CONTRIBUTED PAPER

Combinatorial Optimization of Graphical User Interface Designs

This article surveys combinatorial optimization as a flexible and powerful tool for computational generation and adaptation of graphical user interfaces (GUIs).

By Antti Oulasvirta, Niraj Ramesh Dayama, Morteza Shiripour, Maximilian John, and Andreas Karrenbauer

ABSTRACT | The graphical user interface (GUI) has become the prime means for interacting with computing systems. It leverages human perceptual and motor capabilities for elementary tasks such as command exploration and invocation, information search, and multitasking. For designing a GUI, numerous interconnected decisions must be made such that the outcome strikes a balance between human factors and technical objectives. Normally, design choices are specified manually and coded within the software by professional designers and developers. This article surveys combinatorial optimization as a flexible and powerful tool for computational generation and adaptation of GUIs. As recently as 15 years ago, applications were limited to keyboards and widget layouts. The obstacle has been the mathematical definition of design tasks, on the one hand, and the lack of objective functions that capture essential aspects of human behavior, on the other. This article presents definitions of layout design problems as integer programming tasks, a coherent formalism that permits identification of problem types, analysis of their complexity, and exploitation of known algorithmic solutions. It then surveys advances in formulating evaluative functions for common design-goal foci such as user performance and experience. The convergence of these two advances has expanded the range of solvable problems. Approaches to practical deployment are outlined with a wide spectrum of applications. This article concludes by discussing the position of this application area within optimization and human–computer interaction research and outlines challenges for future work.

KEYWORDS | Combinatorial optimization; computational design; graphical user interfaces (GUIs); human–computer interaction (HCI); integer programming; interactive optimization; meta-heuristic optimization.

I. INTRODUCTION

This article surveys combinatorial optimization approaches for graphical user interface (GUI) design. GUIs have become the prime user interface type for interacting with computing systems. They leverage our perceptual and motor capabilities to cater for elementary interactions with computer programs, such as exploration and invocation of commands, parameter selection, information search, and multitasking. Familiar GUI types include widget layouts, forms, hypertext, toolbars, windows, and menu systems. The design of these GUIs critically affects the usability, usefulness, learnability, and enjoyability of a system, and it shapes the ultimate success and acceptance [1]. Professional titles such as "Interaction Designer" and "User Experience Designer" now represent common professions across major technology companies.

Among the various computational techniques to decide on the GUI design and interactions, combinatorial optimization is distinguished by its algorithmic capacity,

Manuscript received April 8, 2019; revised October 9, 2019 and January 12, 2020; accepted January 17, 2020. Date of publication February 17, 2020; date of current version March 4, 2020. This work was supported in part by the European Research Council (ERC) through the European Union's Horizon 2020 Research and Innovation Program under Grant 637991 and in part by the Academy of Finland projects Bayesian Artefact Design (BAD) and Human Automata. *(Corresponding author: Antti Oulasvirta.)*

Antti Oulasvirta is with the Department of Communications and Networking, School of Electrical Engineering, Aalto University, 02150 Espoo, Finland, and also with the Finnish Center for Artificial Intelligence (FCAI), 02015 Espoo, Finland (e-mail: antti.oulasvirta@aalto.fi).

Niraj Ramesh Dayama and **Morteza Shiripour** are with the Department of Communications and Networking, School of Electrical Engineering, Aalto University, 02150 Espoo, Finland.

Maximilian John and **Andreas Karrenbauer** are with the Max Planck Institute for Informatics, 66123 Saarbrücken, Germany.

Digital Object Identifier 10.1109/JPROC.2020.2969687

controllability, and generalizability [2]. Its potential to complement human designers' work lies in its capability to search for large numbers of possible designs, a target that might otherwise be out of reach. In combinatorial optimization, GUI design is addressed as the algorithmic process of combining design decisions to find acceptable or optimal solutions as defined by an objective function. Design, then, is defined as the process of trying out various decisions to find combinations that yield the highest value for that function. This process is a natural match with user interface design, where a set of elements must be organized and their properties defined. Many interconnected decisions are made, from the selection of functionality to various decisions on how to present them and implement their interaction. A good design must strike a balance among numerous objectives, such as usefulness of functionality, functions' identifiability and ease of use, and complexity and learnability.

In comparison to formal methods, such as logic, combinatorial optimization offers an effective but a flexible way of expressing design knowledge and objectives in a computable manner. Compared to data-driven approaches based on machine learning, such as artificial neural networks, combinatorial optimization allows direct and meaningful control of design outcomes via specific design objectives. It does not rely on the acquisition of a large training set, although its input parameters can be learned from data sets through machine learning methods. Combinatorial optimization approaches can offer proofs for solution quality and, at the same time, can utilize the so-called black-box solvers to include complex models and simulators in objective functions. As a generative method, this can be made as transparent and controllable as desired.

Notwithstanding these apparent benefits, which have been recognized in this domain at least since the 1970s (see Section II), applications beyond simple button layouts have remained out of reach for combinatorial optimization techniques. One reason is that computational optimization techniques tend to insist on explicit and precise inputs. On the contrary, design practice is a characteristically ill-defined process [3]. The fitness of design is best determined in actual use, which is hard to anticipate in advance. Moreover, designers' work is concerned not only with a concrete layout but also with hard-to-formalize conceptual, structure-based, functional, and esthetic aspects of design [4]. To this end, they consider multiple types of constraints when they create, shape, and determine use-oriented qualities [4], [5]. Their success draws from their capabilities of creativity, problem-solving, sensemaking, empathy, and collaboration [4], [6]–[10]. They continuously engage in refining the design objectives and constraints [5], [11]. To explore their ideas, they sketch and implement rapid prototypes [12]. This process is iterative, selective, and corrective: at times, they explore the design space for satisfactory approaches and then switch to an in-depth analysis of the problem to identify a hypothetical best design. They alternate between

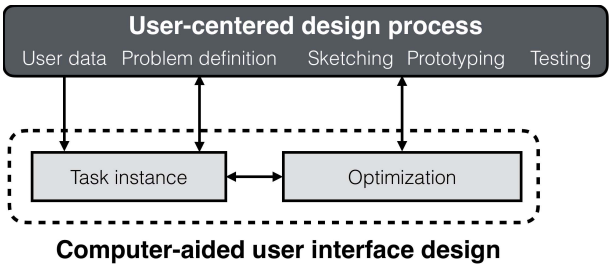

Fig. 1. *Uses of combinatorial design methods in the user-centered design process.*

a constructive and a critical stance. Criticality allows them to assess which aspects of a solution belong together and which are compatible with background data. Design also deploys a multiplicity of representations of human needs and behavior, including user profiles, use cases, storyboards, user requirements, and scenarios. These are used constructively to envision opportunities or to set and refine objectives. Final choices are, for the most part, specified manually for the software by designers and developers. To this end, they use a variety of means, from direct hard-coding to higher level tools offered by software development environments (SDEs). It is fair to ask how combinatorial optimization can assume a role in such a practice.

The goal of this article is to offer a comprehensive survey of the combinatorial optimization approach; this includes the assumptions made, key technical elements, the noteworthy achievements reported over the past few years, and also the major issues encountered. Applications of combinatorial optimization in human–computer interaction (HCI) have been expanding recently. These advances build on convergence of three ideas demonstrated in those areas: 1) definition of appropriate constraints for ruling out infeasible designs, which is sufficient for generating some partial designs (e.g., in GUI layouts, element orders, or placements); 2) definition of user-related design objectives such as movement performance or visual acuity, which makes the outcomes more usable and better customized to individuals; and 3) identification of suitable algorithmic methods, which has made larger and more realistic problem instances solvable. There is increasing understanding of how to formulate objective functions for more comprehensive design objectives, as well as a mathematical understanding of the design decisions involved. These advances have opened up new applications. It is possible to produce usable menu systems, as well as toolbars, and thus document content layouts and collages, web layouts, wireframes, and others. Beyond applications, there is increasing understanding of how best to utilize combinatorial optimization in user-centered design (see Fig. 1). The research surveyed in this article is revealing ways in which design problems can be specified interactively for an optimizer or learned from a data set with machine-learning

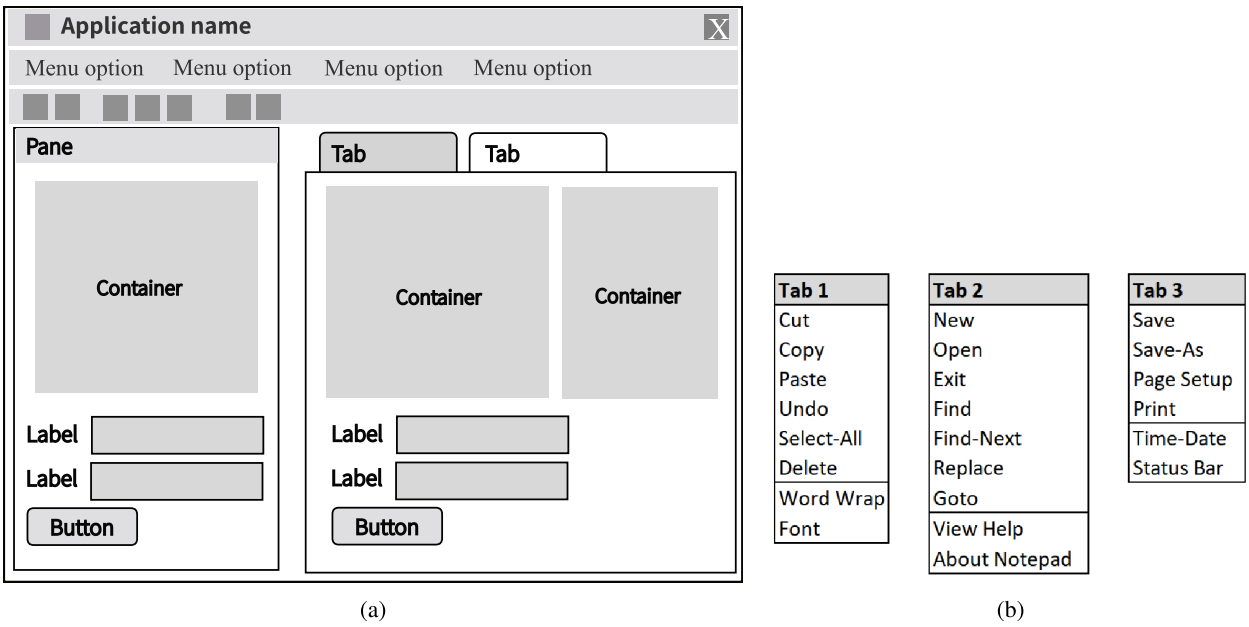

Fig. 2. ***Elementary GUI design tasks reviewed in this article include selecting the functionality, choosing labels and icons, assigning and ordering these (in slots, containers, and hierarchies), and deciding on properties for them such as colors. (a) Desktop application. (b) Hierarchical menu.***

methods, such that the results better capture the relevant objectives and empirical tendencies. Optimizers can be integrated into design tools to assist designers in sketching, wireframing, and prototyping. Moreover, optimizers can be integrated into software systems that adapt user interfaces to individual users.

Optimization offers a rich and flexible toolbox for practical design problems. It can be used to find the optimal design, surprising (different distinct) solutions, designs with particular tradeoffs, or solutions that strike the best compromise among competing objectives or are robust to changes in conditions. A designer can identify how far the present interface is from the best achievable design. Choices behind design can be scrutinized and justified, for better support of building and sharing of knowledge in teams. A computer scientist can use well-established algorithmic techniques—for example, from optimization or operations research (OR)—to analyze problems and hence identify efficient methods to solve them. Advances in algorithms and hardware have made it possible to solve larger problem instances and consider the integration of optimization into design tools. They offer a viable opportunity for finding good or optimal solutions, and, in certain conditions when the so-called exact methods are used, mathematical guarantees (bounds) can be computed. An optimizer can make rigorous statements such as "this design is within 5% of the best achievable design"; such information cannot be delivered by any other known noncomputational methods applied in design. These applications rely on the ability of optimization techniques to search very large numbers of options that designers might have little time to explore manually.

A. Scope

This is the first article to survey combinatorial optimization methods in the area of GUIs. The relevant literature was searched via academic search engines [Google Scholar, IEEE Xplore, and Association for Computing Machinery (ACM) Digital Library (DL)], for combinations of terms related to combinatorial optimization and user interfaces. Ninety-two articles were selected initially, and the pool was expanded by traversing of citation trees.

This article examines combinatorial optimization as a means for generating GUI designs. It focuses on a few recurring issues and design decisions recognized in popular guidance on GUI design (see [13] and [14]): 1) selecting which program functions are to be manipulated and which presented to the user; 2) selecting widget types and their properties, such as labels and colors, and interactive features; 3) deciding how associated interactions map to state-changes in the program; 4) organizing components within the space of their containers, with decisions on positions and on sizes and (overlap-free) shapes; and 5) distributing components across containers to form a hierarchy. Fig. 2 illustrates two common types of GUIs: application GUI and menu. They consist of differing mixtures of elementary decisions and constraints.

Another goal for this article is to expose elements constitutive to graphical interface design. These have been previously scattered around technical articles. To this end, this article surveys formal definitions for commonly used point-and-click GUIs often seen in regular operating systems, apps, and web sites. Specific techniques for interactive graphics [15], command selection [16], and 3-D interaction [17] are left for future

work, as are methods for photo collages [18], document layouts [19], game levels, real-time layout rendering [20], and software-engineering-related topics such as GUI programming [21] and user-interface description languages [22], [23]. Emerging work on human computation and machine-learning-based methods is touched upon only briefly here, and only in the context of their applications in combinatorial optimization. Rule-based [24], probabilistic, and formal methods [25] used for GUI design are similarly discussed from this angle in Section II. Other scholars have provided reviews of design tools using computational methods [2], [13], [26]–[28].

B. Aims

Four technical challenges are addressed in this survey. The authors believe these to have curtailed progress previously.

The first issue is capturing a designer's decision problem as design variables and constraints in a coherent and solvable definition. To formulate a design space requires not only abstraction and mathematical decomposition but also understanding of the designer's subjective and practical problem. Among the recurring design decisions in GUI design are the elements' sizes, colors, and positions and their types. The Cartesian product of design decisions, from which infeasible designs are removed by introducing constraints, forms the design space, also known as the feasible set or the candidate set. The characteristic of interface design, though at times unnoticed by practicing designers, is that the size of design spaces easily gets very large. For example, for n functions there are $2^n - 1$ ways to combine the functions in an application, which, for only 50 functions, means 1 125 899 906 842 623 possibilities. Assuming that 50 commands have been selected, one can organize them into a hierarchical menu in $100! \approx 10^{158}$ ways. Layout problems are even more vast. Determining whether a set of nonoverlapping rectangular objects fits on a display is NP-complete [29]. What optimization offers here is a systematic way of attacking a design problem nonetheless, when the space can be formulated so as to be efficiently searchable.

The second challenge is to define an appropriate evaluative function. This incorporates assumptions about what makes a design "good" for end users. The challenge is to find mathematical descriptions for numerous and often poorly defined objectives: consider learnability versus usefulness, or esthetics versus usability. For example, one well-known objective in menu design is the ease of selecting a command with a pointer. This has been addressed by placing frequently accessed commands closer to the entry point to the menu. The entry point in a tabbed menu is in the top-left corner. Another objective in menu design involves the grouping of items: placing commands close to each other can make it easier to find them. Also, a good menu design balances depth with breadth. This article synthesizes progress in defining evaluative functions, which range from simple rules to full-fledged cognitive simulators.

The third challenge is to formulate the task such that it can be appropriately parameterized (or instantiated or initialized) in a particular project. In optimization parlance, a task instance is a task- and designer-specific parameterization of the design task. Every objective function has parameters, such as weights and coefficients, which must be filled in for a particular task instance. Not only are the objectives different from those in previous research on applied optimization, such as product and assortment optimization in management sciences, but designers understand their work as being iterative, explorative, and corrective rather than about finding the best possible design. How can one define an optimization task such that it addresses the objectives that designers hold to be important and, yet, fits with the practices of designers and developers? These considerations can be addressed by the use of methods familiar from robust and interactive optimization. Instead of just one "optimal design," they generate multiple optimization tasks and account for uncertainty in input data rigorously (see [30]). Machine learning can be used to learn parameterizations in a data-driven manner.

The final challenge is in the definition of a solver strategy that matches practical requirements, particularly as to the availability of such resources as computation time, the uncertainty of inputs, and the need for interactively steering optimization. Previous work either limited the focus to a few decisions at a time or resorted to black-box optimization. Black-box methods, however, neither guarantee global optima nor estimate bounds for the quality of the solution. Exact methods such as integer programming use a structured (nonrandom) search approach that guarantees the optimal solution in finite time. Also, these can provide guarantees (bounds) for the quality of the solution. They allow exploiting widely available and efficient commercial solvers. To sum up, the way these four challenges are solved affects the success or failure of a real-world application.

C. Guidance for Readers

The rest of this article provides a consolidated overview of this rapidly developing area, targeted primarily for researchers in two fields. For optimization researchers, it offers a characterization of the problem (see Section II), definition of key design tasks with integer programming to link them to known problems in operations and optimization research (see Sections III and IV), an overview of evaluative functions (see Section V), and a discussion of characteristic challenges (see Sections VI and VII). An overview of the mathematical problems and their applications is given in Table 1. For HCI researchers, this article offers a brief review of the intellectual positioning of this approach (see Section II), an accessible overview of recurring design problems and advances (see Section VI), and guidance on how to formulate design knowledge and

theories formally as evaluative functions (see Section V), along with an overview of practical issues in deployment, including evaluative methods (see Section VII).

This article may be of interest also for researchers in the fields of human factors and design research. Although there was a significant interest in computational methods in the early years of HCI and human factors research [31], [32], the proposed predictive models of human performance were not linked to research on algorithms that can generate designs. Curiously, the first articles cited OR as an example of how models should be combined with algorithmic research; however, since little advice was given on technical issues such as optimization tasks and methods, applications were practically limited to parameter optimizations and simple mappings. In 2001, a survey of automated usability evaluation [33] was able to identify only a few articles going beyond usability evaluation to algorithmically propose suggestions for improving designs. On the other hand, although OR and management science have examined related topics in product design, these turn out to have few touchpoints with graphical interfaces. The design of product offerings [34], products [35], product lines, and product positionings [36]; supply-chain design [37]; and assortment design [38] focus more on the selection of features for products and much less on their concrete design or end-user use [39]. For example, the Kano model is a product-design model that covers consumer preferences in the features of products [40]. It distinguishes among various types of product qualities, each with differential effects on satisfaction. Although there is more confluence with classic optimization problems such as packing, ordering, and layouts (see Section IV), these definitions must be expanded to account for design and human aspects typifying GUI design; we do this in Section VI.

II. GUI DESIGN AS OPTIMIZATION

A GUI presents the state and control of a computer program visuospatially on a display for interaction with a pointing device [41]. The visual presentation serves two functions: first, the program state can be conveyed to the user, and second, it enables changing the state of the program by interacting with a pointer. Commands are typically carried out by dwelling (e.g., hover-over), clicking, or dragging elements with a pointer. Elements express visually what type of interaction they permit; consider, for example, buttons, widgets, icons, and adjustment handles.

The traditional GUI paradigm is known as WIMP: windows, icons, menus, and pointing device. In addition, modern GUIs offer multiple types of widgets, such as buttons, entry fields, and choosers. Containers of various types are available for media, applications (docks), and documents (folders), and navigation controls such as scrollbars, task switchers, search bars, and tabs translate or update views. In a text-entry mode, text can be entered also via a virtual or physical keyboard. Keyboard shortcuts can be used to invoke commands without pointing.

Since most software and services have extensive functionality to offer, GUIs are often organized hierarchically. Two principles of hierarchical organization are commonly followed.

1) *Visual Containment*: Graphically marked containers such as canvases, windows, and boxes can have other containers and elements within them.
2) *Logical Compositionality*: A program can consist of multiple sub-GUIs, such as a settings panel, a drawing canvas, and a dialog. These can be presented in sequence or parallel. For example, the 3-D modeling software Maya, which offers 1346 functions in a menu, arranges them in a hierarchical fashion [39].

One popular way of arranging elements in a GUI is the grid layout [42]. It uses grid lines to organize slots that determine the possible sizes and positions of graphical elements. Besides purely technical considerations (e.g., software and hardware reliability) and considerations related to marketing and brands, there are end-user-related design objectives in GUI design. They include: 1) usefulness; 2) user performance such as speed and accuracy in completing tasks; 3) learnability; and 4) aspects of user experience such as esthetics, emotions, or perceived value. To understand which objectives are important, companies invest significantly in user research, wherein the methods include, among others, surveys, online logging, controlled evaluations, and observational studies. Such techniques are used to chart the needs, practices, capabilities, and technical contexts of users. However, it is widely accepted that the quality of the design is determined in actual use. This creates a tough challenge for design. A designer must anticipate how well users will perform and how they will use and experience a design candidate. To this end, designers conventionally rely on design heuristics—well-founded rule-like conventions such as "do not use more than four colors to code information"—alongside design patterns, empirical evaluation such as usability and A/B testing, and personal experience [14]. Research in the fields of cognitive psychology and human factors has revealed several mathematical and simulation models that capture aspects of graphical interaction; however, there is no comprehensive predictive model yet for GUI interaction, although the topic has been of sustained interest in research. That said, many practical models exist for focused topics. We review these in Section V.

A. History of Optimization-Based Approaches in HCI Research

Combinatorial optimization of GUIs has attracted interest from a number of fields. However, efforts have been fragmented. The articles surveyed below have been published in the disciplines of applied mathematics and OR, artificial intelligence, machine learning, software engineering, HCI, ergonomics, design research, and cognitive psychology. They cover mathematical definitions of design problems, efficient solvers, the learning of objective

Table 1 Overview of Paper Structure

Integer programming problem Section IV	**Graphical interface design task** Section VI	**Two example design objectives** Sections V and VI
A. Selection	A. Functionality selection	Perceived usefulness, cost
A. Selection	B. Label selection	Consistency, memorability
A. Selection	C. Icon selection	Comprehensibility, identifiability
A. Selection	D. Widget selection	Requirement matching, usefulness
B. Ordering	E. Linear menus	Selection time, association
C. Assignment	E. Hierarchical menus	Foraging time, selection time
C. Assignment	G. Distributing UIs	Accessibility, satisfaction
D. Layouts	F. Grid layouts	Alignment, preferential placement

function parameters from data, description of design tasks and device characteristics, creativity support for designers, the effect on design practice, and psychological models as objective functions. Although a historical review is beyond the scope of this survey, four milestones are worth mentioning.

The first is the expanding scope of mathematically defined design problems. The characterization of keyboard layout tasks as a quadratic assignment task was pioneered by Pollatschek *et al.* [43] and later by Burkard and Offermann [44] who proposed efficient solutions to the problem (see [45], for a recent review). The observation that graphical layouts can be defined as a packing problem was made by Hart and Yi-Hsin [46].

From the study of Sutherland's Sketchpad in 1964 [47], constraints have been used for graphical interaction. Since then, constraints have been discovered that are exploitable to guarantee feasible layouts [48]—for example, elements not overlapping or getting clipped. Layouts can be created via adding constraints incrementally [49]. Chorus [50] addressed nonlinear geometric constraints such as Euclidean geometric, nonoverlapping, and graph layout constraints. It also introduced soft constraints with hierarchical strengths or preferences. Cassowary [51] implemented a linear arithmetic constraint solver to adapt the layout to changing sizes. Some ideas of this sort have been implemented in popular commercial systems such as Apple's Auto Layout, enabling GUIs to adjust their layout dynamically as window or screen dimensions change (these definitions, which form the core of combinatorial solutions, are dealt with in detail in Section IV). Constraint satisfaction alone, however, has been inadequate for producing full designs beyond laying out elements. Also, constraint systems are hard to develop and maintain; furthermore, the criterion of "goodness" of a design is often unclear. Elements can be chosen, ordered, and positioned with constraints, but without added assumptions connected with human attention and motor control, many of the layouts produced remain practically useless.

The second milestone lay in defining design tasks by using formalisms compatible with software engineering practices that employ interactive software. Model-based design engineering (MDE) [52] developed rich notation and languages for expressing user tasks and software- and hardware-related constraints. This field, which emphasizes more on integration with software engineering practices and standards, has been subject to serious criticism [22], [23], [28], [53], [54], which can be summarized in terms of three obstacles: 1) reliance on a large number of hard-coded heuristics, which make the system hard to control, expand, and learn; 2) rigidity of the model, with limited scope of important decisions that can be made; and 3) limited success with demonstrable improvements in usability. However, MDE has proposed formulations of design tasks that allow representing users' tasks, limits of the input and output devices, and properties of the user interface.

The third milestone came in the use of psychologically plausible models as objective functions, pioneered by Fisher [32], [55]. These ideas were explored in the menu and keyboard design, wherein simple navigation and motor performance models were used to model how users navigate or how quickly the fingers move. Zhai *et al.* [56] developed a widely used approach to keyboard layout optimization that is based on Fitts' law. The idea of using sensorimotor models for objective functions was adopted by Gajos *et al.* [57] for widget layout design. However, their approach assumed functional prespecification of the interface, including at times a functionality hierarchy, which imposed an overhead to use and could also strongly constrain the outcomes of optimization.

The fourth milestone lies in the demonstration of viable interaction concepts for integrating optimizers into design tools [58]–[60]. This has made it possible to design a GUI through an optimizer without deeper understanding of optimization. We review interaction concepts at the end of this article.

For overcoming the obstacles faced by previous attempts, a three-pronged approach is needed, one that: 1) uses empirically verifiable models of interaction and user experience as objective functions; 2) expresses design tasks by using a formalism that permits analysis of problems and solutions, such as integer programming; and 3) integrates with design tools that allow defining design tasks for a solver interactively without in-depth understanding of the mathematical underpinnings.

B. Basic Concepts

This section introduces the fundamental concepts of combinatorial optimization in the context of GUI design.

User interface design is formulated as algorithmic combination of discrete design decisions utilized with the goal of obtaining an optimal solution defined by an objective function. Following a standard definition of combinatorial

optimization [61], we define a design task as

$$\text{Find } \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \boldsymbol{X}, \text{ which maximizes } \boldsymbol{f_\theta}(\boldsymbol{x})$$

where $\boldsymbol{x}$ is an n-dimensional design vector, each dimension describing a design variable, $\boldsymbol{X}$ is a set of candidate designs, $\boldsymbol{f}$ is an objective function, and $\boldsymbol{\theta}$ is a set of parameters defining the task instance. Further definitions given in this article fall under this broad definition. This opens a rigorous but rich set of concepts for formally describing a design task.

Design space $\boldsymbol{X}$ is the set of candidate designs, or all to-be-considered design vectors. It is contained in the Cartesian product of the domains of the design variables. A design variable can address any open decision, which may be defined via, for example, Boolean, integer, or categorical variables. Examples include sizes and colors and the positions of elements and their types. A benefit of this formulation is that the size of the design space can be estimated, provided that $\boldsymbol{X}$ is known. In graphical interface design, the number of combinations of such choices grows very large easily.

Evaluative knowledge, or assumptions about what makes a design good for end users, is expressed in the objective function $\boldsymbol{f_\theta}(\boldsymbol{x})$.[1] It assigns an objective value to a design candidate $\boldsymbol{x} \in \boldsymbol{X}$. As discussed later, the functions might be, for example, if–then rules (heuristics), metrics, regression models, or simulators. The choice of a function has a profound effect on outcomes and the process of optimization.

The task instance is a parameterization of the objective function in a given design project. It comprises the particular decision variables, weights of objectives, and constraints that characterize the task at hand. As we discuss later in the article, the acquisition of a task instance is a major challenge arising from the nature of design practice.

Optimization is the process of searching the design space for candidates that yield the most desirable value of f. The candidate that obtains the highest (or lowest, in minimizing) score is the optimum design. A design is globally optimal when the search method guarantees that it achieves the best objective value over the whole design space. Such guarantees are offered by exhaustive search and exact methods (e.g., integer programming). Design is approximately optimal when its objective value is within some margin of the optimal design or when there is a good chance that only marginally better designs exist.

Multicriteria optimization describes the case with multiple objective functions $f_i : X \to \mathbb{R}$ for $i = 1, \ldots, k$ instead of only one. The utopian design, or a solution that optimizes all objective functions simultaneously, is often nonexistent. Therefore, the optimal solutions of multicriteria optimization problems are referred to as Pareto-optimal points. A feasible solution is described as Pareto-optimal if one cannot improve any objective function without the score for another suffering. Several solution strategies for multicriteria optimization problems, among them hierarchical or aggregating approaches, are discussed in Section VII.

In the following material, design problems are modeled via multiobjective models, discussed in more detail in Section VII-D. Throughout, multiple objectives $f_1, \ldots, f_k$ will be denoted by $(f_1|\cdots|f_k)$.

After providing an overview of applications of combinatorial optimization, we proceed to survey mathematical representations of interface design tasks.

C. Applications

1) Generative Design: In generative design, the goals are twofold: 1) to solve problems that are rendered hard by a large problem size or number of objectives and 2) construct designs algorithmically with high quality of the design for end users, encompassing goals such as usability, good ergonomics and user experience, and reliability and recovery from errors [62]. Demonstrated applications for GUIs include generation of graphical layouts [13], [30], [42], [46], [48], [60], [63], [64], creation of document layouts [65], multiplatform design [66]–[69], generation from program code [70], generation of forms from data descriptions [71], [72], retargeting of input spaces for ergonomics [73], retargeting of web pages [74], dialogue box layouts [75], distributed user interfaces [76], pervasive information displays [77], multiappliance services [78], and functionality design [39], as well as design for web pages [79], [80], menus [81], scatterplot visualizations [82], and widget layouts [83].

2) Interactive Design: The paradigm of "one-shot" optimization is often impractical. In design practice, problems are ill-defined and designers learn and update the problem definition and the design space. Human-in-the-loop—or optimizer-in-the-loop, depending on the level of automation—methods can help designers in many ways. Optimizers may aid in completing partial designs, exploring designs, comparing them, and finding a rationale for them. They can be categorized in accordance with the way they coordinate changes with the designer [84]. At one extreme, the designer commands the optimizer prior to optimization, defining objectives, constraints, etc. At the other, the tool recognizes the designer's task for this article or the intent and assists either automatically or by asking the designer to pick or confirm updates. It can even "nudge" a novice designer toward better choices. Strategies in interactive optimization are discussed in Section VII.

3) Adaptation and Personalization: Combinatorial optimization can adapt designs when its objective function

[1] At times referred to as a loss function, merit function, criteria function, reward function, evaluative function, utility function, goodness function, or fitness function.

is able to represent individual-specific or moment-by-moment requirements. For example, a design can be reoptimized when the device orientation changes [85]. Todi *et al.* [79] adapted web layouts in line with the visual history of a user [80], and Lindlbauer *et al.* [86] proposed an algorithm to automatically adapt mixed reality interfaces in line with the current user context, leveraging the environment and the cognitive load imposed by the user's current task. In real time, their optimization-based approach adjusts which information is displayed in the interface and where it gets placed.

In ability-based optimization, designs are generated by taking an individual's motor or cognitive impairments into account [83], [87], [88]. SUPPLE uses a regression model of pointing time (from Fitts' law) and heuristic models of human vision to generate widget layouts for motor- and vision-impaired users [83]. Sarcar *et al.* [88] used simulator-based forward prediction as an objective function when individuating designs for users with tremors and dyslexia. They used a simulator model of text entry, which predicts the movement of the eyes and fingers in touchscreen text entry. In the cases above, parameters were manually individuated by reference to the literature.

4) Decision Support: Algorithmic exploration of a design space can not only generate designs but inform decision-making. Exact solvers can apply bounds to solution quality, indicating, for example, that a given design is within p% of the global optimum. In Pareto front optimization, equally good (nondominated) designs can be sought as alternatives to a given design. In robust optimization, a design can be searched for that is robust to changes in assumptions (e.g., assumptions about users' tasks). With explorative optimization, maximally diverse design options can be sought that are within some margin of tolerance of a given design. In local optimization, the best k updates to a given design can be sought.

III. APPROACHES TO MODELING COMBINATORIAL OPTIMIZATION TASKS

This section briefly discusses two main approaches available to model computational design tasks. In Sections IV and VI, we have opted to use one particular formalism, integer programming, because of its efficient and natural representation of design tasks.

The approaches differ in how the design space can be defined. The design space $\boldsymbol{X}$, as defined above, is contained in the Cartesian product of the domains of the design variables. Typically, the design space is a proper subset since only rarely do all combinations yield feasible designs. Commonly, a list of constraints is used to describe the subset of feasible designs. However, it is possible to discriminate feasible designs from infeasible designs by penalizing the latter in the objective function, avoiding constraints that are hard to formalize, or lifting all constraints. Moreover, the design space can be implicitly described by a starting design and a transition relation; that is, the design space may consist of all designs reachable from the starting solution after a series of transitions.

Along with this discussion, we also illustrate the two approaches by means of a representative example: the menu layout problem. The menu layout task requires to place n different commands under suitable tabs in a menu structure. For the purposes of this discussion, the objective is to attain a balance between the speed of access (selection time) and the logical interrelation between collocated commands (searchability).

A. Black-Box Approaches

One widely applicable and generic principle for expressing arbitrary design spaces is based on oracle queries—calls of a black-box function that returns an evaluation of a given design vector. In essence, any computable function, even simulations [89], can be used as an oracle. However, this is both a strength and weakness at the same time because the expressive power renders good designs hard to find. Appropriate transition relations could provide some structure that makes local neighborhood search heuristics effective. Moreover, machine-learning methods such as reinforcement learning [90] can be used in this context.

To formulate the menu design problem using black-box techniques, we first define the evaluation function as a weighted combination of the performance (access time) and learnability (relative placement of interrelated commands). In practice, this function covers the time required to access individual commands and the placement distance between logically interrelated commands. A large number of diverse solutions is first generated; this can be done via randomization or can be done by using constructive heuristic techniques. Thereafter, several meta-heuristic approaches are available to steadily improve on the original seed solutions.

B. Constraint and Integer Programming

Constraint programming [91] is a more structured approach to separating feasible designs from infeasible designs. To this end, a list of constraints $C_1, \ldots, C_m$ is given, where C_i maps design vectors to $\{0, 1\}$ for each $i \in [m]$—that is, to either `false` or `true`, where we say that a constraint is violated in the former case and satisfied in the latter. A design is feasible if and only if all constraints are satisfied and infeasible otherwise. The constraints can consist of logic formulae, including arithmetic predicates.

However, in this article, we focus mainly on constraints that are linear equations or inequalities over integer variables, which yields the powerful integer program (IP) subclass that we discuss in Section IV. Though integer programming can be considered a special case of constraint programming, the two approaches differ significantly with respect to the solution process: the former is centered more on finding feasible solutions, often by using fundamental concepts from logic or graph theory, whereas the

latter exploits numerical algebra for computing bounds of relaxations. Moreover, there exists an area dubbed constraint integer programming [92], an attempt to exploit both worlds.

The key motivation behind the use of IPs is that it offers a compact abstraction that is compatible with design. In practical projects, the formulations outlined in the next few sections can be employed in a plug-and-play manner: Design knowledge can be implemented as decisions, objectives, and constraints, with the solution then found by an IP solver. Irrespective of whether some consideration (either a constraint or an expression within the objective) is actually used, the IP solver follows the standard algorithm and provides results. An IP is a representation of design tasks that permits analysis and comparison of complexity as well as direct solving by modern solvers. If the type of the necessary functions and expressions so allows, exact methods such as branch-and-bound can be used. In contrast, using other optimization techniques, especially black-box optimization, would often require substantial tuning of parameters whenever any consideration (constraint or objective) is modified. Moreover, when the core design tasks are explicated and linked to known concepts in optimization, one gains a powerful analytical tool, which increases the general understanding of graphical interfaces.

Using menu design as an example, one can formulate the problem in terms of an IP using the following binary decision variables:

$$\begin{aligned} T_i^t &\longrightarrow \text{is command } i \text{ placed in tab } t \\ R_i^r &\longrightarrow \text{is command } i \text{ placed in row } r \\ Z_{ij} &\longrightarrow \text{are commands } i, j \text{ placed in the same tab.} \end{aligned}$$

In addition, we define variable G_i to indicate the time required to "reach" command i as indicated by Fitts' law. Then we can exactly define the objective function verbally characterized in Section III-A

$$\sum_{i \in N} \sum_{j \in N} A_{ij} Z_{ij} + \sum_{i \in N} F_i t_i.$$

Here, F_i is the frequency of usage of command i, and A_{ij} is the logical interrelation between commands i and$j$. Along with this objective, the IP requires constraints such as listed below[2]

$$\begin{aligned} \sum_t T_i^t &= 1 \quad \forall i \in N \\ T_i^t &\geq T_j^t + Z_{ij} - 1 \quad \forall i, j \in N, \forall t \\ R_i^r + R_j^r + Z_{ij} &\leq 2 \quad \forall i, j \in N, \forall r. \end{aligned}$$

[2]The formulation presented above is representative only. More variables and constraints are required for to completely formulate the menu layout problem.

IV. INTEGER PROGRAMMING FORMULATIONS

This section presents elementary integer programming formulations that underpin the applications of combinatorial optimization in user interface design dealt with in Section VI. We provide a brief introduction to integer programming here; the interested reader is referred to [93]–[95].

As discussed above, a GUI design is composed of multiple decisions and specified by a vector $x = (x_1, \ldots, x_n)$ from the set of candidate designs $X \subseteq X_1 \times \cdots \times X_n$, where X_i is the domain for decision x_i, $i \in \{1, \ldots, n\}$—for example, reals, rationals, integers, and (most prominently) binary variables: $X_i = \{0, 1\}$. Typically, however, X does not comprise the whole Cartesian product; it is restricted to some subset defined by problem-specific constraints. For example, consider the design of a fixed-size layout. In this case, the layout specification is considered to be well-defined only if all elements are (explicitly) connected to at least one horizontal and one vertical identifier. These connections can be expressed as a 1-D array, a 2-D table, or a system of constraints [96]. Lok and Feiner [48] provide a classification of these constraints as: 1) relational ones, such as "Element 1 refers to Element 2" and 2) spatial constraints, such as an element occurring in a particular location. A typical case of relational constraints is that a search engine's *Search* button is expected to appear immediately adjacent to the search-text field. A typical case of spatial constraints is visible in most standard web sites, which place the company logo in the page header. Borning *et al.* [97] extended the constraints-based approach to the web, and Zeidler *et al.* [96] applied it to document layouts.

A straightforward yet powerful class of constraints, used below, is obtained from linear inequalities, which define the so-called half-spaces $H := \{x \in \mathbb{R} : a^T x \leq \beta\}$, where $a \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$. The intersection of finitely many half-spaces is called a polyhedron. Many, if not (almost) all, interesting sets of candidate designs can be described as the intersection of a polyhedron with the integer lattice; for example, $X = \{x \in \mathbb{Z}^n : Ax \leq b\}$ where A is an $m \times n$ constraint matrix, b is the so-called right-hand side, and the comparison is done componentwise. Most often, A and b can be chosen to consist of integer entries.

The applications discussed below can be modeled as a maximization problem for a linear function over the intersection of such a polyhedron with $\mathbb{Z}^n$. Such an optimization problem is called an integer linear program (ILP). We will also see models where the objective is a quadratic function, but this generalization does not add any modeling power and can be linearized if necessary. If only a subset of the variables is required to be an integer, it is called a mixed- ILP (MIP). An MIP can be reduced to a series of the so-called feasibility problems, where, instead of maximizing $f(x)$ over all $x \in X$, we ask whether there is a feasible solution to $X \cap \{x \in \mathbb{R}^n : f(x) \geq \theta\}$ for suitable choices of θ, determined, for

example, by binary search. If $f(x)$ is a linear function—that is, $f(x) = c^T x$—then this is just another MIP. If it is a nonlinear or even a black-box function, outer and inner piecewise linear approximations can be considered; that is, $f(x)$ is point-wise overestimated or underestimated by linear functions. In the case of nonlinear functions that are out of reach for standard exact solvers, the problems can be addressed with hybrid solvers.[3]

The elementary tasks reviewed in the following discussion are selection, covering, packing, network design, ordering, assignment, and layouts. Although these are thoroughly studied problems in operations and optimization research, their application in this domain requires specific modifications to objectives and constraints. We review the general design decisions (design variables) and constraints here to expose the structure of these problems. The matter of adequate objective functions and solvers is discussed further along in this article.

A. Selection

Selection problems involve choosing a set of given elements to meet given requirements while optimizing one or multiple objective functions. Selection problems are seen in GUI design when, for example, one is selecting functions for a design. We divide selection problems into four subtypes. These formulations extend a book chapter by some of the authors of this article [98].

1) Covering: In a covering problem, we are given a set of elements $U = \{u_1, \ldots, u_n\}$ and a set of attributes $A = \{a_1, \ldots, a_m\}$. Each element u_i is equipped with a nonempty set of attributes $A_i \subseteq A$. Moreover, every attribute a_j has a certain requirement r_j. The task is to select a subset of U that satisfies all requirements; that is, the number of selected elements with attribute a_j is at least r_j for each attribute. The goal is to find a selection that minimizes the total costs—that is, the sum of the costs for each selected element. Sometimes an element may have an attribute with some multiplicity; for example, for the attribute profit, we would be given values associated with each element. Moreover, it may or may not be allowed to select an element multiple times.

Consider the selection of widget types as described in the next section, for example. A designer imposes a minimum set of requirements for an interface, and there are multiple widgets to select from, each with distinct properties. A good solution to this problem covers all the requirements at a minimal cost.

The quintessential covering problem is a set-covering problem [99], where all requirements are 1 and in the cardinality version all costs are 1 as well. The term "set" refers to the different subsets of elements that share a common attribute. A special case of the set-covering problem is a vertex cover where the elements are the nodes of a given graph and the attributes are the edges; that is, we are asked to select a subset of the nodes of minimum cardinality that contains at least one node of each edge. A vertex cover is NP-hard, and the general set-covering problem is NP-hard even to approximate within a factor of $\mathcal{O}(\log n)$ [100]. On the other hand, a greedy algorithm produces an approximate set cover within this logarithmic optimality guarantee.

2) Packing: Packing problems are similar to covering problems: again, we are given a set of elements $U = \{u_1, \ldots, u_n\}$ and attributes $A = \{a_1, \ldots, a_m\}$. Instead of requirements, though, we now have positive capacities c_j for each attribute a_j, and only those subsets of U are feasible that do not exceed the capacity for each attribute. Analogously, a backpack has a certain volume and a total weight limit over all packed items. The attributes are volume and weight. Also, the elements are typically associated with a valuation (say, p_i for element u_i) that can be considered as a profit. Hence, the goal is to find a selection of the elements that do not exceed the capacities and maximizes the total profit. Again, it may or may not be allowed to select an item multiple times.

Consider again the widget selection problem: Not all available widgets might fit within a canvas of limited size. If an individual reward is associated with each widget, it is natural to ask for a fitting selection that yields the greatest reward. Similarly, not all functionality may be implemented in a limited time budget.

Packing problems are a dual to covering problems: when we switch the role of elements and attributes, the role of capacities and costs, and the role of profits and requirements, we obtain the associated dual covering problem.

An important fact that follows from optimization theory [94] is that the profit of any solution for the packing problem cannot exceed the costs of any solution for the associated dual covering problem. In particular, this holds for the maximum profit and the minimum costs. The dual problem for the vertex cover problem mentioned above is the so-called matching problem wherein we are asked to find a maximum cardinality subset of the edges such that each node is incident to no more than one edge in the matching. Other well-known packing problems are the independent set problem, the set packing problem, and the knapsack problem. The independent set problem often appears as a subproblem for modeling mutually exclusive choices. While matching can be solved in polynomial time, the other problems mentioned in this section are NP-hard. The independent set type is also NP-hard to approximate within a factor of $\mathcal{O}(n^{1-\varepsilon})$ [101].

3) Network Design: Network design problems are a special type of selection problems seen in connection with graphs. Typically, we are given a graph and are supposed to select a subset of the edges such that certain connectivity constraints are satisfied. A simple example is the Steiner Tree problem: for a given undirected graph $G = (V, E)$ and a set of terminals T (i.e., $T \subseteq V$), we shall select edges, such as $E' \subset E$, such that between each two terminals

[3]For example, Local Solver; see http://localsolver.com/.

there is a path consisting only of edges from E'. Observe that at least one edge in a cycle can be removed while connectivity is still maintained, and, hence, every optimum solution is a tree. The nodes of the selected edges that are not terminals are called Steiner nodes. For the Directed Steiner Tree problem, we are given a directed graph $G = (V, A)$, a designated root node $r \in V$, and a set of terminals $T \subseteq V \setminus \{r\}$, and our task is to select arcs such that there is a directed path from r to each terminal consisting only of selected arcs. Typically, a cost is associated with each arc, and we are supposed to find a feasible selection of minimum costs. Sometimes only the root and the terminals are given explicitly and the remaining nodes are only given implicitly—for example, we could have a Steiner node associated with each subset of the terminals. A feasible solution would then define a hierarchical clustering of the terminals.

This problem appears, for instance, in the context of the design of a hierarchically organized user interface. In Section VI, we look at the problem of a hierarchical menu structure as an example. The terminals are the actions that should be available as menu items. For achieving quick access times, it makes sense to avoid clutter by grouping related actions together in a submenu. The root node represents the menu bar, and an arc represents a parent–child relationship; that is, selecting an arc from the root to a menu item means to put this item directly on the menu bar, hence making it accessible via one click.

B. Ordering

In ordering problems, we are asked to provide a permutation of a given ground set. The permutation should minimize the cost of traversing the elements in a chosen order. More specifically, there is a cost associated with the transition from an element to its successor in the designated order [102].

A famous example is the traveling salesman problem (TSP). We are given a set of locations $P = \{p_1, \ldots, p_n\}$ and distances d_{ij} for each pair of locations i, j. The goal is to find the shortest tour that visits every location and then returns to the starting point—that is, to find a permutation $\pi : [n] \to [n]$ of the locations such that $d_{\pi(n),\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)}$ is minimized.

In GUI design, consider the task of ordering items in a linearly organized menu. The goal here is to order a given set of menu items such that skimming the menu becomes as fast as possible. Two items, when placed adjacent to each other, have an associated reading time such that the first primes the other. For example, reading "Save" after "Open" is faster than after "Bookmarks." Another variant, easier, is to organize the menu for fast access to the most commonly used commands. In this case, the problem collapses to that of ordering commands by frequency. Linear ordering is NP-hard. Although many special cases of the TSP admit a PTAS [103], the general TSP does not do so unless $P = \text{NP}$ [104].

C. Assignment

Assignment problems belong to the best-studied class of optimization problems [105]. In the general version, the task is to find a one-to-one correspondence between n items and n locations such that the total cost of the assignment is minimized. In mathematical notation, x being a valid assignment can be written as follows:

$$x \in \Pi_n := \left\{ x \in \mathbb{Z}^{n \times n} : \sum_{i=1}^{n} x_{ik} = 1 \, \forall k \in [n] \right. \\ \left. \text{and} \sum_{k=1}^{n} x_{ik} = 1 \, \forall i \in [n] \right\}.$$

The two most popular variants of this problem are the linear and the quadratic assignment problem. In the former case, the objective function is linear (i.e., of the form $\sum_{i,k} c_{ik} x_{ik}$) and the optimal solution can be computed efficiently. Two famous algorithms for the linear assignment problem are the Hungarian method, which runs in $\mathcal{O}(n^3)$, and the bipartite matching algorithm of Duan and Su [106], which runs in time $\mathcal{O}(n^{5/2} \log C)$ for an integer cost of, at most, C.

In many practical applications, we need to account additionally for the interaction between the items and the interaction between the locations, but the linear assignment problem cannot address this. Therefore, the quadratic assignment problem, which adds a quadratic term of the form $\sum_{i,j,k,\ell} c_{ijk\ell} x_{ik} x_{j\ell}$ to the objective function, becomes more and more relevant.

In the variant of Koopmans and Beckmann [107], the cost can be factored into two parts, one being relevant for either only items or only locations. Mathematically, this means we can write $c_{ijk\ell} = f_{ij} \cdot d_{k\ell}$, where we interpret f_{ij} as the dependence of item i on item j and $d_{k\ell}$ as the distance between the locations k and ℓ. Though this factorization simplifies the problem, the quadratic assignment problem remains one of the hardest combinatorial optimization problems. There even exist unsolved instances of only 30 items and locations. Queyranne [108] showed that it is NP-hard to approximate this problem within any constant factor, even if the cost can be factorized to a symmetric block diagonal flow matrix and a distance matrix describing the distances of a set of points on a line. One systematic way to tackle this complexity is to compute relaxations in a branch-and-bound framework, where the performance of such a relaxation is measured in the quality of the lower bound it produces and also in the time needed to compute its solution. While many of the early relaxations published [109]–[111] replace the quadratic term with an increased number of linear terms, semidefinite programming relaxations have been gaining interest lately [112]–[115]. In-depth treatments of the quadratic assignment problem exist [116], [117].

Examples of assignment problems in user interfaces are the linear and grid menu problems, which are discussed

in Section VI. For a given set of commands and positions, the target is to uniquely assign the commands to positions. Beyond other criteria, one goal with these problems is to minimize the gap between closely interrelated problems, which can be modeled by the quadratic objective function of the assignment problem.

D. Layouts

Layout problems are closely related to packing problems: We have to fit a set of objects onto a canvas. A set of objects is given, and they have to be placed in line with feasibility constraints such that there is no overlap or overflow of objects. The most common instances of this problem class appear in geometric 2-D layout; that is, we are given a set of objects in the plane, such as rectangles or bounding boxes of other shapes, and we are supposed to fit them within a 2-D canvas [118].

The layout of rectangle i can be specified by the coordinates of the top-left corner (x, y) together with its width w and height h. Typically, width and height are given for each rectangle (or at least by lower bounds) while the coordinates are variables. If a grid layout is desired, these variables are constrained to be integers. To avoid an overlap of two rectangles, one of them has to be to the left of or above the other. Rectangle i is to the left of rectangle j if and only if $x_i + w_i \leq x_j$. Similarly, i is above j if and only if $y_i + h_i \leq y_j$. Switching the roles of i and j, we obtain $x_j + w_j \leq x_i$ and $y_i + h_i \leq y_j$, where at least one of these four constraints has to hold and it is clear that not all of them can do so at the same time. Hence, we have to model the disjunction of constraints. This is done via auxiliary binary variables, as in using d_{ij} and a_{ij}, to (de)activate the respective constraints

$$x_i + w_i \leq x_j + W \cdot (1 - d_{ij}) \quad (1)$$

$$y_i + h_i \leq y_j + H \cdot (1 - a_{ij}) \quad (2)$$

$$x_j + w_j \leq x_i + W \cdot (1 - d_{ji}) \quad (3)$$

$$y_j + h_j \leq y_i + H \cdot (1 - a_{ji}) \quad (4)$$

where W and H are the width and height of the canvas and at least one of them has to be activated

$$d_{ij} + a_{ij} + d_{ji} + a_{ji} \geq 1. \quad (5)$$

While these layout problems are typically NP-hard, there often exists a constant approximation or even a PTAS for them [119], [120].

V. OBJECTIVE FUNCTIONS: MATHEMATICAL REPRESENTATION OF EVALUATIVE KNOWLEDGE

Evaluative knowledge can be represented in combinatorial optimization in various ways, ranging from if–then rules to step-by-step-executed simulation models. Table 2 provides an overview of the main categories of models from the perspective of their psychological domain. The factors reported on thus far cover most recognized aspects of usability, including user performance, learnability, experience, and ergonomics. Ivory and Hearst [33] reviewed models and methods for automated usability evaluation, including cognitive models such as GOMS and GLEAN, as well as cognitive task analysis and Petri nets. This section of the article focuses on models from the perspectives of empirical validity and computational efficiency. In Sections IV and VI, indications as to objective functions are given where possible, but the design tasks were defined without imposition of any specific ones. This formulation allows flexibility in replacing and improving those used.

Mathematically, an objective function is a function that maps a design candidate $\boldsymbol{x} \in \boldsymbol{X}$ to n real-valued predictors: $f : \boldsymbol{X} \rightarrow \mathbb{R}^n, n \geq 1$. There are four considerations in model selection:

1) computational efficiency;
2) coverage of relevant factors;
3) quality of design choices for end users;
4) parameter selection for new task instances.

However, state-of-the-art models often come with a performance handicap. Although they may cover a wider range of phenomena and produce the most valid results, they are typically too inefficient for the optimization of large problems. Per-simulation execution times range from seconds to days. Moreover, models with multiple parameters may be harder to fit to data and are prone to overfitting. On the other hand, the simplest linear models are very fast and are easier to fit to data but cover a narrower range of phenomena.

An objective function makes empirically verifiable predictions linking design choices and outcomes for users. To be useful in optimization, they must permit counterfactual reasoning of the type "if the design were like this, these consequences would occur (for end users)." An antecedent ("if speech were used for typing") is linked to a numerically expressed consequent ("words per minute would be 70 and error rate 5%"). For example, in Fitts' law, [124], which is widely used for keyboard optimization [45], the consequent is MT, or movement time, and two design-related antecedent conditions are stated: 1) D, or distance to target from the present location of a pointer and 2) W, or width of target.

An objective function may also have empirical parameters that need to be calibrated for the task instance at hand. This can be done, for example, by preference elicitation methods [125] or via statistical fitting of parameters to some data set. For example, Fitts' law has two free parameters, a and b, which are shaped by user-, posture-, and device-related factors and are set via ordinary least squares (OLS). At the end of this article, we return to discussing parameter inference and empirical evidence for the models presented here.

Table 2 Selected Examples of Objective Functions Used in User Interface Optimization

Model	Objective	Type(*)	Design variables	No. of free parameters	Application	Ref.
Motor performance						
Fitts' law	Target selection: Speed and accuracy	R	Distance and width of a button	2	Button layouts (e.g., keyboards)	[56]
Cue integration theory	Target selection in time: Speed and accuracy	R	Distance and width of a temporal target	2	Temporal selection (e.g., games)	[121]
SDP(a)	Menu-item selection time	R	Menu length, target index	6	Linear and hierarchical menus	[58]
Motor complexity	Learnability of complex coarticulated movements	R	Gestures	None	Gestural input	[122]
Perception & attention						
Familiarity	Positional similarity with previous designs	R	Distance of elements in a menu hierarchy	All	Menu layouts	[58]
Feature congestion	Clutter perception	I	Colors, shapes, and sizes of elements	None	Web layouts	[60]
Grid alignment	Fluency of figure/ground perception	I	Positions of elements	None	Web layouts	[60]
Active vision	Search time for a target	S	Colors, shapes, and sizes of elements	None	Element positions and colors	[60]
Aesthetics						
Color harmony	Perceived harmony of colors	I	Colors of elements	None	Web layouts	[60]
Task performance						
Task completion time	Key-stroke-level task performance	R	GUI state-machine design	6	Point-and-click interfaces	[138]
Ergonomics						
RULA	Maximum postural comfort	R	Arm posture	N/A	Mid-air input	[73]
MA(m)	Minimum fatigue	S	3D element positions	None	Mid-air-gesture interfaces	[123]

Notes: (*) R = regression model, I = image-based metric, H = heuristic, and S = simulator model; (a) Search–Decide–Point; (m) total muscle activation

Keyboard optimization is an application area closely related to combinatorial optimization. This field has seen several objective functions developed, addressing different human factors and types of input devices [45]. Models and heuristics have been proposed to optimize keyboards for one-finger pointing performance, chorded finger-movement performance, touch typing performance, reduction in muscle fatigue and strain, and ideal motor complexity. The usability of a GUI is similarly determined by a multitude of factors. However, more emphasis is put on attention, perception, and cognitive aspects of use, such as learning and navigation, alongside experiential aspects (esthetics, etc.).

A. Mathematical Representation of Evaluative Knowledge

1) Boolean Functions and Equality Constraints: Design heuristics are rules of thumb used in design [126], such as "white space should be minimized," "pixel density should be balanced" [127], or "feedback should be provided for user actions." Hundreds of design heuristics are presented in the literature—with the list growing constantly [128]–[130]—and have been used for menu optimization [131] and graphic design (e.g., poster) optimization [63]. In a pioneering article, AIDE [59] implemented heuristic objectives of efficiency (path length for moving between elements in optimal versus candidate layout), alignment, balance (the weight of one side relative to the other), and (designer-provided) constraints for widget layout design.

Mathematically, some design heuristics are Boolean functions. In the presence of a feature, they yield 1, returning 0 otherwise. The function implicitly partitions the candidate set of designs into heuristic-adhering $\boldsymbol{X_H} \in \boldsymbol{X}$ and nonadhering sets. Some design heuristics can be expressed by means of equality or inequality constraints

$$g_i(\boldsymbol{x}) = c_i, \quad \text{for } i = 1, \ldots, n \text{ equality constraint}$$
$$h_j(\boldsymbol{x}) \geq c_j, \quad \text{for } j = 1, \ldots, n \text{ inequality constraint.}$$

Here, g and h are constraints to be satisfied by a design.

Heuristics have two major drawbacks. The first involves conflict resolution [28]: Often, multiple heuristics are needed for attacking a real-world problem. Because each heuristic influences only a few of the decisions, weights or conflict-resolution rules must be introduced in larger numbers [64]. The optimization system may become fragile—a small change in weight or rule may produce vast differences in results. The second issue is validity. As a product of design practice rather than scientific inquiry, many heuristics are subjective and their predictive validity is questionable [132].

2) Linear and Nonlinear Regression Models: Linear models express a predictor of interest (y) as a linear function of the design variables of interest

$$y = \beta_0 + x_1 + \cdots \beta_n x_n. \tag{6}$$

Nonlinear models generalize this

$$y = f(\mathbf{x}, \boldsymbol{\beta}) \tag{7}$$

where y is an outcome variable of interest, $\boldsymbol{x}$ is a vector of design variables or derivatives thereof, and $\boldsymbol{\beta}$ is a vector of coefficients. For example, search–decide–point (SDP) is a regression model predicting task completion time in menu selection [133]. It postulates a nonlinear term for decision time and a linear term for search time. As the user becomes more experienced with a menu, a calibration parameter expresses a shift from serial visual search to a decision among competing elements. Statistical models such as SDP provide a stronger link between design variables and predicted user performance.

3) Image Metrics: Image metrics are computer programs that output quantitative values for a graphical rendering of interface design. One example of esthetic-related metrics is related to harmonic colors [134]. The system defines a set of colors that combine to produce a perception of a user interface as visually pleasing. Harmony is determined by relative positioning in the color space rather than by specific hues. Templates, provided to test any given set of colors against the harmonic set, consist of one or two sectors of the hue wheel, with given angular sizes. They can be arbitrarily rotated to create new sets. The distance of an interface from a template is computed from the arc-length distance between the hue of an element and the hue of the closest sector border, and from the element's saturation channel. Objective functions in this class are often expensive to compute. Recent advances in deep learning may make it possible to evaluate image metrics in milliseconds [135]. A recently published repository provides a collection of numerous image-based metrics [136].

4) Simulations: Simulators are step-wise-executed functions $M(\theta)$ that map the model's θ parameters to behavior-related predictions. The θ parameters capture characteristics of the user, task, design, or context. If random variables are included in the simulator, its outputs can fluctuate randomly even when the θ values are fixed. Fields witnessing simulators' use for combinatorial optimization so far include cognitive architectures, reinforcement learning models, biomechanical, and neuromechanical modeling, and neural network simulators.

For example, visual search of graphical 2-D layouts is complex cognitive-perceptual activity driven by the oculomotor system and affected by task and by incoming information. The Kieras–Hornof model of visual attention [137] is a simulator predicting fixation locations in looking for a given target in a given graphical layout. It is based on a set of availability functions that determines how perceivable the features of a target are from the user's current eye location. The availability functions are based on the eccentricity from the current eye location and angular size s of the target. Additive Gaussian random noise with variance proportional to the size of the target is assumed. For each feature, a threshold is computed, as is the probability that the feature is perceivable. These determine where to look next in the layout. Simulation terminates when a fixation lands on the target.

Simulators are generative models in the sense that they generate not just a prediction of outcome (e.g., task completion time) but process traces also. They predict not just the outcomes of interaction but their joint production and emergence in interaction with design. Furthermore, most simulators in the HCI field include causal mechanisms that link design variables to outcomes in interaction. This is valuable in design, since one does not merely output predictions for aggregates but also can examine predicted interaction in detail. Another benefit is high representational power: simulators can capture the relationships among multiple factors. Some simulation models are Turing-strong formalisms with memory and the ability to execute programs. These benefits come at the expense of computational efficiency. Certain optimization methods, in particular, most of the exact methods, are practically ruled out as solvers. For example, for a recent application of reinforcement learning with a task-completion time model [keystroke-level model (KLM)], optimization of user interfaces extended to only five states [138].

VI. DESIGN TASK DEFINITIONS

This section surveys definitions of GUI design tasks. The definitions build on the elementary IP tasks reviewed in Section IV and references made to objectives defined in Section V.

A. Functionality Selection

Given a set of functionality candidates V, the task is to select a subset $\mathcal{X} \subseteq V$ that end users find useful, satisfying, and easy to use and that is profitable for the developer. This is a selection problem with $2^n - 1$ possible designs.

A formulation was proposed recently for optimization against four criteria [39]

$$\max(U|S|E|P).$$

The objectives are introduced below.

Usefulness, or U, is modeled thus: Let u_v denote the usefulness of the individual function v. This usefulness depends not just on it but also on the presence of related functions. For example, print setup is useless without print. Therefore, we define dependencies in terms of the presence and absence of features; that is, let u_{vw} denote the usefulness of function v when w too is selected, and

let $u_{v\bar{w}}$ denote the usefulness of v in the absence of w. The total usefulness for $\mathcal{X}$ is given by

$$\begin{aligned} U &= \sum_{v\in\mathcal{X}} \left[\sum_{w\in\mathcal{X}} u_{vw} + \sum_{w\notin\mathcal{X}} u_{v\bar{w}} \right] \\ &= \sum_{v\in V} x_v \sum_{w\in V} u_{vw}x_w + u_{v\bar{w}}(1-x_w) \\ &= \sum_{v\in V}\sum_{w\in V} \bar{u}_{vw}x_v x_w \end{aligned} \tag{8}$$

where $\bar{u}$ is effective usefulness as demonstrated by Oulasvirta *et al.* [39]. Note that, in general, the dependencies are not symmetrical; e.g., print still makes sense without print setup.

Satisfaction S is defined as the sum of the satisfaction scores for the functions included in a design

$$S = \sum_{v\in\mathcal{X}} s_v = \sum_{v\in V} s_v x_v. \tag{9}$$

This score refers to the subjective experience of the functionality, as opposed to usefulness.

Ease-of-use value E can be defined as the aggregated ease of use over all functions included in the design

$$E = \sum_{v\in X} e_v = \sum_{v\in V} e_v x_v \tag{10}$$

where e_v denotes the ease of use of a single function.

Profitability p_v of function v is defined in terms of its business value $v_v \in \mathbb{R}$ and costs $c_v \in \mathbb{R}$: $p_v = v_v - c_v$

$$P = \sum_{v\in\mathcal{X}} p_v = \sum_{v\in\mathcal{X}} (v_v - c_v) = \sum_{v\in V} (v_v - c_v)x_v. \tag{11}$$

Note that if $c_v > v_v$, profitability is negative. Including functions that are of negative profitability is justifiable in light of the other parts of the objectives.

The ILP can be defined as follows. We allow directional dependencies; that is, we consider whether the dependence of a function v on a function w is so strong that having v but not w would yield an objective value that is worse than the trivial solution of selecting all functions. As in [39], the canonical integer programming model with directional dependencies has constraints of the form $x_w \leq x_v$ for all features w that depend on feature v; that is, $x_v = 0$ implies $x_w = 0$ for all dependent features w. Note that these constraints are not active if $x_v = 1$ and thus indeed model unidirectional dependencies. Another possibility is to extend the model of [39] by defining a regret value $u_{vw} \geq 0$, which penalizes the objective value of a solution that selects the dependent feature w without v. Let $b : V \to \mathbb{R}$ denote the aggregated value of each function. Note that a value may be negative, if its development is more expensive than its payoff. However, we might still choose a feature with a negative value to satisfy dependencies that enable obtaining a larger total payoff. Accordingly, the complete ILP is this

$$\begin{aligned} \max\ & \sum_{v\in V} b_v x_v - \sum_{w \text{ depends on } v} u_{vw}y_{vw} \\ \text{s.t. } & x_w - x_v - y_{vw} \leq 0 \quad \forall v,w \in V : w \text{ depends on } v \\ & x_v \in \{0,1\} \quad \forall v \in V. \end{aligned} \tag{12}$$

This ILP can be solved efficiently in polynomial time by exploiting its structural properties. The ILP (12) is equivalent to its linear program relaxation

$$\max\{b^T x - u^T y : A^T x - y \leq 0,\ 0 \leq x \leq \mathbb{1}, y \geq 0\}$$

where A is the node-arc-incidence matrix of the directed graph describing the dependencies. The dual of this problem is

$$\min\{\mathbb{1}^T z' : Az + z' \geq b, 0 \leq z \leq u\}$$

which is used to show that the linear program relaxation is equivalent to a min cost flow problem and can be solved in $O(nm \log n + n^2 \log^2 n)$ time [139]; that is, within the same timebound as the simpler model in [39].

Oulasvirta *et al.* [39] developed an exploratory tool using robust optimization to find diverse design candidates for a stated functionality selection task.

B. Label Selection

A similar approach can be adopted for the problem of label selection. Given a set of commands, each with one or more alternative labels or command names, the task is to select a set that is memorable, consistent, and quickly typed. Formally, we are given a set U of commands as well as a set L_u of possible labels for every command $u \in U$, the goal being to assign every command to one of its possible labels. The naming is modeled by binary variables $x_{u\ell}$ having a value of 1 if and only if we label command u with label ℓ.

We consider three objectives that we want to synchronously optimize. First, the labeling should be memorable, meaning that one should find it to be intuitive and easy to understand, the purpose of a particular command when given its label. We model this by a value $m_{u\ell}$ for every command $u \in U$ and label $\ell \in L_u$, which describes the difficulty of memorizing command u with label ℓ. For the sake of consistency, we interpret smaller values of $m_{u\ell}$ to be easier to memorize. Therefore, we will minimize this objective function. Second, we want the labels chosen to be fast-to-type ones. We can easily measure the time it takes for a certain label to be typed on the underlying keyboard and save the value for every label $\ell \in L$ in t_ℓ.

The last objective requires consistency of the labeling of two separate commands. In the simplest setting, this consistency is independent of the actual commands, so we only compare the consistency of the chosen labels. In this case, we have distance values $d_{k\ell}$ for every pair of labels $k, \ell \in L$ that we want to minimize. The resulting optimization problem can be written as a quadratic problem of the following form:

$$\begin{aligned} \min \quad & \left(\sum_{\ell \in L} t_\ell y_\ell \;\middle|\; \sum_{u \in U} \sum_{\ell \in L_u} m_{u\ell} x_{u\ell} \;\middle|\; \sum_{k,\ell \in L} d_{k\ell} y_k y_\ell \right) \\ \text{s.t.} \quad & \sum_{\ell \in L_u} x_{u\ell} = 1 \quad \forall u \in U \\ & y_\ell \geq \sum_{u \in U_\ell} x_{u\ell} \quad \forall \ell \in L \\ & 0 \leq y_\ell \leq 1 \quad \forall \ell \in L \\ & x_{u\ell} \in \{0,1\} \quad \forall u \in U, \ell \in L. \end{aligned} \tag{13}$$

Note that we introduced the auxiliary variables y_ℓ for every label $\ell \in L$, denoting whether the respective label is selected by any command or not. It is sufficient to require nonnegativity of the auxiliary variables because any optimal solution to this problem (13) will contain only binary values for y. We also introduced the set $U_\ell := \{u \in U | \ell \in L_u\}$, in order to simplify the formulation of the second constraint. While the first constraint guarantees each command being assigned exactly one of its possible labels, the second constraint links the x and y variables. Lastly, we make sure that no label is taken more than once.

In a more complex scenario, we consider the similarity of commands also with regard to consistency. We want to achieve more consistent labeling of similar commands, on the one hand, while, on the other, giving less of a penalty to inconsistent labels for nonsimilar commands. In this scenario, we are given similarity measurements s_{uv} for two commands $u, v \in U$. The resulting minimization problem comprises the same constraints as above (13), but the objective function is slightly more complex

$$\left(\sum_{\ell \in L} t_\ell y_\ell \;\middle|\; \sum_{u \in U} \sum_{\ell \in L_u} m_{u\ell} x_{u\ell} \;\middle|\; \sum_{u,v \in U} \sum_{k,\ell \in L} s_{uv} d_{k\ell} \; x_{uk} x_{v\ell} \right). \tag{14}$$

The number of terms in the third part of the objective function, which is already the hardest part of the previous problem (13) on account of its quadratic nature, is increased by a factor of $|U|^2$. This optimization problem is a special case of the quadratic assignment problem, which is computationally very hard, as we discussed in Section III-C.

C. Icon Selection

A similar approach can be used for icon selection. Comprehensibility and identifiability of icons are two central goals in the design of command labels and icons, as noted by Laursen *et al.* [140], who formulated icon set design as an optimization problem. Candidate icons are assumed to belong to function sets, with one from each getting selected for the final set. The authors presented a crowd-based approach for obtaining input values indicating comprehensibility and identifiability. The task was then to choose a set that is maximally comprehensible with identifiable items that are visually distinct from each other. This can be done in $\mathcal{O}(m^n)$, where n is the number of function sets and m the number of candidates per set.

D. Widget Selection

In menus and simple push-button panels, elements are often homogeneous. With advanced software libraries for GUIs, a designer often has multiple types of widgets and controls to choose from. In the widget selection problem, there are multiple widgets, or widget types, but there is only a limited-size canvas to place them on. Two aspects of this problem can be identified.

The first involves the designer specifying a minimum set of requirements for an interface, where there are multiple widgets to select from, each with distinct properties. For example, there may be a requirement that the user selects one option out of many. This may be implemented via, for example, a menu or a drop-down list, where each option has its associated "costs" in terms of learnability, usability, use of display space, etc. A good solution to this problem covers all the requirements at a minimum cost. This is a practical example of a covering problem, a class discussed in Section III-A. In the second part of the problem, the goal is to find a selection of elements that, without exceeding the capacities, maximizes some positive value, such as usefulness to end users. Again, it may or may not be permissible to select an item multiple times. This is a classical packing problem. The formal details are described in Section IV-A.

E. Menus

Menus are widely used interfaces for command exploration and invocation. Here, we look at menus where options are presented visually and selected by means of a pointing device. From the combinatorial viewpoint, the task can be formulated as assignment [58], [141]: For a given set of commands N and set of positions P, the task is to assign every command $i \in N$ to a unique position $p \in P$. Three evaluation criteria are used. 1) Fitts' law is used to calculate and minimize the selection time required to move the cursor to any intended command i. 2) Association score α_{ij} for any pair of commands $i, j \in N$ is used as a weight for identifying and minimizing the gap between closely related elements. While the one-to-one assignment between sets N and P would usually entail a classical assignment formulation, the two-part (Fitts' law and association) objective function leads to a novel optimization problem. 3) Some way of controlling for group or

tab size is introduced, such as the Hick–Hyman law, which links selection time to the number of commands in a group [ST$= \log_2(N)$].

A generic version of the menu optimization problem can be formulated in terms of the classical assignment decision variable x_{ip}, denoting whether or not command i is assigned to position p. In conjunction with x_{ip}, we define the time cost T_p required to navigate to position p and frequency F_i of command i. Then, the selection time criterion is defined as follows:

$$S = \sum_{i \in C} \sum_{p \in P} F_i T_p x_{ip}. \tag{15}$$

By this criterion alone, commands with higher frequencies would be placed in the top positions. The other criterion is Association, A, aimed at placing the related commands near each other to help the user find them easily. For example, the commands for copying, cutting, and pasting are closely related to each other. We define the parameter $\alpha_{i,j}$ to indicate the logical correlation or association of commands i, j. Also, we define decision variable $y_{i,j}$ for indicating that these two commands are located next to each other. By means of this variable, the overall association score for a given menu layout is quantified as follows:

$$A = \sum_{i \in C} \sum_{j \in C} \alpha_{i,j}\ y_{i,j}. \tag{16}$$

Selection Time and the Association score are the two important criteria in the design of various kinds of menus, discussed in Sections VI-E1–VI-E3. For further details on evaluation criteria, the interested reader is referred to the work of Ramesh *et al.* [81].

1) Linear Menus: Linear menus are organized in a single row or column with no internal substructure such as grouping. Each command is assumed to have some association with the positions for "before" and "after" the command. The following constraints constitute one option for linking positioning variable x_{ip} to the association variable defined earlier, y_{ij}:

$$y_{i,j} + 1 \geq x_{i,p} + x_{j,p+1} + x_{j,p-1} \quad \forall i, j \in C;\ p \in P \tag{17}$$
$$x_{i,p-1} + x_{i,p+1} + 1 \geq y_{i,j} + x_{j,p} \quad \forall i, j \in C;\ p \in P. \tag{18}$$

The objective function typically utilized is the weighted sum of selection time [see (15)] and Association [see (16)].

2) Grid Menus and Toolbars: Now consider toolbars and menus arranged in grids. We have rows, $r \in M$, and columns, $c \in N$, so each position is defined by (r, c). Then, $x_{i,r,c} = 1$ if and only if command i is placed on row r and in column c.

The association of each position with those surrounding it (e.g., top, down, right, left, top-right, top-left, down-right, and down-left) should be calculated. For example, if we use binary decision variable $\gamma_{i,j}$ to indicate whether or not command j is on the right side of command i, the following constraints will identify this scenario:

$$\gamma_{i,j} + 1 \geq x_{i,r,c} + x_{j,r,c+1} \quad \forall i, j \in C, r \in M,\ c \in N \tag{19}$$
$$x_{i,r,c} + 1 \geq \gamma_{i,j} + x_{j,r,c+1} \quad \forall i, j \in C, r \in M,\ c \in N. \tag{20}$$

Similar constraints can be defined, analogously, for other positions. The objective function for a grid menu is similar to that for a linear menu, but in place of a vertical/horizontal list there must be calculated grid positions, and the binary variables for a linear menu must be replaced with those for a grid menu.

3) Hierarchical Menus: Matsui and Yamada [141], [142] represented a hierarchical cell-phone menu as a tree structure. They used simulated annealing (SA) to optimize it for selection time with a mixture of logarithmic pointing time and number of items. Grouping was controlled with a function similarity term and a menu granularity term that penalizes having a large number of descendants. A generalization of this problem is developed below.

Larger menu systems (involving 20+ elements) require some type of *hierarchical organization*, typically achieved via tabbing, groups, folding, cascades, or submenus. Classical approaches for hierarchical menu design use assignment-based formulations [58]. However, as is clear from examining outputs published by Bailly *et al.* [58], assignment alone does not sufficiently represent the organization of individual elements within larger entities. Rather, it leads to frequency-ordered groupings that often ignore the top-level commands' effect of representing the rest of the menu; the topmost items are not necessarily a good indicator of the remaining content of the relevant menu.

This challenge can be addressed by means of an evaluative function based on information foraging theory (IFT) [81], [143]. An IFT-based objective enables assessing search performance in the case of groups. In the context of hierarchically organized menus, it offers a quantitative model of a rational but time-limited agent that navigates a hierarchy composed of discrete areas or sets. The agent must decide whether to explore the current set of commands (group/tab) or instead abandon/skip this set in favor of the next. When used as the objective of an optimizer, that translates to evaluation and minimization of the time wasted by the user in the irrelevant parts of the menu. This minimization results in layouts that place semantically indicative items at the top of the menu.

Menu design with the IFT-based approach can be implemented via MIP by formulating a sample–discard–explore paradigm. This paradigm captures the four logical outcomes that are possible during any search process.

1) *True Positive:* The user guesses that the set contains the target command, and it indeed contains that command. In this case, the cost during search within the set is the time consumed (per Fitts' law) to scan the list and move the pointer over the required command in the set.
2) *True Negative:* The user guesses that the current set does not contain the required command, and the set indeed does not contain it. No further cost is incurred.
3) *False Positive:* The user guesses that the set contains the required command, but it actually does not. The additional cost incurred for this set is the time consumed (under Fitts' law) to navigate all commands in the set. This cost is proportional to the size of the set.
4) *False Negative:* The user guesses that the set does not contain the required command, but in reality it does. Now the user must fruitlessly analyze all succeeding sets, such as subsequent groups on the tab.

We assume that the user begins the search by sequentially analyzing (sampling) the lead elements of every set encountered. On the basis of the decision made to discard or explore any specific set, the user invests the corresponding effort for that set. This process repeats until a true positive (target) is reached. This logic can be applied recursively at any level of a hierarchy where multiple options (sets) are available. In our application, we assume two levels: tab and group. The insight is that the total time expended in locating a specific command is the summation of time spent in the four possible scenarios, weighted by the probability of the user making the corresponding decision for the relevant set. To obtain an estimate for the entire menu structure generated by an optimizer, the estimated costs are further weighted by the frequency of use.

An integer programming formulation based on the above model requires the following decision variables:

$$u_c^j = \begin{cases} 1 \ldots \text{if command } j \text{ leads group } c \\ 0 \ldots \text{otherwise} \end{cases}$$

$$\Phi_i^c = \text{Total time / cost for command } i \text{ computed for group } c$$

$$x_i^c = \begin{cases} 1 \ldots \text{if command } i \text{ is placed in group } c \\ 0 \ldots \text{otherwise} \end{cases}$$

$$r_i^r = \begin{cases} 1 \ldots \text{if command } i \text{ is placed on the } r\text{th row} \\ 0 \ldots \text{otherwise} \end{cases}$$

$$t_i = \text{The time required to reach command } i$$

$$p^c = \text{Starting position (row number) of group } c \text{ within its tab.}$$

Decision variables such as x define the unordered structure of the groups and tabs. The absolute positioning of commands is provided by such variables as r. In addition, further connecting decision variables will be required, for guaranteed sanity of the overall mathematical model.

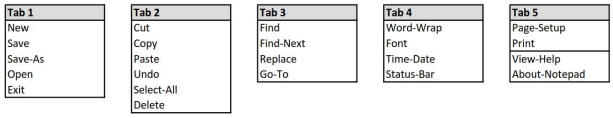

Fig. 3. ***Menu generated for the Windows Notepad text editor via an information-foraging-based approach [81].***

The objective is to minimize weighted cost Φ (the weighting is for frequency of use) for the time taken to reach any command placed within any set

$$\text{Min} \sum_{i \in N} \sum_{c \in C} \mathbb{F}_i \Phi_i^c$$

where $\mathbb{F}_i$ is the frequency of usage of command i in a set N of commands to be subdivided into groups C. The resulting MIP formulation can find optimal menu layouts for a typical data instance (40–50 commands) within around 20–30 min of computational effort. As an example, the optimized menu layout for the classical Windows Notepad application is shown in Fig. 3.

F. Grid Layouts

Grids are an important organizing principle for GUIs [42]. A design grid divides a display via grid lines for assignment of elements: containers, buttons, *etc*. Hart and Yi-Hsin [46] presented an ILP to solve the rectangular packing problem in GUI layouts. Their formulation takes care of ordering of items on a display and avoids infeasible solutions that include overlap, overflow, or illegal orientation of an element. It assumes a predefined grid layout with columns and introduces constraints for making sure elements are presented no more than once, are not clipped, respect the maximum and minimum size specified, and occur in the predefined columns. An efficient branch-and-bound solver was developed.

This section extends earlier layout formulations toward a generic definition. In terms of optimization problems, addressing the topic involves the following factors.

1) *Esthetics:* Overall, end users prefer a grid-like arrangement with generally rectangular placement.
2) *Semantic Association:* Mutually interrelated widgets are preferably grouped together, in close proximity.
3) *Relative Placement:* Some elements may need to be at a specific side of another element. For example, the *Search* button for an Internet search-engine web site is expected to be located to the right side of the search-text field in the English locale.
4) *Multiple Near-Optimal Solutions:* The optimizer is expected to produce a wide variety of dissimilar layouts, so that the designer can choose from diverse options. This option to choose from a broad range of varied layouts induces a mathematical problem of finding a set of diverse layouts instead of a single optimal solution.

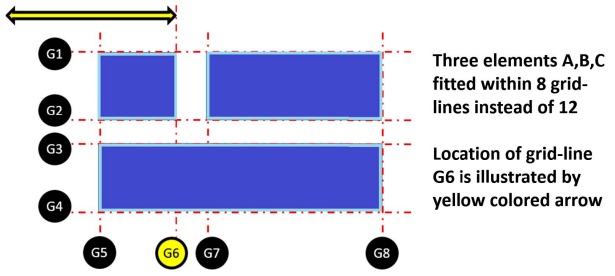

Fig. 4. ***Usage of grid lines to define placement of all elements with the MIP approach.***

5) *Computational Performance:* Designers prefer not to wait an inordinate amount of time for computation of layouts. The formulation must be compact enough to yield near-optimal results within the reasonable computational effort.

For related work in constraint-based approaches to text document layouts, (see the article by Hurst *et al.* [19]).

It is possible to address these challenges with a limited computational effort by using a nondiscretized placement model [30]. The position and size of every element i is modeled by continuous variables ℓ_i, t_i (representing the top-left corner) and b_i, r_i (representing the bottom-right corner). We do not discretize individual pixels, so the formulation size is independent of the size of the canvas. It is easy to enforce direct constraints on ℓ, r, t, b to satisfy sizing requirements and also prevent the overflow of elements. Overlap avoidance is assured through constraints imposed in formulae 1–5. Having made sure that a nonoverflowing, nonoverlapping layout can be obtained via a nondiscretized formulation with good computational performance, we now enforce the other placement objectives. To guarantee grid alignment, we construct a (vertical or horizontal) grid line to cover every edge of every element being placed. It is possible (and desirable) that some grid lines will match the edges of more than one element; in that case, we refer to the relevant elements as edge-aligned with each other. For a problem involving n elements (resulting in a $4n$ edge count), a maximum of $4n$ grid lines is required. In the mathematical construction, we allow all of these $4n$ grid lines in the model; however, we introduce a decision variable to specify whether a specific grid-line is used or not. Furthermore, we introduce decision variables to specify the locations of the grid lines actually used (the X- or Y-coordinate axis intercept of each particular grid line). This concept of grid lines is illustrated in Fig. 4.

The figure shows a layout of three elements represented by means of eight grid lines; four grid lines have not been used. This fact can be captured by using 12 binary decision variables (for a grid line being used or not). Furthermore, the location of grid line $G6$ is shown with the yellow arrow on the top-left corner. This location can be captured by a continuous decision variable. Thus, eight continuous decision variables are sufficient to specify the complete grid structure of Fig. 4. An MIP model working with the objective of minimizing the total number of grid lines actually used will automatically result in a well-aligned grid-like structure for the overall layout (for examples, see Fig. 5).

Another concern within the realm of esthetic factors is that of having an overall rectangular outline for the entire layout. This is not automatically addressed with the foregoing discussion of grid-lines. For example, consider Fig. 6, showing two separate solutions for a five-element problem instance. Both solutions use seven grid lines, but the solution in the right panel has a clean rectangular outline while the solution on the left has a jagged outline. Clearly, the summation for the number of grid lines cannot, on its own, address the esthetic requirement of a rectangular outline.

To resolve the rectangularity issue, consider Fig. 7, which shows a candidate solution with resulting actual overall jagged nonrectangular outline (marked with a red outline) on the left. The ideal rectangular outline is formally identified by the four extremal grid lines (red lines) shown in the right-hand panel. We note that the nonextremal or intermediate grid lines (the black lines in the bottom image) do not play any part in defining the overall rectangular outline. In fact, any element edges aligned with the intermediate (black) lines may potentially cause jagged nonrectangular outlines. Therefore, the rectangularity objective effectively means that the difference between the red-colored jagged polygon (on the left) and the red rectangle (on the right) is to be minimized or eliminated.

Applying this intuition, we augment our MIP formulation as follows: For every element i, we define four binary decision variables to specify whether the four edges of the element align with any intermediate (black) grid line or with any extremal (red) grid line. In the objective function, we penalize every case wherein an element edge aligns with any intermediate (black) grid line. Conversely, we reward every case in which an element edge aligns with any extremal (red) grid line. Returning to Fig. 6, from earlier, we see that the newly defined objective function will penalize the jagged layout on the left side and will prefer the clean rectangular layout on the right.

Next, we consider the proximal collocation requirement we discussed for semantic associations (#2 above). To address this, we define λ_{ij} as the mutual semantic association between elements i and j. The λ_{ij} values are defined in the range 0–100, with higher values indicating greater association. Also, we utilize decision variables h_{ij}, v_{ij} to indicate the horizontal and vertical distance between two elements i and j. Then, the following objective function dictates that the closely interrelated elements be placed in close proximity to each other:

$$\min \sum_{i \in N} \sum_{j \in N} \lambda_{ij} \frac{h_{ij} + v_{ij}}{2}. \tag{21}$$

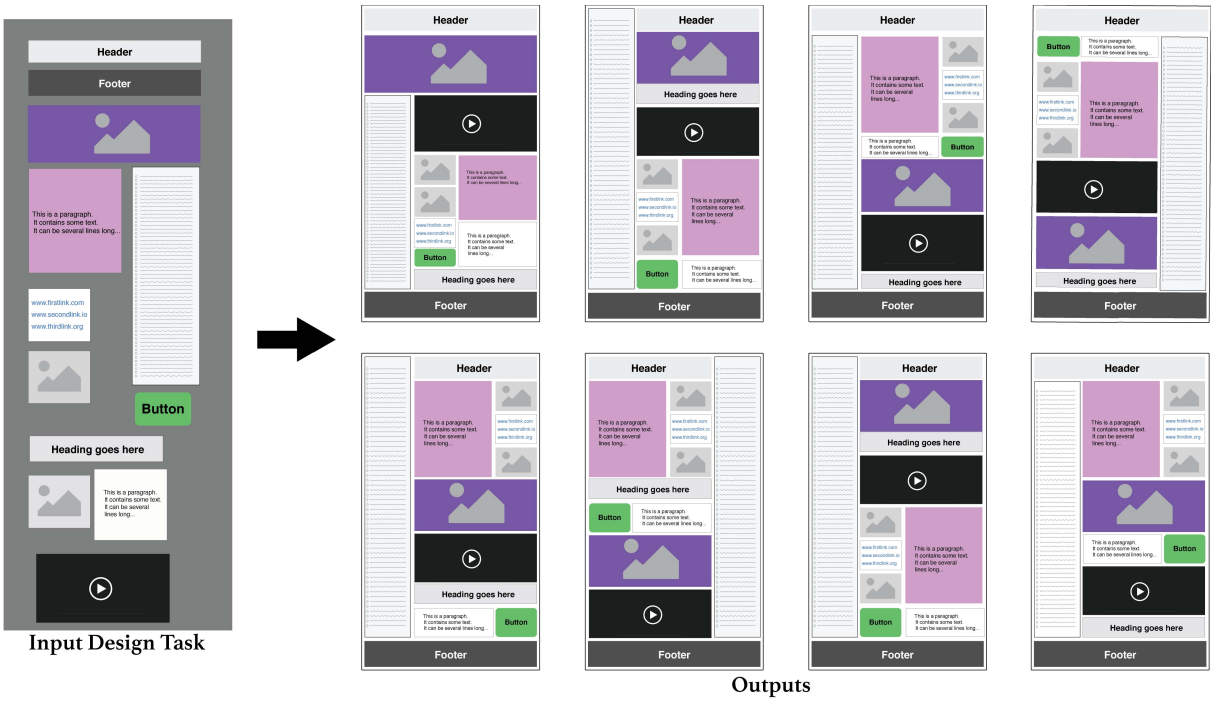

Fig. 5. ***Example grid layouts generated for a web wireframe problem instance [30].***

Requirement #3, dealing with the relative positioning of a pair of elements, is easily addressed by forcing the $\mathbb{A}$ or $\mathbb{D}$ variable to be nonzero. Furthermore, it is possible to situate the elements in the immediate vicinity of each other by adding substantially greater weight to the coefficient of the relevant distance term in the objective function.

The most interesting challenge faced in layout problems is to guarantee that multiple, varied (near-optimal) solutions are provided by the optimizer. The classical MIP approach is typically geared for computing only one optimal solution. Commercial ILP solvers do provide options to generate a wider solution pool, but the solutions in this pool are often indistinguishable (not sufficiently different from each other) for a human user. For example, the Gurobi optimizer considers two solutions to be different if at least one discrete decision variable takes a different value. But the resulting impact from only one decision variable is often negligible for the end user.

We resolve this matter via the following reasoning: For the case where the summation of all $\mathbb{A}$ decision variables is larger than that of all $\mathbb{D}$ decision variables, the solutions will be radically different from the converse case. Accordingly, the diversity of potential solutions can be captured by enforcing different metrics for $(\sum \mathbb{A}/\sum \mathbb{D})$. If the maximum and minimum values of the metric are deduced, then the candidate solutions can be distributed across these known limits. Candidate solutions thus calculated for constrained values of $(\sum \mathbb{A}/\sum \mathbb{D})$ are sufficiently distinct from each other. Fig. 5 shows a set of GUI grid wireframes generated in line with this approach.

G. Distributed Interfaces

Consider a collaborative environment such as a meeting or workshop wherein multiple participants can access devices of multiple types (handheld, computer-screen, large-display, etc.). The resulting plethora of input and output channels can be better utilized if the GUI application automatically distributes its underlying elements among the devices in an efficient and controllable manner.

The automatic distribution of user interface elements across multiple devices can be modeled through mathematical models [76]. The MIP formulation used here utilizes decision variable x_{ed} to indicate that element e is shown via device d. Another decision variable, o_{eu}, indicates whether element e is accessible to user u. Finally, s_{ed} specifies the area of element e on device d. The objective is to maximize the perceived user satisfaction. However, user satisfaction can be quantified in a variety of ways. For example, consider a parameter c_{ed} encapsulating the value realized by displaying element e on device d. Another parameter, a_{ue}, indicates the relative interest of user u in accessing element e. Finally, parameter b_{ud} indicates whether or not user u can access device d. The decision variables listed above are sufficient to model most of the potential metrics.

VII. DEPLOYMENT

This section surveys practical advice on developing and deploying optimizers for GUI design. Fig. 8 shows six interrelated themes that must be addressed in deploying combinatorial optimization to that end:

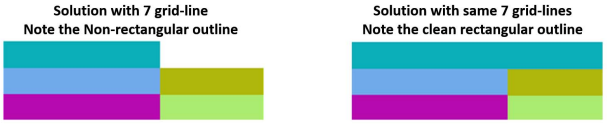

Fig. 6. *Rectangular outlines not achieved by grid-line minimization alone.*

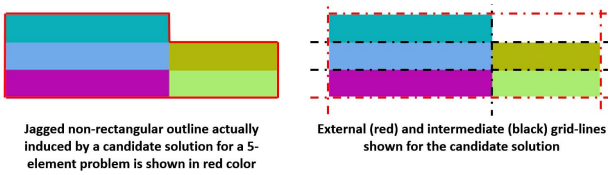

Fig. 7. *Overall rectangular outlines can be achieved if the element edges align more with the extremal (red) grid lines than the intermediate (black) grid.*

1) design space;
2) evaluation function;
3) task instance;
4) multiobjective solution;
5) solver;
6) empirical validation.

This section is geared toward HCI researchers interested in practical applications. However, the discussion in Sections VII-E and VII-F may be of interest for computer scientists, since some topics are specific to GUI design as an application field. Section VII-F is general in nature.

A. Design Space

Numerous methods in the field of combinatorial optimization are available for expressing a design space. For example, placement decisions (locations) are generally discretized to a grid, and sequences of actions (modeled as nodes on a graph) induce Hamiltonian cycles over arcs (sequencing), typically represented by immediate precedence variables. In GUI design, there is often a variety-rich set of solutions that perform comparably well for the stated objectives. However, the final impact of any modeling decision is not immediately evident. Many permissible alternatives for a certain key decision may lead to good (near-optimal) solutions that all are quite different from each other.

Also, the objective functions in GUI design often show great differences from classical OR. For example, human users seldom can discern very small changes in position, size, or color. Hence, the optimizer need not traverse decision trees to such an extent. With these points in mind, we arrive at two considerations when modeling a design space.

1) If a specific metric for the design space can be discretized to a large number of decision variables, then discrete decision variables should not be used to represent it; continuous decision variables should be used wherever possible. Discretization would lead to a loose and voluminous formulation.
2) Generally, discrete variables should be chosen for decisions that will substantially reduce the search space within a short traversal of the decision tree. Often such decision variables may not be of interest to the human user. For example, the variables $\mathbb{D}_{ij}$ in Section VI-F and U in Section VI-E (under 3) are of little interest to human users, but deciding for these variables substantially restricts the remaining search space.

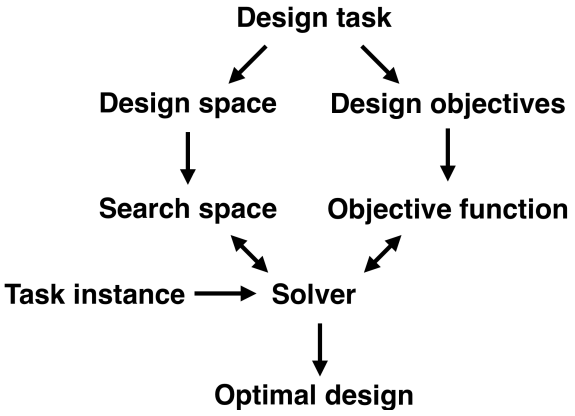

Fig. 8. *Development of an optimization approach for practical deployment in design.*

B. Evaluation Function

An evaluation function can be modeled as an objective or with constraints. The latter allows us to model a wider class of objective functions than just linear ones—for example, convex functions over a discrete domain via raising to a higher dimensional space. The key consideration is the tradeoff among computational performance, model validity, and representational flexibility. While certain models reviewed in the previous section can be modeled as hard constraints and others as soft constraints, some have been implemented programmatically as simulators and cannot be reformulated as constraints. From the optimizer's perspective, they are essentially black boxes. Moreover, most simulators are stochastic and require multiple executions, to build a reliable estimate. The computational cost per evaluation may be too high for practical purposes.

C. Task Instance

"Task instantiation" refers to the provisioning of parameter values that describe the specific design problem at hand. For example, a task instance can express a list of commands that a menu optimizer must organize into a menu, together with scores for association between the commands and their relative priorities for a user. Real projects may involve tens or even hundreds of input

parameters, including: 1) the to-be-considered elements of a design; 2) relative weights of objectives in the evaluative function; 3) empirical parameters of evaluative models; and 4) governing hyperparameters to tune the solver.

Task instantiation is often tricky because the task cannot be precisely defined at the outset: Designers may not know which elements to include or what their types should be. Relative weights too are at the designer's discretion and are hard to determine without exposure to outcome quality. Results may be sensitive to minute changes in input parameters. Finally, setting parameters for the solver necessitates some understanding of solvers' algorithms; this often lies outside the GUI designer's domain.

The literature distinguishes five approaches, which are somewhat complementary.

1) Specification by demonstration is the process wherein a task instance is obtained by creating a full or partial solution. For example, one can draw a full layout for a workspace, which can then be worked on by the optimizer. Alternatively, some elements may be locked or otherwise marked as out of bounds for the optimizer [30].
2) In the process of design mining, weights, objectives, or constraints are learned from data [48], [63], [144]–[146]. For example, a probabilistic model can be developed to represent locations of headers on web pages. One can then bias an optimizer to adhere to a prescribed style. The domain-free and domain-bound aspects of the resulting multiobjective task have to be calibrated, so that the learned domain model cannot dominate the outcomes. Another known challenge is overfitting to training data and out-of-distribution predictions.
3) Robust optimization assumes that the input values either change or are uncertain [147]–[149]. Several techniques exist, only one of which, the Monte Carlo method, has been attempted in GUI design. An explorative optimization method has been presented for the functionality selection problem [39]. On the basis of human-expressed uncertainty in input data, multiple optimization tasks are generated and handled individually. Designer-stated levels of input values' uncertainty inform distributions of input weights (as opposed to point values), and several task instances are generated accordingly. The resulting candidate solutions are then mined for novel or robust designs. Thus, the most probable interpretations from the designer's input get represented in the outputs. However, this approach depends on efficient per-instance solving.
4) Online learning enables arriving at parameter values via online experiment design. Key parameters' values are tried out with a selected group of experimental users, and outcome quality is measured. One can use Bayesian optimization to obtain the values [150]. See Koyama and Igarashi 's work [151] on applications in crowd-based parametric design optimization.
5) Finally, empirical parameter-fitting addresses the frequent need for fitting evaluative functions' coefficients to the case. For example, the two empirical parameters of Fitts' law (a and b) depend on the input device and user group. If these parameters are invalid, incorrect tradeoffs between element size and location propagate to the optimization. In HCI more generally, parameter inference may be hard, especially if the model lacks an easily computable likelihood that could be maximized. Besides tuning parameters manually and adopting values from the literature, common fitting methods include grid search and local optimization. To this end, representative observations from the target domain are needed. Bayesian optimization has been introduced recently [152], using a proxy model (e.g., a Gaussian process) to determine where to sample next. The benefit is that reliably estimating model parameters does not require as many observations.

D. Multiobjective Solutions

GUI design typically involves multiple objectives, comprising ergonomic, esthetic, and other criteria. The resulting objectives f_i for $i = 1, \ldots, k$ may conflict with each other. A solution that is optimal for one objective may perform poorly for others. This is a well-studied topic in the field of combinatorial optimization but considered relatively little in this application area.

One common technique is to combine the objectives into a single objective expression. This can be done in many ways—by using weighted sums, the lexicographic method, goal-programming methods, etc. [153]. The resulting problem can then be solved via well-established algorithms [154]. In the lexicographic method, one defines a hierarchical ordering of the objective functions. The optimizer then iteratively finds an optimal solution to one objective function while not degrading higher-ordered objectives. Recent generations of MIP solvers have introduced the specification of multiple objectives so that the user does not have to manually handle the various techniques discussed above. If, however, there are no preferences among the objective functions and it might, then, be unreasonable to assign weights or a hierarchy to them, so-called no-preference methods provide an alternative strategy. The goal-programming method, for example, calculates the ideal objective vector $z_i^* = \inf_{x \in X} f_i(x)$ for $i = 1, \ldots, k$ and scalarizes the objective function

$$\min_{x \in X} \|f(x) - z^*\|.$$

An alternative is to choose one function f_j as the main objective function and bound the values of the other objective functions in the constraints as $f_i(x) \leq \varepsilon_i$. This method is usually referred to as the ε-constraint method [155].

Other approaches involve metaheuristic algorithms [156]. Here, the problem is solved with all objectives and then the Pareto front of all solutions is computed [157], [158]. Among the successful algorithms used in these are Nondominated Sorting Genetic Algorithm II (NSGA-II) [159] and Strength Pareto Evolutionary Algorithm 2 (SPEA2) [160]. The most common approach in GUI computation has been the weighted sum approach, which is sensitive to objective weights.

E. Interactive Optimization

Interactive optimization refers to an optimization process that can be steered by a human during solving [55], [59], [161], [162]. Interactivity is beneficial when the design problem cannot be precisely articulated. When incorporated into a design tool, an optimizer can assist in creative problem-solving and potentially "nudge" novice designers toward better, more usable designs [28].

Meignan *et al.* [162] offer a comprehensive review of interactive optimization approaches. Search-oriented interaction lets the designer provide additional information during the optimization process. The user can take any of several roles in this process: In the assisting role, the designer modifies solutions. The guiding role entails the designer steering the exploration by affecting decision variables. In the tuning role, a designer sets solver-specific parameters. Model-oriented interaction, an alternative to search-orientation, allows the designer to modify and refine the optimization model during the optimization process. This, in turn, can be divided into two approaches. The adjusting designer can change parameters to the objective function or constraints when the objective is not fully known at the beginning of the process. The enriching designer can add objectives or constraints. The initial model is assumed to be incomplete, and solutions are invalidated or validated in light of interactive feedback. Thus far, the literature does not offer guidelines for choosing between approaches.

A subset of these techniques has been applied to GUI design. Among those for defining input are control panels [59], preference elicitation [125], constraint editing [163], and storyboarding [164]. Techniques for interacting with optimizer outputs include interactive example galleries [60], [64], [165], Pareto front visualizations [58], localized suggestions on a design canvas [58], and localized critique of design outputs [125]. During search, one can steer an optimizer by selecting promising designs (biasing the local search) [60], completing part-solutions [30], visualizing optimization landscapes for steering, and locking part-solutions that are ready [30], [58].

Besides interaction techniques, to support participation of various stakeholders, optimization should allow fast definition and redefinition of the task instance. In an iterative design process, the problem definition is evolving constantly. Also, there should be room for interactive exploration of the design space by the designer, with questions such as "what would happen if I placed the element here?" [58]. There is a need to work with tacit, "subjective" criteria as well: stakeholders often wish to make manual adjustments to proposed solutions, applying background criteria such as assumptions about users' habits, cultural specifics, and subjective preferences. A "subjective function" could be learned, then articulated and used as an additional objective in optimization of future solutions. Finally, it is important to provide explanations and justification for design choices, as opposed to merely producing a design. A stakeholder might ask "why is this element here?" or seek to understand how well the design reflects each particular objective. Techniques that reveal how changes to certain input parameters affect the design and *vice versa* could aid in such navigation of the design space and make the optimization more controllable.

F. Solvers

There are two principle approaches to solving a design problem. First, black-box methods can address the problem irrespectively of the objective function and without making explicit assumptions regarding the objective function. They treat it as a black box or an oracle that decides the objective value of a given candidate. If evaluating the objective function for a given configuration is inexpensive, random-search-based heuristics are often an effective way to obtain high-quality solutions. Such methods offer no guarantees of finding the global optimum, though. Neither do they provide any metrics for solution quality. There is no explicit criterion for stopping either, to trigger termination of the optimization process.

1) Black-Box Solvers: There are deterministic and randomized black-box methods. Though it is possible to randomly try out candidates and return the best solution after a certain time, efficiency is often greater with a more systematic approach. Greedy algorithms divide the construction of a solution into multiple, sequential decisions. These decisions are made in a certain order and are not revoked later. A typical example is packing problems, solved by picking item by item until further items cannot be picked anymore. It is natural to proceed by choosing the next item from among those that still fit and yielding the largest growth in profit relative to the current selection. The main ingredient of local search is the definition of a neighborhood for a given configuration. In GUIs, the neighborhood metric may consist of, for example, spatial distance between elements or their positions relative to each other. Starting with the initial configuration, local search proceeds by choosing a neighboring configuration that has an objective value at least as good as the current one. These choices might be, for example, greedy or randomized.

Metaheuristic black-box methods have some way of controlling lower level heuristics. For example, SA is a special case of local search with added randomness. The main difference is that a neighboring configuration may be chosen not only when it yields a better objective value but also,

with a certain probability, when it is worse. That likelihood decreases with the extent of deterioration of the objective value. Typically, the probability is $\exp(-\beta\Delta)$, where Δ is the difference in objective value and β a parameter referred to as inverse temperature. In this domain, SA has been applied for menu hierarchy design [142] and other uses. Further members of this class used in GUI design have been evolutionary algorithms [141], [166] and ant colony optimization [58].

2) Exact Solvers: If a rigorous analytical description of the design space and the evaluative function is available, exact techniques such as integer programming can be utilized. Exact methods are restricted in the objective functions amenable to them, but they are superior in terms of formal modeling capabilities and the ability to guarantee an optimal solution or a metric for solution quality. Exact methods use a structured (nonrandom) search approach that guarantees the optimal solution, provided that computational effort is sufficient. Modern commercial solvers (e.g., CPLEX and Gurobi) have proven to be efficient and flexible for addressing GUI design.

Explicit enumeration is the simplest exact method. The objective value of each element in the solution space is evaluated, and the current best solution—the so-called incumbent—is updated. In contrast, implicit enumeration makes use of relaxations that can be solved efficiently; that is, only a subset of the constraints is considered, such that the problem becomes tractable. We can thereby obtain information about a subset of the possible solutions, by, for example, partial assignment of the variables, and compare it with the incumbent. If the best objective value for the relaxation with restriction to a subset of solutions is worse than the objective value of the incumbent, we can discard all solutions from this subset at once, because none of them can replace the incumbent. Moreover, if the relaxation of a subset is already infeasible, the subset under consideration must be empty. In either case, we conclude that the negation of the conditions that qualified the subset must hold for all potential new incumbents. This additional information may be helpful to strengthen further relaxations. In addition, relaxations deliver guarantees as to the quality of the incumbent; for example, if the optimum objective value of the incumbent is only 1% away from the optimum objective value of the relaxation, we know also that it can be no more than 1% from the optimum for the original problem.

Branch and bound is one very popular form of implicit enumeration. Among the standard methods for handling integer (linear) programs, this represents search as a tree, where we make a decision at each node to partition the search space. In a simple branching strategy, one variable is fixed in each iteration. Suppose we have already found a feasible solution with an objective value of z. If we now have an oracle that tells us at some node that all feasible integral solutions in the subtree rooted at this node have an objective value of at least z, we can discard the entire subtree because it will not offer an improvement. More generally, two kinds of pruning can be done based on: 1) infeasibility (e.g., two elements cannot be assigned to the same slot) and 2) bounds, where the lower bound is worse than the objective value of the incumbent. Finding a good incumbent early in the process of implicit enumeration usually allows one to discard a larger quantity of subsets. Hence, state-of-the-art IP solvers expend significant effort in implementing heuristics for generating a new primal solution as well as for deciding which subset of partial assignments to consider next. If the designer's insight into the model to be optimized is sufficient to provide a good starting solution to the solver, this small amount of added information can be exploited to speed up the whole optimization process considerably.

3) Techniques for Improving Exact Solver Performance: Improvements in solver performance are possible with known MIP techniques. We will describe them with application examples from graphical interface generation.

First, discrete decision variables can be avoided. Consider a pair of elements i, j with no mutual association. Furthermore, other observations have led to the conclusion that these elements are sufficiently distant from each other on the canvas (say, in different quadrants of the available space). It might then be possible to make $\mathbb{A}, \mathbb{D}$ into continuous variables for this pair. In fact, it may be possible to drop these decision variables completely from the model. Post facto, if it is observed that the elements do overlap, then the variables can be added as separate cuts.

Second, heuristic upper bounds can be computed. After the branch-and-bound tree of the MIP solver has made the initial key decisions, it is possible to utilize a custom repair heuristic that can compute a solution much more quickly than traversing the remaining branch-and-bound tree. Assuming that the $(\sum \mathbb{A}/\sum \mathbb{D})$ value is specified, the heuristic can use the practical domain knowledge via a greedy space-filling approach to get results that would otherwise require substantial traversal of the branch-and-bound tree. This gives the upper bound and also presents a viable solution.

Third, key decisions can be made from prior knowledge. For example, the presence of relatively large elements can often preclude several other potential solutions. Some starting positions specified for key elements can serve as a good option to reduce computational effort while meeting other objectives. However, specification of location need not be a heuristic step that eliminates any good solutions. We can proceed as follows: We modify the branch-and-bound progression of the MIP solver to start with two custom branches. The first branch forces the element e to lie in the top half of the available space. The second branch constrains e not to lie in the top half. The next stage of branching specifies that the element e must lie on the left-hand or right-hand half of the canvas. Collectively, the branches guarantee that all possible options are logically covered.

Fourth and finally, for sufficiently large-sized instances (especially where elements are of relatively small size), it is possible to employ a column generation approach over the $\mathbb{A}, \mathbb{D}$ variables.

G. Empirical Evaluation

The outcomes of computational design are ultimately assessed not by evaluative functions or in terms of numerical performance but in the real benefits afforded to people. To claim "computational design" is to make a falsifiable statement about value produced for end users and other stakeholders. Hence, it is fair to insist on evidence as to whether the designs generated are actually usable, that the optimization can support decision-making and creative processes, or of how the outcomes compare in quality with those of human-designed interfaces.

From the application-oriented articles surveyed, we identified a subset on using combinatorial optimization for generation of GUIs and that presented empirical evaluation carried out with either end users or designers as participants. We excluded writings that failed to report the empirical method or were flagged by the authors as tentative, pilot, or preliminary. The articles can be divided into three main classes on the basis of their purpose and the type of empirical methodology followed.

Design studies are used to assess benefits of optimization as a tool in the design process. Professional or student designers are asked to generate interface designs in response to a given design brief. To generalize to design practice, design studies should be aimed at using representative materials, participants, and conditions. For GUI design, professional background should be related to interaction design, usability engineering, user interface or user experience design, or related areas of software engineering. Interactive grid layout sketching has been assessed in two studies. Using Sketchplorer [60], a black-box multiobjective approach to GUI wireframe generation, professional designers ($N = 10$) were asked to generate layouts with the system. Most designers included a suggestion by the optimizer in their final design, and they deemed the outcomes as good as their own. In a follow-up by Ramesh *et al.* [30], 16 professional interface designers had a similar task, but the optimizer was based on MIP, which was faster and could handle larger instances than the earlier black-box optimizer. Overall, feedback was positive, especially about support for layout exploration, particularly in the early stages of design. Again, most designers included optimized layout suggestions in the final designs. In a design study of functionality selection in product design ($N = 11$), an IP-based tool was compared to a process of nonsupported design [39]. Most designers ranked an optimizer-generated design in their top three designs by quality. From this article and a workshop study at a real-world company, the authors concluded that the tool's most significant benefit is to aid in exploring alternatives to an existing design idea. In a study with MenuOptimizer, novice designers (12 computer science students) [58] were asked to design menu systems with and without interactive optimization support in an SDE. Although the eventual designs were equal in quality levels, design took significantly less effort (with fewer editing steps) when the optimizer was used. In a design study by Park *et al.* [76], participants with a computer science background ($N = 6$) were asked to design two distributed user interfaces, using either an ILP-based optimizer (AdAM) or pen and paper, and rated the results afterward. The two approaches were not significantly different. Although the users commended the optimizer as powerful, using sliders to control the objectives was deemed tricky and took time.

Usability tests use a representative sample of end users who are asked to complete tasks with computer-generated user interfaces. In a study of ILP-generated hierarchical menus, university students ($N = 24$) were asked to find commands and click them. Computer-generated menus were compared to commercially available counterparts. The ILP-generated designs were found significantly faster to use. In a study of IP-assisted adaptation of web GUIs, Todi *et al.* [80] asked users ($N = 16$) to interact with web pages and assessed how quickly they could find items on a previously unseen page. The page layout was either as designed originally or rearranged via a black-box approach wherein a visual-search model fit to the user's history was used as the objective function. The optimizer's goal was to find a layout with elements placed such that they are closer to where the user may expect them to be. The approach demonstrated a significant improvement in the number of fixations needed for finding elements. In a task-based evaluation ($N = 11$) of GUI widget layouts generated by SUPPLE [83], motor-impaired and healthy users were asked to perform simple tasks. In a predesign step, parameters of a model were fit to individuals' motor characteristics. The ability-based interfaces proved significantly less error-prone and faster to use than the nonindividualized ones, with a 28% improvement in task-completion time relative to baselines. In a task-based evaluation of application menus generated by Sketchplorer [60], 20 users were asked to perform icon selections. The computer-generated menus were significantly faster than baseline commercial designs after some experience was gained. The optimized designs were also just as esthetically pleasing, with better color harmony and less symmetry, though this was not an objective for the system. In task-based assessment of user interfaces for printing services, users ($N = 48$) were asked to carry out printing tasks with commercial designs and with ones generated via a rule-based approach [62]. Subjects were twice as fast with computationally generated designs, particularly when the approach accounted for expectations of element locations due to prior experience. In a field experiment on information displays [77], IP-generated information layouts, created with IFT for the objective function, were compared to regular slide-based rotating displays. The attention of passersby was measured

by a computer vision sensor. Computer-generated layouts attracted twice as much attention as the baseline.

Quality ratings involve questionnaire-based studies that can be run for subjectively assessing the quality of the designs produced. In a rating-based study with software developers ($N = 20$), portrait/landscape transformations of GUIs [85] were performed by an optimizer using design heuristics for objectives. No statistical testing was utilized. Although the optimized results received a lower rating than human-generated ones, the respondents perceived the results to be of high quality and a good starting point for design. In a crowdsourced study with DesignScape [64], 40 novices were asked to create graphic designs. Other novices found the designs created with the help of the system better than those created without. In another ratings-based study, computer science students ($N = 15$) evaluated button layouts generated via a constraint-based approach [96]. This article was focused on identifying which layout objective is most useful for which level of complexity in layout managers. Finally, icon sets' comprehensibility was assessed in a crowdsourcing-based study by Laursen *et al.* [140]. The results showed a high correlation between the optimizer's comprehensibility score and human selection of icons for a given prompt but no correlation for the identifiability of icons.

VIII. CONCLUSION

This article has surveyed combinatorial optimization as a computational method for the generation and refinement of GUIs. It enables attacking more realistic design problems with increasing efficacy. Published results cover, among other problems, command selection, functionality selection, widget type selection, icon selection, menu design, layout design, and distribution of functionality. The main takeaway is that combinatorial optimization is emerging as a powerful and flexible complement to existing, mostly noncomputational design methods. It can be used for much more than to find a single optimal design. By virtue of abstraction, multiple instances of the same design problem can be solved. A menu optimizer, for example, can be run for any task instance.

The article has pointed out potentials that can be categorized under eight classes: 1) improved quality and robustness of designs; 2) solutions to very hard problems that are beyond human problem-solving capacity; 3) estimates of the practically achievable improvements possible for a given seed design; 4) information about the complexity and structure of a problem; 5) systems that adapt a design in real time; 6) integration of optimization with interactive design tools, to support both exploration and problem-solving; 7) personalization and customization of interfaces in line with individuals' preferences; and 8) an explicit, scrutinizable approach that can improve understanding of HCI and support knowledge-building in design teams.

Convergence of three advances has made this possible and constitutes a platform for future research. First, predictive models in algorithmic design were previously limited to simple models such as Fitts' law or, worse, hand-coded heuristics. Recent work demonstrates exploitation of a much wider range of user-related objectives, informed by research in psychology and HCI. In areas from performance to errors, ergonomics, and learnability, there is increasing understanding of how to represent the results of behavioral sciences in objective functions. This is a significant step forward in model-based approaches. One benefit of the design task definitions synthesized in this article is that any "oracle" that can produce feasible designs can be used. That could include scientific models but also, in the future, crowd-computing- or data-driven models. This allows separating assumptions about evaluative knowledge from the decision task.

Second, there is increased understanding of how to make combinatorial optimization not only compatible with but more broadly useful in design practice. Several concepts have been developed for integration and interaction with optimizers. These tackle obstacles that previously crippled the approach. Thanks to techniques that allow interactively specifying the task and steering an optimizer, professional designers can more readily delegate suitable hard tasks to a computer. An exciting aspect of this development is that it will open the possibility of novice users designing high-quality GUIs. Optimizers can guide a novice designer toward a good design [28], [30], [60]. We believe, further, that the use of empirically and theoretically sound evaluative functions is critical. Although numerous models have been presented in the literature (especially for vision, motor performance, and cognition), these have not been adopted in design practice. Only a few of them have ended up being taught in design schools and featuring in contemporary design standards and guidelines. If prevailing practice relies on quantitative models at all, it does so less than on empathy, tacit knowledge, heuristics, and empirical testing [167]. By their integration into design tools supported by optimization methods, the models can be made readily useful in design practice. Optimization can be used to inform such determinations in design as estimating the distance of a design from the global optimum. We believe the broader potentials are still unexploited. By defining the interests of several user groups in a multiobjective function, combinatorial optimization can aid in identifying the best compromises in a design space [39]. Using methods of robust optimization, one can find the design that is the best compromise even if assumptions about use change.

Third, combinatorial optimization offers a much-needed, coherent formalism for understanding what design is. It exposes the shape of GUI design problems from a rigorous but actionable mathematical perspective. While our understanding previously was limited to assignment problems, we can now see a spectrum of them—from packing to assignment problems and, further, to layout tasks. Cataloging them will help us understand the properties of these problems and identify efficient solution methods from the literature. We have taken the first steps

here by defining various common design tasks as integer programming tasks.

Combinatorial optimization will supply insights into the complexity and nature of GUI design. From a theoretical point of view, we know that integer programming is NP-hard. There exists a concise formulation—that is, one of polynomial size—for any problem that is in NP [168]. This implies that IP offers great modeling power for real-world problems, including GUI design. Therefore, it is good practice to proceed from an initial model of the problem as an IP, which enables the first experiments and further theoretical analyses. Since not every problem in NP is intractable, it might turn out that the given problem is contained in P. Hence, one should be able to refine the model into a linear program (i.e., avoid integrality constraints), because linear programming is P-complete. An example was given with regard to functionality selection with price-collecting dependencies. Even if this is not the case for the general problem under consideration, there might be confinements to special cases or relaxations that are tractable. Identifying such restrictions and relaxations can be useful for designing approximation algorithms where we embed an instance of the general problem in the restricted space. One can look for a proxy instance that is tractable, find a solution, and transform it for the original problem, or one can try to adjust an optimum solution obtained for the relaxation such that it is feasible for the original problem.

Uncertainty poses an unresolved theoretical problem for further applications in this area. The roots of the issue lie in the characteristically ill-defined nature of design, the degree of variability we see in human behavior, and the lack of precise predictive models (see chemical or civil engineering). Consequently, objective functions should be designed with uncertainty in mind. The problem of optimization in conditions of uncertainty plays a prominent role in optimization theory. It is commonly assumed that the uncertainties can be classified via probability distributions, which is not true in most of the challenges described in this article. Sahinidis [149] has provided a detailed survey of optimization under uncertainty. Related research is emerging also in robust optimization, a field whose problems have uncertain components that are contained in a prespecified uncertainty set. Instead of using a probability distribution, we can assume that a decision-maker may arbitrarily choose values from within this uncertainty set (see [148] for a brief introduction to robust optimization).

However, the scenario of integer programming in GUI design is slightly different. In many cases, it is not possible to define a probability distribution or even an uncertainty set for our models. The reason is that subjective opinions about valuations of various designs can change during the optimization process. For example, after generation of the first solutions that are feasible according to the current model, the user might detect unwanted behavior and change weights in the objective function or even restrict the model by adding further constraints. In practice, this dynamic change in optimization models also restricts us from using off-the-shelf solvers. These solvers cannot benefit from earlier optimization processes if the model changes too much, which leads to solving the problem from scratch after every user input. In conclusion, though GUI design is closely related to the field of optimization amid uncertainty and of robust optimization, the special nature of this subjective and often unstructured uncertainty justifies considering interactive optimization a distinct field of research. Interactive optimization encompasses all optimization wherein users interact with the model or the solution methods during the optimization process, a vast field that poses new challenges to state-of-the-art optimization methods. A survey of the categorization of this broad area and its recent advances is provided by Meignan *et al.* [162].

We note also that there is a dire need to develop empirical methodology specifically for evaluating computational design systems. The empirical evidence reviewed in this article suggests that at least in the case of GUI layouts and element selection tasks, optimizer-generated designs can be of high quality and even comparable to expert-designed interfaces. Designers report benefiting from the approach especially earlier in the design process, when exploration of options can "nudge" them and help them avoid fixating on any one idea too soon. However, empirical studies are still fairly rare, relative to the volume of articles presenting technical solutions. Thus, although published results have been positive, much work remains to be done to collect data in more realistic settings and, to that end, determine the most valid methods. One recognized issue is how to select representative tasks: if the tasks are too complex, studies are expensive to run and can be computationally hard [76]. If they are too easy, optimization may bring little practical value. Also, materials and tasks can be "cherry-picked" to produce confirmatory results. Indeed, we could find only one paper on work wherein computer-generated designs were compared outside the laboratory [77]. Another challenge that remains is to assist designers, who may not have used optimization systems before, to control multiobjective optimization [76]. For example, Bailly *et al.* [58] reported a menu optimizer being hard to understand and the models difficult to control. Others' articles acknowledge an expensive ramp-up stage for defining an optimization task for the optimizer [39], [62]. A third issue is the need for critically testing the objective functions empirically. One of the most significant benefits of the approach is that it yields empirically verifiable predictions. Empirical data's collection serves assessment not only of the outcome itself but also of the objective function that was used. We know of only three studies thus far that have compared empirical data against objective functions [30], [60], [96].

In conclusion, above and beyond these observations, perhaps the most exciting academic prospect of combinatorial optimization for the field of HCI lies in offering a truly interdisciplinary yet principled solution to one of its most profound questions: how data and theory can

inform design [31], [169]. Combinatorial optimization invites computer scientists to figure out what "design" means and how "data" can become models, and it also calls for psychologists, cognitive scientists, and designers to define models for what "good" means for humans in HCI. However, first and foremost, optimization is a call for deriving solutions from first principles, similar to engineering design [170]. This encourages proper formulation of problems and analysis of conditions in which good outcomes can be reached. Optimization could promote a change of culture in design by facilitating the explication, scrutinizing, and accumulation of design knowledge. We expect that the future will see interaction design—which has been considered to be, through and through, empathic, nuanced, tacit, and dynamic activity—that abstracts, decomposes, and algorithmically solves a widening array of problems, all, moreover, in a manner acceptable to designers and end users. ■

REFERENCES

[1] W. R. King and J. He, "A meta-analysis of the technology acceptance model," *Inf. Manage.*, vol. 43, no. 6, pp. 740–755, Sep. 2006.

[2] A. Oulasvirta, X. Bi, and A. Howes, *Computational Interaction*. Oxford, U.K.: Oxford Univ. Press, 2018.

[3] N. Cross, *Designerly Ways of Knowing*. London, U.K.: Springer, 2006.

[4] J. Löwgren and E. Stolterman, *Thoughtful Interaction Design: A Design Perspective on Information Technology*. Cambridge, MA, USA: MIT Press, 2004.

[5] A. Chevalier and M. Y. Ivory, "Web site designs: Influences of designer's expertise and design constraints," *Int. J. Human-Comput. Studies*, vol. 58, no. 1, pp. 57–87, Jan. 2003.

[6] A. Dix, *Human-Computer Interaction*. London, U.K.: Springer, 2009.

[7] L. Hallnäs and J. Redström, "Interaction design: Foundations, experiments," Textile Res. Centre, Swedish School Textiles, Univ. Borås, Interact. Inst., Borås, Sweden, 2006.

[8] J. Preece, H. Sharp, and Y. Rogers, *Interaction Design: Beyond Human-Computer Interaction*. Hoboken, NJ, USA: Wiley, 2015.

[9] D. Saffer, *Designing for Interaction: Creating Innovative Applications and Devices*. London, U.K.: New Riders, 2010.

[10] T. Winograd, "The design of interaction," in *Beyond Calculation*. London, U.K.: Springer, 1997, pp. 149–161.

[11] K. Dorst and N. Cross, "Creativity in the design process: Co-evolution of problem–solution," *Design Studies*, vol. 22, no. 5, pp. 425–437, Sep. 2001.

[12] B. Buxton, *Sketching User Experiences: Getting the Design Right and the Right Design*. San Mateo, CA, USA: Morgan Kaufmann, 2010.

[13] S. Feuerstack, M. Blumendorf, V. Schwartze, and S. Albayrak, "Model-based layout generation," in *Proc. Work. Conf. Adv. Vis. Interfaces*, 2008, pp. 217–224.

[14] W. O. Galitz, *The Essential Guide to user Interface Design: An Introduction to GUI Design Principles and Techniques*. Hoboken, NJ, USA: Wiley, 2007.

[15] J. D. Foley, V. L. Wallace, and P. Chan, "The human factors of computer graphics interaction techniques," *IEEE Comput. Grap. Appl.*, vol. 4, no. 11, pp. 13–48, Nov. 1984.

[16] G. Bailly, E. Lecolinet, and L. Nigay, "Visual menu techniques," *ACM Comput. Surv.*, vol. 49, no. 4, p. 60, 2017.

[17] D. Bowman, E. Kruijff, J. J. LaViola, Jr., and I. P. Poupyrev, *3D User Interfaces: Theory and Practice, CourseSmart eTextbook*. Reading, MA, USA: Addison-Wesley, 2004.

[18] J. Wang, L. Quan, J. Sun, X. Tang, and H.-Y. Shum, "Picture collage," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 1, Jun. 2006, pp. 347–354.

[19] N. Hurst, W. Li, and K. Marriott, "Review of automatic document formatting," in *Proc. 9th ACM Symp. Document Eng.*, 2009, pp. 99–108.

[20] L. A. Meyerovich and R. Bodik, "Fast and parallel Webpage layout," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 711–720.

[21] Ž. Mijailović and D. Milićev, "Empirical analysis of GUI programming concerns," *Int. J. Human-Comput. Studies*, vol. 72, nos. 10–11, pp. 757–771, Oct. 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1071581914000664

[22] J. Vanderdonckt, "Model-driven engineering of user interfaces: Promises, successes, failures, and challenges," in *Proc. ROCHI*, vol. 8, 2008, p. 32.

[23] G. Meixner, F. Paternò, and J. Vanderdonckt, "Past, present, and future of model-based user interface development," *I-Com Zeitschrift Interaktive Kooperative Medien*, vol. 10, no. 3, pp. 2–11, 2011.

[24] R. Popp, J. Falb, D. Raneburger, and H. Kaindl, "A transformation engine for model-driven UI generation," in *Proc. 4th ACM SIGCHI Symp. Eng. Interact. Comput. Syst. (EICS)*, New York, NY, USA, 2012, pp. 281–286, doi: 10.1145/2305484.2305532.

[25] M. Heymann and A. Degani, "Formal analysis and automatic generation of user interfaces: Approach, methodology, and an algorithm," *Human Factors*, vol. 49, no. 2, pp. 311–330, Apr. 2007.

[26] S. E. Hudson and C.-N. Hsi, "A synergistic approach to specifying simple number independent layouts by example," in *Proc. Conf. Human Factors Comput. Syst. (INTERACT CHI)*, 1993, pp. 285–292.

[27] W. C. Kim and J. D. Foley, "Don: User interface presentation design assistant," in *Proc. 3rd Annu. ACM SIGGRAPH Symp. User Interface Softw. Technol. (UIST)*, New York, NY, USA, 1990, pp. 10–20, doi: 10.1145/97924.97926.

[28] B. Myers, S. E. Hudson, and R. Pausch, "Past, present, and future of user interface software tools," *ACM Trans. Comput.-Human Interact.*, vol. 7, no. 1, pp. 3–28, Mar. 2000.

[29] R. J. Beach, "Setting tables and illustrations with style," Univ. Waterloo, Waterloo, ON, Canada, Tech. Rep. AAI0556392, 1985.

[30] N. D. Ramesh, K. Todi, A. Oulasvirta, and T. Saarelainen, "Interactive grid layout design with integer programming," in *Proc. CHI Conf. Human Factors Comput. Syst.*, 2020.

[31] S. K. Card, A. Newell, and T. P. Moran, *The Psychology of Human-Computer Interaction*. Boca Raton, FL, USA: CRC Press, 1983.

[32] D. L. Fisher, "Optimal performance engineering: Good, better, best," *Human Factors, J. Human Factors Ergonom. Soc.*, vol. 35, no. 1, pp. 115–139, 1993.

[33] M. Y. Ivory and M. A. Hearst, "The state of the art in automating usability evaluation of user interfaces," *ACM Comput. Surv.*, vol. 33, no. 4, pp. 470–516, 2001.

[34] A. Shocker and V. Srinivasan, "A consumer-based methodology for the identification of new product ideas," *Manage. Sci.*, vol. 20, no. 6, pp. 921–937, 1974. [Online]. Available: http://www.jstor.org/stable/2630205

[35] M. D. Albritton and P. R. McMullen, "Optimal product design using a colony of virtual ants," *Eur. J. Oper. Res.*, vol. 176, no. 1, pp. 498–520, 2007.

[36] V. Krishnan and K. T. Ulrich, "Product development decisions: A review of the literature," *Manage. Sci.*, vol. 47, no. 1, pp. 1–21, 2001.

[37] J. R. Jiao, T. W. Simpson, and Z. Siddique, "Product family design and platform-based product development: A state-of-the-art review," *J. Intell. Manuf.*, vol. 18, no. 1, pp. 5–29, Jul. 2007.

[38] A. G. Kök and M. L. Fisher, "Demand estimation and assortment optimization under substitution: Methodology and application," *Oper. Res.*, vol. 55, no. 6, pp. 1001–1021, Dec. 2007.

[39] A. Oulasvirta, A. Feit, P. Lähteenlahti, and A. Karrenbauer, "Computational support for functionality selection in interaction design," *ACM Trans. Comput.-Human Interact.*, vol. 24, no. 5, pp. 1–30, Oct. 2017.

[40] N. Kano, N. Seraku, F. Takahashi, and S. Tsuji, "Attractive quality and must-be quality," *J. Jpn. Soc. Quality Control*, vol. 14, no. 2, pp. 39–48, 1984.

[41] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for gui testing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 256–267, 2001.

[42] S. K. Feiner, "A grid-based approach to automating display layout," in *Proc. Graph. Interface*, Toronto, ON, Canada, 1988, pp. 192–197. [Online]. Available: http://dl.acm.org/citation.cfm?id=102313.102339

[43] M. Pollatschek, M. Gershoni, and Y. Tadday, "Improving the hebrew typewriter," Technion, Haifa, Israel, Tech. Rep., 1975.

[44] R. E. Burkard and J. Offermann, "Entwurf von Schreibmaschinentastaturen mittels quadratischer Zuordnungsprobleme," *Z. Oper. Res.*, vol. 21, no. 4, pp. B121–B132, Aug. 1977.

[45] A. M. Feit *et al.*, "Assignment problems for optimizing text input," Aalto Univ., Espoo, Finland, Tech. Rep. 103/2018, 2018.

[46] S. M. Hart and L. Yi-Hsin, "The application of integer linear programming to the implementation of a graphical user interface: A new rectangular packing problem," *Appl. Math. Model.*, vol. 19, no. 4, pp. 244–254, Apr. 1995.

[47] I. E. Sutherland, "Sketchpad a man-machine graphical communication system," *Simulation*, vol. 2, no. 5, pp. R-3–R-20, May 1964.

[48] S. Lok and S. Feiner, "A survey of automated layout techniques for information presentations," in *Proc. SmartGraph.*, 2001, pp. 61–68.

[49] B. N. Freeman-Benson, J. Maloney, and A. Borning, "An incremental constraint solver," *Commun. ACM*, vol. 33, no. 1, pp. 54–63, Jan. 1990, doi: 10.1145/76372.77531.

[50] H. Hosobe, "A modular geometric constraint solver for user interface applications," in *Proc. 14th Annu. ACM Symp. Interface Softw. Technol. (UIST)*, New York, NY, USA, 2001, pp. 91–100, doi: 10.1145/502348.502362.

[51] P. Hertzog, "Binary space partitioning layouts to help build better information dashboards," in *Proc. 20th Int. Conf. Intell. Interfaces (IUI)*, New York, NY, USA, 2015, pp. 138–147, doi: 10.1145/2678025.2701383.

[52] A. R. Puerta, H. Eriksson, J. H. Gennari, and M. A. Musen, "Beyond data models for automated user interface generation," in *Proc. BCS HCI*, 1994, pp. 353–366.

[53] P. A. Akiki, A. K. Bandara, and Y. Yu, "Adaptive model-driven user interface development systems," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 1–33, May 2014.

[54] J. Coutaz, "User interface plasticity: Model driven engineering to the limit!" in *Proc. 2nd ACM SIGCHI Symp. Eng. Interact. Comput. Syst.*, 2010, pp. 1–8.

[55] M. Fisher, "Interactive optimization," *Ann. Oper. Res.*, vol. 5, no. 3, pp. 539–556, 1985.

[56] S. Zhai, M. Hunter, and B. A. Smith, "Performance optimization of virtual keyboards," *Human-Comput. Interact.*, vol. 17, nos. 2–3, pp. 229–269, 2002.

[57] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock, "Automatically generating personalized user interfaces with Supple," *Artif. Intell.*, vol. 174, nos. 12–13, pp. 910–950, Aug. 2010.

[58] G. Bailly, A. Oulasvirta, T. Kötzing, and S. Hoppe, "Menuoptimizer: Interactive optimization of menu systems," in *Proc. 26th Annu. ACM Symp. Interface Softw. Technol.*, 2013, pp. 331–342.

[59] A. Sears, "Aide: A step toward metric-based interface development tools," in *Proc. 8th Annu. ACM Symp. Interface Softw. Technol.*, 1995, pp. 101–110.

[60] K. Todi, D. Weir, and A. Oulasvirta, "Sketchplore: Sketch and explore with a layout optimiser," in *Proc. ACM Conf. Designing Interact. Syst.*, 2016, pp. 543–555.

[61] S. S. Rao, *Engineering Optimization: Theory and Practice*. Hoboken, NJ, USA: Wiley, 2009.

[62] J. Nichols, D. H. Chau, and B. A. Myers, "Demonstrating the viability of automatically generated user interfaces," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2007, pp. 1283–1292.

[63] P. O'donovan, A. Agarwala, and A. Hertzmann, "Learning layouts for single-pagegraphic designs," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 8, pp. 1200–1213, Aug. 2014.

[64] P. O'Donovan, A. Agarwala, and A. Hertzmann, "DesignScape: Design with interactive layout suggestions," in *Proc. 33rd Annu. ACM Conf. Human Factors Comput. Syst.*, 2015, pp. 1221–1224.

[65] E. Schrier, M. Dontcheva, C. Jacobs, G. Wade, and D. Salesin, "Adaptive layout for dynamically aggregated documents," in *Proc. 13th Int. Conf. Intell. Interfaces*, 2008, pp. 99–108.

[66] K. Gajos *et al.*, "Fast and robust interface generation for ubiquitous applications," in *Proc. Int. Conf. Ubiquitous Comput.* Springer, 2005, pp. 37–55.

[67] F. Paterno and C. Santoro, "One model, many interfaces," in *Computer-Aided Design of User Interfaces III*. London, U.K.: Springer, 2002, pp. 143–154.

[68] B. Price, R. Greiner, G. Häubl, and A. Flatt, "Automatic construction of personalized customer interfaces," in *Proc. 11th Int. Conf. Intell. Interfaces*, 2006, pp. 250–257.

[69] M. Nebeling, "Xdbrowser 2.0: Semi-automatic generation of cross-device interfaces," in *Proc. Conf. Human Factors Comput. Syst. (CHI)*, 2017, pp. 4574–4584.

[70] C. M. Beshers and S. Feiner, "Scope: Automated generation of graphical interfaces," in *Proc. 2nd Annu. ACM SIGGRAPH Symp. Interface Softw. Technol. (UIST)*, New York, NY, USA, 1989, pp. 76–85, doi: 10.1145/73660.73670.

[71] C. Janssen, A. Weisbecker, and J. Ziegler, "Generating user interfaces from data models and dialogue net specifications," in *Proc. Conf. Human Factors Comput. Syst. (INTERACT CHI)*, New York, NY, USA, 1993, pp. 418–423, doi: 10.1145/169059.169335.

[72] A. Pizano, Y. Shirota, and A. Iizawa, "Automatic generation of graphical user interfaces for interactive database applications," in *Proc. 2nd Int. Conf. Inf. Knowl. Manage. (CIKM)*, New York, NY, USA, 1993, pp. 344–355, doi: 10.1145/170088.170166.

[73] R. A. M. Murillo, S. Subramanian, and D. M. Plasencia, "Erg-O: Ergonomic optimization of immersive virtual environments," in *Proc. 30th Annu. ACM Symp. Interface Softw. Technol.*, 2017, pp. 759–771.

[74] R. Kumar, J. O. Talton, S. Ahmad, and S. R. Klemmer, "Bricolage: Example-based retargeting for Web design," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2011, pp. 2197–2206.

[75] J. Fogarty and S. E. Hudson, "Gadget: A toolkit for optimization-based approaches to interface and display generation," in *Proc. 16th Annu. ACM Symp. Interface Softw. Technol.*, 2003, pp. 125–134.

[76] S. Park *et al.*, "Adam: Adapting multi-user interfaces for collaborative environments in real-time," in *Proc. Conf. Human Factors Comput. Syst. (CHI)*, 2018, p. 184.

[77] M. L. Montoya Freire, D. Potts, N. R. Dayama, A. Oulasvirta, and M. Di Francesco, "Foraging-based optimization of pervasive displays," *Pervas. Mobile Comput.*, vol. 55, pp. 45–58, Apr. 2019.

[78] J. Nichols, B. Rothrock, D. H. Chau, and B. A. Myers, "Huddle: Automatically generating interfaces for systems of multiple connected appliances," in *Proc. 19th Annu. ACM Symp. Interface Softw. Technol. (UIST)*, New York, NY, USA, 2006, pp. 279–288, doi: 10.1145/1166253.1166298.

[79] K. Todi, J. Jokinen, K. Luyten, and A. Oulasvirta, "Familiarisation: Restructuring layouts with visual learning models," in *Proc. 23rd Int. Conf. Intell. Interfaces*, 2018, pp. 547–558.

[80] K. Todi, J. Jokinen, K. Luyten, and A. Oulasvirta, "Individualising graphical layouts with predictive visual search models," *ACM Trans. Interact. Intell. Syst.*, vol. 10, no. 1, pp. 1–24, Aug. 2019.

[81] N. Ramesh, M. Shiripour, A. Oulasvirta, E. Ivanko, and A. Karrenbauer, "Foraging-based optimization of menu systems," submitted for publication.

[82] L. Micallef, G. Palmas, A. Oulasvirta, and T. Weinkauf, "Towards perceptual optimization of the visual design of scatterplots," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 6, pp. 1588–1599, Jun. 2017.

[83] K. Z. Gajos, J. O. Wobbrock, and D. S. Weld, "Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2008, pp. 1257–1266.

[84] A. Oulasvirta, "User interface design with combinatorial optimization," *Computer*, vol. 50, no. 1, pp. 40–47, Jan. 2017.

[85] C. Zeidler, G. Weber, W. Stuerzlinger, and C. Lutteroth, "Automatic generation of user interface layouts for alternative screen orientations," in *Proc. IFIP Conf. Human-Comput. Interact.* London, U.K.: Springer, 2017, pp. 13–35.

[86] D. Lindlbauer, A. M. Feit, and O. Hilliges, "Context-aware online adaptation of mixed reality interfaces," in *Proc. 32nd Annu. ACM Symp. Interface Softw. Technol. (UIST)*, New York, NY, USA, 2019, pp. 147–160.

[87] J. Abascal, A. Aizpurua, I. Cearreta, B. Gamecho, N. Garay-Vitoria, and R. Miñón, "Automatically generating tailored accessible user interfaces for ubiquitous services," in *Proc. 13th Int. ACM SIGACCESS Conf. Comput. Accessibility*, 2011, pp. 187–194.

[88] S. Sarcar, J. P. Jokinen, A. Oulasvirta, Z. Wang, C. Silpasuwanchai, and X. Ren, "Ability-based optimization of touchscreen interactions," *IEEE Perv. Comput.*, vol. 17, no. 1, pp. 15–26, Jan. 2018.

[89] S. Amaran, N. V. Sahinidis, B. Sharda, and S. J. Bury, "Simulation optimization: A review of algorithms and applications," *Ann. Oper. Res.*, vol. 240, no. 1, pp. 351–380, May 2016, doi: 10.1007/s10479-015-2019-x.

[90] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[91] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming* (Foundations of Artificial Intelligence), vol. 2. Amsterdam, The Netherlands: Elsevier, 2006. [Online]. Available: http://www.sciencedirect.com/science/bookseries/15746526/2

[92] T. Achterberg, T. Berthold, T. Koch, and K. Wolter, "Constraint integer programming: A new approach to integrate CP and MIP," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, L. Perron and M. A. Trick, Eds. Berlin, Germany: Springer, 2008, pp. 6–20.

[93] A. Schrijver, *Theory of Linear and Integer Programming*. Hoboken, NJ, USA: Wiley, 1998.

[94] D. Bertsimas and J. Tsitsiklis, *Introduction to Linear Optimization*, 1st ed. Belmont, MA, USA: Athena Scientific, 1997.

[95] G. Rinaldi and L. A. Wolsey, "Integer programming and combinatorial optimization," in *Proc. IPCO*, vol. 3, 1993.

[96] C. Zeidler, C. Lutteroth, and G. Weber, "Constraint solving for beautiful user interfaces: How solving strategies support layout aesthetics," in *Proc. 13th Int. Conf. NZ Chapter ACM's Special Interest Group Human-Comput. Interact.*, 2012, pp. 72–79.

[97] A. Borning, R. Lin, and K. Marriott, "Constraints for the Web," in *Proc. 5th ACM Int. Conf. Multimedia*, 1997, pp. 173–182.

[98] A. Oulasvirta and A. Karrenbauer, "Combinatorial optimization for user interface design," in *Computational Interaction*. Oxford, U.K.: Oxford Univ. Press, 2018.

[99] A. Caprara, P. Toth, and M. Fischetti, "Algorithms for the set covering problem," *Ann. Oper. Res.*, vol. 98, no. 1, pp. 353–371, Dec. 2000, doi: 10.1023/A:1019225027893.

[100] D. Moshkovitz, "The projection games conjecture and the NP-hardness of LN N-approximating set-cover," *Theory Comput.*, vol. 11, pp. 221–235, Aug. 2015.

[101] N. Alon, U. Feige, A. Wigderson, and D. Zuckerman, "Derandomized graph products," *Comput. Complex.*, vol. 5, no. 1, pp. 60–75, Mar. 1995, doi: 10.1007/bf01277956.

[102] T. Schiavinotto and T. Stützle, "The linear ordering problem: Instances, search space analysis and algorithms," *J. Math. Model. Algorithms*, vol. 3, no. 4, pp. 367–402, 2004, doi: 10.1023/b:jmma.0000049426.06305.d8.

[103] S. Arora, "Polynomial time approximation schemes for Euclidean TSP and other geometric problems," in *Proc. 37th Conf. Found. Comput. Sci.*, Oct. 1996, pp. 2–11.

[104] L. Engebretsen and M. Karpinski, "Approximation hardness of TSP with bounded metrics," in *Automata, Languages and Programming*, F. Orejas, P. G. Spirakis, and J. van Leeuwen, Eds. Berlin, Germany: Springer, 2001, pp. 201–212.

[105] J. Munkres, "Algorithms for the assignment and transportation problems," *J. Soc. Ind. Appl. Math.*, vol. 5, no. 1, pp. 32–38, 1957.

[106] R. Duan and H.-H. Su, "A scaling algorithm for maximum weight matching in bipartite graphs," in *Proc. 23rd Annu. ACM-SIAM Symp. Discrete Algorithms*. Philadelphia, PA, USA: SIAM, 2012, pp. 1413–1424.

[107] T. Koopmans and M. J. Beckmann, "Assignment problems and the location of economic activities," Cowles Found. Res. Econ., Yale Univ., New Haven, CT, USA, Cowles Found. Discuss. Papers 4, 1955.

[108] M. Queyranne, "Performance ratio of polynomial heuristics for triangle inequality quadratic assignment problems," *Oper. Res. Lett.*, vol. 4, no. 5, pp. 231–234, Feb. 1986.

[109] P. C. Gilmore, "Optimal and suboptimal algorithms for the quadratic assignment problem," *SIAM J. Appl. Math.*, vol. 10, pp. 305–313, Jun. 1962.

[110] E. L. Lawler, "The quadratic assignment problem," *Manage. Sci.*, vol. 9, no. 4, pp. 586–599, 1963.

[111] L. Kaufman and F. Broeckx, "An algorithm for the quadratic assignment problem using benders' decomposition," *Eur. J. Oper. Res.*, vol. 2, no. 3, pp. 204–211, 1978.

[112] F. Rendl, G. Rinaldi, and A. Wiegele, "Solving max-cut to optimality by intersecting semidefinite and polyhedral relaxations," *Math. Program.*, vol. 121, no. 2, pp. 307–335, Feb. 2010.

[113] J. Povh and F. Rendl, "Copositive and semidefinite relaxations of the quadratic assignment problem," *Discrete Optim.*, vol. 6, no. 3, pp. 231–241, Aug. 2009.

[114] Q. Zhao, S. E. Karisch, F. Rendl, and H. Wolkowicz, "Semidefinite programming relaxations for the quadratic assignment problem," *J. Combinat. Optim.*, vol. 2, no. 1, pp. 71–109, 1998.

[115] M. John and A. Karrenbauer, "A novel SDP relaxation for the quadratic assignment problem using cut pseudo bases," in *Proc. Int. Symp. Combinat. Optim.*, 2016, pp. 414–425.

[116] C. W. Commander, "A survey of the quadratic assignment problem, with applications," *Morehead Electron. J. Applicable Math.*, vol. 4, no. 4, pp. 1–15, 2005, Paper MATH-2005-01.

[117] E. M. Loiola, N. M. M. de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido, "A survey for the quadratic assignment problem," *Eur. J. Oper. Res.*, vol. 176, no. 2, pp. 657–690, 2007.

[118] A. Caprara and M. Monaci, "On the two-dimensional knapsack problem," *Oper. Res. Lett.*, vol. 32, no. 1, pp. 5–14, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167637703000579

[119] K. Jansen, "A fast approximation scheme for the multiple knapsack problem," in *Proc. Theory Pract. Comput. Sci. (SOFSEM)*, M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán, Eds. Berlin, Germany: Springer, 2012, pp. 313–324.

[120] S. Heydrich and A. Wiese, "Faster approximation schemes for the two-dimensional knapsack problem," in *Proc. 28th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2017, pp. 79–98. [Online]. Available: http://dl.acm.org/citation.cfm?id=3039686.3039692

[121] B. Lee, S. Kim, A. Oulasvirta, J.-I. Lee, and E. Park, "Moving target selection: A cue integration model," in *Proc. Conf. Human Factors Comput. Syst. (CHI)*, 2018, p. 230.

[122] S. Sridhar, A. M. Feit, C. Theobalt, and A. Oulasvirta, "Investigating the dexterity of multi-finger input for mid-air text entry," in *Proc. 33rd Annu. ACM Conf. Human Factors Comput. Syst.*, 2015, pp. 3643–3652.

[123] M. Bachynskyi, G. Palmas, A. Oulasvirta, and T. Weinkauf, "Informing the design of novel input methods with muscle coactivation clustering," *ACM Trans. Comput.-Human Interact.*, vol. 21, no. 6, pp. 1–25, Jan. 2015.

[124] J. Accot and S. Zhai, "Refining Fitts' law models for bivariate pointing," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2003, pp. 193–200.

[125] K. Gajos and D. S. Weld, "Preference elicitation for interface optimization," in *Proc. 18th Annu. ACM Symp. Interface Softw. Technol.*, 2005, pp. 173–182.

[126] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 1990, pp. 249–256.

[127] S. Lok, S. Feiner, and G. Ngai, "Evaluation of visual balance for automated layout," in *Proc. 9th Int. Conf. Intell. Interfaces (IUI)*, New York, NY, USA, 2004, pp. 101–108, doi: 10.1145/964442.964462.

[128] C. Ling and G. Salvendy, "Extension of heuristic evaluation method: A review and reappraisal," *Ergonomia IJE & HF*, vol. 27, no. 3, pp. 179–197, 2005.

[129] J. Tidwell, *Designing Interfaces: Patterns for Effective Interaction Design*. Newton, MA, USA: O'Reilly Media, 2010.

[130] R. Y. Gómez, D. C. Caballero, and J.-L. Sevillano, "Heuristic evaluation on mobile interfaces: A new checklist," *Sci. World J.*, vol. 2014, pp. 1–19, Sep. 2014.

[131] I. Khaddam, S. Bouzit, G. Calvary, and D. Chêne, "Menuergo: Computer-aided design of menus by automated guideline review," in *Proc. Actes 28th Conf. Francophone Interact. Homme-Mach.*, 2016, pp. 36–47.

[132] F. Paz, F. A. Paz, D. Villanueva, and J. A. Pow-Sang, "Heuristic evaluation as a complement to usability testing: A case study in Web domain," in *Proc. 12th Int. Conf. Inf. Technol.-New Generat. (ITNG)*, 2015, pp. 546–551.

[133] A. Cockburn and C. Gutwin, "A predictive model of human performance with scrolling and hierarchical lists," *Human-Comput. Interact.*, vol. 24, no. 3, pp. 273–314, Jul. 2009.

[134] D. Cohen-Or, O. Sorkine, R. Gal, T. Leyvand, and Y.-Q. Xu, "Color harmonization," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 624–630, 2006.

[135] Z. Bylinskii *et al.*, "Learning visual importance for graphic designs and data visualizations," in *Proc. 30th Annu. ACM Symp. Interface Softw. Technol.*, 2017, pp. 57–69.

[136] A. Oulasvirta *et al.*, "Aalto interface metrics (AIM): A service and codebase for computational GUI evaluation," in *Proc. 31st Annu. ACM Symp. Interface Softw. Technol. Adjunct*, 2018, pp. 16–19.

[137] D. E. Kieras and A. J. Hornof, "Towards accurate and practical predictive models of active-vision-based visual search," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2014, pp. 3875–3884.

[138] K. Leino, A. Oulasvirta, and M. Kurimo, "RL-KLM: Automating keystroke-level modeling with reinforcement learning," in *Proc. 24rd Int. Conf. Intell. Interfaces*, 2019, pp. 476–480.

[139] J. Orlin, "A faster strongly polynomial minimum cost flow algorithm," in *Proc. 20th Annu. ACM Symp. Theory Comput. (STOC)*, New York, NY, USA, 1988, pp. 377–387, doi: 10.1145/62212.62249.

[140] L. F. Laursen *et al.*, "Icon set selection via human computation," in *Proc. Pacific Graph. Short Papers*, Goslar, Germany, 2016, pp. 1–6.

[141] S. Matsui and S. Yamada, "Genetic algorithm can optimize hierarchical menus," in *Proc. SIGCHI Conf. Human Factors Comput. Syst. (CHI)*, New York, NY, USA, 2008, pp. 1385–1388, doi: 10.1145/1357054.1357271.

[142] S. Matsui and S. Yamada, "Optimizing hierarchical menus by genetic algorithm and simulated annealing," in *Proc. 10th Annu. Conf. Genetic Evol. Comput. (GECCO)*, New York, NY, USA, 2008, pp. 1587–1594, doi: 10.1145/1389095.1389397.

[143] P. Pirolli and S. Card, "Information foraging," *Psychol. Rev.*, vol. 106, no. 4, p. 643, 1999.

[144] R. Kennard and R. Steele, "Application of software mining to automatic user interface generation," in *Proc. Int. Conf. Softw. Methods Tools*. Amsterdam, The Netherlands: IOS Press, 2008.

[145] R. Kumar *et al.*, "Webzeitgeist: Design mining the Web," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2013, pp. 3083–3092.

[146] J. Talton, L. Yang, R. Kumar, M. Lim, N. Goodman, and R. Měch, "Learning design patterns with Bayesian grammar induction," in *Proc. 25th Annu. ACM Symp. User Interface Softw. Technol.*, New York, NY, USA, Oct. 2012, pp. 63–74, doi: 10.1145/2380116.2380127.

[147] H.-G. Beyer and B. Sendhoff, "Robust optimization—A comprehensive survey," *Comput. Methods Appl. Mech. Eng.*, vol. 196, nos. 33–34, pp. 3190–3218, 2007.

[148] B. L. Gorissen, İ. Yanikoğlu, and D. den Hertog, "A practical guide to robust optimization," *Omega*, vol. 53, pp. 124–137, Jun. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0305048314001698

[149] N. V. Sahinidis, "Optimization under uncertainty: State-of-the-art and opportunities," *Comput. Chem. Eng.*, vol. 28, no. 6, pp. 971–983, Jun. 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0098135403002369

[150] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.

[151] Y. Koyama and T. Igarashi, "Computational design with crowds," *Comput. Interact.*, vol. 9849, p. 153, Jan. 2018.

[152] A. Kangasrääsiö, K. Athukorala, A. Howes, J. Corander, S. Kaski, and A. Oulasvirta, "Inferring cognitive models from data using approximate Bayesian computation," in *Proc. Conf. Human Factors Comput. Syst. (CHI)*, 2017, pp. 1295–1306.

[153] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," *Struct. Multidisciplinary Optim.*, vol. 26, no. 6, pp. 369–395, 2004.

[154] F. Hillier and G. Lieberman, *Introduction to Operations Research* (Introduction to Operations Research), vol. 1. New York, NY, USA: McGraw-Hill, 2001. [Online]. Available: https://books.google.it/books?id=OEhUDQEACAAJ

[155] M. Kaisa, *Nonlinear Multiobjective Optimization* (International Series in Operations Research & Management Science), vol. 12. Boston, MA, USA: Kluwer, 1999.

[156] K.-L. Du *et al.*, *Search and Optimization By Metaheuristics*. Cambridge, MA, USA: Birkhaüser, 2016.

[157] E.-G. Talbi, *Metaheuristics: From Design to Implementation*, vol. 74. Hoboken, NJ, USA: Wiley, 2009.

[158] S. Chand and M. Wagner, "Evolutionary many-objective optimization: A quick-start guide," *Surv. Oper. Res. Manage. Sci.*, vol. 20, no. 2, pp. 35–42, Dec. 2015.

[159] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[160] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength Pareto evolutionary algorithm," Dept. Elect. Eng., Swiss Federal Inst. Technol., Zurich, Switzerland, Tech. Rep. 103, 2001.

[161] G. W. Klau, N. Lesh, J. Marks, M. Mitzenmacher, and G. T. Schafer, "The hugs platform: A toolkit for interactive optimization," in *Proc. Work. Conf. Adv. Vis. Interfaces*, 2002, pp. 324–330.

[162] D. Meignan, S. Knust, J.-M. Frayret, G. Pesant, and N. Gaud, "A review and taxonomy of interactive optimization methods in operations research," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 3, p. 17, 2015.

[163] A. Swearngin, A. J. Ko, and J. Fogarty, "Scout: Mixed-initiative exploration of design variations through high-level design constraints," in *Proc. 31st Annu. ACM Symp. Interface Softw. Technol. Adjunct (UIST)*, New York, NY, USA, 2018, pp. 134–136, doi: 10.1145/3266037.3271626.

[164] M. R. Frank and J. D. Foley, "Model-based user interface design by example and by interview," in *Proc. 6th Annu. ACM Symp. Interface Softw. Technol. (UIST)*, New York, NY, USA, 1993, pp. 129–137, doi: 10.1145/168642.168655.

[165] B. Lee, S. Srivastava, R. Kumar, R. Brafman, and S. R. Klemmer, "Designing with interactive example galleries," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2010, pp. 2257–2266.

[166] J. C. Quiroz, S. J. Louis, and S. M. Dascalu, "Interactive evolution of XUL user interfaces," in *Proc. 9th Annu. Conf. Genetic Evol. Comput. (GECCO)*, New York, NY, USA, 2007, pp. 2151–2158, doi: 10.1145/1276958.1277373.

[167] W. L. P. Wong and D. F. Radcliffe, "The tacit nature of design knowledge," *Technol. Anal. Strategic Manage.*, vol. 12, no. 4, pp. 493–512, 2000, doi: 10.1080/713698497.

[168] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: Freeman, 1991.

[169] J. Nielsen, *Usability Engineering*. Amsterdam, The Netherlands: Elsevier, 1994.

[170] N. Cross and R. Roy, *Engineering Design Methods*, vol. 4. New York, NY, USA: Wiley, 1989.

ABOUT THE AUTHORS

Antti Oulasvirta received the Ph.D. degree in cognitive science from the University of Helsinki, Helsinki, Finland, in 2006.

He leads the User Interfaces Research Group, School of Electrical Engineering, Aalto University, Espoo, Finland, and leads the Interactive Artificial Intelligence Research Program at the Finnish Center for AI FCAI. His research interests include computational design and cognitive modeling.

Niraj Ramesh Dayama received the Ph.D. degree in operations research (OR) from the IIT Bombay, Mumbai, India, in 2014.

He is currently a Postdoctoral Researcher with the User Interfaces Research Group, Aalto University, Espoo, Finland. Besides this research background, he has been a Software Delivery Manager and an IT Consultant. His research interests include mathematical modeling and combinatorial optimization.

Morteza Shiripour is working toward the Ph.D. degree at Aalto University, Espoo, Finland.

He is currently working on the graphical layout optimization. His main research interests are focused on applying mathematical modeling, developing exact, and metaheuristic algorithms.

Maximilian John is working toward the Ph.D. degree at the Max Planck Institute for Informatics, Saarbrücken, Germany, and Saarland University, Saarbrücken.

His research focuses on integer programming.

Andreas Karrenbauer received the Ph.D. degree in computer science from Saarland University, Saarbrücken, Germany, in 2007.

He was a Postdoctoral at École polytechnique fédérale de Lausanne (EPFL), Lausanne, Switzerland, from 2008 to 2010, before moving to the University of Konstanz, Konstanz, Germany, to become a Research Fellow and a Research Group Leader. He has been a Senior Researcher with the Algorithms and Complexity Department, Max Planck Institute for Informatics, Saarbrücken, since 2013. His research interest is in the area of mathematical optimization.