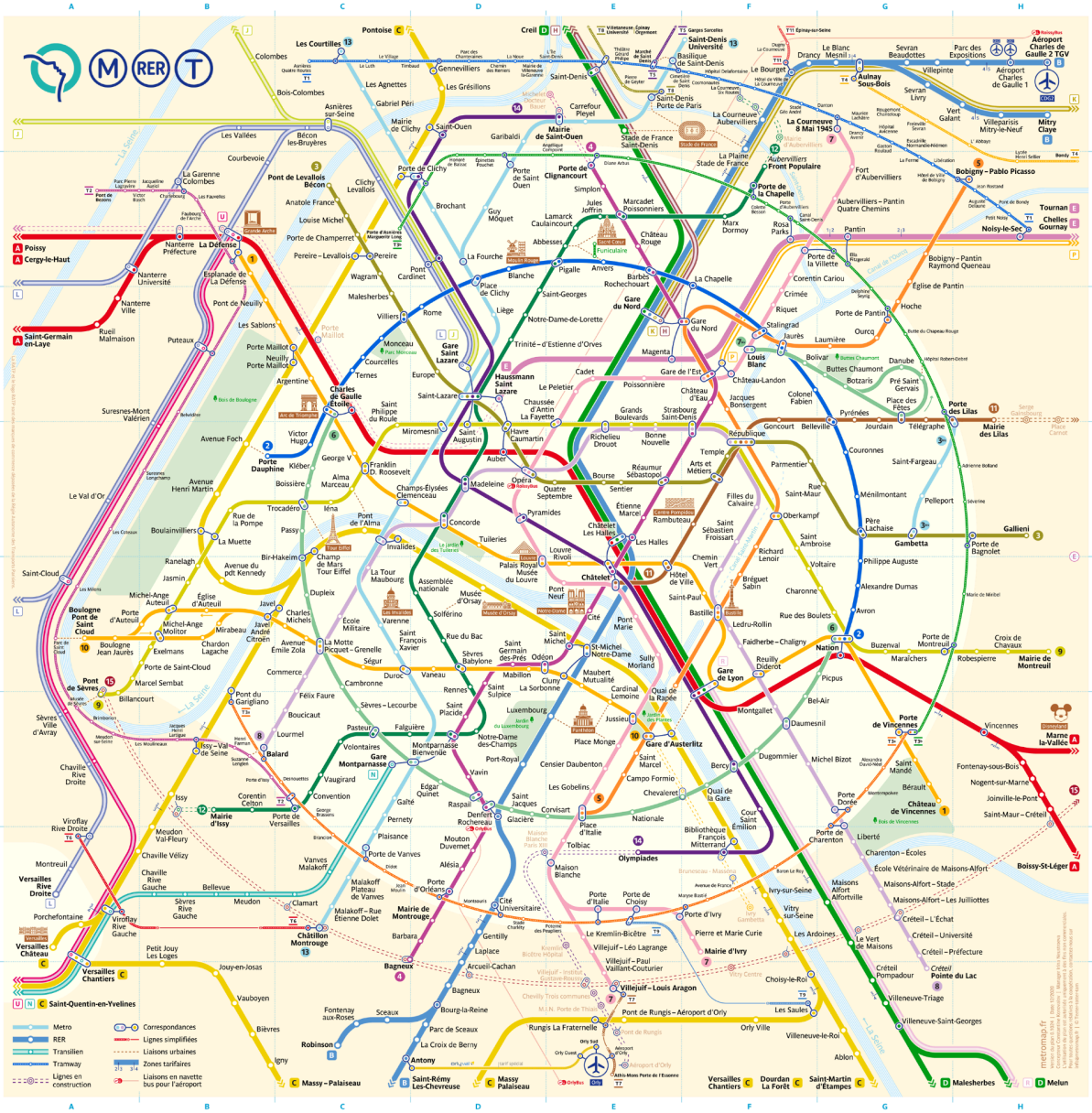
Date: 28/06/2023

SAÉ2.01 : Développement orienté objets

BRIAULT Maxence, JARNOUX Antoine, DE MOURA Thomas, Lotto Maxime, GELIS Adrien et PARISSE Chloé (Info Groupe C)



DÉPARTEMENT INFORMATIQUE

Plan

Présentation

Lors de cette SAÉ sur le développement orienté objets, nous avons eu le choix entre deux sujets. Le premier concernait la mise en place d'une application permettant de visualiser les trajets les plus adaptés et les moins risqués pour le transport de patients dans un contexte difficile. Le deuxième sujet concernait le réseau métropolitain de la capitale française dans lequel il était demandé de mettre en place une application nous spécifiant les trajets les plus courts entre deux stations (ou plus) souhaités.

Nous avons choisi le deuxième sujet car nous avons apprécié la perspective de développer une application qui pourrait nous être utile dans nos vies personnelles.

La problématique principale de ce sujet se positionne autour de la conception et du développement même de l'application. Il s'agissait de trouver des moyens d'optimiser les itinéraires tout en prenant en compte des potentielles contraintes supplémentaires comme les correspondances entre les arrêts, ceci en conservant une interface agréable pour l'utilisateur afin de rendre l'application facile et agréable d'utilisation.

Problématiques

Lors du début de cette SAE, nous avons défini le langage de développement que nous allions privilégier. Notre choix s'est arrêté sur le Java, un langage adapté au développement orienté objets. Par la suite, nous avons intégré les différentes classes dont nous allions avoir besoin au cours du déploiement de l'application. Ainsi nous avions, dès les premières heures dédiées au projet, déjà l'architecture générale de notre code.

Malgré tout, le choix de ce langage a demandé un temps d'adaptation à certains membres du groupe qui n'étaient pas habitués à ce langage de développement. Ils ont donc découvert de nouvelles notions en Java (comme les @Test).

Dans les données initiales du projet se trouvait l'ensemble des relations entre les différentes stations. La représentation de ces données nécessitait d'utiliser la théorie des graphes afin de modéliser les stations du réseau métropolitain. Cependant, nous avons rencontré des difficultés initiales en raison d'une confusion entre la notion de quai (données dans le fichier stations.csv) et notre propre conception d'une station. Par exemple, dans le fichier CSV, "Nation" représentait uniquement une ligne passant par la station Nation, alors que nous pensions que cela englobait l'ensemble de la station Nation.

Le choix de la représentation des résultats fut également un débat au sein du groupe. Nous ne savions pas précisément comment afficher les résultats des demandes effectuées par les utilisateurs. Nous avons fini par choisir une représentation visuelle sous forme de schéma des lignes. L'ensemble des lignes y seraient représentées et le chemin indiqué à l'utilisateur et mis en avant grâce à des une couleur particulière, faisant ressortir le trajet à emprunter. Ce choix nous a demandé beaucoup de temps car nous ne trouvions pas de carte vectorielle correspondant à notre besoin malgré nos recherches.

Ce choix de représentation a notamment été mené par une préoccupation de l'interface utilisateur (UI) de notre application. En effet, nous souhaitions proposer un site agréable à utiliser mais également agréable visuellement. Ce faisant, nous avons cherché à développer un univers graphique cohérent entre le sujet choisi et la conception graphique du site. Ce choix nous a également demandé un investissement particulier afin de s'assurer que nos attentes étaient satisfaites.

Enfin, une fois le code de l'application et le site mis en place, il fallait relier le front et le back end. Cette phase fut la dernière problématique rencontrée. En effet, nous avons dû faire en sorte que notre code soit bien relié au site précédemment créé.

Solutions

Afin de répondre aux différentes problématiques auxquelles nous avons été confrontées, nous avons eu différentes approches. Tout d'abord, concernant les notions en Java qui pouvaient parfois nous manquer, nous avons pu nous appuyer sur les connaissances et l'aide des professeurs présents lors des différentes heures accompagnées de ce projet. De plus, pour des besoins moins contraignants dans le développement du projet, de simples recherches pouvaient également nous indiquer une piste de résolution pour nos difficultés.

En ce qui concerne la réalisation du Graphe, notre professeur, Mr. Delechelle, nous a aidé a y voir plus clair en nous expliquant le concept de stations virtuelles pour représenter une correspondance. Après avoir implémenté cela, les problèmes que nous avions avec le Graphe ont été résolus.
De plus, nous avons migré les informations concernant les stations dans une base de données afin d'avoir un meilleur contrôle sur l'intégrité des données lors de l'insertion des prolongements des métros et des lignes de RER. En effet, dans le CSV donné au début, il manquait les stations ouvertes récemment, nous avons décidé de les ajouter. En ce qui concerne la récupération des données en termes de temps de parcours, nous avons chronométré des vidéo Youtube présentant les trajets souhaités.

Par la suite, le format de représentation des résultats a été abordé. Nous avions donc décidé d'une carte, mais aucune version pré-faite de cette carte ne nous convenait ou n'était au bon format. Nous avons ainsi choisis de refaire une carte (existante seulement en pixel) sous format vectoriel. Pour ce faire, nous avons utilisé le logiciel Inkscape, logiciel permettant de créer des formats vectoriel. La création de cette carte, bien que nous ayant demandé un grand investissement de temps, nous a permis de créer la représentation voulue ainsi que de gérer le graphisme de la carte, que nous avons ainsi pû adapter au mieux au graphisme choisi pour le site web de l'application.

La création de la carte s'est effectué via les étapes suivantes :

- Tout d'abord, nous avons dû décalquer la carte officielle d'IDFM afin de produire notre propre carte (Outils utilisés : InkScape).
- Ensuite, il a fallu découper le chemin SVG en tronçons pour chaque segment de station puis les nommer en « Origine – Arrivé ».
- En outre, il a été nécessaire de prendre un chemin en continu, soit sans branche dans le CSV (indiqué par les balises « parh ») puis fournir ces chemins en entrée du script SVGToJson.js. En ce faisant, le script créé en sortie un fichier JSON contenant l'Id, l'équation du Path, idOrigin, idArrivé tout en faisant bien attention à ne pas faire le chemin dans le mauvais sens. À noter que des paramètres peuvent être utilisés pour changer le sens en cas

d'erreur et que le script vérifie automatiquement la continuité du chemin pour prévenir en cas d'erreur).

- Pour suivre, il est requis de récupérer le fichier JSON créé par le script SVGToJson.js et de le fournir en argument d'entrée à getStationList.js. Dès lors, il est nécessaire d'écrire une par une chaque station suivie de son DisplayName (obtenu directement depuis le Wikipédia des lignes) ainsi que son Id récupéré depuis le fichier CSV. À noter que le script est interactif : les données sont entrées directement dans la console.
- Enfin, il suffit de récupérer le fichier JSON de la ligne produite par getStationList.js et de le glisser dans le dossier principal du projet (FrontEnd).

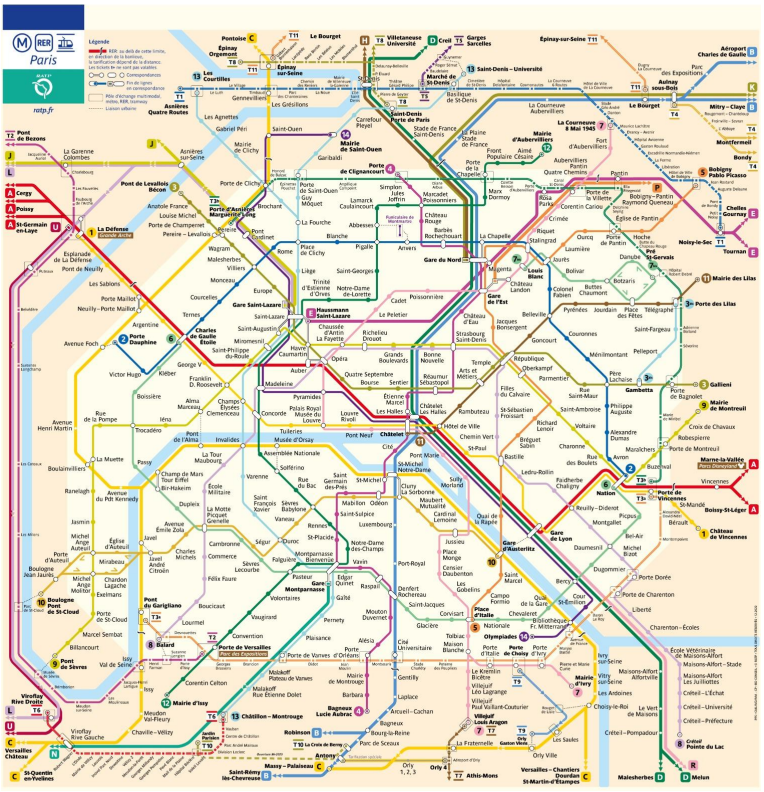
Ce site nous a également, comme précisé dans la rubrique Problématiques, demandé un effort conséquent. Nous avons dû mettre en relation le code en Java et le site internet. Pour ce faire, nous avons utilisé le framework Spring. Spring est un serveur web qui va écouter les requêtes HTTP et répondre. Lors du lancement de l'application, il va instancier l'objet "RéseauMetro" en lui fournissant les données disponible dans la base de données.

De l'autre côté, une application web sous react (next.js) va afficher la carte vectorielle et faire les requêtes nécessaires pour afficher le trajet le plus court. Pour la partie d'affichage de la carte vectorielle, cette partie est trop complexe pour être couverte ici. En résumé, nous utilisons le framework react afin de construire une image SVG en temps réel en fonction des donnés tels que l'équation de la courbe d'un tronçons (courbe entre deux stations) et la position des stations couplé à des données supplémentaire pour contrôler la méthode d'affichage des stations (par exemple : les points des stations en correspondance ne s'affiche pas, les stations superposé ont un contour noir.). Lors de l'affichage du trajet, un algorithme décide quels tronçons affichés en fonction du trajet.

Résultats

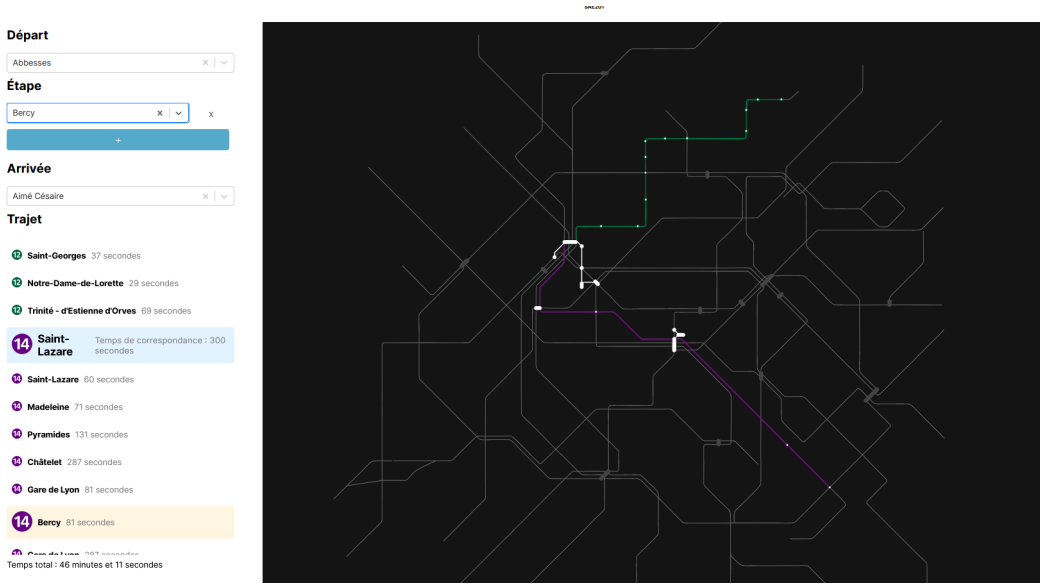
Ainsi dans cette SAÉ, nous avons développé plusieurs méthodes de classe nous permettant d'analyser et d'utiliser le réseau métropolitain parisien. Les différentes méthodes nous permettent de calculer les arbres de couvrance de poids minimal, mais aussi les distances entre des stations et des correspondances, ou encore les différences de distances entre deux stations et un terminus.

Pour ce faire, nous utilisons un site web développé à cet effet présentant une carte vectorielle (d'après le modèle ci-dessous) mettant en valeur le trajet entre les deux stations voulues. Ce trajet est mis en valeur grâce au format de la carte (vectoriel) qui nous permet de colorer le trajet minimal préalablement calculé par l'application développée.



En plus de cela, nous avons porté une grande attention à l'expérience utilisateur de notre site web. Ainsi, la carte est intéractive : les informations des stations sont précisées lorsque l'on survole les stations avec notre souris. Le temps de trajet prévu est également donné lorsque l'on demande un trajet entre deux stations précises.

Ainsi, nous avons un site prenant cette forme (voir ci-dessous) sur lequel nous pouvons choisir la station de départ et d'arrivée, ainsi que des préférences de trajet (chemin le plus court, correspondance). Ici, nous voulons aller de la station Abbesses à la station Aimé Césaire en passant par la station Bercy. Sur la gauche du site, en plus des détails du trajet demandé, nous affichons également les stations par lesquelles l'utilisateur va passer ainsi que leur ligne respective et le temps de trajet entre chacune des stations du trajet.



Conclusion

Ainsi, le développement de cette application orientée objet en Java nous a permis de nous exercer sur des problématiques réelles. En effet, la mise en place de cette application et notamment de son algorithme de calcul de distance de graphe nous a réellement entraînés dans les ressources R2.01 et R2.07 afin de pouvoir analyser les données mises à disposition et les analyser. La mise en place d'une sorte de "GPS" adapté au réseau métropolitain parisien est une tâche qui nous a réellement inspirée car nous aimions penser que nous pourrions nous même utiliser cette application dans notre vie personnelle.
De cette manière, nous avons une application fonctionnelle et facile d'utilisation qui pourrait être déployée pour une utilisation publique car l'ensemble de l'expérience utilisateur a été étudié pour rendre le parcours sur le site accessible à l'ensemble des utilisateurs.

Les pistes d'améliorations que nous avons est notamment d'utiliser les données en temps réel du trafic sur le réseau. En effet, si nous imaginons une application accessible au public, l'incorporation dans nos calculs des différentes heures de pointe, accidents et ralentissements est nécessaire pour le fonctionnement et l'attractivité de cette application.

Dans l'ensemble, ce projet nous a permis de consolider nos compétences en programmation Java et dans la visualisation de données. Nous sommes fiers du travail accompli et convaincus que les compétences acquises seront précieuses pour nos projets futurs.

Annexes

Présentation des différentes données retournées en fonction des méthodes appelées:

Algorithme de Dijkstra

```
 */
public List<Relation> dijkstra_algo(Quai station1, Quai station2, TrajectPreference pref) {
    Map<Quai, Integer> distances = new HashMap<Quai, Integer>();
    Map<Quai, Relation> anteriorite = new HashMap<Quai, Relation>();
    Set<Quai> quaiTraitees = new HashSet<Quai>();

    ArrayList<Relation> relations = new ArrayList<Relation>(this.relations);
    ArrayList<Quai> quais = new ArrayList<Quai>(this.quais);

    // il faut ajouter aux relations déjà présente la relation entre les quai de la
    // station
    if (station1.virtuel) {
        if (!this.stations.keySet().contains(station1)) {
            System.out.println(
                    "Erreur : la station " + station1 + " n'existe pas dans le réseau");
            return null;
        }

        quais.add(station1);

        // on ajoute les relations entre la station et les quais de la station
        System.out.println(
                Couleurs.UNDERLINE + "\nAjout des relations de " + station1.getNom() + " :" + Couleurs.RESET);
```

Algorithme de Bellman Ford

```
static void BellmanFord(int graph[][], int sommet, int arete,
                        int src)
{
    // Initialise la distance de tous les sommets à l'infini.
    int []dis = new int[sommet];
    for (int i = 0; i < sommet; i++)
        dis[i] = Integer.MAX_VALUE;

    // initialise la distance de la source à 0
    dis[src] = 0;

    // Relaxe toutes les arêtes |sommet| - 1 fois. Un chemin le plus court
    // simple de la source à n'importe quel autre
    // sommet peut avoir au plus |sommet| - 1 arêtes
    for (int i = 0; i < sommet - 1; i++)
    {

        for (int j = 0; j < arete; j++)
        {
            if (dis[graph[j][0]] != Integer.MAX_VALUE && dis[graph[j][0]] + graph[j][2] <
                                dis[graph[j][1]])
                dis[graph[j][1]] =
                dis[graph[j][0]] + graph[j][2];
        }
    }
```

Code de la méthode “correspondanceEntre2Lignes”

```java
/**
public void correspondanceEntre2Lignes(String ligne1, String ligne2) {
    // Vérifier si les lignes existent dans la liste des quais
    boolean ligne1Existe = quais.stream().anyMatch(quai -> quai.getLigne().equals(ligne1));
    boolean ligne2Existe = quais.stream().anyMatch(quai -> quai.getLigne().equals(ligne2));

    if (!ligne1Existe || !ligne2Existe) {
        throw new IllegalArgumentException(
                s:"Au moins l'une des lignes spécifiées n'existe pas.");
    }

    // Rechercher les correspondances possibles entre les deux lignes
    List<Quai> correspondances = new ArrayList<>();

    // Parcourir tous les quais
    for (java.util.Map.Entry<Quai, Set<Quai>> entry : this.stations.entrySet()) {
        Quai quai = entry.getKey();
        Set<Quai> correspondancesQuai = entry.getValue();

        boolean contientLigne1 = false;
        boolean contientLigne2 = false;

        for (Quai correspondance : correspondancesQuai) {
            if (correspondance.getLigne().equals(ligne1)) {
                contientLigne1 = true;
            }
            if (correspondance.getLigne().equals(ligne2)) {
                contientLigne2 = true;
            }
```

Affichage de la méthode “correspondanceEntre2Lignes” entre M7 / M4 et M6 / M7

```
Correspondances entre M7 et M4:

Correspondances possibles entre les lignes M7 et M4:
Châtelet
Gare_de_l'Est

Correspondances entre M6 et M7:

Correspondances possibles entre les lignes M6 et M7:
Place_d'Italie

Temps estimé du trajet entre CDGEtoile et Nation (trajet précédent): 832 secondes
```

Code de l’algorithme ACM

```java
 * proposer un algorithme qui permet de trouver l’arbre couvrant minimum (ACM)
 * reliant toutes
 * les stations. En quoi ce réseau serait-il avantageux ou pas pour la RATP ? et
 * pour les
 * utilisateurs ?
 */
public List<Relation> ACM() {
    // Copier les relations existantes
    List<Relation> allRelations = new ArrayList<>(relations);

    // Trier les relations par ordre croissant de temps
    allRelations.sort(new RelationComparator());

    // Créer une liste pour stocker les relations de l'ACM
    List<Relation> acm = new ArrayList<>();

    // Créer un tableau pour stocker les parents de chaque quai
    int[] parent = new int[quais.size()];
    for (int i = 0; i < quais.size(); i++) {
        parent[i] = i;
    }

    // Parcourir toutes les relations triées
    for (Relation relation : allRelations) {
        Quai st1 = relation.getSt1();
        Quai st2 = relation.getSt2();

        // Vérifier si l'ajout de cette relation crée un cycle
        if (!connected(parent, st1, st2)) {
```

Affichage de la méthode ACM

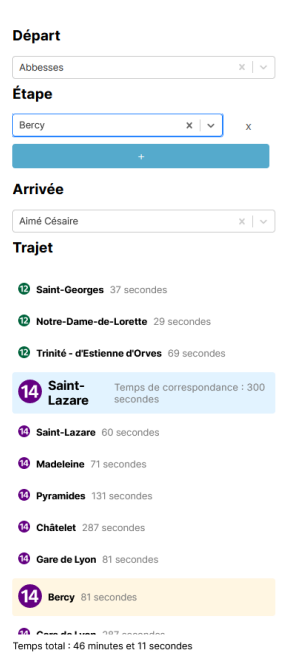
```
- - - ACM - - -

Nombre de relations totale du réseau: 495
Nombre de relations trouvées par l'ACM: 390
Nombre de stations virtuelles trouvées: 60
```

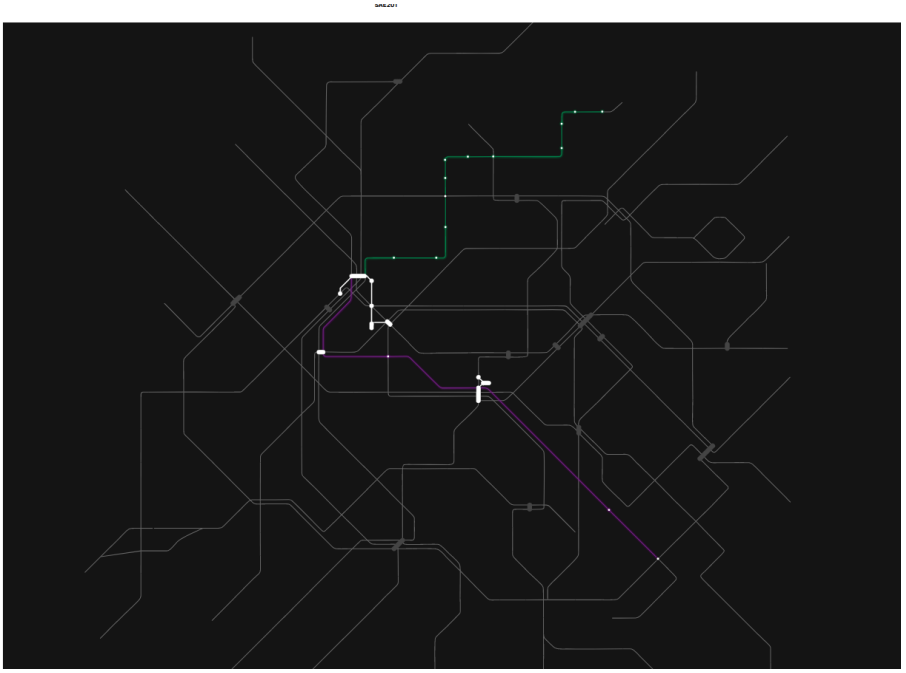
Code méthode comparerStation_A_B

```java
public void comparerStation_A_B(Quai station1, Quai station2, int pDistance,
        TypeAnalyse typeAnalyse) {
    List<Quai> corresp = this.stations.keySet().stream().filter(s -> s.virtuel).collect(Collectors.toList());

    switch (typeAnalyse) {
        case ACCESSIBLE:
            int distanceMinStation1 = Integer.MAX_VALUE;
            Quai stationMinStation1 = null;

            int distanceMinStation2 = Integer.MAX_VALUE;
            Quai stationMinStation2 = null;

            // on va parcourir toutes les correspondances
            for (Quai qVirt : corresp) {
                // on ne prend pas en compte la station elle-même
                if (!qVirt.equals(station1)) {
                    System.out.println(
                            "Trajet entre " + station1.getNom() + " et " + qVirt.getNom());
                    // on calcule le trajet entre la station et la correspondance
                    // on récupère toutes les relations
                    List<Relation> relations = dijkstra_algo(station1, qVirt);
                    // on calcule la distance totale
                    int distance = relations.stream().reduce(identity:0, (acc, r) -> acc + r.getTemps(),
                            Integer::sum);

                    // on compare la distance avec la distance minimale
                    if (distance < distanceMinStation1) {
                        distanceMinStation1 = distance;
```

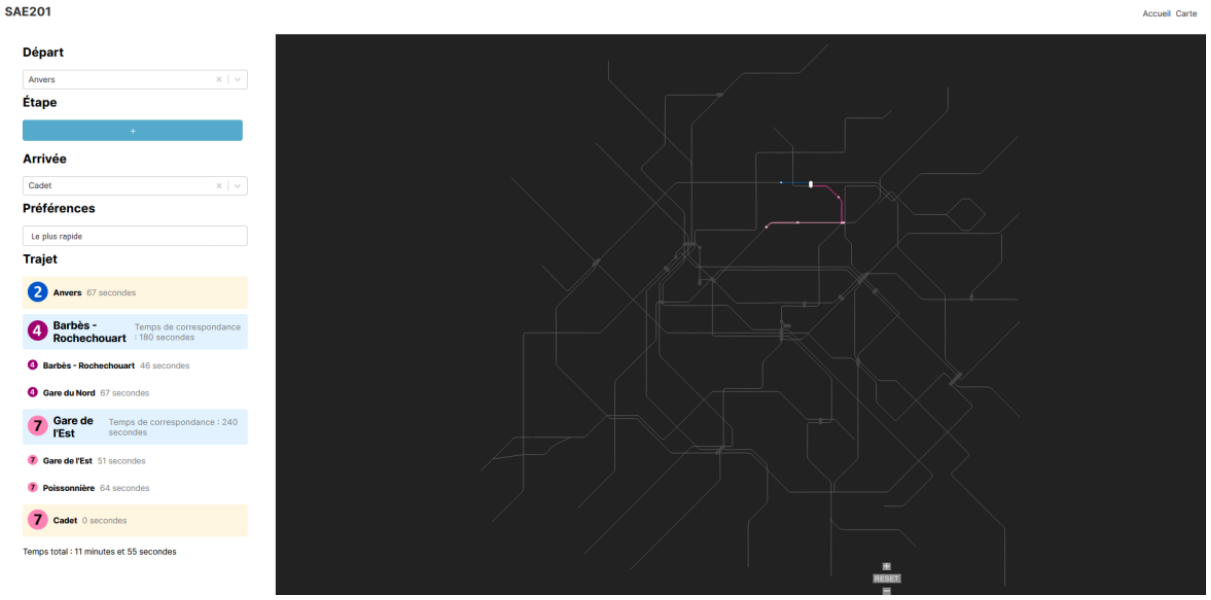
<u>Affichages des différentes possibilités de trajet sur la carte grâce au site:</u>

Trajet en passant par une station donnée





Trajet avec le chemin le plus court



Trajet avec le moins de correspondance possible