Introduction historique

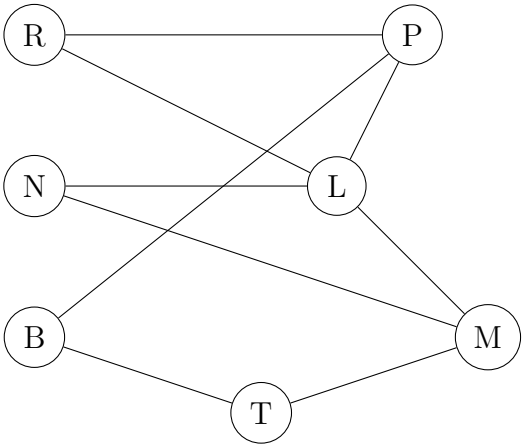
Plan du chapitre

1 Introduction

Un graphe est un objet traduisant des relations binaires orientées ou non entre des éléments. Il est constitué de sommets (les éléments) et d'arrêtes (indiquant une relation entre ces objets).

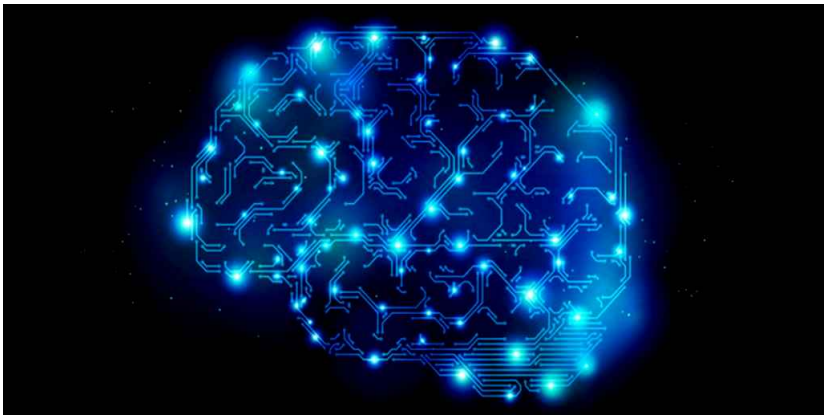
On retrouve ces objets dans de très nombreux domaines :

— **Réseaux aériens (ou routiers, maritimes...)**



— **Réseaux sociaux**

— **Réseaux de neurones**



2 Graphes non orientés

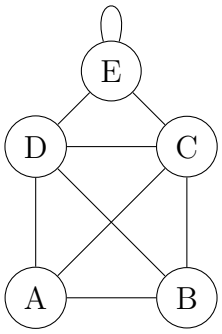
2.A Premières définitions

Définition 1

On appelle graphe non orienté $G = (S, A)$ la donnée de deux ensembles :
- un ensemble de sommets $S = \{s_1, \dots, s_n\}$.
- un ensemble d'arrêtes $A = \{a_1, \dots, a_m\}$.

Une arrête relie deux sommets de G. On note $a_i = \{s_k, s_\ell\}$ si l'arrête a_i relie les sommets s_k et s_ℓ.

Exemple



Dans cet exemple, $G = (S, A)$ avec :
- $S = \{A, B, C, D, E\}$
- $A = \{\{A, B\}, \{B, C\}, \{C, D\}, \{A, D\}, \{C, E\}, \{D, E\}, \{A, C\}, \{B, D\}, \{E, E\}\}$.

Remarques

On appelle l'arrête qui relie le sommet E à lui-même, une **boucle**.
Les sommets sont aussi appelés les **noeuds** du graphe.
Deux sommets reliés par une arrête sont dits **adjacents** ou **voisins**.

Définition 2

On appelle degré d'un sommet $s \in S$, le nombre d'arrêtes ayant pour extrémité le sommet s.
On le note $d(s) \in \mathbb{N}$.

Exemple

Dans l'exemple ci-dessus, on a : $d(A) = 3$ et $d(E) = 3$.

Définition 3

On appelle chemin d'un sommet s_1 à un sommet s_2 une suite, si elle existe, telle que :
- son premier élément est s_1,
- ses élément sont alternativement des sommets et des arrêtes,
- son dernier élément est s_2.

Exemple

Dans l'exemple ci-dessus, il existe plusieurs chemins de A à E. En voici deux :
- $(A, \{A, D\}, D, \{D, E\}, E)$.
- $(A, \{A, B\}, B, \{B, C\}, C, \{C, D\}, D, \{D, E\}, E)$.

Remarques

La description d'un chemin se fait parfois uniquement via la suite des sommets empruntés : (A, D, E) et (A, B, C, D, E) dans les exemples précédents.

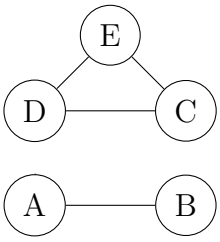
Définition 4

On appelle distance entre deux sommets $(s_1, s_2) \in S^2$ la longueur du plus court chemin entre les sommets s_1 et s_2, c'est-à-dire le chemin nécessitant de passer par le moins d'arrêtes possibles. S'il n'existe pas de chemin entre s_1 et s_2, on note $\text{dist}(s_1, s_2) = \infty$.

Exemple

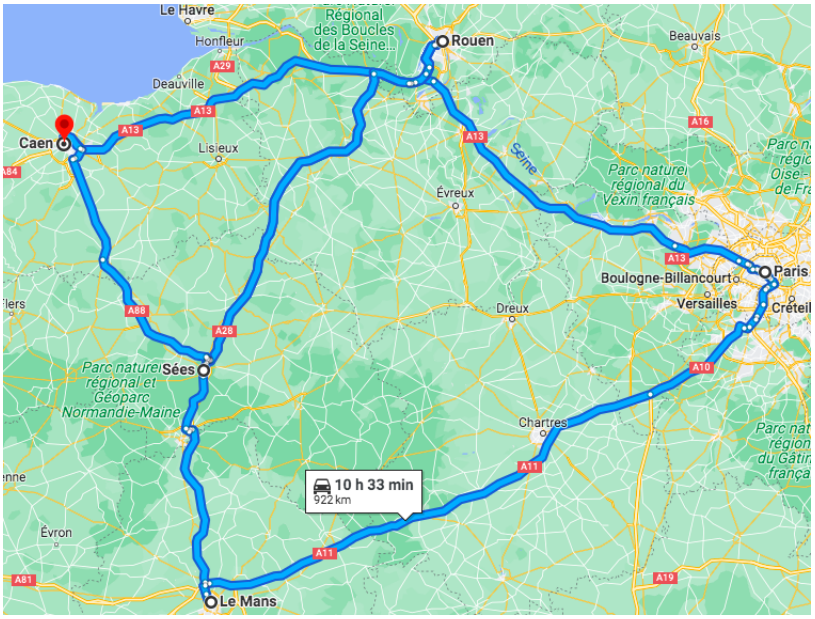
Dans l'exemple ci-dessus : $\text{dist}(A, E) = 2$, $\text{dist}(A, C) = 1$.

Exemple



Ici, on a par exemple :
- $\text{dist}(C, D) = 1$, $\text{dist}(A, B) = 1$.
- $\text{dist}(A, C) = \infty$.

Exercice 5



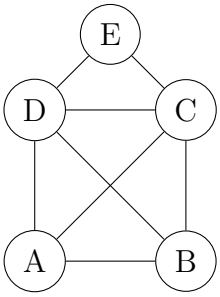
Représenter un graphe :
- dont les sommets sont les villes Caen, Le Mans, Paris, Rouen, Sées
- et dont les arrêtes sont des autoroutes reliant ces villes

Si une autoroute relie deux villes et qu'une troisième est située sur la même autoroute entre ces deux dernières, on ne représentera que deux arrêtes, pas trois.

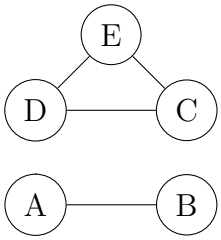
2.B Cycles, connexité dans un graphe

Définition 6

- On dit qu'un graphe $G = (S, A)$ est connexe si pour tous sommets $(s_i, s_j) \in S^2$, il existe un chemin reliant s_i et s_j.
- Si un graphe n'est pas connexe, on appelle composantes connexes les parties connexes de ce graphe.



Graphe connexe



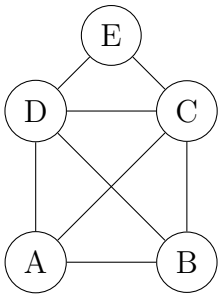
Graphe non connexe.
Ses composantes connexes sont :

$\{A, B\}$ $\{C, D, E\}$.

Définition 7

On appelle cycle, un chemin reliant un sommet à lui-même sans emprunter plusieurs fois une même arrête.

Exemple



Dans ce graphe :
- le chemin $(A, \{A, B\}, B, \{B, C\}, C, \{C, A\}, A)$ est un cycle
- le chemin $(A, \{A, B\}, B, \{B, C\}, C, \{C, B\}, B, \{B, A\}, A)$ n'en est pas un.

2.C Représentation algébrique et informatique des graphes

Définition 8

Soit $G = (S, A)$ un graphe.
On dit que deux sommets s_i et s_j sont adjacents s'il existe une arrête les reliant.

Remarques

On dit également que ces sommets sont voisins.

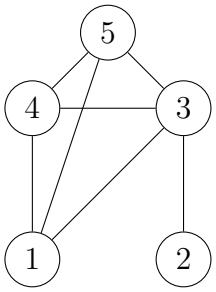
Définition 9

Soit $G = (S, A)$ un graphe avec $n = \text{Card}(S)$. On note $S = \{s_1, \ldots, s_n\}$.
On appelle matrice d'adjacence du graphe G, la matrice $M \in \mathscr{M}_n(\mathbb{R})$ telle que :
- $m_{ij} = 1$ si les sommets s_i et s_j sont adjacents.
- $m_{ij} = 0$ sinon.

Remarques

La matrice d'adjacence d'un graphe non orienté est symétrique.

Exemple



$$M = \begin{pmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{pmatrix}$$

Remarques

— La matrice d'adjacence d'un graphe sans boucle a une diagonale nulle.
— Si les sommets ne sont pas les entiers $1, \ldots, n$, il faudra choisir un ordre sur ces sommets pour écrire la matrice d'adjacence.

Exemple

On implémente les matrices d'adjacence avec Python à l'aide du module `numpy`.
Avec le graphe ci-dessus, on a :

```
1 L=[[0,0,1,1,1],[0,0,1,0,0],[1,1,0,1,1],[1,0,1,0,1],[1,0,1,1,0]] # liste
2
3 import numpy as np # tableau numpy
4 M=np.array([[0,0,1,1,1],[0,0,1,0,0],[1,1,0,1,1],[1,0,1,0,1],[1,0,1,1,0]])
```

Remarques

On peut aussi implémenter un graphe à l'aide d'un dictionnaire représentant les listes d'adjacence.
Avec le graphe ci-dessus :

```
1  import numpy as np
2
3  # On connait déjà la matrice d'adjacence :
4  M=np.array([[0,0,1,1,1],[0,0,1,0,0],[1,1,0,1,1],[1,0,1,0,1],[1,0,1,1,0]])
5
6  # ou avec un dictionnaire des listes d'adjacence :
7  D={}
8  D[1]=[3,4,5] # les voisins de 1 sont 3,4,5
9  D[2]=[3] # 2 a un seul voisin : 3
10 D[3]=[1,2,4,5] # . . .
11 D[4]=[1,3,5]
12 D[5]=[1,3,4]
```

Remarques

On verra en exercice que l'on peut passer de la représentation matricielle à la représentation sous forme de dictionnaire.

3 Graphes orientés

Définition 10

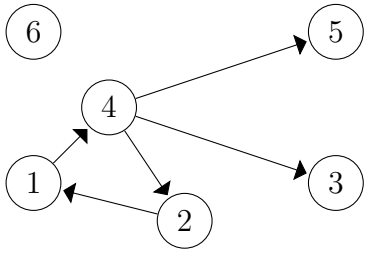
On appelle graphe orienté $G = (S, A)$ la donnée de deux ensembles :
- un ensemble de sommets $S = \{s_1, \dots, s_n\}$.
- un ensemble d'arcs $A = \{a_1, \dots, a_m\}$.

Un arc relie deux sommets de G. On note $a_i = (s_k, s_\ell)$ si l'arc a_i relie les sommets s_k et s_ℓ avec l'orientation $s_k \to s_\ell$.

Remarques

On utilise la notation (s_k, s_ℓ) pour mettre l'accent sur le fait que l'ordre est important dans un graphe orienté. L'arc (s_k, s_ℓ) a pour point de départ (ou **origine**) le sommet s_k et pour point d'arrivée (ou **extrémité**), le sommet s_ℓ.

Exemple



Définition 11

Soit $G = (S, A)$. Soit $s \in S$ un sommet.
- On note $d_+(s)$ le nombre d'arcs dont le point d'arrivée est le sommet s. On appelle $d_+(s)$ le degré entrant de s.
- On note $d_-(s)$ le nombre d'arcs dont le sommet de départ est le sommet s. On appelle $d_-(s)$ le degré sortant de s.
- On note $d(s) = d_+(s) + d_-(s)$ le nombre d'arcs partant ou arrivant à s. On appelle $d(s)$ le degré total de s.

Exemple

Ci-dessus, on a par exemple $d(4) = 4$ car :
- $d_+(4) = 1$
- $d_-(4) = 3$

Définition 12

Soit $G = (S, A)$ un graphe orienté. La matrice d'adjacence $M = (m_{ij})_{1 \leqslant i,j \leqslant n}$ d'un graphe orienté est définie par :

$$m_{i,j} = \begin{cases} 1 & \text{si il existe une arrête de } s_i \text{ vers } s_j \\ 0 & \text{sinon} \end{cases}$$

Remarques

Cette matrice n'est pas nécessairement symétrique comme dans le cas d'un graphe non orienté.

Exemple

Dans le graphe orienté ci-dessus :

$$M = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

```
1 D={}
2 D[1]=[4]
3 D[2]=[1]
4 D[3]=[]
5 D[4]=[2,3,5]
6 D[5]=[]
7 D[6]=[]
```

Exercice 13

On a représenté certains liaisons aériennes, à un instant t, reliant des villes européennes.
Donnez la matrice d'adjacence et le dictionnaire des listes d'adjacence du graphe orienté représentant la situation.
On utilisera l'ordre alphabétique pour construire la matrice d'adjacence.

4 Parcours dans un graphe

On va étudier deux méthodes de parcours :

— Parcours en largeur (Breadth-First Search).
On part d'un sommet s et on découvre tous les voisins de s. Puis on découvre les voisins non visités de ces voisins, etc. On dit que l'on privilégie les sommets frères lors du parcours.
— Parcours en profondeur (Depth-First search).
On part d'un sommet s et on visite un voisin (arbitrairement choisi) de s. On visite alors un voisin non visité de ce voisin, etc. On dit que l'on privilégie les sommets fils lors du parcours en profondeur.

4.A Parcours en largeur - Breadth-First Search (BFS)

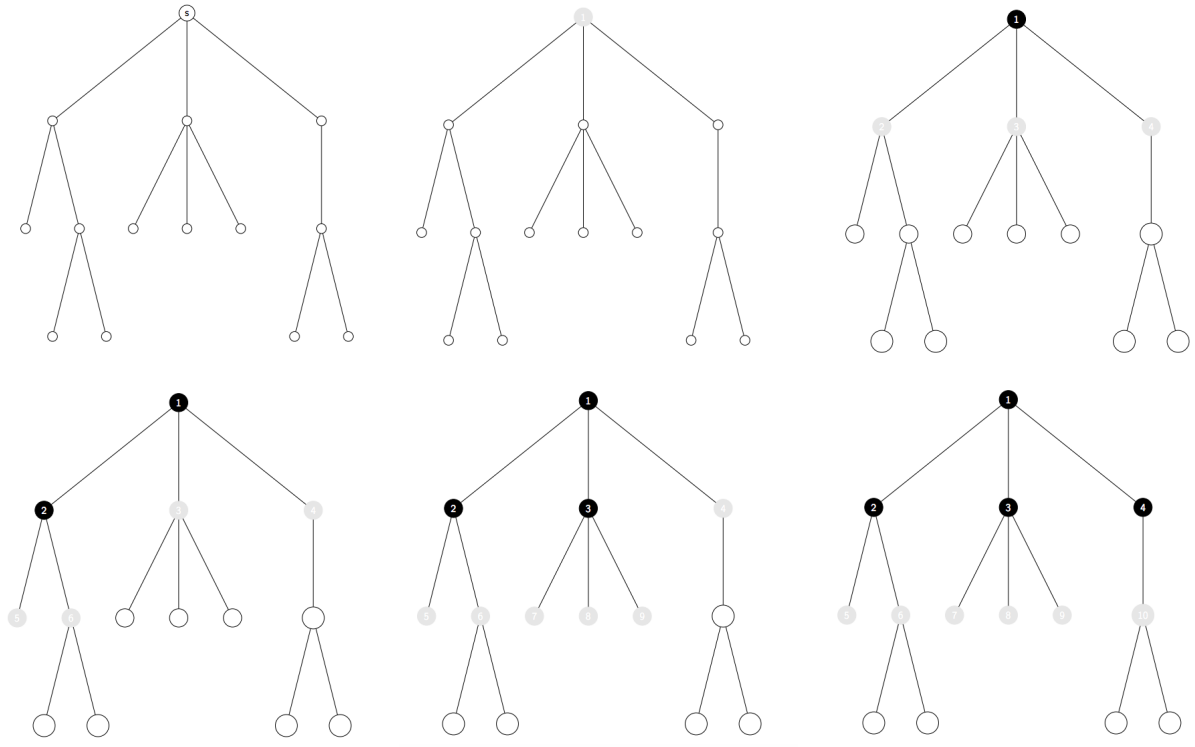
Illustrons cette méthode de parcours par un exemple d'utilisation de la notion de graphe : on souhaite parcourir toutes les pages d'un site internet. Les sommets du graphe sont les pages du site, et un lien entre deux pages est une arrête de ce graphe.

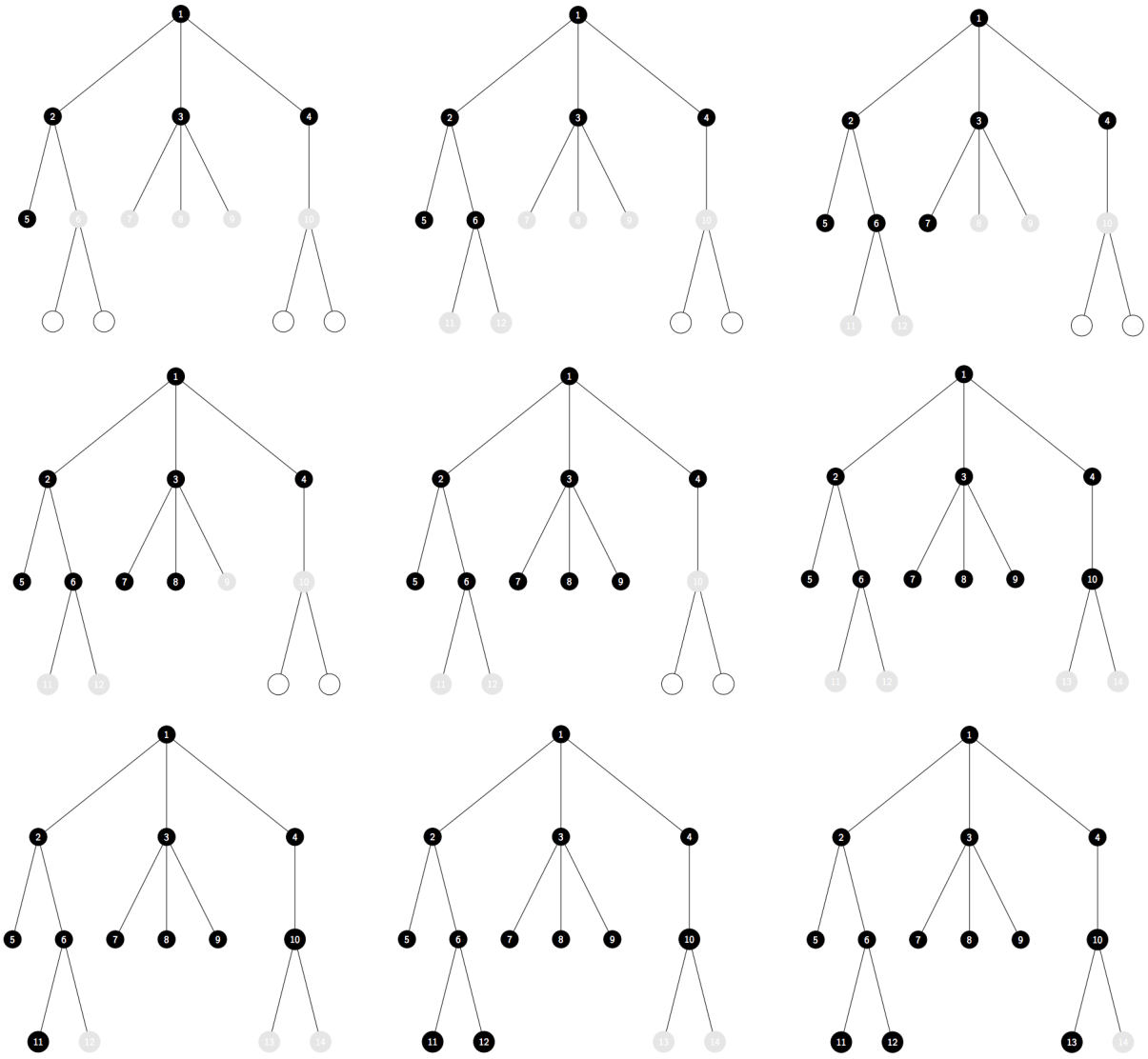
1. On place le sommet correspondant à la page index s_1 du site dans une liste `L`.
2. On découvre les voisins non visités de ce sommet et on les place à la suite de la liste `L`.
3. On supprime le sommet s_1 (page index).
4. On recommence au point 2. tant qu'il reste des éléments dans la liste `L` avec les sommets placés dans `L`.

On traite donc en priorité les liens des pages les plus tôt découvertes.

Remarques

Une manière de comprendre cette méthode de parcours est également de donner une notion de distance au sommet origine. On commence par parcourir les pages à une distance 1 de la page index. Puis on parcourt les pages à une distance 2 et ainsi de suite.





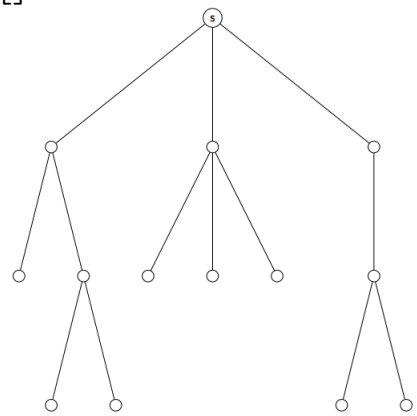
Remarques

— On a coloré en noir les noeuds passés par la liste `L` puis supprimés de la liste : un noeud passe au noir lorsque tous ses voisins ont été découverts.
— On a coloré en gris les noeuds présents dans la liste `L`.
— Après la dernière étape, le noeud 14 sera bien entendu coloré en noir.

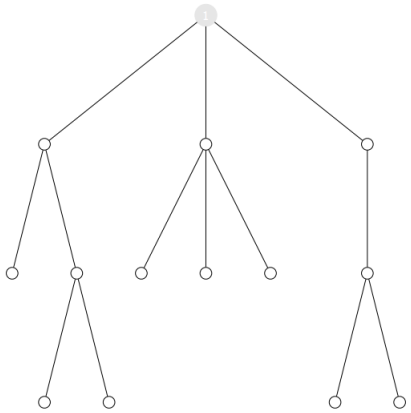
Remarques

Décrivons le contenu de la liste L au cours des premières étapes de l'algorithme.

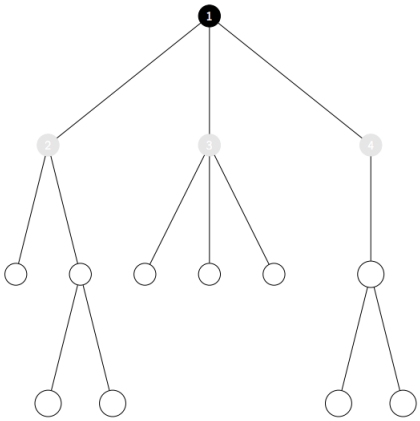
`L=[]`



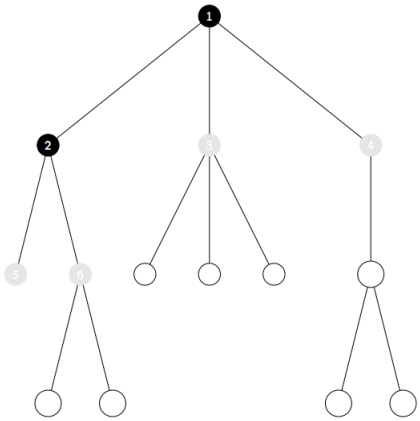
`L=[1]`



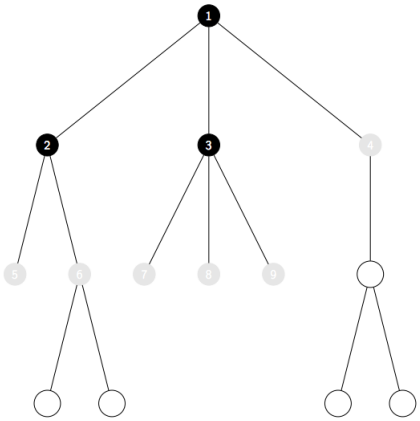
`L=[2,3,4]`



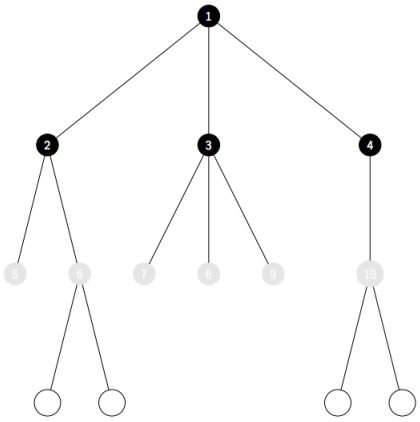
`L=[3,4,5,6]`



`L=[4,5,6,7,8,9]`



`L=[5,6,7,8,9,10]`



Exercice 14

Poursuivre la description du contenu de la liste `L` lors du parcours en largeur du graphe ci-dessus.

Remarques

— Le principe de fonctionnement de la liste `L` est celui d'une **file** : le premier sommet (1) à entrer dans la liste est le premier à être supprimé.
— Le deuxième sommet à être supprimé est celui entré en deuxième dans la liste (c-à-d 2).
— Et ainsi de suite.

Cette méthode suit le principe **first-in first-out** (FIFO).
— Lorsqu'un élément est ajouté à la file, on dit qu'il est **enfilé**.
— Lorsqu'un élément est supprimé de la file, on dit qu'il est **défilé**.

On peut imaginer une file d'attente (à un guichet par exemple) pour se représenter le fonctionnement d'une file.

L'homme à la cravate rouge est défilé, l'homme de dos en chemise blanche est enfilé.

Remarques

On utilisera le module deque. Ce module fonctionne comme une liste mais permet de défiler le premier élément de la file (élément en tête de file, d'indice 0) avec une complexité $O(1)$. Illustration avec la file précédente :

```
1  from collections import deque
2  L=deque([]) # file vide
3  L.append(1) # on enfile 1 en tête de file
4  L.popleft() # on défile 1 i.e. on supprime 1 de la file
5  L.append(2) # on enfile 2 en tête de file : nouvelle tête de file
6  L.append(3) # on enfile 3 à la suite de 2
7  L.append(4) # on enfile 4 à la suite de 3
8  L.popleft() # on défile 2 : 3 est donc la nouvelle tête de file
9  L.append(5) # on enfile 5 à la suite de 4
10 L.append(6) # on enfile 6 à la suite de 5
```

Variables à manipuler lors de l'implémentation de l'algorithme BFS avec Python

- On va écrire une fonction `BFS(G,s)` parcourant le graphe en largeur à partir d'un sommet `s` entré en paramètre.
- On représente le graphe $G = (S, A)$ à l'aide d'un dictionnaire.
- L'algorithme renverra un autre dictionnaire `P` tel qu'en fin de parcours `P[s]` sera le père (ou prédécesseur) du sommet `s`, c'est-à-dire le sommet à partir duquel le sommet `s` a été découvert lors du parcours.
  On a par exemple dans le parcours des pages du site ci-dessus : `P[2]=1`, `P[14]=10`.
- On utilisera une file `L` initialisée à `L=[s]` où `s` est le sommet de départ.

L'algorithme en `pseudo code` :

```
fonction BFS(G,s):
    """ Renvoie le dictionnaire du prédécesseur de chaque sommet
    lors du parcours en largeur du graphe G depuis le sommet origine s"""

    Initialiser la file L à [s]
    P={s : None} # s, sommet de départ, n'a pas de prédécesseur.
    visites={} # dictionnaire des sommets visités : 3 couleurs ...
                                                  # ... Blanc,Gris,Noir
    Pour tout sommet u de G:
        visites[u]="Blanc"

    Tant que L est non vide:
            Défiler la tête de la file L dans une variable u # L.popleft()
            Si visites[u]!="Noir": # Blanc ou Gris
                visites[u]="Noir" # sommet visité
                Pour tout voisin "Blanc" v du sommet u dans G:
                        visites[v]="Gris" # en cours de visite
                        Poser P[v]=u # le prédécesseur de v est u
                        Enfiler v dans L
    Renvoyer P
```

Théorème 15: Complexité

La complexité (dans le pire cas) de l'algorithme de recherche en largeur est $O(n+m)$ où n est le nombre de sommets et m le nombre d'arrêtes (ou d'arcs dans le cas orienté) du graphe parcouru.

4.B Parcours en profondeur - Depth-First Search (DFS)

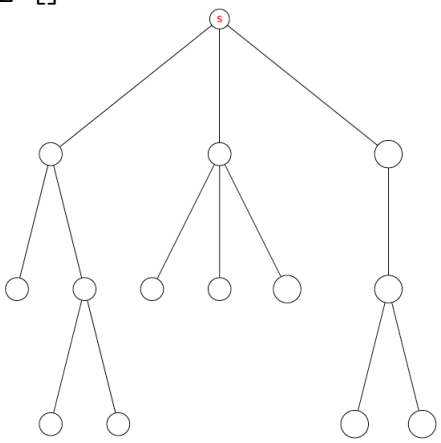
On reprend l'exemple du parcours des pages d'un site internet. Les sommets sont les pages du site et un lien entre deux pages est une arrête du graphe.

1. On place le sommet correspondant à la page index s_1 du site dans une liste `L`.
2. Si ce sommet présente des voisins qui ne sont pas dans `L` alors on sélectionne l'un de ces voisins et on le place à la suite de `L`

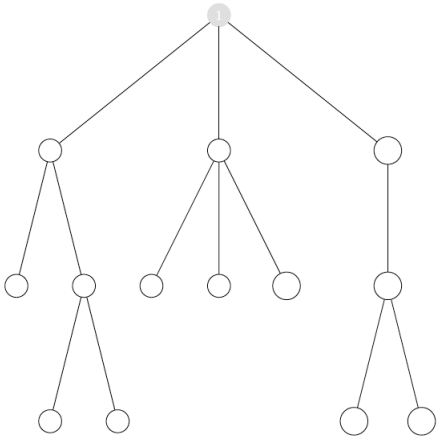
3. Sinon, on supprime cet élément de la liste `L`.
4. On recommence au point 2 tant que la liste `L` n'est pas vide.

On traite donc en priorité les liens des pages les plus tard découvertes.
Décrivons l'évolution de la liste `L` lors du parcours en largeur du graphe.

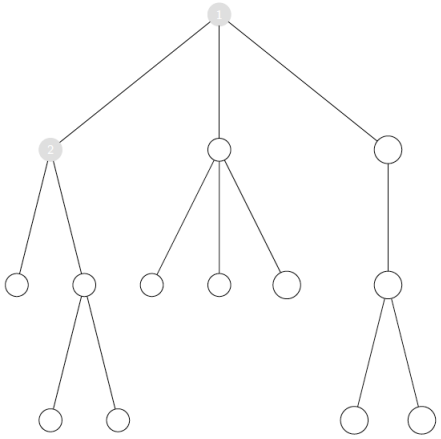
`L=[]`



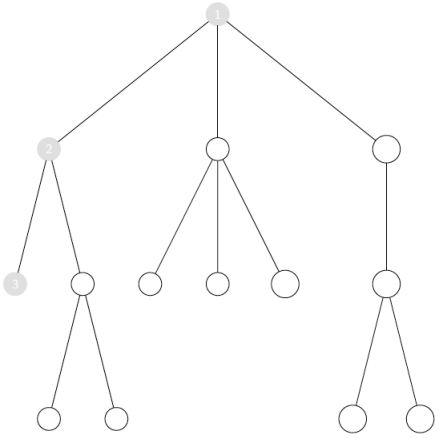
`L=[1]`



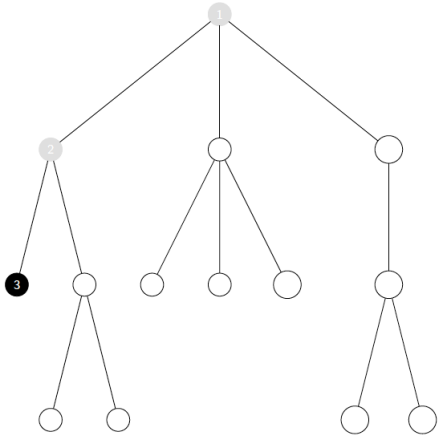
`L=[1,2]`



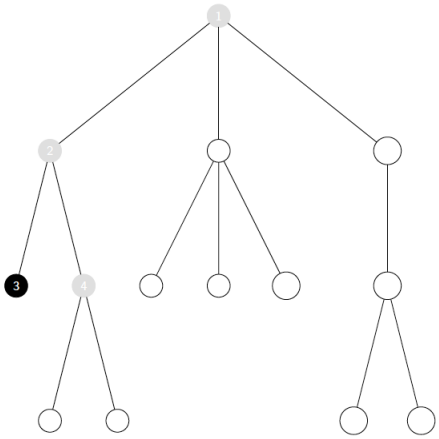
`L=[1,2,3]`



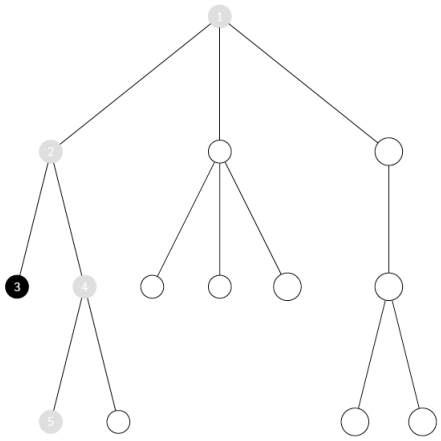
`L=[1,2]`



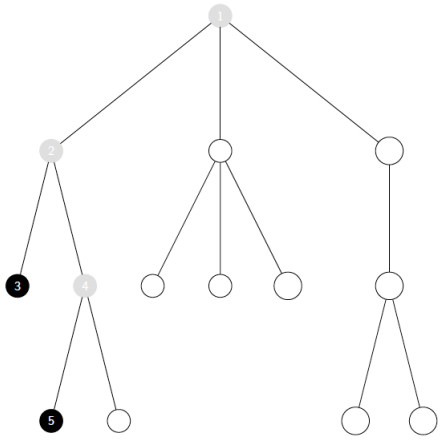
`L=[1,2,4]`



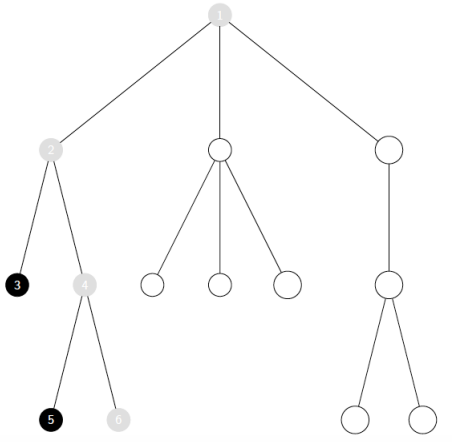
`L=[1,2,4,5]`



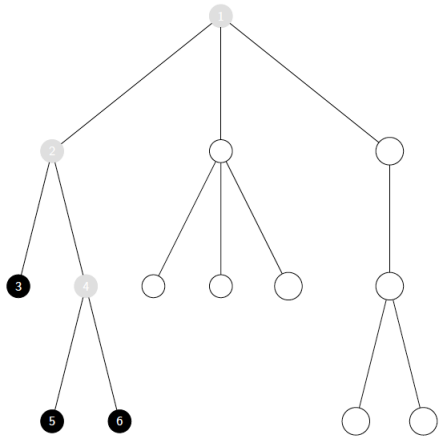
`L=[1,2,4]`



`L=[1,2,4,6]`



`L=[1,2,4]`



Exercice 16

Poursuivre la description du contenu de la liste `L` lors du parcours en profondeur du graphe ci-dessus.

Remarques

- Le principe de fonctionnement de la liste `L` est celui d'une **pile** : le premier sommet (1) à entrer dans la liste est le dernier à être supprimé.
- Le sommet (2) est le deuxième sommet à entrer dans la liste et en sera supprimé après tous ses noeuds fils : (3), (4), (5), (6).
- Et ainsi de suite.

Cette méthode suit le principe **last-in first-out** (LIFO).

- Lorsqu'un élément est ajouté à la file, on dit qu'il est **empilé**.
- Lorsqu'un élément est supprimé de la file, on dit qu'il est **dépilé**.

On peut imaginer une pile d'assiette pour se représenter le fonctionnement d'une pile. La dernière assiette sale à être empilée, sera la première à être lavée (traitée puis supprimée de la pile).
Dessin.

Remarques

On utilisera également le module deque par cohérence avec l'implémentation des files.

```
L=deque([]) # pile vide
L.append(1) # on empile 1 au sommet de la pile
L.append(2) # on empile 2 sur 1 : 2 est le nouveau sommet de la pile
L.append(3) # on empile 3 sur 2 : 3 est le nouveau sommet de la pile
L.pop() # on dépile 3 : 2 est le nouveau sommet de la pile
L.append(4) # on empile 4 sur 2 : 4 est nouveau sommet de la pile
L.append(5) # on empile 5 sur 4 : 5 est le nouveau sommet de la pile
L.pop() # on dépile 5 : 4 est le nouveau sommet de la pile
L.append(6) # on empile 6 sur 4
```

Variables à manipuler lors de l'implémentation de l'algorithme DFS avec Python

1. On va écrire une fonction `DFS(G,s)` parcourant le graphe en profondeur à partir d'un sommet `s` entré en paramètre.
2. On représente le graphe $G = (S, A)$ à l'aide d'un dictionnaire.
3. L'algorithme renverra un autre dictionnaire `P` tel qu'en fin de parcours `P[s]` sera le père (ou prédécesseur) du sommet `s`, c'est-à-dire le sommet à partir duquel le sommet `s` a été découvert lors du parcours.

   On a par exemple dans le parcours des pages du site ci-dessus : `P[2]=1`, `P[6]=4`.
4. On utilisera une Pile `L` initialisée à `L=[s]` où `s` est le sommet de départ.

L'algorithme en `pseudo code` :

```
Entrée : Un graphe G et un sommet de départ s

Sortie : le dictionnaire des prédécesseurs de chaque sommet ...
... lors du parcours en profondeur du graphe G


fonction DFS(G,s):
    """ Renvoie le dictionnaire du prédécesseur de chaque sommet
    lors du parcours en profondeur de G depuis le sommet origine s"""

    Initialiser la Pile L à [s]
    P={s : None} # s, sommet de départ, n'a pas de prédécesseur
    visites={}
    Pour tout u dans G:
        visites[u]="Blanc" # blanc

    visites[dep]="Gris" # gris
```

```
18
19      Tant que L est non vide:
20              u=L[-1] # sans dépiler
21              R=liste des voisins "Blancs" de u # non découverts
22              Si R est non vide:
23                  On choisit un sommet v dans R # le premier par exemple
24                  P[v]=u # le prédécesseur de v est u
25                  On empile v au sommet de L
26                  visites[v]="Gris"
27              Sinon:
28                  On dépile L
29                  visites[u]="Noir"
30      On renvoie P
```

Théorème 17: Complexité

La complexité (dans le pire cas) de l'algorithme de recherche en profondeur est $O(n+m)$ où n est le nombre de sommets et m le nombre d'arrêtes (ou d'arcs dans le cas orienté) du graphe parcouru.

Exercice 18

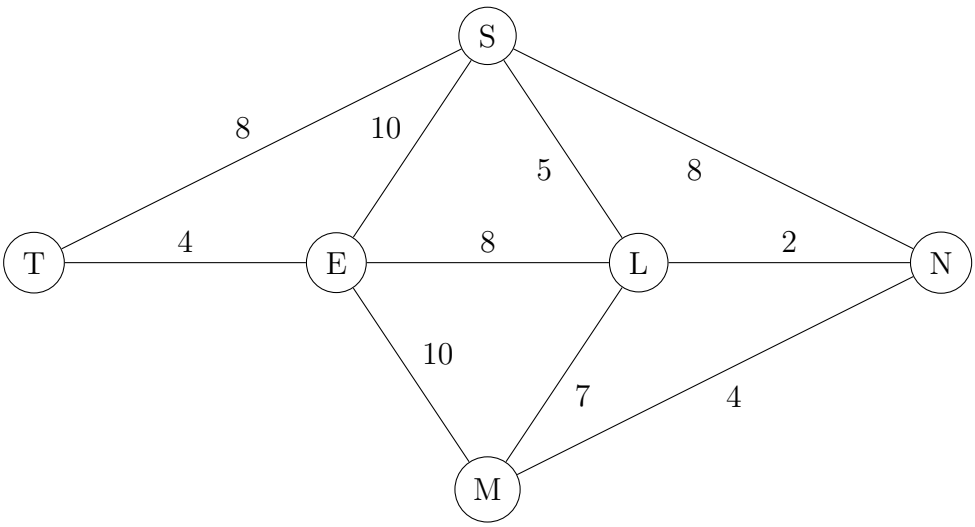
Utiliser un algorithme de parcours pour obtenir, s'il existe, un chemin entre deux sommets d'un graphe.

5 Algorithmes du plus court Chemin

5.A Algorithme de Dijkstra

5.A.1 Graphe pondéré

Considérons le graphe ci-dessous :



On constate que les arrêtes sont étiquetées par de nombres (entiers) positifs. On dit qu'il s'agit d'un graphe pondéré. Ils sont utiles lorsqu'on cherche à modéliser un réseau dont les noeuds sont connectés par des arrêtes plus ou moins simples à parcourir.

L'exemple le plus courant est peut-être celui d'un réseau routier où les sommets représentent des villes et les arrêtes représentent des routes pondérées par la distance entre chaque ville du réseau.

On peut stocker un tel graphe à l'aide d'une matrice d'adjacence en remplaçant les coefficients égaux à 1 par la pondération étiquetant l'arrête. On également stocker un tel graphe avec un dictionnaire dont chaque clef est un sommet `s` et les valeurs associées sont les listes de couples `voisin,distance_s`.

```
1 D={}
2 D['S']=[('T',8),('E',10),('L',5),('N',8)]
3 D['T']=[('S',8),('E',4)]
4 D['E']=[('T',4),('S',10),('L',8),('M',10)]
5 D['L']=[('E',8),('S',5),('N',2),('M',7)]
6 D['N']=[('S',8),('L',2),('M',4)]
7 D['M']=[('E',10),('L',7),('N',4)]
```

5.A.2 Principe de l'algorithme de Dijkstra et exemple

L'algorithme de Dijkstra permet de déterminer le plus court chemin entre deux sommets d'un graphe pondéré. C'est par exemple cet algorithme (ou certaines de ses déclinaisons) qui permet, à l'aide du système GPS, de déterminer l'itinéraire le plus cours entre deux villes.

Remarques

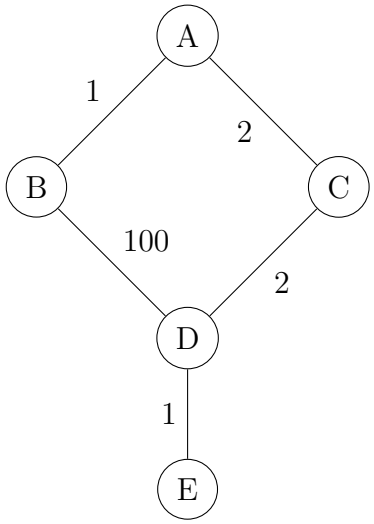
Le graphe dans ce cas là est très naturel :
- Les sommets du graphe sont les villes.
- Les arrêtes sont les routes reliant ces villes.
- Le graphe est pondéré par la distance reliant chacune des villes.

L'idée générale de l'algorithme est la suivante :
- On part d'une origine s_0.
- On détermine le voisin s_1 le plus proche de s_0.
- On recommence avec s_1 en déterminant le voisin s_2 (différent de s_0) le plus proche de s_1.

Avant de continuer, notons que cette recherche gloutonne voisin après voisin ne sera pas optimale et devra être nuancée.

Exemple

Dans le graphe suivant, on souhaite relier le sommet A au sommet E via le chemin le plus court :



En avançant de manière naïve avec une démarche gloutonne, le premier chemin exploré reliant le sommet A au sommet E a une longueur 102.
Cette distance n'est pas minimale car le chemin via C a une longueur 5.
Au cours de la recherche, on notera $D_A^t(D) = 101$ la distance minimale temporaire d'un chemin reliant A à D, c'est-à-dire une distance déterminée avant que tous les chemins n'aient été explorés.
On notera par opposition $D_A^f(D) = 4$ la distance minimale (finale) d'un chemin reliant A à D : c'est-à-dire lorsque tous les chemins auront été explorés.
On aura bien entendu : $D_A^f(E) = 5$.

Le fonctionnement de l'algorithme est le suivant :

❶ On part du sommet s_0 appelé origine. On note $D_{s_0}^f(s_0) = 0$ la longueur du chemin le plus court reliant s_0 à s_0 (bien entendu cette distance est nulle).
Le sommet s_0 ne sera plus utilisé ensuite.

❷ On détermine le voisin s_1 le plus proche de s_0 :

$$D_{s_0}^f(s_1) = \min_{s \in V(s_0)} D_{s_0}^f(s).$$

Tout chemin passant par un autre sommet avant d'atteindre s_1 sera nécessairement plus long car s_1 a été choisi pour minimiser les distances $D_{s_0}^f(s)$.
Le sommet s_1 ne sera plus utilisé ensuite.

❸ On calcule alors $D_{s_0}^f(s_1) + D_{s_1}^f(s)$ pour chaque voisin $s \in V(s_1)$. Si cette distance est inférieure à l'ancienne distance $D_{s_0}^t(s)$, on procède alors à la mise à jour :

$$D_{s_0}^t(s) \leftarrow D_{s_0}^f(s_1) + D_{s_1}^f(s).$$

Les distances $D_{s_0}^t(s)$ pour $s \notin V(s_1)$ sont inchangées.

❹ On sélectionne alors s_2 tel que :

$$D_{s_0}^t(s_2) = \min_{s \in G \setminus \{s_0, s_1\}} D_{s_0}^t(s).$$

Attention, à cette étape s_2 n'est pas nécessairement un voisin de s_1.

On note alors $D^f_{s_0}(s_2) = D^t_{s_0}(s_2)$ et le sommet s_2 ne sera plus utilisé.
En effet, il n'existe pas de chemin plus court reliant s_0 à s_2 car par définition :

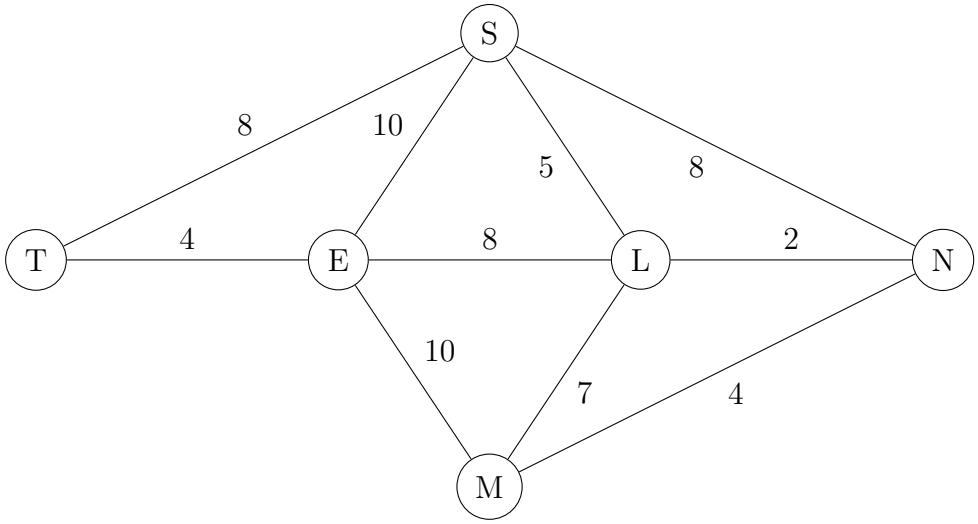
$$D^f_{s_0}(s_2) \leqslant D^f_{s_0}(s_1) + D^f_{s_1}(s),\ \text{pour tout } s \in V(s_1).$$

❹ On recommence au point ❸ avec le sommet déterminé à l'étape précédente.

Remarques

Il sera donc nécessaire de construire, **et de tenir à jour**, un dictionnaire des **distances minimales** de chaque sommet au **sommet origine** s_0.
Ces distances minimales seront des valeurs temporaires potentiellement modifiées, si un chemin plus court apparait en cours de traitement.

Regardons le traitement du graphe suivant où l'on cherche le chemin le plus court de S à M :



Distance min à l'origine	S	T	E	L	N	M	
	0	∞	∞	∞	∞	∞	Initialisation
		8(S)	10(S)	5(S)	8(S)	∞	Itération 1
		8(S)	10(S)		7(L)	12(L)	Itération 2
		8(S)	10(S)			11(N)	Itération 3
			10(S)			11(N)	Itération 4
						11(N)	Itération 5

Exercice 19

Expliquer chacune des lignes du tableau ci-dessus.

5.A.3 Files de priorité

À chaque itération de l'algorithme de Dijkstra, il est nécessaire de déterminer le sommet accessible le plus proche de l'origine. Dans le paragraphe précédent, on a par exemple explicité cette recherche lorsqu'on cherchait s_2 :

$$D^t_{s_0}(s_2) = \min_{s\in G\setminus\{s_0,s_1\}} D^t_{s_0}(s)$$

On va donc construire à mesure que l'on parcourt le graphe, la liste des sommets accessibles en les étiquetant par leur distance à l'origine. Cette distance sera potentiellement modifiée au cours de l'algorithme comme nous l'avons déjà remarqué.

Dans l'exemple étudié dans le paragraphe précédent, cette liste contient initialement :

`L=[(S,0)].`

Puis, le sommet `S` étant visité, la liste `L` devient :

`L=[(E,10),(T,8),(N,8),(L,5)].`

On a supprimé le sommet `S` et ajouté ses voisins (sommets accessibles à cet instant de l'algorithme) étiquetés par leur distance au sommet origine `S`.

Remarquons, c'est un choix important, que l'on a ordonné cette liste suivant les distances décroissantes de l'origine. Ainsi, lorsqu'on veut accéder au sommet le plus proche de l'origine, il suffit d'utiliser la commande `L.pop()`.

On aurait pu organiser suivant les distances croissantes à l'origine mais il aurait fallu utiliser la commande `L.popleft()` réservée au type `deque`. Ici, une simple liste suffira à simuler ce qu'on appelle une **file de priorité**.

Cette structure de données généralise celles de piles et de files. Une file de priorité permet encore de stocker de l'information mais également d'introduire un ordre de priorité sur ces données différent du premier (ou dernier) arrivé.

Dans notre cas, c'est la distance minimale à l'origine qui jouera le rôle de priorité : l'élément de plus forte priorité, placé en fin de liste, aura une distance minimale à l'origine.

```
1  def filePrio():
2      """ stocke les couples (sommet, distance origine)"""
3      return []
4
5  def empty(file):
6      return file==[]
7
8  def delete(file,element):
9      for i in range(len(file)):
10         if file[i][0]==element:
11             return file[0:i]+file[i+1:len(file)]
12     return file
13
14
15 def put(element,priorite,file):
16     """ajoute par ordre décroissant de distance_origine
```

```
17        la priorité est donnée à l'élément en fin de file"""
18        for i in range(len(file)):
19             e,l=file[i]
20             if priorite>l:
21                  return file[:i]+[(element,priorite)]+file[i:]
22        return file+[(element,priorite)]
23
24
25  def get(file):
26        """récupère l'élément de plus forte priorité"""
27        return file.pop()
```

5.A.4 Implémentation de l'algorithme de Dijkstra

```
1   """
2   Entrée : un graphe G, un sommet de départ (dep), un sommet de fin (fin)
3
4   Sortie :
5    - False s'il n'existe pas de chemin reliant dep à fin
6    - Le chemin le plus court si l'on peut relier dep à fin
7   """
8
9   def DIJKSTRA(G,dep,fin):
10        file=filePrio()
11        file=put(dep,0,file) # (sommet de départ, distance=0 à dep)
12        distances_origine={dep:0} # les autres sont infinies
13        for v in G:
14             if v!=dep:
15                  distances_origine[v]=float("inf")
16        parents={dep:None} # dictionnaire des parents
17        chemin=[] # chemin à construire le plus court de dep à fin
18        while not empty(file):
19             (s,d)=..............
20             if s!=fin:
21                  for v,delta in G[s]:
22                       distance=..............
23                       if distance<..............
24                            distances_origine[v]=..............
25                            file=delete(file,v)
26                            file=put(..............)
27                            parents[v]=..............
28             else: # création du chemin
29                  chemin=..............
30                  while s!=dep:
```

```
31                         s=.............
32                         chemin=[s]+chemin
33                     return distances_origine[fin],chemin
34         return False
```

Exercice 20

Compléter l'algorithme à trous ci-dessus.

5.A.5 Complexité

Théorème 21

Soit G un graphe composé de n sommets et m arrêtes. La complexité (dans le pire des cas) de l'algorithme de Dijkstra est $O(n^2)$.
Cette complexité peut être améliorée en $O(m + n\ln(n))$ avec certaines structures de données, appelées tas, que nous ne traitons pas dans ce cours.

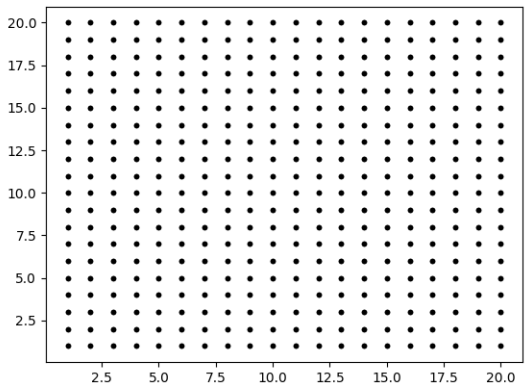
5.B Algorithme A^*

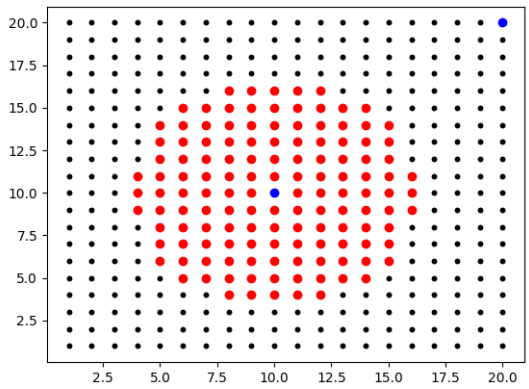
Dans ce paragraphe, on va présenter une amélioration de l'algorithme de Dijkstra permettant d'optimiser, dans la plupart des situations, le temps de recherche du plus court chemin.

L'idée générale est la même que dans l'algorithme de Dijkstra : à chaque itération de l'algorithme, on détermine les voisins à distance minimale du sommet origine. Dans le cas où plusieurs sommets sont candidats, l'algorithme de Dijkstra fait un choix arbitraire (ou aléatoire).

Considérons le graphe $G = (S, A)$ dont les sommets sont les couples $(i, j) \in [\![0, 20]\!]^2$, chaque sommet (i, j) (à part au bord) étant relié à 8 sommets voisins :

$$(i-1, j), (i+1, j), (i, j+1), (i, j-1), (i+1, j+1), (i+1, j-1), (i-1, j-1), (i-1, j+1).$$

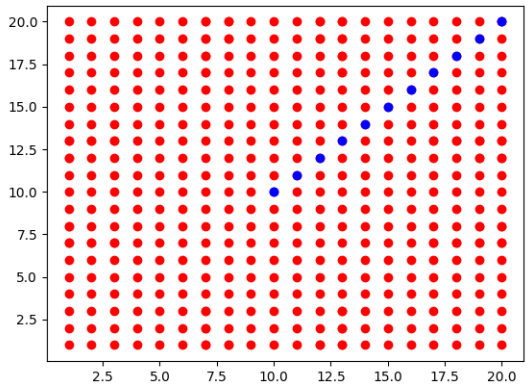




Si l'objectif est de partir du centre $(10, 10)$ du graphe et de rejoindre, par exemple, le point $(20, 20)$, l'algorithme de Dijkstra va effectuer une recherche circulaire en ne privilégiant aucun sommet situé à

distance minimale de l'origine. L'algorithme effectue donc des retours continuels vers des sommets à distance minimale de l'origine.

Pourtant, il y a clairement un cap. Si l'on doit rejoindre, sur un radeau, le continent en partant d'une île isolée dans l'océan, il est plus efficace de se fixer un cap que de rayonner de manière circulaire (en spirale) autour de l'île. Dans l'exemple ci-dessus, tous les sommets sont visités avant de déterminer le plus court chemin.



Avec un réseau de n^2 sommets, on obtient clairement une complexité quadratique $O(n^2)$.

Il faut donc introduire une notion de cap à l'algorithme de Dijkstra. Concrètement, le choix du sommet à placer au sommet de la file de priorité va être motivé par ce qu'on appelle une **heuristique**, une fonction `h` éclairant le choix du meilleur sommet.

Dans l'exemple ci-dessus, une heuristique possible est la distance à vol d'oiseau reliant un sommet (i, j) au but $(20, 20)$, c'est-à-dire la distance euclidienne :

$$\texttt{h((i,j))} = \sqrt{(20-i)^2 + (20-j)^2}.$$

Pour implémenter cette idée, on va maintenir à jour :

— un dictionnaire `distances_origine` (variant) des distances minimales, de chaque sommet, à l'**origine** (comme pour l'algorithme de Dijkstra)

— un dictionnaire `h` (invariant) des distances euclidiennes de chaque sommet au **but**.

— un dictionnaire `score` (variant) des scores, somme des informations précédentes :

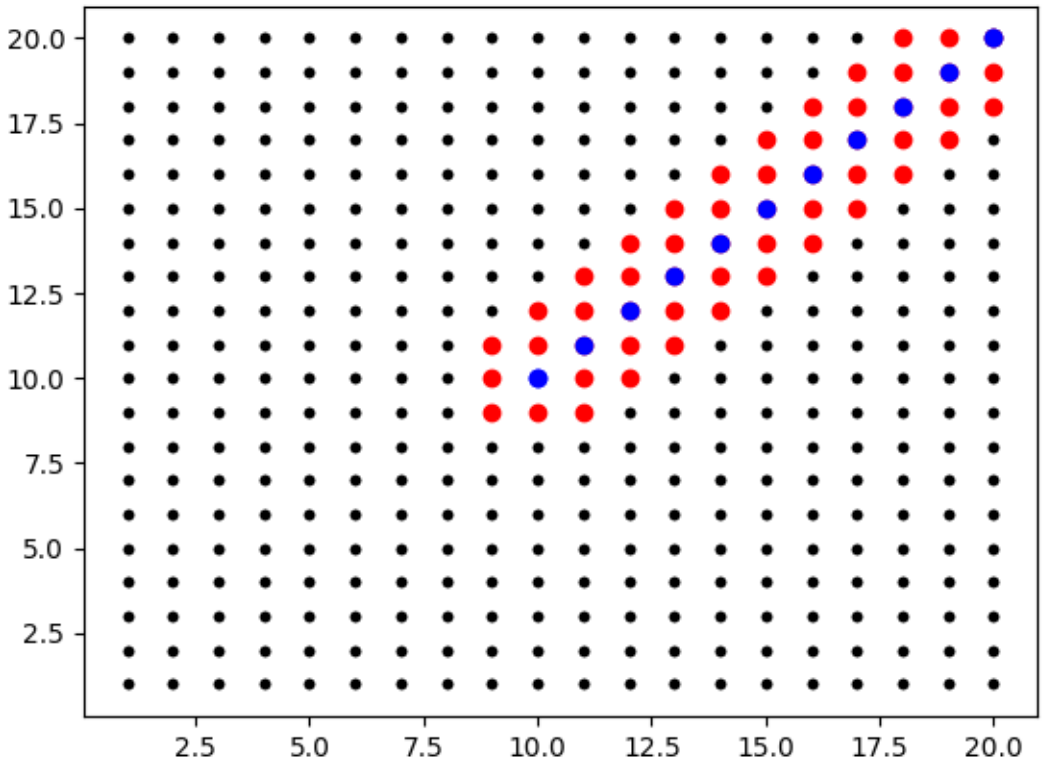
`score[s]=h[s]+distance_origine[s]`.

```
1 """
2 Entrée : un graphe G, un sommet de départ (dep), un sommet de fin (fin)
3
4 Sortie :
5  - False s'il n'existe pas de chemin reliant dep à fin
6  - Le chemin le plus court si l'on peut relier dep à fin
7 """
8
9 def algoAstar(G,dep,fin):
```

```
10      distances_origine={}
11      distances_origine[dep]=0
12      h={} # différence
13      h[dep]=euclide(dep,fin) # différence
14      scores_min={} # différence
15      scores_min[dep]=h[dep] +distances_origine[dep] # différence
16      for v in G:
17           if v!=dep:
18                distances_origine[v]=float("inf")
19                h[v]=euclide(v,fin) # différence
20                scores_min[v]=h[v] +distances_origine[v] # différence
21      file=filePrio()
22      file=put(dep,scores_min[dep],file)
23      parents={dep:None}
24      while not empty(file):
25           (s,d)=..............
26           if s!=fin:
27                for v,delta in G[s]:
28                     distance=..............
29                     if distance<..............
30                          distances_origine[v]=..............
31                          scores_min[v]=.............. # différence
32                          file=delete(file,v)
33                          file=put(..............) # différence
34                          parents[v]=..............
35           else:
36                chemin=..............
37                while s!=dep:
38                     s=..............
39                     chemin=[s]+chemin
40                return distances_origine[fin],chemin
41      return False
```

Exercice 22

Compléter le code à trous ci-dessus. Les différences avec l'algorithme de Dijkstra sont signalées. On suppose que l'on dispose d'une fonction `euclide(s_1,s_2)` renvoyant la distance euclidienne entre deux sommets `s_1,s_2`



La recherche dans ce cas-là a un coût linéaire $O(n)$.