Cocheril Titouan
Salle Ivan
TP1

Iut-Bayonne
TD1

S2.02 - Exploration algorithmique d'un problème

Étape 2 : Recherche de plus courts chemins

Compte rendu de la deuxième partie :

Fonction « plus court chemin entre deux arrêts » en utilisant une heuristique dans un algorithme A* :

Code :

```
def getdistEstim(x,distance):
    return distance[indice_som(x)]

def Astar (depart,arrivee):
    start_time = time.time()
    #initialisation
    n = len(poids_bus)
    pred=[None]*n
    dist=[float("inf")]*n
    dist_estim=[float("inf")]*n #liste des distances estimée
    som=indice_som(depart)
    ferme = []#liste des sommets visités
    ouvert = []#liste des sommets dont on a calculé la distance
estimée
    dist[indice_som(depart)]=0 #on met la distance du sommet de
depart a 0
    ferme.append(depart)
    while nom(som) != arrivee: #tant que le sommets actuel n'est pas
le sommet d'arrivée
        if voisin(nom(som)) != []: #si le sommet a des voisins
                for i in voisin(nom(som)):#on estime les longeurs de
chaque voisins du sommets actuel
                    if i not in ferme:# si le voisin n'est pas dans
la liste des sommets fermé
                        if i not in ouvert or
dist[indice_som(i)]>dist[som] + poids_bus[som][indice_som(i)]: #si
on ne l'a pas encore ouvert(calcul de dist_estim) ou si le nouveau
chemin est meilleur que l'ancien
                            dist[indice_som(i)] = dist[som] +
poids_bus[som][indice_som(i)] #on met a jour la distance pour aller
a i avec la distance du sommet actuel
                            dist_estim[indice_som(i)] =
dist[indice_som(i)] + distarrets(i,arrivee) # on calcul la distance
estimé avec la distance du sommets actuel + la distance pour aller
du sommet actuel au voisin + la distance du voisin au somemt
d'arrivée
                            ouvert.append(i) # on ajoute ce sommet
voisin aux sommets ouverts
                            pred[indice_som(i)] = nom(som) #on met
le sommet actuel comme predecesseur du voisin
                # on prend le meilleur nouveau sommet dans la liste
des sommets ouverts
                ouvert=sorted(ouvert,key=lambda x:
getdistEstim(x,dist_estim),reverse=True) # on tri la liste des
```

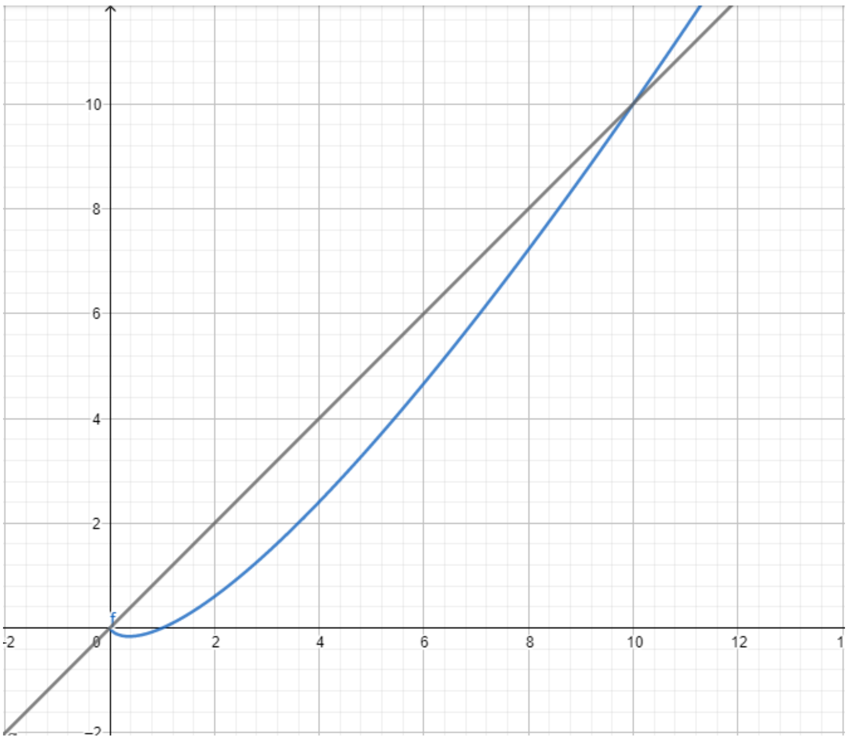
```
sommets ovuerts par leurs distances estimées de la plus grande a la
plus petite
                som = indice_som(ouvert.pop()) # on definie le
sommets de liste comme
                ferme.append(nom(som))
        # cas d'une impasse
        else: # on elimine cette possibilté de chemin et on reviens
au sommet precedent
            ferme.append(nom(som))
            som = indice_som(pred[som])
    #remplissage de la liste des predecesseurs du sommets arrivee
    liste = []
    liste.append(arrivee)
    a = arrivee
    while a != depart:
        liste.insert(0,pred[indice_som(a)])
        a = pred[indice_som(a)]
    print("--- %s seconds ---" % (time.time() - start_time))
    return(liste,dist[indice_som(arrivee)])
```

Explication de l’algorithme:

L’algorithme “A*”, contrairement aux algorithmes vus précédemment ne cherche pas à trouver les chemins les plus court vers chacun des sommets mais directement le chemin le plus court du sommet de départ jusqu’au sommet d’arrivée.
Il est assez proche de l’algorithme de Dijkstra car il choisit chaque fois le meilleur sommet, pour cela il utilise une fonction dite heuristique.
Pour notre part la fonction heuristique utilisée est la distance géographique entre un sommet et le sommet d'arrivée. Pour que le chemin trouvé soit bien le meilleur, cette fonction heuristique doit être “admissible” ce qui signifie qu’elle ne surestime pas la distance calculée. L'algorithme se place sur le sommet de départ et ajoute le sommet voisin à “liste ouverte”, la liste des sommets considérés comme ouvert car on a calculé leur distance estimé grâce a la formule : $F(n) = G(n) + H(n)$ ou $F(n)$ donne la distance estimée de n, G la distance du sommet actuel vers le sommet n et $H(n)$ la distance heuristique de n.
Chaque fois qu’il veut choisir le prochain sommet, on prend le sommet de la liste ouverte qui est triée dans l’ordre décroissant de distance estimée donc on prend le sommet avec la plus petite distance estimé et on l’enlève pour l’ajouter à la “liste fermé”, la liste des sommets déjà visité.
Chaque fois que le nouveau meilleur sommet est choisi, dans le cas où c’est un voisin du sommet actuel et qu’il n’est pas déjà ouvert ou qu’il améliore l’ancien chemin du sommet actuel vers lui, on donne pour prédécesseur au nouveau meilleur sommet le sommet actuel et on lui attribue la nouvelle distance calculée à partir du sommet actuel.
On répète cette opération jusqu’a que le sommet actuel soit le sommet d’arrivée

On peut maintenant retrouver facilement le chemin en ajoutant les prédécesseurs de chaque sommet en partant de l’arrivée jusqu’au départ.
La distance finale est la distance du sommet d’arrivée

Amelioration de dijkstra:
L’algorithme de Dijkstra utilise une liste dont ont récupérer le sommet
Grace a une fonction “exctract_min()” qui parcoure toute la liste donc sa complexité est de O(n).
Pour améliorer la complexité de l’algorithme de Dijkstra nous avons utilisé une file de priorité. Pour cela nous avons utilisé une simple liste qui, chaque fois que des nouveaux éléments sont ajoutés est triée (en fonction de la distance) pour que le premier élément de cette liste soit le meilleur nouveau sommet
L’algorithme de tri utilisée est Timsort dont la complexité moyenne est O(n*log(n)). L’algorithme Timsort est plus efficace dans notre cas car à l’inverse de la fonction “extract_min()”, on ne parcoure pas toute la liste.
Grace au tri, la liste des sommets à visite est déjà ordonnée et à chaque nouveau sommet, on obtient les distances des voisins du sommet actuel.
Or il n’y a jamais plus de 10 voisins, donc pas plus de 10 nouvelles valeurs à trier, nous sommes donc dans le cas ou n appartient à l’intervalle [0,10]
Dans ce cas la complexité de Timsort est inferieure à celle de la fonction extract_min().



Timsort
Extract_min()

<u>Comparer l’efficacité des différents algorithmes :</u>

On a utilisé dans la bibliothèque time la fonction time ()
Qui va afficher le temps d’exécution
On a testé le temps d’exécution des programmes

```
start_time = time.time()
/programme/
print("--- %s seconds ---" % (time.time() - start_time))
```

Test avec NOVE comme arrêt de départ et AGUI comme arrêt d’arrivé

<table>
<tr><th>Algorithme</th><th>Temps d’exécution</th><th>Retour d’exécution</th></tr>
<tr><td>Floyd Warshall</td><td>9.99 secondes</td><td rowspan="5">['NOVE', 'CANT', 'TREY', 'PLAT', 'PCAS', 'OCEAN', 'COTE', 'CAMA', 'MTRS', 'CITA', 'MAUB', 'GRBY2', 'ECHM', 'HDVBAY', 'BASQ-PDB', 'PAUL', 'STLE', 'LACH', 'MARO', 'VILL', 'BEYR', 'UNION', 'LEMB', 'BRNM', 'MANG', 'LOUI', 'MOUR', 'CHSN', 'AGUI', 11444.513027872961]</td></tr>
<tr><td>Bellman</td><td>1.036 secondes</td></tr>
<tr><td>Dijkstra</td><td>1.374 secondes</td></tr>
<tr><td>Dijkstra2 (avec queue de priorité)</td><td>0.956 secondes</td></tr>
<tr><td>A*
(avec queue de priorité)</td><td>0.013 secondes</td></tr>
</table>

Complexité de l'algorithme de Floyd Warshall :

On a séparé l'algorithme de la fonction pour qu'il s'exécute qu'une seule fois. La création des matrices prédécesseurs, distances et l'application de l'algorithme de Floyd Warshall pour créer la matrice finale s'exécute dès le début du programme. Ensuite on a une fonction qui récupère depuis la matrice final le chemin le plus court entre deux arrêts.
La complexité moyenne de l'algorithme de Floyd Warshall est de O(N^3)
Il y a trois itérations imbriquées c'est-à-dire que la première boucle itère n fois la deuxième qui itère n fois la troisième boucle qui itère n fois.
Avec n pour nombre d'arrêts.

Complexité de Bellman :

Il y a au plus N itérations, chacune d'entre elles consistant à regarder les prédécesseurs de tous les sommets, c'est-à-dire exactement les M arcs. Avec une liste d'adjacence comme structure de données du graphe, on a une complexité de O(N.M)

Complexité de Djikstra :

Lorsque la structure de données utilisé est une liste adjacente simple la complexité globale de l'algorithme est de O(N^2)
En utilisant une structure de données plus complexe comme la queue de priorité il est possible d'améliorer l'algorithme avec une complexité de O(M.log(N))

Complexité de A* :

Dans cette algorithme le pire cas, serait de visite tous les nœuds N.
La complexité globale de l'algorithme est de O(N) où N est le nombre de nœuds explorés.
Pour respecter cette complexité il faut que l'heuristique soit admissible.

Codes Complet :

```
def Belmann(arret_dep,arret_arriv):

    #Création d'un dictionnaire contenant pour clé le nom des arrêts, pour première valeur
    #la distance entre les arrêts en partant du premier arrêt
    #comme deuxième valeur le prédécesseur
    #On attribuera à la valeur de départ la distance 0
    dist_pred={arrets: [float('inf'), None] for arrets in noms_arrets}
    dist_pred[arret_dep][0] = 0

    #Fonction de relachement d'un arrêt
    def relachement(a, b):
        arret1=a
        arret2=b
        if dist_pred[arret2][0] > dist_pred[arret1][0] + distarc(arret1, arret2):
            dist_pred[arret2][0] = dist_pred[arret1][0] + distarrets(arret1, arret2)
            dist_pred[arret2][1] = arret1

    #Boucle allant de 0 à 464 (autant de tours de boucles que d'arrêts)
    for i in range(0, len(poids_bus)-1):
        #Boucle qui parcours tous les arrêts possible
        for j in noms_arrets:
            #Boucle qui parcours tous les voisins de l'arrêt traité
            for k in voisin(j):
                relachement(k, j)

    #Création de la matrice résultat
    liste_result=[arret_arriv]
    #Ajout du prédecesseur de l'arrêt d'arrivé
    pred_tmp = dist_pred[arret_arriv][1]
    liste_result.insert(0,pred_tmp)
    #Boucle qui va ajouter tous les arrêts du chemin entre l'arrêt de départ et l'arrêt d'arrivée
    while pred_tmp!=arret_dep:
        pred_tmp=dist_pred[pred_tmp][1]
        liste_result.insert(0,pred_tmp)
```

```
    #Ajout de la distance à la liste résultat
    liste_result.append(dist_pred[arret_arriv][0])
    return liste_result

#print(Belmann('NOVE','AGUI'))
```

```
def exctract_min(sommets,distance):
    min = sommets[0]
    #print(min)
    #print(indice_som(min))
    for i in range(len(sommets)):
       # print(dist[indice_som(min)])
       # print(L[i])
       # print(indice_som(L[i]))
       # print(indice_som(min))
       if distance[indice_som(sommets[i])]<distance[indice_som(min)]:
           min = sommets[i]
    return min


def dijkstra(depart,arrivee):
    n = len(poids_bus)
    pred=[None]*n
    dist=[float("inf")]*n
    a_traiter=noms_arrets.copy()
    som=indice_som(depart)
    dist[indice_som(depart)]=0
    while a_traiter != []:
        for i in voisin(nom(som)):
            if i in a_traiter:
                if dist[som] + poids_bus[som][indice_som(i)] < dist[indice_som(i)]:
                    if pred[som] != i:# evite les boucles
                            dist[indice_som(i)] = dist[som] + poids_bus[som][indice_som(i)]
                            pred[indice_som(i)]=nom(som)
        som = indice_som(exctract_min(a_traiter, dist))
```

```
        a_traiter.remove(nom(som))
    liste = []
    liste.append(arrivee)
    a = arrivee
    while a != depart:
        liste.insert(0,pred[indice_som(a)])
        a = pred[indice_som(a)]
    return liste,dist[indice_som(arrivee)]

#print(dijkstra('NOVE','AGUI'))

def min_estim(traité,dist_estim):
   min = indice_som(traité[0])
   for i in range(len(traité)):
           if dist_estim[indice_som(traité[0])] < dist_estim[min]:
               min = indice_som(traité[0])
   return nom(min) #le nom du sommet de distance estimée minimale

def successeur(sommet):
    suc=[]
    n = len(mat_bus)
    for i in range(n):
        if mat_bus[indice_som(sommet)][i]==1:
            suc.append(nom(i))
    return(suc)

def getdistEstim(x,distance):
    return distance[indice_som(x)]

def Astar (depart,arrivee):
    n = len(poids_bus)
    pred=[None]*n
    pred[indice_som(depart)]=depart
    dist=[float("inf")]*n
    dist_estim=[float("inf")]*n
    som=indice_som(depart)
    som_visite = []#liste des sommets visités
    traité = []#liste des sommets qui sont traité pour trouver le nouveaux sommet de meilleur chemin
    dist[indice_som(depart)]=0
    som_visite.append(depart)
```

```
    while nom(som) != arrivee: #tant que le sommets actuel n'est pas le sommet d'arrivée
        if voisin(nom(som)) != []:
                for i in voisin(nom(som)):#on estime les longeurs de chaque successeurs du sommets actuel
                    if i not in som_visite:
                        if i not in traité or dist[indice_som(i)]>dist[som] + poids_bus[som][indice_som(i)]: #si on ne l'a pas encore traité ou si le nouveau chemin est meilleur
                            dist[indice_som(i)] = dist[som] + poids_bus[som][indice_som(i)]
                            dist_estim[indice_som(i)] = dist[indice_som(i)] + distarrets(i,arrivee)
                            traité.append(i)
                            pred[indice_som(i)] = nom(som)
                # on prend le nouveau meilleur sommet et on lui attribu pour predecesseur le sommet actuel
                traité=sorted(traité,key=lambda x: getdistEstim(x,dist_estim),reverse=True)
                som = indice_som(traité.pop())
                som_visite.append(nom(som))
        # cas d'une impasse
        else: # on elimine cette possibilté de chemin et on reviens au sommet precedent
            som_visite.append(nom(som))
            som = indice_som(pred[som])
    #remplissage de la liste des predecesseurs du sommets arrivee
    liste = []
    liste.append(arrivee)
    a = arrivee
    while a != depart:
        liste.insert(0,pred[indice_som(a)])
        a = pred[indice_som(a)]
    return(liste,dist[indice_som(arrivee)])



#print(Astar('NOVE','AGUI'))

def getdist(x,distance):
    return distance[indice_som(x)]

def dijkstra2(depart,arrivee):
    n = len(poids_bus)
    pred=[None]*n
    dist=[float("inf")]*n
    dist[indice_som(depart)]=0
    a_traiter=noms_arrets.copy()
    som=indice_som(depart)
```

```
    while a_traiter != []:
        for i in  voisin(nom(som)):
            if i in a_traiter:
                if dist[som] + poids_bus[som][indice_som(i)] < dist[indice_som(i)]:
                    if pred[som] != i:# evite les boucles
                        dist[indice_som(i)] = dist[som] + poids_bus[som][indice_som(i)]
                        pred[indice_som(i)]=nom(som)
        a_traiter=sorted(a_traiter,key=lambda x: getdist(x,dist),reverse=True)
        som = indice_som(a_traiter.pop())#on enleve le sommet de meilleur chemin en haut de la queue de priorité
    liste = []
    liste.append(arrivee)
    a = arrivee
    while a != depart:
        liste.insert(0,pred[indice_som(a)])
        a = pred[indice_som(a)]
    return liste,dist[indice_som(arrivee)]

#print(dijkstra2('NOVE','AGUI'))

#Matrice des prédecésseurs
mat_pred = []
for i in (noms_arrets):
  ligne = []
  for j in (noms_arrets):
    if j in voisin(i):
      ligne.append(indice_som(i))
    else:
      ligne.append(0)
  mat_pred.append(ligne)
    #print(mat_pred)


#Matrice des distances
mat_dist=[]
for i in (noms_arrets):
    ligne = []
    for j in (noms_arrets):
        if distarc(i,j) < 1:
            ligne.append(float("inf"))
        else:
```

```
            ligne.append(distarc(i,j))
    mat_dist.append(ligne)
    #print(mat_dist)



#Application de l'algorithme de Floyd Warshall
for t in range (len(noms_arrets)):
    for u in range (len(noms_arrets)):
        for v in range (len(noms_arrets)):
            nouvellesDistances = mat_dist[u][t] + mat_dist[t][v]
            if nouvellesDistances < mat_dist[u][v]:
                mat_dist[u][v] = nouvellesDistances
                mat_pred[u][v] = mat_pred[t][v]



#FLOYD WARSHALL
def FloydWarshall(arret_dep,arret_arriv):
#Extraction du chemin le plus court
    source = indice_som(arret_dep)
    destination = indice_som(arret_arriv)
    pile_result = [arret_arriv]
    distance_final=0
    while nom(destination)!=arret_dep:
        x=mat_pred[source][destination]
        pile_result.append(nom(x))
        y=mat_dist[x][destination]
        distance_final=distance_final+y
        destination=x

#Création de la matrice résultat
    mat_result = []
    while pile_result!=[]:
        s=pile_result.pop(len(pile_result)-1)
        mat_result.append(s)
    mat_result.append(distance_final)
    return mat_result

#print(FloydWarshall('NOVE','AGUI'))
```