université de BORDEAUX

Collège Sciences et technologies

Problème de Sac à dos

Ghita BENCHEIKH

Recherche opérationnelle et aide à la décision

Année universitaire 2015/2016

Contents

1 Introduction

De nombreux problèmes d'optimisation combinatoire issues des applications réelles se modélisent comme des problèmes linéaires en nombres entiers ou des problèmes linéaires mixtes. La plupart de ces problèmes appartiennent à la classe NP ce qui complique leur résolution de manière exacte.
Le problème de sac à dos est un exemple classique de problème de la classe NP. De part ses nombreuses variantes, il est au cœur de nombreux problèmes d'optimisation, et bien qu'il soit étudié dans plusieurs décennies, il reste un sujet actif dans le domaine. L'intérêt du problème n'est pas que théorique. En effet, de nombreux cas pratiques se formulent sous la forme d'un problème de sac à dos ou similaire, c'est un modèle classiquement utilisé dans la gestion du portefeuilles, le transport et bien d'autres domaines.
Dans ce travail, nous présentons le problème du sac à dos ainsi sa formulation mathématique. Nous nous intéressons en particulier à des méthodes de résolution classiques, comme la programmation dynamique et la Branch and Bound. Puis nous étudions le problème du plus long chemin dans un graphe, qui constitue une autre approche qui semble particulièrement intéressante.

2 Problème du sac à dos (KP)

Le problème du sac à dos (Knapsack Problem) est un problème classique d'optimisation combinatoire appartenent à la classse NP-complet [théorème1]. Il tire son nom d'une situation à laquelle est souvent confronté un randonneur au moment de remplir son sac à dos : celui-ci dispose de n objets, caractérisés chacun par une utilité (profit) p_i et un poids (weight) w_i, compte tenu d'un poids W fixé à l'avance à ne pas dépasser, le randonneur cherche à déterminer les objets qu'il va emporter de façon à maximiser l'utilité totale des objets emportés sans dépasser la limite prescrite W pour le poids.
Formellement, un problème de sac à dos binaire est un problème de la forme suivante :

$$(P)\begin{cases} max \sum_{i=1}^{n} \mathrm{p}_i x_i \\ sc. \\ \qquad \sum_{i=1}^{n} \mathrm{w}_i x_i \leq \mathrm{W} \\ \qquad \mathrm{x}_i \in \{0,1\} \qquad \forall i = 1,...,n \end{cases}$$

Par rapport au problème du randonneur, on interprétera une valeur égale à 1 pour une variable x_i comme le choix d'emporter l'objet i, et une valeur 0 comme le choix de ne pas emporter l'objet i. On suppose que les relations suivantes sont satisfaites (sans quoi le problème peut être simplifié):

- $w_i > 0 \quad et \quad p_i > 0 \forall i \in \{1,...,n\}$
- $w_i < W \quad \forall i \in \{1,...,n\}$
- $\sum_{i=1}^{n} w_i > W$

3 Méthodes de résolution

Trouver une solution optimale dans un ensemble discret et fini est un problème facile en théorie, il suffit d'essayer toutes les solutions, et de comparer leurs qualités pour voir la meilleure. Cependant en pratique, l'énumération de toutes les solutions peut prendre trop de temps. Or le temps de recherche de la solution optimale est un facteur très important et c'est à cause de lui que les problèmes d'optimisation combinatoire sont réputés si difficiles. Pour cela, de nombreuses méthodes de résolution ont été développées en recherche opérationnelle. Ces méthodes peuvent être classées sommairement en deux grandes catégories : les méthodes exactes(complètes) et les méthodes approchées(incomplètes).
Avant d'aborder les différentes méthodes de résolution du problème, nous introduisons quelques définitions et notions sur la complexité.

3.1 Notions sur la complexité

La théorie de la complexité a pour but d'apporter des informations sur la difficulté théorique d'un problème à résoudre. Elle permet de classer "du point de vue mathématique" les problèmes selon leur difficulté, cette difficulté est mesurée par la complexité du meilleur algorithme résolvant le problème de façon optimale. Généralement la performance d'un algorithme est évaluée sur la base du temps de calcul (nombre d'opérations élémentaires nécessaires à l'exécution de l'algorithme pour une instance de taille n) et de la taille mémoire (taille nécessaire pour stocker les différentes structures de données) requise pour résoudre le problème. On dit souvent que le temps d'exécution dépend de la taille de l'instance du problème. La complexité algorithmique est un concept fondamental qui permet de mesurer les performances d'un algorithme et de le comparer avec d'autres algorithmes réalisant les mêmes fonctionnalités.
Un algorithme en temps polynomial est un algorithme dont le temps de la complexité est en $o(p(n))$ où p est une fonction polynomiale et n la taille de l'instance. Si k est le plus exposant de ce polynôme en n, le problème correspondant est dit résoluble en $o(n^k)$ et appartient à la classe P, (exemple : connexité dans un graphe).
La classe NP contient les problèmes de décision[1] qui admettent un algorithme polynomial capable de tester la validiter d'une solution du problème indépendamment du temps nécessaire à la recherche de la solution.
Il est clair que $P \subseteq NP$, mais peut-on dire que $P = NP$?
Cette question à été posé en 1970 et n'a toujours pas de réponse, la théorie de la complexité ne permet pas encore d'affirmer que ces deux classes sont différentes, cela signifie que certains problèmes "difficiles" n'ont pu être résolus jusqu'à présent par des algorithmes polynomial.
Un problème est dit NP-complet s'il appartient à la NP et si tous les problèmes de NP se réduisent polynomialement en ce problème. Intuitivement, si on trouve un algorithme pour un NP-complet on trouve alors automatiquement une résolution polynomiale de tous les problèmes de la classe NP.

Théorème1: le problème de sac à dos est NP-Complet.

[1] les problèmes de désions sont les problèmes pour lequels on cherche à répondre pour "oui" ou "non" à une question donnée

Preuve : La façon la plus naturelle pour déterminer qu'un problème est NP-complet est de réduire un autre problème complet au problème étudié et c'est ainsi que nous procéderons pour montrer que le problème du sac à dos est NP-complet.
D'abord il est facile de voir qu'il appartient à la classe NP. On devine un sous-ensemble en temps linéaire, puis on effectue la somme des éléments de ce sous-ensemble et on la compare à W, ceci ce fait en temps polynommial et démontre que le problème dans NP.
démontrons qu'on peut réduire le problème de la somme de sous ensemble à un problème du sac à dos sous sa forme décisionnelle qui à son tour est lié au problème du sac à dos.

- Le problème de la somme de sous ensemble est défini comme suit : On a en entrée n+1 nombre $\{a_1, a_2, ..., a_n, \mathrm{T}\}$ et on cherche à déterminer s'il existe un sous ensemble $I \subseteq \{1, ..., n\}$ tel que $\sum_{i \in I} a_i = T$

- Problème de sac à dos sous la forme décisionnelle : existe-t-il une valeur des x_i pour lequel $\sum_{i=1}^{n} p_i x_i \geq k$?
Si on prend :

$$\begin{array}{ccccc} a_i & = & w_i & = & p_i \\ W & = & T & = & k \end{array}$$

alors : (3) $\sum_{i \in S} \mathrm{a}_i = \mathrm{T} \Longleftrightarrow \begin{cases} \sum_{i \in S} \mathrm{w}_i \leq \mathrm{W} & \Longleftrightarrow \sum_{i \in S} \mathrm{a}_i \leq \mathrm{T} \quad (1) \\ \sum_{i \in S} \mathrm{p}_i \geq \mathrm{k} & \Longleftrightarrow \sum_{i \in S} \mathrm{a}_i \leq \mathrm{T} \quad (2) \end{cases}$

Supposons qu'on a une réponse de "oui" au nouveau problème, cela veut dire qu'on peut trouver un ensemble qui satisfait les deux inégalités (1) et (2), il est immédiat qu'un tel ensemble fournit le nombre recherché. Inversement, si la réponse est "non", automatiquement la réponse du problème original est aussi négative.
Finalement, nous montrons que les deux versions du problème de sac à dos sont équivalentes. En effet, il y a un lien entre la version "décision" et la version "optimisation" du problème dans la mesure où s'il existe un algorithme polynomial qui résout la version "décision", alors on peut trouver la valeur maximale pour le problème d'optimisation de manière polynomial en appliquant itérativement cet algorithme tout en augmentant la valeur de k. D'autre part, si un algorithme trouve la valeur optimale du problème d'optimisation en un temps polynomial, alors un problème de décision peut être résolu en temps polynomial en comparant la valeur de la solution sortie par cet algorithme avec la valeur k. Ainsi les deux versions du problème sont de difficulté similaire.

3.2 Calcul de la borne initiale

Une méthode parfois utilisé pour déterminer une borne initial pour des problèmes de programmation linéaire entiers consiste à relâcher les contraintes d'intégrité, puis à résoudre le problème à variables réelles ainsi obtenu, enfin à arrondir de manière appropriée les valeurs fourni par cette résolution de façon à construire une solution en nombres entiers respectant toutes les contraintes.

Calculer des bornes supérieures ou inférieures permet d'encadrer la valeur de la solution optimale pour les problèmes que l'on tente de résoudre. Ensuite, elles sont utilisées pour le développement de méthodes de résolution exacte s'appuyant sur des procédures d'énumération implicite.
Pour notre problème, la relaxation des contraintes de binarité s'obtient en remplaçant les contraintes $x_i \in \{0,1\}$ pour 1 ≤i≤n par les contraintes $x_i \in [0,1]$. Le problème relâcher (RP) associé à (P) s'écrit alors :

$$(RP)\begin{cases} max \sum_{i=1}^{n} p_i x_i \\ sc. \\ \quad \sum_{i=1}^{n} \mathrm{w}_i x_i \leq \mathrm{W} \\ \quad \mathrm{x}_i \in [0,1] \qquad \forall i = 1, ..., n \end{cases}$$

la résolution du problème (RP) consiste tout d'abord à ordonner les objets selon un ordre décroissant du rapport profit par poids, c'est à dire :

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_n}{w_n}$$

puis à remplir le sac objet après objet et de proche en proche jusqu'à sa saturation. Ensuite, on repère le premier objet ne pouvant pas être mis en totalité dans le sac . Il s'agit d'un élément d'indice j tel que $\sum_{i=1}^{j-1} w_i \leq c \leq \sum_{i=j+1}^{n} w_i$ (On l'appelle l'élément critique).
La solution de (RP) se représente alors comme suit :

$$(x_i^*) = \begin{cases} 1 & i = 1, ..., j-1 \\ \frac{W - \sum_{i=1}^{j-1} w_i}{w_j} & i = j \\ 0 & i = j+1, ..., n \end{cases}$$

Nous pouvons écrire cet algorithme comme suit :

Algorithm 1: Greedy algorithm for the LP relaxed

Data: p_i, w_i, W, n
Result: Z^*, x_i
initialization : $Z^* = 0$, $W^* = 0$ et $x_i = 0$ $\forall i \in \{1,...,\mathrm{n}\}$
Sort item s.t : $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_n}{w_n}$
while $W^* \neq W$ **do**
  if $W^* + w_i > W$ **then**
    $x_i \longleftarrow \frac{W - W^*}{w_i}$
    $W^* \longleftarrow W$
    $Z^* \longleftarrow Z^* + x_i pi$
  else
    $x_i \longleftarrow 1$
    $W^* \longleftarrow W^* + w_i$
    $Z^* \longleftarrow Z^* + p_i$
    $i \longleftarrow i + 1$
  end
end

Théorème2 : cet algorithme donne une solution optimale du problème de sac à dos dans le cas continu.

preuve : Nous prouvons l'optimalité de cette solution, en la comparant avec toute les autres solution réalisable $(y_1, ..., y_n)$ du problème continu.
Soient k et l$\in \{1, ..., n\}$ tel que k correspond au premier indice qui vérifie $y_k < 1$ et l le plus petit indice tel que $k < l$ et $y_l > 0$, remarquer qu'un tel indice existe sûrement, car sinon cette solution $(y_1, ..., y_n)$ sera la même que la solution $(x_1, ..., x_n)$ trouvé par l'algorithme glouton.
Soit $\epsilon = max\{w_k(1 - y_k), w_l y_l\}$.
Si on augmante la valeur de y_k par $\frac{\epsilon}{w_k}$ et qu'on diminue la valeur de y_l par $\frac{\epsilon}{w_k}$ sans toucher les autres variables. Il est clair que ce changement mène à une nouvelle solution de qualité qui n'est sûrement moins bonne que l'ancienne.

3.3 Méthodes approchées

les méthodes approchées constituent une alternative trés intéressante pour traiter les problèmes d'optimisation de grande taille si l'optimalité n'est pas primordiale.
Parmi les méthodes heuristiques, on peut citer la méthode dite gloutonne. Un algorithme glouton construit une solution de manière incrémentale, en faisant à chaque pas un choix maximisant la fonction objectif.
L'algorithme glouton que nous présentons ici (algorithm2) consiste donc à sélectionner à chaque étape un élément selon l'ordre précédemment défini. Si l'élément est admissible, c'est-à-dire si son poids ne dépasse pas la capacité restante aprés fixation des autres éléments, alors, il est mis dans le sac sinon, on sélectionne l'élément qui se situe juste après et qui peut être admissible et ainsi de suite de proche en proche jusqu'à épuisement de tous les objets pouvant être mis dans le sac.

Algorithm 2: greedy approximation for the 0/1 knapsack problem

Data: p_i, w_i, W, n
Result: Z^*, x_i
initialization : $Z^* = 0$, $W^* = 0$ et $x_i = 0$ $\forall i \in$ {1,...,n}
Sort items s.t: $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_n}{w_n}$
for $i = 1$ *to* n **do**
  if $W^* + w_i \leq W$ **then**
    $x_i \longleftarrow 1$
    $W^* \longleftarrow W^* + w_i$
    $Z^* \longleftarrow Z^* + p_i$
  else
end

Théorèm3 : cet algorithme glouton est $\frac{1}{2}$ approximation.

preuve : Soit x^* la solution optimal du problème de sac à dos. Puisque toute solution d'un problème est aussi réalisable pour son problème relaxé, alors:

$Val(x^*) \leq Val(Z^*)$, avec Z^* la solution optimal du problème (RP)

On sait que Z^* est de la forme $(1, ..., 1, \alpha, 0, .., 0)$ avec $\alpha \in [0, 1]$
On pose $x = (1, .., 1, 0, 0, ...0)$ et $y = (1, .., 1, 1, 0, ..0)$
$Val(x^*) \leq Val(Z^*) = Val(x^*) + \alpha p_k \leq Val(x) + Val(y) \leq 2max\{val(x), val(y)\}$

3.4 Méthodes exactes

Une méthode de résolution naïve consisterait à énumérer toutes les solutions réalisables et de ne garder que les meilleurs suivant le critère choisi. En pratique, cette méthode n'est utilisable que pour des problèmes de très petite taille. Cependant, les méthodes exactes utilisées dans la pratique recouvrent à des stratégies bien plus sophestiquées. Par exemple des stratégies basées sur des calculs de bornes pour éviter l'exploration des régions ne contenant pas de meilleur solution que celle déjà trouvée, d'autres stratégies, utilisent l'idée de réduire le problème en un sous problème dont il est facile de le résoudre.

3.4.1 Séparation et évaluation

L'algorithme de séparation et évaluation, plus connu sous son appel anglaise Branch and Bound (BB), repose sur une méthode arborescente de recherche d'une solution optimale, en représentant les états solutions par un arbre d'états avec des nœuds et des feuilles.
Le branch and bound est basé sur deux axes principaux la séparation et l'évaluation.

1. La séparation : Le principe de séparation conditionne la mise en œuvre de la méthode, c'est lui qui définit la façon de construire les sommets S de A. En général, il est issu d'un principe d'énumération exhaustive permettant de ne pas oublier de solution.
   Pour construire l'arborescence de recherche A, on sépare les sommets de A, tant que cela est nécessaire, en commençant par la racine, puis en effectue la séparation selon les valeurs possibles de x_1 puis de x_2, etc.

2. L'évaluation : permet de réduire l'éspace de recherche en éliminant quelques sous ensembles qui ne contiennent pas la solution optimale. Ceci en éliminant tous les choix qui sont incompatibles avec notre problème, c'est à dire qui ne respectent pas la contrainte de sad à dos. aussi une remarque portant sur les feuilles permet d'éliminer quelques éléments inutile de l'arbre de recherche, puisqu'il s'agit du dernier niveau associé à la dernière variable selon effectuer la séparation, il est clair qu'il est inutile d'engendrer la branche associé à $x_n = 0$ quand la branche sœur associé à $x_n = 1$ est présente. On peut encore éliminer toute branche ne peuvant pas conduire à une solution meilleur en utilisant le principe des bornes.

   - Borne inférieur : un minorant de la fonction objectif, que n'importe quelle solution réalisable le fournit.
   - Borne supérieur : valeur maximal de la fonction objectif, c'est à dire quue la solution optimale est nécessairement d'une valeur plus petit. On peut facilement obtenir une telle borne on prenons la somme des profits de tous les objets,

mais il existe une meilleur borne, telle que la solution optimale du problème (RP), puisque cette relaxation à pour effet, d'élargir l'ensemble sur lequel on cherche à maximizer Z.

On initialise notre borne inférieur par le résultat obtenu par l'algorithme1 vu précédement. Pendant la recherche à chaque nœud on calcul la borne supérieur de ce nœud, si jamais à un nœud donné i on a $BI \geq BS(i)$ alors il est inutile d'explorer les nœuds descendants de celui-ci. On dit qu'on coupe l'arbre de recherche. Et on actualise automatiquement la valeur de la borne inférieur lorsqu'on arrive à une feuille (solution réalisable) qui possède une valeur plus grande.

Remarque : cet algorithme est de complexité $o(2^n)$.

preuve : Montrons que le nombre maximum de nœuds au niveau i d'un arbre binaire est 2^i.

- Pour $i = 0$, la racine est le seul nœud au niveau 0, par conséquent, le nombre maximum de nœud au niveau $i = 0$ est $2^0 = 1$. La relation est vérifiée.

- Hypothèse de récurence : Supposons que la relation est vraie jusqu'à $i - 1$, c'est à dire que le nombre maximum de nœuds au niveau j est $2^j \forall j \in 1, .., i - 1$.

Comme chaque nœud dans un arbre binaire possède au plus deux descendants, alors le nombre de maximum de nœuds au niveau i est 2 fois le nombre de nœuds au niveau $i - 1$, c'est à dire $2 \times 2^{i-1} = 2^i$

L'algorithme de Branch and Bound [1] $s'écritcommesuit$:

Algorithm 3: branch and bound algorithm for the knapsack problem

Data: p_i, w_i, W, n
Result: Z^*, x_i^*
Sort items s.t: $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_n}{w_n}$
initialization : $Z^* = 0$, $x_i = 0 \ \forall i \in \{1,...,n\}$
x = solution given by algorithm1
$stage(root) = 0$
$weight(root) = 0$
$value(root) = 0$
$arbre \longleftarrow root$
while $arbre \neq \oslash$ **do**
  $node \longleftarrow$arbre.top()
  $arbre \longleftarrow$arbre / {node}
  if *node is a leaf* **then**
    if $value(node) > value(x)$ **then**
      $x \longleftarrow node$
    end
  else
    $i \longleftarrow stage(node)$
    if $value(x) < value(node) + UB(i, W - weight(node))$ **then**
      this means that this under tree can contains the optimal solution
      if *weight(node)* + $w_{i+1} \leq W$**then**
        $stage(new1) \longleftarrow i + 1$
        $weight(new1) \longleftarrow weight(node) + w_{i+1}$
        $value(new1) \longleftarrow value(node) + p_{i+1}$
      end
      if $stage(node) < n - 1$ *or* ($stage(node) = n - 1$ *and we have added the last object* **then**
        $stage(new2) \longleftarrow i + 1$
        $weight(new2) \longleftarrow weight(node)$
        $value(new2) \longleftarrow value(node)$
      end
    end
  end
end

3.4.2 Programmation dynamique

La programmation dynamique est une approche générale qui se retrouve comme technique de résolution en recherche opérationnelle. Cette technique peut être appliquée quand les sous problèmes ont des sous sous problème en commun, ce qui est le cas pour le problème de sac à dos.

On utilise une table de programmation dynamique V à deux dimensions telle que $V^k(b)$ désigne le gain maximum généré par le choix des k premiers objets dont la somme des poids ne dépasse pas W, avec k= 1,...,n et $b = 0, ..., W$ d'où la complixité de l'algorithme

∘(nW).
$Si\mathrm{V}^{k-1}(b)$ est connu pour toutes les capacités $b = 0, ..., W$, alors la valeur de $V^k(b)$ est donné par la formule suivante :

$$V^k(b) = \begin{cases} \mathrm{V}^{k-1}(b) \text{ si } b < w_k \\ \max\{V^{k-1}(b), V^{k-1}(b - w_k) + p_k\} \end{cases}$$

Autrement dit, $V(k, b)$ est obtenu soit sans prendre l'objet k, il vaut donc $V(k-1, b)$, soit en ajoutant k au meilleur chargement d'objets pris dans $\{1, ..., k-1\}$ de poids au plus $b - w_k$

Il suffit donc de trouver la solution optimale pour les $i-1$ premiers $V^{k-1}(b)$, et alors résoudre le problème revient à trouver la valeur $V^n(W)$.le programme linéaire se formule de la façon suivante :

$$(DP) : \begin{cases} \max \mathrm{V}^n(W) \\ sc. \\ \quad \mathrm{V}^k(b) \geq \mathrm{V}^{k-1}(b - w_k) + p_k \ \forall \mathrm{b} \in \{0, ..., W\}, k \in \{1, ..., n\}, b < w_k \\ \quad \mathrm{V}^k(b) \geq \mathrm{V}^{k-1}(b) \ \forall \mathrm{b} \in \{0,...,\mathrm{W}\}, k \in \{1,...,\mathrm{n}\} \\ \quad \mathrm{V}^k(b) \geq 0 \ \forall \mathrm{b} \in \{0,...,\mathrm{W}\}, k \in \{1,...,\mathrm{n}\} \end{cases}$$

L'algorithme de programmation dynamique[2] $estlesuivant.$

Algorithm 4: Dynamic Programming for 0-1 KP

Data: k, b, p_i, w_i
Result: Z
if $k = 0$ *or* $b = 0$ **then**
  return 0
end
if $w_k > b$ **then**
  return $V^{k-1}(b)$
end
return $\max\{V^{k-1}(b - w_k) + p_k, V^{k-1}(b)\}$

3.4.3 Core d'un problème de sac à dos

Le principe du noyau ou core en anglais, s'appuie sur la réduction du problème initial en un noyau d'objets dont lequel il est difficile de savoir s'ils seront présents dans la solution optimale ou pas, alors que tous les autres ont auparavant une valeur fixe dont on connait. On peut ainsi donner une représentation du problème, cette définition ci-dessous est basé sur la connaissance d'une solution optimale du problème relaxé(RP).

Nous rappelons que le résultat du problème relaxé est composé de trois parties, la première contient tous les variables qui sont fixées à 1, la deuxième correspond au plus à un objet qu'ont l'appelle objet critique ou split item qui désigne la variable fractionnaire, et finalement la troisième partie contient ceux qui sont fixées à 0.

Nous désignons par s l'indice de l'objet critique, ou de base du problème qui est donné par l'inégalité suivante :

$$\sum_{j=1}^{s-1} w_j \leq W \leq \sum_{j=1}^{s} w_j$$

Le core correspond alors au sous ensemble d'articles avoisinant l'article de base d'indice s.
Posons : $a = min\{j : x_j^* = 0\}$ et $b = max\{j : x_j^* = 1\}$
L'ensemble $C = \{a, ..., b\}$ représente le noyau du problème, si on pose

$$\tilde{P} = \sum_{j=1}^{a-1} p_j \text{ et } \tilde{W} = \sum_{j=1}^{a-1} w_j$$

alors le problème du knapsack sur le noyau est formulé ainsi :

$$(KPC) \begin{cases} max \sum_{i \in C} \mathrm{p}_i x_i + \tilde{P} \\ sc. \\ \qquad \sum_{i \in C} \mathrm{w}_i x_i \leq \mathrm{W}\text{-}\tilde{W} \\ \qquad \mathrm{x}_i \in \{0, 1\} \qquad \forall i \in C \end{cases}$$

En pratique, la taille du noyau est petite par rapport à n. Par conséquent, si les valeurs de a et b sont connues a priori, le problème initial peut facilement être résolu en posant $x_j^* = 1$ pour $j = 1, ..., a-1$ et x_j^* pour $j = b+1, ..., n$ et en résolvant tout simplement le knapsack sur le noyau par la méthode de branch and bound ou la programmation dynamique.
Nous présenterons ci-dessous un algorithme de résolution exacte du problème de sac à dos en appliquant la programmation dynamique sur le concept du core.
Soit $V^{(a,b)}(d)$ le gain maximum qu'on peut obtenir avec les objets i du core $C = \{a, ..., b\}$, sans dépasser la capacité d, sachant que les objets d'indice $i < a$ sont fixés à 1 et les objets $i > b$ sont fixés à 0.
Si $V^{(a+1,b)}(d)$ et $V^{(a,b-1)}(d)$ sont connu pour toutes les capacités $d = 0, ..., 2W$, alors on peut facilement calculer la valeur de $V^{(a,b)}(d)$, puisqu'il suffit de tester tous les cas possibles des deux sous problèmes $V^{(a+1,b)}(d)$ et $V^{(a,b-1)}(d)$, soit on gardant l'objet a dans le sac ou pas, et la possiblité d'ajouter l'objet b dans le sac si sa capacité réstante le permet, et de prendre celui qui apporte le plus de gain.
Si on décide de garder l'objet a dans le sac, alors on se retrouve avec un sous-problème de même capacité et de même profit, par contre si on décide de l'enlever, alors on augmente la capacité du sac par w_a unités tout en diminuant sa valure de p_a unités, tandis que pour l'objet b si on l'ajoute dans le sac, ceci permet d'augmenter sa valeur, mais en ne laissant que $d - w_b$ d'esapce libre pour les autres objets du core qui ne sont pas encore traité.
Notons que C=s au départ.

L'algorithme[3] $s'écritalorscommesuit$:

Algorithm 5: Dynamic Programming using Core for 0-1 KP

Data: a, b, d, s, p_i, w_i
Result: Z,x^*
if $a = s$ *et* $b =$ *s-1* **then**
  $x_i \longleftarrow 1$ ∀i∈ {1,..,s-1} $x_i \longleftarrow 0$ ∀i∈ {s,..,n}
  return $\sum_{i=1}^{s-1} p_i$
end
if $a \neq s$ *and* $b \neq s-1$ **then**
  if $w_b <= d$ **then**
    $Z \longleftarrow$max { $V^{(a+1,b)}(d), V^{(a+1,b)}(d+w_a) - p_a, V^{(a,b-1)}(d), V^{(a,b-1)}(d-w_b) + p_b$}
  end
  $Z \longleftarrow$max { $V^{(a+1,b)}(d), V^{(a+1,b)}(d+w_a) - p_a, V^{(a,b-1)}(d)$}
  $i \longleftarrow$ *index of the maximum element*
  if i == 2 **then** $x_a \longleftarrow 0$
  if i == 4 **then** $x_b \longleftarrow 1$
  return Z
end

3.4.4 Modélisation via la théorie des graphes

La théorie des graphes étudie des structures finis (en forme de réseaux) caratérisées par une composition en deux ensembles d'objets :

- l'ensemble des sommets ou nœuds (noté V).
- l'ensemble des arcs ou arêtes E qui symbolisent une connexion, une liaison entre les sommets.

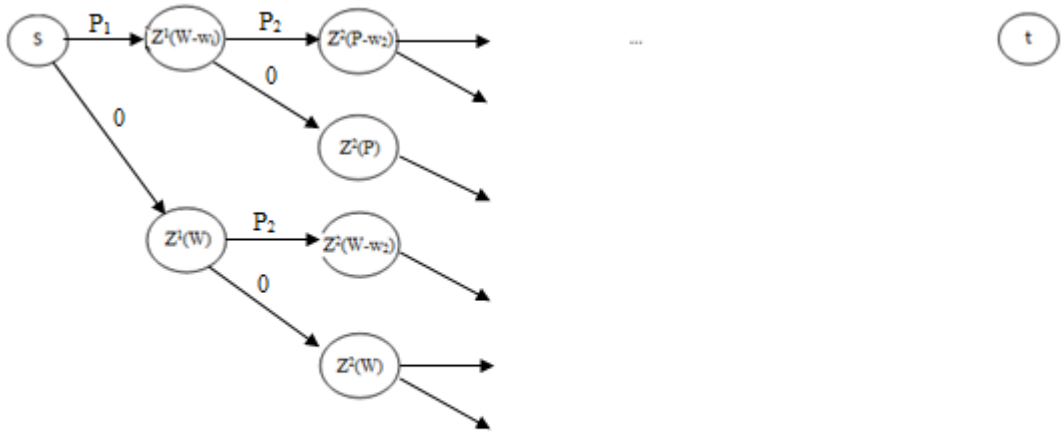
Dans la modélisation, il s'agit de faire aussi attention si les connexions entre les sommets sont en réalité des arêtes qui symbolisent une relation symétrique ou au contraire, des arcs pour lequels le sens est particulièrement important. Nous pouvons alors distinguer deux familles:

- les graphes orientés
- les graphes non orientés

Le problème du sac à dos peut se représenter par un graphe orienté et sans circuit, où nous associons un sommet à chaque état du sac (son poids), nous rajoutons deux sommets artificiels : le sommet source s (de départ) et le sommet puits t (pour la fin). Un arc qui relie un sommet A vers un sommet B signifie qu'on a ajouté un objet, ce qui fait que le poids du sac passe de l'état A à l'état B. Ainsi la valeur de l'arc peut être vue comme la valeur de l'objet ajouté.

Classiquement, ce graphe correspond à une instance du problème de sac à dos est obtenu en associant un sommet à chaque état $V^i(d)$ de la programmation dynamique,

Figure 1: Graphe associé au problème de sac à dos



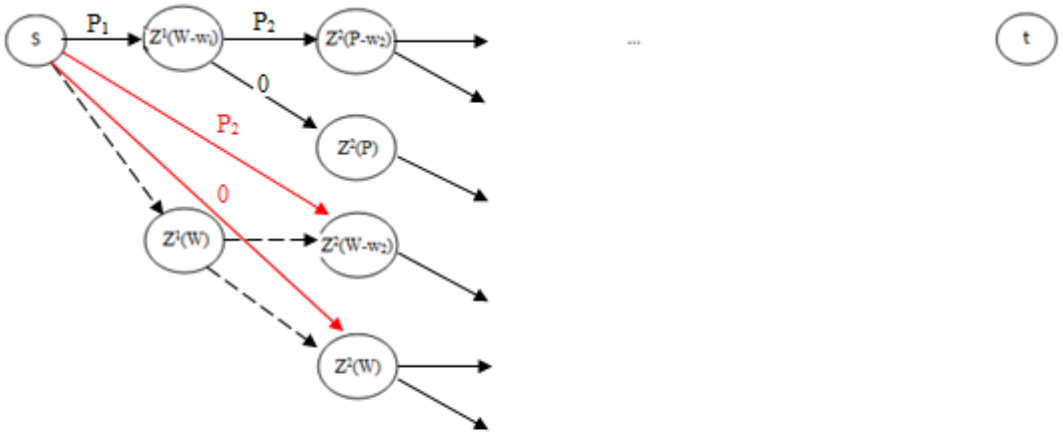
comme le montre la figure ci-dessous.

Algorithm 6: Creation of graph

Data: n, p_i, w_i, W
Result: *Graph*
Notation : P : PILE FIFO
$name(x)$: name of the summit x
$weight(x)$: weight of the knapsack on process
$stage(x)$: stage of x
(k, l) : edge leading to k with a distance of l
$successor(x)$: outgoing edges from x
initialisation : $name = 1$;
$name(s) = 0$; $weight(s) = 0$; $stage(s) = 0$ $name(t) = -1$; $weight(t) = W$;
$stage(t) = n + 1$
$Graph \longleftarrow$ {s,t}
PILE P$\longleftarrow \{s\}$;
while $P \neq \emptyset$ **do**
  $element \longleftarrow pull(P)$
  $P \longleftarrow P/element$
  if $stage(element) \neq n$ **then**
    $i \longleftarrow stage(element)$
    if $weight(element) + w_{i+1} \leq W$ **then**
      $name(new) \longleftarrow name$;$name \longleftarrow name + 1$
      $weight(new) = weight(element) + w_{i+1}$
      $stage(new) \longleftarrow i + 1$
      $successor(element) \longleftarrow \{(new, p_i)\}$
    end
    $name(new) \longleftarrow name$;$name \longleftarrow name + 1$
    $weight(new) = weight(element)$
    $stage(new) \longleftarrow i + 1$
    $successor(element) \longleftarrow \{(new, 0)\}$
  end
  if $stage(element) = n$ **then**
    $successor(element) \longleftarrow \{(p, 0)\}$
  end
end

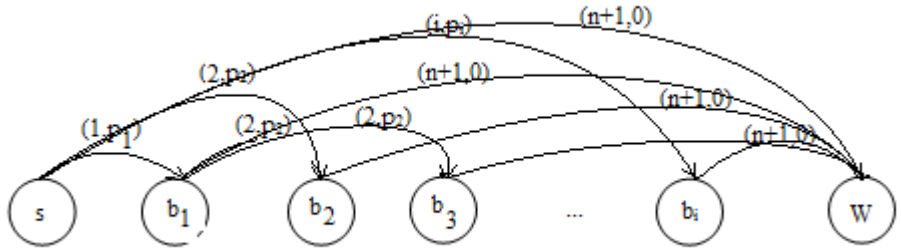
Cependant, on remarque que dans ce graphe, il existe plusieurs sommets inuitles qu'on peut les supprimer et ainsi réduire la taille du graphe, ceci on supprimant tous les sommets qui correspondent au non ajout d'un objet i et de lier son prédécesseur avec ses successeurs, ceci permet de réduire le nombre de sommets du graphe, ainsi que le nombre d'arcs (on gagne 1 arc + un sommet, comme le montre la figure2).

Figure 2: Suppression des arcs et sommets inutile



On répète cette procédure autant de fois que cela est nécessaire, à la fin nous obtenons le graphe de la figure3 :

Figure 3: Graphe associé au problème de sac à dos



Un tel graphe, peut s'obtenir ont appliquant l'algorithme[5] *suivant* :

Algorithm 7: Creation of graph

Data: n, p_i, w_i
Notation : P : PILE FIFO
$name(x)$: name of the summit x
$weight(x)$: weight of the knapsack on process
(k, l) : edge leading to k with a distance of l
$successor(x)$: outgoing edges from x
initialisation : $name = 1$;
$name(s) = 0$; $weight(s) = 0$;
$name(t) = -1$; $weight(t) = W$
$Graph \longleftarrow$ {s,t}
Result: $Graph$
for $i = 1$ *to* n **do**
  for $it = begin(Graph)$ *to* $end(Graph)$ **do**
    $P \longleftarrow \emptyset$;
    if $weight(it) + w_i \leq W$**then**
      $name(new) \longleftarrow name$;$name \longleftarrow name+1$
      $weight(new) \longleftarrow weight(it) + w_i$
      $successor(it) \longleftarrow \{(new, p_i)\}$
      $P \longleftarrow \{new\}$
    end
  end
  link all nodes of the graph with the well
  $Graph \longleftarrow \{elementsof\text{P}\}$
end
for $it = begin(Graph)$ *to* $end(Graph)$ **do**
  $successor(it) \longleftarrow$ {(t,0)}
end

On cherche alors à déterminer le plus long chemin menant du sommet s vers le sommet t. Ce problème peut se modéliser comme un cas particulier du problème du flot de coût maximum. En effet, si l'on considère le nœud s comme étant un nœud de réserve (ou offre) et le nœud t comme étant un nœud de demande, et que tous les autres nœuds soient des nœuds de transition, c'est-à-dire tels que $b_k = 0 \ \forall k \neq s, t$, alors le modèle se présente sous la forme suivante :

$$\begin{cases} max \sum_{(i,k)\in E} \text{p}_i x_{ik} \\ sc. \\ \qquad \sum_{k:(j,k)\in E} x_{jk} - \sum_{i:(i,j)\in E} x_{ij} = 0, \quad \text{j} \neq \text{s,t} \\ \qquad \text{x}_{ij} \geq 0 \quad \forall (\text{i,j}) \in E \end{cases}$$

Mais dans ce travail nous avons utilisé un algorithme qui se base sur la recherche de plus long chemin entre n'importe quel sommet i avec la source s.
Il s'agit de construire progressivement, à partir des données initiales, un sous-graphe, tel qu'au départ, on considère que les distances de chaque sommet au sommet de départ sont nulles. Au cours de chaque itération, on va mettre à jour les distances des sommets reliés

par un arc au dernier du sous-graphe (en ajoutant le poids de l'arc à la distance séparant ce dernier sommet du sommet de départ ; si la distance obtenue ainsi est supérieure à celle qui précédait). Après cette mise à jour, on examine l'ensemble des sommets qui ne font pas partie du sous-graphe, et on choisit celui dont la distance est maximale pour l'ajouter au sous-graphe.
Voici donc l'algorithme correspondant :

Algorithm 8: the longest path search

Data: *Graph*
Result: $dist(s,t)$
Notation :
$dist(v)$: distance between s and v
$f(v)$: father of v
$N(v)$: neighbors of v
$c(v)$: color of v, (such as *white* : not visited, *grey* : discovered, *black* : visited)
initialisation : $\forall v \in V(G)$
$c(v) = white$
$dist(v)$ = -∞
$f(v)$ = NULL
$c(s) = grey \quad d(s) = 0$
while $\exists v$: $c(v)$=*grey***do**
  $v \longleftarrow max\{dist(x):c(x)=grey\}$
  for $w \in N(v)$**do**
    if $c(u)$=*white* **then**
      $dist(w) \longleftarrow dist(v) + d(v,w)$
      $f(w) \longleftarrow v$
      $c(w) \longleftarrow grey$
    end
    if $c(w)$=*grey* **then**
      if $dist(w) > dist(v) + d(v,w)$ **then**
        $dist(w) \longleftarrow dist(v) + d(v,w)$
        $f(w)$
      end
    end
  end
  $c(v) \longleftarrow black$
end

4 Résultats expérimentaux

Nous avons introduit dans la partie précédente quelques algorithmes dédiés au problème du sac à dos. Nous allons, dans cette partie, présenter les résultats expérimentaux obtenus sur plusieurs instances ainsi la comparaison de ces algorithmes. Les algorithmes ont été écrits en langage C++, tous les tests ont été effectués sur un processeur Intel Core i5 1.70 et 6.00 Go RAM.

Initialement, CPLEX est un solveur de programmation linéaire, c'est un outil informatique d'optimisation commercialisé par IBM, comme son nom l'indique il fait référence au langage C et à l'algorithme du simplexe. Cplex est un des solveurs les plus performants disponibles, il peut résoudre compliqué et des problèmes de très grandes tailles. Sur ce fait nous avons implémenté le programme linéaire (P) sous ce logiciel, afin d'obtenir la solution optimale et de s'assurer que les résultats de nos programmes sont bien les résultats optimaux.
Nous avons cherché pour les expérimentations à mettre en valeur l'influence du nombre d'objets et l'influence de la taille maximale du sac sur l'efficacité des algorithmes. Les résultats présentés ont été obtenus sur une moyenne de 10 instances de chaque nombre d'objets $n = \{5, 10, 15, 20, 50.100\}$ et chaque taille $W = \{ W_1, W_2\}$, tel que $W_1 = \frac{\sum_{i=1}^{n} w_i}{n}$ (petite) et $W_2 = \frac{\sum_{i=1}^{n} w_i}{2}$(grande).
Les temps d'exécution en secondes et ont été limité à 360 secondes(6minutes).
Nous proposons 4 groupes d'instances, qui ont été construit de manière à refléter des propriétés particulières qui peuvent influer sur le processus de la recherche de la solution optimale.

4.1 Génération des instances

Dans toutes les instances, les poids sont répartis uniformément dans un intervalle de $[1, R]$, avec $R = 100$, . Les profits sont exprimés en fonction des poids, ce qui donne les propriétés spécifiques de chacun des groupes. Les groupes d'instances sont illustrés graphiquement dans la figure1.

1. Grouep1(Instance non corrélées) : les profits et les poids sont générés aléatoirement et indépendamment dans l'intervalle $[1, R]$, dans ce cas, il n'y a pas de corrélation entre le profit et le poids des articles. les instances non corrélées sont généralement faciles à résoudre soit par l'utilisation de la borne supérieures ou par des relations de dominance.

2. Groupe2(Faible corrélation) : les poids sont choisi aléatoirement dans l'intervalle $[1, R]$, et $p_j \in \left[w_j + \frac{R}{10}; w_j - \frac{R}{10}\right]$, généralement le poids diffère du poids en seulement quelques unités. telles cas sont probablement les plus réaliste.

3. Groupe3(Forte corrélation) : les poids sont répartis uniformément dans $[1, R]$ et $p_j = 5wj + 10$.

4. Groupe4(instance de Subset sum) : les poids sont répartis au hasard dans l'intervalle $[1, R]$, et $p_j = w_j$. Dans ce cas, la plupart des bornes supérieures retourne la même valeur, ce qui n'est pas trés intéressant.

Figure 4: Corrélation des instances

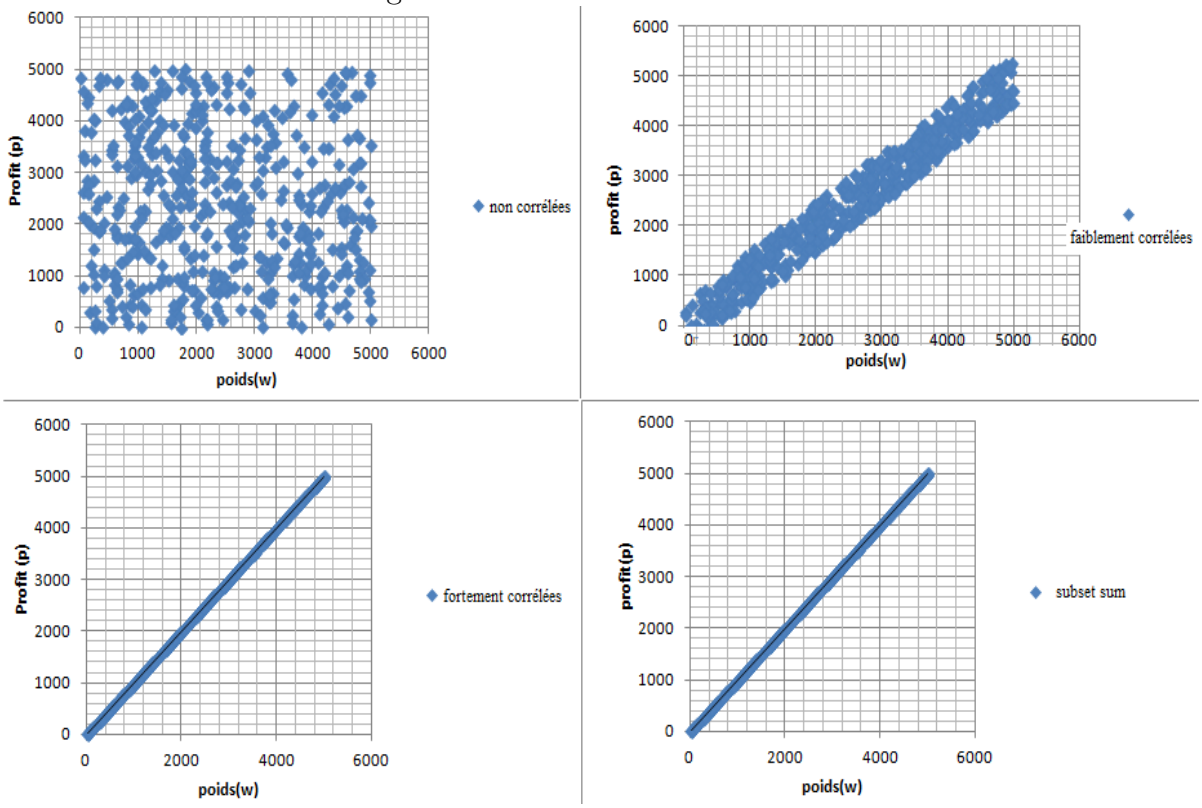


Figure 5: Comportement de chaque méthode face au type d'instance

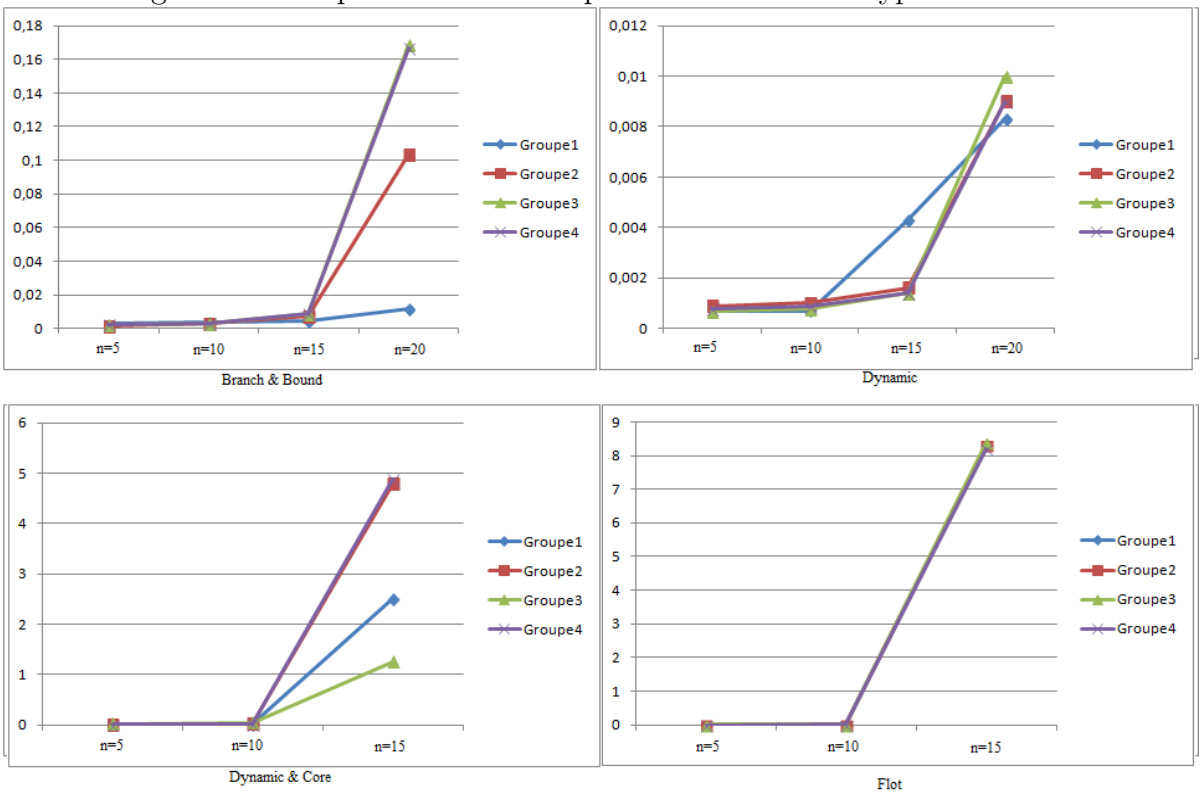
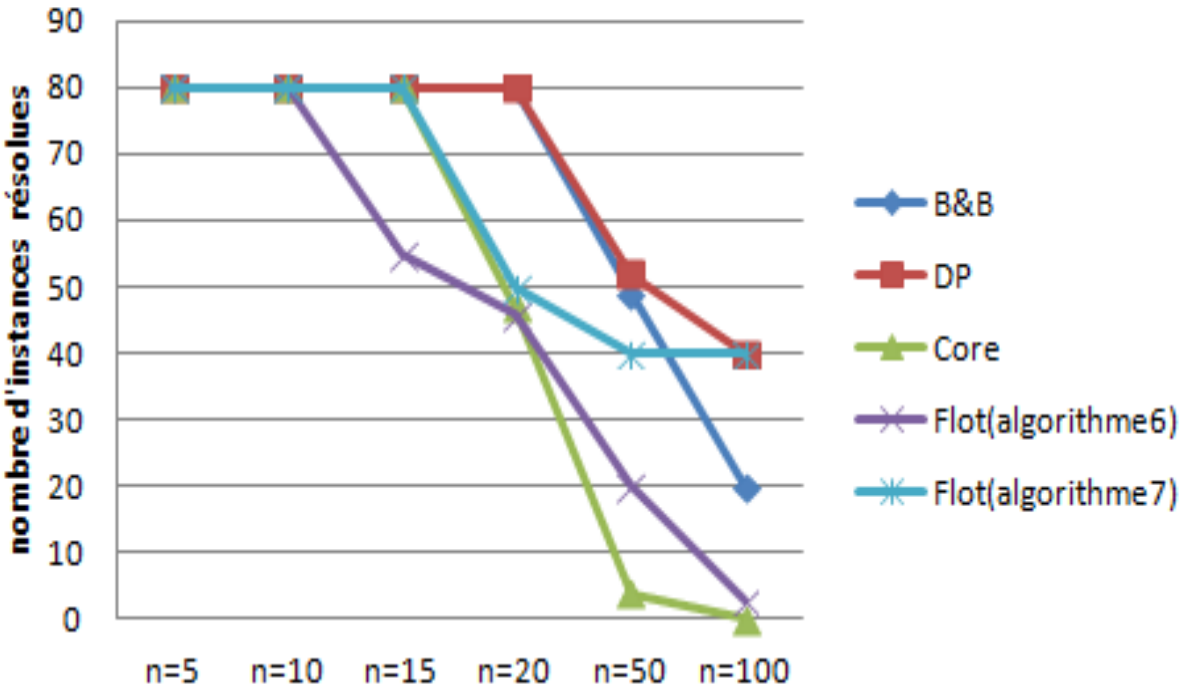


Figure 6: Nombre d'instances résolues par chaque méthode



	Branch and Bound							
	Groupe1		Groupe2		Groupe3		Groupe4	
	W1	W2	W1	W2	W1	W2	W1	W2
n=5	0	0,0016	0	0.0021	0	0,0027	0	0,0022
n=10	0	0,0027	0	0,0037	0	0,0037	0	0,0034
n=15	0	0,0043	0	0,0074	0,0001	0,0091	0,0001	0,0092
n=20	0	0,0118	0,0004	0,1037	0,0004	0,16882	0,00021	0,1669
n=50	0,00344	-	0,007	-	0,007	-	0,0089	-
n=100	0,349	-	0,628	-	-	-	-	-

Table 1: temps de calcul de Branch ans Bound

	Prgrammation dynamique							
	Groupe1		Groupe2		Groupe3		Groupe4	
	W1	W2	W1	W2	W1	W2	W1	W2
n=5	0,0004	0,0007	0,0004	0,0009	0,0006	0,0007	0,0006	0,0008
n=10	0,0004	0,0007	0,0006	0,001	0,0006	0,0008	0,0006	0,0009
n=15	0,001	0,0043	0,0012	0,0016	0,0012	0,0014	0,0012	0,0014
n=20	0,002	0,0083	0,002	0,009	0,002	0,01	0,002	0,009
n=50	0,002	-	0,0022	-	0,0033	-	0,0033	-
n=100	0,0095	-	0,0034	-	0,0082	-	0,0082	-

Table 2: temps de calcul de programmation dynamique

	Core							
	Groupe1		Groupe2		Groupe3		Groupe4	
	W1	W2	W1	W2	W1	W2	W1	W2
n=5	0,0005	0,0007	0,0006	0,009	0,0006	0,03	0,0006	0,007
n=10	0,0005	0,013	0,0006	0,0174	0,0007	0,0415	0,0006	0,171
n=15	0,003	2,5014	0,0031	3,794	0,0033	4,2613	0,0032	4,8861
n=20	0,0306	-	0,0319	-	0,0366	-	0,038	-
n=50	-	-	-	-	-	-	-	-
n=100	-	-	-	-	-	-	-	-

Table 3: temps de calcul de la programmation dynamique avec Core

	Flot1							
	Groupe1		Groupe2		Groupe3		Groupe4	
	W1	W2	W1	W2	W1	W2	W1	W2
n=5	0,0001	0,008	0	0,012	0,0001	0,012	0	0,014
n=10	0,0005	0,1306	0,0005	0,1363	0,0007	0,1341	0,001	0,1069
n=15	0,0033	8,303	0,0033	-	0,0021	-	0,0044	-
n=20	0,0162	-	0,0408	-	0,0158	-	0,0257	-
n=50	82,32	-	64,753	-	-	-	-	-
n=100	-	-	-	-	-	-	-	-

Table 4: temps de calcul du flot (algorithme 6)

	Flot2							
	Groupe1		Groupe2		Groupe3		Groupe4	
	W1	W2	W1	W2	W1	W2	W1	W2
n=5	0	0,0008	0	0,0012	0	0,008	0	0,0014
n=10	0,001	0,0113	0	0,009	0	0,0106	0	0,012
n=15	0,0001	8,303	0	8,306	0,0001	8,3924	0,0001	8,214
n=20	0,0003	-	0,0005	-	0,0005	-	0,0007	-
n=50	0,487	-	0,248	-	0,0973	-	0,1107	-
n=100	81,83	-	85.23	-	109,53	-	121,108	-

Table 5: temps de calcul du flot (algorithme 7)

4.2 Intérprétation des résultats

Sur la figure5 on peut voir l'influence de la corrélation des données sur le temps de calcul de la solution, on remarque que la méthode dynamique classique et la méthode du flot ne sont pas sensibles au type d'instance, tandis que pour la méthode dynamique avec core et la méthode de branch and bound, leurs temps d'exécution évoluent en fonction du degré de corrélation des données, cette évolution n'est en fait pas trés étonnante, puisqu'elles se basent sur le calcul des bornes, la règle de dominance et la solution du problème relaxé qui à leurs tours se basent sur le calcul du rapport profit sur poids, ce qui n'est pas le cas pour les deux autres méthodes (DP et Flot) qui énumèrent toutes les solutions possibles, quelle que soit la nature du problème.
Lorsque l'on augmente le nombre d'objets, on commence à voir apparaître quelques différences entre le nombre d'instances résolues par chaque algorithme, comme on peut le remarquer sur la figure6.
Les tableaux1,2,3,4 et 5, illustrent les temps d'exécution en secondes des différents types de problèmes considérés, à savoir non corrélés, faiblement corrélés, fortement corrélés et subset sum.
Au regard des résultats présentés, on remarque que toutes les méthodes prennent plus de temps à résoudre les problèmes avec une grande capacité du sac, ce qui est logique, puisqu'ont augmentant la capacité on élargit la taille de l'ensemble des solutions réalisables, ce qui rend l'énumération de cet espace plus difficile, on remarque aussi que la plupart des méthodes n'arrivent pas à résoudre les problèmes de taille au delà de 15 objets, sauf la méthode de Branch and Bound et la méthode dynamique qui arrivent jusqu'à 20 objets.
Et on comparant les deux tableaux 4 et 5, on remarque que le temps d'exécution de la première version de création du graphe prend plus de temp que la version améliorée, cette différence n'est pas surprenante à la lumière de la remarque que nous avons introduite lors de la description de l'algorithm7.

5 Conclusion

Dans ce travail, nous avons présenté le problème de sac à dos dans sa version la plus simple. L'importance de ce problème réside dans le domaine vaste de ses applications et ses variantes concrètes.
Nous avons étudié quelques méthodes de résolution exactes de ce problème, que nous les avons implémenté en utilisant le langage C++, finalement nous avons comparé la performance de chacune des méthodes présenté dans ce travail.

Annexes

[1] $Branchandbound$
[1] $programmationdynamique$
[1] $programmationdynamiqueduCore$
[1] $Flotversion1$
[1] $Flotversion2$

Références

[0] $F.Vanderbek.TheKnapsackProblem.IntegerProgramming(2015)$
[1] $OmerHancer.MatematicalProgramming.OR630Fall(2006)$
[2] $ChandraChekuri.ApproximationAlgorithms.CS598CSC(2011)$
[3] $DavidP.Williamson.ORIE6300MathematicalProgrammingI.November2014$