Cours n°6: Résolution de problèmes de sac à dos: algorithmes gloutons, programmation dynamique et séparation-évaluation (Branch&Bound)

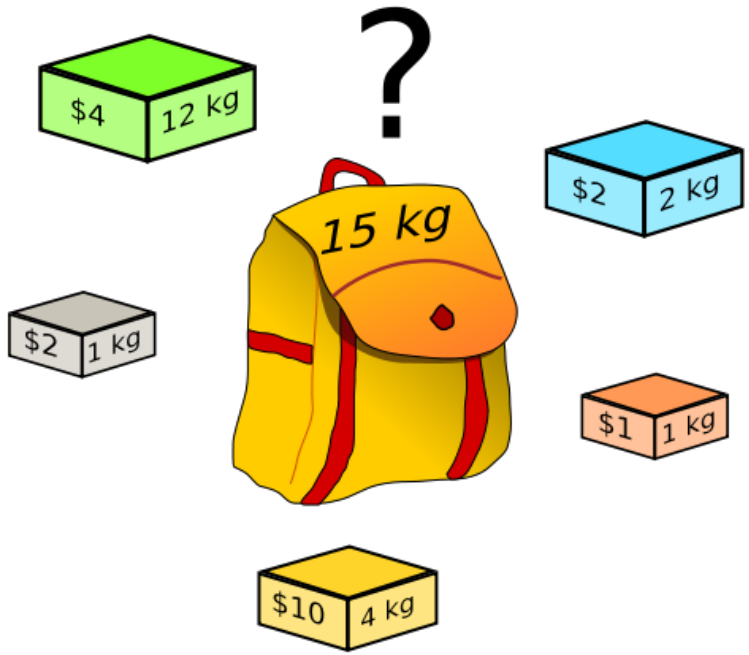
Nicolas DUPIN

https://github.com/ndupin/ORteaching
http://nicolasdupin2000.wixsite.com/research

version du 20 mars 2022

Issu et étendu d'enseignements donnés à l'ENSTA Paris, Polytech' Paris-Saclay, et à l'université Paris-Saclay

Problème de sac à dos, rappel



- On dispose de N objets, de masses en kg (m_i) et de valeur financière (c_i).
- On dispose d'un sac à dos pouvant porter jusqu'à M kg.

Problème : remplir le sac à dos en maximisant la valeur financière contenue dans le sac à dos.

Formulation PLNE du problème de sac à dos

Remplir le sac à dos en maximisant la valeur financière contenue dans le sac à dos se formule comme le problème d'optimisation sous contrainte :

$$\begin{aligned} & \max_{x_i} \sum_{i=1}^{N} c_i \times x_i \\ s.c : \quad & \sum_{i=1}^{N} m_i \times x_i \leqslant M \\ \forall i \in [\![1, N]\!], \quad & x_i \in \{0, 1\} \end{aligned} \tag{1}$$

x_i sont des variables.
c_i, m_i et M sont des paramètres numériques

Réciproquement

Réciproquement, tout problème d'optimisation qui s'écrit sous la forme suivante (qui modélisait bcp de situations concrètes, cf CM1) est un problème de type sac à dos :

$$\begin{aligned} & \max_{x_i} \sum_{i=1}^{N} c_i x_i \\ s.c : \quad & \sum_{i=1}^{N} m_i x_i \leqslant M \\ \forall i \in [\![1, N]\!], \quad & x_i \in \{0, 1\} \end{aligned} \tag{2}$$

avec $m_i \geqslant 0$ pour tout $i \in [\![1, N]\!]$

Remarque 1 : Pour que des solutions existent et que le problème soit réalisable, il faut que $M \geqslant 0$

Remarque 2 : si on a des indices avec $c_i \leqslant 0$, on ne prendra jamais l'objet, on peut retirer de tels objets.

TP/projet C++ : résolution de problèmes de sac à dos

- Séances de TP sur cette problématique.
- Ce cours introduit les notions algorithmiques de ce TP.
- Première partie : Code fourni en C++ à compléter, mais ce n'est pas le plus important.
- Questions expérimentales, pour mieux comparer/analyser les algorithmes et mieux les comprendre en pratique. Un code solution sera fourni pour finaliser ces expériences.
- Des approfondissements optionnels peuvent être réalisés suivant votre intérêt, donnent des points bonus à la note de TP.

Plan

Cas particulier du sac à dos continu

- Ici, on considère que les objets sont sécables

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

$$\begin{array}{ll} \max & 5x_A + 8x_B + 14x_C + 6x_D + 13x_E + 17x_F + 10x_G + 4x_H \\ s.c: & 2x_A + 3x_B + 5x_C + 2x_D + 4x_E + 6x_F + 3x_G + x_H \leqslant 12 \\ \forall i \in \{A, \ldots, H\} & 0 \leqslant x_i \leqslant 1 \end{array} \tag{3}$$

- Quelle stratégie de construction de solution utiliseriez vous ?

Cas particulier du sac à dos

On a intérêt un prendre un objet léger qui a une grande valeur (H par ex).

A priori peu de chance de prendre un objet lourd ayant peu de valeur (comme B).

On trie les objets sur le rapport Prix/Masse. (quel est le prix du kilo ?)

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4

Disposant de 12 kg autorisé, on remplit le sac à dos suivant le rapport Prix/Masse.

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
							•	X

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

On prend l'objet G de masse 3kg, $x_G = 1$, et il reste 8kg à remplir

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
					•		X	X

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

On prend l'objet G de masse 3kg, $x_G = 1$, et il reste 8kg à remplir

On prend l'objet E de masse 4kg, $x_E = 1$, et il reste 4kg à remplir.

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
				•	X		X	X

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

On prend l'objet G de masse 3kg, $x_G = 1$, et il reste 8kg à remplir

On prend l'objet E de masse 4kg, $x_E = 1$, et il reste 4kg à remplir.

On prend l'objet D de masse 2kg, $x_D = 1$, et il reste 2kg à remplir.

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
				X	X	•	X	X

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

On prend l'objet G de masse 3kg, $x_G = 1$, et il reste 8kg à remplir

On prend l'objet E de masse 4kg, $x_E = 1$, et il reste 4kg à remplir.

On prend l'objet D de masse 2kg, $x_D = 1$, et il reste 2kg à remplir.

On prend l'objet F de masse 6kg, $x_6 = \frac{2}{6} = \frac{1}{3}$, et le sac à dos continu est rempli.

SOLUTION : $x_D = x_E = x_G = x_H = 1$, $x_F = \frac{1}{3}$ et $x_A = x_B = x_C = 0$.

Cette stratégie fournit une SOLUTION OPTIMALE du sac à dos continu. (on va le prouver)

Résolution gloutonne du sac à dos continu

$$\begin{array}{ll} s.c : & \max \sum_{i=1}^{n} c_i x_i \\ & \sum_{i=1}^{n} m_i x_i \leqslant M \\ \forall i \in [\![1, n]\!] & x_i \in [0, 1] \end{array} \tag{4}$$

- On trie les objets selon les rapports c_i/m_i décroissants.
- On remplit le sac à dos dans cet ordre. Pour les premiers objets, on leur affecte la valeur 1.
- Si un objet ne peut pas rentrer dans le sac à dos, on prend la valeur fractionnaire maximale qui rentre, et les autres variables sont fixées à 0.

Cette stratégie fournit une SOLUTION OPTIMALE du sac à dos continu. On va le prouver.

Preuve d'optimalité, un lemme essentiel

$$\begin{aligned} & \max_{x \in [0,1]^n} \sum_{i=1}^{n} c_i x_i \\ s.c : \quad & \sum_{i=1}^{n} m_i x_i \leqslant M \\ \forall i \in [\![1, n]\!] \quad & x_i \in [0, 1] \end{aligned}$$

Lemme

Soit $x \in [0,1]^n$ une solution optimale avec deux indices $i \neq j$ tels que $x_i, x_j > 0$ et $c_i/m_i > c_j/m_j$. On a nécessairement $x_i = 1$

Preuve : par contraposée, si $x_i < 1$, on pourrait construire une solution strictement améliorante en faisant augmenter $x_i < 1$ et baisser $x_j > 0$, tout en gardant la contrainte $\sum_{i=1}^{n} m_i x_i$ et les contraintes de bornes, et ainsi ce n'était pas optimal.

Preuve d'optimalité, applications du lemme

$$\begin{array}{ll} & \max_x \sum_{i=1}^{n} c_i x_i \\ s.c : & \sum_{i=1}^{n} m_i x_i \leqslant M \\ \forall i \in [\![1, n]\!] & x_i \in [0, 1] \end{array}$$

Théorème

Soit un problème de sac à dos tels que les rapports c_i/m_i sont tous distincts. Alors, le problème de sac à dos admet une unique solution, donnée par l'algorithme glouton, qui a au plus une valeur fractionnaire

Théorème

L'algorithme glouton précédent fournit toujours une solution optimale du sac à dos continu. Si l'objet ajouté en dernier a un rapport c_i/m_i distinct des autres objets, la solution est unique.
Dans le cas contraire, toute solution optimale contient les objets ayant un meilleur rapport c_i/m_i et ne contient pas les objets ayant un rapport c_i/m_i inférieur.
Toute solution optimale du sac à dos peut être obtenue par différents équilibrages des objets ayant le rapport c_i/m_i critique en satisfaisant la contrainte globale $\sum_{i=1}^{n} m_i x_i = M$

Résolution gloutonne du sac à dos continu (2)

$$\begin{array}{ll} s.c : & \max \sum_{i=1}^{n} c_i x_i \\ \forall i \in [\![1, n]\!] & \sum_{i=1}^{n} m_i x_i \leqslant M \\ & x_i \in [0, 1] \end{array} \tag{5}$$

- On trie les objets selon les rapports c_i/m_i décroissants.
- On remplit le sac à dos dans cet ordre. Pour les premiers objets, on leur affecte la valeur 1.
- Si un objet ne peut pas rentrer dans le sac à dos, on prend la valeur fractionnaire maximale qui rentre, et les autres variables sont fixées à 0.
- Cette solution est optimale
- Algorithme en complexité $\Theta(n \log n)$ (la complexité du tri sachant que le parcours linéaire est en $O(n)$)

ATTENTION : règle propre au sac à dos simple avec une contrainte. Avec deux contraintes de sac à dos, il n'y a pas de propriété équivalente, on utilisera alors un algorithme plus générique de résolution PL.

Lien entre sac à dos continu et entier

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
Solution				1	1	1/3	1	1

Dans notre exemple, la SOLUTION OPTIMALE du sac à dos continu est : $x_D = x_E = x_G = x_H = 1$, $x_F = \frac{1}{3}$ et $x_A = x_B = x_C = 0$.

Prix : $13 + 4 + 10 + 6 + \frac{1}{3}17 \approx 38,66666667$

Si on regarde le problème de sac à dos entier, quel lien entre cette valeur et l'optimum entier ? Quelle stratégie peut on utiliser pour avoir une solution entière ?

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
				X	X	•	X	X

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

On prend l'objet G de masse 3kg, $x_G = 1$, et il reste 8kg à remplir

On prend l'objet E de masse 4kg, $x_E = 1$, et il reste 4kg à remplir.

On prend l'objet D de masse 2kg, $x_D = 1$, et il reste 2kg à remplir.

On prend l'objet F de masse 6kg, > 2, on passe au suivant.

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
			•	X	X	-	X	X

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

On prend l'objet G de masse 3kg, $x_G = 1$, et il reste 8kg à remplir

On prend l'objet E de masse 4kg, $x_E = 1$, et il reste 4kg à remplir.

On prend l'objet D de masse 2kg, $x_D = 1$, et il reste 2kg à remplir.

On regarde l'objet F de masse 6kg, > 2, on passe au suivant.

On regarde l'objet C de masse 5kg, > 2, on passe au suivant.

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
		•	-	X	X	-	X	X

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

On prend l'objet G de masse 3kg, $x_G = 1$, et il reste 8kg à remplir

On prend l'objet E de masse 4kg, $x_E = 1$, et il reste 4kg à remplir.

On prend l'objet D de masse 2kg, $x_D = 1$, et il reste 2kg à remplir.

On regarde l'objet F de masse 6kg, > 2, on passe au suivant.

On regarde l'objet C de masse 5kg, > 2, on passe au suivant.

On regarde l'objet B de masse 3kg, > 2, on passe au suivant.

Dernier objet : A de masse 2kg, rentre dans le sac à dos.

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
		•	-	X	X	-	X	X

On prend l'objet H de masse 1kg, $x_H = 1$, et il reste 11kg à remplir

On prend l'objet G de masse 3kg, $x_G = 1$, et il reste 8kg à remplir

On prend l'objet E de masse 4kg, $x_E = 1$, et il reste 4kg à remplir.

On prend l'objet D de masse 2kg, $x_D = 1$, et il reste 2kg à remplir.

On regarde l'objet F de masse 6kg, > 2, on passe au suivant.

On regarde l'objet C de masse 5kg, > 2, on passe au suivant.

On regarde l'objet B de masse 3kg, > 2, on passe au suivant.

Dernier objet : A de masse 2kg, rentre dans le sac à dos.

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
	X	-	-	X	X	-	X	X

On a sélectionné : $x_H = x_G = x_E = x_D = x_A = 1$ et $x_F = x_B = x_C = 0$.

prix : $13 + 4 + 10 + 6 + 5 = 38$

On avait tout à l'heure la valeur : $38,66666667$

Est-ce logique ?

Relaxation et bornes inférieures (1)

Soit A, B dans $\mathbb{R}^n$ avec $A \subset B$, soit $f : B \longrightarrow \mathbb{R}$ une fonction bornée.

$\inf_{x\in A} f(x) = \inf\{f(x)|x \in A\}$, $\inf_{x\in B} f(x)$, $\sup_{x\in A} f(x) = \sup\{f(x)|x \in A\}$, $\sup_{x\in B} f(x)$ sont bien définis et :

$$\inf_{x\in A} f(x) \geqslant \inf_{x\in B} f(x) \tag{6}$$

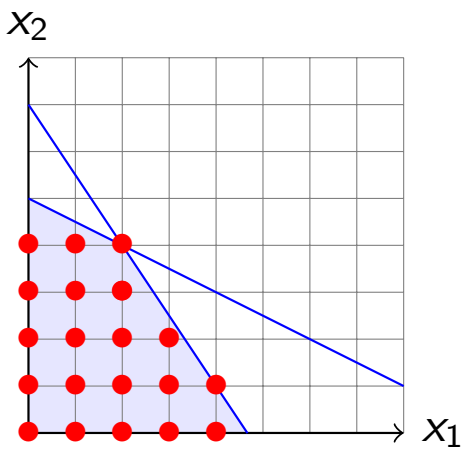
$$\sup_{x\in A} f(x) \leqslant \sup_{x\in B} f(x) \tag{7}$$

Plus on a d'éléments à considérer, plus le minimum est bas
Plus on a d'éléments à considérer, plus le maximum est élevé

Dans le cas où B est bornée dans $\mathbb{R}^n$, f est continue, les hypothèses sont vérifiées. Si de plus B est fermé (alors compact), les bornes inférieures et supérieures sont des min et max : les bornes inférieures et supérieures sont atteintes

Relaxation et bornes inférieures (2)

Application aux problèmes de sac à dos : $\max_{x\in\{0,1\}^n} \sum_{i=1}^{n} c_i x_i$, s.c : $\sum_{i=1}^{n} m_i x_i \leqslant m$



Avec $A = \{x \in \{0,1\}^n \ , \ \sum_{i=1}^{n} m_i x_i \leqslant m\}$ et
$B = \{x \in [0,1]^n \ , \ \sum_{i=1}^{n} m_i x_i \leqslant m\}$.
On a $A \subset B$ et le résultat précédent donne :

$$\max_{x\in\{0,1\}^n} \sum_{i=1}^{n} c_i x_i \quad \leqslant \quad \max_{x\in[0,1]^n} \sum_{i=1}^{n} c_i x_i$$

$$s.c : \sum_{i=1}^{n} m_i x_i \leqslant m \qquad s.c : \sum_{i=1}^{n} m_i x_i \leqslant m$$

$\Longrightarrow$ Le sac à dos continu donne un majorant de l'optimum entier.

N.B : Le raisonnement est valide pour les problèmes de sac à dos généraux. Dans notre exemple de sac à dos simple, on a en plus un algorithme spécifique, simple et efficace.

N.B : En RO, majorant est appelé souvent borne supérieure du pb d'optimisation, ce n'est pas la même borne supérieure qu'en maths, le plus petit des majorants.

Retour à notre exemple

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
	X	-	-	X	X	-	X	X

On a sélectionné : $x_H = x_G = x_E = x_D = x_A = 1$ et $x_F = x_B = x_C = 0$.

Prix sac à dos entier : $13 + 4 + 10 + 6 + 5 = 38$

Dans le cas du sac à dos continu, on avait tout à l'heure la valeur : $38,66666667$

Est-ce logique ?

$$\begin{array}{lll} & \max\limits_{x\in\{0,1\}^n} \sum\limits_{i=1}^{n} c_i x_i & \leqslant \quad \max\limits_{x\in[0,1]^n} \sum\limits_{i=1}^{n} c_i x_i \\ s.c: & \sum_{i=1}^{n} m_i x_i \leqslant m & \quad s.c: \quad \sum_{i=1}^{n} m_i x_i \leqslant m \end{array}$$

La solution donnant 38 est solution réalisable du problème continu dont $38,66666667$ est l'optimum, on attendait bien une valeur inférieure.

N'y a t'il pas quelque-chose de remarquable ici ?

Retour à notre exemple

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4
Prix/Masse	2,5	2,67	2,8	3	3,25	2,83	3,33	4
	X	-	-	X	X	-	X	X

On a une solution de valeur : 38 avec : $x_H = x_G = x_E = x_D = x_A = 1$ et $x_F = x_B = x_C = 0$.

Or, on a le majorant de l'optimum du sac à dos entier : $38,66666667$

Donc le sac à dos entier, a un coût à valeur entière qui est au plus 38.

On a alors prouvé que $x_H = x_G = x_E = x_D = x_A = 1$ et $x_F = x_B = x_C = 0$ est une solution optimale !

Contre-exemple simple

Ce serait trop beau que l'algorithme glouton donne toujours des solutions optimales de problèmes de sac à dos entiers (cf théorie de la complexité) !

Un contre exemple :

Objet	A	B	C	D
Masse (kg)	2	4	5	1
Prix (euros)	15	12	14	1
Prix/Masse	7.5	3	2.8	1

Masse maximale du sac à dos : 7

Solution optimale $x_A = x_C = 1$ et $x_B = x_D = 0$, de valeur 29.

Solution de la relaxation continue : $x_A = x_B = 1$, $x_C = 0.2$ et $x_D = 0$, de valeur $29,8$.

Solution de l'algorithme glouton entier : $x_A = x_B = x_D = 1$ et $x_C = 0$, de valeur 28.

Cas général : relaxation continue

Avec les deux stratégies gloutonnes qu'on vient de voir on peut donc encadrer la valeur optimale d'un problème de type sac à dos, en $O(n \log n)$ (rapide !).

1. La relaxation continue donne une borne supérieure :

$$\begin{array}{rl} \max\limits_{x\in\{0,1\}^n} \sum\limits_{i=1}^{n} c_i x_i & \leqslant \quad \max\limits_{x\in[0,1]^n} \sum\limits_{i=1}^{n} c_i x_i \\ s.c: \quad \sum_{i=1}^{n} m_i x_i \leqslant m & \qquad s.c: \quad \sum_{i=1}^{n} m_i x_i \leqslant m \end{array}$$

2. La stratégie gloutonne en nombre entier donne une borne inférieure puisque toute solution réalisable $x^0 \in \{0,1\}^n$ vérifiant la contrainte $\sum_{i=1}^{n} m_i x_i \leqslant m$ est un minorant de l'optimum :

$$\begin{array}{rl} \max\limits_{x\in\{0,1\}^n} \sum\limits_{i=1}^{n} c_i x_i & \geqslant \quad \sum\limits_{i=1}^{n} c_i x_i^0 \\ s.c: \quad \sum_{i=1}^{n} m_i x_i \leqslant m & \end{array}$$

Implémentation C++ lors des TP

- Support de TP : code fourni en C++, des fonctions à écrire à partir d'une structure définie.
- Objectif : comparaison de différents solveurs (dont le glouton !)
- On a défini par avance une classe solveur générique dont tous les solveurs héritent. Nommée KpSolver, elle contient :

  des fonctions génériques d'import ou d'affichage
  les éléments faisant une instance de sac à dos : nb d'objets, poids, coûts.

- Stratégie : Chaque solveur remplit le champ solution suivant la solution construite par l'approche, de même que l'encadrement avec le minorant de l'optimum (costSolution) et le majorant (upperBoundOPT)

```
#include <vector>
using namespace std;
class KpSolver {
private:
    void sortKnapsack();
    void clearInstance();
protected:
    int nbItems; // Number of items.
    vector<int> weights; //weights of items
    vector<int> values; //values of items
    int knapsackBound;      // Knapsack bound
    vector<bool> solution;
    int costSolution;
    float upperBoundOPT;
public:
    void importInstance(const string& fileName);
    void printKnapsackInstance();
    void printKnapsackSolution(bool printSol);
    int getSolutionCost(){return costSolution;};
    bool isSelected(int item){return solution[item];};
};
```

Jeux de données fournis

- Pour le TP, les jeux de données sont fournis dans le dossier instances/
  - courseExample.in : c'est l'exemple du cours, utile pour déboguer.
  - kp_a_b.in : il y a 6 jeux de données initiaux avec $a \in \{100, 1000, 10000, 100000\}$ le nombre d'objets et $b \in \{1, 2\}$ correspond à deux jeux de données simulés.
  - pour ces instances, il faudra peut être utiliser des jeux de données tronqués pour étudier précisément l'impact du nombre d'objets et modifiés : quel est l'impact de la valeur M sur les caractéristiques de résolution ?
- La fonction importInstance(const string& fileName) permet d'importer de telles données (déjà codée).

Format d'entrée des données

courseExample contient l'exemple du cours, et permet de déboguer assez facilement :

```
8
2 3 5 2 4 6 3 1
5 8 14 6 13 17 10 4
12
```

Première ligne : nombre d'objets

Seconde ligne : les masses des objets

Troisième ligne : les prix des objets

Dernière ligne : la masse maximale du sac à dos

Implémentation C++ des bornes gloutonnes

- Classe KpSolverGreedy hérite de KpSolver, et implémente les deux stratégies gloutonnes de cette section (continu ou entier), pour remplir les champs de bornes et la valeur de la solution.
- A vous d'implémenter, les fonctions void solveLowerBound() et void solveUpperBound().
- void solve() appelle juste les deux calculs de bornes void solveLowerBound() et void solveUpperBound(), rien à modifier.

```
//********* kpSolverGreedy.hpp *********

#ifndef KPSOLVERGREEDY_HPP
#define KPSOLVERGREEDY_HPP

#include "kpSolver.hpp"

class KpSolverGreedy : public KpSolver {

public:

    void solveLowerBound();
    void solveUpperBound();
    void solve();
};

#endif
```

Compiler et lancer des premiers calculs

A ce stade (implémentation des calculs gloutons), la compilation la plus simple est :

```
> g++ *.cpp -o greedySolverKp
```

On peut optimiser la compilation en ajoutant le flag -O3

Après la compilation, l'exécution se fait en spécifiant le chemin de l'instance avec (par exemple) :

```
> ./greedySolverKp instances/courseExample.in
```

Remarque : la compilation d'objets avec makefile permet de ne recompiler que les nouvelles classes, l'objet issus de KpSolver n'a pas à être recompilé, pour un gain de temps à la compilation.

Compiler et lancer des premiers calculs : exemple d'output

```
>  g++ *.cpp -O3 -o greedySolverKp

> ./greedySolverKp instances/courseExample.in

Greedy bounds :
elapsed time: 2.619e-06s
solution cost : 38
proven upper bound  : 38.6667
proven upper bound  after rounding: 38
gap : 0%

> ./greedySolverKp instances/kp_100000_1.in

Greedy bounds :
elapsed time: 0.00047404s
solution cost : 40686621
proven upper bound  : 4.06869e+07
proven upper bound  after rounding: 4.06869e+07
gap : 0.000796331%
```

Bilan partiel

La résolution par l'algorithme glouton est rapide en $O(N \log N)$.

- le calcul continu fournit une borne supérieure, optimiste pour le cas discret ;
- ces premières bornes (continues ou discrètes) se codent facilement ;
- le calcul continu permet de prouver l'optimalité d'une solution construite avec l'algo entier (si c'est optimal),
- ou bien d'évaluer la qualité de solutions construites à la main ;
- mais le procédé n'est pas général pour fournir des solutions prouvées optimales

Plan

Rappel : Programmation dynamique, cadre général

- La programmation dynamique s'applique pour des problèmes d'optimisation combinatoires pour lesquels une solution optimale est composée de bouts de solutions optimales : Principe d'optimalité de Bellman. Par exemple :

  - Le plus court chemin entre deux villes (cf itinéraire Google Map) : les chemins formés par les villes intermédiaire sont optimaux pour joindre ces deux villes.
  - Le plus court chemin entre deux sommets d'un graphe (Algorithme de Bellman-Ford, algorithme de Dijkstra en complexité polynomiale).

- La résolution par programmation dynamique s'apparente à stocker des bouts de solutions optimales et à utiliser des relations de récurrence.

- On reverra la programmation dynamique plus tard dans ce cours dans un cadre spécifique d'optimisation combinatoire et en algorithmie, .

Résolution par programmation dynamique du sac à dos, cadre général

- Variables : $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m}$ représente le coût maximal en remplissant un sac à dos de capacité m, parmi les i premiers objets.
- $C_{N,M}$ donne la valeur du problème d'optimisation.
- Conditions limites :

$$\forall i \in [\![0, N]\!], C_{i,0} = 0$$
$$\forall m \in [\![0, M]\!], C_{0,m} = 0$$

- Formule de récurrence par disjonction de cas :

$$C_{i,m} = C_{i-1,m} \text{ si } m_i > m$$
$$C_{i,m} = \max\left\{C_{i-1,m}, C_{i-1,m-m_i} + c_i\right\} \text{ si } m_i \leqslant m$$

- $C_{N,M}$ est obtenu après MN calculs en construisant les cases de la matrices suivant i croissant. Le calcul de la solution optimale est en $O(MN)$.

Rappel : Illustration sur l'exemple numérique

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5
2 premiers objets	0	0	5	8	8	13	13	13	13	13	13	13	13
3 premiers objets	0	0	5	8	8	14	14	19	22	22	27	27	27
4 premiers objets	0	0	6	8	11	14	14	20	22	25	28	28	33
5 premiers objets	0	0	6	8	13	14	19	21	24	27	28	33	35
6 premiers objets	0	0	6	8	13	14	19	21	24	27	30	33	36
7 premiers objets	0	0	6	10	13	16	19	23	24	29	31	34	37
8 premiers objets	0	4	6	10	14	17	20	23	27	29	33	35	38

Rappel : Illustration du backtrack sur l'exemple numérique

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5
2 premiers objets	0	0	5	8	8	13	13	13	13	13	13	13	13
3 premiers objets	0	0	5	8	8	14	14	19	22	22	27	27	27
4 premiers objets	0	0	6	8	11	14	14	20	22	25	28	28	33
5 premiers objets	0	0	6	8	13	14	19	21	24	27	28	33	35
6 premiers objets	0	0	6	8	13	14	19	21	24	27	30	33	36
7 premiers objets	0	0	6	10	13	16	19	23	24	29	31	34	37
8 premiers objets	0	4	6	10	14	17	20	23	27	29	33	35	38

Algorithme (pseudo-code) : Programmation dynamique, pb de sac à dos

Entrée : N objets définis par leur coût c_i et leur masse m_i pour tout $i \in [\![1; N]\!]$
Sortie : le coût de la solution optimale et la composition d'un sac à dos optimal

Initialisation :
- un vecteur de booléens $s_i = 0$ pour tout $i \in [\![0; N]\!]$
- une matrice C avec $C_{i,m} = 0$ pour tout $i \in [\![0; N]\!]$, $m \in [\![0; M]\!]$

for $i = 1$ to $N - 1$ //Construction de la matrice C
  (parallel) for $m = 1$ à M
    $C_{i,m} = C_{i-1,m}$
    if $m \geqslant m_i$ **then** $C_{i,m} = \max(C_{i,m}, C_{i-1,m-m_i} + c_i)$
  end for
end for

initialise $m = M$
for $n = N$ to 1 avec incrément $n \leftarrow n - 1$
  if $C_{n,m} \neq C_{n-1,m}$ **then** $s_n = 1$ et $m = m - m_n$
end for

retourne $C_{N,M}$ le coût optimal la solution s

Rappel : Complexité

- Construction de la matrice $C_{i,m}$: chaque case se construit en $\Theta(1)$, $\Theta(N.M)$ au total.
- Back-tracking : N opérations en $\Theta(1)$, $\Theta(N)$ au total.
- Complexité totale de la programmation dynamique $C_{i,m}$: $\Theta(N.M)$ au total. NON-POLYNOMIAL, juste PSEUDO-POLYNOMIAL
- Données entrée : deux vecteurs d'entier flottants de taille N, masses et coûts, et un entier M : masse du sac à dos. Taille des données d'entrée : $\Theta(N + \log M)$.
- Complexité non polynomiale : $N.M$ n'est pas polynomial par rapport à $(N + \log M)$.
- Complexité pseudo-polynomiale : à M fixé, la programmation dynamique est polynomiale
- Le problème de sac à dos entier est bien NP-complet.

Implémentation C++ de la programmation dynamique

- Classe KpSolverDP définie dans kpSolver.hpp hérite de KpSolver.
- Fonction solve() implémente de différentes façons l'algorithme de programmation dynamique
- L'appel à solve() remplit les champs de bornes (égaux car algorithme exact) et la valeur de la solution, hérités de la classe mère KpSolver.
- int** matrixDP est la matrice de programmation dynamique à remplir. Les fonctions pour la créer à la taille de l'instance et la supprimer en mémoire sont déjà données.
- Dans l'implémentation, deux modes de construction possibles, et un backtrack commun, d'où la séparation dans le code.

N.B : L'implémentation int** n'est pas la plus efficace, il est préférable d'avoir un stockage sur un espace mémoire contigu de la matrice de programmation dynamique. Un tel choix aide à la lisibilité du code.

```
#include "kpSolver.hpp"

class KpSolverDP : public KpSolver {

private:
    int** matrixDP;

    bool memoizedVersion;
    bool parallelizedVersion;
    bool verboseMode;

    void solveIter();
    int solveMemoized(int i, int m);
    void backtrack();
    void createMatrixDP();
    void deleteMatrixDP();
    void printMatrixDP();
    void fillFirstColumnDP();

public:
    void solve();
};
```

La fonction solve()

- La fonction solve() de la classe KpSolverDP est donnée, ne pas la modifier.
- Dans l'implémentation, deux modes de construction possibles, et un backtrack commun.
- Construction itérative (avec memoizedVersion = false), correspond à l'algorithme écrit précédemment.
- Construction récursive avec mémoïsation (avec memoizedVersion = true), on en parle après
- La fonction fillFirstColumnDP() ; est donnée, elle remplit la matrice de prog. dynamique relative à l'objet 0.

  Remarque : en C++ le premier objet est indexé par 0 et pas 1 donc la matrice est indexée de 0 à N-1 pour les objets et de m=0 à M pour les masses. Les champs $C_{0,m}$ correspondent donc à la ligne de prog. dynamique correspondant au premier objet.
- A vous d'implémenter la construction itérative de la matrice de programmation dynamique et l'algorithme de backtracking.

```
#include "kpSolverDP.hpp"

void KpSolverDP::solve() {
    createMatrixDP();
    fillFirstColumnDP();

    //2 possible constructions for the DP matrix
    if (memoizedVersion)
        costSolution = solveMemoized(nbItems-1,knapsackBound);
        else solveIter();

    upperBoundOPT = costSolution ;
    if (verboseMode) printMatrixDP();
    backtrack();
    deleteMatrixDP();
}
```

N.B : verboseMode permet d'afficher la matrice de programmation dynamique, utile pour déboguer.

Rappel : Memoïsation et programmation dynamique

- Idée : on applique les formules de récurrence. On commence avec une matrice de avec que des -1, sauf sur les cases terminales qu'on peut initialiser. Ensuite :
  - Si on a besoin d'une case déjà calculée, on renvoie la valeur stockée.
  - Si on a besoin d'une case non calculée (de valeur -1), on calcule la valeur correspondante par récurrence et on stocke la valeur calculée.
  - Bonus : on ne calcule que les cases de la matrice dont on a réellement besoin pour calculer la valeur terminale $C_{N,M}$
- La matrice C est indispensable de toute manière pour le backtrack et renvoyer une composition optimale du sac à dos
- La construction de la matrice C par mémoïsation est déjà codée dans le code fourni

```
int KpSolverDP::solveMemoized(int i , int m) {

    if (matrixDP[i][m] > -1)  return matrixDP[i][m];

    if (m < weights[i] )
        matrixDP[i][m] =  solveMemoized(i-1,m);
    else matrixDP[i][m] = max(
                solveMemoized(i-1,m),
                values[i]
                + solveMemoized(i-1,m - weights[i])
                );

      return matrixDP[i][m];
}
```

Pour aller plus loin avec la mémoïsation

La mémoïsation est une technique que vous pouvez étudier dans le cours de Programmatioon Fonctionnelle Avancée (Ocaml)

Dans les langages de programmation fonctionnels tels que OCaml, où les fonctions sont des entités de première classe, il est possible de réaliser la mémoïsation de manière automatique et externe à la fonction (c'est-à-dire sans modifier la fonction récursive initiale)

Mémoïsation automatique en OCaml et en Python :

`https://perso.crans.org/besson/notebooks/agreg/M%C3%A9moisation_en_Python_et_OCaml.html`

`https://www.python-course.eu/python3_memoization.php`

On peut également coder de la mémoïsation automatique en C++, avec plus d'efforts :

`http://slackito.com/2011/03/17/automatic-memoization-in-cplusplus0x/`

`https://cpptruths.blogspot.com/2012/01/general-purpose-automatic-memoization.html?m=1`

Compiler et exécuter la programmation dynamique

L'exécution s'effectue en spécifiant le chemin de l'instance avec un paramètre quelconque. On calcule les différents modes de programmation dynamique (iter ou memoïsation) à la suite de l'algo glouton. Par exemple :

```
> ./kpSolvers instances/courseExample.in 1
```

Avec un second paramètre quelconque (ici, exemple :1), on active le mode verbeux. On affiche la matrice de programmation dynamique. Attention, ca peut afficher beaucoup de lignes !

```
> ./kpSolvers instances/courseExample.in 1 1
```

```
>  g++ *.cpp -fopenmp -O3 -o kpSolvers

> ./kpSolvers instances/courseExample.in 1 1

Max capacity knapsack : 12
Item: 0 Weight : 1 Value : 4 Marginal Cost : 4
Item: 1 Weight : 3 Value : 10 Marginal Cost : 3.33333
Item: 2 Weight : 4 Value : 13 Marginal Cost : 3.25
Item: 3 Weight : 2 Value : 6 Marginal Cost : 3
Item: 4 Weight : 6 Value : 17 Marginal Cost : 2.83333
Item: 5 Weight : 5 Value : 14 Marginal Cost : 2.8
Item: 6 Weight : 3 Value : 8 Marginal Cost : 2.66667
Item: 7 Weight : 2 Value : 5 Marginal Cost : 2.5

Greedy bounds :
elapsed time: 2.631e-06s
solution cost : 38
proven upper bound  : 38.6667
proven upper bound  after rounding: 38
gap : 0%
knapsack composition  : 0 1 2 3 7
Max capacity knapsack : 933877552
```

```
Dynamic Programming iterative version without parallelization  :

elapsed time: 0.000245605s
solution cost by DP: 38
print DP matrix :
0 4 4 4 4 4 4 4 4 4 4 4 4
0 4 4 10 14 14 14 14 14 14 14 14 14
0 4 4 10 14 17 17 23 27 27 27 27 27
0 4 6 10 14 17 20 23 27 29 33 33 33
0 4 6 10 14 17 20 23 27 29 33 34 37
0 4 6 10 14 17 20 23 27 29 33 34 37
0 4 6 10 14 17 20 23 27 29 33 35 37
0 4 6 10 14 17 20 23 27 29 33 35 38
knapsack composition  : 0 1 2 3 7
```

```
Dynamic Programming memoized version :

elapsed time: 0.000249444s
solution cost by DP: 38
print DP matrix :
0 4 4 4 4 4 4 4 4 4 4 4 4
0 4 4 10 14 14 14 14 14 14 14 -1 14
0 4 4 10 14 17 17 23 27 27 27 -1 27
-1 4 6 10 14 17 20 23 -1 29 33 -1 33
-1 -1 6 -1 14 17 -1 23 -1 29 33 -1 37
-1 -1 -1 -1 -1 -1 -1 23 -1 29 33 -1 37
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 33 -1 37
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 38
knapsack composition  : 0 1 2 3 7
```

Des remarques sur ces résultats

On n'a pas exactement les mêmes résultats que la matrice donnée dans le cours

Raison : Les objets sont triés dans la matrice de prog dynamique. Cela ne change pas le résultat final.

On visualise le nombre de cases non nécessaires avec la mémoïsation.

Expérience numérique : quelle est la proportion de calculs de cases inutiles peut être gagnée grâce à la mémoïsation ?

Quelle stratégie d'ordre entre les objets peuvent être implémentés pour maximiser le nombre de cases non calculées ? Stratégies à valider empiriquement, on pourra s'inspirer de la méthode void sortKnapsack() ; de KpSolver.

Résultats en ayant désactivé le tri des instances sur les coûts marginaux :

```
Dynamic Programming iterative version without parallelization  :

elapsed time: 0.00140724s
solution cost by DP: 38
print DP matrix :
0 0 5 5 5 5 5 5 5 5 5 5 5
0 0 5 8 8 13 13 13 13 13 13 13 13
0 0 5 8 8 14 14 19 22 22 27 27 27
0 0 6 8 11 14 14 20 22 25 28 28 33
0 0 6 8 13 14 19 21 24 27 28 33 35
0 0 6 8 13 14 19 21 24 27 30 33 36
0 0 6 10 13 16 19 23 24 29 31 34 37
0 4 6 10 14 17 20 23 27 29 33 35 38
knapsack composition  : 0 3 4 6 7
```

Résultats en ayant désactivé le tri des instances sur les coûts marginaux :

```
Dynamic Programming memoized version :

solution cost by DP: 38
print DP matrix :
0 0 5 5 5 5 5 5 5 5 5 5 5
0 0 5 8 8 13 13 13 13 13 13 13 13
0 0 5 8 8 14 14 19 22 22 27 27 27
-1 0 6 8 11 14 14 20 22 25 -1 28 33
-1 -1 6 8 -1 14 19 -1 24 27 -1 33 35
-1 -1 -1 -1 -1 -1 -1 -1 24 27 -1 33 36
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 34 37
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 38
knapsack composition  : 0 3 4 6 7
```

Expérimentations sur instances plus grandes

En désactivant le mode verbeux, on peut comparer les temps de calculs sans le biais des temps de sortie écran.

Les instances à 1000 objets se résolvent assez bien.

On peut regarder l'impact de la parallélisation. Idéalement, accélération proportionnel au nombre de threads.

Expérience numérique : comparer les temps de calculs des différentes approches sur un même jeu d'instances, regarder l'impact des valeurs de N et M sur les temps de calculs.

N.B : Les résultats numériques se présentent avec les caractéristiques matérielles et logicielles, ici 4 coeurs CPU, Intel(R) Core(TM) i5-3320M CPU, 2.60GHz, linux Ubuntu 20.04 et g++ version 7.5.0.

N.B : C++ n'est pas optimisé pour la récursivité, cela pénalise l'approche mémoïsation ici. Les résultats seraient différents avec OCaml.

```
> ./kpSolvers instances/kp_1000_1.in 1

Greedy bounds :
elapsed time: 1.8941e-05s
solution cost : 396644
proven upper bound  after rounding: 396700
gap : 0.0141185%


Dynamic Programming iterative version without parallelization  :
elapsed time: 0.873885s
solution cost : 396688
proven upper bound  : 396688
gap : 0%

Dynamic Programming memoized version :
elapsed time: 3.18181s
solution cost : 396688
```

Expérimentations optionnelles mémoïsation

Expérimentation : suivant les instances, quelle proportion de calculs sont évités grâce à la mémoïsation ?

ie : compter la proportion de -1 dans la matrice de programmation dynamique.

Remarque : contrairement aux algorithmes gloutons, l'ordre d'indexation des objets n'a pas d'importante.

Par quel ordre sur les objets peut on minimiser le temps de calcul, ie maximiser le nombre de -1 dans la matrice de programmation dynamique ?

Pour cela, on peut s'inspirer de la fonction de la classe KpSolver : void sortKnapsack() ;

On peut recoder de telles fonctions de tri et analyser expérimentalement l'impact sur les résultats.

N.B : les résultats dépendent ici des valeurs numériques en entrée. On validera alors les expérimentations sur un nombre significatif d'instances, et on mesurera la dispersion des résultats.

Expérimentations optionnelles implémentation/parallélisation

Expérimentations : diminuer les temps de calculs par une meilleure implémentation de la version itérative de l'algorithme de programmation dynamique

On comparera alors les temps de calculs sur différentes tailles de données.

piste 1 : le conteneur int** pour matrixDP n'est pas contigu : à remplacer par un simple tableau int* concaténant les différentes lignes de la matrice de prog dynamique

piste 2 : parallélisation avec OpenMP sur les différents threads d'une machine

En effet, de nombreux calculs sont indépendants dans la construction de la matrice de programmation dynamique et peuvent être réalisés indépendamment en parallèle.

Boucles for parallélisée avec Open MP

Avec de la mémoire partagée, (plusieurs threads sur plusieurs cœurs CPU d'un PC), OpenMP permet une parallélisation facile sur les différents threads .

Boucles for parallélisées lorsque les opérations peuvent être réalisées indépendamment.

N.B : Avec Python, numba est un équivalent de parallélisation facile avec une mémoire partagée.
`https://numba.readthedocs.io/en/stable/user/parallel.html`

Pour aller plus loin

La parallélisation GPGPU est la mieux adaptée à la programmation dynamique, voir :

https://hal.archives-ouvertes.fr/hal-01152223/file/ArticleCuda.pdf

Implémentation avec CUDA sous C++ :
https://forums.developer.nvidia.com/t/another-cuda-implementation-of-a-dynamic-programming-problem/31403

https://github.com/OlegKonings/CUDA_MEGA_DP_Example?files=1

Avec Python, numba permet de tels calculs sur GPU :
https://numba.readthedocs.io/en/stable/cuda/index.html

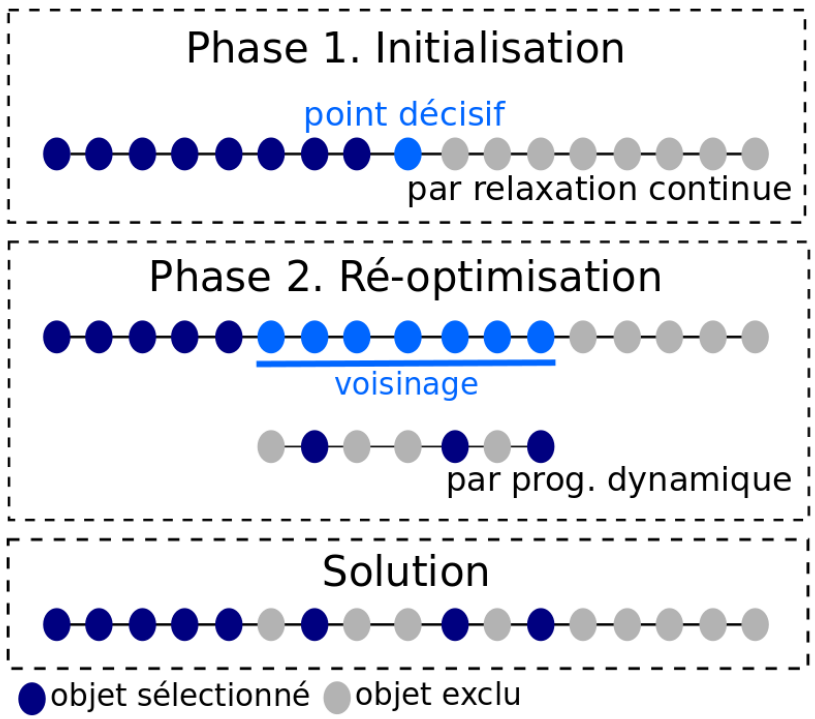
Plan

Heuristique Kernel Search

Parfois les problèmes sont de très grande taille et il faudrait énormément de temps et d'espace pour le résoudre. On peut favoriser une approche par heuristique : on approchera la solution optimale à moindre coût.

Heuristique Kernel Search

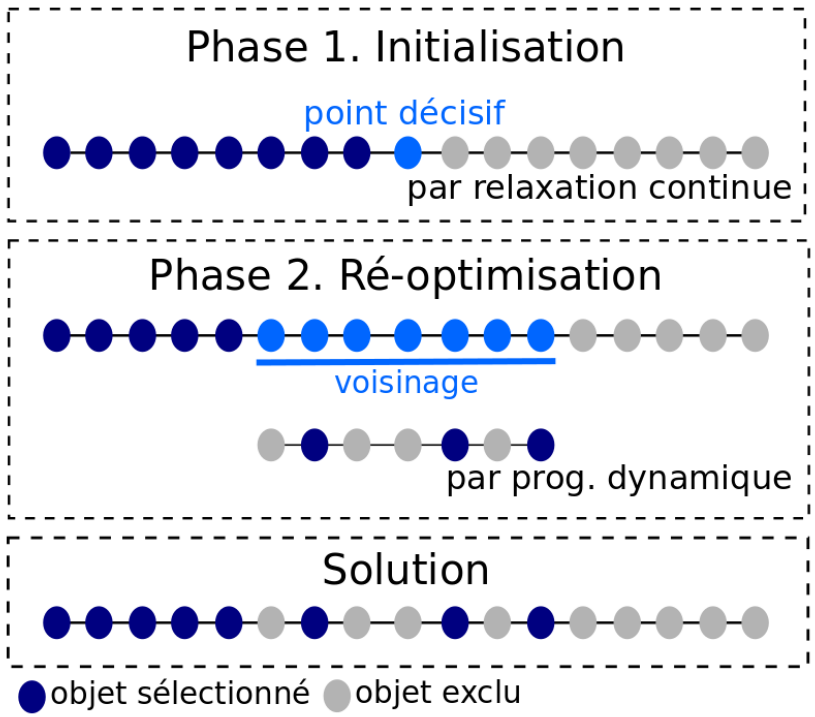
1. Initialisation : Estimation de l'objet décisif par relaxation continue (comme pour l'algo glouton).



Heuristique Kernel Search

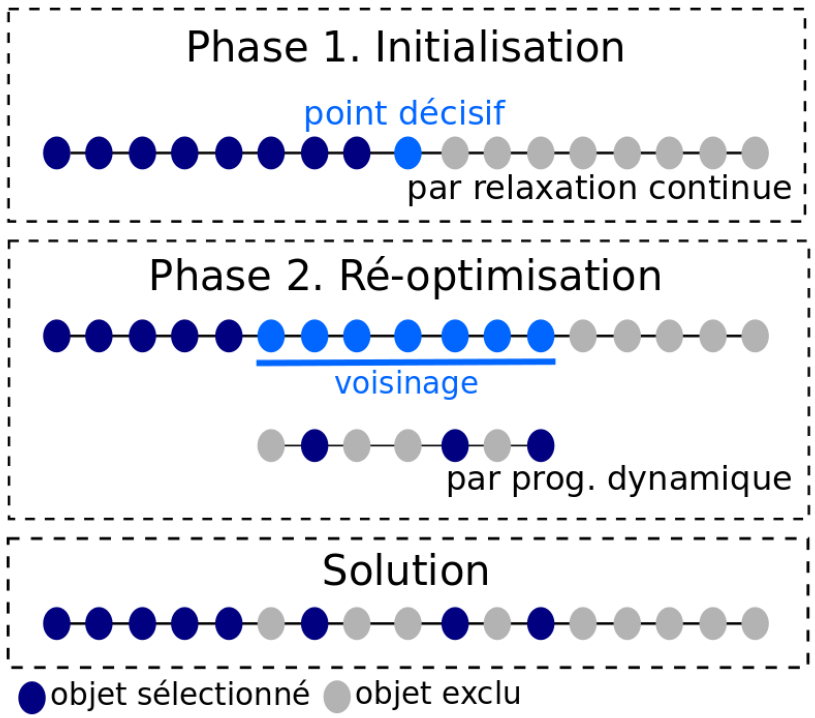
2. Ré-optimisation : Les points hors du voisinage ont leur statut sélectionné ou retiré fixé ; il reste un problème de sac à dos à résoudre avec les objets dans le voisinage de l'objet décisif. Ici, la liste des objets et la capacité du sac à dos est modifié par rapport au problème initial.

- On en déduit la solution.



Heuristique Kernel Search

- On parle d'heuristique paramétrique car le voisinage de réoptimisation autour du noyau est un paramètre du problème. Dans notre cas on peut prendre N=1000 puisque la prog. dynamique est efficace pour des problèmes de cette taille



A vous de l'implémenter !

Conseils :

- faire hériter une telle classe de la classe KpSolverGreedy
- Dans le .cpp, on utilisera la programmation dynamique en définissant le KpSolver correspondant au noyau de la Kernel Search

```
//********* kpSolverHeurDP.hpp *********

#ifndef KPSOLVERHEURDP_HPP
#define KPSOLVERHEURDP_HPP

#include "kpSolverGreedy.hpp"

class KpSolverHeurDP : public KpSolverGreedy {

protected:
    int nbSelectedReopt;
    int nbUnselectedReopt;
    int lastIndex;
    int fixedCost;

public:
    void solveUpperBound();
    void solve();
    void setNeighborhood(int nb1, int nb2);
};
#endif
```

Plan

Rappel : relaxation continue

Avec les deux stratégies gloutonnes qu'on vient de voir on peut donc encadrer la valeur optimale d'un problème de type sac à dos, en $O(n \log n)$ (rapide !).

1. La relaxation continue donne une borne supérieure :

$$\begin{array}{rl} \max\limits_{x\in\{0,1\}^n} \sum\limits_{i=1}^{n} c_i x_i & \leqslant \quad \max\limits_{x\in[0,1]^n} \sum\limits_{i=1}^{n} c_i x_i \\ s.c: \quad \sum_{i=1}^{n} m_i x_i \leqslant m & \qquad s.c: \quad \sum_{i=1}^{n} m_i x_i \leqslant m \end{array}$$

2. La stratégie gloutonne en nombre entier donne une borne inférieure puisque toute solution réalisable $x^0 \in \{0,1\}^n$ vérifiant la contrainte $\sum_{i=1}^{n} m_i x_i \leqslant m$ est un minorant de l'optimum :

$$\begin{array}{rl} \max\limits_{x\in\{0,1\}^n} \sum\limits_{i=1}^{n} c_i x_i & \geqslant \quad \sum\limits_{i=1}^{n} c_i x_i^0 \\ s.c: \quad \sum_{i=1}^{n} m_i x_i \leqslant m & \end{array}$$

Exemple déroulé pour l'algorithme de Branch&Bound

$$\min \quad x_1 + 1,5x_2 + 2x_3 + 5x_4 + 9x_5$$

$$s.c. \quad x_1 + 2x_2 + 3x_3 + 6x_4 + 10x_5 \geqslant 10$$

$$\forall i, \quad x_i \in \{0, 1\}$$

$\Longrightarrow$ Ce problème est il équivalent à un problème de sac à dos ?

Rappel

$$\begin{aligned} & \min_{x_i} \sum_{i=1}^{N} c_i x_i \\ s.c : \quad & \sum_{i=1}^{N} m_i x_i \geqslant M \\ \forall i \in [\![1, N]\!], \quad & x_i \in \{0, 1\} \end{aligned} \tag{8}$$

avec $m_i \geqslant 0$ pour tout $i \in [\![1, N]\!]$, revient à résoudre le problème de sac à dos :

$$\begin{aligned} OBJ = \quad & \max_{y_i} \sum_{i=1}^{N} c_i y_i \\ s.c : \quad & \sum_{i=1}^{N} m_i y_i \leqslant \sum_{i=1}^{N} m_i - M \\ \forall i \in [\![1, N]\!], \quad & y_i \in \{0, 1\} \end{aligned} \tag{9}$$

On récupère le résultat du problème initial avec $OBJ - \sum_{i=1}^{N} c_i$ à l'objectif et on renvoie comme affectation $x_i = 1 - y_i$

Application

Pour le problème :

$$\min \quad x_1 + 1,5x_2 + 2x_3 + 5x_4 + 9x_5$$

$$s.c. \quad x_1 + 2x_2 + 3x_3 + 6x_4 + 10x_5 \geqslant 10$$

$$\forall i, \quad x_i \in \{0, 1\}$$

Ce problème est équivalent à un problème de sac à dos

La relaxation continue se calcule par l'algorithme glouton, optimal pour calculer l'optimisation continue

Exemple déroulé pour l'algorithme de Branch&Bound

$$\min \quad x_1 + 1,5x_2 + 2x_3 + 5x_4 + 9x_5$$

$$s.c. \quad x_1 + 2x_2 + 3x_3 + 6x_4 + 10x_5 \geqslant 10$$

$$\forall i, \quad x_i \in \{0, 1\}$$

Solution continue : $x_2 = x_3 = 1,\ x_1 = x_5 = 0,\ x_4 = \frac{5}{6}$

Relâché continu : $1,5 + 2 + \frac{25}{6} \approx 7,67$

Exemple déroulé pour l'algorithme de Branch&Bound

$$\min \quad x_1 + 1,5x_2 + 2x_3 + 5x_4 + 9x_5$$
$$s.c. \quad x_1 + 2x_2 + 3x_3 + 6x_4 + 10x_5 \geqslant 10$$
$$\forall i, \quad x_i \in \{0,1\}$$

Solution continue : $x_2 = x_3 = 1,\ x_1 = x_5 = 0,\ x_4 = \frac{5}{6}$

Relâché continu : $1,5 + 2 + \frac{25}{6} \approx 7,67$

$x_4 = 0$

$$\min \quad x_1 + 1,5x_2 + 2x_3 + 9x_5$$
$$s.c. \quad x_1 + 2x_2 + 3x_3 + 10x_5 \geqslant 10$$
$$\forall i, \quad x_i \in \{0,1\}$$

$x_4 = 1$

$$\min \quad 5 + x_1 + 1,5x_2 + 2x_3 + 9x_5$$
$$s.c. \quad x_1 + 2x_2 + 3x_3 + 10x_5 \geqslant 4$$
$$\forall i, \quad x_i \in \{0,1\}$$

Exemple déroulé pour l'algorithme de Branch&Bound

$$\min \; x_1 + 1,5x_2 + 2x_3 + 5x_4 + 9x_5$$

$$s.c. \; x_1 + 2x_2 + 3x_3 + 6x_4 + 10x_5 \geqslant 10$$

$$\forall i, \; x_i \in \{0,1\}$$

Solution continue : $x_2 = x_3 = 1,\ x_1 = x_5 = 0,\ x_4 = \frac{5}{6}$

Relâché continu : $1,5 + 2 + \frac{25}{6} \approx 7,67$

$x_4 = 0$ $\qquad$ $x_4 = 1$

$$\min \; x_1 + 1,5x_2 + 2x_3 + 9x_5$$

$$s.c. \; x_1 + 2x_2 + 3x_3 + 10x_5 \geqslant 10$$

Solution continue : $x_2 = x_3 = 1,\ x_1 = 0,\ x_5 = \frac{1}{2}$

Relâché continu : $1,5 + 2 + \frac{18}{2} = 8$

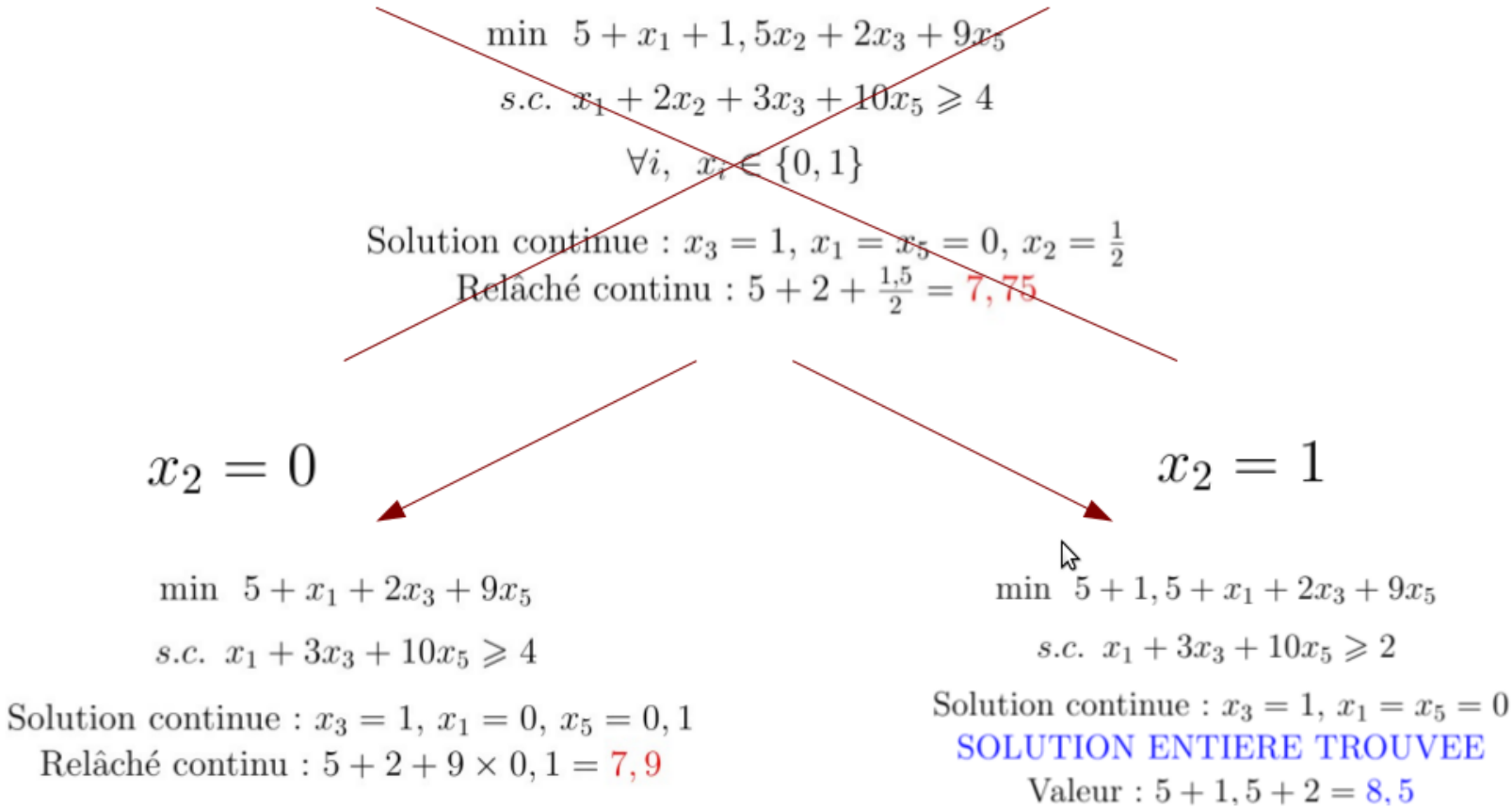
$$\min \; 5 + x_1 + 1,5x_2 + 2x_3 + 9x_5$$

$$s.c. \; x_1 + 2x_2 + 3x_3 + 10x_5 \geqslant 4$$

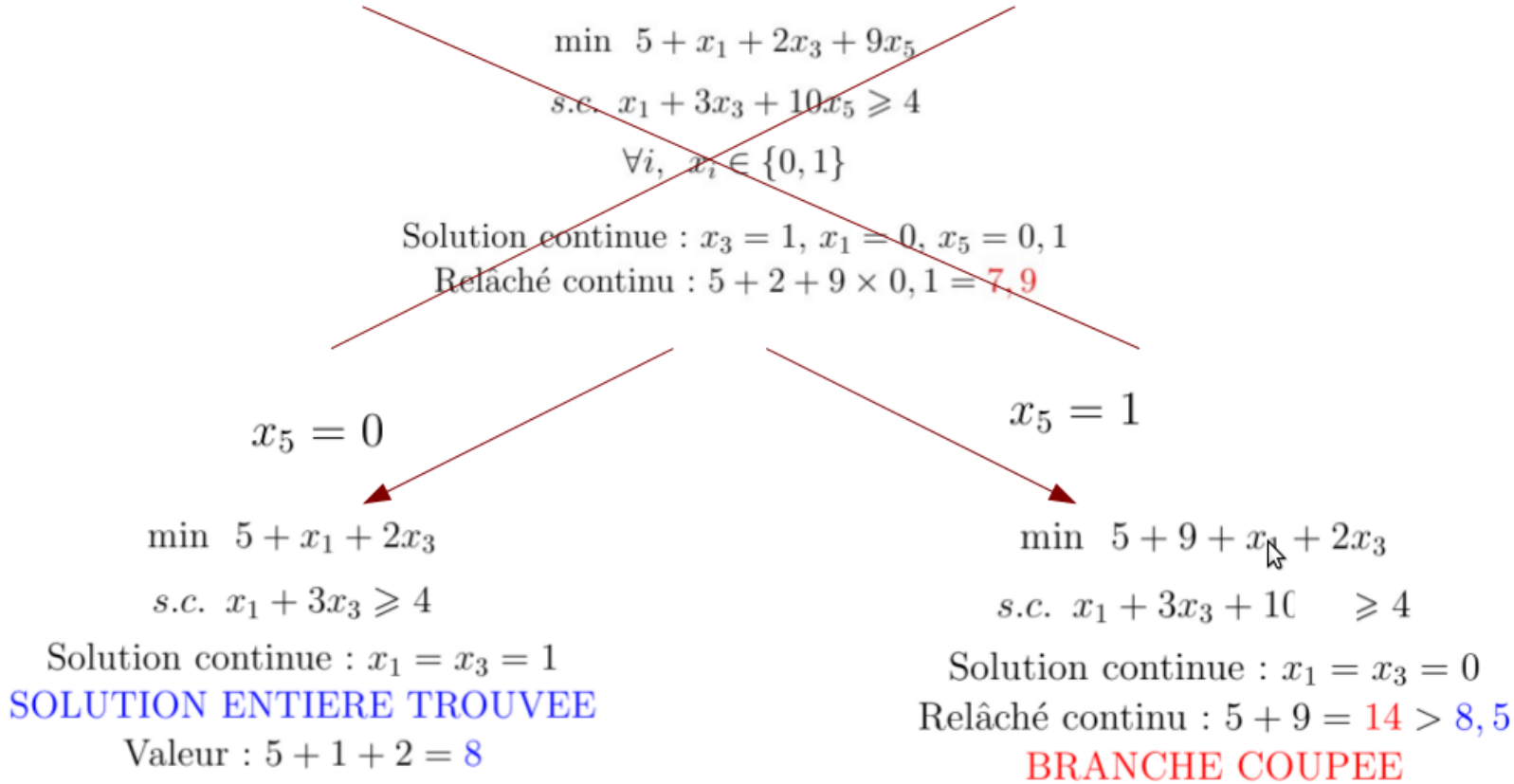
Solution continue : $x_3 = 1,\ x_1 = x_5 = 0,\ x_2 = \frac{1}{2}$

Relâché continu : $5 + 2 + \frac{1,5}{2} = 7,75$

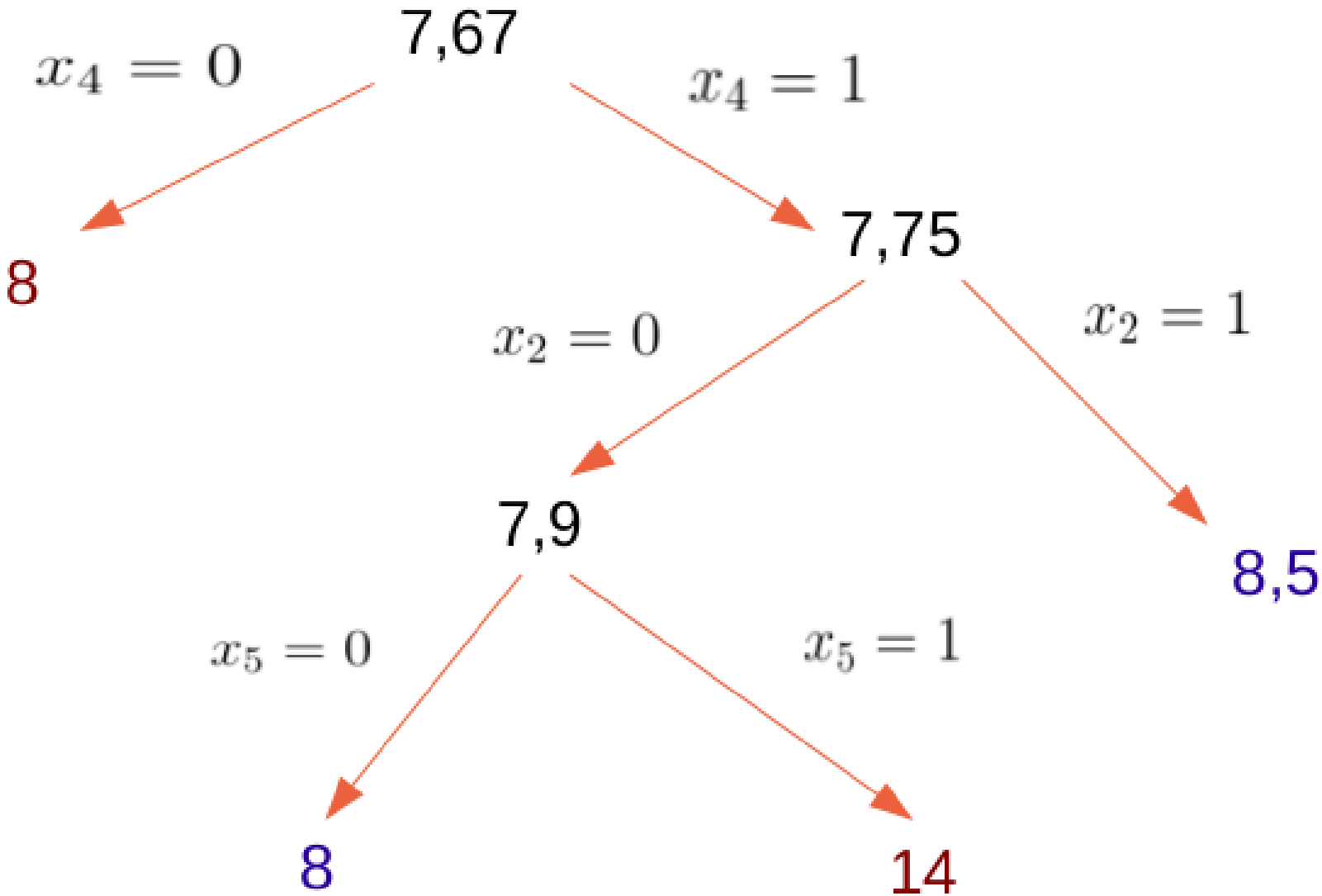
Exemple déroulé pour l'algorithme de Branch&Bound



Exemple déroulé pour l'algorithme de Branch&Bound



Exemple déroulé pour l'algorithme de Branch&Bound



De manière générale

- Ce qui a été vu sur l'exemple est en fait très général.
- Vous pouvez coder l'algorithme de Branch & Bound sur tout problème de sac à dos en partant de l'exemple précédent.
- A tout noeud de l'arbre, fixer des variables à 0 ou 1 induit toujours un problème de sac à dos, la borne de relaxation continue se calcule par l'algorithme glouton sur le sac à dos restreint.
- Trois cas lorsqu'on considère un noeud :
  - avoir une solution entière arrête la résolution
  - la relaxation continue, "borne optimiste", est moins bonne que la meilleure solution courante, pas besoin d'énumérer les sous cas.
  - dans les autres cas, on branche sur la variable fractionnaire
- Un degré de liberté dans l'algorithme : quel noeud courant choisir pour itérer ?

Quel noeud choisir ?

- "BestBound" : dans l'exemple, choix d'uyn noeud ayant le meilleure borne de relaxation continue
- "Random“ : on choisit un noeud au hasard
- "BFS" : parcours en largeur classique
- "DFS" : parcours en profondeur. Comment choisir sur lequel des deux noeuds pon commence ? Cela fait deux variantes, en choisissant comme neoud suivant le noeud après branchement à 0 ou à 1.

N.B : les parcours d'arbres BFS, DFS sont un classique de l'informatique. Une différence ici, l'arbre est créé dans le cours de l'algorithme, ce n'est pas le parcours d'un arbre donné en entrée et bien défini, cela induit des problématiques spécifiques.

Implémentation récursive ?

- Parcours en profondeur se code facilement avec de la récursivité (cf cours Ocaml, Programmation Fonctionnelle)
- Parcours en largeur, ou autres (Bestbound et Random) se coderaient suivant une autres structure de programmation en récursivité.
- Dans la suite, implémentation C++, la plus générique pour avoir ces différentes variantes de Branch&Bound
- Stockage des noeuds comme un deque, liste doublement chaînée, pour avoir accès en O(1) au premier et au dernier élément dans l'implémentation

Implémentation C++

- Une classe NodeBB pour coder un noeuds actif de l'arbre de B&B et les opérations correspondante.
- Une classe KpSolverBB héritant de KpSolver, qui manipule un ensemble de noeuds stockés dans un deque.
- La méthode solve() de KpSolverBB à coder comme précédemment
- L'adaptation des calculs gloutons de relaxation continue et d'heuristique gloutonne pour une solution entière à coder dans nodeBB.cpp

```
#include <vector>
using namespace std;
class  NodeBB {
private:
     vector<bool> isFixed;
     vector<bool> isRemoved;
     vector<bool> primalSolution;
     int criticalIndex;
     float fractionalVariable;
     double localUpperBound;
     long localLowerBound;
     bool overCapacitated;
public:
    bool canBeRemoved();
    void primalHeuristic(int kpB, int nblt, vector<int> & w,
vector<int> & val);
    void solveUpperBound(int kpB, int nblt, vector<int> & w,
vector<int> & val);
    void init(int size);
    void updateAfterbranching0(int id);
    void updateAfterbranching1(NodeBB* nod, int id);
};
```

La classe NodeBB

- Pour encoder qu'une variable x_i est fixée à 1, on utilise isFixed[i]=true
- Pour encoder qu'une variable x_i est fixée à 0, on utilise isRemoved[i]=true
- primalSolution, comme précédemment, indique la composition du sac à dos calculé ,par heuristique gloutonne. en mémoire uniquement quand on a besoin, on libère la mémoire utilisée au plus vite
- Ces choix permettent de minimiser l'espace mémoire occupé, devient critique quand le nombre de noeuds devient grand (algorithme B&B exponentiel en temps et en mémoire en général)

```
enum BranchingStrategies {BestB, DFS10, DFS01, BFS, Random};

class KpSolverBB : public KpSolver {
private:
    deque<NodeBB*> nodes;
    int nbNodes;
    int nbMaxNodeBB;
    bool withPrimalHeuristics;
    bool verboseMode;
    bool withDPinitPrimalHeuristic;
    int sizeDPheur;
    BranchingStrategies branchingStrategy;

    void solveUpperBound();
    NodeBB* selectNode();
    NodeBB* selectNodeRandom();
    void insertNode(NodeBB* nod);
    void insertNodes(NodeBB* nod1, NodeBB* nod2);
    void init();

public:
    void solve();
};
```

La classe KpSolverBB

- La liste des noeuds courants est un deque¡NodeBB*¿ : seule utilisation de pointeurs explicites dans le code, pour libérer de la mémoire dynamiquement. Cela nécessite d'appeler le destructeur.
- nbMaxNodeBB à -1 est la résolution exacte, sinon, on limite le nombre de noeuds parcourus (ce qui limite le temps et l'espace mémoire consommé)
- nbNodes compte le nombre de noeuds parcourus, intéressant pour comparer les convergences de l'algorithme dans différentes configurations.
- BranchingStrategies contient les différentes stratégies de parcours, coda ge assez générique.
- Les fonctions d'insertion et de sélection de noeuds sont déjà codées

```
NodeBB* KpSolverBB::selectNodeRandom() {
    srand (time(NULL));
    int n = nodes.size() ;
    int id= rand() % n ;
    swap(nodes[0],nodes[id]);
    NodeBB* node = nodes.front();
    nodes.pop_front();
    return node;
}

NodeBB* KpSolverBB::selectNode() {
    if(branchingStrategy == Random) return selectNodeRandom();
    if(branchingStrategy == BestBound)
        sort(nodes.begin(),nodes.end(),[](NodeBB* i,NodeBB* j)
        {return i->getNodeUpperBound()<j->getNodeUpperBound();}
        if(branchingStrategy == BFS){
        NodeBB* node = nodes.front();nodes.pop_front();
        return node;
    }
    NodeBB* nod = nodes.back();
    nodes.pop_back();
    return nod;
}
```

```
void KpSolverBB::insertNode(NodeBB *nod) {
    if ((nod->canBeRemoved())) {
        delete nod;
    }
    else nodes.push_back(nod);
}

void KpSolverBB::insertNodes(NodeBB *nod1, NodeBB *nod2) {
        if(branchingStrategy == DFS01) {
                insertNode(nod1);
                insertNode(nod2);
        }
        else {
                insertNode(nod2);
                insertNode(nod1);
        }
}
```

Expériences numériques

- Comparaison de convergence exacte : combien de noeuds parcourus pour prouver l'optimalité de la solution retournée ? Sur différentes configurations, on comparer les valeurs de nbNodes.

- Avec une limite raisonnable à nbMaxNodeBB (100,1000), a t'on une heuristique efficace ? Comment se situent de telles heuristiques par rapport aux approches précédentes ?

Plan

Note sur le critère d'élagage

- Le critère d'élagage pour qu'un noeud fractionnaire soit retiré : $UB \leqslant LB$ où UB est la relaxation continue au noeud considéré, et LB la meilleure solution trouvée jusque-là.
- Avec des coûts entiers, on peut même arrêter quand $\lfloor UB \rfloor \leqslant LB$, ie $UB < LB + 1$, ce qui améliore le critère précédent.
- Comment élaguer un noeud au plus tôt ?
- Piste 1 : avoir au plus tôt la meilleure valeur possible de LB : heuristiques primales
- Piste 2 : pouvoir améliorer les bornes de relaxation continue à chaque noeud, améliorer UB.

Heuristique primale

Pour avoir au plus tôt la meilleure valeur possible de LB : plusieurs types d'heuristiques primales

- A chaque noeud, on peut faire un calcul glouton de solution entière.
- Initialement, on peut lancer n'importe quelle heuristique, par exemple une KernelSearch, pour partir d'une très bonne solution.
- autre stratégie : mélanger les stratégies de branchement/sélection pour pouvoir favoriser de temps à autre la recherche de solution entières.

Implémentation C++

- Le champs de KpSolverBB withPrimalHeuristics active à chaque noeud un calcul glouton de solution entière dans la fonction solve().
- Dans la fonction init(), on peut activer une KernelSearch avec withDPinitPrimalHeuristic= true de paramètre sizeDPheur.

Expériences numériques

- Amélioration de la convergence exacte : qu'apportent les différentes heuristiques comme gain d'exploration de noeuds au prix de temps passé à calculer des heuristiques ?
- Cas idéal : si on a initialement la solution optimale, quel est le nombre de noeud nécessaire pour prouver que la solution in initiale est optimale ?
- Avec une limite raisonnable à nbMaxNodeBB (100,1000), a t'on une heuristique globale efficace en utilisant des heuristiques sur des noeuds de l'arbre de B&B ? Comment se situent de telles heuristiques par rapport aux approches précédentes ?

Coupes d'intégrité, généralités

Partant d'un programme mixte en nombre entiers :

$$\begin{aligned} & \min \sum_{i=1}^{m+p} c_i x_i \\ s.c: \quad & \sum_{i=1}^{m+p} A_{i,c} x_i \geqslant b_c \quad \forall c \in [\![1; C]\!] \\ & x \in \mathbb{N}^m \times \mathbb{R}^p \end{aligned}$$

L'équation $(*) : \sum_{i=1}^{m+p} a_i x_i \geqslant d$ est une coupe valide si l'ajout de $(*)$ ne change pas la faisabilité des solutions entières, et potentiellement élimine des solutions continues ($x \in \mathbb{R}^{m+p}$) vérifiant $\sum_{i=1}^{m+p} A_{i,c} x_i \geqslant b_c$

Algorithme de séparation : algorithme choisissant des coupes à ajouter parmi un ensemble de coupes potentiellement très grand, choix guidé par la relaxation continue.

Coupes d'intégrité sur l'exemple déroulé

$$\begin{aligned} \min \quad & x_1 + 1,5x_2 + 2x_3 + 5x_4 + 9x_5 \\ s.c. \quad & x_1 + 2x_2 + 3x_3 + 6x_4 + 10x_5 \geqslant 10 \\ & \forall i, \quad x_i \in \{0,1\} \end{aligned}$$

Solution continue : $x_2 = x_3 = 1,\ x_1 = x_5 = 0,\ x_4 = \frac{5}{6}$
Relâché continu : $1,5 + 2 + \frac{25}{6} \approx 7,67$

$$x_4 + x_5 \geqslant 1$$

Solution continue : $x_3 = x_4 = 1,\ x_2 = \frac{1}{2},\ x_1 = x_5 = 0$
Relâché continu : $5 + 2 + 0,75 = 7,75$

$\Longrightarrow$ La coupe aurait permis d'énumérer un cas en moins, on retrouvait directement un noeud visité.

Généralisation : coupes de couvertures

Les coupes de couvertures s'appliquent à des contraintes de sac à dos $\sum_i a_i x_i \leqslant d, a_i \geqslant 0$.

Une couverture : un ensemble C tq $\sum_{i \in C} a_i > d$.

On alors la contrainte :

$$\sum_{i \in C} a_i > d \Longrightarrow \sum_{i \in C} x_i \leqslant |C| - 1$$

Pour des contraintes $\sum_i a_i x_i \geqslant d, a_i \geqslant 0$, $x_i \leftrightarrow 1 - x_i$ on a :

$$\sum_{i \in C} a_i < d \Longrightarrow \sum_{i \notin C} x_i \geqslant 1$$

NB : la relaxation continue guide le choix de la couverture C à utiliser.

Expériences numériques avec les coupes ?

Pour ajouter les de coupes de couvertures à l'algorithme B&B du cours, il faudrait savoir résoudre à chaque noeud des problèmes de type :

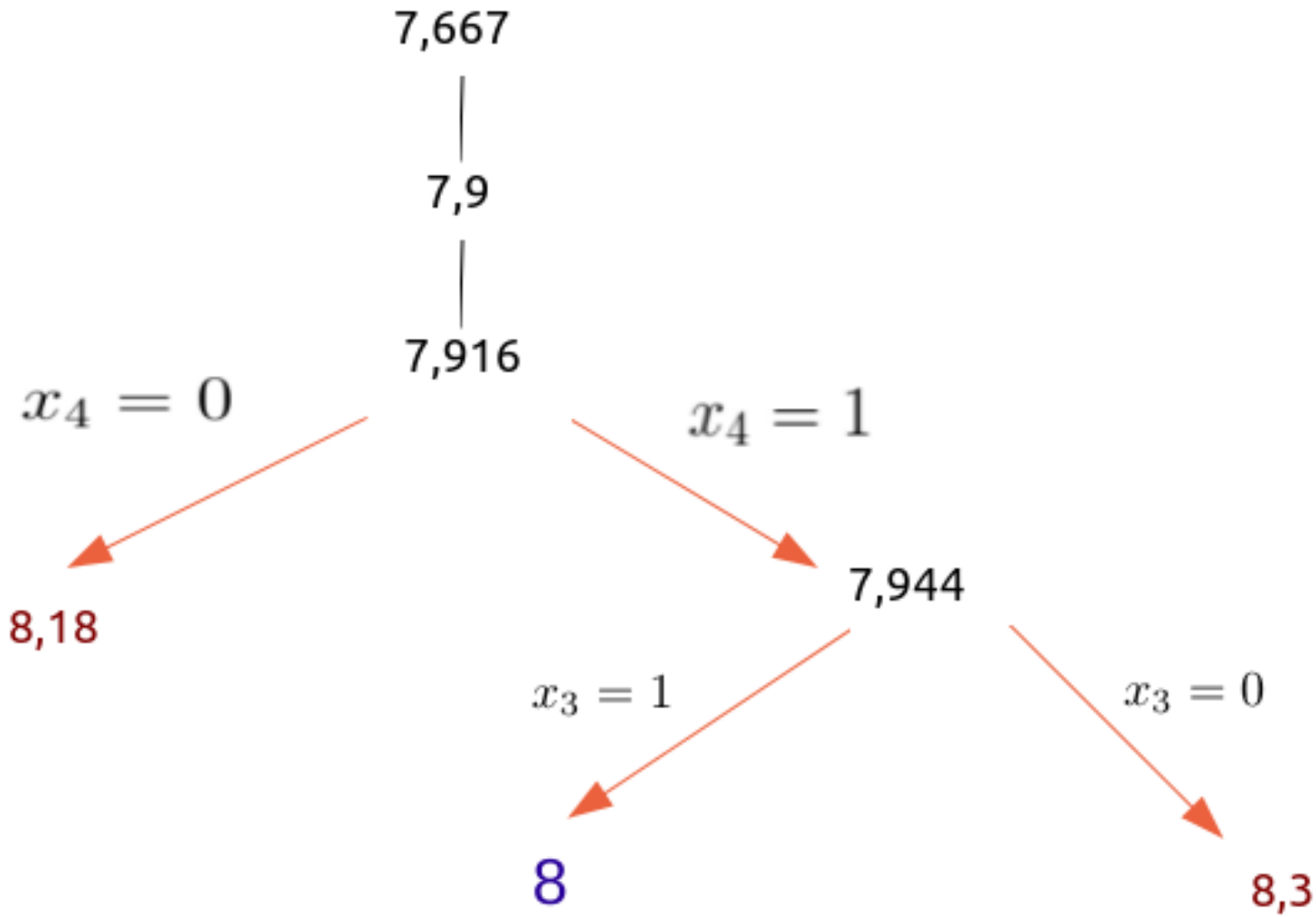
$$\begin{aligned} & \max_{x_i} \sum_{i=1}^{N} c_i x_i \\ s.c : \quad & \sum_{i=1}^{N} m_{i,c} x_i \leqslant M_c \quad \forall c \in [\![1, C]\!] \\ & x_i \in \{0, 1\} \qquad\qquad \forall i \in [\![1, N]\!] \end{aligned} \tag{10}$$

où $m_{i,c} \geqslant 0$ pour tout i, c

Il existe un algorithme pour cela, la méthode des dictionnaires (hors programme pour ce cours)

Remarque : il en serait de même pour implémenter un algorithme de Branch&Bound sur les variantes de problèmes de sac à dos plusieurs contraintes de sac à dos, ou les sac à dos à choix multiple (versions d'architecture automobiles)

$\Longrightarrow$ on verra ça avec GLPK, sur la seconde partie des TP



Sur cet exemple, la réduction peut paraître anodine. L'algorithme de B&B devient impraticable quand la taille de l'arbre augmente de manière exponentielle. Avoir des noeuds éliminés plus tôt peut avoir beaucoup d'impact sur l'explosion exponentielle de l'arbre.

Plan

Bilan et perspectives

On a vu quatre famille d'algorithmes donnant un encadrement de très bonnes qualité de solutions optimales de problèmes de sac à dos :

1. **Algos gloutons** : algos rapide en $O(N \log N)$, donne un encadrement assez bon.
2. **Algo de prog dynamique**, exact mais non polynomial. La valeur optimale n'est obtenue qu'à la toute fin de l'algo donc si le calcul est tronqué il n'y a pas de solution heuristique.
3. **Heuristique Kernel Search** par prog dynamique : approche intermédiaire, on peut contrôler le temps de calcul pour avoir de meilleures solutions que l'approche gloutonne, sans améliorer les bornes supérieures gloutonnes
4. **Algorithme Séparation-Evaluation (Branch&Bound)** mode exact à complexité potentiellement exponentielle et mode heuristique par calcul tronqué, contrôlable, et améliore l'encadrement glouton.

Aux prochains cours nous verrons comment tout cela se généralise pour résoudre des PLNE quelconques.