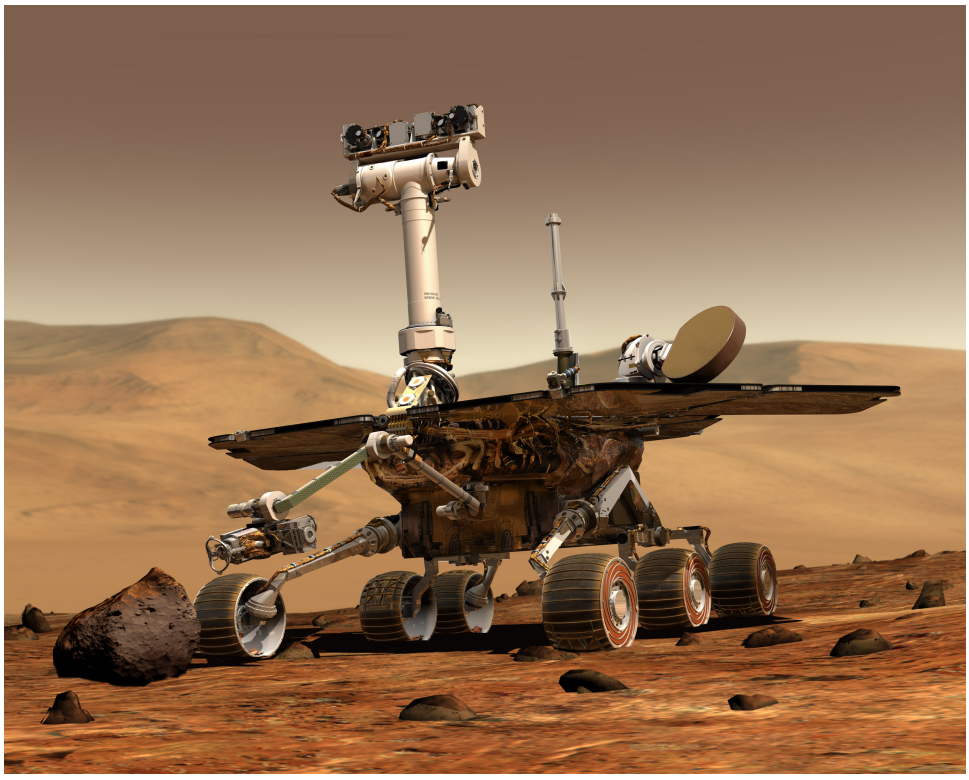
PSI* 2025/26 DS info 1

Exploration martienne

D'après le sujet Centrale 2017, avec une fin qui suit X2012MP/PC, le tout étant sans doute bien trop long. Déjà lire le pavé qui suit n'est pas très utile en fait... vous pouvez aller directement à la section objectifs.

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, 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 44km sur Mars.



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.

Objectifs

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. Dès lors, nous travaillerons en géométrie euclidienne, les points d'intérêts seront repérés par leurs coordonnées cartésiennes à l'intérieur de la zone d'exploration, et la distance entre $M_1(x_1, y_1)$ et $M_2(x_2, y_2)$ est $\delta = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

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` et `random` ont été importées grâce aux instructions

```
import math
import random as rd
```

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

`def maFonction(n:int, x:float, d:str) -> [[int]]:` 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 une liste de listes d'entiers.

Planification d'une exploration : mise en place

Exploration et points d'intérêt

Une *exploration* est un ensemble de points d'intérêts à l'intérieur d'une zone géographique limitée, une série d'analyses étant associée à chaque *point d'intérêt*. Chaque type d'analyse que le robot peut effectuer est codifié et référencé par un nombre entier. Un point d'intérêt est repéré par deux entiers, positifs ou nuls, correspondant à ses coordonnées cartésiennes en millimètres à l'intérieur de la zone d'exploration. L'ensemble des points d'intérêts d'une exploration qui en contient n est représenté par un objet de type n-liste de 2-listes, à éléments, l'élément d'indice `[i][0]` correspondant à l'abscisse du point d'intérêt i et l'élément d'indice `[i][1]` à son ordonnée.

Exemple d'exploration avec quatre points d'intérêt

analyse	x	y
0	345	635
1	1076	415
2	38	859
3	121	582

Calcul des distances

On suppose qu'on dispose de la fonction d'entête

```
def position_robot() -> tuple:
```

qui renvoie un couple donnant les coordonnées actuelles du robot dans le système de coordonnées de l'exploration à planifier. Ainsi l'instruction `x, y = position_robot()` permet de récupérer les coordonnées courantes du robot.

Q1. Tableau des distances
Afin de faciliter l'application des différents algorithmes de recherche de chemin, on souhaite construire un tableau des distances entre les différents points d'intérêt d'une exploration et entre ceux-ci et la position courante du robot au moment du calcul. Chaque point d'intérêt sera repéré par un entier positif ou nul correspondant à son indice dans le tableau des points d'intérêt (entre 0 et `n-1`). Écrire une fonction d'entête

```
def calculer_distances(PI:[[float]]) -> [[float]]:
```

qui prend en paramètre un tableau de n points d'intérêt tel que décrit précédemment et renvoie un tableau (liste de listes) de nombres flottants, de dimension $(n+1) \times (n+1)$, tel que l'élément d'indice `i,j` fournit la distance entre les points d'intérêt i et j, l'indice `n` désignant le point de départ du robot.

Quelques fonctions utilitaires

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.
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.

Q2. Longueur d'un chemin
Écrire une fonction d'entête

```
def longueur_chemin(chemin:list, d:[[float]]) -> 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.

Q3. Normalisation d'un chemin
On verra l'intérêt de cette normalisation ultérieurement. Écrire une fonction d'entête

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

qui prend en paramètre une liste d'entiers (considéré comme une sorte de chemin « invalide ») et renvoie une liste correspondant à un chemin valide, c'est-à-dire contenant une seule fois tous les entiers entre 0 et `n-1` (inclus). 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.
On demande que la fonction produite soit de complexité temporelle linéaire en n. En revanche on pourra introduire une liste de taille n pour mémoriser quels indices ont été employés ou non.

Planification d'une exploration : premières approches

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.

Q4. Déterminer en fonction de n, nombre de points à visiter, le nombre de chemins possibles passant exactement une fois par chacun des points.

Q5. Cet algorithme est-il utilisable pour une zone d'exploration contenant 20 points d'intérêts ? Justifier.

Algorithme du plus proche voisin

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

Q6. Écrire une fonction d'entête

```
def plus_proche_voisin(d:[[float]]) -> list:
```

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

Q7. 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` ?

Q8. 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.

Troisième 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.

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.

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.

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.

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.

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.

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.

On suppose qu'on dispose de la fonction d'entête

```
def chemin_aléatoire(n:int) -> [int]:
```

qui renvoie un chemin, sous forme d'une liste d'entiers de 0 à $n-1$, dans un ordre aléatoire. On considèrera que cette fonction s'exécute avec une complexité en $O(n)$.

Q9. Écrire une fonction d'entête `def créer_population(m:int, d:[[float]]) -> 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 (tuples) *(longueur, chemin)*.

Sélection

Q10. On considère encore que `p` est une liste de couples *(longueur, chemin)*. Écrire une fonction d'entête

```
def réduire(p:list) -> None:
```

qui renvoie une nouvelle liste formée par une population réduite de moitié en ne conservant que les individus correspondant aux chemins les plus courts. *On donne l'autorisation de modifier p, qui ne sera pas conservée. Un algorithme possible consiste à chercher le plus petit élément dans la liste, à le placer en tête par échange, puis à recommencer en cherchant le plus petit parmi les éléments restants et à le placer à la position d'indice 1,etc...*

Mutation

On redonne deux fonctions du module `random`, toutes les deux considérées comme des opérations élémentaires

— `rd.randrange(a, b)` renvoie un entier aléatoire compris entre `a` et `b-1` inclus (`a` et `b` entiers)

— `rd.random()` renvoie un nombre flottant tiré aléatoirement dans $[0, 1[$ suivant une distribution uniforme

Q11. 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 (distincts !).

Q12. Écrire une fonction d'entête `def muter_population(p:list, proba:float, d:[[float]]) -> 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.

Croisement

Q13. Dans cette question on pourra utiliser la concaténation de deux listes : si `l1` et `l2` sont deux listes, on les concatène en effectuant `l=l1+l2`, ce qui est une instruction de complexité égale à la somme des longueurs des deux listes.
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.

Q14. Écrire une fonction d'entête

```
def nouvelle_génération(p:list, d:[[float]]) -> 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.

Algorithme complet

Q15. Écrire une fonction d'entête

```
def algo_génétique(PI:[[float]], m:int, proba:float, g:int) -> float, list:
```

qui prend en paramètre un tableau de points d'intérêts, 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.

Q16. Estimer la complexité de l'algorithme en fonction de `m,g,n`. On pourra supposer $n \leq m$.

Amélioration de l'algorithme génétique

On souhaite améliorer la phase de sélection de l'algorithme. On se contente d'expliquer le principe de cette amélioration : il s'agit de déterminer l'élément médian d'une liste de taille n en temps linéaire en n. Pour simplifier on travaille ici sur un tableau `tab` d'entiers de longueur n.

Pivot et partition

Une fonction annexe nécessaire pour cet algorithme consiste à savoir séparer en deux un ensemble de valeurs. Soit un tableau tab et un réel appelé pivot `p = tab[i]`, il s'agit de réordonner les éléments du tableau en mettant en premier les éléments strictement plus petits que le pivot, puis le pivot p, et en dernier les autres éléments supérieurs ou égaux à p. Par exemple avec

```
tab=[3, 2, 5, 8, 1, 34, 21, 6, 9, 14, 8]
```

en prenant comme valeur de pivot 8 on obtiendra par exemple le tableau résultat suivant :
`[3, 2, 5, 1, 6, 8, 21, 34, 9, 8, 14]`

Notez que dans le résultat les nombres plus grands que le pivot 21, 34, 9, 8, 14 peuvent être dans n'importe quel ordre les uns par rapport aux autres.

Q17. Écrire une fonction `partition(tab, a, b, indicePivot)` qui prend en paramètre un tableau d'entiers, ainsi que des entiers tels que a≤`indicePivot` < b. Soit p = tab[indicePivot]. La fonction devra réordonner les éléments d'indice i, $a \leq i < b$, comme expliqué précédemment en prenant comme pivot le nombre p. La fonction devra également retourner le nouvel indice de la case ou se trouve la valeur p.

Application au calcul de médian

Remarquons que le $\lfloor \frac{n}{2} \rfloor$-ème élément dans l'ordre croissant d'un tableau de taille n est un élément médian du tableau considéré. Nous allons donc non pas programmer une méthode pour trouver le médian mais plus généralement pour trouver le k-ème élément d'un ensemble. Nous allons utiliser l'algorithme récursif suivant :

Recherche du k-ème élément du tableau tab[a :b].
- Si k = 1 et a + 1 = b alors renvoyer tab[a]
- Sinon, soit p = tab[a]. Partitionner le tableau tab[a :b] en utilisant le pivot p en mettant en premier les éléments plus petits que p. Soit i l'indice de p dans le tableau résultant.
- Si i - a + 1 > k chercher le k-ème élément dans tab[a :i] et renvoyer cet élément.
- Si i - a + 1 = k renvoyer le pivot.
- Si i - a + 1 < k chercher le (k - i + a - 1)-ème élément dans tab[i+1 :b] et renvoyer cet élément.

Q18. Écrire une fonction `elementK(tab, a, b, k)` qui réalise l'algorithme de sélection du k-ème élément dans le tableau tab[a :b] décrit précédemment et renvoie cet élément.

Q19. Supposons que dans l'algorithme précédent nous voulions rechercher le premier élément mais qu'à chaque étape le pivot choisi est le plus grand élément, quel est un ordre de grandeur du nombre d'opérations réalisées par votre fonction ?

L'algorithme précédent ne semble donc pas améliorer le calcul du médian. Le problème vient du fait que le pivot choisi peut être mauvais c'est-à-dire qu'à chaque étape un seul élément du tableau a été éliminé. En fait, si l'on peut choisir un pivot p dans tab[a :b] tel qu'il y ait au moins (b - a)/5 éléments plus petits et (b - a)/5 plus grands alors on peut montrer que l'algorithme précédent fonctionne optimalement en temps O(n).

Choix optimal de pivot

Pour choisir un tel élément dans tab[a :b], on réalise l'algorithme `choixPivot` suivant o'u chaque étape sera illustrée en utilisant le tableau donné en introduction en prenant a = 1 et b = 9.

- On découpe le (morceau de) tableau en paquets de 5 éléments plus éventuellement un paquet plus petit. On calcule l'élément médian de chaque paquet.

3 || $\boxed{2\ \boxed{5}\ 8\ 1\ 34}\ \boxed{21\ 6\ \boxed{9}}$ || 14 8

- S'il n'y a qu'un paquet on renvoie son médian. Sinon on place ces éléments médians au début du tableau.

3 || $\boxed{5}\ \boxed{9}$ 2 8 1 34 21 6 || 14 8

- On réalise `choixPivot` sur les médians précédents. Dans notre exemple on recommence donc les étapes précédentes en prenant a = 1 et b = 3.

3 || 5 9 || 2 8 1 34 21 6 14 8

Q20. On supposera l'existence d'une fonction `indice_median5(liste,a,b)` de complexité O(1) qui renvoie l'indice d'un médian pour une zone liste[a :b], à condition qu'elle soit de 5 éléments ou moins.

Écrire la fonction `choixPivot(tab, a, b)` qui réalise l'algorithme précédent et renvoie la valeur du pivot.

Annexe hors sujet de DS - Questions d'origine du concours centrale

---

Q+. 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 ?

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