Rapport PSI Livrable 2

Oscar MARESCHAL

Myriam KHOUJA

Juliette MOTTAY

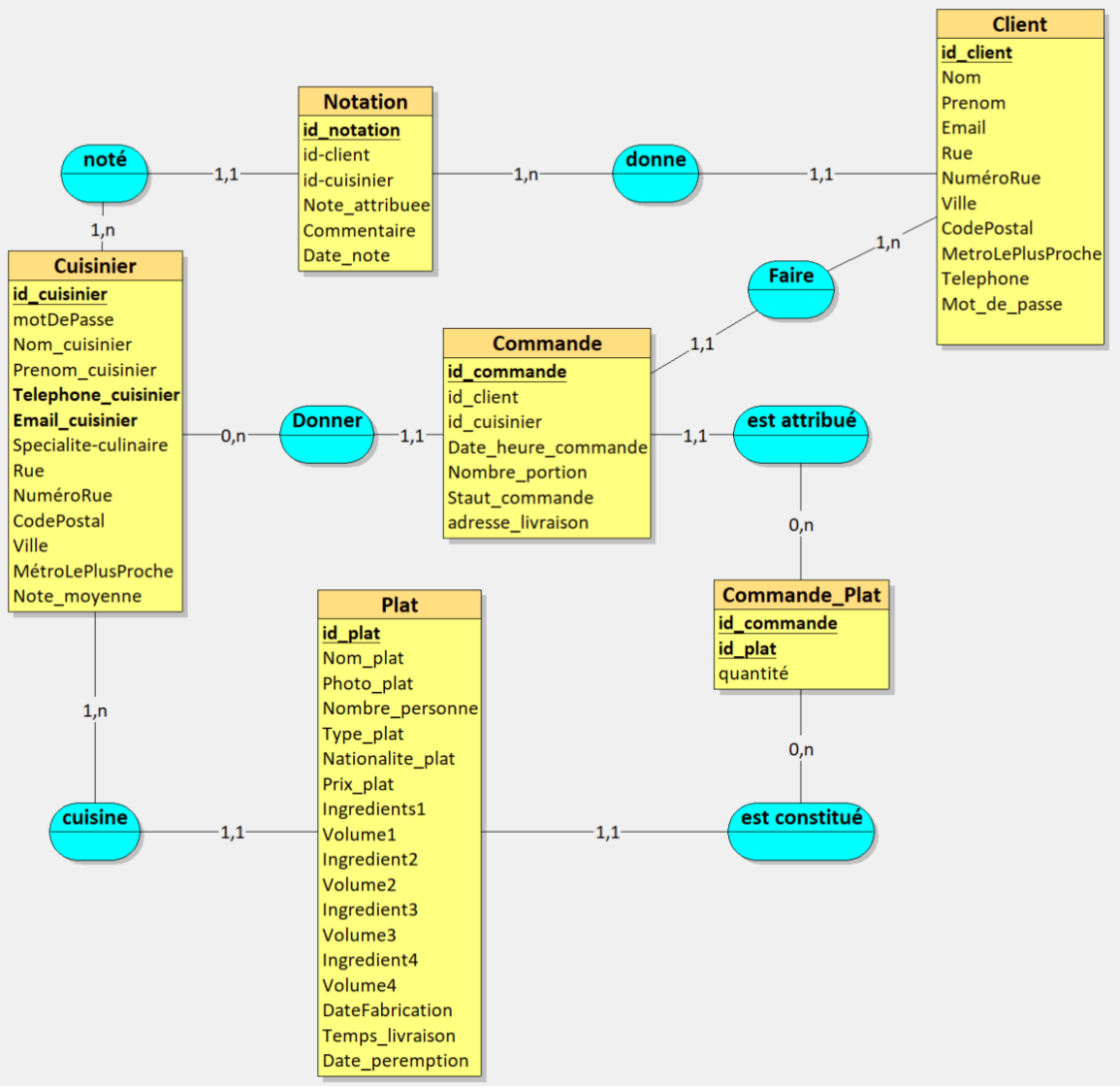
TD B

Lien GitHub :

Livrable-1---MARESCHAL-Oscar---KHOUJA-Myriam---MOTTAY-Juliette

Partie BDD :

Schéma Entité Association



Script SQL de la création de la base

```sql
USE LivInParis;
DROP TABLE commande_plat;
DROP TABLE Commande;
DROP TABLE Notation;
DROP TABLE Client;
DROP TABLE Cuisiner;
DROP TABLE plat;
DROP TABLE Cuisinier;

CREATE TABLE Client (
    id_client INT PRIMARY KEY AUTO_INCREMENT,
    nom VARCHAR(100) NOT NULL,
    prenom VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    rue VARCHAR(100),
    numeroRue VARCHAR(10),
    codePostal INT,
    ville VARCHAR(100),
    metroLePlusProche VARCHAR(100) NOT NULL,
    telephone VARCHAR(20) UNIQUE NOT NULL,
    mot_de_passe VARCHAR(255) NOT NULL
);

CREATE TABLE Cuisinier (
    id_cuisinier INT PRIMARY KEY AUTO_INCREMENT,
    nom VARCHAR(100) NOT NULL,
    prenom VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    telephone VARCHAR(20) UNIQUE NOT NULL,
    rue VARCHAR(100),
    numeroRue VARCHAR(10),
    codePostal INT,
    ville VARCHAR(100),
    metroLePlusProche VARCHAR(100) NOT NULL,
    specialite_culinaire VARCHAR(100),
    mot_de_passe VARCHAR(255) NOT NULL,
    note_moyenne DECIMAL(3,2) DEFAULT 0
);

```

```sql
CREATE TABLE Plat (
    id_plat INT PRIMARY KEY AUTO_INCREMENT,
    id_cuisinier INT NOT NULL,
    nom VARCHAR(100) NOT NULL,
    photo VARCHAR(255),
    nombre_personne INT NOT NULL,
    type_plat VARCHAR(50) NOT NULL,
    nationalite_plat VARCHAR(50) NOT NULL,
    prix DECIMAL(10,2) NOT NULL,
    ingredients TEXT NOT NULL,
    nombre_portion INT NOT NULL,
    date_fabrication DATE NOT NULL,
    date_peremption DATE NOT NULL,
    FOREIGN KEY (id_cuisinier) REFERENCES Cuisinier(id_cuisinier) ON DELETE CASCADE

);

CREATE TABLE Commande (
    id_commande INT PRIMARY KEY AUTO_INCREMENT,
    id_client INT NOT NULL,
    id_cuisinier INT NOT NULL,
    id_plat INT NOT NULL,
    date_heure_commande DATETIME NOT NULL,
    nombre_portion INT NOT NULL,
    statut_commande ENUM('En attente', 'En cours', 'Livré', 'Annulé') DEFAULT 'En attente',
    adresse_livraison VARCHAR(255) NOT NULL,
    FOREIGN KEY (id_client) REFERENCES Client(id_client),
    FOREIGN KEY (id_cuisinier) REFERENCES Cuisinier(id_cuisinier),
    FOREIGN KEY (id_plat) REFERENCES Plat(id_plat)
);
-- table pour gérer les plats commandés
CREATE TABLE Commande_Plat (
    id_commande INT NOT NULL,
    id_plat INT NOT NULL,
    quantite INT NOT NULL DEFAULT 1,
    PRIMARY KEY (id_commande, id_plat),
    FOREIGN KEY (id_commande) REFERENCES Commande(id_commande) ON DELETE CASCADE,
    FOREIGN KEY (id_plat) REFERENCES Plat(id_plat) ON DELETE CASCADE
);

CREATE TABLE Notation (
    id_notation INT PRIMARY KEY AUTO_INCREMENT,
    id_commande INT NOT NULL,
    id_client INT NOT NULL,
    id_cuisinier INT NOT NULL,
    note_attribuee DECIMAL(2,1) CHECK (note_attribuee BETWEEN 0 AND 5),
    commentaire TEXT,
    date_note DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (id_client) REFERENCES Client(id_client),
    FOREIGN KEY (id_cuisinier) REFERENCES Cuisinier(id_cuisinier)
);
```

Partie Algo et Graphe :

Prompts issus des IA génératives pour visualiser le graphe:

Visualisation du Graphe Métro en Utilisant des IA génératives

Exemples de prompts utilisés pour guider l'IA :

> *"Génère une classe en C# permettant d'afficher un graphe orienté avec sommets positionnés selon des coordonnées GPS (latitude, longitude)."*
> *"Écris un algorithme qui lit deux fichiers CSV représentant les stations et les connexions du métro parisien, puis construit un graphe orienté prêt à être affiché dans un `Form` Windows."*
> *"Ajoute la possibilité de zoomer dans la fenêtre graphique pour une exploration intuitive du graphe."*
> *"Affiche les informations du graphe dans la console*

Résultat obtenu :

> Chaque station est représentée par un sommet positionné en fonction de ses coordonnées GPS.
> Les liaisons entre stations sont représentées par des arcs orientés avec flèches
> L'utilisateur peut zoomer et dézoomer
> Toutes les données (stations, connexions) sont également affichées dans la console

Comparaison algo de chemin le plus court

- Dijkstra

L'algorithme de Dijkstra sert à trouver le plus court chemin entre un nœud de départ et tous les autres nœuds dans un graphe pondéré. Il fonctionne en sélectionnant à chaque étape le nœud non visité avec la plus petite distance estimée, puis en mettant à jour les distances de ses voisins. Nous avons été contraints t'utiliser la commande Var dans nos algorithmes car la variable Connexion ne fonctionnait pas et nous affichait une erreur nous avons donc utilisés la commande var sans trouver d'autres solutions.

```
public List<Station> Dijkstra(int départId, int arrivéeId)
{
    bool départ = false;
    bool arrivée = false;

    foreach (Station station in Stations)
    {
        if (station.Id == départId || station.Nom == "départ")
        {
            départ = true;
        }
        if (station.Id == arrivéeId || station.Nom == "arrivée")
        {
            arrivée = true;
        }

        if (départ && arrivée)
        {
            break;
        }
    }

    if (!départ || !arrivée)
    {
        Console.WriteLine("Une des stations n'existe pas");
        return null;
    }

    Dictionary<int, double> distance = new Dictionary<int, double>();
    Dictionary<int, int> station_avant = new Dictionary<int, int>();
    List<int> station_nonvisité = new List<int>();
```

```
    foreach (Station station in Stations)
    {
        station_nonvisité.Add(station.Id);
        if (station.Id == départId)
        {
            distance[station.Id] = 0;
        }
        else
        {
            distance[station.Id] = double.MaxValue;
        }
        station_avant[station.Id] = -1;
    }

    while (station_nonvisité.Count > 0)
    {
        int currentId = -1;
        double distance_min = double.MaxValue;

        foreach (int id in station_nonvisité)
        {
            if (distance[id] < distance_min)
            {
                distance_min = distance[id];
                currentId = id;
            }
        }

        if (currentId == arrivéeId || distance[currentId] == double.MaxValue)
        {
            break;
        }

        station_nonvisité.Remove(currentId);
```

```
List<int> station_voisine = new List<int>();
foreach (var connexion in Connexions)
{
    if (connexion.From.Id == currentId)
    {
        station_voisine.Add(connexion.To.Id);
    }
    else if (connexion.To.Id == currentId)
    {
        station_voisine.Add(connexion.From.Id);
    }
}

Station stationActuelle = null;
foreach (Station station in Stations)
{
    if (station.Id == currentId)
    {
        stationActuelle = station;
        break;
    }
}
if (stationActuelle == null)
{
    Console.WriteLine("Station avec l'ID " + currentId + " non trouvée.");
}
for (int i = 0; i < Stations.Count; i++)
{
    Station autreStation = Stations[i];
    if (autreStation.Nom == stationActuelle.Nom && autreStation.Id != currentId)
    {
        station_voisine.Add(autreStation.Id);
    }
}
```

```
foreach (int voisinId in station_voisine)
{
    if (!station_nonvisité.Contains(voisinId)) continue;

    double alt = distance[currentId] + 1;

    Station currentStation = null;
    foreach (Station station in Stations)
    {
        if (station.Id == currentId)
        {
            currentStation = station;
            break;
        }
    }

    Station voisinStation = null;
    foreach (Station station in Stations)
    {
        if (station.Id == voisinId)
        {
            voisinStation = station;
            break;
        }
    }

    bool changementDeLigne = true;
    foreach (string ligneActuelle in currentStation.Lignes)
    {
        if (voisinStation.Lignes.Contains(ligneActuelle))
        {
            changementDeLigne = false;
            break;
        }
    }

    if (changementDeLigne)
    {
        alt += 1;
    }
```

```
            if (alt < distance[voisinId])
            {
                distance[voisinId] = alt;
                station_avant[voisinId] = currentId;
            }
        }
    }

    List<Station> path = new List<Station>();
    int current = arrivéeId;

    while (current != -1)
    {
        Station station = null;
        foreach (Station s in Stations)
        {
            if (s.Id == current)
            {
                station = s;
                break;
            }
        }
        if (station == null)
        {
            Console.WriteLine("Station avec l'ID " + current + " non trouvée.");
        }
        path.Insert(0, station);
        current = station_avant[current];
    }

    if (path.Count == 0 || path[0].Id != départId)
    {
        Console.WriteLine("Aucun chemin trouvé de " + départId + " à " + arrivéeId);
        return null;
    }

    return path;
}
```

Dijkstra

Avantages :

- **Efficacité** : Pour les graphes avec des poids positifs, Dijkstra est très rapide, particulièrement avec une structure de tas binaire, avec une complexité de **O((V + E) log V)**.
- **Précision** : Il trouve le plus court chemin de manière optimale en tenant compte des poids des arêtes.

Inconvénients :

- **Poids négatifs** : Il ne fonctionne pas si des arêtes ont des poids négatifs.
- **Dépendance à la structure de données** : Sa performance peut être affectée par le choix de la structure de données (tableaux simples vs tas binaires).

Bellman-Ford

Avantages :

- **Poids négatifs** : Il peut gérer les graphes avec des poids négatifs, contrairement à Dijkstra.
- **Détection de cycles négatifs** : Il peut détecter les cycles négatifs dans le graphe.

Inconvénients :

- **Complexité plus élevée** : Sa complexité est **O(V * E)**, ce qui le rend plus lent que Dijkstra, surtout pour des graphes denses.
- **Moins efficace** : Par rapport à Dijkstra, il est moins performant sur des graphes sans poids négatifs.

```
for (int i = 1; i < Stations.Count; i++)
{
    foreach (var connexion in Connexions)
    {
        int departId = connexion.From.Id;
        int arivéeId = connexion.To.Id;

        double alt = distance[départId] + 1;

        if (distance[départId] != double.MaxValue && alt < distance[départId])
        {
            distance[arivéeId] = alt;
            station_avant[arivéeId] = départId;
        }

        alt = distance[arivéeId] + 1;
        if (distance[arivéeId] != double.MaxValue && alt < distance[départId])
        {
            distance[départId] = alt;
            station_avant[départId] = arivéeId;
        }
    }
}
```

```
foreach (var connexion in Connexions)
{
    int departId = connexion.From.Id;
    int arivéeId = connexion.To.Id;

    double alt = distance[départId] + 1;

    if (distance[départId] != double.MaxValue && alt < distance[arivéeId])
    {
        Console.WriteLine("Le graphe contient un cycle négatif.");
        return null;
    }

    alt = distance[arivéeId] + 1;
    if (distance[arivéeId] != double.MaxValue && alt < distance[départId])
    {
        Console.WriteLine("Le graphe contient un cycle négatif.");
        return null;
    }
}
```

Floyd-Warshall

Avantages :

- **Toutes les paires de chemins** : Il calcule le plus court chemin entre toutes les paires de nœuds dans un graphe, ce qui le rend utile dans des situations où tous les chemins sont nécessaires.
- **Simplicité** : Il est relativement simple à implémenter.

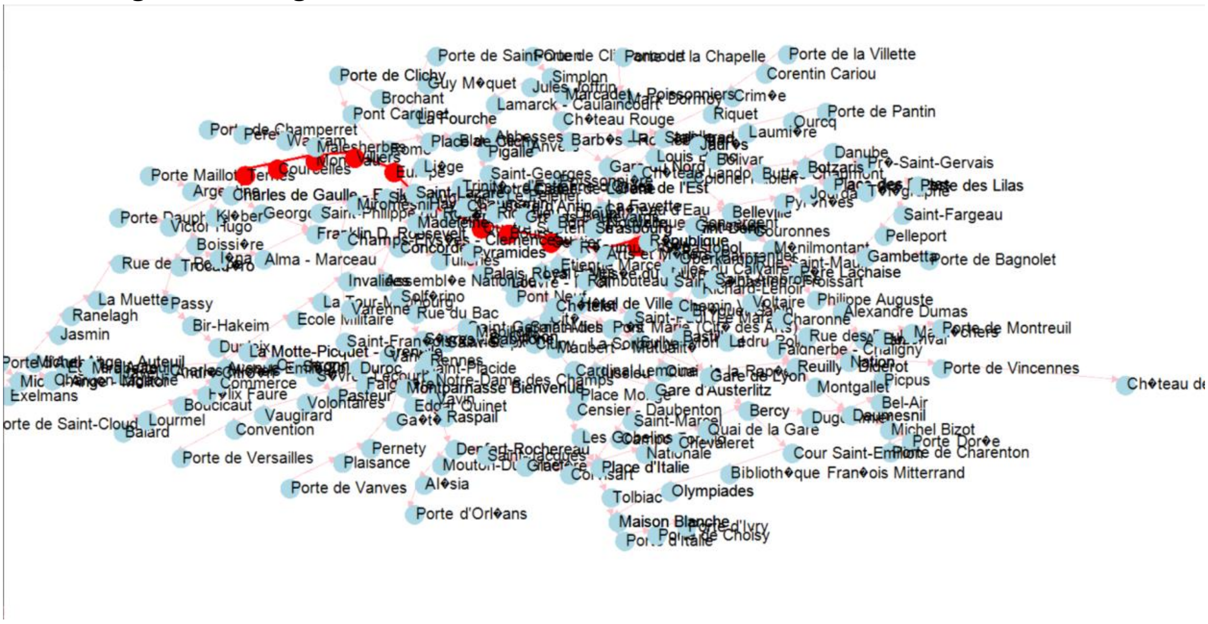
Inconvénients :

- **Complexité élevée** : Sa complexité est **O(V^3)**, ce qui le rend peu adapté aux grands graphes.
- **Utilisation de la mémoire** : Il nécessite une matrice de **V x V** pour stocker les distances, ce qui peut consommer beaucoup de mémoire pour des graphes volumineux.

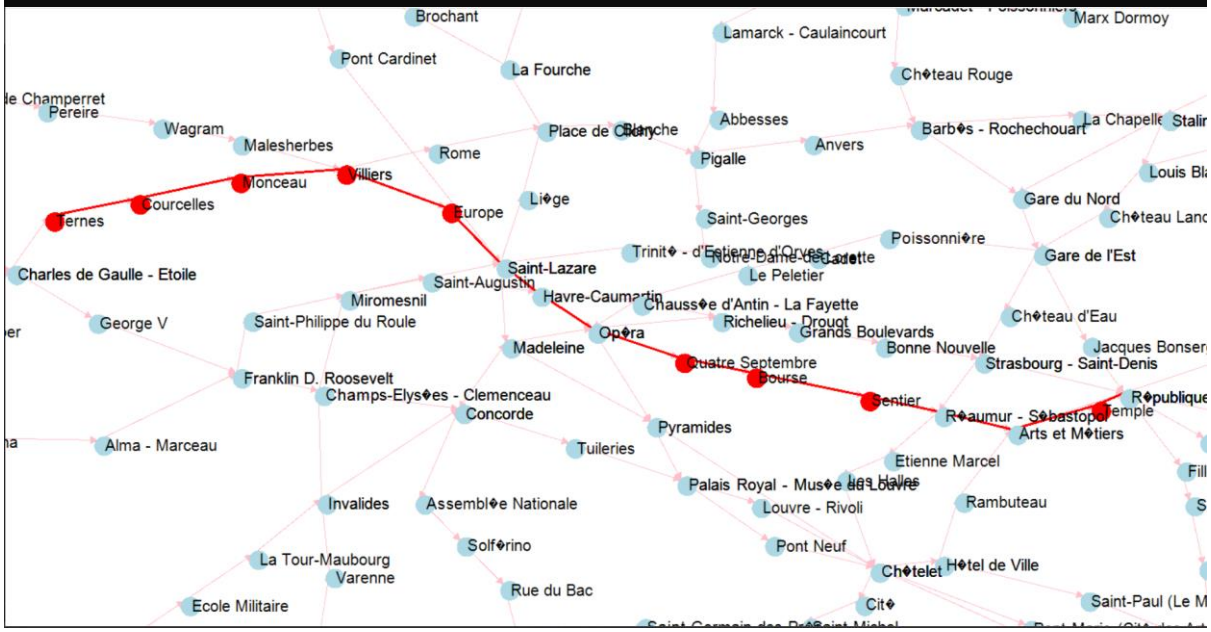
```
int n = Stations.Count;
double[,] distances = new double[n, n];

for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        if (i == j)
        {
            distances[i, j] = 0;
        }
        else
        {
            distances[i, j] = double.MaxValue;
        }
    }
}

foreach (var connexion in Connexions)
{
    int fromIndex = Stations.FindIndex(s => s.Id == connexion.From.Id);
    int toIndex = Stations.FindIndex(s => s.Id == connexion.To.Id);

    distances[fromIndex, toIndex] = 1;
    distances[toIndex, fromIndex] = 1;
}
for (int k = 0; k < n; k++)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (distances[i, j] > distances[i, k] + distances[k, j])
            {
                distances[i, j] = distances[i, k] + distances[k, j];
            }
        }
    }
}
```

D’après nos analyses des différents algorithmes et après les avoir testés, nous avons décidés de garder L’algorithme de Dijkstra car il est plus simple et plus efficace sur des grands graphes comme le métro parisien, de plus les poids des liaisons sont tous positifs. On utilise donc Dijkstra .

Voici le chemin le plus court entre Ternes et République avec une correspondance entre la ligne 2 et la ligne 3 à Villiers.



```
=== SYSTÈME DE NAVIGATION MÉTRO ===
Stations: 332, Connexions: 314

1. Afficher carte
2. Chercher itinéraire
3. Quitter
Choix: 2
ID départ: 23
ID arrivée: 60
Itinéraire (16 stations):
23. Ternes
24. Courcelles
25. Monceau
26. Villiers
49. Villiers
50. Europe
51. Saint-Lazare
52. Havre-Caumartin
53. Op?ra
54. Quatre Septembre
55. Bourse
56. Sentier
57. R?aumur - S?bastopol
58. Arts et M?tiers
59. Temple
60. R?publique
```



Etapes suivantes pour le livrable 3 :

- Relier la partie graphe et la partie BDD en affichant le graphe et le chemin le plus court lorsque le cuisinier sélectionne une commande faite et à envoyer l'application lui affiche le chemin le plus court de chez lui à la station du client.