Analyse et Évaluation de Métaheuristiques pour le Problème du Sac à Dos Multidimensionnel 0-1

Elias BOULANGER, Thomas AUBERT, Rania BADI, Izaak Aubert-Mécibah

Keywords: Ce document présente une analyse approfondie d'un code source en langage C destiné à résoudre le problème du sac à dos multidimensionnel 0-1 par divers algorithmes metaheuristiques (recherche locale, VND, VNS, descente de gradient et algorithme génétique). Nous détaillons la représentation des données, les choix algorithmiques, et estimons la complexité spatiale et temporelle des principales sous-routines. Des tableaux comparatifs expérimentaux illustrent la performance (valeurs de solutions et temps CPU) obtenue à chaque étape du processus d'optimisation.

1 Introduction

Le problème du sac à dos multidimensionnel 0-1 (MKP) consiste à sélectionner un sous-ensemble d'objets de valeur c_j et de poids multiples, de sorte que la somme des poids pour chaque contrainte ne dépasse pas une capacité donnée, tout en maximisant la valeur totale. Ce problème est NP-difficile et apparaît dans de nombreux domaines d'application. Dans le cadre de ce travail, nous avons développé plusieurs métaheuristiques en langage C afin de déterminer une solution (quasi-)optimale pour le MKP. Notre approche intègre notamment :

— La lecture et la représentation des données du problème,
— La construction d'une solution initiale aléatoire,
— La vérification de la faisabilité d'une solution,
— Des procédures de recherche locale (1-flip et swap),
— Des méthodes de descente de voisinage variable (VND) et de recherche à voisinage variable (VNS),
— Une descente de gradient avec mécanisme de réparation,
— Un algorithme génétique (GA) et un schéma multi-start combinant plusieurs méthodes.

L'objectif de ce document est d'analyser en profondeur les choix de conception, la complexité algorithmique et spatiale des différentes sous-routines, et de proposer quelques perspectives d'amélioration.

2 Données et Méthodes

2.1 Structures de Données

La représentation du problème s'effectue via la structure `Problem` qui contient :

— n : le nombre d'objets,
— m : le nombre de contraintes,
— `c` : un tableau de flottants de taille n pour les coefficients (valeurs des objets),
— `capacities` : un tableau de taille m pour les capacités,
— `weights` : un tableau de taille $m \times n$ stocké en ordre ligne, représentant le poids de chaque objet pour chaque contrainte,
— `sum_of_weights` et `ratios` : pré-calculés pour faciliter la recherche locale,
— `candidate_list` : liste d'indices triés selon le ratio valeur/poids, heuristique utilisée pour orienter les choix dans les recherches locales.

La solution candidate est représentée par la structure `Solution` :

— `x` : un vecteur binaire (représenté en flottants 0.0 ou 1.0) de taille n,
— `value` : la valeur objective de la solution,
— `feasible` : un booléen indiquant la faisabilité de la solution.

Analyse de complexité spatiale : La structure principale occupe de l'espace proportionnel à $O(n \times m)$ (pour le tableau des poids) plus $O(n)$ pour les autres tableaux. Ceci est raisonnable pour des instances de taille modérée, mais peut devenir prohibitif pour des très grandes instances, ou lors d'allocations successives de nombreuses solutions.

2.2 Algorithmes et Sous-Routines

Construction de la solution initiale La fonction `construct_initial_solution` réalise une initialisation aléatoire (ou éventuellement gloutonne via plusieurs starts). La complexité est de l'ordre de $O(n)$ par solution, et le choix de plusieurs démarrages permet de diversifier la recherche initiale.

Vérification de la faisabilité La fonction `check_feasibility` parcourt chaque contrainte et calcule la somme des poids pour vérifier si elle est inférieure ou égale à la capacité correspondante. La complexité est de $O(m \times n)$.

2.2.1 Recherche Locale (LS)

Nous présentons ci-dessous le pseudo-code de chacune des procédures de recherche locales ainsi que leur complexité.

LS 1-FLIP Pour chaque itération, on parcourt au maximum k candidats ($k \leq n$) et on réalise une copie en $O(n)$. Si un candidat est viable, la mise à jour de l'utilisation (ou sa vérification) se fait en $O(m)$. Ajoutons également le cout de la réparation $O(n \times (n + 2m))$ qui est détaillée ci-après. Ainsi, le coût d'une itération est de $O(n^2 + 2(mn + m + n))$ avec $k = n$. Il est très difficile d'estimer le temps ou le nombre d'itérations de la boucle globale, c'est pour cela qu'il existe un temps maximal d'exécution. Notons que les prévisions pessimistes sont assez éloignés des cas réels qui ressemblent plutôt à deux phases :

1. Réalise autant de FLIP $0 \rightarrow 1$ que possible en sélectionnant dans l'ordre les éléments les plus intéressant par ratio.
2. Quand une ou plusieurs capacités sont dépassés, enchaîne les réparations et de nouveaux FLIPs $0 \rightarrow 1$ tant qu'ils améliorent.

Les phases une et deux sont simplifiés en général en complexité par les tests $0\ ou\ 1$. C'est surtout la deuxième phase qui est couteuse, se rapprochant de la courbe asymptotique.

LS SWAP Tel que précédemment, on copie en $O(n + m)$. Le parcours des couples se fait dans le pire des cas en $O(n^2)$, la mise à jour de l'utilisation se fait en $O(m)$. Ainsi, le coût d'une itération est de $O(n(n + 1) + m + cout\_repair)$. Les commentaires sont similaires à la version *FLIP*, remarquons cependant que *SWAP* est plus coûteuse dû à sa gestion via des couples et non des items indépendants. Elle permet cependant d'explorer un autre voisinage que *FLIP* qui reste assez simple.

Fonction de Réparation

— La vérification de la faisabilité des contraintes se fait en $O(m)$.
— La recherche de l'objet à retirer nécessite de parcourir n objets, soit $O(n)$.
— La mise à jour de l'utilisation se fait en $O(m)$.

Ainsi, chaque itération coûte $O(m + n + m) = O(n + 2m)$. Dans le pire des cas, si l'on doit retirer jusqu'à n objets, la complexité totale est de $O(n \times (n + 2m))$.

Algorithm 1 LS_Flip(*prob, sol, k, mode*)

```
1: procedure LS_FLIP(prob, sol, k, mode)
2:     current_usage ← Calcul_Usage(sol)                                  ▷ Coût : O(m)
3:     current_value ← Valeur(sol)
4:     while Amélioration détectée do
5:         candidate_sol ← Copie(sol)                                     ▷ Coût : O(n)
6:         candidate_usage ← Copie(current_usage)                         ▷ Coût : O(m)
7:         best_item ← −1, best_value_increase ← 0
8:         for idx ← 1 to k do                                            ▷ Parcourt k candidats
9:             j ← candidate_list[idx]
10:            if sol.x[j] = 0 then
11:                delta ← c[j]
12:                if current_value + delta > current_value then
13:                    if mode = FIRST_IMPROVEMENT then
14:                        best_item ← j
15:                        break
16:                    else if mode = BEST_IMPROVEMENT then
17:                        if delta > best_value_increase then
18:                            best_item ← j
19:                            best_value_increase ← delta
20:                        end if
21:                    end if
22:                end if
23:            end if
24:        end for
25:        if best_item = −1 then
26:            break                                                      ▷ Aucune amélioration trouvée
27:        end if
28:        candidate_sol.x[best_item] ← 1
29:        candidate_usage ← Mise_à_jour(candidate_usage, poids[*, best_item])   ▷ Coût :
   O(m)
30:        if (candidate_usage dépasse les capacités) then
31:            candidate_sol ← Repair_Solution(prob, candidate_sol, candidate_usage)
32:        end if
33:        if Valeur(candidate_sol) > current_value then
34:            sol ← candidate_sol
35:            current_usage ← candidate_usage
36:            current_value ← Valeur(sol)
37:        end if
38:    end while
39: end procedure
```

Algorithm 2 LS_Swap(*prob, sol, k, mode*)

```
1: procedure LS_SWAP(prob, sol, k, mode)
2:     current_usage ← Calcul_Usage(sol)
3:     current_value ← Valeur(sol)
4:     while Amélioration détectée do
5:         candidate_sol ← Copie(sol)                                   ▷ Coût : O(n)
6:         candidate_usage ← Copie(current_usage)                       ▷ Coût : O(m)
7:         best_i ← −1, best_j ← −1, best_delta ← 0
8:         for i ← 1 to n do                                            ▷ Parcourt jusqu'à n items
9:             if sol.x[i] = 1 then
10:                for idx ← 1 to k do                                  ▷ Parcourt k candidats
11:                    j ← candidate_list[idx]
12:                    if sol.x[j] = 0 then
13:                        delta ← c[j] − c[i]
14:                        if delta > 0 then
15:                            if mode = FIRST_IMPROVEMENT then
16:                                best_i ← i, best_j ← j, best_delta ← delta
17:                                break out
18:                            else if mode = BEST_IMPROVEMENT then
19:                                if delta > best_delta then
20:                                    best_i ← i, best_j ← j, best_delta ← delta
21:                                end if
22:                            end if
23:                        end if
24:                    end if
25:                end for
26:            end if
27:            if premier cas d'amélioration trouvé then
28:                break
29:            end if
30:        end for
31:        if best_i = −1 ou best_j = −1 then
32:            break
33:        end if
34:        candidate_sol.x[best_i] ← 0, candidate_sol.x[best_j] ← 1
35:        candidate_usage ← Mise_à_jour(candidate_usage, −poids[∗, best_i] +
   poids[∗, best_j])                                                    ▷ Coût : O(m)
36:        if (candidate_usage dépasse les capacités) then
37:            candidate_sol ← Repair_Solution(prob, candidate_sol, candidate_usage)
38:        end if
39:        if Valeur(candidate_sol) > current_value then
40:            sol ← candidate_sol
41:            current_usage ← candidate_usage
42:            current_value ← Valeur(sol)
43:        end if
44:    end while
45: end procedure
```

Algorithm 3 Repair_Solution(*prob, sol, usage, cur_value*)

```
1:  procedure REPAIR_SOLUTION(prob, sol, usage, cur_value)
2:      for iteration ← 1 to n do                                  ▷ jusqu'à n itérations
3:          if Usage est faisable pour toutes les contraintes then  ▷ Coût : O(m)
4:              sol.feasible ← vrai
5:              return
6:          end if
7:          worst_item ← −1, worst_ratio ← −∞
8:          for j ← 1 to n do                                      ▷ jusqu'à n itérations
9:              if sol.x[j] = 1 then
10:                 ratio ← c[j]/(sum_of_weights[j] + ϵ)
11:                 if ratio < worst_ratio ou worst_item = −1 then
12:                     worst_ratio ← ratio, worst_item ← j
13:                 end if
14:             end if
15:         end for
16:         if worst_item = −1 then
17:             return                                             ▷ Aucun objet à retirer
18:         end if
19:         sol.x[worst_item] ← 0
20:         cur_value ← cur_value − c[worst_item]
21:         for i ← 1 to m do                                      ▷ Coût : O(m)
22:             usage[i] ← usage[i] − poids[i, worst_item]
23:         end for
24:     end for
25: end procedure
```

Descente de Voisinage Variable (VND) La fonction `vnd` combine les deux recherches locales (flip et swap). La stratégie consiste à appliquer successivement ces deux méthodes et à recommencer la recherche dans le voisinage le plus petit dès qu'une amélioration est trouvée.

Forces : Approche simple et modulaire permettant d'exploiter plusieurs voisinages.

Limites : La structure actuelle se repose sur seulement deux voisinages (via flip et swap) ce qui pourrait limiter la diversification.

Sa complexité est difficile à estimer, mais elle dépend directement des coûts combinés *1-flip* et *swap*

Recherche à Voisinage Variable (VNS) La fonction `vns` utilise une procédure de perturbation (shake) combinée à la VND pour échapper aux minima locaux. La phase de `Shake` consiste à effectuer k flips aléatoires dans la solution, ce qui coûte $O(k)$ pour modifier le vecteur solution, plus $O(m)$ pour recalculer l'utilisation en cas de réparation (si nécessaire).

Forces : La perturbation permet d'explorer de nouvelles régions de l'espace de recherche. Tant que la solution n'est pas améliorée, on augmente l'entropie.

Limites : Le mécanisme de shaking et le nombre d'itérations sans amélioration doivent être soigneusement calibrés pour éviter une exploration trop aléatoire ou une convergence prématurée.

Algorithm 4 VND(*prob, sol, max_no_improv, ls_k, ls_mode, start, max_time*)

initialiser no_improv $\leftarrow 0$
while no_improv $<$ max_no_improv **et** `time_is_up(start, max_time)` **= false do**
    improved $\leftarrow$ **false**
    candidate_sol $\leftarrow$ COPYSOLUTION(sol) ▷ Appliquer d'abord 1-flip
    LS_FLIP(prob, candidate_sol, ls_k, ls_mode)
    if VALUE(candidate_sol) $>$ VALUE(sol) **then**
        sol $\leftarrow$ candidate_sol
        improved $\leftarrow$ **true**
    else
        candidate_sol $\leftarrow$ COPYSOLUTION(sol) ▷ Puis appliquer swap
        LS_SWAP(prob, candidate_sol, ls_k, ls_mode)
        if VALUE(candidate_sol) $>$ VALUE(sol) **then**
            sol $\leftarrow$ candidate_sol
            improved $\leftarrow$ **true**
        end if
    end if
    if improved **then**
        no_improv $\leftarrow 0$
    else
        no_improv $\leftarrow$ no_improv + 1
    end if
end while

Algorithm 5 VNS(*prob, sol, max_no_improv, k_max, ls_k, ls_mode, start, max_time, verbose*)

iter $\leftarrow 0$, no_improv $\leftarrow 0$
candidate_sol $\leftarrow$ COPYSOLUTION(sol)
while no_improv $<$ max_no_improv **do**
    k $\leftarrow 0$
    while k $\leq$ k_max **do**
        candidate_sol $\leftarrow$ SHAKE(prob, sol, k)
        VND(prob, candidate_sol, 5, ls_k, ls_mode, start, max_time)
        if VALUE(candidate_sol) $>$ VALUE(sol) **then**
            sol $\leftarrow$ candidate_sol
            k $\leftarrow 0$
        else
            k $\leftarrow$ k + 1
        end if
    end while
    if Amélioration trouvée dans cette itération **then**
        no_improv $\leftarrow 0$
    else
        no_improv $\leftarrow$ no_improv + 1
    end if
    iter $\leftarrow$ iter + 1
end while

Descente de Gradient (GD) La méthode de descente de gradient avec mécanisme de momentum permet d'optimiser une version continue du problème en utilisant une fonction sigmoïde pour obtenir une solution 0-1. Le recours à une phase de réparation assure la faisabilité finale.

Choix de la fonction de perte : La fonction de perte utilisée est :

$$\text{loss}(x_{\text{hat}}) = -\sum_{i=1}^{n} c_i \, x_{\text{hat}}[i] + \frac{1}{2}\lambda \sum_{j=1}^{m} \max\{0,\ \text{usage}[j] - capacity[j]\}.$$

Ce choix traduit une relaxation continue du problème 0-1, où le terme $-\sum c_i \, x_{\text{hat}}[i]$ représente l'objectif à maximiser (transformé en minimisation) et le terme de pénalité est appliqué uniquement lorsque l'utilisation d'une contrainte dépasse sa capacité. L'utilisation d'un $\max\{0, \cdot\}$ assure que la pénalité n'est appliquée qu'en cas de violation, tandis que le facteur $\frac{1}{2}$ facilite le calcul de la dérivée.

Calcul du gradient : Le gradient est calculé par la chaîne de dérivation :

$$\frac{\partial \text{loss}}{\partial \theta_i} = -c_i \cdot \sigma'(\theta_i) + \sum_{j=1}^{m} \lambda \cdot w_{ji} \cdot \sigma'(\theta_i) \cdot \mathbf{1}_{\{usage[j] > capacity[j]\}},$$

où $\sigma(z)$ est la fonction sigmoïde, avec $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. Cette dérivée combine la contribution de l'objectif et celle des pénalités. Le terme $\sigma'(\theta_i)$ assure que l'optimisation est réalisée dans l'espace continu.

Choix de la sigmoïde : La fonction `sigmoid` est choisie pour sa propriété de ramener toute valeur réelle dans l'intervalle $[0, 1]$, ce qui correspond naturellement à la relaxation d'une variable binaire. Ce choix facilite l'usage d'algorithmes de descente de gradient en permettant l'optimisation sur un espace continu tout en obtenant une interprétation probabiliste de l'inclusion des objets.

Méthode de réparation et de rounding : Après la descente, les valeurs continues x_{hat} sont converties en une solution binaire en appliquant un seuil (typiquement 0,5). Si la solution obtenue n'est pas faisable, la fonction `repair_solution` intervient. Celle-ci retire successivement l'objet présentant le pire ratio $\frac{c_i}{\text{sum\_of\_weights}_i}$ jusqu'à ce que la solution devienne faisable. Ce choix, bien qu'heuristique, permet d'obtenir une solution admissible en s'appuyant sur une mesure simple du compromis valeur/poids.

Complexité : À chaque itération, l'algorithme exécute les étapes suivantes :

- Calcul de x_{hat} pour n variables : $O(n)$.
- Calcul de l'utilisation (usage) sur m contraintes et n objets : $O(m \times n)$.
- Calcul du gradient pour chaque variable (avec boucle sur m) : $O(n \times m)$.
- Mise à jour de θ et du vecteur de vitesse : $O(n)$.

Ainsi, la complexité par itération est de $O(n \times m)$.

Algorithm 6 GD(*prob, lambda, lr, max_no_improv, out_sol, verbose, start, max_time*)

procedure GD(prob, lambda, lr, max_no_improv, out_sol, verbose, start, max_time)
    $n \leftarrow$ nombre d'objets, $m \leftarrow$ nombre de contraintes
    Allouer les vecteurs $\theta[1..n]$, $v[1..n]$, $x\_hat[1..n]$, $grad[1..n]$, et le tableau $frozen[1..n]$
    Allouer le vecteur $usage[1..m]$
    for $i \leftarrow 1$ **to** n **do**
        $\theta[i] \leftarrow$ valeur aléatoire dans $[0, 1]$
    end for
    no_improv $\leftarrow 0$, iter $\leftarrow 0$, previous_loss $\leftarrow$ grande valeur (ex. 10^9)
    while no_improv $<$ max_no_improv **and** TIME_IS_UP(start, max_time) = **false do**
        for $i \leftarrow 1$ **to** n **do**
            if $frozen[i]$ est vrai **then**
                $x\_hat[i] \leftarrow 0.0 \; if \; theta[i] \leq 0.0 \; else \; 1.0$
            else
                $x\_hat[i] \leftarrow$ SIGMOID$(\theta[i])$
            end if
        end for
        $usage \leftarrow$ COMPUTE_USAGE(prob, x_hat) ▷ Coût : $O(m \times n)$
        for $i \leftarrow 1$ **to** n **do**
            if non frozen **then**
                $s \leftarrow x\_hat[i]$, $ds \leftarrow s \times (1 - s)$
                $g \leftarrow -c[i] \times ds$
                for $j \leftarrow 1$ **to** m **do**
                    $diff \leftarrow usage[j] - capacity[j]$
                    if $diff > 0$ **then**
                        $g \leftarrow g + \lambda \times w_{ji} \times ds$
                    end if
                end for
                $grad[i] \leftarrow g$
            else
                $grad[i] \leftarrow 0$
            end if
        end for
        for $i \leftarrow 1$ **to** n **do**
            if non frozen **then**
                $v[i] \leftarrow momentum \times v[i] + (1 - momentum) \times grad[i]$
                $\theta[i] \leftarrow \theta[i] - lr \times v[i]$
            end if
        end for
        if iter $>$ n_warmup (par exemple, 10 itérations) **then**
            FREEZE_HIGHEST(prob, θ, frozen)
        end if
        $L \leftarrow$ COMPUTE_LOSS(prob, lambda, x_hat, $usage$)
        if $L \geq$ previous_loss **then**
            no_improv $\leftarrow$ no_improv + 1
        else
            no_improv $\leftarrow 0$
        end if
        previous_loss $\leftarrow L$
        iter $\leftarrow$ iter + 1
    end while
end procedure

Algorithme Génétique (GA)

Analyse et Choix de l'Algorithme Génétique L'algorithme génétique repose sur les étapes classiques suivantes :

- **Initialisation aléatoire de la population** : Chaque individu est généré aléatoirement, avec une probabilité de 50% d'inclure chaque objet. Le coût de cette étape est de $O(\text{pop\_size} \times n)$.
- **Sélection par tournoi** : Pour chaque sélection, un sous-ensemble (de taille *TOURNAMENT_SIZE*) d'individus est tiré aléatoirement, puis l'individu avec la meilleure fitness est choisi. Cette opération se fait en $O(\text{TOURNAMENT\_SIZE})$ par sélection.
- **Croisement à un point** : Le croisement combine deux parents en un enfant, en copiant une partie du vecteur solution d'un parent et le reste de l'autre. Le coût est $O(n)$.
- **Mutation par flip de bits** : Chaque gène (bit) de l'individu a une probabilité de muter, ce qui se fait en $O(n)$.
- **Réparation et évaluation** : Après mutation, la solution est réparée si nécessaire (via l'heuristique de retrait basée sur le ratio valeur/poids) et évaluée. La vérification de la faisabilité et le calcul de la valeur objective ont un coût de $O(n \times m)$ par individu.

La boucle principale se répète pour *max_generations* itérations (ou jusqu'à ce que la limite de temps soit atteinte). Le coût total est donc de l'ordre de

$$O\Big(\text{max\_generations} \times \Big(\text{pop\_size} \times \big(O(n \times m) + O(n)\big)\Big)\Big).$$

La présence de la phase de tri (via `qsort`) sur un tableau de taille pop_size ajoute un coût de $O(\text{pop\_size}\log(\text{pop\_size}))$ par génération, qui reste négligeable lorsque la population est relativement grande par rapport au coût de l'évaluation.

Forces :

- Le GA offre une capacité d'exploration globale grâce à l'initialisation aléatoire et à la diversité induite par le croisement et la mutation.
- La sélection par tournoi favorise l'exploitation des meilleurs individus tout en maintenant une diversité suffisante.

Limites :

- La performance dépend fortement des paramètres (taille de la population, taux de mutation, nombre de générations).
- Le coût computationnel peut devenir élevé pour des populations et un nombre de générations importants, notamment en raison du coût $O(n \times m)$ de l'évaluation de chaque individu.

Algorithm 7 Algorithme Génétique – Boucle principale

procedure GENETICALGORITHM(prob, best_sol, pop_size, max_gens, mut_rate, start, max_time, verbose)
    $population \leftarrow$ allouer un tableau de `Individual` de taille *pop_size*
    $new\_population \leftarrow$ allouer un tableau de `Individual` de taille *pop_size*
    for $i \leftarrow 1$ **to** pop_size **do**
        Allouer la solution de $population[i]$ (de dimension n)
        Allouer la solution de $new\_population[i]$ (de dimension n)
    end for
    GA_INIT_POPULATION(prob, population, pop_size)
    for $gen \leftarrow 1$ **to** max_gens **do**
        $elite\_count \leftarrow \lceil ELITE\_PERCENTAGE \times pop\_size \rceil$
        $sorted\_pop \leftarrow$ tableau de pointeurs vers les individus de $population$, trié par fitness décroissante
        for $i \leftarrow 1$ **to** elite_count **do**
            $new\_population[i] \leftarrow$ copie de $sorted\_pop[i]$
        end for
        for $i \leftarrow elite\_count + 1$ **to** pop_size **do**
            $(parent1, parent2) \leftarrow$ GA_TOURNAMENT_SELECTION(population, TOURNAMENT_SIZE)
            $new\_population[i] \leftarrow$ GA_SINGLE_POINT_CROSSOVER(prob, parent1, parent2)
            GA_MUTATION(prob, new_population[i], mut_rate)
            GA_REPAIR(prob, new_population[i])
            GA_EVALUATE_INDIVIDUAL(prob, new_population[i])
        end for
        for $i \leftarrow 1$ **to** pop_size **do**
            GA_SWAP_INDIVIDUALS(population[i], new_population[i])
        end for
        if TIME_IS_UP(start, max_time) **is true then**
            break ▷ Arrêt si la limite de temps est atteinte
        end if
    end for
    $best\_sol \leftarrow$ individu de $population$ ayant la meilleure fitness
    Libérer la mémoire de $population$ et $new\_population$
end procedure

2.3 Choix Algorithmiques

Choix et Commentaires :

- **Représentation en C** : L'utilisation du langage C permet une gestion fine de la mémoire et une exécution rapide, mais impose une gestion manuelle de l'allocation et de la libération de la mémoire, source potentielle d'erreurs (cf. les vérifications de malloc).
- **Local Search (FLIP et SWAP)** : Ces approches sont simples à mettre en œuvre et offrent une exploitation locale efficace. Cependant, la recherche FLIP se limite à ajouter des objets, ce qui peut limiter l'exploration de l'espace de solution. L'ajout de la recherche swap améliore la diversification, mais reste limitée à des échanges simples.
- **VND et VNS** : La combinaison de plusieurs voisinages (FLIP et SWAP) dans la VND et l'ajout d'une perturbation dans la VNS offrent une bonne capacité à échapper aux minima

locaux. Toutefois, les procédures de perturbation et la stratégie de passage d'un voisinage à l'autre nécessitent un réglage des paramètres.

— **Descente de Gradient** : L'approche par optimisation continue est innovante dans le contexte du MKP, mais dépend fortement des paramètres (learning rate, lambda, nombre d'itérations), du choix de la fonction de perte, et de la procédure de réparation.

— **Algorithme Génétique** : La méthode GA implémente une sélection par tournoi, un croisement à un point et une mutation simple. La structure est classique et permet une bonne exploration globale, mais peut nécessiter un ajustement fin de la taille de la population et du taux de mutation pour obtenir de bonnes performances.

3 Résultats Expérimentaux

Dans nos expériences, nous avons comparé plusieurs approches sur 6 instances standards du MKP (proposées dans le sujet). Le tableau 1 ci-dessous présente, pour chaque instance, sur deux lignes distinctes, la valeur de l'objectif obtenu et le temps CPU (en secondes) consommé par chacune des méthodes.

TABLE 1 – Comparaison expérimentale des solutions. Pour chaque instance, les deux lignes indiquent, respectivement, la valeur de l'objectif et le temps CPU consommé (en secondes) par chaque méthode.

Instance	Initial	LS FLIP	LS SWAP	GD	VND	VNS	GA	GD+VNS	GA+VNS	Références
100M5_1	16125	17666	16818	20327	17666	23666	24167	23795	24282	24381
		0.000082	0.000095	0.000892	0.001333	1.89	4.73	1.68	6.12	
100M30_1	15682	15682	15682	18979	15682	21327	21567	21358	21582	21946
		0.000082	0.000098	0.002791	0.002592	1.953	15.997	2.823	20.134	
250M5_1	40204	40709	43377	52495	40709	57945	59167	57168	59167	59312
		0.000153	0.000657	0.00341	0.006	7.765	12.33	5.37	16.5219	
250M30_1	40489	45628	40489	48307	46536	55170	56132	55123	56177	56842
		0.000241	0.000262	0.0129	0.009	14.23	45.397	16.57	51.64	
500M5_1	86214	91106	90570	118126	94378	115859	119838	118980	119620	120148
		0.000446	0.00243	0.0114	0.025	21.21	26.759	18.48	48.6	
500M30_1	87144	87183	87144	108936	87173	112339	113659	113071	113965	116056
		0.000593	0.000856	0.0408	0.03	85.1548	95.556	32.68	120	

Analyse approfondie des résultats expérimentaux : Performance de la solution :

— Les valeurs initiales, obtenues par une simple initialisation aléatoire, sont inférieures aux solutions améliorées par les méthodes de recherche locale et hybride.

— Les méthodes hybrides, notamment GA+VNS et GD+VNS, produisent généralement des solutions dont la valeur objective se rapproche de celle de la solution de référence. Cela indique que l'ajout d'une phase de perturbation et d'une recherche approfondie dans un voisinage variable permet d'exploiter efficacement l'espace de recherche pour obtenir une solution quasi-optimale.

Temps d'exécution :

— Les méthodes de recherche locale (LS-FLIP et LS-SWAP) affichent des temps d'exécution extrêmement faibles (de l'ordre de la microseconde pour les petites instances), ce qui montre leur efficacité en terme de rapidité, même si leurs améliorations de la valeur objective sont parfois limitées.

— La descente de gradient (GD) présente des temps d'exécutions très faibles pour la valeur obtenue. L'augmenter via une méthode hybride avec un VNS permet d'améliorer sensiblement la descente de gradient, qui se bloque dans un mimimum local très rapidement.

— Les méthodes VND et VNS, qui combinent plusieurs voisinages et intègrent des perturbations, ont un coût CPU significativement plus élevé, en particulier sur les instances les plus volumineuses (par exemple, pour l'instance 500M30_1, VNS peut atteindre 85 secondes).
— L'algorithme génétique (GA) et sa variante hybride GA+VNS sont plus gourmands en temps CPU mais parviennent à obtenir des résultats de haute qualité, souvent supérieurs aux autres approches.

Conclusion sur les résultats : Globalement, les méthodes hybrides (notamment GA+VNS et GD+VNS) offrent un excellent compromis entre qualité de la solution et temps de calcul, en améliorant significativement la solution initiale et en parvenant à explorer efficacement l'espace de recherche. Cependant, pour des instances de grande taille, le coût en temps CPU devient critique et il serait intéressant d'explorer des pistes d'optimisation (parallélisation, optimisation des routines, etc.).

Les résultats sont reproductibles grâce à la seed qui a été définie (ici, à 42).

4 Conclusion et Perspectives

Nous avons présenté une implémentation en langage C d'une série d'algorithmes métaheuristiques pour le problème du sac à dos multidimensionnel 0-1. Le code intègre plusieurs composantes essentielles : lecture et représentation des données, construction d'une solution initiale, vérification de la faisabilité, recherche locale (FLIP et SWAP), descente de voisinage variable (VND) et recherche à voisinage variable (VNS), ainsi qu'une approche par descente de gradient et un algorithme génétique.

Forces :

— L'utilisation de structures de données simples et efficaces permet une bonne gestion de la mémoire et une exécution rapide.
— La modularité des fonctions (LS, VND, VNS, GD, GA) facilite l'expérimentation et la combinaison de différentes stratégies.
— L'intégration d'un algorithme multi-start et d'une gestion du temps d'exécution permet de respecter une contrainte de temps imposée.

Faiblesses et pistes d'amélioration :

— La recherche locale 1-FLIP se limite à l'ajout d'objets, ce qui peut réduire la diversification. L'exploration d'autres types de voisinages (par exemple, 2-FLIP ou la suppression explicite d'objets) pourrait être envisagée.
— Le mécanisme de réparation, bien que fonctionnel, repose sur le retrait d'objets selon le ratio valeur/poids. Des stratégies de réparation plus sophistiquées pourraient être étudiées.
— Les paramètres des algorithmes (taux de mutation, taille de la population, paramètres de la descente de gradient, etc.) nécessitent un réglage fin, pouvant varier selon les instances.
— La complexité temporelle des algorithmes, en particulier pour le GA, peut devenir élevée sur de très grandes instances. L'optimisation et la parallélisation pourraient être envisagées pour améliorer les performances.
— Il a été envisagé d'utiliser du multithreading et une accélération matérielle, mais nous n'avons pas eu le temps de les implémenter complètement ; il subsiste quelques reliquats dans le code.

En conclusion, ce travail offre une base solide pour la résolution du MKP à l'aide de métaheuristiques. Des améliorations sur la diversification des recherches locales, l'adaptation dynamique des paramètres et l'optimisation du code constituent des pistes intéressantes pour des travaux futurs.

5 Contributions

Le projet, bien qu'initialement prévu pour être réalisé par plus de quatre membres, a finalement été principalement développé par Thomas et Elias, en raison du manque d'implication de certains membres. Les contributions sont les suivantes :

- **Thomas** : a développé la structure de données, la procédure VND, la procédure VNS et l'algorithme génétique, et a réalisé les expériences numériques (ainsi que plusieurs fonctions utilitaires).
- **Elias** : a rédigé le rapport, conçu la structure générale du programme ainsi que le fichier `main.c`, développé la structure de données, les deux recherches locales et la descente de gradient (ainsi que plusieurs fonctions utilitaires).
- **Rania** : a développé sa propre version du projet, dont nous nous sommes parfois inspirés pour le projet final. Sa version, disponible sur la branche `MPK` du GitHub, comprend une méthode d'initialisation gloutonne aléatoire, deux fonctions de recherche locale, ainsi qu'une implémentation de VND et de VNS. Note : Cette version du projet est moins avancé et moins optimisé.
- **Izaak** : s'est intéressé au projet, sans toutefois produire de code.

Disponibilité des données et Disponibilité du code

Le code source et les jeux de données utilisés dans ce travail sont disponibles à l'adresse suivante : `https://github.com/Malchemis/MKP`.

Références

Principalement, le coursn, ainsi que :

`https://en.wikipedia.org/wiki/Variable_neighborhood_search`

`https://www.researchgate.net/publication/2906122_A_Tutorial_on_Variable_Neighborhood_Search`

Lien de la template du LaTeX : `https://www.overleaf.com/latex/templates/cibb-template-for-short-papers/yskhdfdgtdpx`