Informatique — MPSI/PCSI, 2023–2024

Correction du TP n° 13

2 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. On appelle la fonction `initparcours` puis, à 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_col(G,r,t):
    # Initialisation
    P = init_parcours(G,r)
    for s in P:
        P[s]["coul"]=JAUNE
    P[r]['coul']=ROUGE
    trace_graphe_adj_cc(P)
    Nv=[s for s in P]
    # Algorithme de Dijkstra jusqu'à atteindre t
    while t in Nv:
        u,g=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
        trace_graphe_adj_cc(P)
    # Remontée du chemin
    s=t
    L=[s]
    while s!=r:
        s=P[s]['parent']
        L.append(s)
    L.reverse()
    return L,P[t]['e']
```

4. On réécrit la même fonction en modifiant condition d'arrêt pour terminer dès que le sommet `t` est atteint.
   Par ailleurs, au cours de l'exécution, on attribue à chaque sommet un sommet parent. Après avoir atteint l'arrivée, on retrouve le chemin de départ en remontant au parent de chaque sommet.

```
def distance(G,r,t):
    """Calcule la distance du sommet r a t par l'algorithme de Dijkstra
    Paramètres :
        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 chemi
    # Initialisation
    P = init_parcours(G,r)
    Nv=[s for s in P]
    # Algorithme de Dijkstra jusqu'à atteindre t
    while t in Nv:
        u,g=plus_faible(P,Nv)
        Nv.remove(u)
        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
    # Remontée du chemin
    s=t
    L=[s]
    while s!=r:
        s=P[s]['parent']
        L.append(s)
    L.reverse()
    return L,P[t]['e']
```

3 Algorithme A*

Exercice 13.2

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.

```python
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.

```python
def pseudo_distance(G,f,C):
    """Remplace la distance associée àchaque arête par la pseudo-distance tenant compte de l
    Paramètres :
        G : graphe pondéré
        f : fonction heuristique
        C : nombre flottant

    Retour :
        P : le graphe modifie"""
    heuristique(G,f)
    for s in G:
        for t in G[s]['adj']:
            G[s]['adj'][t]=G[s]['adj'][t]+C*(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.

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

    Paramètres :
        G : graphe pondéré chaque sommet étant muni des coordonnées XY
        s,t : sommets
        f : heuristique
        C : flottant"""
    # Initialise le parcours et calcule les pseudodistances
    P=init_parcours(G,r)
    pseudo_distance(P,f,C)
    # 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-C*(f(t)-f(r))
```

4 Visualisation des parcours

Exercice 13.3

1. On fait une simple variante de la fonction `distance` en affichant initialisant tous les sommets en couleur `JAUNE`. Ensuite, on modifie la couleur d'un sommet en `BLEU` chaque fois que son étiquette et modifiée et en `ROUGE` quand il est traité.

   On affiche le graphe à l'aide de la fonction `trace_graphe_adj_cc(G)` à chaque modification.

```python
def distance_col(G,r,t):
    # Intialisation
    P = init_parcours(G,r)
    for s in P:
        P[s]["coul"]=JAUNE
    P[r]['coul']=ROUGE
    trace_graphe_adj_cc(P)
    Nv=[s for s in P]
    # Algorithme de Dijkstra jusqu'à atteindre t
```

```
    while t in Nv:
        u,g=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
        trace_graphe_adj_cc(P)
    # Remontée du chemin
    s=t
    L=[s]
    while s!=r:
        s=P[s]['parent']
        L.append(s)
    L.reverse()
    return L,P[t]['e']
```

2. La fonction `Astar_col` appelle maintenant la fonction précédente sur le modèle de `Astar`.

```
def Astar_col(G,f,C,r,t):
    """Calcule le plus court chemin entre les sommets s et t àl'aide de l'algorithme A*\
    et affiche l'état du graphe a chaque étape

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

3. Pour tester l'algorithme $A*$, on commence par définir une fonction `da` qui calcule la distance d'un sommet `s` au sommet arrivé `A55` pour le graphe `G2`

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

Noter que pour être certain que les pseudo-distances restent positives, on doit avoir $0 < C < 1$.

L'exécution de `dijkstra_col(G2,'A21','A55'))` montre normalement que tous les sommets du graphe sont visités sauf $A45$ dont l'étiquette est modifiée mais qui n'est pas complètement traité.

L'algorithme $A*$ est d'autant plus efficace que C est proche de 1 : on voit que de moins en moins de sommet sont traités et que le chemin optimal est trouvé de plus en plus rapidement.

* *
*