

# Le jeu du Parisien



Arnaud De Laborderie

Germain Le

Romain Llorca

Théorie des graphes

ING2 – Groupe 3

Année 2008-2009



# Sommaire

- 1. Introduction**
  
- 2. Organisation générale du code**
  - 2.1 Structures de données
  - 2.2 Algorithmes utilisés
  - 2.3 Interface graphique
  
- 3. Fonctionnement de l'équipe et conclusion**



## 1. Introduction

Le principe du jeu du Parisien que nous avons développé est très simple. Il s'agit de faire deviner au joueur le nom de l'une des 299 stations du métro parisien (RER et Tramways non inclus). Lorsque le joueur trouve la station (choisie aléatoirement par le programme), celui-ci ne prend pas de pénalités. Dans le cas contraire, il perd un point par station d'écart et cinq par correspondance empruntée. Le jeu devait comprendre au minimum trois niveaux de jeu différents, et proposer l'affichage du chemin le plus court entre la station désignée par le programme et celle choisie par l'utilisateur. Nous avons choisi comme niveau le plus simple un niveau où seule une seule ligne s'affiche. Les deux niveaux suivants reprendront la carte entière de Paris, mais dans le premier les lignes s'affichent, alors que le second correspond à une carte entièrement vierge. Toutes ces contraintes nous ont obligés à nous tourner vers des algorithmes de théorie des graphes et de recherche opérationnelle, que nous décrirons dans la suite de ce rapport. Pour réaliser ce jeu, nous avons disposé d'environ cinq semaines.

Ce rapport permettra d'expliquer les choix que nous avons dû faire durant le développement de ce projet, et montrera les premiers résultats de notre programme, en attendant la version définitive qui sera montrée lors de la soutenance.

## 2. Organisation générale du code

L'organisation de notre code est relativement simple. Toute la partie événementielle et graphique est gérée dans le *main.cpp*, tandis que les algorithmes et les classes correspondant à de la théorie des graphes est répartie sur les fichiers *.cpp* et *.h*. Les classes et les méthodes ont été codées en C++, et l'interface graphique est codée en SDL. Le choix de cette librairie sera expliqué dans la dernière partie de ce paragraphe.

### 2.1 Structures de données

Nous avons choisi de séparer les deux modes de jeux principaux de notre programme, à savoir la partie où l'utilisateur joue avec toutes les stations, et celle où il ne joue qu'avec une seule ligne. Il y a donc une classe **Map**, qui représente la carte en entier avec les 299 stations, ainsi qu'une classe **Line**, qui ne représente qu'une seule ligne. Sur le principe, les deux classes fonctionnent de la même façon et ont beaucoup de méthodes et d'attributs quasiment identiques. Nous avons en plus une classe **Station**, qui permet de décrire une station avec ses coordonnées (**x,y**), son numéro (**numero**) et un tableau **lines[5]** qui répertorie les lignes passant par cette station. Pour pouvoir appliquer les algorithmes de théorie des graphes vus en cours, nous avons repris les classes **Noeud** et **ListeNoeuds** du TP2. Dans la suite, on prendra  $NBMAX = 300$ . En effet, les stations commencent de 1 (Abbesses) à 299 (Wagram).

- Construction de la carte entière de Paris

Pour construire la carte entière, nous avons donc rentré les 299 stations en mémoire dans une matrice **Station\* numeros[NBMAX]**. Les stations sont triées par ordre alphabétiques.

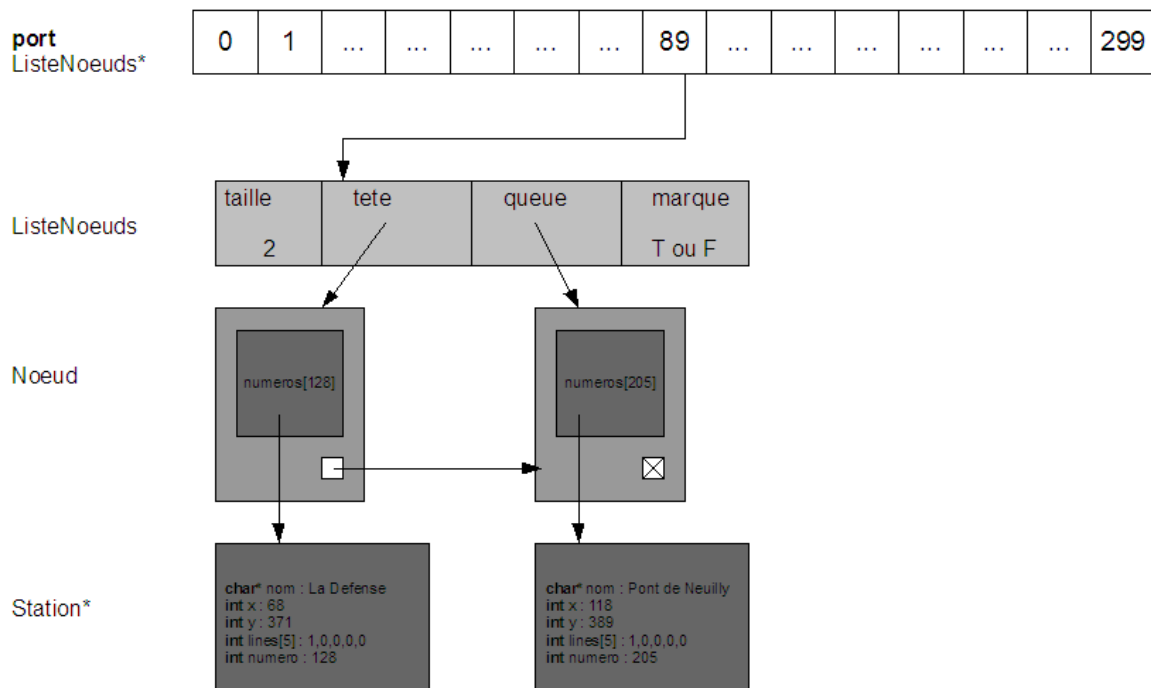
Par exemple, **cout << numeros[1]->nom**; affichera 'Abbesses'.

Les lignes sont ensuite construites une par une à l'aide d'une matrice **int\*\* liste**; qui répertorie les stations formant une même ligne.

Par exemple, **liste[14]** contiendra les numéros '263, 149, 233, 56, 105, 21, 69, 22, 183', qui sont les neuf stations composant la ligne 14. Comme nous ne travaillons qu'avec des entiers, nous avons choisi d'assimiler les lignes 3bis et 7bis à des lignes 15 et 16.

On aura ainsi **liste[15][0] = 102** (Gambetta), qui est la première station de la ligne 3bis.

Enfin, il ne reste plus qu'à faire les connexions entre les stations. Pour cela, nous avons utilisé une matrice **ListeNoeuds\* port[NBMAX]**; qui répertorie les voisins d'un noeud. Par exemple, pour la station numéro 89 (Esplanade de la Défense), on aura la représentation mémoire suivante :



- Lignes seules

La seule différence entre la carte entière et les lignes réside dans le fait que lorsque le constructeur de ligne est appelé, il alloue le tableau **Station\*\*numeros** en fonction de la longueur de la ligne. Ainsi, après avoir construit la ligne 2, on aura par exemple **cout << numeros[0] << endl;** qui affiche à l'écran 'Porte Dauphine'. L'autre différence est que la matrice **liste** n'est plus dans la classe **Line**, étant donné qu'on n'a besoin de construire qu'une seule ligne à la fois. Cependant, l'organisation générale reste la même, avec la matrice **port**. Lors de la construction des lignes, les zones dites linéaires se font automatiquement à l'aide d'une boucle. Mais pour les lignes où il y a des boucles ou des branches, il a fallu relier de façon plus « manuelle » les stations connectées.

Ces différentes structures de données ont l'avantage d'être simple. Cependant, l'inconvénient principal est qu'il a fallu coder les méthodes pour chaque classe, ce qui a été assez long. Mais cette organisation est parfaitement adaptée aux algorithmes mis en place pour la recherche du chemin le plus court.

### 2.2 Algorithmes utilisés

Afin d'apporter une solution efficace à ce projet, nous avons mis en place des algorithmes utilisés en théorie des graphes, notamment la recherche du chemin le plus court à partir d'un parcours en largeur (BFS, Breadth First Search). A l'aide d'une file d'attente, cet algorithme visite, à partir d'une station de départ, toutes les autres stations en passant par ses voisins successifs. Cette méthode a besoin de deux matrices **Station\* pred[NBMAX];** et **int l[NBMAX];**, qui correspondent respectivement au tableau des prédécesseurs de chaque station et de sa distance par rapport au point de départ (en nombre de stations). L'algorithme utilisé est le suivant :

```

Créer une file vide f

Marquer le sommet de départ s à TRUE

Enfiler s dans f

TANT QUE f non vide ET x est différent du sommet d'arrivée

    Défiler f dans une variable temporaire x

    POUR tous les sommets y voisins de x et marqués à FALSE

        Marquer y à TRUE

        Enfiler y dans f

        Stocker x dans la case pred[y] (x précède y)

        Stocker la distance entre s et y dans l[y] (on a l[y]=[x]+1)

    FIN POUR

FIN TANT QUE
  
```

L'algorithme s'arrête dès que le sommet d'arrivée a été défilé. Pour retrouver le chemin le plus court, il suffit de remonter les stations une par une, depuis la station d'arrivée, jusqu'à la station de départ. L'algorithme est donc :

```

Créer une pile vide bfs

Stocker la station d'arrivée dans une variable temporaire tmp

TANT QUE tmp est différent de la station de départ

    Empiler tmp dans bfs

    Stocker le prédécesseur de tmp dans tmp

FIN TANT QUE
  
```

Pour afficher le chemin le plus court, il n'y a plus qu'à parcourir la pile.

Voici un test de cet algorithme codé dans la classe **Map**. Nous avons voulu chercher le chemin le plus court entre Rue du Bac (252) et Bir-Hakeim (24) :

```

"D:\Docs\Docs ECE\ING2\Informatique\Théorie des graphes\Projet\test\bin\Debug\test.exe"
Entrez deux nombres entre 1 et 299
Depart : 24
Bir-Hakeim
Arrivee : 252
Rue du Bac
Appuyez sur une touche pour continuer... _
  
```

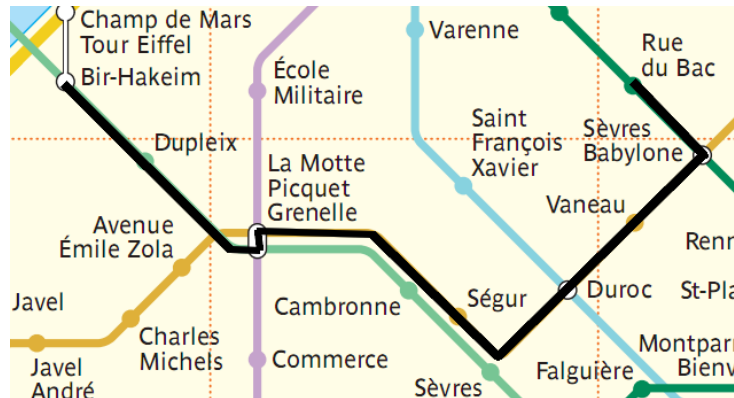
```

"D:\Docs\Docs ECE\ING2\Informatique\Théorie des graphes\Projet\test\bin\Debug\test.exe"
Plus court chemin entre Bir-Hakeim et Rue du Bac
Nb de stations : 7

Bir-Hakeim
Dupleix
La Motte-Picquet - Grenelle
Segur
Duroc
Vaneau
Sevres - Babylone
Rue du Bac

Process returned 0 (0x0)   execution time : 4.188 s
Press any key to continue.
_
  
```

En effet, après vérification sur un vrai plan de métro, il s'agit bien du chemin le plus court pour rallier ces deux stations.

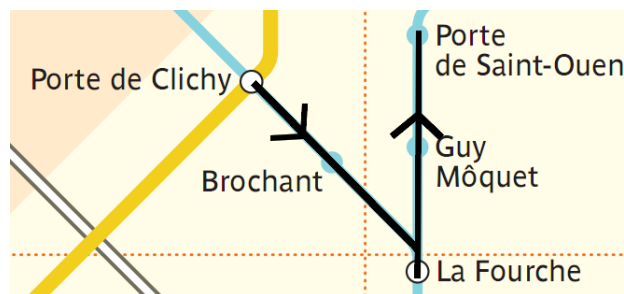


Nous avons également codé cette méthode pour les lignes seules, en suivant le même principe. Pour vérifier notre algorithme, nous avons décidé de chercher le plus court chemin entre Porte de Clichy (ligne 13, numéro 19) et Porte de Saint-Ouen (ligne 13, numéro 25).

```

D:\Docs\Docs ECE\ING2\Informatique\Théorie des graphes\Projet\test\bin\Debug\test.exe
Choisissez une ligne
13
Creation de la ligne 13
Selectionner une station de depart : 19
Selectionner une station d'arrivee : 25
Nb de stations : 4
Porte de Clichy
Brochant
La Fourche
Guy Mocquet
Porte de Saint-Ouen
Process returned 0 (0x0)   execution time : 11.235 s
Press any key to continue.
    
```

Le trajet se vérifie également sur la vraie carte du métro parisien. Par la même occasion, nous avons vérifié que les deux branches de cette ligne ont bien été connectées.



Attention, cet algorithme ne cherche pas le plus court chemin en terme de points, mais bien en terme de nombre de stations. Un chemin peut être plus court en terme de points (par exemple, 6 stations d'écart sans changement, ce qui vaut -6 points), mais plus long en terme de stations (par exemple, il peut exister un chemin qui ne fait que 3 stations, mais avec un

changement. Ceci correspond à -8 points). L'algorithme sélectionnera le deuxième, moins long en nombre de stations mais plus coûteux en points.

### 2.3 Interface graphique

Pour la partie graphique de notre jeu, nous avons choisi d'utiliser la librairie SDL.

Nous avons d'abord effectué un bref comparatif entre diverses librairies graphiques à notre portée. GTK nous semblait trop peu efficace au niveau de l'événementiel (par exemple, on ne peut envoyer qu'une donnée à une fonction callback). Qt, une librairie développée pour utiliser le C++, paraissait difficile à maîtriser suffisamment vite pour terminer le jeu dans le délai imparti. Il nous restait donc à choisir entre Allegro (apprise l'année dernière) et SDL (dont un bon tutoriel se trouve sur le Site du Zéro).

Ces deux librairies sont toutes deux relativement accessibles (leurs fonctions d'affichage fonctionnent d'ailleurs de façon assez similaire). Toutefois, les fonctions sont globalement plus simples (en termes de noms et de nombres de paramètres) du côté de SDL. De plus, l'affichage en SDL peut gérer le double buffer (qui empêche le clignotement de l'écran) de façon automatique sur simple déclaration à l'initialisation du mode vidéo de la librairie. Enfin, la librairie SDL possède une gestion des événements à la fois simple et puissante.

Comme Allegro, la librairie SDL ne gère nativement que l'affichage des BITMAP. Néanmoins, il nous a suffi d'installer le complément **SDL\_image** pour pouvoir utiliser des JPEG ou des PNG.

De même, afin d'afficher du texte dans notre jeu, nous avons installé la librairie additionnelle **SDL\_ttf**.

L'un des aspects les plus pratiques de SDL est que tout l'affichage (couleurs, images, textes...) se fait à partir d'un même type de variable : **SDL\_Surface**. Ainsi, ces éléments représentent en fait des rectangles définis par leurs dimensions et par leur position. Les dimensions d'une surface sont des sous-variables dont les valeurs sont détectées automatiquement à l'allocation de la surface (chargement de l'image, par exemple). La position de chaque surface, en revanche, doit être déclarée pour pouvoir l'afficher. En effet, toute surface a une position par rapport à la "surface générale" (en général appelée "**ecran**") qui définit le mode vidéo. Cette position est une structure standard de deux entiers x et y.

Une autre structure standard SDL, par exemple, est disponible pour déclarer une couleur. On utilise pour cela les trois paramètres R, G, B (composantes primaires) de la couleur.

La construction de l'écran se fait donc un peu comme avec Allegro : on charge les images ou les textes, on gère leurs dimensions et leurs coordonnées, on les plaque sur la surface finale, puis on affiche cette surface finale (cela se fait sous forme de "mise à jour" en SDL). C'est ce principe que nous avons utilisé pour mener à bien notre projet.

La partie la plus essentielle de ce jeu étant la gestion d'un graphe en mémoire, il nous a semblé plus cohérent de nous contenter d'un affichage sobre et allégé. Nous avons donc essayé d'afficher les éléments de façon à la fois complète et rapide.

Lorsque l'application est lancée, l'utilisateur voit d'afficher la fenêtre d'accueil. SDL ne permettant l'affichage que d'une seule fenêtre, celle-ci sera la même tout au long du jeu. Dès que l'événementiel entre en jeu, on se contente de « nettoyer » la surface principale correspondant à la fenêtre et de recommencer l'affichage.

Cet écran d'accueil montre le nom du jeu en haut de l'écran, ainsi que plusieurs zones de texte en bas de l'écran :

- « Règles » : affichage d'un court texte présentant les règles du jeu.
- « Jouer (niveau 1) » : lancement d'une partie au niveau facile.
- « Jouer (niveau 2) » : lancement d'une partie au niveau intermédiaire.
- « Jouer (niveau 3) » : lancement d'une partie au niveau difficile.
- « Quitter » : fin du programme.

Toutes ces zones de texte sont affichées à l'aide des fonctions de la librairie SDL\_ttf. Au même titre que l'image de fond, ce sont des surfaces SDL. La différence notable est qu'on affiche ici du texte, avec une police et une taille d'écriture définies dans des variables.



*Affichage de l'écran d'accueil*

On entre ici dans la partie événementielle du programme. En effet, l'affichage du menu est suivi d'une boucle durant laquelle un démon attend certaines actions de l'utilisateur. Par exemple, un clic sur la croix en haut à droite de la fenêtre ferme l'application. Et bien sûr, un clic gauche sur l'une des options du menu déclenche l'une des actions déjà citées. Pour cela, lorsque le clic gauche de la souris est détecté, on récupère les coordonnées courantes x et y de la souris. Si ces coordonnées sont comprises dans la surface d'une option, cette option est lancée.

Pour pouvoir quitter cette boucle événementielle, il faut bien sûr qu'elle dépende d'une condition simple. Nous avons donc utilisé une variable « continuer » qui est initialisée à 1. Si l'une des actions lancées lors de la boucle fait passer cette variable à 0, la boucle ne se répétera pas au tour suivant.

D'autre part, pour que les menus puissent être « consultés » à volonté (il faut par exemple que le joueur ait le temps de lire les règles du jeu), leur affichage est également suivi d'une « mini-boucle » événementielle dépendant d'une nouvelle variable de type « continuer ». Tout cela est codé à l'intérieur de la grande boucle événementielle.



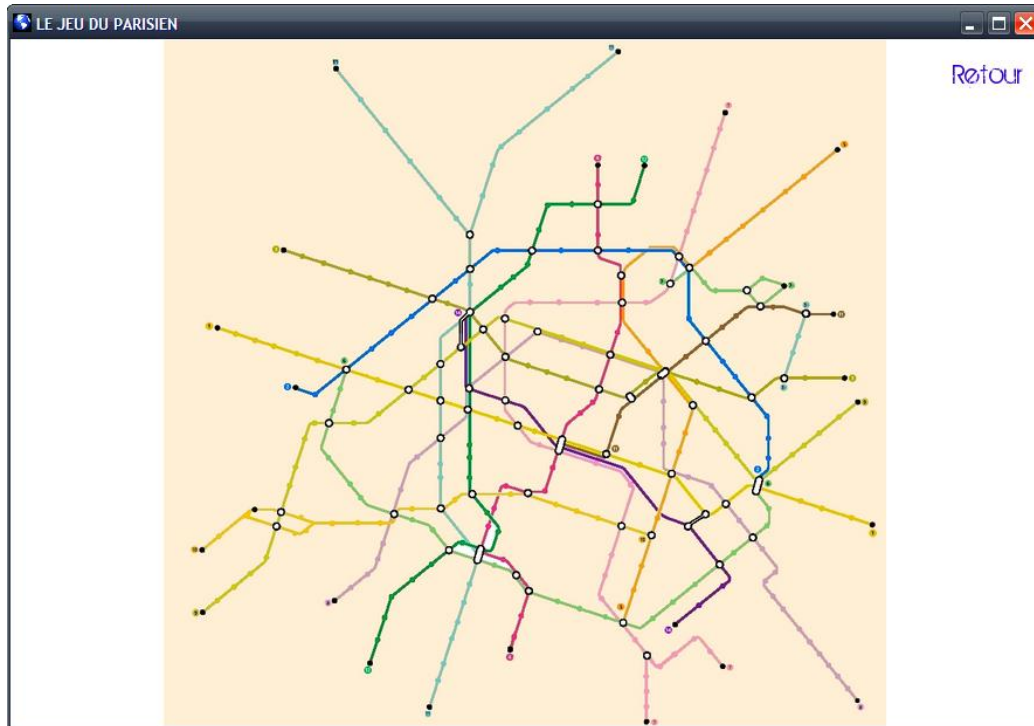
*Affichage des règles du jeu*

Une fois que le joueur a lancé une partie, l'affichage des éléments devient un peu plus complexe, puisqu'il dépend des variables caractéristiques du graphe.

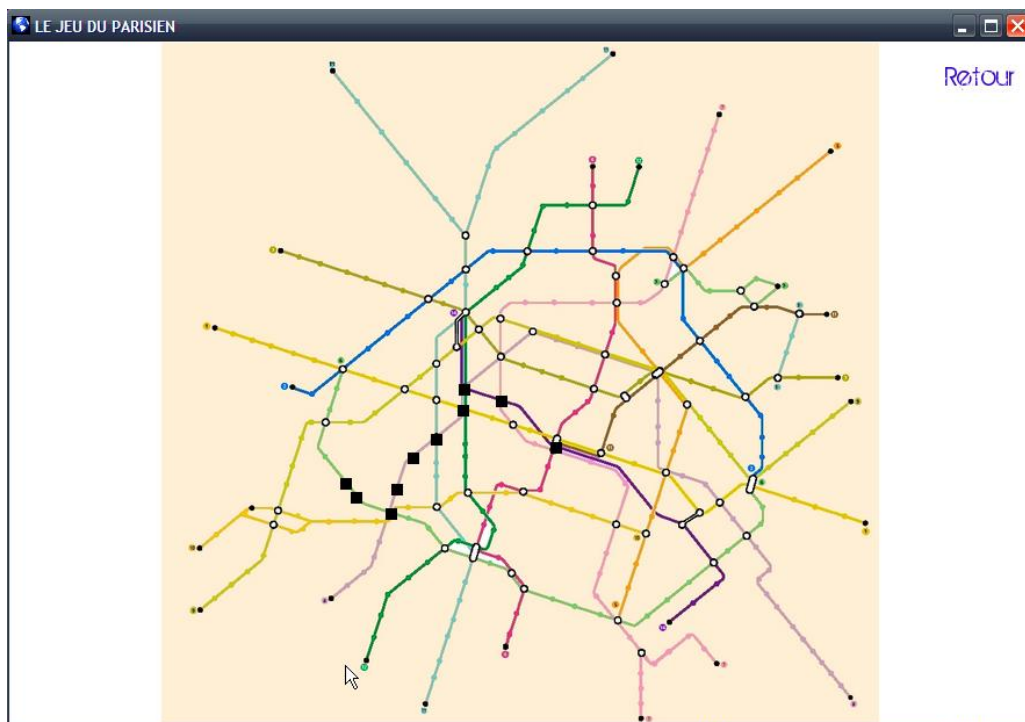
L'une des difficultés consiste à faire en sorte que le programme appelle les fonctions d'affichage au bon moment, afin d'éviter les problèmes d'affichage lors du déroulement de la boucle événementielle.

En ce qui concerne l'affichage du chemin sur la carte, nous avons récupéré manuellement les coordonnées de toutes les stations des différentes cartes que nous avons récupérées. Cette opération fut assez longue et laborieuse mais nous est bien utile pour ce qui est de la reconnaissance des stations sur nos différents plans. Cela nous permet d'afficher correctement la station qui clignotera et nous est d'une grande utilité pour la recherche du plus court chemin.

Voici quelques captures d'écran de l'affichage actuel de notre boucle de jeu. Ce n'est qu'une version temporaire de notre programme, tout comme les carrés noirs servant pour le moment à montrer le chemin le plus court entre deux stations.

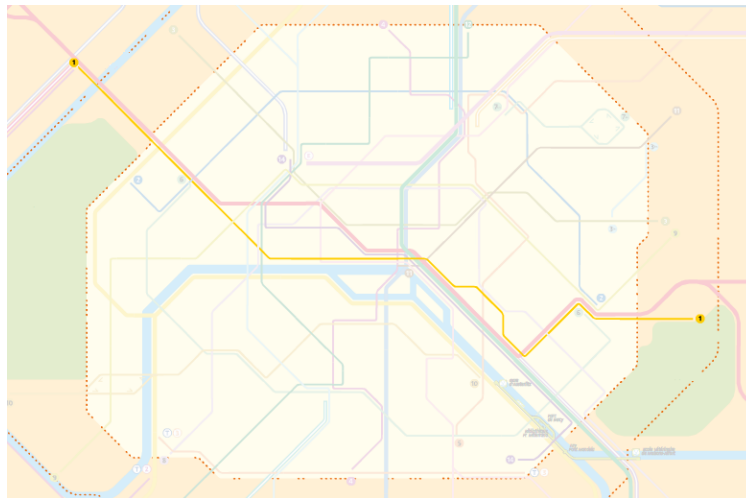


*Affichage du niveau 2*

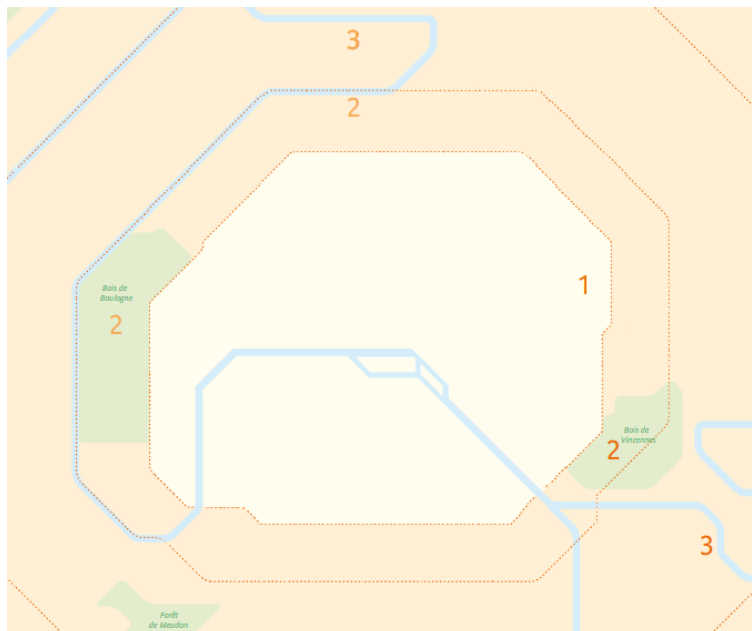


*Affichage d'un plus court chemin entre deux stations*

Voici les autres cartes que nous utiliserons pour notre jeu :



Carte du niveau 1 (avec les lignes seules)



Carte du niveau 3 (sans aucune ligne ni station)

### 3. Fonctionnement de l'équipe et conclusion

Au sein de l'équipe, Germain Le s'est occupé de la partie « mémoire », à savoir l'implémentation des classes et des méthodes, ainsi que la construction du graphe. Arnaud De Laborderie s'est chargé de l'aspect graphique (code en SDL), et Romain Llorca s'est chargé des cartes, ainsi que de la récupération des coordonnées des stations. Pendant toute la durée du projet, nous estimons avoir plutôt bien géré notre temps. Pour la soutenance, nous espérons avoir le jeu avec ses trois niveaux terminés.