Introduction Python: Examen 2026

Philipp Ahrendt

January 12, 2026

Contents

1 Problème du sac-à-dos

Supposons que l'on veut remplir un sac-à-dos avec une collection de n objets précieux. Chaque objet i a une valeur v_i et un poids p_i, et le sac-à-dos a une capacité S, limitant le poids que l'on peut mettre; on ne peut donc pas prendre tous les objets.

L'objectif est de choisir des objets $i_1,\ldots,i_k$ tels que leur valeur $\sum_{1 \leq j \leq k} v_{ij}$ soit maximale, tout en respectant la capacité du sac: $\sum_{1 \leq j \leq k} p_{ij} \leq S$. On s'intéresse ici uniquement au calcul de la valeur `valeur(L,S)`, qui doit valoir $\sum_{1 \leq j \leq k} v_{ik}$ pour une liste `L` contenant les paires (v_i,p_i), et une capacité `S`.

1.1 Question

Ce problème admet une décomposition de type "diviser pour régner": Considérons les sous-problèmes de calculer `valeur(L,S,n)` pour un nombre `n`

`<= len(L)`, c'est-à-dire maximiser la valeur et respecter `S` en n'utilisant que les `n` premiers objets.

- Pour `n = 0` cela vaut 0, aucun objet à mettre
- Pour `n = len(L)` c'est le problème initial.

On peut relier `valeur(L,S,n)` à `valeur(L,S,n-1)`;

- Si $p_{n-1} > S$, alors le n-ième objet ne peut pas être utilisé, il est déjà trop lourd tout seul. On est donc ramené à calculer `valeur(L,S,n-1)`
- Sinon il y a deux possibilités:
  1. Le meilleur choix est de le faire rentrer dans le sac; dans ce cas les `n-1` objets avant ne dépassent pas $S - p_{n-1}$. La valeur optimale est donc `valeur(L,S,n) = v_n + valeur(L,S-p_(n-1),n-1)`.
  2. Le meilleur choix est de le laisser dehors; le sac va être rempli qu'avec les objets avant `n`, et la valeur optimale est `valeur(L,S,n) = valeur(L,S,n-1)`

Comment décide-t-on laquelle des options 1 et 2 est la bonne?

En déduire une implémentation de `valeur(L,S)` sous forme "diviser pour régner".

2 Robot explorateur

Le but de cet exercice est d'implémenter un modèle d'un robot qui explore un environnement inconnu, où il cherche à collectionner des "rubis".

2.1 L'environnement

L'environnement est une carte 2D contenant des obstacles pour le robot, et des rubis que le robot doit collectionner. Elle contient également un "point relais", où le robot peut déposer les rubis (i.e les remettre à un contrôleur externe), ainsi que recharger sa batterie.

```
O O O O O O O O O : O O O O O O     Exemple d'une carte 16x12:
O O O O O : O O O : O O O O O O
O O O O O : O O O : O O $ O O O     O - Espace vide
O : : : : O O O O : O O O O O O     : - Obstacle
O : O O O O O O O : O O O O O O     $ - Rubin
O O O O O O + O O O O O O O O O     # - Point relais
O O O O O O O O O O O O O O O O     + - Position du robot
O : : : O O O O O O : : : : : :
O O O O O O O O O O O O O O O O
O O O O O O : O O O O O O O O O
O O O O O O O : : : : O O O O O
O O O O O O O O O O : : O O O O
# O O O O O O O O O O : O O O O
```

A chaque instant, il y a un seul rubis présent sur la carte. Lorsque le robot passe sur un rubis, il peut le collectionner. Le rubis disparaît alors de la carte, et un nouveau rubis apparaît quelque part sur la carte, de manière aléatoire.

2.1.1 Question

Écrire une fonction `coordonnees_libres(L,obs)`, qui prend en entrée une liste de listes `L` et une liste `obs`, et qui renvoie une liste `coords_libres` de tous les tuples `(i,j)` tels que `L[i][j]` n'est pas dans `obs`.

Exemple: Si `L` est la carte dans l'exemple ci-dessus, `coordonnees_libres(L,[':','#'])` contient tous les coordonnées de la liste sauf les obstacles et le point relais.

2.1.2 Question

Implémenter une classe `Env`:

- **Attributs**
  - Une carte `carte` avec des obstacles, un point relais, et un rubis (elle ne contiendra pas la position du robot!).
  - Une liste `coords_libres` des coordonnées sur la carte qui ne sont ni obstacles, ni point de relais.

  - Un tuple `pos_rubis`, indiquant la position du rubis.
  - Un tuple `pos_robot`, indiquant la position actuelle du robot.
  - Un entier `rubis_remis`, indiquant combien de rubis ont déjà été remis.

- **Initialisation**
  - Prend en entrée une carte `carte_init` avec des positions libres `'O'` , des obstacles `':'` et un point relais `'#'`.
  - Calcule `coords_libres`.
  - Initialise `pos_rubis` en choisissant aléatoirement une position de `coords_libres` (on pourra utiliser la fonction `element_aleatoire` dans le document).
  - Initialise `carte`, en plaçant un rubis `'$'` en `pos_rubis`.
  - Initialise `pos_robot` sur la position du point de relais `'#'`.
  - Initialise `rubis_remis` à 0.

- **Méthodes de classe**
  - `deplacer_robot(self, direction)`: Prend en entrée une des directions `'l','r','u','d'` (left, right, up, down), et déplace le robot d'une case dans cette direction en modifiant `pos_robot`. Si la position d'arrivée est un obstacle, ou en dehors de la carte, le robot ne bouge pas.
  - `replacer_rubis(self)`: enlève le rubis de sa position, et remet un rubis en une position aléatoire (non obstacle ou point relais) sur la carte.
  - `remettre_rubis(self, n)` augmente le nombre de rubis remis par `n`.

2.2 Le robot

Le robot a deux fonctions: explorer la carte, et ramener des rubis. Pour cela, il a trois actions qu'il peut effectuer.

- **Avancer** d'un pas dans une direction (gauche, droite, haut, bas).
- **Observer** la carte autour de sa position, il voit alors, depuis sa position, dans un carre de 3 cases dans chaque direction. Il ne peut pas voir à travers des obstacles.

```
Quelques exemples de comment les obstacles bloquent la
vision du robot. Le modèle est très simple: si l'obstacle
est droit devant dans une direction, il bloque la ligne
doite derrière (voir exemple 2), sinon il bloque toutes les
cases dans le rectangle derrière, vu depuis l'obstacle (voir
exemple 1). Les obstacles qui bloquent la vision sont
eux-mêmes reconnus par le robot comme obstacles.
```

```
Exemple 1:
* * O O : * *
* * O O O O O
* : O O O O O
O O O + O O O
O O O O : * *
O O O O * * *
O O O O * * *
```

```
Exemple 2:
O O O O O O O
O O O O O O O
O O O O O O O
* * : + O : *
O O O O O O O
O O O O O O O
O O O : O O O
```

```
Exemple 3:
O O O * O O O
O O O : O O O
O O O O O O O
* : O + O O O
* : O O O : *
* * O O : * *
* * O O * * *
```

```
Exemple 4:
Quand le robot se trouve au bord de la carte

* * * * * * *
* * * * * * *
* O O O O O O
* O O + O O O
* O O O O O O
* * : O O O O
* * * O O O O
```

- **Collectionner** un rubis. Si le robot est sur une case avec un rubis, il peut le prendre. A partir de ce moment, le robot porte le rubis.

Le robot est muni d'une batterie, qui s'épuise lorsqu'il se déplace ou observe. Le coût du déplacement augmente quand le robot porte des rubis. Le robot ne peut pas effectuer une action dont le coût est supérieur au niveau de sa batterie.

Quand le robot est au point relais, il a deux actions supplémentaires:

- **Remettre** les rubis qu'il porte.
- **Recharger** sa batterie

2.2.1 Question

Implémenter une classe `Robot`:

- **Attributs**
  - Un un objet `env` de type `Env`, l'environnement sur lequel se trouve le robot.
  - Un entier `bat_max` indiquant le niveau de charge maximal de la batterie.
  - Un entier `bat` indiquant le niveau de charge actuel de la batterie.
  - Un entier `rubis` indiquant le nombre de rubis portés par le robot.
- **Méthodes de classe**
  - `avance(self, direction)`: effectue le déplacement du robot dans une des directions `'l','r','u','d'`. Coûte 1 point de batterie, et 1 point de plus pour chaque rubis porté.
  - `observe(self)`: observe la carte autour du robot. Doit renvoyer une liste de listes représentant ce que voit le robot. Voir les exemples ci-dessus. Coûte 3 points de batterie.
  - `collectionne(self)`: si le robot est sur la case d'un rubis, il l'enlève de la carte et le rajoute aux rubis qu'il porte.
  - `remettre(self)`: si le robot est sur le point relais, il enlève ses rubis et les rajoute au nombre de rubis remis associé à la carte.

2.3 Un contrôleur externe

On souhaite maintenant implémenter quelques commandes plus complexes à partir de ces classes:

- **Revenir:** Le robot doit revenir le plus rapidement possible au point relais, remettre ses rubis et recharger.
- **Explorer:** Le robot doit explorer une nouvelle région de la carte.
- **Chercher rubis:** Le robot doit chercher le rubis dans la partie déjà explorée.
- **Récupérer rubis:** Le robot doit aller prendre un rubis déjà détecté.

Ces commandes doivent assurer que le robot n'est jamais "perdu". Si le niveau de batterie devient trop faible, il doit interrompre son action et revenir au point de relais pour recharger.

2.3.1 Question

Pour assurer que le robot puisse revenir au point relais, il faut connaître le chemin le plus court entre la position du robot et le point relais. Dans ce contexte il faut trouver le chemin le plus court *connu*, c'est-à-dire un chemin qui évite les obstacles et ne passe pas par des régions non explorées.

Écrire une fonction `chemin(carte, obs, depart, arrivee)`, qui prend en entrée une liste de listes `carte`, une liste `obs` de "symboles d'obstacles", et des tuples `depart` et `arrivee`, et qui renvoie le chemin le plus court entre `depart` et `arrivee` dans `carte`, en évitant les cases contenant un symbole de `obs`. Le chemin est renvoyé sous forme de liste de coordonnées.

Indication: On peut créer un graphe qui représente la situation, et ensuite utiliser les cours/exercices.

2.3.2 Question

Pour regrouper les données et méthodes dont on a besoin, on introduit une nouvelle classe.

Implémenter une classe `Controleur`:

- **Attributs**
  - Un objet `robot` de type `Robot`: le robot contrôlé.
  - Une liste de listes `carte_connue`, qui contient tout ce qui est déjà exploré de la carte sur laquelle se déplace le robot.
  - Une liste `chemin`: le chemin le plus court du robot pour revenir au point relais.
- **Initialisation**
  - `robot` est donné en entrée.
  - `carte_connue` a les dimensions de la carte d'environnement du robot, et ne contient initialement que des `'*'` et le point relais: la position de départ du robot.
  - `chemin` est vide: le robot est au point relais.
- **Méthodes**
  - `revenir(self)`: fait les déplacements du robot le long du chemin le plus court pour revenir au point relais. Au point relais, le robot remet ses rubis et recharge.

- `explorer(self)`: explore une nouvelle région en suivant le procédé suivant:
  1. Sur `carte_connue`, on cherche un point `'*'` (par exemple le plus proche).
  2. On calcule le chemin le plus court `chemin_exp` entre le robot et ce point.
  3. Le robot se déplace pas par pas vers ce point, et s'arrête juste avant, à *l'avant-dernière* position dans `chemin_exp`. Au cours du déplacement, le robot agit comme suit:
     * Avant de faire un pas, on calcule le chemin le plus court à partir de la position suivante (le `chemin` que l'on aurait si on faisait le pas), et on compare le coût de ce chemin (compte tenu des rubis portés) avec le niveau de batterie qu'on aurait après ce pas:
       · Si le niveau de batterie serait inférieur au coût de revenir, le pas n'est pas effectué; on lance `revenir(self)`.
       · Sinon, le pas est effectué. On remplace `chemin` par le chemin le plus court à partir de la nouvelle position.
     * Si le robot se trouve sur un rubis. On vérifie si, avec un rubis en plus, on peut toujours revenir en suivant `chemin`, si c'est le cas, le robot prend le rubis, et il disparaît de `carte_connue` (il va falloir trouver la nouvelle position).
  4. Si le niveau de batterie est suffisant, le robot observe:
     * Ce qui est observé par le robot devient visible sur `carte_connue`, en particulier, si le rubis est dans la partie observée, il apparaît sur `carte_connue`.
     * Ce qui est bloque n'est pas modifié sur `carte_connue`.

     Sinon on revient recharger.
- `recuperer_rubis(self)`: Calcule le chemin le plus court du robot au rubis, fait les déplacements et le récupère. Comme dans `explorer(self)`, si le niveau de batterie devient trop faible, on abandonne et revient au point relais.
- `chercher_rubis(self, direction)`: Une procédure très simple de d'exploration: le robot se déplace tout droit dans une direction, et fait une observation à chaque pas. On arête soit si on détecte le rubis, soit si on ne peut plus avancer (obstacle ou fin de la carte). On prend toujours les mêmes précautions de batterie. A

chaque observation, `carte_connue` est mise à jour comme dans `explorer(self)`.

Indication: Il peut être judicieux d'introduire ici des méthodes en plus qui "sécurisent" les méthodes du robot. Par exemple, une méthode `pas_securise` qui teste le niveau de batterie et le chemin pour revenir avant de faire le pas, et l'effectue uniquement si la batterie n'est pas trop faible. Elle pourra renvoyer `True` si le pas a été fait, et `False` sinon.

Indication: On peut aussi introduire, dans la fonction `observe_securise`, la mise à jour de `carte_connue`