# Contexte

## Cahier des charges

Lors de ce mini-projet, nous devons appliquer l’algorithme de Warshall et de Dijkstra à des graphes.

Pour cela il nous faudra modéliser les graphes, les nœuds, les liaisons à l’aide de classes. On devra aussi rajouter des accesseurs, des setteurs et d’autres méthodes comme une méthode pour connaître le point dont la liaison à partir d’un point source est la plus petite, ou pour savoir s’il existe une liaison entre deux points.

## Ma solution

Il faut donc qu’on choisisse un langage de programmation Objet qui permet de définir facilement des graphes. J’ai opté pour le langage Python.

J’aurai pu choisir le langage Java que j’utilise régulièrement ou le C# que j’utilise en entreprise, cependant l’apprentissage d’un nouveau langage me paraissait intéressante, de plus, nous utiliserons ce langage lors de notre voyage en Irlande le semestre prochain.

Pour l’implémentation, j’ai décidé d’utiliser trois classes : Liaison, Nœud et Graphe. Dans ces classes, nous retrouverons des méthodes pour faciliter la manipulation de ces données.

Je définirai aussi des fonctions extérieures :

* Pour initialiser des graphes de test
* La fonction Warshall
* La fonction Dijkstra

# Implémentation

## Classes

Tout d’abord, je crée la classe Nœud, cette classe possède un nom et une liste de Liaison (dont on parlera plus tard). Elle possède des méthodes pour ajouter des liaisons, pour vérifier s’il existe ou non un nœud, une méthode pour récupérer un objet Liaison entre deux points.

On crée ensuite la classe Liaison, qui permet de lié deux sommets. Elle contient un poids positif et une instance de Nœud. Elle sera possédée par un nœud, le nœud qui possède sera le nœud d’origine et le Nœud de Liaison, le nœud de destination (le poids sera le poids de cette liaison). Si l’on veut simuler des liaisons non orientées, il faudra mettre une liaison dans les deus sens.

La dernière classe à être crée est la classe Graphe qui contient un nom et une liste de Nœud. On a plusieurs fonctions qui permettent de récupérer le nombre de lien, le nombre de nœud, les nœuds. On a aussi une fonction qui permet de vérifier si une liaison existe entre deux points. Deux fonction pour afficher les matrices dans la console, une sous forme de liste de nœud et l’autre sous forme de matrice. La dernière fonction est la fonction min\_chemin, elle prend en paramètre un nœud et cherche le nœud le plus proche (utilisé dans Dijkstra).

## Warshall

Pour Warshall, l’implémentation est assez simple, on parcourt d’abord le sommet central puis on regarde ses prédécesseurs puis les nœuds fils et on les relie ensemble.

|  |
| --- |
| def WARSHALL(graphe):  graphe\_ = graphe  nb\_noeud\_add = 0  for i in graphe\_.lst\_noeud:  lst\_noeuds = []  for j in graphe\_.lst\_noeud:  for k in graphe\_.lst\_noeud:  if(graphe\_.arc(j,i) and graphe\_.arc(i,k)):  j.add\_noeud\_oriente(k,1)  return graphe\_ |

On peut remarquer qu’on ajoute bien un nœud orienté lorsque les deux liens entre j,i et i,k existent.

## Dijkstra

Pour l’algorithme de Dijkstra, l’implémentation a été un peu plus complexe. On prend un paramètre un graphe, le nœud de départ et le nœud vers lequel on veut calculer le chemin le plus court.

On commence par initialiser deux listes, une liste qui contient les nœuds traités et une liste qui contient les nœuds non traités.

Ensuite l’algorithme commence, on continue de calculer tant qu’on a des nœuds dans la liste de nœuds non traités.

Le traitement est simple, on vérifie grâce à la fonction graphe.arc(noeud1, noeud2) si une liaison existe (renvoie false si non et la liaison si oui), si oui on regarde si le poids et le plus petit qu’on ai recensé dans le parcours. Si c’est aussi le cas, alors on stocke le poids, le nœud et le nœud père.

Lorsqu’on sort de la boucle, on ajoute ce nœud dans la liste des nœuds utilisés et on le supprime de la liste des restants.

Lorsqu’on le while est fini, on retourne les informations du nœud utilisé dont le chemin.

|  |
| --- |
| def Dijkstra(graphe, noeud\_start, noeud\_end):  #Nous allons représenter les noeuds sous forme d'un tableau à 3 dimensions  # 1 : Le noeud  # 2 : Le poids du chemin le plus court pour arriver jusque là  # 3 : Le chemin le plus court pour arriver jusque là  #Initialisation de l'algorithme  lst\_noeuds\_utilises = []  lst\_noeuds\_restants = []  for noeud in graphe.lst\_noeud:  if noeud != noeud\_start:  lst\_noeuds\_restants.append({"noeud" : noeud , "poids" : float('inf') , "chemin" : [] })  lst\_noeuds\_utilises.append({ "noeud" : noeud\_start, "poids" : 0 , "chemin" : [noeud\_start]})  #Début algorithme  #tant qu'il y a reste des chemins à calculer  while len(lst\_noeuds\_restants):  #On calcule le point le plus près  chemin\_min = float('inf')  noeud\_min = None  noeud\_min\_pere = None  chemin = []  for noeud\_utilise in lst\_noeuds\_utilises:  for noeud\_restant in lst\_noeuds\_restants:  #Si un arc existe et que (son poids + celui du chemin pour aller jusqu'au noeud\_utilise)  #est inférieur au poid minimum trouvé pour l'instant  result1 = graphe.arc(noeud\_utilise["noeud"] , noeud\_restant["noeud"])  if graphe.arc(noeud\_utilise["noeud"] , noeud\_restant["noeud"]) and noeud\_utilise["noeud"].get\_liaison(noeud\_restant["noeud"]).poids + noeud\_utilise["poids"] < chemin\_min:  #print noeud\_restant["noeud"].name  chemin\_min = noeud\_utilise["noeud"].get\_liaison(noeud\_restant["noeud"]).poids  noeud\_min = noeud\_restant  noeud\_min\_pere = noeud\_utilise  #On enlève le noeud des noeuds restants à explorer  lst\_noeuds\_restants.remove(noeud\_min)  #On met le bon poids ainsi que le bon chemin  noeud\_min["poids"] = noeud\_min\_pere["poids"] + noeud\_min\_pere["noeud"].get\_liaison(noeud\_min["noeud"]).poids  chemin = noeud\_min\_pere["chemin"]  chemin.append(noeud\_min["noeud"])  noeud\_min["chemin"] = chemin  #On ajoute le chemin aux chemins traités  lst\_noeuds\_utilises.append(noeud\_min)  #On a traité tous les chemins  for noeud in lst\_noeuds\_utilises:  if noeud["noeud"] == noeud\_end:  return noeud |