Bilkent University

Department of Computer Engineering

Senior Design Project

Willow: Graph-Based Browsing

Final Report

Efe Dağdemir, Tuana Türkmen, Sezin Zeydan, Can Cebeci, Cem Cebeci

Supervisor: Uğur Doğrusöz

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Table of Contents

1. Introduction

The internet has become an essential part, almost a necessity, in our everyday lives. We make use of the internet in so many different ways but one of the most important ones is through surfing the web. Millions of people are accessing the World Wide Web every single minute and web browsers are helping them do it. Although reaching the web by a browser is simple and easy, navigating between numerous tabs and managing different browsing sessions are not. One can easily get lost in between abundant tabs and lose track of the tasks in hand. As the number of tabs get larger, the amount of time lost due to confusion increases. Furthermore, backtracking on multiple tabs gets troublesome on the current model offered by the browsers. The features offered by web browsers for such tasks are limited and outdated. For instance, the linear structure of the tabs becomes hard to use as the size of the tabs gets smaller and identifying the desired tab between many others becomes burdensome with each new tab. Additionally, the concept of not being able to access a collection of tabs from previous browsing experiences is a flexibility limitation for the users.

In the search for a solution to these problems, we came up with Willow. Willow is a browser extension that could help to ease the process of browsing via understandable and intuitive visual navigation and session management features. Willow aims to enable users to efficiently visualize and manage their browsing sessions in a graph-based, interactive structure. The goal of our project is to take the current method of browsing to the next level by providing a visual structure that allows for smooth and intuitive navigation between the web pages browsed by the users as well as management features for browsing sessions.

2. Requirements Details

In this section we will list currently satisfied requirements of Willow.

2.1. Functional Requirements

- Visualization of the browsing session in a graph
- Navigation among websites using the graph
- Navigation through the graph, the graph can be panned and zoomed in order to focus on different parts of the browsing session.
- Organisation and visualisation of the browsing history
- The browsing history is portrayed as a set of sessions. To navigate around the browsing history, a session may be chosen to be visualized. The graph related to this session is displayed and can be searched and traversed, just like an ordinary session graph.
- Modification/customization of the graph
- Session saving and restoration
- Automatic node size determination done through applying PageRank to nodes.
- The user can create textual notes to be displayed on the graph as reminders or additional information.

2.2. Nonfunctional Requirements

Performance

The response time of Willow should be at most 30 milliseconds.

Scalability

The performance degradation of Willow with respect to the number of nodes should be sublinear.

Usability

The user interface of Willow should be intuitive enough that the average user can use the application to its full potential within 10 minutes of exploration.

Extensibility

The architecture of the software should be extensible enough to enable us to add new features based on user feedback before CSFair.

3. Final Architecture and Design Details

This section presents the overall architecture of Willow in terms of its functional decomposition into subsystems, packages and objects. The object design for the contents of each package is also presented in various UML[1] diagrams. Section 4 describes the implementation challenges faced, and choices made, in implementing the design presented here as a Chrome extension.

Below is the package diagram that describes the final subsystem decomposition of Willow.

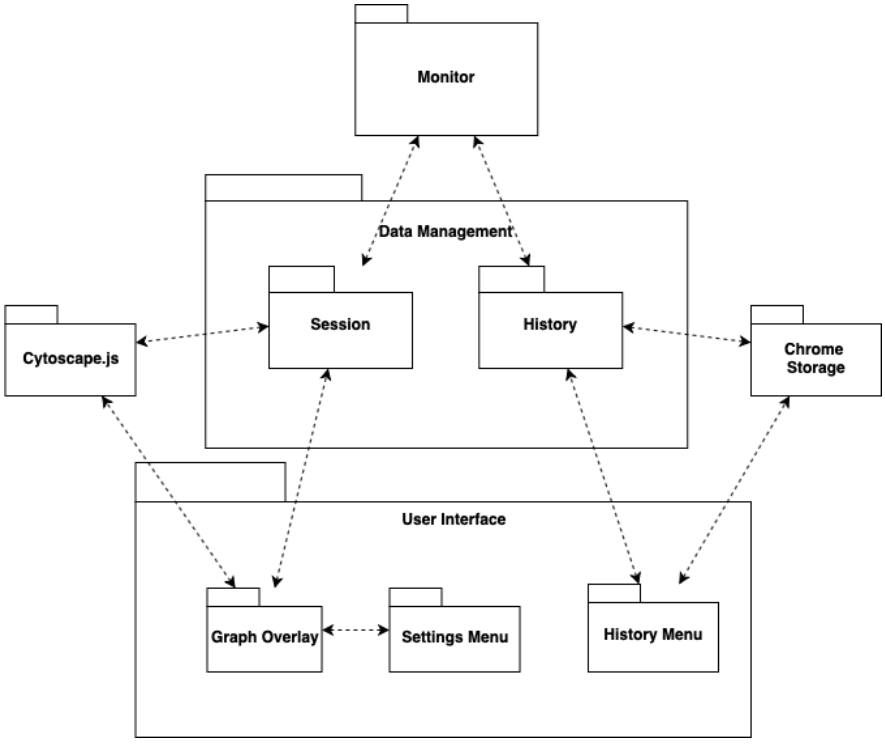

Figure 1: High-level subsystem decomposition of Willow

3.1 Data Management

3.1.1. Session

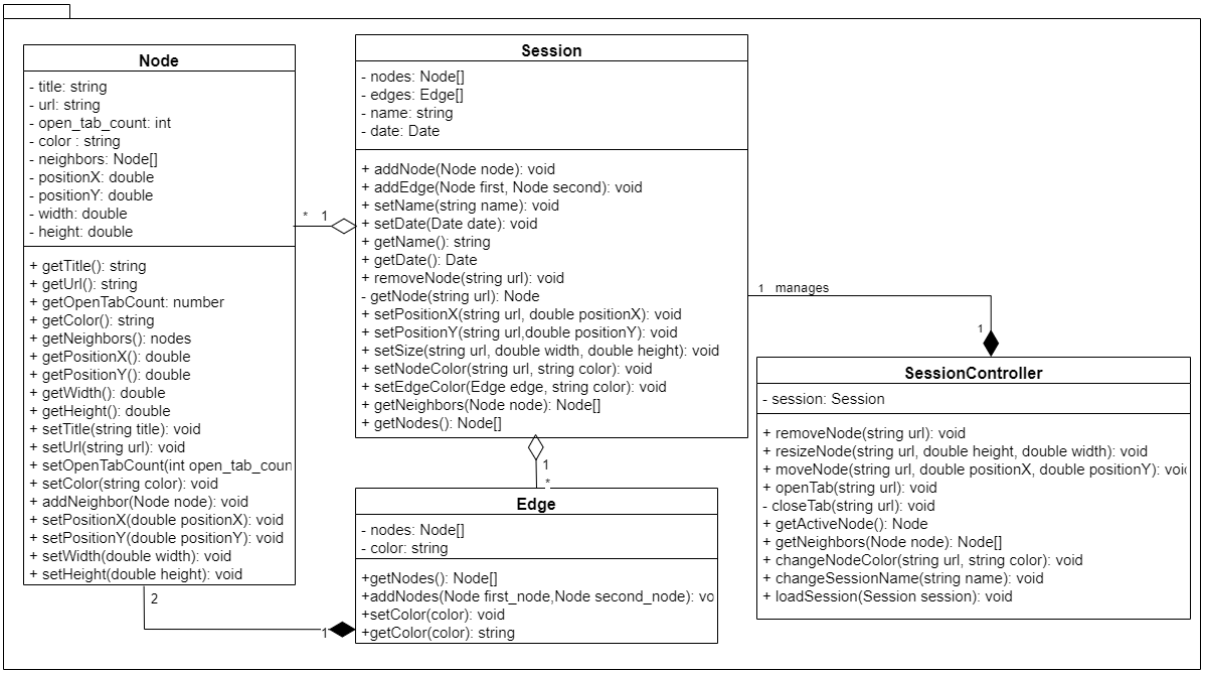

Figure 2: Class diagram of the classes of the Session package

Class: Node
Node class corresponds to the nodes in the graph. It has attributes to distinguish different cases of a node. Alongside the basic get and set operations, a function to add a neighbor to a node is presented.

Attributes:
- **title:** string
- **url:** string
- **openTabCount:** int
- **color:** string
- **neighbors:** Node[]
- **positionX:** double
- **positionY:** double
- **width:** double
- **height:** double

Functions:
- **addNeighbor(Node node):** is called whenever a neighboring node is added to a node. The added node becomes one of the neighbors.

Class: Edge
Edge class is the other part of the graph other than nodes. Used to link two nodes together.

Attributes:
- **nodes:** Node []
- **color:** string

Functions:
- **addNodes(Node firstNode, Node secondNode):** is used to attach the two end nodes of an edge.

Class: Session
This class represents the browsing session of a user. It has nodes and edges (representing a graph). Alongside the get and set operations, a node can be added to a session or can be removed from it.

Attributes:
- **nodes:** Node[]
- **edges:** Edge[]
- **name:** string
- **date:** Date

Functions:

- **addNode(Node node):** is called when a new node will be added to a session. The
- **removeNode(Node node):** is called when a node is removed from session.
- **getNode(string url):** is used to find the node in the graph which contains the provided url as its url.

Class: SessionController

This class only contains a session and is used to manage it. In order to make operations on a session, sessionController directs the operation via presenting the corresponding url and if needed additional information (such as color) to the session. Additionally it can open a Chrome tab with the given url and can load a session to be the current one.

Attributes:

- **session:** Session

Functions:

- **addNode(Node predecessor, Node node):** this method adds a new node to session.
- **removeNode(Node node):** is called when a node is removed from the session graph. This method subsequently calls close_tab(string url) method with the url of node to also close the tab from browser.
- **Private closeTab(string url):** this method is called from remove_node(Node node) method to close the tab that corresponds to removed node.
- **resizeNode(Node node, double height, double width):** is called when the size of node is changed.
- **moveNode(Node node, double positionX, double positionY):** is called when the node is moved on the session graph.
- **openTab(Node node):** is called to open a new Chrome tab which has the url of the provided node.
- **changeNodeColor(Node node, string color):** is called when a node's color will be changed.
- **changeSessionName(string name):** changes the current session's name to the provided name.
- **loadSession(Session session):** is called when a session will be loaded as the current session.

3.1.2 History

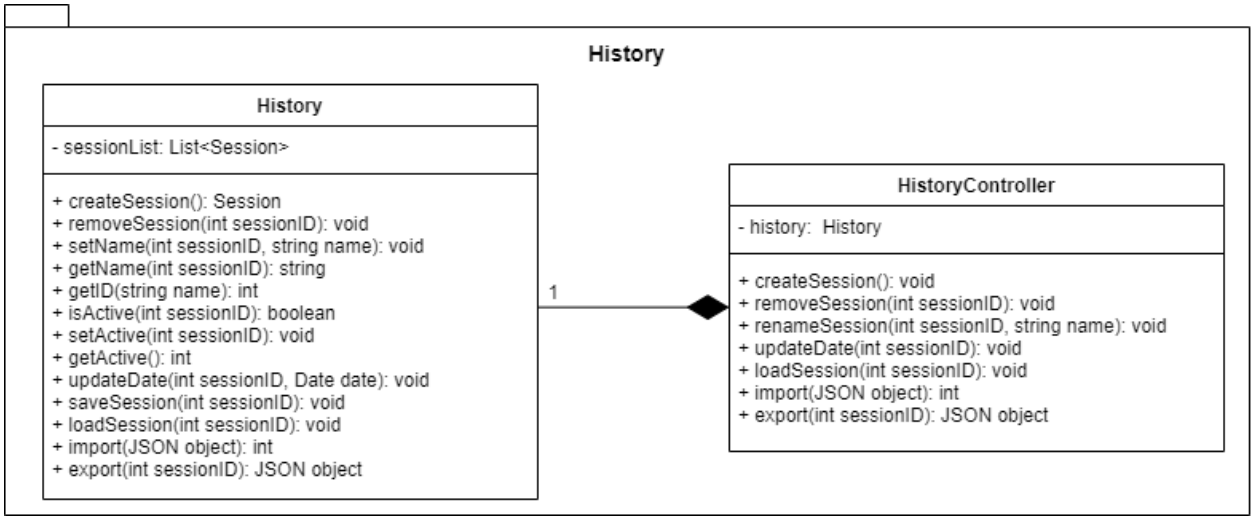

Figure 3: Class diagram of the classes of the History package

The HistoryService package consists of two classes that work together to provide the underlying structure that both deals with the storage of the history and the operations that can be done regarding the history. Classes in this package keep information about the sessions stored in the local machine as a list and perform several different types of actions starting from adding/removing sessions to importing/exporting sessions.

Class: History

History class is responsible for the storage of sessions and the various actions performed on the history, although the actions are designed to be not directly invoked on the object but rather from the controller object.

Attributes:

- **sessionList:** This is the list that holds all the session objects.

Functions:

- **createSession():** Gives the call to create a session and inserts the created session into the *sessionList*.
- **removeSession(int sessionID):** Removes the given session from the *sessionList*.
- **setName(int sessionID, string name):** Sets the name of the given session with the given name.
- **getName(int sessionID):** Gives the name of the session with the given sessionID.
- **getID(string name):** Gives the ID of the session with the given session name.
- **isActive(int sessionID):** Specifies whether the given session is active.
- **setActive(int sessionID):** Sets the given session as active.
- **getActive():** Gives the sessionID of the active session.

- **updateDate(int sessionID, Date date):** Sets the last accessed date of the given session.
- **saveSession(int sessionID):** Saves the session to the local machine.
- **loadSession(int sessionID):** Loads the given session as the current session.
- **import(JSON object):** Import a session outside of Willow's storage as the current session.
- **export(int sessionID):** Export the specified session as a JSON object.

Class: HistoryController

HistoryController class is the controller of the HistoryService package, it provides the useful actions that can be performed on the history as an interface.

Attributes:

- **history:** An instance of a *History* object, this is what this controller class will perform actions on.

Functions:

- **createSession():** Invokes the *createSession()* function on the History object to create a new session.
- **removeSession(int sessionID):** Invokes the *removeSession()* function on the History object to remove a session from the history.
- **renameSession(int sessionID):** Invokes the *setName()* function on the History object to rename the session.
- **updateDate(int sessionID):** Invokes the *updateDate()* function on the History object to update the last accessed date of the session.
- **loadSession(int sessionID):** Invokes the *loadSession()* function on the History object to load the selected session as the current session.
- **import(JSON object):** Invokes the *import()* function on the History object to import a session outside of Willow's storage as the current session.
- **export(int sessionID):** Invokes the *export()* function on the History object to export the selected session as a JSON object.

3.2 User Interface

3.2.1 Graph Overlay

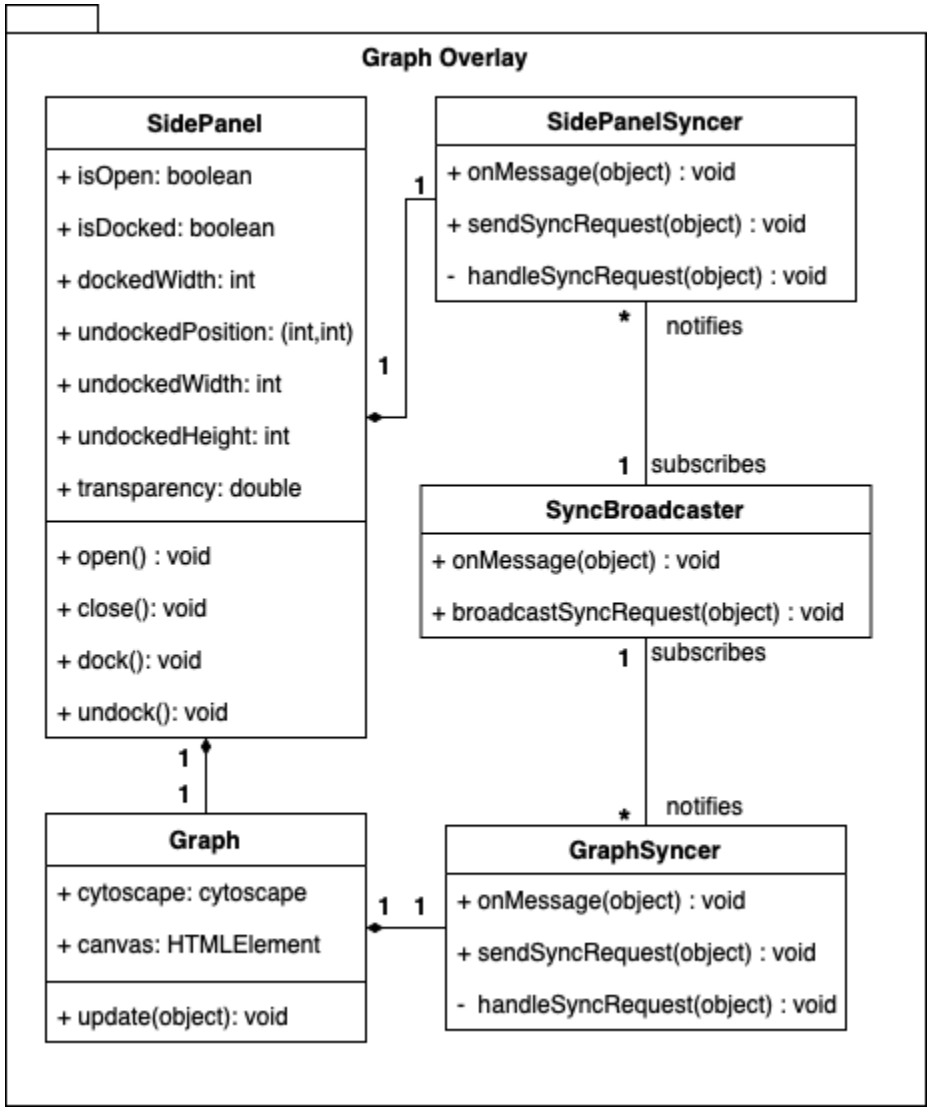

Figure 4: Class diagram of the classes of the Graph Overlay package

Class: SidePanel

A user's main method of interaction with Willow is a collapsible, undockable side panel. Due to the limiting boundaries of what a Chrome extension can do, this side panel is injected as an HTML element by a content script [2] to the web page open in each tab in a Chrome window. This results in the proliferation of many instances of the side panel, each associated with an instance of the SidePanel class. This class encapsulates various properties regarding the state of the html element and defines methods which modify this state.

Attributes:

- **isOpen:** indicates whether the side panel is in its expanded (open) or collapsed (closed) form.
- **isDocked:** indicates whether the expanded form of the side panel resides along an edge of the window, in which case it is said to be in the docked state. An undocked panel, in contrast, can be moved anywhere on the page and scaled freely both in height and width.
- **dockedWidth:** the width of the panel in its docked state. Irrelevant when the panel is undocked. Retained for use to restore the panel to its width when it is docked again.
- **undockedPosition:** the pixel coordinates of the top left corner of the panel in its undocked state. Irrelevant when the panel is docked. Retained for use to restore the panel to its position when it is undocked again.
- **undockedWidth**: the width of the panel in its undocked state.
- **undockedHeight**: the height of the panel in its undocked state.
- **transparency**: the transparency of the panel. Can be configured by the user so that the page behind the panel is visible to a desired degree.

Functions:

- **open()**: expands the panel, bringing it to its open form, covering a portion of the page behind it and allowing it to display the session graph.
- **close():** collapses the panel, bringing it to its closed form, displayed as a small icon.
- **dock():** docks the panel, attaching it to an edge of the window and keeping it from being dragged.
- **undock():** undocks the panel, allowing it to be dragged and placed anywhere on the page.

Class: SidePanelSyncer

As described above, Willow injects a separate HTML element, associated with a separate instance of the side panel into the web page open at each panel. In the face of this non-singularity, the need to synchronize the state of the panel across all tabs arises. For example, when the panel in the active tab is opened, the panels in the other open tabs should be opened as well. This also applies to closing, docking, undocking, resizing and moving the panel. This type of synchronization hides the multiplicity of the panel and gives the impression of a single panel existing across all tabs. SidePanelSyncer instances reside in content scripts and ensure this synchronization by sending and receiving sync requests to/from the SyncBroadcaster.

Functions:

- **onMessage(object):** called whenever the page receives a message. Forwards the message to handleSyncRequest if the message is a sync request that was broadcast by the SyncBroadcaster.
- **sendSyncRequest(object)**: called when the state of a panel or any of its visible properties changes. Sends a sync request to the SyncBroadcaster so that the side panels in other tabs are notified of the change.
- **handleSyncRequest(object):** updates the associated SidePanel accordingly to the received syncRequest.

Class: SyncBroadcaster

The SyncBroadcaster is a singleton object that resides in a background script [4]. Its purpose is to listen for sync requests and broadcast them. Such an intermediary object is needed since the SidePanel objects themselves reside in content scripts and due to security considerations, Chrome does not allow direct messages between content scripts.

Functions:

- **onMessage(object):** much like the mechanism in SidePanelSyncer, this function is the generic message receptor that is called by Chrome. It forwards the message to broadcastSyncRequest if it is a sync request.
- **broadcastSyncRequest(object):** sends the request to all SidePanelSyncer objects except the one that originated the sync request.

Class: Graph

The graph is displayed within the side panel and together they make up the graph overlay. To display the graph, we rely on cytoscape.js[3], which draws the graph on a dedicated HTML div, referred to as the cytoscape canvas. Just like SidePanel, the graph display is duplicated across open tabs and GraphSyncer instances are used to synchronize them.

Attributes:

- **cytoscape**: the cytoscape.js instance which displays itself of the canvas
- **canvas:** the dedicated HTML div element which contains the graph display

Functions:

- **update(object):** called by the associated GraphSyncer object. The input object indicates the parameters that need updating. Modifies the cytoscape instance so that it is synchronized with the instances in other tabs.

Class: GraphSyncer

Just like SidePanelSyncer, GraphSyncer acts as the communication channel between the Graph objects separately injected to each page and the singleton SyncBroadcaster. It sends sync requests when changes are made to its associated Graph object and likewise modifies the Graph object when sync requests are received.

Functions:

- **onMessage(object):** called whenever the page receives a message. Forwards the message to handleSyncRequest if the message is a sync request that was broadcast by the SyncBroadcaster.
- **sendSyncRequest(object):** called when the state of the related graph or any of its visible properties changes. Sends a sync request to the SyncBroadcaster so that the side graphs in other tabs are notified of the change.
- **handleSyncRequest(object):** updates the associated Graph accordingly to the received syncRequest.

3.2.2 Settings Menu

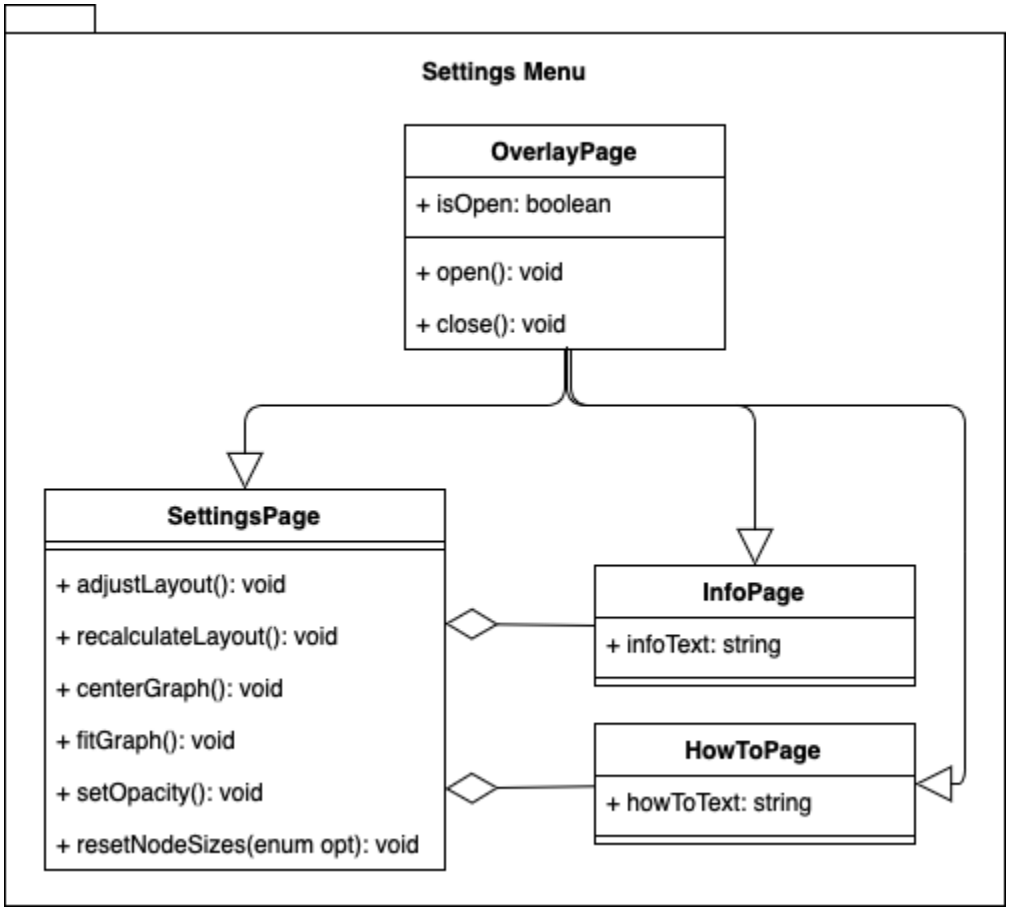

Figure 5: Class diagram of the classes of the Settings Menu package

Class: OverlayPage

The settings menu and its submenus share certain functionalities, which are abstracted into a superclass called OverlayPage. These functionalities maintain the state of the variable isOpen, which needs to be synchronized across the instances of an OverlayPage inserted in each Chrome tab.

Attributes:

- **isOpen:** Indicates whether the page is open and visible to the user. Needs synchronization across tabs as well as storage of its state for the initialization of the page in newly-opened websites.

Functions:

- **open():** opens the overlay page and makes it visible.
- **close():** closes the overlay page and makes it invisible.

Class: SettingsPage

The SettingsPage class constructs and displays a settings page popup on top of the session graph when the user presses the settings button. Using this popup, the user can perform various actions. Since all logic is handled at the Data Management package, this class simply forwards the operations there.

Functions:

- **adjustLayout():** Makes incremental changes to the session layout to make it more understandable. Preserves the current layout.
- **recalculateLayout():** Discards the current layout and calculates a new layout only using the topology from scratch.
- **centerGraph():** Brings the viewport to the graph's center.
- **fitGraph():** Adjusts the viewport so that the user can see all nodes in the graph.
- **setOpacity():** Changes the opacity of the injected sidePanel.
- **resetNodeSizes(enum opt):** Sets the sizes of all nodes in the graph. Depending on the parameter opt, this can set them to a uniform size or apply a PageRank algorithm to determine new sizes.

Class: InfoPage

The InfoPage class is responsible for displaying and managing the information page, a submenu that can be accessed through the settings page and presents the user with

information about Willow. This information includes contact details, presentation of the project team and attribution of the used resources to their owners.

Attributes:

- **infoText:** the body of hypertext that is formatted and presented within the InfoPage.

Class: HowToPage

Similar to InfoPage, this class displays a popup which informs the user. The contents of this popup include information on how to use the software.

Attributes:

- **howTo:** the body of hypertext that is formatted and presented within the HowToPage.

3.2.3 History Menu

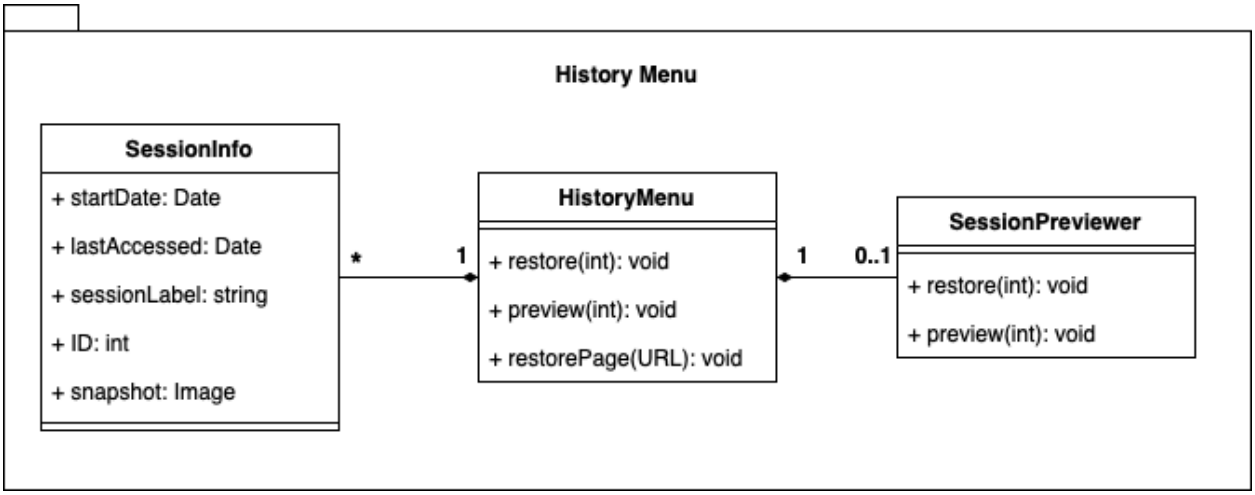

Figure 6: Class diagram of the classes of the History Menu package

The history menu package contains classes that interact with each other and exchange messages with the background pages to override Google Chrome's history page and display the browsing history as a list of sessions. The user may then preview some sessions to identify the one they are looking for and eventually choose to either restore a page and open it in the current session or restore a previous session and keep browsing in it. Classes in this package do not maintain data about the history or the previous sessions, they only read it from the History package described at section 3.1.2. Similarly, they do not perform the restore actions but request them from the History package instead.

Class: HistoryMenu

The class HistoryMenu shares the name of the package and predictably, it's the class that manages the rest of the classes in the package. It's responsible for initializing the history menu by retrieving the history data, organizing it and then presenting it to the user by overriding the Chrome history page. It's public functions correspond to the history menu use cases.

Functions:

- **void restoreSession(int):** restores the open tabs from the session with the given session id in the current Chrome window. Willow will start displaying the browsing graph of that session session as well. Any browsing actions done by the user will result in changes in the restored session. The function does not perform any changes on its own but requests them from other packages.
- **preview(int):** presents a preview of the session with the given session id. This request is received from one of the SessionInfo objects and passed to a SessionPreviewer object. HistoryMenu does not handle this request.

Class: SessionInfo

The class SessionInfo contains the information about a single session. A list of SessionInfo objects is maintained and the same list is represented visually to the user. Objects of this class are simply data items.

Attributes:

- **Date startDate**: the date at which the session started.
- **Date lastAccessed:** the date at which the session was last accessed.
- **string sessionLabel:** the label the user enters for the session.
- **int ID:** the unique id of the session.
- **Image snapshot**: a snapshot of the session's graph in image format.

Class: SessionPreviewer

The SessionPreviewer class's responsibility is to draw a preview of a session for the user to interact with. The user uses the session labels in conjunction with these previews to find the session they are searching for. The session previews support zoom and pan operations but no actual modifications to the session graphs.

Functions:

- **void restore(int)**: a request sent by the graphic elements for the session with the given id to be restored. This request is passed to the HistoryMenu object managing the SessionPreviewer instance.
- **void preview(int):** a request sent by the HistoryMenu for the session with the given id to be previewed. This class creates an appropriate HTML element to contain the graph representation and draws on it.

3.3 Monitor

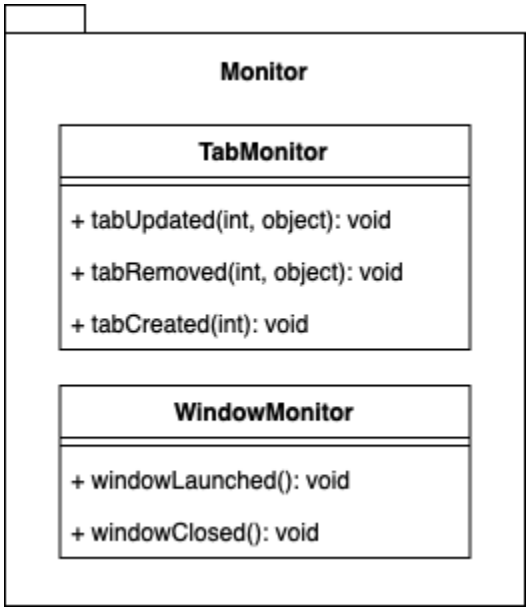

Figure 7: Class diagram of the classes of the Monitor package

Classes in the Monitor package are responsible for monitoring the user's browsing behaviour and reporting certain events to the relevant system components. The Chrome API's event listeners are used for this purpose. The monitored events are divided into two parts. User actions regarding tabs and user actions regarding windows.

Class: TabMonitor
Monitors user behaviour on tabs.
Functions

- **void tabUpdated(int, object):** A listener for the chrome.tabs.onUpdated event.
- **void tabRemoved(int, object):** A listener for the chrome.tabs.onRemoved event.
- **void tabCreated(int):** A listener for the chrome.tabs.onCreated event.

Class: WindowMonitor
Monitors user behaviour on windows.
Functions

- **void windowLaunched(chrome.Window):** A listener for the chrome.window.onCreated event.
- **void windowClosed(int):** A listener for the chrome.window.onRemoved event

The diagrams above represented our object oriented design on how to implement the system. The final implementation, however, is not a standalone software; it is a Chrome Extension. The constraints imposed on Chrome extensions such as languages used and source code organization necessitated some alterations on this design. Our final implementation uses this organization and architectural outline but we do not include a file for all classes shown here. Instead, we merge some combinations into single scripts and preserve the logical separation between the classes via functions. Section 4 dives deeper into the implementation and it's organization.

4. Development/Implementation Details

4.1 Adaptation of the object design to Chrome extension architecture

As described in Section 3, the implementation of our object design was heavily constrained by the architectural style Chrome imposes on extensions. We begin Section 4 by discussing how we have organised the design so that Chrome could accommodate it.

Chrome extensions comprise different types of HTML pages and scripts. The two main classes are content scripts, which are JavaScript files that are run on web pages that match certain selectors, and background scripts, which are inserted into a so-called "background page" that is generated by Chrome when the extension is installed. Aside from these two types of scripts, our extension includes HTML documents that are presented to the user, either by opening them in a new tab (as in the case of the history menu) or by loading them into an iframe that is injected into a webpage by a content script.

4.1.1 The Monitor package:

The Monitor package is completely implemented in a background script. Background scripts are authorized to use various Chrome API's such as tabs and windows. The background script adds a listener to the `chrome.tabs.onUpdated` and `chrome.tabs.onRemoved` events for the TabMonitor and `chrome.window.onRemoved` for the WindowMonitor. Using these Chrome events, the Monitor package can track what the user does.

TabMonitor
When a tab is updated to change it's URL, the TabMonitor checks to see if a new node needs to be created. If there does not exist a node with the same URL in the SessionGraph already, a node needs to be created.
Then, the monitor needs to decide whether the URL update should result in an edge in the graph. For this, it uses the `chrome.history.getVisits` function. We have resorted to this method because the `chrome.tabs` API does not include any functionality to determine how a new URL is loaded. `chrome.history.getVisits` returns an array of visit objects sorted by their dates. The last one is always the last visit, which should have been saved at the brand new visit caused by our changing URL. If the last visit's type is `link` or `form_submit`, a new edge should be created. To determine the source node of the new edge, the TabMonitor also keeps a Map that tracks the URL's tabs contain.

WindowMonitor
The WindowMonitor's task is much simpler since conceptually, all windows have the same session running among them. Also, when Google Chrome shuts down, all data structures on the background pages are torn down. However, this means the latest updates to the SessionGraph will also be torn down. Thus, the WindowMonitor signals the History subsystem to save the current SessionGraph as the last window closes.

4.1.2 The Graph Overlay Package

The main means of interaction a user has with Willow is through a collapsible side panel where the session graph is displayed, this side panel is implemented and managed by the Graph Overlay Package. Since Chrome's API does not include functionalities that allow extensions to add new browser UI elements, implementing this package proved to be highly challenging.

Broadly, our implementation injects the side panel as an HTML element with fixed positioning and a very large z-index (so that it floats over the page content) to all web pages the user visits. A content script with a universal match selector (declared with `"matches": ["<all_urls>"]` in the extension manifest) achieves this injection. A number of synchronization mechanisms we have implemented make sure that the numerous instances, one per open tab, of the side panel HTML element appear the same at all times and are positioned at exactly the same locations in each tab. This creates the impression that the side panel is a UI element and not a part of the web page.

SidePanel and SidePanelSyncer
The aforementioned universally-inject content script implements the SidePanel class and injects the HTML elements that constitute its user interface to all web pages the user visits. The same script injects SidePanelSyncer as well. These two objects have access to each other's methods since they reside within the same web page, which results in them sharing their JavaScript namespace.

Graph and GraphSyncer
The Graph is conceptually and visually a part of the SidePanel. In terms of implementation, however, it differs from the SidePanel significantly. The HTML elements that constitute the SidePanel are injected directly into the host page (the page the user has visited) while the Graph is implemented in a separate extension-domain HTML page. This page is loaded into an iframe which the SidePanel script inject within the panel.

This fundamental distinction in the implementation of these two classes is not an arbitrary implementation choice. The Graph has to be implemented in an extension-domain page since it accesses various extension assets (UI icons, web page favicons that are displayed inside nodes representing those pages etc.). If the Graph were injected as an HTML element to the host page, the host page, having a remote domain, would try to access the extension assets in the local extension domain. This would cause a plethora of security issues and hence is not allowed by Chrome's security policies.

SidePanel, on the other hand, has to be injected directly within the host page since it needs to have the capabilities to resize and move itself, which involves interacting with the host page's DOM. Due to the aforementioned security policies, such interaction would not be allowed if the SidePanel were implemented in an extension-domain script.

Much like SidePanelSyncer, GraphSyncer is injected into the very same extension-domain page as Graph so that these two objects can access the methods of each other.

SyncBroadcaster
Chrome prevents web pages, and thus content scripts, from sending messages to one another directly due to security concerns. Therefore, SyncBroadcaster is implemented as a background script. This allows the Syncer classes to send messages to SyncBroadcaster (since one is a background script) and then SyncBroadcaster broadcasts the messages to numerous content scripts.

Being a background script, the implementation of SyncBroadcaster resides in Willow's generated background page. This lets other background scripts access SyncBroadcaster as well, which has proven surprisingly useful.

4.1.3 The Data Management Package

The Data Management Package is implemented completely in background scripts. Different parts of this package are distributed to separate scripts that interact with each other. The classes in this package are not displayed to the user directly but they send messages to the content scripts to update the information on them.

The Session package

Session, Node and Edge
The session class has an implementation much simpler than how we designed it to be. This is due to a design decision we made to include a Cytoscape instance for the current session running in the background. The background Cytoscape instance is never rendered and its sole purpose is to keep track of the current session's model and perform layout operations. Maintaining a separate Cytoscape instance was a unifying decision because thanks to this decision, the persistent model, the logical model and the displayed model all have the same representation. Since Cytoscape handles the model classes for us, we did not implement separate classes for Session, Node and Edge.

SessionController
The SessionController is an entity that knows how to interact with the background Cytoscape instance. It maintains the information held by the model. It comprises many functions in the background page. These functions are called by either the Monitor package or the UI to make changes in the session. Most of the SessionController functionality merges Cytoscape's interface with various Chrome API to first make changes in the graph then illustrate the changes to the user.

The import/export functionality is also implemented in the SessionController. When exporting, the SessionController constructs a JSON object that represents the current cytoscape instance and uses the `chrome.downloads` API to have the user download this JSON representation. When importing, the SessionController reads that same representation to set up its graph. Then, the important Session's tab state needs to be restored. The SessionController then creates a new tab with url `chrome://newtab`,

closes all other open tabs, creates new tabs for all pages open in the newly imported session and finally closes the extra tab.

The History package

The history model is kept entirely in the `chrome.storage.local` API as a list of past sessions. The sessions are responsible of exporting themselves to be restored later. The HistoryController class appends a lastUpdated date, a session id, a session name and a png snapshot to the sessions exported by the SessionController and saves them to local storage in a list. This class also performs periodic saves of the current Session, in addition to the ones made on Session switches and Chrome shutdowns. The HistoryController gets running as Chrome does to signal SessionController to start a new session.

Settings Menu

The settings menu is implemented in an extension-domain script that resides in the same HTML document as the Graph. This script listens to messages from the SidePanel (since the button that toggles the settings menu resides there) and upon request, opens and closes the menu by injecting and removing a statically-defined HTML string into the extension-domain document.

Implementing the settings menu in a content script and injecting it into the host document would have made the interaction between the side panel and the settings menu much easier. This would allow for a more elegant implementation of the setting functionalities that affect the SidePanel's state (e.g. its opacity). However, this alternative causes the CSS rules defined in the host page to apply to the DOM elements of the setting menu, which causes the menu to have an inconsistent appearance in different web pages. This issue is present in styling the SidePanel's DOM elements as well. Fortunately, those elements are very few in number and they are mostly icons, which are not affected significantly by the host page's CSS. This lets us handle the issue very easily for the SidePanel, merely by overriding and enforcing a few styling traits using `!important` statements.

The implementation of the sub-menus (the info and how-to pages) are much like the settings menu itself. Aside from the synchronization mechanisms designed to make sure that they are either open in all tabs or closed in all tabs, which are described in detail in Section 4.2.1, they are implemented trivially by means of injection and removal of static DOM elements.

History Menu

The History Menu is an independent HTML document with its own CSS that is loaded on a new tab when the user presses the “History” button from the settings menu. This HTML document is also an extension page and hence has access to Chrome extension API. As discussed in the implementation of the History package, the history is entirely saved into the `chrome.storage.local` API. The History Menu reads that same stored data to present it to the user. The SessionInfo’s are handled as such.

4.2 Various Problems Faced and Solutions Implemented

4.2.1 User Interface Synchronization

Since Willow injects each tab as a different instance of graph. One of the challenges we faced while developing Willow was to synchronize UI components across tabs. There were many UI components we had to sync and each had their own logic. The synced components include: browsing graph, settings menu, how to page, information page, notes.

An example synchronization of Information Page / How to Page goes like this:

Whenever a user decides to open one of the above pages, that page should open in all other tabs as well. So that when users move on to another tab, they will be able to continue their experience from where they left off exactly. In order to succeed, we had to use some techniques. Firstly, `chrome.storage` API was used to keep track of the situation of the pages (whether they are open or not). Secondly, the message passing technique of `chrome.runtime` was used. Basically, whenever Information Page or How to Page opened, a message is sent to the event listeners within the extension and the state is set to “open”. After this message passing process, other tabs are instructed to open that page as well. The same process goes for closing these pages.

A situation that needed attention was, Information Page or How To Page can be closed from a different button other than their respective “cross” button which is the Settings button. In our design, Information Page and How To Page are considered as submenus of the Settings Menu. Thus, if the button to close the Settings Menu were to be clicked, then the state of these pages is checked from the `chrome.storage` API and if one of them is open, firstly that page is closed in all tabs. Then the Settings Menu is closed everywhere.

Another example was synchronising notes pop up.

We tried to implement the notes feature like the concept of sticky notes in which if a user opens notes page a small pop-up where users can take notes will show up on the side panel. However we quickly realized during testing that if a user opens notes on tab A, tab B did not open notes and that showed an inconsistency in our system. To solve this issue we also used the chrome.runtime API to send messages between different tabs. The logic is as follows: at first we initialize the notes state as false and whenever a user initiates the notes page we set that message to true and this message is sent to event listeners and when extension detects a change in event listeners it instructs other pages to open notes as well. Same logic applies to closing the notes pop up as well.

4.2.2 Treating links starting with “chrome://” differently

When a user visits an address starting with “chrome://”, in our design, Willow does not count that as a valid website and does not add it to the graph. To do this, firstly the links starting with this prefix were simply ignored. However, then it was discovered that when a user goes to one of these sites from an active page, the page user left remained as active on the graph. Because Willow did not see a “valid transformation from one site to another”. Thus, another option was added. This option first checks if any website with an address starting with the prefix is visited from a website that was visited before. And if the answer is yes, it is saved that the left website is now not open and this information is passed to every open tab via message passing. Therefore, if a node becomes “not active” because a page starting with “chrome://” is visited, every tab can know about it immediately.

4.2.3 Styling with Cytoscape.js Function values

Styling a graph in Cytoscape.js is mostly managed through the values that are inserted to the JSON style tag. However, some of the values of the nodes change with some actions. To exemplify, an active node which is shown with a green little circle above it can become “not active”. If this is the case, the green little circle should transform into grey. In order to achieve this, Willow needs to check if a node’s openTabCount is larger than 0. Thus, in order to manage this, we used Cytoscape.js’ function values which return whatever is appropriate according to other circumstances in a video. This way Willow is able to change the graph style dynamically.

4.2.4 Having different instances of Cytoscapes

Willow has different Cytoscape instances. One is where it can be considered as “background” and the other is in the “content part/front”. The layout is run on the background Cytoscape instance, the instance is then transformed into a JSON format and is passed to the front. When the sent Cytoscape instance is mounted at the front

(for every open tab), the style of the graph is applied to the mounted ones. Even though this design helps us makes many things easier, some obstacles were faced. Some of the features of Cytoscape are unfortunately not serializable. Thus when one tries to adjust a feature of Cytoscape such as Wheel Sensitivity, where to put it is important.

To restore a session, the History Menu uses the `chrome.runtime.sendMessage` function to pass the responsibility to some background page that loads the session with the given id. The History class catches this and performs the necessary operations.

For session previews, we opted to use simple PNG's exported by cytoscape itself. Any SessionInfo object includes with it a png, appended there by the History class. Session previews simply read that PNG and present it to the user.

4.2.5 CSPs Causing Interface Problems with Google Fonts

Content Security Policy (CSP) is a security measure to prevent attackers from injecting malicious code in websites. It is a simple yet effective layer of security that has become a standard across the web. Because the Willow side panel is displayed by injecting HTML code into web pages, the CSP settings of certain websites prevented Willow from displaying its user interface correctly.

During the design of the Willow icon, the typeface *Montserrat* was used. To stay consistent with our brand identity, we decided to use the same font throughout our extension. However, the font was not readily available for use in pure CSS, so we imported the font from Google Fonts. This decision caused many problems with CSPs and hindered the user interface unusable.

After trying many different approaches to resolve this problem, we realized that this was a common problem and that it was not possible to import a remote font source without fallbacks. Upon further research we came across web safe fonts. Web safe fonts are fonts that are readily available in most operating systems, which resolved the issue of using a remote font source for us. Weighing between losing functionality on certain sites versus visual design consistency, we decided to use a web safe font that matched our design to get the best of both worlds.

4.2.6 Turning the drawbacks of a resizable panel into a design feature

Making dynamic UI elements like a resizable panel come with certain design challenges. For us those challenges were about the synchronization, minimum size threshold and the fate of the UI elements when the panel is below a certain size. The former challenge and its solution is discussed above. This section will discuss the remaining.

At first we thought about setting the minimum size threshold to the width of all of the elements in the panel header but that made the smallest panel size too large and the resizability to be meaningless. Next, we considered eliminating elements from the header completely but all of them proved to be valuable. While some elements had their values in their functionality, some had it in their aesthetic visuals.

The solution was to make the header flexible, to implement a dynamic header for a dynamic panel. If we could make some parts of the header disappear below a size threshold and appear when it is back above it, it would make the header a dynamic component. It was certain that we could not sacrifice any functionality, this meant that we needed to sacrifice a component that brought aesthetic value. Choosing the visual element to sacrifice was easy, the icon was too valuable to sacrifice, so we decided on the label. However, the abrupt disappearance of the label made it seem like a bug rather than a feature. The answer to this problem was to animate the disappearance of the label. After making the label animated, everything seemed better than before. The rigid, problematic header transformed into a responsive, dynamic component that brought even more value than its previous state.

5. Testing Details

We did not use any specific tools for testing Willow therefore there are no test modules included in our codebase. However we did test the user interface frequently, to do that we followed our continuous integration pipeline where whenever one of us made a change to our main codebase we all tested the new feature extensively.

So for the verification of each component of the application we performed alpha tests, these tests are conducted by team members that were not involved in the implementation of the particular component they were testing. For validating the application we regularly met up with our supervisor and showed demos and took his feedback.

5.1 Testing the UI

User interface is a big part of our application so we had to make sure every aspect of it was working correctly. To test our UI we left Willow open in the back and left it to work while we were browsing through the web and after a while we checked to see if the result was correct and there was an easy to understand graph that represented our browsing session. For each new feature we applied this technique. This continuous testing significantly helped us to detect and solve UI bugs fast, and design a better and easy to use interface.

We also tested Willow using the input from our friends and families. Since Willow is fairly easy to upload and check on Google Chrome Extensions page we made them upload and set up Willow for their own browsers and they used Willow during their browsing sessions and then reported back to us various bugs they found along with their opinion on user experience design and flow. They also reported to us any inconsistency they felt in the system such as titles not loading or synchronization issues. We realized some bugs thanks to them. We got some really useful advice from them considering none of them were from software engineering backgrounds and they were just regular users.

Another crucial part of testing Willow was to see if Willow unknowingly changed the content of any external web page it ran on. To do that we tried to diversify the way we use Willow while testing as much as possible meaning that we did not just test Willow using more commonly used pages while browsing such as Wikipedia, popular news sites or social media we also tested Willow with as many different options as possible. We are aware of the magnitude of the internet resources so the best way to conduct these tests was to leave Willow on in the background while we were browsing deep on the internet whatever the subject is. This helped us diversify our test range and see some edge cases where Willow might possibly modify the page it is used on.

5.2 Performance and Communication Testing

We performed timer testing and tested the performance metrics of Willow, for example we made sure to have a fast response time even though this is not something that depends only on us. This depends on the connection quality of the user, the rate Chrome Web API sends the information about users' tabs. However we still tested and the response time of Willow is under 45 ms in ordinary situations.

We also extensively use Chrome Local Storage for communication between our packages so we had to make sure this communication was working correctly. To do that while implementing system communication we sent various test messages and ensured that they were received correctly on the other side. In addition to this we overloaded the system with very big messages to test where the limit of Chrome Local Storage messages lay and we made sure to implement communication in a way that it was fast, simple and did not violate the limit.

5.3 Continuous Integration

All things considered since we had a continuous integration development pipeline, we were able to continuously test our application as we were developing and this helped us locate our mistakes very early on and we were able to handle them before there were

any big and irreversible consequences of these mistakes. We tested components of our application as isolated as possible and followed a build up approach where we did not continue implementing that component until we were sure it was working as intended.

6. Maintenance Plan and Details

Willow does not use any remote servers for storing user data so in terms of maintenance it is very easy to maintain because core functionalities work on the user's local computer. In this situation we do not pay any extra fees for keeping Willow up except for 5 dollars for publishing it on the Chrome Web Store and except for that there are no monthly fees or anything for Willow.

Currently Willow is implemented using Manifest V2 and even though V2 is still supported, in the future Google will migrate to Manifest V3 and when the time comes we will update Willow accordingly and also migrate our application to Manifest V3.

7. Other Project Elements

7.1.Consideration of Various Factors in Engineering Design

Willow is a tool designed to enhance people's internet browsing experience. Hence, it does not affect and is not affected by public health, public safety and public welfare in any way at all. Social factors are also irrelevant for the design of Willow.

Global Factors:
Willow is imagined as a software to be used by people all around the world. However, people speak many different languages all around the globe. Considering that most people that browse the internet speak English, having the interface in English will allow people from different countries to access the software.

Cultural Factors:
Different cultures have different understandings of privacy. The European Union has very strict rules about data protection. On the other hand, the Chinese are used to a much more invasive mode of surveillance. In designing Willow, the privacy boundaries of different cultures need to be respected.

7.2.Ethics and Professional Responsibilities

Willow follows ACM Code of Ethics and Professional Conduct [5] and IEEE Code of Ethics [6].
In projects where the history of a user is managed, privacy is an important value. However, Willow does not store its users' data on a remote server. Users' data are stored locally. Additionally, we do not share the data with any other third-party without informing the users.
The sources (libraries, APIs, etc.) we used are all used according to their licences and the correct reference to them were done. And lastly, we gave importance to the accuracy of used data in order to not deceive our users. Thus, the accuracy of visualization of the were given great importance [6].

7.3.Judgements and Impacts to Various Contexts

As discussed in section 7.1 Willow takes into account various factors such as global and cultural. Also we tried to publish Willow before the deadline to get useful feedback from users regarding the user experience and any other advice. In this context Willow will be used by people who will shape how it feels best to them since it's all about making browsing easy for its users.

7.4 Teamwork Details

7.4.1 Contributing and functioning effectively on the team

As the Willow team we believe that the road to proper teamwork is through healthy communication. To do that during this year we have met at least once a week and discuss our progress that week. In these meetings new tasks are assigned to every group member in a fair manner and important decisions about design, implementation or feasibility are made. We do every task on a schedule. We gave proper deadlines for each task and tried to keep up with those deadlines and this ensured that we kept proper time tables and never got behind schedule. Also, all of our team members have previously worked on various course projects together. We believe that our familiarity with each other's strengths, weaknesses, working habits and communication styles helped us maintain a respectful and comfortable work environment and kept us on track.

Another caution we took to ensure proper and effective teamwork is that we did not do every task individually. For example if it is a very big task we divided the task in parts and we also divided the group and then started doing it each smaller task for each group. When it's time to combine the mini tasks we got together to review and discuss

the tasks done by each group. This approach was advantageous because it was very efficient and also gave everyone in the group to review the task as a whole in the end.

7.4.2 Helping creating a collaborative and inclusive environment

Our primary aim as Team Willow was to follow the software development life cycle properly. In order to achieve this we analyzed and designed our project comprehensively. We tried to create a prototype before CSFair to allow us to test our product on the field and make continuous improvements in the necessary areas. After making sure our product is shaped to meet the initially planned functionalities and the demands of the users, we moved on to implementing our extended features. In the future we plan on testing our extended features with users and receiving feedback in the same way with the prototype of the core product.

7.4.3 Taking lead role and sharing leadership on the team

According to our plan we have 9 work packages: Analysis Report, High-Level Design Report, First Prototype, Marketing, Low-Level Design Report, Second Prototype, Marketing, Final Implementation Changes, Final Report.

Table 1: List of work packages

WP#	Work package title	Leader	Members involved
WP0	Requirements Specification	Sezin Zeydan	All members
WP1	Analysis Report	Tuana Türkmen	All members
WP2	High-Level Design Report	Cem Cebeci	All members
WP3	First Prototype	Efe Dağdemir	All members
WP4	Marketing	Can Cebeci	Sezin Zeydan, Efe Dağdemir

WP5	Low-Level Design Report	Sezin Zeydan	All members
WP6	Second Prototype	Can Cebeci	All members
WP7	Marketing	Tuana Türkmen	Cem Cebeci
WP8	Final Implementation Changes	Cem Cebeci	All members
WP9	Final Report	Efe Dağdemir	All members

Table 2: Explanation of the work packages

WP 0: *Requirements Specification*			
Start date: *Sep 16, 2020* **End date:** *Oct 12, 2020*			
Leader:	*Sezin Zeydan*	**Members involved:**	*All members (Tuana Türkmen, Efe Dağdemir, Cem Cebeci, Can Cebeci)*
Objectives: *Name and briefly describe the project. Identify initial project requirements, constraints and professional & ethical responsibilities.*			
Tasks: ***Task 0.1 Functional Requirements*** **:** Identifying the functional requirements. ***Task 0.2 Constraints*****:** *Determining the constraints.*			

Deliverables

D0.1: Project Specification Report

WP 1: *Analysis Report*

Start date: *Oct 12, 2020* **End date:** *Nov, 21 2020*

Leader:	*Tuana Türkmen*	**Members involved:**	*All members (Efe Dağdemir, Sezin Zeydan, Cem Cebeci, Can Cebeci)*

Objectives: *Having a comprehensive analysis of the project so that the design and implementation of the project goes as smoothly as possible.*

Tasks:

***Task 1.1 Research about the current system*:** *The purpose is to know the current system. This is important for the design and implementation of the project because we would know what is already available, what should be extra implemented and what can be changed in the current system.*

Task 1.2 Analyzing the proposed system: *The purpose is to decide on the functionalities and the models of the system.*

Task 1.3 Considering social conditions: *The purpose is to decide on risks, project plan, responsibilities, etc.*

Deliverables

D1.1: *Analysis Report*

WP2: *High-Level Design Report*

Start date: *Nov, 21 2020* **End date:** Dec 21, 2020			
Leader:	*Cem Cebeci*	**Members involved:**	*All members (Efe Dağdemir, Sezin Zeydan, Tuana Türkmen, Can Cebeci)*
Objectives: *Having a comprehensive design of the project so that the implementation of the project goes as smoothly as possible.*			
Tasks: ***Task 2.1 Architecture***: *The purpose is to decide on which architecture we will use while implementing our project.* ***Task 2.2 System Models*****:** *The purpose is to have more precise system models that we will use during our implementation.*			
Deliverables ***D2.1:*** *High-Level Design Report*			
WP3: *First Prototype*			
Start date: Dec 21, 2020 **End date:** Before the end of fall semester			
Leader:	*Efe Dağdemir*	**Members involved:**	*All members (Sezin Zeydan, Tuana Türkmen, Can Cebeci, Cem Cebeci)*
Objectives: *Implementing a presentable prototype with basic functionality.*			

Tasks:

Task 3.1 Implementation: *Implementing the project so that it has basic functionality and can be tested by presenting to ısers.*

Deliverables

D3.1: *A basic functioning prototype*

WP4: *Marketing*

Start date: After implementation of the prototype **End date:** Before the demo on the fall semester

Leader:	*Can Cebeci*	**Members involved:**	*Sezin Zeydan, Efe Dağdemir*

Objectives: *Marketing strategies for populating the product between users.*

Task 4.1 Social Media: *Using social media platforms to introduce users with the product.*

WP5: *Low-Level Design Report*

Start date: *1st week of the spring semester* **End date:** *3rd week of the spring semester*

Leader:	*Sezin Zeydan*	**Members involved:**	*All members (Efe Dağdemir, Tuana Türkmen, Can Cebeci, Cem Cebeci)*
Objectives: *Having a comprehensive low-level design of the project so that the implementation of the project goes as smoothly as possible.*			
Tasks: ***Task 5.1 Architecture***: *The purpose is to decide on which architecture we will use while implementing our project.* ***Task 5.2 System Models*****:** *The purpose is to have more precise system models that we will use during our implementation.*			
Deliverables ***D5.1:*** *Low-Level Design Report*			
WP6: *Second Prototype*			
Start date: 3rd week of the spring semester **End date:** Before the Final Report deadline			
Leader:	*Can Cebeci*	**Members involved:**	*All members (Sezin Zeydan, Tuana Türkmen, Efe Dağdemir, Cem Cebeci)*
Objectives: Implementing a second prototype according to the user feedback we received which optimistically will have more features.			
Tasks:			

Task 6.1 Changes: *The purpose is to change the first prototype according to the feedback received.*

Task 6.1 Extended Features: *The purpose is to add the extended features to the implementation.*

Deliverables

D6.1: Second Prototype

WP7: *Marketing*

Start date: After implementation of the second prototype **End date:** Before the final changes

Leader:	*Tuana Türkmen*	**Members involved:**	*Cem Cebeci*

Objectives: *Marketing strategies for populating the product between users.*

Task 7.1 Social Media: *Using social media platforms to introduce users with the product.*

WP8: Final Implementation Changes

Start date: After implementation and marketing of the second prototype **End date:** Before CSFair

Leader:	*Cem Cebeci*	**Members involved:**	*All members (Efe Dağdemir, Sezin Zeydan, Tuana Türkmen, Can Cebeci)*
Objectives: *Having the final implementation of the project.*			
Task 8.1 Finalizing: *Finalizing the project implementation according to the latest user feedback and having an extensive testing procedure.*			
Deliverables ***D8.1:*** *Final Product*			
WP9: *Final Report*			
Leader:	*Efe Dağdemir*	**Members involved:**	*All members (Sezin Zeydan, Tuana Türkmen, Can Cebeci, Cem Cebeci)*
Objectives: *Provide a comprehensive final report of the project that gives complete information about the system.*			
Tasks: ***Task 9.1 Architecture***: Final architecture of the system. ***Task 9.2 Maintenance Plan:*** Provide a maintenance plan for the system. ***Task 9.3 Social Impact:*** Analyse social impact of the system.			
Deliverables ***D9.1:*** *Final Report*			

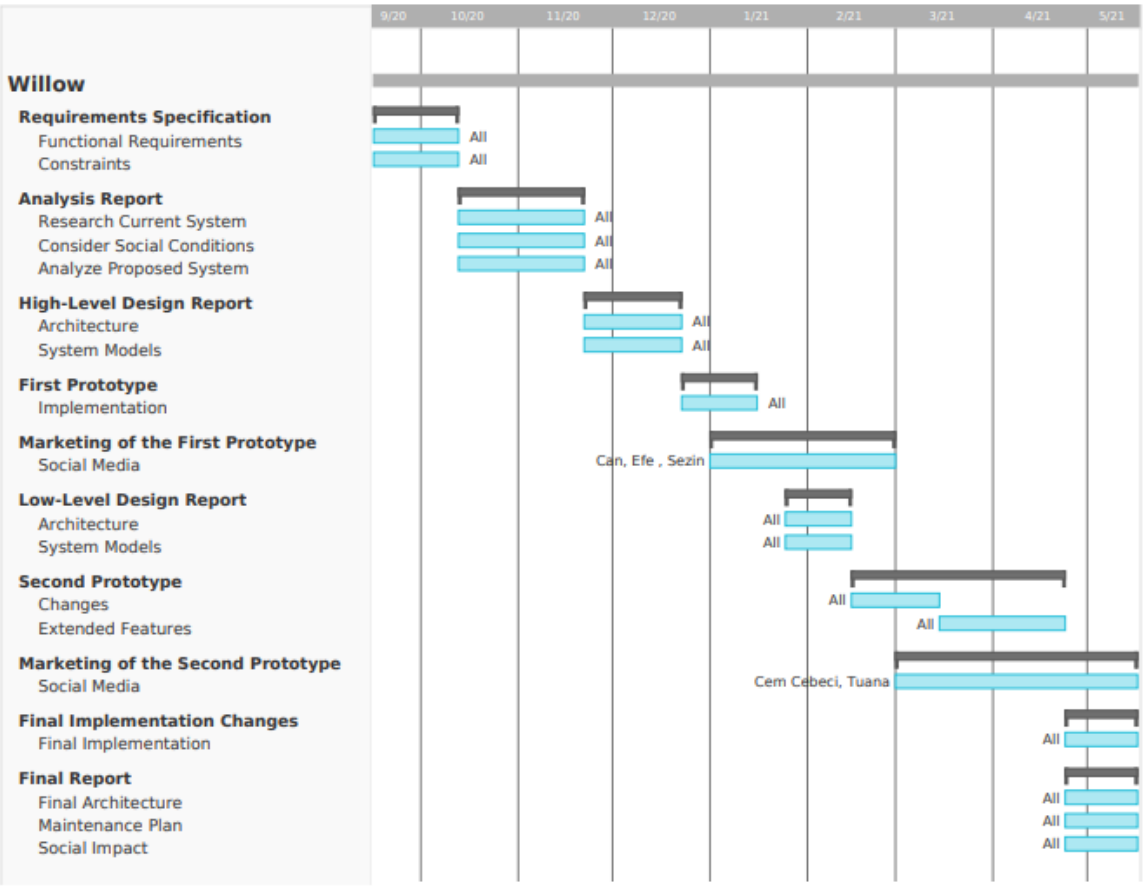

Figure 9: Gantt chart of the project plan

7.4.4 Meeting objectives

The objectives promised to deliver in the previous reports have all been accomplished. Willow is now a successfully working browser assistant that provides its users an easy to use interface that visualizes their browsing session and also gives its users the ability to see and manage their old sessions. Willow also implements extra features such as applying the PageRank algorithm to show its users a more detailed graph of their browsing session. Willow is also currently under consideration from Google for publishing on Chrome Store.

7.5 New Knowledge Acquired and Applied

While implementing Willow we tried to utilize each member's strongest sides. Thus the learning process was quite smooth. To acquire new knowledge we mostly used internet sources such as youtube, stackoverflow etc. We also tried to get really familiar with

Cytoscape and Javascript as well as the architectural styles of Chrome extensions and the utilities of the Chrome extension API.

8. Conclusion and Future Work

In the end we succeeded in creating an application that visualizes its users' browsing session and gives them a chance to reorganize their tabs in a more easier way. Willow is up and working and currently uploaded to Chrome Web Store and will be published officially when Google approves. Willow is free and open-source, all the code written for Willow is available on Github. We are happy with what we created and learned a lot in the process. However we are still very open to the idea of improving it so any feedback from our users will be taken into account.

As mentioned in the maintenance section all in all Willow is very easy to maintain however we do plan on migrating our application from Chrome extension Manifest V2 to Manifest V3, which was introduced by Google a couple months into the development period of Willow, when the need arises.

9. Glossary

- **Session:** Sessions are continuous instances of browsing activity. A new session starts every time the user launches the Chrome app. The session is terminated upon exit. Sessions can be saved and restored, allowing them to last across multiple runs of Chrome.
- **Session Graph:** A visual graph structure that shows the relationships between nodes. The session graph contains an edge $u \rightarrow v$ if and only if the first instance of access to the website associated with node v was through a link contained in the website associated with node u.
- **Node:** A vertex on the session graph which represents a web page visited within the session.
- **Open Node:** A node whose associated page is currently open in a tab.
- **Closed Node:** A node whose associated page was visited within the session and is not currently open in a tab. The tab that once contained the webpage need does not have to be closed, it may simply be navigated to another webpage.
- **Active Tab:** The tab whose content is currently displayed. At any time, the browser contains exactly one active tab and any number (possibly zero) of inactive tabs.
- **Active Node:** The node that is associated with the webpage contained in the active tab. Trivially, there is exactly one active node at all times and the active node is an open node.

- **Willow Overlay:** The hideable and extendable panel that is added to Chrome's UI when Willow is installed. The overlay contains the session graph and is the main medium of interaction with Willow

10. References

[1] *What is Unified Modeling Language (UML)?* [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/. [Accessed: 07-Feb-2021].

[2] “Content scripts,” *Chrome Developers*. [Online]. Available: https://developer.chrome.com/docs/extensions/mv2/content_scripts/. [Accessed: 07-Feb-2021].

[3] M. Franz, “Cytoscape.js: Graph theory (network) library for visualisation and analysis,” *Cytoscape.js*. [Online]. Available: https://js.cytoscape.org/. [Accessed: 07-Feb-2021].

[4] “Manage events with background scripts,” *Chrome Developers*. [Online]. Available: https://developer.chrome.com/docs/extensions/mv2/background_pages/. [Accessed: 07-Feb-2021].

[5] “The Code affirms an obligation of computing professionals to use their skills for the benefit of society.,” Code of Ethics. [Online]. Available: https://www.acm.org/code-of-ethics. [Accessed: 20-Nov-2020].

[6] “IEEE Governing Documents,” IEEE. [Online]. Available: https://www.ieee.org/about/corporate/governance/index.html. [Accessed: 20-Nov-2020].

Bilkent University

Department of Computer Engineering

Senior Design Project

Willow: Graph-Based Browsing

User Manual

Efe Dağdemir, Tuana Türkmen, Sezin Zeydan, Can Cebeci, Cem Cebeci

Supervisor: Uğur Doğrusöz

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Table of Contents

1. Getting Started

1.1. What is Willow?

Willow is a browser extension that could help to ease the process of browsing via understandable and intuitive visual navigation and session management features. Willow aims to enable users to efficiently visualize and manage their browsing sessions in a graph-based, interactive structure.

1.2. System Requirements

Willow only requires the Google Chrome web browser to work. Since browser extensions are not available on mobile platforms, Willow can only be used on computers. The following system requirements are needed for Google Chrome [1]:

Windows

- Windows 7, Windows 8, Windows 8.1, Windows 10 or later
- An Intel Pentium 4 processor or later that's SSE3 capable

Mac

- OS X El Capitan 10.11 or later

Linux

- 64-bit Ubuntu 14.04+, Debian 8+, openSUSE 13.3+, or Fedora Linux 24+
- An Intel Pentium 4 processor or later that's SSE3 capable

1.3. Download & Installation

There are two different ways users can add Willow to their browsers. The first is through Chrome Web Store and the second is by adding the extension manually to Chrome.

Chrome Web Store

All of the necessary steps are taken to publish Willow in the Chrome Web Store and Willow is currently going through a review process by Google before it will be available for download on the Chrome Web Store. Installing Willow via Chrome Web Store can be achieved with the following steps:

1. Open the Chrome Web Store.
2. Search for “Willow: Graph-Based Browsing” and select it.
3. Click “Add to Chrome”.
4. Click “Add extension” to allow for permissions.

Click the Willow icon to the right of the address bar to start using Willow.

Manually installing Willow
Manually installing Willow requires the extension to be downloaded from the GitHub repository.

1. After the project is downloaded/cloned from this repository, Chrome Extension Management page should be opened. This can be achieved by navigating to *chrome://extensions* or by following the steps *Chrome menu -> More Tools -> Extensions* or by following the steps *Extensions menu button -> Manage Extensions.*
2. Then the “Developer mode” option should be enabled (toggle switch on the top right).
3. Next, “Load unpacked” button should be clicked and the extension directory (willow/demo/) should be chosen. After the above steps, the extension will be loaded.

Click the Willow icon to the right of the address bar to start using Willow.

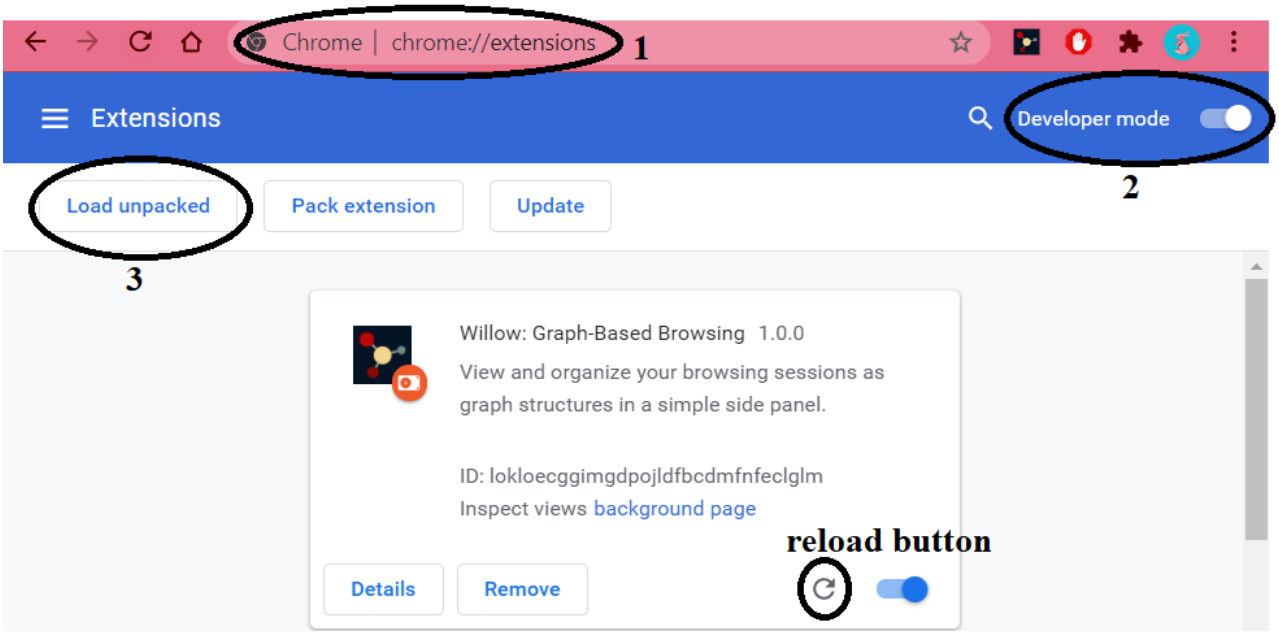

Figure 1: Numbered steps for manual installation

2. Using Willow

2.1. Graph Operations

Create a node on the graph
Since nodes correspond to the web pages that are visited, whenever a web page is visited a node will be formed on the graph immediately.

Create an edge on the graph

Edges are formed between nodes on the graph whenever a new web page/link is visited from a page that was visited before. Edges represent the connection between these web pages. And they are formed in a way such that it is a directed arrow that starts from the source node and goes towards the target node.

Discovering edges

Discovering edges are created when a first time link between two nodes are formed. Meaning that the corresponding web page is discovered for the first time. They are shown with continuous arrows.

Non-discovering edges

Non-discovering edges simply form whenever a node is visited but this visit is not the first time an edge was formed between these two nodes. They are shown with dashed lines.

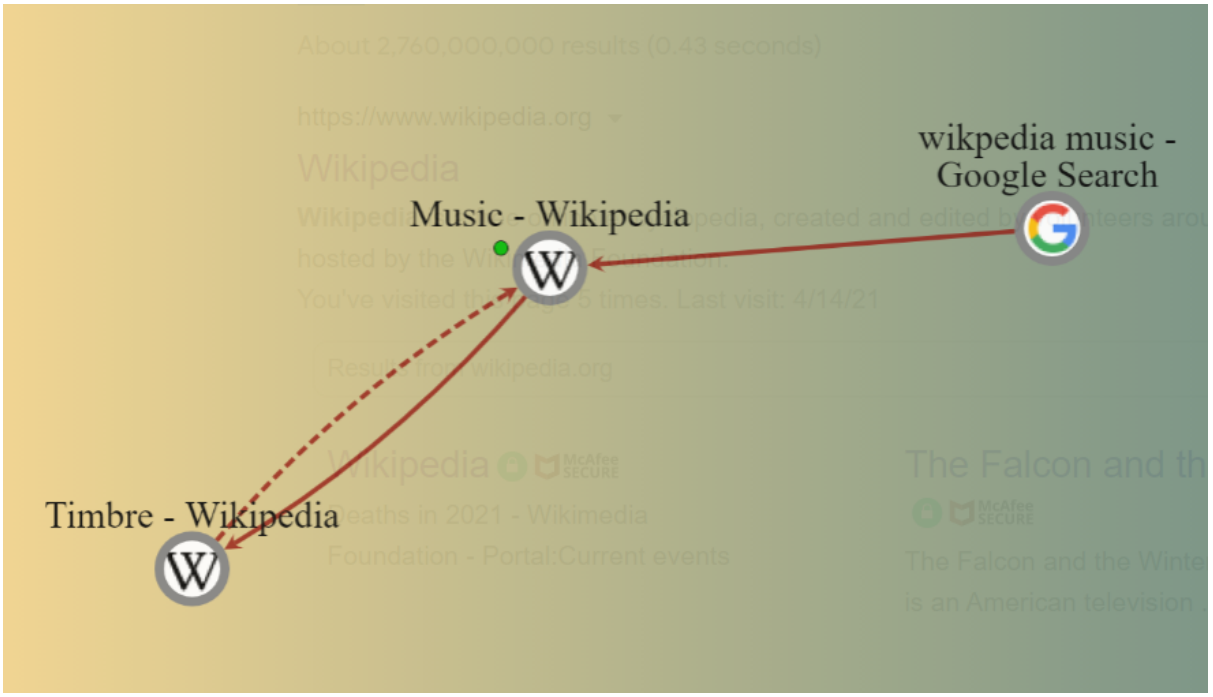

Figure 2: Graph with 2 discovering edges and a non-discovering edge

Remove a node from the graph

A node can be removed from the graph if that node is right clicked and the Remove node option is selected from the formed context menu. With this operation, the connected edges are removed alongside the chosen node.

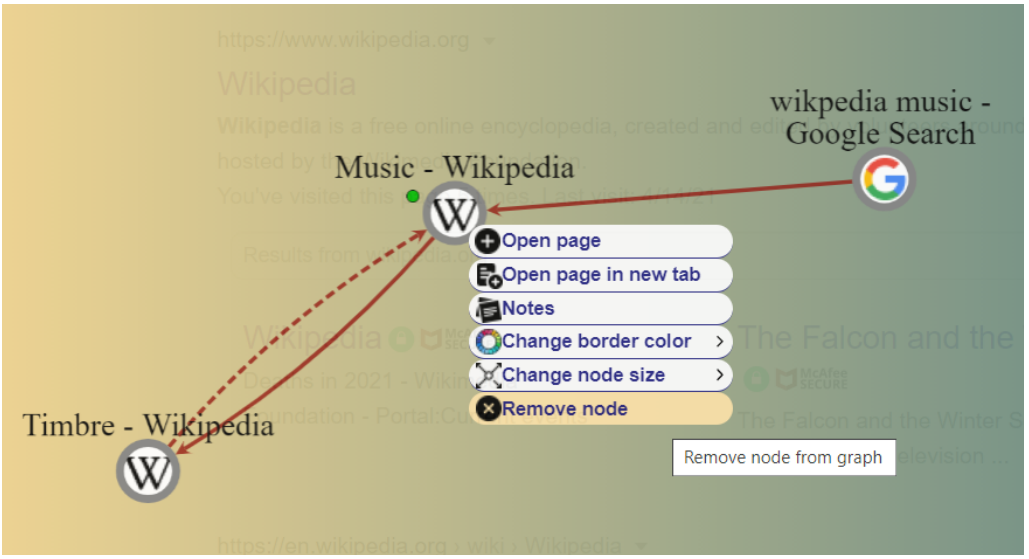

Figure 3: Remove node option in the context menu

Fit the graph

In order to make the graph fit to the extension overlay, the context menu option that is created when the graph background is right clicked should be opened and the Fit graph option should be selected. If this operation is done, some pan and zoom operations are performed to make the graph fit to the overlay.

Center the graph

The context menu that is formed when the background is right clicked presents the option to center the graph. If this operation is chosen, then the graph will be centered according to the overlay.

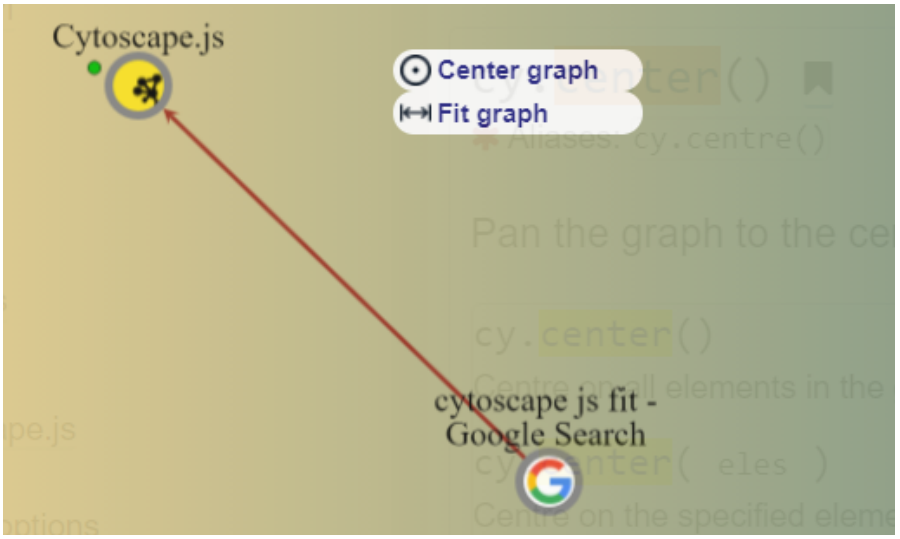

Figure 4: Center graph and fit graph options in the context menu

Export/ Import graph

The options to import or export the graph are present in the Settings Menu (see section 2.6). If the export option is selected, a new window opens which asks the user the basic saving options like confirming the name or the saving place of the graph.

If the import option is selected, again a new window opens that asks the user to choose from the previously exported (saved) graph sessions. If one of the sessions is selected, then the current graph will turn into that session's graph and the tabs of that session will be opened.

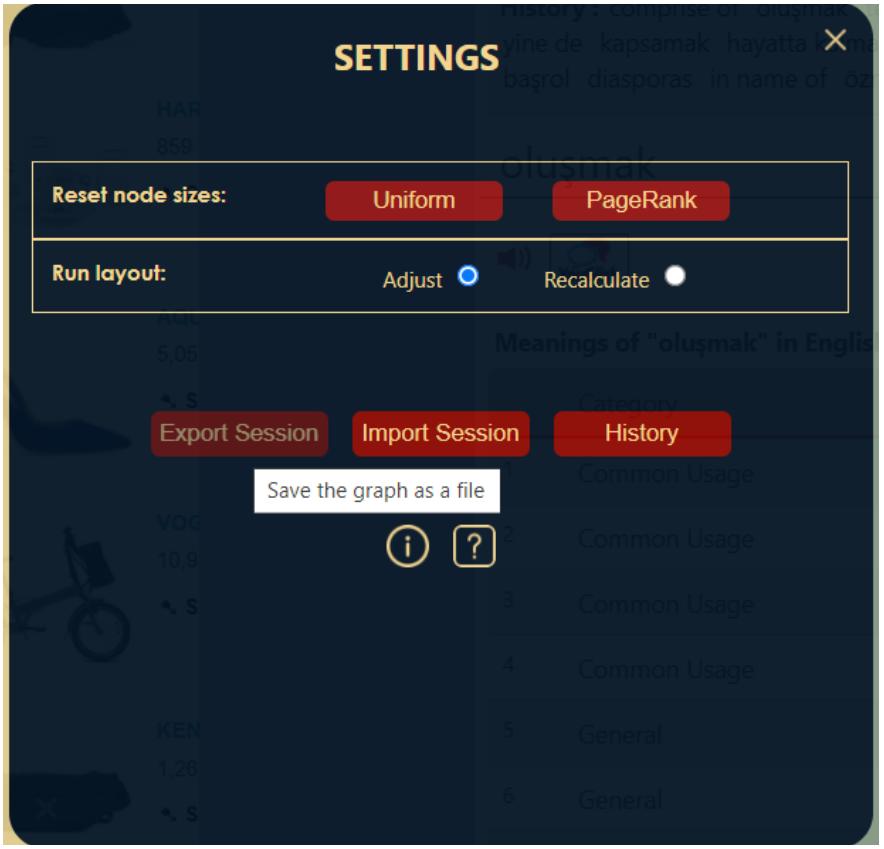

Figure 5: Export session button in the settings menu

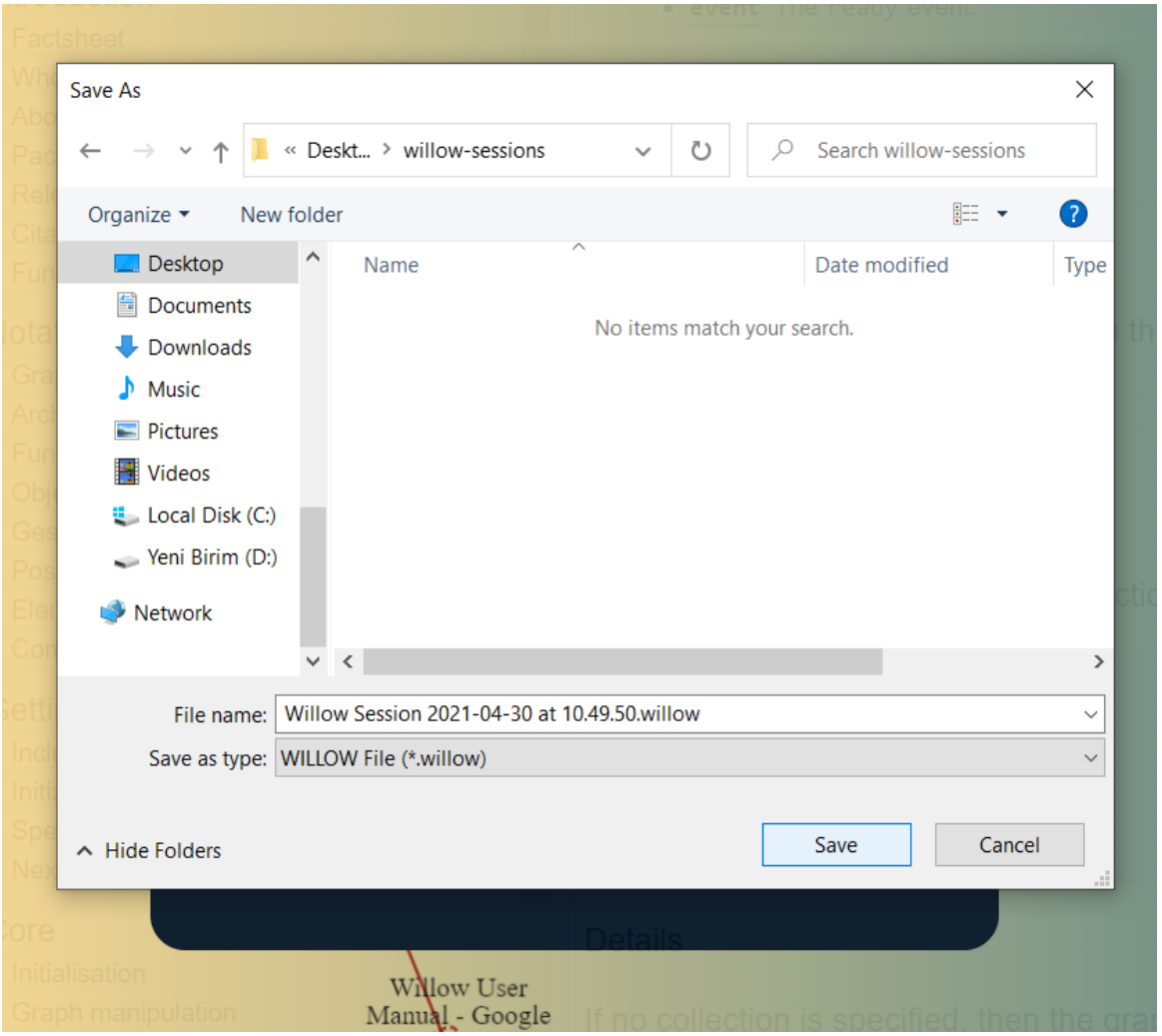

Figure 6: Save As menu that is opened after "Export Session" button is clicked

2.2. Node Manipulations

Reposition nodes

Nodes can be repositioned individually or as a group. Nodes need to be selected before they can be repositioned. Selected nodes appear with a black border.

Reposition individual nodes

If a node is clicked on and dragged, that node will be carried along the way of the drag and the node will be placed to the place where the click of the mouse is lifted.

Reposition a group of nodes

A group of nodes can be selected in two ways. Firstly, they can be chosen via holding down the Shift Key and selecting the nodes with the mouse. Or secondly if a node is clicked down for a while, that node and its connected nodes will be selected. After selecting the nodes, they can be repositioned by dragging any of the selected nodes.

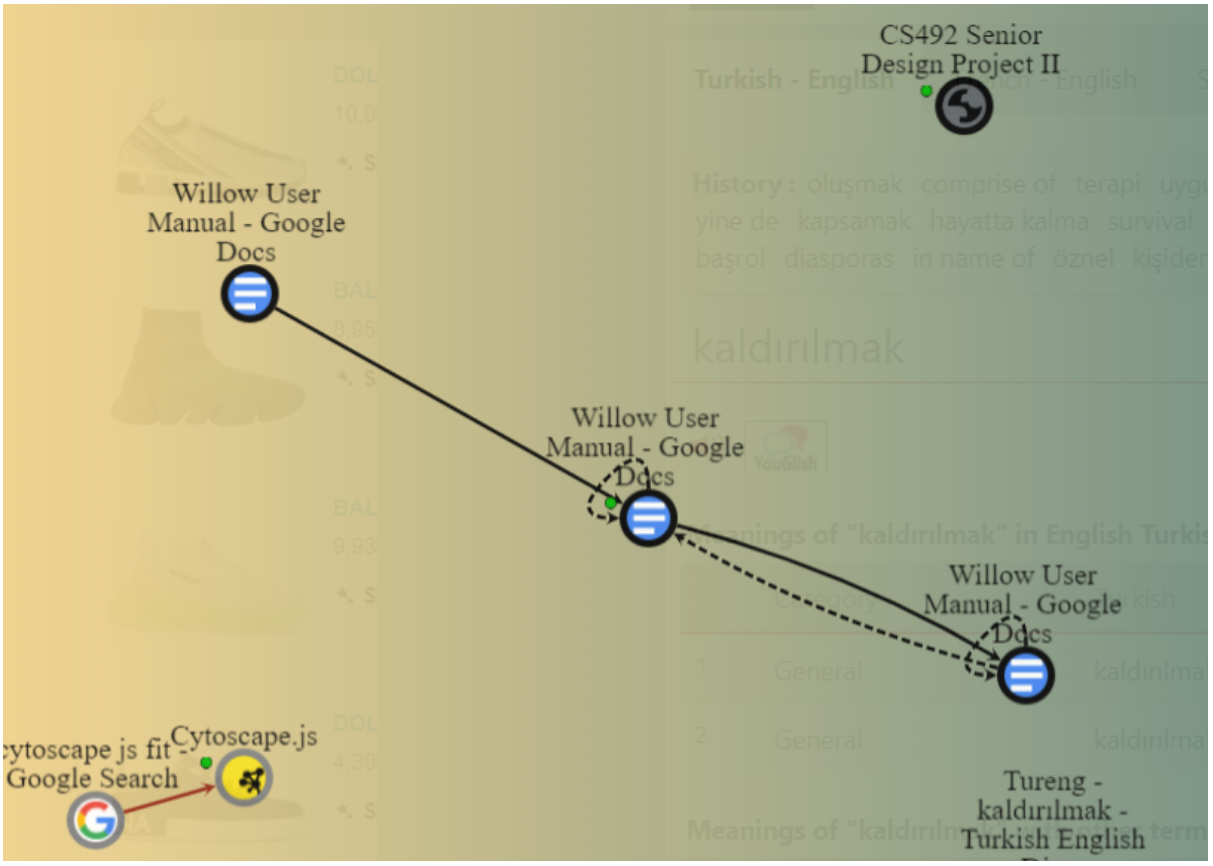

Figure 7: A group of selected nodes (shown with black borders)

Add notes to a node and see notes of a node

The option to add notes to and/or see existing notes of a node can be found by right clicking on the node and selecting the Notes option. When this option is selected, a pop up opens that shows the current notes if there is any. At the same time, this pop up is where new notes can be added from. If a node has notes, then it is indicated by the notes icon on the button right of the node.

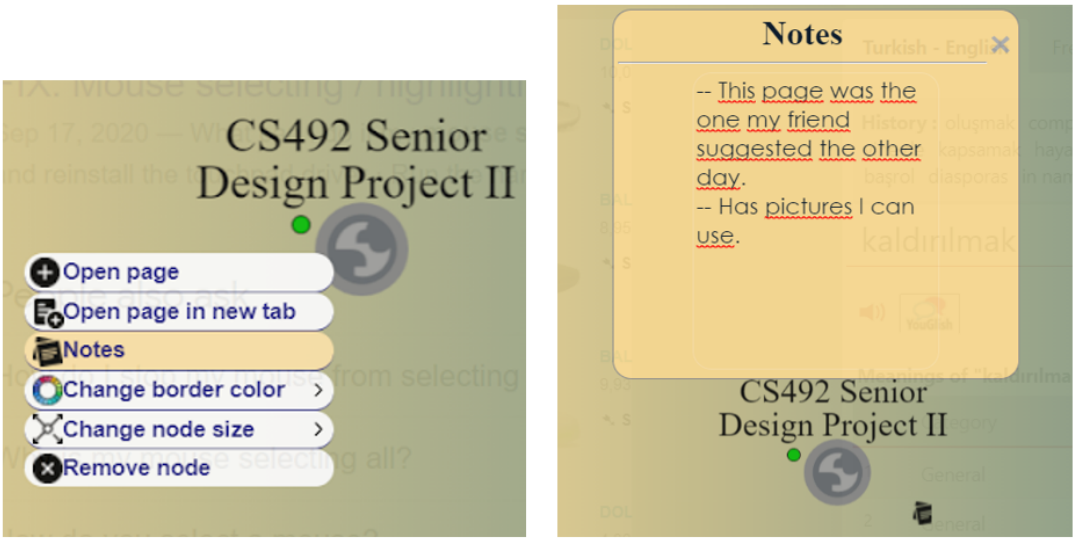

Figure 8: Notes option and its input interface

Change border color

Border color of a node can be changed to 7 different colors: red, pink, yellow, green, purple, blue and black. To change the border color to one of the 7 colors available, right click on a node and hover over the “Change border color” option and select the desired color.

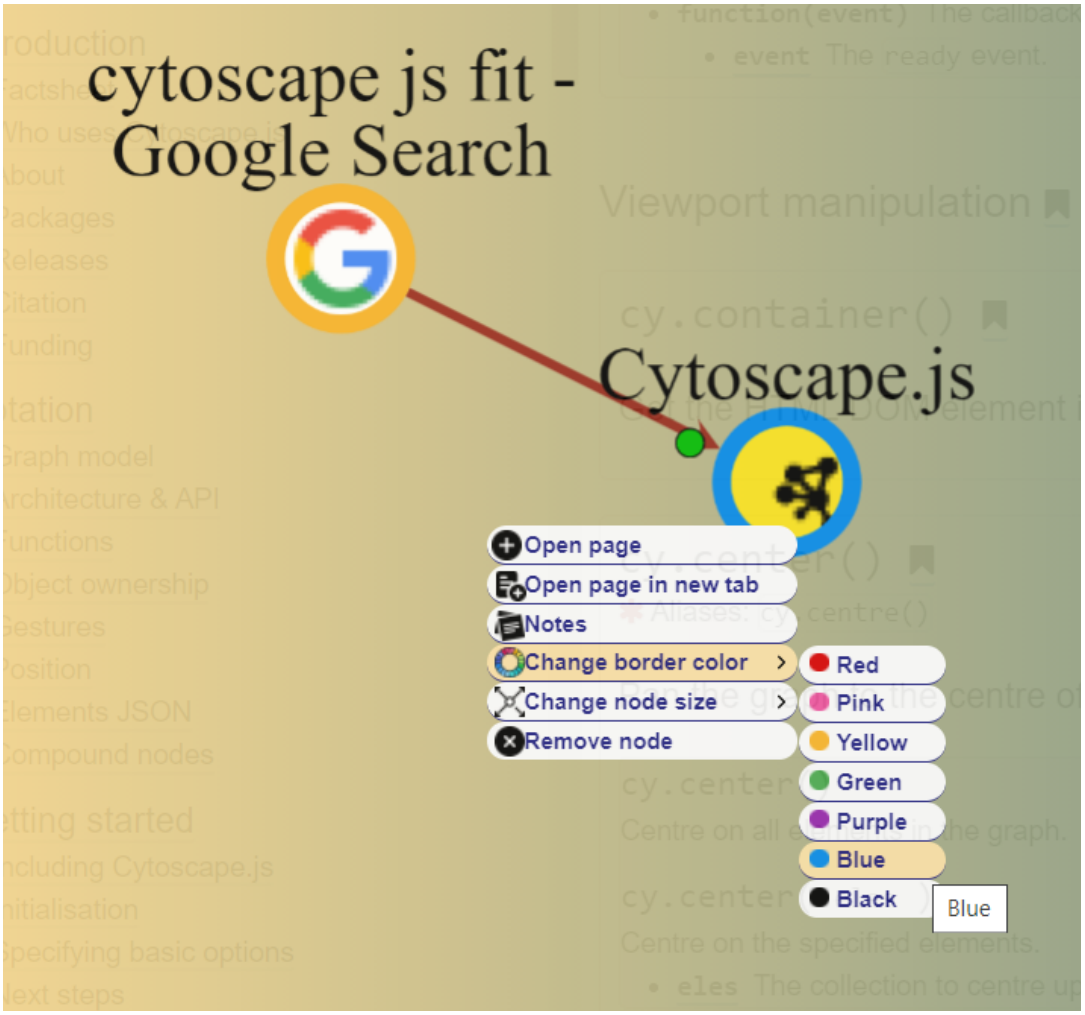

Figure 9: Change border color options in the context menu

Change size

Node sizes can be changed individually or entirely. Individual nodes can be resized more freely while the entirety of nodes can be resized in one of two ways.

Change the size of an individual node

Individual nodes can be resized with a preset of 3 sizes (small, medium, large) or in an incremental fashion.

- **Preset Sizes:** Right click on a node and hover over “Change node size” to reveal the preset sizes and select the size you desire.

- **Incremental sizing:** Right click on a node and hover over “Change node size” to reveal the “Arrange size” option. Hover over “Arrange size” to reveal the incremental sizing options and select to increase or decrease the size of the node.

Change the size of all nodes
To change the size of all nodes, open the Settings menu to find and select one of the two “Node sizes” options in the first row.

- **Uniform sizing:** Sets all nodes to the default size.
- **PageRank sizing:** Set node sizes according to the PageRank algorithm. PageRank sizing sizes nodes based on their topological importance. How many nodes point to a node and how important their pointees in turn are determines the topological importance of a node, a bigger node means a more important node. Keep in mind that this option overrides the previously set sizes of all nodes.

2.3. Graph Layouts
Graph layouts determine how a graph is visualized.

Run a layout
A layout can be run on the graph anytime by the Run Layout button on the top of the overlay.

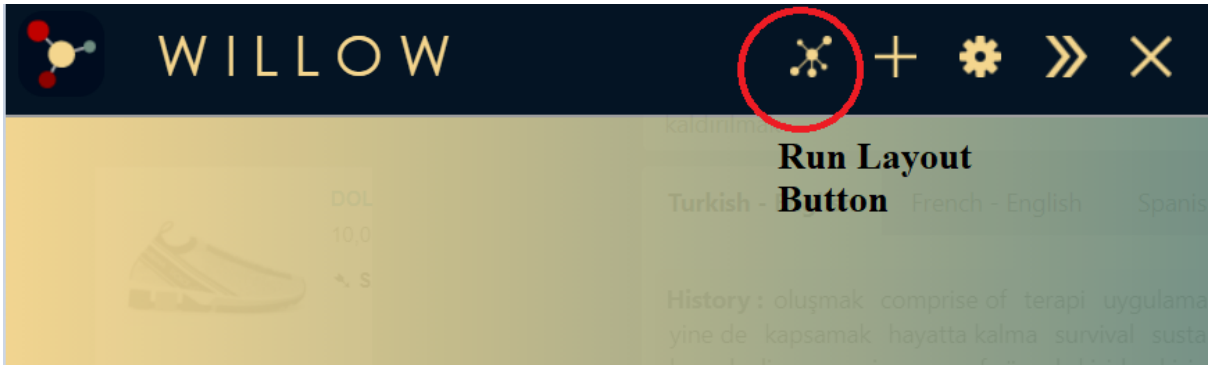

Figure 10: Run Layout button in the panel header

Adjusting vs. Recalculating
The kind of the layout that will be run can be chosen from the Settings Menu. The *Adjust* option reruns the layout algorithm without making drastic changes to the current situation of the graph. The *Recalculate* option reruns the algorithm in a randomized manner.

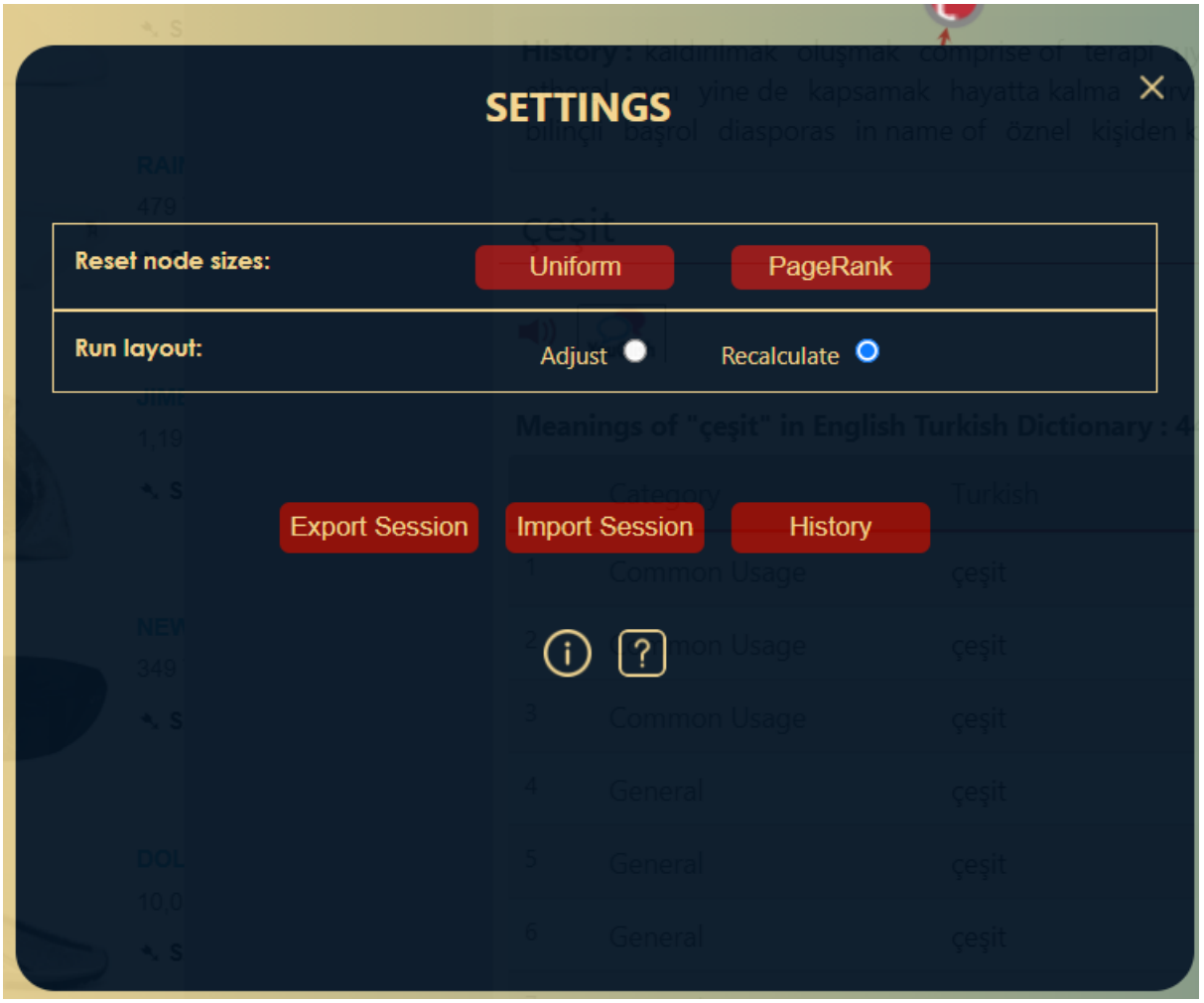

Figure 11: Layout options in the settings menu

2.4. Session Operations

Start a new session

A new session can be started via the New Session button on top of the overlay. If this button is clicked, the current session will be closed alongside the open tabs and a new session will start.

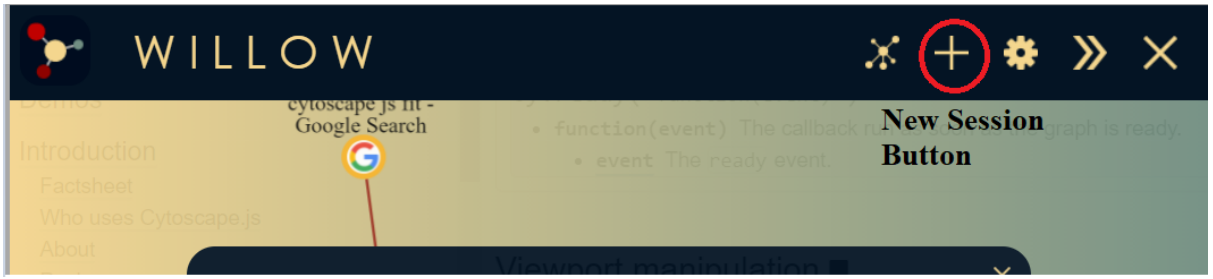

Figure 12: New Session button in the panel header

History Page
History page is where users can see the history of their sessions.

From this page, sessions can be restored, renamed or deleted. These operations are made possible via the buttons on the bottom right of the page.

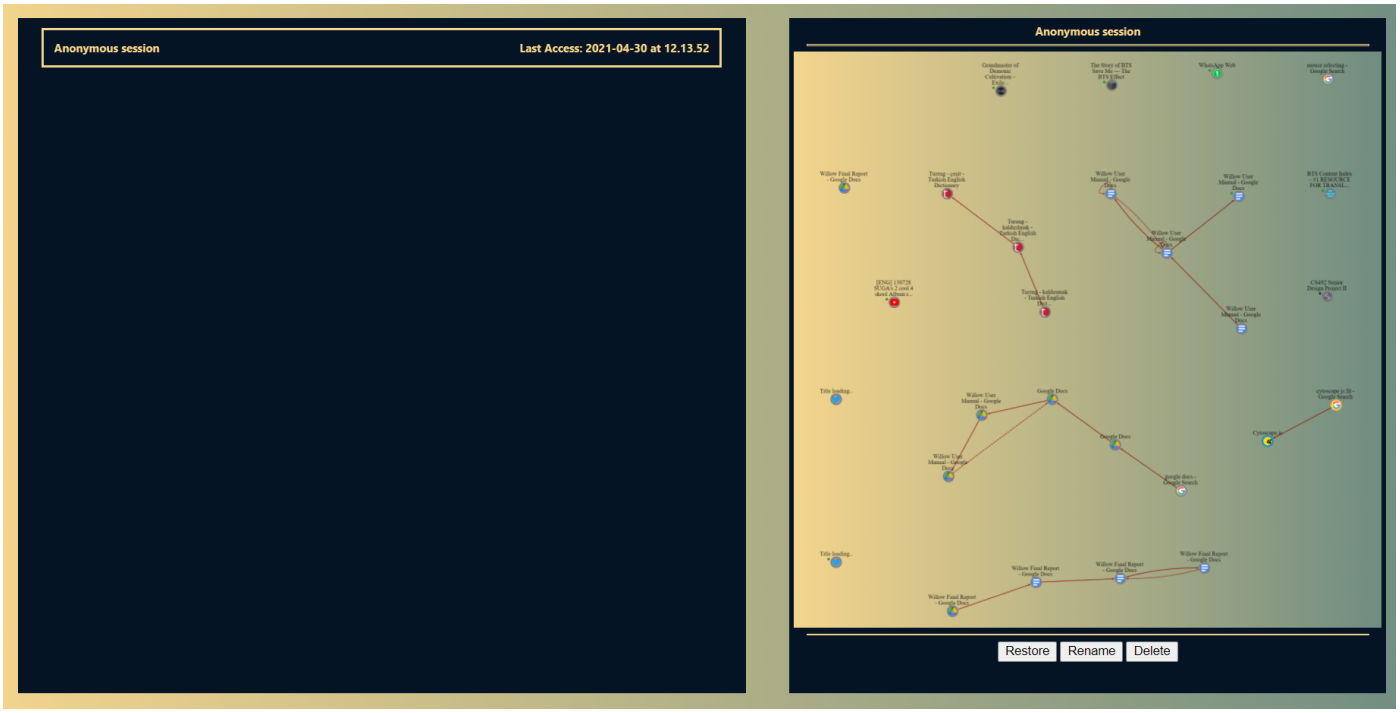

Figure 13: Willow's History Page

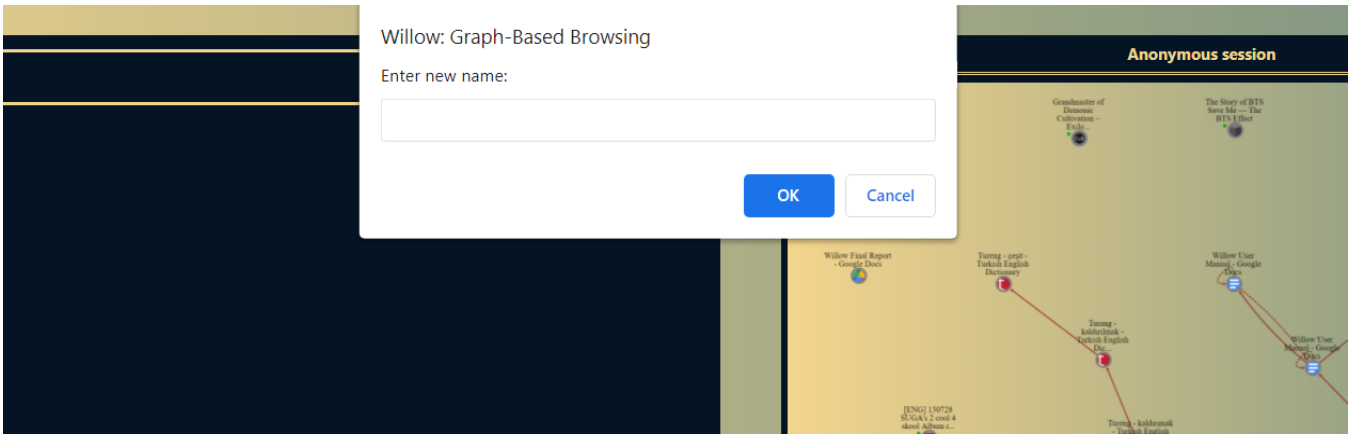

Figure 14: Input interface for renaming a session

2.5. Panel Operations

Open panel
Opening the Willow side panel (i.e. the overlay) can be achieved by clicking the Willow icon to the right of the address bar.

Close panel
There are two ways to close the side panel.

- **Willow icon:** Identical to opening the side panel, clicking the Willow icon in the extensions section to the right of the address bar will close the panel.
- **Close button:** Clicking the "*Close*" button on the panel header will close the panel.

Figure 15: Close button in the panel header

Dock/Undock panel
The Willow side panel can be docked and undocked. Docking/undocking can be achieved by clicking the "Undock" and "Dock" icons. Undock and Dock buttons will replace each other depending on whether the panel is docked or undocked.

- **Docked:** While docked the panel stays fixed to the left side of the screen.
- **Undocked:** While undocked the panel can be moved freely around the screen.

Figure 16: Undock button in the panel header

Resize panel
The side panel is resizeable on all sides. While docked, the panel can only be resized from the right, otherwise the panel is resizeable on all sides.

2.6. Miscellaneous Features

Settings Menu

Settings Menu contains many of the features that are talked about in this manual. It opens/closes via the Settings button on top of the overlay. Information Page, How To Page and History are all opened from this menu.

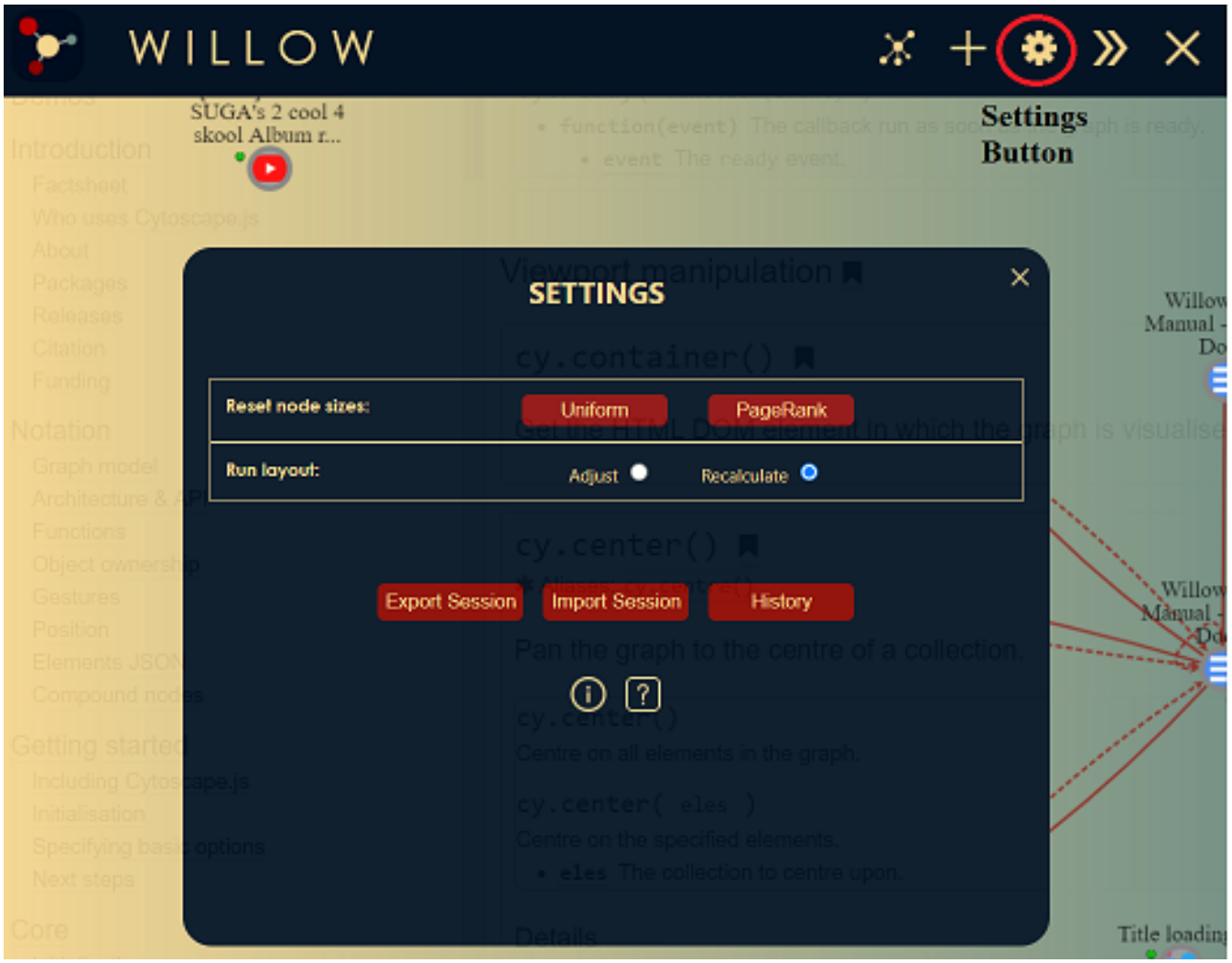

Figure 17: Settings button in the panel header and the Settings Menu

How To Page

How To Page is an informative page that describes the usage of the extension. It is opened by the button that is displayed as a question mark in the Settings Menu.

Figure 18: How To Page

Information Page

Information Page displays information about the extension like its version, the contact information, supervisor etc. It is opened by the button that is displayed as the letter “i” in the Settings Menu.

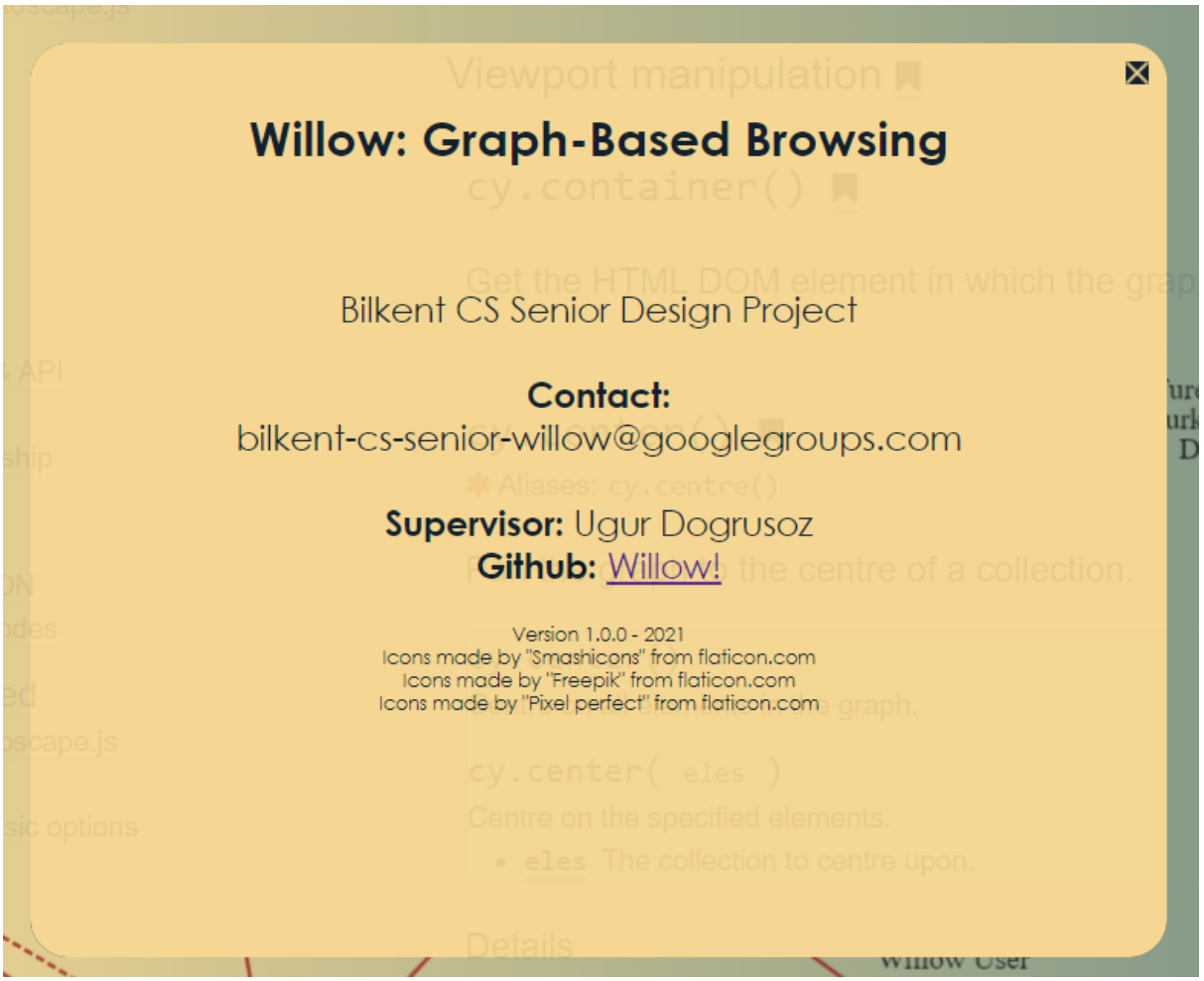

Figure 19: Information Page

3. Troubleshooting

3.1. Resolving a Crash

In the case of a crash, the Willow overlay may not open when the Willow icon is being clicked. The following steps should be taken to resolve the crash:

1. Open the Chrome Extension Page
2. Find Willow among the extensions.
3. Click the reload icon at the bottom right.

Keep in mind that this action will reset the graph and the sessions.

4. References

[1] “Chrome Browser system requirements,” *Google Chrome Enterprise Help*. [Online]. Available: https://support.google.com/chrome/a/answer/7100626?hl=en. [Accessed: 30-Apr-2021].