Stage Optimisation

Javier Peña Castaño

August 2025

Introduction

Le problème du sac à dos quadratique binaire (0–1 QKP) consiste à maximiser une fonction quadratique pseudo-booléenne avec des coefficients positifs, sous une contrainte de capacité linéaire.

Dans ce rapport, nous présentons une méthode exacte pour calculer une borne supérieure pour le QKP, dérivée d'une décomposition lagrangienne afain de résoudre ce problème. Elle permet de trouver l'optimum pour des instances comportant jusqu'à 150 variables, quelle que soit leur densité, et jusqu'à 300 variables pour des densités moyennes ou faibles.

Dans le cadre de ce stage, nous allons nous concentrer sur le calcul de la borne supérieure (UB), étape par étape, en utilisant la décomposition lagrangienne.

Voici la formulation du problème :

$$Maximiser f(x) = \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij} x_i x_j$$

$$S.C \sum_{i=1}^{n} a_i x_i \leq b, \quad x_i \in \{0, 1\}$$

Voici quelques notations:

- $X = \{x_1, x_2, \ldots, x_n\}$ est l'ensemble des variables du problème,
- $\{X_1, X_2, \ldots, X_p\}$ est une partition de X $(1 \leq p \leq n)$.
  Chaque X_k est appelé le *cluster* k,
- $Y_k = X \setminus X_k$,
- I_k (resp. J_k) est l'ensemble des indices des variables de X_k (resp. Y_k),
- x_{I_k} est le vecteur des variables x_i, $i \in I_k$ (appelées variables *compliquantes*),
- $cl(i)$ est l'indice du cluster contenant la variable x_i.

Semaine 1

Lecture de l'article de base sur le stage, compréhension de toutes les méthodes et familiarisation avec le sujet. Recherches complémentaires en lien avec cette thématique.

Semaine 2

Début de l'implémentation du code du calcul de la première borne supérieure, en divisant le problème en clusters de 5 variables. Ensuite, on calcule la borne supérieure de chaque cluster par énumération, en calculant sépraément la partie linéaire :

$$\sum_{i=1}^{n} c_i x_i$$

et la partie quadratique:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij} x_i x_j$$

tout en respectant la contrainte suivante :

$$\sum a_i x_i \leq b$$

La première borne supérieure va correspondre à la somme des bornes supérieures de chaque cluster, qui va être toujours supérieur a la borne optimale recherché.

Semaine 3

Codage des étapes 3, 4 et 5 du calcul de la borne supérieure

Durant cette semaine, j'ai implémenté les étapes suivantes du calcul de la borne supérieure dans le cadre de la décomposition lagrangienne.

Implémentation de la fonction $L_k(x_{I_k}, y^k_{J_k}, \lambda, \mu)$

La fonction suivante a été codée :

$$\begin{aligned} L_k(x_{I_k}, y^k_{J_k}, \lambda, \mu) = & \sum_{i \in I_k} \left(c_i + \sum_{h \neq k} \lambda_i^h \right) x_i - \sum_{j \in J_k} \lambda_j^k y_j^k \\ & + \sum_{i \in I_k} \sum_{i' \in I_k i' \neq i} \frac{1}{2} c_{ii'} x_i x_{i'} \\ & + \sum_{i \in I_k} \sum_{j \in J_k j > i} \left(\frac{1}{2} c_{ij} + \mu_{ij} \right) x_i y_j^k + \sum_{i \in I_k} \sum_{j \in J_k j < i} \left(\frac{1}{2} c_{ij} - \mu_{ij} \right) x_i y_j^k. \end{aligned}$$

Cette fonction est essentielle car elle est utilisée dans le calcul de la borne supérieure duale via la fonction $w(\lambda, \mu)$ suivante :

$$w(\lambda, \mu) = \sum_{k=1}^{p} \max \left\{ L_k(x_{I_k}, y^k_{J_k}, \lambda, \mu) \ : \ x_{I_k}, y^k_{J_k} \ ; \ S.C(4), (5), (6) \right\}.$$

Fonction duale $w(\lambda, \mu)$

La fonction duale $w(\lambda, \mu)$ est centrale dans la méthode de décomposition lagrangienne. Elle est définie comme suit :

$$w(\lambda, \mu) = \sum_{k=1}^{p} \max \left\{ L_k(x_{I_k}, y^k_{J_k}, \lambda, \mu) \ : \ x_{I_k}, y^k_{J_k} \ ; SC(4), (5), (6) \right\}.$$

Cette fonction représente une borne inférieure (dual bound) sur le problème initial, obtenue par relaxation de certaines contraintes à l'aide de multiplicateurs de Lagrange.

Implémentation Deux versions de cette fonction ont été programmées :

- **Version 1** : Évalue uniquement la valeur de $w(\lambda, \mu)$ en résolvant, pour chaque cluster k, le sous-problème associé à la fonction L_k.
- **Version 2** : En plus de la valeur de $w(\lambda, \mu)$, cette version retourne les meilleurs vecteurs x et y trouvés (solutions primales correspondantes aux maximisations locales de chaque L_k).

Étapes principales du code

1. **Boucle sur les clusters** : on parcourt les clusters X_k de la partition des variables X.
2. **Résolution du sous-problème local** :
   - Pour chaque cluster k, on construit et maximise la fonction $L_k(x_{I_k}, y^k_{J_k}, \lambda, \mu)$.
   - Cette maximisation est effectuée en respectant les contraintes du problème d'origine, limitées au cluster considéré.
3. **Agrégation des résultats** :
   - On somme les valeurs maximales obtenues pour chaque cluster.
   - Cela donne la valeur de la fonction duale $w(\lambda, \mu)$.
4. **Sauvegarde des argmax** :
   - Si la version enrichie est utilisée, les vecteurs x_{I_k} et $y^k_{J_k}$ maximisant chaque L_k sont enregistrés et retournés avec la valeur duale.

Utilité Cette fonction est appelée à chaque itération de l'algorithme de sous-gradient pour :

- Évaluer la qualité des multiplicateurs de Lagrange actuels,
- Calculer les sous-gradients (différences entre les solutions couplées et découplées),
- Et mettre à jour ces multiplicateurs afin de progresser vers un optimum dual.

Algorithme de sous-gradient

Enfin, un algorithme de sous-gradient a été implémenté pour minimiser la fonction duale $w(\lambda, \mu)$. Voici le principe de fonctionnement de l'algorithme, basé sur la fonction `subgradient_solve(...)` :

1. Initialisation des multiplicateurs de Lagrange λ et μ.
2. À chaque itération :
   - Évaluation de la fonction duale $w(\lambda, \mu)$ à l'aide de la fonction `solve_dual_primal(...)`.
   - Calcul des sous-gradients :
     - Pour λ, via $g_\lambda^{k,j} = x_j^* - y_j^{k*}$.
     - Pour μ, via $g_\mu^{i,j} = x_i y_{ji} - x_j y_{ij}$, à partir des clusters et des indices associés.
   - Mise à jour des multiplicateurs selon la règle :

$$\lambda^{t+1} = \max(0, \lambda^t - \alpha_t \cdot g_\lambda^t), \quad \mu^{t+1} = \mu^t - \alpha_t \cdot g_\mu^t$$

   - Réduction du pas de mise à jour : $\alpha_{t+1} = p \cdot \alpha_t$, avec typiquement $p = 0.9$.
3. La meilleure valeur de $w(\lambda, \mu)$ rencontrée est conservée, ainsi que les multiplicateurs associés.

Ce processus est répété jusqu'à convergence ou jusqu'au nombre maximal d'itérations. Cet algorithme permet d'obtenir une bonne borne inférieure sur la solution optimale.

Installation du solveur CBC

Pour résoudre les sous-problèmes d'optimisation (lors du calcul de $\max L_k$ et l'algo de sous-gradient), le solveur open-source **CBC (Coin-or branch and cut)** a été installé et configuré dans l'environnement de développement.

Semaine 4

Amélioration de l'algorithme de sous-gradient

Durant cette semaine, j'ai introduit deux nouvelles stratégies de réduction de pas dans l'algorithme de sous-gradient, en suivant les méthodes classiques de la littérature :

Méthodes de réduction de pas

Méthode 2 (série convergente)
Le pas est défini comme :

$$\lambda_k = \lambda_0 \cdot \alpha^k, \quad avec \quad 0 < \alpha < 1$$

Cette méthode permet une décroissance exponentielle du pas. Elle est souvent utilisée lorsqu'on ne connaît pas de borne inférieure fiable. Elle est citée par Shor (1968) et Goffin (1977).

Méthode 3 (relaxation)
Ici, le pas est défini par :

$$\lambda_k = \rho \cdot \frac{f(x^k) - \bar{f}}{\|\gamma^k\|}$$

où :

- $\bar{f}$ est une estimation de la valeur optimale f^* (borne inférieure),
- ρ est un coefficient de relaxation, souvent ≤ 2,
- γ^k est le sous-gradient à l'itération k.

Cette méthode est plus agressive et efficace lorsque l'on dispose d'une bonne estimation de la solution optimale.

Heuristique pour l'estimation de la borne inférieure

Pour pouvoir appliquer efficacement la méthode 3, j'ai implémenté une heuristique de type glouton pour estimer $\bar{f}$. Deux versions sont proposées :

1. `greedy_linear` Cette heuristique fonctionne en $O(n \log n)$:
   1. Les objets sont triés selon le rapport c_i/a_i de façon décroissante.
   2. Les objets sont ajoutés un à un tant que la contrainte de capacité est respectée.
   3. La fonction objectif (linéaire + quadratique) est ensuite évaluée.

   Code simplifié :

```
order = sortperm(c ./ a, rev=true) # indices triés par ratio
for idx in order
    if a[idx] <= cap
        X[idx] = true
        cap -= a[idx]
    end
end
```

2. `greedy_plus_oneflip` Cette version ajoute une amélioration locale à `greedy_linear` :

1. Applique `greedy_linear` pour obtenir une solution initiale.
2. Essaye d'activer un seul item inactif supplémentaire, s'il y a un gain marginal positif, sans dépasser la capacité.

Extrait du code :

```
for idx in 1:n
    if !X[idx] && a[idx] <= cap
         = c[idx] + sum(Q[idx,k] for k in 1:n if X[k])
        if  > 0
            X[idx] = true
            fbar +=
            break
        end
    end
end
```

Tests et modifications

Enfin, j'ai modifié le système de test pour forcer certaines décisions :

- Certaines variables sont obligatoirement activées ou désactivées dans la solution initiale.
- Cela permet de tester des scénarios plus contrôlés, utiles pour évaluer la robustesse des heuristiques et la qualité de la borne $\bar{f}$.

Toutes ces implémentations sont documentées et intégrées dans le fichier `DecompositionLagrangienne.ipynb`.

Semaine 5

Création d'une instance de test réaliste

Pour tester la robustesse et la scalabilité de mes fonctions, j'ai construit un exemple à $n = 30$ variables avec les caractéristiques suivantes :

- **Coûts linéaires** $c = [1.0, 2.0, \ldots, 30.0]$.
- **Matrice de couplage** Q en chaînes par blocs :
  - poids 1.0 pour les couples $(i, i+1)$ si $i \leq 10$,
  - poids 2.0 si $10 < i \leq 20$,
  - poids 3.0 au-delà.
  - Quelques arêtes supplémentaires croisées : $(5, 15)$, $(10, 20)$, $(12, 25)$.
- **Poids** a_i : tous initialisés à 2.0, avec deux variables par cluster fixées à 1.0 pour introduire de la diversité.
- **Capacité** b : fixée à 40% de la somme des poids.
- **Clusters** : formés par groupes de 5 variables.
- **Initialisation des multiplicateurs** $\lambda_{k,j}$ et $\mu_{i,j}$ à zéro.

Correction du calcul des sous-gradients

Une erreur a été corrigée dans le calcul de g_λ, le sous-gradient associé à $\lambda_{k,j}$:

- **Avant** : le code ne récupérait pas correctement la valeur globale x_j^*.
- **Après** : on utilise les dictionnaires `cluster_of` et `pos_in_cluster` pour localiser précisément la position de x_j dans son cluster :

```
kj = cluster_of[j]
pj = pos_in_cluster[j]
xj = sol.xstar[kj][pj]
val = xj - yk[idx]
glambda[(k,j)] = val
```

Début de la stratégie de fixation de variables

J'ai commencé à traduire en Julia des fonctions écrites initialement en C, dédiées à la **fixation de variables** binaires. L'objectif est de réduire la taille du problème sans perdre d'optimalité, en exploitant des bornes supérieures.

Structure utilisée

On introduit une structure `VarFixInfo` qui permet de stocker pour chaque variable :

- si une information de fixation existe,
- une borne inférieure et une borne supérieure fixées (`xi_binf`, `xi_bsup`).

Fonctions principales

- `rewrite_problem(...)` : permet de fixer une variable $x_i = 0$ ou 1, en réécrivant l'instance du problème et en mettant à jour :
  - le vecteur des coûts c,
  - la matrice Q (symétrique),
  - les contraintes et le côté droit.

  Le terme constant généré est mémorisé et ajouté à l'évaluation finale.
- `calcul_borne2(...)` : appel a la fonction initial `compute_first_ub(...)`
- `calcul_borne2(...)` : calcule une borne supérieure par une relaxation lagrangienne du problème modifié.
- `fix_variables_0(...)` : tente de fixer par paquets de 2 variables toutes celles pour lesquelles $x_i^* \approx 0$, si la borne associée devient inférieure à la borne inférieure connue.
- `fix_variables_01(...)` : essaie de fixer chaque variable individuellement à 0 puis à 1, et vérifie à chaque fois si la borne supérieure obtenue reste compatible. Si oui, la variable est fixée.

Ces outils sont précieux pour améliorer l'efficacité du solveur, surtout dans les grandes instances. Leur intégration progressive dans l'algorithme principal est en cours.

Semaine 6

Développement d'heuristiques supplémentaires pour le calcul de bornes inférieures

Durant cette semaine, deux nouvelles heuristiques ont été implémentées à partir des algorithmes décrits dans le rapport théorique .

1. Heuristique gloutonne améliorée (`greedy_heuristic`)

Cette heuristique reproduit en Julia l'algorithme suivant:

- Tous les objets sont initialement placés dans le sac (ensemble K_1).
- Tant que la contrainte de capacité est violée, on retire l'objet ayant le plus faible ratio c_j.
- Les coûts c_j sont mis à jour en fonction des interactions quadratiques avec les autres objets du sac.

- Ensuite, on tente de rajouter les objets restants (dans K_0) ayant un ratio amélioré $c_j > 0$.

L'objectif est de construire une solution admissible avec un bon compromis linéaire-quadratique.

L'implémentation calcule ensuite la valeur complète de la fonction objectif pour évaluer la borne inférieure correspondante. La complexité reste raisonnable grâce à l'approche incrémentale.

2. Recuit simulé (`simulated_annealing`)

Cette deuxième méthode est une méta-heuristique basée sur le **recuit simulé**. Elle permet de sortir de minima locaux et de mieux explorer l'espace des solutions admissibles.

Le fonctionnement est le suivant :

1. Initialisation avec une solution admissible x (par exemple celle produite par `greedy_heuristic`).
2. Une température τ est initialisée à une valeur haute.
3. Pour chaque palier de température :
   - Un certain nombre d'itérations sont effectuées (L).
   - À chaque itération, un voisin y de x est généré en échangeant un article du sac avec un article hors du sac.
   - Si y est faisable et améliore la fonction, il est accepté.
   - Sinon, il est accepté avec une probabilité dépendant de la détérioration Δ et de la température τ :

$$proba = e^{-\Delta/\tau}$$

4. La température est réduite progressivement : $\tau \leftarrow \rho \cdot \tau$.
5. À la fin, la meilleure solution rencontrée est retournée.

Cette méthode peut trouver de meilleures bornes inférieures (LB), au prix d'un coût de calcul plus élevé. Dans les tests effectués, elle peut prendre plus d'une minute pour 100 variables, contre moins d'une seconde pour l'heuristique gloutonne seule.

En résumé

Ces deux heuristiques offrent un bon compromis entre qualité de la borne inférieure et temps de calcul. La première (`greedy_heuristic`) est rapide et fiable, tandis que la seconde (`simulated_annealing`) permet d'explorer plus largement les configurations admissibles, en contournant les pièges de l'optimisation locale. Avec ces deux heuristiques on obtient une meilleure borne inférieur qu'avec les heuristiques implémentées la semaine 4.

Développement d'un solveur exact TSP avec Bonobo.jl (prototype)

En parallèle, j'ai commencé le développement d'un **solveur exact pour le TSP (voyageur de commerce)** basé sur le framework `Bonobo.jl`. L'objectif est de répliquer et personnaliser l'approche décrite dans :
https://opensourc.es/blog/tspsolver.jl-using-bonobo.jl-to-solve-our-first-instance

Structure du solveur

Le solveur s'appuie sur :

- une structure `Root` contenant le graphe complet et la matrice de coûts,
- des `Node` représentant les nœuds de l'arbre BnB avec :
  - une tournée candidate (`tour`),
  - un arbre couvrant minimal (`mst`),
  - les arêtes fixées et interdites.
- un `evaluate_node!` qui utilise un 1-tree pour calculer la LB, et une heuristique gloutonne pour l'UB,
- une stratégie de branchement personnalisée `LONGEST_EDGE` qui sélectionne l'arête la plus longue non encore fixée dans le MST.

Problème rencontré : branchements infinis

Malgré un comportement initial correct, le solveur entre dans une boucle infinie. Le message suivant est affiché à de nombreuses reprises :

```
No branching possible - all edges are fixed or disallowed
```

Cela signifie que la fonction `get_branching_variable` ne trouve plus d'arêtes éligibles, mais que Bonobo continue tout de même à explorer des nœuds (probablement vides). Ce comportement vient de deux causes :

- La fonction `get_branching_nodes_info` retourne quand même deux enfants, même si aucun branchement réel n'est possible.
- Le système ne détecte pas explicitement que le nœud courant est terminal.

Solutions possibles :

- Ajouter une condition dans `get_branching_nodes_info` pour retourner `[]` lorsque `branching_edge === nothing`.
- Utiliser `return nothing` dans la stratégie pour signaler l'impossibilité de branched.
- S'assurer que les UB/LB `NaN` soient correctement interprétés par Bonobo comme invalides.

Références

- Alain Billionnet, Éric Soutif, *An exact method based on Lagrangian decomposition for the 0–1 quadratic knapsack problem*, *European Journal of Operational Research*, Vol. 157, 2004, pp. 565–575.