Arnaud Bégyn arnaud.begyn[AT]wanadoo.fr

1 Base de données - Exemple de fret maritime de conteneurs

Q1.
```
SELECT idC, taille/val AS ratio FROM CONTENEURS
WHERE portDepC = "Marseille" AND portDestC = "Barcelone" AND dateDisp < 2025-01-01
ORDER BY ratio DESC
```

Q2.
```
SELECT N.idN, COUNT(C.idC) AS nb_conteneurs FROM NAVIRES AS N
JOIN CONTENEURS AS C
ON N.idN = C.idN GROUP BY C.idN
```

2 Structure de données pour l'espace des objets

Q3.

```python
def profit(obj, S) :
    P = 0
    n = len(obj)
    for k in range(n) :
        p = obj[k][1]
        x = S[k]
        P = P + p*x
    return P
```

Q4.

```python
def contrainte(obj, S, b) :
    C = 0
    n = len(obj)
    for k in range(n) :
        r = obj[k][0]
        x = S[k]
        C = C + r*x
    return C <= b
```

3 Structure de données pour l'espace des solutions

Q5. Pour `b` : `S = [1, 1, 0]`.
Pour `c` : `S = [1, 0, 1]`.

Q6. Le nombre de feuilles est 2^n. À chaque feuille on calcule le profit et la contrainte ce qui a un coût en $O(n)$ et on teste la contrainte (coût en $O(1)$). Au total on a donc un coût en $O(n2^n)$. On doit aussi recher le profit maximum parmi au plus 2^n listes ce qui a un coût en $O(2^n)$. Le coût total est donc $O(n2^n) + O(2^n) = O(n2^n)$.

4 Résolution approchée par un algorithme glouton

Q7. `Lqi = [4, 3/2, 1]` et `Li = [1, 0, 2]`.
On choisit l'objet `1` et alors `b = 5-1 = 4`.
On choisit ensuite l'objet `0` et alors `b = 4 - 2 = 2`.
On ne peut pas choisir l'objet `2` car `4 > b`.
Il n'y a plus d'objets à tester.
On a alors `S= [1, 1, 0]` avec `P = 7` et `C = 3`.

Q8.

```
def construitLi(obj) :
    Lqi = []
    Li = []
    for i in range(len(obj)) :
        Lqi.append(obj[i][1]/obj[i][0])
        Li.append(i)
    for i in range (1, len(Lqi)) :
        x = Lqi[i]
        j = i
        while j>0 and Lqi[j-1] < x :
            Lqi[j] = Lqi[j-1]
            Li[j] = Li[j-1]
            j -= 1
        Lqi[j] = x
        Li[j] = i
    return Li
```

Q9. La première boucle s'effectue dans tous les cas avec un coût en $O(n)$ (les instructions qu'elle effectue sont de coût constant).
La seconde s'effectue n fois. Dans le pire des cas la boucle `while` est effectuée j fois et elle effectue des instructions à coût constant donc au total dans le pire des cas le coût de cette boucle est en $O(1+2+\cdots+n) = O(n^2)$. Le coût total de la fonction est alors $O(n) + O(n^2) = O(n^2)$.
Dans le meilleur des cas la boucle `while` n'est pas effectué (liste déjà triée dans l'ordre décroissant) donc le coût total est $O(n) + O(n) = O(n)$.

Q10.

```
S = len(obj)*[0]
Li = construitLi(obj)
j = 0
while j < len(obj) :
    if obj[Li[j]][0] <= b :
        S[Li[j]] = 1
        b = b - obj[Li[j]][0]
    j += 1
```

Q11. Avec la contrainte `b = 5` on ne peut prendre que deux objets :

- ▷ 0 et 1 qui donnent un profit de 7, c'est le choix glouton ;
- ▷ 1 et 2 qui donnent un profit de 8, c'est la solution optimale.

Sans surprise l'approche gloutonne ne donne pas la solution optimale mais une bonne valeur approchée.

5 Résolution exacte par un algorithme de programmation dynamique

Q12. À l'issue des lignes 2 à 8 `T` est la matrice nulle à `n+1=4` lignes et `b+1=6` colonnes.
Ensuite `i` va de `0` à `2` et alors `T[i+1]` vaut successivement :
`[0, 0, 3, 3, 3, 3]`
`[0, 4, 4, 7, 7, 7]`
`[0, 4, 4, 7, 7, 8]`
La fonction renvoie `8` qui correspond au profit maximal (d'après **Q11.**)

Q13. Les deux boucles imbriquées exécutent des intructions de coût constant donc la complexité est en $O(nb) + O(nb) = O(nb)$.

Q14. Initialement : `S = [0, 0, 0]` et `k = 2` et `r = 5`.
À la fin de la première itération : `S = [0, 0, 1]` et `k = 2` et `r = 1`.
À la fin de la seconde itération : `S = [0, 1, 1]` et `k = 0` et `r = 0`.
La boucle s'arrête car r = 0.
Toutes les instructions sont de coût constants.
Les boucles décrémentent `r` et `k` qui sont initialisées à `b` et `n-1`.
Donc dans le pire des cas la complexité est en $O(nb)$.
Au total on a donc une complexité en $O(nb) + O(nb) = O(nb)$. La complexité est inchangée.

6 Résolution exacte par un algorithme PSE

Q15.

```
def estFeuille(a) :
    return a['g'] == {} and a['d'] == {}
```

Q16.

```
def possible(obj, Sk, b) :
    n = len(obj)
    # premier test
    k = len(Sk)
    S = Sk + (n-k)*[0]
    test1 = contrainte(obj, S, b)
    # second test
    S = Sk + (n-k)*[1]
    test2 = profit(obj, S) > profit(obj, glouton(obj, b))
    # conclusion
    return test1 and test2
```

Q17.

```
else :
    Sk = arbre['g']['S']
    if possible(obj, Sk, b) :
        KPpse(arbre['g'], obj, b)
        KPpse(arbre['d'], obj, b)
```

7 Résolution approchée par colonie de fourmis (ACO)

Q18.

```
# Utiliser le nombre de lignes nécessaires
n = len(obj)
for i in range(n) :
    T[i] = rho * T[i]
```

Q19.

```
#Utiliser le nombre de lignes nécessaires
nbAnts = len(S)
kmax = 0
for k in range(1, nbAnts) :
    if S[k] > S[kmax] :
        kmax = k
Pmax = profit(obj, S[kmax])
```

Q20.

```
#Utiliser le nombre de lignes nécessaires
for i in range(n) :
    if S[kmax][i] == 1 :
        T[i] =  T[i] + 1 / ( 1 +  PbestOfAll - Pmax)
        if T[i] < Tmin :
            T[i] = Tmin
        if T[i] > Tmax :
            T[i] = Tmax
```

Q21.

```
Oo = randint(0, n-1)
#2)b)ii.
S[k][Oo] = 1
b2 = b2 - obj[Oo][0]
```

Q22.

```
def construitProb(obj, candidats, b, T) :
    m = len(candidats)
    prob = { i : 0 for k in range(m) }
    for i in range(m) :
        oi = candidats[i]
        prob[i] = T[oi]**alpha * ( b * obj[oi][1] / obj[oi][0] )**beta
    s = sum(prob.values())
    for i in range(m) :
        oi = candidats[i]
        prob[i] = prob[i]/s
    return prob
```

Q23.

```
#Utiliser le nombre de lignes nécessaires
prob = construitProb(obj, candidats, b2, T)
i = choixCandidat(candidats, prob)
Oi = candidats[i]
S[k][Oi] = 1
candidats.remove(Oi)
b2 = b2 - obj[Oi][0]
miseAjourCandidats(obj, candidats,b2)
```

8 Analyse des résultats

Q24. L'algorithme glouton est le plus rapide mais ne donne pas la solution optimale. Parmi ceux qui donnent la solution optimale c'est l'algorithme de programmation dynamique qui est le plus rapide.
Pour l'algorithme ACO on voit qu'un nombre d'itérations (`nbit`) trop petit ne donne pas la solution optimale, mais qu'un nombre d'itérations trop grand ralenti beaucoup l'algorithme. On peut faire la même conjecture pour le nombre de fourmis `nbants` mais aucun exemple n'est donné pour la confirmer. On peut aussi jouer sur les paramètres `alpha` et `beta` dans le but d'améliorer la précision.