PROJET INFO 1A : COMPTE RENDU

2020-2021
Grenoble INP - Phelma
Sacha HEITZ - Hugo BRISSET

Introduction

Les structures humaines les plus élaborées et les plus complexes qu'il existe sont sans aucun doute les ***réseaux***. Qu'ils s'agissent de réseaux internet, de réseaux de transports ou encore de réseaux énergétiques, ces structures, faites de ***nœuds*** et ***d'interconnexions***, sont d'une incroyable complexité. Pour les modéliser, on utilise en informatique une structure abstraite bien connue nommée ***graphe***. Cette dernière est particulièrement adaptée à l'ensemble des problématiques de réseaux. Elle se constitue d'un ensemble de ***sommets*** ou de nœuds, reliés entre eux par des ***arcs***. Ces derniers peuvent être orientés ou non selon les problématiques. Nous allons nous intéresser ici à la modélisation par le graphe de réseaux de transport routier et ferroviaire (métro à l'échelle urbaine) dans le but d'établir le ***plus court chemin*** (ensemble d'arcs) entre deux nœuds du graphe. Pour ce faire, nos arcs seront pondérés, c'est-à-dire que chaque arc possède un ***poids*** qui lui est propre, le poids d'un chemin correspondant à la somme des poids des arcs de ce chemin. Nous allons donc nous intéresser dans un premier temps à l'implantation du type abstrait de données graphe puis à l'implantation du calcul de plus court chemin par deux algorithmes différents : ***Astar*** et ***Dijkstra***. Nous nous intéresserons ensuite à l'étude comparative des performances de chacune de nos implémentations.

I. Implantation

1. Etat du logiciel

Notre logiciel actuel, nommé ***pcc***, est fonctionnel et permet de rechercher le plus court chemin entre deux sommets d'un graphe par l'algorithme Astar et Dijkstra, qu'il s'agisse d'un réseau routier ou métropolitain.
Notre programme propose de sélectionner les sommets de départ et d'arrivée par leur numéro pour les réseaux routiers ou par leur nom pour les réseaux de métro.
Le graphe est lue à partir de son chemin sur la machine actuelle qui est demandé à l'utilisateur, notre logiciel est donc portable d'une machine à l'autre.
Ces choix se font par le biais d'une interface utilisateur textuelle, qui s'exécute en ligne de commande.
Notre logiciel intègre aussi un affichage graphique du graphe et du résultat de plus court chemin obtenu à l'aide de la bibliothèque SDL. Il intègre évidemment un affichage textuel du chemin en console afin d'indiquer sommet par sommet le chemin à suivre.

2. Structures de données

- Ensemble **Atraiter** : C'est l'ensemble des sommets que l'on a identifiés comme étant toujours à traiter, c'est à dire que l'on à toujours pas trouvé le chemin optimal pour y arriver. Cette structure nécessite notamment des fonctionnalités d'ajout, de modification et de recherche du plus petit élément. Nous avons donc fait le choix d'un ***tas*** (*heap_t*) trié par ordre de coût croissant afin de représenter cet ensemble. Ainsi, nous sommes assurés que l'élément de coût le plus faible sera toujours en tête du tas. On aura alors qu'à le récupérer pour avoir le plus petit élément, cela afin d'éviter un parcours long et fastidieux pour chercher le plus petit élément.

- Ensemble **Atteint** : C'est l'ensemble des sommets que l'on a déjà atteint, c'est à dire que l'on à déjà trouvé le chemin optimal pour y arriver. Cette structure nécessite notamment des fonctionnalités d'ajout et de test de présence. Nous avons décidé d'utiliser un ***tableau de pointeurs sur sommets*** (*tabver_t*). Chaque pointeur sera stocké dans l'élément du tableau correspondant à l'indice du sommet sur lequel il pointe. Ce choix est un peu gourmand en mémoire car il nécessite de prévoir un tableau de la taille du nombre de sommets du graphe, mais il nous permet un gain de temps lors du test de présence car il suffit alors de regarder l'emplacement du tableau correspondant à l'indice du sommet que l'on veut tester.

- **graphe.data** : Cette structure correspond aux données du graphes, c'est à dire que c'est elle qui va stocker les sommets et les arcs qui constituent le graphe que l'on étudie. Pour cette dernière, le sujet nous donnait déjà le choix à effectuer : ***un tableau de sommet*** (*vertex_t**), chaque sommet stockant lui-même l'ensemble des arcs dont il est le départ. Dans l'ensemble des autres structures et pour des raisons d'optimisation mémoire, nous utilisons alors des pointeurs sur sommet afin que ces derniers ne soient pas dupliqués inutilement et qu'ils soient stockés une fois et une seule dans ce tableau.

- **vertex.edges** : Il s'agit de l'ensemble des arcs partant du sommet vertex. La structure la plus adaptée à cet ensemble est évidemment la ***liste chaînée dynamique*** d'arcs (*listedge_t*). La liste chaînée dynamique permet un parcours simple et est optimisée du point de vue de la mémoire car elle contient uniquement le nombre de maillons nécessaires et est facilement modifiable (ajout ou suppression d'un maillon).

- **predecesseurs** : Cet ensemble regroupe les prédécesseurs de chaque sommet présent dans Atteint, c'est-à-dire que pour chaque sommet atteint, on stock le sommet précédent permettant d'arriver à ce sommet depuis celui de départ de manière optimisé. Cet ensemble nous sert ensuite à construire le plus court chemin en départ et arrivée en remontant de prédécesseur en prédécesseur depuis le sommet d'arrivée. Nous avons opté pour un ***tableau de pointeur sur sommet*** *(tabver_t)* où l'on stock le pointeur dans la case du tableau correspondant à l'indice du sommet dont il est le prédécesseur.

- **path** : c'est l'ensemble des sommets constituant le plus court chemin. On le construit à partir du tableau de pointeurs sur sommet **predecesseurs**. Nous avons naturellement choisi d'utiliser une ***pile de pointeurs sur sommets*** (*lifover_t*), contenant de haut en bas les sommets du départ jusqu'à l'arrivée. Nous n'avons plus alors qu'à dépiler les sommets un par un pour reconstruire le plus court chemin.

Nous avons alors défini à l'aide de ces Types Abstraits de Données les structures suivantes : arcs (*edge_t*), sommet (*vertex_t*) et graphe (*graph_t*).

3. Tests et exécution

Afin de vérifier le bon fonctionnement de notre code, nous avons procédé à des tests de chaque module au fur et à mesure qu'ils étaient réalisés. Cette méthodologie nous a permis d'avoir en permanence un travail fiable, et de ne pas avoir à tout débugger d'un seul coup, ce qui complique généralement considérablement l'identification des sources d'erreurs. Pour ce faire, nous avons créé des fichiers sources de test pour chacun des modules, dans lesquels nous exploitons les différentes fonctions du module afin de vérifier leur fonctionnement.

Bien que suffisant pour vérifier le fonctionnement de nos codes dans le cas général, ces tests ne couvrent pas l'ensemble des cas de figure et nécessiteraient d'être complétés afin de s'assurer de la robustesse du code quel que soit le cas de figure rencontré.

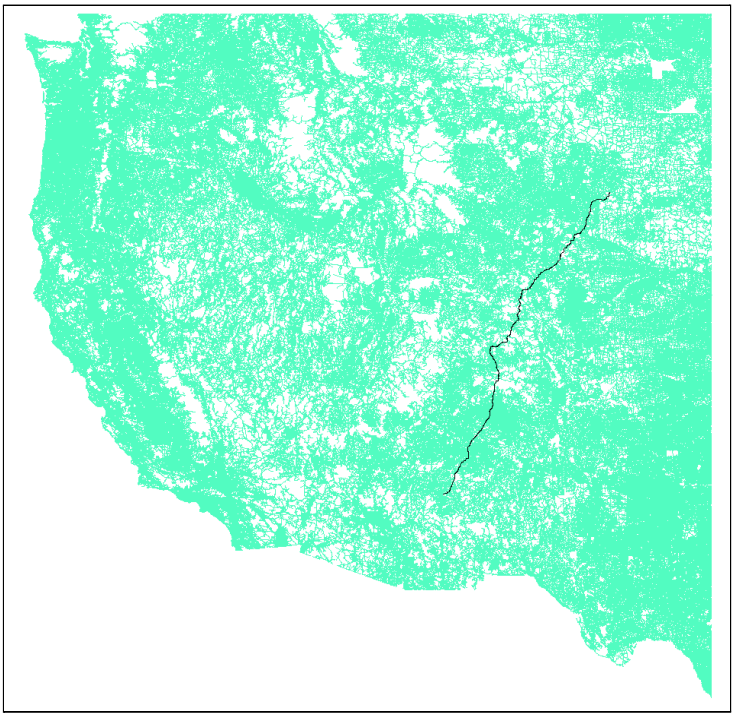
Pour exécuter nos deux algorithmes, nous avons écrit plusieurs codes :

- Une interface dite "utilisateur", ***pcc.c***, permettant de choisir dans la console l'algorithme utilisé, le graphe à étudier et les sommets de départ et d'arrivée.
- Deux programme dit "développeur" (***test_astar.c*** et ***test_dijkstra.c)*** qui se lance en ligne de commandes et dont il faut changer les paramètres directement dans le code source permettant d'exécuter respectivement les algorithmes Astar et Dijkstra afin de les tester, de les déboguer et de les améliorer.

Voici un exemple de l'exécution en console du programme principal **pcc** :

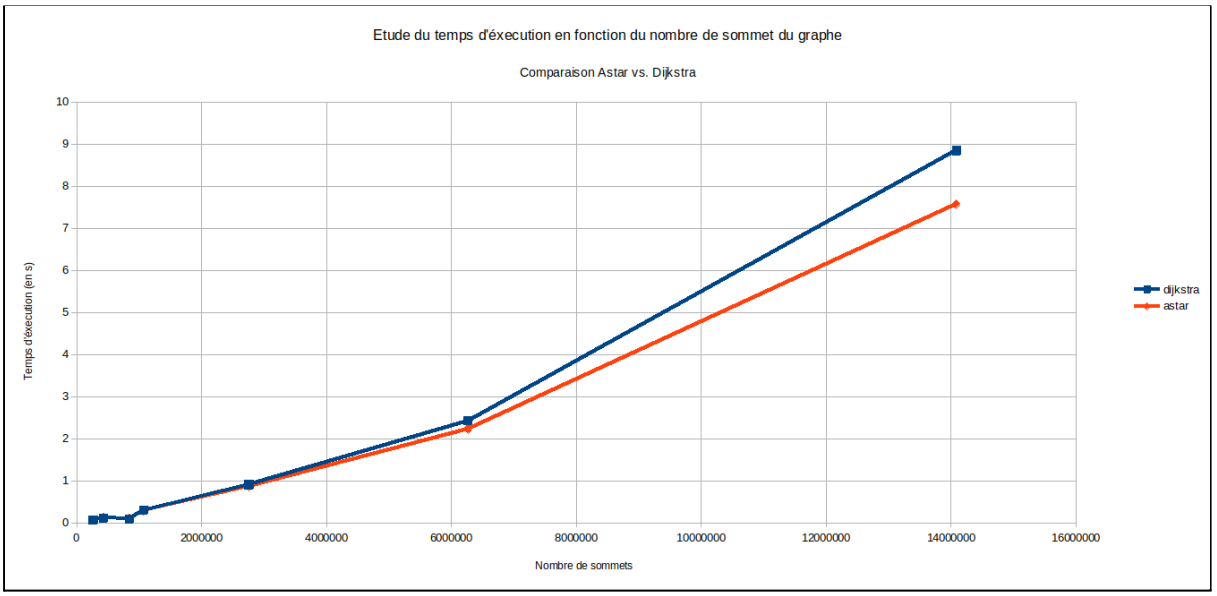
```
sacha@MBP-de-Sacha heitzsbrissethprojetS2 % bin/pcc
Recherche par nom de stations ou par numéro de sommets?
1:nom
2:numéro
>> 2
Veuillez saisir l'emplacement du graphe à traiter
>> /Users/sacha/Downloads/DATA/grapheUSAOuest.txt
Voulez vous utiliser Astar ou Dijkstra ? [A/D]
>> A
Veuillez choisir un numéro de sommet de départ (entre 0 et 6262103)
>> 6587
Veuillez choisir un numéro de sommet d'arrivee (entre 0 et 6262103)
>> 4895632
Le poids du plus court chemin vaut : 14112672.000000
Le plus court chemin pour aller de 6587 à 4895632 est :
Le sommet Sommets6588 numero 6587
Le sommet Sommets6590 numero 6589
```

Et voici le résultat graphique obtenu :



4. Analyse de performances

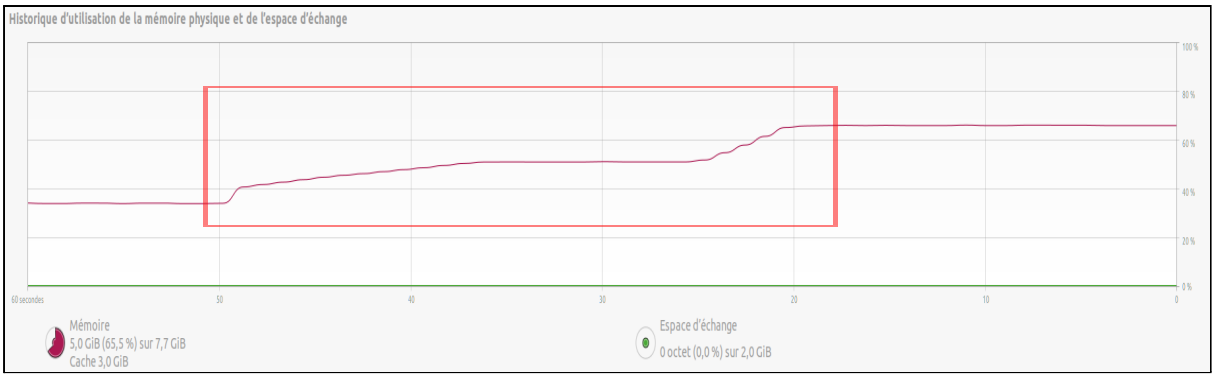
Une fois l'ensemble de notre code opérationnel, nous nous sommes penchés sur ses performances, et notamment sur le temps d'exécution moyen en fonction du nombre de sommets du graphe. Pour ce faire, nous avons utilisé le module ***time.h*** afin de mesurer le temps d'appel des fonctions Astar et Dijkstra sur 300 répétitions afin d'obtenir un temps moyen d'exécution moyen en fonction du nombre de sommets du graphe étudié. Afin d'avoir une image fiable du temps d'exécution, les sommets de départ et d'arrivée sont choisis au hasard à chaque répétition grâce à la fonction ***random()***. Pour nous faciliter la tâche, nous avons écrit un code ***test_temps.c*** qui mesure le temps moyen d'appels aux fonctions astar et dijkstra pour un graphe donné. Nous avons alors obtenu les résultats suivant :



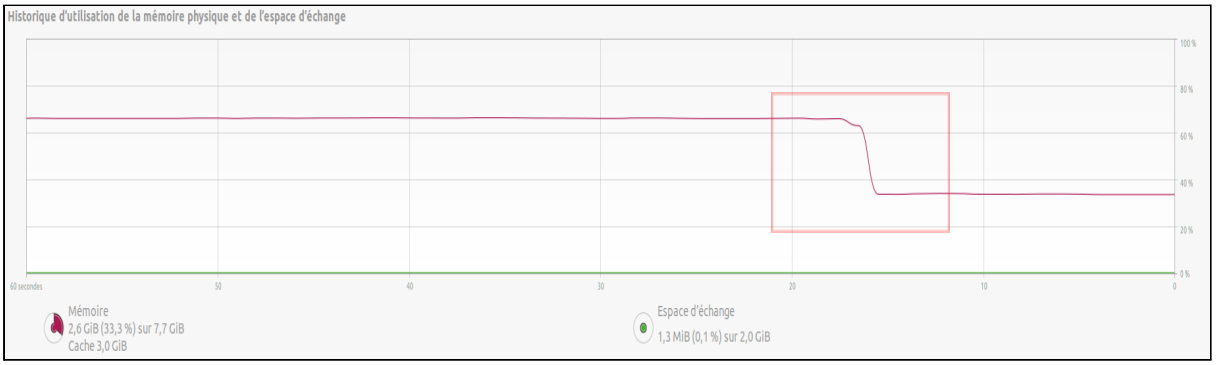
Les deux algorithmes semblent avoir une efficacité d'exécution temporelle quasi similaire, bien que l'algorithme Astar semble légèrement plus performant à mesure que le nombre de sommets du graphe augmente. Leur complexité respectives semblent se rapprocher d'un *O(log(n))*. Cette analyse ne prend pas en compte la densité du graphe étudié, c'est-à-dire le nombre d'arcs par rapport au nombre de sommets. Il convient aussi de préciser que le code de mesure du temps d'exécution ne mesure que le temps mis pour l'appel à la fonction Astar ou Dijkstra, et qu'il est indépendant de la fonction de lecture du graphe, qui est de toute façon commune aux deux algorithmes.

Nous nous sommes aussi penché sur la gestion mémoire de notre code grâce à l'outil ***valgrind***. Cet outil nous a permis de corriger de nombreuses allocations non-libérées et ainsi de ne pas gaspiller de mémoire. Cependant, certaines allocations, notamment dues à la librairie *SDL_Phelma*, restent non libérées sans que nous sachions réellement comment les libérer.

D’un point de vue consommation mémoire, notre code est assez gourmand, notamment car nous avons opté pour un tableau de pointeurs sur sommet de la taille du nombre de sommet contenu dans le graph pour l’ensemble ***Atteint***. Cependant, afin d’éviter de dupliquer inutilement les informations stockées, toutes les structures manipulant des sommets autres que le tableau de sommets du graphe utilise des pointeurs sur sommets (*vertex_t**) ou des entiers (*int*). Lors de l'exécution des deux algorithmes Astar et Dijkstra sur le graphe de 1,4 GB *“grapheUSACentral.txt”*, nous avons pu constater qu’environ 6 GB de mémoire étaient utilisés, soit environ 4 fois la taille du graphe étudié. Un PC possédant une mémoire RAM d’au moins 8GB est donc théoriquement capable de faire tourner les algorithmes sans être limité. Voici un exemple de la consommation de mémoire lors de l’exécution d’Astar sur le graphe *“grapheUSAOuest.txt”* :



Augmentation de la mémoire utilisé



Mémoire libérée à la fin du programme

On remarque que l’on consomme environ 30% de la mémoire vive disponible (8GB) soit 2,4GB. Le fichier du graphe faisant 600MB, on retrouve bien le facteur 4 trouvé précédemment.

II. Suivi

1. Organisation

Lors de la première séance de projet, nous avons pris connaissance du sujet et nous sommes attelés à découper ce dernier en plusieurs parties et à se répartir le travail à faire. Nous avons alors identifié les différentes tâches suivantes :

- Coder les différentes structures de données : liste, tas, etc ... (traitée par Hugo)
- Ecrire les scripts de lecture du fichier et de construction du graphe a partir du fichier texte (traitée par Sacha)
- Ecrire les différentes fonctions de test des différents modules (traitée par Hugo)
- Implémenter les scripts Astar et Dijkstra (traitée par Hugo et Sacha)
- Ecrire des tests de chacun des deux algorithmes (traitée par Sacha)
- Faire une fonction permettant de tester le temps d'exécution des deux algos (traitée par Hugo)
- Permettre l'affichage graphique du réseau et du résultat (traitée par Hugo)
- S'occuper d'adapter les algorithmes pour les métros : recherche par nom, affichage différent, etc … (traitée par Sacha)
- Ecrire un programme "utilisateur" propre permettant d'exploiter les différents scripts développés (traitée par Hugo)

Évidemment, ce découpage est théorique et même s'il a été relativement bien respecté, nous nous sommes évidemment aidés mutuellement sur de nombreux problèmes que nous rencontrions chacun de notre côté.

Nous nous sommes régulièrement réuni afin de faire un point sur l'avancement des différentes tâches et nous avons aussi eu souvent besoin de s'expliquer mutuellement le fonctionnement de nos codes afin de pouvoir les prendre correctement en main et de les compléter.

Nous avons aussi dû nous coordonner par rapport au travail sur GitLab afin de ne jamais écraser le travail de l'autre et d'être certains de ne jamais travailler simultanément sur les mêmes fichiers. Enfin, nous avons opté pour un éditeur de texte partagé en ligne afin d'écrire ce rapport de manière commune. Cela nous a permis de compléter les parties que nous avions traitées et de gagner en efficacité.

2. Outils de développement

Sous ***Linux Ubuntu*** (Hugo Brisset) nous avons utilisé l'IDE ***Atom***, qui intègre une gestion avancée des dépôts ***GitLab,*** l'utilitaire de partage et de gestion des versions que nous avons utilisé pour collaborer. La compilation et l'exécution des programmes a été intégralement réalisée en ligne de commande grâce au ***terminal Ubuntu-Gnome***. Les programmes ont tous été compilés avec ***GNU Compiler Collection*** (GCC). Pour les tests de gestion mémoire, nous avons utilisé le logiciel ***Valgrind***.

Sous ***MacOS*** (Sacha Heitz) nous avons utilisé les mêmes outils que sous linux à ceci près que le terminal utilisé était celui de ***MacOS X*** . Pour les tests de gestion mémoire, nous avons utilisé le debugger ***GDB*** puisque valgrind n'est pas disponible sous MacOS.

3. Problèmes rencontrés

A propos des outils, nous avons rencontré au début du projet quelques difficultés avec l'utilisation du Makefile, notamment avec l'utilisation des bibliothèques graphiques, et il nous a fallu un peu de temps pour appréhender correctement cet outil. Nous avons aussi rencontré quelques problèmes lors de l'installation de la bibliothèque SDL Phelma sous Mac OS, notamment en raison de la localisation des fichiers .h sur la machine.

D'un point de vue du code en lui-même, nous n'avons pas rencontré de réel problème, si ce n'est pour la recherche par nom des sommets lorsque ces derniers contiennent des espaces. Lorsque le nom du sommet contient un ou des espaces et que ce dernier est recherché comme sommet d'arrivée, le programme ne trouve pas le sommet correspondant, alors qu'il le trouve s'il s'agit d'une recherche pour le sommet de départ. Nous n'avons malheureusement pas eu le temps de nous pencher plus sur la question et n'avons donc pas su identifier la cause de ce problème. En revanche, nous avons trouvé que le sujet était assez complexe de part la multitude de programmes et de modules différents qu'il nous fallait produire, et notamment par le nombre de structures imbriquées les une dans les autres qui demandent une grande rigueur lors de l'appel d'un paramètre particulier d'une des structures emboîtées.

4. Enseignements

Nous avons beaucoup appris de ce projet, notamment du point de vue de l'organisation, de la coordination et de la gestion du timing d'un projet . Nous nous sommes rendus compte de l'importance d'une bonne répartition des tâches, et d'une communication régulière afin de toujours être au fait de ce que fait l'autre.

Nous avons aussi appris à organiser un projet fait d'une multitude de programmes et de modules différents, mais aussi à nous adapter au travail et aux méthodes de notre binôme. Nous avons dû apprendre à collaborer ensemble sur Git et à se coordonner et se synchroniser pour ne jamais écraser le code de l'autre.

Enfin, d'un point de vue technique, nous avons beaucoup appris de l'utilisation de différents outils tels que Valgrind et GDB, mais aussi sur la manipulation de la bibliothèque SDL.

Il reste encore de nombreux points d'améliorations à notre travail. Dans un premier temps, il serait nécessaire de rendre l'ensemble de nos codes plus robustes afin qu'ils ne plantent jamais quelle que soit la situation rencontrée, car nous n'avons fait que très peu de gestion d'erreur par manque de temps. Il serait pour cela intéressant de développer plus les tests effectués sur chaque module et chacun des deux algorithmes.

Afin d'étoffer un peu plus le projet, nous avions imaginé créer une interface graphique permettant à l'utilisateur de choisir ses points de départ et d'arrivée en cliquant directement sur le graph affiché à l'écran. Cela nous permettrait de manipuler la gestion d'événements souris qui nous semble particulièrement intéressante. Cette interface graphique impliquerait d'implémenter un code permettant de zoomer sur la carte affichée car les graphes fournis sont généralement illisible à petite échelle du fait du nombre de sommets et d'arcs qu'ils contiennent.

Conclusion

Ce projet nous a tous les deux beaucoup intéressé et nous a permis de découvrir en profondeur la notion de graphe que nous ne connaissions pas du tout. Il nous a permis de développer aussi bien nos compétences en informatique que nos compétences transversales telles que l'organisation, la gestion de projet ou la communication. Il nous a demandé beaucoup de rigueur et une certaine efficacité dans notre collaboration. Nous avons beaucoup appris et avons pris beaucoup de plaisir à construire et développer l'ensemble de ce projet, et avons notamment apprécié la satisfaction lorsque nous arrivions à un résultat visuel concret et un code opérationnel.

Finalement, ce projet nous a tous les deux confortés dans nos choix d'orientation pour l'année prochaine : SEOC.