Rapport du projet JS\_Graph\_Sage

Plan

Introduction

1. La nature du projet (Sage -> besoin de Js\_graph)

* Idée et origines
* Notre équipe
* Structure du logiciel (Regarde général + fonctionnement d'attach)

1. L'utilisation pratique

* User guide
* Cas d'utilisation

1. Affichage du graphe avec D3.js
2. Connexion interface-terminal

* portes
* ouverture et fermeture de websocket
* import - export des changements

1. Nouvelles fonctionnalités

* affichage de nouvelles informations
* coloration

1. Amélioration de l'interface

* Sélection groupée
* Zoom?
* Repositionnement du graphe
* Customisation

1. Difficultés et idées abandonnées

* création de groupes ?
* getters JS?

Conclusion

# Connexion interface-terminal

La connexion au serveur est initiée dans la méthode `show\_CustomJS` dans 'customJSGraph.py'. Il vérifie si la variable `current\_server` est none, si c'est le cas il ajoute le graphe en paramètre dans un dictionnaire `graph\_client\_dict` sous la clé 1 et appelle la méthode `launch\_connection` ('connection.py'). Sinon, il appelle la méthode `client\_dictionnary\_verification` ('connection.py') qui vérifie si le graphe en paramètre est déjà dans `graph\_client\_dict` (c'est-à-dire déjà ouvert dans un onglet) et déconnecte l'onglet ouvert du websocket si c'est le cas avec la méthode `end\_connection\_client`. Après cela, `show\_CustomJS` ajoute également un graphe à `graph\_client\_dict` sous la clé numérique qui est égale à 1 + `current\_server.id\_counter` (la valeur de clé/id client le plus élevé jamais donné).

## Connexion au port

Remarque : la connexion au port 9001 se fait à la fois en Python et en JavaScript, nous devons donc changer les deux côtés pour assurer la connexion aux autres ports…

## Ouverture et fermeture de websockets

Alors que la connexion à un port est initiée dans le code Python, la connexion websocket démarre dans le navigateur. Le fichier ‘graph\_d3js.js’ définit entre autres ce qui se passe lors du chargement de la fenêtre. Pour garantir que le contenu de la page n'est pas affiché à l'utilisateur tant que les données du graphe ne sont pas entièrement chargées, nous définissons l'affichage du body du document comme "none".

Ensuite, la méthode `InitWebSocketConnection` est appelée. C'est l'une des méthodes JavaScript les plus importantes du projet car elle crée un nouveau websocket et définit ce qui doit être fait lorsque ce socket s'ouvre, se ferme, reçoit un message ou une erreur.

Bien qu'il n'y ait pas d'instructions pour l'ouverture du websocket (`websocket.onopen`) dans la solution d'origine, nous avons dû en ajouter quelques-unes pour nous assurer que la connexion n'est pas perdue lors du rechargement de la page.

La création d'un nouvel objet WebSocket dans `InitWebSocketConnection` implique apparemment la connexion d'un nouveau client en code Python correspondant à ce websocket et l'appel d'une méthode `new\_client` dans 'connection.py'.

## Garder la connexion lors du rechargement de la page

Le problème de l'API WebSocket en JavaScript est que la connexion sera inévitablement perdue au rechargement de la page. Cependant, le premier réflexe d'un client lorsqu'il rencontre un problème est de recharger la page, il était donc important de trouver un moyen d'implémenter cette fonctionnalité.

Ainsi, nous avons modifié le comportement du code JavaScript lors de la fermeture de webSocket (`websocket.onclose`). Nous utilisons la nouvelle option de JavaScript permettant de définir si la page a été accédée par rechargement (`performance.navigation.type == performance.navigation.TYPE\_RELOAD`) et appelons `InitWebSocketConnection` lorsque le socket est fermé à cause du rechargement de la page.

Lorsqu'un nouveau websocket est créé après le rechargement, le côté Python du code crée immédiatement un nouveau client en réponse. Cependant, la connexion de ce client serait immédiatement terminée par la méthode `new client` original car ce nouvel id client ne serait pas dans `graph\_client\_dict` (rappel : les id client sont ajoutés au dict dans `show\_CustomJS` qui n'est pas appelé dans ce cas).

Par conséquent, nous avons dû modifier les méthodes `client\_left` et `new\_client`. Initialement, `client\_left` se contenterait de 'pop' (supprimer) la paire clé-valeur du graphe-client dans `graph\_client\_dict` chaque fois qu'un websocket est fermé. Il ferme également complètement la connexion au serveur si le dict reste vide après la suppression.

Pour nous assurer que le graphe est réinséré dans dict avec un nouvel id client si la page a été rechargée, nous enregistrons le graphe supprimé dans une variable `reloaded\_graph`, qui est ensuite ajoutée à `graph\_client\_dict` sous le nouvel identifiant dans la méthode `new\_client`. Nous devons également faire attendre la méthode `client\_left` pendant une demi-seconde avec `time.sleep(0.5)` pour éviter la fermeture du serveur au cas où le `graph\_client\_dict` resterait vide avant que le `new\_client` ne soit exécuté.

## Affichage du graphe après le rechargement de la page

Lorsqu'une page est ouverte pour la première fois, une méthode `InitNewGraph` est appelée qui s'assure que le graphe s'affiche bien dans la fenêtre. Malheureusement, à moins qu'un graphe JSON n'ait été fourni dans les paramètres, cette méthode charge les données du graphe à partir de html, qui ne sont modifiées en Python qu'une seule fois lorsque `show\_CustomJS` est appelé. Ainsi, si nous devions appeler `InitNewGraph` sans argument lors du rechargement de la page, nous recevrons le graphe sans toutes les modifications que nous lui avons apportées. Par conséquent, nous avons créé une méthode `RequestRenewGraph` à appeler dans `PageOpenOrReload` si la page était rechargée.

Le `RequestRenewGraph` envoie simplement un message à Python demandant les dernières données de graphe sous forme JSON.

Ensuite, dans la méthode `TreatResponse` de 'Connection.js', nous appelons finalement `InitNewGraph` avec le résultat de la requête comme argument. (Chaque fois que nous utilisons `InitNewGraph`, nous devons également appeler `UpdateGraphProperties` pour nous assurer que les propriétés du graphe s'affichent correctement dans le menu de gauche).

Enfin, après `PageOpenOrReload` dans `webSocket.onopen`, nous changeons l'affichage du body en `inline` pour afficher le graphe et les propriétés qu'on vient de charger.

## Importation des modifications apportées au graphe dans le terminal Sage

Comme l'appel de la méthode `RequestRenewGraph` importe en fait les dernières données de graphe directement depuis Sage, le rechargement de page est un moyen efficace d'importer toute modification si un utilisateur décide de travailler avec le graphe dans le terminal plutôt que dans l'interface graphique. La même méthode est appelée lorsque vous cliquez sur le bouton 'Redraw Graph' en haut du menu, afin que l'on puisse importer les modifications sans recharger la page.

Un bonus qui vient avec l'utilisation de la fonction "InitNewGraph" lors du traitement de la réponse, est que le graphe nouvellement affiché sera redessiné en plein centre de l'écran et s'adaptera parfaitement à la taille de la fenêtre.

# Nouvelles Fonctionnalités

## Coloration

Pour une coloration à long terme, les éléments du graphe sont divisés en groupes, sinon ils perdraient de la couleur à chaque clic de souris.

Les groupes sont identifiés par des numéros. Pour ajouter un élément à un groupe particulier, il faut utiliser la fonction `SetGroupElement` dans graph\_d3.js et passer en paramètre un objet de classe `ValueRegisterer` qui a l'élément et le numéro de groupe parmi ses arguments.

Les éléments sont des objets de la classe `Element` avec des arguments contenant l'élément lui-même (sommet, arête...) et le type d'élément (EdgeType, NodeType, etc.)

La coloration des éléments selon leurs groupes est implémentée par les fonctions `RefreshNodes` et `RefreshEdge`.

Pour diviser les éléments en groupes, nous pouvons appeler les méthodes `SetEdgesColoration` ou `SetNodesColoration` et passer un tableau contenant les tableaux de tuples (pour les arêtes) ou ? (pour les sommets), chacun d'eux représentant un groupe distinct.

**Pour afficher l'arbre couvrant :**

Nous demandons à SageMath un tableau de tuples représentant des arêtes avec la méthode UpdateGraphProperties. Le tableau est ensuite transmis à la fonction `SetEdgesColoration` à l'intérieur d'un autre tableau (comme mentionné ci-dessus, cette méthode prend un tableau à deux dimensions comme argument). La méthode crée ainsi un nouveau groupe d'arêtes et le marque avec une couleur différente.