

# Analyse

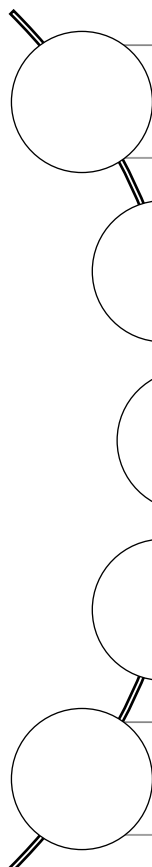
*Trouver la ou les routes qui rapportent le plus de points entre des points données à parcourir dans une grille 2D.*

**Licence RGI - Groupe ERP CISCO :**

**Maël RHUIN**

# Sommaire

---



Approche.....	p3<>p4
Cahier des charges .....	p5
Analyse.....	p6<>p17
Logigramme fonctionnel.....	p18
Conlusion.....	p19

# 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
```

**Réflexion autour de l'algorithme Tabou**

L'algorithme Tabou est une méthode heuristique utilisée pour résoudre des problèmes d'optimisation combinatoire. Il repose sur l'idée de maintenir une liste de mouvements interdits appelée liste tabou, afin d'éviter de revisiter les solutions déjà explorées. Cela permet à l'algorithme d'explorer l'espace de recherche de manière plus efficace et d'éviter de rester bloqué dans des optima locaux.

```
// 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

---

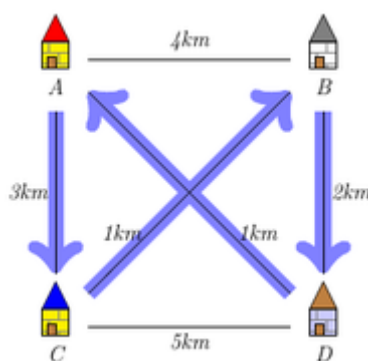
**Sujet** : Trouver la ou les routes qui rapportent le plus de points entre des points données à parcourir dans une grille 2D.

Le problème du voyageur de commerce (TSP - Traveling Salesman Problem en anglais) est un problème d'optimisation combinatoire dans lequel un voyageur doit visiter un ensemble de villes donné, en minimisant la distance totale parcourue. Le voyageur part d'une ville de départ, visite chaque ville une seule fois, et retourne finalement à la ville de départ.

Le lien entre le problème du voyageur de commerce et le sujet de trouver les routes qui rapportent le plus de points entre des points donnés dans une grille 2D réside dans la nature de l'optimisation. Dans le TSP, l'objectif est de minimiser la distance totale parcourue, tandis que dans le sujet donné, l'objectif est de maximiser le nombre de points accumulés.

En considérant les points donnés dans une grille 2D comme des villes à visiter, le problème devient similaire au TSP, mais avec une métrique différente pour l'optimisation. Au lieu de minimiser la distance parcourue, il s'agit de trouver les routes qui maximisent la somme des points accumulés. Cela implique de trouver le chemin qui passe par les points donnés et qui permet de collecter le plus grand nombre de points possible.

Ainsi, bien que les objectifs diffèrent entre le TSP classique et le sujet donné, la problématique d'optimisation combinatoire de trouver le meilleur chemin à travers des points spécifiques est un point commun entre les deux.



En utilisant une approche de programmation orientée objet, j'ai découpé la résolution du problème en différentes classes qui interagissent entre elles pour atteindre l'objectif final :

La partie "**MAIN**" du programme est responsable de l'acquisition des données nécessaires, de la construction de l'objet **Carte**, de la résolution du problème à partir de cette carte et de l'écriture de la solution dans un fichier. C'est la partie principale du programme qui orchestre les différentes étapes.

L'objet **Carte** représente la carte du problème. Il contient des informations telles que la grille, le point de départ, les points d'intérêt avec leur score, les points stratégiques et les points obligatoires. La carte est également responsable de la création du graphe correspondant à la grille. Elle contient la méthode **pathFinder**, qui sera utilisée pour rechercher le chemin optimal via mes méthodes adaptée et notamment l'algorithme **Tabou**.

L'objet **Graphe** représente le graphe associé à la carte. Il contient une liste de sommets. Le graphe est responsable de l'implémentation de l'algorithme de **Dijkstra** pour calculer les chemins les plus courts. Il contient également d'autres méthodes pour le calcul des distances et des scores.

L'objet **Sommet** représente un sommet dans le graphe. Il possède des coordonnées (x, y) pour sa position dans la grille. Il contient également des listes de prédécesseurs, de successeurs et d'arcs, ce qui permet de modéliser les connexions entre les sommets. De plus, il a une propriété indiquant s'il s'agit d'un obstacle ou non.

L'objet **Arc** représente un arc entre deux sommets dans le graphe. Il possède une référence vers le sommet de départ et le sommet d'arrivée, ainsi qu'un poids qui représente la distance entre ces deux sommets.

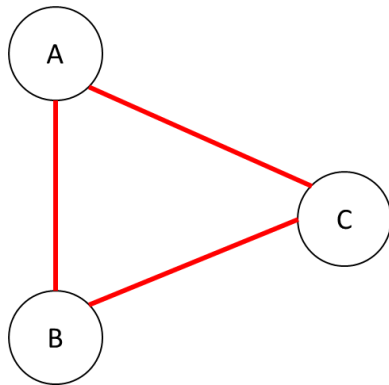
# Analyse

---

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  $\geq 0$ ).

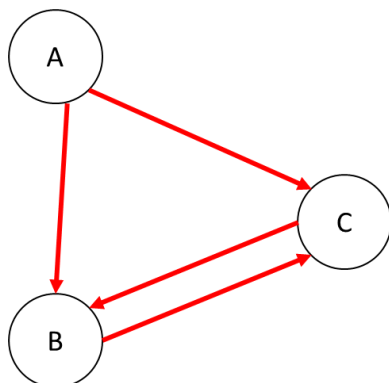
## Qu'est-ce qu'un graphe ?

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.

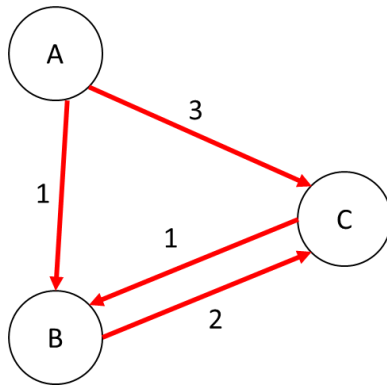
Un graphe **orienté** est un graphe où les arcs ont une direction représentée par une flèche.



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 :

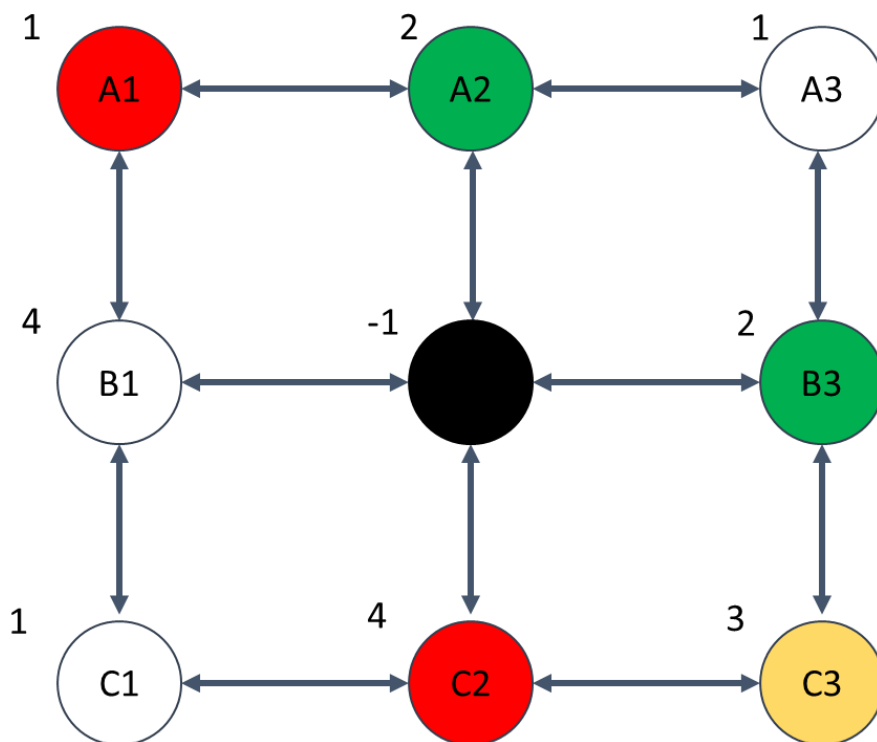
$$a = -1 \text{ et } b = 4 ; \quad l = 3 \text{ et } h = 3$$

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 :

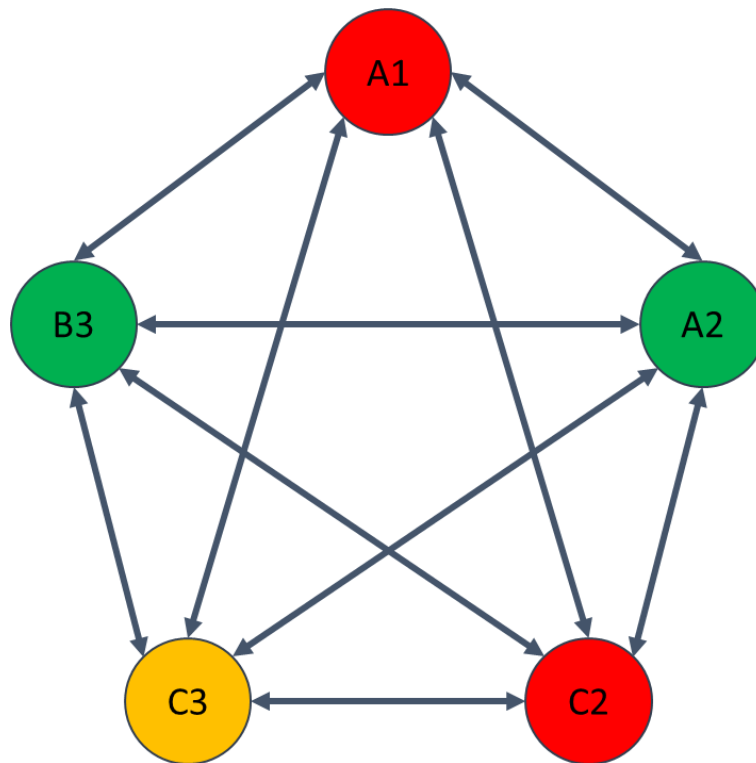


Sur ce graphe on y a placé pour exemple les points « types » du cahier des charges :

- 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  $\geq 0$ , 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 » .



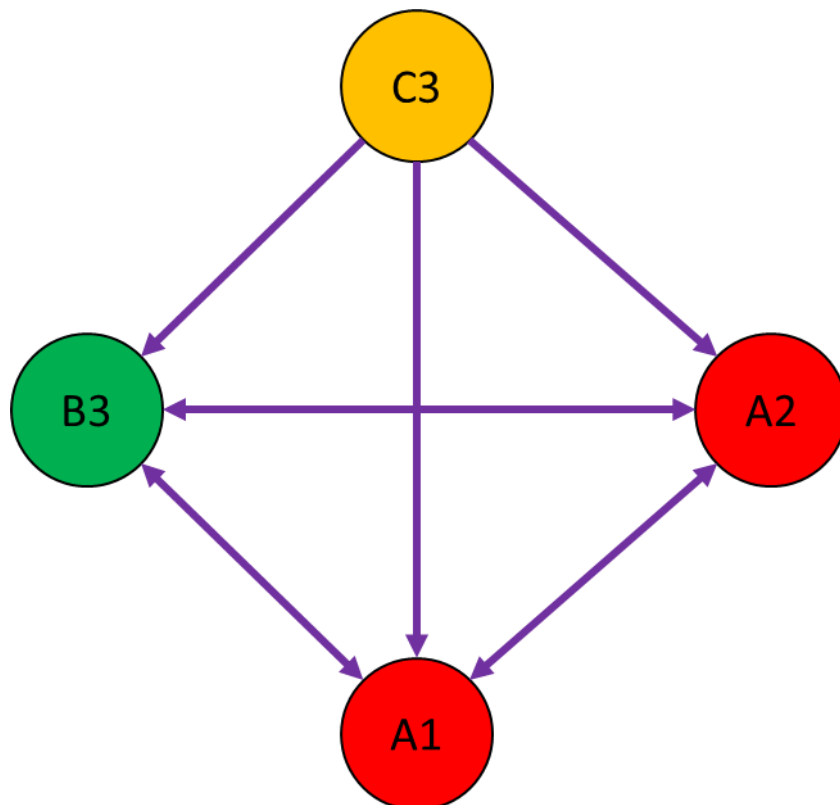
Nombre de villes $n$	Nombre de chemins candidats $\frac{1}{2}(n-1)!$
3	1
4	3
5	12
6	60
7	360
8	2 520
9	20 160
10	181 440
15	43 589 145 600
20	$6,082 \times 10^{16}$
71	$5,989 \times 10^{99}$

Afin de palier au problème d'un trop grand nombre de sommet qui pourrait amener à ce qui s'apparente à une explosion combinatoire montrer par le tableau précédent, il s'agit de réduire davantage ce graphe mais de manière méthodique afin de n'exclure que les « bonnes » parties. Le problème au final n'est pas tant la taille de la grille mais le nombre de points à prendre en compte dans notre recherche. Il faut le minimiser en dessous de 15 de préférence.

Pour cela j'ai :

- Supprimé les arcs retournant vers le sommet de départ
- Pour chaque sommet du graphe supprimé son arc le plus lourd
- Evalué pour chaque sommet son cout pour y aller, on le supprime dans le cas où y aller coûte plus cher que ce qu'il peut nous apporter.

On obtient à partir du graphe illustré précédemment celui-ci :



Fonction **streamlineGraphe**(sommets: Liste de Sommet, graphe: Graphe) :  
Graphe

```
sommets_creés <- nouvelle Liste de Sommet

// On ajoute les sommets

Pour chaque sommet dans sommets

    sommets_creés.ajouter(nouveau Sommet(sommet.getX(),
sommet.getY(), sommet.isUnObstacle()))

Fin Pour

// On crée des arcs pour relier chaque sommet à un autre

Pour chaque sommet_en_cours dans sommets_creés

    Pour chaque sommet dans sommets_creés

        Si sommet_en_cours.getNom() != sommet.getNom() Alors

            A <- sommet_en_cours dans le graphe minimisé
            B <- sommet dans le graphe minimisé
            dijkstra <- graphe.dijkstra(A, B)
            cout <- computeCoutChemin(dijkstra, graphe)

            // On ne crée pas d'arc si le chemin contient un obstacle

            contient_obstacle <- faux

            Pour chaque s dans dijkstra

                Si s.isUnObstacle() Alors

                    contient_obstacle <- vrai ; Sortir de la boucle

            Fin Si

        Fin Pour

        Si !contient_obstacle ET B != sommet_de_depart) Alors

            sommet_en_cours.ajouter(

                nouvel Arc(sommet_en_cours, sommet, cout))

            sommet.ajouterPredecesseur(sommet_en_cours)

            sommet_en_cours.ajouterSuccesseurs(sommet)

        Fin Si

    Fin Si

Fin Pour

Fin Pour

Retourner nouveau Graphe(sommets_creés)

Fin Fonction
```

```
Procédure reduceGraphe(graphe: Graphe)
    // Suppression des arcs à fortes distances
    Pour chaque sommet dans graphe.getSommets()
        // Calcul des distances vers les successeurs
        distances y ajouter calculerDistancesVersSuccesseurs(sommet,
graphe)

        // Trouver le successeur le plus éloigné
        sommetLePlusLoin <- trouverSuccesseurLePlusLoin(distances)

        // Supprimer l'arc entre le sommet en cours et son successeur
le plus lointain
        supprimerArc(sommet, sommetLePlusLoin)

        // Retirer le successeur le plus lointain des successeurs du
sommet en cours
        supprimerSuccesseur(sommet, sommetLePlusLoin)

        // Retirer également le sommet en cours des prédécesseurs du
successeur le plus lointain
        supprimerPredecesseur(sommetLePlusLoin, sommet)
    Fin Pour

    // Suppression des sommets onéreux : stratégiques et intérêts
    sommets_a_supprimer y ajouter trouverSommetsOnereux(graphe)
    Pour chaque sommet_a_supprimer dans sommets_a_supprimer
        supprimerSommetsAdjacents(sommet_a_supprimer, graphe)
        graphe.getSommets().supprimer(sommet_a_supprimer)
    Fin Pour
Fin Procédure
```

On peut désormais à partir du nouveau graphe trouver tous les chemins possibles dans notre cas et sélectionner celui qui résout notre problème, qui maximise le score :

```
Fonction pathFinder(): Liste de Sommets

// Minimisation du graphe
graphe_minimise <- streamlineGraphe()
afficherGraphe(graphe_minimise)

// Réduction du graphe
reduceGraphe(graphe_minimise)
afficherGraphe(graphe_minimise)

// Récupération du départ et des sommets obligatoires
depart <- depart du graphe minimisé
sommets_obligatoires <- sommets obligatoires du graphe minimisé

// Recherche des chemins possibles
Pour chaque sommet du graphe
  On lance un traitement en parallèle :
    [chemins <- findAllPaths(depart, sommet, sommets_obligatoires)]
Fin pour
afficherNombreChemins(chemins)

// Filtrage des chemins possibles
chemins_possibles <- filtrerChemins(chemins, sommets_obligatoires)
afficherNombreCheminsFiltrés(chemins_possibles)

// Calcul des scores des chemins
scores <- computePathScore(chemins_possibles, graphe_minimise)

// Sélection du chemin avec le meilleur score
meilleur_chemin <- sélectionnerMeilleurChemin(scores)

Retourner construireChemin(meilleur_chemin, graphe)
Fin Fonction
```

Le chemin trouvé est maintenant à développer dans le graphe initial via l'algorithme suivant :

```
Fonction buildPath(sommets: Liste de Sommets, graphe: Graphe): Liste de Sommets

    chemin <- Liste de Sommets
    chemin_tmp <- Liste de Sommets

    Pour i de 0 à sommets.size() - 1:
        A <- null
        B <- null

        Pour i allant de 0 au nombre de sommets dans sommets
            Si sommet est égal à sommets.get(i), alors A <- sommet
            Si sommetest égal à sommets.get(i + 1), alors B <- sommet
        Fin pour
        chemin_tmp <- dijkstra(A, B)
        chemin_tmp.supprimer(0)

        chemin.ajouterTout(chemin_tmp)

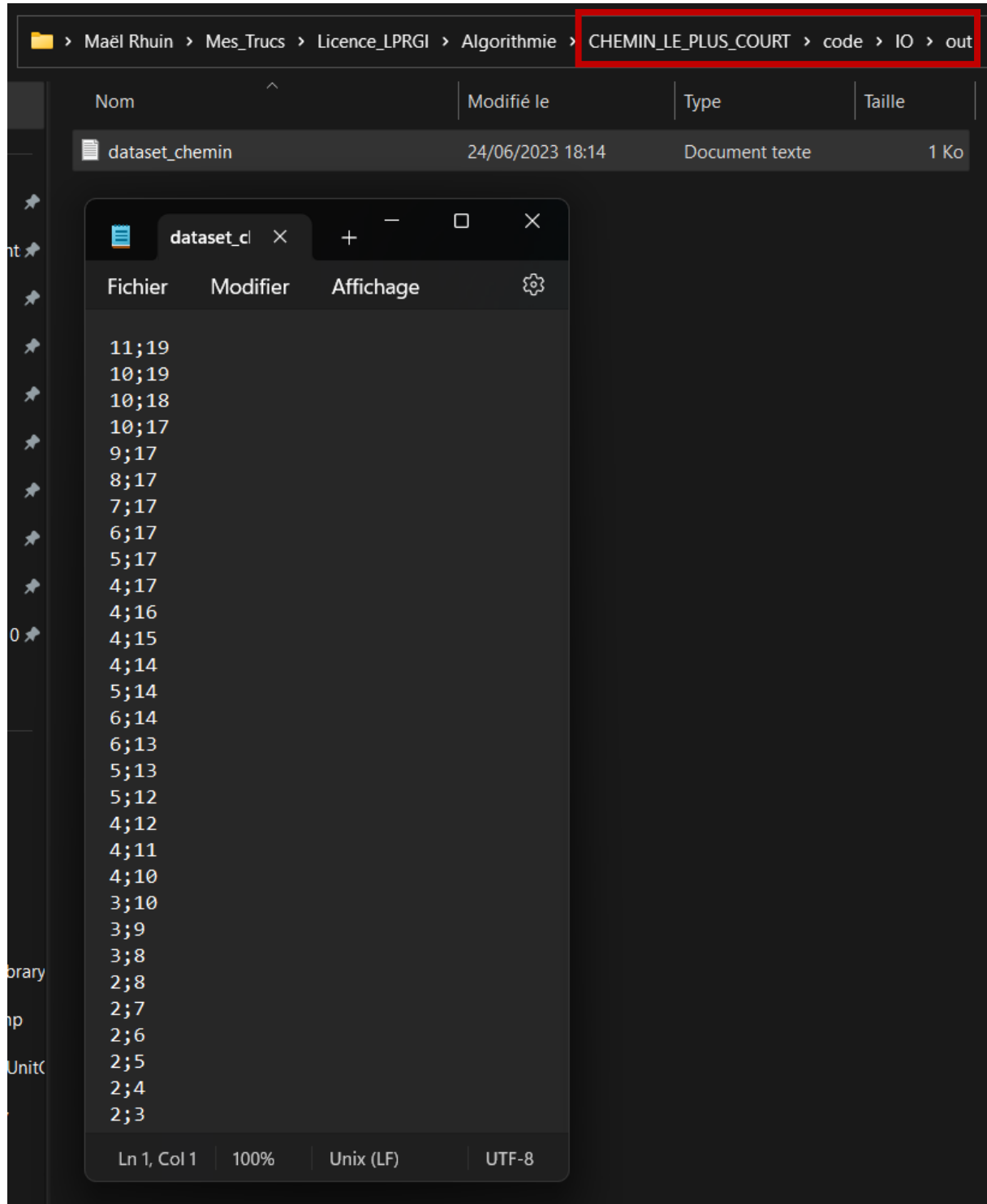
    chemin_tmp.vider()
    chemin_tmp.ajouterTout(chemin)

    chemin.vider()
    chemin.ajouter(sommets.get(0))
    chemin.ajouterTout(chemin_tmp)

    Retourner chemin
Fin Fonction
```

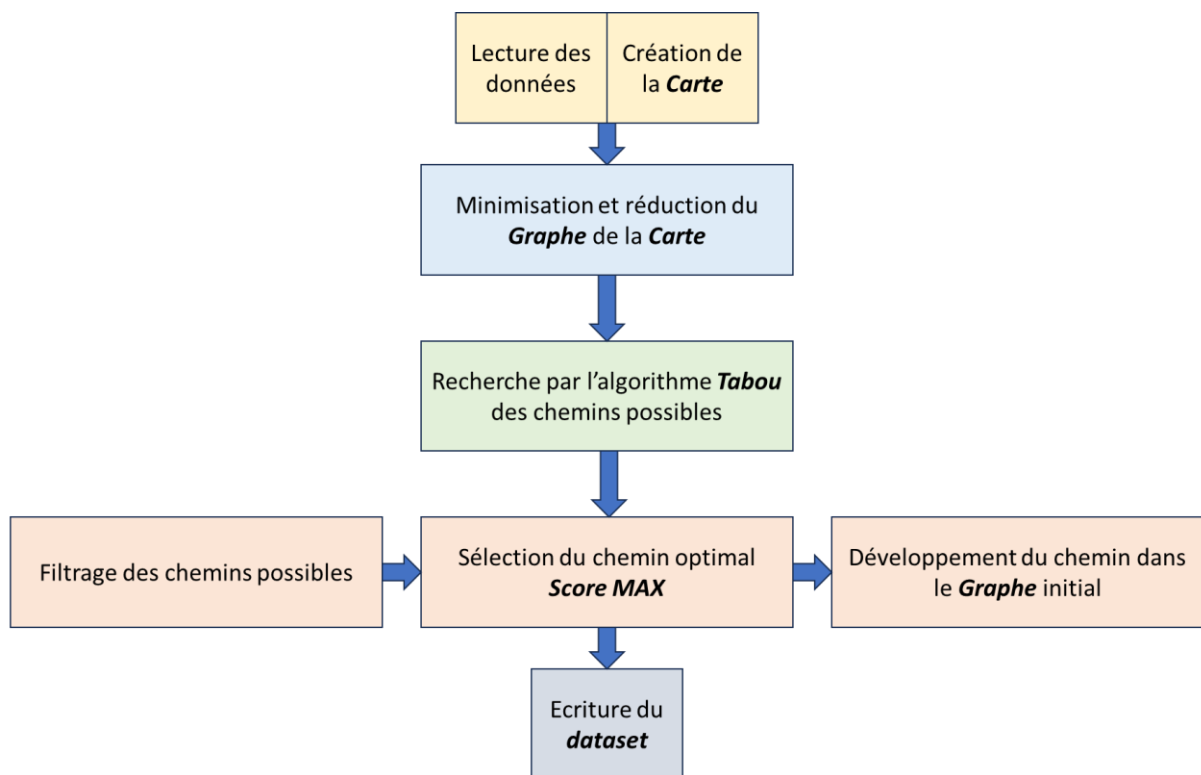


On peut enfin extraire sous forme d'un **dataset** le chemin final obtenu. De façon à avoir les coordonnées **{ colonne ; ligne }** de chaque sommet traversé.



# Logigramme fonctionnel

---



## Conclusion

---

En conclusion, ce code présente des avantages indéniables en utilisant l'algorithme de Dijkstra pour trouver efficacement les chemins les plus courts dans le but de former un chemin global pour un ensemble de points donné. Cela offre une grande flexibilité, car la fonction de construction du chemin final peut être ajustée pour privilégier la vitesse ou maximiser le score.

Durant le développement de ce code, plusieurs difficultés ont été rencontrées. L'implémentation objet des graphes et de l'algorithme de Dijkstra en utilisant le langage JAVA a posé des défis techniques. De plus, l'optimisation des boucles et des parcours pour réduire le temps de calcul et éviter les opérations redondantes a également nécessité une attention particulière.

Enfin, il est important de souligner que ce code met en œuvre des concepts clés tels que les graphes, l'algorithme de Dijkstra, le parcours en profondeur et en largeur des graphes, ainsi que le problème classique du voyageur de commerce (TSP). Il convient de noter que le problème TSP est connu pour sa complexité combinatoire, car le nombre de chemins possibles augmente de manière exponentielle avec le nombre de points à parcourir. Cela signifie que, pour des instances de problèmes de grande taille, la recherche de la meilleure solution devient extrêmement coûteuse en termes de temps de calcul.

Malgré ces défis, ce code constitue une base solide pour résoudre le problème du TSP dans un contexte spécifique, en prenant en compte les contraintes et les objectifs définis. Des améliorations supplémentaires pourraient être apportées pour gérer les obstacles et améliorer les performances, mais cela dépendrait des spécifications et des exigences spécifiques du problème à résoudre.