Présentation de l’algorithme T-AIA-901:

L’objectif de la partie pathfinding du projet T-AIA-901 est, en recevant un ID de phrase, une gare ou ville de départ et de destination, de renvoyer un trajet optimal en train ou bus voire les deux pour aller du départ à l’arrivée.

Explications des différents algorithmes:

Il existe différents algorithmes pouvant être utilisés pour ce genre de travaux, certains étant plus adaptés que d’autres en fonction de leurs caractéristiques et du type de trajets que l’on souhaite obtenir.

Les plus connus sont :

- Breadth First Search (BFS)
- Depth First Search (DFS)
- Dijkstra
- A*

Le BFS correspond à une sorte de vague commençant du point de départ analysant toutes les nodes de plus en plus éloignés, jusqu’à atteindre la node d’arrivée et retracer le chemin. C’est l’un des plus coûteux en temps et ressources car explorant toutes les nodes dans toutes les directions sans distinction.

Le DFS choisit une direction ou un chemin de manière arbitraire et explore le plus possible dans cette direction jusqu’à atteindre une impasse où elle la prochaine choisie sera alors la node la plus éloignée du départ parmi celles explorées et ayant d’autres nodes voisines à explorer.
Il est moins coûteux que le BFS mais n’est pas adapté pour les zones ouvertes et reste tout potentiellement coûteux car pouvant également explorer toute la carte si la direction choisie arbitrairement n’est pas bonne.

Le Dijkstra est un algorithme est un plus élaboré que les précédents sélectionnant les nodes avec la distance parcourue la plus faible pour arriver à sa node actuelle, continuant ainsi jusqu’à ne plus avoir de node à explorer ou avoir atteint le point d’arrivée. Cela permet d’avoir de manière sûre le chemin le plus court pour aller d’un point A à B.
Il est plus intéressant que les deux précédents car utilisant un système de sélection plus sophistiqué explorant des nodes ayant davantage de chance d’être intéressantes pour la résolution du chemin.

Le A* est une version évoluée du Dijkstra, où l’on garde la majeure partie du système de distance minimum pour la sélection, mais où l’on ajoute à cette distance le résultat d’une fonction heuristique permettant de moduler avec, en théorie plus de précision la node choisie, notamment par l’ajout de poids sur certaines nodes, en rendant certains beaucoup plus ou beaucoup moins intéressantes en fonction du résultat voulu.
Le poid d’une node devient alors:
Coût = Distance de départ + Distance d’arrivée + Fonction heuristique

C’est l’algorithme le plus intéressant de la liste car combinant les avantages du Dijkstra en termes de ressources et temps utilisé, en plus d’avoir davantage de flexibilité par le système d’heuristique tout en ayant encore plus de chance d’avoir un chemin optimal rapidement.

Deux notes supplémentaire sur le A*:
L’heuristique est ce qui rend l’algorithme “admissible”, c'est-à-dire qu’il trouvera toujours le chemin le plus court.
Un algorithme A* sans coût ou heuristique est un simple Dijkstra , qui est lui-même un algorithme admissible, mais plus lent la majorité du temps.

C’est pour les raisons ci-dessus que l’algorithme retenu pour ce projet est le A*.

Utilisation et déroulement du programme:

Le programme est écrit en Python, la base de données en PostgreSQL et le gestionnaire de base de données utilisé est Adminer, le tout orchestré via Docker et Docker compose, bien que les fichiers python puissent être utilisés sans passer par une image.

Création de la base de donnée:
Le programme utilise les fichiers donnés leur de la présentation du projet, c’est à dire:

- agency.txt
- calendar_dates.txt
- calendar.txt
- routes.txt
- stops_times.txt
- stops.txt
- transferts.txt
- trips.txt
- timetables.csv

Pour chacun de ces fichiers une table est créée puis remplie avec les données contenues dedans afin de pouvoir les réutiliser par la suite.

Présentation des classes:
Les 5 classes définies sont les classes Map, TrainStation, Node et STATIONS_DICTS:

La classe Map est le point principal du programme. Il s’agit d’un Singleton possèdant de nombreuses fonction Factory qui s’occupe de la gestion des TrainStations, l’attribution des trips et du déroulement de A*.

Pour des soucis de ressources, elle ne charge pas toute les gares et trips, mais seulement celles nécessaire pour les algorithmes, qu’elle garde en mémoire pour les futures utilisations pour gagner du temps.

La classe TrainStation correspond simplement aux différentes gares, que ça soit de Train ou de Bus, contient les informations relatives à celle-ci

La classe Trip représente les liens entre les TrainStations, les reliant entre elles via un trajet à sens unique et servant de chemins pour l'algorithme pour passer de TrainStation en TrainStation pour essayer d'atteindre l'objectif.

Les Nodes sont utilisée par le A* pour contenir les informations tel que les coûts, l'ancêtre et la génération d'une Station, ainsi que le Trip qui a permis de l'atteindre.

STATIONS_DICTS est une énumération contenant les villes dites spéciales, c'est-à dire les très grandes villes possèdant plusieurs gares et donc généralement des transports en commun ou à pied possibles facilement. Les villes dans cette liste ont des Trips supplémentaires récupérés par l'API Google pour se déplacer via Métro, Tramway, Bus non répertoriés ou bien à pied.

Déroulement de l'algorithme:

La classe Map est initialisée sans TrainStation par défaut. Il est cependant possible d'initialiser toutes les gares avec tous leurs trajets via load_all_stations_and_trips.
Cela permet de ne plus avoir de d'appel à la base de données à faire, en échange d'une attente initiale longue malgré le système de multi threading et d'une consommation de ressource élevée.

On passe à la fonction load_path une liste d'objets possédant une gare valide en "departure" et en "destination", ainsi qu'une sentenceID. On passe également la date et le jour de départ.

Une gare est considérée comme valide si l'on passe le nom de la gare mot pour mot, si le nom de la ville où se trouve la gare, si elle en partage le nom ou bien si l'on passe un alias correct.
Exemple: "Gare de Lille Flandres", "calais ville", "la capitale", "paris"
A noter que l'algorithme essaye de corriger le nom de la ville s'il y a une faute dedans mais que le string reçu est suffisamment proche de celui de la ville ("Strasbourg" en "Starsbourt")

Si une ville à plusieurs gares, l'algorithme essayera de prendre la gare la plus proche géographiquement parlant de la gare à laquelle est liée, bien que prône à l'erreur car les gare n'étant pas toujours les plus proches (Lille => Paris qui sélectionne Lille Europe => Paris Gare de l'Est).

Les TrainStations non chargées dans la Map sont alors récupérées en base de données ainsi que leurs Trips.
Si invalides alors on arrête la fonction, sinon on passe à l'algorithme.

Pour toutes les étapes passées dans la liste d'objet, on calcule la distance de la node initiale de la node finale ainsi que sa distance heuristique.
Dans cette implémentation de A*, la "distance" utilisée est le temps nécessaire pour aller d'une node à une autre. S'il n'y a pas de distance directe, on utilise une approximation calculée via la distance géographique.

On récupère la node a la valeur heuristique la plus basse, s'il s'agit de la node finale on arrête l'itération actuelle et construit le chemin via les ancêtre ayant menés à cette node.

Si la node n'a pas ses Trips chargés, alors on les récupère en base de données.
Les TrainStations liées par un Trip à la node actuelle sont initialisées mais sans leurs trips pour l'instant pour soucis de consommation de ressources inutiles.

Si la TrainStation est dans l'énumération expliquée plus tôt, on lui attribue des Trips Spéciaux via l'API Google.

On regarde ensuite la liste des nodes atteignable par la node actuelle, et on leur attribue les ancêtres, générations et différentes "distances".
La distance heuristique prend en compte le temps d'attente qu'il peut exister avant le départ du transport.
Si la node n'a pas déjà été visitée et n'est pas présente dans les futures nodes explorées, on l'ajoute dans les futures explorations.

On continue cette boucle jusqu'à ce qu'il n'y ai plus de nodes explorables ou que l'on atteigne la node finale.

On renvoie alors un objet avec les informations relatives au trajets, que l'on peut alors formaté selon les besoins.