CARNET DE BORD

suivi du projet du 03/04/18 au 17/05/18

03/04/18

Fait pendant la séance :

Prise en main du sujet :

- élaboration du diagramme d'utilisation ;
- élaboration d'un premier diagramme de classes sans regard sur la base de données (réflexion sur les hypothèses à poser ect) ;
- recherche de bases de données sur les sites de la RATP et stif ;
- déchiffrage des données ;
- élaboration d'un diagramme de classe tenant compte de la structure de la base de données.

A Faire : Relire les consignes. Lire la documentation des algorithmes présentés, préciser le diagramme de classes.

09/04/18

Fait pendant la séance :

- Diagramme de cas d'utilisation ok ;
- Diagramme de classe (partie réseau) ok ;
- Début diagramme d'activités ;
- Établissement des étapes principales de développement : création de la BDD(lecture de fichier , écriture BDD à partir de Java) ; création des classes ; instantiation des algorithmes de détermination d'itinéraires ;

Hypothèses mise en avant durant la séance :

- on travaille seulement sur le réseau ferré (rer+métro) ;
- l'utilisateur ne peut que choisir son heure de départ (maintenant ou plus tard) et non pas d'arriver avant telle heure ...

Remarques :

Le programme comprend 3 algorithmes différents :

- Le CSA pour l'itinéraire le plus rapide ;
- Le CSA modifié qui ne prend en compte que la table horaire des temps de marche lié aux corespondances ;
- Le naïf qui compte seulement le nombre de correspondance OU un dijskra avec comme poids 0 pour une connexion sur une même ligne et 1 pour les connexions par correspondance. ON A EU UNE TRES BONNE IDEE !

A Faire : Demain : se concentrer sur les bases de données !
Aelaïg : lecture des fichiers.
Fanny : création de la BDD avec Java.

Pour plus tard :

- créer la table horaire : créer une méthode ;
- implémenter Dijskra (fluide, possibilité de changer la pondération) ;
- écrire diagramme de séquence des différents algorithmes ;
- rédiger rapport d'Analyse ;

10/04/18

Fait pendant la séance :

- Création des bases de données par JAVA ;
- Première idée : remplir la base de donnée via JAVA avec lecture de fichier puis insertion dans la BDD créée.

Or cette méthode s'avère très laborieuse et longue (perte de temps), donc on a choisi de remplir la base de donnée vierge par importation des fichiers de données .txt convertis en .csv directemet depuis l'interface SQLiteman.

- Ainsi on a : 1) converti tous les fichiers utiles en .csv et supprimer les entêtes de chaque fichier (Données RATP pour BDD) ;
- puis on a importer un premier fichier dans notre base de donnée créée le matin.

A Faire :

Aelaïg :

- Commencer le rapport d'analyse et envoyer les fichiers à Fanny ;
- Reprendre dans le diagramme des classe avec Enum d'État en Travaux →booléen dans station/ligne.

Fanny :

- Créer le GitHub pour partage du fichier ;
- Remplir la base de donnée + supprimer colonnes non utilisées + création de colonne par JAVA.

Commencer algorithme : se répartir les tâches !

02/05/18

Fait pendant les vacances :

- Remplissage de la BDD (les colonnes ne sont pas supprimées, et il manque le métro 6-arrêts) et Connexion avec java
- Rédaction du rapport d'analyse (diagramme gantt ect)
- Création des classes réseau sur java

A Faire :

Fanny :

- Supprimer les colonnes de la BDD
- Finir le rapport d'analyse (diagramme activités et hypothèses)

Aelaïg :

- Commencer à réfléchir à l'implémentation de la table horaire
- question : devons nous garder les données sur la BDD et utiliser tout au long de l'algorithme de plus court chemin des requêtes SQL ou transférons nous directement toutes les données sur JAVA (très gros objets) ?

03/05/18

Fait :

- Recherches pour créer la table Horaire liée aux courses
- Problèmes rencontrés : erreur nom de colonne BDD table course entre id_route et id_course
- id_course doit être intégrer dans une variable Long
- Question : comment gérer les horaires / durées sur JAVA.

04/05/18

Fait pendant la séance :

Aelaïg :

- Résolution des problèmes rencontrés précédemment
- Questionnement concernant le temps de calcul : très long pour afficher tous les id_course…

- Problème : j'utilise un result.previous() pour créer ma table horaire car j'ai besoin d'avoir accès à deux lignes à chaque itération. Cependant erreur : `SQLite only supports TYPE_FORWARD_ONLY cursors` ; la BDD SQLiteman ne peut gérer un curseur en marche arrière. Donc nous avons chercher une solution alternative : garder la valeur précédente dans une sauvegarde. Cela fonctionne très bien !
- Documentation de toutes les tables créées

Fanny :

- Création d'une classe « Time » pour palier au soucis du format de date/horaire -> cette classe nous permet de faire nos calculs de durée dans le monde des nombres et pas dans le monde du temps (base 60)
- Documentation et commentaire de mes classes
- Finition des diagrammes en tout genre
- Début de l'algorithme CSA + documentation

A faire :

Fanny :

- finir le rapport

Aelaïg :

- créer getters setters pour Troncon

05/05/18

A faire :

- créer la table horaire des correspondances : vérifier que les correspondances de transfers réfèrent aux stations choisies puis que je créer la table en 1) sélectionnant toutes les stations ayant une correspondance, puis pour chacune sélectionner tous les arrêts à cette station et créer pour chaque arrêt autant de connexions qu'il y a de correspondances avec heure de départ + temps de marche = heure d'arrivée
- fusionner les 2 tables (des courses et des correspondances).
- Trier cette table selon les heures de départ des connexions = tronçons.
-

Résultats de test requête :
Nombre de stations dans notre réseau = 1315
Nombre de stations origine de correspondance = 4129
Nombre de stations destination de correspondance = 3883
Nombre de correspondances = 18338

Il faut donc garder seulement les correspondances entre 2 stations du réseau…

Nombre de correspondances entre 2 stations du réseau = 1548

INSERT INTO correspreseau SELECT * FROM correspondance WHERE id_station_origine IN (SELECT id_station FROM station) AND id_station_destination IN (SELECT id_station FROM station)

Fait :

- création d'une nouvelle table dans la BDD RATP « correspreseau » ne contenant que les correspondances entre stations du réseau (soit les stations répertoriées dans la table station)

Nombre de stations_origine = 262
Nombre de stations_destination = 256

Je remarque que des lignes de correspreseau sont en double
Après suppression des doublons en réalité 774 correspondances. →Création d'une nouvelle table au propre correspondance ne comprenant aucun doublon. L'ancienne table est conservée sont le nom toutescorresp, la table correspreseau est supprimée. Ainsi désormais, nous ne travaillons qu'avec la table CORRESPONDANCE.

- Problème : le temps de trajet entre 2 stations d'une correspondance est exprimé en secondes… alors que Time est en minute →SOLUTION : simplification, on exprime le temps de trajet en minute arrondi à l'entier le plus proche.

- Réalisation de la Table Horaire des Correspondances achevée
- Fusion des deux Tables Horaires
- Tri de la Table Horaire finale selon les heures de Départ des connexions (=Tronçon). Cette étape a nécessité la redéfinition de la méthode compareTo(Object o) dans les classes Tronçon ainsi que Time (car on veut comparer des horaires) afin de pouvoir directement trier la liste de Tronçons grâce à la méthode Collections.sort(TableHoraire).
- La Table Horaire est prête à être utilisée. Seule inquiétude : sa taille (pour le moment j'ai travaillé sur une Table réduite).

05/05/18

Fait :

- Rapport fini et validé
- Répartition des tâches : Aelaïg : CSA + enregistrement de la table horaire et Fanny : Dijkstra + document de synthèse

Remarque : Pour le calcul d'itinéraires, nous remarquons que nous n'avons besoin que d'une petite partie de la Table Horaire ainsi, nous décidons de la calculer et de l'enregistrer dans une table de BDD. Celle ci ne sera pas recalculée à chaque calcul d'itinéraire mais seulement lorsqu'un gestionnaire déclarera un travaux.
Créer une base de donnée pour sauvegarder la table horaire !

09/05/18

Fanny :

• code de Dijkstra :

▪ création d'une méthode de recherche du poid minimal dans le ArrayList de poids sur le principe de on cherche l'indice d'un poidMin initialisé à 0, si il n'y en a pas alors on l'incrémente pour chercher égal à 1 et ainsi de suite => ce n'est surement pas le plus efficace mais je savais pas trop comment faire autrement car il faut chercher une station avec un poids minimal mais ce n'est pas forcement l'unique

▪ création d'une méthode de calcul du poid d'un troncon qui prenant une station de départ et une station d'arrivée cherche s'il elles sont en correspondance (1), sur la même ligne et consécutives (0) ou pas en lien (-1) et leur attribut le poids cohérent. La correspondance est gérée, reste à gérer si même ligne et consécutive avec la table horaire sauvegardée quand elle existera => il manque juste à gérer le poids d'un tronçon sans correspondance

▪ implémentation de Dijkstra : normalement est terminé sous couvert de finir la méthode de calcul du poids du troncon. /!\ pas encore testé

• requête utilisateur :

début ▪ définition des attributs et création du constructeur par le biais d'un scanneur qui interroge directement l'utilisateur

• pb avec la gestion de l'heure courante : est-ce que l'utilisateur rentre l'heure meme quand il veut partir maintenant ?

▪ critère sous la forme d'un int pour simplifier après (système de condition en fonction de la valeur de critère pour lancer l'algo adéquat

▪ création de la méthode itineraire() qui lance la création d'une requete utilisateur puis fait appel à l'algo adéquate en fonction du critère choisi

---

10/05/18

• Tel avec Aelaig : (repartition taches)

▪ Tous les algorithmes retournent un ArrayList<Integer> et on le traite après avec une méthode qui l'affiche proprement à l'utilisateur

▪ Fanny :

- Continue sur requete utilisateur
- Cherche moyen d'afficher l'itinéraire à l'utilisateur
- Demain lance la sauvegarde table horaire depuis l'école

▪ Aelaig :

- Rajoute dans tronçon l'info sur correspondance ou non
- Continue CSA
- Ajout d'une colonne « ligne » qui permet d'associer un id_station au nom de la ligne à laquelle la station appartient -> ce code est présent dans la classe LienStationBdd du sous-package bdd.autre
- Creation de afficherItineraire(itineraire) dans Requete qui permet de transformer un ArrayList en Chaine de caractère

▪ Pour l'instant c'est sous la forme « Station : « nom_station » ligne « nom_ligne » » donc qui liste TOUTES les stations présentes dans l'itinéraire avec leur nom et leur numéro de ligne ▪ Si on arrive à avoir la table des courses, faudra ajouter la possibilité de n'afficher que les stations de début de prise de la ligne et les stations de correspondances

---

12/05/18

Fait : Aelaïg :

- La Table Horaire est beaucoup trop volumineuse donc je tente une autre approche permettant de la limiter : en classant les arrêts dans la table BDD selon l'heure de départ, nous pouvons nous restreindre à l'évaluation des arrêts concernant une plage d'horaires prédéfinie : ainsi si la requête utilisateur impose comme heure de départ 6h30 alors on considère tous les arrêts entre 6h30 et 7h30 ou 8h30.
- Modification de reseau.Arret
- Après mise en place de cette nouvelle stratégie, le calcul est encore trop long… ce qui prend du temps c'est la multitude d'interactions entre le script java et la BDD. Ainsi j'ai tenté de concentré le gros du traitement dans le programme JAVA en instanciant ma liste ArretsTempo (= liste des arrêts dans la fourchette d'horaires prédéfinie) directement dans JAVA et en redéfinissant tout le calcul de la table horaire sur JAVA. Le calcul est certe plus rapide or une erreur s'affiche : `Exception in thread "main" java.lang.OutOfMemoryError: Java heap space.` En effet on remarque que la BDD arret compte 3000 arrets à exactement 14:00:00 ainsi 3000*60 (1heure) = 180 000 itérations. Le nombre de connexions est beaucoup trop élevé.

13/05/18

A Faire :

- mettre au clair le code, documenter
- obtenir une bonne fois pour toute une table horaire exploitable
- finir les algorithmes CSA et CSA modifié
- finir requête utilisateur.

Fait :

- en reprenant le code j'arrive à obtenir rapidement une table horaire restreinte à 1 ou 2 heures.
- La sauvegarde en BDD est l'étape qui prend le plus de temps, au final est n'est pas indispensable donc finalement nous ne ferons pas de sauvegarde sur BDD.
- Clarification du code + documentation
- création d'une fonction dans Troncon : rechercher Troncon dans une liste de Troncon grâce à ses identifiants de stations.

Problème quand on désigne une station de départ et d'arrivée = plusieurs stations/quais donc plusieurs listStations

Algo Dijkstra

`Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 1628 out-of-bounds for length 1314 ligne 169`

- modifications Dijkstra (pour moi TronconDijkstra est inutile → modification de Troncon)
- l'erreur a retrouvé dans l'algorithme
- modification de l'algorithme CSA : listerStation via ArretsTempo ; mise à jour mais encore quelques problèmes de compréhension
- modification de requête ; résolution du problème de l'heure actuelle

A Faire :

- finir les algorithmes : nous y sommes presque !
- gérer les stations en travaux
- gérer le fait que le départ et l'arrivée puissent se faire sur plusieurs stations/quais.

14/05/18

A Faire :

- gérer les erreurs des algorithmes : nous y sommes presque !
- Obtenir des itinéraires propres !
- gérer les stations en travaux
- gérer le lien entre ligne et station OK
- gérer le fait que le départ et l'arrivée puissent se faire sur plusieurs stations/quais.

Fait :

- mise en commun des modifications
- création de la colonne dans la BDD permettant le lien entre la station et la ligne
- reprise du code de l'Algorithme de Dijkstra
- le calcul du trajet au minimum de correspondances.
- reprise du code de l'Algorithme du CSA

- gérer dans listStation supprimer les stations de la ligne 6

problème plein de stations don la ligne est null (donc 6) il faut supprimer toutes les stations dont la ligne est null et tous les arrets desservant ces stations !!!

Remarque : on découvre qu'une station est réellement un quai : 1 station

à refaire redéfinir la méthode equals pour les classe où on utilise contains.

- MERCREDI : mettre au propre le code, javadoc , faire le document de synthèse, réfléchir à la présentation.

15/05/18

Fait :

- reprise de l'algorithme du CSA - fonctionne très bien
- reprise de l'algorithme Dijstra - fonctionne très bien
-

A Faire :

- Aelaïg ce soir :
  - algorithme CSA modifié pour correspondances
- documentation de tout le code et restructuration si besoin.

- Demain 16/05/18 :
- rédiger le document de Synthèse
- gérer l'affichage final → gérer problème de plusieurs identifiants pour 1 seule station (idée : l'utilisateur rentre le nom , et on affiche les différents identifiants possibles)

- gérer les stations en travaux. + recréer colonne en travaux. (Aelaïg) + mise ligne en travaux et enlever ligne en travaux.

- Prendre du temps pour mettre au propre le code et rédiger un document de synthèse parfait + read me.
- Refaire le diagramme de GANTT.
- Restructurer le code.

16/05/18

Fait :

- continuer le document de Synthèse
- gérer l'affichage final → gérer problème de plusieurs identifiants pour 1 seule station (idée : l'utilisateur rentre le nom , et on affiche les différents identifiants possibles) OK

- gérer les stations en travaux. + recréer colonne en travaux. (Aelaïg) + mise ligne en travaux et enlever ligne en travaux. OK

- Refaire le diagramme de GANTT. OK
- Restructurer le code. OK

- Créer table arrettries + enTravaux - fait dans la soirée
- gérer enTravaux et gérer méthode nom de station = identifiants a choisir - fait dans la soirée

---

17/05/18

A Faire :

- Gérer les dernières erreurs
- Finir de rédiger les rendus.