**Analyse**

***Trouver la ou les routes qui rapportent le plus de points entre des points données à parcourir dans une grille 2D.***

***Licence RGI - Groupe ERP CISCO :***

**Maël RHUIN**

**Sommaire**

**Approche**

***Sujet*** : Trouver la ou les routes qui rapportent le plus de points entre des points données à parcourir dans une grille 2D.

***Réflexion autour de l’algorithme de Dijkstra :***

L’[algorithme de Dijkstra](https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra) est une méthode qui permet de trouver le plus court chemin entre deux sommets d’un graphe pondéré par des réels positifs. Il a été inventé par le mathématicien et informaticien néerlandais [Edsger Dijkstra](https://fr.wikipedia.org/wiki/Edsger_Dijkstra) en 1956. L’idée principale de l’[algorithme](https://www.maths-cours.fr/methode/algorithme-de-dijkstra-etape-par-etape) est de maintenir un ensemble de sommets dont les distances minimales à la source sont connues, et d’ajouter progressivement le sommet le plus proche de la source à cet ensemble. Ci-après l’algorithme de Dijkstra en pseudo code :

// Entrée : un graphe G = (V, E) pondéré par des réels positifs et un sommet source s

// Sortie : un tableau dist qui contient les distances minimales de s à tous les autres sommets

// Initialisation

Pour chaque sommet v de V

dist[v] = +infini // Distance infinie à l'origine

visité[v] = faux // Sommet non visité

Fin pour

dist[s] = 0 // Distance nulle à la source

// Boucle principale

Tant qu'il existe un sommet non visité

u = le sommet non visité ayant la plus petite distance // Choix glouton

visité[u] = vrai // Marquer u comme visité

Pour chaque voisin v de u

si dist[u] + poids(u, v) < dist[v] // Relâchement des arêtes

dist[v] = dist[u] + poids(u, v) // Mise à jour de la distance de v

Fin si

Fin pour

Fin tant que

Retourner dist

**Cahier des charges**

La réalisation de cette analyse donnant lieu à un programme informatique est conditionnée par un cahier des charges :

Sur un plateau donné de largeur 20 et longueur 20 disponible dans le fichier Excel joint.

* Case négative (**noire**) : infranchissable
* Case positive : possible de se déplacer avec le coût indiqué
* Case stratégiques (**rouge**) :
* Case d'intérêts (**vert**) :

Construire un ***dataset*** contenant les routes (les plus courtes) et le coût de déplacement (sommes des cases traversées) entres :

* tous les points stratégiques de la carte
* tous les points d'intérêts de la carte
* tous les points stratégiques et les points d'intérêts

La personne arrive en coordonnées {x ; y}, x = 11 et y = 19, et doit se rendre aux points stratégiques 1, 3, 6 & 7 :

* Chaque point stratégique lui rapporte 30 points.
* Chaque point d’intérêt lui rapporte sa valeur donnée.

Déplacements autorisés : horizontaux et verticaux (pas de diagonales)

Calculer le ou les chemins pour que le personnage se rende aux lieux stratégiques indiqués et récolte au passage le maximum de points en passant par des lieux d'intérêts

Calcul des points

Points des lieux stratégiques + points des lieux d'intérêts traversés - poids de chaque déplacement

L'objectif est de trouver la ou les routes donnant le plus de point !

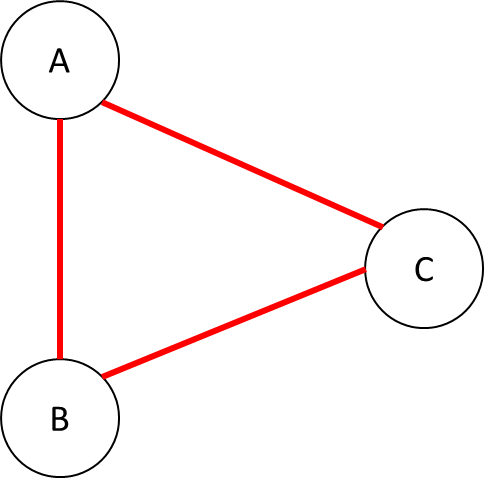
**Analyse de Niveau 0**

***Sujet*** : Trouver la ou les routes qui rapportent le plus de points entre des points données à parcourir dans une grille 2D.

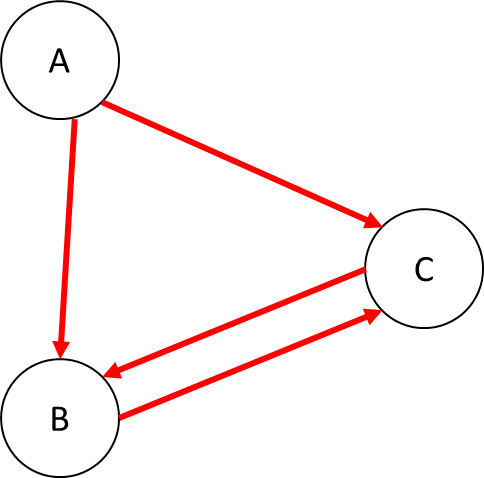
Après avoir trouvé une approche à la résolution de notre sujet. Il nous faut l’appliquer. Pour résumé l’algorithme de Dijkstra requiert un graphe orienté et pondéré (chaque arc possède un poids qui doit être ≥ 0.

***Qu'est-ce qu'un graphe ?***

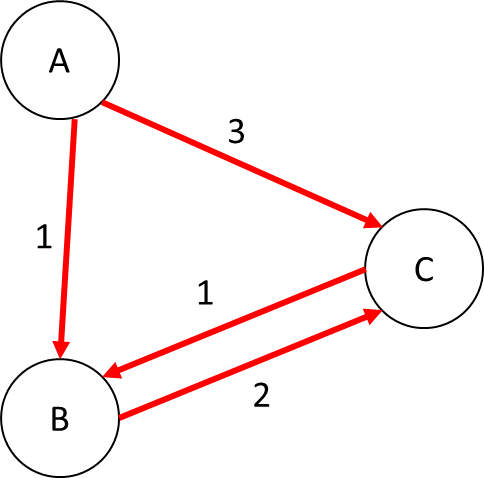
Un [graphe](https://fr.wikipedia.org/wiki/Graphe_(math%C3%A9matiques_discr%C3%A8tes)) en mathématiques est une structure composée d’objets appelés sommets et de relations entre eux appelées arêtes (ou arcs).

À gauche un graphe composé de trois sommets A, B, C relié par des arcs en rouge.

Un graphe ***orienté*** est un graphe où les arcs ont une direction représentée par une flèche.

 À gauche un graphe composé de trois sommets A, B, C relié par des arcs orientés en rouge.

Un graphe ***pondéré*** est un graphe où chaque arc porte un nombre appelé **poids**. Les poids peuvent représenter des coûts, des longueurs ou des capacités selon le problème.

 À gauche un graphe composé de trois sommets A, B, C relié par des arcs orientés et pondérés en rouge.

Il faut donc d’abord transformer notre matrice d’entiers en un graphe orienté et pondéré pour pouvoir lui appliquer Dijkstra.

***Comment faire ?***

Il nous faut découper le problème en plusieurs outils de résolution. Pour former notre graphe il nous faut des sommets, des arcs et des poids. Pour cela on décompose en objets :

Notre ***carte*** possède une retraduction en un graphe :

Un ***graphe*** est composé de sommets :

Un ***sommet*** possède un nom unique et des arcs orientés :

Un ***arc*** possède un sens (d’un sommet A vers B) et un poids positif.

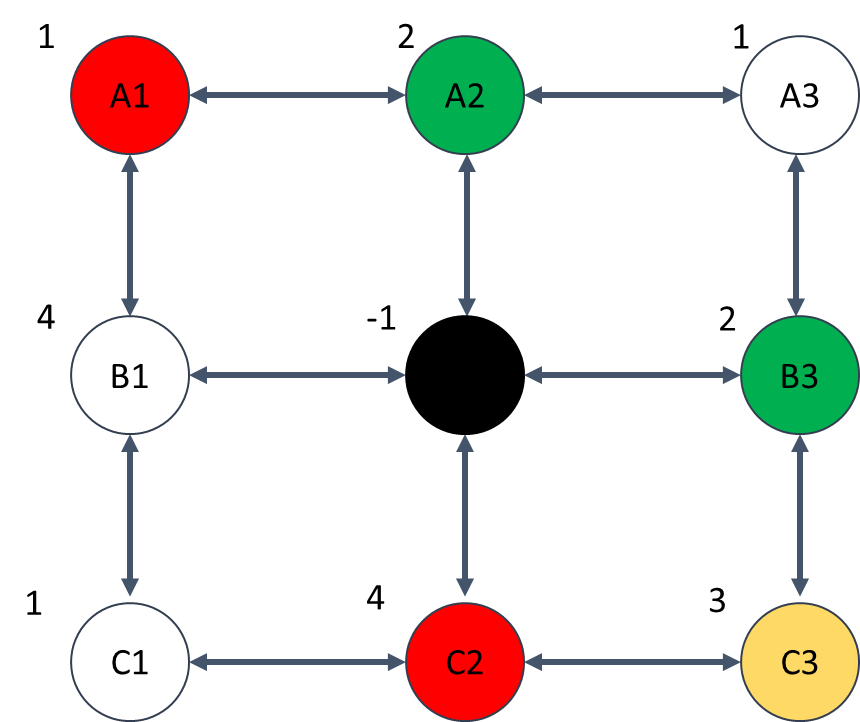
Soit une matrice d’entiers compris entre un minimum ***a*** et un maximum ***b*** de taille donnée par deux entiers ***l*** (largeur)et ***h*** (hauteur) :

***Exemple :***

***a = -1 et b = 4 ; l = 3 et h = 3***

On affecte à chaque ***case*** de cette matrice un nouveau ***sommet*** qui peut être un obstacle ou non, de manière à obtenir une liste de sommets :

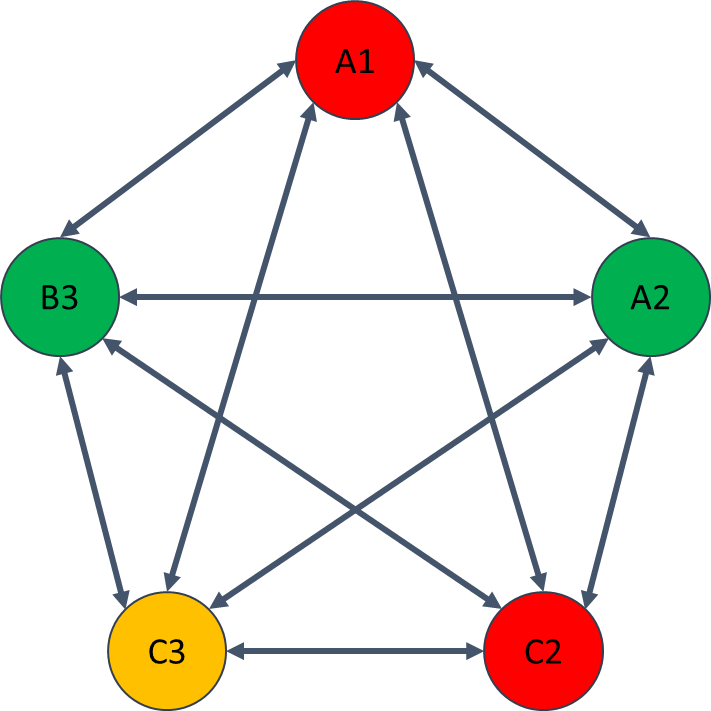
Pour chaque ***voisin*** (haut, bas, gauche, droite) de chaque ***sommet***, on crée un ***arc*** de ***poids*** correspondant à la ***case*** de notre matrice pour obtenir le graphe correspondant :



Sur ce graphe on y a placé pour exemple les points « types » du cahier des charges :

* En noir l’obstacle
* En rouge les points stratégiques
* En vert les points d’intérêts
* En jaune le point de départ

Comme Dijkstra requiert des arcs de poids ≥ 0, dans le cas des obstacles nous déclarons le sommet comme étant un obstacle et son poids entrant est définit à un nombre très grand proche de « l’infini ».

**** La suite de notre raisonnement va être de « minimisé ce graphe » en un plus petit ne contenant que les points qui nous intéresse. Chaque arc sera cette fois-ci de poids correspondant au score calculé du chemin le plus court entre 2 sommets. Chaque sommet de ce graphe est connecté à un autre. On appelle ce type de graphe : « graphe complet » .

On peut ensuite à partir de ce graphe sélectionner en le parcourant les arcs qui maximise notre score tant qu’il nous reste des sommets « obligatoires » à traverser. Pour cela on utilise un algorithme, présenté à la page suivante.

// Entrée : un graphe\_minimise G et un sommet de départ

// Sortie : une liste de sommets représentant le chemin parcouru

// Initialisation

Créer une liste vide chemin

sommet\_en\_cours = sommet de départ

Ajouter sommet\_en\_cours à chemin

Créer une liste sommets\_a\_parcourir contenant tous les sommets obligatoires

// Boucle principale

Tant que sommets\_a\_parcourir n'est pas vide

Créer un arc\_poids\_max non null bouclé sur lui même

Pour chaque arc dans les arcs de sommet\_en\_cours

Si l'arc a un poids supérieur à arc\_poids\_max et que l'arrivée de l'arc n'est pas déjà dans le chemin

Mettre à jour arc\_poids\_max avec l'arc

Fin Si

Fin pour

Si arc\_poids\_max a un poids >= 0

mettre à jour sommet\_en\_cours avec l'arrivée de arc\_poids\_max

ajouter sommet\_en\_cours à chemin

Si sommet\_en\_cours est dans sommets\_a\_parcourir

supprimer sommet\_en\_cours de sommets\_a\_parcourir

Fin Si

Sinon

Si sommet\_en\_cours est dans sommets\_a\_parcourir

Mettre à jour sommet\_en\_cours avec l'arrivée de arc\_poids\_max

Ajouter sommet\_en\_cours à chemin

Supprimer sommet\_en\_cours de sommets\_a\_parcourir

Pour chaque sommet\_restant dans sommets\_a\_parcourir

Ajouter sommet\_restant à chemin

Supprimer sommet\_restant de sommets\_a\_parcourir

Fin Pour

Sinon

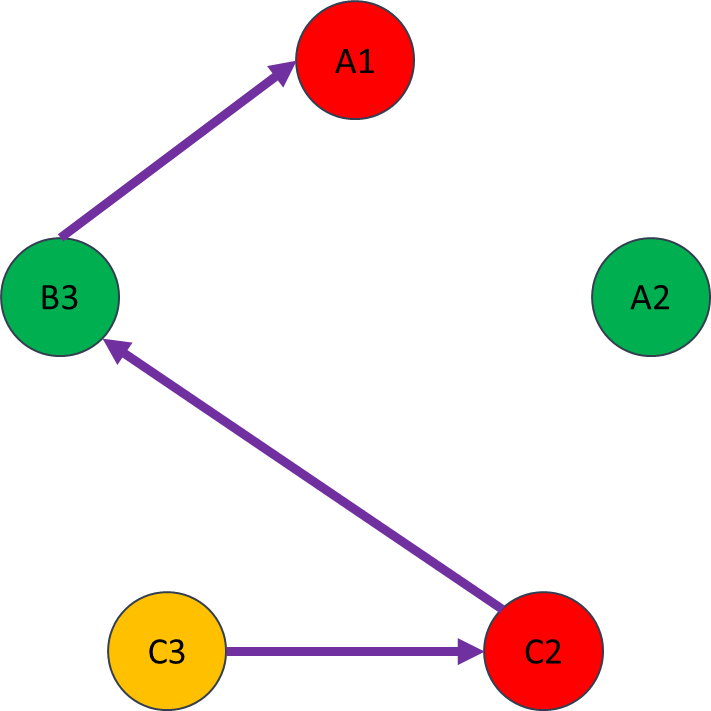
Sortir de la boucle principale

Fin si

Fin si

Fin Tant que

Retourner le chemin



Le chemin obtenu par cette algorithme dans notre graphe complet est maintenant à « développer » pour pouvoir être superposé avec notre graphe initial. Il nous faut alors une fonction qui va rechercher une nouvelle fois le chemin le plus court entre chacun des sommets successif de notre chemin pour obtenir le chemin global final.

// Entrée : une liste de sommets représentant un chemin dans un graphe pondéré et connexe

// Sortie : une liste de sommets représentant un chemin développé par la recherche du plus court chemin entre chaque paire de sommets consécutifs

// Initialisation

Créer une liste vide chemin\_developpe

Créer une liste vide chemin\_tmp

// Ajout du premier sommet du chemin dans la liste développée

Ajouter le premier sommet de la liste chemin dans la liste chemin\_developpe

// Boucle principale

Pour chaque paire de sommets consécutifs sommet\_A et sommet\_B du chemin

Appliquer l'algorithme de Dijkstra entre sommet\_A et sommet\_B pour obtenir le plus court chemin chemin\_tmp

Supprimer le premier sommet de chemin\_tmp car il est déjà présent dans la liste chemin\_developpe

Ajouter tous les sommets de chemin\_tmp dans la liste chemin\_developpe

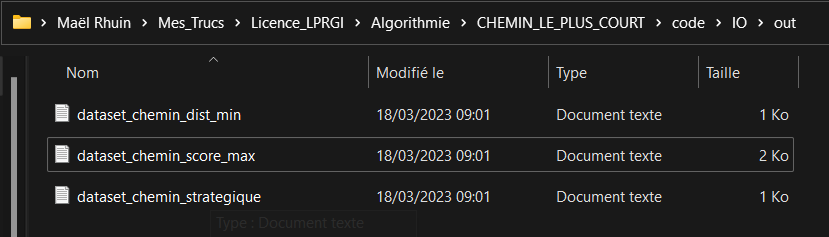
Fin pour

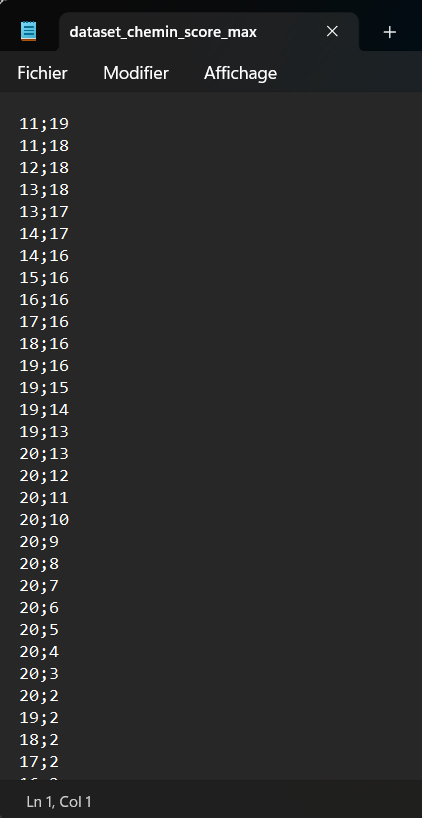
// Ajout du dernier sommet du chemin dans la liste développée

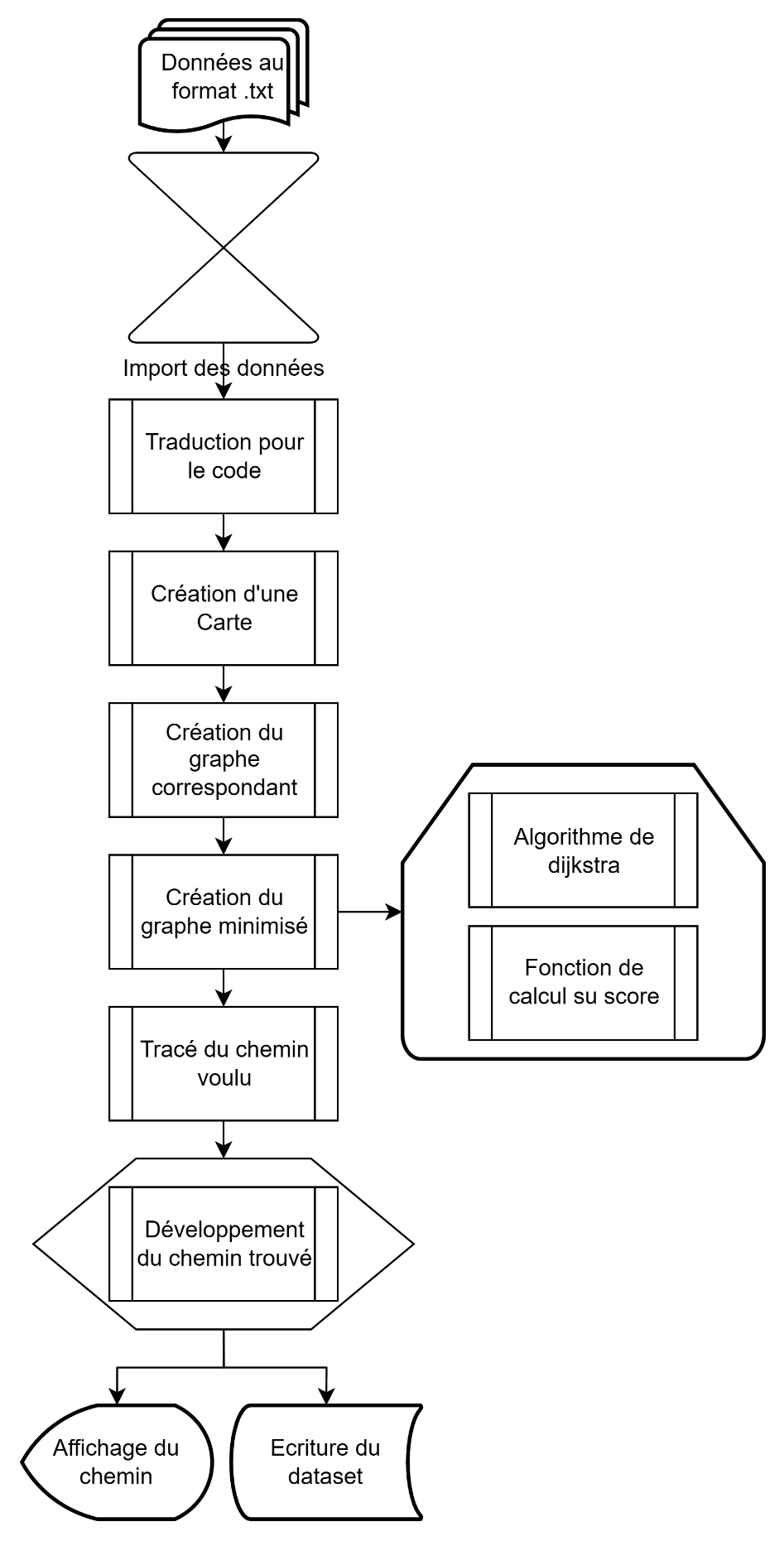
ajouter le dernier sommet de la liste chemin dans la liste chemin\_developpe

Retourner le chemin\_developpe

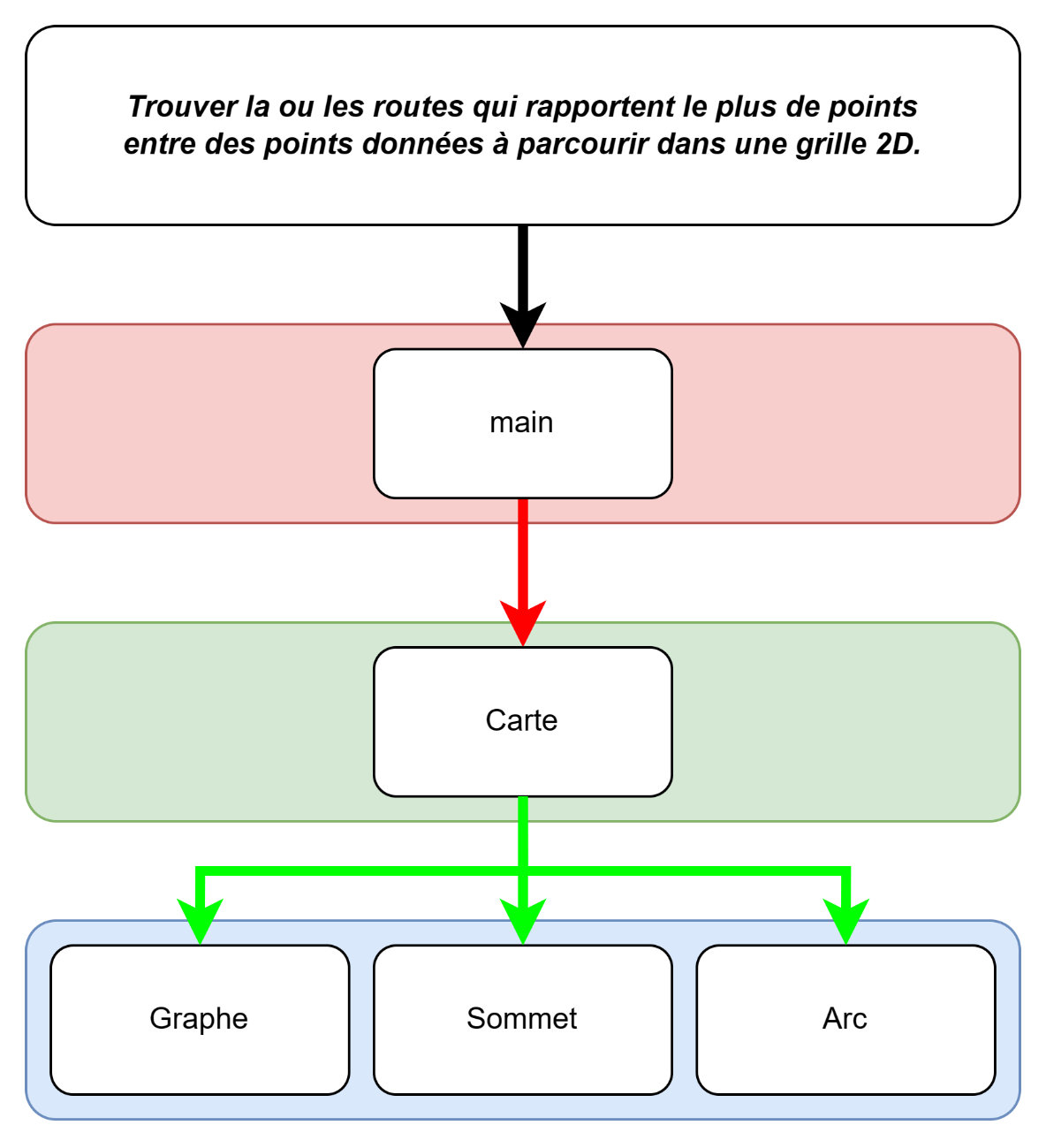
On peut enfin extraire sous forme d’un ***dataset*** le chemin final obtenu. De façon à avoir les coordonnées ***{ colonne ; ligne }*** de chaque sommet traversé.

****

****

**Logigramme fonctionnel**

**Arbre hiérarchique**



**Analyse de niveau 1**

|  |  |
| --- | --- |
| FP0  main |  |
| **Valeur Ajoutée**  Classe principale du programme. Elle répond directement à la problématique via ses composantes. | |
| **INPUT**  depart.txt, obligatoire.txt, matrice.txt, matrice\_interet.txt, matrie\_strategique.txt | **OUTPUT**  dataset\_score\_max.txt |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Début du programme  Définir la variable "repertoire\_courant" comme le répertoire courant  Afficher "[Création de la carte]"  Importer les données à travailler à partir du répertoire d'entrée et stocker la carte dans la variable "c"  Afficher le graphe de la carte  Afficher "[Terminée]"  Initialiser une liste "chemin\_score\_max" avec le chemin de score maximal de la carte  Définir la variable "repertoire\_dataset" comme le répertoire de sortie  Afficher "[Chemin obtenu de score max]"  Afficher le chemin de score maximal  Afficher "[Fin]"  Créer un fichier de données contenant le chemin de score maximal dans le répertoire de sortie  Fin du programme | |

**Analyse de niveau 2**

|  |  |
| --- | --- |
| FS0.1  Carte |  |
| **Valeur Ajoutée**  Cette classe est une représentation objet de notre problème. | |
| **INPUT**  …………… | **OUTPUT**  …………… |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Début  Attributs de la classe :   * + depart: un tableau d'entiers représentant les coordonnées du point de départ   + cases\_obligatoires: un tableau d'entiers représentant l'indice des cases obligatoires parmi les cases stratégiques   + grille: une matrice d'entiers représentant la grille de la carte   + cases\_interets: une matrice d'entiers représentant les coordonnées des cases d'intérêts et de leurs valeurs   + cases\_strategiques: une matrice d'entiers représentant les coordonnées des cases stratégiques   + graphe: un objet Graphe correspondant à la carte   Constructeur de la classe :  Carte(depart: tableau d'entiers, cases\_obligatoires: tableau d'entiers, grille: matrice d'entiers, cases\_interets: matrice d'entiers, cases\_strategiques: matrice d'entiers)   * + Associer chaque paramètre à l’attribut correspondant.   Fin | |

|  |  |
| --- | --- |
| FS0.2  afficherChemin |  |
| **Valeur Ajoutée**  Cette méthode permet l’affichage compréhensible d’une liste de sommets. | |
| **INPUT**  Liste de sommets | **OUTPUT**  …………… |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Procédure afficherChemin (chemin : liste de sommets)  Début  Pour i allant de 0 à Taille du chemin – 1 Faire  Afficher le nom du sommet à l’indice i dans le chemin  Afficher « -> »  Fin pour  Afficher le nom du dernier sommet du chemin  Fin | |

|  |  |
| --- | --- |
| FS0.3  traduirePourLeCode |  |
| **Valeur Ajoutée**  Ce code modifie les indices des tableaux en soustrayant 1 à chaque valeur. | |
| **INPUT**  matrice\_interets, matrice\_strategies, obligatoire, depart | **OUTPUT**  …………… |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Procédure traduirePourLeCode (matrice\_interets : matrice d’entiers, matrice\_strategies : matrice d’entiers, obligatoire : vecteur d’entiers, depart : vecteur d’entiers)  Début  Pour chaque entier de chaque paramètre  On soustrait 1 à la valeur de l’entier  Fin pour  Fin | |

|  |  |
| --- | --- |
| FS0.4  importMatrice |  |
| **Valeur Ajoutée**  Cette méthode retourne une matrice d'entiers construite à partir du contenu d’un fichier txt. | |
| **INPUT**  separateur, chemin\_fichier | **OUTPUT**  Matrice d’entiers |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Fonction importMatrice(separateur : chaine de caractère, chemin\_fichier : chaine de caractère)  Début  Pour chaque ligne du fichier  On découpe cette ligne selon le séparateur en un vecteur d’entiers  On attribut les valeurs de ce vecteur à la matrice qui sera renvoyée  Fin pour  On renvoie la matrice obtenue  Fin | |

|  |  |
| --- | --- |
| FS0.5  importerDonneesATravailler |  |
| **Valeur Ajoutée**  Cette méthode retourne une instance de "Carte" construite à partir des données importées depuis les fichiers. | |
| **INPUT**  repertoire | **OUTPUT**  Carte |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Fonction importerDonneesATravailler (rerpertoire : chaine de caractère)  Début  Pour chaque fichier du repertoire  On en extrait le vecteur ou la matrice d’entiers pour la création de notre carte  FinPour  On retourne une instance de Carte crée selon les données extraites.  Fin | |

|  |  |
| --- | --- |
| FS0.6  dataset |  |
| **Valeur Ajoutée**  Cette méthode permet de créer un dataset à partir d'un chemin donné et de l'écrire dans un fichier spécifié. | |
| **INPUT**  chemin, chemin\_fichier | **OUTPUT**  --------- |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Fonction dataset(chemin : liste de sommets, chemin\_fichier : chaine de caractère)  Début  Pour chaque sommet du chemin  On récupère les coordonées x + 1 et y + 1 du sommet  On les inscrit dans le fichier au chemin\_fichier  Fin pour  Fin | |

**Analyse de niveau 3**

|  |  |
| --- | --- |
| FT0.1.1  Graphe |  |
| **Valeur Ajoutée**  Cette classe est une représentation objet d’un graphe. | |
| **INPUT**  …………… | **OUTPUT**  …………… |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Début  Attributs de la classe :   * + sommets : liste des sommets du graphe   Constructeur de la classe :  Graphe(List<Sommet> sommets) : initialise la liste de sommets à partir de la liste donnée en paramètre  Graphe(int[][] matrice\_adjacences) :   * + Crée les sommets du graphe à partir de la matrice d'adjacences donnée en paramètre   + Pour chaque sommet créé :     - Récupère ses voisins dans la matrice d'adjacences     - Pour chaque voisin :       * Calcule le poids de l'arc entre le sommet courant et le voisin (en gérant les obstacles)       * Crée l'arc entre le sommet courant et le voisin avec le poids calculé       * Ajoute l'arc au sommet courant       * Met à jour la liste de sommets en remplaçant le sommet courant par la version mise à jour avec l'arc ajouté     - Fin pour   + Fin Pour   Fin | |

|  |  |
| --- | --- |
| FT0.1.2  Sommet |  |
| **Valeur Ajoutée**  Cette classe est une représentation objet d’un sommet. | |
| **INPUT**  …………… | **OUTPUT**  …………… |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Début  Attributs de la classe :   * + nom : chaîne de caractères   + x : entier coordonnée x   + y : entier coordonnée y   + predecesseurs : liste des sommets prédécesseurs   + successeurs : liste des sommets successeurs   + arcs : liste des arcs du sommet   Constructeur de la classe :  Sommet(nom: String, x: int, y: int, predecesseurs: List<Sommet>, successeurs: List<Sommet>, arcs: List<Arc>, isUnObtascle: boolean)   * + Associer chaque paramètre à l’attribut correspondant.   Fin | |

|  |  |
| --- | --- |
| FT0.1.3  Arc |  |
| **Valeur Ajoutée**  Cette classe est une représentation objet d’un arc. | |
| **INPUT**  …………… | **OUTPUT**  …………… |
| **Logigramme Fonctionnel** | |
| **Service Fonctionnel**  Début  Attributs de la classe :   * + Depart : Sommet de départ   + Arrivee : Sommet d’arrivée   + poids : entier poids de l’arc   Constructeur de la classe :  Arc(depart: Sommet, arrivee: Sommet, poids: int)   * + Associer chaque paramètre à l’attribut correspondant.   Fin | |

**Ressources**

Le pseudo code est découpé en plusieurs fichiers et placé dans le répertoire du projet ***docs***.

Le code complet est placé dans le dossier ***code***, il s’agit sur projet complet développé sur ***Intellij***.

Une ***Javadoc*** a été générée et placée dans le dossier ***javadoc***.

Les données d’entrées sont placés dans 5 fichiers txt dans le répertoire ***code/IO/in***.

Le ***dataset*** de sortie est à retrouver dans le répertoire ***code/IO/out***.

Le ***.jar*** du projet est à retrouver dans le répertoire ***code/out/artifacts/Algorithmie\_jar.*** Son exécution demande d’avoir un ***JavaRuntimeEnvironment*** supportant ***Java 16*** : il faut ensuite se placer dans ce répertoire et exécuter : ***java -jar .\Algorithmie.jar***

**Conclusion**

En conclusion, ce code présente des avantages certains. Tout d'abord, l'utilisation de l'algorithme de Dijkstra permet de trouver rapidement les chemins les plus courts pour former un chemin plus grand pour le ***dataset***.

De plus, la fonction de tracé du chemin final est facilement réglable selon que l'on souhaite aller au plus vite ou maximiser le score. Cependant, un inconvénient majeur est que le code ne peut pas détecter les situations où il n'y a pas de chemin possible si les obstacles bloquent complètement l'accès à un ou plusieurs points obligatoires.

Au cours de la rédaction de ce code, certaines difficultés ont été rencontrées. La mise en place et l'adaptation des graphes et de l'algorithme de Dijkstra ont été complexes, pour leur implémentation objet avec le langage JAVA. De plus, l'optimisation des boucles et parcours pour réduire le temps de calcul et éviter la redondance des opérations ont également posé quelques problèmes.

Enfin, les concepts sources et/ou utilisés dans ce code incluent les graphes, l'algorithme de Dijkstra, le parcours en profondeur et en largeur d'un graphe ainsi que le problème du voyageur TSP. Ces concepts sont tous essentiels à la mise en place d'un algorithme efficace pour trouver la ou les routes qui rapportent le plus de points entre des points données à parcourir dans une grille 2D.