Épreuve d'informatique X/ENS - 2016 - PSI/PT

Détection de collisions entre particules

Partie I. Simulation du mouvement des particules

Question 1 : On écrit la fonction de déplacement d'une particule en tenant compte des rebonds sur les bords.

```
1  def deplacerParticule(particule, largeur, hauteur):
2      x,y,vx,vy = particule
3      if (x + vx <= 0) or (x + vx >= largeur):
4          vx = -vx
5      x += vx
6      if (y + vy <= 0) or (y + vy >= hauteur):
7          vy = -vy
8      y += vy
9      return((x,y,vx,vy))
```

Partie II. Représentation par une grille

Question 2 : La fonction « `nouvelleGrille` » retourne un tableau de tableaux correspondant à une grille de taille *largeur* × *hauteur* rempli de `None` (grille vide).

```
1  def nouvelleGrille(largeur, hauteur):
2      grille = []
3      for i in range(largeur):
4          grille.append([])
5          for j in range(hauteur):
6              grille[i].append(None)
7      return(grille)
```

Question 3 : On commence par récupérer les largeur et hauteur de la grille (lignes 2 et 3). Dans la double boucle des lignes 5 à 15, on détecte les particules présentes. On déplace chacune des particules trouvées que l'on place dans la nouvelle grille (ligne 15). Si une particule est déjà présente dans la case cible, il y a collision, le programme s'arrête et retourne `None` (ligne 13). S'il n'y a pas eu collision, la fonction retourne la nouvelle grille (ligne 16).

(PSI*)

```
1  def majGrilleOuCollision(grille):
2      largeur = len(grille)
3      hauteur = len(grille[0])
4      Grille = nouvelleGrille(largeur,hauteur)
5      for i in range(largeur):
6          for j in range(hauteur):
7              if grille[i][j] != None:
8                  particule = deplacerParticule(grille[i][j],largeur,hauteur)
9                  x,y,vx,vy = particule
10                 X = int(x)
11                 Y = int(y)
12                 if Grille[X][Y] != None:
13                     return(None)
14                 else:
15                     Grille[X][Y] = particule
16     return(Grille)
```

Question 4 : La boucle conditionnelle des lignes 3 à 7 effectue des mises à jour de la grille tant qu'il n'y a pas de collision et que le temps maximum n'a pas été atteint. S'il y a collision, la fonction retourne l'instant de la collision (ligne 7). Sinon, la fonction retourne `None` (ligne 8).

```
1  def attendreCollisionGrille(grille, tMax):
2      t = 0
3      while grille != None and t < tMax:
4          t = t+1
5          grille = majGrilleOuCollision(grille)
6          if grille == None:
7              return(t)
8      return(None)
```

Question 5 : Dans le pire des cas, il n'y a pas de collision et la boucle conditionnelle de la fonction «`attendreCollisionGrille`» est exécutée `tMax` fois. Dans cette boucle conditionnelle, il y a trois affectations de coût constant et un appel à la fonction «`majGrilleOuCollision`». La fonction «`majGrilleOuCollision`» fait une fois appel à la fonction «`nouvelleGrille`» de coût $O(largeur \times hauteur)$ et effectue au plus $largeur \times hauteur$ fois les instructions de la double boucle itérative. Le coût de «`deplacerParticule`» et des autres instructions de la boucle est en $O(1)$ donc le coût total de la fonction «`majGrilleOuCollision`» est en $O(largeur \times hauteur)$.
On en déduit que le coût global de «`attendreCollisionGrille`» est donc en $O(tMax \times largeur \times hauteur)$

Partie III. Représentation par liste de particules

Question 6 : On commence par écrire une fonction «`carreDistance`» qui retourne le carré de la distance entre un point de coordonnées `(x1,y1)` et un point de coordonnées `(x2,y2)`.
La fonction «`detecterCollisionEntreParticules`» récupère les informations des particules `p1` et `p2` et retourne `vrai` si la distance entre leurs centres est inférieure à deux fois le rayon (ligne 10) et `faux` sinon (ligne 12).

```
1  def carreDistance(x1,y1,x2,y2):
2      X = x2-x1
3      Y = y2-y1
4      return(X*X+Y*Y)
5
6  def detecterCollisionEntreParticules(p1, p2):
7      x1,y1,vx1,vy1 = p1
8      x2,y2,vx2,vy2 = p2
9      if carreDistance(x1,y1,x2,y2) <= 4*rayon*rayon:
10         return(True)
11     else :
12         return(False)
```

Question 7 : La fonction `maj` commence par récupérer les informations de l'ensemble de particules (ligne 2). On crée un nouvelle liste de particules (ligne 3). Dans la boucle de lignes 4 à 5, on considère chaque particule, on en calcule la nouvelle position et la nouvelle vitesse. On met le résultat dans la nouvelle liste de particules. On retourne le résultat en sortie de boucle (ligne 6).

```
1  def maj(particules):
2      largeur,hauteur,liste = particules
3      Liste = []
4      for i in range(len(liste)):
5          Liste.append(deplacerParticule(liste[i],largeur,hauteur))
6      return((largeur,hauteur,Liste))
```

Question 8 : On commence par mettre à jour les particules (ligne 2). Dans la double boucle itérative, on recherche une éventuelle collision entre les particules. Si c'est le cas, on retourne `None` (ligne 8). Si aucune collision a été détectée, on retourne le nouvel ensemble de particules (ligne 9).

```
1  def majOuCollision(particules):
2      Particules = maj(particules)
3      liste = Particules[2]
4      nbParticules = len(liste)
5      for i in range(nbParticules):
6          for j in range(i+1,nbParticules):
7              if detecterCollisionEntreParticules(liste[i],liste[j]):
8                  return(None)
9      return(Particules)
```

Question 9 : Dans la fonction « attendreCollisionGrille », on initialise le temps à 0 (ligne 2). Dans la boucle conditionnelle des lignes 3 à 7, on effectue au plus `tMax` mis à jour. S'il y a eu collision (lignes 6 et 7), on retourne la valeur du temps `t`. Si aucune collision n'a eu lieu, on retourne `None` (ligne 8).

```
1  def attendreCollision(particules, tMax):
2      t = 0
3      while particules != None and t < tMax:
4          t = t+1
5          particules = majOuCollision(particules)
6          if particules == None:
7              return(t)
8      return(None)
```

La boucle conditionnelle des lignes 3 à 7 est exécutée au plus `tMax` fois (quand il n'y a aucune détection de collision). La fonction «`majOuCollision`» a un coût en $O(n^2)$: on effectue $(n-1)+(n-2)+...+1=\frac{n(n-1)}{2}$ fois les instructions du cœur de la double boucle et la fonction «`detecterCollisionEntreParticules`» a un coût en $O(1)$.

En conclusion, le coût de la fonction «`attendreCollision`» est en $O(n^2.\texttt{tMax})$.

Question 10 : Entre les instants t et $t+1$, une particule se déplace d'au plus `vMax` donc deux particules peuvent rentrer en collision si à l'instant $t+1$ les deux particules sont à une distance inférieure ou égale à $2\,$`rayon`. Sachant que chaque particule peut parcourir une distance de `vMax`, la distance est inférieure à $(2*\texttt{vMax}+2*\texttt{rayon})$.

conclusion : La distance maximale entre deux particules à l'instant t est $(2*\texttt{vMax}+2*\texttt{rayon})$ pour qu'elles puissent rentrer en collision.

Question 11 : On suppose que la liste des particules est triée par abscisses croissantes. On met à jour les positions des particules (ligne 2). On récupère la liste des particules (ligne 3). Dans la boucle itérative des lignes 7 à 14, on teste si la i-ième particule est entrée en collision. On limite les tests aux particules d'indice k d'abscisse plus grande que celle de i (donc $k > i$) et telles que la différence entre les abscisses des points P_i et P_k avant la mise à jour était inférieure à $2\,(v_{max} + r)$. S'il y a une collision, on retourne `None` (ligne 13), sinon, on retourne la liste des particules mise à jour (ligne 15).

```
1  def majOuCollisionX(particules):
2      Particules = maj(particules)
3      Liste = particules[2]
4      NouvelleListe = Particules[2]
5      nbParticules = len(Liste)
6      d = 2*(rayon+vMax)
7      for i in range(nbParticules):
8          P = Liste[i]
9          k = i+1
10         while k < nbParticules and abs(Liste[k][0]-P[0]) < d:
11             if detecterCollisionEntreParticules(NouvelleListe[i],
12                                                 NouvelleListe[k]):
13                 return(None)
14             k += 1
15     return(Particules)
```

Partie IV. Trier des listes partiellement triées

Question 12 Dans la fonction `scm`, la variable `n` représente le nombre d'éléments de la liste `s`, la variable `d` l'indice du premier terme de la suite croissante maximale à l'étude (initialisée à 0), `res` la liste résultat et `i` un indice de parcours de la liste. Dans la boucle des lignes 6 à 12, on parcourt tous les éléments de la liste et on détecte les indices de fin de suite croissante maximale (variable f). Dès que l'on a détecté la fin d'une telle suite, on ajoute le couple (d, f) à la liste résultat `res` (ligne 10) et on met à jour la variable `d`. En sortie de boucle, on ajoute le couple $(d, n-1)$ qui caractérise la dernière *scm* (ligne 13).

```
1  def scm(s):
2      n = len(s)
3      d = 0
4      res = []
5      i = 0
6      while i < n:
7          if i == d or s[i] >= s[i-1]:
8              f = i
9          else :
10             res.append((d,f))
11             d = i
12         i += 1
13     res.append((d,n-1))
14     return(res)
```

Question 13 : Dans la fonction « `fusionner` », on commence par récupérer les indices de début et de fin des deux *scm* `r1` et `r2` (lignes 2 et 3). On recopie dans une liste `l1` la partie de la liste correspondant à `r1` (ligne 4). Dans la boucle itérative des lignes 9 à 21, on trie les éléments de la liste entre les indices `d1` et `f1`. On parcourt simultanément la liste `l1` et la portion de liste correspondant à la *scm* `r2` à l'aide des indices `i1` et `i2`. Si `i2` $\geqslant$ `n1`, c'est que tous les éléments de la *scm* `r1` ont été rangés dans `s`, donc l'élément suivant à ranger appartient à la liste représentée par `r2` (ligne 10 à 11). On a le cas symétrique avec `i2` $>$ `f2` (ligne 12 à 13). Sinon on compare les éléments d'indice `i1` dans `l1` et d'indice `i2` dans `s` et suivant le cas de figure, on «range» `l1[i1]` ou `s[i2]` dans `s` (lignes 14 à 18).

```
1  def fusionner(s, r1, r2):
2      d1, f1 = r1
3      d2, f2 = r2
4      l1 = copier(s, d1, f1)
5      i = d1
6      i1 = 0
7      n1 = f1-d1+1
8      i2 = d2
9      for i in range(d1,f2+1):
10         if i1 >= n1:
11             s[i] = s[i2]; i2 += 1
12         elif i2 > f2:
13             s[i] = l1[i1]; i1 += 1
14         elif l1[i1] < s[i2]:
15             s[i] = l1[i1]; i1 += 1
16         else :
17             s[i] = s[i2];
18             i2 += 1
```

Question 14 : On écrit la fonction `depileFusionneRemplace` qui dépile les deux *scm* au sommet de la pile, effectue leur fusion et empile la *scm* résultat.

```
1  def depileFusionneRemplace(s, pile):
2      scm2 = pile.pop()
3      scm1 = pile.pop()
4      fusionner(s,scm1,scm2)
5      pile.append((scm1[0],scm2[1]))
```

Question 15 : On commence par écrire une fonction `tailleSCM` qui retourne la taille de la *scm* passée en argument.
Dans la procédure `alphaTri`, on applique l'algorithme α-tri. On calcule les *scm* (ligne 7). Dans la boucle des lignes 8 à 11, on insère dans la pile les *scm* une à une en appliquant la politique de fusion décrite par l'énoncé : si deux fois la taille de la dernière *scm* de la pile est inférieure à la taille de l'avant dernière `scm` de la pile, on les fusionne et on met le résultat dans la pile. On effectue cette dernière opération tant que c'est possible (boucle conditionnelle des lignes 10 et 11). Dans la boucle conditionnelle des lignes 12 et 13, on effectue les fusions des `scm` restant dans la pile jusqu'à ce qu'il ne reste plus qu'une seule *scm*. La liste est alors triée, on peut quitter la procédure.

```
1  def tailleSCM(scm):
2      return(scm[1]-scm[0]+1)
3
4  def alphaTri(s):
5      pile = []
6      SCM = scm(s)
7      n = len(SCM)
8      for i in range(0,n):
9          pile.append(SCM[i])
10         while len(pile) > 1 and 2*tailleSCM(pile[-1])>tailleSCM(pile[-2]):
11             depileFusionneRemplace(s,pile)
12     while len(pile) > 1:
13         depileFusionneRemplace(s,pile)
14     return()
```