Informatique — MPSI/PCSI

Correction du TP n° 13

15 Algorithme de Dijkstra

Exercice 13.1

1. On commence par faire une copie profonde du graphe `G`, copie à laquelle ont peut affecter des caractéristiques non intrinsèques au graphe de départ.

```
def init_parcours(G,r):
    """Initialise le parcours du graphe"""
    P = copy.deepcopy(G)
    for s in P:
        P[s]['e']=float('inf')
        P[s]['vu']=0
    P[r]['e']=0
    P[r]['vu']=1
    return P
```

2. On recherche le sommet d'étiquette minimum par la méthode du candidat.

```
def plus_faible(G,S):
    """Retourne de le sommet de la liste S d'étiquette la plus faible
    Donnes :
        G : graphe étiquette
        S : liste de sommets
    Retour :
        S[indmin],min : sommet d'étiquette minimum"""
    smin=S[0]
    min=G[smin]['e']
    for s in S:
        if G[s]['e']<min:
            min=G[s]['e']
            smin=s
    return smin
```

3. Pour appliquer l'algorithme de Dijkstra, on commence par initialiser le parcours, puis on définit la liste Nv des sommets non visités. Tant que Nv est non vide, on choisit alors le sujet d'étiquette la plus faible et on applique l'algorithme.

```
def dijkstra(G,r):
    """Calcule la distance des sommets de G àr par l'algorithme de Dijkstra
    Parametres :
        G : graphe
        r : sommet racine
    Retour :
        P : graphe etiquette avec les distances a r"""
    P = init_parcours(G,r)
    Nv=[s for s in P]
    while Nv:
        u=plus_faible(P,Nv)
        Nv.remove(u)
        P[u]['vu']=1
        for v in P[u]['adj']:
            if P[v]['vu']==0 and P[u]['e']+P[u]['adj'][v]<P[v]['e']:
                P[v]['e']=P[u]['e']+P[u]['adj'][v]
    return P
```

Exercice 13.2
On initialise le graphe et on applique l'algorithme de Dijkstra, en modifiant condition d'arrêt pour terminer dès que le sommet `t` est atteint. A chaque étape de l'algorithme, on attribue aussi à tout sommet dont on modifie l'étiquette son sommet parent : cela sera utile pour "remonter" les chemins les plus courts à la section suivante.

```
def distance(G,r,t):
    """Calcule la distance du sommet r a t par l'algorithme de Dijkstra
    Parametres :
        G : graphe
        r : sommet racine
        t : sommet arrive
    Retour :
        L,P[t]['e'] : chemin le plus court de r a t sous forme de liste et longueur du chemin"""
    # Initialisation
    P = init_parcours(G,r)
    Nv=[s for s in P]
    # Algorithme de Dijkstra jusqu'à atteindre t
    while t in Nv:
        u=plus_faible(P,Nv)
        Nv.remove(u)
        P[u]['vu']=1
        for v in P[u]['adj']:
            if P[v]['vu']==0 and P[u]['e']+P[u]['adj'][v]<P[v]['e']:
                P[v]['e']=P[u]['e']+P[u]['adj'][v]
                # Ne pas oublier de préciser le sommet parent
                P[v]['parent']=u
    # Remontée du chemin
    s=t
    L=[s]
    while s!=r:
        s=P[s]['parent']
        L.append(s)
    L.reverse()
    return L,P[t]['e']
```

Exercice 13.3
On reprend le code précédent, en initialisant la couleur des sommets et en modifiant celle-ci chaque fois qu'un sommet est vu (il passe en bleu) ou complètement traité (il passe en rouge). Pour la suite, il est utile d'introduire des paramètre facultatifs *Etapes* (détermine si on doit attendre à chaque étape ou non), *etiquettes* (affiche l'etiquette de chaque sommet ou non) et d (taille des sommets) pour des raisons de lisibilité.

```
def distance_col(G,r,t,etapes=True,etiquettes=True,d=20**2):
    # Initialisation
    P = init_parcours(G,r)
    for s in P:
        P[s]['coul']=JAUNE
    P[r]['coul']=ROUGE
    if etapes==True:
        trace_graphe_adj_cc(P,d,etiquettes)
        input('Attente...')
    Nv=[s for s in P]
    # Algorithme de Dijkstra jusqu'à atteindre t
    while t in Nv:
        u=plus_faible(P,Nv)
        Nv.remove(u)
        P[u]['coul']=ROUGE
        for v in P[u]['adj']:
```

```
            if P[u]['e']+P[u]['adj'][v]<P[v]['e']:
                P[v]['e']=P[u]['e']+P[u]['adj'][v]
                P[v]['parent']=u
                P[v]['coul']=BLEU
        if etapes==True:
            trace_graphe_adj_cc(P,d,etiquettes)
            input('Attente...')
    #Affichage final
    trace_graphe_adj_cc(P,d,etiquettes)
    # Remontée du chemin
    s=t
    L=[s]
    while s!=r:
        s=P[s]['parent']
        L.append(s)
    L.reverse()
    return L,P[t]['e']
```

16 Algorithme A*

Exercice 13.4

1. La fonction prend entrée un graphe tel qu'à chaque sommet est associé une clef `XY` contenant un couple de coordonnée entière et retourne la distance de Manhattan entre deux sommets.

```
def Manhattan(G,s,t):
    """Distance de Manhattan entre les sommets s et t
    Paramètres :
        G : graphe ou chaque sommet est muni des coordonnées XY
        s,t : sommets"""
    return abs(G[s]['XY'][0]-G[t]['XY'][0])+abs(G[s]['XY'][1]-G[t]['XY'][1])
```

2. On associe à chaque sommet une nouvelle clef `h` dont le contenu est l'heuristique définie par la fonction `f` et on retourne le graphe ainsi complété.

```
def heuristique(G,f):
    for s in G:
        G[s]['h']=f(s)
    return G
```

3. On utilise la fonction précédente puis modifie le graphe `P` en remplaçant la distance associée à chaque arête par une pseudo-distance.

```
def pseudo_distance(G,f):
    """Remplace la distance associée àchaque arete par la pseudo-distance tenant compte de l
    Paramètres :
        G : graphe pondéré
        f : fonction heuristique
        C : nombre flottant"""
    heuristique(G,f)
    for s in G:
        for t in G[s]['adj']:
            G[s]['adj'][t]=G[s]['adj'][t]+(G[t]['h']-G[s]['h'])
```

4. On commence par initialiser une copie du graphe avec `init_parcours`, puis utilise la fonction précédente, puis on lance l'algorithme de Dijkstra sur le graphe modifié.

   On recalcule bien la distance initiale à partir de la pseudo-distance avant de la retourner.

```
def Astar(G,f,r,t):
    """Calcule le plus court chemin entre les sommets r et t àl'aide de l'algorithme A*

    Paramètres :
```

```
        G : graphe pondere chaque sommet etant muni des coordonnees XY
        s,t : sommets
        f : heuristique
        C : flottant"""
    # Initialise le parcours et calcule les pseudodistances
    P=init_parcours(G,r)
    pseudo_distance(P,f)
    # Dijkstra avec les pseudo-distances
    L,e=distance(P,r,t)
    # Attention, on retourne bien la distance en utilisant les distances initiales
    return L,e-(f(t)-f(r))
```

5. On reprend le code précédent en appelant cette fois la fonction `distance_col`.

```
def Astar_col(G,f,r,t,etiquettes=True,etapes=True,d=20**2):
    """Calcule le plus court chemin entre les sommets s et t àl'aide de l'algorithme A*
    et affiche l'etat du graphe a chaque etape

    Paramètres :
        G : graphe pondere chaque sommet etant muni des coordonnees XY
        s,t : sommets
        f : heuristique
        C : flottant"""
    # Initialise le parcours et calcule les pseudodistances
    P=init_parcours(G,r)
    pseudo_distance(P,f)
    # Dijkstra avec les pseudo-distances
    L,e=distance_col(P,r,t,etiquettes,etapes,d)
    # Attention, on retourne bien la distance en utilisant les distances initiales
    return L,e-(f(t)-f(r))
```

6. On doit se limiter à $C \in]0;1[$ pour veiller à ce que les pseudo-distances restent positives. Plus la valeur de C est proche de 1, plus le poids donné à l'heuristique est important.

   On rappelle que l'heuristique est la fonction qui à un sommet s associée $0,5 \times Manhattan(G2, s,' A55')$. On peut par exemple définir une fonction associant à s cette valeur de la manière suivante.

```
def da(s):
    return Manhattan(G,s,'A55')
```

   puis appeler la fonction `Astar` en donnant en paramètre $C \times f$, où C est à remplacer par sa valeur.

   Une autre méthode est d'utiliser une fonction lambda en tapant par exemple en ligne de commande `Astar_col(G2,lambda s: 0.5*Manhattan(G2,s,'A55'),'A12','A55',etapes=False)`

   Les figures 1, 2 et 3 montrent l'état final du graphe selon si on utilise l'algorithme de Dijkstra ou l'algorithme A^* pour $C = 0,5$ et $C = 0,9$: le chemin optimal est obtenu d'autant plus rapidement que C est proche de 1.

Exercice 13.5

Il suffit de recopier le code contenu dans `graphe_routier.py` et de l'exécuter après avoir modifié le chemin d'accès passé en argument de la fonction `os.chdir`.

Pour des raisons de lisibilité, il est préférable de diminuer la valeur du paramètre d (diamètre des sommets) et de na pas afficher les noms des sommets. En tapant par exemple dans la console la commande `trace_graphe_adj(Gis,d=25,etiquettes=False)`, on obtient la figure 4.

Exercice 13.6

1. On peut traduire la formule mathématiques en fonction Python à l'aide du module `math` (ici renommé en `ma`), la latitude et la longitude du sommet s étant obtenues par `G[s]['XY'][0]` et `G[s]['XY'][1]`.
   Il faut être attentif au fait que ces latitudes et longitudes sont exprimées en degrés alors que les fonctions du module `math` prennent comme argument des angles en radians : il faut donc commencer par faire une conversion d'unité.

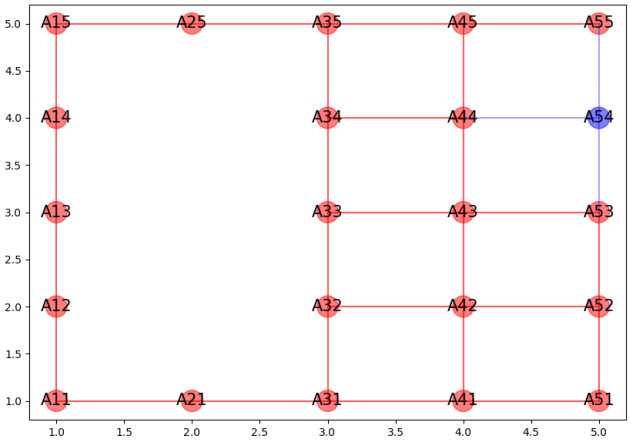

FIGURE 1 – Etat final du graphe après l'algorithme de Dijkstra

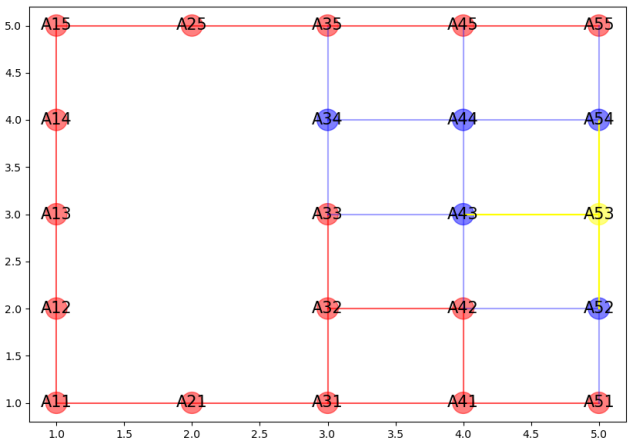

FIGURE 2 – Etat final du graphe après l'algorithme A^* avec $C = 0,5$

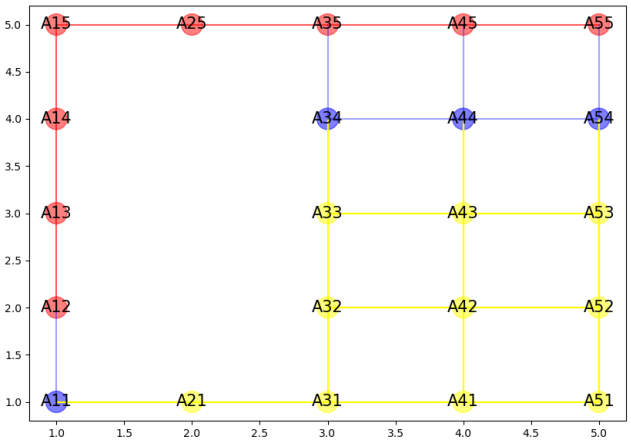

FIGURE 3 – Etat final du graphe après l'algorithme A^* avec $C = 0,9$

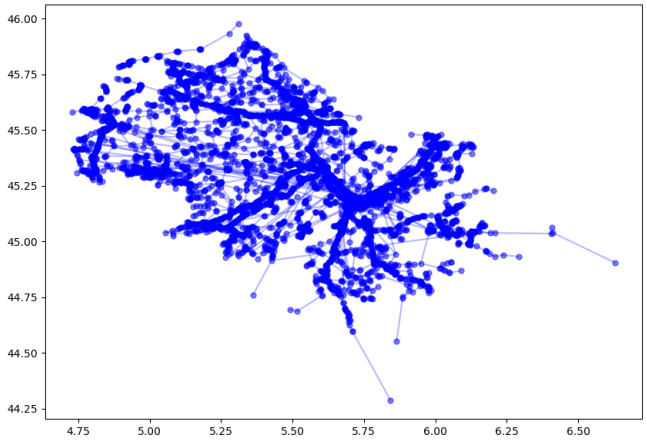

FIGURE 4 – Graphe associé au réseau routier de l'Isère

```
def euclidienne(G,s,t):
    """Distance Euclidienne entre les sommets s et t
    Parametres :
        G : graphe etiquette avec les coordonnees XY
        s,t : sommets"""
    pA,pB,lA,lB=G[s]['XY'][1]/180*ma.pi,G[t]['XY'][1]/180*ma.pi,\
        G[s]['XY'][0]/180*ma.pi,G[t]['XY'][0]/180*ma.pi
    angle=ma.acos(ma.sin(pA)*ma.sin(pB)+ma.cos(pA)*ma.cos(pB)\
                *ma.cos(lB-lA))
    return 6378137*angle
```

2. On peut par exemple exécuter en ligne de commande
   `distance(Gis,'B1122','B363')`
3. On exécute de même
   `Astar(Gis,lambda s: euclidienne(Gis,s,'B363'),'B1122','B363')`
   Les deux algorithmes retournent le même chemin le plus court entre les sommets $B1222$ et $B363$ et sa longueur (42334 m).
4. Pour obtenir un graphe lisible et éviter de détailler toutes les étapes, on peut taper en ligne de commande
   `distance_col(Gis,'B1122','B363',d=25,etiquettes=False,etapes=False)`
   puis

```
Astar_col(Gis,lambda s:euclidienne(Gis,s,'B363'),
'B1122','B363',d=25,etiquettes=False,etapes=False)
```

Les figures 5 et 6 montrent que l'algorithme A^* explore moins de sommets que Dijkstra.

* *
*

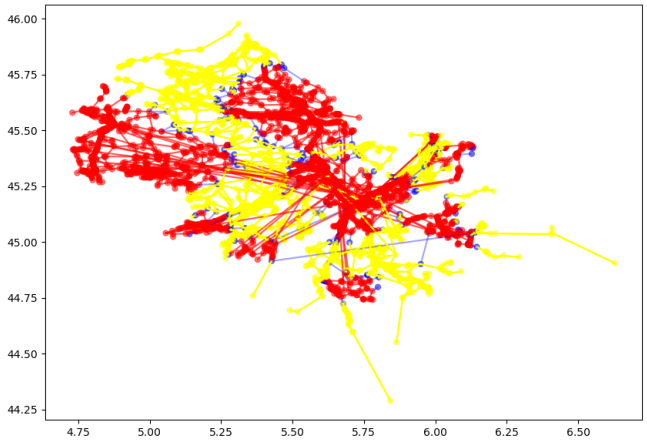

FIGURE 5 – Plus court chemin entre les sommets $B1222$ et $B363$ par l'algorithme de Dijkstra

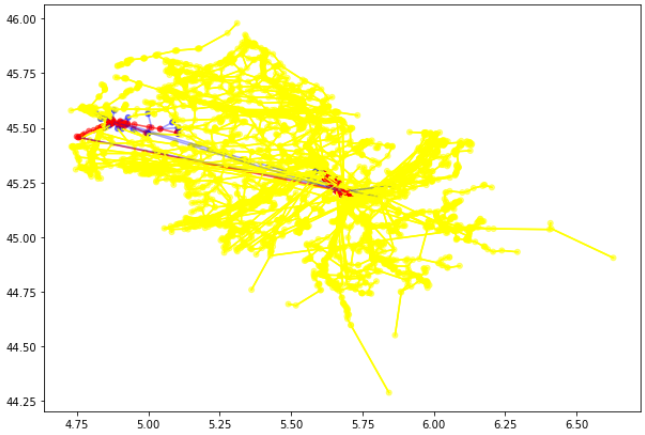

FIGURE 6 – Plus court chemin entre les sommets $B1222$ et $B363$ par l'algorithme A^*