Sujet : Centrale Supélec IPT 2016
Prévention des collisions aériennes

I Plan de vol

Cette partie est intégrée dans les applications de la synthèse sur les Bases de Données proposée au cours & TD 7.

I.A -

```
SELECT COUNT(*) FROM vol WHERE jour = '2016-05-02' AND heure < '12:00' ;
```

I.B -

```
SELECT id_vol FROM vol JOIN aeroport ON arrivee = id_aero
WHERE ville = 'Paris' AND jour = '2016-05-02' ;
```

I.C - La requête renvoie la liste des identifiants des vols intérieurs à la France le 2 mai 2016.

I.D -

```
SELECT vol1.id_vol AS Id1, vol2.id_vol AS Id2
FROM vol AS vol1, vol AS vol2
WHERE
Id1 < Id2 AND
vol1.niveau = vol2.niveau AND
vol1.jour = vol2.jour AND
vol1.depart = vol2.arrivee AND
vol1.arrivee = vol2.depart ;
```

II Allocation des niveaux de vol

II.A - Implantation du problème

Le sujet propose d'utiliser la matrice d'adjacence `conflit` comme variable globale, ce que je ne trouve pas du tout pertinent.

Mes propositions de fonctions seront donc avec un argument par défaut.

II.A.1) Voici trois propositions. Les deux dernières utilisent la fonction intégrée `sum()` avec une liste par compréhension. La première variante utilise un test dans la définition de la liste par compréhension. La seconde utilise le fait que `True` a une valeur de 1 en *Python* et `False` de 0.

Ce sont des syntaxes *Python* à retenir car elles sont claires et concises.

```
def nb_conflits(mat_adj = conflit):
    taille = len(mat_adj)
    nbre_conflits = 0
    for ligne in range(taille):
        for colonne in range(ligne + 1, taille): #matrice symétrique
            if conflit[ligne][colonne] != 0:
                nbre_conflits += 1
    return(nbre_conflits)

def nb_conflits_variante1(mat_adj = conflit):
    taille = len(mat_adj)
    nbre_conflits = 0
    for ligne in range(taille):
        nbre_conflits += sum([1 for col in range(ligne + 1, taille) \
        if conflit[ligne][col] != 0])
    return(nbre_conflits)
```

```
def nb_conflits_variante2(mat_adj = conflit):
    taille = len(mat_adj)
    nbre_conflits = 0
    for ligne in range(taille):
        nbre_conflits += sum([conflit[ligne][col] != 0 \
        for col in range(ligne + 1, taille)])
    return(nbre_conflits)
```

II.A.2) Les deux boucles parcourent tous les éléments de la matrice symétrique, c'est à dire $3 \cdot n \times \frac{3 \cdot n}{2}$ tests. La complexité est donc en $\mathcal{O}(n^2)$.

II.B - Régulation

II.B.1) Voici encore deux propositions différentes. La complexité de la seconde est en $\mathcal{O}(3 \cdot n)$, et non en $\mathcal{O}(n)$ pour la première, mais cela ne change rien à notre échelle de complexité.

```
def nb_vol_par_niveau_relatif(regulation):
    nb_vol = [0, 0, 0]
    for r in regulation:
        nb_vol[r] += 1
    return(nb_vol)

def nb_vol_par_niveau_relatif_variante(regulation):
    a = sum([x == 0 for x in regulation])
    b = sum([x == 1 for x in regulation])
    c = sum([x == 2 for x in regulation])
    return([a, b, c])
```

II.B.2) Coût d'une régulation

De même que pour **II.A.1)**, j'ajoute `conflit` en argument par défaut.

a)
```
def cout_regulation(regulation, mat_adj = conflit):
    cout = 0
    taille = len(regulation)
    for kligne in range(taille):
        for kcolonne in range(kligne + 1, taille):
            cout += mat_adj[3 * kligne + regulation[kligne]]\
            [3 * kcolonne + regulation[kcolonne]]
    return(cout)
```

b) De même que pour la complexité de `nb_conflits()`, les deux boucles parcourent un élément sur trois de la matrice symétrique, soit $n \times \frac{n}{2}$ opérations à coût constant. On se retrouve là encore avec une complexité en $\mathcal{O}(n^2)$.

c)
```
def cout_RFL(mat_adj = conflit):
    n = len(mat_adj) // 3
    regul0 = [0] * n
    return(cout_regulation(regul0))
```

II.B.3) Chaque vol ajouté au plan de vol offre trois possibilités supplémentaires de relations avec chacun des vols déjà enregistrés. Il y aura donc 3^n régulations possibles pour n vols. Et la complexité du calcul de chaque régulation est en $\mathcal{O}(n^2)$. La complexité du calcul de toutes les régulations possibles serait donc en $3^n \times \mathcal{O}(n^2) = \mathcal{O}(3^n)$.

C'est une complexité exponentielle, il est donc inenvisageable d'appliquer la *force brute* et de calculer les coûts de toutes les régulations possibles dès que l'on souhaite traiter un grand nombre de vols, ce qui doit être le cas en pratique.

II.C - L'algorithme Minimal

II.C.1)

a)
```
def cout_du_sommet(s, etat_sommet, mat_adj = conflit):
    '''Pré-condition : s est un numéro de sommet n'ayant pas été supprimé.'''
    cout = sum([mat_adj[s][k] for k in range(len(mat_adj)) \
    if etat_sommet[k] != 0])
    return(cout)
```

Il faut bien faire attention à ne pas prendre en compte les sommets supprimés.

b) La fonction calcule la somme sous condition des éléments d'une ligne de la matrice d'adjacence, elle effectue donc un nombre d'opérations proportionnel à $3 \cdot n$. C'est donc une complexité linéaire en $\mathcal{O}(n)$.

II.C.2)

a)
```
def sommet_de_cout_min(etat_sommet, mat_adj = conflit):
    # Initialisation du cout mini au premier sommet d'un vol non choisi
    vol_init = 0
    n_vols = len(etat_sommet) // 3
    while etat_sommet[3 * vol_init] != 2 and vol_init < n_vols :
        vol_init += 1
    if vol_init == n_vols :
        # Si tous les vols ont déjà été choisis
        return(None)
    else :
        s_min = 3 * vol_init
        cout_min = cout_du_sommet(s_min, etat_sommet, mat_adj)
        for vol in range(vol_init, n_vols):
            if etat_sommet[3 * vol] == 2:
                # Ce test permet de sauter les vols déjà choisis
                for k in range(3):
                    cout_k = cout_du_sommet(3 * vol + k, \
                    etat_sommet, mat_adj)
                    if cout_k < cout_min :
                        s_min = 3 * vol + k
                        cout_min = cout_k
    return(s_min)
```

En voici une variante qui évacue le problème de l'initialisation du coût minimum.

```
def sommet_de_cout_min_variante(etat_sommet, mat_adj = conflit):
    # Pour éviter l'initialisation de la fonction précédente
    s_min = 0
    cout_min = -1
    for s in range(len(etat_sommet)):
        # Chaque sommet est testé
        if etat_sommet[s] == 2:
            cout = cout_du_sommet(s, etat_sommet, mat_adj)
            if cout_min == -1 or cout < cout_min:
                cout_min = cout
                s_min = s
    return(s_min)
```

Remarque 1. *On pourrait aussi avoir envie d'utiliser* `inf` *pour initialiser* `cout_min`*. Cependant* `inf` *n'est pas intégré directement dans Python. On peut l'appeler en important le module* `numpy`*, mais il existe aussi une astuce pour l'utiliser sans ce module, c'est en déclarant* `inf = float('inf')`*, et dans ce cas, vous pouvez l'utiliser dans vos codes.*

b) L'initialisation coûte au pire des cas n entrées dans la boucle `while` qui ne comporte que des opérations en coût constant. Elle est donc en $\mathcal{O}(n)$. Ensuite, la boucle `for` parcourra les vols restants. Dans le pire des cas pour la suite, aucun sommet n'a déjà été choisi et la boucle `for` parcourra tous les sommets, soit $3 \cdot n$ entrées dans la seconde boucle `for` qui appelle la fonction `cout_du_sommet()`, elle-même de complexité en $\mathcal{O}(n)$.
La fonction `sommet_de_cout_min()` a donc une complexité quadratique en $\mathcal{O}(n^2)$.

II.C.3)

a)
```
def minimal(mat_adj = conflit):
    # initialisation de etat_sommet
    n_vols = len(mat_adj) // 3
    etat_sommet = [2] * 3 * n_vols
    # Parcourt des vols
    nb_vols_choisis = 0
    while nb_vols_choisis < n_vols :
        s_min = sommet_de_cout_min(etat_sommet, mat_adj)
        etat_sommet[s_min] = 1
        etat_sommet[s_min - s_min % 3 + (s_min + 1) % 3] = 0
        etat_sommet[s_min - s_min % 3 + (s_min + 2) % 3] = 0
        nb_vols_choisis += 1
    # Construction de regulation
    regulation = [s % 3 for s in range(len(etat_sommet)) \
    if etat_sommet[s] == 1]
    return(regulation)
```

Voici une variante qui construit `regulation` à chaque itération et qui utilise une affectation sous condition, autre syntaxe *Python* utile.

```
def minimal_variante(mat_adj = conflit):
    n_vols = len(mat_adj) // 3
    etat_sommet = [2] * 3 * n_vols
    regulation = [None] * n_vols
    for vol in range(n_vols):
        s_min = sommet_de_cout_min(etat_sommet, mat_adj)
        vol_min = (s_min // 3)
        for s in range(vol_min * 3, vol_min * 3 + 3):
            # Affectation conditionnée
            etat_sommet[s] = 1 if s == s_min else 0
        regulation[vol_min] = s_min % 3
    return(regulation)
```

b) La boucle `while` s'itère n fois, elle appelle `sommet_de_cout_min()` à chaque itération. Cette boucle a donc une complexité polynomiale en $\mathcal{O}(n^3)$. Puis la construction de `regulation` a une complexité en $\mathcal{O}(n)$.

La fonction `minimal()` a donc une complexité polynomiale en $\mathcal{O}(n^3)$.

C'est un algorithme dit *glouton* car il recherche à chaque étape un optimum local. On n'est pas alors sûr de trouver l'optimum global. C'est ce qu'on appelle une *heuristique*, c'est à dire une méthode de calcul qui fournit *rapidement* une solution réalisable mais pas nécessairement optimale pour un problème d'optimisation difficile.

II.D - Recuit simulé

Les opérations et fonctions *Python* disponibles précisées en fin de sujet supposaient l'importation de différents modules. Ces importations n'étaient pas demandées dans le sujet. Voici comment en pratique je les importe pour pouvoir les utiliser avec leurs préfixes.

```
import random as rd
import numpy as np
import time as tm
```

Un *recuit* est un terme issu du domaine des traitements thermiques des métaux et alliages. Il s'agit d'une montée en température du matériau au delà d'une température particulière suivi d'une descente contrôlée de sa température. Ce procédé permet notamment de relaxer les contraintes dites résiduelles dans le matériau dues à une transformation physique (déformation plastique par exemple) ou à une transformation thermique (trempe ou soudage par exemple).

Cette méthode est transposée en optimisation numérique pour trouver les extrema d'une fonction.

```
def recuit(regulation, T = 1000, mat_adj = conflit):
    cout = cout_regulation(regulation, mat_adj)  #calcul du coût
    n = len(regulation)
    while T >= 1:
        vk = rd.randint(n)  # vol vk tiré au hasard
        rk = regulation[vk]
        nouv_rk = (rk + rd.randint(1, 2)) % 3
        regulation[vk] = nouv_rk    # modification de rk
        # calcul du nouveau coût
        cout_modif = cout_regulation(regulation)
        Delta_c = cout_modif - cout
        if Delta_c < 0 and rd.random() > np.exp(-(Delta_c) / T):
            regulation[vk] = rk # reprise du coût précédent
        T *= 0.99   # Diminution du parmètre T de 1%
    # Pas de renvoi nécessaire, car regulation est modifiée par effet
    # de bord (cette fonction est donc une procédure).
```

La complexité de cette fonction dépend alors de la complexité de `cout_regulation()` qui est en $\mathcal{O}(n^2)$ et d'autres opérations en coût constant. Cette fonction est appelée dans la boucle `while` qui est parcourue ici N fois, avec N tel que $0.99^N \cdot T < 1$.

La complexité de `recuit()` est donc simplement quadratique en $\mathcal{O}(n^2)$. C'est donc une solution non optimale préférable à l'algorithme Minimal.

III Système d'alerte de trafic et d'évitement de collision

III.A - Acquisition et stockage des données

III.A.1) Le débit binaire est de 10^6 bits par seconde, soit un bit par μs. Une émission ne dépassant pas 128 μs est donc une émission ne dépassant pas 128 bits. 16 bits sont consacrés aux marques de début et de fin et au contrôle, il reste donc 112 bits au maximum pour les données dans une émission de transpondeur.

III.A.2) Sur les 112 bits de données maximum du message, 24 sont consacrés au numéro d'identification de l'avion. Il en reste donc 88.

Il reste à coder l'altitude et la vitesse ascensionnelle. La précision requise n'est pas indiquée, il faut donc en faire l'hypothèse. Je considère que chaque grandeur doit être connue à l'unité près.

Il faut donc $(66\,000 - 2\,000) + 1 = 64\,001$ valeurs pour l'altitude, que l'on peut coder sur $\lceil \log(64\,001)/\log(2) \rceil = 16$ bits, et $(5\,000 + 5\,000) + 1 = 10\,001$ valeurs pour la vitesse ascensionnelle que l'on peut coder sur 14 bits.

Ces 30 bits sont donc tout à fait compatibles pour la taille d'un message de transpondeur. Les informations peuvent donc être obtenues en une seule fois.

III.A.3) 100 heures de fonctionnement exigent de mémoriser $100 \times 3\,600 \times 100$ données. Chaque donnée demande une place mémoire de 8×4 octets.

Le volume de stockage nécessaire pour conserver toutes les données est donc de $1,152$ Go.

Ce volume de données n'est plus aujourd'hui une contrainte technique forte.

III.B - Estimation du CPA

III.B.1) $\overrightarrow{OG}(t) = \overrightarrow{OG}(t_0) + (t - t_0) \cdot \overrightarrow{V} = (x(t), y(t), z(t))_{\mathcal{R}_0}$

Donc, en utilisant les données sous la forme précisée à la question **III.A.3**, on peut calculer les coordonnées du vecteur $\overrightarrow{OG}(t)$.

$$\left\{ \begin{array}{rcl} x(t) & = & x + (t - t0) \cdot vx \\ y(t) & = & y + (t - t0) \cdot vy \\ z(t) & = & z + (t - t0) \cdot vz \end{array} \right.$$

III.B.2) Si les deux avions passent par une distance minimale, alors cette distance minimale à lieu à l'instant $t = t_c$ tel que :

$$\frac{d\|\overrightarrow{OG}(t)\|^2}{dt} = 2 \cdot \overrightarrow{OG}(t) \cdot \frac{d\overrightarrow{OG}(t)}{dt} = 2 \cdot \overrightarrow{OG}(t) \cdot \overrightarrow{V}(G/\mathcal{R}_0) = 0$$

C'est à dire pour $\overrightarrow{OG}(t) \cdot \overrightarrow{V} = \left(\overrightarrow{OG}(t_0) + (t - t_0) \cdot \overrightarrow{V}\right) \cdot \overrightarrow{V} = 0$.

Donc, si cette distance minimale existe, elle aura lieu à $t = t_c > t_0$ tel que :

$$t_c = t_0 - \frac{\overrightarrow{OG}(t_0) \cdot \overrightarrow{V}}{\|\overrightarrow{V}\|^2}$$

Avec les données renvoyées par la fonction `acquerir_intrus()`, cela donne l'expression suivante :

$$t_c = t0 - \frac{x \times vx + y \times vy + z \times vz}{vx^2 + vy^2 + vz^2}$$

III.B.3 Si le produit scalaire $\overrightarrow{OG}(t_0) \cdot \overrightarrow{V}$ est positif, cela veut simplement dire que l'avion *intrus* s'éloigne de l'*avion propre*. L'expression de t_c de la question précédente montre également que si $\overrightarrow{OG}(t_0) \cdot \overrightarrow{V} > 0$, alors le minimum est atteint pour $t = t_0$.

III.B.4 Fonction `calculer_CPA()` :

```
def calculer_CPA(intrus):
    '''intrus = [id, x, y, z, vx, vy, vz, t0] renvoyé par acquerir_intrus().'''
    OGt0 = intrus[1:4]
    V = intrus[4:7]
    #calcul du produit scalaire avec la fonction zip
    OGt0_dot_V = sum(pos * vit for (pos, vit) in zip(OGt0, V))

    if OGt0_dot_V > 0:
        # test sur le signe du produit scalaire
        return(None)
    else:
        V_carre = sum(pow(vit, 2) for vit in V) # calcul de V^2
        tCPA = intrus[7] - OGt0_dot_V / V_carre
        OGtc = [pos + (tCPA - intrus[7]) * vit \
        for (pos, vit) in zip(OGt0, V)] # calcul de OG(tc)
        dCPA = pow(sum(pow(pos, 2) for pos in OGtc), 0.5)
        zCPA = OGtc[2] / 0.3048 # conversion mètre en pieds
    return([tCPA, dCPA, zCPA])
```

III.C - Mise à jour de la liste des CPA

III.C.1) Fonction `mettre_a_jour_CPAs()` :

```
def mettre_a_jour_CPAs(CPAs, id, nv_CPA, intrus_max, suivi_max):
    '''Modifie la liste CPAs par effet de bord, et renvoie None si l'avion a
été supprimé ou n'a pas été ajouté, ou un entier indiquant l'indice de la ligne
de CPAs qui a été modifiée ou ajoutée.'''
    n_suivis = len(CPAs)
    # recherche de l'indice de l'intrus s'il est déjà suivi,
    # n_suivis s'il ne l'est pas
    indice_id = 0
    while indice_id < n_suivis and indice_id != CPAs[indice_id][0]:
        indice_id += 1
    if indice_id != n_suivis: # l'intrus est déjà suivi
        if nv_CPA[0] > suivi_max or nv_CPA == None:
            # son CPA est prévu dans plus de suivi_max secondes OU
            # l'intrus ne présente pas de risque de collision
            del(CPAs[indice_id])    # l'intrus est supprimé de la liste
            return(None)
        elif nv_CPA[0] <= suivi_max:
            # son CPA est prévu dans moins de suivi_max
            CPAs[indice_id] = [id] + nv_CPA
            # mise à jour avec les nouvelles infos sur le CPA
            return(indice_id)
    elif indice_id == n_suivis and nv_CPA[0] <= suivi_max:
        # l'intrus n'est pas déjà suivi ET
        # son CPA est prévu dans moins de suivi_max secondes
        if n_suivis < intrus_max:
            # il reste de la place dans la liste
            CPAs += [[id] + nv_CPA]
        elif n_suivis >= intrus_max and nv_CPA[0] < CPAs[-1][1]:
            # la liste est pleine ET
            # le tCPA du nouvel intrus est inférieur à celui du dernier
            # de la liste
            CPAs[-1] = [id] + nv_CPA
        else : # si n_suivis >= intrus_max et nv_CPA[0] >= CPAs[-1][1]
            return(None)
        return(len(CPAs) - 1)
```

III.C.2) Fonction `replacer()` :

```
def replacer(ligne, CPAs):
    tCPA_intrus = CPAs[ligne][1]
    n_suivis = len(CPAs)
    while ligne > 0 and CPAs[ligne-1][1] > tCPA_intrus:
        CPAs[ligne-1], CAPs[ligne] = CPAs[ligne], CPAs[ligne-1]
        ligne -= 1
    while ligne < n_suivis-1 and CPAs[ligne+1][1] < tCPA_intrus:
        CPAs[ligne+1], CAPs[ligne] = CPAs[ligne], CPAs[ligne+1]
        ligne += 1
```

III.C.3) Fonction `enregistrer_CPA()` :

```
def enregistrer_CPA(intrus, CPAs, intrus_max, suivi_max):
    CPA = calculer_CPA(intrus)
    if CPA != None :                        # si risque de collision
        nv_CPA = [intrus[0]] + CPA
        ligne = mettre_a_jour_CPAs(CPAs, intrus[0], nv_CPA, intrus_max, suivi_max)
        if ligne != None:       # si modification du CPAs
            replacer(ligne, CPAs)
```

III.D - Évaluation des paramètres généraux du système TCAS

III.D.1) Le pire des cas est lorsque les deux avions font route l'un vers l'autre, soit à une vitesse relative de 2×900 km $\cdot$ h^{-1}. Dans ce cas, la collision éventuelle entre les deux avions aurait lieu à $t = 60/1800 = 1/30$ h $= 2$ mn.

Cela représente 120 s, ce qui est supérieur aux 100 s de `suivi_max` laissé pour que les pilotes puissent réagir.

III.D.2) Avec une vitesse ascensionnelle de 1500 pieds par minute, il faut 20 s pour augmenter son altitude de 500 pieds. Si les deux avions veulent se croiser avec une différence d'altitude d'au moins 500 pieds, l'un doit commencer à manœuvrer 20 s avant leur `tCPA`, ou 10 s avant si les deux pilotes coordonnent leurs actions.

III.D.3) Cette durée de 25 s est supérieure à la durée de 20 s calculée à la question précédente. Cela laisse donc le temps au pilote de réagir.

III.D.4) Pour vérifier 30 intrus, chacun au moins une fois par seconde, cela implique un temps de boucle de $1/30$ s $= 33$ μs pour la fonction `TCAS()`.

III.D.5) La fonction `TCAS()` appelle dans sa boucle `while` trois fonctions.

- La fonction `acquerir_intrus()` dépend du temps d'acquisition des messages des transpondeurs. La question **III.A.1)** précise que l'émission du message n'excède pas 128 μs. Sa porteuse est une fréquence radio, donc une onde électromagnétique de célérité proche de la vitesse de la lumière.
- La fonction `enregistrer_CPA()` appelle la fonction `calculer_CPA()`, la fonction `mettre_a_jour_CPAs()` et la fonction `replacer()`.
  - La fonction `calculer_CPA()` ne fait appel qu'à des opérations en coût constant, elle a donc une complexité en $\mathcal{O}(1)$ et sera donc très rapide, de l'ordre de la nano seconde.
  - La fonction `mettre_a_jour_CPAs()` parcourt la liste `CPAs` qui ne contient que 30 lignes comportant elles-mêmes 4 entiers. Elle a donc une complexité en $\mathcal{O}(30 \times 4) = \mathcal{O}(1)$, là encore très rapide.
  - La fonction `replacer()` est un tri par insertion sur la liste `CPAs`, soit sur 30 lignes. Son temps de calcul sera encore de l'ordre de la nano seconde.
- La fonction `traiter_CPAs()` examine les CPA des intrus (30 au maximum), décide si l'un d'eux présente un risque de collision (un simple calcul en $\mathcal{O}(1)$, détermine la manœuvre à effectuer (monter ou descendre, là encore, sans doute un simple calcul en $\mathcal{O}(1)$), en coordination avec le système TCAS de l'intrus concerné (par émission de transpondeur), et génère une alarme et une consigne à destination du pilote (utilisation de simples signaux électriques).

En conclusion de toute cette étude, on peut en déduire que le facteur limitant la vitesse d'éxécution de la fonction `TCAS()` est l'émission et la réception des messages par les transpondeurs, dont les temps caractéristiques sont inférieurs à la milliseconde, soit très en deça du temps de boucle calculé à la question précédente.

La mise en œuvre de la fonction `TCAS()` est donc compatible avec les spécifications du système TCAS.