Projet IA41 A06

Problème du sac à dos

Binôme : Monneret Nicolas
Haffner Alexandre

Enoncé du problème

On dispose d'un ensemble d'objets. Chaque objet i possède un poids p(i) > 0 et une utilité u(i) > 0. p(i) et u(i) sont des entiers naturels. On dira qu'un chargement est un sous-ensemble d'objets. Le poids d'un chargement est naturellement la somme des poids des objets qu'il contient; son utilité est la somme des utilités des objets qu'il contient.
L'objectif consiste à déterminer le chargement d'utilité maximale, dont le poids n'excède pas un poids P donné.

Analyse du problème et démarche

Le problème posé implique de développer un algorithme (efficace si possible) permettant de trouver un sac optimal pour un ensemble d'objets et un poids maximal donnés.

Nous avons tout d'abord tenté de mettre à plat nos différentes idées :

La première façon de prendre le problème que nous avons envisagée fut la suivante. Nous avions dans l'idée de créer un arbre contenant l'ensemble des possibilités en fonction des données. Une fois cet arbre créé il aurait suffi de choisir la possibilité la plus intéressante. Cette idée nous a paru immédiatement beaucoup trop coûteuse. En effet, établir ce genre d'arbre revient à prendre ou ne pas prendre chacun des objets, ce qui donne 2^n possibilités. Au vu des jeux d'essais proposés dans le sujet (50 objets) nous avons rapidement abandonné l'idée.

Nous avons ensuite imaginé un moyen plus rapide pour effectuer ce travail, chercher en quelque sorte l'intérêt d'un objet en calculant le rapport utilité/poids. Apres avoir trié ces objets par intérêt nous aurions pu remplir le sac avec les objets les plus intéressant en premier. Apres avoir simulés quelques jeux d'essais sur papier nous avons remarqué que cette méthode ne nous menait pas toujours à un résultat final exact. Prenons par exemple le jeu d'essai suivant (nous prendrons pour les objets la convention de l'énoncé (poids utilité)) :

Objets : (8 4), (3 3)
Poids maximum : 10

L'objet (8 4) possède un intérêt de 0,5.
L'objet (3 3) possède un intérêt de 1.

Il faut alors placer dans le sac l'objet (3 3) car il possède le plus grand intérêt. A partir de là il n'est plus possible d'ajouter quelque chose dans le sac sans dépasser les 10kg. Nous obtenons donc un sac final ayant une utilité de 3 pour un poids de 3.

Le problème est que nous recherchons le sac ayant la plus grande utilité possible. En ayant choisi l'objet (8 4) nous aurions obtenu un sac ayant une utilité de 4. La solution qu'aurait fournie cet algorithme aurait été erronée dans bien des cas, nous avons donc écarté cette idée.

Nous avons finalement imaginé mettre en place un algorithme (récursif) basé sur le choix : prendre l'objet ou ne pas prendre l'objet, tout en limitant le nombre de choix en supprimant en quelque sorte des branches de l'arbre. L'idée est la suivante :
Nous avons une liste d'objets et un sac optimal à trouver, donc son poids maximum. Lorsque nous prenons le premier objet nous avons le choix de le mettre ou non dans le sac. L'important étant de choisir la meilleur des deux façons de faire il suffi en fait de choisir celle qui nous permettra d'obtenir le meilleur sac (ayant l'utilité maximum). Pour ce faire il faudra comparer les utilités totales de deux sacs :

- Le sac optimal sans cet objet (il suffira donc de relancer la routine avec le reste des objets).
- Le sac optimal avec cet objet, qui sera en fait le sac optimal pouvant contenir le reste des objets avec suffisamment de place pour pouvoir ajouter l'objet courant (ici il faudra relancer la routine avec le reste des objets mais aussi avec un poids maximum plus petit de p(i) kg (avec p(i) le poids de l'objet courant afin de pouvoir l'ajouter).

Le meilleur de ces deux sacs sera le sac optimal pour une liste d'objets et un poids maximum donné. L'avantage ici est que lorsqu'un sac ne peut plus contenir l'objet il ne reste plus qu'un choix et beaucoup de possibilités sont ainsi supprimées.
La récursivité est assez évidente il suffit donc de trouver les cas limites qui sont les suivants :
Si il n'y a pas d'objets à prendre ou si la capacité du sac est égale à zéro alors le sac optimal est le sac vide avec une utilité de zéro. On ne considèrera donc pas les sac ayant une capacité de zéro comme valide puisqu'il ne peuvent pas contenir d'objet ayant un poids supérieur à zéro (comme spécifié dans l'énoncé), ce qui nous permettra dans certain cas de gagner quelques étapes pour arriver au résultat.

Nous allons devoir manipuler un ensemble d'objets et de sacs (contenant ces objets) afin d'obtenir un résultat au problème posé. Ceci implique de définir les structures de données adéquates afin de pouvoir travailler aisément sur celles-ci :

Un objet : (poids utilite)
Un sac : (utilité_totale (obj1 obj2 ... objN))

Ainsi qu'un ensemble de fonctions permettant de les manipuler :

Utilité : (1) utilite (objet)
(utilite '(2 3)) => 3
(utilite '(7 9)) => 9

Poids : (1) poids (objet)
(poids '(2 3)) => 2
(poids '(7 9)) => 7

Créer un sac : (0) creerSac ()
(creerSac) => (0 ( ))

Ajouter un objet dans un sac : (2) ajoutObjetSac (sac objet)

(ajoutObjetSac '(7 ((5 3) (4 4)))
'(2 3))

=> (10 ((2 3) (5 3) (4 4)))

Comparer deux sac : (2) meilleurSac (sac1 sac2)

(meilleurSac '(7 ((5 3) (4 4)))
'(10 ((2 3) (5 3) (4 4))))

=> (10 ((2 3) (5 3) (4 4)))

Apres avoir défini tout ceci nous avons partagé le travail et nous sommes arrivés à la solution suivante.

Solution

```
;renvoyer l' utilite d'un objet
(defun utilite (objet)
  (cadr objet)
)


;renvoyer le poid d'un objet
(defun poid (objet)
  (car objet)
)


;creer un sac (Utilité_totale (liste d'objets))
(defun creerSac ()
  (list 0 ())
)


;ajouter un objet a un sac en additionnant l'utilité de l'objet à l'utilité totale du sac
(defun ajoutObjetSac (sac objet)
  (cons (+ (car sac) (utilite objet)) (list (cons objet (cadr sac))))
)


;renvoyer le sac ayant la meilleure utilité totale
(defun meilleurSac (sac1 sac2)
  (if (> (car sac1) (car sac2))
    sac1
    sac2
  )
)


;fonction principale, renvoie le sac optimal
(defun sacOptimal (poidMax listeObjets)
      (cond

            ;s'il n'y a plus d'objet à traiter le sac optimal est le sac vide
            ((null listeObjets) (creerSac))

            ;de meme si l'on cherche le sac optimal ayant une capacité maximum de 0
            ((eq 0 poidMax) (creerSac))

            ;si le poid du premier objet de la liste est supérieur au poid maximum du sac,
            ;autrement dit, si le sac ne peut pas contenir l'objet le sac optimal est donc le sac
            ;optimal avec le reste des objets et le meme poid maximum étant donné que l'on n'a
            ;pas ajouté le premier objet.
            ((> (poid (car listeObjets)) poidMax)
                        (sacOptimal poidMax (cdr listeObjets))
            )

            ;si le sac peut contenir le premier objet il faut choisir le meilleur sac résultant
            ;des actions : prendre l'objet ou ne pas le prendre
            (t    (meilleurSac

                  ;si on ne prend pas l'objet on calcul le sacOptimal avec le reste des objets
                  (sacOptimal poidMax (cdr listeObjets))
```

```
                        ;si on prend l'objet on calcul le sac optimal avec le reste des objets
                        ;pouvant le contenir, c'est à dire avec un poid diminué de
                        ;(poid objet_courant) auquel on ajoute l'objet courant
                        (ajoutObjetSac
                              (sacOptimal (- poidMax (poid (car listeObjets)))
                                    (cdr listeObjets)
                              )
                              (car listeObjets)
                        )
                  )
            )
      )
)
```

Jeux d'essais

Nous avons pu procédé, à partir de là, à différents tests en prenant plusieurs jeux d'essais. Nous avons mesuré les temps d'exécution grâce à la fonction suivante qui nous était fournie :

```
;Version Clisp
(defun getElapsedTime (fnc)
      (let ((v1 (get-internal-real-time)))
            (print (eval fnc))
            (let ((v2 (get-internal-real-time)))
                  (/ (float (- v2 v1)) (float internal-time-units-per-second))
            )
      )
)
```

Les temps d'exécution sont relatifs à nos PC. Nous les indiquons ici à titre indicatif et serviront à des comparaisons relativement à plusieurs algorithmes. Commençons par les jeux d'essais imposés par le sujet.

Jeu d'essai #1 : Les 10 objets suivants :

(1 6), (2 2), (3 3), (4 1), (5 5), (10 7), (4 3), (3 4), (2 5), (1 3)

Nous obtenons les résultats suivants pour les différents poids :

Poids = 5 :	**(13 ((1 6) (2 5) (1 3)))**	temps : 0.005 sec
Poids = 10 :	**(21 ((1 6) (3 3) (3 4) (2 5) (1 3)))**	temps : 0.010 sec
Poids = 15 :	**(26 ((1 6) (3 3) (5 5) (3 4) (2 5) (1 3)))**	temps : 0.016 sec
Poids = 20 :	**(29 ((1 6) (3 3) (5 5) (4 3) (3 4) (2 5) (1 3)))**	temps : 0.016 sec
Poids = 25 :	**(33 ((1 6) (3 3) (5 5) (10 7) (3 4) (2 5) (1 3)))**	temps : 0.016 sec
Poids = 30 :	**(36 ((1 6) (3 3) (5 5) (10 7) (4 3) (3 4) (2 5) (1 3)))**	temps : 0.016 sec

Jeu d'essai #2 : Les 20 objets suivants :

(1 6), (1 2), (1 3), (2 1), (2 5), (2 7), (2 3), (2 4), (2 5), (3 3),
(3 5), (3 8), (4 4), (4 8), (4 5), (5 3), (5 6), (6 8), (7 5), (8 5)

Nous obtenons les résultats suivants :

Poids = 5 :	**(18 ((1 6) (1 2) (1 3) (2 7)))**	temps : 0.016 sec
Poids = 10 :	**(31 ((1 6) (1 2) (1 3) (2 5) (2 7) (3 8)))**	temps : 0.083 sec
Poids = 15 :	**(42 ((1 6) (1 3) (2 5) (2 7) (2 5) (3 8) (4 8)))**	temps : 0.550 sec
Poids = 20 :	**(51 ((1 6) (1 2) (1 3) (2 5) (2 7) (2 3) (2 4) (2 5) (3 8) (4 8)))**	temps : 1.900 sec
Poids = 25 :	**(58 ((1 6) (1 2) (1 3) (2 5) (2 7) (2 4) (2 5) (3 5) (3 8) (4 8) (4 5)))**	temps : 4.500 sec
Poids = 30 :	**(64 ((1 6) (1 2) (1 3) (2 5) (2 7) (2 3) (2 4) (2 5) (3 3) (3 5) (3 8) (4 8) (4 5)))**	temps : 8.183 sec

Jeu d'essai #3 : Les 50 objets suivants :

(1 8), (1 1), (1 9), (1 1), (1 5), (2 7), (2 4), (2 4), (2 2), (2 8),
(3 9), (3 9), (3 5), (3 8), (3 3), (4 7), (4 4), (4 2), (4 5), (4 1),
(5 5), (5 8), (5 4), (5 8), (5 5), (6 3), (6 6), (6 8), (6 5), (6 5),
(7 5), (7 8), (7 4), (7 8), (7 5), (8 3), (8 6), (8 8), (8 5), (8 5),
(9 5), (9 8), (9 4), (9 8), (9 5), (10 3), (10 6), (10 8), (10 5), (10 5)

Nous obtenons les résultats suivants :

Poids = 10 :	**(46 ((1 8) (1 9) (1 5) (2 7) (2 8) (3 9)))**	temps : 1.100 sec
Poids = 15 :	**(59 ((1 8) (1 9) (1 5) (2 7) (2 4) (2 8) (3 9) (3 9)))**	temps : 14.38 sec
Poids = 20 :	**(71 ((1 8) (1 9) (1 5) (2 7) (2 4) (2 4) (2 8) (3 9) (3 9) (3 8)))**	temps : 130.2 sec

Nous n'irons pas ici, au dessus d'un poids supérieur à 20, le temps d'exécution étant trop long. Toutefois nous pourrons exécuter ces jeux d'essais plus tard dans une version plus optimisée de l'algorithme.

Jeu d'essai #4 : Pas d'objet avec n'importe quel poids maximum :
résultats : (0 NIL)

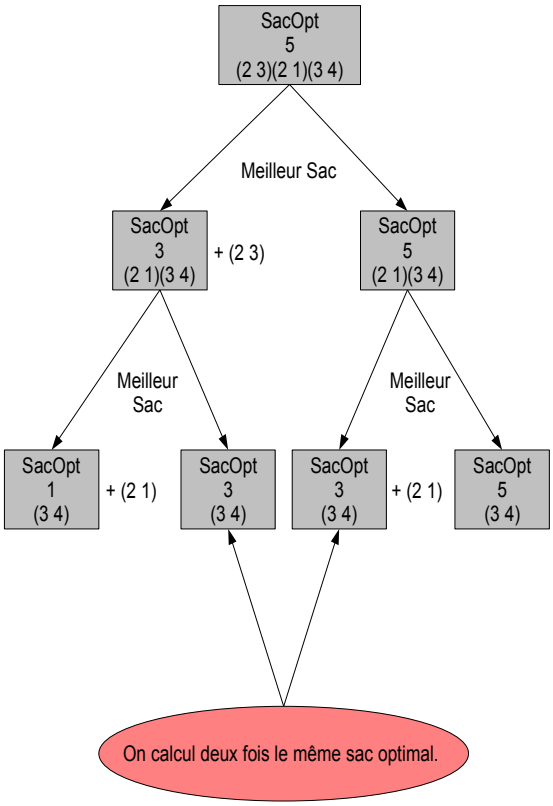
Jeu d'essai #5 : N'importe quelle liste d'objet avec un poids maximal de 0 :
résultats : (0 NIL)

Améliorations

Comme vous avez pu le constater dans les jeux d'essais, le temps d'exécution devient vite très important. Le temps pour arriver à la solution avec cette démarche ne dépendra pas uniquement du nombre d'objets mais aussi du poids maximum et du jeu d'essais proposé : en effet, avec certaines liste d'objets la plupart des « branches » crées lors du calcul du maximum de deux sac optimaux peuvent rapidement arriver à un cas limite. Cependant la plupart du temps l'algorithme implique de calculer plusieurs fois le même sac optimal ce qui est clairement une perte de temps. Prenons par exemple le jeu d'essai suivant :

Poids maximum : 5
Liste d'objets : (2 3) (2 1) (3 4)

Nous pouvons définir le schéma suivant en appliquant l'algorithme précédent.



On calcul deux fois le même sac optimal. Ici l'influence est limitée car il y a peu d'objets. Effectuer deux fois ce calcul revient à effectuer tous les calcul qui en découle deux fois et ainsi de suite pour chaque redondance.

Nous avons donc tenté de trouver une solution à ce problème. Nous avons pensé à travailler itérativement de façon à pouvoir réutiliser les résultats pour les calculs suivants. Nous avons consulté des ouvrages sur Internet traitant de ce genre de problèmes et nous avons trouvé une méthode qui semblait convenir : le programmation dynamique.

« *Le problème du sac à dos possède la propriété de sous structure optimale, c'est-à-dire que l'on peut construire la solution optimale du problème à i variables à partir du problème à i-1 variables. Cette propriété permet d'utiliser une méthode de résolution par programmation dynamique.* »
Source : Wikipedia

Nous avons donc exploré un peu plus cette voie et découvert qu'il était tout simplement possible de prendre l'algorithme que nous avions conçu à l'envers. Le but étant de commencer à partir des cas limite pour chercher le sac optimal pour chaque nouvel objet et chaque poids maximum. L'astuce est de conserver l'ensemble de ces résultats dans un tableau afin de pouvoir le réutiliser pour y chercher les sacs optimaux déjà calculés.

On peu assimiler cette méthode à celle permettant de créer un triangle de pascal.

On connaît les cas limites qui sont les 1 placé dans la première colonne et la diagonale :

```
1
11
1 1
1  1
```

A partir de ces cas limite il est possible de calculer chacun des éléments du triangle si tant est qu'on les calculs dans l'ordre afin de pouvoir réutiliser les éléments précédents.

```
1
11
121
1331
.....
```

Le principe dans notre cas est exactement le même. Nous allons créer un tableau à deux dimensions avec en colonnes les différentes listes d'objets et en lignes les différents poids. Par exemple :

Poids maximum : 5
Liste d'objets : (2 3) (2 1) (3 4)

SV = SacVide

	( )	(2 3)	(2 1)	(3 4)
0	SV	SV	SV	SV
1	SV			
2	SV			
3	SV			
4	SV			
5	SV			

Une fois le tableau mis en place il ne reste qu'à le remplir colonne par colonne jusqu'à arriver à la dernière case : (3 4), 5. Prenons par exemple la première case à remplir : (2 3), 1.

Comme dans l'algorithme précédent on doit choisir entre prendre ou ne pas prendre l'objet (2 3). La seule des deux options possible est de ne pas prendre l'objet étant donné que la capacité du sac est de 1. Le sac optimal pour un poids maximum de 1 et pouvant contenir l'objet (2 3) est le sac vide. Remplissons une case i, j : on supposera donc que les cases k, m avec k < i ont étés remplies dans les itérations précédentes. Le sac optimal i, j sera la meilleure façon de faire entre prendre ou ne pas prendre l'objet. Si l'on ne prend pas l'objet le sac optimal sera le sac optimal pouvant contenir tous les objets précédents ayant un poids maximum égal à celui que l'on cherche, autrement dit le sac de la case (i-1), j. Ce sac optimal a déjà été calculé. Si l'on prend l'objet le sac optimal sera le sac optimal pouvant contenir les objets précédent ayant un poids maximum de j-p(i), avec p(i) le poids de l'objet que l'on traite, afin de pouvoir y ajouter l'objet. Le sac qui se trouve à la case (i-1), (j-p(i)) a lui aussi été calculé. On peut donc lui ajouter l'objet courant pour connaître son utilité. Il suffit ensuite de comparer les deux sac résultants des deux possibilités pour choisir la meilleur (donc obtenir le sac optimal pour la case i, j).

Il ne reste plus qu'à continuer jusqu'à considérer tous les objets, et tous les poids. Le sac optimal final sera donc le contenu de la case : dernierObj, poidMax. Dans l'exemple ce sera la case (3 4), 5.

Afin de mettre en oeuvre cet algorithme en lisp nous avons du développer un nouveau panel de fonctions pour nous permettre entre autre de manipuler un tableau à deux dimensions : le créer, rechercher un élément, modifier une case, le parcourir...

Créer un tableau à deux dimensions :	T2D (i j val)
Accéder à une case du tableau :	accesT2D (i j tab)
Modifier une case du tableau :	change (val i j tab)
Créer une liste de couples d'index :	creerIndex (debut_i fin_i debut_j fin_j)
Renvoie le i-ème objet de la liste :	iemeObjet (i listeObjets)

(T2D 2 3 '*)	=>	((* * *) (* * *))
(accesT2D 1 2 '((1 2 3) (4 5 6) (7 8 9)))	=>	6
(change '* 1 2 '((1 2 3) (4 5 6) (7 8 9)))	=>	((1 2 3) (4 5 *) (7 8 9))
(creerIndex 0 1 0 2)	=>	((0 0) (0 1) (0 2) (1 0) (1 1) (1 2))
(iemeObjet 2 '((2 4) (6 4) (7 4)))	=>	(6 4)

Nous allons créer une variable locale à la fonction de résolution de type T2D puis la modifier grâce à une fonctions que nous appellerons « remplirCase (i j objet tab) » qui sera chargé de remplir la case i, j de tab suivant les règles précédemment définies. Afin de parcourir le tableau en colonne nous avons développé la fonction creerIndex qui fournis une liste de couples d'index que nous pourrons passer à la fonction mapcar.

Voici donc la nouvelle solution :

```
; Acceder à un élément de coordonnées i j dans un tableau deux dimensions
(defun accesT2D (i j tab)
  (cond
    ((> i 0) (accesT2D (- i 1) j (cdr tab)))
    ((> j 0) (accesT2D 0 (- j 1) (list (cdr (car tab)))))
    (t (car (car tab)))
  )
)


; Création d'un tableau à deux dimensions de taille i j (fonction T2D)
(defun T2D (i j val)
   ; On appel la fonction colonne une unique fois de cette manière
  (tab i (colonne j val))
)


; Associée à T2D : ajouter i colonnes pour créer le tableau
(defun tab (i C)
  (if (= 0 i)
    ()
    (if (= 1 i)
      (list C)
      (cons C (tab (- i 1) C))
    )
  )
)


; Associée à T2D : créer une colonne de taille size remplie avec la valeur val
(defun colonne (size val)
  (if (= 0 size)
    ()
    (cons val (colonne (- size 1) val))
  )
)


; Modifier une valeur dans le tableau à deux dimensions
(defun change (val i j tab)
       (cond
            ;on atteint la ieme colonne
            ((> i 0) (cons (car tab) (change val (- i 1) j (cdr tab))))
```

```
            ;puis la jieme ligne
            ((> j 0) (cons (cons    (car (car tab))
                                    (car (change      val
                                                      0
                                                      (- j 1)
                                                      (list (cdr (car tab)))
                                          )
                                    )
                           )
                           (cdr tab)
                    )
            )

            ;on modifie la case
            (t (cons (car (list (cons val (cdr (car tab))))) (cdr tab)))
      )
)


; On fourni à la fonction un tableau à deux dimensions T2D et l'objet courant, la fonction se charge
;de trouver le sac optimal pour cette configuration en se basant sur les sac optimaux précédents.
(defun remplirCase (i j objet tab)
      (cond
            ; Si la case à modifier se trouve sur la premier ligne ou sur la première colonne on ne
            ; modifie rien car le sac optimal est le sac vide(tab est initialisé a (creerSac))
            ((eq 0 i) tab)
            ((eq 0 j) tab)

            ;si le poid de l'objet est supérieur au poid maximum du sac, autrement dit,
            ;si le sac ne peut pas contenir l'objet le sac optimal est donc le sac optimal avec le reste
            ;des objets et le meme poid maximum étant donné que l'on n'a pas
            ;ajouté le premier objet.
            ((> (poid objet) j) (change (accesT2D (- i 1) j tab) i j tab))

            ;si le sac peut contenir le premier objet il faut choisir le meilleur sac résultant
            ;des actions : prendre l'objet ou ne pas le prendre
            (t (change
                  (meilleurSac

                        ;si on ne prend pas l'objet on copie le sacOptimal
                        ;avec le reste des objets
                        (accesT2D (- i 1) j tab)

                        ;si on prend l'objet on prend le sac optimal avec le reste des objets
                        ;pouvant le contenir, c'est à dire avec un poid diminué de
                        ;(poid objet_courant) auquel on ajoute l'objet courant
                        (ajoutObjetSac
                              (accesT2D (- i 1) (- j (poid objet)) tab)
                              objet
                        )
                  )
                  i
                  j
                  tab
               )
            )
      )
)
```

```
;Permet de créer une liste d'index pour parcourir un T2D colonnes par colonnes.
(defun creerIndex (debut_i fin_i debut_j fin_j)
      (cond
            ((> debut_i fin_i) ())
            (t    (append (creerColonne debut_i debut_j fin_j)
                        (creerIndex (1+ debut_i) fin_i debut_j fin_j)
                  )
            )
      )
)


;Associée à créerIndex : permet de créer une colonne avec un certain index, allant de début à fin.
(defun creerColonne (index debut fin)
      (cond
            ((> debut fin) ())
            (t (cons (list index debut) (creerColonne index (1+ debut) fin)))
      )
)


; Atteindre le ieme objet de liste
(defun iemeObjet (i listeObjets)
      (cond
            ;le 0-ieme objet de la liste n'existe pas, on lui donne une valeure par défaut
            ((eq 0 i) '(0 0))
            ((eq 1 i) (car listeObjets))
            (t (iemeObjet (1- i) (cdr listeObjets)))
      )
)


;Deuxième version pour résoudre le problème
(defun sacOptimalb (poidMax objets)
      ;On crée une variable contenant un tableau initialisé à creerSac (sac vide)
      (set 'X (T2D (1+ (length objets)) (1+ poidMax) (creerSac)))

      ;On crée une variable contenant la liste des objets pour pouvoir la parcourir dans
      ;la fonction suivante
      (set 'O objets)

      ;On utilise mapcar sur la liste d'index pour appliquer à chaque case du tableau la fonction f
      (mapcar 'f (creerIndex 1 (length objets) 1 poidMax))

      ;On retourne la solution finale
      (accesT2D (length objets) poidMax X)
)



; Index contient une liste d'index : pour chaque couple d'index on met à jour X
(defun f (index)
  (set 'X   (remplirCase      (car index)
                              (cadr index)
                              (iemeObjet (car index) O)
                              X
            )
  )
)
```

Jeux d'essais

A partir de ce nouvel algorithme, nous pouvons tester les différents jeux d'essais précédents et comparer les temps d'exécution. Nous noterons ici uniquement les temps d'exécution et non les résultats, ceux-ci étant identiques aux premiers. Aussi nous commencerons au jeu d'essais #2, le #1 étant sensiblement identique.

Jeu d'essai #2 : Les 20 objets suivants :

(1 6), (1 2), (1 3), (2 1), (2 5), (2 7), (2 3), (2 4), (2 5), (3 3),
(3 5), (3 8), (4 4), (4 8), (4 5), (5 3), (5 6), (6 8), (7 5), (8 5)

Nous obtenons les résultats suivants :

Poids = 5 :	temps : 0.016 sec	(précédent : 0.016 sec)
Poids = 10 :	temps : 0.016 sec	(précédent : 0.083 sec)
Poids = 15 :	temps : 0.083 sec	(précédent : 0.550 sec)
Poids = 20 :	temps : 0.116 sec	(précédent : 1.900 sec)
Poids = 25 :	temps : 0.200 sec	(précédent : 4.500 sec)
Poids = 30 :	temps : 0.233 sec	(précédent : 8.183 sec)

Jeu d'essai #3 : Les 50 objets suivants :

(1 8), (1 1), (1 9), (1 1), (1 5), (2 7), (2 4), (2 4), (2 2), (2 8),
(3 9), (3 9), (3 5), (3 8), (3 3), (4 7), (4 4), (4 2), (4 5), (4 1),
(5 5), (5 8), (5 4), (5 8), (5 5), (6 3), (6 6), (6 8), (6 5), (6 5),
(7 5), (7 8), (7 4), (7 8), (7 5), (8 3), (8 6), (8 8), (8 5), (8 5),
(9 5), (9 8), (9 4), (9 8), (9 5), (10 3), (10 6), (10 8), (10 5), (10 5)

Nous obtenons les résultats suivants :

Poids = 10 :	temps : 0.200 sec	(précédent : 1.100 sec)
Poids = 15 :	temps : 0.383 sec	(précédent : 14.38 sec)
Poids = 20 :	temps : 0.667 sec	(précédent : 130.2 sec)

A partir de ce nouvel algorithme, nous pouvons désormais effectuer le jeu d'essais #3 pour des valeurs supérieures à 20, ce qui n'était pas envisageable avant.

Poids = 30 :	**(87 ((5 8) (5 8) (3 8) (3 9) (3 9) (2 8) (2 4) (2 4) (2 7) (1 5) (1 9) (1 8)))**	temps : 1.067 sec
Poids = 50 :	**(115 ((7 8) (6 8) (5 8) (5 8) (4 7) (3 8) (3 5) (3 9) (3 9) (2 8) (2 4) (2 4) (2 7) (1 5) (1 9) (1 8)))**	temps : 2.650 sec
Poids = 100 :	**(166 ((9 8) (8 8) (7 8) (7 8) (6 8) (6 6) (5 5) (5 8) (5 8) (5 5) (4 5) (4 4) (4 7) (3 8) (3 5) (3 9) (3 9) (2 8) (2 2) (2 4) (2 4) (2 7) (1 5) (1 9) (1 8)))**	temps : 7.417 sec
Poids = 150 :	**(207 ((10 8) (9 8) (9 8) (8 8) (8 6) (7 5) (7 8) (7 8) (6 5) (6 5) (6 8) (6 6) (5 5) (5 8) (5 8) (5 5) (4 5) (4 4) (4 7) (3 3) (3 8) (3 5) (3 9) (3 9) (2 8) (2 2) (2 4) (2 4) (2 7) (1 5) (1 1) (1 9) (1 8)))**	temps : 15.53 sec
Poids = 200 :	**(239 ((10 8) (10 6) (9 8) (9 8) (8 5) (8 5) (8 8) (8 6) (7 5) (7 8) (7 4) (7 8) (7 5) (6 5) (6 5) (6 8) (6 6) (5 5) (5 8) (5 4) (5 8) (5 5) (4 5) (4 2) (4 4) (4 7) (3 3) (3 8) (3 5) (3 9) (3 9) (2 8) (2 2) (2 4) (2 4) (2 7) (1 5) (1 1) (1 9) (1 1) (1 8)))**	temps : 30.90 sec

Les jeux d'essais #4 et #5 précédents (pas d'objet et poids maximum nuls) renvois toujours (0 NIL). Au vu de l'efficacité du programme, nous pouvons ajouter le jeu d'essais #6 suivant :

Jeu d'essai #6 : Les 100 objets suivants :

(1 2) (1 3) (1 9) (1 11) (1 13) (1 15) (1 17) (1 18) (11 10) (11 13) (2 3) (2 3) (2 5) (2 8) (2 12) (2 14) (2 17) (2 19) (12 2) (12 4)(3 2) (3 4) (3 5) (3 8) (3 9) (3 9) (3 9) (3 12) (13 7) (13 11) (4 1) (4 4) (4 4) (4 5) (4 5) (4 11) (4 14) (4 15) (14 8) (14 16) (5 2) (5 3) (5 4) (5 8) (5 14) (5 15) (5 15) (5 18) (15 4) (15 13) (6 3) (6 6) (6 8) (6 8) (6 10) (6 13) (6 15) (6 17) (16 6) (16 9) (7 5) (7 5) (7 7) (7 8) (7 11) (7 14) (7 17) (7 17) (17 2) (17 9) (8 3) (8 6) (8 8) (8 9) (8 10) (8 11) (8 11) (8 15) (18 9) (18 10) (9 1) (9 3) (9 6) (9 7) (9 7) (9 16) (9 16) (9 17) (19 4) (19 19) (10 3) (10 6) (10 8) (10 9) (10 12) (10 15) (10 16) (10 16) (20 6) (20 8)

Nous obtenons les résultats suivants :

Poids = 20 : **(168 ((4 15) (2 19) (2 17) (2 14) (2 12) (2 8) (1 18) (1 17) (1 15) (1 13) (1 11) (1 9)))** temps : 1.917 sec

Poids = 50 : **(265 ((6 17) (5 18) (5 15) (4 15) (4 14) (3 12) (3 9) (3 9) (2 19) (2 17) (2 14) (2 12) (2 8) (1 18) (1 17) (1 15) (1 13) (1 11) (1 9) (1 3)))** temps : 6.733 sec

Poids = 100 : **(392 ((7 17) (7 17) (7 14) (6 17) (6 15) (5 18) (5 15) (5 15) (5 14) (4 15) (4 14) (4 11) (3 12) (3 9) (3 9) (3 9) (3 8) (2 19) (2 17) (2 14) (2 12) (2 8) (2 5) (1 18) (1 17) (1 15) (1 13) (1 11) (1 9) (1 3) (1 2)))** temps : 22.34 sec

Nous constatons que par cette méthode les temps d'exécutions ont été nettement diminués.

Difficultés rencontrées et conclusion

Les difficultés que nous avons rencontrées résultent directement du niveau d'exigence que nous nous sommes fixé : donner un résultat rapide à tous les jeux d'essais. Nous avons donc du développer deux méthodes, qui bien qu'étant liées, implique de retravailler le problème à la base.

Nous avons aussi rencontré une difficulté d'ordre technique. En effet, le logiciel Emacs supporte très mal les variables de taille importantes. Il est donc impossible sous ce logiciel de travailler (avec le second algorithme) si nbObjets * poidMax > 1300 environ, car ceci implique une variable contenant plus de 1300 sacs. Ce problème ne s'est pas posé sous Clisp, logiciel avec lequel nous avons effectué tous nos jeux d'essais. En revanche, tous les jeux d'essais que l'on pouvait envisager de résoudre avec le premier algorithme n'ont posés aucun problème ni sous Emacs, ni sous Clisp.

Nous avons pu, par le biais de cet exercice, appréhender de nouvelles méthodes de résolution de problèmes, efficaces et couramment utilisés.