A2025 – INFO COMMUNE

**ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

**Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).**

CONCOURS 2025

ÉPREUVE D'INFORMATIQUE COMMUNE

Durée de l'épreuve : 2 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

*Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :*

INFORMATIQUE COMMUNE

Cette épreuve est commune aux candidats des filières MP, PC et PSI.

L'énoncé de cette épreuve comporte 14 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence
Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.

Téléchargé gratuitement sur Doc-Solus.fr .

Autour du sac à dos

Introduction

Le problème du sac à dos, noté également KP (knapsack problem), est un problème d'optimisation combinatoire. Il modélise une situation analogue à celle du remplissage d'un sac à dos ne pouvant pas supporter plus d'un certain poids, avec tout ou partie d'un ensemble donné d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale (le profit), avec la contrainte de ne pas dépasser le poids maximum admissible (la quantité totale de ressources disponibles).

D'un point de vue mathématique, la formulation du problème du sac à dos est la suivante :

$$\textit{Maximiser le profit total } P = \sum_{i=0}^{n-1} p_i x_i \textit{ sous la contrainte } \sum_{i=0}^{n-1} r_i x_i \leq b$$

où

- n est le nombre d'objets candidats,
- i est l'entier caractérisant l'objet ($i \in [\![0, n-1]\!]$),
- p_i est le profit (ou valeur) associé à l'objet i,
- x_i est la variable de décision associée à l'objet i : $x_i = 1$ si l'objet i est sélectionné et $x_i = 0$ sinon,
- r_i est la quantité de ressources consommée par l'objet i (ou poids de l'objet i),
- b est la quantité totale de ressources disponibles (ou poids total).

On ne considère que des cas où chaque ressource r_i, chaque profit p_i et la quantité de ressources disponibles b, sont des entiers.

On fait aussi l'hypothèse que $\forall i \in [\![0, n-1]\!], r_i \leq b$.

Dans le problème du sac à dos multidimensionnel, noté MKP, on considère plusieurs sacs à dos ayant chacun un poids maximum admissible. L'objectif est alors de maximiser la valeur totale des objets contenus dans l'ensemble des sacs à dos.

Les applications pratiques sont nombreuses : transport de marchandises, découpe de matériaux, gestion de portefeuilles financiers, allocation de tâches à des systèmes multiprocesseurs, ...

1 Base de données - Exemple de fret maritime de conteneurs

On donne les tables `NAVIRES` et `CONTENEURS` suivantes :

- `NAVIRES(idN,evp,portDepN,dateDep,portDestN)`
  - idN : clé primaire, identifiant du navire (type entier, non nul)
  - evp : capacité en équivalent conteneurs de longueur 20 pieds (type entier)
  - portDepN : port de départ (type chaîne de caractères)
  - dateDep : date de départ (type date)
  - portDestN : port de destination (type chaîne de caractères)

Téléchargé gratuitement sur Doc-Solus.fr .

idN	evp	portDepN	dateDep	portDestN
12	1500	"Marseille"	2025-06-23	"Hambourg"
2	16000	"Valence"	2025-06-18	"Algésiras"
5	500	"Le Havre"	2025-10-06	"Rotterdam"
8	23000	"Hongkong"	2025-07-02	"Shanghai"
...	...	...	...	...

- `CONTENEURS(idC,idN,taille,portDepC,dateDisp,portDestC,val)`
  - idC : clé primaire, identifiant du conteneur (type entier)
  - idN : identifiant du navire attribué (type entier, valeur 0 si non encore attribué).
  - taille : longueur du conteneur, 20 ou 40 pieds (type entier). La hauteur et la profondeur sont identiques.
  - portDepC : port de départ (type chaîne de caractères)
  - dateDisp : date de mise à disposition (type date)
  - portDestC : port de destination (type chaîne de caractères)
  - val : tarification, valeur entière de 1 à 5 par ordre croissant de profit pour l'entreprise (type entier)

idC	idN	taille	portDepC	dateDisp	portDestC	val
103	12	20	"Marseille"	2025-08-08	"Hambourg"	2
218	12	40	"Marseille"	2025-07-08	"Valence"	1
5	0	40	"Le Havre"	2025-10-01	"Shangai"	5
885	8	20	"Hongkong"	2025-07-01	"Barcelone"	1
...	...	...	...	...	...	...

Pour les deux questions qui suivent, on demande d'écrire les requêtes en langage `SQL`.

On notera que pour les valeurs d'attributs de type date (format AAAA-MM-JJ), les relations d'ordre ($<, \leq, >, \geq, =, \neq$) sont autorisées. Par exemple, 2025-07-02 < 2025-10-03.

❏ **Q1** – Écrire une requête permettant de retourner la liste des identifiants de conteneurs et leur ratio tarification/longueur trié par ordre décroissant, partant de Marseille et devant aller à Barcelone, avec une mise à disposition avant le 01/01/2025.

❏ **Q2** – Écrire une requête permettant de retourner les identifiants de navire et leur nombre respectif de conteneurs attribués.

2 Structure de données pour l'espace des objets

On demande d'utiliser exclusivement le langage Python pour toute la suite du sujet.

On implémente l'espace des objets i ($i \in [\![0, n-1]\!]$) avec une structure informatique `obj` de type liste de taille n.

La valeur de `obj[i]` est un tuple `(ri,pi)` où `ri` et `pi` sont les valeurs respectives des paramètres r_i et p_i de l'objet i.

Par exemple, `(10,5)` permet de définir pour un objet étiqueté 3 les paramètres $r_3 = 10$ et $p_3 = 5$.

Une solution au problème du sac à dos $S = [x_0, x_1, ..., x_{n-1}]$, est implémentée par une liste Python. On rappelle que $x_i = 1$ si l'objet i est sélectionné et $x_i = 0$ sinon.

Téléchargé gratuitement sur Doc-Solus.fr .

Pour les deux questions qui suivent, la fonction python `sum` ne sera pas utilisée.

❑ **Q3** – Écrire une fonction `profit(obj:[(int)],S:[int])->int` prenant en argument une liste `obj` et une liste `S`, et retournant la valeur du profit `P`.

❑ **Q4** – Écrire une fonction `contrainte(obj:[(int)],S:[int],b:int)->bool` prenant en argument une liste `obj`, une liste `S` et un nombre `b`, et retournant le booléen `True` si la contrainte de ressources du problème du sac à dos est respectée, `False` dans le cas contraire.

3 Structure de données pour l'espace des solutions

On choisit de modéliser l'espace des solutions au problème du sac à dos avec un arbre binaire (figure 1). On ne s'intéresse pas pour le moment au respect de la contrainte $\sum_{i=0}^{n-1} r_i x_i \leq b$.

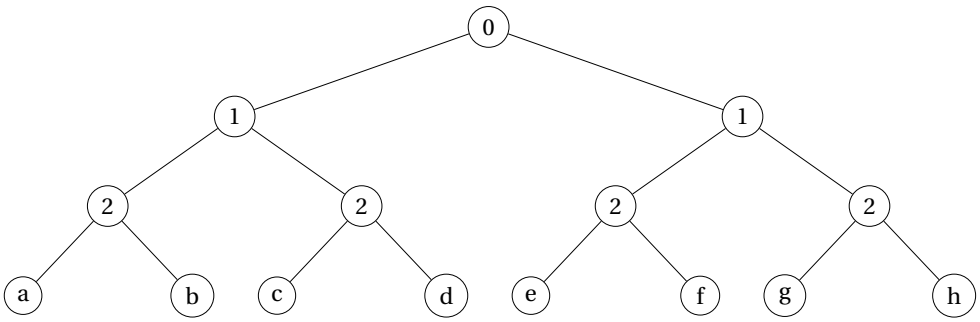

Figure 1 – Arbre binaire et espace des solutions pour le cas particulier où $n = 3$

À chaque nœud, deux choix sont possibles :
— Fils gauche : l'objet i ($i \in [\![0, n-1]\!]$) est sélectionné ($x_i = 1$).
— Fils droit : l'objet i ($i \in [\![0, n-1]\!]$) n'est pas sélectionné ($x_i = 0$).

Au niveau le plus élevé (la racine), le choix se porte sur l'objet 0. Au niveau immédiatement inférieur, le choix se fait sur l'objet 1, et ainsi de suite ... jusqu'à l'objet $n-1$.

Au niveau le plus bas, sont définies les feuilles (sans successeur). À chacune d'entre-elles correspond une solution du type $S = [x_0, x_1, ..., x_{n-1}]$ décrivant les choix effectués le long du chemin menant de la racine à la feuille considérée.

❑ **Q5** – Sur l'exemple de la figure 1 où $n = 3$, à quoi est égale la liste S pour les feuilles b et c ?

❑ **Q6** – Donner le nombre de feuilles de l'arbre binaire en fonction du nombre d'objets n. Indiquer en la justifiant la complexité temporelle en fonction du nombre d'objets n d'un algorithme de résolution du problème du sac à dos de type "force brute" qui envisagerait toutes les combinaisons possibles de sélection d'objets.

Lorsque le nombre d'objets n devient grand, un parcours de toutes les solutions n'est pas possible en un temps raisonnable.

Téléchargé gratuitement sur Doc-Solus.fr .

4 Résolution approchée par un algorithme glouton

On envisage une stratégie gloutonne pour la résolution du problème du sac à dos. Lorsque le choix de sélectionner un objet est fait, il ne sera plus remis en question. Les étapes sont les suivantes :

1) On construit la liste `Lqi` qui contient pour chaque objet $i \in [\![0, n-1]\!]$ le rapport profit sur ressource consommée $q_i = p_i / r_i$. En parallèle, on construit la liste `Li`, telle qu'initialement `Li[i]` vaut i. La liste `Li` est telle que `Li[j]` ($j \in [\![0, n-1]\!]$) donne l'indice dans la liste `obj` de l'objet décrit par le rapport situé en position j dans `Lqi`.

2) On trie la liste `Lqi` par ordre décroissant et on modifie simultanément la liste `Li` de sorte que $\forall j \in [\![0, n-2]\!], Lqi[Li[j]] \geq Lqi[Li[j+1]]$.

3) En partant de la quantité totale de ressources disponibles b, on sélectionne l'objet i de rapport q_i le plus grand possible tel que $r_i \leq b$.

4) On met à jour la quantité de ressources disponibles $b = b - r_i$ et on réitère le processus à partir de 3) jusqu'à ce qu'il ne soit plus possible de sélectionner un objet supplémentaire.

❏ **Q7** – On prend le cas particulier où `obj=[(2,3),(1,4),(4,4)]` et `b=5`. Expliquer quelle liste `S` est construite par la stratégie gloutonne précédente.

Le code incomplet de la fonction `construitLi(obj:[(int)])->[int]` qui prend en argument la liste `obj` et qui retourne la liste `Li` définie précédemment, est donné ci-après. Cet algorithme correspond aux étapes 1 et 2 décrites en introduction de la stratégie gloutonne.

```
1  def construitLi(obj):
2      Lqi=[]
3      Li=[]
4      for i in range(len(obj)):
5          Lqi.append(.........)
6          Li.append(i)
7      for i in range (1,len(Lqi)):
8          x=Lqi[i]
9          j=i
10         while j>0 and Lqi[j-1]<x:
11             Lqi[j]=Lqi[j-1]
12             Li[j]=.........
13             j-=1
14         Lqi[j]=x
15         Li[j]=.........
16     return Li
```

❏ **Q8** – Proposer le code Python complet des lignes 5, 12 et 15.

❏ **Q9** – Identifier le meilleur des cas et le pire des cas de la méthode de tri utilisée dans la fonction `construitLi` en précisant et justifiant leur complexité temporelle respective.

On donne en page suivante le code incomplet qui calcule la solution au problème du sac à dos `S` sous la forme `S=[x0,x1,...]` avec une stratégie gloutonne.

Téléchargé gratuitement sur Doc-Solus.fr .

Une fois la liste `Li` construite en ligne 2, cet algorithme correspond aux étapes 3 et 4 décrites en introduction de la stratégie gloutonne. Les variables `obj` et `b` ont été définies en amont du code.

```
1 S=len(obj)*[0]
2 Li=construitLi(obj)
3 j=0
4 while .........:
5     if .........<=b:
6         S[.........]=.........
7         b=.........
8     j+=1
```

❑ **Q10** – Proposer le code Python complet des lignes 4, 5, 6 et 7.

La complexité temporelle optimale d'une méthode de tri dans le pire des cas en fonction du nombre n de données à trier est quasi linéaire $C(n) = \Theta(nlog(n))$. On peut en déduire que la complexité optimale de la stratégie gloutonne est aussi quasi linéaire.

❑ **Q11** – On reprend le cas particulier où `obj=[(2,3),(1,4),(4,4)]` et `b=5`. Donner la solution optimale pour ce cas particulier. Conclure sur la pertinence d'une approche gloutonne.

5 Résolution exacte par un algorithme de programmation dynamique

On utilise une méthode dite de programmation dynamique pour résoudre le problème du sac à dos à n objets.

On note pour tout $k \in [\![0, n]\!]$ et tout $r \in [\![0, b]\!]$, $P_{max}(k, r)$ le profit maximum réalisable avec les objets d'indice i compris entre 0 et k non inclus pour une quantité maximale de ressources consommées r.

On pose $P_{max}(0, r) = 0$ pour tout $r \in [\![0, b]\!]$, et on considère acquise la formule de récurrence suivante, donnant pour tout $i \in [\![0, n-1]\!]$ et tout $r \in [\![0, b]\!]$, le profit maximum :

$$P_{max}(i+1, r) = max\{P_{max}(i, r), P_{max}(i, r - r_i) + p_i\}$$

où

- p_i est le profit associé à l'objet i ($i \in [\![0, n-1]\!]$),
- r_i est la quantité de ressources consommée par l'objet i ($i \in [\![0, n-1]\!]$),
- r est la quantité maximale de ressources consommées.

On stocke les valeurs de $P_{max}(k, r)$ dans un tableau T à deux dimensions sous forme de liste de listes.

On rappelle que b est la quantité totale de ressources disponibles.

La fonction `KPprogDynamique(obj:[(int)],b:int)->int` donnée en page suivante, implémente un algorithme de programmation dynamique itératif.

Téléchargé gratuitement sur Doc-Solus.fr .

```
1  def KPprogDynamique(obj,b):
2      T=[]
3      n=len(obj)
4      for k in range(n+1):
5          L=[]
6          for r in range(b+1):
7              L.append(0)
8          T.append(L)
9      for i in range(n):
10         for r in range(b+1):
11             if obj[i][0]<=r:
12                 T[i+1][r]=max(T[i][r],T[i][r-obj[i][0]]+obj[i][1])
13             else:
14                 T[i+1][r]=T[i][r]
15     return T[n][b]
```

❏ **Q12** – On reprend le cas particulier où `obj=[(2,3),(1,4),(4,4)]` et `b=5`. Donner sans justifications les valeurs du tableau `T` à l'issue de l'exécution des lignes 2 à 8. Puis, à la fin de chacune des trois itérations de la boucle `for` en ligne 9, donner la valeur de `T[i+1]`. Préciser ce que retourner la fonction `KPprogDynamique(obj,b)` et à quoi correspond cette valeur.

❏ **Q13** – Déterminer en la justifiant la complexité asymptotique temporelle de la fonction `KPprogDynamique(obj,b)`.

On conserve les 14 premières lignes de la fonction proposée précédemment. On complète avec les lignes de code suivantes afin de retourner la solution `S` sous la forme `S=[x0,x1,...]` et la valeur de `T[n][b]`.

```
1  def KPprogDynamique(obj,b):
2      #
3      #Lignes 2 à 14 précédentes conservées
4      #
5      S=n*[0]
6      k=n-1
7      r=b
8      while r>0 and k>=0 and T[k+1][r]!=0:
9          while k>0 and T[k+1][r]==T[k][r]:
10             k-=1
11         S[k]=1
12         r=r-obj[k][0]
13         k-=1
14     return S,T[n][b]
```

❏ **Q14** – Donner les valeurs des variables `S`, `k` et `r` après chaque itération de la boucle `while` en ligne 8. Indiquer en justifiant si la complexité asymptotique temporelle de la fonction `KPprogDynamique(obj,b)` est ainsi modifiée.

Téléchargé gratuitement sur Doc-Solus.fr .

6 Résolution exacte par un algorithme PSE

Un algorithme par séparation et évaluation (PSE) ou *branch and bound* permet d'éliminer de l'arbre des solutions, les branches qui ne peuvent pas contenir la solution optimale. Cet "élagage" de l'arbre permet d'éviter l'exploration complète de celui-ci.

On applique dans un premier temps un algorithme glouton afin d'avoir en un temps raisonnable une solution approchée de la solution optimale dont le profit est noté P_{min}.

La reformulation du problème initial est donc :

$$\textit{Maximiser le profit total } P = \sum_{i=0}^{n-1} p_i x_i \textit{ sous les contraintes } \sum_{i=0}^{n-1} r_i x_i \leq b \textit{ et } P > P_{min}.$$

L'arbre binaire des solutions `arbreSol` est implémenté par une structure récursive (figure 2). Chaque nœud est lui-même un arbre binaire qui construit niveau par niveau une solution. On modélise chacun par un dictionnaire possédant trois clés de type `str` :

- `'S'` : le vecteur courant solution :
  - `arbresol['S']=[ ]` pour la racine de l'arbre (liste vide),
  - `arbresol['S']=[`$x_0, x_1, ..., x_{k-1}$`]` à la profondeur k ($k \in [\![1, n]\!]$),
- `'g'` : le nœud "fils" de gauche,
- `'d'` : le nœud "fils" de droite.

À la profondeur $k = n$,

- `arbresol['S']=[`$x_0, x_1, .., x_{n-1}$`]` : le vecteur est une solution,
- `arbresol['g']={}` : le nœud "fils" de gauche est un dictionnaire vide,
- `arbresol['d']={}` : le nœud "fils" de droite est un dictionnaire vide.

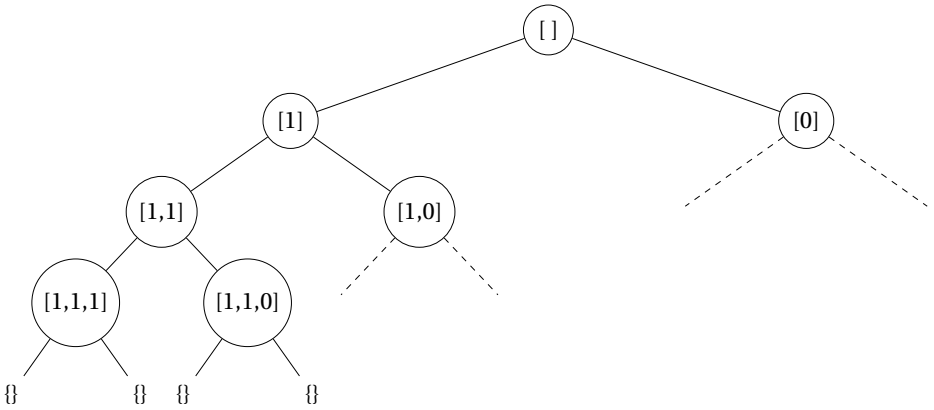

Figure 2 – Arbre binaire partiel pour $n = 3$ avec les valeurs de `arbresol['S']` à chaque nœud

La fonction récursive `construireArbreBinaire(a:dict,n:int)` permet, en page suivante, de construire un tel arbre.

Téléchargé gratuitement sur Doc-Solus.fr .

```
def construireArbreBinaire(a,n):
    if n==0:return a
    else:
        filsGauche={'S':a['S']+[1],'g':{},'d':{}}
        filsDroit={'S':a['S']+[0],'g':{},'d':{}}
        a['g']=filsGauche
        a['d']=filsDroit
        construireArbreBinaire(filsGauche,n-1)
        construireArbreBinaire(filsDroit,n-1)
```

Par exemple, le code suivant permet de construire un arbre des solutions pour $n = 3$:

```
arbreSol={'S':[],'g':{},'d':{}}
construireArbreBinaire(arbreSol,3)
```

❑ **Q15** – Écrire une fonction `estFeuille(a:dict)->bool` qui prend en argument un arbre `a` et qui retourne le booléen `True` si `a` est une feuille, `False` dans la cas contraire.

Le profit P_{min} obtenu par un algorithme glouton est stocké dans la **variable globale** `Pmin`.

❑ **Q16** – Écrire une fonction `possible(obj:[(int)],Sk:[int],b:int)->bool` qui prend en argument la liste des objets `obj`, une liste `Sk` valeur d'une solution courante d'un nœud de niveau $k \in [\![1, n]\!]$ (`len(Sk)=k`) et la quantité de ressources disponibles `b`. La fonction retourne le booléen `True` s'il est utile de poursuivre l'exploration des nœuds inférieurs. La fonction retourne `False` dans le cas contraire. On s'assure ici que :

- la condition de ressources est satisfaite : on utilise pour cela la fonction `contrainte(obj:[(int)],S:[int],b:int)->bool` avec un vecteur solution `S` de longueur n égal à la liste `Sk` complétée par des 0. On vérifie ainsi dans un premier temps que le simple ajout de l'objet à la profondeur k ($k \in [\![1, n]\!]$) ne consomme pas trop de ressources disponibles,
- il est possible de trouver une meilleure solution que l'algorithme glouton : on utilise la fonction `profit(obj:[(int)],S:[int])->int` avec un vecteur solution `S` égal à la liste `Sk` complétée par des 1 en une liste de longueur n pour simuler le profit maximal de la branche avec l'ajout de tous les objets non encore traités à la profondeur k ($k \in [\![1, n]\!]$).

On adopte un algorithme récursif `KPforceBrute(arbre:dict,obj:[(int)],b:int)`, de résolution optimale par « force brute » du problème du sac à dos. Le code Python est donné en page suivante.

La variable `Pmin` est réactualisée s'il existe une solution meilleure que celle trouvée par une stratégie gloutonne et la solution optimale correspondante est alors mise à jour dans la **variable globale** `Sol`.

Téléchargé gratuitement sur Doc-Solus.fr .

```
1  def KPforceBrute(arbre,obj,b):
2      global Pmin,Sol
3      if estFeuille(arbre) :
4          if contrainte(obj,arbre['S'],b):
5              P=profit(obj,arbre['S'])
6              if P>Pmin:
7                  Pmin=P
8                  Sol=arbre['S']
9      else:
10         KPforceBrute(arbre['g'],obj,b)
11         KPforceBrute(arbre['d'],obj,b)
```

On souhaite à la fois :

— "élaguer" les branches de l'arbre des solutions qui ne peuvent pas contenir de solution de profit supérieur à la valeur P_{min} obtenue par un algorithme glouton, et

— "élaguer" les branches de l'arbre des solutions qui dépassent la ressource maximale b.

❑ **Q17** – Le nom de la fonction `KPforceBrute(arbre:dict,obj:[(int)],b:int)` est modifié en `KPpse(arbre:dict,obj:[(int)],b:int)`. Cette fonction prend en argument l'arbre des solutions `arbre`, la liste des objets `obj` et un nombre ressources `b`. Elle affecte aux variables globales `Pmin` et `Sol` respectivement la valeur du profit maximal et le vecteur solution correspondant, conformément à l'algorithme PSE. Réécrire le code Python à partir de la ligne 9 (autant de lignes supplémentaires que nécessaire). Ne pas réécrire les autres lignes.

7 Résolution approchée par colonie de fourmis (ACO)

Principe de la méthode

L'optimisation par colonies de fourmis (ACO : *ants colony optimization*) est une méthode inspirée du comportement de fourmis lorsque celles-ci sont à la recherche de nourriture.

Le choix pour une fourmi artificielle $k \in [\![0, nb_{Ants}-1]\!]$ de sélectionner un objet plutôt qu'un autre afin d'élaborer une solution S_k, se fait avec une loi de probabilité qui évolue au cours des itérations.

Les fourmis artificielles déposent des traces de phéromone sur les objets sélectionnés dans chacune des solutions S_k élaborées, en fonction du profit engendré. La probabilité de sélectionner un objet contenu dans une solution de profit important sera augmentée.

Elles élaborent alors de nouvelles solutions relativement aux traces de phéromone précédemment déposées.

De plus, ces traces subissent une loi d'évaporation au cours des itérations.

Intuitivement, cette communication indirecte fournit une information sur la qualité des solutions envisagées, afin d'attirer les fourmis vers les zones potentiellement intéressantes de l'espace des solutions.

On adopte alors la stratégie générique suivante :

1) Initialisation

a) Initialiser les traces de phéromone $\tau(o_i) \leftarrow \tau_{max}$ pour chaque objet o_i ($i \in [\![0, n-1]\!]$)

b) Initialiser la meilleure solution $S_{bestOfAll} \leftarrow \emptyset$ et le profit $P_{bestOfAll} \leftarrow 0$

Téléchargé gratuitement sur Doc-Solus.fr .

2) Processus itératif de nb_{it} itérations (nb_{it} : valeur entière définie afin d'obtenir une solution approchée de qualité en un temps d'exécution raisonnable).

Tant que le nombre d'itérations nb_{it} n'est pas atteint, faire :

a) Initialiser les solutions $S = \{S_0, S_1, ..., S_{nb_{Ants}-1}\} / S_k \leftarrow \emptyset$, $k \in [\![0, nb_{Ants} - 1]\!]$.

b) Pour chaque fourmi $k \in [\![0, nb_{Ants} - 1]\!]$, construire une solution S_k comme suit :

i. Choisir aléatoirement un premier objet $o_0 \in [\![0, n-1]\!]$ avec une loi uniforme.

ii. $S_k \leftarrow \{o_0\}$ et la quantité de ressources disponibles b est mise à jour.

iii. $candidats \leftarrow \{o_i \in [\![0, n-1]\!] / o_i \neq o_0$ et respect de la contrainte maximale de ressources$\}$.

iv. Tant que $candidats \neq \emptyset$, faire :

— Choisir un objet $o_i \in candidats$ avec la probabilité

$$prob(o_i) = \frac{(\tau(o_i))^{\alpha}(\eta(o_i))^{\beta}}{\sum_{o_j \in candidats}(\tau(o_j))^{\alpha}(\eta(o_j))^{\beta}}$$

La probabilité de choisir un objet $prob(o_i)$ est définie avec un facteur phéromonal $\tau(o_i)$ et un facteur heuristique $\eta(o_i)$.

Le facteur heuristique $\eta(o_i)$ doit permettre de choisir les objets les plus «intéressants» de part leurs caractéristiques intrinsèques. Il est défini par un ratio profit p_i/ressource r_i tel que :

$$\eta(o_i) = b \times \frac{p_i}{r_i}$$

avec $b = b - \sum_{o_i \in S_k} r_i$, la quantité restante de la ressource lorsque la fourmi construit la solution S_k.

La quantité de ressources disponibles b est réactualisée après chaque ajout d'objet dans la solution S_k.

— Enlever de $candidats$ chaque objet qui viole des contraintes de ressources.

c) Mettre à jour les traces de phéromone $\tau(o_i)$ en fonction des solutions S_k et $S_{bestOfAll}$

i. Toutes les traces de phéromone sont diminuées afin de simuler l'évaporation. On multiplie chaque composant phéromonal par un coefficient réel de persistance $\rho \in [0, 1]$.

ii. On détermine la meilleure fourmi k_{max} du cycle courant de la boucle «Tant que le nombre d'itérations nb_{it} ...». C'est celle qui a trouvé la solution $S_{max} \in S = \{S_0, S_1, ..., S_{nb_{Ants}-1}\}$, c'est-à-dire la solution ayant le profit maximal $P(S_{max}) = P_{max}$ trouvé durant le cycle courant.

iii. On met à jour la meilleure solution $S_{bestOfAll}$ ainsi que le meilleur profit $P_{bestOfAll} = P(S_{bestOfAll})$ trouvé depuis le début de l'exécution du processus itératif (y compris le cycle courant).

iv. La meilleure fourmi k_{max} du cycle courant dépose de la phéromone.

La quantité déposée est égale à : $1/(1 + P_{bestOfAll} - P_{max})$.

Cette quantité de phéromone est ajoutée sur chacun des composants phéromonaux $\tau(o_i)$ de S_{max}, c'est-à-dire correspondant aux objets o_i sélectionnés dans S_{max}.

Téléchargé gratuitement sur Doc-Solus.fr .

v. Si une trace de phéromone est inférieure à τ_{min} alors la mettre à τ_{min}.
vi. Si une trace de phéromone est supérieure à τ_{max} alors la mettre à τ_{max}.

Implémentation de l'algorithme

Afin qu'une fourmi artificielle k puisse choisir un objet $o_i \in candidats$ au cours de la construction d'une solution S_k, on implémente l'ensemble des $candidats$ avec une liste `candidats` d'indices $i \in [\![0, n-1]\!]$ représentant la position dans la liste `obj` des objets o_i compatibles avec la quantité de ressources disponibles b.

De plus, on utilise un dictionnaire `prob` tel qu'à la clé `i` (indice de l'objet o_i) correspond la valeur de la probabilité $prob(o_i)$ (comprise en 0 et 1).

Les deux paramètres α et β sont définis dans deux variables globales. Par exemple, `alpha=2` et `beta=5`.

La liste `T` de taille n contient les facteurs phéromonaux associés aux objets $\tau(o_i)$ ($o_i \in [\![0, n-1]\!]$).

La quantité de ressources disponibles est stockée dans la variable `b`.

Les paramètres ρ, τ_{max} et τ_{min} sont définis dans des variables globales. À titre d'exemple, `rho=0.8`, `Tmax=6` et `Tmin=0.01`.

La liste `S` contient les listes `S[k]` des solutions S_k ($k \in [\![0, nb_{Ants}-1]\!]$) construites par chacune des fourmis artificielles lors d'une itération de l'algorithme.

On rappelle que chaque solution est de la forme $S_k = [x_0, x_1, .., x_{n-1}]$ avec $x_i \in \{0, 1\}$ ($i \in [\![0, n-1]\!]$).

La meilleure de toutes les solutions construites depuis le début de l'exécution de l'algorithme et le profit associé, sont stockés dans les variables globales respectives `SbestOfAll` et `PbestOfAll`.

On donne ci-après le code incomplet de la fonction `mettreAjourTetSolution(obj:[(int)],T:[float],S:[[int]])`, qui prend en argument la liste des objets `obj`, la liste des facteurs phéromonaux `T` et la liste des solutions `S`, et qui met à jour la liste des facteurs phéromonaux `T`, la meilleure solution `SbestOfAll` et le meilleur profit `PbestOfAll`. Cette fonction ne retourne rien et correspond à l'item 2)c) de la description de l'algorithme.

```
def mettreAjourTetSolution(obj,T,S):
    global SbestOfAll,PbestOffAll
    # 2)c)i.
    # Utiliser le nombre de lignes nécessaires
    ..............................
    # 2)c)ii.
    #Utiliser le nombre de lignes nécessaires
    ..............................
    # 2)c)iii.
    if Pmax>PbestOfAll:
        PbestOfAll=Pmax
        SbestOfAll=S[kmax]
    # 2)c)iv.v.vi.
    #Utiliser le nombre de lignes nécessaires
    ..............................
```

Téléchargé gratuitement sur Doc-Solus.fr .

❏ **Q18** – Proposer le code Python complet des lignes 4 et plus de la partie 2)c)i. de la fonction ci-dessus.

❏ **Q19** – Proposer le code Python complet des lignes 7 et plus de la partie 2)c)ii.

❏ **Q20** – Proposer le code Python complet des lignes 14 et plus de la partie 2)c)iv.v.vi.

La fonction `randint(a,b)` de la bibliothèque `random` a été importée. Elle permet de retourner de manière aléatoire un entier x / $a \leq x \leq b$.

La fonction `miseAjourCandidats(obj:[(int)],candidats:[int],b:int)` a déjà été implémentée. Elle permet de supprimer de la liste `candidats` tous les objets de la liste `obj` ne respectant pas la contrainte de ressource b. (item 2)b)iii.).

La fonction `L.remove(x)` appliquée à une liste `L` permet de retirer un élément `x` de la liste.

On donne ci-après le code incomplet qui permet de trouver une solution au problème posé en appliquant un algorithme par colonie de fourmis.

```
1  #1)a)
2  n=len(obj)
3  T=[Tmax for i in range(n)]
4  #1)b)
5  PbestOfAll=0
6  SbestOfAll=n*[0]
7  #2)
8  for it in range(nbit):
9      #2)a)
10     S=[n*[0] for i in range(nbAnts)]
11     #2)b)
12     for k in range(nbAnts):
13         b2=b
14         #2)b)i.
15         ................
16         #2)b)ii.
17         ................
18         ................
19         #2)b)iii.
20         candidats=[i for i in range(n) if i!=o0]
21         miseAjourCandidats(obj,candidats,b2)
22         #2)b)iv.
23         while b2>0 and candidats!=[ ]:
24             #Utiliser le nombre de lignes nécessaires
25             ..............................
26     mettreAjourTetSolution(obj,T,S)
```

❏ **Q21** – On donne le nom de variable `o0` (o zéro) pour l'indice de l'objet choisi. Proposer le code Python complet des lignes 15, 17 et 18, définies ci-dessus.

Téléchargé gratuitement sur Doc-Solus.fr .

On propose ci-dessous l'ébauche d'une fonction
`construitProb(obj:[(int)],candidats:[int],b:int,T:[float])->{int:float}`,
qui prend en argument la liste des objets `obj`, la liste des indices `candidats`, l'entier b et la liste `T`, et qui retourne le dictionnaire `prob` des valeurs de probabilité associées à chacun des objets $o_i \in candidats$. Cette fonction participe à la réalisation des tâches décrites à l'item 2)b)iv. de la description de l'algorithme.

```
1  def construitProb(obj,candidats,b,T):
2      m=len(candidats)
3      prob=..........
4      for i in range(m):
5          oi=..........
6          ..........
7      s=sum(prob.values())
8      for i in range(m):
9          oi=..........
10         ..........
11     return prob
```

❏ **Q22** – Proposer le code Python complet des lignes 3, 5, 6, 9 et 10.

On dispose d'une fonction
`choixCandidat(candidats:[int],prob:{int:float})->int`,
qui prend en argument la liste `candidats` et le dictionnaire `prob`, et qui retourne l'indice du candidat sélectionné. Cette fonction participe à la réalisation des tâches décrites à l'item 2)b)iv. de la description de l'algorithme.

❏ **Q23** – Proposer le code Python complet des lignes 24 et plus, définies en page précédente.

8 Analyse des résultats

On réalise une simulation avec 20 objets ayant chacun une quantité de ressources consommées comprise entre 1 et 100, et un profit respectif compris entre 1 et 100.

La quantité de ressources disponibles a été fixée à 200.

Les résultats sont affichés pour chaque méthode avec le format :

— Nom de la méthode : temps d'exécution en secondes
— Solution, profit maximum
— Liste T pour l'algorithme ACO.

```
1 Force brute :  1.275291500001913
2 [0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 567
3
4 Glouton :  2.300000051036477e-05
5 [0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 548
6
7 PSE :  0.026752800011308864
8 [0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 567
9
```

Téléchargé gratuitement sur Doc-Solus.fr .

```
Programmation dynamique :  0.00085039998521097
[0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 567

ACO avec nbAnts=5 et nbit=1 :  0.00029949998133815825
[0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 548
Liste T :  [4.800..., 5.800..., 4.800..., 4.800..., 5.800..., ...]

ACO avec nbAnts=5 et nbit=5 :  0.0012731999740935862
[0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 567
Liste T :  [1.966..., 2.967..., 1.966..., 1.966..., 4.767..., ...]

ACO avec nbAnts=5 et nbit=50 :  0.011390200001187623
[0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 567
Liste T :  [0.01, 0.065..., 0.01, 0.01, 3.799..., ...]
```

❑ **Q24** – Commenter les résultats quant aux critères de performance (temps d'exécution et qualité de la solution proposée). Commenter les résultats obtenus par l'algorithme ACO en précisant sur quels paramètres de la méthode on peut jouer, et quelles sont les conséquences sur les performances.

Fin du problème

Téléchargé gratuitement sur Doc-Solus.fr .