CENTRALE - 2016 - INFORMATIQUE TOUTES FILIÈRES

Rédigé par Jérémy Larochette (Lycée Carnot, Dijon).

On utilise `Python 3`.

I plan de vol

I.A – Si midi est inclus :

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

Sinon :

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

I.B – Trois interprétations :

- Numéro des vols (`v1`) tels qu'il existe un vol (`v2`) partant du même aéroport et arrivant dans l'un des aéroports de Paris le 2 mai 2016 :

```sql
SELECT v1.id_vol FROM vol v1 JOIN vol v2   ON v1.depart = v2.depart
                             JOIN aeroport ON v2.arrivee = id_aero
WHERE ville = 'Paris' AND v2.jour = '2016-05-02';
```

- Numéro des vols arrivant dans l'un des aéroports de Paris le 2 mai 2016 :

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

- Numéro des vols partant de l'un des aéroports de Paris le 2 mai 2016 :

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

I.C – Cette requête renvoie les numéros de vols entre deux villes françaises le 2 mai 2016.

I.D –

```sql
SELECT v1.id_vol, v2.id_vol FROM vol v1
  JOIN vol v2 ON v1.niveau  = v2.niveau
             AND v1.depart  = v2.arrivee
             AND v1.arrivee = v2.depart
             AND v1.jour    = v2.jour
  WHERE v1.id_vol < v2.id_vol; -- pour éviter les doublons
```

II Allocation des niveaux de vol

II.A – Implantation du problème

II.A.1)

```python
def nb_conflits():
    "Renvoie le nombre de conflits potentiels dans conflit"
    N = len(conflit) # N = 3n
    nb = 0
    for i in range(N):
        for j in range(i): # au dessous de la diagonale seulement
            if conflit[i][j] != 0:
                nb += 1
    return nb
```

Remarque : la ligne `global conflit` n'est pas nécessaire ici.

II.A.2) La boucle interne a une complexité en $O(i)$ donc la fonction a une complexité en $O(n^2)$.

II.B – Régulation

II.B.1)

```python
def nb_vol_par_niveau_relatif(regulation):
    """regulation est une liste de n entiers
    renvoie [a, b, c] où :
      - a est le nombre de vols à leurs RFL
      - b est le nombre de vols au-dessus de leurs RFL
      - c est le nombre de vols au-dessous de leurs RFL"""
    nb_vols = [0, 0, 0]
    for r in regulation:
        nb_vols[r] += 1
    return nb_vols
```

II.B.2.a)

```python
def cout_regulation(regulation):
    """Renvoie le coût de la régulation"""
    n = len(regulation)
    cout = 0
    v = [] # liste des n vols
    for i in range(n):
        v.append(3*i + regulation[i])
    for i in range(n):
        for j in range(i):
            cout += conflit[v[i]][v[j]]
    return cout
```

II.B.2.b) Le coût de la première boucle est en $O(n)$ et celle des deux autres est comme précédemment en $O(n^2)$ soit un coût total en $O(n^2)$.

II.B.2.c)

```python
def cout_RFL():
    """coût de la régulation où chaque avion vole à son RFL"""
    n = len(conflit) // 3
    return cout_regulation( [ 0 for _ in range(n) ] )
```

II.B.3) Pour chaque vol, il y a 3 niveaux relatifs, donc il y a 3^n régulations possibles. Le calcul de tous les coûts possibles serait exponentiel, ce qui est déraisonnable.

II.C – L'algorithme Minimal

II.C.1.a)

```python
def cout_du_sommet(s, etat_sommet):
    """Renvoie le cout du sommet s et le paramètre etat_sommet"""
    assert etat_sommet[s] != 0
    N = len(etat_sommet)
    cout = 0
    for i in range(N):
        if etat_sommet[i] != 0:
            cout += conflit[i][s]
    return cout, etat_sommet
```

II.C.1.b) Une simple boucle dont le corps est en $O(1)$, la complexité est en $O(N) = O(n)$ ($N = 3n$.)

II.C.2.a)

```
def sommet_de_cout_min(etat_sommet):
    """Renvoie le sommet de coût minimal s'il y a encore des sommets non supprimés,
-1 sinon."""
    N = len(etat_sommet)
    cout_min, s_min = -1, -1
    for s in range(N):
        if etat_sommet[s] == 2:
            c = cout_du_sommet(s, etat_sommet)[0]
            if cout_min == -1 or c < cout_min:
                cout_min, s_min = c, s
    return s_min
```

II.C.2.b) On applique au plus N fois la fonction `cout_du_sommet`, donc le complexité est en $O(N^2) = O(n^2)$, le reste est négligeable.

II.C.3.a)

```
def minimal():
    """renvoie la régulation résultant de l'algorithme Minimal"""
    N = len(conflit)
    n = N // 3
    etat_sommet = [ 2 for _ in range(N) ]
    regulation = [ 0 for _ in range(n) ]

    for _ in range(n): # On traite tous les sommets
        s = sommet_de_cout_min(etat_sommet)
        i = s // 3

        # changement d'état du sommet de cout min
        for j in range(3):
            etat_sommet[3*i + j] = 0
        etat_sommet[s] = 1

        regulation[i] = s % 3

    return regulation
```

II.C.3.b) On applique $N/3 = n$ fois la fonction `sommet_de_cout_min` car à chaque étape 3 sommets changent d'état ce qui donne du $O(n^3)$. La deuxième boucle étant en $O(n)$, on obtient une complexité en $O(n^3)$. C'est polynomial et beaucoup mieux que la force brute proposée en II.B.3 !

II.D – Recuit simulé

```
def recuit(regulation):
    """recuit simulé appliqué à une régulation"""
    n = len(regulation)
    T = 1000
    c = cout_regulation(regulation)
    while T >= 1:
        v = randint(n)       # choix du vol aléatoire
        r = regulation[v]
        regulation[v] = (r + randint(2) + 1) % 3 # choix du niveau aléatoire != r
        c_alea = cout_regulation(regulation)
        if c_alea >= c and random() > exp(-(c_alea - c) / T):
            regulation[v] = r # retour à la régulation précédente
        else:
            c = c_alea # mise à jour du coût de la régulation
        T *=  .99
```


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

III.A – Acquisition et stockage des données

III.A.1) Pour chaque émission d'au plus 128μs, au plus 128 bits sont utilisés dont 16 bits réservés soit 112 bits pour le message.

III.A.2) Pour l'altitude, il y a $66000 - 2000 + 1 = 64001$ entiers naturels à coder ce qui peut se faire sur 16 bits ($2^{16} = 65536$) (ou sur 17 bits en codant les entiers directement en binaire).

Pour la vitesse comprise entre -5000 et 5000, codée par exemple en complément à 2, il suffit de 14 bits ($2^{13} < 10001 < 2^{14}$).

Il y a donc largement de quoi coder cela en une seule fois dans les 112 bits disponible dans le transpondeur.

III.A.3) 100 heures représentent 360 000 secondes soit 36 000 000 appels à la fonction. Pour chaque appel, il y a besoin de $8 \times 4 = 32$ octets, soit en tout 1 152 000 000 o = 1,152 Go ce qui est raisonnable avec les supports de données actuels.

III.B – Estimation du CPA

III.B.1) Comme la vitesse est supposée constante et le mouvement supposé rectiligne, $\boxed{\overrightarrow{OG}(t) = \overrightarrow{OG}(t_0) + (t - t_0)\overrightarrow{V}.}$

III.B.2) On calcule $\left\|\overrightarrow{OG}(t)\right\|^2 = \left\|\overrightarrow{OG}(t_0)\right\|^2 + 2(t-t_0)\overrightarrow{OG}(t_0)\cdot\overrightarrow{V} + (t-t_0)^2\left\|\overrightarrow{V}\right\|^2$. On veut $t \geqslant t_0$ et $\left\|\overrightarrow{OG}(t)\right\|^2$ minimal. Le minimum sera atteint au sommet de la parabole si celui-ci est atteint pour $t \geqslant t_0$, et en t_0 sinon. En dérivant, on obtient $\left(\left\|\overrightarrow{OG}\right\|^2\right)'(t) = 2\overrightarrow{OG}(t_0)\cdot\overrightarrow{V} + 2(t-t_0)\left\|\overrightarrow{V}\right\|^2$.

On a donc $\boxed{t_c = t_0 - \dfrac{\overrightarrow{OG}(t_0)\cdot\overrightarrow{V}}{\left\|\overrightarrow{V}\right\|^2} \text{ si } \overrightarrow{OG}(t_0)\cdot\overrightarrow{V} \leqslant 0 \text{ et } t_0 \text{ sinon.}}$

III.B.3) D'après la question précédente, si $\overrightarrow{OG}(t_0)\cdot\overrightarrow{V} > 0$, $t_c = t_0$ donc il n'y a pas collision.

III.B.4) On suppose que dans l'énoncé, « s'il n'y a pas de risque de collision » se traduit par « si $\overrightarrow{OG}(t_0)\cdot\overrightarrow{V} > 0$ ».

```
def calculer_CPA(intrus):
    """intrus est une liste de la forme [id, x, y, z, vx, vy, vz, t0] renvoyée par
 acquerir_intrus et renvoie :
    - None s'il n'y a pas de risque de collision,
    - [tCPA, dCPA, zCPA] sinon où tCPA est l'instant prévu pour le CPA, dCPA est la
    distance en mètres entre les deux avions au moment du CPA, zCPA est la
    différence d'altitude en pieds."""

    id, x, y, z, vx, vy, vz, t0 = intrus
    OGdotV = x*vx + y*vy + z*vz
    normV2 = vx**2 + vy**2 + vz**2

    if  OGdotV <= 0:
        dt = -OGdotV/normV2      # = t0 - tCPA
        tCPA = t0 + dt
        dCPA = sqrt( (x + dt*vx)**2 + (y + dt*vy)**2 + (z + dt*vz)**2 )
        zCPA = (z + dt*vz)/.3048
        return [tCPA, dCPA, zCPA]

    else: return None # ou bien juste return, ou bien rien.
```

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

III.C.1)

```
def mettre_a_jour_CPAs(CPAs, id, nv_CPA, intrus_max, suivi_max):
    """met à jour la liste des CPAs et renvoie :
    - None si l'avion id 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."""

    deja_suivi = False # L'intrus est-il déjà suivi ?
    numero = -1        # Position dans CPAs si c'est le cas, -1 sinon

    while numero < len(CPAs) - 1 and not deja_suivi:
        numero += 1
        deja_suivi = (CPAs[numero][0] == id)


    if nv_CPA is None:              # Pas de risque de collision
        if deja_suivi:
            del CPAs[numero]
            return None
    else:                           # Risque de collision
        tCPA, dCPA, zCPA = nv_CPA
        dt = tCPA - time()

        if deja_suivi:

            if dt > suivi_max:
                del CPAs[numero]
                return None
            else:
                CPAs[numero] = [id] + nv_CPA
                return numero

        elif dt <= suivi_max:

            if len(CPAs) < intrus_max:
                CPAs.append([id] + nv_CPA)
            elif tCPA < CPAs[-1][1]:
                CPAs[-1] = [id] + nv_CPA
            return len(CPAs) - 1
```

III.C.2)

```
def replacer(ligne, CPAs):
    """Remet la ligne CPAs[ligne] à sa place dans l'ordre
    des tCPA croissants."""

    def echange(L,i,j): L[i], L[j] = L[j], L[i]

    position = ligne

    while position > 0 and CPAs[position][1] < CPAs[position - 1][1]:
        echange(CPAs, position, position - 1)
        position -= 1

    while position < len(CPAs) - 1 and CPAs[position][1] > CPAs[position + 1][1]:
        echange(CPAs, position, position + 1)
        position += 1
```


ou, de manière moins fine :

```
def replacer(ligne, CPAs):
    """Remet la ligne CPAs[ligne] à sa place dans l'ordre des tCPA croissants."""
    cpa = CPAs[ligne]
    del CPAs[ligne]

    i=0
    while i < len(CPAs) and CPAS[i][1] < cpa[1]:
        i+=1

    CPAs.insert(i, cpa)
```

III.C.3)

```
def enregistrer_CPA(intrus, CPAs, intrus_max, suivi_max):
    """Enregistrement ou modification d'un intrus"""
    ligne = mettre_a_jour_CPAs(CPAs, intrus[0], calculer_CPA(intrus), intrus_max,
                               suivi_max)
    if ligne is not None:
        replacer(ligne, CPAs)
```

III.D – Évaluation des paramètres généraux du système TCAS

III.D.1) Le temps minimum entre la détection et le CPA correspond au cas où les avions sont face à face : le CPA se trouve alors à 30 km et sera atteint en $\frac{30}{900}$ h = $\frac{1}{30}$ h = 2 min = 120 s qui est du même ordre que 100 s et un peu plus grand, donc cela permet de détecter à temps, de réagir et de suivre les intrus pour éviter les collisions.

III.D.2) Si l'un des deux avions seulement change d'altitude, il lui faudra 20 s pour obtenir une différence d'altitude de 500 pieds, si les deux avions changent d'altitude dans le sens opposé, il suffit de 10 s.

III.D.3) Vu la réponse à la question précédente, 25 secondes suffisent à s'éloigner raisonnablement du CPA et donc de l'autre avion.

III.D.4) Il faut 30 vérifications par seconde, donc la boucle doit s'exécuter en moins de $\frac{1}{30}$ s ≈ 33,3 ms.

III.D.5) Un certain nombre de paramètres inconnus interviennent dans la fonction `TCAS` ce qui rend difficile la réponse à cette question. Qu'appelle-t-on une boucle ? un tour du `while` ou une mise à jour de chacun des intrus ? Le seul facteur limitant sur lequel on peut discuter ici est `intrus_max` car dans la boucle `while` de la fonction `TCAS`, on trouve :

- une exécution de la fonction `acquerir_intrus` qui fait intervenir le matériel de détection d'intrus de l'avion dont nous ne savons pas grand chose,
- une exécution de la fonction `enregistrer_CPA` faisant intervenir `calculer_CPA`, `mettre_a_jour` et `replacer`, s'exécute en O(`intrus_max`) en faisant l'hypothèse raisonnable que `del` et `insert` ont une complexité linéaire,
- une exécution de la fonction `traiter_CPAs` dont nous ne savons pas grand chose car nous ne savons pas comment sont choisis et traités les intrus « les plus dangereux »(premiers de liste `CPAs` ?), comment réagir (monter, descendre) ni le temps que cela demande.

Comment, dans ces conditions, donner un ordre de grandeur du temps minimal d'exécution d'une boucle ?
Autre piste : en faisant l'hypothèse (!) que le temps d'exécution des fonctions inconnues est négligeable, le facteur limitant la vitesse serait le temps nécessaire à échanger les données. Pour une distance de l'ordre de la dizaine de kilomètre, à la vitesse de la lumière, l'échange sera de l'ordre de la centaine de μs, du même ordre de grandeur que la durée d'émission de message d'un intrus. On peut estimer le temps d'exécution d'une boucle de l'ordre de la milliseconde ce qui est compatible avec la question précédente.