TP (S2) 5 — Plus courts chemins & Parcours de graphes pondérés

1 Plus courts chemins ....
2 Parcours de graphes pondérés ..

Objectifs

- S'approprier les algorithmes de parcours de graphes pondérés (Dijkstra, et A*).
- Modifier les algorithmes de parcours de graphe pour répondre à de demandes spécifiques.

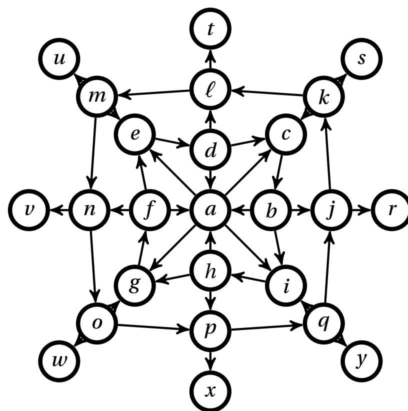
Fichier externe ?

OUI `ex2.py`, `ex2.py`, `ex4.py`, `ex5.py`, `ex6.py`, `ex6.py`, `ex7.py`, `ex8.py` *(présents dans le répertoire partagé de la classe)*

1. PLUS COURTS CHEMINS

Exercice 1 Distances dans un graphe non pondéré [Sol 1]

On considère le graphe G suivant.



1) Donner les longueurs des chemins (a,c,b,j,r) et $(a,e,d,\ell,m,n,o,p,q,j,r)$ dans G.

2) Soit γ un chemin dans G. Quel lien a-t-on entre la longueur de γ et le nombre de sommets qui le composent ?

3) En appliquant à la main le parcours BFS, déterminer des plus courts chemins et leurs distances associée entre a et r, puis entre n et c.

Exercice 2 Plus court chemin entre deux sommets d'un graphe non pondéré [Sol 2] Le fichier `ex2.py` contient le code de la fonction `bfs_dist(g, v)` vue en cours ainsi que le dictionnaire `gex1` codant le graphe de l'exercice 1.

Dans cet exercice, G sera un graphe codé par le dictionnaire g. Soient v et w deux sommets d'un graphe G codé par un dictionnaire g et soit un voisin u de w, on dit que w est le prédécesseur de u dans le parcours en largeur de G depuis v si, dans l'exécution de `bfs_dist(g,v)`, u n'était pas déjà visité lors de sa découverte comme voisin de w.

1) Modifier la fonction `bfs_dist` du cours pour définir une fonction `bfs_pred(g,v)` qui renvoie un dictionnaire des prédécesseurs lors d'un parcours en largeur d'un graphe `g` depuis un sommet `v`.

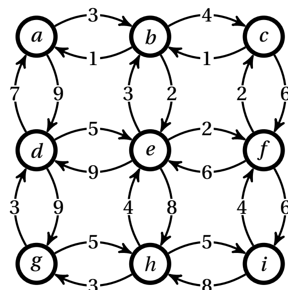
2) En déduire une fonction `shortest_path(g,u,v)` qui renvoie le couple formé de la distance et de la liste associée à un plus court chemin du sommet `u` au sommet `v` du graphe `G`.

3) Retrouver les résultats de la question 3 de l'exercice 1.

2. PARCOURS DE GRAPHES PONDÉRÉS

Lycée Michel MONTAIGNE – Bordeaux

Exercice 3 DIJKSTRA à la main [Sol 3] Construire, **à la main**, le dictionnaire d_a des distances de a à tous les sommets du graphe G_2 du cours.



On pourra présenter l'algorithme sous forme d'un tableau à l'allure ci-dessous :

Sommets visités	a	b	c	d	e	f	g	h	i
{}	0	∞	∞	∞	∞	∞	∞	∞	∞
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Exercice 4 Plus court chemin entre deux sommets d'un graphe pondéré [Sol 4] Le fichier `ex4.py` contient le code de la fonction `dijkstra(g, v_init)` vue en cours ainsi que le dictionnaire `gex3` codant le graphe de l'exercice 3.
Écrire une fonction `dijkstra_path(g,v_init,v_end)` qui reçoit un dictionnaire `g` codant un graphe G, un sommet de départ `v_init`, un sommet d'arrivée `v_end` et qui :

- lorsque `v_fin` n'est pas accessible depuis `v_init`, affiche un message l'indiquant,
- lorsque `v_fin` est accessible depuis `v_init`, renvoie le triplet `(N,d,C)` où `N` est le nombre de sommets qui ont été visités pour détecter un plus court chemin, `d` est la distance de `v_init` à `v_fin`, `C` est un meilleur chemin de `v_init` à `v_fin`.

```
>>> dijkstra_path(gex3, 'a','g')
(9, 16, ['a', 'b', 'e', 'h', 'g'])
```

Exercice 5 DIJKSTRA et matrice d'adjacence [Sol 5] Un graphe dont les sommets sont numérotés par des entiers de 0 à $N-1$ peut être représenté par sa matrice d'adjacence, *i.e.* le tableau `numpy` A où, pour tout $(i,j) \in [\![0, N-1]\!]^2$, $A[i,j]$ vaut le poids de l'arc de i à j si j est un voisin de i, $+\infty$ sinon. On trouvera dans le fichier `ex5.py` le code de la fonction `dijkstra_path` (cf exercice 4) ainsi que la matrice d'adjacence `A3` du graphe de l'exercice 3 (où le sommet `'a'` est numéroté `'0'`, le sommet `'b'` est numéroté `'1'`, *etc.*...).

Reprendre l'exercice précédent et écrire, puis tester avec `A3`, la fonction `dijkstra_path_adja(A, v_init,v_fin)` qui reçoit la matrice d'adjacence `A` d'un graphe et non plus le dictionnaire le représentant.

Exercice 6 Chemin dans un tableau d'entiers [Sol 6] On considère un tableau carré d'entiers positifs, comme celui ci-dessous. Dans un tel tableau, on peut se déplacer d'une case vers la gauche, la droite, le haut ou le bas. Le poids d'un chemin est la somme des entiers rencontrés sur le chemin. On se pose la question du poids minimal d'un chemin reliant la case en haut à gauche vers la case en bas à droite. Dans le tableau T_1 ci-dessous, ce poids est 2297, et un chemin correspondant est indiqué en gras.

131	673	**234**	**103**	**18**
201	**96**	**342**	965	**150**
630	803	746	**422**	**111**
537	699	497	**121**	956
805	732	524	**37**	**331**

On trouvera dans le fichier `ex6.py` les fonctions `dijkstra_path` et `dijkstra_path_adja` des deux exercices précédents, le tableau `T1` ci-dessus ainsi qu'un tableau `T2` de taille 80×80.

1) Soit $n \geq 3$. À un tableau d'entiers `T` de taille $n \times n$ comme ci-dessus, on associe un graphe à n^2 sommets numérotés de 0 à $n^2 - 1$: chaque case (i,j) du tableau est un sommet (numérotation dans le sens de lecture) et il y a un arc entre deux cases si et seulement si ces deux cases sont voisines, le poids de l'arc étant alors l'entier contenu dans la case de destination.

1.1) Écrire la fonction `sommet(i,j,n)` qui renvoie le numéro a du sommet de la case (i,j) et la fonction `indices(a,n)` qui renvoie le couple d'indice (i,j) de la case correspondant au sommet a.

```
>>> sommet(2,3,5)
13
>>> indices(13,5)
(2, 3)
```

1.2) Écrire une fonction `cases_voisines(i,j,n)`, de complexité en $O(1)$, qui renvoie la liste des couples d'indices des cases voisines de (i,j).

```
>>> cases_voisines(0,0,5)
[(0, 1), (1, 0)]
>>> cases_voisines(1,2,5)
```

```
[(1, 1), (1, 3), (0, 2), (2, 2)]
```

1.3) Écrire une fonction `array2adja(T)` qui renvoie la matrice d'adjacence du graphe correspondant tableau d'entiers `T`.

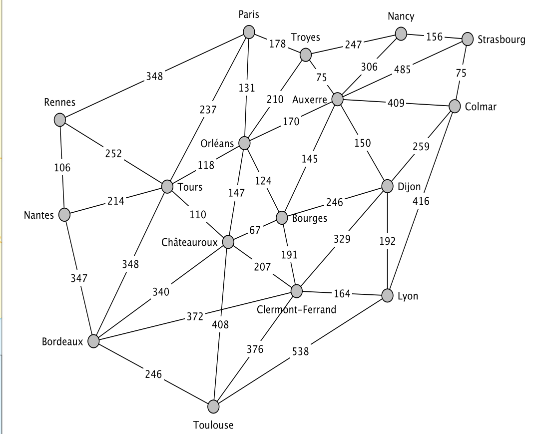
1.4) À l'aide de la fonction `dijkstra_path_adja` et des fonctions précédentes, écrire une fonction `dijkstra_array_adja(T)` qui prend en entrée un tableau carré d'entiers `T`, recherche un chemin de poids minimal du coin supérieur gauche de `T` à son coin inférieur droit et qui renvoie le couple `(p,C)` où `p` est le poids du chemin sélectionné (attention à ne pas oublier l'entier contenu dans la case de départ!) et où `C` est le meilleur chemin chemin détecté *i.e.* la liste des couples d'indices des cases traversées.
Tester cette fonction sur le tableau `T1` et vérifier le résultat.

2) Appliquer `dijkstra_array_adja` au tableau `T2` permet de résoudre le problème 83 du site Project-Euler : `https://projet-euler.fr/`.
Néanmoins la temps d'exécution est relativement long : la matrice d'adjacence est de taille $80^2 \times 80^2$ ce qui impose, dans la fonction `dijkstra_path_adja(A, v_init,v_fin)`, de tester 80^2 sommets à chaque extraction (ligne 52 de `ex6.py`) alors qu'un sommet n'a au maximum que 4 voisins! Il est donc plus pertinent, pour les grands les tableaux, de coder le graphe correspondant non pas par matrice d'adjacence mais par un dictionnaire.

2.1) Écrire une fonction `array2dico(T)` qui renvoie la dictionnaire codant le graphe correspondant à `T`, les sommets étant le couple d'indice de la case.

2.2) À l'aide de la fonction précédente et de la fonction `dijkstra_path_adja`, écrire une fonction `dijkstra_array_dico(T)` qui prend en entrée un tableau carré d'entiers `T`, fais la recherche d'un chemin de poids minimal du coin supérieur gauche de `T` à son coin inférieur droit et qui renvoie le couple `(p,C)` où `p` est le poids du chemin sélectionné (attention à ne pas oublier l'entier contenu dans la case de départ!) et où `C` est le meilleur chemin chemin détecté *i.e.* la liste des couples d'indices des cases traversées.
Tester cette fonction sur les tableaux `T1` et `T2` et commenter.

Exercice 7 Sur l'autoroute des vacances... [Sol 7] On donne ci-dessous la carte de France et de ses principaux axes routiers ainsi qu'un graphe G formé de certaines villes (sommets du graphe) reliées par des routes (arêtes du graphe, le poids étant la distance séparant les deux villes).
On trouvera dans le fichier `ex7.py` le dictionnaire `gfr` codant le graphe G, le dictionnaire `dir_stras` des distances à vol d'oiseaux entre les villes représentées sur le graphe et Strasbourg, le code de la fonction `dijkstra_path` (cf exercice 4) et enfin le code de la fonction `a_star_path` vu en cours.



Recherchez le plus court chemin de Colmar à Strasbourg puis de Bordeaux à Strasbourg avec Dijkstra et A* puis commentez les résultats obtenus.

Exercice 8 Dijkstra vs A* [Sol 8] Le fichier `ex8.py` contient le code de la fonction `dijkstra_path(g,v_init,v_end)` (cf exercice 4), le code de la fonction `a_star_path(g, v_init,v_fin, h)` vu en cours, et les dictionnaires `g1`, `g2` et `g3` codant trois graphes G_1, G_2 et G_3, également rencontrés en cours.
Dans cet exercice, les étiquettes des sommets sont des couples de réels; on redonne de plus la définition des distances d_p du plan réel : si $u(x_1,y_1)$ et $v(x_2,y_2)$ sont deux points de $\mathbb{R}^2$, on pose :

- $d_p(u,v) = (|x_2-x_1|^p + |y_2-y_1|^p)^{\frac{1}{p}}, \quad \forall p \in [1,+\infty[$,
- $d_\infty(u,v) = \max(|x_2-x_1|,|y_2-y_1|)$.

1) Écrire la fonction `dp(u,v,p)` qui renvoie la distance d_p entre les sommets u et v.

2) Écrire la fonction `a_star_path_dp(g, v_init,v_fin, p)` qui effectue la recherche du plus court chemin de `v_init` à `v_fin` avec l'algorithme A* avec l'heuristique d_p.

3) **3.1)** Recherchez le plus chemin de $(2,2)$ à $(4,4)$ dans G_1 avec Dijkstra et avec A* pour l'heuristique d_1, commentez.

3.2) Recherchez le plus chemin de $(4,4)$ à $(8,8)$ dans G_2 avec Dijkstra et avec A* pour l'heuristique d_2, commentez.

3.3) Recherchez le plus chemin de $(4,4)$ à $(8,8)$ dans G_3 avec DIJKSTRA, avec A* pour l'heuristique d_1 et avec A* pour l'heuristique d_∞ commentez.

SOLUTIONS DES EXERCICES

Solution 1

1) On trouve : $\delta(\gamma_3) = 4 \qquad \delta(\gamma_4) = 10$.

2) Notons n le nombre de sommets de γ et $|\gamma|$ sa longueur, on a $|\gamma| = n-1$

3) Le chemin (a, c, b, j, r) est de longueur minimale entre a et r : $\mathrm{d}_a[r] = 4$.

Le chemin (n, o, g, f, a, c) est de longueur minimale entre n et c : $\mathrm{d}_n[c] = 5$.

Solution 2

1) Il suffit de remplacer le dictionnaire des distances par celui des prédécesseurs dans le code du cours et d'adapter le code dans la condition de la boucle.

```
def bfs_pred(g, v):
    q = deque()
    visited = {x : False for x in g.keys()}
    pred = {x : None for x in g.keys()}
    q.append(v)
    visited[v] = True
    while len(q) > 0:
        w = q.popleft()
        for u in g[w]:
            if not visited[u]:
                visited[u] = True
                q.append(u)
                pred[u] = w
    return pred
```

2) Le parcours du dictionnaire des prédécesseurs depuis le sommet d'arrivée permet la construction d'une liste des sommets visités. La distance est calculée dans la boucle qui parcourt les prédécesseurs. Dans la solution proposée, on utilise une file pour éviter le retournement de liste en fin de traitement; la fonction `list` convertit la file en liste.

```
def shortest_path(g, u, v):
    pred, d, path = bfs_pred(g, u), 0, deque()
    while v != None:
        path.appendleft(v)
        d += 1
        v = pred[v]
    return d-1, list(path)
```

3)

```
>>> shortest_path(gex1, 'a', 'r')
(4, ['a', 'c', 'b', 'j', 'r'])
>>> shortest_path(gex1, 'n', 'c')
(5, ['n', 'o', 'g', 'f', 'a', 'c'])
```

Solution 3

Sommets visités	a	b	c	d	e	f	g	h	i
$\{\}$	0	∞	∞	∞	∞	∞	∞	∞	∞
$\{a\}$	.	3	∞	9	∞	∞	∞	∞	∞
$\{a,b\}$	.	.	7	9	5	∞	∞	∞	∞
$\{a,b,e\}$	.	.	7	9	.	7	∞	13	∞
$\{a,b,e,c\}$	.	.	.	9	.	7	∞	13	∞
$\{a,b,e,c,f\}$	.	.	.	9	.	.	∞	13	13
$\{a,b,e,c,f,d\}$	.	.	.	.	.	.	18	13	13
$\{a,b,e,c,f,d,h\}$	.	.	.	.	.	.	16	.	13
$\{a,b,e,c,f,d,h,i\}$	.	.	.	.	.	.	16	.	.
$\{a,b,e,c,f,d,h,i,g\}$	.	.	.	.	.	.	.	.	.
Distances à a	0	3	7	9	5	7	16	13	13

Solution 4

```
def dijkstra_path(g, v_init,v_fin):
    visited = {x : False for x in g}              # dico des sommets visités
    pred = {x : None for x in g}                  # dico des predecesseurs
    dist = {x : float('inf') for x in g}          # dico des distances
    dist[v_init] = 0                              # vinit est à distance 0 de lui-même
    hq = [(0, v_init)]
    heapq.heapify(hq)                             # création de la FP
    N = 0                                         # compteur des sommets visités
    while len(hq) > 0 and not(visited[v_fin]):    # visite des sommets
```

```python
        dv, v = heapq.heappop(hq)
        if not visited[v]:
            visited[v] = True
            N += 1
            for w, dvw in g[v]:
                if not visited[w]:
                    dw = dv + dvw
                    if dw < dist[w]:
                        dist[w] = dw
                        pred[w] = v
                        heapq.heappush(hq, (dw, w))
    if not visited[v_fin]:
        print("Pas de chemin de "+str(v_init)+" à "+str(v_fin))
    else:
        C = [v_fin]
        while C[0] != v_init:
            w = pred[C[0]]
            C = [w] + C
        return N, dist[v_fin], C
```

extraction du sommet de prio min
maj du compteur
parcours des voisins non visités de v
relâchement de l'arête (v,w)
maj de la distance min
maj du prédécesseur
ajout dans la FP
cas où vfin n'est pas accessible
construction du chemin

```python
>>> dijkstra_path(gex3, 'a', 'g')
(9, 16, ['a', 'b', 'e', 'h', 'g'])
```

Solution 5

```python
def dijkstra_path_adja(A, v_init,v_fin):
    n = len(A)
    visited = {x : False for x in range(n)}
    pred = {x : None for x in range(n)}
    dist = {x : float('inf') for x in range(n)}
    dist[v_init] = 0
    hq = [(0, v_init)]
    heapq.heapify(hq)
    N = 0
    while len(hq) > 0 and not(visited[v_fin]):
        dv, v = heapq.heappop(hq)
        if not visited[v]:
            visited[v] = True
            N += 1
            for w in range(n):
                dvw = A[v,w]
                if not visited[w] and dvw < float('inf'):
                    dw = dv + dvw
                    if dw < dist[w]:
                        dist[w] = dw
                        pred[w] = v
                        heapq.heappush(hq, \
                            (dw, w))
    if not visited[v_fin]:
        print("Pas de chemin de "\
            +str(v_init)+" à "+str(v_fin))
    else:
        C = [v_fin]
        while C[0] != v_init:
            w = pred[C[0]]
            C = [w] + C
        return N, dist[v_fin], C
```

le nombre de sommets
dico des sommets visités
dico des predecesseurs
dico des distances
vinit est à distance 0 de lui-même
création de la FP
compteur des sommets visités
visite des sommets
extraction du sommet de prio min
maj du compteur
parcours des voisins non visités de v
dvw est le poid de l'arc de v à w (inf si pas d'arc)
relâchement de l'arête (v,w)
maj de la distance min
maj du prédécesseur
ajout dans la FP
cas où vfin n'est pas accessible
construction du chemin

```python
>>> dijkstra_path_adja(A3, 0, 7)
(7, 13.0, [0, 1, 4, 7])
```

Solution 6

1) 1.1)

```python
def sommet(i,j,n):
    return n*i+j
def indices(a,n):
    return a//n, a%n
```

```python
>>> sommet(2,3,5)
13
>>> indices(13,5)
(2, 3)
```

1.2)

```python
def cases_voisines(i,j,n):
    V = []  # liste des cases voisines
    D = [-1,1] # liste des déplacements
    for dx in D: # test des voisins horizontaux
        if 0 <= j+dx < n:
            V.append((i,j+dx))
    for dy in D: # test des voisins verticaux
        if 0 <= i+dy < n:
            V.append((i+dy,j))
    return V
```

```python
>>> cases_voisines(0, 0, 5)
[(0, 1), (1, 0)]
>>> cases_voisines(1, 2, 5)
[(1, 1), (1, 3), (0, 2), (2, 2)]
```

1.3)

```python
def array2adja(T):
    n = len(T)
    N = n**2
    A = np.ones((N,N)) * float('inf') # initialisation de A
    for x in range(N): # pour chaque sommet x
        i, j = indices(x,n)
        V = cases_voisines(i, j, n) #V est la liste des \
↪ indices des voisins x
```

```
        for v in V:
            k, l = v
            A[x, sommet(k, l, n)] = T[v] #modification de \
↪ la case de A correspondante
    return A
```

```
>>> T1=np.array([
... [131,673,234,103,18],
... [201,96,342,965,150],
... [630,803,746,422,111],
... [537,699,497,121,956],
... [805,732,524,37,331]
... ])
>>> array2adja(T1)[0:7] # seulement les 7 premières lignes \
↪ ici
array([[ inf, 673.,  inf,  inf,  inf, 201.,  inf,  inf,  \
↪ inf,  inf,  inf,
         inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  \
         ↪ inf,  inf,  inf,
         inf,  inf,  inf],
       [131.,  inf, 234.,  inf,  inf,  inf,  96.,  inf,  \
       ↪ inf,  inf,  inf,
         inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  \
         ↪ inf,  inf,  inf,
         inf,  inf,  inf],
       [ inf, 673.,  inf, 103.,  inf,  inf,  inf, 342.,  \
       ↪ inf,  inf,  inf,
         inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  \
         ↪ inf,  inf,  inf,
         inf,  inf,  inf],
       [ inf,  inf, 234.,  inf,  18.,  inf,  inf,  inf, \
       ↪ 965.,  inf,  inf,
         inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  \
         ↪ inf,  inf,  inf,
         inf,  inf,  inf],
       [ inf,  inf,  inf, 103.,  inf,  inf,  inf,  inf,  \
       ↪ inf, 150.,  inf,
         inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  \
         ↪ inf,  inf,  inf,
         inf,  inf,  inf],
       [131.,  inf,  inf,  inf,  inf,  inf,  96.,  inf,  \
       ↪ inf,  inf, 630.,
         inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  \
         ↪ inf,  inf,  inf,
         inf,  inf,  inf],
       [ inf, 673.,  inf,  inf,  inf, 201.,  inf, 342.,  \
       ↪ inf,  inf,  inf,
        803.,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  \
        ↪ inf,  inf,  inf,
         inf,  inf,  inf]])
```

On peut voir que la taille de la matrice d'ajacence est inutilement importante (très peu de voisine par case).

1.4)

```
def dijkstra_array_adja(T):
    A = array2adja(T)
    n = len(T)
    N, d, C = dijkstra_path_adja(A, 0, n**2-1)
    Cind = [] #Conversion de la liste des sommets en liste \
↪ d'indices des cases
    for a in C:
        Cind.append(indices(a,n))
    return d+T[0,0],Cind
```

```
>>> dijkstra_array_adja(T1)
(2297.0, [(0, 0), (1, 0), (1, 1), (1, 2), (0, 2), (0, 3), \
↪ (0, 4), (1, 4), (2, 4), (2, 3), (3, 3), (4, 3), (4, 4)])
```

2) 2.1)

```
def array2dico(T):
    n = len(T)
    g = {}  #le dictionnaire, initialement vide
    D = [-1,1]  #liste des déplacements
    #Les 4 coins
    g[(0,0)] = [((0,1),T[0,1]),((1,0),T[1,0])]
    g[(0,n-1)] = [((0,n-2),T[0,n-2]),((1,n-1),T[1,n-1])]
    g[(n-1,0)] = [((n-1,1),T[n-1,1]),((n-2,0),T[n-2,0])]
    g[(n-1,n-1)] = \
↪ [((n-1,n-2),T[n-1,n-2]),((n-2,n-1),T[n-2,n-1])]
    #les bords verticaux privés des coins
    for i in range(1,n-1):
        g[(i,0)] = [((i,1),T[i,1])]
        g[(i,n-1)] = [((i,n-2),T[i,n-2])]
```

```
        for dy in D:
            g[(i,0)].append(((i+dy,0),T[i+dy,0]))
            g[(i,n-1)].append(((i+dy,n-1),T[i+dy,n-1]))
    #les bords horizontaux privés des coins
    for j in range(1,n-1):
        g[(0,j)] = [((1,j),T[1,j])]
        g[(n-1,j)] = [((n-2,j),T[n-2,j])]
        for dx in D:
            g[(0,j)].append(((0,j+dx),T[0,j+dx]))
            g[(n-1,j)].append(((n-1,j+dx),T[n-1,j+dx]))
    #le reste
    for i in range(1,n-1):
        for j in range(1,n-1):
            g[(i,j)] = []
            for dx in D:
                g[(i,j)].append(((i,j+dx),T[i,j+dx]))
            for dy in D:
                g[(i,j)].append(((i+dy,j),T[i+dy,j]))
    return g
```

```
>>> array2dico(T1)
```

```
{(0, 0): [((0, 1), 673), ((1, 0), 201)], (0, 4): [((0, 3), \
↪ 103), ((1, 4), 150)], (4, 0): [((4, 1), 732), ((3, 0), \
↪ 537)], (4, 4): [((4, 3), 37), ((3, 4), 956)], (1, 0): \
↪ [((1, 1), 96), ((0, 0), 131), ((2, 0), 630)], (1, 4): \
↪ [((1, 3), 965), ((0, 4), 18), ((2, 4), 111)], (2, 0): \
↪ [((2, 1), 803), ((1, 0), 201), ((3, 0), 537)], (2, 4): \
↪ [((2, 3), 422), ((1, 4), 150), ((3, 4), 956)], (3, 0): \
↪ [((3, 1), 699), ((2, 0), 630), ((4, 0), 805)], (3, 4): \
↪ [((3, 3), 121), ((2, 4), 111), ((4, 4), 331)], (0, 1): \
↪ [((1, 1), 96), ((0, 0), 131), ((0, 2), 234)], (4, 1): \
↪ [((3, 1), 699), ((4, 0), 805), ((4, 2), 524)], (0, 2): \
↪ [((1, 2), 342), ((0, 1), 673), ((0, 3), 103)], (4, 2): \
↪ [((3, 2), 497), ((4, 1), 732), ((4, 3), 37)], (0, 3): \
↪ [((1, 3), 965), ((0, 2), 234), ((0, 4), 18)], (4, 3): \
↪ [((3, 3), 121), ((4, 2), 524), ((4, 4), 331)], (1, 1): \
↪ [((1, 0), 201), ((1, 2), 342), ((0, 1), 673), ((2, 1), \
↪ 803)], (1, 2): [((1, 1), 96), ((1, 3), 965), ((0, 2), \
↪ 234), ((2, 2), 746)], (1, 3): [((1, 2), 342), ((1, 4), \
↪ 150), ((0, 3), 103), ((2, 3), 422)], (2, 1): [((2, 0), \
↪ 630), ((2, 2), 746), ((1, 1), 96), ((3, 1), 699)], (2, \
↪ 2): [((2, 1), 803), ((2, 3), 422), ((1, 2), 342), ((3, \
↪ 2), 497)], (2, 3): [((2, 2), 746), ((2, 4), 111), ((1, \
↪ 3), 965), ((3, 3), 121)], (3, 1): [((3, 0), 537), ((3, \
↪ 2), 497), ((2, 1), 803), ((4, 1), 732)], (3, 2): [((3, \
↪ 1), 699), ((3, 3), 121), ((2, 2), 746), ((4, 2), 524)], \
↪ (3, 3): [((3, 2), 497), ((3, 4), 956), ((2, 3), 422), \
↪ ((4, 3), 37)]}
```

2.2)

```
def dijkstra_array_dico(T):
    g = array2dico(T)
    n = len(T)
    N, d, C = dijkstra_path(g,(0,0),(n-1,n-1))
    return d+T[0,0],C
```

```
>>> dijkstra_array_dico(T1)
(2297, [(0, 0), (1, 0), (1, 1), (1, 2), (0, 2), (0, 3), (0, \
↪ 4), (1, 4), (2, 4), (2, 3), (3, 3), (4, 3), (4, 4)])
>>> dijkstra_array_dico(T2)
```

```
(425185, [(0, 0), (1, 0), (1, 1), (1, 2), (0, 2), (0, 3), \
↪ (0, 4), (0, 5), (0, 6), (1, 6), (1, 7), (2, 7), (2, 8), \
↪ (3, 8), (3, 9), (3, 10), (3, 11), (4, 11), (4, 12), (4, \
↪ 13), (4, 14), (4, 15), (5, 15), (6, 15), (6, 16), (6, \
↪ 17), (6, 18), (5, 18), (5, 19), (5, 20), (5, 21), (5, \
↪ 22), (5, 23), (6, 23), (6, 24), (6, 25), (6, 26), (6, \
↪ 27), (6, 28), (6, 29), (7, 29), (8, 29), (8, 30), (9, \
↪ 30), (9, 31), (10, 31), (11, 31), (11, 30), (12, 30), \
↪ (13, 30), (14, 30), (15, 30), (16, 30), (17, 30), (17, \
↪ 31), (17, 32), (17, 33), (17, 34), (18, 34), (19, 34), \
↪ (20, 34), (21, 34), (22, 34), (22, 35), (23, 35), (24, \
↪ 35), (24, 36), (24, 37), (25, 37), (25, 38), (26, 38), \
↪ (26, 39), (26, 40), (26, 41), (27, 41), (27, 42), (27, \
↪ 43), (28, 43), (29, 43), (30, 43), (31, 43), (32, 43), \
↪ (32, 44), (32, 45), (32, 46), (33, 46), (34, 46), (35, \
↪ 46), (36, 46), (37, 46), (38, 46), (39, 46), (39, 47), \
↪ (39, 48), (40, 48), (41, 48), (42, 48), (43, 48), (44, \
↪ 48), (44, 49), (45, 49), (46, 49), (47, 49), (47, 50), \
↪ (48, 50), (48, 51), (49, 51), (49, 52), (50, 52), (50, \
↪ 53), (50, 54), (50, 55), (51, 55), (52, 55), (52, 56), \
↪ (52, 57), (52, 58), (53, 58), (53, 59), (53, 60), (54, \
↪ 60), (55, 60), (56, 60), (57, 60), (58, 60), (59, 60), \
↪ (60, 60), (61, 60), (62, 60), (62, 61), (62, 62), (62, \
↪ 63), (62, 64), (62, 65), (62, 66), (63, 66), (64, 66), \
↪ (65, 66), (66, 66), (67, 66), (67, 67), (67, 68), (67, \
↪ 69), (67, 70), (67, 71), (68, 71), (68, 72), (69, 72), \
↪ (69, 73), (69, 74), (69, 75), (69, 76), (70, 76), (71, \
↪ 76), (71, 77), (72, 77), (72, 78), (73, 78), (74, 78), \
↪ (75, 78), (76, 78), (77, 78), (78, 78), (79, 78), (79, \
↪ 79)])
```

L'exécution de cette dernière fonction sur T2 est instantanée.

Solution 7

```
>>> dijkstra_path(gfr, "Colmar", "Strasbourg")
(2, 75, ['Colmar', 'Strasbourg'])
>>> a_star_path(gfr, "Colmar","Strasbourg",dir_stras)
(2, 75, ['Colmar', 'Strasbourg'])
>>> dijkstra_path(gfr, "Bordeaux","Strasbourg")
(17, 998, ['Bordeaux', 'Châteauroux', 'Bourges', 'Dijon', \
↪ 'Colmar', 'Strasbourg'])
>>> a_star_path(gfr, "Bordeaux","Strasbourg",dir_stras)
(13, 998, ['Bordeaux', 'Châteauroux', 'Bourges', 'Dijon', \
↪ 'Colmar', 'Strasbourg'])
```

Sur ces deux exemples, les deux algorithmes trouvent le même chemin, mais A* le trouve plus rapidement lorsque l'on s'éloigne de Strasbourg.

Solution 8

1)
```
def dp(u,v,p):
    x, y = u
    z, t = v
    a, b = abs(z-x), abs(t-y)
    if p == float('inf'):
        return max(a,b)
    else:
        return (a**p+b**p)**(1/p)
```

2)
```
def a_star_path_dp(g, v_init,v_fin, p):
    #construction du dico h de la distance d_p des sommets du \
↪ graphe à v_fin
    h = {}
    for u in g:
        h[u] = dp(u, v_fin, p)

#on lance la recherche avec a_star_path avec h pour heuristique
    return a_star_path(g, v_init, v_fin, h)
```

3) 3.1)
```
>>> dijkstra_path(g1,(2,2),(4,4))
(15, 8, [(2, 2), (2, 3), (3, 3), (3, 4), (4, 4)])
>>> a_star_path_dp(g1,(2,2),(4,4),1)
(8, 8, [(2, 2), (2, 3), (3, 3), (3, 4), (4, 4)])
```

A* trouve le même chemin que Dijkstra mais en visitant moins de sommets.

3.2)
```
>>> dijkstra_path(g2,(4,4),(8,8))
(14, 7, [(4, 4), (4, 2), (4, 0), (6, 0), (8, 0), (8, 4), \
↪ (8, 6), (8, 8)])
>>> a_star_path_dp(g2,(4,4),(8,8),2)
```

```
(8, 11, [(4, 4), (4, 6), (6, 6), (6, 8), (8, 8)])
```

A* trouve un moins bon chemin que DIJKSTRA mais en visitant moins de sommets.

3.3)
```
>>> dijkstra_path(g3,(4,4),(8,8))
(7, 6, [(4, 4), (0, 4), (0, 0), (8, 0), (8, 4), (8, 8)])
>>> a_star_path_dp(g3,(4,4),(8,8),1)
(9, 13, [(4, 4), (4, 6), (4, 8), (6, 8), (6, 6), (6, 4), \
↪ (8, 4), (8, 8)])
>>> a_star_path_dp(g3,(4,4),(8,8),float('inf'))
(7, 6, [(4, 4), (0, 4), (0, 0), (8, 0), (8, 4), (8, 8)])
```

A* trouve un moins bon chemin que DIJKSTRA et en visitant plus de sommets avec l'heuristique d_1, alors qu'il donne les mêmes résultats que DIJKSTRA avec l'heuristiqued_∞. Le choix de l'heuristique n'est donc pas anodin.