teaching sciences UPSTI for innovation

Proposition de corrigé

Concours : Concours Centrale-Supélec

Année : 2016

Filière : MP - PC - PSI - TSI

Épreuve : Informatique

Ceci est une proposition de corrigé des concours de CPGE, réalisée bénévolement par des enseignants de Sciences Industrielles de l'Ingénieur et d'Informatique, membres de l'UPSTI (Union des Professeurs de Sciences et Techniques Industrielles), et publiée sur le site de l'association :

https://www.upsti.fr/espace-etudiants/annales-de-concours

A l'attention des étudiants

Ce document vous apportera des éléments de corrections pour le sujet traité, mais n'est ni un corrigé officiel du concours, ni un corrigé détaillé ou exhaustif de l'épreuve en question.

L'UPSTI ne répondra pas directement aux questions que peuvent soulever ces corrigés : nous vous invitons à vous rapprocher de vos enseignants si vous souhaitez des compléments d'information, et à vous adresser à eux pour nous faire remonter vos éventuelles remarques.

Licence et Copyright

Informez-vous !

Retrouvez plus d'information sur les Sciences de l'Ingénieur, l'orientation, les Grandes Ecoles ainsi que sur les Olympiades de Sciences de l'Ingénieur et sur les Sciences de l'Ingénieur au Féminin sur notre site : www.upsti.fr

L'équipe UPSTI

Centrale 2016 Informatique
MP PSI PC TSI

I. Plan de Vol

1. Requête permettant d'obtenir le nombre de vol décollant avant midi le 02 mai 2016 :
   La comparaison pour les jours et les heures fonctionne si les types sont **chaînes de caractères** mais pas nécessairement avec le format **date** et **time** spécifié.

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

2. Requête donnant la liste des numéros de vols au départ d'un aéroport desservant Paris le 02 mai 2016 :

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

   Il est possible d'utiliser "=" au lieu de "LIKE" .

3. Que fait cette requête ?

```
SELECT id_vol
    FROM vol
        JOIN aeroport AS d ON d.id_aero = depart
        JOIN aeroport AS a ON a.id_aero = arrivee
    WHERE d.pays = "France" AND a.pays = "France" AND jour = "2016-05-02"
```

   Elle renvoie les numéros des vols intérieurs à la France pour le jour du 2 mai 2016.

4. Requête permettant d'obtenir les couples de numéros de vols susceptibles d'engendrer des conflits :

```
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
```

   La suppression des doublons de la liste est obtenue par le test Id1<Id2 ;

II. Allocation des niveaux de vol

A. Implantation du problème

1. Fonction renvoyant le nombre de conflits potentiels

```
def nb_conflits():
    long = len(conflit)
    nbre_conflits = 0
```

```
    for i in range(long):
        for j in range(i+1,long): # matrice symétrique
            if conflit[i][j]!=0:
                nbre_conflits += 1
    return nbre_conflits

"""autres manières de faire"""
def nb1_conflits():
    long = len(conflit)
    nbre_conflits = long**2
    for elt in conflit:
        nbre_conflits -= elt.count(0)
    return nbre_conflits

def nb2_conflits():
    return sum( sum(x != 0 for x in l) for l in conflit)
```

2. Complexité de cette fonction

On réalise deux boucles pour parcourir la variable **nb_conflits**, qui est une liste de listes de dimension $(3n \times 3n)$ la complexité de cet algorithme est en $O(n^2)$.

B. Régulation

1. fonction **nb_vol_par_niveau_relatif**

```
def nb_vol_par_niveau_relatif(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]

""" autre solution"""

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

2. fonction **cout_regulation**

```
def cout_regulation(regulation):
    sommets = [3*k+regulation[k] for k in range(len(regulation))]
    cout_regul = 0
    for i in sommets:
        for j in sommets:
            cout_regul += conflit[i][j]
    return cout_regul/2
```

3. complexité de cette fonction

Il y a deux boucles imbriquées de longueur n chacune donc une complexité en $O(n^2)$.

4. Fonction **cout_RFL**

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

5. Nombre possible de régulations

Il y a n vols, pour chaque vol trois possibilités donc 3^n régulations possibles. Ce nombre de régulations est important (surtout pour n grand...$n = 19$ nombre de régulations supérieur au milliard). Il n'est donc pas envisageable de traiter tous les cas possibles.

C. L'algorithme minimal

1. Fonction **coût_du_sommet**

Il faut évaluer le coût d'un sommet en ayant pris soin de retirer les sommets déjà supprimés.

```
def cout_du_sommet(s,etat_sommet):
    cout = sum(conflit[s][i] for i in range(len(etat_sommet))
                             if etat_sommet[i] != 0)
    return cout
```

la complexité de cette fonction est linéaire (du fait de la somme) donc en $O(n)$.

2. Fonction **sommet_de_coût_minimal**

```
def sommet_de_cout_min(etat_sommet):
    rang = 0
    cout_min = float("inf") #initialisation du coût à la valeur infinie
    for s in range(len(etat_sommet)):
        if etat_sommet[s]==2:
            cout = cout_du_sommet(s,etat_sommet)
            if  cout < cout_min:
                cout_min = cout
                rang = s
    return rang
```

la complexité de cette fonction est en $O(n^2)$, car il y a une boucle sur n avec un appel à la fonction **coût du sommet**.

3. Fonction **minimal**

```
def minimal():
    etat_sommet = [2]*len(conflit)
    long = len(etat_sommet)
    for s in range(long//3):
        rang = sommet_de_cout_min(etat_sommet)
        for r in range(rang//3 *3, rang//3*3 + 3):
            etat_sommet[r] = 1 if r==rang else 0
    regulation = [i%3 for i in range(len(etat_sommet)) if etat_sommet[i]==1]
    return regulation
```

La complexité de cette fonction est en $O(n^3)$, car il y a une boucle en $O(n)$ dans laquelle un appel à **sommet_de_cout_minimal** est fait. C'est beaucoup mieux que la « force brute » qui est en $O(3^n)$.

D. Recuit simulé

```
"""le sujet proposait d'utiliser la fonction random de numpy (sans le dire),
ici c'est celle du module random qui est utilisée"""

import random
import numpy as np
def recuit(regulation):
    T = 1000
    cout = cout_regulation(regulation)  #calcul du coût
    n = len(regulation)
    while T>= 1:
        v = random.randint(0,n-1)       #tirage aléatoire du vol
        choix = regulation[v]           #
        c = (choix + random.randint(1,2))%3
        regulation[v] = c               #modification de la valeur de régulation
        cout_alea = cout_regulation(regulation) #calcul du nouveau coût
        if cout_alea >= cout and random.random()> np.exp(-(cout_alea-cout)/T):
            regulation[v] = choix       #reprise du coût précédent
        else :
            cout = cout_alea            #prise en compte de la modif
        T *= 0.99
    return regulation,cout
```

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

A. Acquisition et stockage des données

1. Nombre de bits disponibles pour les données.
   Il y a 16 bits utilisés pour les marques et le contrôle, il en reste 112 pour les données (128 bits en $128\mu s$).
2. Taille du message
   Il y a 112 bits disponibles ; Il en faut 24 pour le numéro d'identification. Pour coder l'altitude il faut $66000-2000+1=64001$ codes, soit 16 bits (ou bien 17 bits si on code directement en binaire). Pour coder la vitesse il faut $5000-(-5000)+1=10001$ codes, en complément à deux par exemple cela nécessite 14 bits. Cela fait un total de 54 (ou 55) bits, ce qui est largement compatible avec la taille disponible. Le message sera transmis en une seule fois.
3. Occupation mémoire.
   Il y a 8 nombres codés chacun sur 4 octets, soit 32 octets. Pour 100h de fonctionnement avec 100 appels par seconde cela représente $100\times100\times3600=36000000$ appels. Cela correspond à un total de $32\times36000000=1.152Go$. C'est aujourd'hui une taille de stockage raisonnable.

B. Estimation du CPA

1. Vecteur position
   $\overrightarrow{OG}(t)=\overrightarrow{OG}(t_0)+(t-t_0)\vec{V}(G/R_0)=(x+v_x\cdot(t-t_0))\vec{i}+(y+v_y\cdot(t-t_0))\vec{j}+(z+v_z\cdot(t-t_0))\vec{k}$
2. Instant de la distance minimale
   Il faut minimiser la distance de l'intrus à l'avion propre, donc la norme du vecteur position $\overrightarrow{OG}(t)$. Il suffit d'exprimer la dérivée temporelle de la norme au carré et de chercher la valeur de

t pour laquelle elle s'annule :
$\frac{\mathrm{d}\|\overrightarrow{OG}(t)\|^2}{\mathrm{d}t} = 2\overrightarrow{OG}(t)\cdot\vec{V}(G/R_0) = 2(\overrightarrow{OG}(t_0) + (t_c - t_0)\vec{V}(G/R_0))\cdot\vec{V}(G/R_0) = 0.$
Ce qui conduit à $t_c = -\frac{\overrightarrow{OG}(t)\cdot\vec{V}(G/R_0)}{\|\vec{V}(G/R_0)\|^2} + t_o$. Cet instant doit être supérieur ou égal à t_0, il faut donc que $\overrightarrow{OG}(t)\cdot\vec{V}(G/R_0) < 0$.

3. Risque de collision
Il n'y a donc pas de risque de collision si $\overrightarrow{OG}(t)\cdot\vec{V}(G/R_0) > 0$ car dans ce cas l'intrus s'éloigne de l'avion propre et $t_c = t_0$.

4. Fonction **calculer_CPA(intrus)**
On se sert des éléments déterminés précédemment et des fonctions de numpy. Il est aussi possible de récupérer les composantes des vecteurs (position et vitesse id,x,y,z,vx,vy,vz,to = intrus) et d'effectuer les calculs « à la main ».

```
def calculer_CPA(intrus):

# récupération de la position et de la vitesse
# calcul du produit scalaire
# détermination de tCPA-t0
    pos,vit = np.array(intrus[1:4]),np.array(intrus[4:7])
    posdotvit = np.dot(pos,vit)  #produit scalaire fourni par numpy
    dt = -posdotvit/np.dot(vit,vit)   #dt = tCPA-t0
#test sur le signe
    if posdotvit > 0:
        return None
    else:
        tCPA = intrus[7]+dt        #calcul du temps
        dCPA = (np.sum((pos+dt*vit)**2))**0.5 # norme du vecteur position
        zCPA = (pos[2]+dt*vit[2])/0.3048   # conversion mètre en pieds
    return [tCPA,dCPA,zCPA]

"""AUTRE SOLUTION (sans numpy)"""
def calculer_CPA(intrus):
    id,x,y,z,vx,vy,vz,t0 = intrus
    tc = t0-(x*vx+y*vy+z*vz)/(vx**2+vy**2+vz**2)
    if tc < t0:
        return None
    dCPA = sqrt((x+vx*(tc-t0))**2 + (y+vy*(tc-t0))**2+(z+vz*(tc-t0))**2)
    zCPA = z+vz*(tc-t0)
    return [tc,dCPA,zCPA]
```

5. Fonction **mettre_a_jour_CPAs**

```
def mettre_a_jour_CPAs(CPAs, id, nv_CPA, intrus_max, suivi_max):
    list_id = [l[0] for l in CPAs] # liste des id
    if id in list_id:
        ind_id = list_id.index(id) # indice dans list_id
        if nv_CPA == None or nv_CPA[0] > suivi_max  :
            del (CPAs[ind_id])
            return None
        if nv_CPA[0] < suivi_max :
            CPAs[ind_id] = [id] + nv_CPA # modification du CPA
```

```
            return ind_id
    elif nv_CPA[0]<suivi_max :
        if len(CPAs) < intrus_max : # liste pas pleine
            CPAs.append([id] + nv_CPA)
        elif nv_CPA[0] < CPAs[-1][1]:
            CPAs[-1] = [id] + nv_CPA
        return len(CPAs) - 1
    return None
"""Autre solution """
def mettre_a_jour_CPAs(CPAs,id,nv_CPA,intrus_max,suivi_max):
    i=0
    while i<len(CPAs) and CPAs[i][0]!=id: #recherche de id
        i+=1
    if i<len(CPAs) and nv_CPA==None:
        del(CPAs[i])
        return None
    elif i<len(CPAs) and nv_CPA[1]<suivi_max:
        CPAs[i]=[id]+nv_CPA
        return i
    elif i=len(CPAs) and nv_CPA[1]<suivi_max and len(CPAs)<intrus_max:
        CPAs.append([id]+nv_CPA)
        return i+1
    elif i=len(CPAs) and nv_CPA[1]<CPAs[-1][1] and len(CPAs)=intrus_max:
        CPAs[-1]=[id]+nv_CPA
        return i
```

6. Fonction **replacer(ligne,CPA)**

```
def replacer(ligne, CPAs):
    cpa = CPAs[ligne]
    ind = len([l[1] for l in CPAs if l[1]<cpa[1] ]) # temps < tps du cpa
    del(CPAs[ligne])      # suppression de la ligne
    CPAs.insert(ind, cpa) # insertion de la ligne à la bonne place

"""autre solution plus efficace (tri par insertion) """
def replacer1(ligne, CPAs):
    t = CPAs[ligne][1]
    n = len(CPAs)
    while ligne > 0 and CPAs[ligne-1][1] > t:
        CPAs[ligne-1], CAPs[ligne] = CPAs[ligne], CPAs[ligne-1]
        ligne -= 1
    while ligne < n-1 and CPAs[ligne+1][1] < t:
        CPAs[ligne+1], CAPs[ligne] = CPAs[ligne], CPAs[ligne+1]
        ligne += 1
```

7. Fonction **enregistrer_CPA**

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

C. Évaluation des paramètres généraux du système TCAS

1. Temps entre détection et CPA
   Le cas le plus défavorable est si les deux avions volent face à face. Dans ce cas à 900km/h s'ils sont situés à 60km de distance alors ils auront chacun parcouru 30km au CPA. Le temps écoulé entre détection et le CPA est $\frac{30}{900}$h = 2 minutes. Cela représente 120 secondes ce qui est supérieur au temps 100 secondes du suivi max. Cela doit permettre aux pilotes de réagir.
2. Temps de manœuvre avant CPA
   Avec une vitesse ascensionnelle de 1500 pieds par minute, il faut 20 secondes pour obtenir une différence de 500 pieds ou 10 si les deux pilotes réagissent en sens opposés.
3. Alarme dans le cockpit
   La durée de 25 secondes, supérieure aux 20 secondes calculées précédemment, permet au pilote de réagir et de pouvoir dégager afin d'avoir une différence d'altitude minimale de 500 pieds.
4. Temps maximum d'exécution de boucle
   S'il faut vérifier chaque intrus au moins une fois par seconde et qu'il y a 30 intrus à surveiller cela implique un tour de boucle en $\frac{1}{30}$ = 33 ms.
5. Facteur limitant
   je ne vois pas quoi répondre ? peut être que le facteur limitant est la durée des mesures (temps de réponse des capteurs) plus que des temps de calcul ?