Rapport de Soutenance 1 -

Team MAMAROCH
Nathan Chevalier
Sachintha Mallawa Tantirige Dias

InfoSpe Epita

Table des matières

1 Introduction **3**
1.1 Présentation du projet 3
1.2 Edition des cartes 4
1.3 Intérêts et enjeux 5

2 Etat de l'art **6**
2.1 Applications/Editeurs d'itinéraire 6

3 Réalisation **10**
3.1 Implémentation des graphes 10
3.1.1 Principe 10
3.1.2 Listes 11
3.1.3 Implémentation des graphes 13
3.1.4 Lecture et enregistrement 15
3.1.5 Ajouts et améliorations 17
3.2 Interface 18
3.3 Site Web 20

4 Organisation pratique **23**
4.1 Choix Techniques 23
4.2 Répartition des tâches 23
4.3 Planning prévisionnel 23

5 Conclusion **24**

1 Introduction

1.1 Présentation du projet

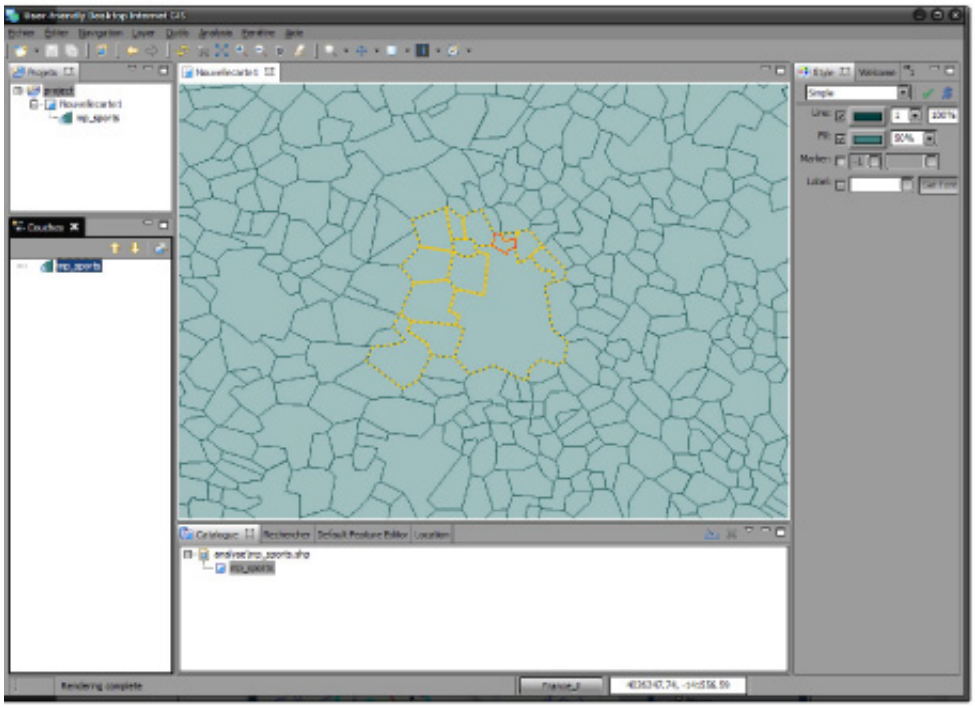
Le but de ce projet est de créer une application de planification des trajets du métro parisien. L'application devra trouver une ou plusieurs alternatives pour rejoindre un point d'arrivée depuis un point de départ. La principale difficulté sera de pondérer les arrêtes en fonction des temps de trajet ainsi que de prendre en compte toutes les stations où des changements de lignes peuvent être effectués.

Les nombreuses lignes de métro et les interconnexions augmentent fortement le nombre de possibilités pour passer d'un point A à un point B, mais une seule est la plus rapide. Le but d'une telle application étant de faire gagner du temps, l'algorithme devra privilégier des trajets qui pourrait sembler peu logique en premier abord.

1.2 Edition des cartes

La plus-value de cette application est l'intégration d'un système d'édition des réseaux de transport urbains. En effet, pour pouvoir s'adapter à n'importe quelle ville, une photo de la carte des tranports d'une ville inconnue peut-être importée. A partir de cette image, l'utilisateur peut créer la version numérique de ce plan de transports en ajoutant des stations et des points d'arrêts, en connectant deux stations...

Cette édition doit pouvoir se faire dans une interface fluide permettant le déplacement et l'agrandissement afin d'avoir la possibilité d'éditer des plans complexes. Une fois l'édition terminée, ce plan doit être enregistré sous forme de graphe dans un format standardisé pour pouvoir être par la suite parcouru afin de trouver le plus court chemin.



La principale difficulté de cette partie sera de faire déplacer tous les points sur l'interface utilisateur qaund ce dernier souhaite se déplacer sur le plan. Cela va nécessiter d'utiliser des positionnements relatifs et non absolus. A l'heure actuelle nous n'avons pas fais le choix de la bibliothèque GUI que nous utiliserons pour répondre au mieux à cette problématique.

1.3 Intérêts et enjeux

Ce projet fait suite à une question originale : quel trajet permet de parcourir les 304 stations du métro parisien le plus vite possible. Nous pouvons également imaginer dans un second temps ajouter une note d'intérêt à chaque station tel que les lieux touristiques à proximité, les commerces et services à proximité ou bien encore les meilleurs endroits pour faire de la photo à proximité. Ainsi l'application pourrait proposer une alternative un peu plus longue mais plus intéressante en fonction de votre profil.

Pour répondre à ces questions, l'enjeu principal sera de trouver une implémentation optimale du système de transport qui rendra plus facile le travail des algorithmes de recherche par la suite. Cette implémentation pourrait également se baser sur un format de fichier standart permettant de modifier aisément les informations sur les stations, d'ajouter de nouvelles stations, de nouvelles lignes (notamment celle de la ligne 14 qui desservira bientôt notre magnifique campus Epita Villejuif) ou bien même de changer totalement de ville.

2 Etat de l'art

2.1 Applications/Editeurs d'itinéraire

Voici une liste des applications qui permettent de calculer les itinéraire en fonction de la position GPS.

Citymapper :

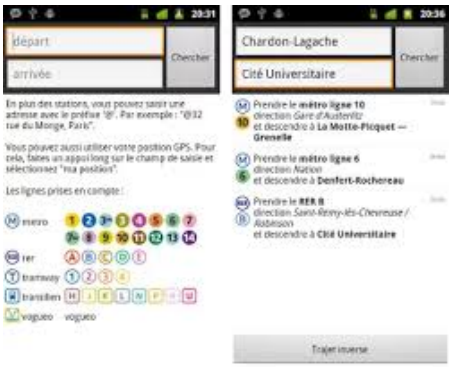
Citymapper est une application connu sur Paris pour avoir la capacité de planifier les itinéraires en temps réel avec une grande variété de transports(Train, metro, bus, taxi, velo ,et bien plus) ce qui permet pour les utilisateurs d'avoir un très grand choix d'itinéraire de déplacement.

Google maps :

C'est une application très basique, un calculateur d'itinéraire classique qui permet d'avoir des cartes précises avec des informations sur le traffic(que ce soit dans le metro ou sur les routes/autoroutes) en temps réel.

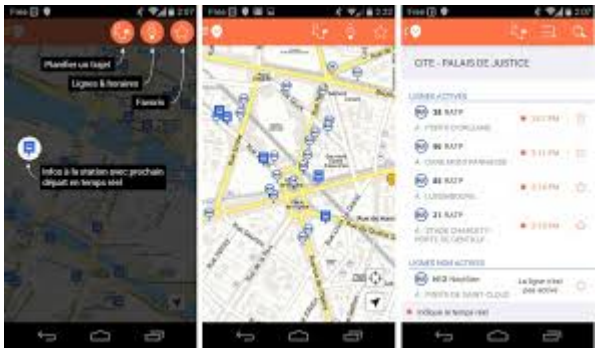
Metro01 :

C'est l'une des seules applications/systemes(apk) pour utilisateurs qui permet de calculer l'itinéraire d'un point A à un point B sans avoir besoin d'internet ce qui peut sauver le temps des utilisateurs s'il se retrouvent sans services. Mais reste un calculateur classique avec des options qui s'ajoute avec l'accès à internet(GPS etc...)



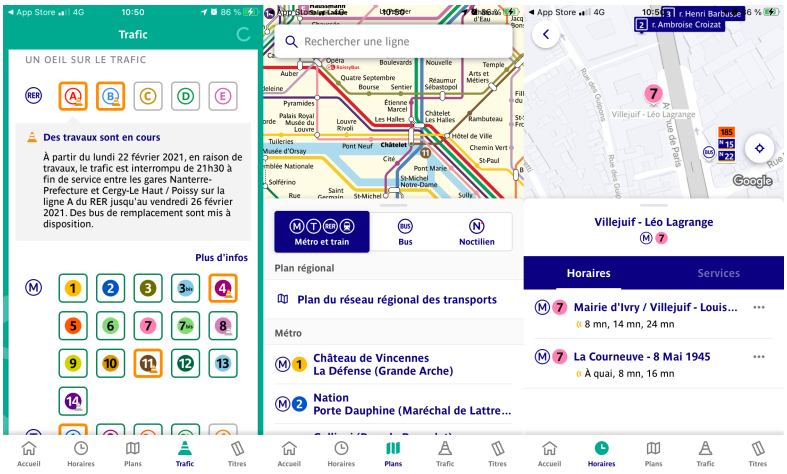
Moovit :

Comme citymapper, c'est un calculateur d'itinéraire qui à une très grande variété de modes de transports. Mais qui a pour particularité de regrouper l'itineraire de plus de 2000 villes du monde qui permet un choix d'itinéraire très varié en temps réel. Que ce soit sur Paris ou ailleurs c'est une application qui est très utilisée.



RATP :

C'est le classique pour ce qui est du deplacement dans le metro parisien car il regroupe toutes les informations sur le plan des metros, les horaires,et le traffic en temps réel qui permet de prevoir des changements de ligne en cas de problèmes.



NOUVEAU

AllyApp :

Il permet aux utilisateurs de l'application d'alimenter de manière collaborative les données et offre une multitude d'API. Il dispose également d'une fonctionnalité de calcul d'itinéraire.

Fulcrum :

Il permet les collectes de données géo-référencées sur téléphone et peut être modifié.Et l'utilisation de la cartographie des transports avec le GPS.

JungleBus :

Cette application Android gratuite et open-source est concentrée exclusivement sur la collecte d'informations sur les arrêts de bus, dont les données géographiques/de localisation.

OSM tracker :

Cet outil offre un suivi GPS, associé à un formulaire d'enquête basé sur une application du type « build-it-yourself », permettant de collecter des informations sur les arrêts et itinéraires. Les données peuvent également être directement synchronisées avec OpenStreetMap.

3 Réalisation

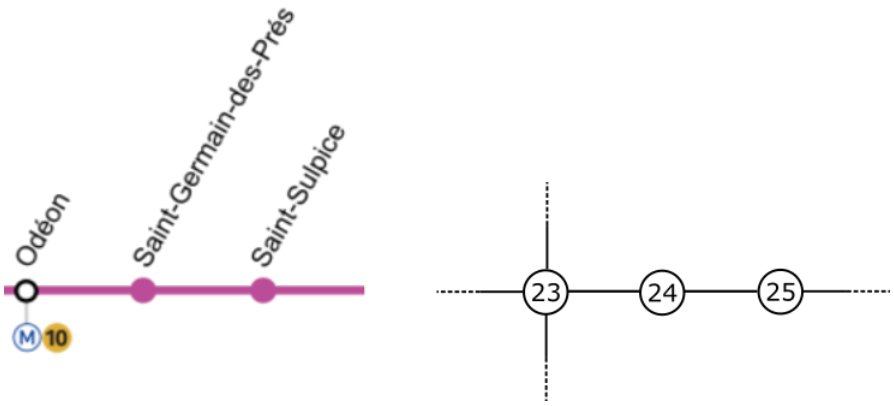
3.1 Implémentation des graphes

3.1.1 Principe

La première chose nécessaire à la gestion d'un réseau de transport urbain est une structure permettant d'une part de stocker et d'organiser toutes les données relatives à ce réseau et d'autre part de traiter ces données afin de trouver des parcours entre deux stations. Pour cela on choisit évidemment de représenter le réseau de transport urbain à l'aide de graphes.

Chaque station sera représentée par un sommet du graphe qui pourra avoir une étiquette contenant le nom littéraire de la station (par exemple : « Villejuif – Paul Vaillant Couturier »). Les trajet entre deux stations sont donc représentés par des arcs.

En effet, le graphe doit être un graphe orientés car certains trajets entre deux stations ne se font que dans un seul sens. Par exemple, sur la ligne 10 entre les stations Javel et Boulogne, le circuit est divisé en deux sens, certaines stations ne sont accessibles que dans un sens et inversement.



Représentation en graphe d'une portion de la ligne 4

La représentation du graphe sera en liste d'adjacence et non en matrice d'adjacence. En effet, chaque station comporte peu d'arcs, dans la plupart des cas seulement deux (une station en amont et une en aval), cette représentation semble donc plus adaptée, une matrice d'adjacence serait lourde et contiendrait énormément de valeurs nulles.

3.1.2 Listes

Commençons donc par créer une structure de listes. Une liste permet de stocker des données en séries tout en ajoutant, supprimant ou accédant librement à n'importe quel élément. Evidemment en C cela peut paraître compliqué à cause de la nécessité d'allouer une certaines place en mémoire variant continuellement. Heureusement pour nous, les pointeurs permettent de récupérer l'emplacement de chaque donnée.

On va donc chainer ces données, c'est-à-dire les attacher les unes aux autres grâce à leur adresse. De manière très concrète, chaque case mémoire contenant une donnée pointera vers l'élément suivant de la liste. Pour cela nous avons donc besoin d'une structure contenant la donnée elle-même ainsi que le pointeur de l'élément suivant de la chaine.

```
struct list
{
    struct list *next;
    int data;
};
```

Cette chaîne est très pratique mais comporte un problème : nous ne savons pas où elle commence. Pour cela nous utilisons une sentinelle, un faux élément indiquant le départ de la chaîne. C'est le pointeur de cet élément qui permet d'accéder à la liste toute entière. Le dernier élément est lui facilement reconnaissable car n'ayant aucun élément à sa suite son champ *next vaut le pointeur NULL. Sur cette base nous implémentons donc un certain nombre de fonctions qui nous serviront d'outils par la suite. Pour chacune d'entre elles, le pointeur de la sentinelle et donc celui de la liste chainée est passé en premier argument afin de modifier directement cette liste.

```
void list_init(struct list *list);
```

Cette fonction initialise les champs de la liste : elle ne comporte que la sentinelle dont le champ *next contient le pointeur nul.

```
int list_is_empty(struct list *list);
```

Une liste est vide quand le champ *next de la sentinelle ne pointe sur aucun autre élément. Elle renvoie 1 quand la liste est vide.

```
int list_len(struct list *list);
```

Cette fonction parcourt toute la chaine de la liste et incrémente un compteur pour chaque élément traversé. Elle retourne la valeur finale de ce compteur.

```
void list_push(struct list *list, int value);
```

Cette fonction ajoute un élement au début de la liste, elle créé une structure list supplémentaire contenant la donnée passée en paramètre et l'insère juste après la sentinelle.

```
int list_pop(struct list *list);
```

Cette fonction retire le premier élément de la liste (en laissant la sentinelle) et retourne la valeur qui était contenue dans cet élément.

```
void list_insert(struct list *list, int value, int place);
```

Cette fonction insère une valeur dans la liste mais à une place donnée. Si l'indice de l'élément est négatif il sera inséré en tant que premier élément de la liste. Si cette place est plus grande que la longueur de la liste il sera inséré en tant que dernier élément de la liste.

```
int list_remove(struct list *list, int place);
```

Cette fonction supprime l'élément situé à la place désignée et retourne sa valeur. Il faut évidemment rechainer l'élément précédent avec l'élément suivant pour garder toute la chaine restante de la liste.

```
int list_remove_val(struct list *list, int value);
```

Cette fonction supprime la première donnée égale au paramètre value dans la liste. Si la valeur n'est pas contenue dans la liste, cette dernière reste intacte.

```
struct list *list_find(struct list *list, int value);
```

Cette fonction cherche et retourne l'élément de la liste dont la valeur est égale au paramètre value. L'intérêt de retourner l'élément sous sa forme de structure est de relancer une recherche sur la suite la liste.

```
int list_contains(struct list *list, int value);
```

Cette fonction retourne 1 si la liste contient au moins une occurrence de la valeur recherchée. Cette fonction est basée sur la fonction précédente.

```
void print_list(struct list *list);
```

Cette fonction sert à afficher une liste. Elle affiche empty si la liste est vide et les valeurs contenues dans la liste séparées par un espace sinon.

Cette implémentation des listes est basique mais suffisante.

3.1.3 Implémentation des graphes

Maintenant que nous avons un système de listes, nous allons pouvoir implémenter des graphes par listes d'adjacences. Pour cela nous définissons une nouvelle structure, qui contient 3 propriétés, l'ordre du graphe, si il est orienté ou non ainsi qu'un tableau de listes d'adjacences, une pour chaque sommet. Si les deux premières sont simples à mettre en oeuvre (on utilise des entiers) la dernière est plus complexe, notamment du fait de l'allocation dynamique de mémoire.

```
struct graph
{
    int directed;
    int order;
    struct list *adjlists[1];
};
```

Nous devons allouer sufissament de mémoire pour stocker la structure même du graphe mais aussi la taille d'une structure de liste multipliée par le nombre de sommets, soit l'ordre du graphe

```
struct graph *g = (struct graph*)malloc(sizeof(struct graph) +
        sizeof(struct list*) * order);
```

Une fois que tout l'espace nécessaire est réservé, nous allons parcourir le tableau des listes d'adjacences, et pour chaque sommet, donc pour chaque case, initialiser une liste. Nous pouvons ensuite retourner le pointeur de ce nouveau graphe.

Comme pour les listes, nous définissons les fonctions utilitaires pour manipuler un graphe : ajouter ou supprimer un arc, ajouter ou supprimer un sommet.

```
void graph_add_edge(struct graph *g, int src, int dst);
void graph_remove_edge(struct graph *g, int src, int dst);
```

Ces deux fonctions ont un fonctionnement similaire. Pour ajouter ou supprimer un arc il faut en réalité ajouter ou supprimer le sommet dont l'arc est incident de la liste d'adjacence du sommet dont l'arc est sortant. Nous appelerons ces deux sommets source et destination. Il faut tout d'abord nous assurer que les entiers src et dst passés en paramètres des fonctions pour représenter respectivement l'indice des sommets source et destination appartiennent bien au graphe, donc qu'ils sont compris entre 0 et l'ordre du graphe. Dans le cas contraire, le programme est arrêté avec un mesage d'erreur.

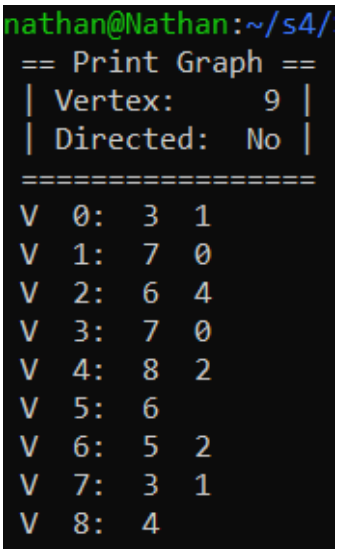
Suite à ces vérifications, si l'indice du sommet de destination n'est pas présent dans la liste d'adjacence du sommet source, on l'ajoute. Pour cela on utilise les fonctions `list_contains` et `list_push` vues précédemment. Si le graphe est non orienté on répète cette opération en inversant les sommets source et destination.

```c
struct graph *graph_add_vertex(struct graph *g);
```

Ajouter un sommet au graphe revient à faire deux actions. D'une part augmenter l'ordre du graphe, ce qui est assez simple, d'autre part étendre son tableau de liste d'adjacence pour y ajouter la liste d'adjacence de son nouveau sommet. Ce second point a été plus difficile à implémenter bien qu'il soit finalement assez simple à saisir. Il suffit de garder le pointer renvoyé par la fonction realloc utilisée pour augmenter l'espace mémoire réservé au graphe et de retourner ce pointeur. Il ne faut donc pas oublier de récupérer le nouveau pointeur de structure graphe à chaque fois que l'on appelle cette fonction `graph_add_vertex`

```c
void graph_print(struct graph *g);
```

Cette fonction affiche le graphe en deux parties : d'abord l'entête avec les informations relatives au graphe (ordre, orienté ou non) puis la liste d'adjacence pour chaque sommet. Cette fonction est surtout utile pour visualiser des tests.



Exemple d'affichage d'un graphe simple non orienté d'ordre 9

3.1.4 Lecture et enregistrement

La construction de graphe n'est utile pour répondre à notre besoin que si nous pouvons enregistrer le graphe qui représente notre réseau de transport urbain afin de pouvoir l'éditer par la suite mais surtout le fournir comme donnée à l'application utilisateur qui créé les parcours. Pour cet enregistrement, les données doivent être formalisées et écrites dans un fichier. Nous choisissons le type de fichier .gra standardisé par Microsoft.

```
#labels: algo,maths,prog,archi,elec,phys
1
6
0 2
1 0
1 3
1 4
1 5
2 1
2 3
3 2
3 0
4 3
5 4
```

Ce fichier est composé de 3 zones principales : les options et les commentaires, les informations d'entêtes et la liste des arcs. Les lignes de la première parties commencent par un symbole dièse et peuvent contenir des options telles que les étiquettes des sommets du graphe. Dans un premier temps nous ignorerons cette partie.

Viens ensuite la seconde partie composée de deux lignes d'un nombre chacune. La première concerne le type de graphe, la ligne contient 1 si le graphe est orienté, 0 sinon. La seconde ligne contient l'ordre du graphe, c'est à dire le nombre de sommet.

Enfin les lignes suivantes (dont le nombre peut varier) constituent la troisième partie. Chacune de ces lignes représente un arc du graphe. Pour chacun de ces arcs, la ligne comprend d'abord l'indice du sommet source puis un espace de séparation puis l'indice du sommet de destination.

Nous sommes donc prêt à lire un fichier gra et à charger son contenu dans une structure de graphe. Pour cela on commence par ouvrir le fichier en lecture et récupérer le descripteur de fichier associé. La fonction `getline(&line, &len, fd)` nous permet de lire ligne après ligne le contenu du fichier et de stocker chaque ligne dans le buffer créé par avance. Ce buffer pourra être converti en nombre

à l'aide de la fonction `atoi(line)` ou traitée à posteriori. Ce sera le cas pour les lignes comprenant les arcs. Cette ligne est séparée par la reconnaissance du caractère espace. Voici un code partiel permettant ces séparations successives.

```
while (getline(&line, &len, fp) != -1)
{
        char *ptr;

        ptr = strtok(line, " ");
        int src = atoi(ptr);
        ptr = strtok(NULL, " ");
        int dst = atoi(ptr);
        graph_add_edge(g, src, dst);
}
```

La fonction `struct graph *graph_load(char *file)` va donc constituer le graphe correspondant aux données stockées dans le fichier .gra passé en paramètre. Ce graphe sera stockée dans la structure de graphe créée précédemment et sera renvoyé par la fonction.

Pour enregistrer le graphe on se sert de la même norme mais dans le sens inverse. On écrit dans un fichier le type de graphe puis son ordre puis tous les arcs le constituant en s'aidant de la fonction `fprintf` dans le but de pouvoir le réouvrir plus tard.

Bien sûr, on n'oublie pas dans chacun des deux cas de refermer le descripteur du fichier utilisé ainsi que de libérer les espaces mémoires réservés.

3.1.5 Ajouts et améliorations

Notre implémentation de graphe est une implémentation complète mais basique. En effet, elle est faite de sorte à pouvoir supporter les graphes usuels. Cependant nous voudrons utiliser notre implémentation dans un cas spécifique : celui des réseaux de transports. Nous avons donc d'ores et déjà ciblé deux types d'améliorations : celles d'ordre générale, les optimisations de codes et celles d'ordre spécifique permettant une meilleure appréhension des graphes dont nous nous servirons.

Les optimisations générales concernent notamment les graphes non-orientés. Effectivement, la totalité des arcs (donc des arrêtes dans ce cas) sont enregistrés dans le fichier .gra quand seulement une sur deux servirait puisque lors du chargement du graphe on redouble les arcs dans les deux sens si le graphe est orienté. Cela pourrait permettre de réduire la taille de fichiers.

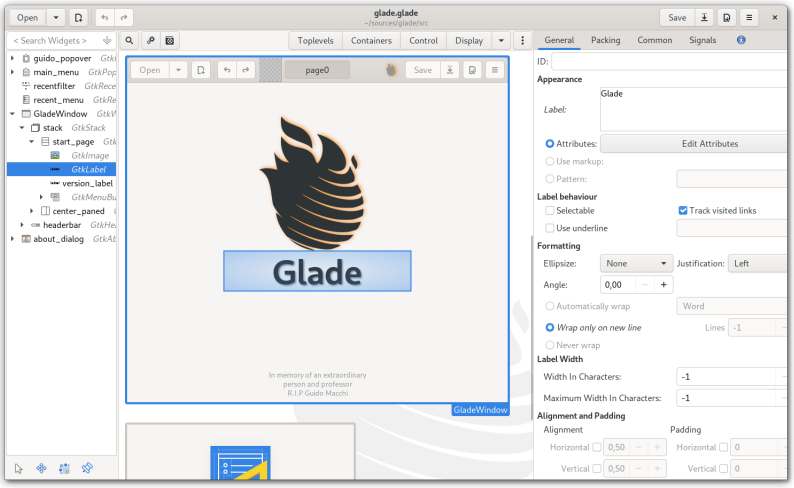
D'autre part pour se rapprocher d'un système de transport urbain, nous devrons intégrer deux informations importantes permettant aux usagers de se réperer, à savoir les noms littéraux des stations ainsi que les identifiants des différentes lignes. Concernant les noms des stations, nous devrions utiliser le champ étiquette dans la première partie du fichier .gra qui et actuellement ignoré. Nous pourrions y stocker d'autres informations telles que l'image du plan correspondant.

Enfin concernant les numéros de lignes, il faudrait que chaque arc ait le numéro d'identifiant de la ligne de transport à laquelle il appartient. Dans la structure graphe, cela reviendrait à ajouter une propriété dans les listes d'adjacences et dans les fichiers .gra on pourrait assez simplement ajouter une troisièeme valeure entière sur chaque ligne représentant un arc.

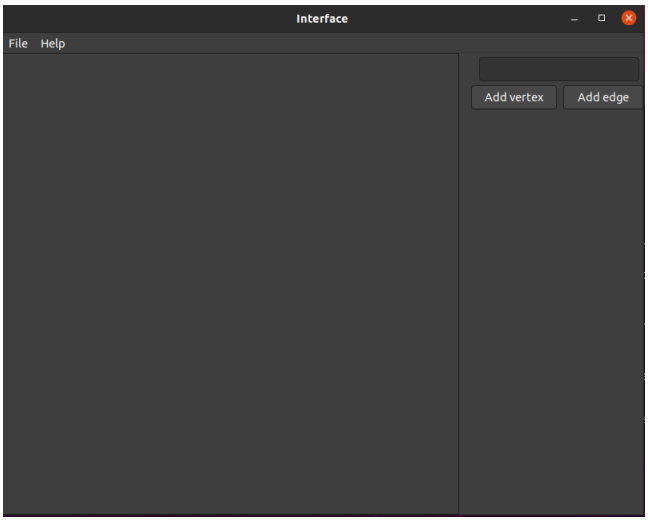
Ces ajouts pourront être finalisés lorsque nous fusionnerons l'interface avec le système de graphe.

3.2 Interface

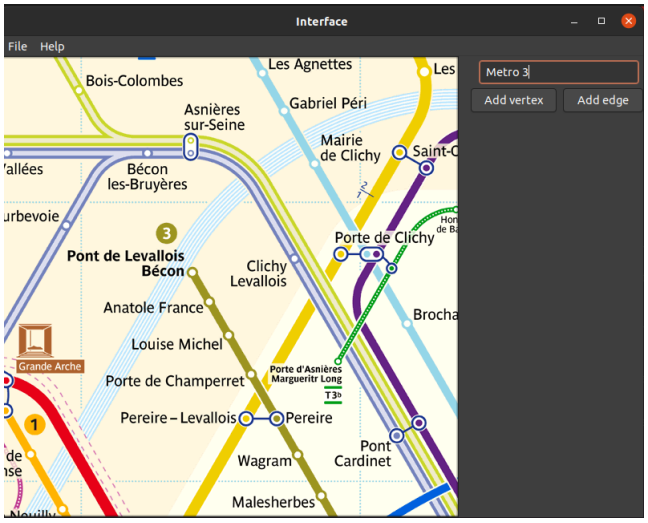
Pour cette première soutenance, l'interface graphique pour l'utilisateur doit être très basique pour la facilité d'utilisation.
Nous étions parti dans l'idée d'utiliser SDL pour le réaliser mais le management des evenements devient très brouillons à partir d'un point.
Nous avons donc décidé d'utiliser Gtk avec glade qui facilite énormément la forme et la structure de l'interface.



Nous avons opté pour une interface séparé en deux avec une partie qui est un lecteur d'image à gauche et la partie de droite où l'on peut retrouver des bontons permettant d'ajouter des sommets ou des arrêtes grâce à la structure des graphes cité juste au dessus qui seront retranscrite comme étant des ligne de metros ou des noms de stations.



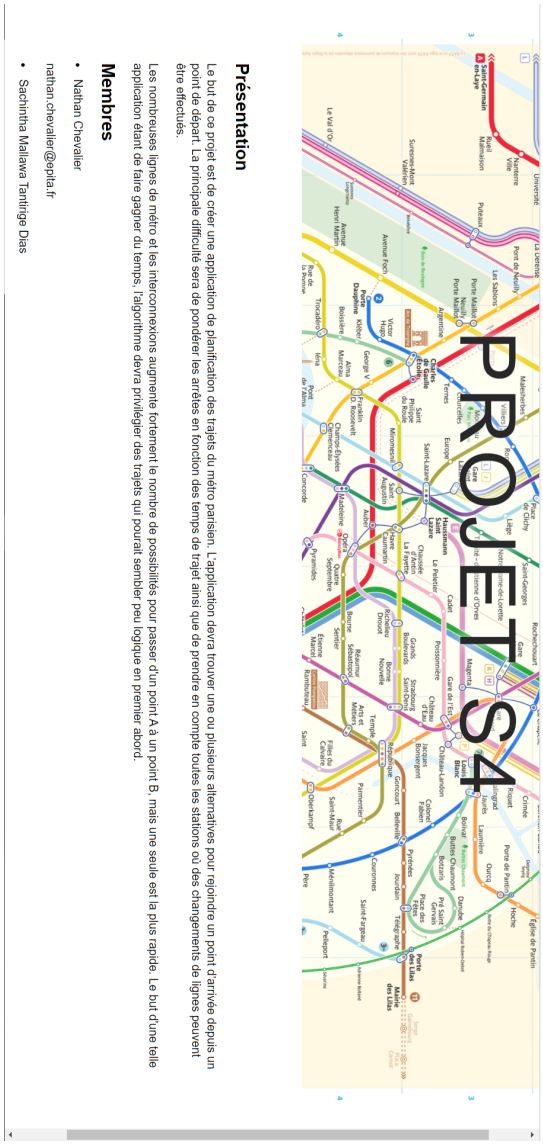
Avec le plan ouvert :



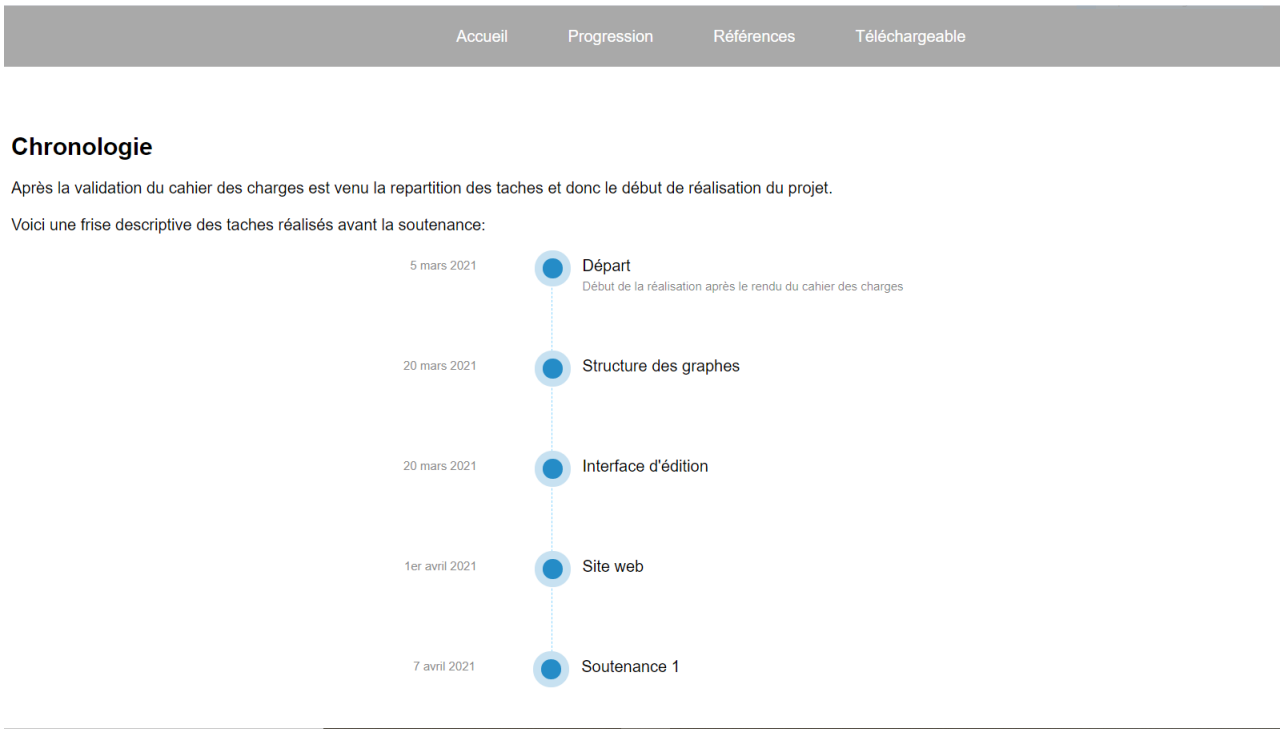
3.3 Site Web

Pour la réalisation du site web, nous sommes restés sur un code html très simple qui va à l'essentiel donc ce qui est demandé pour la soutenance et du code css assez débutant.

Nous avons donc en première page une présentation rapide du sujet de notre projet avec une description de nos membres.



Sur la deuxième page, nous retrouvons la progression du projet avec la partie Chronologie completé d'une frise chronologique.
Et à la fin il y a une liste des problèmes rencontrés durant la réalisation du projet avec les solutions envisagées pour remédier à ces problèmes.



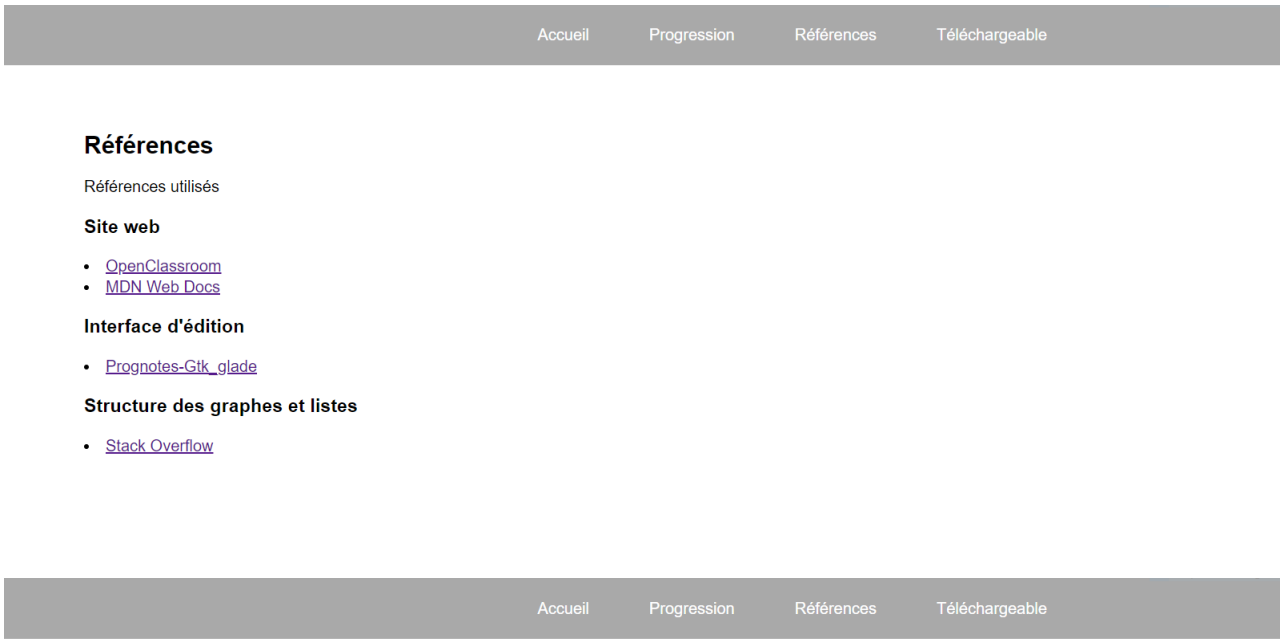
Problèmes rencontrés

- Surestimation des capacités avec le temps restant avant la soutenance
- Manque d'organisation
- Mise en commun des codes

Solutions envisagées

- Plus de communications
- Avoir une meilleur organisation(Gantt)

La troisième et quatrième page qui ne sont pas encore complète en terme de mise en page regroupe les références, les liens utilisés pour faire le projet et enfin des liens de téléchargement de notre cahier de charges, du rapport de soutenance et du projet(non-fonctionnel)



4 Organisation pratique

4.1 Choix Techniques

Ce projet sera l'occasion de mettre en pratique les connaissances acquises sur certaines types de structures complexes tels que les graphes. Afin de respecter les contraintes fixées dans le cadre du projet, l'application sera développée à l'aide du langage C sous Unix. Nous devrons donc dans un premier temps créer des outils pour utiliser les structures de travail (ie les graphes).

Enfin, pour le site web dédié au projet, nous utiliserons les langages HTML, CSS et JS obligatoires en front-end et ferons le choix de PHP pour le back-end et MySQL pour un éventuel système de base de données.Tout ces choix techniques n'engendrent pas de coût, puisque certains membres du groupes peuvent utiliser leurs serveurs web personnels.

Nous avons à présent arrêté notre choix de bibliothèque de GUI qui permettra de créer l'interface de l'éditeur de plan et nous nous sommes arrêtés sur GTK qui a une meilleure gestion des évènement et peut-être couplé avec glade pour créer une interface soignée.

Enfin, le CRI ne nous en ayant pas fourni, nous avons également créé un répertoire sur GitHub, un outil basé sur git, afin de pouvoir collaborer sur notre projet ainsi que d'archiver les différentes versions de l'avancée de notre projet. Le lien est le suivant : https ://github.com/NathanCHEVALIER/S4Project

4.2 Répartition des tâches

Tâche à effectuer	Référent + Suppléant
Implémentation des structures	Nathan + Sachintha
Algorithme de chemins	Nathan + Sachintha
Editeur de plans	Nathan + Sachintha
Saisie des données	Nathan + Sachintha
Interface	Nathan + Sachintha
Site et docu	Nathan + Sachintha

4.3 Planning prévisionnel

Tâches à effectuer	Soutenance 1	Soutenance 2	Soutenance 3
Implémentation des structures	100%	100%	100%
Algorithme de chemins	0%	70%	100%
Editeur de plans	40%	70%	100%
Saisie des données	50%	100%	100%
Interface	20%	40%	100%
Options	0%	0%	100%
Site et docu	20%	60%	100%

5 Conclusion

Au-delà des aspects techniques, la première condition pour la réussite d'un tel projet est la gestion des ressources humaines. Nous avons la particularité de n'être que deux pour ce projet ce qui s'avère être une chance. Effectivement, il n'y a pas de tensions, chacun a suffisament à faire et doit dans le même temps s'impliquer équitablement. En bref, l'équilibre est plus simple à trouver.

La gestion du temps, et donc un travail régulier sera une clé pour l'aboutissement du projet. Il faudra également prendre du recul pour s'assurer au fur et à mesure de respecter les objectifs fixés. Nous sommes actuellement satisfaits de ce qui a été effectué et confiants pour la suite.