Graphes : colorations et avions

INFORMATIQUE COMMUNE - Devoir n° 3 - Olivier Reynet

Consignes :

1. utiliser une copie différente pour chaque partie du sujet,
2. écrire son nom sur chaque copie,
3. écrire de manière lisible et intelligible,
4. préparer une réponse au brouillon avant de la reporter sur la feuille.

R Les parties A,B,C et D sont indépendantes. Il est **fortement** conseillé de les aborder dans l'ordre.

A Coloration de graphe

■ **Définition 1 — Coloration valide.** Une coloration d'un graphe est valide lorsque deux sommets adjacents n'ont jamais la même couleur.

■ **Définition 2 — Nombre chromatique.** Le nombre chromatique d'un graphe G est le plus petit nombre de couleurs nécessaires pour obtenir une coloration valide de ce graphe. On le note généralement $\chi(G)$.

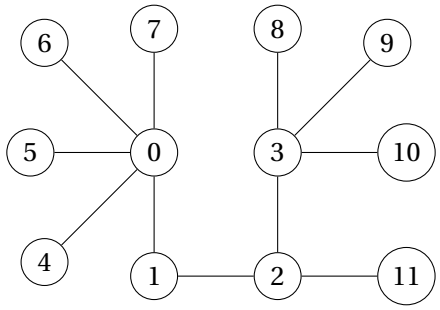


FIGURE 1 – Graphe `g`

On se donne un graphe `g` (cf. figure 1) ainsi qu'une liste de couleur utilisables pour colorer une graphe.

```
colors = ["pink", "turquoise", "yellow", "blue","magenta", "orange", "lime"]
```

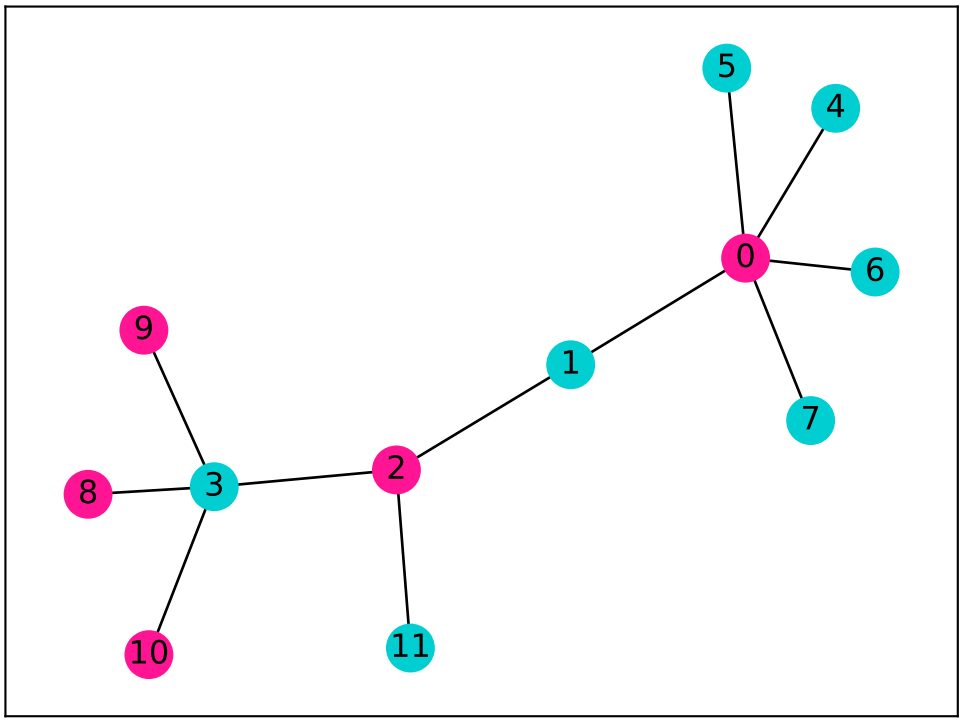
A1. En utilisant les liste Python, représenter par une liste d'adjacence le graphe `g` de la figure 1.

Solution :

```
g = [[1, 4, 5, 6, 7], [0, 2], [1, 3, 11], [2, 8, 9, 10],
     [0], [0], [0], [0],
     [3], [3], [3],
     [2]
     ]
```

A2. Quel est le nombre chromatique du graphe 1 ? Trouver ce nombre par le dessin.

Solution : On colorie à la main le graphe. on voit que le graphe est planaire, donc on sait qu'il existe au moins une solution avec quatre couleurs. Peut-on cependant colorer avec trois ou même deux couleurs ? Oui (cf. ci-dessous), donc $\chi_g = 2$.



L'algorithme de Welsh-Powell (cf. algorithme 1) est un algorithme de coloration de graphe glouton. Cependant, plutôt que de parcourir en largeur le graphe pour le colorer, cet algorithme traite les sommets dans l'ordre décroissant de leur degré. **Pour chaque sommet, dans l'ordre décroissant des degrés,** il choisit une nouvelle couleur et l'affecte aux autres sommets si cela est possible, c'est-à-dire lorsqu'ils ne sont ni déjà colorés et ni directement adjacents entre eux.

R L'idée de cet algorithme part de la constatation suivante : dans l'algorithme glouton, une nouvelle couleur est requise uniquement lorsqu'un voisin plus densément connecté à d'autres déjà colorés est découvert. En traitant en premier les sommets les plus connectés aux autres sommets, on espère minimiser le nombre de couleurs à utiliser.

Algorithme 1 Welsh-Powell

```
1 : Fonction WP(g, couleurs)
2 :     sommets ← COUNTING_SORT(SOM_DEG(g))
3 :     cmap ← un dictionnaire vide
4 :     tant que sommets n'est pas vide répéter
5 :         s ← retirer le premier sommet de la liste sommets
6 :         c ← retirer la première couleur de la liste couleurs
7 :         cmap[s] ← c
8 :         tant que certains sommets peuvent être colorés en c répéter
9 :             v ← un sommet qu'il est possible de colorer en c
10 :            cmap[v] ← c
11 :            supprimer v de la liste sommets
12 :    renvoyer cmap
```

A3. Écrire une fonction de prototype `som_deg(g)` dont le paramètre est un graphe sous la forme d'une liste d'adjacence. Cette fonction renvoie la liste des tuples (sommet, degré du sommet). Par exemple, `[[1], [0, 2], [1, 3], [2]]` renvoie `[(0, 1), (1, 2), (2, 2), (3, 1)]`.

Solution :

```
def som_deg(g):
    n = len(g)
    sd = []
    for i in range(n):
        sd.append((i, len(g[i])))
    return sd
    # return [(i,len(voisins)) for i, voisins in enumerate(g)]
```

A4. Écrire une fonction de prototype `deg_max(sd)` dont le paramètre est la liste issue de la fonction `som_deg` et qui renvoie le degré maximal du graphe. Si la liste passée en paramètre est vide, la fonction renvoie `None`.

Solution :

```
def deg_max(sd):
    if len(sd) > 0:
        dmax = sd[0][1]
        for s,deg in sd:
            if deg > dmax:
                dmax = deg
        return dmax
    return None
```

A5. Écrire une fonction de signature `c_color(g, s, remaining)` qui renvoie la liste des sommets de `g` que l'on peut colorer de la même couleur que `s` parmi les sommets de la liste `remaining`.

Solution :

```
def c_color_others(g, s, remaining):
    not_colorable = [v for v in g[s]]
    to_color = []
    for v in remaining:
        if v not in not_colorable:
            to_color.append(v)
            for k in g[v]:  # garder la trace de ceux qu'on ne peut plus colorer ainsi
                if k not in not_colorable:
                    not_colorable.append(k)
    return to_color
```

On suppose qu'on dispose d'une fonction `counting_sort(sd)` qui renvoie la liste des tuples (sommet, degré du sommet) **triée** dans le sens décroissant des degrés en un temps linéaire en fonction de la taille de la liste.

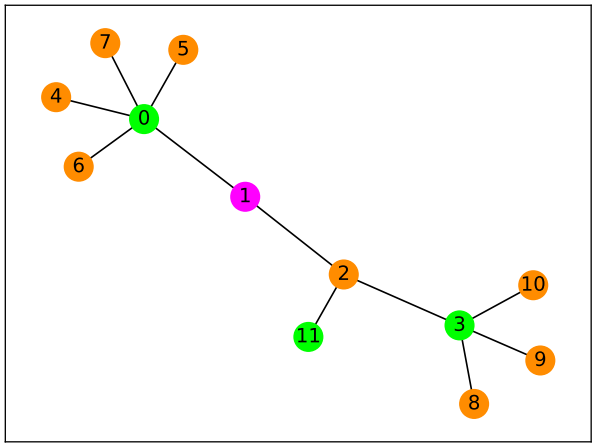
A6. Écrire une fonction de prototype `welsh_powell(g, colors)` qui implémente l'algorithme de Welsh-Powell. Cette fonction renvoie un dictionnaire `cmap` : chaque clef est le numéro d'un sommet et la valeur associée à cette clef est la couleur affectée au sommet. On pourra s'appuyer sur les fonctions précédentes et utiliser la fonction `lst.remove(v)` qui retire l'élément de valeur `v` de la liste `lst`.

Solution :

```
def welsh_powell(g, colors):
    sd = som_deg(g)
    sommets = counting_sort(sd)
    color_map = dict()
    while len(sommets) > 0:
        s = sommets.pop(0)  # choix glouton
        c = colors.pop()
        color_map[s] = c  # coloration
        to_color = c_color_others(g, s, sommets)
        for v in to_color:
            color_map[v] = c
            sommets.remove(v)
    return color_map
```

A7. Appliquer à la main l'algoritme de Welsh-Powell sur le graphe `g` de la figure 1. Que pouvez-vous en conclure ?

Solution : L'algorithme de Welsh-Powell n'est pas optimal. Sur cet exemple, on a besoin de trois couleurs alors que $\chi_g = 2$.



B Plus court chemin

On cherche à programmer l'algorithme de Dijkstra en Python mais sans utiliser la bibliothèque `queue` pour implémenter une file de priorité. Dans ce but, on opte pour l'utilisation des listes Python et la programmation d'une fonction d'insertion particulière.

B1. Écrire une fonction de signature `insert_int(L: list, e: int)->None` dont les paramètres sont une liste `L` d'entiers triée dans l'ordre croissant et `e` un entier à insérer dans la liste triée **afin que celle-ci reste triée**. Cette fonction modifie la liste passée en paramètre. **On utilisera obligatoirement une boucle `while` ainsi que la fonction `insert` dont l'usage est rappelé ci-dessous.**

> **lst.insert(i, x)** Insère un élément x à la position i dans la liste lst. Le premier argument est la position de l'élément avant lequel l'insertion doit s'effectuer. Donc lst.insert(0, x) insère l'élément en tête de la liste et lst.insert(len(lst), x) est équivalent à lst.append(x).

Solution :

```
def insert_int(L: list, e: int)->None:
    i = 0
    while i < len(L) and L[i] <= e:
        i += 1
    L.insert(i, e)
```

B2. Donner la complexité de la fonction `insert_int` lorsque l'insertion de l'élément se fait en tête de liste, en fin de liste ou dans un cas quelconque. On supposera que la complexité de la fonction `insert` est en $\mathcal{O}(n-i)$ si la position d'insertion est i et la longueur de la liste vaut n.

Solution : Dans un cas quelconque, la boucle `while` effectue i itérations et la fonction `insert` est de complexité $\mathcal{O}(n-i)$, puisqu'elle doit décaler les $n-i$ éléments restant sur la droite. Au final,

$$C(n) = i + n - i = \mathcal{O}(n)$$

La complexité est en $\mathcal{O}(n)$.

B3. Appliquer à la main l'algorithme de Dijkstra au graphe défini par la liste d'adjacence

```
gp = [[(1, 2), (2, 5), (4, 7), (5, 9)],
      [(0, 2), (2, 2)],
      [(0, 5), (1, 2), (3, 1)],
      [(2, 1), (4, 14), (5, 1)],
      [(0, 7), (3, 14)],
      [(0, 9), (3, 1)]]
```

On prendra soin de représenter le déroulement dans un tableau dont les colonnes sont les sommets du graphe et l'ensemble des sommets visités et les lignes les distances aux sommets à chaque itération.

Solution : `{0 : 0, 1 : 2, 2 : 4, 3 : 5, 4 : 7, 5 : 6}`

Δ	0	1	2	3	4	5
$\{0\}$	0	2	5	$+\infty$	7	9
$\{0,1\}$	0	2	4	$+\infty$	7	9
$\{0,1,2\}$	0	2	4	5	7	9
$\{0,1,2,3\}$	0	2	4	5	7	6
$\{0,1,2,3,5\}$	0	2	4	5	7	6
$\{0,1,2,3,5,4\}$	0	2	4	5	7	6

B4. Dans le cadre de l'algorithme de Dijsktra, la file de priorité utilisée contient des tuples `(distance, sommet)` et la priorité la plus grande est la distance la plus faible. Écrire une fonction de signature `insert_elem(Q: list, e: (int,int))->None` qui permet d'insérer un élément dans la file de priorité `Q`. Cette fonction est une simple adaptation de la fonction `insert_int`.

Solution :

```
def insert_elem(Q: list, e: (int,int))->None:
    i = 0
    while i < len(Q) and Q[i][0] <= e[0]:
        i += 1
    Q.insert(i, e)
```

B5. Dans notre version de Dijkstra, on décide de ne retourner que les distances les plus courtes. Pour stocker ces distances, qui évoluent au cours de l'algorithme, on utilise un dictionnaire Python. Écrire une fonction de signature `init_distances(G: list)-> dict` dont le paramètre est un graphe `G` donné sous la forme d'une liste d'adjacence. Cette fonction renvoie le dictionnaire dont les clefs sont les sommets du graphe `G`. Les valeurs associées aux clefs sont toutes initialisées à `math.inf`.

Solution :

```
import math

def init_distances(G: list) -> dict:
    d = {}
    for v in range(len(G)):
        d[v] = math.inf
    return d
```

B6. Écrire une fonction de signature `dijkstra(G: list, start: int)-> dict` dont les paramètres sont un graphe donné sous la forme d'une liste d'adjacence et un entier qui représente le sommet de départ à partir duquel on calcule les distances. Elle renvoie le dictionnaire des distances les plus courtes, par exemple, `{0 : 0, 1 : 12, 2 : 3, 3 : 14, 4 : 21, 5 : 42}`. On fera obligatoirement appel aux fonctions `init_distances` et `insert_elem` et on utilisera une file de priorité implémentée par une liste de tuples `(distance, sommet)`.

Solution :

```
def dijkstra(G: list, start: int) -> dict:  # simple and naive pq
    d = init_distances(G)
    d[start] = 0
    q = [(0, start)]
    visited = []
    while len(q) > 0:
        delta, s = q.pop(0)
        if s not in visited:
            for v, dv in G[s]:
                if v not in visited and d[v] > delta + dv:
                    d[v] = delta + dv
                    insert_elem(q,(d[v], v))
    return d
```

B7. Quelle est la complexité de la fonction `dijsktra` ainsi programmée dans le pire des cas ?

Solution : Soit n l'ordre du graphe.

- `init_distances` est de complexité $\mathcal{O}(n)$
- `q.pop(0)` est de complexité $\mathcal{O}(n)$ (transfert)
- `insert_elem` est de complexité $\mathcal{O}(n)$ (mise à jour de la file, enfiler)

Donc on peut écrire :

$$C(n) = n + (n \times n + m \times n) = \mathcal{O}(n(n+m))$$

Si le graphe est complet, le pire des cas, on a alors $m = n(n-1)/2$ et la complexité est en $\mathcal{O}(n^3)$.

C 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.

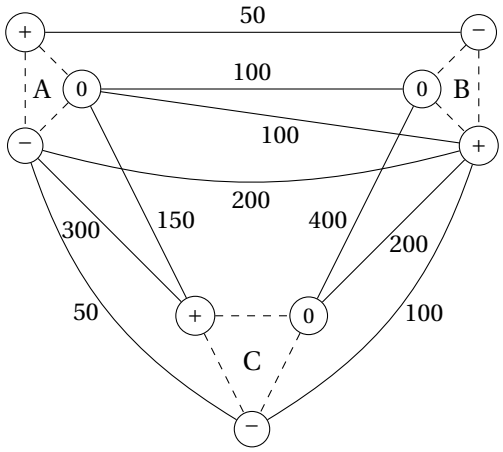


Figure 3 Exemple de conflits potentiels entre trois vols

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

C1. 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.

C2. 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 $(r_0, r_1, \ldots, r_{n-1})$. Par exemple, la régulation $(0, 0, \ldots, 0)$ 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 $k = \lfloor s/3 \rfloor$.

C3. 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.

Cout d'une régulation

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

C4. 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.

C5. Évaluer en fonction de n, la complexité de cette fonction.

C6. Déduire de la question *a)* une fonction `cout_RFL()` qui renvoie le cout de la régulation pour laquelle chaque avion vole à son RFL.

C7. Combien existe-t-il de régulations possibles pour n vols ?

C8. 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.

C9. 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`.

C10. Exprimer en fonction de n la complexité de la fonction `cout_du_sommet`.

C11. 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.

C12. Exprimer en fonction de n la complexité de la fonction `sommet_de_cout_min`.

C13. En déduire une fonction `minimal()` qui renvoie la régulation résultant de l'application de l'algorithme Minimal.

C14. Comment pourrait-on qualifier l'algorithme Minimal ?

C15. Quelle serait la complexité d'un algorithme résolvant ce problème par la force brute ?

C16. Quelle est la complexité de Minimal ?

II.D - Recuit simulé (question bonus)

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.

C17. (bonus points) 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 $T = 1\,000$ 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.

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

L'optimisation globale des niveaux de vol d'un système d'avions étudiée précédemment est complétée par une surveillance locale dans l'objectif de maintenir à tout instant une séparation suffisante avec tout autre avion. La règlementation actuelle impose aux avions de ligne d'être équipé d'un système embarqué d'évitement de collision en vol, ou TCAS pour *traffic collision avoidance system*.

Nous nous intéressons au fonctionnement du TCAS vu d'un avion particulier que nous appelons *avion propre*. Les avions qui volent à proximité de l'avion propre sont qualifiés d'*intrus*.

Le système TCAS surveille l'environnement autour de l'avion propre (typiquement dans un rayon d'une soixantaine de kilomètres) pour identifier les intrus et déterminer s'ils présentent un risque de collision. Pour cela, le TCAS évalue, pour chaque intrus, le point où sa distance avec l'avion propre sera minimale. Ce point est appelé CPA, pour *closest point of approach*. Le fonctionnement global (simplifié) du système TCAS est décrit par la fonction TCAS donnée figure 5.

Cette fonction maintient une liste des CPA des avions situés dans son périmètre de surveillance (variable `CPAs`). Cette liste est limitée à un nombre restreint d'intrus (`intrus_max`) dont l'instant du CPA est proche (dans moins de `suivi_max`) de façon à garantir la détection et le traitement d'un éventuel danger dans un temps suffisamment court.

```
def TCAS() :
    """Fonction principale du syst\'eme TCAS."""
    intrus_max = 30 # nombre maximum d'avions suivis
    suivi_max = 100 # d$\color{olive}\mathbf{\mathtt{\'e}}$lai maximum pour le
         CPA (en secondes)
    CPAs = [] # liste des CPA des avions suivis
    while (TCAS_actif()) :
        intrus = acquerir_intrus()
        enregistrer_CPA(intrus, CPAs, intrus_max, suivi_max)
        traiter_CPAs(CPAs)
```

Figure 5

La fonction `TCAS_actif` renvoie la position de l'interrupteur principal du système.

La fonction `acquerir_intrus` détermine la position et la vitesse d'un intrus particulier. À chaque appel, elle s'intéresse à un avion différent dans le périmètre de surveillance du TCAS. Lorsque tous les intrus ont été examinés, l'appel suivant revient au premier intrus qui est toujours dans le périmètre.

La fonction `enregistrer_CPA` intègre dans la liste CPA de nouvelles informations telles que fournies par la fonction `acquerir_intrus`.

Enfin la fonction `traiter_CPAs` examine les CPA des intrus les plus dangereux pour décider si l'un d'eux présente un risque de collision. Si c'est le cas, cette fonction détermine la manoeuvre à effectuer (monter ou descendre) en coordination avec le système TCAS de l'intrus concerné et génère une alarme et une consigne à destination du pilote.

Acquisition et stockage des données

Chaque avion est équipé d'un émetteur radio spécialisé, appelé *transpondeur*, qui fournit automatiquement, en réponse à l'interrogation d'une station au sol ou d'un autre avion, des informations sur l'avion dans lequel il est installé. La fonction `acquerir_intrus` utilise les données du système de navigation de l'avion propre, les données fournies par le transpondeur de l'intrus, le relèvement de son émission et les informations fournies par le système de contrôle aérien au sol.

D1. Les transpondeurs utilisent tous la même fréquence radio. Afin d'éviter la saturation de cette fréquence, en particulier dans les zones à fort trafic, chaque émission ne doit pas durer plus de 128 μs. Le débit binaire utilisé est de 10^6 bits par seconde ; chaque message commence par une marque de début de 6 bits et se termine par 4 bits de contrôle et une marque de fin de 6 bits.

$$\underbrace{d\,d\,d\,d\,d\,d}_{\text{début}}\underbrace{x\,x\,x\ldots x\,x\,x}_{\text{données}}\underbrace{e\,e\,e\,e}_{\text{contrôle}}\underbrace{f\,f\,f\,f\,f\,f}_{\text{fin}}$$

Déterminer le nombre maximum de bits de données dans une émission de transpondeur.

D2. Le système TCAS souhaite récupérer l'altitude et la vitesse ascensionnelle de chaque intrus en interrogeant son transpondeur. La réponse du transpondeur contient systématiquement un numéro d'identification de l'avion sur 24 bits. Les autres informations sont des entiers codés en binaire qui peuvent varier dans les intervalles suivants :

- altitude de 2 000 à 66 000 pieds ;
- vitesse ascensionnelle de −5 000 à 5 000 pieds par minute.

La taille d'un message de transpondeur est-elle suffisante pour obtenir ces informations en une seule fois ?

D3. Après avoir récupéré les informations nécessaires, la fonction `acquerir_intrus` calcule la position et la vitesse de l'intrus par rapport à l'avion propre et renvoie une liste de huit nombres :

```
[id, x, y, z, vx, vy, vz, t0]
```

où

- `id` est le numéro d'identification de l'intrus ;
- `x`, `y`, `z` les coordonnées (en mètres) de l'intrus dans un repère orthonormé $\mathcal{R}_0$ lié à l'avion propre ;
- `vx`, `vy`, `vz` la vitesse (en mètres par seconde) de l'intrus dans ce même repère ;
- `t0` le moment de la mesure (en secondes depuis un instant de référence).

À des fins d'analyse une fois l'avion revenu au sol, la fonction acquerir_intrus conserve chaque résultat obtenu. Chaque nombre est stocké sur 4 octets. En supposant que cette fonction est appelé au maximum 100 fois par seconde, quel est le volume de mémoire nécessaire pour conserver les données de 100 heures de fonctionnement du TCAS ?

D4. Ce volume de stockage représente-t-il une contrainte technique forte ?