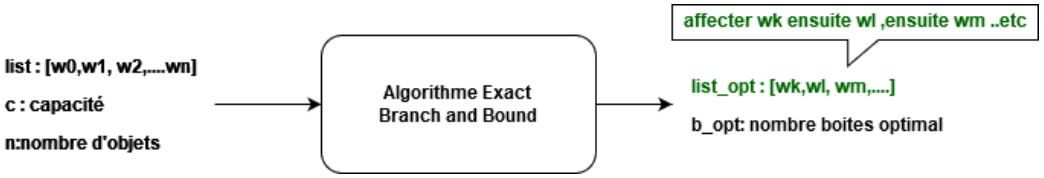
Dans cette partie, nous allons présenter la conception détaillée des méthodes exactes sur lesquelles notre choix d'implementation s'est porté:

1. Le branch and bound
2. Une version améliorée du branch and bound
3. La recherche exhaustive

Dans le but de montrer l'applicabilité de ces méthodes, comparer leurs performances et montrer leurs limites, nous effectuerons des tests empiriques et comparatifs sur des benchmarks d'un coté, et sur des instances générées par notre propre générateur d'instances d'un autre coté.

1 Branch and bound

- L'algorithme Branch-and-Bound (B &B) que nous avons implémenté tente de ranger un objet à la fois en fonction de l'ordre initial des objets.
- Au niveau j de l'arbre, B &B crée un noeuds fils pour chaque boite ouverte et range l'objet j dans cette boîte si c'est possible. il crée aussi un noeuds supplémentaire qui représente l'ouverture d'une nouvelle boîte, et il range l'objet j dans cette boîte.
- En pratique, au niveau 1 de l'arbre l'objet 1 est rangé dans la boîte 1 , au niveau 2 l'objet 2 est rangé dans la boîte 1 ou dans une nouvelle boîte 2 ,...etc
- A chaque noeud, on résout un sous problème de taille (n-k) du bin packing, où les k premiers objets ont déjà été emballés.
- L'opération du rangement d'un objet i au niveau k consiste à permuter entre les éléments list(K) et list(i). On va avoir comme sortie une liste d'objets ordonnées selon l'ordre de rangement, il suffit ensuite de remplir les boîtes par les objets dans leur nouvel ordre pour générer la solution (l'emplacement de chaque objet dans les boîtes)



1.1 Pseudocode

Soient:

- n : le nombre d'articles
- $list[0 \ldots n-1]$: la liste des articles en entrées
- opt_list: la liste ordonnée fournissant la solution optimale en sortie
- opt_cost: le nombre de boîtes optimal
- C:la capacité maximale d'une boîte.

L'algorithme proposé est une fonction récursive packBins ayant comme paramètres:

- k:l'ordre de l'élément à être rangé (le niveau dans l'arbre).
- $sumwt$:la somme des poids des éléments restants à être rangés
- $bcount$:la somme cumulée des boîtes déjà utilisées (depuis la racine jusqu'à ce nœud)
- $capa\_restante$:l'espace libre restant dans la boîte ouverte.

Algorithm 1 Branch & Bound

if $n = k$ **then**
  //Noeud feuille (n objets rangés)
  if *bcount* $<$ *opt_cost* **then**
    //Solution exacte obtenue par cette branche est meilleure que celle trouvée auparavant
    Mise à jor du coût optimal *opt_cost* = *bcount* .
    sauvegarde de la solution liste *opt_list* = *liste*.
  else
    Continuer le parcours (monter d'un niveau dans l'arbre).
  end if
else
  //L'ensemble des articles restants sont dans les positions list[k... n-1].

  //Chaque nœud fils i signifie qu'on a rangé le ième article parmi les articles restants (ayant la position k+i dans la liste list) à la position k.

  for chaque nœud fils i **do**
    Mettre l'article list[k+i] dans une boîte en permutant l'article[k+i] avec l'article[k] : permuter(k+i,k) .
    Incrémenter le nombre de boîtes utilisées (bcount) si on a ouvert une nouvelle boite.
    Mettre à jour la capacité restante (capa_restante).
    Mettre à jour la somme des volumes des articles restants à être rangés (Sumwt)
    Calculer l'évaluation du nœud fils courant (borne L1) : $Bound = bcount + \frac{(sumwt - capa\_restante)}{C}$
    Comparer l'évaluation du nœud avec la solution optimale courante :
    if *bcount* $\geq$ *opt_cost* **then**
      //Solution exacte obtenue par cette branche est meilleure que celle trouvée auparavant
      Le nœud est éliminé. Dans ce cas on re-permute pour revenir à l'état précédant (permute (k+i,k)).
      Sauvegarder la solution liste *opt_list* = *liste*
    else
      On exploite le nœud courant encore, en faisant un appel récursif à la fonction avec la valeur k+1 dans le premier paramètre,en utilisant les nouvelles valeurs des autres paramètres.
    end if
  end for
end if

1.2 Evaluation d'un noeud (Borne L1)

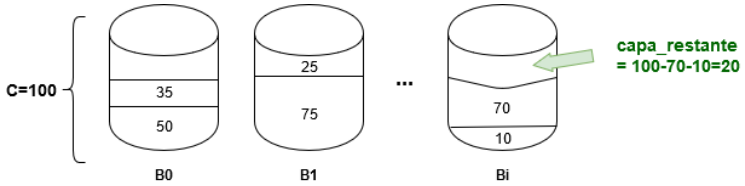
L'évaluation d'un nœud est calculée en sommant 2 parties, le nombre de boîtes déjà utilisées bcount et une estimation du nombre de boîtes qu'on va ouvrir encore pour contenir les objets restants. Cette estimation est obtenue en divisant la somme des poids restants sumwt sur la capacité d'une boîte. On soustrait de la somme des poids restants, l'espace vide restant dans la dernière boîte ouverte, car ce dernier peut contenir des objets. On obtient ainsi la formule suivante :

$$bound = bcount + \frac{capa_{restante} - sumwt}{C}$$

Nombre de boîtes ouvertes

Estimation du nombre de boites à ouvrir encore

exemple 1: List= 10,50,25,80,70,75,35,70 ; C = 100



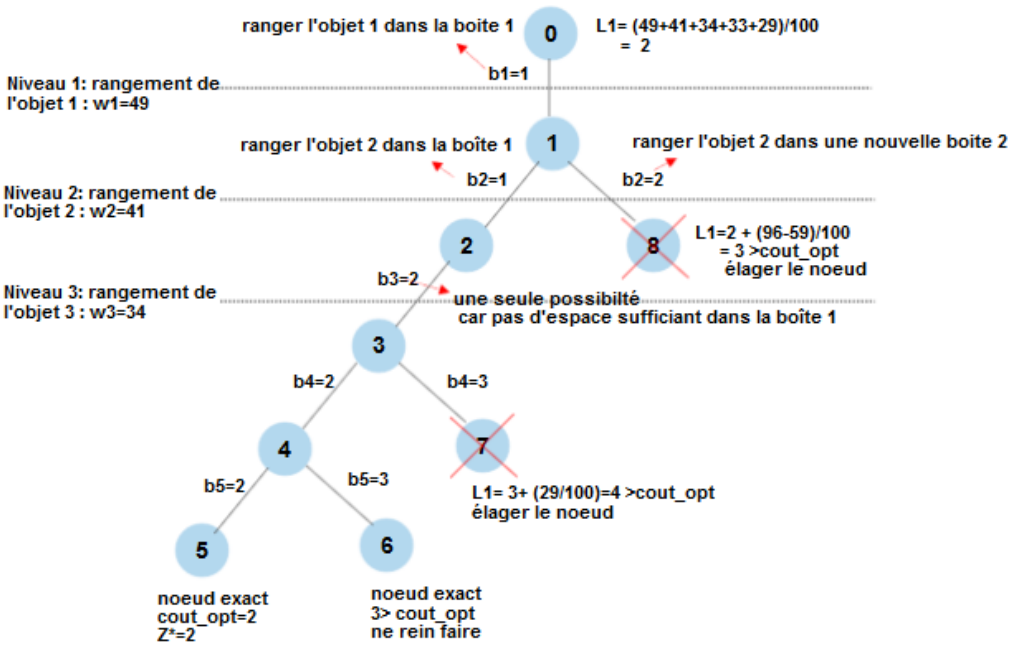
bound =((80+70)-20)/100)=2

exemple 2: n= 5 ; W_j=49,41,34,33,29 ; c=100 on pose : b_j= le numéro de boîte qui contient l'objet j.

2 Branch and bound amélioré

Une version améliorée de l'algorithme Branch and Bound présentée ci-dessus. L'amélioration est faite en 2 étapes:

1. Utilisation de l'heuristique WFD (Worst Fit Decreasing) pour initialiser la solution optimale.
2. Changement de la borne L1 utilisée par une autre borne plus puissante appelée L2.



2.1 Evaluation d'un noeud (Borne L2)

Il a été prouvé que la borne L1 n'est efficace que quand les poids des objets sont petits, c'est à dire qu'on peut mettre plusieurs objets dans la même boîte. Si ce n'est pas le cas, et que les objets ont de grands poids (proches de C) , cette borne n'aura aucun effet et l'algorithme fera une recherche exhaustive. C'est pour cela que la borne L2 à été proposée par Martello et Toth ,pour remédier à ce problème. On rappelle la formule de la borne L2, qui a été déjà présentée dans l'état de l'art:

rappel Soit α un entier tels que :

$$0 \leq \alpha \leq C/2$$

On definie des classes d'articles suivantes:

$$C_1 = \{a_i, \quad C - \alpha < v_i\}$$

$$C_2 = \{a_i, \quad C/2 < v_i \leq C - \alpha\}$$

$$C_3 = \{a_i, \quad \alpha < v_i \leq C/2\}$$

$BI(I)$ est donnée par la formule suivante:

$$BI(I) = max\{L(\alpha), \quad 0 \leq \alpha \leq C/2\}$$

Avec

$$L(\alpha) = |C_1| + |C_2| + max(0, \lceil \frac{\sum_{j \in C_3} v_j - (|C_2| * C - \sum_{j \in C_2} v_j)}{C} \rceil)$$

Explication de la formule : Etant donnée que les objets des classes C_1 et C_2 ont un poids supérieur à $C/2$ chacun d'eux sera placé dans une boîte séparée pour le contenir, donc $|C_1| + |C_2|$boîtes sont utilisées quelque soit la solution. De plus,aucun objet de l'ensemble C_3 ne peut être rangé dans une boîte contenant un objet de C_1 (à cause de la contrainte de capacité). La capacité résiduelle (espace libre) des $|C_2|$ boîtes est de : $C^* = |C_2| * c - \sum_{j \in C_2} w_j$ Donc dans le meilleur des cas, cette capacité résiduelle va être remplie par les objets de C_3, et dans ce cas le nombre de nouvelles boîtes qu'on doit ouvrir est de : $\frac{\sum_{j \in C_3 - C^*} w_j}{c}$ (cette dernière formule utilise le même principe que la borne L1).

2.2 Pseudocode

Algorithm 2 Branch and bound amélioré

Appliquer WFD sur l'instance pour initialiser le coût optimal: cout_opt=WFD(problème)
Appliquer l'algorithme Branch and Bound sur le problème en utilisant la borne L2

3 Recherche exhaustive

Dans cette 3ème solution, on a implémenté une recherche exhaustive, qui consiste à parcourir l'ensemble des nœuds et leurs fils, sans aucun élagage de nœuds. Donc on aura le même algorithme que celui du branch and bound, en supprimant l'étape de l'évaluation du nœud pour décider de son élagage.