Assessing the User Interaction Cost of Keyboard Navigation in Web Applications

Christina Z. Chaniotaki
Department of Computer Science
University of Southern California
Los Angeles, USA
cchaniot@usc.edu

Paul T. Chiou
Department of Computer Science
University of Southern California
Los Angeles, USA
paulchio@usc.edu

William G.J. Halfond
Department of Computer Science
University of Southern California
Los Angeles, USA
halfond@usc.edu

***Abstract*—Keyboard usage is an important aspect of accessing the web, as it is an interface supported by a wide range of accessible technologies, providing flexibility for those who cannot use point-and-click interfaces. However, navigating the web with a keyboard is not always easy. In this paper, we present a quantitative look at the disparate experiences that keyboard users face when interacting with web applications (apps) compared to what is perceived as the "standard" conventional mouse-based interaction. Specifically, we assess the effort required to navigate web apps using the keyboard and examine how the web User Interface (UI) behaves differently for keyboard and mouse users. We implemented and ran a keyboard-based UI crawler on a large set of web UIs, and analyzed the keyboard navigation patterns from the perspective of keyboard users. Our findings highlight significant disparities, revealing common obstacles that keyboard users face when performing basic tasks on web apps.**

***Index Terms*—Keyboard Navigation, User Experience, Empirical Study, Keyboard Accessibility, Web Accessibility, Interaction Cost.**

I. Introduction

The web is an integral part of our life, enabling communication, entertainment, socializing, studying, and work through web applications (apps). Ensuring accessibility to web apps is essential for everyone, including individuals with disabilities. Prioritizing web accessibility not only helps companies meet legal and ethical requirements but also enhances user experience and inclusivity. Standards such as the Americans with Disabilities Act (ADA) and laws around the world provide guidelines to support accessibility efforts [1]–[5]. However, for the 16% of the global population with disabilities, navigating the web can still present significant challenges [6].

Web users with limited vision or motor abilities who cannot use point-and-click or touch devices rely on alternative navigation methods. The World Wide Web Consortium (W3C) identifies the keyboard as the most universally supported input method for people with disabilities [7]. When content can be navigated using a keyboard, it becomes accessible via Assistive Technologies (ATs) such as speech input, sip-and-puff switches, voice controls, and scanning software, all simulating keystrokes [8]–[10]. No other input method offers this flexibility, making keyboard access to web User Interfaces (UIs) a critical aspect of accessibility.

Despite the importance of keyboard usage [11], the web remains largely unfriendly to keyboard users. Many web apps contain accessibility barriers; up to 48% of popular sites lack keyboard access, often causing frustration or fatigue for users with disabilities [10], [12], [13]. For example, interactive elements that can be accessed with a mouse may not be reachable via the keyboard. Even in the cases where the interactive elements are reachable, simply entering a search term and navigating to the first result can take tens of keystrokes [14]. These keyboard barriers hinder the usability of the web and negatively affect the web navigation experience of those who rely on the keyboard interface or ATs.

Existing research on web keyboard usage primarily focuses on identifying various accessibility violations in web apps (e.g., [12], [15]–[19]). While this research helps ensure compliance with accessibility standards, no study has measured the impact of keyboard navigation on users in terms of effort and available UI interactions. Several studies have attempted to evaluate how users effectively interact with web apps through various interaction modalities such as snapshot-based interaction [20], facial expression, and emotion recognition [21], [22]. These studies, however, do not address keyboard-based modalities. Therefore, a gap remains in understanding the effort required to navigate and interact with a web app using a keyboard and the differences in UI behaviors compared to mouse interaction.

To better understand keyboard navigation in web apps, we conducted an extensive empirical study on popular websites to measure and analyze their keyboard navigation. Specifically, our study examined the effort required for keyboard navigation and interactions versus that of mouse interactions and the discrepancies in UI behaviors between these two modalities. Our results revealed stark differences: on average, keyboard users required seven times more time than point-and-click users needed to complete the same task. Additionally, over one in seven pages contained UI behaviors that were inaccessible or unavailable to keyboard users, including submitting forms, accessing drop-down menus, or triggering dynamic components. Based on these observations, our findings highlight design patterns and anti-patterns that can guide developers improve keyboard navigation in web apps.

In summary, this paper makes the following contributions:

- We conducted an extensive empirical study of 216 popular websites, quantifying the interaction costs and different behaviors of keyboard navigation compared to mouse.
- As part of the effort, we developed Krawler, a novel keyboard-based web crawling infrastructure that models user interactions via a graph-based Keyboard Experience Representation (KER), enabling large-scale, automated analysis of keyboard navigation across web apps.
- We identified design patterns and anti-patterns that affect interaction cost, offering actionable insights for improving keyboard accessibility in modern web interfaces.

II. Overview of the Study

The goal of our study is to investigate and understand the differences in users' experiences when interacting with a web app using a keyboard-based (KB) interface versus a point-and-click (PNC) interface. Fundamentally, the two modalities are very different. A PNC user (typically using a mouse) can simply click on any element in a web page to activate or interact with it. In contrast, KB users must press the Tab key[1] to move the keyboard focus forward (or Shift+Tab to move the keyboard focus backward) to reach an element they want to interact with (i.e., if a user is currently on element n and wants to interact with element $n+i$, then the user must press the Tab i times until focus reaches that element.) Once an element has focus, a KB user can use a standard set of keys: arrows ↑ ↓ ← → to navigate within groups of elements (e.g., menus, radio buttons); Space or Enter to activate elements; and Esc to exit prompts or dialogs [24]–[27]. Although these modalities use different interactions, they both aim to make the web equally accessible and usable. The keyboard interface plays a crucial role in ensuring accessibility, yet the interaction costs it imposes are less understood—an aspect that this study aims to address. For instance, while a website may allow navigation via the keyboard, the nature of the designed keyboard interface can make even simple tasks require significantly more effort. An example of this would be a news website where a mouse user can open an article with a single click, whereas a keyboard user must press Tab multiple times to navigate through menus, ads, and other links before reaching it. Such challenges highlight the gap between accessibility compliance and the accessibility interaction cost for KB users.

In this study we focus on two aspects of accessibility interaction cost: (1) the effort required to navigate and interact with a web app's UI, and (2) the number of UI inconsistencies that emerge—or even break—when using KB navigation instead of a traditional PNC approach. To address these two aspects of accessibility interaction cost, our study investigates the following three research questions (RQs):

[1]Findings from the Web Accessibility Survey show that 93.5% of KB users rely on the Tab key for navigation [23].

RQ1 What is the accessibility interaction cost for keyboard-based users to navigate and interact with a web page?
RQ2 What effort is required for keyboard-based users to complete typical use cases in web apps?
RQ3 What are the differences in UI behavior when interacting with a web page via the keyboard versus the mouse?

A. Selection of Subject Web Apps and Pages

Our study evaluates these RQs on a broad sample of popular web apps. To identify our sample, we started with the top 1,000 most popular websites from the Tranco list [28] (February 2nd, 2025). From this list, we removed: 11 subjects focused on adult content [29]; 303 subjects with expired SSL certificates; 66 subjects flagged as malicious in VirusTotal [30]; and 69 subjects that were locale-based variants (e.g., amazon.com, amazon.it), retaining only the highest rank variant. To ensure reproducibility of our results, we also excluded 59 subjects that could not be cached automatically (due to bot protection mechanisms such as Captchas). From the remaining 492 subjects, we randomly selected 216 subjects using Cochran's formula with 95% confidence and 5% margin of error [31]. The sample covers diverse web app categories: *News* (24), *Entertainment* (17), *Shopping and Travel* (22), *Social Media* (5), *Services* (72), *Business* (37), *Education and Science* (23), *Government* (11), *Healthcare* (3), and *Financial Platforms* (2). Table I shows ten representative websites from our sample. The table presents the website URL, its category, Tranco ranking, and the number of interactive elements (IE), where IE refers to any web page components—such as buttons, links, or form fields—that users can interact with to perform an action. The complete list of websites is available in our replication package [32].

For RQ1 and RQ3, we analyzed the initial page (homepage) of each subject, as it is the main entry point from which users navigate to key pages (e.g., login, ordering, etc.). This focus ensured consistency and feasibility across our subjects while still capturing a representative view of each subject's keyboard navigation accessibility. For RQ2, we conducted the analysis at the use case level (e.g., ordering a product), which may span multiple web pages starting from the homepage.

TABLE I
Ten Example Websites from Our Sample Dataset

Ranking	URL	Category	IE
13	https://www.instagram.com	Social Media	138
30	https://www.wikipedia.org	Education and Science	174
147	https://www.nist.gov	Government	56
151	https://www.paypal.com	Financial Platforms	77
265	https://booking.com	Shopping and Travel	224
278	https://www.nvidia.com	Business	352
438	https://www.autodesk.com	Services	62
482	https://www.duolingo.com	Entertainment	185
574	https://www.nbcnews.com	News	106
627	https://www.mayoclinic.org	Healthcare	333

B. Research Protocol Overview

At a high level, evaluating our study's RQs requires comparing user interactions between PNC and KB users. To do this for our subjects, we employed automated web crawling techniques to capture information about each of the subjects. For computing information from a PNC user's perspective, we conducted a comprehensive review of available web crawling solutions. These crawlers simulate mouse navigation by clicking or hovering over the interactive elements of a web page to capture the different UI states. During this process, we found that several tools were no longer maintained, accessible, or functioning properly [33]–[39], while others produced incompatible output that made comparison difficult [40]–[43]. We successfully customized two web crawlers, Crawljax [44], and Qualstate [45], to capture PNC user interactions. However, comparable techniques for KB crawling do not exist; thus, in Section III, we present Krawler, our KB crawling infrastructure for comparable data collection.

C. Experimental Testbed

Our experiments were browser-based, with tools automatically launching the respective browser on each Operating System (OS). We utilized Ubuntu 20.04 LTS, Windows 10 Pro 19045.5854, and macOS Sequoia 15.1 with Google Chrome 133.0.6943.54, Mozilla Firefox 0.35.0, and Microsoft Edge 132.0.2957.127. All experiments were systematically performed for all OS–browser combinations using Krawler, Qualstate, and Crawljax, with identical browser versions across systems to ensure comparability.

III. Keyboard-Based Web Crawling Architecture

Developing a crawler that can capture keyboard interaction data for a given web page is essential for our empirical study. A key challenge in computing this information is that interactive elements and their responses cannot be inferred from the source code due to modern web pages' dynamic nature. To address this, we designed Krawler to be a dynamic KB web crawler that systematically interacts with all elements in a web page and records the interaction results in a model that we can use later to compute the relevant keyboard interaction statistics.[2] In this section, we describe the model and formally define Krawler's crawling algorithm.

A. Modeling Keyboard Navigation

For a given Page Under Test (PUT), Krawler builds a graph model of KB navigation and interactions, analogous to how traditional crawlers capture this information for a PNC user. We call this model a Keyboard Experience Representation (KER). Figure 1 shows an example and its corresponding UIs.

We define the KER as a set G of UI states that the PUT can be in. A **UI state** is defined as *a unique set of visible interactive elements on the PUT,* representing different configurations of interactive elements as encountered through keyboard navigation. Each UI state $g \in G$ is represented as a directed graph $g = \langle V, E, v_0 \rangle$ where V is the set of nodes representing interactive elements in the UI state; E is the set of directed edges that define KB navigation flows between elements in V; and v_0 is the entry node, representing the element that initially receives keyboard focus in the UI state.

Edges of KER—The edge set E represents the keyboard navigation flows triggered by keyboard actions Φ (i.e., Tab, Shift + Tab, ↑, ↓, ←, →, Enter, Space, Esc), the standard keyboard actions defined by the W3C Authoring Practices [46]. A directed edge $e \in E$ is a tuple $\langle v_s, \phi, v_t \rangle$ which represents that pressing a key $\phi \in \Phi$ shifts focus from v_s to v_t with $v_s, v_t \in V$. Edges can be either inter-state or intra-state. *Inter-state edges* indicate actions that create a new UI state g', where the target node v_t is in g' and becomes the new state's entry node v_0. *Intra-state edges* occur when no new state is created as a result of the key press. If an action does not shift focus (e.g., pressing Space to select a box), a self-edge is created.

In Figure 1a, pressing Enter on the *Log In* link (g_0) opens the "Log In" UI (g_1). This is modeled in the KER (Figure 1b), where the *inter-state edge* transitions from state g_0 to g_1.

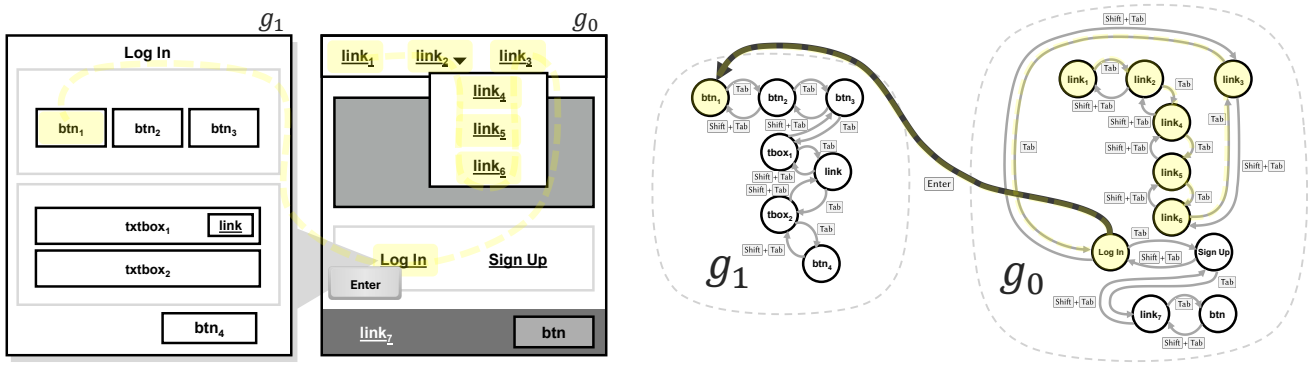
B. Building the Keyboard Experience Representation (KER)

To build the KER, Krawler dynamically explores the UI of a given PUT using Φ. The high-level intuition of this process is to iterate over all the interactive elements V in the PUT and execute all the keyboard operations Φ on each element to identify keyboard navigation. During this process, whenever a UI state changes—such as expanding or collapsing menus, switching tab panels, or triggering dialogs—Krawler continues modeling keyboard navigation within the new state. This process repeats until all possible UI states have been explored. Algorithm 1 shows the process of creating the KER for a PUT.

The algorithm begins with an empty set of UI states G (line 1) and uses a worklist to process newly identified UI states (line 2). The worklist contains a set of edges, each in the form $\langle v, \phi, v' \rangle$, representing the transition to a particular UI state. The worklist starts with the edge (line 5) that transitions to the initial UI state g_0 after loading the PUT. When an edge $\bar{e}$ is processed from the worklist, the approach first crawls to the new UI state g where $\bar{e}.v_t$ resides (line 9), then builds the node set $g.V$ (line 11) by rendering the PUT in the browser and analyzing its rendered Document Object Model (DOM) to identify its visible interactive HTML elements.

The algorithm then crawls the UI to explore all possible keyboard navigation paths by triggering every keyboard action $\phi \in \Phi$ on each interactive element (line 14). Each time ϕ is triggered, the approach observes the browser's focus shift and the resulting UI state, then creates a directed edge representing the transition (line 16). The approach then analyzes the resulting UI state by rendering the PUT in the browser and

[2]The source code of Krawler is at: https://github.com/USC-SQL/krawler



(a) Visualization of two UI states.

(b) KER of the two UI states.

Fig. 1. A simplified version of the KER (right) with its corresponding UI states (left).

examining its DOM to identify each visible interactive element (line 17). If the resulting UI state is new, the approach adds the newly created inter-state edge to the worklist (line 18). If no new UI state results from executing the action ϕ, an intra-state edge is created as part of the existing state's edge set (line 19). This construction process iteratively explores all possible UI states of the PUT and their keyboard navigation paths until the worklist is empty (i.e., the graph reaches a fixed point).

Algorithm 1 Building the KER Graph

Input 1: Page Under Test (*PUT*)
Input 2: Φ: Keyboard navigation buttons
Output: KER: Keyboard Experience Representation

1: Initialize $G \leftarrow \emptyset$
2: Initialize *worklist* $\leftarrow \emptyset$
3: $g_0 \leftarrow$ initial state of *PUT* after page load
4: $v_0 \leftarrow$ initial node in g_0 to receive focus after page load
5: add initial edge $\langle null, \epsilon, v_0 \rangle$ to *worklist*
6: Initialize $E_{visited} = \emptyset$
7: **while** *worklist* is not empty **do** dequeue edge $\bar{e}$ in the form $\langle v_s, \bar{\phi}, v_t \rangle$
8: **if** $\bar{e} \in E_{visited}$ continue
9: $g \leftarrow$ set state with $\bar{e}.v_t$ /*state currently in construction process*/
10: add g to G
11: **for all** $v \in g.V$ **do**
12: **for all** $\phi \in \Phi$ **do**
13: set v to focus
14: execute ϕ on v and save new UI state as g'_t
15: $v' \leftarrow$ node currently in focus in $g'_t.V$
16: create new edge e as $\langle v, \phi, v' \rangle$
17: **if** $g.V \neq g'_t.V$ **then**
18: add e to *worklist* /*inter-state edge for exploration*/
19: add e to $g.E$
20: add $\bar{e}$ to $E_{visited}$
21: **return** $\langle G, v_0 \rangle$

IV. RQ1: What is the accessibility interaction cost for keyboard-based users to navigate and interact with a web page?

The goal of RQ1 is to evaluate web accessibility for KB users by measuring (1) the keystrokes required for a full-page traversal and (2) the ability to interact with all elements. A full-page traversal enables users with visual impairments to discover all content and interactive elements available via the keyboard. Interaction with all elements measures the ability to activate or manipulate an element rather than navigate to it.

A. Protocol

To measure full-page navigation and interactivity, we executed Krawler on the homepage of each of the 216 subjects and analyzed the intra-state edges of the resulting KERs. For (1), we defined a **full-page iteration** as *the number of* intra-state edges labeled with Tab *or* Enter[3] that must be followed from the starting element (v_0) until returning to v_0. For (2), to **activate each interactive element** $\alpha \in V$, we measured the number of intra-state edges needed to reach and activate α from v_0. We then averaged this value across all interactive elements on the page. This metric has been used in related work for measuring UI task efficiency [47]. In cases where a *"skip content"* link was identified—features that enable keyboard users to bypass certain parts of the UI and jump directly to another section—we computed (1) and (2) twice: once without following its activation edge and once after following it, to provide a holistic view.

In this process, we encountered anomalous behavior that we marked as *navigation failures*. These cases occurred when Krawler remained stuck on the same interactive element for multiple keystrokes, could not reach an element after multiple iterations of all interactive elements, or there was unexpected behavior in which pressing Tab loaded another page. We excluded these web pages from (1) and (2) and marked them as navigation failures. However, if Krawler was able to complete a full-page cycle, even without accessing all the visible elements of the page, we included this iteration in our results and analysis.

B. Results

Figure 2 shows the Cumulative Distribution Function (CDF) of the average number of keystrokes required for a full-page

[3] Enter presses required to dismiss modal dialogs.

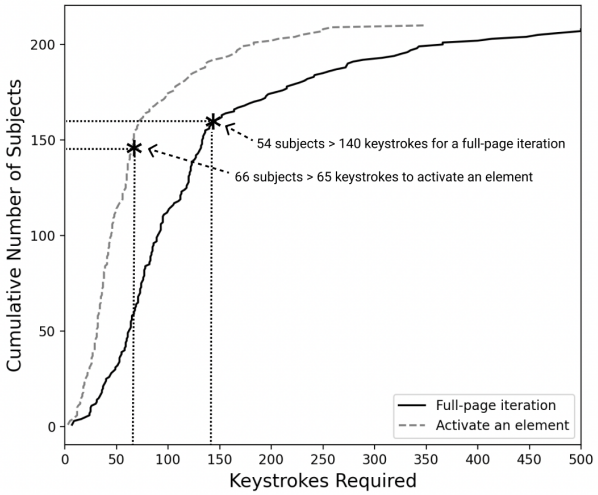


Fig. 2. Average cumulative keystrokes required across pages.

iteration and the activation of an interactive element. The x-axis represents the number of keystrokes required, while the y-axis shows the cumulative number of websites that required that number of keystrokes. During the evaluation, we identified six web pages with navigation failures across all browsers and five others with failures in a single browser. The numbers presented in Figure 2 only include those without navigation failures and are zoomed in to display a maximum of 500 keystrokes. The visualization excludes four websites that required more than 500 keystrokes to complete a full-page traversal, as they represent extreme outliers that would distort the visualization.

C. Discussion

1) Full page iteration: We found that the majority (73%) of subjects required under 140 keystrokes for a full-page iteration (corresponding to the asterisk in Figure 2), while 54 (25%) subjects required more. Additionally, 2% of the subjects, not shown in the graph, were inaccessible for a full-page iteration (i.e., navigation failures). High keystroke requirements are problematic for KB users with disabilities, as navigation speed is limited by an average of 1,500 ms per keystroke [48], [49], meaning that these 54 subjects could take 3.6 to 30 minutes to complete a traversal—making even basic interactions highly time-consuming. Looking deeper, we found subjects that contained lists of items with structured content—such as news articles with an image, title, and description—where all elements within each item linked to the same target. While this is user-friendly for PNC users, for KB users, when default keyboard navigation is used, they must Tab through each duplicate link within an item, which further increases effort. Unlike KB users, PNC users need a single click to reach the target, regardless of the number or structure of elements. In five extreme cases, keystrokes reached 683—peaking at 1,200—due to **infinite scrolling**. These subjects dynamically loaded new items as users navigated downward, preventing full-page iteration via Tab. While cached versions of the page allowed us to have a full-page iteration, real-world users would be unable to reach the footer as new content continuously loads. Instead, they must rely on Shift+Tab to move backwards, reach the top, navigate through the browser controls, and then enter the page via the footer, further increasing navigation effort.

We found that in 101 (47%) subjects, a KB user must perform **more keystrokes than the initial number of interactive elements** to complete a full-page iteration. Specifically, these subjects required a min/avg/max of 7/132/700 keystrokes for full navigation, while containing a min/avg/max of 5/107/516 visible interactive elements. Therefore, KB users needed additional min/avg/max of 2/25/184 keystrokes beyond the element count. This discrepancy arose due to the *dynamic UI behavior* of web apps, where pressing Tab can trigger actions that load new elements—either increasing effort for KB users or helping them navigate more efficiently. For instance, in 12 subjects, drop-downs expanded upon focus, and in 43 cases, carousels revealed additional interactive elements that were initially hidden. Consequently, KB users had to navigate through all elements—including those not immediately visible—before completing a full-page traversal. However, in 79 cases, skip content buttons were implemented, enabling KB users to bypass sections and reducing interaction effort by a min/avg/max of 2/19/85 keystrokes. Thus, the effort required for KB navigation is not strictly proportional to the number of visible interactive elements.

Regarding the extra UI elements during navigation, in 29 (13%) cases, **top navigation menus expanded automatically on keyboard focus**, forcing users to Tab through all sub-menus (and sometimes sub-sub-menus) with no way to prevent expansion or exit (e.g., Esc). Therefore, the Tab presses needed to reach the end of the page increased by a min/avg/max of 2/34/79. These menus deviate from standard design practices, which typically require ↓, Space, or Enter to trigger expansion. We manually examined these cases and found five web pages where the Tab key navigated inside the submenu, but the focus was invisible. As a result, users could not determine their position and had to perform more than 60 extra keystrokes to reach a visible focus indicator again.

2) Activate each interactive element: The distribution of subjects shows that most of the subjects (69%) required an average of 65 keystrokes (corresponding to the asterisk in Figure 2), with only 66 (31%) subjects requiring more keystrokes to activate an interactive element. For these 66 subjects, we found that the number of keystrokes required for keyboard users to activate an interactive element ranged from an avg/max of 133/600 per interactive element and a min/avg/max of 135/323/1,200 interactive elements per subject. Although an upward trend in keystroke count correlated with an increased number of interactive elements, this correlation held true for only about half of the subjects. Specifically, 76 subjects required **fewer keystrokes than the number of interactive elements** on the page. This lower interaction cost occurred

either because certain sections or elements were *ignored during keyboard navigation* (e.g., the *Conviva* subject ignored header drop-down menu actions, rendering them inaccessible via keyboard) or because interactive elements were *grouped together*, allowing users to navigate efficiently within groups using arrow keys (e.g., ←, →) before using Tab to exit. For instance, the *National Geographic* subject collapses article information to minimize redundant keyboard navigation on multiple links pointing to the same target. Similarly, the *RedHat* subject groups information into cards and utilizes additional keys (e.g., ↓, ↑) to access underlying components, significantly reducing the required keystrokes since many subcomponents are not accessed using Tab alone. Lastly, in the extreme cases surpassing 200 keystrokes, infinite scrolling, as outlined in Section IV-C1 was responsible, as dynamically loaded content notably increases navigation effort.

3) Inaccessible UI components causing navigation failures: For 11 subjects (5%) that we excluded from Figure 2 due to **navigation failures**, we manually analyzed their keyboard issues. We found five UIs where the keyboard cursor became stuck—a phenomenon known as a *keyboard trap*, either on a single element (three subjects) or cycling within a group of elements (two subjects). These traps blocked KB users from fully interacting with content. For instance, on *MySpace*, the first `<input>` in the top-right form created a single-element trap with no escape even using Shift+Tab. For cycling within a group of elements (*cycle traps*), keyboard focus remained stuck inside the group, preventing users from reaching other parts of the page unless they activated a navigation button within that group. Implementation-wise, these issues often stem from JavaScript listeners (e.g., `element.focus()`) that unintentionally redirect focus back to a specific element whenever the keyboard focus moves away.

4) Variations in full-page navigation effort across browsers: We observed that the **number of keystrokes required for a full-page iteration varied across different browsers**. This discrepancy occurred when navigating from the last interactive element on a page back to the first one. The number of Tab presses needed to cycle through browser UI elements differs depending on the browser. For example, in Chrome, returning to the first element via Tab requires only two extra presses, while in Edge, only one extra press is needed. However, in Firefox, this number can vary, often requiring at least eight additional Tab presses, depending on the number of installed plugins. This behavior occurs because, after reaching the last element of a web page, Firefox shifts focus to various browser UI components (e.g., search input, bookmark button, and plugins) before allowing navigation to return to the web page, increasing the effort required for keyboard users.

5) Summary of findings: Our analysis reveals significant accessibility challenges for KB users, increasing interaction costs across many websites due to dynamic UI behaviors, automatic menu expansions, and redundant interactive elements. Additionally, issues such as infinite scrolling, keyboard traps, and improperly hidden elements create barriers that can make sections of a web page completely inaccessible. Browser inconsistencies further impact navigation effort, as different tabbing behaviors alter the number of keystrokes required to traverse a page. Our findings highlight the importance of designing more seamless UI experiences by ensuring structured keyboard navigation, minimizing redundant interactions, and improving cross-browser consistency.

V. RQ2: What effort is required for keyboard-based users to complete typical use cases in web apps?

The goal of RQ2 is to evaluate the effort required for KB users to navigate and interact with web apps to complete various use cases. After quantifying the effort for simple KB operations in RQ1, we were also interested in evaluating more complex operations. These operations (i.e., use cases) involve extended UI interactions spanning multiple pages to complete actions like ordering products.

A. Protocol

To estimate the effort required for a user to complete various use cases (e.g., search and buy a product), starting from loading the web app, we measured the number of clicks for PNC users and the number of keystrokes for KB users needed to complete every step in each use case. This quantification allowed a direct comparison of navigation efforts. We omitted the number of steps to key in data (i.e., the number of keystrokes for typing input such as username and password) for both mouse and keyboard since they are the same effort, and we focused solely on the navigation/activation effort. **For PNC navigation**, we manually completed each use case and recorded the number of mouse clicks needed to complete the task. **For KB navigation**, we replicated the use case, activating the same interactive elements as in the PNC version by following the sequence of Tab-labeled intra-state edges in the KER graph until the next element was found and then activating it. Since these edges encode the web page's forward focus order, the navigation path was determined by sequential tabbing reflecting the standard navigation behavior available to KB users. When a skip link was available, we also quantified the alternative path obtained by activating it, in addition to the regular sequential-Tab path. We omitted shortcuts, as they are generally considered a feature-level concern rather than a core accessibility requirement [50]–[52].

Identification of use cases—Our process consisted of two steps: first, we defined general *use case categories*, and then we chose an *individual use case* for each website in our sample. Based on the website categories (listed in Section II-A) and drawing from relevant literature and industry reports on common web app tasks [53], [54], we identified 13 distinct categories of use cases: *Content Exploration*: e.g., browsing features; *Knowledge Discovery*: e.g., reading news, blog posts; *Explore and Compare*: e.g., product specs; *Explore and Watch*:

e.g., watching videos; *Search, Access, and Purchase*: e.g., ordering products; *Access and Connect*: e.g., contact forms; *Access and Download*: e.g., software, media; *Connect and Meet*: e.g., schedule appointments, meetings; *Start Your Journey*: e.g., get started, request information; *Sign Up and Start*: e.g., log-in, setup a new service; *Social Media Engagement*: e.g., chat, share, like, comment; *Create Mode*: e.g., design, writing; and *Play Online*: e.g., play online games. Then, for each website, we chose a use case representing its primary functionality based on its category. For example, Shopping and Travel sites were assigned either a Search, Access, and Purchase task or an Explore and Compare task.

B. Results

Figure 3 presents a comparison of the average number of mouse clicks versus keystrokes required to complete the use cases. The vertical axis represents the use case categories, with the number of completed use cases using KB navigation in each category shown in parentheses. The horizontal axis represents the average number of keystrokes and clicks required to complete each use case, plotted on a logarithmic scale. Note that this data does not include 30 use cases that could not be completed using the KB navigation. In those scenarios, accessibility issues manifested in the page's UI (as discussed earlier in Section IV-C), preventing us from completing the use case. This included all three use cases from the *Play Online* category, which is why this category is not shown.

C. Discussion

The results indicate that, on average, a PNC user needs 10 clicks to complete a use case, while a KB user needs 50 keystrokes to complete the same use case. To contextualize this effort, we converted the number of steps into a wall-clock time estimate, following the same approach as in Section IV-C. On average, a PNC user would need 11 seconds to complete a use case, given an average time of 1,100 ms per mouse click [55]. In contrast, a KB user would require 75 seconds, assuming an average keystroke time of 1,500 ms for people with disabilities [48], [49]. In an extreme case, the keyboard users required 491 keystrokes—a 3,992% increase in effort compared to those using a mouse. In this extreme case, each time the user navigated to a page, they had to Tab through 83 sub-menu items in the top navigation menus (the sub-menus items can be tabbed to despite the menu being closed, and the skip link aid was not working). For this use case, a keyboard user would need 12 minutes just to search for a book.

The difference in effort between mouse and keyboard navigation is not proportional to UI density; **KB navigation effort grows faster than mouse navigation in dense UIs**. For PNC users, interaction cost remains nearly constant, as they can directly click any item. For KB users, effort scales with the number of elements, since they must sequentially Tab through each one. Therefore, in many use cases involving dense UIs, the differences in navigation efforts were more pronounced.

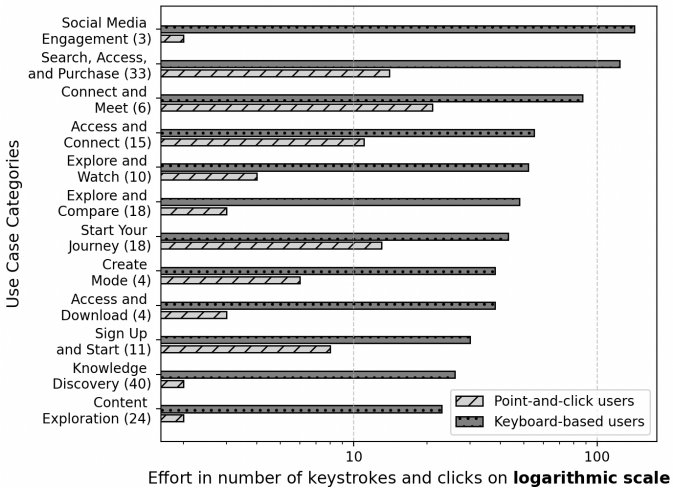


Fig. 3. Average number of clicks and keystrokes required to complete various use cases.

Eight of the use cases with the greatest disparity between PNC and KB navigation were among the top 20% of the densest UIs, averaging 615 elements per page. Infinite scrolling further increases effort. For example, on *Instagram*, KB users must sequentially navigate through all elements (e.g., stories, like, share, comment) to reach a post, requiring approximately 200 Tab presses to comment, whereas mouse users need only two clicks. By contrast, 29 use cases with sparse interfaces (averaging 140 elements) exhibited minimal disparities.

We found that **certain design strategies, such as skip mechanisms, significantly reduced the navigation effort required for keyboard users**. We observed that the effort required for a PNC user to complete a use case remained fairly consistent, whereas for a KB user, it varied significantly, with standard deviations of 9 clicks for PNC users and 26 keystrokes for KB users. Upon further investigation into the factors contributing to this difference, we discovered that in many keyboard use cases, there are effective strategies to noticeably reduce the overall effort required. We found that in cases where a skip content mechanism existed to help keyboard users bypass sections, the navigation time was reduced significantly. For example, in the *Amazon* marketplace subject, instead of navigating through all filter buttons and product listings to reach the main content, KB users could utilize the available skip buttons at the beginning of the navigation to jump directly to specific sections like orders, search, products, and the cart. This skip content mechanism, found in 79 (37%) of our subjects, reduced navigation steps by nearly 50% when completing a use case, compared to navigating without it.

We also found that **automated focus redirection reduced keyboard effort**. In 10 (5%) of use cases involving login or data entry—common on web forms—the navigation entry points were modified to set the initial keyboard focus on the `<input>` fields. This allowed users to start typing upon page load instead of navigating from the top. This resulted in a min/avg/max savings of 12/77/95% in effort,

respectively. Conversely, when focus redirection was missing, particularly during UI transitions, unnecessary keyboard effort was required. We observed this in 28 use cases. For instance, on *Autodesk*, adding a product to the cart triggered a right-side navigation drawer, but the focus was not automatically set within it. As a result, a KB user had to press Tab 93 times to reach the cart, traversing all other products and the rest of the page.

Overall, our results show that the keyboard's sequential nature of navigation impacted the effort in completing use cases. This effort occurs because a KB user must traverse multiple elements to reach the target, unlike a mouse, which provides direct access. This difference is even larger in subjects with high UI density, which led to greater KB navigation effort compared to mouse users. The effort can be reduced significantly through accessibility enhancements, such as skip mechanisms that allow users to bypass sections and automatic focus management for dynamically changing content. Our findings highlight the importance of incorporating such accessibility considerations into UI design to reduce interaction cost disparities between keyboard and mouse navigation.

VI. RQ3: What are the differences in UI behavior when interacting with a web page via the keyboard versus the mouse?

The goal of RQ3 is to evaluate whether the UI behavior differs when using a keyboard compared to mouse interaction. While we focused mainly on the effort to navigate to/and activate elements in RQ1 and RQ2, in this RQ, we are interested in what happens *after* activating the elements. The UI behavior provides insight into the effects of keyboard interaction and how KB and PNC users' experiences differ.

A. Protocol

To capture differences in UI behavior, we systematically explored each web app by activating every interactive element via mouse clicks and Enter keystrokes, recording the resulting UI state transitions, URLs, and DOM snapshots.

Modeling mouse-based interaction—For PNC exploration, we used two web crawlers, Crawljax [44], and Qualstate [45]. Both tools explore UI states—as defined in Section III-A—by simulating mouse `clicks` (and `hover` actions for Qualstate), and then capturing the resulting DOM and URL. Each tool required some basic configuration and modification to be used in our study. We configured Crawljax and Qualstate to remain on the initial page, avoid external links, and have an unlimited crawling depth to ensure that every interactive element on the web page was activated, similar to how Krawler operates. We kept all other tool configurations at their default settings. The complete configuration details and any tool modifications are provided in the replication package [32].

We ran Krawler, Crawljax, and Qualstate on our sample of 216 web pages to investigate whether UI behaviors vary across OSs and browsers. Using the experimental environment described in Section II-C, we evaluated each web page under all OS–browser combinations, resulting in nine configurations per page and a total of 1,944 executions (216×9). Qualstate was incompatible with Firefox due to its reliance on a Chromium-based implementation.

In web usage, whenever a user interacts with a web app—whether through the mouse or keyboard—the UI typically provides a visual cue to acknowledge the action. Building on the intuition that each UI state change represents a distinct outcome of an interaction, we use this notion as a metric for representing UI behaviors. Any divergence in the resulting UI states can therefore be interpreted as differences in the app's behavior (e.g., performing an action on an element that alters the UI in different ways). To capture such behavioral discrepancies between keyboard and mouse navigation, we adopt the concept of UI state coverage—a metric widely used in prior work to evaluate the exploration capabilities of web crawlers [34], [39], [44], [56]–[67]. We calculated each tool's UI state coverage as the ratio between the unique number of UI states identified by the tool (i.e., $|G|$) and the total number of distinct UI states (the union) among all the tools.

B. Results

Table II presents the average UI state coverage results for our 216 subjects, organized by OS (macOS, Windows, and Linux) and browser (Chrome, Edge, and Firefox) for each tool. Due to space constraints, we present only the average state coverage per configuration; full results are in our replication package [32]. Our results indicate that all three tools—Krawler, Crawljax, and Qualstate—were able to achieve similar amounts of UI experiences by achieving relatively similar state coverage. Crawljax maintains a consistent value of approximately 85% state coverage across all configurations, where Krawler and Qualstate report slightly higher averages, hovering near 91% and 90% respectively. Note that Qualstate was incompatible with Firefox (i.e., NA).

C. Discussion

A natural question is whether Krawler and Qualstate perform the *same* exploration, given their comparable coverage. We

TABLE II
AVERAGE UI STATE COVERAGE OF TOOLS ACROSS BROWSERS AND OSS IN OUR SAMPLE OF 216 SUBJECTS.

OS	Browser	Crawljax	Qualstate	Krawler
Linux	Chrome	86%	89%	92%
Linux	Edge	87%	89%	92%
Linux	Firefox	86%	NA	92%
macOS	Chrome	85%	91%	91%
macOS	Edge	85%	91%	91%
macOS	Firefox	85%	NA	91%
Windows	Chrome	86%	89%	91%
Windows	Edge	86%	90%	90%
Windows	Firefox	87%	NA	91%

found that 94 states (44%) are covered by both tools, while 122 states (56%) are unique to one or the other.

1) Similarities in UI behaviors: Of the UI states covered by both Krawler and Qualstate, 90% were triggered through interactions with standard HTML controls (e.g., `<a>`, `<button>`). These elements include built-in support for both keyboard and mouse navigation, which invoke the same event handling function. As a result, the same UI (e.g., a dialog) is produced regardless of input method. This observation suggests that using standard accessible HTML controls can help create similar experiences between keyboard and mouse.

2) Differences in UI behaviors: When looking at the differences in UI behaviors, we noticed they impacted keyboard users in varying ways, some with little to no effect, some prevented interaction entirely, and others increased the effort required to navigate and use the page.

a) Differences with negligible effect: In 10% of the different UI behaviors, the differences stem from the inherent distinctions between how keyboards and mouse interact with web interfaces, with little usage impact on KB users. For example, clicking on a search `<input>` field with a mouse places the cursor inside it for text entry, whereas pressing Enter often triggers an immediate search submission, navigating to the results page. This difference arises because keyboard users must first navigate to the field and type before activation, while a mouse click directly engages the element.

b) Differences impacting accessibility: Looking at the differences in coverage (UI states explored in one but not the other), we observed two causes: In the first cause, in 82 out of 216 subjects (38%), **Krawler reached more states than the PNC crawlers**. The reason is that click-based approaches do not trigger keyboard actions. Many subjects had additional UI states triggered only by KB interactions (e.g., `keydown`, `keyup`, or `focus` event handlers). For example, on the *Costco* subject, we found that while the mouse-based tools captured approximately 14 states, Krawler identified a total of 27. The extra states manifest when Krawler navigates through Tab in the header menu, triggering extra expansion buttons—hidden for PNC users—to help KB users activate and navigate the sub-menu items. Similar examples happen with the skip content buttons and other hidden buttons that only appeared during KB navigation, creating "keyboard-only" states.

In the second cause, in 40 (19%) cases, **Krawler detected fewer UI states than the mouse-based tools**. This occurred when elements lacked proper keyboard actionability (i.e., event handling)—typically custom `<div>` or `<span>` controls bound to `click` or `hover` events to respond to the mouse but not keyboard actions. These "mouse-only" states cannot be triggered through keyboard input. When examining these mouse-only states (i.e., states missing from Krawler), almost all of the non-actionable elements are also not reachable via Tab; in five cases, this was due to keyboard traps, as we discussed in Section IV-C, and in one case, the page redirected on the Tab press to an element. Consequently, all these accessibility barriers (whether non-actionable, non-reachable, or both) prevented Krawler from discovering those states.

c) Differences impacting ease of use: Subtle browser differences sometimes require more navigation effort from KB users. These **UI differences reflect variations in keyboard navigation across browsers when interacting with `<input>` elements**. For example, Chrome treats *radio-button groups* as a single block navigable with arrow keys, whereas Firefox allows tabbing to each button individually unless a default selection is present. Similarly, *date-related* `<input>` types differ: in Chrome, the date picker is activated by pressing Enter or Space on a designated button inside the input field, while Firefox's event handling requires users to press Enter or Space on a specific subfield (i.e., day, month, year) to open and modify the date picker. Mouse effort remains unchanged, but keyboard users take on average three steps in Chrome and six in Firefox to complete the same selection.

Discrepancies in keyboard interactions across OSs can cause unexpected behavior during keyboard navigation. **Native OS components affect how `<select>` drop-down elements behave**. On Windows and Linux, pressing ↓ on a focused drop-down immediately selects the next option, while macOS opens all options, letting users browse without confirming until Enter. These differences stem from browsers relying on each OS's native UI components to manage drop-down controls. In our subjects, most drop-downs triggered unintended navigation (e.g., switching languages) due to default `onChange` handlers for `<select>` elements. Consequently, KB users need to navigate a min/avg/max of 20/70/95 steps to return to the same drop-down element.

3) Summary of findings: Our analysis revealed that while Krawler, Crawljax, and Qualstate exhibit similar UI state coverage, the pathways and states they explore differ. Krawler uncovers additional states from keyboard-specific interactions (e.g., Tab triggering hidden elements), while mouse-based tools find states inaccessible via keyboard due to missing handlers. Environmental factors—including browser handling of radio buttons and input pop-ups, and OS-level differences in drop-downs—also introduce effort differences in keyboard navigation. These findings highlight both the inherent differences between keyboard and mouse interactions and the challenges keyboard users face from inconsistent implementations across platforms. Addressing these inconsistencies is critical to ensuring that keyboard users can access and interact with web apps as efficiently as mouse users.

VII. Design Patterns and Anti-Patterns

Based on our findings and the discussions in Sections IV to VI, we identified recurring *design patterns* as well as *anti-patterns*. These patterns provide actionable guidance for developers seeking to enhance the usability and inclusiveness of interactive systems. Our results show that well-implemented accessibility features can substantially reduce user effort,

whereas common design shortcomings often introduce friction, redundancy, or even complete inaccessibility.

Design Patterns: We identified three design patterns that support keyboard accessibility: skip content links, automatic focus shifts, and the use of standard HTML controls. **Skip content links** allow users to bypass components and reach main content quickly, reducing keystrokes (Sections IV-C and V-C). For example, *Taboola* provides skip links to access the main content and footer. **Automatic focus shifts** keep users oriented after dynamic updates, such as modals (Section V-C). For example, *Zendesk* moves focus to modal dialogs automatically, while *Autodesk* lacks focus shift, requiring extra keystrokes. Lastly, the **use of standard HTML controls** (i.e., `<button>`, `<input>`, `<a>`, and `<select>`) ensures built-in keyboard accessibility (Section VI-C). For instance, *Wikipedia* homepage use `<a>` for language links, and `<input>`, `<select>`, and `<button>` for the search bar, all focusable and actionable. These patterns help reduce effort and support efficient, inclusive keyboard navigation.

Design Anti-Patterns: We identified four anti-patterns that hinder keyboard accessibility: mismanaged keyboard events, grouped components with no escape, infinite scrolling, and non-standard custom components. **Mismanaged keyboard events** occur when custom handlers override native tabbing behavior, trapping focus within a set of elements or a single field (Section IV-C). For example, on *Myspace*, focus stuck in the search input, preventing normal navigation. **Grouped components with no escape** increase navigation effort by forcing users to tab through all sub-items (Section IV-C). For example, carousels on *National Geographic* trap focus, preventing users from leaving until all items are traversed. **Infinite scrolling without an escape mechanism** prevents users from reaching the page end, as new content continuously shifts the focus order (Sections IV-C and V-C). For instance, *Avito* dynamically loads listings, making it impossible to tab to the footer. Lastly, **non-standard custom components** (e.g., `<div>` or `<span>`) lack built-in keyboard support (Section VI-C). For example, on *Equativ*, icons implemented with `<div>` are not focusable, creating barriers for keyboard users. These anti-patterns disrupt keyboard flow, increase effort, and can make interfaces partially or fully inaccessible.

VIII. Threats to Validity

External Validity: Our study uses a random sample of the top 1,000 Tranco websites [28], so findings generalize only to these popular sites. However, they cover roughly 50% of Chrome page loads [68]. Since their popularity implies they are highly maintained, frequently updated, and generally strive to meet legal accessibility obligations; they can provide a reasonable lower bound on the accessibility challenges users may face in the broader population of websites.

Internal Validity: Running our tools on three OSs and browsers introduces potential variation in browser versions. To ensure consistency, we used identical versions on all platforms and standardized the window size to 1280×1024 with 100% zoom to prevent layout differences from affecting results.

Another threat is potential inaccuracy in manually verifying experiments, as counting clicks and keystrokes may introduce human error. To mitigate this, we recorded all interactions to allow recounting and validation. Because our analysis focused on differences between conditions rather than absolute values, small errors would not materially affect results. This approach ensured accurate, reproducible measurements and minimized the impact of human error.

Another threat is the presence of skip content buttons on some websites. Because Krawler assumes deterministic keyboard navigation—where each keystroke (e.g., Tab, Shift+Tab) moves focus predictably—ignoring these buttons would overestimate effort. To avoid this, Krawler checked for skip mechanisms at the start of navigation and used them when available to ensure accurate effort measurement.

A potential issue is that we did not run all three tools simultaneously, so website content may have changed due to updates or bot detection. To mitigate this, we captured and stored each site on February 4th, 2025, using mitmproxy, and used these archived versions for all tests. This ensured consistency by running every tool on identical page versions.

Construct Validity: Another potential threat is that Krawler does not use keyboard shortcuts (key combinations) when navigating web pages, which could affect accuracy in measuring interaction effort. However, shortcuts are not part of standard keyboard usage in web apps and are generally considered a feature-level concern rather than a core accessibility requirement [50]–[52]. Thus, our study focuses on standard navigation keys and skip content buttons, which do not rely on key combinations.

IX. Related Work

A. Keyboard Accessibility Studies of the Modern Web UI

There is extensive research on automatically detecting keyboard accessibility issues that violate Web Content Accessibility Guidelines (WCAG) standards [12], [17], [43], [69]–[71]. Existing work by Chiou et al. [12], [17], [69], [70] and Bostic et al. [43] focuses on detecting such violations. However, these tools do not evaluate the actual navigation experience or challenges faced by keyboard users. In contrast, our study empirically investigates disparities in user experience, emphasizing the effort required and differences in UI behaviors between PNC and KB interactions.

Previous empirical studies on KB navigation used cognitive modeling frameworks like GOMS (Goals, Operators, Methods, and Selection rules) to quantify performance and differences between keyboard and mouse interactions [27], [72]–[74]. However, these studies are over a decade old and covered a limited set of websites. With evolving web technologies and accessibility guidelines, updated studies are needed to understand keyboard navigation in modern web apps. Islam

et al. [75] proposed a probabilistic model for desktop apps accessibility, emphasizing non-visual, keystroke-based interactions. In contrast, our research empirically evaluates real-world disparities between keyboard and mouse navigation across diverse web apps, focusing on practical user experience challenges rather than predictive metrics.

Other studies have focused on the prevalence of accessibility issues in mobile apps and their impact on end-users [76]–[81]. In contrast, our study provides an empirical assessment focused explicitly on the disparities in user experience between KB and mouse-based users.

B. UI State Exploration Web Crawlers

Existing research around web UI crawlers focused on analyzing the state exploration of web pages from the PNC user's standpoint [33]–[39], [41], [43]. A popular and well-cited web UI crawler, Crawljax [44], uses iterative depth-first search to identify UI states and transitions, generating a State Flow Graph (SFG). It is widely used for testing and verification and has gained significant attention from researchers [37], [38], [56], [59]–[61], [63], [64], [66], [67]. Another recent tool is QExplore [39], which uses Q-learning exploration techniques to capture the different UI states using a guided search strategy and creates a SFG. Lastly, Qualstate [45] is a tool that interacts with each actionable element in order to explore the UI states of web pages. It can also be configured to interact with certain interactive elements or fill out login forms. While these tools detect broken links, security issues, and functional errors, they do not capture keyboard navigation, relying on PNC interactions. Our study addresses this gap with Krawler, which simulates keyboard interactions to explore interfaces, trigger dynamic content, and record focus movements, representing diverse UI behaviors as a graph (Section III).

X. Conclusion

In this paper, we conducted an extensive empirical study on keyboard-based navigation in web apps, assessing accessibility, effort, and UI behavioral differences compared to point-and-click navigation. Our analysis revealed significant disparities, with many web apps requiring a much larger number of keystrokes for basic tasks, making keyboard navigation inefficient. We identified issues such as excessive navigation loops, keyboard traps, and browser-specific inconsistencies that increase the interaction cost. Worse, some UI behaviors were inaccessible via the keyboard, highlighting the limitations of current web design practices. We also identified design patterns that support keyboard accessibility, as well as anti-patterns that hinder accessibility and increase user effort. Our findings emphasize the need for user-friendly keyboard navigation to improve accessibility. Addressing these challenges can ensure a more inclusive web experience, enabling keyboard users to interact with web apps as efficiently as mouse users.

References

[1] World Wide Web Consortium (W3C), "Making the web work," 1994, accessed on November 28, 2023. [Online]. Available: https://www.w3.org

[2] W3C, "Web Content Accessibility Guidelines," 2023, accessed on November 28, 2023. [Online]. Available: https://www.w3.org/TR/WCAG21

[3] S. L. H. James Nurthen, Michael Cooper, "Web Accessibility Initiative-Accessible Rich Internet Applications," 2022, accessed on November 28, 2023. [Online]. Available: https://www.w3.org/WAI/standards-guidelines/aria

[4] European Commission, "Web Accessibility Directive-Standards and Harmonisation," 2021, accessed on November 28, 2023. [Online]. Available: https://digital-strategy.ec.europa.eu/en/policies/web-accessibility-directive-standards-and-harmonisation

[5] Section 508, "Our Mission," 2017, accessed on November 28, 2023. [Online]. Available: https://www.section508.gov

[6] World Health Organization, "Disability," 2023, accessed on November 28, 2023. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/disability-and-health

[7] WCAG 2.2 Understanding Docs, "Understanding Guideline 2.1: Keyboard Accessible," 2025, accessed on March 7th, 2025. [Online]. Available: https://www.w3.org/WAI/WCAG22/Understanding/keyboard-accessible

[8] UC Denver, Center for Innovative Design and Engineering, "Alternative Keyboards," 2025, accessed on March 7th, 2025. [Online]. Available: https://www.ucdenver.edu/centers/center-for-inclusive-design-and-engineering/community-engagement/colorado-assistive-technology-act-program/technology-and-transition-to-employment/alternative-keyboards

[9] Top 5 Accessibility, "Accessible keyboards and their main features, explained," 2025, accessed on March 7, 2025. [Online]. Available: https://top5accessibility.com/blog/accessible-keyboards-their-main-features-explained

[10] U.S. Department of Justice and Civil Rights Division, "Guidance on Web Accessibility and the ADA," 2022, accessed on May 28, 2025. [Online]. Available: https://www.ada.gov/resources/web-guidance

[11] WCAG, "Understanding Success Criterion 2.4.3: Focus Order," 2023, accessed on January 8th, 2024. [Online]. Available: https://www.w3.org/WAI/WCAG21/Understanding/focus-order.html

[12] P. T. Chiou, A. S. Alotaibi, and W. G. J. Halfond, "Detecting and localizing keyboard accessibility failures in web applications," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*, August 2021.

[13] W. A. R. W. M. Isa, A. I. H. Suhaimi, N. Ariffrn, N. F. Ishak, and N. M. Ralim, "Accessibility evaluation using Web Content Accessibility Guidelines (WCAG) 2.0," in *Proceedings of the 4th International Conference on User Science and Engineering (i-USEr)*, August 2016.

[14] T. Haanperä and M. Nieminen, *Usability of Web Search Interfaces for Blind Users - A Review of Digital Academic Library User Interfaces*. Springer Berlin Heidelberg, 2013, p. 321–330.

[15] WebAIM, "WAVE Web Accessibility Evaluation Tools," 2023, accessed on November 28, 2023. [Online]. Available: https://wave.webaim.org

[16] N. Fernandes, D. Costa, S. Neves, C. Duarte, and L. Carriço, "Evaluating the accessibility of rich internet applications," in *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility (W4A 2012)*, April 2012.

[17] P. T. Chiou, A. S. Alotaibi, and W. G. J. Halfond, "BAGEL: An approach to automatically detect navigation-based web accessibility barriers for keyboard users," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI 2023)*, April 2023.

[18] W. M. Watanabe, R. P. M. Fortes, and A. L. Dias, "Acceptance tests for validating ARIA requirements in widgets," *Universal Access in the Information Society*, vol. 16, no. 1, pp. 3–27, October 2015.

[19] Deque, "The accessibility testing tool built for development teams," 2023, accessed on November 28, 2023. [Online]. Available: https://www.deque.com/axe-devtools-accessibility-testing

[20] S. Yang, C. Lee, H. V. Shin, and J. Kim, "Snapstream: Snapshot-based interaction in live streaming for visual art," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI 2020)*, Apr. 2020, pp. 1–12.

[21] A. Jaimes and N. Sebe, "Multimodal human–computer interaction: A survey," *Computer Vision and Image Understanding*, vol. 108, no. 1, pp. 116–134, 2007, special Issue on Vision for Human-Computer Interaction.

[22] M. Imani and G. A. Montazer, "A survey of emotion recognition methods with emphasis on E-Learning environments," *Journal of Network and Computer Applications*, vol. 147, 2019.

[23] Web Accessibility Survey, "Survey Results," 2023, accessed on May 28, 2025. [Online]. Available: https://webaccessibilitysurvey.com/survey-results

[24] AccessibleWeb, "How Do I Navigate a Website Using Only a Keyboard?" accessed on July 7th, 2025. [Online]. Available: https://accessibleweb.com/question-answer/navigate-website-keyboard

[25] Colorado State University, College of Health and Human Sciences, "Keyboard Navigation," accessed on July 7th, 2025. [Online]. Available: https://www.chhs.colostate.edu/accessibility/best-practices-how-tos/keyboard-navigation

[26] Accessible Technology, University of Washington, "Keyboard accessibility," accessed on July 7th, 2025. [Online]. Available: https://www.washington.edu/accesstech/checklist/keyboard

[27] M. Schrepp, "On the efficiency of keyboard navigation in web sites," *Universal Access in the Information Society*, vol. 5, no. 2, pp. 180–188, June 2006.

[28] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, 2019.

[29] Emilio Dallatorre, "Adult Host List," 2024, accessed on March 20th, 2024. [Online]. Available: https://github.com/emiliodallatorre/adult-hosts-list

[30] Virus Total, "Analyse suspicious files, domains, IPs and URLs to detect malware and other breaches, automatically share them with the security community," 2024, accessed on March 20th, 2024. [Online]. Available: https://www.virustotal.com

[31] W. G. Cochran, *Sampling techniques*, 3rd ed. John Wiley & Sons, Inc, 1977.

[32] C. Z. Chaniotaki, P. T. Chiou, and W. G. Halfond, "Replication Package for the paper "Assessing the User Interaction Cost of Keyboard Navigation in Web Applications"," 2026. [Online]. Available: https://zenodo.org/records/20647979

[33] M. Biagiola, F. Ricca, and P. Tonella, *Search Based Path and Input Data Generation for Web Application Testing*. Springer International Publishing, 2017, p. 18–32.

[34] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, *jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications*. Springer International Publishing, 2015, p. 295–316.

[35] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Synthesizing web element locators (T)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, 2015, pp. 331–341.

[36] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, 2011, pp. 571–580.

[37] Y. Li, P. Han, C. Liu, and B. Fang, "Automatically Crawling Dynamic Web Applications via Proxy-Based JavaScript Injection and Runtime Analysis," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC 2018)*. IEEE, Jun. 2018, pp. 242–249.

[38] C.-H. Liu, W.-K. Chen, and C.-C. Sun, "Guide: An interactive and incremental approach for crawling web applications," *The Journal of Supercomputing*, vol. 76, no. 3, pp. 1562–1584, March 2018.

[39] S. Sherin, A. Muqeet, M. U. Khan, and M. Z. Iqbal, "QExplore: An exploration strategy for dynamic web applications using guided search," *Journal of Systems and Software*, vol. 195, p. 111512, 2023.

[40] S. Khodayari and G. Pellegrino, "JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*. USENIX Association, Aug. 2021, pp. 2525–2542. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/khodayari

[41] T. S. d. Moura, E. L. G. Alves, H. F. d. Figueirêdo, and C. d. S. Baptista, "Cytestion: Automated GUI testing for web applications," in *Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES '23)*, 2023, pp. 388–397.

[42] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black Widow: Blackbox data-driven web scanning," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1125–1142.

[43] T. Bostic, J. Stanley, J. Higgins, D. Chudnov, J. F. Brunelle, and B. Tracy, "Automated evaluation of web site accessibility using a dynamic accessibility measurement crawler," *arXiv preprint arXiv:2110.14097*, 2021, https://api.semanticscholar.org/CorpusID:239998141.

[44] A. Mesbah, E. Bozdag, and A. Van Deursen, "Crawling AJAX by inferring user interface state changes," in *Proceedings of the Eighth International Conference on Web Engineering*, 2008, pp. 122–134.

[45] F. Rosa Martins, L. Seixas Pereira, and C. Duarte, "QualState: Finding website states for accessibility evaluation," in *Proceedings of the 21st International Web for All Conference (W4A 2024)*, 2024, pp. 96–105.

[46] W3C, "How People with Disabilities Use the Web," 2024, accessed on Augst 4th, 2024. [Online]. Available: https://www.w3.org/WAI/people-use-web

[47] S. Saha, A. Senapati, and R. Maity, "An Approach to Predict the Task Efficiency of Web Pages," *Multimedia Tools and Applications*, vol. 82, no. 16, pp. 25 217–25 233, February 2023.

[48] E. F. LoPresti, H. H. Koester, and R. C. Simpson, "Measuring keyboard performance for people with disabilities," 2006, available online: https://api.semanticscholar.org/CorpusID:61660804.

[49] Koester Performance Research, "Text Entry Rate for People with Physical Disabilities (Infographic)," 2025, accessed on March 1, 2025. [Online]. Available: https://kpronline.com/blog/text-entry-rate-for-people-with-physical-disabilities-infographic

[50] Deque, "Digital accessibility is the new global standard," 2025, accessed on July 7th, 2025. [Online]. Available: https://www.deque.com

[51] Tpgi, "Your Accessibility Partner," 2025, accessed on July 7th, 2025. [Online]. Available: https://www.tpgi.com

[52] Level Access, "End-to-End Digital Accessibility," 2025, accessed on July 7th, 2025. [Online]. Available: https://www.levelaccess.com

[53] B. Shneiderman and C. Plaisant, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5th ed. Pearson, 2004.

[54] A. Cooper, R. Reimann, D. Cronin, and C. Noessel, *About Face: The Essentials of Interaction Design*, 4th ed. Wiley, 2014.

[55] J. Raskin, *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Professional, 2000.

[56] N. Sunman, Y. Soydan, and H. Sözer, "Automated web application testing driven by pre-recorded test cases," *Journal of Systems and Software*, vol. 193, p. 111441, 2022.

[57] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Dependency-aware web test generation," in *Proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST 2020)*, 2020, pp. 175–185.

[58] F. R. P. T. Matteo Biagiola, Andrea Stocco, "Diversity-based web test generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, 2019, pp. 142–153.

[59] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of AJAX user interfaces," in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, 2009, pp. 210–220.

[60] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*, 2018, pp. 503–514.

[61] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "Apogen: Automatic page object generator for web testing," *Software Quality Journal*, vol. 25, pp. 1007–1039, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:207237213

[62] F. R. P. T. Andrea Stocco, Maurizio Leotta, *Clustering-Aided Page Object Generation for Web Testing*. Springer International Publishing, 2016, p. 132–151.

[63] A. M. Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models," in *Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE 2013)*, 2013, pp. 278–287.

[64] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 35–53, 2012.

[65] H. Tanida, M. R. Prasad, S. P. Rajan, and M. Fujita, *Automated System Testing of Dynamic Web Applications*. Springer Berlin Heidelberg, 2013, p. 181–196.

[66] C.-H. Liu, S. D. You, and Y.-C. Chiu, "A reinforcement learning approach to guide web crawler to explore web applications for improving code coverage," *Electronics*, vol. 13, no. 2, p. 427, 2024.

[67] W.-K. Chen, C.-H. Liu, and K.-M. Chen, *A Web Crawler Supporting Interactive and Incremental User Directives*. Springer Singapore, 2018, p. 64–73.

[68] K. Ruth, A. Fass, J. Azose, M. Pearson, E. Thomas, C. Sadowski, and Z. Durumeric, "A world wide view of browsing the world wide web," in *Proceedings of the 22nd ACM Internet Measurement Conference*, ser. IMC 2022. ACM, Oct. 2022, p. 317–336.

[69] P. T. Chiou, R. Winn, A. S. Alotaibi, and W. G. J. Halfond, "Automatically detecting reflow accessibility issues in responsive web pages," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024)*, April 2024, pp. 1–13.

[70] P. T. Chiou, A. S. Alotaibi, and W. G. J. Halfond, "Detecting dialog-related keyboard navigation failures in web applications," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*, 2023, pp. 1368–1380.

[71] M. Tafreshipour, A. Deshpande, F. Mehralian, I. Ahmed, and S. Malek, "Ma11y: A mutation framework for web accessibility testing," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*, 2024, pp. 100–111.

[72] S. Shao and H. Qin, *Application research on the webpage accessibility based on the GOMS model*. Springer Berlin Heidelberg, 2015, pp. 161–166.

[73] M. Schrepp and P. Fischer, *A GOMS model for keyboard navigation in web pages and web applications*. Springer Berlin Heidelberg, 2006, pp. 287–294.

[74] F. P. Schrepp Martin, "GOMS models to evaluate the efficiency of keyboard navigation in web units," *Eminds - International Journal of Human Computer Interaction*, vol. 1, pp. 33–46, January 2007.

[75] M. T. Islam, D. E. Porter, and S. M. Billah, "A probabilistic model and metrics for estimating perceived accessibility of desktop applications in keystroke-based non-visual interactions," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI 2023)*, 2023, pp. 1–20.

[76] S. Yan and P. G. Ramachandran, "The current status of accessibility in mobile apps," *ACM Transactions on Accessible Computing*, vol. 12, no. 1, pp. 1–31, February 2019.

[77] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez, "Can everyone use my app? An empirical study on accessibility in Android apps," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 41–52.

[78] A. Alshayban, I. Ahmed, and S. Malek, "Accessibility issues in Android apps: State of affairs, sentiments, and ways forward," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE 2020)*, 2020, pp. 1323–1334.

[79] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock, "An epidemiology-inspired large-scale analysis of Android app accessibility," *ACM Transactions on Accessible Computing*, vol. 13, no. 1, pp. 4:1–4:36, April 2020.

[80] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond, "Automated Detection of TalkBack Interactive Accessibility Failures in Android Applications," in *Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST 2022)*, 2022, pp. 232–243.

[81] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond, "Automated Repair of Size-Based Inaccessibility Issues in Mobile Applications," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*, 2021, pp. 730–742.