Graph4GUI: Graph Neural Networks for Representing Graphical User Interfaces

Yue Jiang
yue.jiang@aalto.fi
Aalto University
Finland

Changkong Zhou
changkong.zhou@aalto.fi
Aalto University
Finland

Vikas Garg*
vgarg@csail.mit.edu
YaiYai Ltd and Aalto University
Finland

Antti Oulasvirta*
antti.oulasvirta@aalto.fi
Aalto University
Finland

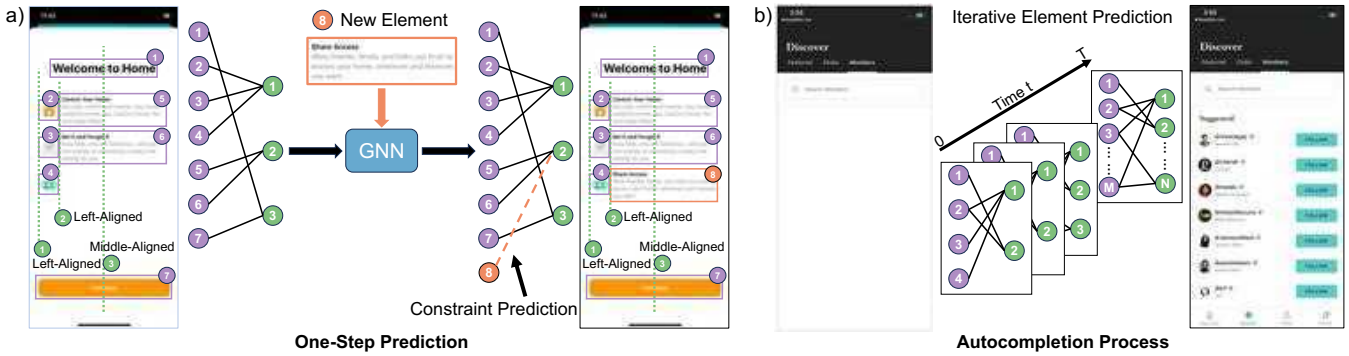

Figure 1: Graph4GUI is a graph-based GUI representation that captures the connections between GUI element properties and constraints. Such representation can capture the visual–spatial–semantic structure of a GUI such that it could be effectively employed in computational design. a) To represent the GUIs, bipartite graphs comprising element nodes (colored purple) convey the GUI elements' properties and constraint nodes (colored green) that can be integrated into graph neural networks. Such representation can serve various downstream tasks, such as predicting constraints (dotted orange edge) for an unplaced element (colored orange). b) By iteratively predicting the sizes and locations of yet-unplaced elements, we can support designers by autocompleting partially completed GUI designs.

ABSTRACT

Present-day graphical user interfaces (GUIs) exhibit diverse arrangements of text, graphics, and interactive elements such as buttons and menus, but representations of GUIs have not kept up. They do not encapsulate both semantic and visuo-spatial relationships among elements. To seize machine learning's potential for GUIs more efficiently, Graph4GUI exploits graph neural networks to capture individual elements' properties and their semantic—visuo-spatial constraints in a layout. The learned representation demonstrated its effectiveness in multiple tasks, especially generating designs in a challenging GUI autocompletion task, which involved predicting the positions of remaining unplaced elements in a partially completed GUI. The new model's suggestions showed alignment and visual appeal superior to the baseline method and received higher subjective ratings for preference. Furthermore, we demonstrate the practical benefits and efficiency advantages designers perceive when utilizing our model as an autocompletion plug-in.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**; **Interaction techniques**.

KEYWORDS

Graphical User Interface, User Interface Representation, Constraint-based Layout, Graph Neural Networks

ACM Reference Format:
Yue Jiang, Changkong Zhou, Vikas Garg, and Antti Oulasvirta. 2024. Graph4GUI: Graph Neural Networks for Representing Graphical User Interfaces. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24), May 11–16, 2024, Honolulu, HI, USA*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3613904.3642822

*Co-last authors, equal contribution.

CHI '24, May 11–16, 2024, Honolulu, HI, USA

ACM ISBN 979-8-4007-0330-0/24/05
https://doi.org/10.1145/3613904.3642822

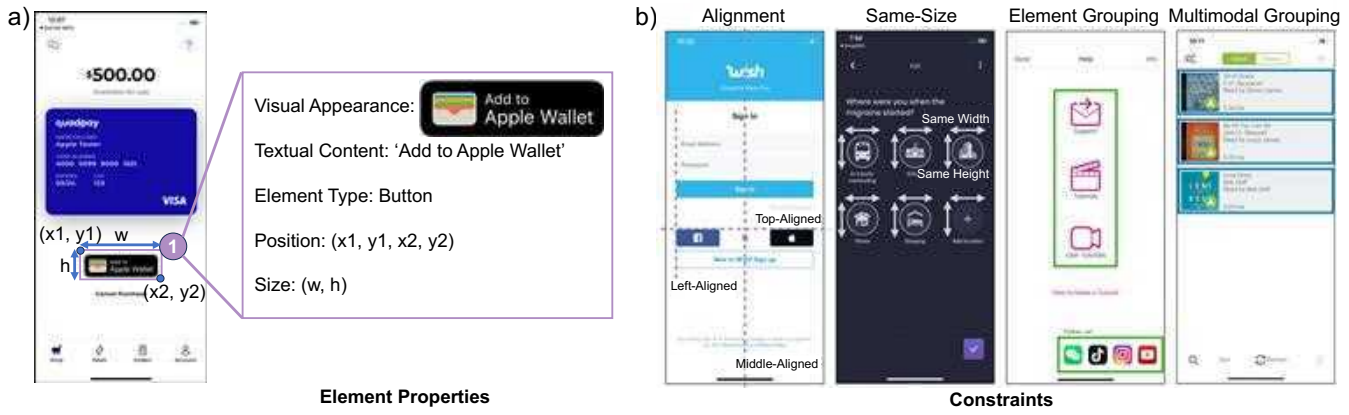

Figure 2: a) Graph4GUI represents each GUI element through a separate node with properties. GUI element nodes convey the element properties, including visual appearance, textual content, element type, position, and size. b) Constraint nodes express four types of constraints: alignment, same-size, element grouping, and multimodal grouping constraints.

1 INTRODUCTION

Modern graphical user interfaces (GUIs) are replete with diverse elements like text, graphics, buttons, checkboxes, sliders, and icons, arranged in various ways. GUIs deploy visual, spatial, and textual cues to guide users in their design. For example, colors convey grouping and attention, while visual cues like proximity or shared visual areas signal element associations [11]. Elements are ordered and grouped based on grid lines; for instance, lists are often left-aligned [53]. In addition, textual elements, such as headers, labels, and annotations, are needed to communicate the "semantics" of the various shapes and images. Despite architectural commonalities, each GUI genre and application has its unique conventions. The question of how to represent a GUI's visual–spatial–semantic structure such that it could be effectively conveyed in computational design remains open [32, 34–36].

Prior methods for representing GUIs and their constituent elements fall short of capturing these integrative aspects. Some work has focused exclusively on textual content in GUIs, but neglected the visual aspects of design and the variety of GUI elements [45, 46]. In contrast, other approaches emphasize visual appearance and GUI element types but overlook the content of the elements [1, 15, 50]. This results in similar treatment for GUIs sharing structural and visual similarities but differing in content. Layout constraints represent the layout relationships between GUI elements, such as alignment, same-size, and grouping. Most existing methods, employing Convolutional Neural Networks (CNNs) to learn GUI images, face challenges because they have to learn layout constraints from pixels. This makes training a model to predict constraints a challenge.

To address this gap, we propose a novel graph-based GUI representation Graph4GUI that integrates GUI elements with layout constraints (Figure 1a). We formulate a bipartite graph to express GUI elements and their relationships via two kinds of nodes: element and constraint nodes. As shown in Figure 2, each GUI element node represents element properties, including visual appearance, textual content, element type, position, and size. Constraint nodes include four types of constraints: alignment, same-size, element grouping, and multimodal grouping constraints. We then employ Graph neural networks (GNNs) to learn a domain-specific representation from the graph-structured data.

Compared to other GUI representations [1, 13, 15, 45–47, 50], the design of our GNN aims to balance two representation learning goals. On the one hand, we want to maximally exploit domain knowledge, particularly using stable, universal GUI design characteristics without learning from scratch. On the other, we want to capture contingent design tendencies – such as color palettes and fonts – without manual specification. If successful, this would make it possible to learn a useful representation with fewer samples. To this end, our approach is to represent relatively universal principles of layouts as constraints in a graph, thus reducing the need to learn them from scratch. At the same time, genre-specific tendencies are learned by applying GNNs to capture the design features unique to GUIs. In contrast, traditional structured representations in computational design, such as DOM trees, are not designed for learning but for specifying GUIs. While they represent the view hierarchy of a layout, they do not lend themselves to many machine learning methods. For example, there is no natural way to represent the concept of grid alignment with DOMs, as this information is split into the leaf nodes of the tree. To compute whether two elements are aligned, the whole tree needs to be parsed. In contrast, our method directly embeds connections between GUI elements and their constraints in the graph.

To examine the effectiveness of our graph-based representation, we applied it to different applications: GUI autocompletion, GUI topic classification, and GUI retrieval. Our primary emphasis lies in autocompletion due to its complexity. This autocompletion problem is challenging, not only because exploring potential GUI element combinations is computationally costly but also because good solutions must consider visual, spatial, and semantic constraints among the to-be-placed elements and those already present. We present a method for iteratively recommending the position and size of unplaced GUI elements, as illustrated in Figure 1b. To augment the model's usability for designers, we introduce alternative element suggestion options, including the recommendation of element

Approach	Textual Content	Visual Appearance	Element Type	View Hierarchy	Layout Constraints
SUGILITE [45]	✓	✗	✗	✓	✗
SOVITE [46]	✓	✗	✗	✓	✗
ERICA [15]	✗	✓	✓	✓	✗
HAMP [1]	✗	✓	✓	✗	✗
Liu et al. [15]	✗	✓	✓	✓	✗
Screen2Vec [47]	✓	✗	✓	✓	✗
Li et al. [48]	✗	✗	✓	✓	✗
Brückner et al. [10]	✗	✗	✓	✗	✗
GRIDS [13]	✗	✗	✓	✗	✓
Ours	✓	✓	✓	✗	✓

Table 1: A comparison of existing approaches, marked based on if they *explicitly represent*: textual content, visual appearance, GUI element type, view hierarchy, and layout constraints. The view hierarchy, akin to a DOM tree, begins with a root view and organizes all its descendants in a tree structure. Layout constraints denote relationships like alignment and grouping among GUI elements. "✓" indicates that the model can capture the factor, while "✗" indicates that it does not.

groups and simultaneous suggestions for all elements. For evaluation, we conducted Study 1, comparing it with GRIDS [13], an approach for autocompletion using integer programming. Our model produced suggestions with superior alignment and visual appeal compared to the baseline, consistent with participants' preferences. In Study 2, we integrated our model into a plug-in for a design tool. It allows GUI designers to utilize autocompletion capabilities in real time while maintaining full control over the design process within the interactive design tool. We interviewed six GUI designers to study our tool's practical benefits and efficiency advantages.

Our work makes the following contributions:

(1) A novel graph representation for GUIs, Graph4GUI, which incorporates GUI element properties such as textual content, visual appearance, and element types, along with their relationships and constraints.
(2) A graph neural network method for learning the graph representation of the GUI to optimize the GUI element dimensions and positions.
(3) An autocompletion framework that serves to demonstrate the graph representation facilitating interactive GUI design. The framework's effectiveness was evaluated through a comparison study and a designer study.
(4) Applying the graph representation to other applications, including GUI topic classification and GUI retrieval.

2 RELATED WORK

This section focuses on the limitations of preexisting representations of GUIs, the GUI-related applications of graph neural networks, and constraint-based approaches to layout generation.

2.1 Representations of GUIs

Existing GUI representations often prioritize specific properties while neglecting others. Table 1 provides a comparison of existing approaches, marked based on whether they explicitly represent textual content, visual appearance, GUI element type, view hierarchy, and layout constraints. Some representations focus on textual content, ignoring visual appearance and GUI element types [45, 46]. In contrast, alternative methods prioritize visual appearance and the types of GUI elements [1, 15, 50], but often overlook the importance of textual content. This can lead to similar treatment of structurally and visually similar GUIs that differ significantly in textual content. Screen2Vec [47] addresses this by generating GUI representations incorporating textual content, element types, and screen hierarchy. Our method, Graph4GUI, extends this consideration by incorporating constraints and interrelationships between GUI elements. ILuvUI [37] proposed a vision language model to create a language representation of GUI. A relevant method, GRIDS [13], is an integer programming method optimizing grid layouts using layout constraints. We also consider layout constraints since they are important in developing a well-structured GUI design. With our approach, Graph4GUI, we propose a solution that considers not only textual content, visual appearance, and GUI element types but also the constraints and interrelationships between GUI elements.

2.2 Graph Neural Networks on GUIs

Graph neural networks [24, 25, 62, 71] are state-of-the-art models for encoding graph-structured data. Whereas CNNs rely on convolution over spatial neighborhoods and enjoy widespread application to encode GUI images, GNNs aggregate information from neighborhoods defined by an input graph that are not restricted to the spatial domain. This gives them the potential to exploit information about the GUIs beyond pixel level. Li et al. applied GNNs to denoise an existing user-interface dataset [44], and performed GUI autocompletion from the GUI layout hierarchy but failed to generate visually realistic GUI results [48]. Brückner et al. [10] looked into constructing a graph using GUI elements' relative positioning to predict elements; however, it proved challenging to learn the layout structure from only relative positions. HAMP [2], introduced a graph representation with nodes for app descriptions, GUI screens, GUI classes, and element images to perform GUI tasks. Still, such detailed metadata cannot be extracted from GUI screenshots without extensive manual annotations. In contrast, our application of GNNs is geared toward modeling the layout graph of GUI elements, thereby enabling us to capture both the topological intricacies of the GUI layout and the properties of individual GUI elements.

2.3 Constraint-based Layout Generation

Constraint-based layout models have been widely used in GUI layouts [3, 5, 6, 8, 28, 40, 51, 52, 60, 61, 63, 66, 68, 72, 75, 76] and document layouts [7, 29, 30, 41]. Early methods like Peridot [55, 57] and Lapidary [74] proposed programming by demonstration, automatically generate constraints for user interfaces based on designer interactions. These models offer greater flexibility for layout generation than simple layout models such as group, grid, table, and grid-bag layouts [54, 56, 58]. Prior work proposed constraint-based layout generation [17, 69]. For instance, SUPPLE [20–22] presented constraints for alternative widgets and groupings, and ORCLayout [33, 38, 39] introduced OR-constraint as a mixture of hard and soft constraints to unify flow-based and constraint-based layouts. Constraints have functioned also to enable layout personalization [18], maintaining consistency [19], giving layout-alternative suggestions based on user-defined constraints [4, 65], generating layout alternatives from templates or modifiable suggestions [31, 64, 73], and allowing both author and viewer to specify the layout [7]. Finally, recent work has explored applying deep-learning approaches to automatic layout generation, eliminating the need for manually defined constraints [42, 77]. However, none of these methods predict constraints for GUI elements. Incorporating GUI element relationships as constraints enables our model to predict them within the network. This enhances the network's ability to establish connections and deepen its comprehension of both element properties and constraints.

3 GUI LAYOUT PROBLEM

Designing a GUI involves carefully selecting elements and organizing them into a structure that is usable and aesthetically appealing. This brings with it a large number of both element-specific and layout-related decisions and is typically iterative in nature [23]. Our objective is to provide a more comprehensive characterization of GUIs compared to existing GUI representations by factoring in visual, spatial, and semantic features. We formulate the problem by partitioning it into elements and layouts while also defining the GUI design problem as an optimization process.

3.1 Element Properties

The first major consideration is *visual appearance*, a broad notion encompassing such properties as color combinations, geometric shapes, and GUI styles. For example, employing tranquil hues such as blues and greens may encourage a calming interface, requiring a more spacious and streamlined layout. In contrast, energetic shades (reds, yellows, etc.) might necessitate a more condensed and high-energy design. Likewise, the arrangement of the various shapes plays a crucial role in the overall visual appeal of the layout. Equally essential is the *textual content* – encompassing all forms of text content visible in the user interface. Labels can play a crucial role from the design perspective. The alignment and arrangement of labels that contain long paragraphs require larger spaces, to avoid clutter and overlapping. Conversely, elements containing brief text strings or bullet-point-style items may allow for compaction, thereby leaving room for other pertinent features. The format of the content is also critical; condensed or expanded layouts especially require careful consideration of element spacing, alignment, and the overall design arrangement. Finally, each *element type*, such as button, checkbox, or text field, has inherent functionality and objectives. For example, buttons need to be easily reachable if user interaction is to be effective, whereas checkboxes might not require such prominent positioning on account of their less frequent use. Text fields' usual dominance as the focal point is due to their role in data entry, which necessitates adequate design space.

3.2 Layout Constraints

Layout-level properties can be approached as constraints. Imposing constraints can help maintain consistency across GUIs and aid users in understanding them. Among commonplace GUI constraints are keeping similar elements the same size, aligning elements along a shared grid, and grouping related elements together. The *alignment* constraint, for instance, enforces uniform positioning and visual consistency by arranging elements along a shared axis, thus maintaining a structure within the layout [53]. In addition to establishing relationships among elements, alignment strengthens the synergy between the elements and the broader layout. The *same-size* constraint is equally essential in that it guarantees maintaining appropriate sizes across GUI elements. This enhances visual harmony by making sure of consistent sizing among related elements. *Element grouping* is important for enhancing the layout's organization and logical structure. This is achieved by bringing together related elements with similar functions. The strategy promotes user-friendly navigation. Finally, *multimodal grouping* constraints lend a structured feel to varied elements, with coherent organization across distinct types. This allows for a harmonious combination of text, images, and other GUI components while remaining respectful of the layout's coherence and uniformity principles.

3.3 Formulation of GUI Layout Problem

We can now define the GUI layout problem as an optimization problem. With this formulation, we decide on the positions and sizes of elements in a GUI, denoted as $e_i = (x_i, y_i, w_i, h_i)$, where the coordinates (x_i, y_i) represent the top-left corner of the i-th element and (w_i, h_i) represents its width and height. Here, we focus on a setting where all the elements are in rectangular bounding boxes.

We define two objective terms, the element loss term ($\mathcal{L}_{\text{ele}}$) and the constraint loss term ($\mathcal{L}_{\text{cons}}$). The first of these encapsulates the properties of the GUI elements, such as visual appearance, texture content, and element type. The constraint loss term covers the layout constraints that guide the GUI design toward an optimal arrangement. The objective function we defined above becomes

$$\begin{aligned}\mathcal{L}(\hat{e}_1, \hat{e}_2, ..., \hat{e}_N, \mathbf{F}; \lambda, \eta) = \mathcal{L}_{\text{ele}}(e_1, e_2, ..., e_N, \hat{e}_1, \hat{e}_2, ..., \hat{e}_N; \eta) \\ + \lambda\, \mathcal{L}_{\text{cons}}(\hat{e}_1, \hat{e}_2, ..., \hat{e}_N, \mathbf{F}),\end{aligned} \tag{1}$$

where $\mathbf{F} = \{f_1, f_2, ..., f_N\}$ pertains to the set of properties for the N GUI elements, including the visual appearance, textual content, and element type. The predicted size and position of each GUI element are denoted as $\hat{e}_i = (\hat{x}_i, \hat{y}_i, \hat{w}_i, \hat{h}_i)$, while the ground-truth sizes and positions are represented by $e_i = (x_i, y_i, w_i, h_i)$. $\lambda > 0$ is the weight of the constraint loss. $\eta > 0$ is the weight of the boundary loss as a part of the element loss term described below.

The element loss term ($\mathcal{L}_{\text{ele}}$) refers to the discrepancy between the predicted and actual values for both positions and sizes of the GUI elements, with a penalty imposed if the predicted elements are outside the interface area:

$$\begin{aligned}&\mathcal{L}_{\text{ele}}(e_1, e_2, ..., e_N, \hat{e}_1, \hat{e}_2, ..., \hat{e}_N; \eta)\\&= \text{MSE}(e_1, e_2, ..., e_N, \hat{e}_1, \hat{e}_2, ..., \hat{e}_N) + \text{B}(\hat{e}_1, \hat{e}_2, ..., \hat{e}_N; \eta).\end{aligned} \tag{2}$$

We implement the Mean Squared Error (MSE) loss function to quantify the level of discrepancy between the predicted and the actual positions and sizes of GUI elements, represented as

$$\text{MSE}(e_1, e_2, ..., e_N, \hat{e}_1, \hat{e}_2, ..., \hat{e}_N) = \frac{1}{N}\sum_{i=1}^{N} |\hat{e}_i - e_i|^2. \tag{3}$$

The boundary constraint is used to penalize a predicted element lying outside the interface's screen space:

$$\begin{aligned}\text{B}(\hat{e}_1, \hat{e}_2, ..., \hat{e}_N; \eta) &= \sum_{t=1}^{N} \text{B}(\hat{e}_t; \eta)\\&= \eta \cdot \sum_{t=1}^{N} \Big(max(-\hat{x}_t, 0) + max(-\hat{y}_t, 0)\\&\qquad + max(\hat{x}_t + \hat{w}_t - w_{\text{UI}}, 0)\\&\qquad + max(\hat{y}_t + \hat{h}_t - h_{\text{UI}}, 0)\Big),\end{aligned} \tag{4}$$

where w_{UI} and h_{UI} are the width and height of the GUI.

We introduce the constraint loss term ($\mathcal{L}_{\text{cons}}$) for estimating the discrepancy between the predicted constraints and the constraints present in the GUI design. This is measured by means of Binary Cross Entropy (BCE), whereby evaluates a binary decision for each constraint, namely, whether it is satisfied or not,

$$\begin{aligned}&\mathcal{L}_{\text{cons}}(e_1, \ldots, e_N, \hat{e}_1, \ldots, \hat{e}_N, \mathbf{F}, C)\\&\qquad = \text{BCE}(C(e_1, \ldots, e_N, \mathbf{F}),\ C(\hat{e}_1, \ldots, \hat{e}_N, \mathbf{F})),\end{aligned} \tag{5}$$

where C represents the constraints based on the elements and element properties calculated as

$$\text{BCE}(c, \hat{c}) = -c \log(\hat{c}) - (1 - c) \log(1 - \hat{c}). \tag{6}$$

The term $-c \log(\hat{c})$ serves to penalize the model when a constraint c that should be satisfied has a predicted probability $\hat{c}$ approaching 0 (where the ground-truth value is 1). This encourages the model to increase the likelihood of satisfying the required constraints. Conversely, the term $-(1-c) \log(1-\hat{c})$ punishes the model when the predicted probability $\hat{c}$ is near 1 for a constraint c that should not be satisfied (since the ground-truth value is 0). This term encourages the model to reduce the likelihood of constraints that ought not be satisfied.

As a result, the formulation of the GUI optimization problem is

$$\begin{aligned}(\hat{e}_1^*, \hat{e}_2^*, ..., \hat{e}_N^*) = \underset{\{\hat{e}_1, \hat{e}_2, ..., \hat{e}_N\}}{\arg\min} \Big(&\frac{1}{N}\sum_{i=1}^{N} (\hat{e}_i - e_i)^2 + \text{B}(\hat{e}_1, \hat{e}_2, ..., \hat{e}_N; \eta)\\&+ \lambda\ \text{BCE}(C(e_1, \ldots, e_N, \mathbf{F}), C(\hat{e}_1, \ldots, \hat{e}_N, \mathbf{F}))\Big).\end{aligned} \tag{7}$$

4 GUI REPRESENTATION

Our proposed method is designed to enrich GUI representation by developing a heterogeneous bipartite graph that covers both GUI element properties and layout constraints, thereby dealing with the intricate arrangement and interrelationships among GUI elements. This graph comprises nodes of two types: GUI element nodes and constraint nodes. The former expresses element properties specific to individual GUI elements, and the latter defines layout constraints for GUI elements in the interface display. Integrating element properties and layout constraints into a single unified graph facilitates a thorough representation of a GUI's elements and layout. While earlier work has integrated element properties into GUI representation [47, 49, 59, 67], our approach brings further benefits by not only accounting for the properties of individual GUI elements but also capturing their interrelationships and spatial arrangements within the overall layout.

To ascertain the linkages between GUI elements and constraints, we connect the respective GUI element nodes to the constraint nodes with edges. Specifically, GUI element nodes can only connect to GUI constraint nodes, and *vice versa*. Consequently, the graph constructed represents the GUI structure by establishing the relations between elements and constraints through its edges.

4.1 Graph Nodes for GUI Elements

In our graph representation Graph4GUI, each GUI element is signified by a separate node with properties identifying its position, size, visual appearance, textual content, and type. We encode these properties into an embedding vector and concatenate them to form a single attribute vector for the node (see Figure 2 a).

4.1.1 Position Embedding. We define a GUI element's position by the coordinates of its top-left and bottom-right points, represented as (x_1, y_1) and (x_2, y_2), respectively. These coordinates specify the element's position and size in the GUI. To represent the position within the graph node, a trainable parametric matrix of size $(max(w, h) + 1) \times 256$ is used to encode the position, where w and h are the GUI's width and height, respectively. This matrix maps any coordinate to a 16-dimensional vector. On feeding the four coordinates into this matrix and flattening the resulting embedding, a 64-dimensional vector is output as the position embedding.

4.1.2 Size Embedding. While an element's size can be derived from its position, it is useful to include a size embedding in our representation, especially for tasks such as GUI autocompletion that require a new element to be placed in the GUI at an unknown position. We embed the element size using a process similar to position embedding. This yields a 256-dimensional-vector size embedding.

4.1.3 Visual Appearance Embedding. We encode the visual appearance of GUI elements by extracting high-level features and converting them into a feature vector, which serves as the element's visual representation.

4.1.4 Textual Content Embedding. The textual content of elements is represented by encoding textual information into a vector that captures the semantic meaning and context properties of the text.

4.1.5 Element Type Embedding. GUI element types are represented as one-hot vectors. For example, in a dataset that contains the

three element types Text, List Item, and Button, a button element will be assigned a vector of [0, 0, 1] as its type. In the case of our dataset, which contains 18 distinct element types, the element type embedding is a one-hot vector of length 18, which is processed by a trainable matrix to produce the embedding for the element type.

4.2 Graph Nodes for Constraints

Our constructed graph is designed to generalize the definition of constraints. We represent constraints of different types as separate nodes in the graph, which enables easy extension of the graph to include additional kinds of constraints in the future. Currently, we represent four types of constraints as nodes in the graph: alignment constraints, same-size constraints, element grouping constraints, and multimodal constraints (see Figure 2 b).

4.2.1 Alignment Constraint Nodes. We incorporate alignment constraint nodes into our graph to stipulate the positional affiliation among GUI elements. Each node comprises attributes symbolizing the kind of alignment and the line employed for element alignment. Edges are established between GUI elements and their respective alignment constraint nodes to signify their correlation with alignment. Alignment constraints can express any of six distinct alignments – namely "left-aligned," "top-aligned," "right-aligned," "bottom-aligned," "vertical midline-aligned," and "horizontal midline-aligned." We employ a one-hot vector to express the alignment type. For instance, a left-alignment type is expressed as $[1, 0, 0, 0, 0, 0]$, indicating left-alignment. We further characterize the alignment line using a two-dimensional vector – e.g., $[a, 0]$ represents the elements being aligned with $x = a$. We concatenate the alignment type and line representations to produce an eight-dimensional vector.

4.2.2 Same-Size Constraint Nodes. To portray GUI elements of the same width or height within the GUI in graphical terms, we devised the notion of uniform size constraint nodes. There are two types of size constraints in our design: identical width and similar height. We consolidate the size type and size value into a single-node attribute vector instead of defining them as a one-hot vector. For example, the identical width constraints are defined by $[w, 0]$, where w is the width value, while constraints dictating identical height are defined by $[0, h]$, where h is the height value.

4.2.3 Element Grouping Constraint Nodes. Incorporating consideration of related components enhances the structure of GUIs, particularly with regard to elements with comparable functions or belonging to the same category. To depict the element grouping constraints, we define element grouping constraint nodes. We then connect related GUI element nodes to the corresponding grouping constraint node, signifying their inclusion in a particular group.

4.2.4 Multimodal Grouping Constraint Nodes. Multimodal grouping constraints enable structured organization of elements of differing types: text, pictures, and other GUI components. We create a set of nodes for each multimodal grouping constraint and correlate the relevant elements with their respective nodes. By establishing ties between GUI element nodes and multimodal grouping constraint nodes, we signify placing elements that differ in mode within the same group. In cases of additional multimodal grouping constraints, we create new types of multimodal grouping constraint nodes and repeat the process.

4.3 Learning GUI Layout Design with Graph Neural Networks

As Figure 1a illustrates, we create a graph representation of a GUI layout by organizing GUI element nodes and constraint nodes. Again, these nodes are connected by edges, representing the relationships between elements and constraints. To facilitate GUI design, we can train a graph neural network to take this graph as input and optimize the layout.

4.3.1 Graph Construction. The heterogeneous bipartite graph, $\mathcal{G} = (E \cup C, A)$, is constructed from M GUI element nodes and their N corresponding constraint nodes. The former set of nodes is represented by $E = \{e_1, e_2, ..., e_M\}$, and the set of their constraint nodes is denoted by $C = \{c_1, c_2, ..., c_N\}$. In constructing links between element nodes and constraint nodes, we assign the adjacency matrix A. The value of an element $a_{i,j}$ in that adjacency matrix is set to 1 when the ith GUI element is satisfied with the jth constraint; otherwise, it is 0.

4.3.2 Predicting GUI Element Dimensions and Positions. The primary task of our GNN model is to predict the dimensions and positions of the GUI elements. The predicted GUI element attributes are denoted by $\hat{e}_i = (\hat{x}_i, \hat{y}_i, \hat{w}_i, \hat{h}_i)$. These predictions are produced through parameterized functions specific to the GNN model. The model parameters are denoted by θ, and the model that generates the GUI element predictions is denoted as GNN_θ. The predicted GUI elements $\hat{e}_i$ are then computed as

$$\hat{e}_i = \text{GNN}_\theta(\mathcal{G}). \tag{8}$$

4.3.3 Optimization of GNN Parameters. The GNN model's θ parameters are optimized by minimizing the previously defined objective function $\mathcal{L}$; see Subsection 3.3. The optimization process can be represented as

$$\theta^* = \arg\min_\theta \mathcal{L}(\hat{e}_1, \hat{e}_2, ..., \hat{e}_N, \mathbf{F}; \theta), \tag{9}$$

where θ^* denotes the optimized parameters of the GNN model. After optimizing the GNN model's parameters, θ^*, we can use this model to design new GUI layouts. For a new design, the trained GNN takes the graph-form representation of GUI elements and constraints as input, and then outputs its predicted dimensions and positions for GUI elements. Our approach makes it easy to incorporate new design constraints by modifying the construction of the graph or the objective function.

5 GUI AUTOCOMPLETION

To demonstrate the utility of our graph representation, we propose an autocompletion method that uses our representation approach to enable interactive iterative design. GUI autocompletion is challenging due to the computational complexity involved in accurately predicting suitable GUI elements. Given fixed screen dimensions, our method automatically generates suggestions for finishing a partially completed GUI layout by iteratively predicting the positions of remaining unplaced GUI elements. Our method suggests

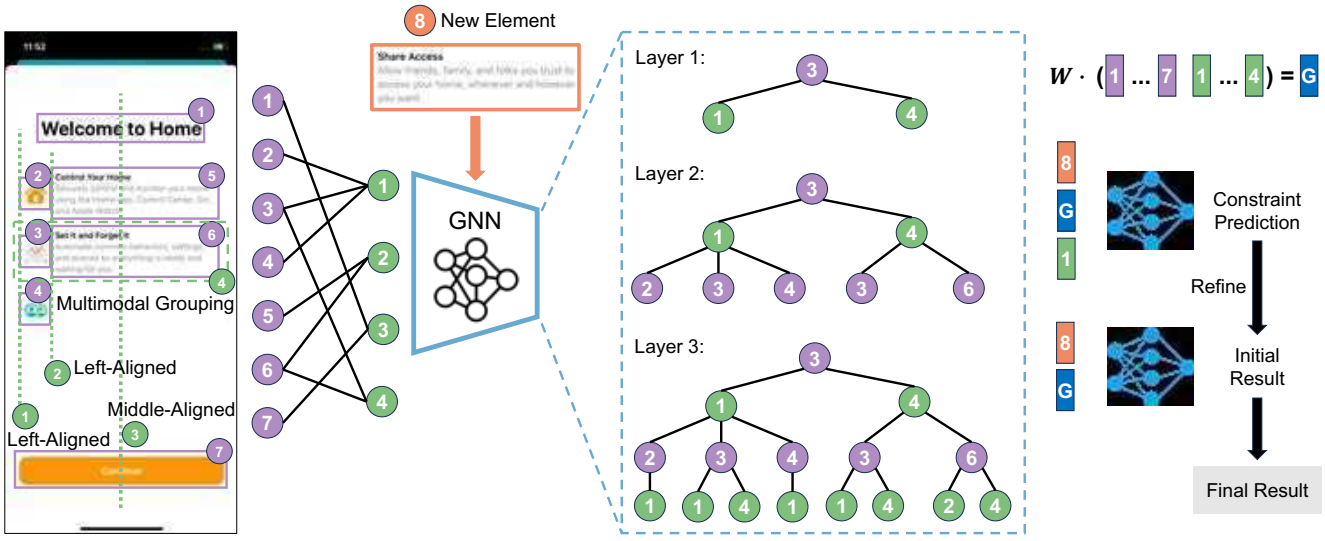

Figure 3: Graph4GUI was adapted for the autocompletion task: We first encode the graph representation of the GUI via the GNN. We only illustrate some parts of the graph for simplicity. Element 8 is the target to-be-placed element. In each GNN layer, nodes perform aggregation from their respective neighbors. To illustrate, consider element node 3. As it goes through the GNN layers, it accumulates information from related constraint nodes and other element nodes. This process results in feature embedding vectors for all nodes, including both element nodes and constraint nodes within the graph. We compute the graph embedding as a weighted average of the node embeddings with the weight matrix W. We then concatenate the target element's embedding vector, the graph embedding, and a constraint embedding and send it to fully connected layers to predict whether the target to-be-placed element should satisfy the constraint. Simultaneously, we concatenate the target element's embedding and the graph embedding to predict the initial position and size of the target element. Integrating these predictions with the constraints, we subsequently refine the position and size to obtain the final results.

position, size, and confidence level for each unplaced element based on the partial GUI. It enables designers to receive suggestions when they complete the design of each element, without the need to predefine all GUI elements beforehand. Moreover, if designers have additional unplaced GUI elements ready, our method iterates over each, providing suggestions for their positions, sizes, and confidence levels. It can significantly reduce the manual effort required for design. Autocompletion that produces high-quality GUIs is rendered difficult by the high computational complexity of evaluating all possible combinations of GUI elements.

Prior studies have explored the autocompletion task; however, they were only capable of handling wireframes. Li et al. [48] used the GUI layout hierarchy to perform GUI autocompletion. Although the hierarchy does capture the structure of the layout, it accounts for only the grouping and containment relationships between GUI elements. It neglects the alignments and relative sizes, which are important layout constraints. On the other hand, Brückner et al. [10] proposed a method of constructing a graph with reference to differences in position between GUI elements. However, it does not consider the properties of GUI elements. GRIDS [13] is a grid-layout-based optimization approach for autocompletion considering constraints such as alignment and grouping. With integer programming, GRIDS produces results by searching for optimal available placements for unplaced elements. Our method, considering both element properties and constraints, fills the gap, crossing the void to generate more desirable predictions.

5.1 Target GUI Element Prediction

Given a partial GUI, we set out to predict both the size and the position of a target element (with a fixed aspect ratio) and the associated constraints it should follow. As shown in Figure 3, the process begins with constructing a graph representation of the partial GUI, denoted as $\mathcal{G}_p$. Exploiting GNNs, we encode this graph to yield feature vectors for all the nodes, including element nodes and constraint nodes within the graph. Within each GNN layer, nodes are aggregated from their respective neighbors. Going through the GNN layers, each node iteratively accumulates information from its associated constraint nodes and other element nodes.

$$\mathcal{G}_p \xrightarrow{\text{GNN}} \{\mathbf{h}_{ele,1}, \mathbf{h}_{ele,2}, ...\}, \{\mathbf{h}_{align,1}, \mathbf{h}_{align,2}, ...\}, \{\mathbf{h}_{size,1}, \mathbf{h}_{size,2}, ...\}, \{\mathbf{h}_{eg,1}, \mathbf{h}_{eg,2}, ...\}, \{\mathbf{h}_{mg,1}, \mathbf{h}_{mg,2}, ...\}, \tag{10}$$

where $\{h_{ele,1}, h_{ele,2}, ...\}$ are the feature vectors for element nodes, $\{h_{align,1}, h_{align,2}, ...\}$ those for alignment constraint nodes, $\{h_{size,1}, h_{size,2}, ...\}$ those for size constraint nodes, $\{h_{eg,1}, h_{eg,2}, ...\}$ the ones for element grouping constraint nodes, and $\{h_{mg,1}, h_{mg,2}, ...\}$ the vectors for multimodal grouping constraint nodes.

After this, the feature vector $h_{\mathcal{G}_p}$ for the entire graph $\mathcal{G}_p$ is computed. This vector is obtained by way of the weighted summation of the average feature vectors of each node type. The weight matrices $\mathbf{W}ele$, $\mathbf{W}align$, $\mathbf{W}size$, $\mathbf{W}eg$, and $\mathbf{W}_{mg}$ are trained alongside GNN parameters, making sure an end-to-end training process ensues that eliminates manual selection:

$$\begin{aligned} h_{\mathcal{G}} =& \mathbf{W}_{ele} \cdot \text{avg}\{\mathbf{h}_{ele,1}, \mathbf{h}_{ele,2}, ...\} + \mathbf{W}_{align} \cdot \text{avg}\{\mathbf{h}_{align,1}, \mathbf{h}_{align,2}, ...\} \\ &+ \mathbf{W}_{size} \cdot \text{avg}\{\mathbf{h}_{size,1}, \mathbf{h}_{size,2}, ...\} + \mathbf{W}_{eg} \cdot \text{avg}\{\mathbf{h}_{eg,1}, \mathbf{h}_{eg,2}, ...\} \\ &+ \mathbf{W}_{mg} \cdot \text{avg}\{\mathbf{h}_{mg,1}, \mathbf{h}_{mg,2}, ...\}. \end{aligned} \tag{11}$$

The resulting graph feature vector and the embedding of the target element are concatenated and fed into fully connected layers. This facilitates position and size predictions for the target GUI element $\hat{e}_t = (\hat{x}_t, \hat{y}_t, \hat{w}_t, \hat{h}_t)$. Furthermore, our approach extends to predicting the constraints that should be satisfied by the target GUI element. We utilize the objective function outlined in Section 3 to perform fine optimization of the target element's positioning, dimensions, and adherence to constraints. For each constraint within the partial GUI, we concatenate the graph feature vector, the embedding of the target element, and the specific constraint's feature vector. This concatenated vector is propagated through fully connected layers to predict the probability of each constraint required for the target GUI element's satisfaction. Simultaneously, we concatenate the target element's embedding and the graph embedding to predict the initial position and size of the target element. Integrating these predictions with the constraints, we subsequently refine the position and size to obtain the final results.

5.2 Confidence Levels

To guide the process of ascertaining the level of confidence in the outcomes, below we describe how we compute confidence levels such that we can avoid offering potentially questionable predictions. Conveying the confidence level – whether it is low, medium, or high – of a certain prediction enables software tools and designers to take suitably informed actions. An application of this feature will be discussed later in the designer study.

5.2.1 High Confidence. High confidence is validated in terms of alignment and uniform size constraints. When the disparity between the predicted alignment line and the position of the target element falls below the threshold value σ, refinement is performed by aligning the position with the projected alignment line. For instance, if the predicted constraint entails left-alignment with the line $x = a$, and if $|\hat{x}_t - a| \leq \sigma$, the $\hat{x}_t$ value is adjusted to match a. This threshold was set to $\sigma = 20$ pixels in our experiments. Likewise, when the difference between the size of the target element and the sizes of other elements, as predicted by constraints promoting uniformity, is below the σ limit, the size is adjusted to match the uniform size value. For example, if the projected constraint dictates that the target element should possess the same width as elements with a width of b, and if $|\hat{w}_t - b| \leq \sigma$, the $\hat{w}_t$ value is adjusted to align with b. When both $(\hat{x}_t, \hat{y}_t)$ and at least one of $\hat{w}_t$ and $\hat{h}_t$ can be verified, we assign a *high* confidence level to the outcome, since the fixed aspect ratio of the target element allows deducing the remaining attribute.

5.2.2 Medium Confidence. Medium confidence is established via element and multimodal grouping constraints. In scenarios wherein confirmation is unattainable for $(\hat{x}_t, \hat{y}_t)$ and at least one of $\hat{w}_t$ and $\hat{h}_t$, grouping constraints come into play. These constraints, encompassing both element and multimodal grouping, reveal patterns among elements and can be exploited for refinement of positions and sizes. For example, vertical groupings often entail consistent widths and equidistant spacing between elements vertically. Consequently, if $|\hat{w}_t - \text{avg}(w_i)| \leq \sigma$, where w_i represents widths of other elements in the target's vertical group, $\hat{w}_t$ is adjusted to match $\text{avg}(w_i)$. If $(\hat{w}t - w_l) - \text{avg}(|w_i - wi - 1|) \leq \sigma$, the $\hat{w}t$ value is set to $w_l + \text{avg}(|w_i - wi - 1|)$. In instances where $(\hat{x}_t, \hat{y}_t)$, along with at least one of $\hat{w}_t$ and $\hat{h}_t$, can be verified, the result gets accorded a *medium* confidence rating.

5.2.3 Low Confidence. In all other cases, the outcomes are assigned a *low* confidence rating.

6 EXPERIMENTS FOR AUTOCOMPLETION

We conducted experiments for the autocompletion task to show the effectiveness of our representation. We created a dataset with partial GUIs and then evaluated our method's prediction quality through qualitative and quantitative experiments. Additionally, we conducted an ablation study to demonstrate the necessity of each constraint type taken into account.

6.1 Dataset and Training Process

For the evaluation, we took the ENRICO dataset [43], a subset of the RICO dataset [14] including cleaner mobile GUI information and the VINS [12] GUI dataset, as our basis for creating a mobile dataset for GUI autocompletion. We improved the dataset's quality through several steps. Initially, we excluded layouts in the dataset that contain three or fewer. GUI elements. After this, we employed the UIED model [70] to enhance the precision of element types and refine the bounding boxes of the GUI elements. We then made further adjustments manually to correct the bounding boxes of the elements. Our refined dataset contains 5,653 GUIs in total.

To evaluate our model's performance, we followed a fivefold cross-validation approach; this technique involved partitioning the GUI dataset into five equal-sized folds. Four of the folds, with approximately 4,522 GUIs, served for training our model, while we reserved one fold, encompassing around 1,131 GUIs, for testing. In our experiments, each fold was utilized once for testing, with the remaining four folds serving as the training data. Using fivefold cross-validation helps to validate the generalization of the model to unseen data and offers a more comprehensive view of the model's behavior by averaging its performance across multiple test sets, thus reducing the impact of random variations in the data.

To create our dataset for GUI autocompletion, for each GUI, we randomly kept a chunk of GUI elements on the display. We removed other GUI elements to create a partial GUI and store the potential "next GUI element" to be added for completing the given partial GUI. By this mechanism, we obtained a partially completed GUI with missing elements and a target element that we need to predict, given the partially completed GUI. Note that each partial GUI often had more than one potential target GUI element. With this method, we can generate various partial GUIs and corresponding target

elements. We generated partial GUIs for training from the complete GUIs in the training data, doing similarly for the test data. In total, each fold of complete GUIs yielded approximately 171,212 pairs of partial GUIs and corresponding target elements for training. Consequently, for each experiment, we used a training dataset containing about 684,849 incomplete-GUI–target pairs and a test dataset comprising approximately 171,212 pairs.

6.2 Implementation Details

6.2.1 Embeddings. We encoded the visual appearance of the element by using a pre-trained ResNet152 model [27]. Through this model, which is able to extract high-level features from images, we generated a feature vector that represents the element's visual appearance. For encoding the textual content of the GUI elements, we used a pre-trained BERT model [16]. A Transformer-based neural-network architecture pre-trained on a large corpus of text data, BERT can generate a 768-dimensional vector representing the text, which we applied to extract features from the interface elements. In a technique that improves the efficiency of models utilizing this representation, we introduced an "unknown" token for infrequent words. Infrequent words are often inadequately represented in training data. That can lead to overfitting. Incorporating an unknown token enables the model to generalize its predictions for previously unseen words and simplify the representation to facilitate the model's processing. To implement this approach, we began by computing the frequency of each text element. If the text occurred fewer than three times, we replaced it with the special token [UNK] in BERT, representing an unknown word. We then used BERT to generate the text content embedding.

6.2.2 Graph Neural Networks. We applied the SAGEConv model [26], a variety of GNN models that is suited to training heterogeneous graphs. The SAGEConv model employs a message-passing technique to propagate information through the graph to convey it from a node's neighborhood to the node itself, thereby improving its feature representation. SAGEConv can capture the relationships between nodes of different types in the graph. The output feature vectors are 256-dimensional $h \in \mathbb{R}^{256}$. The trainable weights for computing $\mathbf{W}_{ele}$, $\mathbf{W}_{align}$, $\mathbf{W}_{size}$, $\mathbf{W}_{eg}$, $\mathbf{W}_{mg} \in \mathbb{R}$ are in the dimension of 256×256.

6.3 Qualitative Evaluation

To enhance the usability of the model for designers, we introduce three types of element suggestion options and show the results.

6.3.1 Suggesting a Single Element. As Figure 1b indicates, iteratively predicting the sizes and positions of yet-unplaced elements by means of the updated graph representation helps support designers by autocompleting partially completed designs. By default, we loop over all elements still to be placed and select the one with the highest associated confidence level to add (Figure 4a). Furthermore, our setting allowed designer-in-the-loop interaction wherein designers can make adjustments to the GUI design as their preferences dictate after every iteration. They could move the element, resize it, or select an alternative GUI element for placement. The changes are visible immediately, and the underlying graph representation gets updated accordingly, so that subsequent predictions can work from it, as demonstrated in Figure 4b.

6.3.2 Suggesting a Group of Elements. Our model predicts grouping constraints for each element based on the partial GUI. As shown in Figure 4c, if multiple elements share the same grouping constraint, our model suggests these grouped elements together, thereby expediting the prediction process.

6.3.3 Suggesting All Elements. Alternatively, in Figure 4d, the model can predict all elements simultaneously. The final results iterate over each element, placing those with the highest confidence levels first, based on the updated partial GUI. While providing a complete view, modifying results can be more challenging compared to adjustments in the iterative prediction process.

6.4 Quantitative Evaluation

To evaluate the accuracy of our autocompletion approach, we assessed its single-step prediction by three metrics. For this purpose, we denoted the top-left point of the predicted GUI element as $(\hat{x}, \hat{y})$ and its corresponding ground truth as (x, y). Similarly, we denoted the predicted size of the target GUI element as $(\hat{w}, \hat{h})$ and its corresponding ground truth as (w, h). Finally, w_{UI} and h_{UI} represent the width and height of the user interface.

6.4.1 Metrics. We established separate metrics for assessing the predictions' accuracy with regard to position, size, and alignment. All three metrics have a range between 0 and 1, where a lower value indicates a better prediction.

- **Position Error (PosError):** The PosError metric measures the relative distance between the predicted position of the GUI element and the corresponding ground-truth position. Calculating the error entails ascertaining the distance between the predicted and ground-truth positions, then normalizing it by the maximum possible distance that the element can move,

$$\mathcal{L}_{\text{PosError}} = \frac{||(\hat{x}, \hat{y}) - (x, y)||_2}{\sqrt{(w_{\text{UI}} - \hat{w})^2 + (h_{\text{UI}} - \hat{h})^2}}. \tag{12}$$

- **Area Error (AreaError):** The AreaError metric evaluates the difference between the predicted size of the GUI element and the corresponding ground-truth size. The difference between the predicted and ground-truth sizes is normalized in terms of the maximum size between the predicted size and the ground-truth size,

$$\mathcal{L}_{\text{AreaError}} = \frac{|\hat{w} \cdot \hat{h} - w \cdot h|}{\max(\hat{w} \cdot \hat{h}, w \cdot h)}. \tag{13}$$

- **Alignment Error (AlignError):** The AlignError metric judges the proportion of the alignments predicted correctly. This figure is calculated by dividing the number of correctly predicted alignments for the target element by the total number of alignments that the predicted element should satisfy.

6.4.2 Comparison. We compared our model and GRIDS [13], an optimization approach based on grid layout designed for autocompletion while considering constraints, including alignment, element location, rectangular outline, and preferred element positions. We

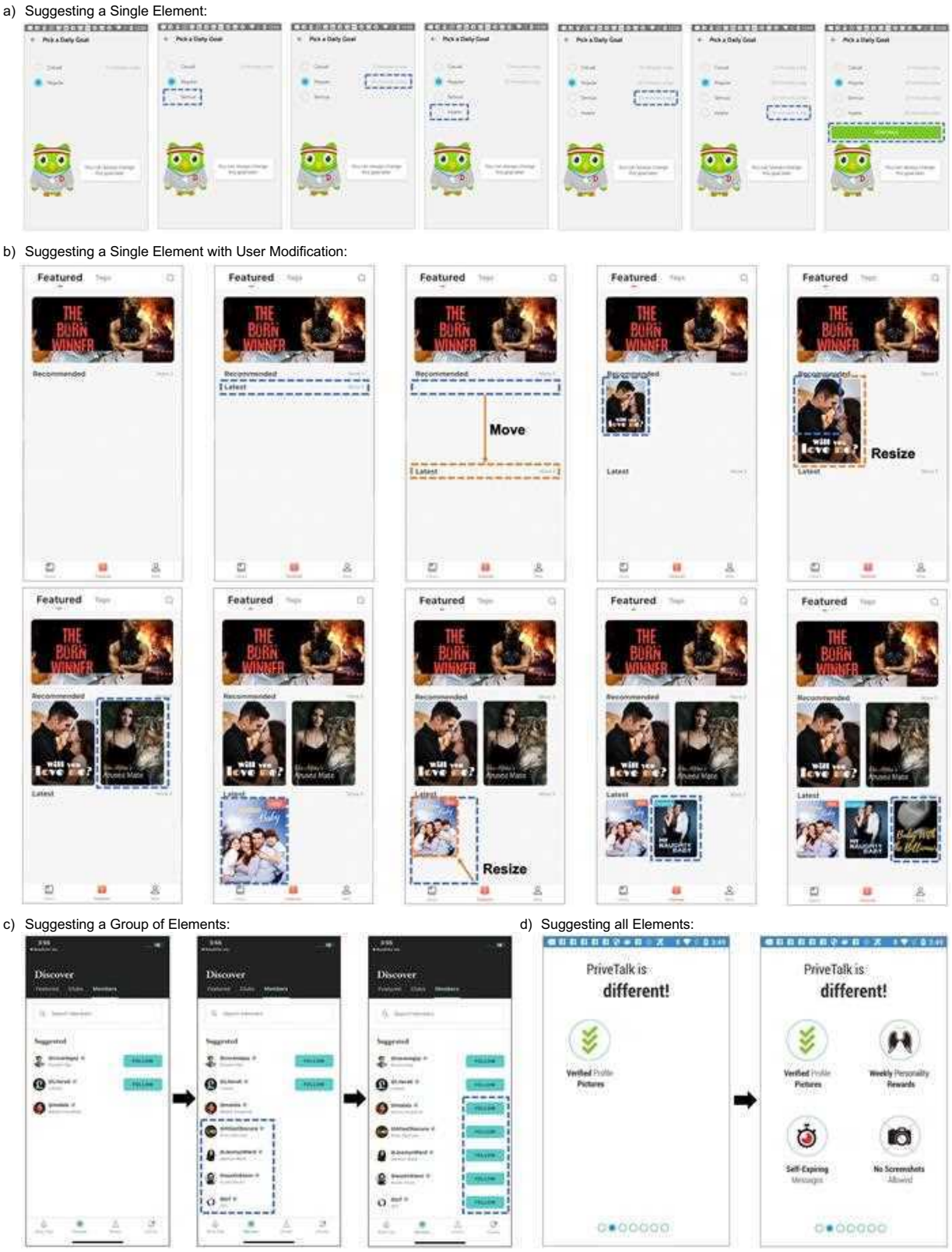

Figure 4: a) Our model can iteratively predict unplaced GUI elements (shown in blue bounding boxes). b) Designers can make adjustments (orange), including moving, resizing, or re-selecting GUI elements. c) The model's capability to predict groupings allows for the placement of elements together as a group. d) The model can also predict all the elements simultaneously.

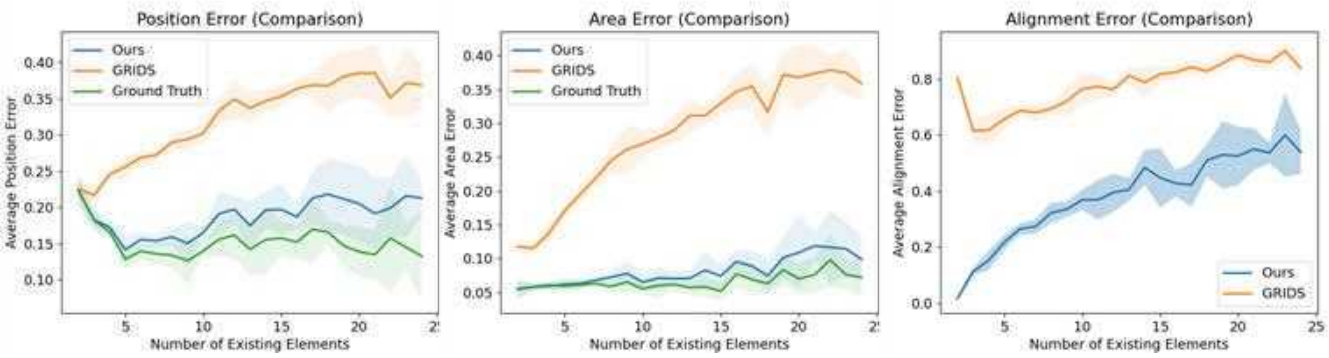

Figure 5: Comparison of our model with GRIDS [13], an autocompletion approach using integer programming, and the established upper bound for the model's performance exploiting ground-truth constraints to predict positions and sizes. The evaluation used three metrics: position error, size error, and alignment error. This comparison incorporates fivefold cross-validation to assure reliability, with the mean and standard deviation illustrated in the corresponding plots.

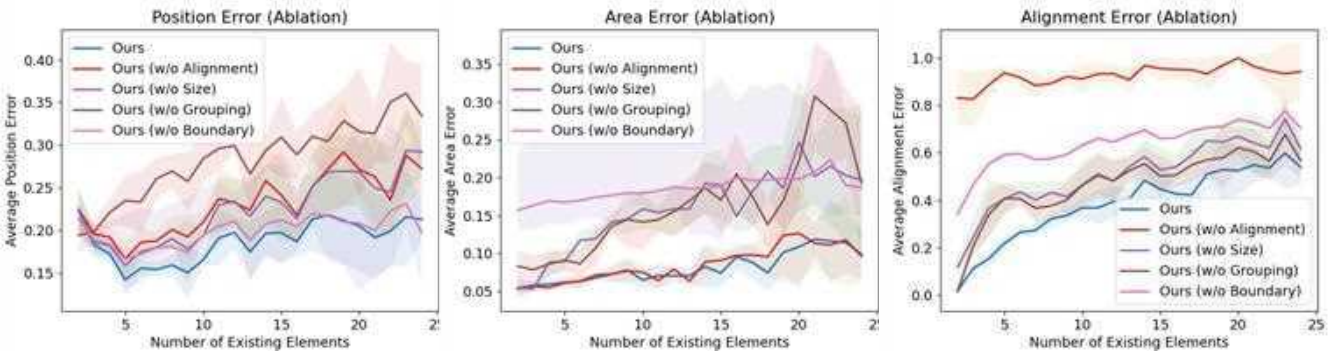

Figure 6: Results from an ablation study comparing our model's performance to ablated models in which each type of constraint has been removed.

chose GRIDS for comparison due to the following reasons: 1) it also takes constraints into account, 2) the majority of existing GUI presentations do not perform autocompletion, and enabling autocompletion is non-trivial, 3) other relevant prior works [10, 48] addressing autocompletion tasks are not open-sourced. With integer programming, GRIDS produces results by seeking the optimal placements available for unplaced elements. It generates multiple optimized solutions, each accompanied by a confidence value. We chose the solution with the highest confidence value among each model's first 10 solutions. In addition, we established an upper bound for the model's performance by employing ground-truth constraints to predict element positions and sizes. Because of the ambiguity of GUI element placement, the ground truth thus defined was not assigned a zero-loss value. The ambiguity arises from the fact that some elements do not satisfy enough constraints; that issue, in turn, makes accurate prediction of their placement challenging. Our comparison by all three metrics was performed relative to the number of preexisting elements in the partial GUIs, as Figure 5 illustrates. We used fivefold cross-validation for the comparisons (our plots present both mean and standard deviation values), with a sample of 10,000 test data from each fold used for model evaluation. Since our technique uses ground-truth alignments in predicting the ground truth, we do not have ground-truth results for alignment error. The results show that our model predicts more accurate positions and sizes, with more accurate alignments, than GRIDS. Finally, we computed the inference time needed by the models. Our model performs one-step predictions in 0.148 seconds, on average, with our test data on a single RTX4090 GPU, while GRIDS takes much longer, at 67.4 seconds.

6.4.3 The Ablation Study. Our ablation study compared the proposed model to ablated models, each lacking one specific type of constraint. This ablation study, the results of which are depicted in Figure 6, showed the necessity of each constraint for our model's performance.

7 COMPARISON STUDY

We performed a comparison study to evaluate our model against GRIDS [13] for one-step prediction. We got test images randomly sampled from the test dataset described in subsection 6.1. All the images are mobile GUIs. As Graph4GUI optimizes positions and sizes with content and graphics, while GRIDS only handles wireframe layouts, the comparison focuses on wireframes. To ensure fairness, we trained our model on wireframe GUIs without visual appearance, textual content, or element type, showcasing higher perceived quality despite not fully utilizing Graph4GUI's capabilities.

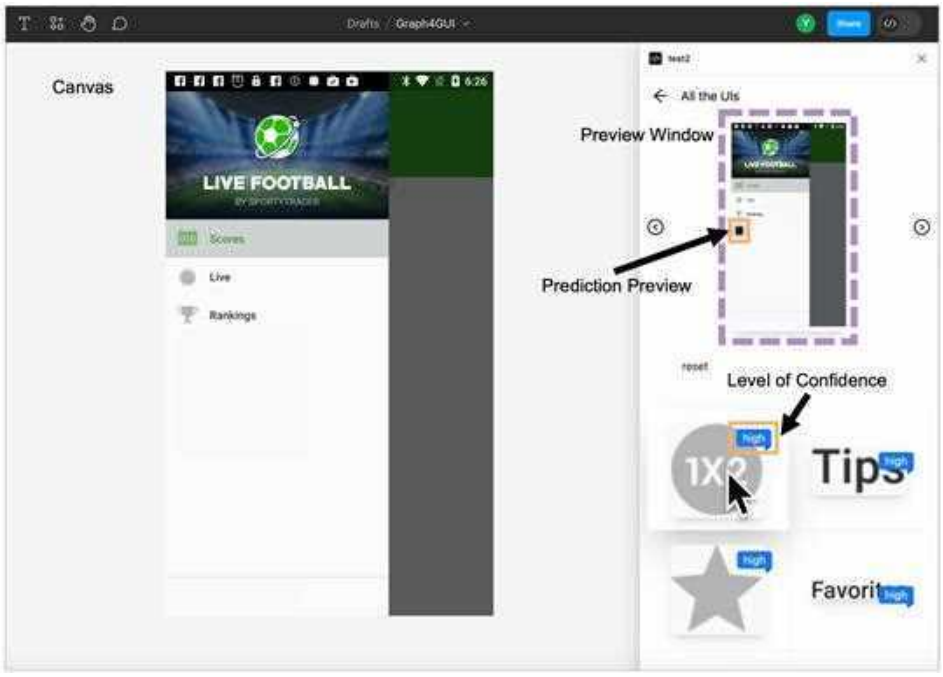

Figure 7: We implemented our method as a Figma plug-in. The plug-in offers GUI element prediction suggestions with confidence levels, helping designers prioritize element selection. It also features a preview window that shows a prediction preview of the element placed on the GUI when hovering over an element, providing an intuitive view to help decision-making. After selecting a GUI element, the plug-in can automatically place it in the suggested position and size on the canvas.

7.1 Method

7.1.1 Participants. We enlisted 35 participants (25 female, 9 male, 1 other) through social media, averaging 26.49 years (SD = 3.07). All had normal or corrected-to-normal vision, and none were colorblind. Local regulations did not mandate formal ethics review.

7.1.2 Experimental Design. From 1,000 randomly sampled partial GUIs with a to-be-placed element, we randomly selected 100 presented to each participant for comparison between our method and the GRIDS method.

7.1.3 Apparatus. Pairs of GUI images were presented side by side on a custom webpage in randomized order.

7.1.4 Procedure. After completing a demographics questionnaire, participants viewed GUI pairs and selected the preferred one based on personal criteria like design, layout, or aesthetics. Preferences were indicated by choosing the left or right image, or "They are equally good". Participants could assess up to 100 pairs, stopping at their discretion.

7.2 Findings

We received responses for 3,367 image pairs from 35 participants. Preferences were as follows: 456 for GRIDS (13.54%), 2,368 for our model (70.33%), and 543 expressing no preference (16.13%). The difference between our method and GRIDS was statistically significant ($\chi^2 = 3115.8, p < .001$). This finding attests that our model indeed produces more visually appealing suggestions that show better alignment than the baseline method's output, backing up our conclusions with evidence from participants' preferences.

8 DESIGNER STUDY

We conducted a user study to evaluate the effectiveness and usability of our method for assisting with GUI autocompletion. The aim was to assess both the impact on design efficiency and the subjective experience. To guide the design of the study, we established the following objectives:

(1) Determine whether our technique enables designers to enhance the efficiency of the design process.
(2) Evaluate the quality of suggestions provided by our model.
(3) Explore how designers utilize each function of the tool, specifically the element prediction preview, the element prediction, and the confidence rating for the predictions.
(4) Ascertain whether designers perceived our technique as helpful for their GUI design practice.

8.1 Method

8.1.1 Plug-in. Our method is implemented as a Figma plug-in (Figure 7), offering GUI element predictions with confidence levels. Given a partially completed GUI and a list of elements to be placed, the plug-in computes confidence levels for each prediction, aiding designers in prioritizing placements. The plug-in includes a preview window that displays a wireframe version of the GUI when hovering over an element, allowing designers to assess results intuitively.

Upon selecting a GUI element, the plug-in automatically positions and sizes it on the canvas.

8.1.2 Participants. Six GUI designers with diverse experience levels were recruited through email lists, local networks, and social media platforms. Participants, aged 21 to 33 (mean age 26.5), were either UI/UX designers or HCI/design students, all experienced in Figma for GUI design. The group comprised four females and two males. Prior to the study, participants received full information about the conditions and gave informed consent. Local regulations do not require formal ethics review.

8.1.3 Materials. Participants used the lab's laptops for various activities, including interacting with the Figma interface using our plug-in and filling out a questionnaire. The study involved a practice task to acquaint participants with Figma and our plug-in, and six GUI design tasks (three with our plug-in and three without). Each task provided a brief outlining the GUI design purpose, a GUI to be completed, and a list of elements to place.

8.1.4 Experiment Design. The study used a within-subject design, exposing participants to two conditions. In one condition, they freely used our plug-in with Figma's standard features to complete three GUIs (login, shopping, and menu pages). In the baseline condition, participants completed the same GUIs without the plug-in. Task and condition order were fully counterbalanced to eliminate any potential bias.

8.1.5 Procedure. After signing the consent form, participants completed a design background questionnaire and underwent a Figma tutorial. They were then tasked with creating six GUIs in two conditions: with and without our plug-in. After each task, participants rated their designs and assessed the perceived task load. In the with-plug-in condition, they also evaluated the plug-in's specific features. We used the System Usability Scale (SUS) for overall usability assessment. We conducted final interviews for participants to compare their experiences, provide feedback on plug-in features, and identify weaknesses in our tool.

8.2 Quantitative Findings

Our tool's usability and helpfulness were quantitatively evaluated using the System Usability Score, participant ratings of features and resulting GUIs, and task completion times.

8.2.1 System Usability Score. Following established practices [9], SUS scores were computed, yielding an average of 87.08 (SD = 5.48). This significantly surpasses the average SUS score of 68, indicating excellent usability. Participants found our plug-in easy to use, with design features enhancing their GUI design process.

8.2.2 Ratings of Plug-in Features. Participants rated each plug-in feature on a scale of 1 to 7, with the model scoring 6.77 for preview (SD = 0.47), 5.83 for element prediction (SD = 0.69), and 6.17 for confidence level (SD = 0.69). Participants prefer features of prediction confidence and previews before actual element placement.

8.2.3 Ratings of Result GUIs. Participants provided an integer score out of 7 for each GUI design, with no significant differences between conditions in ratings for completed GUIs ($t = 0.396, p = 0.695$). The average rating with the plug-in was 6.28 (SD = 0.65), and without the plug-in was 6.17 (SD = 0.96). Participants reported having terminated their design process upon achieving satisfaction with the GUIs.

8.2.4 Timing. Without our plug-in, the average finishing time was approximately 5.6 minutes (SD = 1.02), while with the plug-in, times mostly ranged from 1 to 3 minutes (Mean = 2.22, SD = 0.78). The significant difference ($t = -11.71, p < .001$) indicates a 40% improvement in GUI design task completion time with our plug-in.

8.2.5 Summary. The high SUS score suggests ease of use and benefits for participants in their GUI design process. Our plug-in significantly improved efficiency, reducing task completion time by approximately 40%. Participants favored features such as element prediction preview, confidence level indicator, and element prediction. However, no significant difference in GUI quality was observed between plug-in usage and non-usage, as indicated by participant ratings of result GUIs.

8.3 Qualitative Findings

Alongside quantitative analysis, we performed qualitative analysis. Overall, participants gave positive feedback, especially on providing proper suggestions for GUI element prediction and omitting many manual design operations, with P2 mentioning that *"suggestions for element prediction are reasonable and have saved me some time on manual editing and aligning elements."*.

8.3.1 Workflow. Participants appreciated the integration of our method as a Figma plug-in. Since Figma is a popular design software, P3 concluded, *"It is very useful to have this kind of integration; I do not need to spend time learning a completely new tool to use these functions, and now I can simply use the software I normally use at work."* P5 held the same opinion, stating, *"This plug-in doesn't interrupt my design process; it's more like an add-on that helps with my design and provides inspiration. The operations are intuitive, so there is no need for us to learn how to use it specifically."* Some participants also praised the ease of element placement and thought the plug-in makes the design process more efficient; e.g., *"I often had to place the elements one by one, but now I can just click, and the plug-in directly suggests the proper placement. It is easy and less time-consuming."* (P6).

8.3.2 Functionality. Participants highlighted the usefulness of the preview window, with P5 noting its role in exploration and inspiration: *"It is interesting to see element prediction previews for each element in the preview window. I could hover over each element to get intuition as to how the GUI looks after placing it without the need to actually place it and undo it if I do not like the result. This can be used as an exploration process to help me compare different elements intuitively without additional effort. "* and P3 emphasized that the preview window gives designers *"a good way to visualize element suggestions."* In addition, P2 mentioned that the combination of preview and confidence level makes for better exploration: *"' I used the preview to compare the predictions among elements with high confidence – or medium if no element with high confidence exists – to decide which one I preferred to place first. I do not need to think much about which element I want to place since the confidence level helped*

Model	Overall	Profile	Menu	Login	Settings	Tutorial	Form	Gallery	List
ResNet50	28.35	0.00	16.10	0.00	81.65	0.00	0.00	0.00	47.62
Nearest Neighbors	30.62	65.96	46.19	43.33	14.75	65.57	68.00	19.01	7.44
Random Forest	82.39	58.51	95.34	70.00	80.22	54.10	69.33	86.97	91.07
Ours	**91.53**	**79.79**	**96.19**	**75.00**	**85.97**	**90.16**	**88.67**	**97.89**	**95.24**

Table 2: GUI topic classification results: Comparing the accuracy rates of our graph-based GUI representation method and the ResNet50, Nearest Neighbors, and Random Forest models for GUI topics.

me narrow down the options. " Additionally, all the participants appreciated the convenience of automatically predicting alignments and sizes; for instance, P4 stated, *"It works particularly well when the to-be-placed element needs to align with some existing elements or have the same size as them."*

8.3.3 Limitations. While acknowledging the advantages, participants identified some limitations. Specifically, P2, P3, and P5 criticized the method's accuracy when the unplaced element does not need to align or group with any existing GUI element. Furthermore, P3 and P4 pointed out that the method provides only one suggestion per element, limiting exploration possibilities.

9 OTHER APPLICATIONS

In addition to GUI autocompletion, we further explored other applications using our graph-based GUI representation.

9.1 GUI Topic Classification

GUI topic classification involves categorizing GUIs based on their topics and usage. For instance, "Gallery" GUIs exhibit a grid-like layout with images, while "Profile" GUIs display information related to user profiles or products. Our approach utilized GUI representation for classification, employing eight GUI topics derived from the Enrico dataset [43]. We sampled 10,000 GUIs, comprising both complete and partial instances from the Enrico GUIs associated with these eight topics, with a maximum of 2,000 instances for each GUI topic. The dataset was split into 85% for training and 15% for testing. The graph representation of each GUI was fed into a Graph Neural Network (GNN) to obtain the graph embedding, following the same process used in the autocompletion task (refer to Section 5). The classification process involved three fully connected layers and a softmax function, resulting in an accuracy rate of 91.53%, higher than other baselines. A comparison with the ResNet50, Nearest Neighbors, and Random Forest models is presented in Table 2.

9.2 GUI Retrieval

GUI retrieval is the process of finding the most similar GUI to a given one. Utilizing the graph embedding from our trained GUI topic classification model, we applied the nearest neighbor approach to identifying the closest GUIs. Samples demonstrating the performance of our model and the Screen2Vec model [47] in retrieving both complete and partial GUIs are shown in Figure 8 (More results can be seen in the supplementary materials).

9.2.1 User Study. A comparison study was conducted to assess our model against Screen2Vec.

Participants. Fifteen participants (9 female, 6 male) were recruited through social media promotion. All participants had normal vision or vision corrected to normal with glasses. None were colorblind. Local regulations do not require formal ethics review.

Experimental Design. From a pool of 1,500 randomly sampled partial and complete GUIs, we randomly selected 100 GUIs for each participant and presented the results retrieved by our method and the Screen2Vec method.

Apparatus. Pairs of GUI images, one predicted by our method and one by Screen2Vec, were displayed side by side on a custom webpage in randomized order.

Procedure. Participants began with a demographics questionnaire, followed by evaluating GUI images and selecting their preferences based on personal assessment criteria. Each participant could assess up to 100 pairs and could stop comparisons at any point.

Findings. We obtained 1,144 responses from 15 participants. Preferences were as follows: 37 for Screen2Vec (3.23%), 426 for our model (37.24%), and 681 for images perceived as equally good (59.53%). The difference between our method and Screen2Vec was statistically significant ($\chi^2 = 827.5, p < .001$). This indicates that our model retrieved more visually similar GUIs compared to Screen2Vec.

10 DISCUSSION AND CONCLUSION

This paper addressed the challenges of representing GUIs through a graph-based deep learning model. Prior deep learning-based GUI representations failed to consider the constraints for GUI elements and the visual-spatial-semantic structure of a GUI, which are important in computational design. Although many modern GUI tools use constraints to optimize GUIs, training a model to predict constraints remains a challenge. Our proposed novel graph-based GUI representation captures both the properties of GUI elements, such as textual content, visual appearance, and element types, and their relationships in the visual, spatial, and semantic dimensions of a GUI. It can be computed efficiently in computational design. We further trained graph neural networks (GNNs) to take the graph as input to optimize the GUI. We will release our code and data.

Our work has achieved the following results in the GUI autocompletion task.

(1) Our method predicts the position, size, and alignment of GUI elements more accurately. As shown in Figure 5, it achieves less than half of the error values in these three metrics (position, size, and alignment) compared to GRIDS [13], an approach for autocompletion using integer programming. When the number of existing elements on the GUI increases,

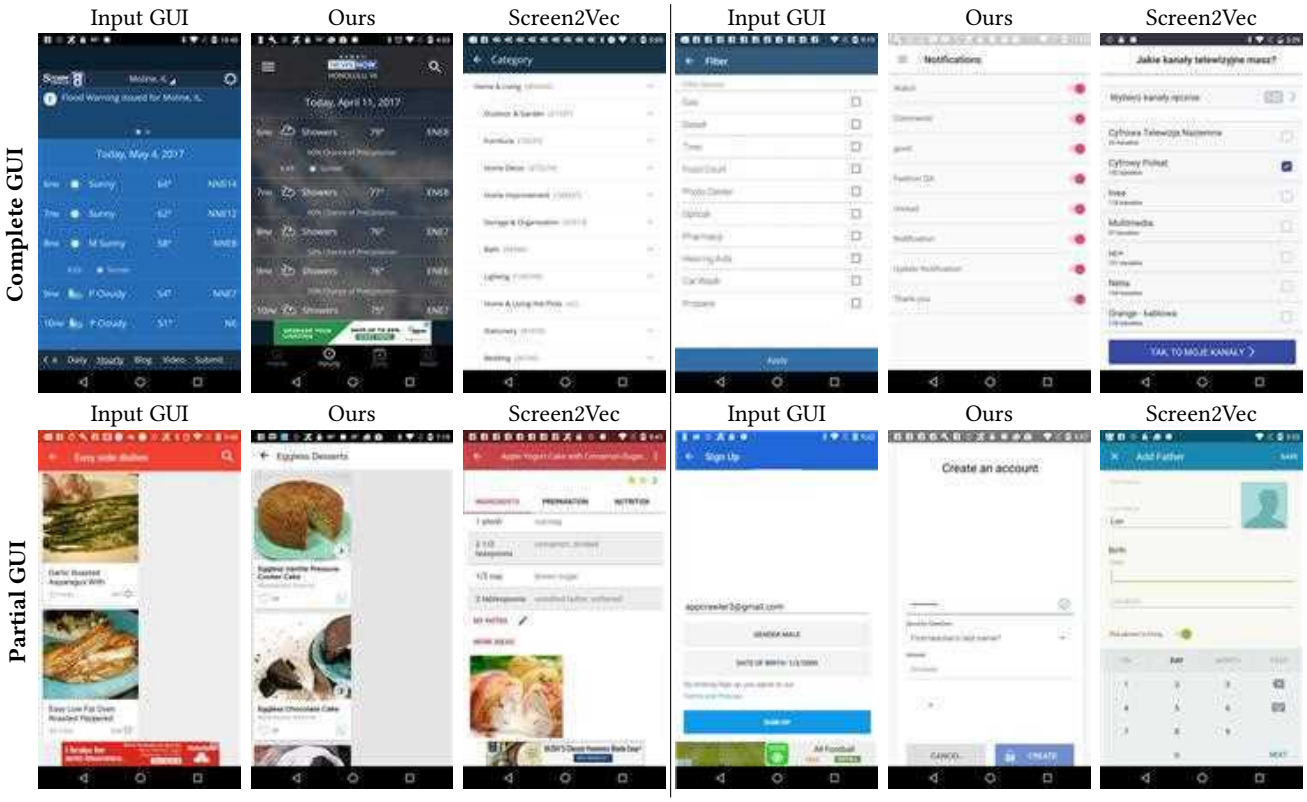

Figure 8: GUI retrieval results of our model and Screen2Vec, including both complete GUIs and partial GUIs.

it remains to have low error rates while GRIDS's errors dramatically increase.

(2) Our model offers superior alignment and visual appeal compared to the baseline method, and is better aligned with participants' preferences. In our comparison study, 70.33% of the responses preferred results from our model compared to 13.54% for GRIDS.

(3) Our method enhances flexibility by integrating as a plug-in within a popular existing design tool, Figma. This integration allows designers to apply workflows they are already familiar with, eliminating the need to learn new tools or switch between different design software tools. Participants in the designer study praised the plug-in for accelerating their design process without disrupting the existing functionalities of their design applications.

In addition to the demonstrated capability of our graph-based GUI representation in the GUI autocompletion task, we show that our GUI representation can be applied to other applications, such as GUI topic classification and GUI retrieval. Our model demonstrated superior accuracy in GUI topic classification compared to baseline methods like ResNet50, Nearest Neighbors, and Random Forest. Furthermore, user feedback highlighted our model's effectiveness in retrieving visually similar GUIs compared to the Screen2Vec model. Compared to other data-driven approaches, our graph-based representation facilitates the understanding of GUI structure, improving the explainability of the model. This capacity enables our representation to potentially extend to diverse downstream tasks. For example, accessibility needs can also be represented as constraints [22], and our method can train and predict layout constraints, thus it could potentially enhance accessibility.

10.1 Limitations and Future Work

As pointed out by participants in our designer study, our method has limited ability to generate accurate predictions if the unplaced element does not need to align or group with any existing element on the GUI. We currently assign a low confidence level to it to avoid uncertain predictions. Future work can improve the prediction of underconstrained GUI elements by considering more design priors or including more complicated constraints. As shown in Table 1, our representation does not explicitly represent the view hierarchy. The view hierarchy provides structural data, aiding models in understanding the layout and relationships of elements. We do not currently represent view hierarchies since they are not always available and often contain errors with incorrect structure information. However, future work can connect related element nodes in the graph representation to represent the view hierarchy. Moreover, while our method offers suggestions for each element to be placed, it provides only a single suggestion per element, thus constraining the possibility of exploration. In addition, we focus on a setting where all the elements are rectangular in shape or in rectangular bounding boxes. There are no datasets available with non-rectangular bounding boxes. To accommodate various shapes

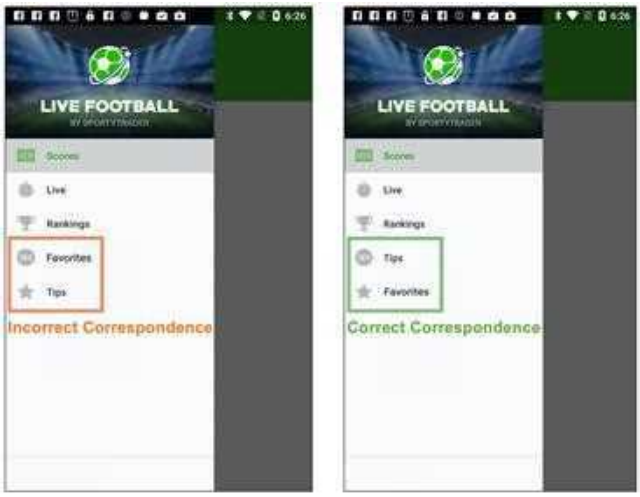

Figure 9: Limitation of our method: It cannot capture the semantic correspondence between different types of GUI elements, like associating the "Favorite" text with a "star" icon, which could be explored further in future research.

of bounding boxes, we can augment the element node with additional parameters. These parameters would facilitate the description of common shapes, such as rectangles with rounded corners and circles. Subsequently, the model can be retrained to incorporate this information when present in the training dataset. Furthermore, we observe that even for element prediction with high confidence levels, sometimes it does not predict ideal results. For example, as illustrated in Figure 9, our method cannot capture the semantic correspondence between different types of GUI elements, e.g., it cannot detect that the "Favorite" text should correspond to the "star" icon. Future research could explore more about GUI element correspondence and constraints across UI types.

ACKNOWLEDGMENTS

We appreciate the active discussion with Zixin Guo and Haishan Wang. This work was supported by Aalto University's Department of Information and Communications Engineering, the Research Council of Finland (flagship program: Finnish Center for Artificial Intelligence, FCAI, grants 328400, 345604, 341763; Subjective Functions, grant 357578), and the Meta PhD Fellowship.

REFERENCES

[1] Gary Ang and Ee-Peng Lim. 2022. Learning and Understanding User Interface Semantics from Heterogeneous Networks with Multimodal and Positional Attributes. *ACM Trans. Interact. Intell. Syst.* (Dec 2022). https://doi.org/10.1145/3578522

[2] Gary Ang and Ee-Peng Lim. 2022. Learning and Understanding User Interface Semantics from Heterogeneous Networks with Multimodal and Positional Attributes. *ACM Transactions on Interactive Intelligent Systems* (2022).

[3] Greg J. Badros, Alan Borning, and Peter J. Stuckey. 2001. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Trans. Comput.-Hum. Interact.* 8, 4 (Dec. 2001), 267–306. https://doi.org/10.1145/504704.504705

[4] Pavol Bielik, Marc Fischer, and Martin Vechev. 2018. Robust Relational Layout Synthesis from Examples for Android. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 156 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276526

[5] Thomas Bill, Bertil Lundell, John Alan McDonald, and Michael Sannella. 1992. *Bricklayer: Window Layout Using Linear Programming.* Technical Report. University of Washington, New York, NY, USA.

[6] Alan Borning and Robert Duisberg. 1986. Constraint-Based Tools for Building User Interfaces. *ACM Trans. Graph.* 5, 4 (oct 1986), 345–374. https://doi.org/10.1145/27623.29354

[7] Alan Borning, Richard Kuang-Hsu Lin, and Kim Marriott. 2000. Constraint-based document layout for the Web. *Multimedia systems* 8, 3 (2000), 177–189.

[8] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. 1997. Solving Linear Arithmetic Constraints for User Interface Applications. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology.* ACM, Banff, Alberta, Canada, 87–96. https://doi.org/10.1145/263407.263518

[9] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[10] Lukas Brückner, Luis A Leiva, and Antti Oulasvirta. 2022. Learning GUI Completions with User-defined Constraints. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 12, 1 (2022), 1–40.

[11] Duncan P. Brumby and Susan Zhuang. 2015. Visual Grouping in Menu Interfaces. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (2015). https://api.semanticscholar.org/CorpusID:14994819

[12] Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. 2021. VINS: Visual Search for Mobile User Interface Design. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21).* Association for Computing Machinery, New York, NY, USA, Article 423, 14 pages. https://doi.org/10.1145/3411764.3445762

[13] Niraj Ramesh Dayama, Kashyap Todi, Taru Saarelainen, and Antti Oulasvirta. 2020. *GRIDS: Interactive Layout Design with Integer Programming.* Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376553

[14] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology.* 845–854.

[15] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16).* Association for Computing Machinery, New York, NY, USA, 767–776. https://doi.org/10.1145/2984511.2984581

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] James Fogarty and Scott E. Hudson. 2003. GADGET: a Toolkit for Optimization-Based Approaches to Interface and Display Generation. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology* (Vancouver, Canada) *(UIST '03).* ACM, 125–134. https://doi.org/10.1145/1186562.1015789

[18] Krzysztof Gajos and Daniel Weld. 2005. Preference Elicitation for Interface Optimization. *UIST: Proceedings of the Annual ACM Symposium on User Interface Softaware and Technology*, 173–182. https://doi.org/10.1145/1095034.1095063

[19] Krzysztof Gajos, Anthony Wu, and Daniel S Weld. 2005. Cross-device consistency in automatically generated user interfaces. In *Proceedings of the 2nd Workshop on Multi-User and Ubiquitous User Interfaces.* 7–8.

[20] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2008. Decision-Theoretic User Interface Generation. In *AAAI'08.* AAAI Press, 1532–1536.

[21] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically Generating Personalized User Interfaces With Supple, In Proceedings of the 9th International Conference on Intelligent User Interfaces. *Artif. Intell* 174, 12-13, 910–950. https://doi.org/10.1016/j.artint.2010.05.005

[22] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2008. Improving the Performance of Motor-Impaired Users With Automatically-Generated, Ability-Based Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy) *(CHI '08).* ACM, 1257–1266. https://doi.org/10.1145/1357054.1357250

[23] Wilbert O Galitz. 2007. *The essential guide to user interface design: an introduction to GUI design principles and techniques.* John Wiley & Sons.

[24] Vikas Garg, Stefanie Jegelka, and Tommi Jaakkola. 2020. Generalization and Representational Limits of Graph Neural Networks. In *Proceedings of the 37th International Conference on Machine Learning (ICML).* PMLR, 3419–3430. https://proceedings.mlr.press/v119/garg20c.html

[25] M. Gori, G. Monfardini, and F. Scarselli. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Vol. 2. 729–734 vol. 2. https://doi.org/10.1109/IJCNN.2005.1555942

[26] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[28] Hiroshi Hosobe. 2000. A Scalable Linear Constraint Solver for User Interface Construction. In *Proceedings of the 6th International Conference on Principles and*

Practice of Constraint Programming (CP '02). Springer-Verlag, Berlin, Heidelberg, 218–232. https://doi.org/10.1007/3-540-45349-_17

[29] Hiroshi Hosobe. 2005. Solving Linear and One-Way Constraints for Web Document Layout. In *Proceedings of the 2005 ACM Symposium on Applied Computing* (Santa Fe, New Mexico) *(SAC '05)*. Association for Computing Machinery, New York, NY, USA, 1252–1253. https://doi.org/10.1145/1066677.1066959

[30] Nathan Hurst, Kim Marriott, and Peter Moulder. 2003. Cobweb: A Constraint-Based WEB Browser. In *Proceedings of the 26th Australasian Computer Science Conference - Volume 16* (Adelaide, Australia) *(ACSC '03)*. Australian Computer Society, Inc., AUS, 247–254.

[31] Charles Jacobs, Wilmot Li, Evan Schrier, David Bargeron, and David Salesin. 2003. Adaptive Grid-Based Document Layout. In *ACM SIGGRAPH 2003 Papers* (San Diego, California) *(SIGGRAPH '03)*. ACM, 838–847. https://doi.org/10.1145/1201775.882353

[32] Yue Jiang. 2024. Computational Representations for Graphical User Interfaces. In *Extended Abstracts of the 2024 CHI Conference on Human Factors in Computing Systems (CHI EA '24)*.

[33] Yue Jiang, Ruofei Du, Christof Lutteroth, and Wolfgang Stuerzlinger. 2019. ORC Layout: Adaptive GUI Layout with OR-Constraints. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, Article 413, 12 pages. https://doi.org/10.1145/3290605.3300643

[34] Yue Jiang, Yuwen Lu, Clara Kliman-Silver, Christof Lutteroth, Toby Jia-Jun Li, Jeffrey Nichols, and Wolfgang Stuerzlinger. 2024. Computational Methodologies for Understanding, Automating, and Evaluating User Interfaces. In *Extended Abstracts of the 2024 CHI Conference on Human Factors in Computing Systems (CHI EA '24)*.

[35] Yue Jiang, Yuwen Lu, Christof Lutteroth, Toby Jia-Jun Li, Jeffrey Nichols, and Wolfgang Stuerzlinger. 2023. The Future of Computational Approaches for Understanding and Adapting User Interfaces. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, Article 367, 5 pages. https://doi.org/10.1145/3544549.3573805

[36] Yue Jiang, Yuwen Lu, Jeffrey Nichols, Wolfgang Stuerzlinger, Chun Yu, Christof Lutteroth, Yang Li, Ranjitha Kumar, and Toby Jia-Jun Li. 2022. Computational Approaches for Understanding, Generating, and Adapting User Interfaces. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 74, 6 pages. https://doi.org/10.1145/3491101.3504030

[37] Yue Jiang, Eldon Schoop, Amanda Swearngin, and Jeffrey Nichols. 2023. ILuvUI: Instruction-tuned LangUage-Vision modeling of UIs from Machine Conversations. *arXiv preprint arXiv:2310.04869* (2023).

[38] Yue Jiang, Wolfgang Stuerzlinger, and Christof Lutteroth. 2021. ReverseORC: Reverse Engineering of Resizable User Interface Layouts with OR-Constraints. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 316, 18 pages. https://doi.org/10.1145/3411764.3445043

[39] Yue Jiang, Wolfgang Stuerzlinger, Matthias Zwicker, and Christof Lutteroth. 2020. ORCSolver: An Efficient Solver for Adaptive GUI Layout with OR-Constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376610

[40] Solange Karsenty, James A. Landay, and Chris Weikart. 1993. Inferring Graphical Constraints With Rockit. In *Proceedings of the Conference on People and Computers VII* (York, United Kingdom) *(HCI'92)*. Cambridge University Press, 137–153. https://doi.org/10.1007/3-540-58601-_91

[41] Markku Laine, Yu Zhang, Simo Santala, Jussi P. P. Jokinen, and Antti Oulasvirta. 2021. Responsive and Personalized Web Layouts with Integer Programming. *Proc. ACM Hum.-Comput. Interact.* 5, EICS, Article 213 (May 2021), 23 pages. https://doi.org/10.1145/3461735

[42] Hsin-Ying Lee, Lu Jiang, Irfan Essa, Phuong B Le, Haifeng Gong, Ming-Hsuan Yang, and Weilong Yang. 2019. Neural Design Network: Graphic Layout Generation with Constraints. *arXiv e-prints* (2019), arXiv–1912.

[43] Luis A Leiva, Asutosh Hota, and Antti Oulasvirta. 2020. Enrico: A dataset for topic modeling of mobile UI designs. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*. 1–4.

[44] Gang Li, Gilles Baechler, Manuel Tragut, and Yang Li. 2022. Learning to denoise raw mobile UI layouts for improving datasets at scale. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–13.

[45] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6038–6049. https://doi.org/10.1145/3025453.3025483

[46] Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M. Mitchell, and Brad A. Myers. 2020. Multi-Modal Repairs of Conversational Breakdowns in Task-Oriented Dialogs. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 1094–1107. https://doi.org/10.1145/3379337.3415820

[47] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. 2021. Screen2Vec: Semantic Embedding of GUI Screens and GUI Components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 578, 15 pages. https://doi.org/10.1145/3411764.3445049

[48] Yang Li, Julien Amelot, Xin Zhou, Samy Bengio, and Si Si. 2020. Auto completion of user interface layout design using transformer-based tree decoders. *arXiv preprint arXiv:2001.05308* (2020).

[49] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping Natural Language Instructions to Mobile UI Action Sequences. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 8198–8210. https://doi.org/10.18653/v1/2020.acl-main.729

[50] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 569–579. https://doi.org/10.1145/3242587.3242650

[51] Christof Lutteroth, Robert Strandh, and Gerald Weber. 2008. Domain specific high-level constraints for user interface layout. *Constraints* 13, 3 (2008), 307–342.

[52] Ethan Marcotte. 2011. *Responsive Web Design*. A book apart.

[53] Aliaksei Miniukovich and Antonella De Angeli. 2015. Computation of interface aesthetics. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 1163–1172.

[54] Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (March 2000), 3–28. https://doi.org/10.1145/344949.344959

[55] Brad A. Myers. 1990. Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints. *ACM Trans. Program. Lang. Syst.* 12, 2 (apr 1990), 143–177. https://doi.org/10.1145/78942.78943

[56] Brad A. Myers. 1995. User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 2, 1 (March 1995), 64–103. https://doi.org/10.1145/200968.200971

[57] Brad A. Myers and William Buxton. 1986. Creating Highly-Interactive and Graphical User Interfaces by Demonstration. *SIGGRAPH Comput. Graph* 20, 4 (1986), 249–258. https://doi.org/10.1145/15922.15914

[58] Brad A. Myers, Richard G. Mcdaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Ieee Computer Society, Ieee Computer Society, Andrew Mickish, Alex Klimovitski, and Patrick Doane. 1997. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering* 23 (1997), 347–365.

[59] Panupong Pasupat, Tian-Shun Jiang, Evan Liu, Kelvin Guu, and Percy Liang. 2018. Mapping natural language commands to web elements. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 4970–4976. https://doi.org/10.18653/v1/D18-1540

[60] Erica Sadun. 2013. *iOS Auto Layout Demystified*. Addison-Wesley Professional, Boston, US.

[61] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. Insights Into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (London, United Kingdom) *(EICS '13)*. ACM, Gothenburg, Sweden, 275–284. https://doi.org/10.1145/3197231.3197249

[62] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[63] Adriano Scoditti and Wolfgang Stuerzlinger. 2009. A New Layout Method for Graphical User Interfaces. In *Science and Technology for Humanity (TIC-STH), 2009 IEEE Toronto International Conference*. IEEE, 642–647. https://doi.org/10.1016/j.infsof.2015.10.005

[64] Nishant Sinha and Rezwana Karim. 2015. Responsive Designs in a Snap. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 544–554. https://doi.org/10.1145/2786805.2786808

[65] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J. Ko. 2020. *Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376593

[66] Pedro Szekely and Brad Myers. 1988. A User Interface Toolkit Based on Graphical Objects and Constraints. *SIGPLAN Not.* 23, 11 (jan 1988), 36–45. https://doi.org/10.1145/62084.62088

[67] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning. In *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology (UIST)*, (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 498–510. https://doi.org/10.1145/3472749.3474765

[68] Gerald Weber. 2010. A Reduction of Grid-Bag Layout to Auckland Layout. In *Proceedings of the 2010 21st Australian Software Engineering Conference (ASWEC '10)*. IEEE Computer Society, 67–74. https://doi.org/10.1109/ASWEC.2010.38

[69] Daniel S. Weld, Corin Anderson, Pedro Domingos, Oren Etzioni, Krzysztof Gajos, Tessa Lau, and Steve Wolfman. 2003. Automatically Personalizing User Interfaces. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence* (Acapulco, Mexico) *(IJCAI'03)*. Morgan Kaufmann Publishers Inc., 1613–1619. http://dl.acm.org/citation.cfm?id=1630659.1630944

[70] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. UIED: a hybrid tool for GUI element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1655–1659.

[71] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=ryGs6iA5Km

[72] Brad Vander Zanden and Brad A. Myers. 1990. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) *(CHI '90)*. ACM, Seattle, Washington, USA, 27–34. https://doi.org/10.1007/978-3-319-67744-_2

[73] Brad Vander Zanden and Brad A. Myers. 1990. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) *(CHI '90)*. ACM, 27–34. https://doi.org/10.1007/978-3-319-67744-_2

[74] Brad Vander Zanden and Brad A Myers. 1991. The Lapidary Graphical Interface Design Tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 465–466. https://doi.org/10.1145/108844.109005

[75] Clemens Zeidler, Christof Lutteroth, Gerald Weber, and Wolfgang Stürzlinger. 2012. The Auckland Layout Editor: An Improved GUI Layout Specification Process. In *Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction* (Dunedin, New Zealand) *(CHINZ '12)*. Association for Computing Machinery, New York, NY, USA, 103. https://doi.org/10.1145/2379256.2379287

[76] C. Zeidler, G. Weber, A. Gavryushkin, and Christof Lutteroth. 2017. Tiling algebra for constraint-based layout editing. *J. Log. Algebraic Methods Program.* 89 (2017), 67–94.

[77] Xinru Zheng, Xiaotian Qiao, Ying Cao, and Rynson W. H. Lau. 2019. Content-Aware Generative Modeling of Graphic Design Layouts. *ACM Trans. Graph.* 38, 4, Article 133 (July 2019), 15 pages. https://doi.org/10.1145/3306346.3322971