Graphes et réseaux

Sujets de concours

CONCOURS CENTRALE•SUPÉLEC

Informatique

MP, PC, PSI, TSI

3 heures Calculatrices autorisées

2017

Mars Exploration Rovers
Mission d'exploration martienne

Mars Exploration Rovers (MER) est une mission de la NASA qui cherche à étudier le rôle joué par l'eau dans l'histoire de la planète Mars. Deux robots géologues, Spirit et Opportunity (figure 1), se sont posés sur cette planète, sur deux sites opposés, en janvier 2004. Leur mission est de rechercher et d'analyser différents types de roches et de sols qui peuvent contenir des indices sur la présence d'eau. Ils sont équipés de six roues et d'une suspension spécialement conçue pour leur permettre de se déplacer quelle que soit la nature du terrain rencontré. Leur cahier des charges prévoyait une durée de vie de 90 jours martiens (le jour martien est environ 40 minutes plus long que le jour terrestre). Spirit a cessé d'émettre le 22 mars 2010, soit 2210 jours martiens après son arrivée sur la planète. Début 2017, Opportunity est toujours en activité et il a parcouru plus de 44 km sur Mars.

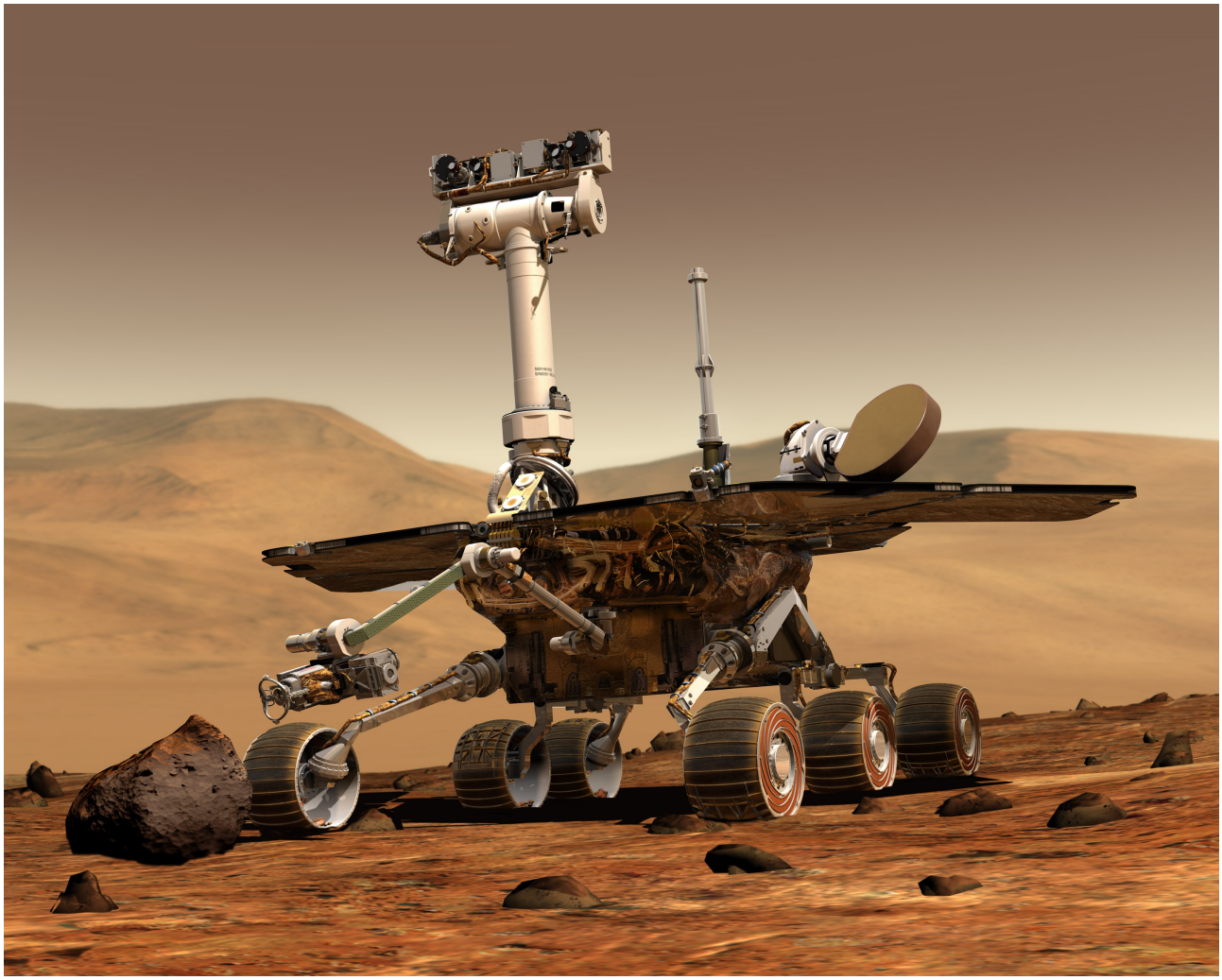

Figure 1 Vue d'artiste d'un robot géologue de la mission *Mars Exploration Rovers* (NASA/JPL – Caltech/Cornell)

Chaque robot est équipé de plusieurs instruments d'analyse (caméra, microscope, spectromètres) et d'un bras qui permet d'amener les instruments au plus près des roches et sols dignes d'intérêt. À partir de photographies de la surface de la planète, prises à plusieurs longueurs d'ondes par différents satellites et par le robot lui-même, les scientifiques de la NASA définissent une liste d'emplacements (*points d'intérêt* ou PI) où effectuer des analyses. Cette liste est transmise au robot qui doit se rendre à chaque emplacement indiqué et y effectuer les analyses prévues. Chaque robot est capable d'effectuer un certain nombre de types d'analyses géologiques correspondant aux différents instruments dont il dispose. Une fois tous les points d'intérêts visités et les résultats des analyses

transmis à la Terre, le robot reçoit une nouvelle liste de points d'intérêts et démarre une nouvelle *exploration*. Compte-tenu des contraintes de transmission entre la Terre et les robots (latence, périodes d'ombre, faible débit, etc.) il est prévu que les robots travaillent en autonomie pour planifier le parcours de chaque exploration. Ainsi, une fois la liste des points d'intérêt reçue, le robot analyse le terrain afin de détecter d'éventuels obstacles et détermine le meilleur chemin lui permettant de visiter l'ensemble de ces points en dépensant le moins d'énergie possible.

Après s'être intéressé à l'enregistrement des explorations, des points d'intérêts correspondants et des analyses à y mener, ce sujet aborde trois algorithmes qui peuvent être utilisés par le robot pour déterminer le meilleur parcours lui permettant de visiter chaque point d'intérêt une et une seule fois. Pour cela nous faisons quelques hypothèses simplificatrices.

— La zone d'exploration est dépourvue d'obstacle : le robot peut rejoindre directement en ligne droite n'importe quel point d'intérêt.
— Le sol est horizontal et de nature constante : l'énergie utilisé pour se déplacer entre deux points ne dépend que de leur distance, autrement dit le meilleur chemin est le plus court.
— La courbure de la planète est négligée compte tenu de la dimension réduite de la zone d'exploration : nous travaillerons en géométrie euclidienne et les points d'intérêts seront repérés par leurs coordonnées cartésiennes à l'intérieur de la zone d'exploration.

Les seuls langages de programmation autorisés dans cette épreuve sont Python et SQL. Toutes les questions sont indépendantes. Néanmoins, il est possible de faire appel à des fonctions ou procédures créées dans d'autres questions. Dans tout le sujet on suppose que les bibliothèques `math`, `numpy` et `random` ont été importées grâce aux instructions

```
import math
import numpy as np
import random
```

Si les candidats font appel à des fonctions d'autres bibliothèques ils doivent préciser les instructions d'importation correspondantes.

Ce sujet utilise la syntaxe des annotations pour préciser le types des arguments et du résultat des fonctions à écrire. Ainsi

```
def maFonction(n:int, x:float, d:str) -> np.ndarray:
```

signifie que la fonction `maFonction` prend trois arguments, le premier est un entier, le deuxième un nombre à virgule flottante et le troisième une chaine de caractères et qu'elle renvoie un tableau numpy.

Une liste de fonctions utiles est donnée à la fin du sujet.

II Planification d'une exploration : première approche

Avant de démarrer une nouvelle exploration, le robot doit déterminer un chemin qui lui permet de passer par tous les points d'intérêts une et une seule fois. L'enjeu de l'opération est de trouver le chemin le plus court possible afin de limiter la dépense d'énergie et de limiter l'usure du robot.

Chaque point d'intérêt sera repéré par un entier positif ou nul correspondant à son indice dans le tableau des points d'intérêt. Un chemin d'exploration sera représenté par un objet de type `list` donnant les indices des points d'intérêt dans l'ordre de leur parcours.

II.A – Quelques fonctions utilitaires

II.A.1) Longueur d'un chemin

Écrire une fonction d'entête

```
def longueur_chemin(chemin:list, d:np.ndarray) -> float:
```

qui prend en paramètre un chemin à parcourir et la matrice des distances entre points d'intérêt (telle que renvoyée par la fonction `calculer_distances`) et renvoie la distance que doit effectuer le robot pour suivre ce chemin en partant de sa position courante (correspondant à la dernière ligne/colonne du tableau `d`) et en visitant tous les points d'intérêt dans l'ordre indiqué.

II.A.2) Normalisation d'un chemin

Écrire une fonction d'entête

```
def normaliser_chemin(chemin:list, n:int) -> list:
```

qui prend en paramètre une liste d'entiers et renvoie une liste correspondant à un chemin valide, c'est-à-dire contenant une seule fois tous les entiers entre 0 et `n` (exclu). Pour cela cette fonction commence par supprimer les éventuels doublons (en ne conservant que la première occurrence) et les valeurs supérieures ou égales à `n`, sans modifier l'ordre relatif des éléments conservés, puis ajoute à la fin les éventuels éléments manquants en ordre croissant de numéros.

II.B – Force brute

Pour rechercher le plus court chemin, on peut imaginer de considérer tous les chemins possibles et de calculer leur longueur. On obtiendra ainsi à coup sûr le chemin le plus court.

II.B.1) Déterminer en fonction de n, nombre de points à visiter, le nombre de chemins possibles passant exactement une fois par chacun des points.

II.B.2) Cet algorithme est-il utilisable pour une zone d'exploration contenant 20 points d'intérêts ? Justifier.

II.C – Algorithme du plus proche voisin

Une idée simple pour obtenir un algorithme utilisable est de construire un chemin en choisissant systématiquement le point, non encore visité, le plus proche de la position courante.

II.C.1) Écrire une fonction d'entête

```
def plus_proche_voisin(d:np.ndarray) -> list:
```

qui prend en paramètre le tableau des distances résultat de la fonction `calculer_distances` (question I.A.2) et fournit un chemin d'exploration en appliquant l'algorithme du plus proche voisin.

II.C.2) Quelle est la complexité temporelle de l'algorithme du plus proche voisin en considérant que cet algorithme est constitué des deux fonctions `calculer_distances` et `plus_proche_voisin` ?

II.C.3) En considérant les trois points de coordonnées $(0,0)$, $(0,3000)$, $(0,7000)$ et en choisissant un point de départ adéquat pour le robot, montrer que l'algorithme du plus proche voisin ne fournit pas nécessairement le plus court chemin.

Dans la pratique, on constate que, dès que le nombre de points d'intérêt devient important, l'algorithme du plus proche voisin fournit un chemin qui peut être 50% plus long que le plus court chemin.

III Deuxième approche : algorithme génétique

Les algorithmes génétiques s'inspirent de la théorie de l'évolution en simulant l'évolution d'une population. Ils font intervenir cinq traitements.

1. Initialisation
   Il s'agit de créer une population d'origine composée de m individus (ici des chemins pour l'exploration à planifier). Généralement la population de départ est produite aléatoirement.
2. Évaluation
   Cette étape consiste à attribuer à chaque individu de la population courante une note correspondant à sa capacité à répondre au problème posé. Ici la note sera simplement la longueur du chemin.
3. Sélection
   Une fois tous les individus évalués, l'algorithme ne conserve que les « meilleurs » individus. Plusieurs méthodes de sélection sont possibles : choix aléatoire, ceux qui ont obtenu la meilleure note, élimination par tournoi, etc.
4. Croisement
   Les individus sélectionnés sont croisés deux à deux pour produire de nouveaux individus et donc une nouvelle population. La fonction de croisement (ou reproduction) dépend de la nature des individus.
5. Mutation
   Une proportion d'individus est choisie (généralement aléatoirement) pour subir une mutation, c'est-à-dire une transformation aléatoire. Cette étape permet d'éviter à l'algorithme de rester bloqué sur un optimum local.

En répétant les étapes de sélection, croisement et mutation, l'algorithme fait ainsi évoluer la population, jusqu'à trouver un individu qui réponde au problème initial. Cependant dans les cas pratiques d'utilisation des algorithmes génétiques, il n'est pas possible de savoir simplement si le problème est résolu (le plus court chemin figure-t-il dans ma population ?). On utilise donc des conditions d'arrêt heuristiques basées sur un critère arbitraire.

Le but de cette partie est de construire un algorithme génétique pour rechercher un meilleur chemin d'exploration que celui obtenu par l'algorithme du plus proche voisin.

III.A – Initialisation et évaluation

Une population est représentée par une liste d'individus, chaque individu étant représenté par un couple *(longueur, chemin)* dans lequel

- *chemin* désigne un chemin représenté comme précédemment par une liste d'entiers correspondant aux indices des points d'intérêt dans le tableau des distances produit par la fonction `calculer_distances` ;
- *longueur* est un entier correspondant à la longueur du chemin, en tenant compte de la position de départ du robot.

Écrire une fonction d'entête

```
def créer_population(m:int, d:np.ndarray) -> list:
```

qui crée une population de m individus aléatoires. Cette fonction prend en paramètre le nombre d'individus à engendrer et le tableau des distances entre points d'intérêt (et la position courante du robot) tel que produit par la fonction `calculer_distances`. Elle renvoie une liste d'individus, c'est-à-dire de couples *(longueur, chemin)*.

III.B – Sélection

Écrire une fonction d'entête

```
def réduire(p:list) -> None:
```

qui réduit une population de moitié en ne conservant que les individus correspondant aux chemins les plus courts. On rappelle que `p` est une liste de couples *(longueur, chemin)*. La fonction `réduire` ne renvoie pas de résultat mais modifie la liste passée en paramètre.

III.C – Mutation

III.C.1) Écrire une fonction d'entête

```
def muter_chemin(c:list) -> None:
```

qui prend en paramètre un chemin et le transforme en inversant aléatoirement deux de ses éléments.

III.C.2) Écrire une fonction d'entête

```
def muter_population(p:list, proba:float, d:np.ndarray) -> None:
```

qui prend en paramètre une population dont elle fait muter un certain nombre d'individus. Le paramètre `proba` (compris entre 0 et 1) désigne la probabilité de mutation d'un individu. Le paramètre `d` est la matrice des distances entre points d'intérêt.

III.D – Croisement

III.D.1) Écrire une fonction d'entête

```
def croiser(c1:list, c2:list) -> list:
```

qui crée un nouveau chemin à partir de deux chemins passés en paramètre. Ce nouveau chemin sera produit en prenant la première moitié du premier chemin suivi de la deuxième moitié du deuxième puis en « normalisant » le chemin ainsi obtenu.

III.D.2) Écrire une fonction d'entête

```
def nouvelle_génération(p:list, d:np.ndarray) -> None:
```

qui fait grossir une population en croisant ses membres pour en doubler l'effectif. Pour cela, la fonction fait se reproduire tous les couples d'individus qui se suivent dans la population `(p[i], p[i+1])` et `(p[m-1], p[0])` de façon à produire m nouveaux individus qui s'ajoutent aux m individus de la population de départ.

III.E – Algorithme complet

III.E.1) Écrire une fonction d'entête

```
def algo_génétique(PI:np.ndarray, m:int, proba:float, g:int) -> float, list:
```

qui prend en paramètre un tableau de points d'intérêts (figure 2), la taille `m` de la population, la probabilité de mutation `proba` et le nombre de générations `g`. Cette fonction implante un algorithme génétique à l'aide des différentes fonctions écrites jusqu'à présent et renvoie la longueur du plus court chemin d'exploration et le chemin lui-même obtenus au bout de `g` générations.

III.E.2) Est-il possible avec l'implantation réalisée, qu'une itération de l'algorithme dégrade le résultat : le meilleur chemin obtenu à la génération $n+1$ est plus long que celui de la génération n ?

Dans l'affirmative, comment modifier le programme pour que cette situation ne puisse plus arriver ?

III.E.3) Quelles autres conditions d'arrêt peut-on imaginer ? Établir un comparatif présentant les avantages et inconvénients de chaque condition d'arrêt envisagée.

Opérations et fonctions Python disponibles

Fonctions

- `range(n)` renvoie la séquence des `n` premiers entiers ($0 \to n-1$)
- `list(range(n))` renvoie une liste contenant les `n` premiers entiers dans l'ordre croissant : `list(range(5))` $\to$ `[0, 1, 2, 3, 4]`
- `random.randrange(a, b)` renvoie un entier aléatoire compris entre `a` et `b-1` inclus (`a` et `b` entiers)
- `random.random()` renvoie un nombre flottant tiré aléatoirement dans $[0, 1[$ suivant une distribution uniforme
- `random.shuffle(u)` permute aléatoirement les éléments de la liste `u` (modifie `u`)
- `random.sample(u, n)` renvoie une liste de `n` éléments distincts de la liste `u` choisis aléatoirement, si `n > len(u)`, déclenche l'exception `ValueError`
- `math.sqrt(x)` calcule la racine carrée du nombre x
- `math.ceil(x)` renvoie le plus petit entier supérieur ou égal à `x`

- `math.floor(x)` renvoie le plus grand entier inférieur ou égal à `x`
- `sorted(u)` renvoie une nouvelle liste contenant les éléments de la liste `u` triés dans l'ordre « naturel » de ses éléments (si les éléments de `u` sont des listes ou des tuples, l'ordre utilisé est l'ordre lexicographique)

Opérations sur les listes

- `len(u)` donne le nombre d'éléments de la liste `u` :
  `len([1, 2, 3])` → 3 ; `len([[1,2], [3,4]])` → 2
- `u + v` construit une liste constituée de la concaténation des listes `u` et `v` :
  `[1, 2] + [3, 4, 5]` → `[1, 2, 3, 4, 5]`
- `n * u` construit une liste constitué de la liste `u` concaténée `n` fois avec elle-même :
  `3 * [1, 2]` → `[1, 2, 1, 2, 1, 2]`
- `e in u` et `e not in u` déterminent si l'objet `e` figure dans la liste `u`, cette opération a une complexité temporelle en $O($`len(u)`$)$
  `2 in [1, 2, 3]` → `True` ; `2 not in [1, 2, 3]` → `False`
- `u.append(e)` ajoute l'élément `e` à la fin de la liste `u` (similaire à `u = u + [e]`)
- `del u[i]` supprime de la liste `u` son élément d'indice `i`
- `del u[i:j]` supprime de la liste `u` tous ses éléments dont les indices sont compris dans l'intervalle $[$`i`$,$ `j`$[$
- `u.remove(e)` supprime de la liste `u` le premier élément qui a pour valeur `e`, déclenche l'exception `ValueError` si `e` ne figure pas dans `u`, cette opération a une complexité temporelle en $O($`len(u)`$)$
- `u.insert(i, e)` insère l'élément `e` à la position d'indice `i` dans la liste `u` (en décalant les éléments suivants) ; si `i >= len(u)`, `e` est ajouté en fin de liste
- `u[i], u[j] = u[j], u[i]` permute les éléments d'indice `i` et `j` dans la liste `u`
- `u.sort()` trie la liste `u` en place, dans l'ordre « naturel » de ses éléments (si les éléments de `u` sont des listes ou des tuples, l'ordre utilisé est l'ordre lexicographique)

Opérations sur les tableaux (`np.ndarray`)

- `np.array(u)` crée un nouveau tableau contenant les éléments de la liste `u`. La taille et le type des éléments de ce tableau sont déduits du contenu de `u`
- `np.empty(n, dtype)`, `np.empty((n, m), dtype)` crée respectivement un vecteur à `n` éléments ou une matrice à `n` lignes et `m` colonnes dont les éléments, de valeurs indéterminées, sont de type `dtype` qui peut être un type standard (`bool`, `int`, `float`, ...) ou un type spécifique numpy (`np.int16`, `np.float32`, ...). Si le paramètres `dtype` n'est pas précisé, il prend la valeur `float` par défaut
- `np.zeros(n, dtype)`, `np.zeros((n, m), dtype)` fonctionne comme `np.empty` en initialisant chaque élément à la valeur zéro pour les types numériques ou `False` pour les types booléens
- `a.ndim` nombre de dimensions du tableau `a` (1 pour un vecteur, 2 pour une matrice, etc.)
- `a.shape` tuple donnant la taille du tableau `a` pour chacune de ses dimensions
- `len(a)` taille du tableau `a` dans sa première dimension (nombre d'éléments d'un vecteur, nombre de lignes d'une matrice, etc.) équivalent à `a.shape[0]`
- `a.size` nombre total d'éléments du tableau `a`
- `a.flat` itérateur sur tous les éléments du tableau `a`
- `a.min()`, `a.max()` renvoie la valeur du plus petit (respectivement plus grand) élément du tableau `a` ; ces opérations ont une complexité temporelle en $O($`a.size`$)$
- `b in a` détermine si `b` est un élément du tableau `a` ; si `b` est un scalaire, vérifie si `b` est un élément de `a` ; si `b` est un vecteur ou une liste et `a` une matrice, détermine si `b` est une ligne de `a`
- `np.concatenate((a1, a2))` construit un nouveau tableau en concaténant deux tableaux ; `a1` et `a2` doivent avoir le même nombre de dimensions et la même taille à l'exception de leur taille dans la première dimension (deux matrices doivent avoir le même nombre de colonnes pour pouvoir être concaténées)

• • • FIN • • •

CONCOURS CENTRALE•SUPÉLEC

Informatique

MP, PC, PSI, TSI

3 heures Calculatrices autorisées

2016

Prévention des collisions aériennes

Ce problème s'intéresse à différents aspects relatifs à la sécurité aérienne et plus précisément au risque de collision entre deux appareils. Dans la première partie nous abordons l'enregistrement des plans de vol des différentes compagnies aériennes. La deuxième partie explique la manière d'attribuer à chaque vol un couloir aérien répondant au mieux aux souhaits des compagnies aériennes. Enfin, la troisième partie s'intéresse à la procédure de détection d'un risque de collision entre deux avions se croisant à des altitudes proches.

Une liste d'opérations et de fonctions qui peuvent être utilisées dans les fonctions Python figure en fin d'énoncé. Les candidats peuvent coder toute fonction complémentaire qui leur semble utile. Ils devront dans ce cas préciser le rôle de cette fonction, la signification de ses paramètres et la nature de la valeur renvoyée.

II Allocation des niveaux de vol

Lors du dépôt d'un plan de vol, la compagnie aérienne doit préciser à quel niveau de vol elle souhaite faire évoluer son avion lors de la phase de croisière. Ce niveau de vol souhaité, le RFL pour *requested flight level*, correspond le plus souvent à l'altitude à laquelle la consommation de carburant sera minimale. Cette altitude dépend du type d'avion, de sa charge, de la distance à parcourir, des conditions météorologiques, etc.

Cependant, du fait des similitudes entre les différents avions qui équipent les compagnies aériennes, certains niveaux de vols sont très demandés ce qui engendre des conflits potentiels, deux avions risquant de se croiser à des altitudes proches. Les contrôleurs aériens de la région concernée par un conflit doivent alors gérer le croisement de ces deux avions.

Pour alléger le travail des contrôleurs et diminuer les risques, le système de régulation s'autorise à faire voler un avion à un niveau différent de son RFL. Cependant, cela engendre généralement une augmentation de la consommation de carburant. C'est pourquoi on limite le choix aux niveaux immédiatement supérieur et inférieur au RFL.

Ce problème de régulation est modélisé par un graphe dans lequel chaque vol est représenté par trois sommets. Le sommet 0 correspond à l'attribution du RFL, le sommet + au niveau supérieur et le sommet − au niveau inférieur. Chaque conflit potentiel entre deux vols sera représenté par une arête reliant les deux sommets concernés. Le cout d'un conflit potentiel (plus ou moins important en fonction de sa durée, de la distance minimale entre les avions, etc.) sera représenté par une valuation sur l'arête correspondante.

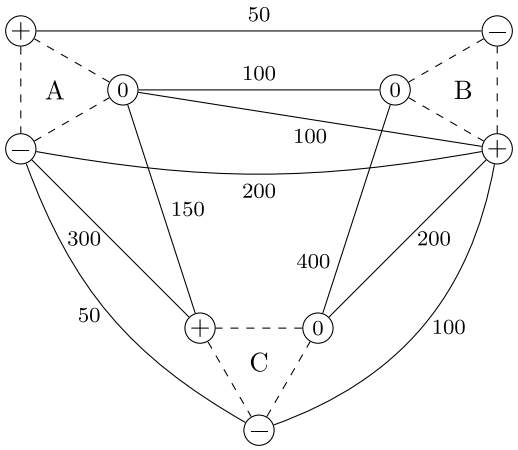

Figure 3 Exemple de conflits potentiels entre trois vols

Dans l'exemple de la figure 3, faire voler les trois avions à leur RFL engendre un cout de régulation entre A et B de 100 et un cout de régulation entre B et C de 400, soit un cout total de la régulation de 500 (il n'y a pas de

conflit entre A et C). Faire voler l'avion A à son RFL et les avions B et C au-dessus de leur RFL engendre un conflit potentiel de cout 100 entre A et B et 150 entre A et C, soit un cout total de 250 (il n'y a plus de conflit entre B et C).

On peut observer que cet exemple possède des solutions de cout nul, par exemple faire voler A et C à leur RFL et B au-dessous de son RFL. Mais en général le nombre d'avions en vol est tel que des conflits potentiels sont inévitables. Le but de la régulation est d'imposer des plans de vol qui réduisent le plus possible le cout total de la résolution des conflits.

II.A – Implantation du problème

Chaque vol étant représenté par trois sommets, le graphe des conflits associé à n vols $v_0, v_1, ..., v_{n-1}$ possède $3n$ sommets que nous numéroterons de 0 à $3n-1$. Nous conviendrons que pour $0 \leqslant k < n$:
- le sommet $3k$ représente le vol v_k à son RFL ;
- le sommet $3k+1$ représente le vol v_k au-dessus de son RFL ;
- le sommet $3k+2$ représente le vol v_k au-dessous de son RFL ;

Le cout de chaque conflit potentiel est stocké dans une liste de $3n$ listes de $3n$ entiers (tableau $3n \times 3n$) accessible grâce à la variable globale `conflit` : si i et j désignent deux sommets du graphe, alors `conflit[i][j]` est égal au cout du conflit potentiel (s'il existe) entre les plans de vol représentés par les sommets i et j. S'il n'y a pas de conflit entre ces deux sommets, `conflit[i][j]` vaut 0. On convient que `conflit[i][j]` vaut 0 si les sommets i et j correspondent au même vol (figure 4).

On notera que pour tout couple de sommets (i, j), `conflit[i][j]` et `conflit[j][i]`, représentent un seul et même conflit et donc `conflit[i][j] == conflit[j][i]`.

```
conflit = [ [   0,   0,   0, 100, 100,   0,   0, 150,   0],
            [   0,   0,   0,   0,   0,  50,   0,   0,   0],
            [   0,   0,   0,   0, 200,   0,   0, 300,  50],
            [ 100,   0,   0,   0,   0,   0, 400,   0,   0],
            [ 100,   0, 200,   0,   0,   0, 200,   0, 100],
            [   0,  50,   0,   0,   0,   0,   0,   0,   0],
            [   0,   0,   0, 400, 200,   0,   0,   0,   0],
            [ 150,   0, 300,   0,   0,   0,   0,   0,   0],
            [   0,   0,  50,   0, 100,   0,   0,   0,   0] ]
```

Figure 4 Tableau des couts des conflits associé au graphe représenté figure 3

II.A.1) Écrire en Python une fonction `nb_conflits()` sans paramètre qui renvoie le nombre de conflits potentiels, c'est-à-dire le nombre d'arêtes de valuation non nulle du graphe.

II.A.2) Exprimer en fonction de n la complexité de cette fonction.

II.B – Régulation

Pour un vol v_k on appelle *niveau relatif* l'entier r_k valant 0, 1 ou 2 tel que :
- $r_k = 0$ représente le vol v_k à son RFL ;
- $r_k = 1$ représente le vol v_k au-dessus de son RFL ;
- $r_k = 2$ représente le vol v_k au-dessous son RFL.

On appelle *régulation* la liste $(r_0, r_1, ..., r_{n-1})$. Par exemple, la régulation $(0, 0, ..., 0)$ représente la situation dans laquelle chaque avion se voit attribuer son RFL. Une régulation sera implantée en Python par une liste d'entiers.

Il pourra être utile d'observer que les sommets du graphe des conflits choisis par la régulation r portent les numéros $3k + r_k$ pour $0 \leqslant k < n$. On remarque également qu'au sommet s du graphe correspond le niveau relatif $r_k = s \bmod 3$ et le vol v_k tel que $k = \lfloor s/3 \rfloor$.

II.B.1) Écrire en Python une fonction `nb_vol_par_niveau_relatif(regulation)` qui prend en paramètre une régulation (liste de n entiers) et qui renvoie une liste de 3 entiers `[a, b, c]` dans laquelle a est le nombre de vols à leurs niveaux RFL, b le nombre de vols au-dessus de leurs niveaux RFL et c le nombre de vols au-dessous de leurs niveaux RFL.

II.B.2) Cout d'une régulation

On appelle *cout d'une régulation* la somme des couts des conflits potentiels que cette régulation engendre.

a) Écrire en Python une fonction `cout_regulation(regulation)` qui prend en paramètre une liste représentant une régulation et qui renvoie le cout de celle-ci.

b) Évaluer en fonction de n, la complexité de cette fonction.

c) Déduire de la question a) une fonction `cout_RFL()` qui renvoie le cout de la régulation pour laquelle chaque avion vole à son RFL.

II.B.3) Combien existe-t-il de régulations possibles pour n vols ?

Est-il envisageable de calculer les couts de toutes les régulations possibles pour trouver celle de cout minimal ?

II.C – L'algorithme Minimal

On définit le *cout d'un sommet* comme la somme des couts des conflits potentiels dans lesquels ce sommet intervient. Par exemple, le cout du sommet correspondant au niveau RFL de l'avion A dans le graphe de la figure 3 est égal à $100 + 100 + 150 = 350$.

L'algorithme *Minimal* consiste à sélectionner le sommet du graphe de cout minimal ; une fois ce dernier trouvé, les deux autres niveaux possibles de ce vol sont supprimés du graphe et on recommence avec ce nouveau graphe jusqu'à avoir attribué un niveau à chaque vol.

Dans la pratique, plutôt que de supprimer effectivement des sommets du graphe, on utilise une liste `etat_sommet` de $3n$ entiers tels que :
- `etat_sommet[s]` vaut 0 lorsque le sommet s a été supprimé du graphe ;
- `etat_sommet[s]` vaut 1 lorsque le sommet s a été choisi dans la régulation ;
- `etat_sommet[s]` vaut 2 dans les autres cas.

II.C.1)

a) Écrire en Python une fonction `cout_du_sommet(s, etat_sommet)` qui prend en paramètres un numéro de sommet s (n'ayant pas été supprimé) ainsi que la liste `etat_sommet` et qui renvoie le cout du sommet s dans le graphe défini par la variable globale `conflit` et le paramètre `etat_sommet`.

b) Exprimer en fonction de n la complexité de la fonction `cout_du_sommet`.

II.C.2)

a) Écrire en Python une fonction `sommet_de_cout_min(etat_sommet)` qui, parmi les sommets qui n'ont pas encore été choisis ou supprimés, renvoie le numéro du sommet de cout minimal.

b) Exprimer en fonction de n la complexité de la fonction `sommet_de_cout_min`.

II.C.3)

a) En déduire une fonction `minimal()` qui renvoie la régulation résultant de l'application de l'algorithme Minimal.

b) Quelle est sa complexité ? Commenter.

II.D – Recuit simulé

L'algorithme de *recuit simulé* part d'une régulation initiale quelconque (par exemple la régulation pour laquelle chacun des avions vole à son RFL) et d'une valeur positive T choisie empiriquement. Il réalise un nombre fini d'étapes se déroulant ainsi :
- un vol v_k est tiré au hasard ;
- on modifie r_k en tirant au hasard parmi les deux autres valeurs possibles ;
  - si cette modification diminue le cout de la régulation, cette modification est conservée ;
  - sinon, cette modification n'est conservée qu'avec une probabilité $p = \exp(-\Delta c/T)$, où Δc est l'augmentation de cout liée à la modification de la régulation ;
- le paramètre T est diminué d'une certaine quantité.

Écrire en Python une fonction `recuit(regulation)` qui modifie la liste `regulation` passée en paramètre en appliquant l'algorithme du recuit simulé. On fera débuter l'algorithme avec la valeur $T = 1000$ et à chaque étape la valeur de T sera diminuée de 1%. L'algorithme se terminera lorsque $T < 1$.

Remarque. Dans la pratique, l'algorithme de recuit simulé est appliqué plusieurs fois de suite en partant à chaque fois de la régulation obtenue à l'étape précédente, jusqu'à ne plus trouver d'amélioration notable.