

PROJET TUTORE :
MANIPULATION DE GRAPHES AVEC SAGE



Réalisé par :

DUBAN Mathis
EL HIDRAOUI Jawad
TABBAB Obada
TSETSERAVA Dziyana

DUT Informatique 21-22

Rapport de projet présenté à :

Monsieur VALICOV Petru - Enseignant à l'IUT Montpellier Sète



Le 27-mai-2022

SOMMAIRE :

1. La nature du projet	6
1.1 L'origine du projet.....	6
1.2 L'essence de SageMath.....	6
1.3 Les graphes de Sage	7
1.4 Utilisation pratique de JS_Graph_Sage.....	9
2. Cahier des charges	13
2.1 Structure du logiciel	13
2.2 Fonctionnalités demandées.....	20
2.3 Optimisations ergonomique	22
3. Rapport technique	24
3.1 Affichage du graph avec D3.js.....	24
3.2 Connexion interface-terminal.....	31
3.3. Affichage des informations complémentaires.....	34
3.4 Amélioration de l'interface	37
4. Résultats	43
4.1 Comparaison des versions.....	43
4.2 Difficultés et idées abandonnées	44
4.3 Comparaison avec une solution concurrente	45
5. Gestion de projet	46
5.1 L'équipe en charge du projet	46
5.2 Méthode de travail – SCRUM.....	47
5.3 Difficultés rencontrées	50



INTRODUCTION

Français

Ce rapport est écrit dans le cadre du projet tutoré du second semestre visant à préparer un DUT Informatique à l'IUT de Montpellier-Sète. Il a pour but de retranscrire l'ensemble des étapes par lequel le groupe est passé pour mettre en œuvre et développer la solution que l'on connaît aujourd'hui.

Le projet a été réalisé par une équipe de quatre étudiants avec l'utilisation des langages de programmation JavaScript et Python, ainsi qu'une bibliothèque D3.js.

L'objectif principal du projet était d'améliorer et d'enrichir les fonctionnalités d'une solution développée par un groupe d'étudiants en Licence Pro il y a deux ans. Il s'agit d'une interface pour la manipulation de graphes dans une application mathématique bien connue : SageMath.

Dans ce rapport, vous trouverez l'ensemble des informations relatives au développement de la solution. Un manuel d'utilisation illustré sera fourni en début de rapport pour vous permettre de manipuler le logiciel et de tester par vous-même la solution développée.

Dans une seconde partie, vous prendrez conscience du cahier des charges du projet. Vous y trouverez une analyse de la solution existante et une liste des nouvelles fonctionnalités qui ont dû être ajoutées.

Une partie suivante sera consacrée au rapport technique. Vous apprendrez plus sur la visualisation des graphes avec D3.js, la connexion entre l'interface et le terminal, l'affichage des informations du graphe et la customisation de l'interface. Dans cette partie vous serez confrontée aux outils choisis par l'équipe, aux problèmes rencontrés lors de son travail et les solutions mises en place pour les résoudre. Vous découvrirez les fonctionnalités développées et les directions prises par le groupe. Des directions qui seront justifiées et expliquées tout au long de la lecture.

La quatrième partie fera le récapitulatif du projet et de ses résultats. Une comparaison sera faite entre les deux versions du projet et nous ferons une rétrospective critique des difficultés rencontrées dans une perspective d'évolution personnelle.

Enfin une partie entière sera consacrée à la gestion du projet où vous serez plongés au cœur de notre quotidien en tant que développeur SageMath.



Ce rapport est rédigé à l'attention de nos évaluateurs, mais aussi à tous ceux qui souhaitent comprendre et utiliser le logiciel ou bien qui souhaitent récupérer le projet et continuer à le développer, ainsi qu'à toutes les personnes curieuses, souhaitant découvrir comment ce projet s'est mené de bout en bout. Aucun prérequis n'est attendu pour lire le rapport.

INTRODUCTION

English

This report was written as part of the second semester tutored project aimed at completing the DUT in Computer Science at the IUT of Montpellier-Sète. Its purpose is to describe and analyze all the steps the group went through to develop and implement the solution we see today.

The project was undertaken by a team of four students using the JavaScript and Python programming languages, as well as the D3.js library.

The main goal of the project was to add improvements and new functionalities to a solution developed by a group of License Pro students two years ago. The said solution is an interface for manipulating graphs in the well-known mathematical application SageMath.

In this report, you will find all the information related to the development of the solution. An illustrated user manual will be provided at the beginning of the report to allow you to use the software and to test the developed solution yourself.

In the second part, you will get to know the specifications of the project. There you will find an analysis of the existing solution and a list of new features that needed to be added.

The following part will be devoted to the technical report. You will learn more about visualizing graphs with D3.js, establishing connection between the interface and the terminal, displaying graph information and customizing the interface. In this part you will take a look at the tools chosen by the team, the problems encountered during their work and the solutions put in place by the team to solve them. You will discover the functionalities developed and the directions taken by the group. Directions that will be justified and explained throughout the reading.

The fourth part will summarize the project and its results. A comparison will be made between the two versions of the project and we will do a critical retrospective of the difficulties encountered in order to allow some personal evolution.



Finally, an entire chapter will be devoted to project management. It will immerse you in the heart of our daily lives as SageMath developers.

This report is written for the attention of our evaluators, but also for all those who wish to understand and use the software, for those who wish to take this project and continue our development work and simply for all curious people, wishing to discover how this project went from start to finish. There are no prerequisites for reading this report.



REMERCIEMENTS

L'ensemble du groupe tient à remercier M. VALICOV pour son implication et son suivi régulier sur le projet. Ainsi que pour sa patience et ses conseils qui ont sû nous aiguiller.

Nous adressons également nos remerciements à Mme. MESSAOUI pour ses conseils concernant la rédaction de ce rapport.

NOTE POUR LE LECTEUR

Afin de faciliter la compréhension du rapport aux lecteurs certains codes couleur ont été mis en place.

nomDefonction : Les noms de fonctions présentes dans nos fichiers de projet seront toujours en gras.

Commande terminale : les commandes qu'il faut appeler à partir du terminal Sage seront surlignées en gris.



1. La nature du projet

Lors de notre attribution sur ce projet réalisé par d'anciens élèves de L3 pro, nous avons dû nous imprégner de leur travail afin de comprendre et avancer sur leur solution pour proposer une application fonctionnelle et intuitive visant à répondre à toutes les spécifications demandées. Mais tout d'abord nous devons expliquer l'origine du projet car cela va permettre de mettre en exergue énormément d'éléments et d'informations qui seront essentielles pour le bon déroulement du projet.

1.1 L'origine du projet

Dans un monde où la recherche et les technologies se développent de plus en plus vite, il est primordial que nous puissions comprendre le flot d'informations toujours plus grandissant que nous recevons. Aussi, il est très fréquent de traiter et de modéliser ces groupes d'informations par des graphes car leurs représentations visuelles permettent une compréhension plus rapide. Les outils permettant de visualiser toutes ces données sont donc destinées à un public scientifique connaissant l'essence des graphes et sont dans l'optique de pouvoir en tirer de nombreuses informations.

Actuellement, plusieurs outils sont à la disposition des scientifiques pour créer et étudier leurs propres graphes. *SageMath*, logiciel open-source fait partie de l'un d'entre eux. Même si les fonctionnalités proposées par le logiciel sont poussées et permettent beaucoup, il n'est pas spécifiquement destiné aux traitements et à l'affichage des graphes. Ainsi, pour pouvoir avoir une solution permettant de faciliter cette nécessité de traitement de graphes, ce projet est né. Il a été dans un premier temps développé par des étudiants de L3 pro, Aymeric GOUJON, Benoît PRIGENT et Tristan VALADE de la promotion de 2020 (cf. Annexe 1) qui ont su créer les premières fonctionnalités importantes du projet que vous pouvez retrouver dans leur rapport de projet (cf. Annexe 2). Nous avons donc par la suite fait un fork de leur travail pour poursuivre le projet tout en corrigeant l'existant.

1.2 L'essence de SageMath

SageMath est un logiciel de mathématiques open-source délivré sous licence GPL-GNU (cf. annexe 3) développé en février 2005. Ainsi des développeurs de tous les horizons collaborent et contribuent à l'enrichissement des fonctionnalités du service. Il permet à une vaste communauté d'informaticiens, de mathématiciens et d'individus de soutenir leurs projets et leurs recherches grâce aux outils proposés par *Sage*. Ce logiciel fournit une interface en ligne de commande Python permettant d'exécuter les fonctions fournies par les bibliothèques Sage mais également une interface graphique appelée Notebook qui fonctionne dans une interface web qui permet notamment de se connecter sur un serveur *SageMath* distant.

Voici la forme sous laquelle se présente le logiciel :

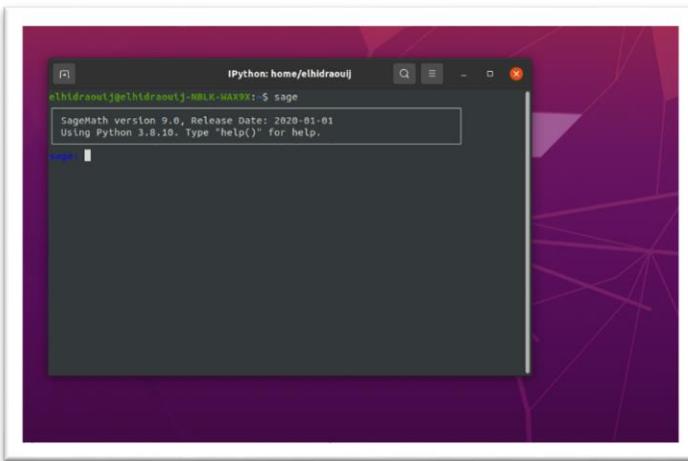


Figure 1 : interface native de SageMath

Aujourd’hui le logiciel est devenu une référence open-source grâce à sa gratuité (comparé à d’autres logiciels comme MATLAB, Mathematica, Maple) et à sa capacité de traiter de multiples champs des mathématiques notamment l’algèbre, l’analyse numérique, les calculs statistiques et un domaine qui va plus particulièrement nous intéresser : le champ des graphes. La partie graphe de SageMath contient plus d’un millier de fonctionnalités réalisables depuis l’interface Python cependant aucune de ces fonctionnalités ne permettent de manipuler des graphes facilement tout en ayant les autres fonctionnalités Sage à portée de main.

Le but de ce projet serait de permettre à n’importe quel utilisateur (scientifique ou non) d’utiliser les fonctionnalités *Sage* de manière très intuitive et visuelle afin de générer et manipuler n’importe quel graphe de manière très simple. Pour cela nous devons comprendre comment les graphes sont composés et comment ils sont utilisés dans SageMath.

1.3 Les graphes de Sage

Un graphe est une structure composée d’objets dans laquelle certaines paires d’objets sont en relation. Les objets correspondent à des abstractions mathématiques et sont appelés sommets (ou nœuds ou points), et les relations entre sommets sont des arêtes (ou liens ou lignes). Ils permettent de modéliser toutes sortes de réseaux (sociaux, routiers, informatiques etc...), de données (évolutions d’états, arbre de probabilités...) mais aussi de résoudre des problèmes complexes (problème du postier chinois...).

Voici à quoi peut ressembler un graphe :

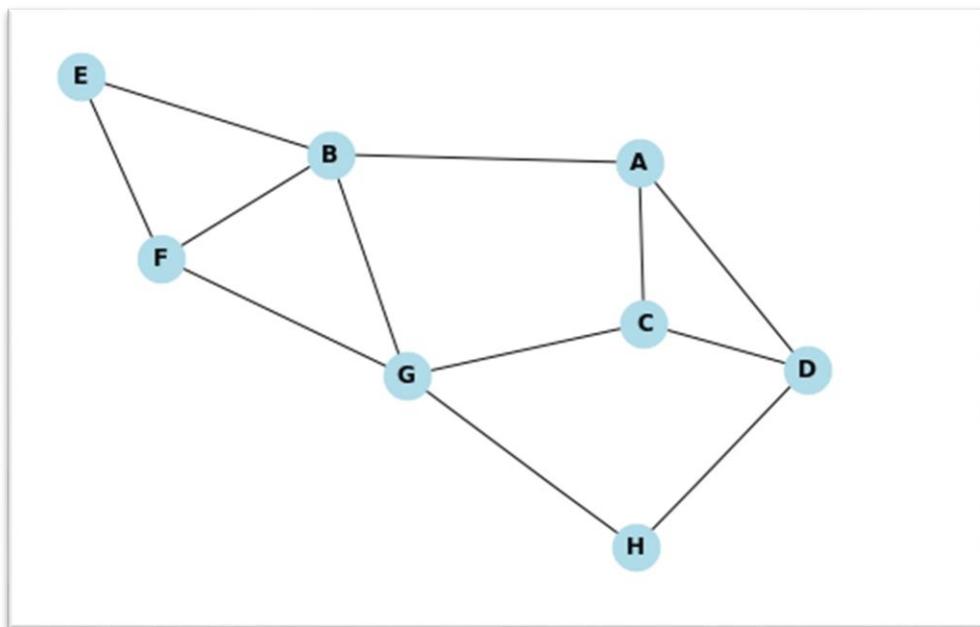


Figure 2 : graphe classique

Très prisés par notre tuteur les graphes ont été un pilier central à l'origine de ce projet.

Voici un exemple de l'affichage d'un graphe SageMath :

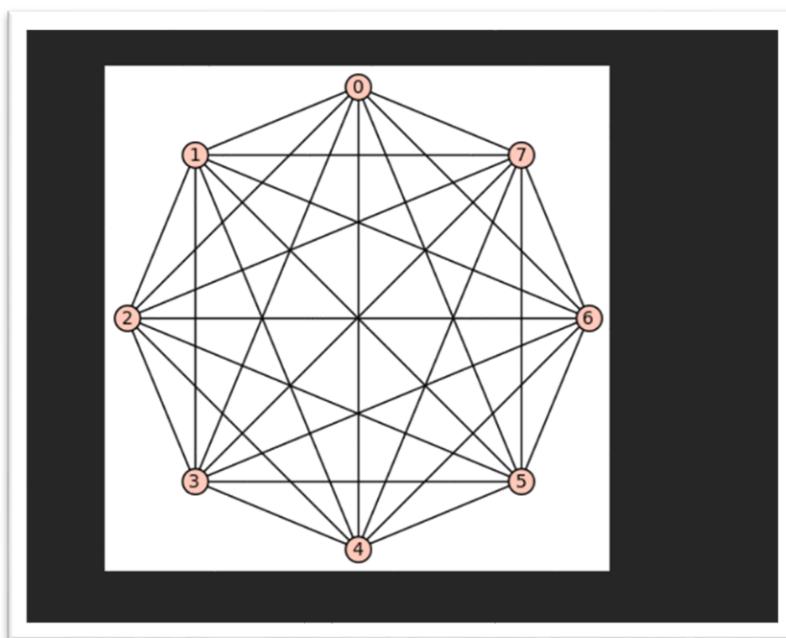


Figure 3 : graphe dans SageMath



Comme dit au préalable, le logiciel fonctionne par le biais d'interactions avec l'utilisateur qui se réalise à la manière d'un informaticien qui manipule son système de sa pleine volonté, sous forme d'insertion de commande, à travers le terminal. Une manipulation rigoureuse, rigide et bien commode aux habitués des consoles et des terminaux mais plutôt primitive, complexe et âcre aux regards des non-initiés.

Et c'est bien là que réside l'enjeu de notre projet. SageMath est destiné à un public varié et pas seulement à des informaticiens. Par exemple, les mathématiciens, premières cibles du logiciel, ne sont pas nécessairement familiers à cette forme de manipulation et il en va de même pour la majorité des potentiels utilisateurs. Ainsi il n'est pas évident pour eux de comprendre et de maîtriser un tel outil.

De plus, manipuler un élément graphique qui se présente essentiellement sous la forme de noeuds et de liens à travers des lignes de commandes et sans quelque représentation est tout sauf ergonomique/instinctif.

Non seulement le confort de travail des spécialistes est amélioré, mais surtout on permet au logiciel initial SageMath de s'ouvrir à une plus grande partie des utilisateurs. C'est une toute nouvelle cible qui s'offre au logiciel. Cela lui permettrait d'étendre ses parts de marché et d'affirmer un peu plus sa place légitime sur le marché.

C'est pourquoi nous proposons une interface graphique, directement liée au terminal de SageMath qui permet la manipulation graphique et l'affichage quasiment instantanée des graphes et de leurs propriétés principales qui permettront plus de fonctionnalités et plus de visibilité comparé à l'affichage de base implémenté dans Sage.

Notre but est donc de partir d'une solution existante et de rendre cette dernière fonctionnelle afin de remplir toutes les attentes envers ce dernier.

1.4 Utilisation pratique de JS_Graph_Sage

Maintenant que vous comprenez mieux le contexte du projet, nous allons expliquer étape par étape comment l'utilisateur peut télécharger et utiliser l'interface JS_Graph_Sage sur laquelle nous avons travaillé. Ce manuel d'utilisation sera suivi d'un diagramme de cas d'utilisation qui devrait donner à nos lecteurs une bonne idée de ce qu'est notre projet.

Manuel d'Utilisation

Etape 1 – récupération du projet

Installer le logiciel SageMath (cf. Annexe 4) puis faire un fork du code GitHub ou simplement télécharger l'archive.

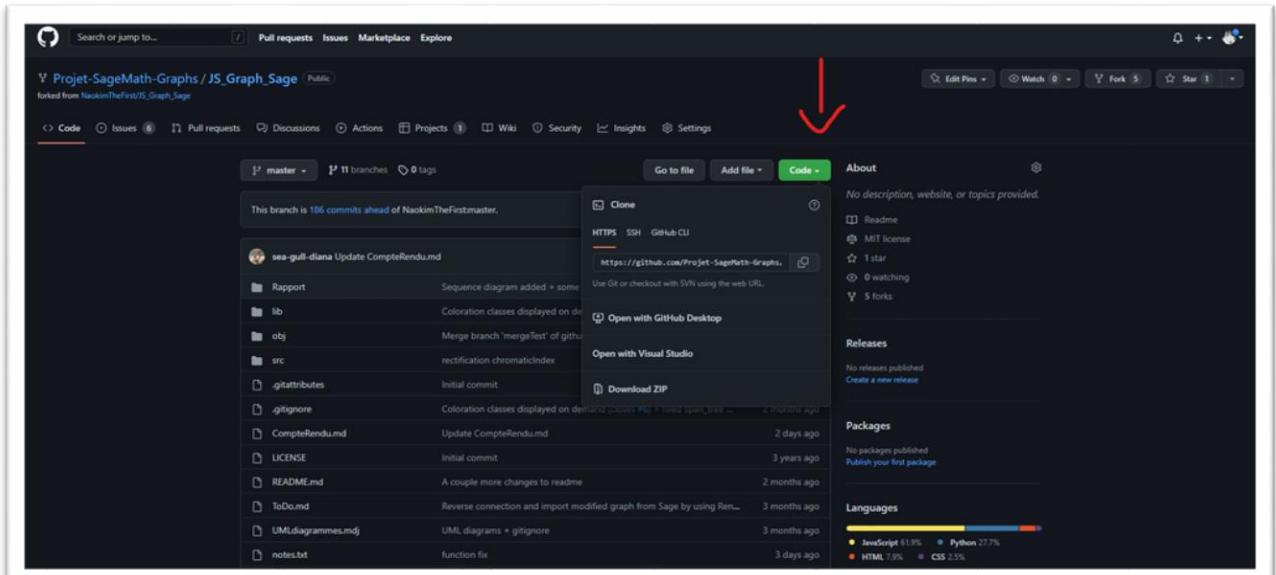


Figure 4 : Dépôt GitHub de JS_Graph_Sage Version 2.0

Etape 2 – attacher Sage au projet récupéré

Lancer le logiciel SageMath et utiliser la commande `attach("path_to_file_init_CustomJS.py")` pour lier le code Python du fork GitHub et SageMath. Par exemple, si le projet est situé dans un fichier nommé sage, vous devrez taper

```
attach("sage/JS_Graph_Sage/src/Python/init_CustomJS.py")
```

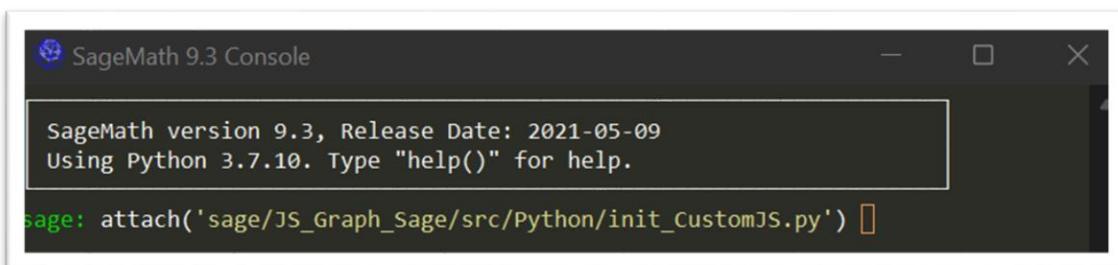


Figure 5 : attach du programme dans le terminal Sage



Etape 3 – Création de graph

Pour lancer l'affichage d'un graphe, il faut d'abord créer un graphe sur le terminal sage. Heureusement, il existe plein de graphes préexistants dans la librairie SageMath. On peut faire par exemple :

```
sage: g=graphs.CompleteGraph(5);
```

Cela va créer un graphe correspondant à la demande de l'utilisateur (dans cet exemple un graph complet d'ordre 5).

Etape 4 – Affichage du graph

Puis, il suffit de taper `show_CustomJS(g)` (on passe `g`, le graphe précédemment créé en paramètre) afin de lancer l'interface graphique sur votre navigateur par défaut de votre machine assurez-vous avoir une connexion internet active.

```
sage: show_CustomJS(g);
```

Importer des changements du terminal vers l'interface graphique

Si jamais vous devez réaliser des changements sur le graphe qui ne peuvent pas être réalisés depuis l'interface, effectuez les commandes depuis le terminal de SageMath puis cliquez sur le bouton Redraw Graph de l'interface pour importer les changements ou recharger la page du navigateur web.

Redraw Graph

Diagramme des Cas d'utilisation

Le diagramme ci-dessous (figure 6) va vous donner une liste détaillée (mais non exhaustive) des cas d'utilisation de notre solution.

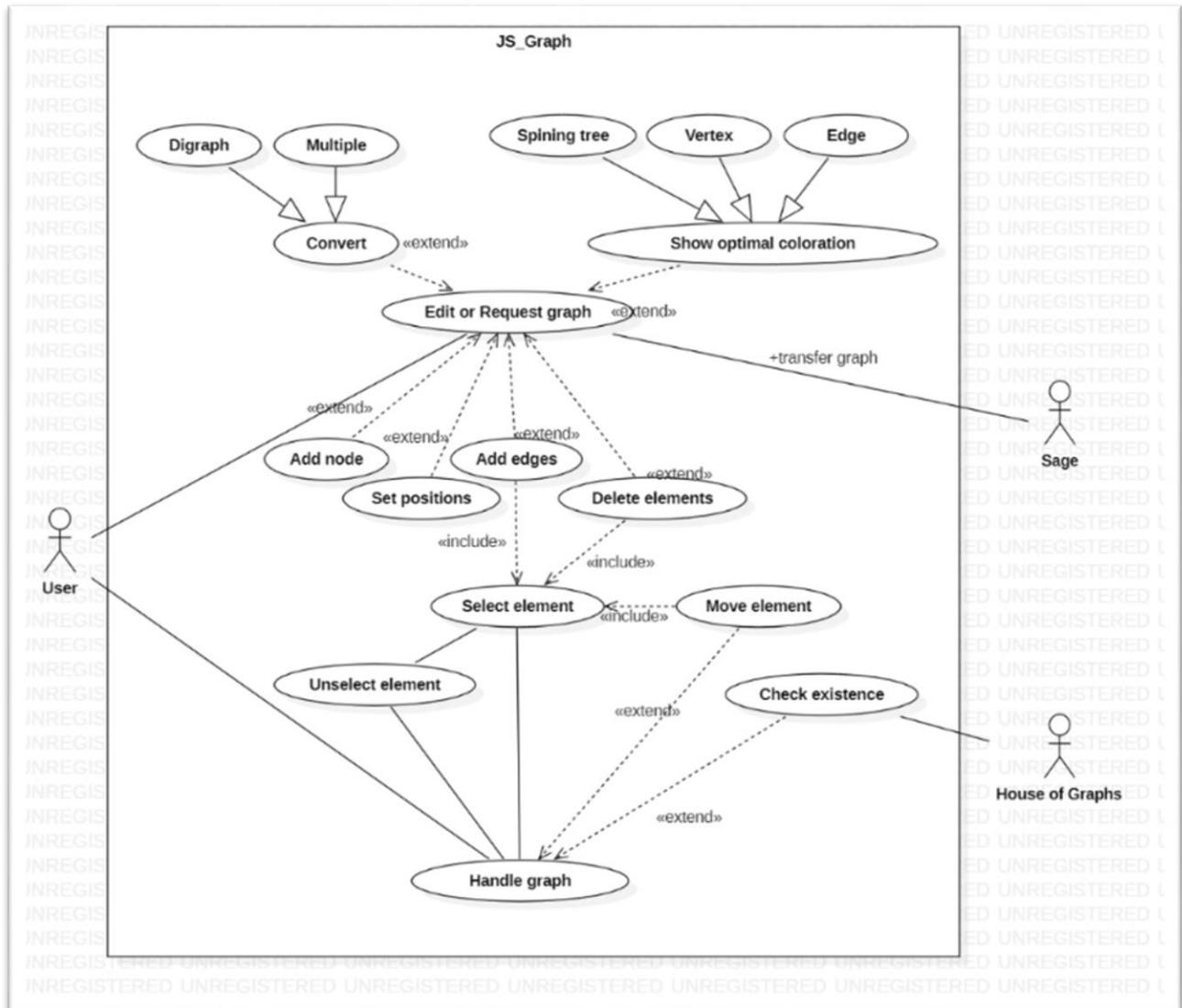


Figure 6 : Use Case Diagram



2. Cahier des charges

Après avoir vu ce qu'avait fait l'ancienne équipe sur le projet, nous avons dû analyser et lister les fonctionnalités qu'ils ont créées pour comprendre leur choix et pouvoir les améliorer.

2.1 Structure du logiciel

Le projet laissé par les étudiants de Licence pro contenant le socle de fonctionnalités essentielles au fonctionnement du projet avait des lacunes assez importantes qui devaient être corrigées.

Ce chapitre a pour but de vous dépeindre une vision générale du projet préexistant le nôtre afin que vous puissiez mieux comprendre les explications plus détaillées du chapitre trois, où nous présenterons notre propre contribution technique au projet.

L'ensemble des fonctionnalités présentes à ce moment-là sont des algorithmes exécutés par le terminal sage dont le résultat est par la suite, transféré à l'interface graphique qui les affichera aux yeux de l'utilisateur.

Tout d'abord lorsque la commande de l'affichage sur l'interface graphique est appelée dans le terminal, un objet de type graphe est chargé dans la fenêtre navigateur de l'utilisateur correspondant à l'interface graphique du projet. Puis, tout changement apporté sur le graphe affiché et chargé sur l'interface est par la suite transmis au terminal de SageMath, afin qu'il puisse à son tour, mettre à jour le graphe qui se trouve sur le terminal.

Technologies utilisées et organisation des fichiers

Technologie :

Cette interface graphique était construite par l'équipe précédente. La représentation du graphe est réalisée grâce à un Framework/bibliothèque appelé D3.js qui permet de représenter des données sous forme graphique (explication du Framework en détails - cf. 2.1.2).

Afin de pouvoir établir une connexion de l'interface graphique vers le terminal sage, les étudiants ont été amenés à concevoir une architecture dite, client-serveur. Le serveur est créé grâce à un script Python ainsi qu'à des librairies JavaScript pour son fonctionnement. Le client peut par la suite interroger le serveur grâce à un script de synchronisation rédigé en JavaScript.



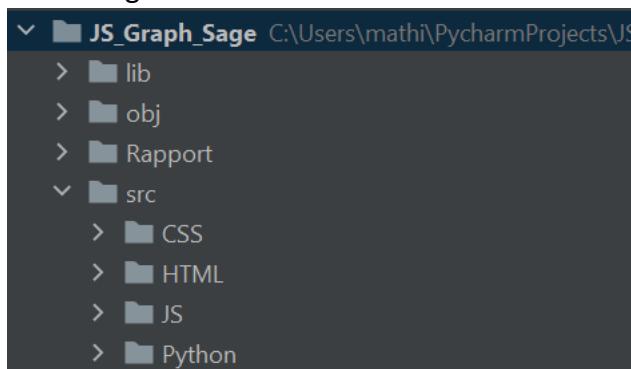
Arborescence des fichiers :

Les fichiers du projet étaient organisés de la manière suivante :

Un répertoire “src” qui contenait le code métier du projet avec des répertoires correspondants aux différents langages de programmations qui contenait eux-mêmes les codes correspondants (Python JS, ...).

Un répertoire “lib” contenant la librairie du Framework D3.js.

Un répertoire “obj” contenant des résultats représentés par un ensemble des fichiers JSON et HTML générés suite à l’exécution des commandes de l’interface Sage.



Fonctionnalités présentes

Affichage d’informations relative au Graphe :

Régulier (Is regular), Planaire (Is planar), et Bipartie (Is bipartite) - l’ensemble de ces informations sont contenu dans les caractéristiques du graphe qui sont perpétuellement mis à jour lorsque le graphe est modifié.

Ajouter des sommets : tout d’abord l’utilisateur se place sur un endroit de l’écran, il peut ajouter un sommet en appuyant sur la touche A du clavier. Le nouveau sommet est donc affiché et le terminal enregistre le nouveau sommet avec ses coordonnées dans l’objet de type Graph.

Ajouter des arêtes : l’utilisateur doit sélectionner les sommets entre lesquels il souhaitera ajouter une ou plusieurs arêtes et appuie sur E, une arête est donc ajoutée sur l’interface graphique entre les sommets choisis, et ceci est enregistré auprès du terminal sage.

Coloration de sommets : la coloration de sommets est présentée sous forme de boutons, lors de l’appui sur ce dernier, les sommets se colorent avec une coloration optimale.

Coloration des arêtes : de même que pour la coloration des sommets, lors de l’appui sur le bouton, les arrêtes du graphe se colorent souvent pour donner une coloration optimale.



Diviser une arête en ajoutant un sommet intermédiaire : en choisissant une arête, lors de l'appui sur la touche D du clavier, un sommet se rajoute au milieu de l'arête, la séparant en deux arêtes.

Orientation forte /Orientation aléatoire : à l'appui sur un des boutons d'orientation du graphe, l'interface graphique redessine le graphe avec une orientation.

2.1.1 CommandManager

Pour permettre à l'utilisateur de faire un retour en arrière ainsi qu'en avant (respectivement et communément appelé CTRL Z et CTRL Y) sur les modifications apportées à un graphe, les étudiants de licence professionnelle ont mis en place une classe appelée **CommandeManager**. Celle-ci contient deux piles, la première est appelée **commandStack** et correspond aux commandes invoquées par l'utilisateur qui apportent des modifications sur le graphe. Cette pile est la porte vers la possibilité de revenir en arrière après un changement sur le graphe. La seconde s'appelle **revertedCommandStack** et contient, inversement à la première, les commandes qui ont été annulées afin de pouvoir les réexécuter si besoin.

Dans cette classe on retrouve plusieurs méthodes :

- **Do** qui permet de rajouter les commandes exécutées par l'utilisateur à la pile des commandes.
- **Undo** qui permet de vérifier que la pile des commandes (**commandStack**) est remplie et fait revenir le graphe à l'état d'avant exécution de la dernière commande, sinon le programme affiche que le retour en arrière n'est pas possible car l'état du graphe est indifféré de l'état du graphe initial : "nothing to revert".
- **Redo** qui permet de regarder dans la pile des commandes annulées (**revertedCommandesStack**) et remettre l'état du graphe à l'état après exécution de la dernière commande annulée. Si la pile est vide, le programme affichera qu'il n'y a pas de commande à faire revenir : "Nothing to redo".
- **Execute**, la toute dernière méthode de cette classe, initialise la pile des commandes et appelle la méthode **Do** pour faire l'ajout de la commande exécutée dans le **commandStack**.

2.1.2 D3.js - cas d'utilisation

Lorsqu'il s'agit d'analyser le travail effectué par nos prédecesseurs, nous sommes obligés de mentionner le cœur de la plupart des fonctionnalités de notre interface - la bibliothèque D3.js.

Plus tard, nous aurons un sous-chapitre entier qui lui sera consacré (cf. 3.1), mais pour l'instant nous allons nous contenter de donner un bref aperçu de ce qu'est la bibliothèque et comment elle a été utilisée dans la version initiale de l'application.



Comme nous l'avons dit, D3.js est une bibliothèque JavaScript. Elle permet de visualiser différents types de données, tels que des graphiques, des diagrammes, ou bien des graphes, et d'interagir avec eux. Nos prédécesseurs utilisaient la version 3 de la bibliothèque en l'important avec un lien dans une balise <script> html.

D3.js est utilisé pour plusieurs fonctionnalités de base de notre interface.

Affichage du graphe

Tout d'abord, elle permet d'afficher le graphe à l'écran avec sa méthode **d3.layout.force()**. Le 'force layout' dans D3.js est essentiellement une stratégie de visualisation des éléments de données qui positionne les sommets liés selon une simulation physique. Le résultat de la méthode a été attribué à une variable **force** globale avec les informations sur les sommets et les liens reçues depuis le code Python.

Sélection des éléments du graphe

Deuxièmement, D3.js était à l'origine de la fonction de sélection. L'objet **d3.svg.brush** permet à l'utilisateur de dessiner un demi-rectangle transparent sur l'écran avec sa souris. Le rectangle est supprimé dès que l'utilisateur arrête de dessiner (sur l'événement 'end' du **brush**), mais d'abord les éléments du graphique à l'intérieur du rectangle sont marqués comme sélectionnés avec les fonctions **SelectElementsInsideExtent** et **SelectElement** et colorés en rouge avec une propriété 'stroke' de SVG.

Le déplacement des sommets

Enfin, d3.js est crucial pour faire glisser les parties du graphe. La méthode **drag()** permettait aux utilisateurs de faire glisser le sommet sur lequel ils avaient cliqué, bien qu'il soit impossible dans la version initiale de l'interface de faire glisser plusieurs sommets à la fois.

En plus des fonctionnalités de base mentionnées ci-dessus, les méthodes D3.js sont également utilisées dans JS_Sage_Graph pour certains objectifs plus petits, tels que la sélection des éléments SVG de DOM (méthodes **d3.select()** et **d3.selectAll()**) et la création d'une palette de couleurs pour colorer les parties du graphe (fonction **d3.scale.category20()**).

Pour conclure, D3.js est la bibliothèque déjà utilisée par SageMath qui permet à l'utilisateur d'afficher des graphes dans le navigateur en exécutant la commande `<graph_variable>.show(method="js")` dans le terminal Sage. C'est pour cette raison que nos prédécesseurs l'ont choisi comme le moteur principal de JS_Graph_Sage. Par conséquent, nous reviendrons sur le sujet de D3.js plus tard dans le troisième chapitre de notre rapport.



2.1.3 Connexion avec le terminal – transfert des messages

Comme nous l'avons mentionné précédemment, la solution JS_Graph_Sage se compose de deux parties, l'une écrite en Python et l'autre en JavaScript.

Le code Python est connecté au terminal Sage via la méthode **attach** de Sage et peut donc servir de pont à la manipulation des graphes avec toutes les méthodes de Sage.

La partie JavaScript de la solution est responsable de l'interface. Elle affiche les données reçues du terminal Sage et manipule les éléments graphiques.

En clair, le fonctionnement de la solution nécessite que ses deux parties puissent communiquer entre elles, ce qu'elle fait grâce à l'API websocket et son architecture client-serveur. La partie Python joue le rôle de serveur et l'interface JavaScript de client. Nous parlerons plus en détail de la façon dont cette connexion est établie dans le chapitre 3.2 du rapport.

Une fois qu'une connexion socket est établie, le client est capable d'envoyer des messages au serveur (terminal Sage) afin d'obtenir des informations sur le graphe ou de le mettre à jour dans Sage au moindre changement dans la simulation D3.js. Nous pouvons donc essentiellement transmettre des données entre les codes Python et JavaScript du programme en utilisant respectivement les fonctions de l'API **<server>.send_message()** et **<webSocket>.send()**.

Le message est envoyé par la fonction **SubmitMessage** définie dans le fichier Connection.js. Elle crée le message contenant toutes les données du graphe (en tant qu'objet JSON) et un nom de paramètre de requête (en tant que chaîne). Le message préparé est ensuite envoyé au code Python (terminal Sage) avec la fonction API mentionnée ci-dessus.

En Python, le message est traité par une fonction **message_received** qui prend en paramètres l'**id** du client qui envoie le message, le serveur et le message lui-même. Vous pouvez trouver cette fonction sur la figure 7.

```
def message_received(client, server, message):
    global graph_client_dict, reload_in_process

    if client['id'] in graph_client_dict :
        ...
        targetGraph = graph_client_dict[client['id']]
        JSONmessage = DataGraph(message)

        newGraph = ConstructGraphFromJSONObject(JSONmessage)
        response, newGraph = handle_message(JSONmessage.parameter,newGraph)
        ...
        update_graph(targetGraph, newGraph)

        if response[1] != None :
            returnMessage = JSONEncoder().encode({"request":response[0], "result": response[1]})
            server.send_message(client,returnMessage)
        else :
            end_connection_client(client, server)
```

Figure 7 : function **message_received**

Comme vous pouvez le voir, cette méthode crée un nouveau graphe dans Sage à partir de l'objet JSON qui lui a été envoyé à l'aide de la fonction **ConstructGraphFromJSONObject**. Elle appelle ensuite une fonction **handle_message** qui renvoie la réponse au message envoyé par le client. Dans la figure 8 vous pouvez voir le diagramme de séquence de ce processus. Et comme vous pouvez le remarquer, avant que la réponse ne soit envoyée au code JavaScript, notre fonction **message_received** appelle également une autre fonction extrêmement importante : **update_graph**, qui prend l'ancien graphique et le nouveau graphique en paramètres.

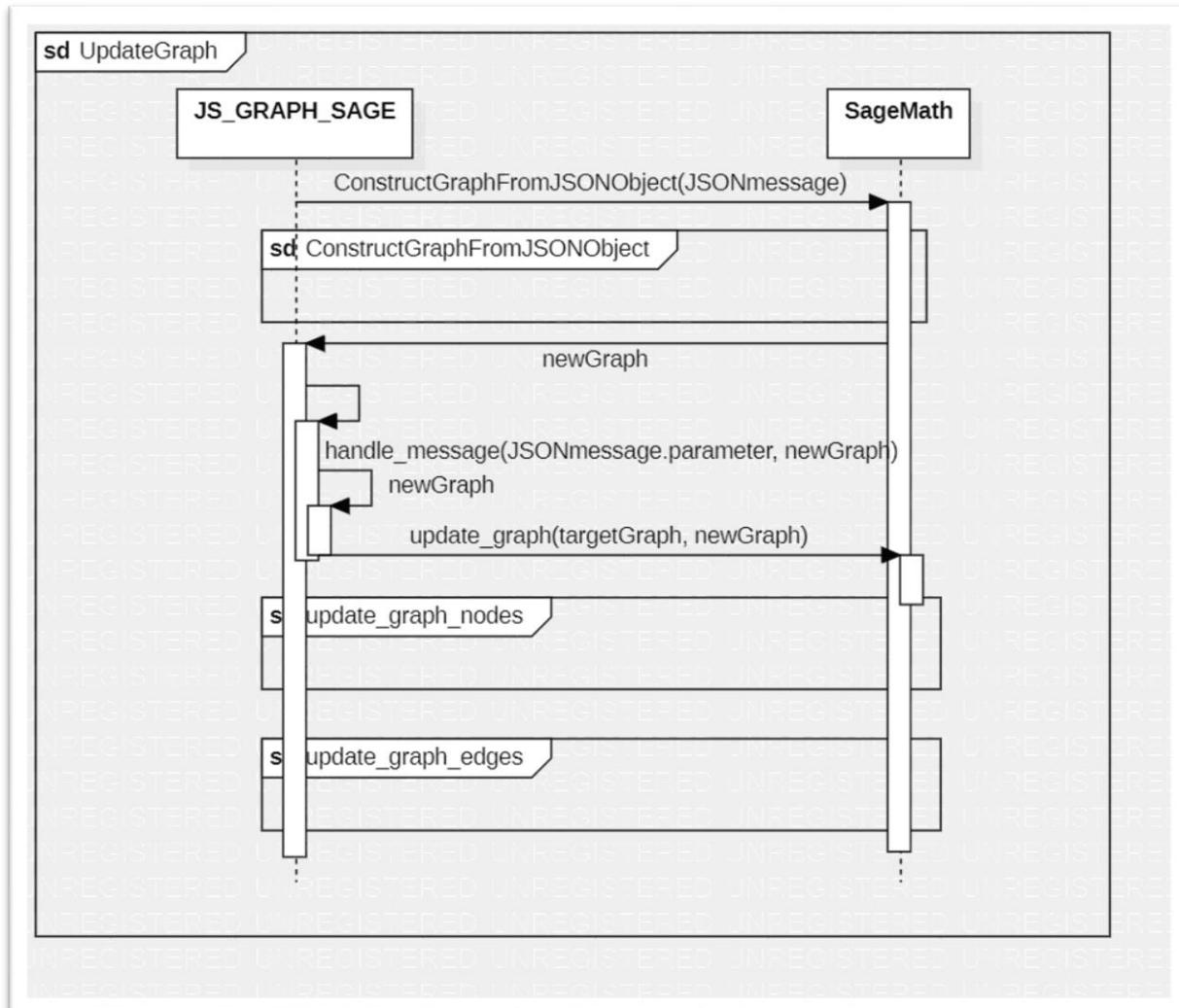


Figure 8 : Sequence Diagram (Graph update process)

Les diagrammes de séquence détaillés des méthodes `update_graph_edges` et `update_graph_nodes` se trouvent en annexes (cf. annexe 7), mais en un mot, toutes ces fonctions sont destinées à comparer l'ancien et le nouveau graphe. Et si, par exemple, le nouveau graphe a un sommet que l'ancien graphe n'a pas, alors la fonction appellera la méthode `add_vertex` de Sage pour ajouter l'élément manquant. Il en va de même pour les arêtes, les boucles et les positions des sommets.

Lorsque le message résultant est renvoyé à JavaScript, il est reçu par la fonction **TreatResponse** qui prend alors les mesures appropriées en fonction du paramètre de requête du message.

Maintenant, vous pouvez voir qu'il y a un processus assez complexe entre les deux parties du programme. Et pour bon nombre des nouvelles fonctionnalités dont nous avions besoin, nous avons dû bien comprendre ce système de traitement des messages.



2.2 Fonctionnalités demandées

Après avoir analysé ce que l'autre groupe avait développé, nous avons réalisé avec l'aide de M. VALICOV une liste de propriétés (du graphe) et de fonctionnalités essentielles, auxquelles l'utilisateur devait absolument avoir accès.

Informations élémentaires du graphe :

Le projet laissé par nos précédents collaborateurs n'était pas encore assez poussé pour que n'importe quel utilisateur voulant pleinement travailler sur un graphe soit satisfait.

Une des facettes les plus importantes est d'avoir accès à toutes les informations essentielles du graphe car ces dernières définissent le graphe en lui-même. La première étape a donc été d'afficher toutes les informations relatives au graphe grâce à Sage dans pour que l'utilisateur puisse capter le maximum d'informations le plus rapidement possible.

Questionner le nom du graphe

Une fonctionnalité qui nous a été également demandée, c'est de pouvoir questionner une base de données de graphes du site [The House Of Graphs](#), afin d'obtenir le nom du graphe en question pour savoir si ce dernier est un graphe connu. Cela serait une information très utile pour l'utilisateur. Pour cela on utilisera une notation G6 du graphe (cf. 3.3.3) que l'utilisateur devra également voir sur le menu de son interface.

Repositionnement du graphe

Dans la première version développée, le graphe pouvait être déplacé mais il manquait une fonctionnalité afin de recentrer le graphe au milieu de l'écran pour avoir une meilleure visibilité. La création d'un bouton permettant cette fonctionnalité devait donc être présente dans la version finale du projet.

Sauvegarde du graphe en cours

Dans la version précédente, tout changement du graphe était directement sauvegardé et cela posait soucis car l'utilisateur n'avais pas forcément envie que le graphe soit sauvegardé sans-cesse, c'est alors que l'idée de pouvoir enclencher la sauvegarde continue ou non a été proposé (via une touche du clavier).

Affichage du graphe par rapport à ses positions

Chaque graphe possède de nombreuses caractéristiques différentes. Ainsi, dans certains graphes les sommets ont des positions bien précises, alors que dans d'autres non.



Par exemple, dans un graph complet d'ordre 5, on peut voir grâce à la fonction `g.get_pos()` que chaque sommet à bien une position prédéfinie.

```
sage: g=graphs.CompleteGraph(5)
sage: g.get_pos()
{0: (0.0, 1.0),
 1: (-0.9510565163, 0.3090169944),
 2: (-0.5877852523, -0.8090169944),
 3: (0.5877852523, -0.8090169944),
 4: (0.9510565163, 0.3090169944)}
```

Figure 9 : terminal Sage – fonction `get_pos`

Et ici le Graphe de Paley(5) ne possède pas de sommets avec des positions

```
sage: g=graphs.PaleyGraph(5)
sage: g.get_pos()
sage: ||
```

Il faut donc un bouton qui permet de redessiner de manière différente le graphe s'il n'a pas de positions prédéfinies.

Bloquer/débloquer la position des sommets

Ajouter la possibilité pour les graphes ayant des sommets avec des positions non-définies de bloquer ou débloquer les positions des sommets via une touche du clavier.

Coloration des sommets/arêtes

Afficher les classes de coloration des sommets/arrêtes sous forme du texte.

Possibilité de fusionner un ensemble de sommets

Demander de redessiner le graphe avec un ensemble de sommets sélectionnés réunis dans un seul sommet (opération primordiale dans la théorie des graphes).

Permettre de changer le graphe affiché dans l'interface depuis le terminal

Précédemment, il n'existait qu'une connexion unidirectionnelle entre l'interface et le terminal Sage. On pouvait modifier le graphe Sage à partir de l'interface, mais on ne pouvait pas modifier le graphe de l'interface à partir du terminal Sage. Vu qu'on ne pouvait pas implémenter toutes les fonctionnalités proposées par Sage dans notre interface, c'était important de permettre à l'utilisateur de pouvoir travailler à la fois sur l'interface et dans le terminal.

Montrer l'arbre couvrant du graphe

Permettre à l'utilisateur de visualiser sur l'interface web l'arbre couvrant du graphe en cochant une checkbox.



Améliorer la liaison entre le projet et SageMath

La fonctionnalité `attach` développé par la première équipe demandait de saisir à la fois le chemin vers le projet et le chemin vers le fichier “`init_CustomJS.py`” du projet. Nous étions demandés de refaire la fonctionnalité pour que l’utilisateur ne devait saisir le lien qu’une seule fois.

Migrer de la version 4 vers la version 7 de D3.js

Nous voulions que notre application soit crédible et à jour le plus longtemps possible. Pour cela, nous devons avoir la version la plus récente possible de la bibliothèque D3.js. De plus, c’était une modification exigée explicitement par notre tuteur. Cela nous sera également utile si on veut ajouter des nouvelles fonctionnalités utilisant la dernière version de D3 JS.

2.3 Optimisations ergonomique

La prépondérance de la question de l’ergonomie n’est plus à démontrer et pourtant trop souvent peu considérée lorsque l’on parle de logiciel et d’interface graphique utilisateur. Penser l’ergonomie de son application, c’est la professionnaliser et l’adapter à l’utilisateur. Etant dans une démarche d’adaptation de l’outil envers son public, il est impensable de passer à côté de cette démarche primordiale. C’est pourquoi de légères modifications pointilleuses mais nécessaires ont été réalisé sur le projet.

Laisser la place nécessaire maximale à la manipulation du graphe

Imaginez-vous devoir dessiner, faire du montage vidéo, de la manipulation 3D ou bien de la programmation sur une toute petite partie de votre écran. Vous conviendrez que la commodité et le confort de travail ne sont pas optimaux et qu’une fenêtre un tant soit peu plus grande se révélerai être du luxe. Eh bien, tout cela est l’enjeux de cette modification visuelle. L’idée était de laisser la place maximale à la manipulation du graphe tout en réservant une place minimale mais confortable aux propriétés et paramètres du graphe. Nous avons donc revu la largeur du menu apparent sur la gauche pour qu’il prenne la taille de la chaîne maximale contenue à l’intérieur. De cette manière nous parvenons à maximiser la fenêtre de manipulation tout en garantissant une lecture confortable et claire des caractéristiques du graphe.

Adapter l’overlay/menu à son utilisation

Dans une dynamique de personnalisation nous voulions rendre l’expérience utilisateur unique et au maximum personnalisable au regard des compétences que nous disposons. De plus, avec les fonctionnalités que l’on ajoutait au fur et à mesure, ce bloc gris, épais, semblable à un rigide pilier qu’est le menu des propriétés occupait sans cesse de plus en plus d’espace et semblait envahir peu à peu l’ensemble de la fenêtre. La solution prise par le groupe a été



d'ajouter des boutons permettant de réduire ou de développer les différentes sections du menu permettant à l'utilisateur de ne garder visible que les sections dont il a besoin.

Adapter la taille des sommets et des arêtes

Dans le projet initial, la taille des sommets et des arêtes étaient codées de telle sorte que si notre résolution était grande, ils étaient affichés de la même manière que si notre résolution était limitée. Cela pouvait donner lieu à des situations délicates dans lesquelles on ne pouvait pas vraiment distinguer les sommets des arêtes. Après concertation, il a été convenu d'adapter automatiquement la taille des éléments du graphe pour que toutes les configurations puissent offrir une expérience utilisateur optimale.

Redimensionner le graphe selon la taille de la fenêtre

Un autre aspect qui se devait d'être pris en considération est le redimensionnement du graphe par rapport à la taille de la fenêtre. Cela nous est tous déjà arrivé de travailler avec deux applications en simultané. On s'organise de telle sorte que l'on se retrouve avec un logiciel sur la partie droite de notre écran et un autre sur la partie gauche. C'est une méthode de travail très courante qui peut très vite monter jusque trois ou quatre applications sur un même écran. Le problème avec le projet initial est que lorsque nous redimensionnions la taille de la fenêtre, le graphe conservait ses dimensions brutes et donc débordait de la fenêtre. Ainsi, dans une démarche de praticité et de confort, l'équipe a implémenté le redimensionnement automatique et adapté du graphe à la taille de la fenêtre. Désormais, il est donc possible pour un utilisateur de redimensionner sa fenêtre comme bon lui semble et de travailler en parallèle sur une autre application.

Drag de la sélection groupée

Pouvoir sélectionner un ensemble de sommets et les tous déplacer en même temps.

Rétraction des sous-menus et le menu responsif

Avoir un menu avec des sous-menus rétractables pour une meilleure visibilité tout en ayant le menu global responsive pour n'importe quelle taille de l'écran.

Mode nuit/jour

Pour une meilleure visualisation du graphe, la possibilité d'avoir un mode clair ou sombre.



3. Rapport technique

Dans cette partie nous allons voir plus techniquement comment nous sommes arrivés à réaliser toutes ces opérations et traitements de graphes via Sage.

3.1 Affichage du graph avec D3.js

Comme vous le savez déjà, afin de représenter le graphe, le projet utilise la bibliothèque D3.js qui permet d'afficher le graphe sur le navigateur web de l'utilisateur.

3.1.1 Fonctionnement D3.js

Une fois que l'utilisateur va utiliser la commande **show_CustomJS**, un onglet du navigateur favoris de l'utilisateur va afficher le graphe grâce à D3.js.

Comme nous avons dit précédemment, D3.js est une bibliothèque JavaScript open-source (dernière version actuelle 7.4.4). Elle vise à manipuler des données et les afficher dans un document HTML grâce aux éléments SVG générés par D3.js dans n'importe quel navigateur moderne. Les versions les plus récentes de D3.js proposent également des options d'affichage supplémentaires au SVG, mais ce n'était pas encore le cas pour la version 3 héritée de nos prédecesseurs.

Il suffit de télécharger la bibliothèque dans le projet pour avoir accès à toutes les fonctionnalités de D3.js en l'incluant dans une balise de 'script' de cette manière :

```
<script src="https://d3js.org/d3.v7.min.js"></script>
```

Les éléments SVG sont utilisés pour générer toute sorte d'éléments représentant différentes données (tableau, diagrammes bâtons, camembert, nuages de points...). Chaque représentation se trouve dans une zone SVG qui contient également des objets SVG représentant des points, barres...

Voici un exemple de représentations possibles :

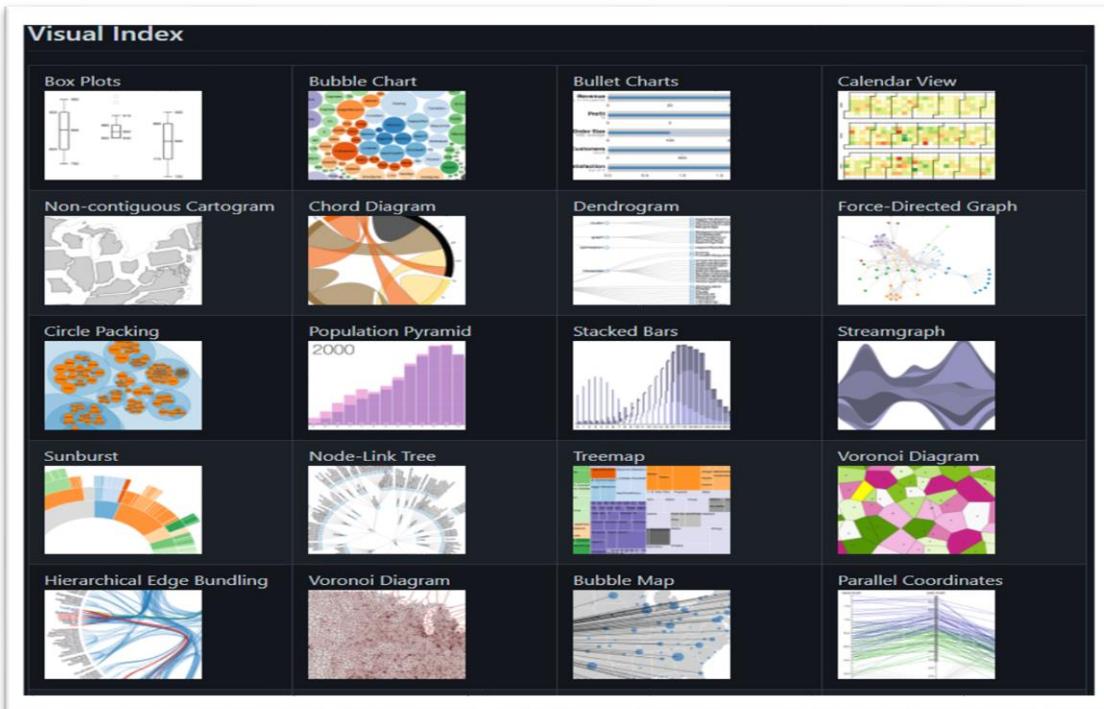


Figure 10 : représentation des données dans D3.js

Ces éléments se comportent comme n'importe quel éléments HTML, c'est-à-dire que ces derniers peuvent être en interaction avec des fonctionnalités CSS et Javascript. Les données utilisées pour leurs représentations sont récupérées depuis de nombreux formats de fichiers (JSON, CSV, GeoJSON).

Voici un exemple sur la déclaration d'une zone SVG (attribution de la largeur et longueur de la zone pour stocker un diagramme en D3 JS) :

```
// Attention ici il faut que le body possède déjà un DIV dont l'ID est chart
const svg = d3.select("#chart").append("svg")
  .attr("id", "svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom)
  .append("g")
  .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

Figure 11 : creation du SVG



3.1.2 Lancement de l'affichage D3.js

Une fois que l'utilisateur lance la génération du graphe, la fonction **InitNewGraph()**, qui contient plusieurs autres fonctions importantes, va s'occuper de créer et d'afficher le graphe en son entiereté (voir figure 12). Les informations du graphe vont être récupérées depuis le document HTML et transformées en format JSON (contenant les nœuds et arêtes) approprié pour D3.js (objets SVG) via la fonction **LoadGraphData()**. Chaque élément de ce fichier JSON (sommets et arrêtes) deviendra par la suite un objet SVG grâce aux fonctionnalités de D3.js dans la fonction **InitGraph()**. Et enfin la totalité du graphe sera affiché avec l'appel de la méthode **force.start()**.

```
function InitNewGraph(graph :null = null) {
    if (force) force.stop();

    LoadGraphData(graph); // transformation des données JSON et format D3 JS
    InitGraph(); // on rajoute les premiers attributs SVG et réalise les calculs des placements des points
    InitInterface(); // on place l'interface en fonction de la taille du navigateur
    ManageAllGraphicsElements(); //implementation de tout les éléments graphique
    InitForce();
    //Start the automatic force layout
    force.start();
}
```

Figure 12 : fonction **InitNewGraph**

C'est lors de l'appel de la fonction **InitGraph()** que la création du graphe a lieu (voir figure 13). Comme nous avons déjà mentionné dans le chapitre précédent (cf. 2.1.2) la fonction **d3.layout.force()** utilisée dans la version initiale de l'application joue le rôle crucial dans l'affichage du graphe. Les méthodes de "layout" permettent non seulement de créer les sommets et les arêtes du graphe, en y mettant les parties correspondantes de notre objet JSON, mais aussi de définir ses propriétés, comme les espacements entre les sommets et sa taille en général.

```
Function InitGraph() {
    ...
    force = d3.forceSimulation()
        .force("charge", d3.forceManyBody().strength(graphJSON.charge))
        .force("link", d3.forceLink()
            .distance(graphJSON.link_distance)
            .strength(graphJSON.link_strength))
        .force("x", d3.forceX(graphJSON.gravity).x(width()))
        .force("y", d3.forceY(graphJSON.gravity).y(height()));
    ...
    force.nodes(graphJSON.nodes);
    force.force("link").links(graphJSON.links);
    ...
}
```

Figure 13 : fonction **InitGraph**

Une autre fonction importante appelée par **InitNewGraph()** est à détailler. Il s'agit là de **InitForce()** (voir figure 14). La fonction anonyme créée pour l'évènement "tick" de D3.js assure la mise à jour de certains attributs des sommets et autres éléments du graphe avec chaque changement de notre "force layout".

```
function InitForce() {
    force.on("tick", function () {
        // Position of vertices
        nodes.attr("cx", function (d) {
            return d.fx;
        })
            .attr("cy", function (d) {
                return d.fy;
            })
        ...
    })
}
```

Figure 14 : fonction **InitForce**

Enfin, le "force layout" a les méthodes **stop()** et **start()** qui arrêtent et démarrent la simulation de notre "layout" respectivement. La méthode **start()** doit être appelée chaque fois que nous apportons des modifications aux éléments SVG afin qu'elles soient prises en compte par la simulation.



3.1.3 Migration D3.js vers la version 7

À ce stade, nous avons présenté comment le graphe est affiché avec la version obsolète 3 de D3.js. En effet, la bibliothèque possède déjà les quatrième, cinquième et même septième versions. De plus, l'écart entre la troisième et la quatrième version est énorme. L'équipe de développement D3.js a réécrit la bibliothèque presque complètement, en changeant les noms et les fonctionnements de plusieurs méthodes. Ainsi, la migration vers la dernière version de la bibliothèque nous a demandé pas mal d'efforts et d'études.

Migration vers V4

Forces :

Parmi les gros changements que nous avons dû prendre en compte lors de la migration vers la version 4, figurait l'implémentation des forces dans D3.js. La fonction **d3.layout.force()** a été remplacée par **d3.forceSimulation()** et les méthodes de cet objet de simulation ont subi d'importants changements de syntaxe (voir figure 15). Par exemple, au lieu d'appeler **.links(graphJSON.links)**, nous devons maintenant appeler **.force("link").links(graphJSON.links)**. Et au lieu de **.charge(graphJSON.charge)**, nous avons dû utiliser **.force("charge", d3.forceManyBody().strength(graphJSON.charge))**.

```
force = d3.forceSimulation()
    .force("charge", d3.forceManyBody().strength(graphJSON.charge))
    .force("link", d3.forceLink()
        .distance(graphJSON.link_distance)
        .strength(graphJSON.link_strength))
    .force("x", d3.forceX(graphJSON.gravity).x(width()))
    .force("y", d3.forceY(graphJSON.gravity).y(height()));

force.nodes(graphJSON.nodes);
force.force("link").links(graphJSON.links);
```

Figure 15 : creation de **forceSimulation**

Donc, fondamentalement, chaque ligne de code liée à D3.js dans la fonction **InitGraph()** a dû être réécrite selon les nouvelles règles de syntaxe.

Brush :

Toutes les autres fonctionnalités majeures de d3.js que nous avons mentionnées précédemment devaient également être réécrites.



Remplaçant **d3.svg.brush**, il existe désormais trois classes de “brush” (pinceau), parmi lesquels nous avons utilisé le **d3.brush** capable de brosser à la fois sur les dimensions x et y. Aussi, au lieu d'appeler **.x(d3.scale.identity().domain([-100000, 100000]))** (et la même chose pour 'y') pour déterminer la zone de dessin, nous devions appeler **.extent([[0, 0], [100000, 100000]])**.

Et comme l'objet **d3.event.target** n'existe plus, nous avons dû utiliser la nouvelle fonction **d3.brushSelection(this)** pour obtenir l'étendue du pinceau (la zone de sélection), au lieu de **d3.event.target.extent()**.

Il ne s'agit bien sûr pas d'une liste exhaustive des changements que nous devions ajouter. Par exemple, comme nous ne pouvions plus utiliser **d3.event.target.clear()** pour supprimer le rectangle de sélection et que nous n'avions pas trouvé de solution prête en ligne, nous avons dû trouver notre propre solution et appeler **d3.selectAll ("rect.selection").style("display", "none")** à la place.

Drag :

Tous les types de fonctions impliquant le repositionnement des sommets ont dû être modifiés, car les sommets du graphe D3.js n'avaient plus les propriétés **px** et **py**. Au lieu de cela, il y avait maintenant des propriétés **fx** et représentant les positions fixes des sommets, ainsi que des propriétés **vx** et **vy** pour la vitesse des sommets. Comme nous n'avions pas besoin que nos graphes se déplacent d'eux-mêmes, nous n'avions donc pas besoin de définir la vitesse. Nous avons donc simplement supprimé **px** et **py** partout et attribué les mêmes valeurs à **fx** et **fy** à la place. De tels changements ont dû être apportés non seulement aux fonctions du drag, mais à bien d'autres comme la fonction **center_and_scale** par exemple.

Parmi les autres modifications à apporter à la fonctionnalité drag de notre programme, nous avons dû appeler sur les sommets une fonction **d3.drag()** au lieu de la méthode **drag()** de ‘force layout’ pour les rendre “draggables”.

Les événements 'dragstart' et 'dragend' ont été renommés pour devenir simplement 'start' et 'end'. Et quelques modifications supplémentaires ont dû être apportées afin que les sommets suivent la souris au lieu de simplement sauter à un autre endroit lorsque le drag est terminé. Ainsi, nous avons dû ajouter un 'listener' pour un événement 'drag' modifiant les positions des sommets en fonction des positions actuelles du curseur. D'autres lignes importantes à ajouter dans le même but étaient **if (!d3.event.active) force.alphaTarget(0.3).restart()** dans la fonction de 'listener' 'start' et **if (!event.active) force.alphaTarget(0)** dans la fonction de 'end'. Celles-ci sont nécessaires pour d'abord faire bouger la simulation, puis la ramener à l'état statique.

Autres changements :

Dans l'ensemble, la quantité de modifications que nous avons dû apporter était probablement trop importante pour tout expliquer en détail. Parmi les autres modifications



notables, on peut citer par exemple la palette de couleurs que nous avons dû transformer de **d3.scale.category20()** en **d3.scaleOrdinal(d3.schemeCategory20)**. Nous avons également dû appeler la fonction **merge()** après chaque appel de **enter().append()** sur les éléments SVG sélectionnés, car **append()** ne fusionnait plus automatiquement les éléments entrants dans la sélection. La création d'une arête courbe a également eu des changements de syntaxe importants : les lignes **d3.svg.line().interpolate("cardinal").tension(.2)...** ont dû être remplacées par **d3.line().curve(d3.curveCardinal.tension(.2))...**

Migration vers V5

Assez étonnamment, la migration vers la version 5 n'a demandé qu'une petite modification. D3 ne fournissait plus les schémas de couleurs catégoriques **d3.schemeCategory20***, nous avons donc dû changer à nouveau notre palette de couleurs en **d3.scaleOrdinal(d3.schemePaired)**. Le seul inconvénient est que la nouvelle palette n'a que 12 couleurs différentes au lieu de 20.

Migration vers V6 et V7

La migration vers les versions 6 et 7 de D3.js n'avait aussi qu'un seul changement à prendre en compte. D3 a commencé à transmettre des événements directement aux listeners, remplaçant la variable globale **d3.event**. Ce qui signifiait que nous devions remplacer toutes les lignes comme

```
.on('start', function (d) {  
  if (!d3.event.active) force.alphaTarget(0.3).restart();...
```

par

```
.on(' start', function (event, d) {  
  if (!event.active) force.alphaTarget(0.3).restart();...
```

Ce qui nous a pris un peu plus de temps à comprendre, c'est que maintenant toutes les fonctions de "listeners" avaient l'événement comme leur premier paramètre, donc même si nous n'utilisions pas **d3.event** à l'intérieur de la fonction elle-même, nous devions définir deux arguments pour accéder à la seconde un remplaçant des lignes comme **.on("dblclick", function (currentData)** par **.on("dblclick", function (e, currentData)**.

Pour conclure, la migration vers une nouvelle version d'une bibliothèque a été une tâche étonnamment difficile, mais nous sommes fiers de dire que nous l'avons réussi. Après tout, l'utilisation d'une version à jour des bibliothèques et des frameworks est indispensable pour tout projet de développement logiciel réel.

3.2 Connexion interface-terminal

La connexion client-serveur entre l'interface et le terminal Sage est initiée dans la méthode **show_CustomJS** localisé dans le fichier « customJSGraph.py » du projet. Vous pouvez voir ci-dessous le diagramme de séquence pour cette partie du programme (voir figure 16).

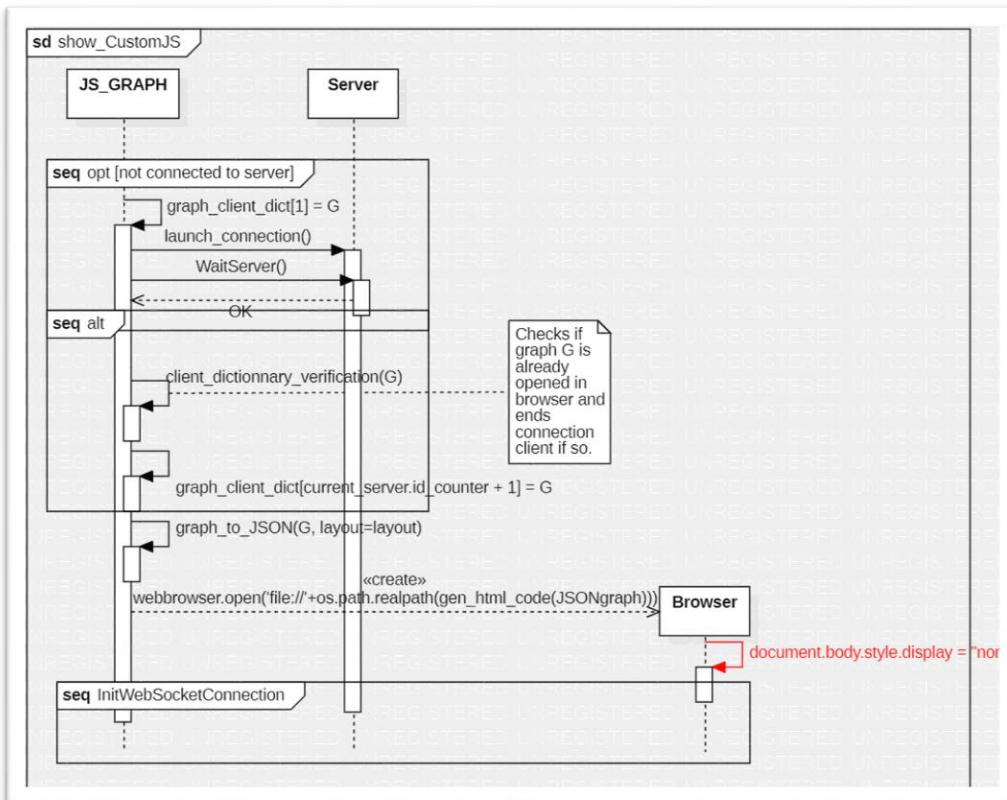


Figure 16 : Sequence diagram – fonction **show_CustomJS**

La méthode vérifie si la variable **current_server** est indéfinie (égale à **None**). Si c'est le cas, elle ajoute le graphe en paramètre dans un dictionnaire **graph_client_dict** sous la clé 1 et appelle la méthode **launch_connection** (« connection.py »). Sinon, elle appelle la méthode **client_dictionary_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 la fenêtre ouverte 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 égal à **1 + current_server.id_counter** (la valeur de clé/id client le plus élevé jamais donné).

Vous pouvez voir les autres diagrammes sur le processus de connexion dans les annexes (cf. annexe 6).



3.2.1 Ports et Sockets

Alors que la connexion à un port est initiée dans le code Python, la connexion client-serveur à travers le 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 le display du body du document html 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 ».

3.2.2. Relancement de la page

Le problème de l'API **WebSocket** en JavaScript est que la connexion sera inévitablement perdue au rechargement de la page. Malheureusement, le premier réflexe d'un client lorsqu'il rencontre un problème lors de sa navigation est de recharger la page sur laquelle il navigue. Il était donc important de trouver un moyen de fixer ce problème.

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 la fonction **new_client** ne soit exécutée.

3.2.3. Réaffichage du graphe et import des changements depuis Sage

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 rechargeement de la page, nous recevrions 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.

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 l'exécution de **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** qui importe les dernières données du graphe directement depuis Sage, le rechargeement 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.

3.3. Affichage des informations complémentaires

3.3.1. Information des caractéristiques du graph

Afin d'afficher toutes les informations essentielles du graphe, nous avons dû utiliser différentes fonctions Sage afin de pouvoir les transmettre dans le menu utilisateur. Pour cela nous demandons chaque caractéristique dans un message qui est transmis vers sage et duquel nous revenons une réponse contenant toutes les informations. Ce message est mis constamment à jour lors de la modification du graphe, cela permet à l'utilisateur d'avoir les bonnes informations du graphe en cours de modification. Voici une partie du message qui demande toutes les informations à Sage :

```
response[1].append(radius)
response[1].append(diameter)
response[1].append(graph.is_regular())
response[1].append(graph.is_planar())
response[1].append(graph.is_bipartite())
response[1].append(len(graph.vertices()))
ds = graph.degree_sequence()
```

Figure 17 : Demande et ajout des informations dans le tableau **response**

- *Girth* (taille de la maille) qui vérifie la taille du cycle le plus court, cette information est récupérée grâce à la fonction **graph.girth()** qui renvoie la taille du cycle le plus petit du graphe ou infini si le graph ne contient pas de cycle.

Pour empêcher l'échec de l'exécution lors d'un renvoi de la réponse infinie, il faut tester si le graphe n'est pas un arbre ou une forêt (dans ce cas précis le graphe n'a pas de cycle et on peut renvoyer nous-même infini en chaîne de caractères et éviter l'arrêt de l'application).

- *Is Eulerian* qui affiche si le graphe en question est un graphe Eulérien (un graphe qui possède un chemin qui parcours toutes les arêtes, une fois par arête). Cette information est gérée par la fonction **graph.is_eulerian()**.
- *Vertex Connectivity* qui affiche le nombre de d'arêtes connectées à tous les autres sommets grâce à la fonction **graph.vertex_connectivity()**.
- *Edge Connectivity* qui affiche le nombre de sommets connectés à tous les autres sommets grâce à la fonction **graph.edge_connectivity()**.



- *Chromatic number* qui affiche le nombre minimal de couleurs suffisantes pour colorier tous les sommets du graphe dans son entièreté notamment grâce à la fonction **graph.chromatic_number()**.
- *Chromatic index* qui affiche le nombre minimal de couleurs pour colorier toutes les arêtes du graphe. Cette information est donnée par la fonction **graph.chromatic_index()**.
- *Is Hamiltonian* qui affiche si le graphe possède un cycle hamiltonien (un chemin qui passe par tous les sommets une seul fois) grâce à la fonction **graph.is_hamiltonian()**.
- *Number of nodes* (nombre de sommets) qui affiche le nombre de nodes grâce à la fonction **graph.size()**.
- *Number of edges* (nombre d'arête) qui affiche le nombre d'arêtes grâce à la fonction **len(graph.edges())**.
- *Maximal degree* qui affiche le degré max du graphe grâce à la fonction

```
ds = graph.degree_sequence()  
ds[len(ds) - 1].
```
- *Minimal degree* qui affiche le degré min du graphe : **ds[0]**.

3.3.2 Visualisation des données à travers les couleurs

Pour une coloration à long terme (au moins plus long que la coloration en rouge des sommets et arêtes sélectionnés), 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** du fichier « `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 (sommel, arête...) et le type d'élément (**EdgeType**, **NodeType**, etc.)

La coloration des éléments selon leurs groupes est implémentée par les fonctions **RefreshNodes** et **RefreshEdge** en utilisant la palette de couleurs créée par la méthode de la bibliothèque D3.js (cf. 2.1.2 ou 3.1.3).

Pour diviser les éléments en groupes et les colorer, nous pouvons appeler les méthodes **SetLinksColoration** ou **SetNodesColoration** et passer un tableau à deux dimensions contenant des tuples (pour les arêtes) ou des chaînes de caractères (pour les sommets), chacun des sous-tableaux représentant un groupe distinct.



Pour afficher l'arbre couvrant :

Nous demandons à SageMath un tableau de tuples représentant des arêtes à travers la méthode **UpdateGraphProperties** qui envoie un message à Sage en lui demandant les propriétés du graphe, y compris l'arbre couvrant (renvoyé par la fonction `<nomGraph>.min_spanning_tree()` du Sage). Le tableau est ensuite transmis à la fonction **SetLinksColoration** 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.

Nous avons ajouté un checkbox à notre code HTML et y avons attaché un listener afin de donner à l'utilisateur la possibilité de choisir lui-même s'il souhaite afficher l'arbre couvrant.

Affichage de la coloration optimale :

La solution originale propose déjà des méthodes pour colorer les arêtes et les sommets des graphes en fonction de leurs classes de coloration qui ont été trouvées par Sage avec les fonctions `graph_coloring.edge_coloring(<graph>)` et `<graph>.coloring()` respectivement. Comme la méthode `coloring()` semblait fonctionner très lentement avec de grands graphes, nous l'avons remplacée par une méthode de Sage `graph_coloring.vertex_coloring(<graph>)` qui est plus efficace.

Si le graphe comporte plusieurs classes de coloration, elles deviennent rapidement difficiles à distinguer, car le programme utilise des couleurs similaires pour les marquer. Ainsi, nous avons fait en sorte que notre menu d'interface affiche également les classes de coloration sous la forme du texte. L'affichage se fait sous le bouton de coloration correspondant une fois qu'il a été cliqué.

Nous le faisons en appelant une nouvelle fonction **DisplayColoringInText** qui prend comme argument l'identifiant d'une ligne de tableau html `<tr>` où le texte doit être affiché et le tableau de coloration reçu comme message depuis Sage. Cette fonction place également au début du texte un bouton html qui supprime le contenu de toute la ligne du tableau une fois cliqué.

La partie la plus difficile de la coloration du graphe consistait à trouver comment éviter les conflits entre l'affichage de l'arbre couvrant et l'affichage de la coloration optimale des arêtes. Nous avons fini par créer une fonction **DeleteAllEdgeGroups()** qui supprime efficacement l'attribut **group** de chaque arête dans **graphJSON**. Cette méthode est appelée chaque fois qu'un utilisateur a besoin d'afficher ou de masquer un arbre couvrant et dans la fonction **TreatResponse** lorsque la coloration des arêtes a été demandée. Lorsqu'elle est suivie de la méthode **ManageEdges()**, elle permet de supprimer toutes les colorations d'arêtes existantes.



Cette méthode permet également de supprimer la coloration de l'arbre couvrant si l'utilisateur décoche la checkbox.

3.3.3 G6 et Di-G6

Que se cache-t-il derrière les termes G6 et Di-G6 ?

Avant de parler de G6 et de Di-G6 il faut savoir que les graphes peuvent être représentés de différentes façons. Il existe la représentation graphique, qui est la représentation la plus répandue et la plus visuelle, il existe également la matrice d'adjacence qui est une sorte de tableau à double entrée qui représente la présence ou non d'arêtes entre deux sommets et il en existe bien d'autres dont G6 et Di-G6 les acronymes de Graph6 et Digraph6.

Graph6 et Digraph6 sont des formats de stockages de graphes qui ont la particularité de représenter un ou plusieurs graphes sous forme de chaîne de caractères. C'est l'une si ce n'est la forme de stockage la plus économique en termes d'espace et c'est aussi celle qui est la plus facilement transportable et transférable.

Ces formules sont calculées automatiquement à la modification d'un graphe par Sage et sont renvoyées à travers le "websocket" vers les propriétés du graphe pour être finalement mises à jour sur l'interface.

Avant de calculer l'une de ces formules le graphe est soumis à quelques tests afin de ne pas provoquer d'erreur ni de fournir une information erronée. Il est convenu de vérifier d'abord que le graphe ne possède pas de boucle (car G6 et Di-G6 ne sont applicables que sur les graphes qui en sont dépourvus) grâce à la fonction **has_loop()** de Sage qui renvoie un booléen, **True** si le graphe à une boucle, **False** sinon. Puis de déterminer si le graphe est orienté ou non grâce à la fonction **is_directed()** de Sage qui renvoie un booléen, true si le graphe est orienté, false sinon. En fonction, de ces résultats nous affichons les résultats suivants :

<u>Condition</u>	<u>Résultat</u>
A une boucle	None
Est orienté	DiGraph6
N'est pas orienté	Graph6

3.4 Amélioration de l'interface

Dans la version du projet laissé par les étudiants de L3 pro, l'interface avec toutes leurs fonctionnalités étaient présents mais l'interface n'avait pas la possibilité d'être customisée. Il



était alors important de pouvoir proposer une expérience utilisateur différente pour chaque besoin spécifique. Nous avons décidé d'axer la customisation de notre interface sur différents points :

- Pouvoir avoir une vision plus claire du travail en cours en rétractant ou non le menu et ses sous-menus
- Pouvoir avoir une colorimétrie adaptée à l'utilisateur via 2 modes (mode clair et mode sombre)

3.4.1 Sélection groupée

La sélection groupée était une fonctionnalité implémentée par le groupe précédent, ou en cours d'implémentation. Elle était incomplète, car elle ne permettait pas à l'utilisateur de profiter pleinement du potentiel offert par l'interface. En effet, même si les éléments étaient d'apparence sélectionnés grâce au changement de couleur, il était impossible de les déplacer simultanément. Tout là est l'enjeu de cette fonctionnalité : permettre à l'utilisateur de déplacer en même temps, l'ensemble des nœuds sélectionnés au préalable.

La première étape fut de parvenir à créer le lien entre les sommets du graphe et les points affichés à l'écran. Pour cela on s'est tout simplement servi de l'attribut unique "name" de chacun des sommets pour répercuter le changement de coordonnées des sommets vers les points.

Ensuite on s'est servi de la fonction **drag()** de d3js qui comme son nom l'indique va permettre de déplacer un élément grâce au maintien du clic de la souris. Si on s'arrête là, il n'y a qu'un sommet qui sera déplacé, en effet même si un ensemble E de sommets est sélectionné, il n'y en a qu'un qui active l'événement drag et c'est celui-ci qui sera déplacé.

La prochaine étape a donc été de spécifier les actions mises en place lors du déclenchement de l'événement et celles mises en place lors de sa fin respectivement grâce aux méthodes **on('start')** et **.on('end')**. Il suffisait de calculer l'offset de la souris lors du déplacement des points et de le répartir sur l'ensemble des sommets (et donc des points) au relâchement du clic. Ce faisant, les points se déplaçaient bel et bien mais de manière saccadée et l'animation du drag se faisait seulement sur le point qui était manuellement déplacé.

Après plusieurs tentatives d'implémentation de cette fonctionnalité qui furent soldées par un échec, le groupe a finalement trouvé la solution au problème. Il a simplement fallu ajouter la méthode **on('drag')** de d3js qui spécifient les actions à mettre en place lors du drag. Et c'est donc en appliquant continuellement lors du drag, l'offset de la souris aux points sélectionnés, que le déplacement groupé des sommets a fini par se faire de manière fluide.

3.4.2 Redimensionnement et repositionnement du graphe

L'une des principales faiblesses du programme initial était le fait que le rectangle d'interface où le graphe était affiché, ainsi que la position et la taille du graphe n'étaient définis qu'une seule fois : lors de la première ouverture du graphe. Cela signifie que lorsqu'un utilisateur modifiait la taille de la fenêtre du navigateur, le graphe n'était pas redimensionné avec. Il en va de même pour le rectangle d'interface : si vous ouvrez d'abord une petite fenêtre et puis augmentez sa taille, la zone en dehors de l'étendue initiale de la fenêtre ne fera pas partie de l'élément rectangle SVG servant de canevas pour notre graphe.

C'était clairement un problème important à résoudre. Ainsi, nous avons ajouté une fonction à **window.onresize** "listener" (voir figure 18). Dans cette fonction, nous appelons une méthode **center_and_scale()** utilisée dans la fonction **InitGraph()** pour redimensionner et recentrer le graphique en fonction des nouvelles dimensions de la fenêtre. La méthode affecte de nouvelles valeurs aux attributs **fx** et de **force.nodes()** (**force** est le nom de l'objet renvoyé par la fonction **forceSimulation()** dont nous avons parlé au chapitre 3.1). Cependant, pour que ces mises à jour soient prises en compte, nous avons également dû créer une fonction **UpdateLayout()** qui arrêterait la force, appellerait la fonction **ManageAllGraphicsElements()** qui recrée tous les éléments SVG, puis exécuterait **force.restart()**. En plus du redimensionnement du graphe, la taille de ses sommets et l'épaisseur de ses arêtes sont également modifiées dans la méthode **OptimizeVertexSize()** que nous avons créée.

```
window.onresize = function () {
    if (typeof graphJSON != "undefined") {
        OptimizeVertexSize();
        var resizeParameters = center_and_scale();
        MyManager.ModifyPositionsOnResize
        (resizeParameters);
        UpdateLayout();
    }
}
```

Figure 18 : fonction **window.onresize**

Cette solution a amélioré l'apparence de l'interface lors du redimensionnement de la fenêtre, mais a apporté un problème inattendu : le fonctionnement de Ctrl+Z et Ctrl+Y. Comme vous vous en souvenez peut-être, lorsque l'utilisateur supprime un élément ou modifie la position des sommets, ces informations sont stockées dans la variable **commandStack** d'un objet **CommandManager**, qui est un tableau d'objets de la classe **Command**. Ainsi, lorsqu'un utilisateur fait un drag d'un sommet vers une autre position, puis modifie la taille de sa fenêtre et clique sur Ctrl + Z, les sommets reviennent à la position exacte où ils se trouvaient auparavant, ce qui peut même être en dehors de la fenêtre actuelle. Pour corriger ce bogue,



nous avons dû créer une fonction **ModifyPositionsOnResize** qui mettrait à jour toutes les positions à l'intérieur d'un **commandStack** et d'un **revertedCommandStack** (tableau utilisé pour Ctrl+Y).

Le problème est que les objets **Command** dans ces tableaux ont des propriétés **value** différentes selon le type d'action auquel ils se rapportent. Ainsi, une commande de suppression de sommet (**SupprNodeCommand**) aurait comme **value** un objet JSON représentant le sommet en question, tandis que pour la commande de repositionnement de sommet (**MoveNodeCommand**), la propriété **value** serait un objet de la classe **ValueRegisterer** qui a à son tour les propriétés **oldValue** et **newValue** où x et y positions du sommet repositionné sont représentés sous forme de tableaux de taille deux. Et si la commande ne repositionne pas un seul sommet, mais une sélection d'entre eux (**MoveSelectedNodesCommand**), la propriété **value** contiendrait un tableau des objets **ValueRegisterer**. Donc, dans l'ensemble, nous avons dû créer trois types de boucles différents pour modifier toutes ces positions dans notre fonction **ModifyPositions**, ce qui, pour être honnête, n'est pas une solution très efficace et soignée. Cette partie est quelque chose qui pourrait certainement être améliorée dans notre projet.

3.4.3 Affichage des noms et positions des sommets

Mapping des sommets

L'un des problèmes de la solution initiale était l'affichage du graphe avec des types de sommets non-string. Ou plutôt c'était la modification non demandée du graphe après un tel affichage qui posait le problème.

Si vous vous souvenez, dans le chapitre 2.1.3, nous avons brièvement expliqué comment le graphe dans Sage est mis à jour après chaque modification qui se produit sur l'interface. L'objet JSON représentant la nouvelle version du graphe mis à jour est transformé en graphe Sage et est comparé sommet par sommet, arête par arête avec l'ancienne version du graphe pour savoir quelles modifications sont nécessaires.

Mais bien sûr, le programme ne pouvait pas savoir à partir d'un seul objet JSON quels étaient les types de sommets d'origine, il a donc simplement utilisé des chaînes de caractères à la place.

Imaginez maintenant ce qui se passe si l'ancien graphe a des sommets de type int et que les sommets du nouveau graphe sont de type string. Bien sûr, la comparaison les considérera comme des sommets différents. C'est pourquoi, dans la solution d'origine, dès qu'un utilisateur ouvrait un graphe avec des sommets non-string dans l'interface JS_Graph_Sage, tous ses anciens sommets et, par extension, les arêtes étaient supprimées, et de nouveaux sommets étaient créés avec des noms de type string.



Pour résoudre ce problème, nous avons créé un dictionnaire global **original_nodes** qui "mapperait" les valeurs d'origine des sommets à leur représentation sous forme de chaîne de caractères. Ce dictionnaire reçoit des valeurs lors de la conversion d'un graphe en objet JSON (méthode **graph_to_JSON**). Et ensuite, il est utilisé dans la fonction **ConstructGraphFromJSONObject** pour créer un graphe Sage avec des valeurs de sommets corrects.

La question des positions des sommets du graphe

Une autre modification que nous avons ajoutée à la solution d'origine était liée au traitement des positions des sommets sur l'interface.

Initialement, chaque fois que le graphe Sage était mis à jour dans la partie Python du programme, les positions des sommets étaient réinitialisées à l'aide de la méthode **set_pos** du Sage.

Cependant, les graphes dans Sage ne sont pas obligés d'avoir des positions. Certains des graphes, tels que CompleteGraph les ont, certains, comme PaleyGraph, n'ont pas. Et si vous appelez une fonction SageMath **show()**, le graphe avec des positions non définies sera affiché différemment à chaque fois.

Ainsi, le fait que JS_Sage_Math réattribuait les positions du graphe dès le moment même où ce graphe a été créé pourrait être considéré comme un problème.

Par conséquent, nous avons décidé de ne pas appeler la méthode **set_pos** à chaque mise à jour. Et afin d'attribuer les positions aux sommets affichés sur l'interface dans la fonction **graph_to_JSON**, nous avons décidé d'utiliser le layout 'spring' de SageMath (mais uniquement pour les graphes dont les positions n'étaient pas déjà définies). Ainsi, nous avons ajouté quelques lignes de code testant si la fonction du graphe **get_pos** renvoyait quelque chose, et sinon, appelant la méthode **G.graphplot(layout='spring')._pos** afin de définir le dictionnaire de positions pour l'objet JSON.

Nous avons également ajouté une nouvelle fonctionnalité à l'interface que nous avons appelée "freeze positions". Chaque fois qu'un utilisateur appuie sur la touche "F", les positions des sommets sont mises à jour comme elles l'étaient dans la version précédente de la solution.

Ainsi, l'utilisateur pourrait désormais décider lui-même si les positions doivent être définies. Et si les positions ne sont pas définies, chaque chargement de page ou en appuyant sur un bouton "Redraw Graph" redessinerait complètement le graphe en fonction du layout 'spring'.

Plus tard cependant, nous avons découvert un nouveau problème. Maintenant que les positions n'étaient pas définies à chaque mise à jour, le graphe pouvait avoir des positions fixes de certains sommets, mais pas tous. Par exemple, si un utilisateur a cliqué sur "F", et puis a ajouté un nouveau sommet.



Pour éviter les erreurs d'interface lié à une telle situation, nous avons donc dû créer la fonction **CheckForUnsetPositions** que vous voyez sur la figure 19.

```
def CheckForUnsetPositions(targetGraph, newGraph, newGraphJSON):
    Gpos = targetGraph.get_pos()
    if Gpos and len(Gpos) < len(newGraphJSON.nodes):
        posdict = {}
        for n in newGraphJSON.nodes:
            posdict[original_nodes[n.get("name")]] = (n.get("x"),n.get("y"))
    newGraph.set_pos(posdict)
```

Figure 19 : fonction **CheckForUnsetPositions**

Cette fonction serait appelée dans la méthode **message_received** juste après la construction du nouveau graphe à partir de l'objet JSON. Comme vous pouvez le voir, il vérifie si la longueur du dictionnaire de positions du nouveau graphe est la même que celle de l'ancien graphe. Et sinon, il réaffecte toutes les positions.

Donc, fondamentalement, ajouter un sommet à un graphe avec des positions définies dans notre interface est devenu équivalent à appuyer sur la touche "F".

3.4.4 Customisation de l'interface

Ce projet que nous réalisons doit être un vrai outil de travail scientifique de graphes pour des professionnels (professeurs, chercheurs...). Pour cela les fonctionnalités doivent être claires et simples d'utilisation. L'utilisateur peut désormais rétracter les sous-menus afin d'obtenir un gain de place supplémentaire sauf la partie « Properties » car on l'estime son importance trop élevée pour la laissée retractée, en effet c'est la partie centrale du menu.

Notre deuxième implémentation est la possibilité d'utiliser le dark ou light mode qui va changer les couleurs de l'interface. Le choix du mode sera réalisé via un bouton qui va changer de mode et sauvegarder ce dernier dans les cookies de l'utilisateur (quel que soit son navigateur par défaut) afin que lors de sa prochaine visite son choix sauvegardé soit affiché.

Nous avons utilisé les fonctions JavaScript **Windows.localStorage()**. Ces fonctions permettent de générer des cookies via un « couple de valeur » avec deux attributs : “Key” et “Value”. Il nous suffit donc d'avoir un couple pour les thèmes choisis et **ShowCustomJS** va pouvoir générer l'interface voulue, voici un exemple de cookies générés par notre application dans le navigateur de l'utilisateur :

Key	Value
themeSelect	lightMode

Lors du chargement du graph, les contenus des cookies de Key et Value vont être lus et le graph sera affiché dans le mode correspondant.

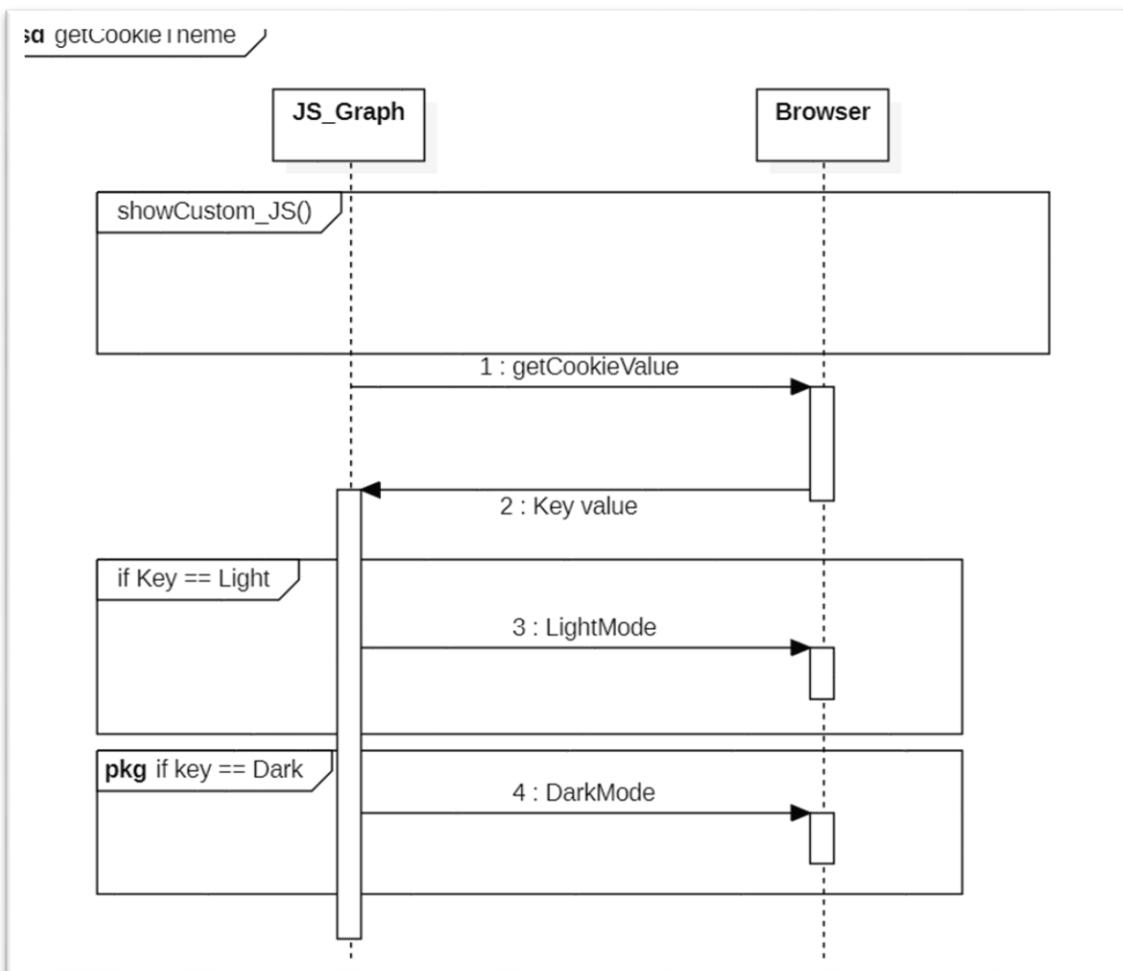


Figure 20 : Sequence Diagram sur le procéssus de la recuperation du cookie

4. Résultats

Cette partie est une démonstration des résultats que nous avons pu tirer après l'ajout de nos différentes fonctionnalités et améliorations dans le projet.

4.1 Comparaison des versions

Nous allons maintenant faire le point sur les résultats obtenus à l'issue de ce projet tutoré en comparant les deux versions de **JS_Graph_Sage** : la première développée par les étudiants de l'IUT il y a deux ans et la nôtre. Dans les chapitres précédents, nous avons beaucoup parlé de ce qui manquait à la solution d'origine et des nouvelles fonctionnalités que nous avons ajoutées. Donc, dans cette partie, nous allons d'abord résumer brièvement tout cela.



Mais en plus de parler de la façon dont la solution a été améliorée, nous examinerons également les aspects dans lesquels la solution d'origine était peut-être meilleure que celle que nous avons maintenant.

Tout d'abord, la dernière version de JS_Graph_Sage a toute une série de nouvelles fonctionnalités. Parmi les plus importantes, on peut citer le rechargement de la page sans perdre la connexion, l'affichage de multiples informations du graphe dans le menu à gauche de l'interface, le déplacement d'une sélection et la possibilité d'importer dans l'interface les modifications apportées au graphe dans le terminal Sage.

L'interface elle-même est devenue plus personnalisable et plus réactive, car l'utilisateur peut désormais basculer entre les modes sombre et clair et le graphe est redimensionné et recentré à chaque fois que la taille de la fenêtre est modifiée.

De plus, la nouvelle version permet de mieux préserver la nature du graphe, car il est désormais possible d'ouvrir un graphe avec différents types de sommets sans qu'ils soient remplacés par des chaînes de caractères, et les positions des sommets ne sont plus affectées par défaut.

Et enfin, la solution utilise maintenant la dernière version de la bibliothèque D3.js au lieu d'une version obsolète.

Dans l'ensemble, la nouvelle version s'était beaucoup améliorée par rapport à la première. Cependant, certaines des modifications ont causé de nouveaux problèmes qui n'existaient pas dans la version originale. Ainsi, le fonctionnement de Ctrl+Z et Ctrl+Y est devenu beaucoup moins stable et plus complexe car nous devons réécrire chaque position à l'intérieur des commandes sauvegardées à chaque fois que la fenêtre est redimensionnée. Si l'utilisateur utilise la nouvelle fonctionnalité de fusion de sommets, Ctrl+Z sera impossible pour cet événement et pourrait produire un comportement d'interface étrange. De plus, la palette de couleurs n'a plus que 12 couleurs différentes au lieu de 20 en raison de la migration D3.js.

Malheureusement, nous n'avons pas eu le temps de réparer chaque bug qui accompagnait les modifications que nous avons apportées, mais nous avons réussi à corriger un nombre important de bugs et d'imperfections de la solution d'origine.

Pour conclure, nous pensons que même si notre version n'est pas 100% meilleure par rapport à la précédente, nous avons réussi à faire un énorme pas en avant dans le développement de JS_Graph_Sage, en réalisant un programme très prometteur et utile.

4.2 Difficultés et idées abandonnées

Au fur et à mesure que le projet avançait nous avons rencontré différentes difficultés qui nous ont ralenti dans notre avancement.

Tout d'abord, nous avons récupéré une version réalisée par une autre équipe qui n'avait malheureusement pas écrit de rapport complet visant à expliquer leur démarche et leur code (qui n'était pas commenté). Cela nous a pris beaucoup de temps de nous imprégner du code afin de pouvoir travailler par-dessus leur travail. Certaines de leur fonctionnalité avaient des choix de conception différent de notre point de vue et malheureusement par faute de temps



nous n'avons pas pu retravailler cette base et avons dû continuer tout en respectant leur travail.

De plus, travailler sur un projet qui nécessite une connexion entre plusieurs applications contenant des langages différents était une première pour nous également car chaque programme et langage possèdent leurs propres caractéristiques. Par exemple pour les fonctions pour **merge_vertices** ou **girth**, ces fonctions Sage possèdent leur propre type de renvoi (**girth** peut renvoyer infini par exemple, il faut donc prendre cela en compte et changer le code en conséquence et fait donc rallonger le temps d'implémentation).

De nombreuses idées n'ont pas pu être réalisées faute de temps, voici une liste d'idées intéressantes qui pourront être réalisées dans une version future :

- Permettre à l'utilisateur d'annuler la merge des sommets avec Ctrl+Z.
- Pouvoir avoir une location de port Dynamique sur la machine de l'utilisateur et ne pas imposer un seul et unique port de communication (le travail sur cette fonctionnalité est commencé, mais pas fini).
- Pouvoir réaliser une fonctionnalité de sélection de sommets via un lasso et non via une zone rectangulaire
- Pouvoir zoomer sur le graphe
- Avoir la fonction **merge_vertices** qui permet de redessiner uniquement les sommets réunis par rapport au barycentre des sommets sélectionnés et non le graphe complet

4.3 Comparaison avec une solution concurrente

Il est important de noter l'existence d'une solution concurrente qui semble, au premier abord, répondre à une problématique similaire à la nôtre. Cette solution, proposée par J. F. Raymond est disponible ici (cf. Annexe 5), possède néanmoins des différences avec celle proposé par M. Valicov qui sont importantes de mettre en lumière, car elles permettent de nous positionner plus précisément sur le marché.

Suite à une analyse de la solution concurrente nous avons pu déterminer des différences notables entre les fonctionnalités présentes sur notre projet et celles proposées par J.F. Raymond.

Représentation d'un graphe : dans notre projet nous proposons une fonctionnalité qui permet de redessiner le graphe, souvent depuis une disposition ("layout") de sommets appelée la disposition ressort (ou "spring layout"). Dans la solution concurrente, c'est le choix entre différentes dispositions des sommets qui est proposé.

Ajouter des sommets ou des arêtes : Notre solution propose des raccourcis claviers répertoriés par le key-helper qui nous permettent de rajouter des sommets et/ou des arêtes



directement depuis les touches du clavier. Dans la solution concurrente, l'opération est un peu plus fastidieuse puisque l'on appuie d'abord sur un bouton d'ajout de sommets ou d'arêtes pour ensuite les ajouter en cliquant sur le bouton gauche de la souris dans le cas des sommets, ou en cliquant sur deux sommets dans le cas des arêtes.

Zoom : la solution concurrente propose toutefois un bouton "zoom" qui permet de zoomer sur le graphe (fonctionnalité que nous n'avons pas implémentée dans la nôtre). Une fonction qui a pour but de réaliser une action similaire existe toutefois dans notre projet qui permet le redimensionnement automatique et adapté du graphe en fonction de la taille de la fenêtre.

Ajout de bout de chaîne : Cette fonctionnalité proposée uniquement dans la solution concurrente permet d'ajouter automatiquement à la file un bout de chaîne supplémentaire depuis un sommet sélectionné par l'utilisateur.

Ajouter une clique : La solution concurrente propose un ajout de clique sur un graphe donné une fonctionnalité que notre solution ne propose pas.

Mise à part des différences entre les deux solutions, la solution d'interface graphique proposé par J.F. Raymond est une solution visualisation de graphe très sophistiquée, tandis que la nôtre, est une solution n'est pas orienté seulement vers la représentation mais aussi vers la modification de graphes visant à travailler sur ce dernier tout en les modifiant avec une expérience utilisateur agréable.

5. Gestion de projet

Acheminer un projet vers sa réussite est une tâche complexe, d'autant plus lorsque ce dernier allie la rigueur mathématique à la délicatesse de la communication interprocessus. Il faut savoir tirer profit des spécificités, des appétences de chacun et prendre en compte les contraintes inhérentes aux membres de l'équipe. Aussi, il est primordial d'avoir une méthode de travail claire, organisée et stable permettant de poser un cadre autour duquel s'articule le développement du projet. Enfin, le groupe a connu des difficultés managériales lors de certaines phases du projet qu'il est important d'identifier et d'analyser pour assurer la progression future du projet.

5.1 L'équipe en charge du projet

L'équipe qui s'est vu attribuer le projet, est composée de profils à la fois différents et complémentaires. Voici un audit des ressources de chacun des membres :

<u>Identité</u>	<u>Historique</u>	<u>Compétences</u>	<u>Appétences</u>	<u>Faiblesses</u>
TABBAB	DUT GEII	Système	Back-end	Github



Obada		Réseaux Programmation orientée objet	Système et Réseaux	
DUBAN Mathis	BTS : système numérique option électronique et communication	Programmation orientée objet D3js JavaScript	Front-end Web	Python
TSETSERAVA Dzyiana	Master Négociation de Projets Internationaux	Managériale Programmation orientée objet Git et ses outils	Scrum master Back-end	None
EL HIDRAOUI Jawad	Licence STAPS	JavaScript Python Programmation Orientée objet	Front-end Back-end	D3js Github

Des ressources diverses et pertinentes à la réalisation du projet avec des compétences à la fois polarisées vers des thématiques spécifiques et à la fois plus diffuse avec des intérêts vers des domaines variés.

L'enjeu de la gestion du projet sera de mobiliser ces ressources de manière optimale en favorisant à la fois la progression du projet et à la fois la progression personnelle des membres de l'équipe dans leur évolution personnelle. C'est à la méthode de travail que va revenir la tâche d'équilibrer ces deux notions.

5.2 Méthode de travail – SCRUM

POSTULAT DE BASE :

Travailler en groupe, collaborer avec d'autres individus n'est pas une chose aisée et naturelle. Dans tout domaine, lorsqu'un groupe d'individus doit se coordonner en vue de réaliser une performance, on peut commencer à parler de gestion de projets et de gestion d'équipe. À chacun sa stratégie, et il en va de la réussite d'un projet que de choisir la méthode idoine.

C'est pourquoi cette problématique nous est apparue dès les premiers instants, dès la création de l'équipe. L'enjeu était de trouver une méthode qui nous garantissait à la fois un



progrès effectif et une souplesse de travail. En effet, bien des contraintes étaient à prendre en compte lors du choix de la méthode, en voici une liste :

- Nous partions d'une solution déjà existante, on ne pouvait se permettre de recommencer entièrement le projet
- Il fallait que notre méthode de travail puisse nous permettre de nous investir dans notre parcours universitaire
- Il fallait que chacun d'entre nous puisse apprendre et évoluer dans chacun des domaines du projet
- Il fallait pouvoir délivrer des résultats à notre tuteur régulièrement pour qu'il puisse nous faire des retours et ainsi connaître la marche à suivre
- Il nous fallait une certaine souplesse vis-à-vis des contraintes inhérentes aux membres du groupe (vie personnelle, professionnelle, développement personnelle...).

Et il s'avère qu'une méthode allait bien nous être utile.

LA METHODE SCRUM :

Au regard des contraintes énoncées précédemment, nous avons décidé d'opter pour la méthode de gestion de projet SCRUM. Son principe basé sur la livraison itérative de solution nous accorde une souplesse et une flexibilité de travail. Elle se base sur le principe de la livraison itérative des fonctionnalités ("user stories") au client.

La méthode SCRUM nous paraissait la plus pertinente car cela nous assurait qu'il y ait déjà des fonctionnalités prêtes pour la livraison finale du projet. De plus comme notre projet reposait sur le fait que l'on devait améliorer la solution déjà existante en y ajoutant de nombreuses fonctionnalités, SCRUM avec son "backlog" nous correspondait mieux, car il était facile à transformer nos tâches en petites "user stories" quasiment indépendantes.

Nous nous sommes coordonnés sur un rythme hebdomadaire de réunions avec notre tuteur/client M. Valicov afin de recevoir régulièrement son retour sur le travail que nous avons réalisé précédemment.

Ces réunions étaient l'occasion de présenter les fonctionnalités que nous avions implémentées, recevoir le feedback du client, définir les nouvelles tâches à faire et discuter de ce qu'il n'allait pas dans le projet.

Ces rendez-vous marquaient donc la fin d'un sprint et le commencement du prochain et nous permettaient d'alimenter notre backlog avec de nouvelles "user stories".

À la suite de chacun de nos briefings avec notre tuteur nous avons rédigé un compte rendu (cf. Annexe 10)



5.2.1 Récupération du travail existant

Lors du début du projet nous avons récupéré le travail laissé par des étudiants de L3 pro qui était stocké sur leur propre GitHub. La première chose à faire était de nous rajouter en tant que collaborateur afin de pouvoir créer de nouvelles branches et rajouter du code. Nous aurions pu créer un projet de notre côté mais cela n'aurait pas respecté l'essence du projet, car on aurait continué ce projet sans prendre en compte ce que les autres ont réalisé de leur côté, ce n'était pas possible de partir sur cette base-là car on aurait perdu énormément d'informations sur « l'histoire » du projet comme l'historique des “commits”. Nous nous sommes donc tous rajoutés en tant que collaborateurs en faisant le fork du projet pour pouvoir continuer le travail des autres étudiants sans perdre le lien avec le dépôt GitHub original.

Nous avons utilisé GitHub en tant qu'outil de travail collaboratif pour travailler et partager le travail entre tous les membres de notre équipe, car c'était l'outil choisi par les étudiants de L3 pro. Nous avons donc créé 5 branches individuelles (une par personne) et une autre nommée **mergeTest** qui nous a permis de fusionner et tester nos codes provenant des différentes branches pour le push après sur la branche master.

5.2.2 Réunions hebdomadaires et Comptes Rendus

Comme nous avons choisi Scrum comme méthodologie de gestion de projet, nous devions diviser notre travail en sprints et décider des user stories à livrer à la fin de chaque sprint.

Vu que nous avions une réunion avec notre tuteur/client toutes les semaines où toutes les deux semaines, nous avons décidé que chacune de ces réunions sera une fin d'un sprint et donc le temps de présenter des résultats et d'obtenir un “feedback”.

Au cours d'une telle réunion, notre tuteur jetterait un coup d'œil sur ce que nous avons réussi à accomplir jusqu'à présent, donnerait son avis sur ce qui était prêt et ce qui devait être amélioré. Lors de ces réunions, nous avons également reçu de nouvelles fonctionnalités et des idées d'optimisation à ajouter à notre programme.

Dès le début du projet, nous rédigions des comptes rendus de ces réunions enregistrant les fonctionnalités que nous avions livrées avec succès et les nouvelles user stories à ajouter à notre backlog, ainsi que les bugs à corriger et les améliorations à apporter. Ces comptes rendus peuvent être trouvés en annexes (cf. annexe 10).

À la fin de chaque réunion ou au moins le lendemain, nous avions une petite discussion d'équipe concernant les choses que nous devions faire où nous pouvions assigner des missions et partager les problèmes. De temps en temps, notre équipe organisait des réunions plus longues pour que plusieurs personnes puissent travailler ensemble sur une tâche difficile.



5.2.3 GitHub Project Board – Backlog

Mis à part les réunions et les comptes rendus, un projet Scrum a besoin d'un bon "backlog" pour se dérouler sans heurts. Il existe de nombreuses applications qui peuvent être utilisées pour créer et conserver un "backlog", telles que Trello ou GitHub Project Board. Au final, nous avons décidé d'utiliser ce dernier, car il pourrait être lié directement à notre dépôt GitHub. L'outil lui-même mérite quelques mots. Nous avons utilisé une version bêta du Project Board et cela s'est avéré être un moyen très pratique d'organiser nos "user stories". Le "backlog" peut être affiché sous forme de "board" (avec les catégories *Todo*, *In progress* et *Done*) ou de tableau (où vous pouvez immédiatement voir à qui la story est attribuée et quelles sont sa complexité et sa valeur en chiffres). Vous pouvez voir notre "backlog" en annexes (cf. annexe 9).

L'un des principaux avantages du Project Board était la possibilité pour chaque membre de l'équipe de voir quelles "user stories" n'avaient pas encore été assignées et de choisir celles sur lesquelles il souhaitait travailler.

Une autre fonctionnalité intéressante offerte par GitHub est la conversion des "user stories" en "issues". De telles stories pourraient être référencées dans des "commits" avec un hashtag et un numéro de l'issue unique. Le statut de ces stories pourrait également être changé en *Done* directement à partir d'un commit, en écrivant quelque chose comme "closed #4" ou "fixing #11" dans le message de commit. Nous n'avons pas beaucoup exploité cette fonctionnalité, mais c'était quand même une expérience intéressante et potentiellement.

5.3 Difficultés rencontrées

Au fil du déroulement du projet nous avons rencontré différentes difficultés qui nous ont ralenti, ces erreurs nous serviront de leçon pour nos futurs projets.

Notre organisation du Git/GitHub

Au fil de la réalisation de ce projet, nous avons eu à faire des choix concernant la gestion du versionnage du projet, pour pouvoir faciliter la fusion du code de chacun.

Au fil du temps nous nous sommes concentrés sur le travail en autonomie et nous avons mis de côté l'intégration continue du code. Dans les mois qui ont suivi, notre tuteur nous a fait remarquer (**le 30 mars 2022**) que l'intégration continue du code était essentiel pour l'avancement du projet. Lors de la fusion de toutes les branches nous avons rencontré énormément de conflits sur les fichiers édités sur toutes les branches. Cela a été coûteux en temps et en ressources, puisque notre équipe a passé deux semaines à réconcilier les branches et tout intégrer au sein d'une seule branche appelée "mergeTest".



Également nous nous sommes trop concentrés sur la réalisation du code sans prendre en compte la rédaction du rapport qui pourtant est une des étapes les plus importantes du projet dans sa globalité.

Nous aurions dû nous forcer à moins nous concentrer sur le code et prendre du temps chaque semaine pour réaliser une rédaction incrémentale du rapport pour avoir plus de temps pour finaliser le rapport.

Perturbation externe au projet :

Au cours de la réalisation de notre projet tutoré nous avons eu à réaliser plusieurs projets liés au cours de notre formation de DUT, ce qui nous a obligé de mettre en pause le projet tutoré pendant quelques semaines, durant lesquelles nous n'avons pas contacté notre tuteur.

CONCLUSION

À travers ce projet nous avons pu entrevoir le logiciel SageMath tout en découvrant comment un projet complexe pouvait interagir avec le monde fascinant des graphes. Nous avons appris à travailler sur une solution déjà existante réalisée par une autre équipe tout en ajoutant de nouvelles fonctionnalités à travers des demandes réalisées par un client (représenté par notre tuteur de Projet M.Valicov).

Réaliser ces différentes tâches nous a permis de mettre en lumière certains points clés de gestion d'équipe qui nous ont aidés à corriger les points où nous avions des lacunes (notamment en gestion de projet, nécessitant une bonne compréhension des enjeux).

Ce projet a été très instructif car il nous a permis à chacun d'évoluer dans plusieurs domaines dans lesquels nous n'avons pas forcément l'habitude de travailler. Cette motivation de permettre à un maximum de personnes d'utiliser notre solution pour traiter les graphes a été notre moteur et nous souhaitons de tout cœur que notre projet servira à de nombreuses personnes.



Annexes

1. Rapport technique du projet JS_Graph réalisé par les étudiants de L3 pro :

https://github.com/Projet-SageMath-Graphs/JS_Graph_Sage/blob/master/Rapport/Projet%20Sage%20Graph.pdf

2. Lien vers le dépôt GitHub avec le projet des étudiants de L3 pro :

[NaokimTheFirst/JS_Graph_Sage \(github.com\)](https://NaokimTheFirst/JS_Graph_Sage.github.com)

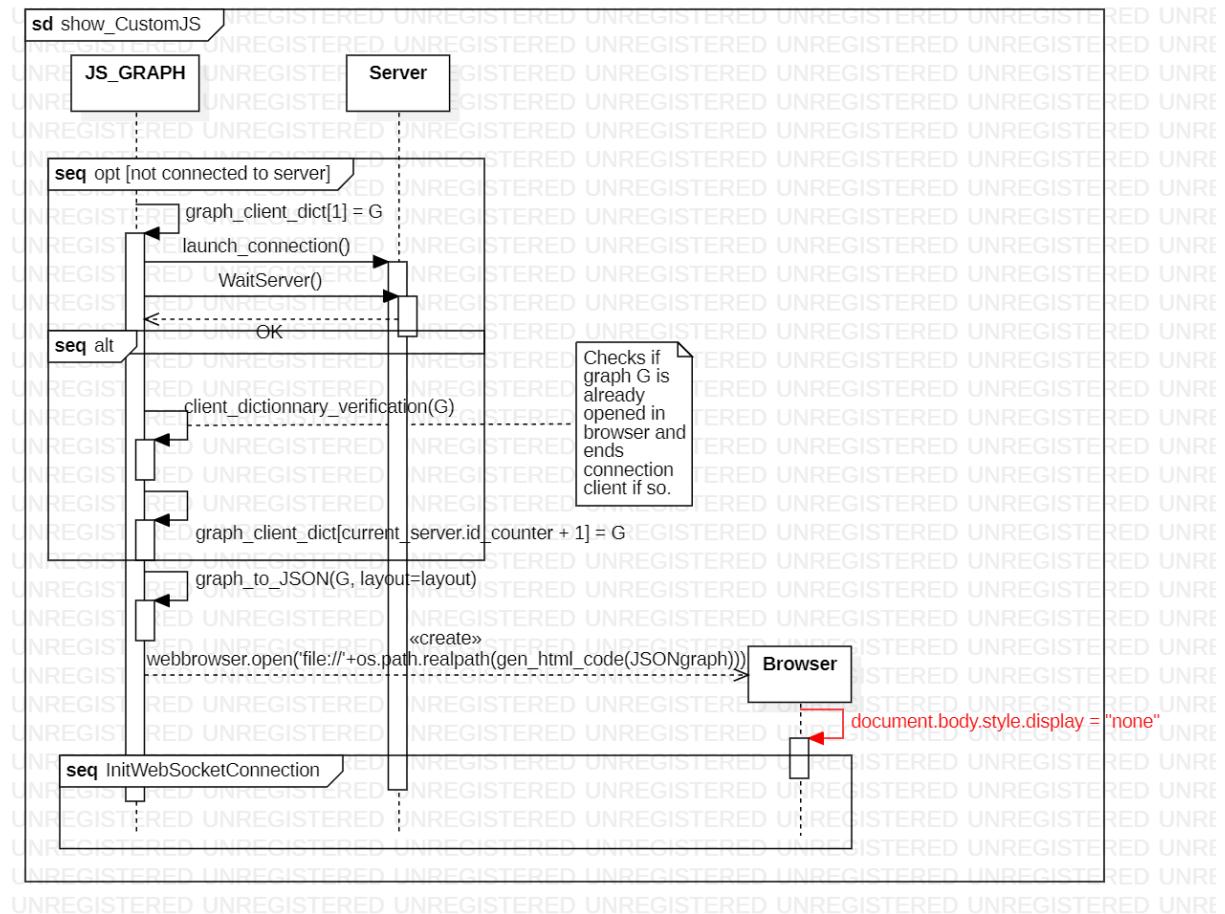
3. Licence GPL-GNU : La licence publique générale GNU, ou GNU General Public License (son seul nom officiel en anglais, communément abrégé GNU GPL, voire simplement « GPL »), est une licence qui fixe les conditions légales de distribution d'un logiciel libre du projet GNU.

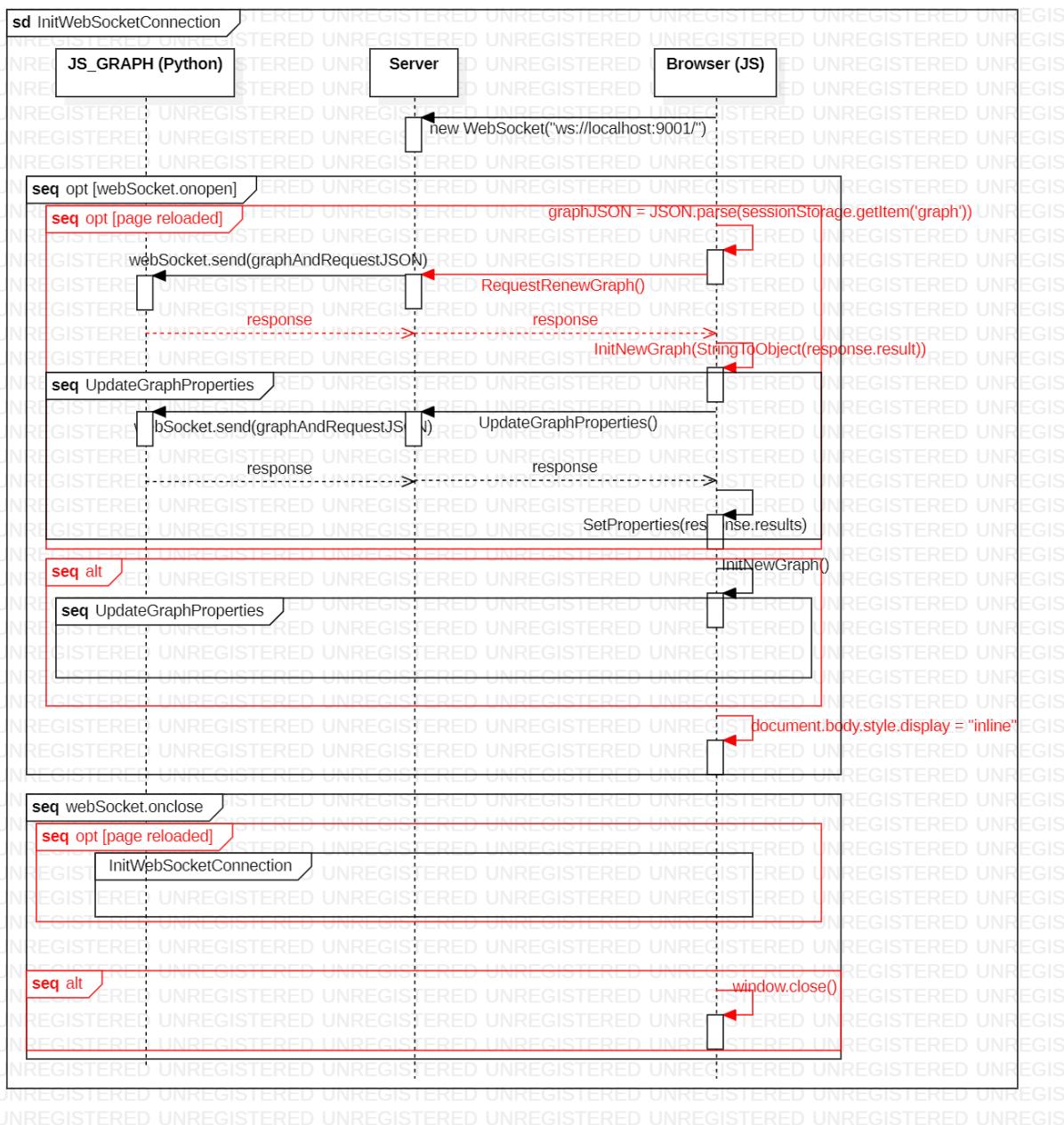
4. Lien Téléchargement Sage : <https://doc.sagemath.org/html/en/installation/>

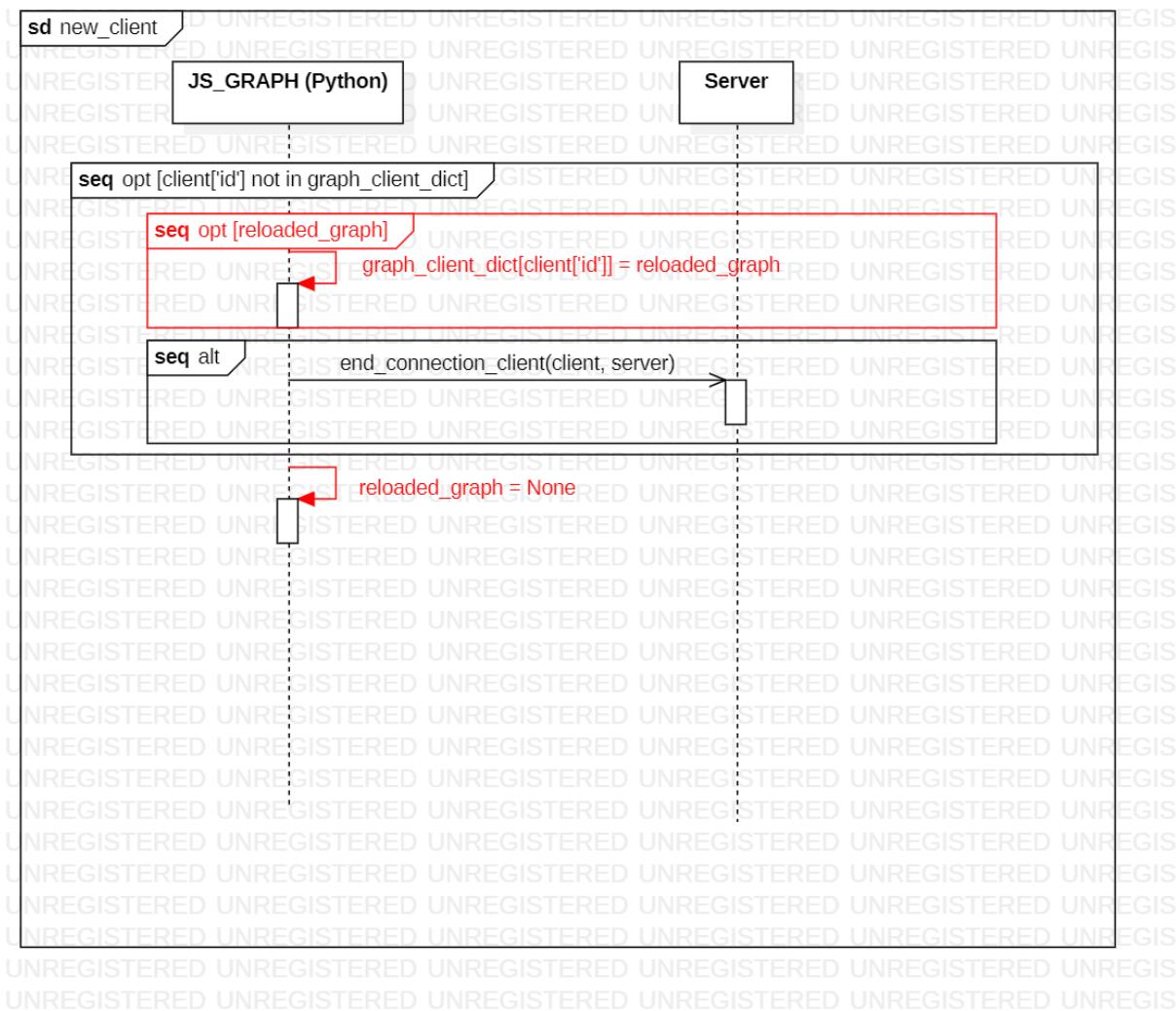
5. Solution de J. F. Raymond : <https://github.com/jfraymond/phitigra>

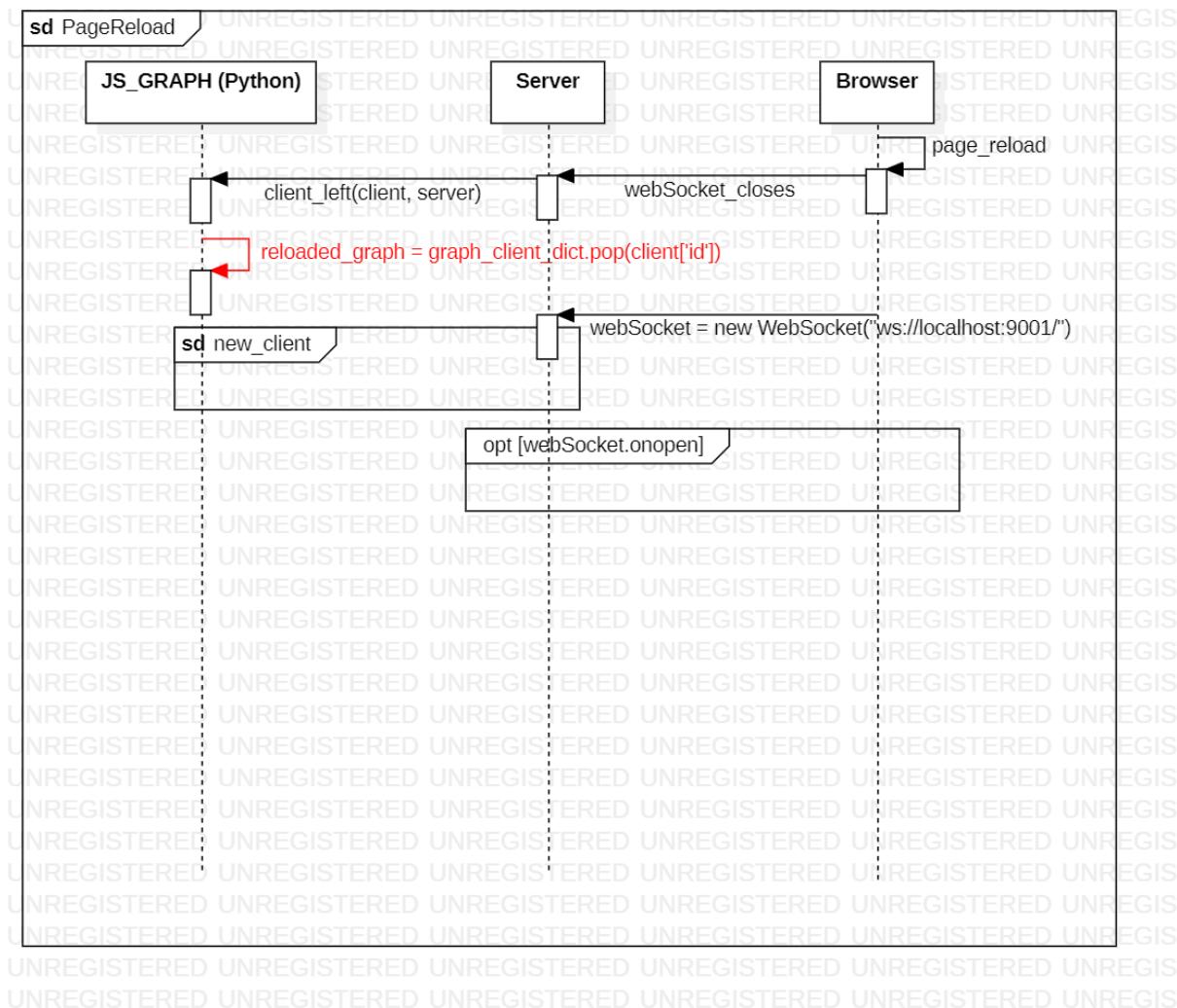
6. Sequence diagrams – Connexion terminal

En rouge : les changements faits pour permettre de recharger la page.

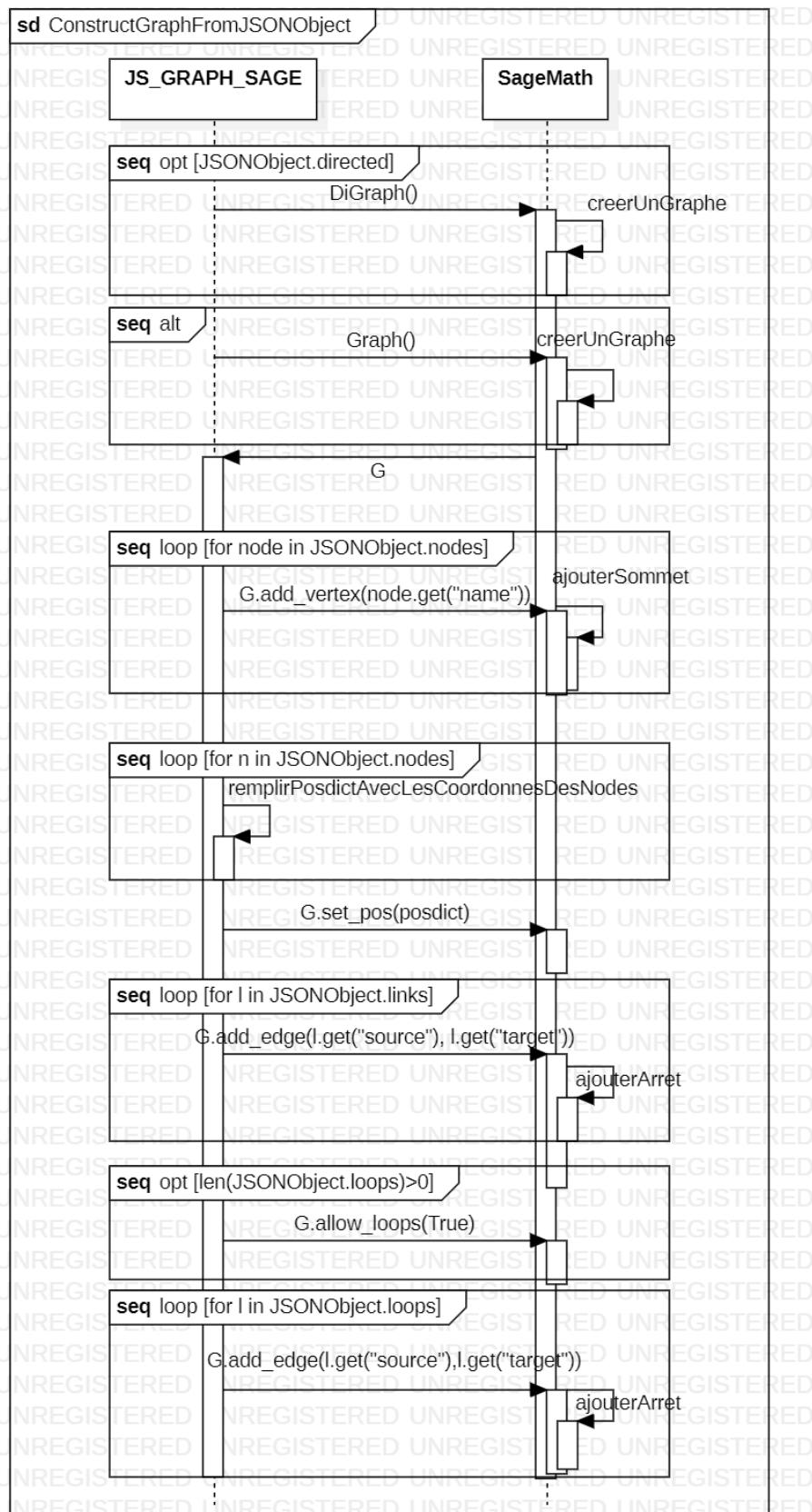


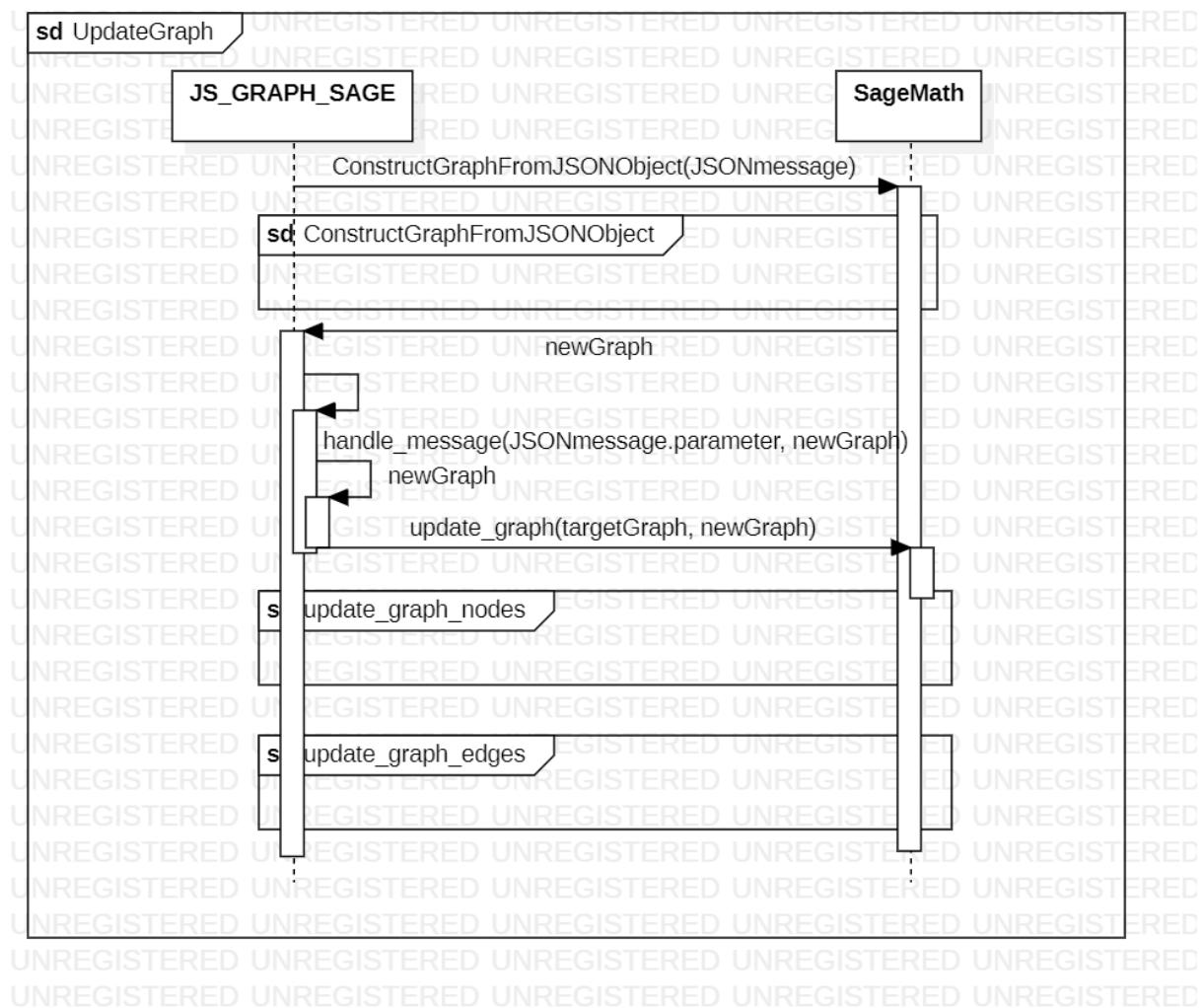


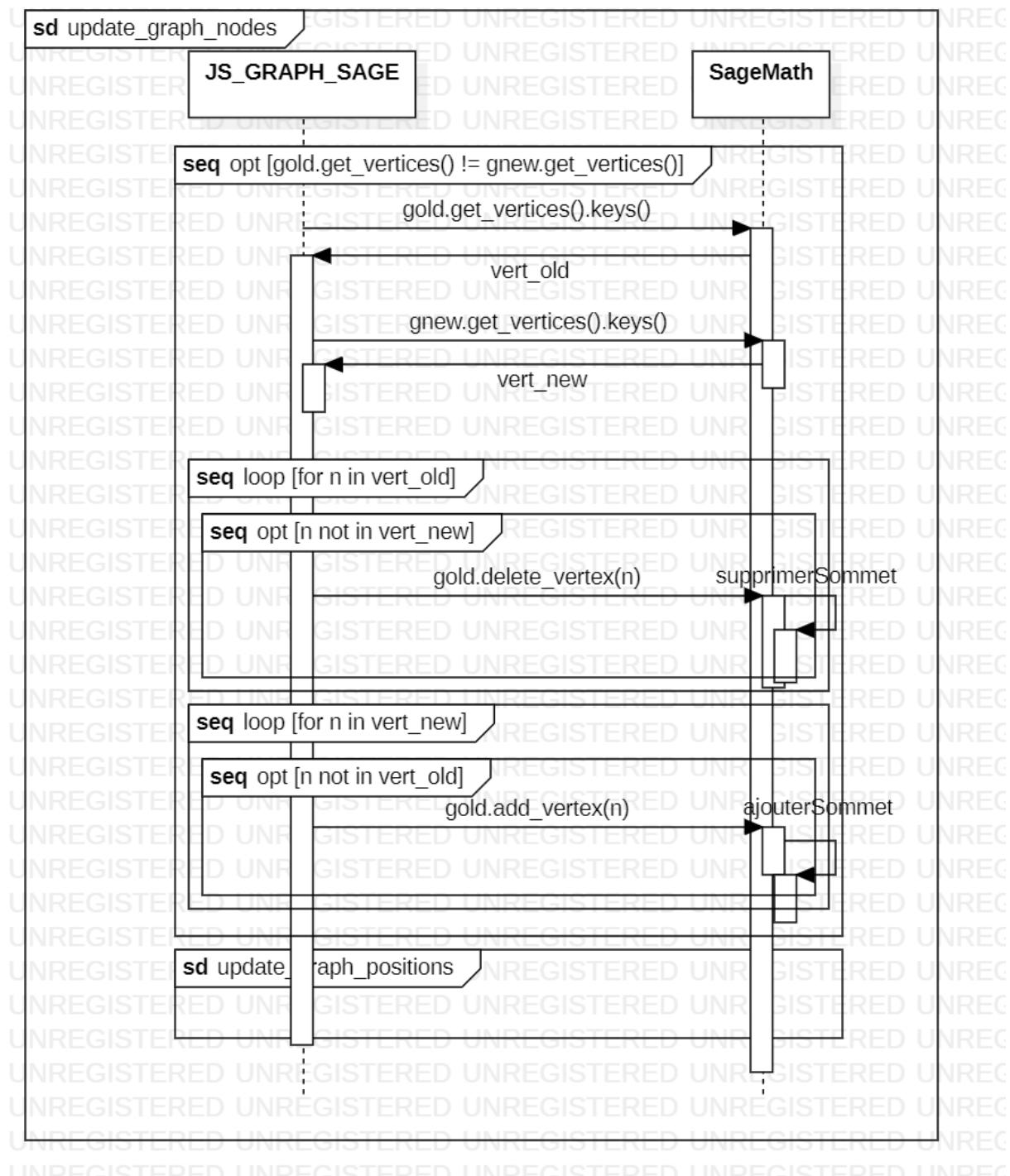


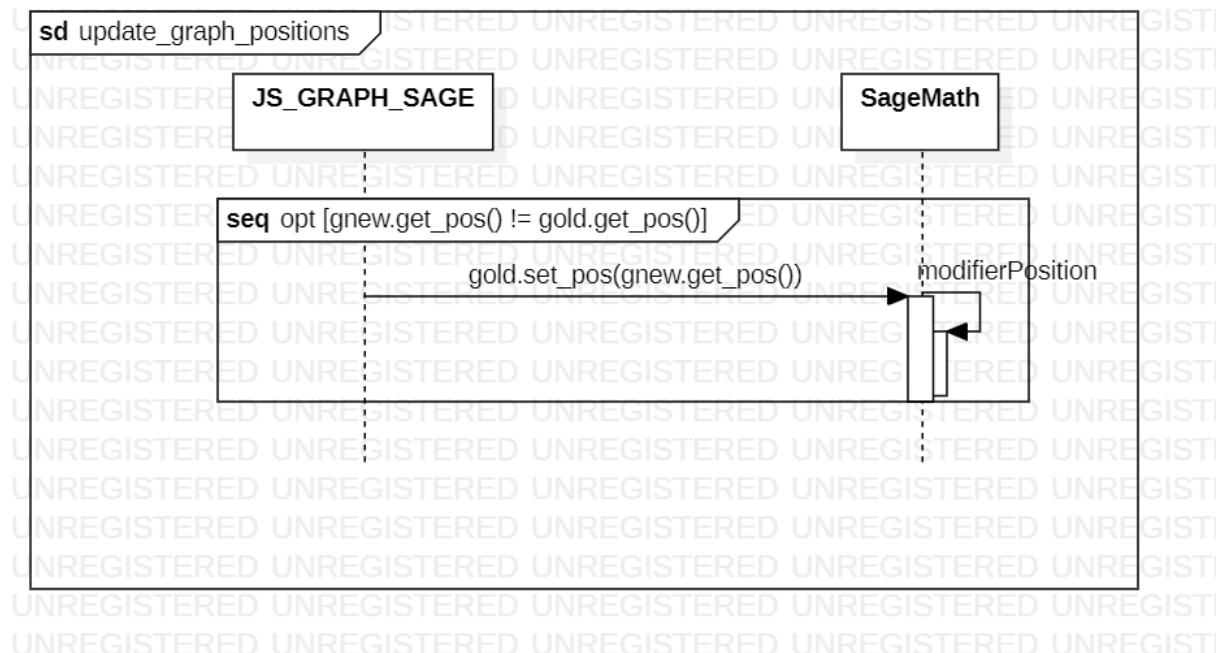


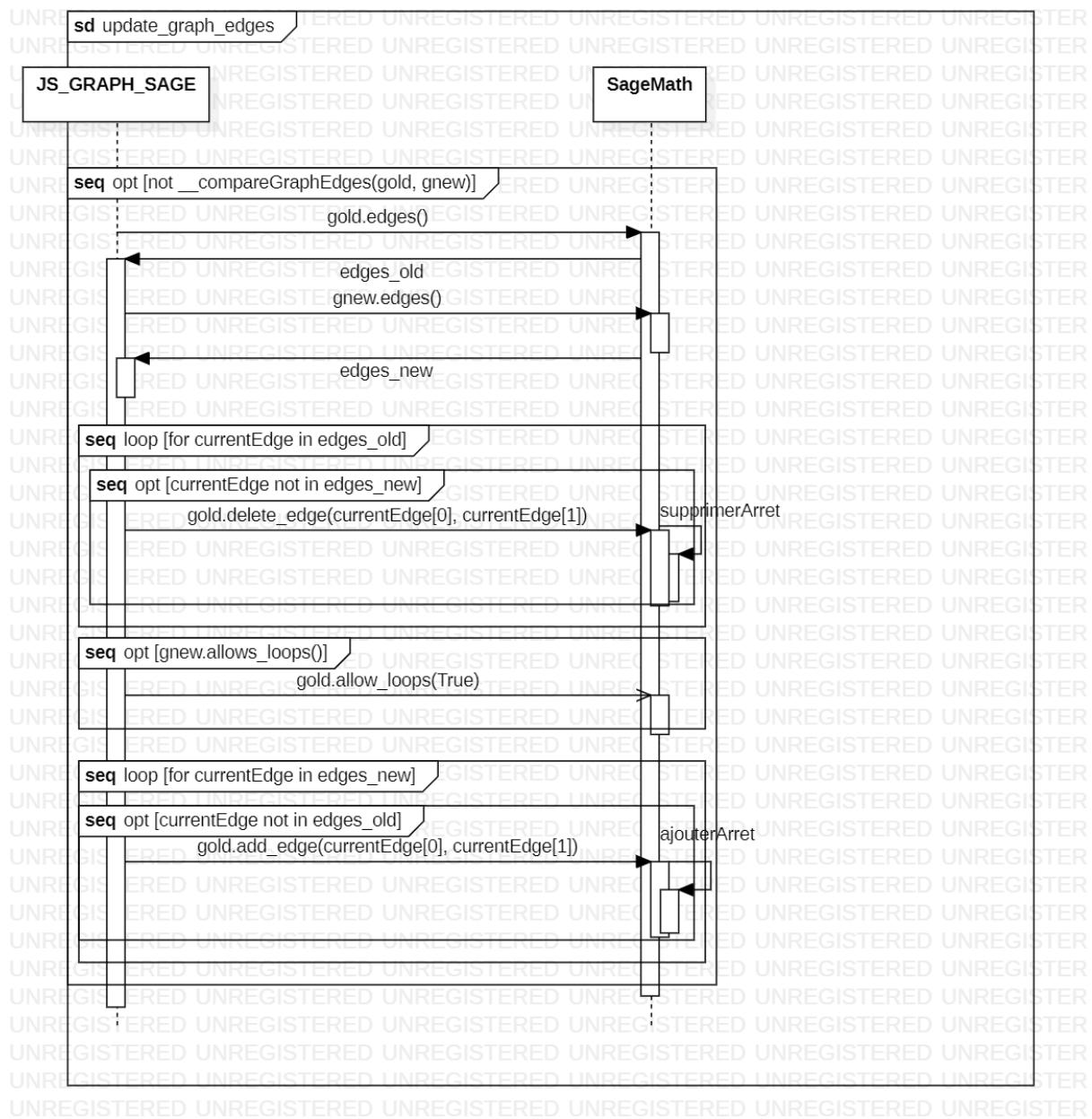
7. Sequence diagrams – Graph Update



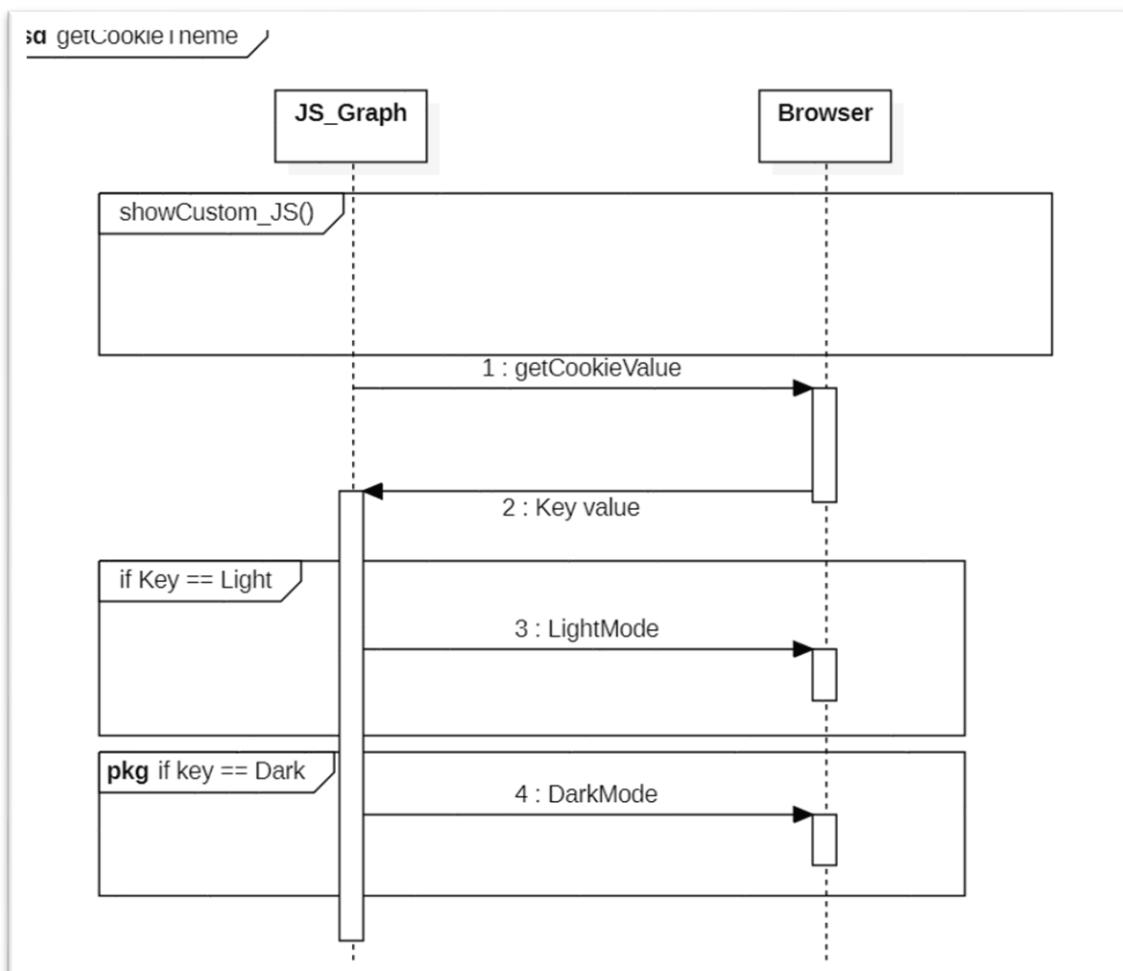


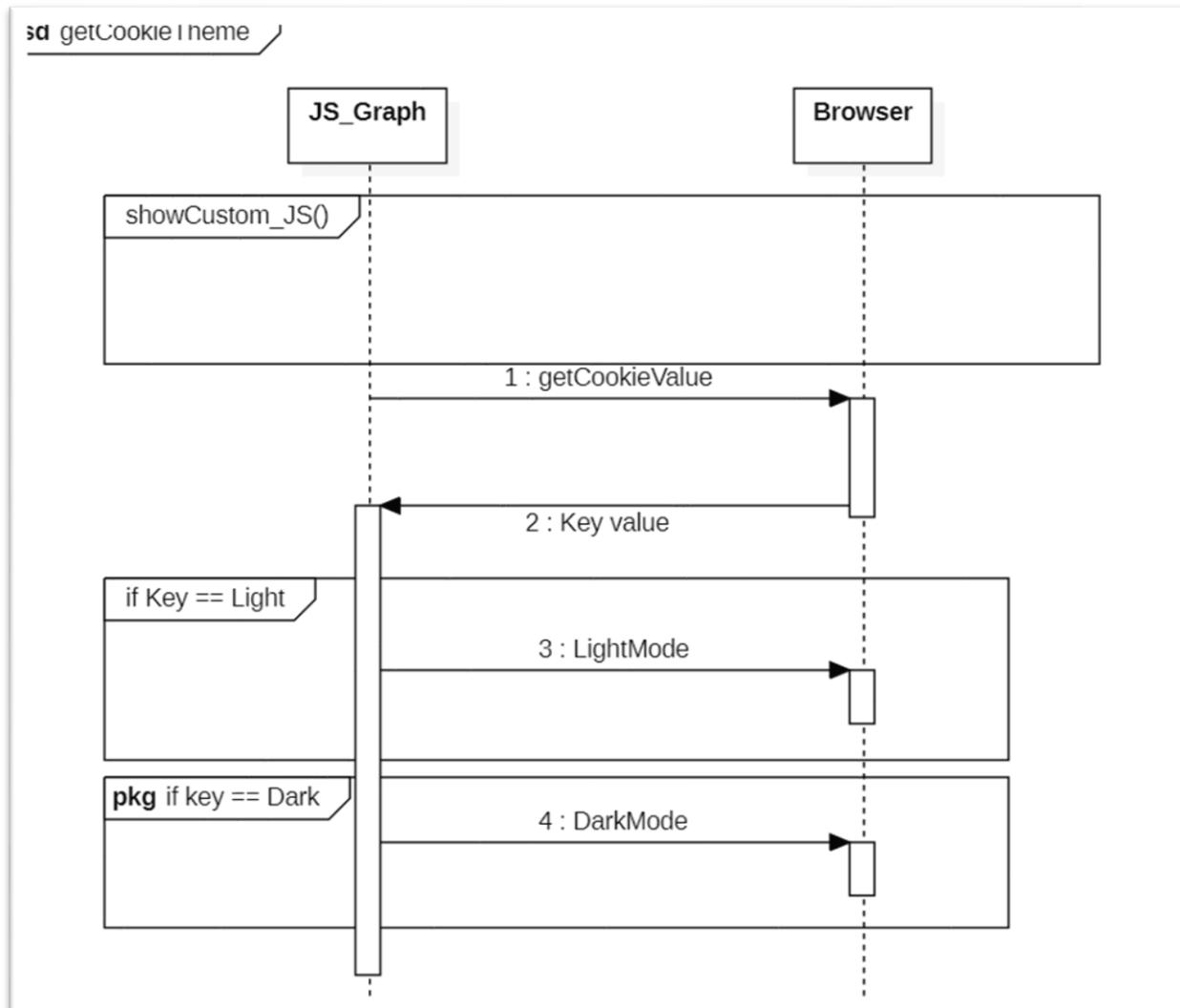






8. Sequence diagrams – Get cookieTheme







9. Backlog :

⊕ Projet JS_Graph_Sage

Table Board + New view

Title	Assignees	Status	Complexity	Value
1 ⚡ Modifier le fonctionnement d'attach	Dalwaj	Done	1	5
2 ⚡ Permettre le drag d'une sélection groupée	Dalwaj	Done	3	3
3 ⚡ Diagrammes de séquence sur update	sea-gull-diana	Done	2	1
4 ⚡ Ajouter le nb des sommets / arêtes dans le menu gauche	Akkuun	Done	2	3
5 ⚡ Mapping des sommets pour préserver leur type original	sea-gull-diana	Done	1	4
6 ⚡ Diagrammes de séquence sur le processus de connexion	sea-gull-diana	Done	2	1
7 ⚡ Connexion inverse entre SageMath et interface graphique	sea-gull-diana	Done	4	4
8 ⚡ Améliorer l'animation du drag d'une sélection	sea-gull-diana	Done	2	1
9 ⚡ Keeping websocket connection on window reload	sea-gull-diana	Done	5	4
10 ⚡ Faire fonction end_connection(graph)	ObadaTabbab	In Progress	2	1
11 ⚡ Zoom in / zoom out du graphe + scrolling responsif	Dalwaj	Todo	4	3
12 ⚡ Encoder un graph en G6 et l'afficher sur la page	Dalwaj	Done	2	2
13 ⚡ Possibilité de questionner le site The House of Graphs si le graphe obtenu est déjà connu	Dalwaj	Done	4	1
14 ⚡ Afficher l'info sur edge connectivity	Akkuun	Done	1	1
15 ⚡ Afficher les degrés maximal et minimal	Akkuun	Done	1	1
Title	Assignees	Status	Complexity	Value
16 ⚡ Afficher l'info sur girth	Akkuun	Done	1	1
17 ⚡ Utiliser un nouveau port si 9001 est occupé	ObadaTabbab	Done	5	5
18 ⚡ Possibilité de choisir un sous-graphe et faire un graphe complémentaire	ObadaTabbab	Todo	3	2
19 ⚡ Bloquer l'ajout d'un sommet en cliquant 'A' si les autres sommets ne sont pas de type int	sea-gull-diana	Todo	2	3
20 ⚡ Afficher 'is eulerian'	Akkuun	Done	1	1
21 ⚡ Afficher chromatic index et number (comme bouton) (Hard Stuff)	Akkuun	Done	2	1
22 ⚡ Afficher hamiltonicity : is_hamiltonian() (Hard Stuff)	Akkuun	Done	2	1
23 ⚡ Montrer un spanning tree (directement sur le graphe : marquer les arêtes avec couleur)	sea-gull-diana	Done	3	1
24 ⚡ Afficher vertex-connectivity	Akkuun	Done	2	1
25 ⚡ Possibilité de customiser l'interface (couleurs, paramètres affichés dans le menu...)	Akkuun	Done	5	2
26 ⚡ Trouver un moyen de remplacer les getters dans InterfaceAndMisc.js par un seul getter	sea-gull-diana	Todo	1	1
27 ⚡ Possibilité de sortir du terminal de sage avec exit	ObadaTabbab	Todo	2	2
28 ⚡ Migration de D3JS	sea-gull-diana	Done	10	5
29 ⚡ Scrolling du menu		Todo	3	2
30 ⚡ Optimiser les méthodes de coloration des arêtes et des sommets	sea-gull-diana	Done	3	3



31	<input checked="" type="checkbox"/> Afficher les classes de la coloration si l'utilisateur le demande (Hard Stuff)	 sea-gull-diana	<input type="button" value="Done"/>	2	2
32	<input type="checkbox"/> Création de la documentation (rapport du projet)	 Akkuun, Da...	<input type="button" value="Done"/>	10	10
33	<input type="checkbox"/> Implementer 'edge contraction' : remplacer la partie sélectionnée par un sommet	 ObadaTabbab	<input type="button" value="Done"/>	4	2
34	<input type="checkbox"/> Bouton pour afficher maxClique (Hard Stuff)	 Akkuun	<input type="button" value="Todo"/>	2	2
35	<input type="checkbox"/> Bouton pour afficher max independent set (Hard Stuff)	 Akkuun	<input type="button" value="Todo"/>	2	2
36	<input checked="" type="checkbox"/> Modification: assurer qu'on n'ajoute pas les coordonnées aux sommets par défaut	 sea-gull-diana	<input type="button" value="Done"/>	3	3
37	<input checked="" type="checkbox"/> Fonctionnalité freeze : ajouter les positions courantes aux sommets si on tape 'F'	 sea-gull-diana	<input type="button" value="Done"/>	3	3
38	<input type="checkbox"/> Appliquer layout 'spring' du Sage lors d'affichage du graph	 Dalwaj	<input type="button" value="Done"/>	2	3
39	<input type="checkbox"/> Repositionner et redimensionner le graphe on window resize	 sea-gull-diana	<input type="button" value="Done"/>	5	4
40	<input type="checkbox"/> Fonctionnalité de save (possibilité de désactiver auto sauvegarde)	 Dalwaj	<input type="button" value="Done"/>	4	3

10. Comptes Rendus

Manipulation interactive des graphes avec Sage

Le logiciel SageMath, est un logiciel connu libre et open-source de calcul mathématiques très populaire auprès des scientifiques. Il contient une grande partie implémentée concernant les graphes et leurs algorithmes (le backend programmé en Python). En revanche le frontend est actuellement assez limité et l'utilisateur doit presque entièrement passer par la ligne de commandes. On peut visualiser les graphes en différents formats, et notamment sur un navigateur sous forme d'objets graphiques avec la librairie Javascript d3js, mais on ne peut pas les manipuler "à la main". L'an dernier une équipe d'étudiants de l'IUT avait produit une solution pour résoudre ce problème et permettre une manipulation interactive des graphes à travers un navigateur. Leur solution est ici :

https://github.com/NaokimTheFirst/JS_Graph_Sage

Le but du projet est de terminer le travail en enrichissant cette solution. Il s'agit d'ajouter des nouvelles fonctionnalités interactives de manipulations des dessins et les intégrer au projet SageMath. Une possibilité serait de programmer des déroulements d'algos de Sage existants sur un navigateur. Technos : Python (avec SageMath), Javascript, Git.

Tuteur : Petru Valicov

24/01/2022 - 2e rdv avec tuteur

- Creation d'un fork : https://github.com/Dalwaj/JS_Graph_Sage
- Pull des modifications pour 'attach' par Jawad
- A faire : Diagramme de classes (+ séquences), ajouter les fonctionnalités à l'interface graphique (afficher nombre de sommets, déplacer la partie du graph sélectionnée...)

03/02/2022 - 3e rdv avec tuteur

Problèmes à résoudre : - animation du drag d'une multiselection (à voir si c'est lié au renouvellement trop rapide du fenêtre), - perte de connexion lors de renouvellement du fenêtre, - absence de connexion inverse entre SageMath et JS_Graph, - les types des sommets sont transformés en string lors de la connexion.



Solution proposée pour le problème des strings : créer un dictionnaire (une map) qui associe à chaque sommet v du graphe initial une chaîne de caractères cv qui est sa représentation textuelle que vous allez utiliser dans le format JSON. Toutes les opération que vous allez effectuer dans d3js sur cv seront transposé sur v à travers ce dictionnaire.

10/02/2022 - 4e rdv

Nouvelles choses à faire : - voir les fichiers *things-to-add.md* et *ToDo.md*. - enovoyer le graph en JSON de Python à JavaScript pour remplacer le graphe precedent avec innerHTML (mission déjà accomplie :) . - pouvoir relancer la page sans perdre la connexion.

Travail fait : - Mapping des sommets dans un dict pour preserver les types d'origin. - Possibilite de modifier le graphe dans ls terminal et importer les changements dans l'interface graphique avec le bouton “Renew Graph”.

18/02/2022 - 5e rdv

Nouvelles choses à faire : - Pour tout le monde : 1. faire le menage sur le dépôt en effaçant les fichiers de configuration (personnels), du type : .vscode, .idea, .class, .o etc. 2. faire git rm --cache obj/result.html 3. ajouter à gitignore : .idea/ .vs/ .vscode/ *.class *o 4. faire git add . et git commit - Comprendre comment D3.js marche dans le code et faire migrer vers une version plus récente. Voir les liens : <https://observablehq.com/@d3/d3v6-migration-guide> <https://blog.devgenius.io/d3-js-whats-new-in-version-6-5f45b00a85cb> - Permettre à l'utilisateur de sortir du terminal sage avec exit. - Trouver un moyen de remplacer les getters dans InterfaceAndMisc.js par un seul getter. - Les propriétés “lourdes” (qui prennent le temps pour que le sage les compte), ne doivent être affichées que si l'utilisateur les demande explicitement. Voir *things-to-add.md/Hard Stuff*. - Chercher le moyen à copier le project board GitHub à un autre compte (organisation ou utilisateur) pour pouvoir le lier au dépôt GitHub. - Pour l'affichage de girth : Bug avec graphs.ClawGraph(), car girth est infinie. Il faut passer une exception dans le code.

Travail fait : - Possibilite de modifier le graphe dans ls terminal et importer les changements dans l'interface graphique avec le bouton “Renew Graph” (Bouton à renommer et repositionner). - Possibilité de refraichir la page sans perdre la connexion (permet d'importer les changements de même manière que “Renew Graph”). - Refraichir la page ou “Renew Graph” peuvent être utilisés pour repositionner le graph au centre de l'écran et optimiser ça taille. - Diagramme de séquence pour le processus de connexion (mes changements dans le code original sont marqués en rouge). - Affichage de plusieurs nouvelles propriétés du graph dans le menu : degrés max et min, is eulerian, hamiltonicity. - Affichage du graph sous format G6. - Possibilité de questionner le site *The House Of Graphs* si le graphe obtenu est déjà connu. - Project Board crée sur GitHub pour le backlog de nos user stories (à déplacer). Voir : <https://github.com/sea-gull-diana/projects/1/>

25/02/2022 - 6e rdv

Nouvelles choses à faire : - Changer ownership du dépôt vers une organisation (et copier-coller le project board vers cette organisation). - Comment attribuer des portes ? Coder en dur ou choisir n'importe quel porte disponible (voir si les portes sont regroupées par famille



et on peut choisir une famille à utiliser). - Changer la méthode de coloration optimale. Utiliser la fonction du sage plus optimale (voir email). - La division en groupes à enlever (fonctionnalité non finie, donc le champ du groupe dans le menu sert à rien). - La première ébauche du rapport à faire et envoyer à M. Valicov pendant les vacances.

Travail fait : - Connexion aux portes différentes selon la disponibilité (bugs à fixer voir au-dessus). - Spanning tree (coloration des arêtes d'un arbre couvrant).

30/03/2022 - 7e rdv

Nouvelles choses à faire : - Faire une réunion du groupe pour merge tout ce qu'on a fait sur GitHub (LE PLUS IMPORTANT - faire avant le rdv suivant). - Encore quelques boutons à ajouter (voir Project Board). - Edge contraction

Travail fait : - Affichage des classes de coloration sous la forme du texte - Bug fixé dans l'affichage de l'arbre recouvrant - Plan du rapport préparé ; - Redaction du rapport commencé : [voir document sur OneDrive](#).

11/04/2022 - 8e rdv

Nouvelles choses à faire : - Separer dans le menu Orientations et Algorithmic/Hard Stuff - Resoudre le probleme : les sommets recoivent les coordonées qu'ils n'ont pas a l'origin Lorsqu'on dessine le graph avec show_CustomJS, on ajoute les coordonées aux sommets, l'objet change donc de nature. **Solution :** Faire un bouton("save") qui permet de fixer l'embedding (la possibilité de freeze les coordonées des sommets). Mais si on ne clique pas ce bouton, les sommets ne doivent pas avoir des coordonnes. **Fonctions utiles :** set_pos() pour donner les coordonnees aux sommets. graph.set_embedding() position des sommets par rapport l'un a l'autre. - Utiliser le layout spring de Sage par defaut dans show_CustomJS pour bien dessiner le graphe. - Edge contraction : plutot que graph.contract_edge(0,1), utiliser la methode graph.merge_vertices()

Travail fait : - merge des branches effectué (branche obada et les dernieres versions des branches jawad et mathis en attente) - Problème de "Girth" réglé, mais pas encore ajouté à master - Bug corrigé dans l'affichage des graphes avec des sommets de type objet

Prochaine reunion : mercredi 20/04 a 9h00 ##### 20/04/2022 - 9e rdv *Nouvelles choses à faire :* - Nettoyer le code - Merge des sommets (effectuer le merge sur newgraph et pas targetgraph, puis l'importer vers interface). - Migration D3.js - Restructurer l'interface - Merge de tout ce qui reste sur la branche master.

Travail fait : - Fonctionnalité Save (autosauvegarde par defaut, possible de le désactiver en cliquant 'U' et sauvegarder les changements en cliquant 'S'). - Fonctionnalité Freeze positions en cliquant 'F' (layout spring par defaut s'il n'y a pas de positions fixes). - Dark et Light Mode avec l'utilisation des cookies pour sauvegarder les préférences de l'utilisateur.



03/05/2022 - 10e rdv

Nouvelles choses à faire : - Regarder la fonctionnalité de Zoom dans la solution [IPE](#), voir si on peut faire le même. - Changer la taille des sommets on window.resize. - Penser comment régler le problème de sommets qui sortent de la fenêtre quand on diminue sa taille. - Fixer la taille du menu. - Is Hamiltonian mettre en *Hard Algorithmes*. - Penser à afficher le cycle Hamiltonian (comme on fait pour le spanning tree). - Mettre Girth et les 2 connectivités en *Properties* - ce n'est pas Hard stuff. - Dans la section *Hard algorithmes* mettre : Coloring, chromatic, is Hamiltonian. - Le reste mettre dans la section *Oriented graphs*. - Corriger les bugs de coloration et de *Redraw Graph* avec le graph de type graphs.HortonGraph(). - Ne pas appeler UpdateGraphProperties lorsqu'on bouge des sommets ou les renomme. - Remplacer is_tree() par is_forest() dans le calcul de Girth. - Décider si on veut faire une demande pour présenter notre solution pour les Sage Days à Montpellier (la semaine du 13 juin). - Voir [phitigra](#) par J. F. Raymond - solution qui permet de manipuler les graphes comme le notre. Se positionner par rapport à lui. Voir si son truc est trop dépendant de Jupiter.

Travail fait : - Problème d'affichage des graphes vides corrigé. - L'interface graphique () s'adapte à la taille de l'écran on window.resize(). - Amélioration des styles de dark et light modes (quelques bugs corrigés).

09/05/2022 - 11e rdv

Nouvelles choses à faire : - Garder seulement la fonctionnalité de 'U' (renommer comme "On/Off automatic save"), pas besoin de 'S' (on peut obtenir le même résultat en appuyant 'U' 2 fois). - Pour le "merge" des sommets prendre le premier sommet aléatoirement pour ne pas rendre le processus trop compliqué. - Ne pas redessiner le graphe lors du merge et mettre le nouveau sommet résultant du merge dans le centre du rectangle formé par les sommets fusionnés. Algorithme pour faire un merge sans la fonction sage : liste = [] s = set() for i in liste: s.union(g.neighbors(i)) s.difference(liste) - Ajouter des interlignes dans Key Helper pour une mise en page plus claire. - Renommer "Subdivide Edge" dans Key Helper en "Subdivide Edges". - Enlever le resize manuel du menu dans css. - Ne pas afficher les classes de coloration en texte après "Redraw". - Edge coloring ne marche pas après la subdivision des arêtes. - Ne pas permettre à l'utilisateur de déplacer les sommets en dehors de la fenêtre !!! - Renommer "Exist" en "Exists on House of Graphs" (mettre House of Graphs sous le style d'un lien). - Repositionner le bouton G6 et la zone de texte.

Travail fait : - Resize des sommets et arêtes on window resize. - UpdateGraphProperties n'est plus appelé lors de la repositionnement des sommets. - Bugs du HortonGraph corrigés. - Taille du menu fixé. - Améliorer le positionnement du graphe dans le centre (problème du décalage vers le droit fixé).



19/05/2022 - 12e rdv

Nouvelles choses à faire : - Reécrire les positions des sommets dans command stack (utilisés pour Ctrl+Z) chaque fois qu'on appelle la fonction center_and_scale. - Augmenter la taille de menu. - Rendre la taille l'interface graphique égale à celle de la fenêtre. Ne pas avoir de l'interface derrière le menu (actuellement on peut créer un sommet derrière le menu et c'est un bug. - Ajouter un checkbox 'Allow Multiple Edges' dans le menu. - Renommer Key Helper en Key Commands. - Rendre Ctrl+Z possible après le merge des sommets. - Limiter le nombre de changements sauvegardé dans le command stack. - Fixer les bugs de G6 window style. - Ne pas permettre d'ajouter les changements sur interface avant RedrawGraph si on a fait des changements dans le terminal sage. - Bug avec les sommets qui portent le même nom.

Travail fait : - Migration d3js de v3 à v7. - Drag de la selection (bug fixé). - Resize du graph lors du resize de la fenêtre. - Problème avec l'ajout des sommets dans un graph avec des sommets positionnés réglé. - Coloration des arêtes après edge subdivide. - Renommage des boutons. - Merge des sommets.

25/05/2022 - 13e rdv (dernier)

Nouvelles choses à faire (au moins essayer si on a du temps) : - Corriger un bug: quand on change la taille de la fenêtre, les éléments selectionnés ne sont plus selectionnés, mais sont toujours rouges. - Ctrl+Z pour merge ou assurer que Ctrl+Z fonctionne bien pour les autres éléments après merge. - Mettre coloration et chromatic dans la section Hard Algorithmes. - Augmenter la taille du menu. - Enlever tous les logs du terminal avant la présentation finale.

Travail fait : - Bug de Ctrl+Z après redimensionnement du fenêtre corrigé (presque). - On est en train de rédiger notre rapport.

THE END