Révisions de MPSI/PCSI

Les graphes

Plan du cours

I Définitions

Définitions : Graphe non orienté

- Un **graphe non orienté** est un couple $G = (S, A)$ dans lequel S est un ensemble non vide (dont les éléments sont appelés **sommets**) et A un ensemble de parties à deux éléments de S (dont les éléments sont appelés arêtes du **graphe**).
- Deux sommets sont **adjacents** ssi il existe une arête qui les relie.
- Le **degré** d'un sommet est égal au nombre d'arêtes dont il est extrémité. On le notera $d(s_i)$.
- Un graphe est **connexe** si deux sommets distincts sont toujours reliés par un chemin.
- L'**ordre** d'un graphe est le nombre de sommets.

Définitions : Chemin

- Un chemin d'un sommet s_0 à un sommet s_n est une séquence $(s_0, s_1, ..., s_n)$ où deux sommets consécutifs sont adjacents.
- La longueur d'un chemin est le nombre d'arêtes utilisées pour aller du sommet de départ au sommet d'arrivée.
- La distance entre deux sommets est la longueur minimale d'un chemin reliant ces deux sommets.
- Un cycle est un chemin tel que le sommet de départ et le sommet d'arrivée sont identiques.

Définitions : Graphe orienté

Un **graphe orienté** est un couple d'ensembles (S, A) avec $A \subset S^2$. Les éléments de A que l'on appelle alors des **arcs** sont des couples (s_i, s_j) de sommets. On pourra aussi les notes $s_i \to s_j$ et parler d'extrémités initiale et terminale. Les arêtes ont un sens de parcours.
On appelle **degré sortant** d'un sommet s_i le nombres d'arcs $(s_i, s_j) \in A$. Ce degré sortant est noté $d_+(s_i)$. Le degré entrant est défini de façon analogue et est noté $d_-(s_i)$.

Définitions : Graphe pondéré

Ajouter des poids ou des étiquettes aux arêtes d'un graphe apporte des informations supplémentaires.

- Un graphe est **pondéré** si un nombre, un **poids**, est associé à chaque arête ou chaque arc.
- Un graphe est **étiqueté** si une **étiquette**, est associée à chaque arête ou chaque arc.

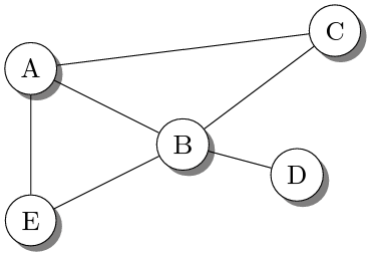
Dans un réseau routier par exemple, un poids peut être le nombre de kilomètres d'une route liant deux lieux, une étiquette peut être le nom de cette route.

II Représentations

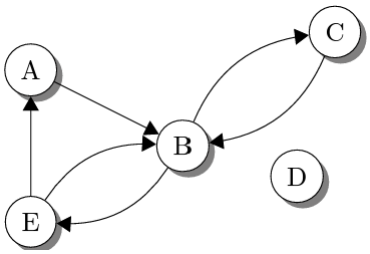
II.1 Représentation schématique

La manière la plus simple de représenter un graphe est de faire un dessin. Les sommets sont représentés par des points, les arêtes par des lignes, chacune reliant deux poids.

Exemple 1.



Ce graphe n'est pas orienté, et connexe.
Sommet A : $d(A) = 3$, ses voisins sont B, C, E.
$[ABD]$ est un chemin de longueur 2.
$[AEBD]$ est un chemin de longueur 3.
$[ACBEA]$ est un cycle



Ce graphe est orienté, et non connexe.
D est un sommet isolé.
Degré entrant de B : $d_-(B) = 3$
Degré sortant de B : $d_+(B) = 2$

II.2 Représentation en python

II.2.a) Liste d'adjacence

On peut représenter un graphe non orienté en précisant pour chacun des sommets la liste de ses voisins. Ces listes s'appellent des **listes d'adjacence**. L'ordre d'écriture n'a pas d'importance.

Dans le cas de graphes orientés, on peut présenter des **listes de successeurs**.

Ces **listes d'adjacence peuvent être représentées en `Python` à l'aide** :

— **liste de listes** : chaque élément de la liste est une liste contenant un sommet et la liste de ces voisins.

```
# Pour le graphe non orienté :
G=[ ["A",["B","C","E"]] , ["B",["A","C","D","E"]] , ["C",["A","B"]] , ["D",["B"]] , ["E",["A","B"]] ]
# Pour le graphe orienté :
G =[ ["A",["B"]] , ["B",["C","E"]] , ["C",["B"]] , ["D",[]] , ["E",["A","B"]] ]
```

— **dictionnaire** : les clés sont les sommets et les valeurs correspondent aux clés sont les listes des voisins.

```
# Pour le graphe non orienté :
G = {"A":["B","C","E"] , "B":["A","C","D","E"] , "C":["A","B"] , "D":["B"] , "E":["A","B"] }
# Pour le graphe orienté :
G = {"A":["B"] , "B":["C","E"] , "C":["B"] , "D":[] , "E":["A","B"] }
```

Si le graphe est pondéré, on complète les listes d'adjacence avec les poids.

```
# Avec une liste de listes :
G=[ ["A",[("B",2),("E",5)]] , ["B",[("A",2),("C",1),("E",3)]] , ["C",[("B",1)]] , ["D",[]] , ["E",[("A",5),("B",3)]]]
# Avec un dictionnaire
G={"A":[("B",2),("E",5)] , "B":[("A",2),("C",1),("E",3)] , "C":[("B",1)] , "D":[] , "E":[("A",5),("B",3)]}
```

II.2.b) Matrice d'adjacence

En mathématiques, on peut associer à un graphe, une matrice carrée (n, n) ou un tableau, où n est le nombre de sommets. Les sommets sont numérotés de 0 à $n-1$. À l'intersection d'une ligne i et d'une colonne j le nombre représenté la présence ou non d'une arrête enter les sommets i et j : 1 pour la présence, 0 pour l'absence.

Le tableau correspondant au graphe non orienté dessiné précédemment est le suivant :

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	1	1
C	1	1	0	0	0
D	0	1	0	0	0
E	1	1	0	0	0

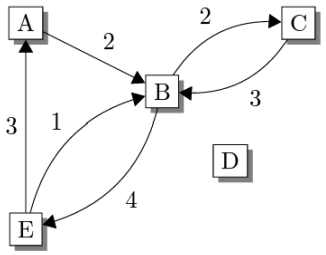
La diagonale ne contient que des 0 et est un axe de symétrie du tableau, c'est le cas pour les graphes non orientés.

Le même graphe peut être représenté par des matrices différentes qui dépendent de l'ordre des sommets qui est pris en compte.

La matrice du graphe non orienté est la suivante : $\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$

Dans le cas du graphe orienté précédent, on obtient la matrice d'adjacence suivante, qui n'est plus symétrique :
$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$

Dans le cas de graphe pondéré ou étiqueté, on place les informations le long des arêtes. On peut utiliser la matrice en remplaçant les 1 par les poids par exemple.



$\begin{pmatrix} 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 4 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{pmatrix}$

En `Python`, on pourra utiliser les **listes de listes** ou les tableaux `numpy` pour représenter une matrice d'adjacence.

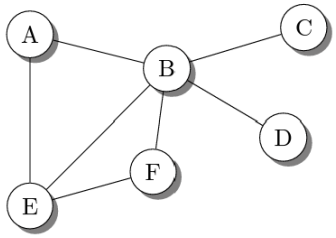
III Parcours d'un graphe

III.1 Parcours en profondeur

Définition : Parcours en profondeur

À partir d'un sommet, on passe à un de ses voisins, puis à un voisin de ce voisin et ainsi de suite. S'il n'y a pas de voisin, on revient au sommet précédent et on passe à un autre de ses voisins.

Exemple 2.



Parcours en profondeur à partir de A : On commence par un voisin de A : B (par ex., on aurait pu commencer par E), puis un voisin de B : C. C n'a pas de voisin, on remonte à B et on visite un autre voisin : D qui n'a pas de voisin. On remonte à B et on visite un autre voisin : E, puis un voisin de E : F.

On utilise une pile pour placer les sommets en attente.

Définition : pile

Une pile est une structure de données linéaire (les données sont rangées sur une ligne) ayant pour maxime « dernier entré premier sorti » (Last In, First Out), LIFO.
Imaginez une pile d'assiette : vous pouvez l'assiette sur le dessus, et vous récupérer l'assiette sur le dessus, la dernière rangée.
Les piles peuvent être implémentées de plusieurs manières. Nous utiliserons les listes.

Effet	Python
Ajouter un élément	`pile.append(x)`
Retirer un élément et renvoyer sa valeur	`pile.pop()`

Le parcours en profondeur est en général utilisé pour parcourir tout un graphe, ainsi tant que des sommets restent non visités ils seront empilés. Tant que la pile n'est pas vide, nous dépilons son sommet et regardons s'il a déjà été exploré. Si ce n'est pas le cas, nous mettons à jour son père et empilons ses voisins. Lorsque la pile est vide, la liste des pères permet de reconstituer le parcours en profondeur du graphe. Le graphe est ici représenté sous forme de liste d'adjacence.

```
def parcours_prof(graphe,sommet):
    visite=[] # liste des voisins visités
    marque={} # dictionnaire des voisins visités
    attente=deque() # pile qui garde les sommets en attente
    attente.append(sommet) # on ajoute au sommet de la pile le sommet de départ
    while len(attente)>0: # tant que la pile n'est pas vide
        sommet=attente.pop() # on dépile le sommet de la pile
        if sommet not in marque: # si le sommet n'a pas déjà été visité
            visite.append(sommet) # on l'ajoute à la liste des sommets visités
            marque[sommet]=True # on l'ajoute au dictionnaire
            for s in graphe[sommet] : # pour les voisins de sommet
                if s not in marque: # le voisin s n'a pas déjà été visité
                    attente.append(s) # on le place dans la pile des sommets en attente d'être visité
    return visite
>>> parcours_prof_it(g,'A')
['A', 'E', 'F', 'B', 'D', 'C']
```

III.2 Parcours en largeur

Définitions : Parcours en largeur

À partir d'un sommet, on explore tous ses voisins immédiats. Puis à partir d'un voisin, on explore tous ses voisins immédiats sauf ceux déjà explorés. Et ainsi de suite.

Exemple 3. Pour le même graphe que précédemment.
Parcours en largeur à partir de A : On commence par tous les voisins de A : B et E, puis on repart de B et on visite tous ses voisins restants : C, D et F.

Une structure de données particulière est naturellement utilisée pour le parcours en largeur de graphes : il s'agit de la notion de file (d'attente).

Définition : file

Une file est une structure de données linéaire (les données sont rangées sur une ligne) ayant pour maxime « premier entré premier sorti » (First In, First Out), FIFO.
Imaginez une file d'attente : la première personne à en sortir, est la première à y être rentrée.
Les files peuvent être implémentées de plusieurs manières. Une solution est d'utiliser une liste doublement chaînée : un `deque`, qui est une structure composée de données et de moyens d'accéder à la donnée suivante et à la donnée précédente. Cette structure est fournie en Python par le module `collections.deque`.

Ainsi :

Effet	Python
Ajouter un élément	`f.append(x)`
Retirer un élément et renvoyer sa valeur	`f.popleft()`
Créer une file vide	`f = deque()`

Pour implémenter le parcours en largeur, la file contiendra initialement le sommet de départ. Tant qu'elle n'est pas vide, nous traitons le premier sommet et regardons s'il a déjà été exploré. Si ce n'est pas le cas, nous mettons à jour son père et enfilons ses voisins. Lorsque la file est vide, la liste des pères permet de reconstituer le parcours en profondeur du graphe. Le graphe est ici représenté sous forme de liste d'adjacence.

```
def parcours_largeur(graphe,sommet):
    visite=[] # liste des voisins visités
    marque={} # dictionnaire des voisins visités
    attente=deque() # file qui garde les sommets en attente
    attente.append(sommet) # on ajoute au sommet de la file le sommet de départ
    while len(attente)>0: # tant que la file n'est pas vide
        sommet=attente.popleft() # on visite le sommet au début de la file
        if sommet not in marque: # si le sommet n'a pas déjà été visité
            visite.append(sommet) # on l'ajoute à la liste des sommets visités
            marque[sommet]=True # on l'ajoute au dictionnaire
            for s in graphe[sommet] : # pour les voisins de sommet
                if s not in marque: # le voisin s n'a pas déjà été visité
                    attente.append(s) # on le place dans la pile des sommets en attente d'être visité
    return visite
>>> parcours_largeur(g,'A')
['A', 'B', 'E', 'C', 'D', 'F']
```

IV Recherche d'un plus court chemin

On s'intéresse maintenant à un graphe pondéré avec des poids positifs. Le chemin le plus court est celui de coût, c'est-à-dire la somme des poids des arêtes, le plus faible.

IV.1 Algorithme de Dijkstra

Cet algorithme, publié par Edsger Dijkstra en 1959, utilise un **parcours en largeur** et calcul le plus court chemin entre un sommet et chacun des autres sommets.

On suppose le graphe connexe et non orienté.

IV.1.a) Principe

Si le plus court chemin entre deux sommets D et A passe par un sommet I, alors la partie de ce chemin entre D et I est le plus court chemin de D à I, et la partie entre I et A est le plus court chemin entre I et A. À chaque étape, on effectue donc le meilleur choix possible. C'est un algorithme glouton.

L'algorithme est semblable à celui d'un parcours en largeur d'abord, mais au lieu d'utiliser une file pour les sommets en attente, on utilise une fil de priorité. Cela signifie qu'on extrait le sommet ayant la priorité, dans ce cas c'est celui qui correspond à la distance minimale.

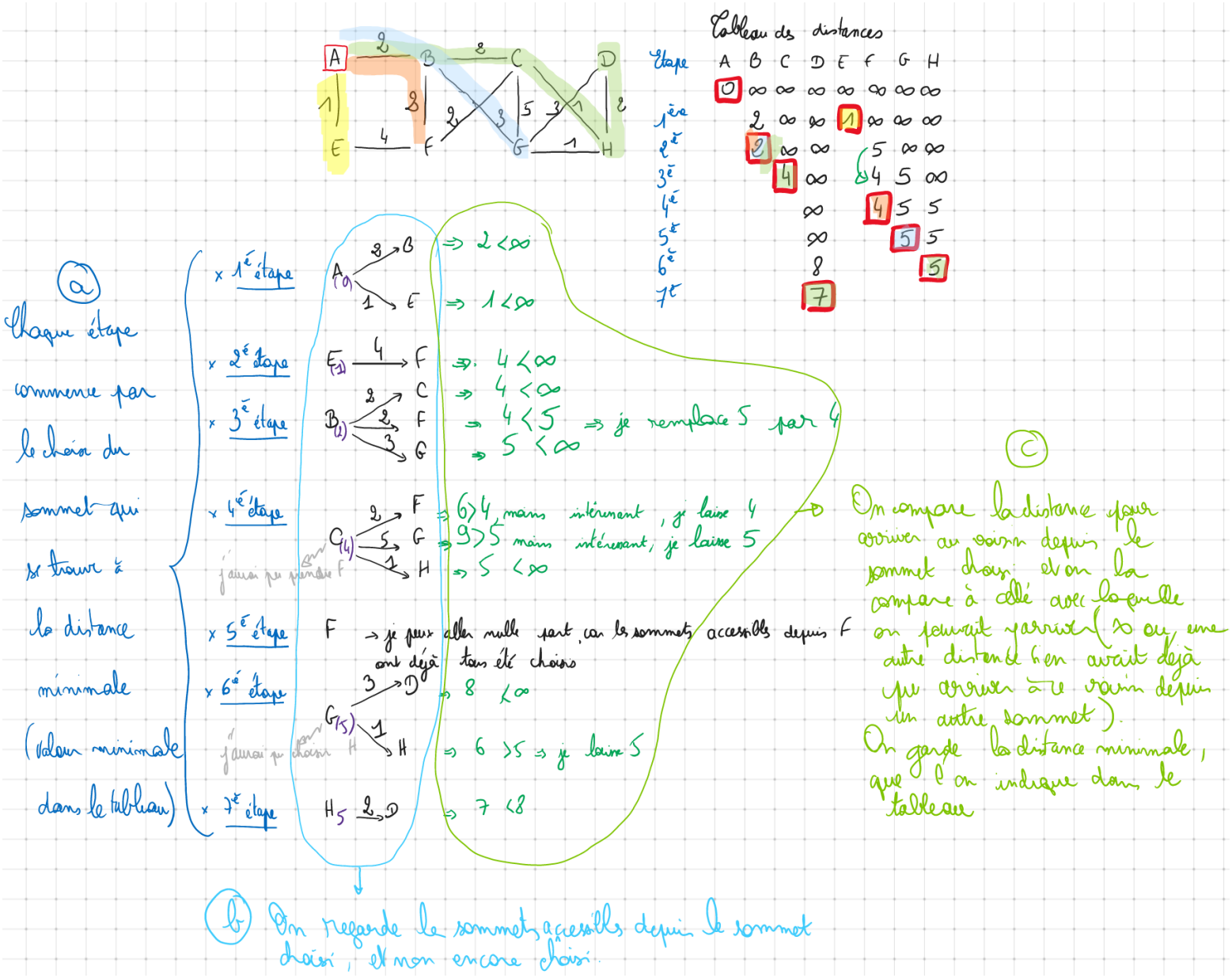
IV.1.b) Exemple

On considère le graphe représenté ci-dessous et on cherche le plus court chemin entre le sommet A et chacun des autres sommets du graphe. On affecte la valeur ∞ à chaque sommet, sauf au sommet A de départ, à qui on affecte la valeur 0. À chaque étape :

(a) On choisit le sommet dont la distance depuis A dans le tableau est minimale.

(b) On regarde ses différents voisins encore accessibles (c'est-à-dire qui n'ont pas déjà été choisis).

(c) On compare la distance avec laquelle on arrive aux différents voisins depuis ce sommet, à la distance avec laquelle on avait pu y arriver jusque là (l'infini, ou une autre distance par un autre chemin). On garde la distance minimale avec laquelle on peut arriver à ce voisin.



IV.1.c) Implémentation en python

Pour représenter l'ensemble des sommets et leur distance, nous allons utiliser un dictionnaire dont les sommets sont les clés et les distances leurs valeurs.

À chaque étape, il faut comparer la distance à laquelle se trouve chaque sommet. Il faut donc commencer par écrire une fonction qui renvoie la clé de valeur minimale.

```
def minimum(dico):
    mini=float('inf') # initialisation de la valeur minimale
    for cle in dico :  # parcours des clés de dico
        if dico[cle]<mini: # clé de valeur <au minimum local
            mini=dico[cle] # on a trouvé un nouveau minimum local
            cle_min=cle # clé de valeur=minimum local actuel
    return cle_min
```

```
def Dijkstra(graphe,S):
    res={} # dictionnaire des valeurs minimales de distance entre le sommet S et chacun des autres sommets
    dist={cle:float('inf') for cle in graphe} # initialisation du dictionnaire des distances
    dist[S]=0 # sommet S de départ, distance=0
    while len(dist)>0: # tant que d n'est pas vide, il reste des sommets à visiter
        # (a)
        smin=minimum(dist) # sommet de distance minimale
        res[smin]=dist[smin] # on copie le sommet smin et sa distance dans le dictionnaire des résultats
        for k in range(len(graphe[smin])): # parcours des voisins de smin
            # (b)
            v,d=graphe[smin][k] # v : nom du voisin, d distance à laquelle il se trouve de smin
            if v not in res: # on ne s'intéresse qu'au voisin non déjà choisi
                # (c)
                dist[v]=min(dist[v],dist[smin]+d) # on choisit la distance minimale entre celle avec laquelle on aurait déjà pu arriver à v (dist[v]), et celle avec laquelle on arrive depuis smin (dist[smin]+d)
        res[smin]=dist[smin] # on copie le sommet smin et sa distance dans le dictionnaire des résultats
        del dist[smin] # on supprime smin de dist, qui contient les sommets qu'il reste à visiter
    return res
```

IV.2 Algorithme A^*

IV.2.a) Principe

Peter E. Hart, Nils John Nilsson et Bertram Raphael ont proposé un algorithme de recherche d'un chemin nommé **algorithme** A^* qui fournit un chemin entre deux sommets donnés. C'est une extension de l'algorithme de Dijkstra.

Cet algorithme fournit l'une des meilleurs solutions rapidement. Il est utilisé en intelligence artificielle et dans des applications de jeux vidéos pour lesquels le plus important est la vitesse d'obtention d'une solution, même si elle n'est pas optimale.

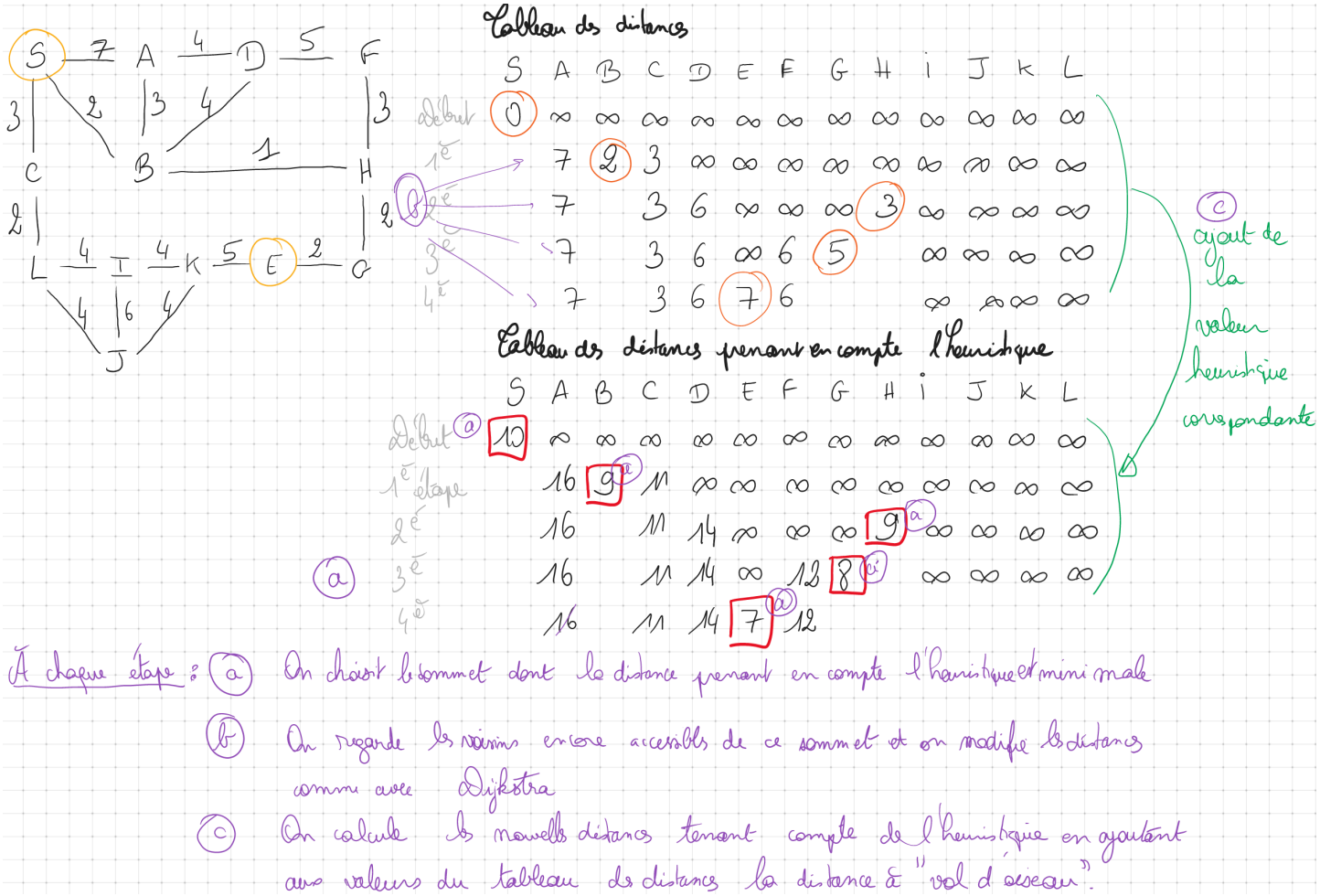
L'algorithme utilise une évaluation heuristique sur chaque sommet afin de parvenir à trouver le meilleur chemin. Les sommets sont visités suivant l'ordre donné par cette évaluation.

Une méthode heuristique est une méthode de résolution utilisée pour obtenir une solution rapidement, pas forcément la meilleure, quand d'autres algorithmes ont une complexité en temps trop élevée. On n'explore pas toutes les possibilités pour trouver la solution optimale, mais on les filtre à l'aide de données supplémentaires provenant de mesures, d'expériences, ou de statistiques.

Dans une recherche de distance minimale dans un graphe représentant un réseau routier, les valeurs heuristiques peuvent être les distances « à vol d'oiseau. »

IV.2.b) Exemple

Exemple 4. On étudie le graphe ci-dessous, et on souhaite déterminer le chemin le plus court pour aller du sommet `'S'` au sommet `'E'`. Ce graphe est représenté par le dictionnaire des listes d'adjacences, et on choisit l'heuristique donnée par les valeurs heuristiques distance « à vol d'oiseau » jusqu'à `'E'`.



On obtient le parcourt suivant : S-B-H-G-E, pour une distance totale de 7.
Le résultat dépend de l'heuristique choisi.

IV.2.c) Implémentation en python

```
def A_etoile(graphe,deb,fin,h):
    res={} # dict. des distances minimales entre deb et les autres
    dist={cle:float('inf') for cle in graphe} # dict. des distances
    dist[deb]=0 # sommet deb de départ, distance=0
    disth={cle:dist[cle]+h(cle) for cle in graphe} # dictionnaire des distances tenant compte de l'heuristique
    while fin in dist: # tant qu'on n'est pas arrivée au sommet fin
        smin=minimum(disth) # sommet de dist min pour démarrer une étape
        res[smin]=dist[smin] # copie de smin et sa distance dans res
        for k in range(len(graphe[smin])): # parcours des voisins de smin
            v,d=graphe[smin][k] # v : nom du voisin, d distance à laquelle il se trouve de smin
            if v not in res: # voisin non déjà choisi
                dist[v]=min(dist[v],dist[smin]+d) # on choisit la dist min entre celle avec laquelle on aurait déjà pu arriver à v (dist[v]), et celle avec laquelle on arrive depuis smin (dist[smin]+d)
                disth[v]=dist[v]+h(v) # nouvelle distance minimale tenant compte de l'heuristique
        res[smin]=dist[smin] # on copie le sommet smin et sa distance dans le dictionnaire des résultats
        del dist[smin] # on supprime smin de dist, qui contient les sommets qu'il reste à visiter
        del disth[smin]
    return res
```