


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.....	p3
Cahier des charges	p4
Analyse de Niveau 0.....	p5<>p11
Logigramme fonctionnel.....	p12
Arbre hiérarchique.....	p13
Analyse de niveau 1 à 3.....	p14<>p23
Ressources & Conclusion	p24<>p25

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](#) 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](#) en 1956. L'idée principale de l'[algorithme](#) 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 !

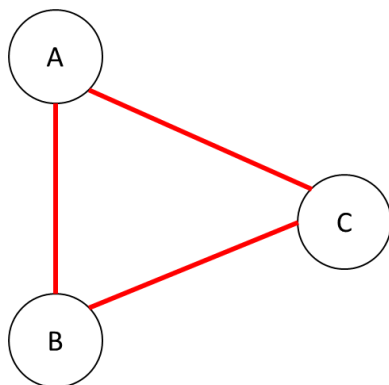
Analyse de Niveau 0

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 résumer l'algorithme de Dijkstra requiert un graphe orienté et pondéré (chaque arc possède un poids qui doit être ≥ 0).

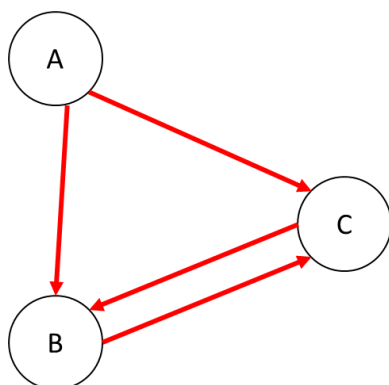
Qu'est-ce qu'un graphe ?

Un [graphe](#) en mathématiques est une structure composée d'objets appelés sommets et de relations entre eux appelées arêtes (ou arcs).



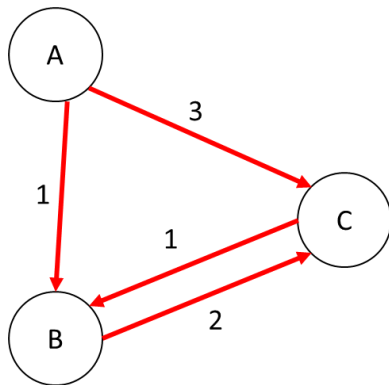
À gauche un graphe composé de trois sommets A, B, C relié par des arcs en rouge.

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 :

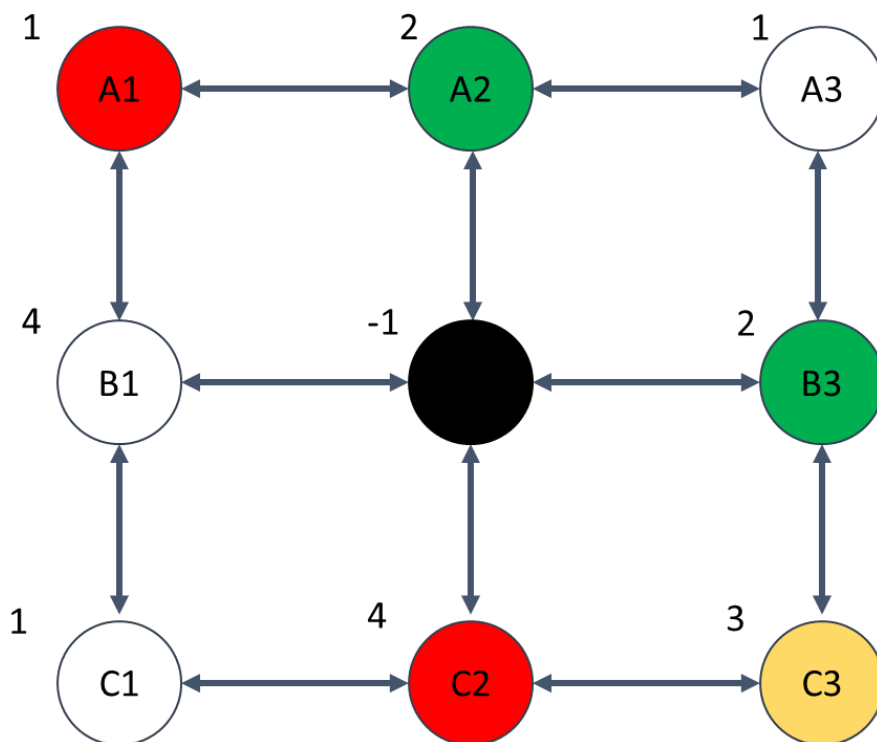
$$a = -1 \text{ et } b = 4 ; \quad l = 3 \text{ et } h = 3$$

1	2	1
4	-1	2
1	4	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 :

<i>Sommet A1</i>	<i>Sommet A2</i>	<i>Sommet A3</i>
<i>Sommet A4</i>	<i>Sommet A5</i>	<i>Sommet A6</i>
<i>Sommet A7</i>	<i>Sommet A8</i>	<i>Sommet A9</i>

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 :

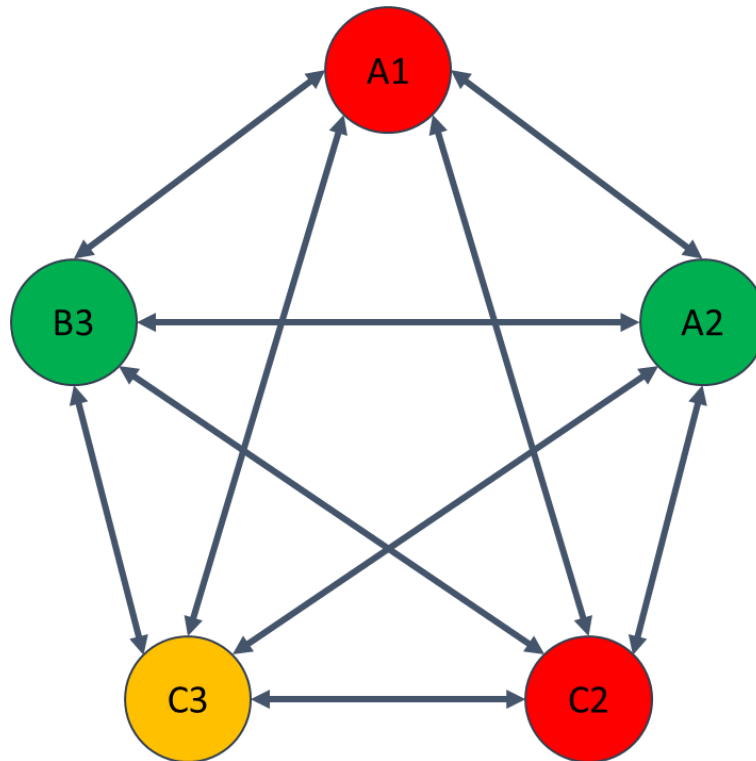


Sur ce graphe on y a placé pour exemple les points « types » du cahier des charges :

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

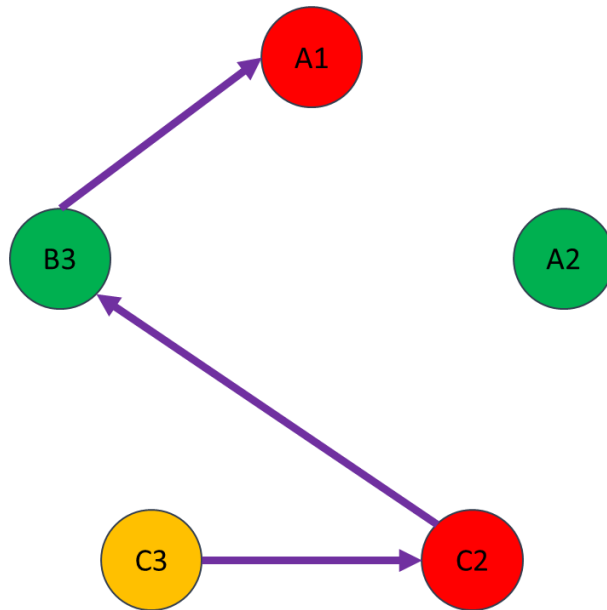
Comme Dijkstra requiert des arcs de poids ≥ 0 , dans le cas des obstacles nous déclarons le sommet comme étant un obstacle et son poids entrant est défini à un nombre très grand proche de « l'infini ».

La suite de notre raisonnement va être de « minimiser ce graphe » en un plus petit ne contenant que les points qui nous intéressent. 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 maximisent 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 algorithmie 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

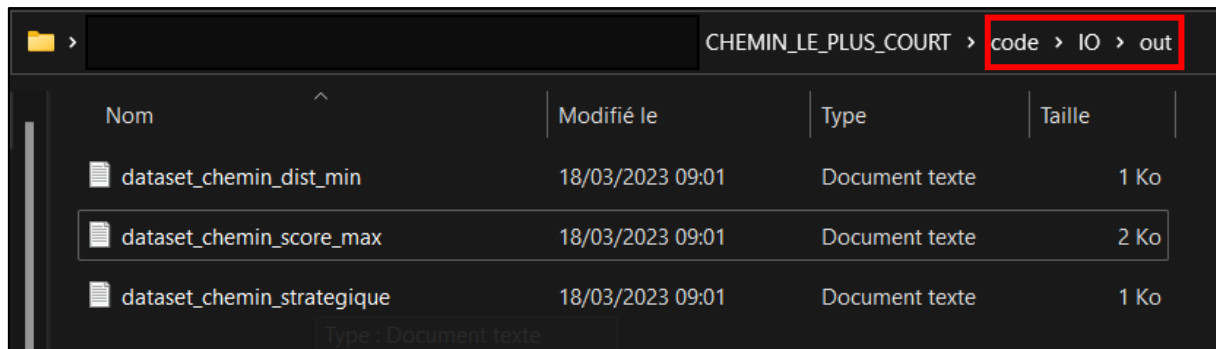
    Supprimer le premier sommet de chemin_tmp car il est déjà présent dans la liste
chemin_developpe

    Ajouter tous les sommets de chemin_tmp dans la liste chemin_developpe
Fin pour

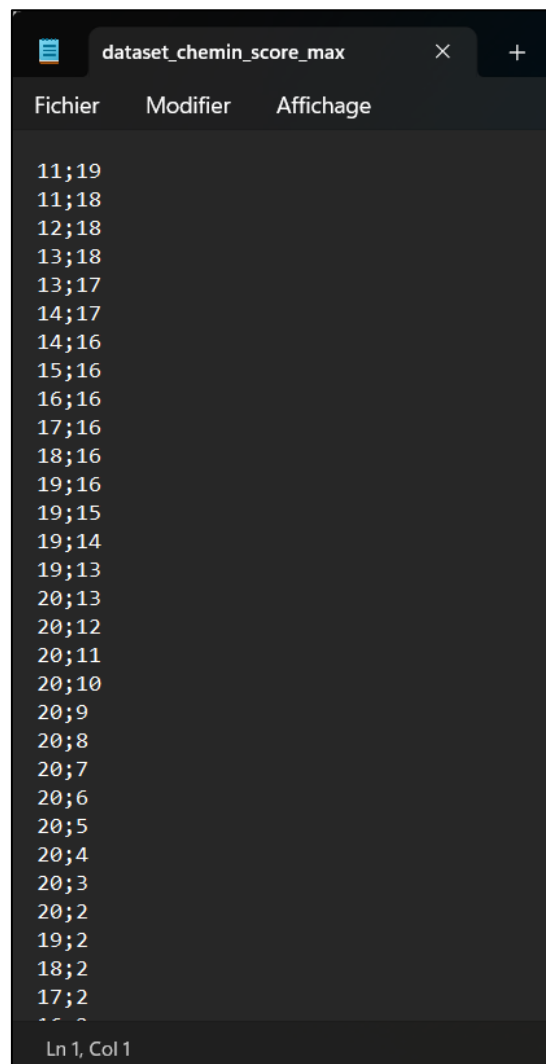
// Ajout du dernier sommet du chemin dans la liste développée
ajouter le dernier sommet de la liste chemin dans la liste chemin_developpe

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.

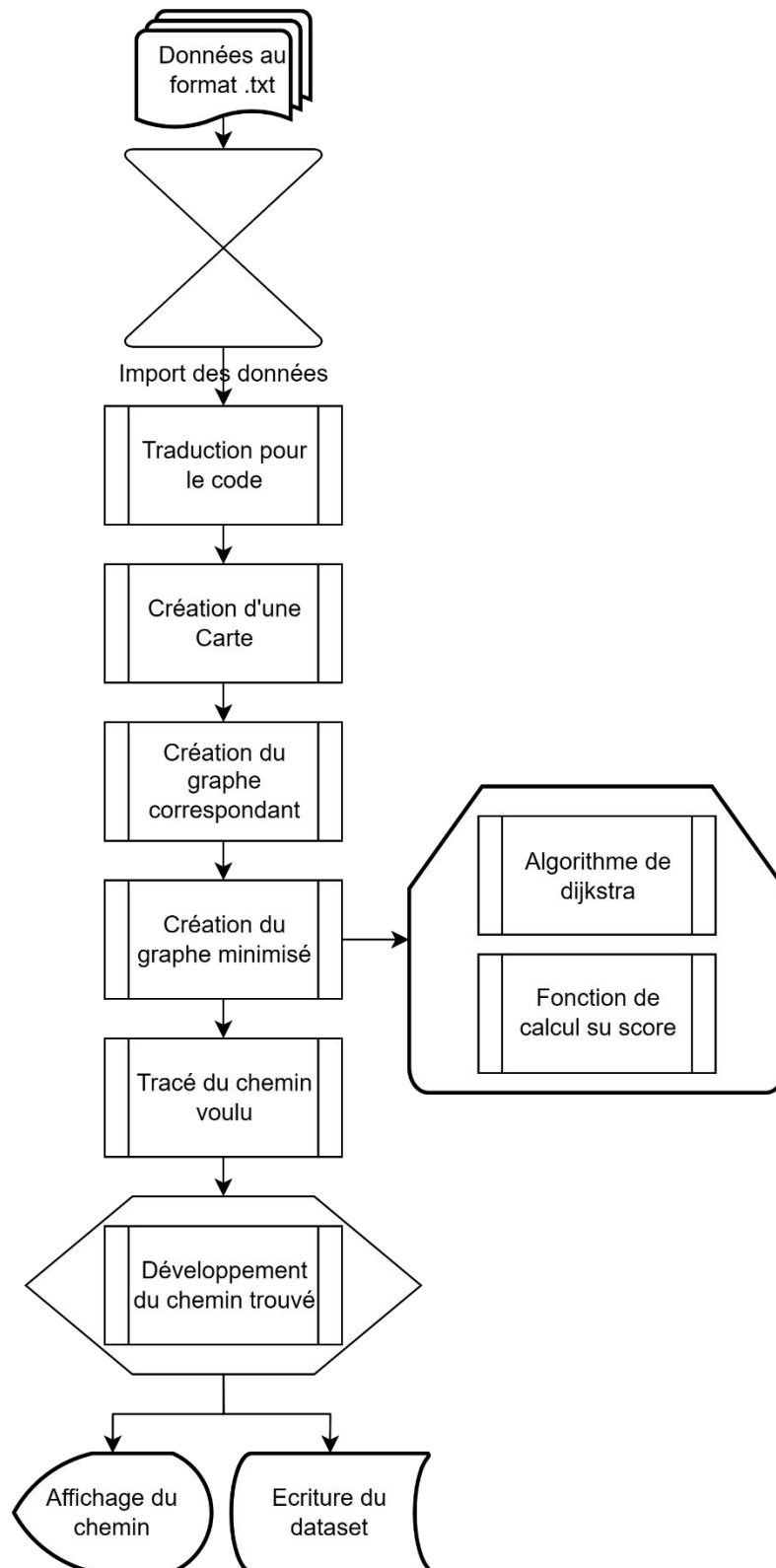


Nom	Modifié le	Type	Taille
dataset_chemin_dist_min	18/03/2023 09:01	Document texte	1 Ko
dataset_chemin_score_max	18/03/2023 09:01	Document texte	2 Ko
dataset_chemin_strategique	18/03/2023 09:01	Document texte	1 Ko

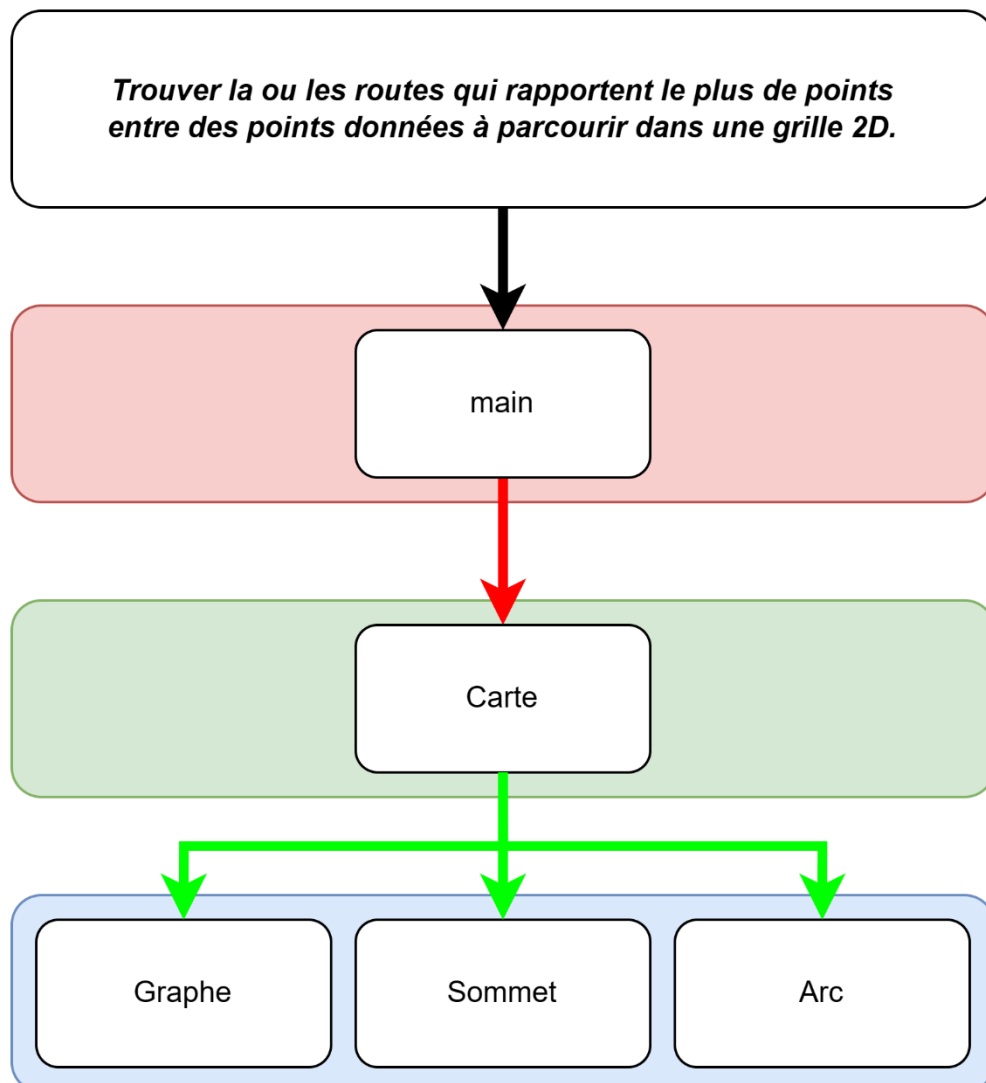


```
11;19
11;18
12;18
13;18
13;17
14;17
14;16
15;16
16;16
17;16
18;16
19;16
19;15
19;14
19;13
20;13
20;12
20;11
20;10
20;9
20;8
20;7
20;6
20;5
20;4
20;3
20;2
19;2
18;2
17;2
```

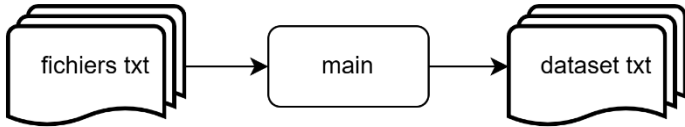
Logigramme fonctionnel



Arbre hiérarchique

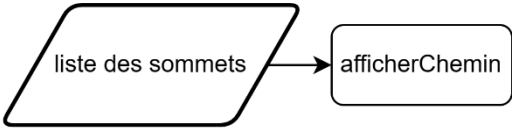


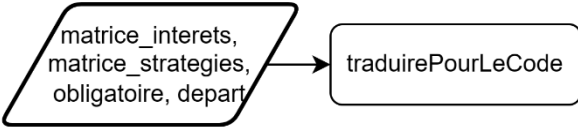
Analyse de niveau 1

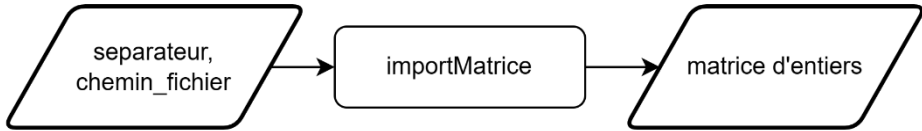
FPO main	
<p style="text-align: center;">Valeur Ajoutée</p> <p>Classe principale du programme. Elle répond directement à la problématique via ses composantes.</p>	
<p style="text-align: center;">INPUT</p> <p>depart.txt, obligatoire.txt, matrice.txt, matrice_interet.txt, matrice_strategique.txt</p>	<p style="text-align: center;">OUTPUT</p> <p>dataset_score_max.txt</p>
<p style="text-align: center;">Logigramme Fonctionnel</p>  <pre> graph LR A[fichiers txt] --> B(main) B --> C[dataset txt] </pre>	
<p style="text-align: center;">Service Fonctionnel</p> <p>Début du programme</p> <pre> 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 </pre> <p>Fin du programme</p>	


Analyse de niveau 2


FS0.1 Carte	
<p style="text-align: center;">Valeur Ajoutée</p> <p style="text-align: center;">Cette classe est une représentation objet de notre problème.</p>	
<p style="text-align: center;">INPUT</p> <p style="text-align: center;">.....</p>	<p style="text-align: center;">OUTPUT</p> <p style="text-align: center;">.....</p>
<p style="text-align: center;">Logigramme Fonctionnel</p> <div style="text-align: center; margin-top: 20px;"> <div style="border: 1px solid black; border-radius: 10px; padding: 10px; display: inline-block;"> Carte </div> </div>	
<p style="text-align: center;">Service Fonctionnel</p> <p>Début</p> <p>Attributs de la classe :</p> <ul style="list-style-type: none"> o depart: un tableau d'entiers représentant les coordonnées du point de départ o cases_obligatoires: un tableau d'entiers représentant l'indice des cases obligatoires parmi les cases stratégiques o grille: une matrice d'entiers représentant la grille de la carte o cases_interets: une matrice d'entiers représentant les coordonnées des cases d'intérêts et de leurs valeurs o cases_strategiques: une matrice d'entiers représentant les coordonnées des cases stratégiques o graphe: un objet Graphe correspondant à la carte <p>Constructeur de la classe :</p> <p>Carte(depart: tableau d'entiers, cases_obligatoires: tableau d'entiers, grille: matrice d'entiers, cases_interets: matrice d'entiers, cases_strategiques: matrice d'entiers)</p> <ul style="list-style-type: none"> o Associer chaque paramètre à l'attribut correspondant. <p>Fin</p>	

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  <pre>graph LR; Input[/matrice_interets, matrice_strategies, obligatoire, depart/] --> Process(traduirePourLeCode);</pre>	
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 séparateur, chemin_fichier	OUTPUT Matrice d'entiers
Logigramme Fonctionnel  <pre>graph LR; A[/séparateur, chemin_fichier/] --> B(importMatrice); B --> C[/matrice d'entiers/];</pre>	
Service Fonctionnel Fonction importMatrice(séparateur : chaîne de caractère, chemin_fichier : chaîne 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  <pre>graph LR; repertoire[/repertoire/] --> importerDonneesATravailler(importerDonneesATravailler); importerDonneesATravailler --> Carte[/Carte/];</pre>	
Service Fonctionnel <code>Fonction importerDonneesATravailler (rerpertoire : chaine de caractère)</code> 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	
<p align="center">Valeur Ajoutée</p> <p align="center">Cette méthode permet de créer un dataset à partir d'un chemin donné et de l'écrire dans un fichier spécifié.</p>	
<p align="center">INPUT</p> <p align="center">chemin, chemin_fichier</p>	<p align="center">OUTPUT</p> <p align="center">-----</p>
<p align="center">Logigramme Fonctionnel</p>  <pre> graph LR Input[/chemin, chemin_fichier/] --> Process(dataset) Process --> Output[dataset txt] </pre>	
<p align="center">Service Fonctionnel</p> <pre> Fonction dataset(chemin : liste de sommets, chemin_fichier : chaine de caractère) Début Pour chaque sommet du chemin On récupère les coordonnées x + 1 et y + 1 du sommet On les inscrit dans le fichier au chemin_fichier Fin pour Fin </pre>	

Analyse de niveau 3

FT0.1.1 Graphe	
<p style="text-align: center;">Valeur Ajoutée Cette classe est une représentation objet d'un graphe.</p>	
<p style="text-align: center;">INPUT </p>	<p style="text-align: center;">OUTPUT </p>
<p style="text-align: center;">Logigramme Fonctionnel</p> <div style="text-align: center; border: 1px solid black; width: 100px; margin: 20px auto; padding: 5px;"> Carte </div>	
<p style="text-align: center;">Service Fonctionnel</p> <p>Début</p> <p>Attributs de la classe :</p> <ul style="list-style-type: none"> o sommets : liste des sommets du graphe <p>Constructeur de la classe :</p> <p>Graphe(List<Sommet> sommets) : initialise la liste de sommets à partir de la liste donnée en paramètre</p> <p>Graphe(int[][] matrice_adjacences) :</p> <ul style="list-style-type: none"> o Crée les sommets du graphe à partir de la matrice d'adjacences donnée en paramètre o Pour chaque sommet créé : <ul style="list-style-type: none"> ▪ Récupère ses voisins dans la matrice d'adjacences ▪ Pour chaque voisin : <ul style="list-style-type: none"> • 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 o Fin Pour <p>Fin</p>	

FT0.1.2 Sommet	
<p style="text-align: center;">Valeur Ajoutée Cette classe est une représentation objet d'un sommet.</p>	
<p style="text-align: center;">INPUT </p>	<p style="text-align: center;">OUTPUT </p>
<p style="text-align: center;">Logigramme Fonctionnel</p> <div style="text-align: center; border: 1px solid black; width: 100px; margin: 0 auto; padding: 5px;"> Sommet </div>	
<p style="text-align: center;">Service Fonctionnel</p> <p>Début</p> <p>Attributs de la classe :</p> <ul style="list-style-type: none"> o nom : chaîne de caractères o x : entier coordonnée x o y : entier coordonnée y o predecesseurs : liste des sommets prédécesseurs o successeurs : liste des sommets successeurs o arcs : liste des arcs du sommet <p>Constructeur de la classe :</p> <p>Sommet(nom: String, x: int, y: int, predecesseurs: List<Sommet>, successeurs: List<Sommet>, arcs: List<Arc>, isUnObstacle: boolean)</p> <ul style="list-style-type: none"> o Associer chaque paramètre à l'attribut correspondant. <p>Fin</p>	

FT0.1.3 Arc	
Valeur Ajoutée Cette classe est une représentation objet d'un arc.	
INPUT	OUTPUT
Logigramme Fonctionnel <div data-bbox="715 931 877 1014">Arc</div>	
Service Fonctionnel Début Attributs de la classe : <ul style="list-style-type: none">o Depart : Sommet de départo Arrivee : Sommet d'arrivéeo poids : entier poids de l'arc Constructeur de la classe : Arc(depart: Sommet, arrivee: Sommet, poids: int) <ul style="list-style-type: none">o 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.