Projet Algorithmique : Métro

BATTISTON Ugo 21805659
JAM Mathys 21801747

BATTISTON Ugo 21805659
JAM Mathys 21801747

Dans le cadre de notre projet, nous avons utilisé la librairie graphique SFML en c++. Cet librairie est donc nécessaire à la bonne compilation du projet et est obtenable par le package sfml-devel sur linux, ou est téléchargable sur https://www.sfml-dev.org/

L'interface graphique a été conçu pour être utilisable pour toute tailles d'écran et devrait s'agrandir ou se rétrécir sans problème. Cependant, en cas de difficulté d'utilisation, ou problème d'installation des librairies, vous trouverez en indexe des images de démonstration, et restons joignable par mail si besoin est.

De plus, voici quelques points sur l'interface graphique :

- La cartes des stations est zoomable. Le zoom sera centré sur votre souris : attention, il est très sensible.
- A un certain niveau de zoom, le nom des stations s'affiche au dessus d'elles.
- Les stations sélectionné s'affichent sur la gauche. Vous pouvez tapez le nom que vous cherchez directement dans les barres. Vous pouvez cliquez pour sélectionner une recommandation.
- Cependant, vous pouvez aussi cliquez sur les stations sur la carte. Les stations sélectionnées sont facilement identifiable par leur taille, et couleurs.
- Cliquer sur la carte pour sélectionner une stations.
- Cliquer sur une station sélectionnée pour la retirez.
- Cliquer sur une icône de ligne en haut de l'écran pour en désactiver l'affichage.
- Un fois deux stations sélectionner, cliquer sur calculer le trajet pour afficher le trajet en textuel. Le trajet s'affiche sur la carte en rouge.

Les fichiers source se trouvent normalement dans l'archive avec ce document. Ils sont séparés en deux catégories : Graph, et Graphics. Graph représente l'ensemble des fichier nécessaire au dijkstra, tandis que Graphics contient tous les fichier nécessaire à l'affichage graphique du métro.

Contact :

Mathys JAM : mathys.jam@ens.uvsq.fr
Ugo BATTISTON : ugo.battiston@ens.uvsq.fr

BATTISTON Ugo 21805659
JAM Mathys 21801747

Stations et terminus

Pour pouvoir dessiner les stations ainsi que de pouvoir afficher les lignes prises lors du chemin, il était nécessaire de compléter le fichier fournis dans le sujet. Dans un premier temps, nous avons réfléchi à une manière de déterminer quels stations appartiennent à quels ligne. Nous avons remarqué que les “stations” présente dans le fichier correspondait plutôt à des quais de gare, et ainsi qu'un trajet entre deux station du même nom correspondait à un trajet à pied. Ainsi, les stations d’une même lignes était relié par des trajet en métro tandis qu’un changement de ligne par un trajet à pied. Ainsi, il s’agissait d’un problème de coloration de graphe : il suffisait de propager la couleur, ici le numéro de ligne, de proche en proche, et de s'arrêter si deux stations était relié par un trajet à pied. Nous avons donc pour chaque ligne, pris tout les terminus (peut importe la direction) et les avons ajouté au fichier fournis. Ils sont différencié par un T au lieu d’un V en début de ligne, ainsi que le numéro de ligne.

```
V 0036 Boulogne, Jean Jaurès
T 0037 10 Boulogne, Pont de Saint-Cloud, Rond Point Rhin et
Danube
```

figure 1
metro.txt

BATTISTON Ugo 21805659
JAM Mathys 21801747

Ainsi, après le chargement du graphe, nous propageons les numéros de lignes et obtenons donc les lignes correspondant à la réalité (voir figure 2).

```
void Graph::spread_line()
{
   // Pile pour stocker les ids des stations à traiter
   std::stack<unsigned int> to_treat;
   // Pour chaque station dans le graph
   for (auto& station : vertices)
   {
       // Si la station a une ligne définie et n'est pas marqué
       // (correspond à un nouveau terminus)
       if (station.second.getLine() == "" || station.second.getMarked())
           continue;
       station.second.setMarked(1);
       // on rajoute la station sur la pile
       to_treat.push(station.second.getId());
       // parcours en profondeur
       while (!to_treat.empty())
       {
           Vertex& buffer = vertices[to_treat.top()];
           to_treat.pop();
           // pour chaque connexions de la station courantes
           for (auto& e : buffer.getEdges())
           {
               //Si la liaison n'est pas de la marche (même ligne) et que la station n'a pas
de ligne définie
               if (e.getMetro() == false || vertices[e.getDestination()].getLine() != "" ||
vertices[e.getDestination()].getMarked())
                   continue;
               // on définie la ligne et on push la nouvelle station sur la pile
               Vertex& tmp = vertices[e.getDestination()];
               tmp.setLine(buffer.getLine());
```

```
                tmp.setMarked(1);
                to_treat.push(e.getDestination());
            }
        }
    }
    unmarkAll();
}
```

figure 2
src/Graph/graph.cpp l.72

Grâce à cet algorithme, nous n'avons pas eu besoin de compléter tout le fichier avec toutes les lignes manuellement, uniquement les terminus.

Ensuite, nous avons eu besoins d'obtenir les positions de chaque stations pour pouvoir les dessiner dans l'espace correctement. Nous avons opté pour ne pas utiliser de carte de fond, mais plutôt de représenter chaque station par une position en x/y. Pour ceci, nous avons utiliser l'API d'Ile de france mobilité pour obtenir une liste de toutes les stations de métro. Nous avons ensuite éliminer les stations et connexion superflu (antérieur au fichier fourni) et ainsi obtenu une position "réel" de chaque station. Pour éviter les conflits au fait que plusieurs stations ait le même nom ou les problème d'accentuations (notamment à l'affichage), tous les accent on été supprimé du fichier fourni, et tous les quais sont considérés comme étant à la même position.

```
   x        y         noms
648294  6861006  Avenue Emile Zola
655847  6861443  Avron
```

figure 3
positions.txt

```
        // on map les positions réels sur l'écran
        x = mapTo(643000 , 662000 , 50, 3 * (double) window.getSize().x / 5 - 50, x);
        y = mapTo(6853000, 6873000, window.getSize().y - 50, 50, y);
```

figure 4
src/Graphics/GraphDrawer.cpp l.56

BATTISTON Ugo 21805659
JAM Mathys 21801747

Pathfinding

Nous avons commencé par réfléchir à une structure de donnée permettant une bonne implémentation de l'algorithme de Dijkstra. Nous avons choisi de faire une classe Vertex (figure 5), Edge (figure 6), pour les sommets et les arrête.

```
class Vertex
{
private:
   std::list<Edge> edges; // connexions sortantes
   std::string name;      // noms de la station/quais, sans accents
   std::string line;      // nom de la ligne
   unsigned int id;       // id unique
   int marked;
   bool is_terminus;
```

figure 5
headers/Vertex.hpp

```
class Edge
{
private:
   unsigned int idSource, idDestination; // Les arêtes sont orienté
   unsigned int duration;                // Durée du trajet, poids de l'arrete
   bool isMetro = true;                  // si la connexions corresponds a un trajet à
pieds ou à métro
```

figure 6
headers/Edge.hpp

Et une classe englobant les différentes arêtes et les sommets dans une classe Graph. Nous avons fait le choix de différencier la partie graphique de la partie algorithme afin que la classe Graph soit utilisable indépendamment d'un affichage graphique.

```
class Graph
{
private:
   std::map<unsigned int, Vertex> vertices;
public:
   Graph(std::string fileName);
   int load_from_file(std::string fileName);
   std::map<unsigned int, Vertex>& getVertices();
   std::list<Vertex> dijkstra(unsigned int source, unsigned int destination);
    int add_vertex(const std::string name, const unsigned int id, std::string line = "");
   int add_edge(const unsigned int id1, const unsigned int id2, const unsigned int
duration, bool isMetro = true);
   std::list<std::string> vertex_to_string(std::list<Vertex>& vertices_path);
```

figure 7
Graph.hpp

On peut utiliser le graph directement en tant que telle, via dijkstra(src, dest), la fonction renvoi une liste contenant chaque étape du trajet. L'interface graphique interagit avec le graph par cet méthode.

Pour l'algorithme, nous avons fait le choix d'utiliser l'algorithme de Dijkstra, en le modifiant pour qu'il soit plus optimisé en arrêtant l'algorithme lorsque que le sommet de destination du l'utilisateur a été analysé. A la place d'obtenir un one to all, on obtient des chemin jusqu'à obtenir le chemin souhaité et d'arrêter l'algorithme.

```
while(V.size() < vertices.size())
{
   a++;
   unsigned int indice_min = dijkstra_find_indice_min_distance(distance, V);
   V.push_back(indice_min);

   std::list<Edge> edges = dijkstra_find_valid_edge(indice_min, V);

   for (auto& i : edges)
   {
       unsigned int new_duration = distance[indice_min] + i->getDuration();
       unsigned int old_duration = distance[i->getDestination()];

       if(new_duration < old_duration)
       {
           distance[i->getDestination()] = new_duration;
           pere[i->getDestination()] = indice_min;
       }
   }
   if(vertices.find(idDestintion)->second.getName() ==
vertices.find(indice_min)->second.getName())
           break;
}
```

figure 8
src/Graph/Graph.cpp dijkstra
La version commentée en détails de l'algorithme est disponible dans les sources

Nous avons pensé à d'autres algorithmes comme l'algorithme d'A*, mais il nous aurait fallu changer la structure de notre classe graph, mais sur conseil de Monsieur Thierry Mautor, nous sommes dirigé vers un Dijkstra. Permettant de garder une classe Graph générique fonctionnant sur tous les graph.

On s'est ensuite poser la question de comment obtenir la direction : nous savons quels lignes notre chemin empruntait, mais nous n'avions pas alors considéré la notion de "direction" d'une ligne. Hors, dans la réalité, cet notion est vitale pour identifier quels quai emprunter. Nous avons donc dû trouver une solution s'accommodant avec ce que nous avions déjà fait. En premier lieu, nous avons pensez à propager la direction en même temps que nous propagions

la ligne : vu que cet dernière se propage en partant des terminus, nous aurions donc pu savoir dans quels sens nous allions. Cependant, cet solutions ne marche pas dans certains cas :

Au lieu de tout recommencer, nous avons opter pour une solution qui nous semblait moin élégante, et plus cher en terme de complexité. Une fois que nous quittons une ligne dans notre trajet, nous effectuons un parcours en profondeur dans la même direction jusqu'à temps d'avoir trouver un terminus. Cet méthode permet aussi de résoudre le problème de boucle. Nous marquons les deux derniers sommet, permettant effectivement de “donner une direction” au parcours en profondeur, et ainsi ne pas trouver le terminus d’origine.

```
std::string Graph::calcul_terminus(unsigned int id1, unsigned int id2)
{
    std::stack<unsigned int> pile;
    vertices[id1].setMarked(1);
    vertices[id2].setMarked(1);
    pile.push(id2);
    while (!pile.empty())
    {
        int current = pile.top();
        pile.pop();
        if (vertices[current].getTerminus())
            return vertices[current].getName();
        for (auto& i : vertices[current].getEdges())
        {
            if (!(vertices[i.getDestination()].getMarked()) &&
vertices[i.getDestination()].getLine() == vertices[current].getLine())
            {
                if (vertices[i.getDestination()].getTerminus())
                    return vertices[i.getDestination()].getName();
                pile.push(i.getDestination());
                vertices[i.getDestination()].setMarked(1);
            }
        }
    }
    throw std::runtime_error("Unknown terminus :) !");
}
```

figure 9

Graph.cpp l.373

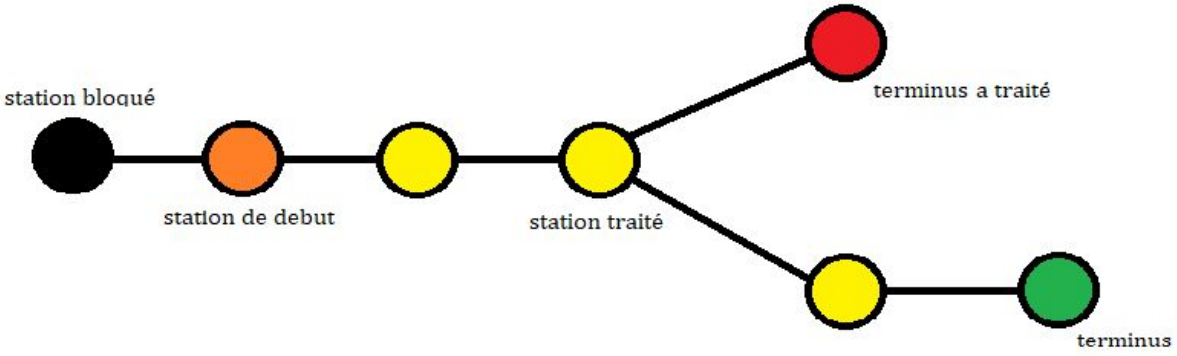
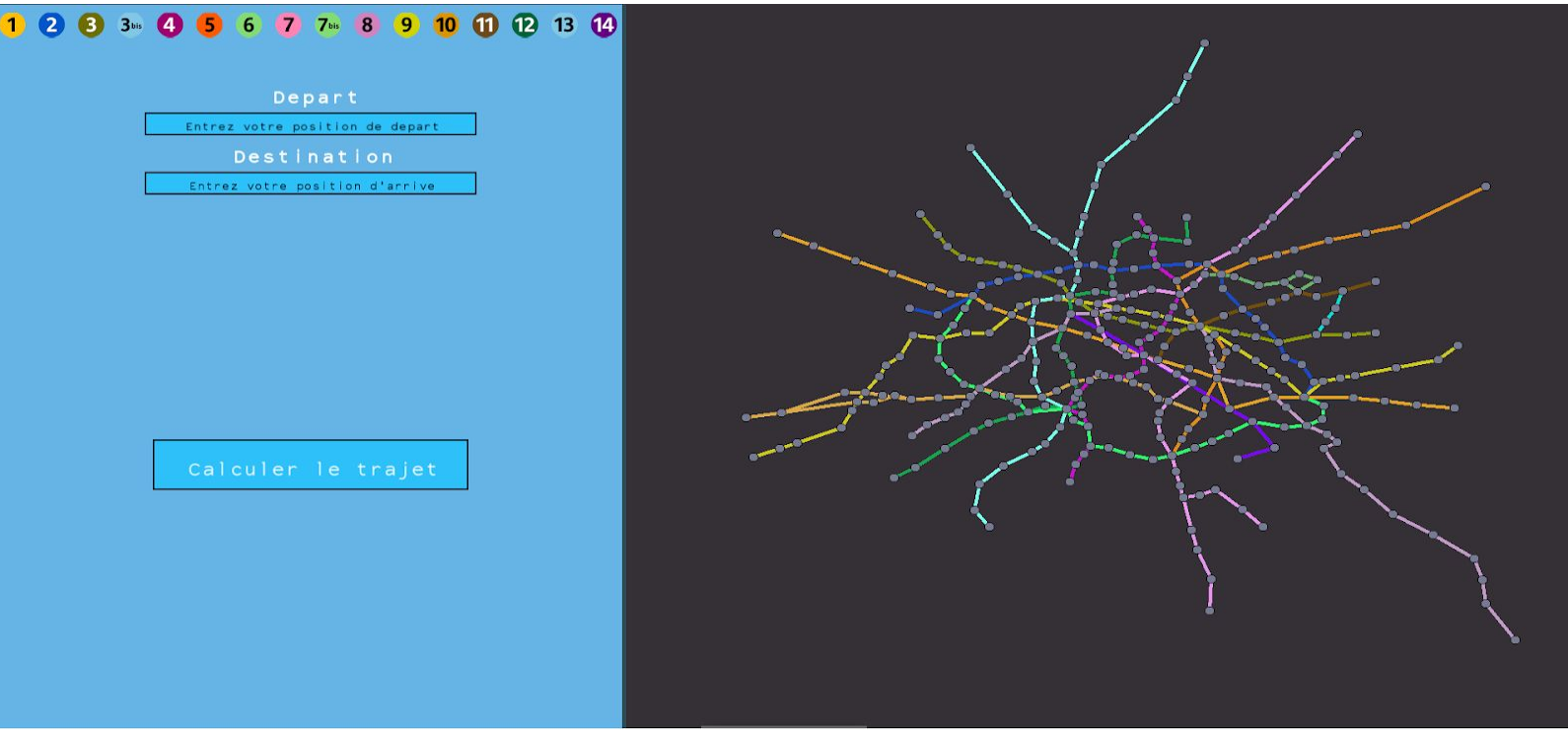


figure 10
représentation du parcours
dessiné par Ugo

Interface graphique / Annexe

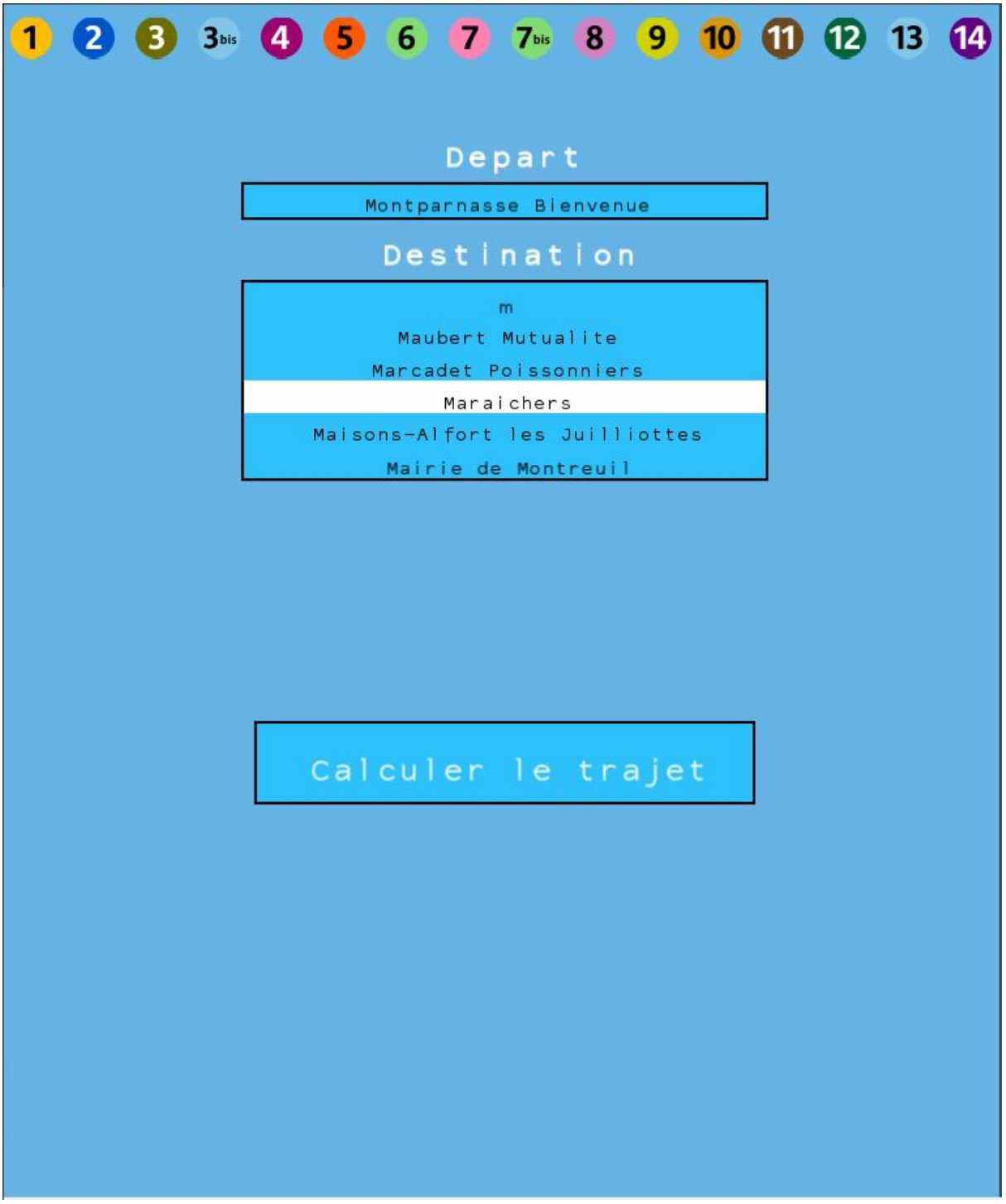
La classe GraphDrawer est la classe principal pour dessiner un graph : ce qui permet donc bel et bien de séparer efficacement le graph de sa représentation graphique. En plus de cela, nous avons plusieurs classe représentant chacun des boutons, ainsi qu'une classe s'occupant de dessiner un "menu" (partie de gauche permettant de choisir départ et destination) et une classe permettant de représenter les Stations (cet fois ci sous forme graphique plutôt que dans le Graphe)

Interface au complet



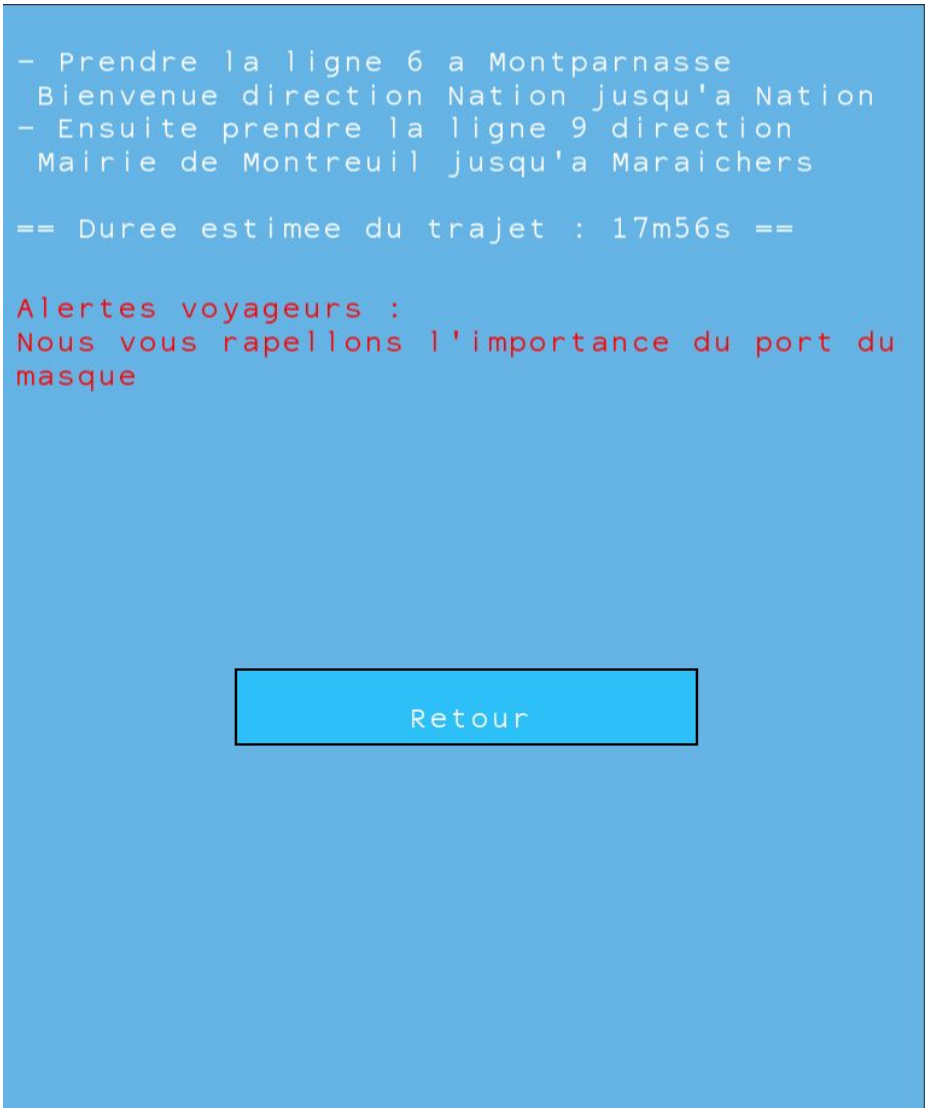
BATTISTON Ugo 21805659
JAM Mathys 21801747

Exemple de sélections de stations via les boutons d'entrés.

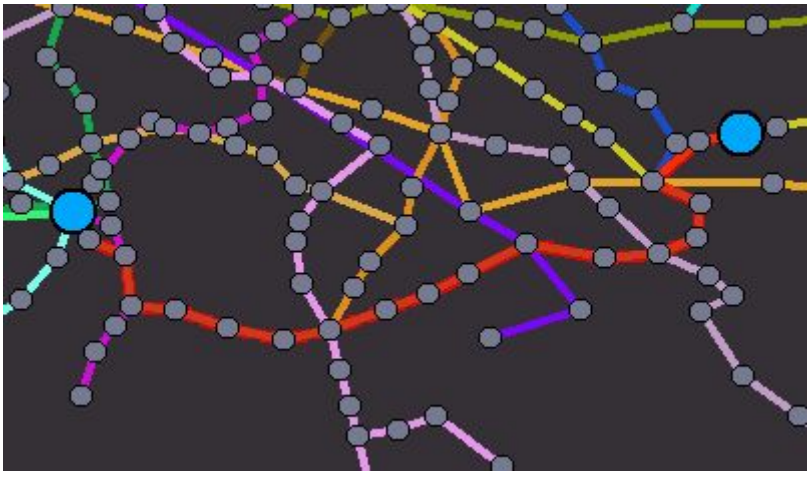


BATTISTON Ugo 21805659
JAM Mathys 21801747

Exemple de trajet

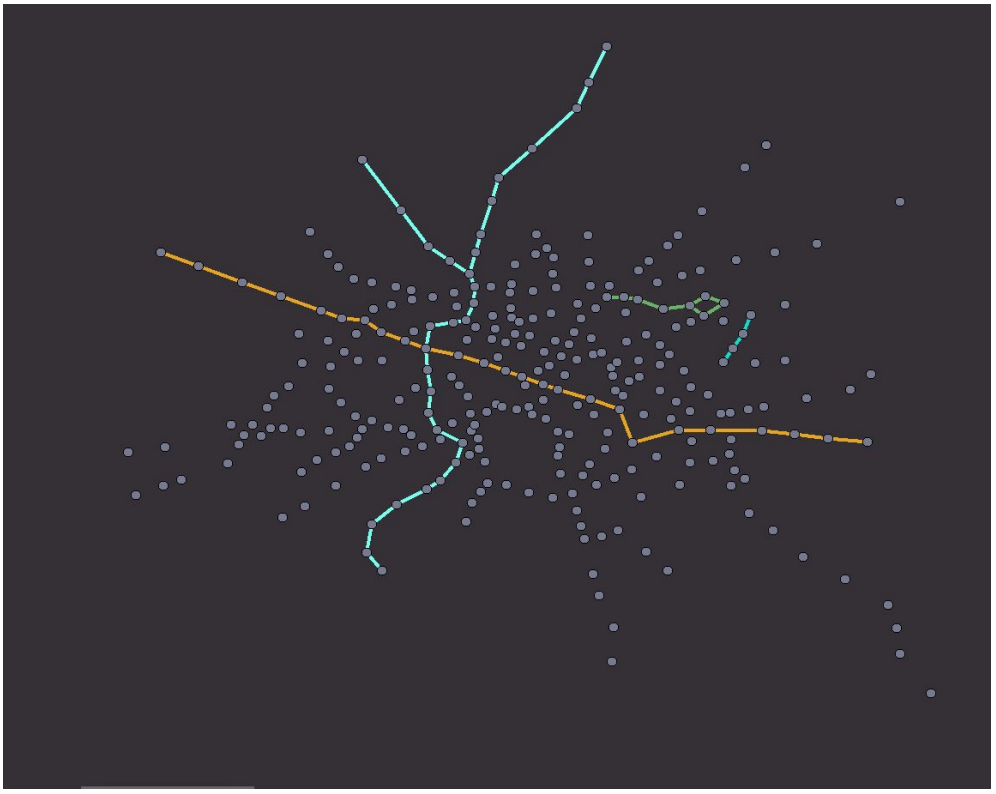


Affichage du trajet sur la carte

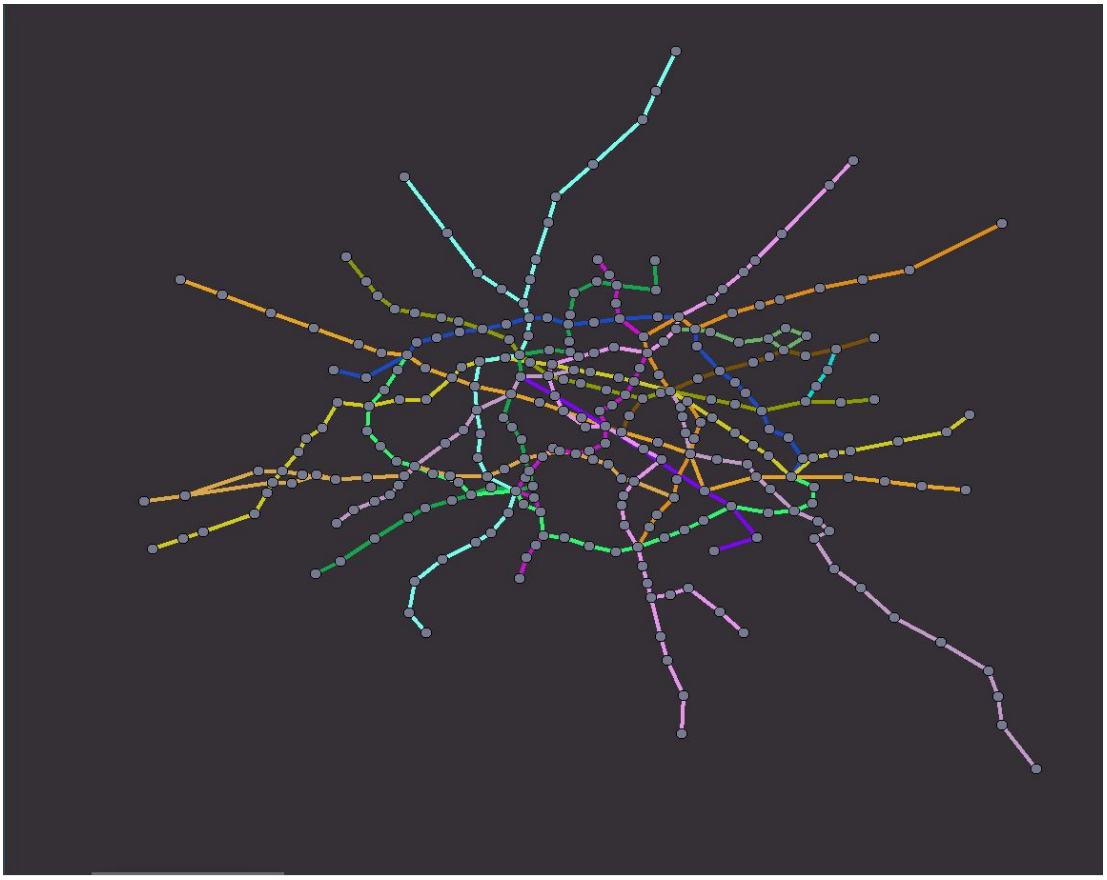


BATTISTON Ugo 21805659
JAM Mathys 21801747

Exemple de la fonction de désactivation des lignes



Plan au complet



BATTISTON Ugo 21805659
JAM Mathys 21801747

Exemple de la fonctions de zoom