2015-020-COR engendré le 8 mars 2016 à 17 :12
à partir du fichier source modifié le 2016-03-08 17 :12 :52
ConTEXt : 20151007 12 :03 MKIV
Engine : LuaTeX 0.80.0
PGF : 3.0.1

Durée : 3 heures — Calculatrices autorisées

I Plan de vol

I.A – Nombre de vols qui décollent le 2 mai 2016 avant midi.

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

I.B – Nécessité d'une jointure pour déterminer les aéroports parisiens.

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

I.C – Elle liste les numéros de tous les vols domestiques français (partant de France et arrivant en France) qui doivent décoller le 2 mai 2016.

I.D – Vols suivant des chemins inverses au même niveau, le même jour.

```
SELECT
    v1.id_vol,
    v2.id_vol
FROM
    vol AS v1 JOIN
    vol AS v2 ON
        v1.depart = v2.arrivee AND
        v1.arrivee = v2.depart AND
        v1.jour = v2.jour AND
        v1.niveau = v2.niveau
WHERE
    v1.jour = '2016-05-02' AND
    v1.id_vol < v2.id_vol -- pour éliminer les doublons
```

II Allocation de niveaux de vol

II.A – Implantation du problème

II.A.1) Nombre de conflits

Première solution : on parcourt toutes les cases de la matrice et on divise par deux.


```
def nb_conflits():
    """Compte le nombre de conflits potentiels"""
    nb = 0
    for ligne in conflit:
        for c in ligne:
            if c != 0: nb += 1
    return nb // 2
```

Deuxième solution : on ne parcourt que le triangle (strictement inférieur).

```
def nb_conflits():
    """Compte le nombre de conflits potentiels"""
    nb = 0
    for i in range(len(conflit)):
        for j in range(i):
            if conflit[i][j] != 0: nb += 1
    return nb
```

II.A.2) Complexité

Cas 1 : boucle externe exécutée $3n$ fois, boucle interne exécutée $3n$ fois corps de boucle $O(1)$ ce qui donne $9n^2\,O(1) = O(n^2)$.

Cas 2 : corps de la boucle en O(1) exécutée k fois pour k variant de 0 à $3n-1$ ce qui donne une complexité de $(\sum_{k=0}^{3n-1} k)\,O(1) = O(n^2)$.

Il s'agit du parcours de toutes (ou la moitié) les cases d'une matrice $3n \times 3n$, le résultat est donc cohérent.

II.B – Régulation

II.B.1) Nombre de vols par niveau relatif

On parcourt la liste en accumulant dans la case adéquate.

```
def nb_vol_par_niveau_relatif(regulation):
    """Compte le nombre de vol par niveau relatif"""
    res = [0, 0, 0]
    for k in regulation:
        res[k] += 1
    return res
```

II.B.2) Cout d'une régulation

a) On somme le cout de tous les couples de vols en sachant que le vol k est représenté par le sommet $3k + r_k$.

Première solution : on somme tous les cas possibles et on divise par deux.

```
def cout_regulation1(regulation):
    """Calcule le cout d'une régulation"""
    cout = 0
    for v1, r1 in enumerate(regulation):
        for v2, r2 in enumerate(regulation):
            cout += conflit[3*v1+r1][3*v2+r2]
    return cout // 2
```

Deuxième solution on ne considère que les $j < i$.

```
def cout_regulation2(regulation):
    """Calcule le cout d'une régulation"""
    cout = 0
    for v1 in range(len(regulation)):
        for v2 in range(v1):
            cout += conflit[3*v1+regulation[v1]][3*v2+regulation[v2]]
    return cout
```

b) Complexité Même calcul qu'en II.A.2 avec n à la place de $3n$: $O(n^2)$.

c) Comme le précise l'énoncé, la régulation où tous les avions volent à leur RFL est la liste qui ne contient que des zéros. La seule « difficulté » est de calculer n.

```
def cout_RFL():
    """Cout si tous les avions volent à leur RFL"""
    return cout_regulation((len(conflit)//3 * [0])
```

II.B.3) Nombre de régulations

Pour chaque vol, trois choix sont possibles. Il y a donc 3^n régulations possibles.

D'après l'introduction, $n > 10000$ il y a donc au moins $3^{10000} \approx 10^{4800}$ calculs de cout à faire. Ce qui est totalement inaccessible, il suffit pour s'en convaincre de comparer à l'âge de l'univers : 5×10^{17} s.

Même en réduisant le nombre de vol à une centaine (en travaillant par exemple par tranche horaire et zone géographique) le nombre de calculs de cout à faire $\approx 10^{48}$ dépasserait d'un facteur incommensurable le nombre de nanosecondes dans la vie de l'univers !

II.C – Algorithme minimal

II.C.1) Cout d'un sommet

a) On somme la ligne des conflits correspondant au sommet s en sautant les sommets supprimés.

Version avec `enumerate`.

```
def cout_du_sommet(s, etat_sommet):
    """Calcul du cout d'un sommet"""
    cout = 0
    for j, c in enumerate(conflit[s]):
        if etat_sommet[j] != 0:
            cout += c
    return cout
```

Version avec indice explicite.

```
def cout_du_sommet(s, etat_sommet):
    """Calcul du cout d'un sommet"""
    cout = 0
    for j in range(len(etat_sommet)):
        if etat_sommet[j] != 0:
            cout += conflit[s][j]
    return cout
```

b) Boucle exécutée $3n$ fois avec un corps en $O(1)$: $O(n)$.

II.C.2) Sommet de cout minimum

a) On parcourt les sommets non supprimés ni attribués (à l'état 2) et on sélectionne celui de cout minimal. Au cas où plusieurs sommets ont même cout, la solution présentée sélectionne le premier ; en cas d'égalité, il pourrait être intéressant de donner la priorité aux sommets RFL.

```
def sommet_de_cout_min(etat_sommet):
    """Recherche le sommet de cout minimal"""
    cout_min = None # valeur du cout min (inconnue au départ)
    sommet = None # numéro du sommet de cout min (initialisation utile si le graphe est vide)
    for s, e in enumerate(etat_sommet):
        if e == 2:
            cout = cout_du_sommet(s, etat_sommet)
            if cout_min is None or cout < cout_min:
                cout_min = cout
                sommet = s
    return sommet
```

b) Corps de boucle en $O(n)$ exécuté au pire $3n$ fois : $O(n^2)$.

II.C.3) Minimal

a) On part du graphe complet (tous les sommets à l'état 2). Et on recherche successivement les sommets de cout minimal. Le point délicat pourrait être de définir la condition d'arrêt, mais comme à chaque étape, on traite un vol, on sait qu'il y aura exactement n étapes.


```
def minimal():
    """Calcul d'une régulation par l'algorithme minimal"""
    n = len(conflit) // 3   # nombre de vols
    etat_sommet = 3*n * [2] # aucun sommet sélectionné ni supprimé
    regulation = n * [None] # niveaux relatifs inconnus au départ

    for _ in range(n): # on sait qu'il y a n vols à affecter
        smin = sommet_de_cout_min(etat_sommet)
        vol = smin // 3  # numéro du vol correspondant à ce sommet
        regulation[vol] = smin % 3 # attribution du niveau relatif à ce vol
        # sélection du sommet min et suppression des deux autres sommets du vol
        etat_sommet[3*vol] = etat_sommet[3*vol+1] = etat_sommet[3*vol+2] = 0
        etat_sommet[smin] = 1
    return regulation
```

b) Corps de la boucle en $O(n^2)$ (`sommet_de_cout_min`) exécutée n fois : $O(n^3)$.

Cet algorithme est donc abordable, mais ne garantit pas d'obtenir une solution optimale.

II.D – Recuit simulé

```
def etape(regulation, T, cout):
    """
    Réalise une étape de recuit.

    Paramètres
        regulation (E): la régulation courante
        regulation (S): la régulation modifiée
        T - nombre: la température du recuit
        cout - int: le cout de la régulation courante
    Résultat
        le cout de la régulation modifiée
    """
    k = randint(n)
    ancien_niveau = regulation[k]
    regulation[k] = (ancien_niveau + 1 + randint(2)) % 3
    nouveau_cout = cout_regulation(regulation)
    if nouveau_cout <= cout:
        # si baisse du cout, on conserve la nouvelle régulation
        return nouveau_cout

    # sinon, un petit coup de proba
    proba = exp((cout - nouveau_cout)/T)
    if random() < proba:
        # random suit la loi uniforme sur [0,1[
        # donc pour tout x dans [0,1] : P(random() < x) = x
        return nouveau_cout

    # si random >= proba, on restaure la valeur initiale
    regulation[k] = ancien_niveau
    return cout

def recuit(regulation):
    """Application du recuit simulé à une régulation"""
    cout = cout_regulation(regulation)
    T = 1000
    while T > 1:
        cout = etape(regulation, T, cout)
        T *= 0.99
```

III TCAS

III.A – Acquisition et stockage des données

III.A.1) Longueur max du message

Un débit de 10^6 bits pas seconde, donne 1 bit par microseconde. Donc en 128 µs on peut émettre 128 bits. En enlevant les $6+4+6=16$ bits de service, il reste $128-16=112$ bits utiles.

III.A.2) Longueur nécessaire

Pour chaque grandeur à représenter, il faut déterminer le nombre de valeurs possibles. Le nombre de bits optimal pour un codage en binaire est alors $\lceil \log_2(\text{nb valeurs}) \rceil$. En effet, n bits permettent de coder 2^n valeurs différentes.

On obtient ainsi

Information	Nombre de valeurs	Nombre de bits
Altitude	$66000-2000+1=64001$	16
Vitesse	$5000+5000+1=10001$	14

Soit un total de $24+14+16=54$ bits. Il est donc parfaitement possible de récupérer ces informations en une seule interrogation.

On peut remarquer qu'être précis au pied près n'est pas réaliste. On peut donc réduire encore l'occupation de la fréquence en codant l'altitude et la vitesse par incréments de 10, voire 100 pieds.

III.A.3) Volume mémoire

8 valeurs × 4 octets × 100 appels/seconde × 3600 secondes/heure × 100 heures $= 1{,}152 \times 10^9$ octets.

La mémorisation des données sur 100 heures de vol nécessite de l'ordre d'un Go. Ce type de volume mémoire est disponible dans une large gamme de technologies. Il est donc fort probable que disposer à un cout raisonnable d'un type de mémoire de cette taille répondant aux exigences de l'aviation ne pose pas de problème. Il ne s'agit donc pas d'une contrainte technique forte.

III.B – Estimation du CPA

III.B.1) Extrapolation de la trajectoire

Les deux avions étant en mouvement rectiligne uniforme, le mouvement de l'intrus par rapport à l'avion propre est également rectiligne uniforme de vitesse constante $\vec{V}$. Il vient donc

$$\overrightarrow{OG}(t) = \overrightarrow{OG}(t_0) + (t-t_0)\vec{V}$$

III.B.2) Instant du CPA

La distance entre les deux avions à l'instant t est $d(t) = \|\overrightarrow{OG}(t)\|$. On cherche l'instant t_c pour lequel $d(t_c)$ est minimum. Autrement dit l'instant où $d'(t)=0$ ($d(t)$ est trivialement dérivable et n'a qu'un seul extrémum qui est son minimum).

Il est plus simple et cela revient au même de calculer l'instant où la dérivée de $d^2(t)$ s'annule.

$$\begin{aligned} d^2(t) &= \left(\overrightarrow{OG}(t_0) + (t-t_0)\vec{V}\right)^2 \\ (d^2)'(t) &= 2\,\vec{V}\cdot\left(\overrightarrow{OG}(t_0) + (t-t_0)\vec{V}\right) \\ &= 2\,\vec{V}\cdot\overrightarrow{OG}(t_0) + 2(t-t_0)\vec{V}^2 \end{aligned}$$

En posant $(d^2)'(t_c)=0$ il vient

$$\begin{aligned} (t_c-t_0)\vec{V}^2 &= -\vec{V}\cdot\overrightarrow{OG}(t_0) \\ t_c &= t_0 - \frac{\vec{V}\cdot\overrightarrow{OG}(t_0)}{\vec{V}^2} \end{aligned}$$

III.B.3) Risque de collision

Si $\overrightarrow{OG}(t_0)\cdot\vec{V} \geqslant 0$, alors $t_c \leqslant t_0$, le CPA est antérieur à la dernière acquisition de l'intrus, les deux avions sont donc en train de s'éloigner, il n'y a plus de risque de collision (sauf si un des deux avions modifie sa trajectoire).

III.B.4) Fonction `calculer_CPA`

Il suffit d'appliquer directement la réponse aux deux questions précédentes. Puisqu'on suppose que l'avion propre vole horizontalement la coordonnée z du CPA donne directement la différence d'altitude à cet instant entre les deux avions. Il faut juste penser à la convertir en pieds (facteur de conversion donné au début de la partie I).


```
def calculer_CPA(intrus):
    """Calcul du CPA d'un intrus."""
    id, x, y, z, vx, vy, vz, t0 = intrus

    deltat = - (x*vx + y*vy + z*vz) / (vx*vx + vy*vy + vz*vz) # tc - t0
    if (deltat <= 0): return None

    xc, yc, zc = x + deltat * vx, y + deltat * vy, z + deltat * vz # CPA
    dc = sqrt(xc*xc + yc*yc + zc*zc)
    return [t0+deltat, dc, zc/0.3048] # on demande la différence d'altitude en pieds
```

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

III.C.1) Mise à jour

Une fonction de recherche peut être utile. Il n'y a pas 36 solutions, la liste n'est pas triée par identifiant.

```
def rechercher_intrus(CPAs, id):
    """
    Recherche un intrus dans la liste des CPA.

    Paramètres
        CPAs : liste des CPA ([id, ...])
        id : identifiant de l'intrus à rechercher
    Résultat
        None si l'intrus ne figure pas dans la liste
        son indice sinon
    """
    for i, intrus in enumerate(CPAs):
        if intrus[0] == id: return i
    return None
```

Il faut veiller à imbriquer correctement les `if`, ne pas oublier que `suivi_max` est à interpréter par rapport à l'heure courante et bricoler un peu avec l'identifiant.

```
def mettre_a_jour_CPAs(CPAs, id, nv_CPA, intrus_max, suivi_max):
    """Met à jour la liste CPAs avec un nouveau CPA."""
    pos = rechercher_intrus(CPAs, id)
    if nv_CPA is None or nv_CPA[0] > time() + suivi_max:
        if pos is not None: del CPAs[pos]
        res = None
    elif pos is not None:
        CPAs[pos] = [id] + nv_CPA
        res = pos
    elif len(CPAs) < intrus_max:
        CPAs.append([id] + nv_CPA)
        res = len(CPAs) - 1
    elif nv_CPA[0] < CPAs[-1][1]:
        CPAs[-1] = [id] + nv_CPA
        res = len(CPAs) - 1
    return res
```

III.C.2) Tri du tableau

Plusieurs stratégies sont possibles :

- on supprime la ligne à replacer de la liste, on obtient ainsi une liste triée et
  - on ajoute une case « vide » à la fin et on décale la fin de la liste jusqu'à arriver à l'endroit où placer la ligne ;
  - on recherche l'emplacement de la ligne et on utilise `insert` pour l'y placer ; puisque la liste est triée, on peut imaginer utiliser une recherche rapide (dichotomie) pour trouver l'emplacement, mais compte tenu du petit nombre d'éléments, le jeu n'en vaut probablement pas la chandelle ;
- on regarde si la ligne est à déplacer vers le début ou la fin de la liste et on pratique par décalages successifs dans le bon sens

- soit par permutation entre deux éléments contigus ;
- soit en utilisant une variable intermédiaire pour mémoriser la ligne à replacer ;

– on applique un algorithme de tri classique à l'ensemble de la liste sans utiliser le fait que la liste est déjà quasiment triée.

Dans tous les cas, sauf à imaginer une solution aberrante, la performance n'est pas un critère compte tenu de la taille extrêmement réduite de la liste. Il me semble raisonnable de privilégier la lisibilité de l'algorithme.

Solution par décalage vers l'avant ou l'arrière. On pourrait se passer des `if` car ils sont immédiatement retestés dans les boucles `while`, mais cela rendrait le programme encore plus difficile à lire.

```
def replacer(ligne, CPAs):
    """Remet la ligne indiquée à sa place."""
    cpa = CPAs[ligne] # le CPA à replacer
    tc = cpa[1] # le tCPA correspondant
    if ligne > 0 and tc < CPAs[ligne-1][1]:
        # il faut avancer la ligne vers le début de la liste
        i = ligne
        while i > 0 and tc < CPAs[i-1][1]:
            CPAs[i] = CPAs[i-1]
            i -= 1
        CPAs[i] = cpa
    elif ligne < len(CPAs) - 1 and tc > CPAs[ligne+1][1]:
        # il faut reculer la ligne vers la fin de la liste
        i = ligne
        while i < len(CPAs) - 1 and tc > CPAs[i+1][1]:
            CPAs[i] = CPAs[i+1]
            i += 1
        CPAs[i] = cpa
```

Solution par suppression et réinsertion.

```
def replacer(ligne, CPAs):
    """Remet la ligne indiquée à sa place."""
    cpa = CPAs[ligne] # le CPA à replacer
    tc = cpa[1] # le tCPA correspondant
    del CPAs[ligne]
    i = len(CPAs)
    while i > 0 and tc < CPAs[i-1][1]: i -= 1
    CPAs.insert(i, cpa)
```

III.C.3) Fonction `enregistrer_CPA`

Pas de problème particulier si le candidat sait lire le programme de la figure 5. Il faut cependant veiller à ne pas appeler `replacer` si `mettre_a_jour_CPAs` a renvoyé `None`, à moins que `replacer` ait prévu que son paramètre `ligne` puisse être `None`.

```
def enregistrer_CPA(intrus, CPAs, intrus_max, suivi_max):
    """Traite l'acquisition d'un nouvel intrus."""
    cpa = calculer_CPA(intrus)
    pos = mettre_a_jour_CPAs(CPAs, intrus[0], cpa, intrus_max, suivi_max)
    if pos is not None: replacer(pos, CPAs)
```

III.D – Paramètres généraux

III.D.1) Temps d'approche

Si le CPA est proche de l'avion propre, cela signifie que les deux avions volent à peu près l'un vers l'autre. Autrement dit il faut additionner leur vitesse pour obtenir leur vitesse relative : ils se rapprochent à 1800 km/h, soit 30 km/min. Ainsi deux avions séparés de 60 km se rejoindront en 2 minutes.

On note que `suivi_max` est inférieur mais du même ordre de grandeur. On peut raisonnablement supposer plus un intrus est loin, moins sa localisation sera précise et que `suivi_max` est réglé pour ne pas prendre en considération les avions en limite de portée du système et qui risqueraient de déclencher de fausses alarmes, suite par exemple à une réception de mauvaise qualité.

III.D.2) Temps de montée

Si les deux avions manœuvrent simultanément en sens opposé leur différence de vitesse ascensionnelle sera de 3000 pieds par minute, soit 50 pieds par seconde. Il leur suffira donc de 10 secondes pour s'écarter de 500 pieds.

Si un seul des deux avions manœuvre (l'autre n'ayant pas perçu le danger), il faudra deux fois plus de temps, soit 20 secondes pour atteindre un écart vertical de 500 pieds.

III.D.3) Zone d'alerte

Émettre une alarme 25 secondes avant le CPA laisse donc au pilote quelques secondes pour analyser la situation puis lui permet de changer confortablement d'altitude, même si l'intrus ne réagit pas.

Émettre une alarme pus tôt pourrait risquer, dans les zones à fort trafic, de créer un conflit avec un autre intrus.

III.D.4) Temps maximum pour effectuer une boucle

Chaque boucle met à jour la position d'un seul intrus à la fois. Pour acquérir la position de chaque intrus chaque seconde, il faut donc effectuer au minimum 30 boucles par seconde. Une boucle ne doit donc pas durer plus d'une trentaine de millisecondes.

III.D.5) Facteur limitant et durée minimum

Les algorithmes étudiés précédemment ont, au pire, des complexités en $O(n)$. Bien que n'ayant pas étudié le détail des fonctions `acquerir_intrus` et `traiter_CPAs` on ne voit pas quels algorithmes complexes elles pourraient mettre en œuvre. De plus le système travaille sur un volume de données extrêmement limité. Ainsi, le facteur limitant la vitesse d'exécution du système est très probablement les entrées-sorties, autrement dit le temps nécessaire à échanger des données avec l'extérieur (avion intrus, station au sol, etc.)

Nous avons vu que la durée d'émission du message d'un intrus donnant son identité, son altitude et sa vitesse est de l'ordre d'une centaine de microsecondes. Si l'intrus est situé à 30 km le temps de parcours de l'onde radio entre les deux avions sera du même ordre de grandeur : $3\times10^4\,\mathrm{m}/3\times10^8\,\mathrm{m\cdot s^{-1}} = 100\,\mu\mathrm{s}$. Comme le transpondeur de l'intrus répond à un message d'interrogation de l'avion propre, il faut doubler ce temps. Ainsi l'acquisition d'information auprès d'un intrus situé à 30 kilomètres prend au moins 200 µs (temps de l'aller-retour) plus 100 µs pour la durée des messages. Si on prend en compte le fait que le système TCAS reçoit également des données de stations au sol, et que la fonction `traiter_CPAs` peut être amenée à contacter un intrus pour coordonner une manœuvre d'évitement, on peut estimer que les temps d'entrée-sortie pour une boucle peuvent être de l'ordre de grandeur d'une milliseconde, ce qui constitue donc le temps d'exécution minimum de cette boucle.

Ce temps est largement inférieur au temps maximum (30 millisecondes) et permet donc de répondre aux spécifications du système. À condition toutefois de gérer une éventuelle alarme de manière asynchrone (l'émission d'une alarme dans le cockipt nécessitant vraisemblablement plusieurs secondes pour être perçue par le pilote).