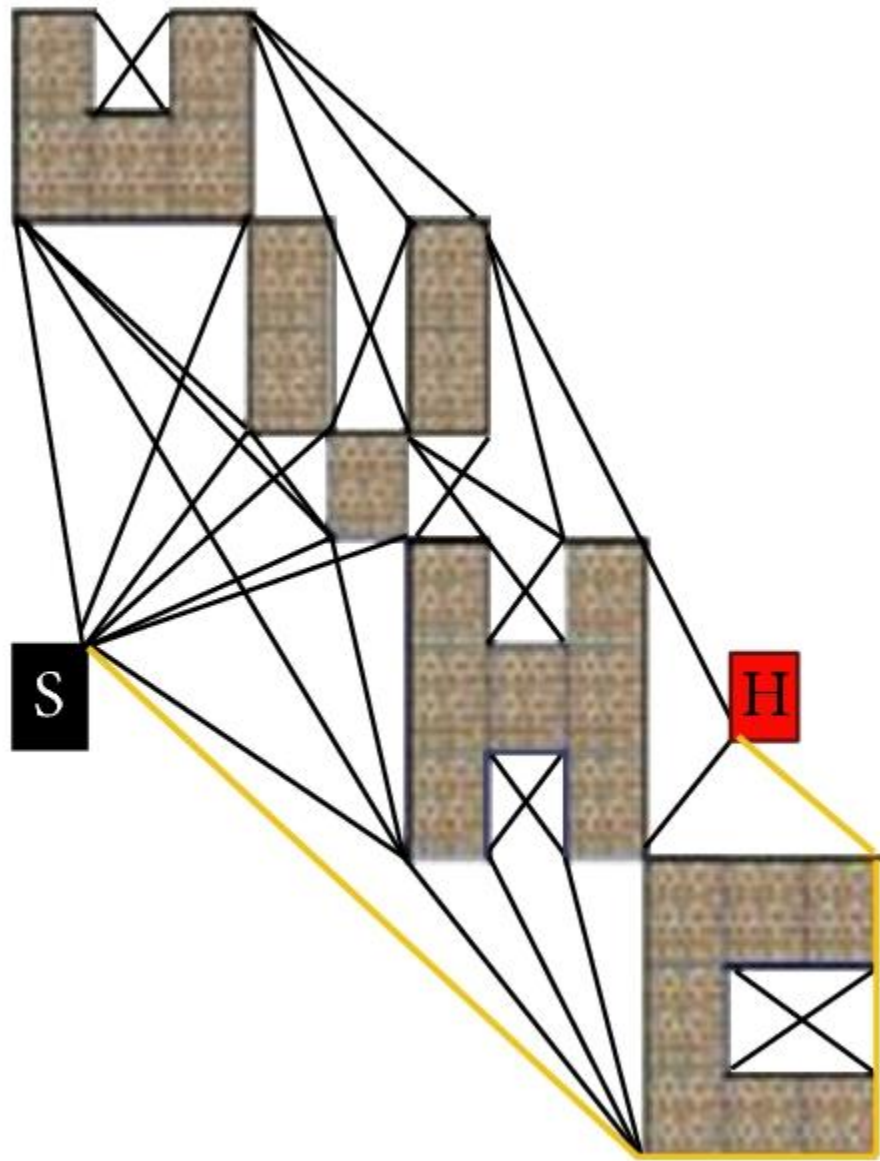


25 JUIN 2023



***Image 1 :** Graphe de l'ensemble des chemins du point S (point de départ) au point d'arriver (point H).*

ANALYSE FONCTIONNELLE

LE PATHFINDER

Projet effectué par :

RÉMI OMETZ – Alternant LP RGI CISCO

SÉBASTIAN ZITOUNI – Alternant LP RGI ERP

Table des matières

I.	Cahier des charges	4
II.	Analyse descriptive.....	4
A.	Description du projet	4
B.	Principe.....	5
C.	L'objectif	6
D.	Les contraintes	7
E.	Résultat du programme	7
III.	Arbre hiérarchique fonctionnelle	9
IV.	Pseudo-code du projet	10
A.	Classe Dijkstra	10
1.	Dijkstra	10
2.	SelectionPoids	11
3.	ParcoursVoisinCase.....	12
4.	VerificationConditionChemin.....	13
5.	ReconstitutionChemin	14
B.	Classe Trajet.....	16
1.	Attribut	16
2.	Constructeur	16
3.	AfficheTrajet	16
D.	Classe Map.....	18
1.	Attribut	18
2.	Constructeur	18
3.	AfficheMapEtTrajet.....	18
4.	ValeurPoint	21
E.	Classe Point	22
1.	Attribut	22
2.	Constructeur	22
3.	Getter/setter	23
4.	Getter	23
F.	Classe Program.....	24
1.	Main.....	24
2.	RecuperationPoints.....	25
3.	ChoixDuTrajet	25
4.	NombreDePointsAAteindre	26

5.	ChoixEntrePointDepartEtUnPointImportant.....	27
6.	SuppressionDesTrajetsContenantLesPointsParcourues.....	28
7.	ChoixDuTrajetEntre2PointsEnFonctionDuScore	29
V.	Conclusion	30

I. 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) :

X	Y	N#
2	2	1
12	12	2
6	13	3
8	6	4
14	17	5
15	3	6
19	2	7

Case d'intérêts (vert) :

X	Y	Valeur
2	7	18
2	19	12
7	2	13
11	14	9
13	5	25
18	6	8
19	17	5

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 XY (11;19) et doit se rendre aux points stratégiques 1, 3, 6 & 7

Chaque point stratégique lui rapporte 30 points.

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 !

II. Analyse descriptive

A. Description du projet

C'est un projet qui a pour but de nous faire réfléchir à un algorithme qui doit parcourir toute une carte. Cette carte possède un point de départ, un point d'arrivée, des obstacles, des points de passages obligatoires et des points de passages secondaires. Le point d'arrivée est l'endroit où l'algorithme commence et qui doit aller jusqu'au point d'arrivée. Une fois qu'il a atteint l'arrivée, le programme se termine.

Cependant, il peut rencontrer des obstacles sur son chemin. L'algorithme ne peut que les contourner. De plus, pour que l'algorithme soit considéré comme valide, il doit passer obligatoirement par tous les points obligatoires.

Concernant les points de passages secondaires, ils ne sont pas obligatoires et ne rapporte que des bonus.

B. Principe

Le principe du programme est dans la détection des points, la détermination des chemins et la sélection du chemin le plus court.

	A	B	C
A			
B			
C			
D			
E		Arriver	
F			
G			
H	Départ		

Figure 2 : Exemple de carte montrant un chemin possible pour atteindre l'arriver

Le programme doit comprendre qu'il doit commencer au départ qu'il a deux points accessibles. Pour qu'il puisse déterminer le quel il doit choisir, des « poids » sont attribués à toutes les cases de cartes. Ces poids vont de 1 à 4.

	A	B	C
A	3	1	2
B	1	1	4
C	3	4	2
D	2	3	4
E	1	Arriver	1
F	1	2	1
G	1	2	2
H	Départ	2	3

Figure 3 : Exemple de carte en ajoutant un "poids" à chaque case

De cette manière, deux chemins sont possibles pour atteindre le plus rapidement l'arrivée :

1. [A,H] → [A,G] → [A,F] → [A,E] → [B,E] (Chemin bleu) avec un poids total de 3
2. [A,H] → [B,H] → [B,G] → [B,F] → [B,E] (Chemin rose) avec un poids total de 6

Sachant que l'on veut le chemin avec le plus faible poids, le programme doit choisir la première solution.

Maintenant, la carte peut posséder des obstacles. Ils sont modélisés par des poids de valeurs -1.

	A	B	C
A	3	1	2
B	-1	1	4
C	3	4	-1
D	2	3	4
E	1	Arriver	1
F	1	2	1
G	1	-1	2
H	Départ	2	3

Figure 4 : Exemple de carte avec les obstacles modélisés

A cause de ces obstacles, le programme ne peut plus prendre le chemin suivant :

$[A,H] \rightarrow [B,H] \rightarrow [B,G] \rightarrow [B,F] \rightarrow [B,E]$

Il devra contourner la case [B,G], ce qui donne comme chemin possible :

$[A,H] \rightarrow [B,H] \rightarrow [C,H] \rightarrow [C,G] \rightarrow [C,F] \rightarrow [C,E] \rightarrow [B,E]$

	A	B	C
A	3	1	2
B	-1	1	4
C	3	4	-1
D	2	3	4
E	1	Arriver	1
F	1	2	1
G	1	-1	2
H	Départ	2	3

Figure 5 : Exemple de carte avec le chemin contournant l'obstacle

Pour finir, le programme doit posséder la notion de score. Les points obligatoires rapportent 30 points et les points secondaires rapportent un nombre de point prédéfinis dans le cahier des charges.

Formule de calcul des points :

Points des lieux obligatoires + points des lieux secondaires traversés - poids de chaque déplacement.

C. L'objectif

L'objectif du programme est de trouver le chemin ayant le score le plus haut, passant par tous les points obligatoires.

D. Les contraintes

Voici la liste des contraintes :

1. La détection des points accessibles ne peut pas se faire en diagonale. Les points accessibles peuvent être : haut, bas, droite, gauche ;

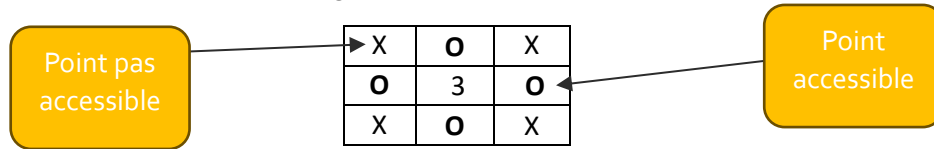


Figure 6 : Modélisation des points accessibles

2. La carte à 20 lignes et 20 colonnes ;
3. Les points ayant comme poids -1 ne sont jamais accessibles ;
4. Les points ayant comme poids -1 doivent être contourner et pas sautés ;
5. Le programme doit passer au moins 1 fois par tous les points obligatoires
6. Le programme doit calculer tous les chemins possibles entre tous les points obligatoires vers obligatoire, obligatoires vers secondaires et secondaires vers secondaires.
7. Tous les chemins doivent être enregistré dans un dataset

E. Résultat du programme



Légende :

Case bleu : Case parcourue par le programme

Case rouge : Point obligatoire

Les nombres sont les poids de chaque case

Le score est affiché avant la carte

Figure 7 : Ecran de résultat du programme

```

-----
Point Départ : ( 2 , 14 )
Point Arrivée : ( 6 , 13 )
Cout : 12
( 6 , 14 ) - ( 5 , 14 ) - ( 4 , 14 ) - ( 3 , 14 ) - ( 2 , 14 ) - ( 2 , 14 ) -
Score : 18
-----
Point Départ : ( 6 , 13 )
Point Arrivée : ( 11 , 14 )
Cout : 14
( 10 , 14 ) - ( 10 , 13 ) - ( 9 , 13 ) - ( 8 , 13 ) - ( 7 , 13 ) - ( 6 , 13 ) - ( 6 , 13 ) -
Score : 13
-----
Point Départ : ( 11 , 14 )
Point Arrivée : ( 19 , 2 )
Cout : 48
( 19 , 3 ) - ( 19 , 4 ) - ( 19 , 5 ) - ( 19 , 6 ) - ( 19 , 7 ) - ( 19 , 8 ) - ( 19 , 9 ) - ( 18 , 9 ) - ( 17 , 9 ) - ( 16 , 9 ) - ( 15 , 9 ) - ( 15 , 10 ) - ( 14 , 10 ) - ( 13 , 10 ) - ( 13 , 11 ) - ( 12 , 11 ) - ( 12 , 12 ) - ( 12 , 13 ) - ( 12 , 14 ) - ( 11 , 14 ) - ( 11 , 14 ) -
Score : -5
-----
Point Départ : ( 19 , 2 )
Point Arrivée : ( 13 , 5 )
Cout : 21
( 14 , 5 ) - ( 14 , 4 ) - ( 14 , 3 ) - ( 14 , 2 ) - ( 15 , 2 ) - ( 15 , 1 ) - ( 16 , 1 ) - ( 17 , 1 ) - ( 18 , 1 ) - ( 19 , 1 ) - ( 19 , 2 ) - ( 19 , 2 ) -
Score : -1
-----
Point Départ : ( 13 , 5 )
Point Arrivée : ( 2 , 2 )

```

Figure 8 : Liste de tous les chemins calculés par le programme affiché avant le score et la carte

III. Arbre hiérarchique fonctionnelle

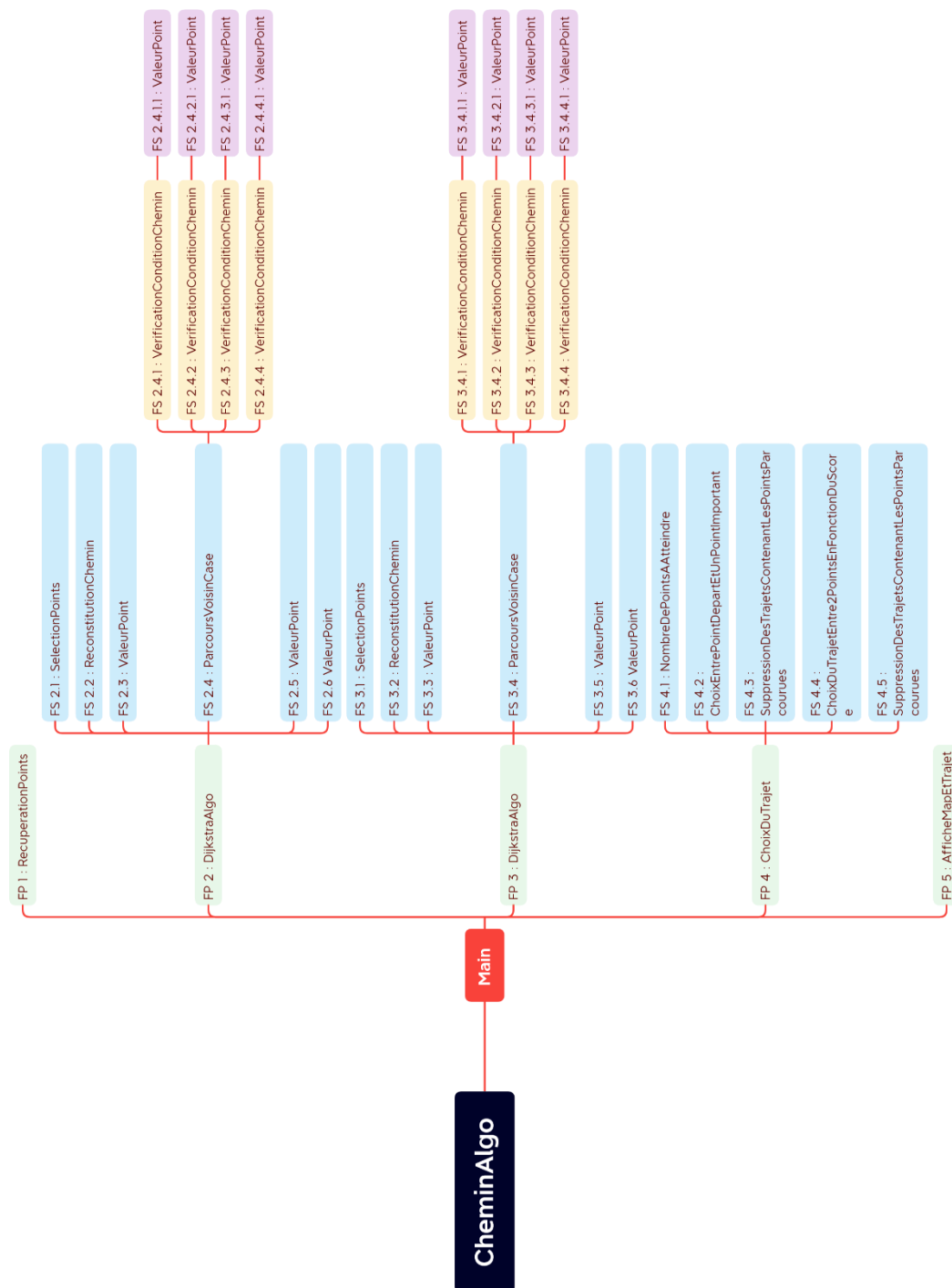


Figure 9 : Arbre hiérarchique fonctionnel

IV. Pseudo-code du projet

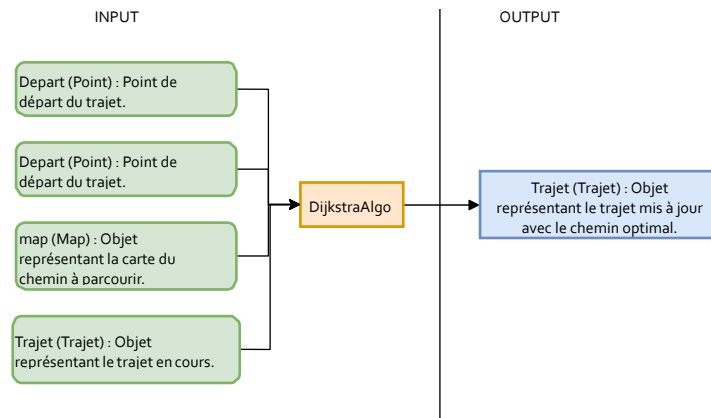
A. Classe Dijkstra

1. Dijkstra



Valeur ajoutée : Détermination du chemin le plus court entre deux points sur une carte.

Service fonctionnel



Pseudo-code

```
DijkstraAlgo
VARIABLE
  Liste de point FileAttente
  Liste de point ListVoisin

  Point PointActuel

  Entier DistanceTotal

  Booléen Arrive = faux

  Tableau de caractère TabAvance
DEBUT
  PointDep de Trajet <- Depart
  PointArr de Trajet <- Arriver

  Distance de Depart <- 0
  Ajouter Depart a FileAttente

  Tant que FileAttente n'est pas vide ET Arrive = faux Faire

    PointActuel <- SelectionPoinds(FileAttente)

    Retirer PointActuel de FileAttente

    SI X de PointActuel = X de Arriver ET Y de PointActuel = Y de Arriver ALORS
      TabAvance[X de PointActuel][Y de PointActuel] <- '0'

      ReconstitutionChemin(PointActuel, Trajet)

      DistanceTotal <- Distance de PointActuel + map[X de PointActuel][Y de PointActuel]
      CoutTrajet de Trajet <- DistanceTotal
      Arriver <- vrai
    SINON
      ListVoisin <- ParcoursVoisinCase(PointActuel, map)

      POUR chaque Voisin de ListVoisin FAIRE
        SI Distance de Voisin > Distance de PointActuel + map[X de PointActuel][Y de PointActuel]
          FAIRE
            Distance de Voisin <- Distance de PointActuel + map[X de PointActuel][Y de PointActuel]
            Parent de Voisin <- PointActuel

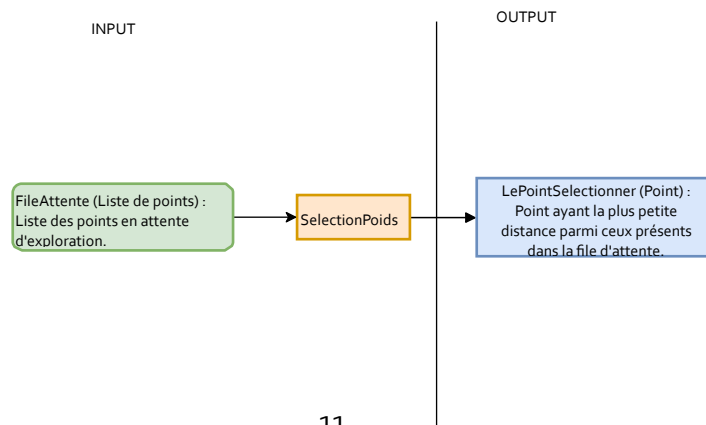
            SI pas X de Voisin ET Y de Voisin n'existe pas dans FileAttente FAIRE
              Ajouter Voisin a FileAttente
              TabAvance[X de PointActuel][Y de PointActuel] <- 'X'
            FIN
          FIN
        FIN
      FIN
      Vider ListVoisin
    FIN
  FIN
FIN
```

2. SelectionPoids



Valeur ajoutée : Sélection du point avec la plus petite distance dans la file d'attente.

Service fonctionnel



```

SelectionPoids
DEBUT
    Point LePointSelectionner <- premier élément de FileAttente

    POUR chaque Point de FileAttente ALORS
        SI Distance de Point < Distance de LePointSelectionner
            ALORS LePointSelectionner <- Point
        FIN
    FIN

    RETOURNER LePointSelectionner
FIN

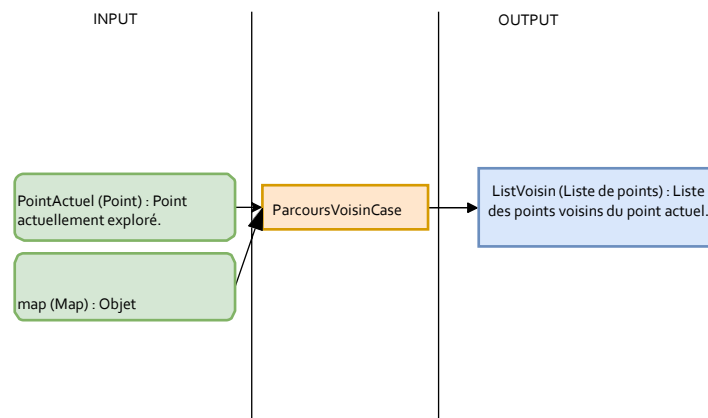
```

3. ParcoursVoisinCase



Valeur ajoutée : Identification des points voisins du point actuel sur la carte.

Service fonctionnel



Pseudo-code

```
ParcoursVoisinCase
VARIABLE
    Liste de Point ListVoisin
    Entier PositionX
    Entier PositionY
DEBUT
    PositionX <- X de PointActuel
    PositionY <- Y de PointActuel

    SI PositionY - 1 >= 0 ALORS
        SI VerificationConditionChemin(map, PositionX, PositionY - 1) FAIRE
            Nouveau Point UnVoisin <- Point(PositionX, PositionY - 1)
            Distance de UnVoisin <- 999999
            Ajouter UnVoisin a ListVoisin
        FIN
    FIN

    SI PositionY + 1 < 20 ALORS
        SI VerificationConditionChemin(map, PositionX, PositionY + 1) FAIRE
            Nouveau Point UnVoisin <- Point(PositionX, PositionY + 1)
            Distance de UnVoisin <- 999999
            Ajouter UnVoisin a ListVoisin
        FIN
    FIN

    SI PositionX + 1 < 20 ALORS
        SI VerificationConditionChemin(map, PositionX + 1, PositionY)
FAIRE
        Nouveau Point UnVoisin <- Point(PositionX, PositionX + 1)
        Distance de UnVoisin <- 999999
        Ajouter UnVoisin a ListVoisin
    FIN
    FIN

    SI PositionX - 1 >= 0 ALORS
        SI VerificationConditionChemin(map, PositionX - 1, PositionY)
FAIRE
        Nouveau Point UnVoisin <- Point(PositionX, PositionX - 1)
        Distance de UnVoisin <- 999999
        Ajouter UnVoisin a ListVoisin
    FIN
    FIN

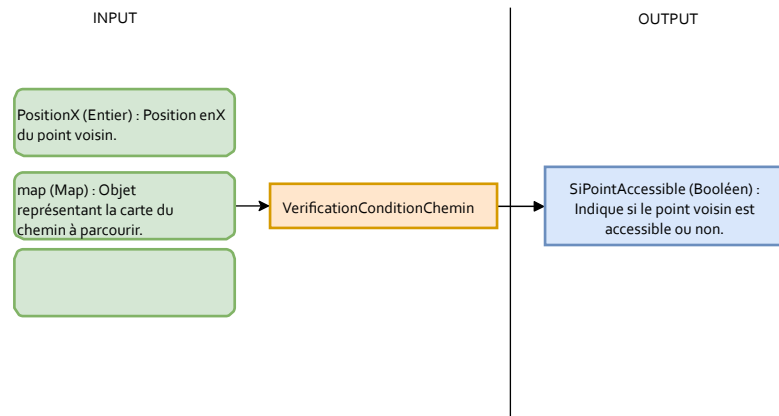
    RETOURNER ListVoisin
```

4. VerificationConditionChemin



Valeur ajouté : Vérification de l'accessibilité d'un point voisin sur la carte

Service fonctionnel




Pseudo-code

```
VerificationConditionChemin
VARIABLE
    Booléen SiPointAccessible
DEBUT
    SiPointAccessible <- faux

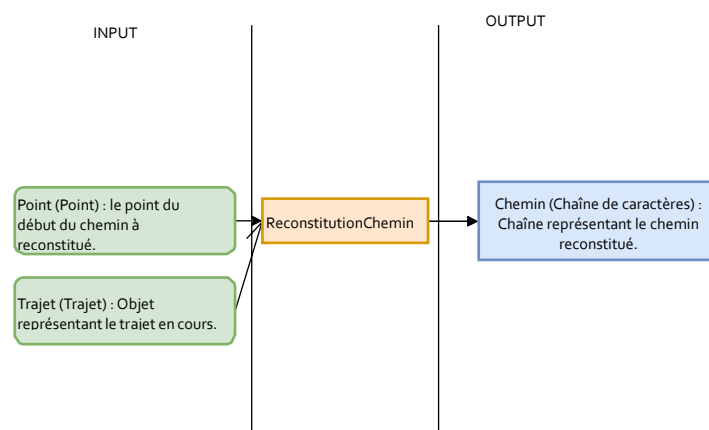
    SI pas map[X de PointActuel][Y de PointActuel] = -1
    FAIRE    SiPointAccessible <- vrai
    FIN

    RETOURNER SiPointAccessible
FIN
```

▼ 5. ReconstitutionChemin

 **Valeur ajoutée :** Reconstitution du chemin parcouru à partir d'un point spécifié jusqu'au point de départ, avec mise à jour du trajet.

Service fonctionnel



Pseudo-code



```
ReconstitutionChemin
VARIABLE
    Chaîne de caractère Chemin
DEBUT
    Chemin <- ""

    Tant que Parent de Point n'est pas null ALORS
        Chemin <- "(" + X de Point + ";" + Y de Point + ")" +
Chemin Point <- Parent de Point
    FIN

    Chemin <- "(" + X de Point + ";" + Y de Point + ")" + Chemin
    RETOURNER Chemin
FIN
```

B. Classe Trajet

1. Attribut

Description des variables

PointDep : Le point de départ du trajet

PointArr : Le point d'arriver (final) du trajet

ListPointPracourure : Liste de tous les points du trajet

CoutTrajet : Coût total du trajet

Pseudo-code

```
public Point PointDep, PointArr  
public Liste de Point  
ListPointPracourue public  
Entier CoutTrajet
```



2. Constructeur



Valeur ajouté : Construit un objet trajet.

Service fonctionnel

Aucune variable est en entrée. Néanmoins, l'objet construit est en sortie.

Pseudo-code



```
PointDep <- null  
PointArr <- null  
ListPointParcoursure <- Nouvelle Liste de  
PointTrajet <- 0
```

3. AfficheTrajet



Valeur ajouté : Affiche tous les points du trajet.

Service fonctionnel

Aucunes variables sont en entrées comme en sortie.

Pseudo-code


```

AfficheTrajet
DEBUT
    Afficher("-----")
    Afficher("Point Départ : ( " + X du PointDep du Trajet + " , " + Y du PointDep du Trajet + "
)") Afficher("Point Départ : ( " + X du PointArr du Trajet + " , " + Y du PointArr du Trajet + "
)") Afficher("Cout : " + CoutTrajet du Trajet)

    POUR chaque Point point de ListPointParcourue FAIRE
        Afficher(" " X du point + " , " + Y du point + " ) - ")
    FIN
    Afficher("")
FIN
```

D. Classe Map

1. Attribut

Description des variables

TabMap : Carte du programme. Cette carte comporte toutes les informations du fichier Excel.

Pseudo-code

Privé TabMap est un tableau d'entier de dimension 20,20

2. Constructeur



Valeur ajouté : Construit un objet map.

Service fonctionnel

Aucune variable est en entré. Néanmoins, l'objet construit est en sortie.

Pseudo-code

```
VARIABLE
  Entier ligne
  Entier colonne
DEBUT
  ligne <- 0

  Lecture du fichier CSV map.csv

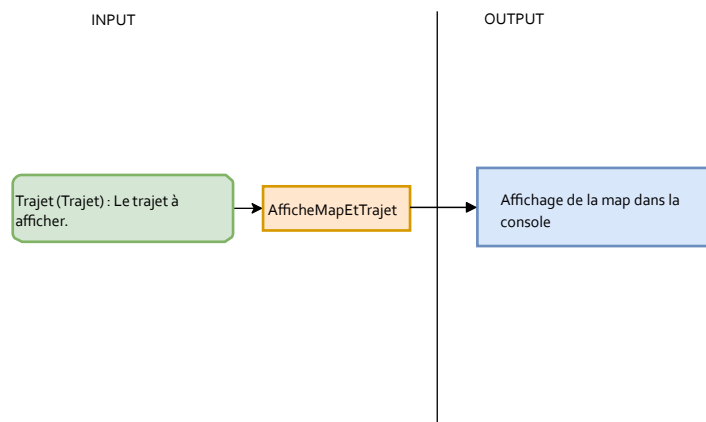
  TANT QUE la fin du fichier n'est atteinte FAIRE
    Chaîne de caractère line <- la ligne lu
    Tableau de chaîne de caractère values <- Separation de line avec ";" comme
séparateur
    POUR colonne allant de 0 à longueur du tableau values FAIRE
      SI TabMap[ligne, colonne] <- Convertir en entier values[colonne]
    FIN
    ligne <- ligne + 1
  FIN
FIN
```

3. AfficheMapEtTrajet



Valeur ajouté : Affiche la carte et le trajet du programme.

Service fonctionnel



Pseudo-code

```
VARIABLE
    Booléen PointEcritEnCouleur
    Booléen DejatEcrit
    Liste de point PointDejaEcrit
    Entier Ligne
    Entier Colonne
DEBUT
    PointEcritEnCouleur <- faux

    Pour Ligne allant de 0 à 20 par pas de 1 faire
        Afficher("|")
        Pour Colonne allant de 0 à 20 par pas de 1 faire
            Pour chaque Point p dans ListePointsParcourue de Trajet faire
                Si X de p = Ligne ET Y de p = Colonne alors
                    DejatEcrit <- faux
                    Pour chaque Point p2 dans la liste PointDejaEcrit
faire
                        Si X de p = X de p2 ET Y de p = Y de p2 alors
                            DejatEcrit <- vrai
                        Fin Si

                    Si DejatEcrit = faux faire
                        Si TabMap[ligne, colonne] = -1

                            Si GetUtile de p = 1 faire
                                Fond de l'affichage en rouge
                                Afficher(TabMap[ligne, colonne])
                            Sinon
                                Fond de l'affichage en bleu
                                Couleur de la police en noir
                                Afficher(TabMap[ligne, colonne])
                            Fin Si

                                Fond de l'affichage en noir
                                Couleur de la police en blanc
                                Afficher("|")
                            Fin Si
                        Sinon

                            Si GetUtile de p = 1 alors
                                Fond de l'affichage en rouge
                                Afficher(TabMap[ligne, colonne])
                            Sinon
                                Fond de l'affichage en bleu
                                Couleur de la police en noir
                                Afficher(TabMap[ligne, colonne])
                            Fin Si

                                Fond de l'affichage en noir
                                Couleur de la police en blanc
                                Afficher(" |")
                            Fin Si
                        PointEcritEnCouleur <- vrai
                        Ajouter p dans PointDejaEcrit
                    Fin Si
                Fin Si
            Fin pour chaque

            Si PointEcritEnCouleur = faux alors
                Si TabMap[ligne, colonne] = -1
                    Afficher(TabMap[ligne, colonne] + "|")
                Sinon
                    Afficher(TabMap[ligne, colonne] + " |")
                Fin Si
            Fin Si

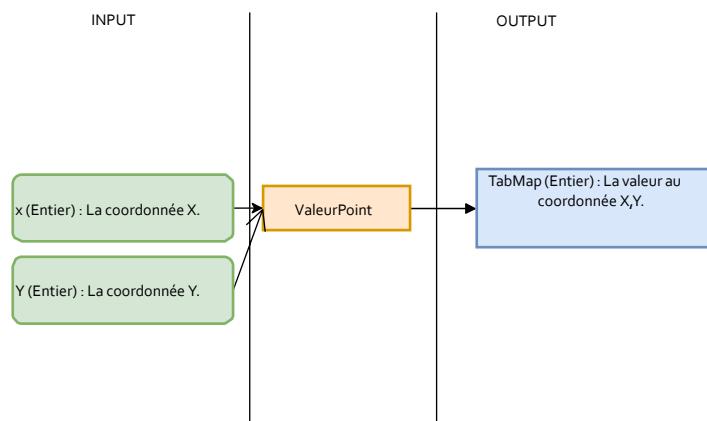
            PointEcritEnCouleur <- faux
        Fin Pour
    Afficher("")
Fin Pour
FIN
```

4. ValeurPoint



Valeur ajouté : Donne la valeur au coordonnée X, Y de la carte.

Service fonctionnel



Pseudo-code

```
ValeurPoint
DEBUT

  Retourne TabMap[x, y]
FIN
```

E. Classe Point

1. Attribut

Description des variables

x : Coordonnée X du

point.y : Coordonnée Y

du point.

valeur : Valeur au coordonnée X,Y de la carte.

utile : Si les points est utile à la parcourir ou non. Il peut être égal à 1 : Il est utile OU 0 : il n'est pas utile.

Pseudo-code

```
privé Entier x
privé Entier y
privé Entier valeur
privé Entier utile
```

2. Constructeur

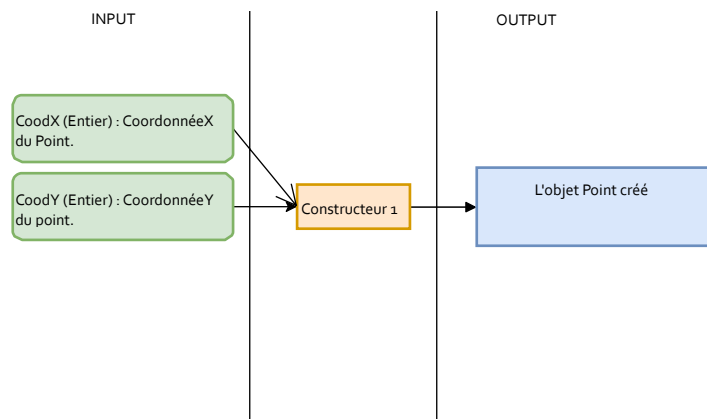


Valeur ajouté : Construit un objet trajet.

▼ Pseudo-code

Constructeur 1 :

Service fonctionnel

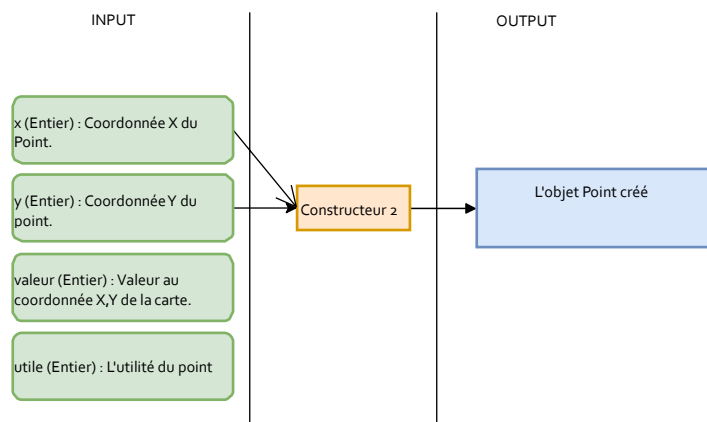


Pseudo-code

```
DEBUT
  x de Point <- CoordX
  y de Point <- CoordY
  utile de Point <- 0
FIN
```

Constructeur 2 :

Service fonctionnel



Pseudo-code

```
DEBUT
  x de Point <- 0
  y de Point <- 0
  valeur de Point <- valeur
  utile de Point <- utile
FIN
```

3. Getter/setter

SetDistance : Set la distance du point
GetSetX : Get ou Set la coordonnée X du point
GetSetY : Get ou Set la coordonnée Y du point
GetSetParent : Get ou Set le parent du point

4. Getter

GetUtile : Get l'utilité du point
GetValeur : Get la valeur du point

F. Classe Program

1. Main



Valeur ajouté : Lance toutes les fonctions pour que le programme fonctionne.

Description des variables

map : La carte du programme

PointDeDepart : le point de départ

PointsImportants : Liste de tous les points important (obligatoire)

TabTrajet : Tableau de tous les trajets possibles entre les points obligatoires et d'intérêt.

TrajetFinal : Le trajet total. Celui qui commence au point de départ et se termine par le point d'arrive en passant pour tous les points obligatoires.

Ligne : Numéro de ligne.

Colonne : Numéro de colonne.

Pseudo-code

```
VARIABLE
  Objet Map map
  Objet Point PointDeDepart
  Liste de Point PointsImportants
  Tableau de Trajet TabTrajet
  Trajet TrajetFinal
  Entier Ligne
  Entier Colonne

DEBUT
  map <- Nouvelle Map()
  PointDeDepart <- Nouveau Point(2, 14)
  PointsImportants <- Nouvelle Liste de Point()
  PointsImportants <- RecuperationPoints()
  TabTrajet <- Nouveau Tableau de Trajet[Le nombre d'éléments de 'PointsImportants', Le nombre d'éléments
de 'PointsImportants']

  // Calcul des trajets entre tous les points importants
  POUR chaque Ligne allant de 0 à Le nombre d'éléments de 'PointsImportants' - 1 FAIRE
    POUR chaque Colonne allant de 0 à Le nombre d'éléments de 'PointsImportants' - 1 FAIRE
      SI PointsImportants[Ligne] est différent de PointsImportants[Colonne] ALORS
        Trajet CalculTrajet <- Nouveau Trajet()
        DijkstraAlgo(PointsImportants[Ligne], PointsImportants[Colonne], map, CalculTrajet)
        TabTrajet[Ligne, Colonne] <- CalculTrajet
        CalculTrajet <- Null
      SINON
        TabTrajet[Ligne, Colonne] <- Null
    FIN SI
  FIN POUR
FIN POUR

  // Calcul des trajets entre le point de départ et tous les autres points
  Tableau de Trajet TabTrajetEntreDepartPoint <- Nouveau Tableau de Trajet[Le nombre d'éléments de
'PointsImportants']
  POUR chaque Ligne allant de 0 à Le nombre d'éléments de 'PointsImportants' - 1 FAIRE
    Trajet CalculTrajet <- Nouveau Trajet()
    DijkstraAlgo(PointDeDepart, PointsImportants[Ligne], map, CalculTrajet)
    TabTrajetEntreDepartPoint[Ligne] <- CalculTrajet
    CalculTrajet <- Null
  FIN POUR

  TrajetFinal <- ChoixDuTrajet(PointsImportants, TabTrajetEntreDepartPoint, TabTrajet, PointDeDepart)

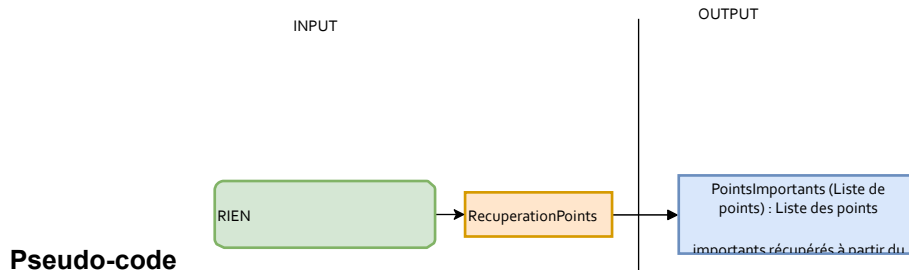
  map.AfficheMapEtTrajet(TrajetFinal)
FIN
```


2. RecuperationPoints



Valeur ajouté : Récupération des points importants à partir d'un fichier CSV.

Service fonctionnel



Pseudo-code

```
VARIABLE
    Tableau d'Entier TabPoint
    Entier ligne
    Liste de Point PointsImportants
    Objet StreamReader reader
    Chaîne line
    Tableau de Chaîne values

DEBUT
    TabPoint <- Nouveau Tableau d'Entier[14, 4]
    ligne <- 0
    PointsImportants <- Nouvelle Liste de Point
    reader <- Nouveau StreamReader(OuvrirFichier(@"..\..\..\Point.csv"))

    Tant que non la FinDuFichier Faire
        line <- reader.ReadLine()
        values <- line.Split(';')

        Pour colonne allant de 0 à Longueur(values) - 1 Faire
            TabPoint[ligne, colonne] <- Converti en entier values[colonne]
        Fin Pour

        Si TabPoint[ligne, 2] est différent de 30 OU TabPoint[ligne, 3] est différent de 0 Alors
            PointsImportants.Ajouter(Nouveau Point(TabPoint[ligne, 0], TabPoint[ligne, 1], TabPoint[ligne, 2], TabPoint[ligne, 3]))
        Fin Si

        ligne <- ligne + 1
    Fin Tant que

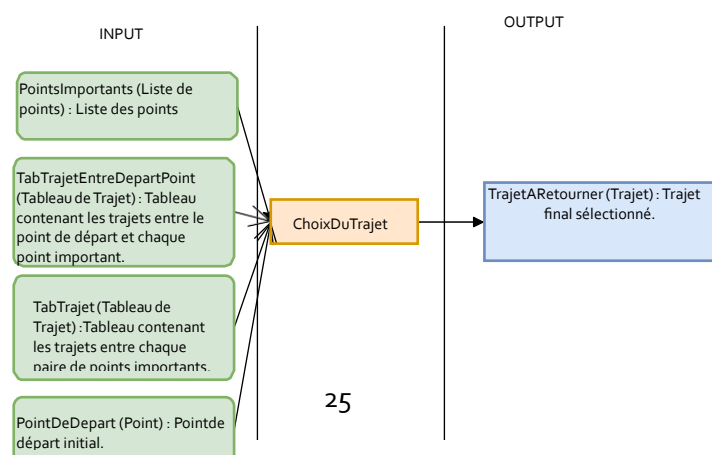
    Retourner PointsImportants
FIN
```

3. ChoixDuTrajet



Valeur ajouté : Sélection du trajet optimal pour parcourir tous les points importants en maximisant le score potentiel.

Service fonctionnel



Pseudo-code

```
VARIABLE
Entier nombrePointImportantAAteindre
Entier nombrePointImportantsAtteint
Booléen Continuer
Objet Trajet TrajetChoisiAAjouter
Objet Trajet TrajetEmprunter
Objet Trajet TrajetAReturner
Entier ScorePotentiel
Entier Score

DEBUT
nombrePointImportantAAteindre <- 0
nombrePointImportantsAtteint <- 0
Continuer <- vrai
TrajetChoisiAAjouter <- null
TrajetEmprunter <- null
TrajetAReturner <- null
ScorePotentiel <- null
Score <- 0

NombreDePointsAAteindre(PointsImportants, TabTrajetEntreDepartPoint, nombrePointImportantAAteindre)
ChoixEntrePointDepartEtUnPointImportant(PointsImportants, TabTrajetEntreDepartPoint, Score,
ScorePotentiel, TrajetChoisiAAjouter, TrajetEmprunter, TrajetAReturner, nombrePointImportantsAtteint)

SuppressionDesTrajetsContenantLesPointsParcourues(PointsImportants, TrajetEmprunter, TabTrajet)

Tant que Continuer est vrai Faire
    TrajetChoisiAAjouter <- null
    ScorePotentiel <- null

    ChoixDuTrajetEntre2PointsEnFonctionDuScore(PointsImportants, TrajetEmprunter, TabTrajet, Score,
    ScorePotentiel, TrajetChoisiAAjouter, TrajetAReturner, nombrePointImportantsAtteint)

    Si nombrePointImportantsAtteint est égal à nombrePointImportantAAteindre Alors
        Continuer <- faux
    Fin Si

    SuppressionDesTrajetsContenantLesPointsParcourues(PointsImportants, TrajetEmprunter, TabTrajet)
Fin Tant que

Retourner TrajetAReturner

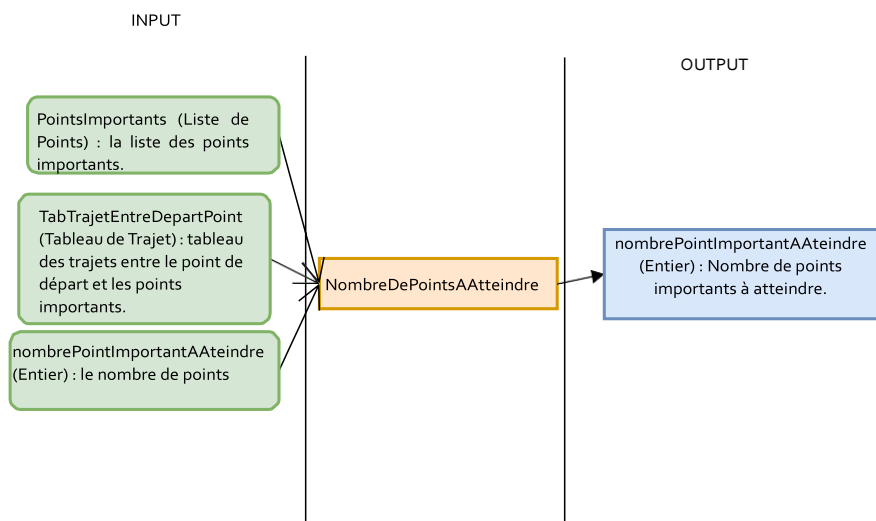
FIN
```

4. NombreDePointsAAteindre



Valeur ajoutée : Sélection du trajet optimal pour parcourir tous les points importants en maximisant le score potentiel.

Service fonctionnel



Pseudo-code

```
VARIABLES
  Liste de Point PointsImportants
  Tableau de Trajet TabTrajetEntreDepartPoint
  Entier nombrePointImportantAAteindre

DEBUT
  nombrePointImportantAAteindre <- 0

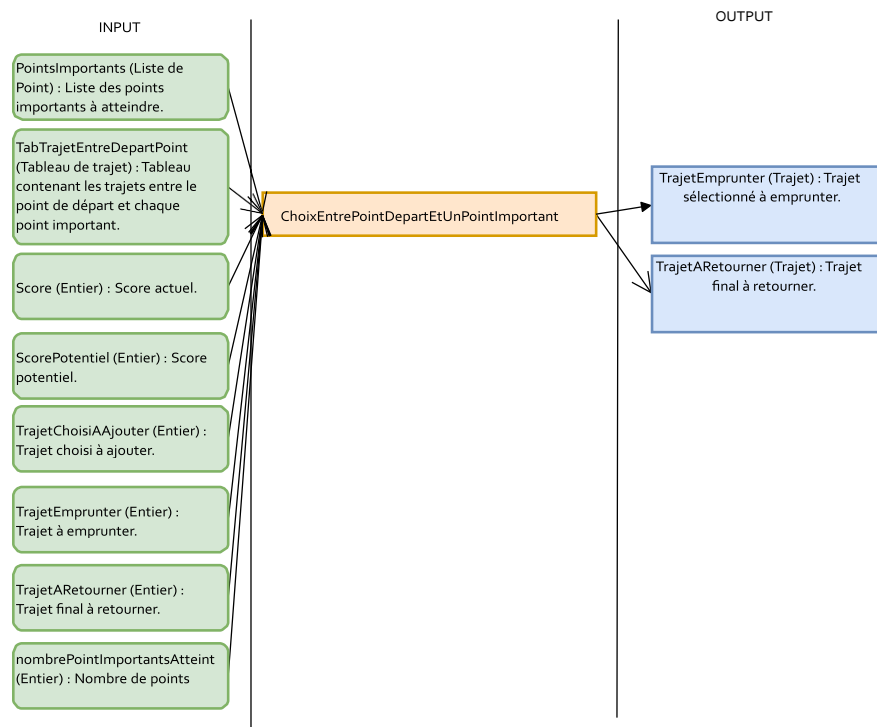
  Pour chaque lignes dans PointsImportants Faire
    Si GetUtile de PointArr de TabTrajetEntreDepartPoint[lignes] est égal à 1
  Alors      nombrePointImportantAAteindre <- nombrePointImportantAAteindre + 1
    Fin Si
  Fin Pour
FIN
```

5. ChoixEntrePointDepartEtUnPointImportant



Valeur ajoutée : Sélection du trajet optimal pour parcourir tous les points importants en maximisant le score potentiel.

Service fonctionnel



Pseudo-code

```

VARIABLES
  Liste de Point PointsImportants
  Tableau de Trajet TabTrajetEntreDepartPoint
  Entier Score
  Entier ScorePotentiel
  Trajet TrajetChoisiAAjouter
  Trajet TrajetEmprunter
  Trajet TrajetARetourner
  Entier nombrePointImportantsAtteint

DEBUT
  Pour chaque lignes dans PointsImportants Faire
    Si Score + (GetValeur de PointArr de TabTrajetEntreDepartPoint[lignes]) - (CoutTrajet de
    TabTrajetEntreDepartPoint[lignes]) > Score + ScorePotentiel Alors
      TrajetChoisiAAjouter <- TabTrajetEntreDepartPoint[lignes]
      ScorePotentiel <- (GetValeur de PointArr de TabTrajetEntreDepartPoint[lignes]) - (CoutTrajet de
      TabTrajetEntreDepartPoint[lignes])
    Fin Si
  Fin Pour

  Si TrajetChoisiAAjouter n'est pas null Alors
    AfficherTrajet de TrajetChoisiAAjouter
    TrajetEmprunter <- TrajetChoisiAAjouter
    TrajetARetourner <- TrajetEmprunter

    Si GetUtile de PointArr de TrajetChoisiAAjouter est égal à 1 Alors
      nombrePointImportantsAtteint <- nombrePointImportantsAtteint + 1
    Fin Si

    Score <- Score + ScorePotentiel
    Afficher("Score : " concaténé avec Score)
  Fin Si

FIN

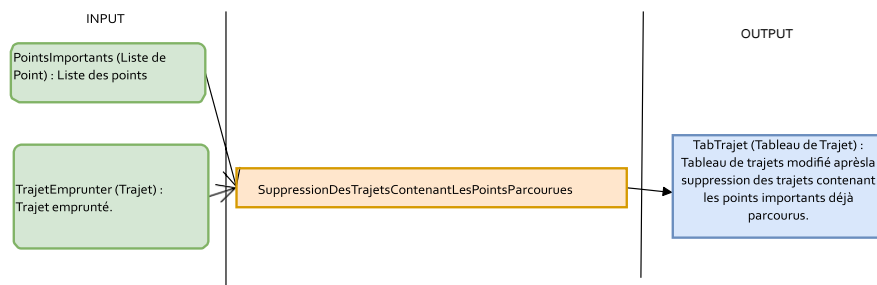
```

6. SuppressionDesTrajetsContenantLesPointsParcourses



Valeur ajouté : Suppression des trajets contenant les points déjà parcourus pour optimiser l'itinéraire.

Service fonctionnel



Pseudo-code

```

VARIABLE
  Liste de Point PointsImportants Trajet TrajetEmprunter
  Tableau de Trajet TabTrajet Entier lignes
  Entier colonnes

DEBUT
  Pour chaque lignes dans PointsImportants Faire
    Pour chaque colonnes dans PointsImportants Faire
      Si TabTrajet[lignes, colonnes] n'est pas nul ET TrajetEmprunter n'est pas nul ET
      GetSetX de PointArr de TabTrajet[lignes, colonnes] est égal à GetSetX de PointArr de
      TrajetEmprunter ET GetSetY de PointArr de TabTrajet[lignes, colonnes] est égal à GetSetY de
      PointArr de TrajetEmprunter Alors
        TabTrajet[lignes, colonnes] <- null
      Fin Si
    Fin Pour
  Fin Pour

FIN

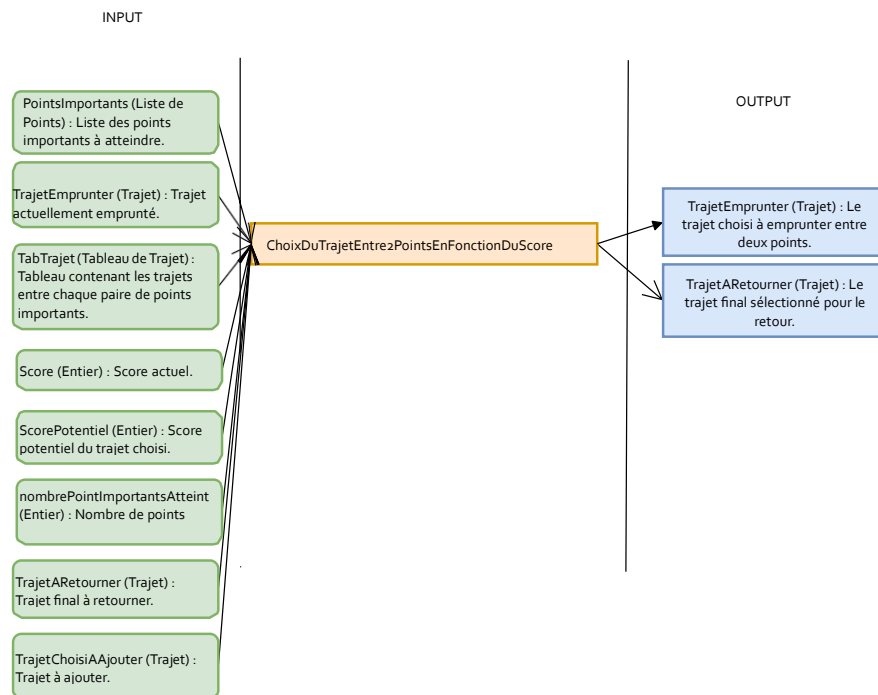
```

7. ChoixDuTrajetEntre2PointsEnFonctionDuScore



Valeur ajouté : Sélection d'un trajet optimal entre deux points en fonction du score potentiel.

Service fonctionnel



Pseudo-code

```
VARIABLES
    Liste de Point PointsImportants
    Trajet TrajetEmprunter
    Tableau de Trajet TabTrajet
    Entier Score
    Entier ScorePotentiel
    Trajet TrajetChoisiAAjouter
    Trajet TrajetARetourner
    Entier nombrePointImportantsAtteint

DEBUT
    POUR chaque lignes allant de 0 à PointsImportants.Count - 1 FAIRE
        POUR chaque colonnes allant de 0 à PointsImportants.Count - 1 FAIRE
            SI TrajetEmprunter n'est pas nul ET TabTrajet[lignes, colonnes] n'est pas nul ET
                TabTrajet[lignes, colonnes].PointDep.GetSetX est égal à TrajetEmprunter.PointArr.GetSetX
            ET
                TabTrajet[lignes, colonnes].PointDep.GetSetY est égal à TrajetEmprunter.PointArr.GetSetY
            ALORS
                SI Score + (TabTrajet[lignes, colonnes].PointArr.GetValeur - TabTrajet[lignes,
colonnes].CoutTrajet) > Score + ScorePotentiel OU ScorePotentiel est nul ALORS
                    TrajetChoisiAAjouter <- TabTrajet[lignes, colonnes]
                    ScorePotentiel <- TabTrajet[lignes, colonnes].PointArr.GetValeur - TabTrajet[lignes,
colonnes].CoutTrajet
                FIN SI
            FIN SI
        FIN POUR
    FIN POUR

    SI TrajetChoisiAAjouter n'est pas nul ET TrajetChoisiAAjouter n'est pas égal à TrajetEmprunter ALORS
        TrajetChoisiAAjouter.AfficheTrajet()
        TrajetEmprunter <- TrajetChoisiAAjouter

    SI TrajetARetourner n'est pas nul ALORS
        TrajetARetourner.PointArr <- TrajetEmprunter.PointArr
        POUR chaque p dans TrajetEmprunter.ListePointsParcourue FAIRE
            TrajetARetourner.ListePointsParcourue.Ajouter(p)
        FIN POUR
        // Ajoute le point d'arrivée car il n'est pas ajouté automatiquement
        TrajetARetourner.ListePointsParcourue.Ajouter(TrajetARetourner.PointArr)
    FIN SI

    SI TrajetChoisiAAjouter.PointArr.GetUtile est égal à 1 ALORS
        nombrePointImportantsAtteint <- nombrePointImportantsAtteint + 1
    FIN SI

    Score <- Score + ScorePotentiel
    afficher("Score : " + Score)
FIN SI
FIN
```

V. Conclusion

Pour conclure, ce programme arrive bien à trouver un chemin parmi tout ceux qui existe. La plus grosse plus grosse difficulté était le développement de l'algorithme de Dijkstra permettant de trouver le chemin le plus court entre deux points.