

Informatique Tronc Commun

Correction du DS n°4

Lycée Fabert de Metz, MPSI – PCSI (2025 – 2026)

Barème de correction (total : 37 pt)

Respect syntaxique (sous forme de malus)

Le correcteur contrôle la syntaxe sur l'ensemble de la copie, en repérant les erreurs suivantes (erreurs qui seraient directement signalées par l'interpréteur Python lors de l'exécution du code) :

Erreurs de syntaxe détectées	
	Non respect des « : » (dans les définitions de fonction, les boucles, les instructions conditionnelles)
	Non respect des mots-clés (mot-clé oublié, For / If au lieu de for / if par exemple, ...)
	Non respect de la typographie (confusion entre parenthèses et crochets, par exemple L(i) au lieu de L[i] pour une liste, range[n] au lieu de range(n), fonction[args] au lieu de fonction(args))
	Autres erreurs (par exemple L = L.append(a) au lieu de L.append(a), confusion entre « = » et « == », entre « <= » et « < », affectation val = Nom au lieu de Nom = val, ...)

Le malus appliqué à la note finale est déterminé en fonction du nombre de critères non respectés. Il s'élève à 25 % de la note si l'étudiant n'a respecté aucun des quatre critères.

A. Implémentation du graphe (total : 8 pt)

A.1. Implémentation de la fonction `degre_In(Routes, A)` :

- respect de la spécification de la fonction (structure `def ...(...) : / return int`, indentation) **(0,5 pt)**
- énumération cohérente de la liste `Routes` **(0,5 pt)**
- calcul correct du degré sortant (initialisation + incrémentation sous condition) **(1 pt)**
- complexité temporelle avec justification succincte impérative **(1 pt)**

A.2. Identification de l'aéroport le plus « accueillant » :

- exploitation cohérente de la fonction `degre_In` (ou opération équivalente) **(0,5 pt)**
- algorithme de recherche de maximum (initialisation + énumération cohérente de la structure de données idoine + mise à jour sous condition de l'identité du maximum) **(3 x 0,5 pt)**
- affichage du résultat attendu dans la console (`print`) **(0,5 pt)**
- complexité temporelle cohérente, avec justification succincte impérative **(1 pt)**

A.3. Création du dictionnaire `G` :

- attribution des clés (exploitation de la liste `Airports`) **(0,5 pt)**
- attribution des valeurs (construction des listes des successeurs en exploitant la liste `Routes`) **(1 pt)**

B. Parcours en largeur (total : 14 pt)

B.1. Stratégie « FIFO » liée à la structure de file (0,5 pt)

B.2. Propriété du parcours BFS (identification de plus courts chemins) (1 pt)

B.3. Evolution de la file d'attente :

- extraction systématique du sommet placé en tête de la file (0,5 pt)
- ajout systématique de nouveaux sommets en queue de file (respect de l'identité et de l'ordre imposé de ces sommets, les sommets déjà découverts ne sont plus ajoutés à la file) (1 pt)
- arrêt du parcours à l'issue du traitement du sommet 'HIP' (file vide) (0,5 pt)
- conclusion (existence de sommets inaccessibles en partant de 'PIF') (0,5 pt)

B.4. Implémentation de la fonction `BFS(G, A_start)` :

- respect de la définition de la fonction (structure `def ...(...)` : / `return dict`, indentation) (0,5 pt)
- gestion de la file d'attente (initialisation de `File_attente` + extraction du sommet placé en tête de file + ajout de nouveaux sommets) (3 x 0,5 pt)
- mise en œuvre du parcours du graphe : exploitation d'une structure de données indiquant si un sommet a déjà été traité ou non + condition d'arrêt (boucle `while`) (1 + 0,5 pt)
- gestion du dictionnaire `Pere` (initialisation + actualisation durant le parcours) (0,5 + 0,5 pt)

B.5. Analyse de la terminaison de `BFS(G, A_start)` :

- sélection de « variant de boucle » (0,5 pt)
- analyse argumentée de `len(File_attente)` (1 pt)
- analyse argumentée de `len(Pere)` (1 pt)

B.6. Implémentation de la fonction `distance(G, A_start, A_end)` :

- respect de la définition de la fonction (structure `def ...(...)` : / `return int / bool`, indentation) (0,5 pt)
- création du dictionnaire `Pere` (appel de la fonction `BFS`) (0,5 pt)
- cas d'un sommet inaccessible (traduction de ce critère et renvoi du booléen `False`) (0,5 pt)
- cas d'un sommet accessible (traduction de ce critère + création et incrémentation d'une variable `dist` en exploitant convenablement les données du dictionnaire `Pere`) (0,5 + 1 pt)

C. Identification des composantes fortement connexes (total : 15 pt)

C.1. Implémentation de la fonction `inverse(L)` :

- construction effective d'une liste inversée (1 pt)
- absence d'effet de bord (liste `L` inchangée après appel de la fonction) (0,5 pt)

C.2. Matrices d'adjacence :

- expression de $\mathcal{M}(H)$ (taille adéquate de la matrice, valeurs 0 ou 1, cohérence du remplissage avec l'ordre des sommets et la liste des arcs) (1 pt)
- expression correcte de $\mathcal{M}(H^t)$ (0,5 pt)
- justification de la dénomination « graphe transposé » (0,5 pt)

C.3. Justification pertinente du choix d'implémentation (complexité en espace) (1 pt)

C.4. Implémentation de la fonction transpose(G) :

- construction du dictionnaire G_t associé au graphe transposé **(1 pt)**
- renvoi du dictionnaire G_t et absence d'effet de bord **(0,5 pt)**

C.5. Implémentation de la fonction explorer(G, A, DejaVu, Visite) :

- attribution de la valeur True à la clé A pour le dictionnaire DejaVu **(0,5 pt)**
- énumération des successeurs de A qui n'ont pas encore été découverts **(0,5 pt)**
- appel récursif (appel de la fonction explorer avec les bons arguments et absence de return) **(1 pt)**
- ajout du sommet A dans la liste Visite (après les appels récursifs) **(0,5 pt)**

C.6. Implémentation de la fonction DFS(G) :

- initialisation du dictionnaire DejaVu **(0,5 pt)**
- mise en œuvre du parcours du graphe (énumération de tous les sommets du graphe avec sélection des sommets non encore découverts + appel récursif) **(1 + 0,5 pt)**
- gestion de la liste Visite (initialisation, inversion à la fin du parcours, renvoi) **(1 pt)**

C.7. Implémentation de la fonction Kosaraju(G) :

- création du dictionnaire G_t associé au graphe transposé **(0,5 pt)**
- énumération des sommets A dans l'ordre attendu (appel de la fonction DFS + boucle for) **(0,5 + 0,5 pt)**

C.8. Complexité temporelle de la fonction Kosaraju(G) :

- complexité de la fonction DFS **(0,5 pt)**
- complexité de la fonction transpose **(0,5 pt)**
- complexité du parcours en profondeur du graphe transposé **(0,5 pt)**
- conclusion sur la complexité de l'algorithme de KOSARAJU **(0,5 pt)**

A. Implémentation du graphe

A.1. Le degré entrant d'un sommet A correspond au nombre de ses prédécesseurs. Il faut pour cela compter le nombre d'occurrences de A dans le second membre des tuples contenus dans la liste `Routes` (le sommet correspond ainsi à l'extrémité d'un arc).

Cet algorithme implique une énumération complète de la liste `Routes` (de taille p), donc il possède une complexité en $\mathcal{O}(p)$.

```
def degre_In(Routes, A):  
    d = 0  
    for arc in Routes :  
        if arc[1] == A :  
            d += 1  
    return d
```

A.2. Recherche de l'aéroport le plus « accueillant », c'est-à-dire celui qui maximise le degré entrant :

```
# Initialisation  
A_max = Airports[0]  
d_max = degre_In(Routes, A_max)  
# Recherche du maximum  
for i in range(1, len(Airports)) :  
    A = Airports[i]  
    d = degre_In(Routes, A)  
    if d > d_max :  
        A_max = A  
        d_max = d  
# Affichage du résultat dans la console  
print("Aéroport le plus accueillant :", A_max)
```

A chaque itération de l'énumération de la liste `Airports` (de taille n), l'opération la plus coûteuse est l'appel de la fonction `degre_In` (de complexité $\mathcal{O}(p)$). L'affichage final dans la console est une opération élémentaire en $\mathcal{O}(1)$ donc, au final, cette séquence d'instructions possède une complexité en $\mathcal{O}(n \times p)$.

Mais on peut également envisager une autre version ne faisant pas appel à la fonction `degre_In`, en profitant de l'énumération de la liste `Routes` pour directement comptabiliser les degrés entrants de tous les aéroports :

```
# Construction d'un dictionnaire {clé = aéroports ; valeur = degrés entrants}  
D = {A:0 for A in Airports}  
for arc in Routes :  
    D[arc[1]] += 1  
# Recherche de l'aéroport maximisant le degré entrant  
d_max = 0  
for A in D.keys() :  
    if D[A] > d_max :  
        A_max = A  
        d_max = D[A]  
# Affichage du résultat dans la console  
print("Aéroport le plus accueillant :", A_max)
```

La création du dictionnaire `D` a un coût $\mathcal{O}(n)$ (taille du dictionnaire) et son remplissage a un coût $\mathcal{O}(p)$ (énumération de la liste `Routes`). La recherche du maximum revient à énumérer tous les couples clé / valeur du dictionnaire `D` donc possède un coût $\mathcal{O}(n)$. L'affichage final dans la console étant une opération élémentaire en $\mathcal{O}(1)$, cette séquence d'instructions possède au final une complexité en $\mathcal{O}(n + p)$.

A.3. On commence par créer un dictionnaire G avec les clés demandées (sommets du graphe contenus dans la liste `Airports`) en les associant à des listes vides. Puis on remplit ces listes en parcourant la liste `Routes` contenant tous les arcs du graphe :

```
G = {A : [] for A in Airports}
for arc in Routes :
    G[arc[0]].append(arc[1])
```

D'autres solutions sont également envisageables, par exemple la suivante :

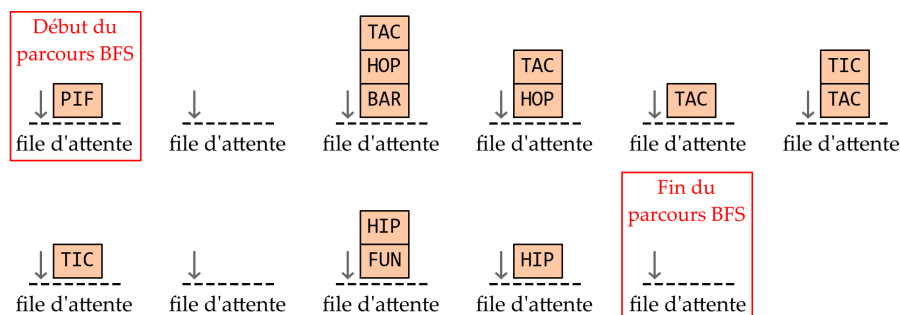
```
G = {}
for A in Airports :
    # Construction de la liste L des successeurs du sommet A
    L = []
    for arc in Routes :
        pre, suc = arc
        if pre == A :
            L.append(suc)
    # Ajout dans le dictionnaire
    G[A] = L
```

B. Parcours en largeur

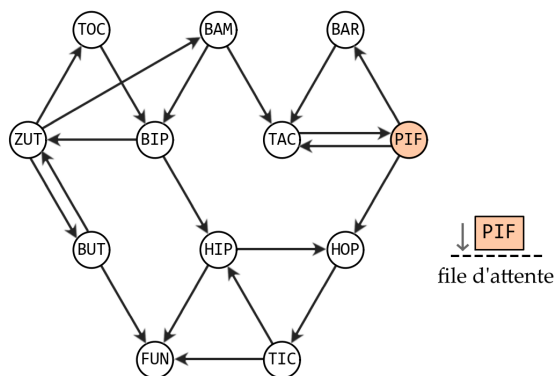
B.1. Le parcours en largeur adopte une stratégie « **FIFO** » (*First In, First Out*). En effet, pour une structure de donnée de type « file », l'élément extrait est systématiquement le plus ancien à avoir été ajouté (la « tête » de file).

B.2. A l'issue d'un parcours en largeur, chaque chemin identifié correspond à un **plus court chemin** (en nombre d'arcs empruntés).

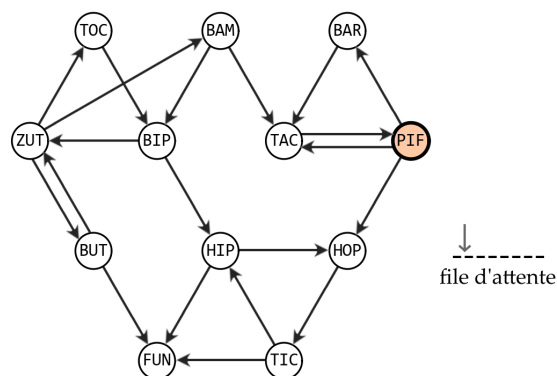
B.3. On représente ci-dessous l'évolution de la file d'attente au cours du parcours en largeur du graphe G_{bis} en partant du sommet 'PIF' (représentation des étapes significatives) :



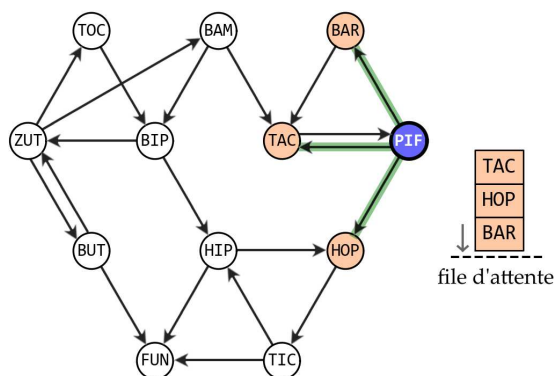
Pour justifier cette évolution de la file d'attente, on représente ci-dessous les principales étapes de ce parcours en profondeur du graphe G_{bis} .



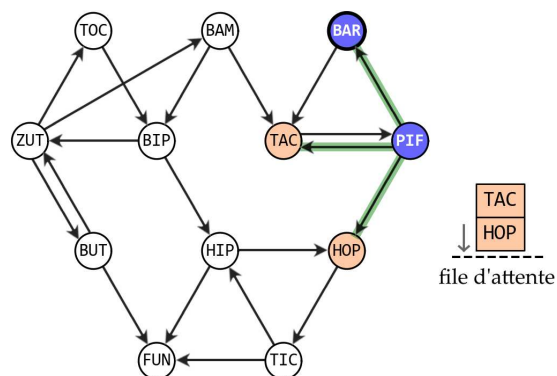
Initialement, seul le sommet de départ 'PIF' est présent dans la file d'attente.



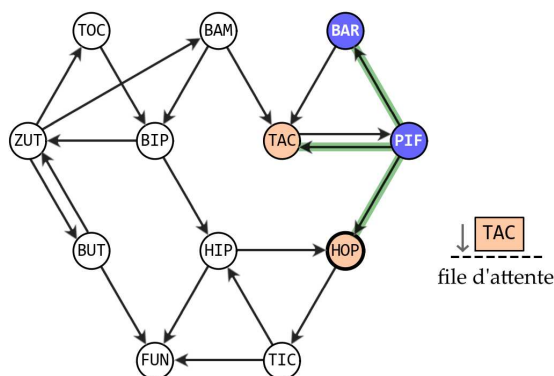
Ce sommet est retiré de la file d'attente : son traitement débute.



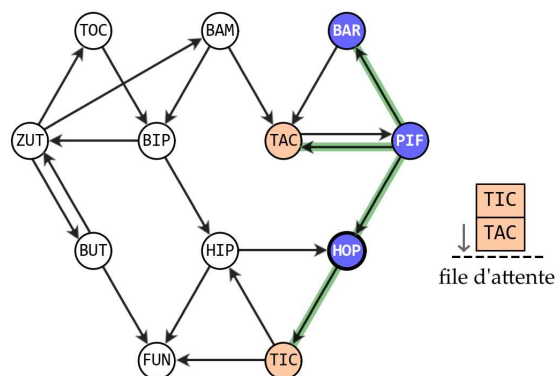
Les successeurs du sommet 'PIF' ('BAR', 'HOP' et 'TAC') sont découverts et placés en queue de la file d'attente.



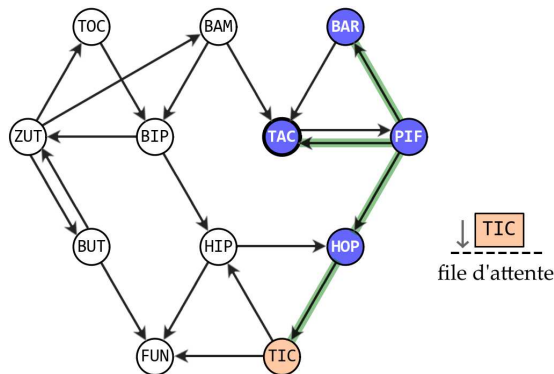
Le sommet 'BAR' (tête de file) est retiré de la file d'attente. Ce sommet ne possède pas de successeur non encore découvert : son traitement est donc terminé.



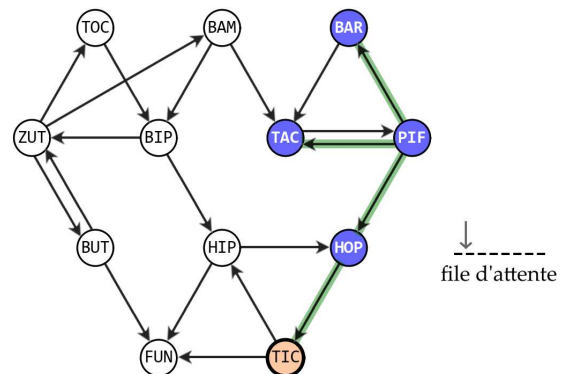
Le sommet 'HOP' (tête de file) est retiré de la file d'attente : son traitement débute.



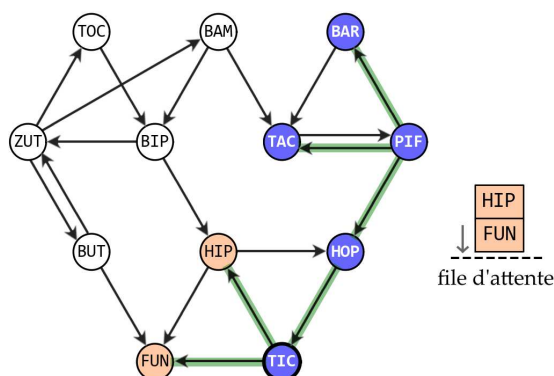
Son successeur 'TIC' est découvert et placé en queue de la file d'attente.



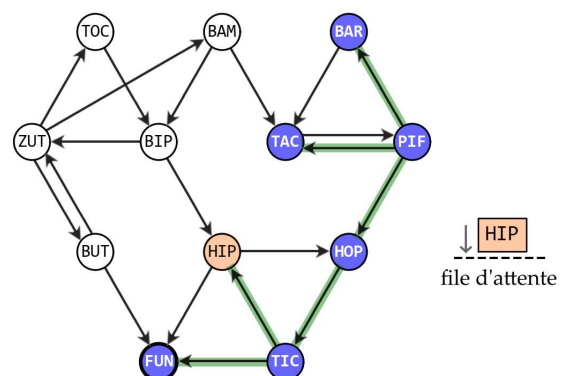
Le sommet 'TAC' (tête de file) est retiré de la file d'attente. Ce sommet ne possède pas de successeur non encore découvert : son traitement est donc terminé.



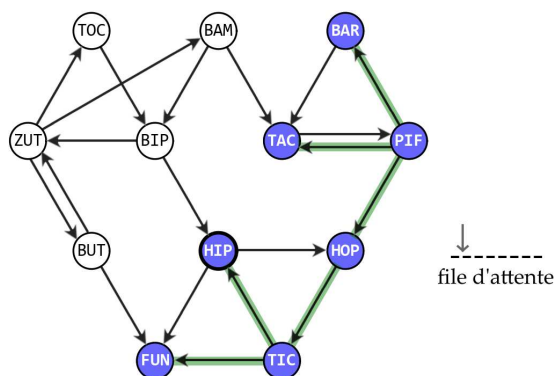
Le sommet 'TIC' (tête de file) est retiré de la file d'attente : son traitement débute.



Ses successeurs 'FUN' et 'HIP' sont découverts et placés en queue de la file d'attente.



Le sommet 'FUN' (tête de file) est retiré de la file d'attente. Ce sommet ne possède pas de successeur : son traitement est donc terminé.



Le sommet 'HIP' est retiré de la file d'attente. Ce sommet ne possède pas de successeur non encore découvert : son traitement est donc terminé.

La file d'attente est vide (plus aucun sommet en attente de traitement) : l'algorithme de parcours en largeur est donc terminé. On constate ainsi que plusieurs sommets du graphe ('TOC', 'BAM', 'ZUT', 'BIP' et 'BUT') ne sont pas accessibles en partant du sommet 'PIF'.

B.4. Implémentation de la fonction BFS(G, A_start) :

```
from collections import deque
def BFS(G, A_start) :
    # Initialisation de la file d'attente et du dictionnaire des sommets « pères »
    File_attente = deque()
    File_attente.append(A_start)
    Pere = {A_start : None}
    # Dictionnaire indiquant si un sommet a déjà été traité ou non
    DejaVu = {A : False for A in G.keys()}
    DejaVu[A_start] = True
    # Mise en oeuvre de l'algorithme de parcours
    while len(File_attente) > 0 :
        A = File_attente.popleft()
        for S in G[A] :
            if not DejaVu[S] :
                File_attente.append(S)
                Pere[S] = A
                DejaVu[S] = True
    return Pere
```

Le dictionnaire DejaVu est utilisé pour éviter de placer plusieurs fois un même sommet dans la file d'attente. Plusieurs implémentations sont possibles :

Initialisation	Test « S non encore découvert »	Instruction « S vient d'être découvert »
DejaVu = {A : False for A in G.keys()} DejaVu[A_start] = True	not DejaVu[S] (ou DejaVu[S] == False)	DejaVu[S] = True
DejaVu = {A : 0 for A in G.keys()} DejaVu[A_start] = 1	DejaVu[S] == 0	DejaVu[S] = 1
DejaVu = {A_start : None}	S not in DejaVu	DejaVu[S] = None

Si l'on adopte la dernière version, on peut même noter que l'introduction du dictionnaire DejaVu est facultative puisque son rôle peut être directement joué par le dictionnaire Pere :

```
def BFS(G, A_start) :
    # Initialisation de la file d'attente
    File_attente = deque()
    File_attente.append(A_start)
    # Dictionnaire des sommets « pères »
    Pere = {A_start : None}
    # Mise en oeuvre de l'algorithme de parcours
    while len(File_attente) > 0 :
        A = File_attente.popleft()
        for S in G[A] :
            if S not in Pere :
                File_attente.append(S)
                Pere[S] = A
    return Pere
```

B.5. Pour prouver la terminaison d'une boucle, on cherche à identifier un **variant de boucle**. Il s'agit d'une expression définie à partir des variables de l'algorithme, à valeur entière, évoluant de manière strictement monotone (strictement décroissante ou strictement croissante) et qui est bornée. L'existence de ce variant de boucle assure ainsi un nombre fini d'itérations.

Analyse de la terminaison de l'algorithme BFS :

- Comme mis en évidence à la question **B.3**, la taille de la file d'attente ne varie pas de façon monotone (elle peut diminuer ou augmenter lors des itérations de la boucle `while`), ce qui implique que `len(File_attente)` ne peut pas constituer un variant de boucle.
- L'évolution de la taille du dictionnaire `Pere` est croissante (on ne peut qu'ajouter des éléments dans ce dictionnaire), mais cette croissance n'est pas forcément « stricte ». Lors du traitement d'un sommet (extrait de la file d'attente), il est en effet possible qu'il ne possède pas de successeurs non encore découverts, donc la taille du dictionnaire `Pere` restera inchangée à l'itération correspondante.
- Pour prouver la terminaison de l'algorithme, un variant de boucle pourrait en fait être le « nombre de sommets extraits de la file d'attente ». Une extraction de la file est effectuée à chaque itération, donc ce nombre croît strictement, en étant borné par le nombre total de sommets dans le graphe : il s'agit donc bien d'un variant de boucle.

B.6. Pour implémenter la fonction `distance(G, A_start, A_end)` :

- on commence par créer le dictionnaire `Pere` associé au parcours en largeur issu de `A_start`
- si `A_end` n'est pas une clé valide du dictionnaire `Pere`, cela signifie que l'aéroport `A_end` n'est pas accessible par liaisons aériennes en partant de `A_start`
- sinon, on remonte la liste des pères en partant de `A_end`, jusqu'à tomber sur `A_start` (ou sur `None`), en incrémentant à chaque étape une variable `dist` correspondant à la distance recherchée.

```
def distance(G, A_start, A_end):
    Pere = BFS(G, A_start)
    if A_end in Pere : # A_end accessible
        dist = 0
        A = A_end
        while A != A_start :
            A = Pere[A]
            dist += 1
        return dist
    else : # A_end non accessible
        return False
```

```
def distance(G, A_start, A_end):
    Pere = BFS(G, A_start)
    if A_end in Pere : # A_end accessible
        dist = 0
        A = A_end
        while Pere[A] != None :
            A = Pere[A]
            dist += 1
        return dist
    else : # A_end non accessible
        return False
```

C. Identification des composantes fortement connexes

1. Composantes fortement connexe et algorithme de KOSARAJU

C.1. Implémentation de la fonction `inverse(L)` :

```
def inverse(L) :  
    return [L[len(L)-1-i] for i in range(len(L))]
```

D'autres implémentations sont évidemment possibles :

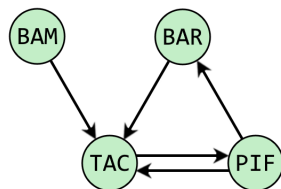
```
def inverse(L) :  
    new_L = []  
    n = len(L)  
    for i in range(n) :  
        new_L.append(L[n-1-i])  
    return new_L
```

```
def inverse(L) :  
    return L[::-1]
```

C.2. En considérant les sommets dans l'ordre attendu 'BAM', 'BAR', 'TAC', 'PIF', on obtient les matrices d'adjacence ci-dessous (0 = absence d'un arc ; 1 = présence d'un arc).

Matrice d'adjacence :

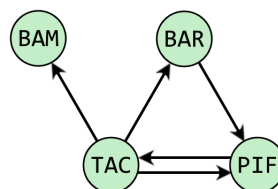
$$\mathcal{M}(H) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$



Graphe orienté H

Matrice d'adjacence :

$$\mathcal{M}(H^T) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$



Graphe transposé H^T

On parle de graphe « transposé » car la matrice d'adjacence $\mathcal{M}(H^T)$ est la transposée de la matrice d'adjacence $\mathcal{M}(H)$.

C.3. Le graphe G des liaisons aériennes possède $n \gg 1$ sommets (aéroports) et p arcs (liaisons) :

- l'implémentation sous la forme de listes d'adjacence correspond à une complexité en espace en $\mathcal{O}(p)$
- celle sous la forme d'une matrice d'adjacence correspond à une complexité en espace en $\mathcal{O}(n^2)$.

Puisque le graphe G est peu dense (la plupart des aéroports sont en effet reliés à un nombre de sommets très inférieur à n), ce qui implique que $p \ll n^2$. Il est ainsi plus avantageux de travailler avec des listes d'adjacence (la matrice d'adjacence contiendrait un très grand nombre de 0).

C.4. Implémentation de la fonction transpose(G) (chaque arc $A \rightarrow S$ est remplacé par un arc $S \rightarrow A$) :

```
def transpose(G) :  
    # Initialisation du dictionnaire G_t  
    G_t = {A : [] for A in G.keys()}  
    # Transformation des arcs A --> S en arc S --> A  
    for A in G.keys() :  
        for S in G[A] :  
            G_t[S].append(A)  
    return G_t
```

3. Parcours en profondeur du graphe (étape n°1)

C.5. Implémentation de la fonction récursive explorer(G, A, DejaVu, Visite) :

```
def explorer(G, A, DejaVu, Visite):  
    DejaVu[A] = True  
    for S in G[A] :  
        if not DejaVu[S] :  
            explorer(G, S, DejaVu, Visite) # Appel récursif  
    Visite.append(A)
```

C.6. Implémentation de la fonction DFS(G) :

```
def DFS(G):  
    # Initialisation  
    DejaVu = {A : False for A in G.keys()}  
    Visite = []  
    # Parcours de l'ensemble du graphe  
    for A in G.keys() :  
        if not DejaVu[A] :  
            explorer(G, A, DejaVu, Visite)  
    return inverse(Visite)
```

4. Algorithme de KOSARAJU (étapes n°1 et n°2)

C.7. Implémentation de la fonction Kosaraju(G) :

```
def Kosaraju(G):  
    # Initialisation  
    CFC = {} # Création du dictionnaire des composantes fortement connexes  
    num = 0 # Numéro de la première composante fortement connexe  
    Ordre = DFS(G) # Ordre des sommets défini à l'issue de l'étape n°1  
    G_t = transpose(G) # Graphe transposé  
    # Parcours en profondeur de l'ensemble du graphe transposé  
    DejaVu = {A : False for A in G_t.keys()}  
    for A in Ordre :  
        if not DejaVu[A] :  
            explorer_K(G_t, A, DejaVu, CFC, num)  
            num += 1  
    return CFC
```

C.8. Evaluation de la complexité temporelle de la fonction Kosaraju :

- l'appel de la fonction DFS possède une complexité en $\mathcal{O}(n + p)$ (sachant qu'au cours de ce parcours en profondeur du graphe G , chaque arc n'est examiné qu'une seule fois)
- l'appel de la fonction transpose possède une complexité en $\mathcal{O}(n + p)$ (création du dictionnaire G_t en $\mathcal{O}(n)$, puis examen de tous les p arcs du graphe)
- comme la fonction DFS, le parcours en profondeur du graphe transposé G^T possède une complexité en $\mathcal{O}(n + p)$ (sachant que les graphes G et G^T possèdent le même nombre de sommets et le même nombre d'arcs).

Au final, l'algorithme de KOSARAJU possède donc une complexité $\mathcal{O}(n + p)$.