Nantes Université — UFR Sciences et Techniques
Master informatique
Année académique 2024-2025

Dossier Exercice d'implémentation

Métaheuristiques

YADEL GAYA - BEN HAMOUDI MELYSSA

26 novembre 2024

Livrable de l'exercice d'implémentation 1 : Heuristiques de construction et d'amélioration gloutonnes

Formulation du SPP

Présenter la formulation du SPP.

Le set packing problem (SPP) est un problème d'optimisation combinatoire qui a comme onjectif de maximiser un profit sous une contrainte qu'une ressource doit etre consommée par une seule activité.
On présente le modèle comme suit :

Maximiser

$$Z = \sum_{j \in S} c_j x_j$$

Sous les contraintes :

$$\sum_{j : i \in S_j} x_j \leq 1, \quad \forall i \in N$$

$$x_j \in \{0, 1\}, \quad \forall j \in S$$

Exemple d'un problème avec SPP

Exemple d'un problème d'affectation ressources-taches : Une entreprise dispose de plusieurs équipes et souhaite affecter ces équipes à différents projets. Cependant, chaque équipe doit participer à un seul projet au maximum. L'objectif est de maximiser le bénéfice total en sélectionnant un ensemble de projets, tout en respectant cette contrainte. Prenons l'exemple de 5 équipes (ressources) E1, E2, E3, E4, E5 et 5 projets(taches) P1, P2, P3, P4, P5.
Les bénéfice de chaque projet est donné succèssivement par : 100, 150, 200, 250, 300

La contrainte assure qu'une équipe ne peut être assignée qu'à un seul projet :

$$\sum_{j=1}^{5} x_{ij} \leq 1 \quad \textit{pour chaque équipe } i \in \{1, 2, 3, 4, 5\} \tag{1}$$

La fonction objectif est donnée par :

$$Maximiser \quad Z = 100x_1 + 150x_2 + 200x_3 + 250x_4 + 300x_5 \tag{2}$$

Modélisation JuMP (ou GMP) du SPP

La modélisation du problème de *Set Packing Problem* (SPP) à l'aide de JuMP se fait en suivant les étapes suivantes :

- On définit un modèle d'optimisation avec la fonction `Model()`.
- On crée une variable binaire `x[i]` pour chaque élément du problème, indiquant si l'élément est sélectionné (`1`) ou non (`0`).
- L'objectif est de maximiser la somme des coefficients associés à chaque élément, c'est-à-dire $\sum C_i x_i$.

— Une contrainte est ajoutée pour chaque ligne de la matrice de contraintes, assurant que la somme des éléments sélectionnés dans chaque ensemble ne dépasse pas 1.

Le code suivant implémente cette modélisation en JuMP :

```
function setSPP(C, A)
    m, n = size(A)  # Dimensions de la matrice A
    spp = Model()  # Création du modèle JuMP
    @variable(spp, x[1:n], Bin)  # Définition des variables binaires
    @objective(spp, Max, dot(C, x))  # Fonction objectif : maximiser la somme pondérée
    @constraint(spp, cte[i=1:m], sum(A[i,j] * x[j] for j=1:n) <= 1)  # Contraintes sur les ensemb
    return spp  # Retour du modèle JuMP
end
```

Instances numériques de SPP

Présenter les 10 instances sélectionnées sous forme de tableau.

pb_100rnd0100.dat
pb_100rnd0200.dat
pb_100rnd0300.dat
pb_100rnd0400.dat
pb_100rnd0500.dat
pb_100rnd0600.dat
pb_100rnd0700.dat
pb_100rnd0800.dat
pb_100rnd0900.dat
pb_100rnd1000.dat

Heuristique de construction appliquée au SPP

Présentation de l'algorithme mis en œuvre.
Nous avons créé une fonction `construct_heuristic` qui génère une solution initiale pour le problème du Set Packing en sélectionnant les colonnes dans l'ordre décroissant de leur rapport coût/occurrences, tout en respectant les contraintes sur les lignes (pas plus d'une colonne sélectionnée par ligne). Il s'agit d'une heuristique gloutonne qui maximise le ratio tout en assurant la faisabilité des solutions générées, retournant ainsi une solution initiale réalisable x_0.

Illustration sur l'exemple Didactic
Ici, le nombre de colonnes est 9 et les coûts associés à chaque colonne sont $C = [10, 5, 8, 6, 9, 13, 11, 4, 6]$, et le nombre de contraintes est 7. L'initialisation de x_0 est $x_0 = [0, 0, 0, 0, 0, 0, 0, 0, 0]$. En appelant la fonction `construct_heuristic`, on obtient :

$$x_0 = \texttt{construct\_heuristic}(C, A)$$

Le résultat est le suivant :

$$\textbf{Solution initiale : } x_0 = [0, 0, 0, 1, 0, 1, 1, 0, 0]$$

La valeur de la fonction objectif pour la solution initiale est $z_0 = 30$.

Présentation de l'algorithme mis en œuvre.

Nous avons créé une fonction `local_search` qui effectue une recherche locale en utilisant un échange de type 1-2 (deux éléments sont ajoutés à la solution, et un élément est retiré) pour améliorer une solution initiale x_0 au problème de Set Packing. La recherche s'arrête lorsqu'aucune amélioration n'est détectée après une itération complète. La faisabilité est vérifiée à chaque étape pour s'assurer que les solutions générées respectent les contraintes du problème.

Illustration sur l'exemple Didactic

Ici, le nombre de colonnes est de 9 et les coûts associés à chaque colonne sont $C = [10, 5, 8, 6, 9, 13, 11, 4, 6]$. Le nombre de contraintes est de 7

Nous prenons la valeur de x_0 obtenue par la méthode de construction gloutonne et appelons la fonction `local_search` :

$$x_{opt}, z_{opt} = \texttt{local\_search}(x_0, C, A)$$

Le résultat est le suivant :

> Début de la recherche locale (1-2 exchange)...
> Recherche locale terminée, aucune amélioration supplémentaire.

La solution optimisée obtenue est :

$$x_{opt} = [0, 0, 0, 1, 0, 1, 1, 0, 0]$$

La valeur optimisée de la fonction objectif est :

$$z_{opt} = 30$$

Dans ce cas, la recherche locale n'a pas trouvé d'amélioration par rapport à la valeur initiale x_0.

Expérimentation numérique

Présentation de l'environnement machine sur lequel les algorithmes vont tourner (référence).

L'environnement machine est un PC Lenovo Thinkpad T460 avec le système d'exploitation Windows 10 Professionnel. La référence complète de la machine est :

Référence complète : LENOVO_MT_20FM_BU_Think_FM_ThinkpadT460

Présentation sous forme synthétique (tableau, graphique, etc.) des résultats obtenus pour les 10 instances sélectionnées.

Instances	**Construct_Heuristic**	**Local_Search**	**GLPK**
pb_100rnd0100.dat	342	353	372
pb_100rnd0200.dat	29	32	34
pb_100rnd0300.dat	193	194	203
pb_100rnd0400.dat	13	14	16
pb_100rnd0500.dat	512	622	639
pb_100rnd0600.dat	45	63	64
pb_100rnd0700.dat	460	498	503
pb_100rnd0800.dat	32	38	39
pb_100rnd0900.dat	442	452	463
pb_100rnd1000.dat	36	38	40

TABLE 1 – Tableau des résultats pour différentes instances

Discussion

Questions type pour mener votre discussion :

— au regard des temps de résolution requis par le solveur MIP (GLPK) pour obtenir une solution optimale à l'instance considérée, l'usage d'une heuristique se justifie-t-il ?

Réponse : Pour moi, l'usage d'une heuristique se justifie lorsque le temps requis par un solveur exact, comme GLPK, pour obtenir une solution optimale devient trop important, surtout pour des instances de grande taille ou dans des contextes où le temps est limité. Si l'heuristique produit rapidement des solutions proches de l'optimum, elle offre un bon compromis entre qualité et efficacité. En revanche, pour des petites instances où GLPK résout rapidement, l'heuristique peut être moins pertinente. Cette pertinence peut être démontrée en comparant les temps de résolution et les écarts entre les solutions obtenues par les deux approches.

— avec pour référence la solution optimale, quelle est la qualité des solutions obtenues avec l'heuristique de construction et l'heuristique d'amélioration ?
Sur le plan des temps de résolution, quel est le rapport entre le temps consommé par le solveur MIP et vos heuristiques ?

Réponse : En se basant sur la solution optimale obtenue avec le solveur MIP (GLPK), on évalue les performances des heuristiques de construction et d'amélioration en termes de qualité des solutions fournies. L'heuristique d'amélioration, grâce à des techniques comme la recherche locale, permet souvent de réduire l'écart par rapport à l'optimum en affinant les solutions initiales générées par l'heuristique de construction.
Les temps d'exécution montrent que les heuristiques sont nettement plus rapides que le solveur exact GLPK, en particulier pour des instances plus complexes. On peut calculer le rapport des temps pour illustrer cette différence. Par exemple, si le solveur met 60 secondes pour trouver l'optimum alors qu'une heuristique ne nécessite que 2 secondes.

— Le recours aux (méta)heuristiques apparaît-il prometteur ?
Entrevoyez-vous des pistes d'amélioration à apporter à vos heuristiques ?

Réponse : Oui, le recours aux (méta)heuristiques est clairement prometteur, surtout lorsqu'il s'agit d'optimiser les temps de calcul tout en obtenant des solutions approximatives de bonne qualité. Ces heuristiques offrent un compromis entre la rapidité d'exécution et la précision des résultats, ce qui les rend particulièrement adaptées pour traiter des problèmes de grande taille ou des instances complexes où les méthodes exactes, comme les solveurs MIP, sont trop lentes.

Livrable de l'exercice d'implémentation 2 : Métaheuristique GRASP, ReactiveGRASP et extensions

Présentation succincte de GRASP appliqué sur le SPP

Présenter l'algorithme mis en oeuvre. Illustrer sur un exemple didactique (poursuivre avec l'exemple pris en DM1). Présenter vos choix de mise en oeuvre.

Dans cette partie du projet on a implémenté une métaheuristique GRASP (Greedy Randomized Adaptive Search Procedure) pour résoudre le problème de Set Packing.

Fonction `calculate_utilities(C, num_elements, e )`

La fonction `calculate_utilities` calcule les "utilités" de chaque colonne pour une solution partielle. Les utilités représentent une mesure de l'attractivité de chaque colonne dans le cadre de la construction de la solution.

- Si une colonne est déjà utilisée (`e [j] == 1`), son utilité est définie à 0.
- Sinon, l'utilité est calculée en divisant le coût de la colonne (`C[j]`) par le nombre d'éléments qu'elle contient (`num_elements[j]`).

Fonction greedyrandom `(C, A, alpha)`

La fonction greedyrandom construit une solution initiale en suivant l'algorithme GRASP. Voici les étapes principales :

- Elle sélectionne les colonnes en fonction de leurs utilités, définies par la fonction `calculate_utilities`.
- Une recherche est effectuée dans un sous-ensemble appelé RCL (Restricted Candidate List), qui contient les colonnes dont l'utilité est au-dessus d'un certain seuil calculé avec un facteur `alpha`.
- Après chaque sélection, les conflits sont pris en compte en marquant les colonnes en conflit avec celles déjà choisies.
- Le processus se répète jusqu'à ce que toutes les colonnes soient utilisées ou que la RCL soit vide.

Fonction `GRASP(C, A, alpha)`

La fonction `GRASP` combine la phase constructive et la phase de recherche locale. Elle effectue les actions suivantes :

- Elle génère une solution initiale en appelant la fonction greedyrandom
- Ensuite, elle améliore cette solution en utilisant une recherche locale via la fonction `local_search` (non détaillée ici).
- Enfin, elle retourne la meilleure solution trouvée et sa valeur (fonction objectif).

Exemple didactic

Le paramètre `alpha` de GRASP est fixé à 0.9. On appelle la fonction suivante :

$$x, z = GRASP(C, A, \alpha)$$

Ensuite, on affiche les résultats obtenus avec les lignes de code suivantes :

```
println("meilleure sol = ", x)

println("meilleure val = ", z)
```

Les résultats sont les suivants :

Running GRASP...
Début de la recherche locale (1-2 exchange)...

Recherche locale terminée, aucune amélioration supplémentaire.
meilleure sol = [0, 0, 0, 1, 0, 1, 1, 0, 0]
meilleure val = 30

Présentation succincte de ReactiveGRASP appliqué sur le SPP

Présenter l'algorithme mis en oeuvre. Illustrer sur un exemple didactique (poursuivre avec l'exemple pris en DM1). Présenter vos choix de mise en oeuvre.

Présentation de l'algorithme `ReactiveGRASP`

L'algorithme `ReactiveGRASP` est une extension du classique GRASP (Greedy Randomized Adaptive Search Procedure). Son objectif est d'améliorer l'efficacité de la recherche en adaptant dynamiquement le paramètre α, qui influence le processus de sélection dans la phase de construction de la solution.

Fonctionnement de l'algorithme

L'algorithme commence par définir plusieurs paramètres importants :

- α : Un paramètre qui détermine l'intensité de la recherche dans la phase de construction.
- Une liste de valeurs discrètes pour α, permettant de tester différentes stratégies de recherche.
- Des probabilités associées à chaque valeur de α, qui sont initialement uniformes.

1. Sélection du paramètre α

À chaque itération, l'algorithme choisit une valeur de α en fonction des probabilités pk. Cette sélection est effectuée par un tirage pondéré, où les valeurs de α qui ont donné de bons résultats dans les itérations précédentes sont plus susceptibles d'être choisies.

2. Exécution de GRASP

Une fois le paramètre α sélectionné, l'algorithme exécute la fonction GRASP pour construire une solution initiale. Cette phase est basée sur une heuristique gloutonne, suivie d'un choix aléatoire pondéré pour sélectionner les éléments qui formeront la solution.

3. Mise à jour des statistiques

Après chaque itération, l'algorithme met à jour plusieurs statistiques :

- La moyenne des résultats obtenus avec chaque valeur de α (notée $zAvg$).
- Le nombre d'utilisations de chaque valeur de α (noté $count\_alpha$).
- La meilleure solution ($best\_solution$) et la meilleure valeur de la fonction objectif ($best\_value$) obtenue jusqu'à présent.
- Les solutions extrêmes ($zBest$ et $zWorst$), représentant respectivement la meilleure et la pire valeur obtenue.

4. Mise à jour des probabilités

Tous les N_α itérations, l'algorithme réévalue les probabilités associées aux valeurs de α :

- Il calcule une moyenne des résultats obtenus pour chaque α et ajuste les probabilités en fonction de la qualité des solutions.
- Si certaines valeurs de α ont conduit à de bonnes solutions, leur probabilité d'être sélectionnées augmente. Si d'autres n'ont pas donné de bons résultats, leur probabilité diminue.

5. Calcul de la solution finale

Après *max_iterations* itérations, l'algorithme retourne la meilleure solution trouvée, ainsi que sa valeur, ainsi que la pire valeur et la moyenne entre les deux.

Illustration avec un exemple didactique

Dans un exemple simple, supposons que nous avons une matrice de contraintes A et un vecteur de coûts C, avec différentes valeurs possibles pour α. L'algorithme commence avec des probabilités égales pour chaque valeur de α et effectue plusieurs itérations de recherche. À chaque itération, l'algorithme sélectionne un α, exécute GRASP pour générer une solution, puis ajuste les probabilités en fonction des résultats obtenus.

Après un certain nombre d'itérations, l'algorithme retourne la solution optimale (ou la meilleure trouvée jusqu'à présent) et fournit également des informations sur la dispersion des résultats (la meilleure, la pire solution et la moyenne).

Choix de mise en œuvre

Le choix de l'algorithme `ReactiveGRASP` repose sur plusieurs facteurs :

- **Adaptabilité de** α : Le paramètre α influence directement la qualité des solutions obtenues. L'ajustement dynamique de ce paramètre permet de mieux explorer l'espace de recherche et d'éviter de se retrouver dans des optima locaux.
- **Exploration vs exploitation** : En ajustant α, l'algorithme équilibre l'exploration de nouvelles solutions et l'exploitation des meilleures solutions déjà trouvées, ce qui améliore la convergence vers une solution optimale.
- **Probabilités dynamiques** : L'utilisation de probabilités pour sélectionner α permet de mieux adapter la recherche en fonction des performances passées, ce qui rend l'algorithme plus robuste à la diversité des instances de problème.

Exemple didactique
Les paramètres de la condition `ReactiveGRASP` sont :

- `max_iterations` $= 30$
- `alpha_values` $= [0.1, 0.5, 0.9]$
- `Nalpha` $= 3$

Ensuite, on appelle la fonction :

$$\texttt{best\_solution, best\_value, zWorst, mean\_value} = \texttt{ReactiveGRASP}(C, A, \texttt{max\_iterations}, \texttt{alpha\_values}, \texttt{Nalph}$$

et on obtient les résultats suivants :

- Meilleure solution trouvée : $[0, 0, 0, 1, 0, 1, 1, 0, 0]$
- Valeur de la meilleure solution : 30
- Valeur de la pire solution ($zWorst$) : 30.0
- Moyenne entre la meilleure et la pire valeur : 30.0

Expérimentation numérique de GRASP

Présenter le protocole d'expérimentation (environnement matériel ; budget de calcul ; condition(s) d'arrêt ; réglage des paramètres). **Présentation de l'environnement machine sur lequel les algorithmes vont tourner (référence).**

L'environnement machine est un PC Lenovo Thinkpad T460 avec le système d'exploitation Windows 10 Professionnel. La référence complète de la machine est :

Référence complète : LENOVO_MT_20FM_BU_Think_FM_ThinkpadT460

Budget de calcul : 60 secondes

Condition d'arrêt : Approcher la valeur exacte trouvée par GLPK

Réglage des paramètres : $\alpha = 0.9$

Rapporter l'étude de l'influence du paramètre α :

Le paramètre α joue un rôle crucial dans l'algorithme `GRASP`. Lorsque α est proche de 1, l'algorithme devient plus exploratoire, générant des solutions plus aléatoires. Cela est dû au fait que les valeurs des utilités dans la RCL (Restricted Candidate List) sont relativement proches, ce qui augmente l'incertitude dans le choix des solutions à explorer. En revanche, lorsque α est proche de 0, l'algorithme devient plus déterministe et privilégie les solutions proches des valeurs de coût C, favorisant ainsi des choix plus optimisés et moins aléatoires. Cette variation de α permet de contrôler le compromis entre exploration et exploitation dans la recherche de la solution optimale.

Présenter sous forme de tableau les résultats finaux obtenus pour les 10 instances sélectionnées.

Instances	**Construct_Heuristic**	**Local_Search**	**GLPK**	**GRASP**
pb_100rnd0100.dat	342	353	372	368
pb_100rnd0200.dat	29	32	34	32
pb_100rnd0300.dat	193	194	203	195
pb_100rnd0400.dat	13	14	16	15
pb_100rnd0500.dat	512	622	639	630
pb_100rnd0600.dat	45	63	64	64
pb_100rnd0700.dat	460	498	503	499
pb_100rnd0800.dat	32	38	39	38
pb_100rnd0900.dat	442	452	463	453
pb_100rnd1000.dat	36	38	40	39

TABLE 2 – Tableau des résultats pour différentes instances

Expérimentation numérique de ReactiveGRASP

Présenter le protocole d'expérimentation (env. matériel ; budget de calcul ; condition(s) d'arrêt).

L'environnement machine est un PC Lenovo Thinkpad T460 avec le système d'exploitation Windows 10 Professionnel. La référence complète de la machine est :

Référence complète : LENOVO_MT_20FM_BU_Think_FM_ThinkpadT460

Budget de calcul : 60 secondes

Condition d'arrêt : 100 itérations

Réglage des paramètres : $\text{alpha}_values = [0.1, 0.5, 0.9]$
$Nalpha = 3$

Rapporter l'apprentissage du paramètre $N\alpha$ réalisé par ReactiveGRASP, les valeurs saillantes établies.

L'apprentissage du paramètre $N\alpha$ dans l'algorithme ReactiveGRASP consiste à ajuster dynamiquement les probabilités de sélection des valeurs de α pendant l'exécution de l'algorithme. Ce paramètre contrôle l'impact de la valeur de α sur la stratégie de recherche. Lors de chaque itération, un tirage pondéré est effectué pour choisir une valeur de α, et les probabilités sont mises à jour après un certain nombre d'itérations, $N\alpha$, en fonction des performances des solutions obtenues avec chaque valeur de α. Ce processus permet à l'algorithme de s'adapter aux différentes instances du problème en privilégiant les valeurs de α qui conduisent à de meilleures solutions. Les valeurs saillantes observées montrent que $N\alpha$ influence directement l'efficacité de l'exploration de l'espace de solutions et peut améliorer la qualité des résultats obtenus par GRASP. En ajustant $N\alpha$, l'algorithme peut équilibrer l'exploration et l'exploitation pour obtenir des solutions optimales ou proches de l'optimal.

Présenter sous forme de tableau les résultats finaux obtenus pour les 10 instances sélectionnées.

Instances	**Construct_Heuristic**	**Local_Search**	**GLPK**	**GRASP**	**ReactiveGRASP**
pb_100rnd0100.dat	342	353	372	368	372
pb_100rnd0200.dat	29	32	34	32	33
pb_100rnd0300.dat	193	194	203	195	202
pb_100rnd0400.dat	13	14	16	15	15
pb_100rnd0500.dat	512	622	639	630	639
pb_100rnd0600.dat	45	63	64	64	64
pb_100rnd0700.dat	460	498	503	499	500
pb_100rnd0800.dat	32	38	39	38	38
pb_100rnd0900.dat	442	452	463	453	454
pb_100rnd1000.dat	36	38	40	39	40

TABLE 3 – Tableau des résultats pour différentes instances

Discussion

Tirer des conclusions en comparant les résultats collectés avec vos deux variantes de métaheuristiques.

En comparant les résultats obtenus avec les deux variantes de métaheuristiques, nous observons que la méthode GRASP a permis une amélioration significative des solutions par rapport aux méthodes classiques, avec une valeur optimale de 368 pour l'instance `pb_100rnd0100.dat`. Cependant, l'algorithme ReactiveGRASP a montré une performance encore supérieure en offrant des solutions de meilleure qualité, comme en témoigne l'amélioration du résultat pour la même instance, où la solution obtenue est de 372. Cela met en évidence l'efficacité de la réactivité dans l'optimisation, permettant d'explorer des régions de solution encore plus prometteuses grâce à l'ajustement dynamique du paramètre α. Ainsi, ReactiveGRASP a permis d'obtenir des solutions améliorées par rapport à GRASP, renforçant son intérêt pour des problèmes d'optimisation complexes.

Quelles sont les recommandations que vous émettez à l'issue de l'étude et avec quelle variante continuez vous l'aventure des métaheuristiques ?

À l'issue de cette étude, plusieurs recommandations peuvent être émises. Tout d'abord, bien que l'algorithme GRASP ait montré une amélioration des solutions par rapport aux méthodes classiques, il reste limité par son choix fixe du paramètre α. L'algorithme ReactiveGRASP, grâce à son mécanisme adaptatif de mise à jour du paramètre α, a démontré une capacité supérieure à explorer efficacement l'espace de recherche et à trouver des solutions encore meilleures.

Ainsi, il est recommandé de poursuivre l'optimisation avec ReactiveGRASP pour les prochaines étapes, en particulier pour des problèmes complexes où l'ajustement dynamique des paramètres peut offrir un avantage considérable. Cette approche réactive semble plus prometteuse pour obtenir des solutions de qualité optimale, et elle pourrait être utilisée comme base pour des améliorations futures, telles que l'intégration de nouvelles heuristiques ou l'expérimentation avec d'autres techniques d'optimisation hybride.

Livrable de l'exercice d'implémentation 3 : Battle of metaheuristics

Présentation succincte des choix de mise en œuvre de la métaheuristique concurrente à GRASP appliquée au SPP

Présentation de l'algorithme de recherche tabou

La **recherche tabou** est une métaheuristique d'optimisation combinatoire permettant d'échapper aux optima locaux. Elle repose sur l'utilisation d'une *liste tabou* pour mémoriser temporairement certaines solutions ou mouvements afin d'éviter leur répétition. Les étapes principales de l'algorithme sont les suivantes :

1 Initialisation

- Définir la **solution initiale** x_0 : Il s'agit de la solution de départ, qui peut être générée de manière aléatoire ou choisie de manière spécifique.
- Initialiser la **liste tabou** : Une liste vide est créée pour enregistrer les solutions déjà explorées, afin d'éviter de revisiter les mêmes solutions et d'encourager la recherche vers de nouvelles zones.
- Initialiser la **solution courante** $x_{current}$ et la **meilleure solution** x_{best} avec x_0.

2 Recherche des voisins avec recherche locale

- **Générer les voisins** : La fonction `generate_neighbors_with_local_search` génère des voisins en inversant un élément de la solution courante $x_{current}$, ce qui peut être vu comme une technique de "flip" des bits.
- **Recherche locale pour chaque voisin** : Après avoir généré un voisin, une recherche locale est appliquée pour l'améliorer, utilisant la fonction `local_search2`. Cette recherche locale cherche à améliorer chaque voisin en fonction de la fonction objectif et des contraintes du problème.
- **Ajouter le voisin amélioré à la liste des voisins** : Une fois qu'un voisin a été amélioré, il est ajouté à la liste des voisins.

3 Sélection du meilleur voisin faisable

- **Évaluation des voisins** : Chaque voisin généré (après recherche locale) est évalué à l'aide de la fonction objectif `evaluate_solution`, qui retourne la valeur de la fonction objectif pour une solution donnée.
- **Sélection du meilleur voisin faisable** : Parmi les voisins faisables (qui respectent les contraintes), le voisin avec la meilleure valeur de la fonction objectif est sélectionné.
- **Mise à jour de la meilleure solution** : Si la solution d'un voisin améliore la meilleure solution connue, celle-ci est mise à jour.

4 Mise à jour de la liste tabou

— **Ajouter le voisin à la liste tabou** : La solution courante est ajoutée à la liste tabou pour éviter de la revisiter.
— **Contrôler la taille de la liste tabou** : Si la liste dépasse la taille maximale ($tabu\_size$), le premier élément (le plus ancien) est retiré.

5 Arrêt

— **Critère d'arrêt** : L'algorithme s'arrête soit après un nombre maximal d'itérations (max_iter), soit lorsqu'une condition de convergence est atteinte (par exemple, si aucun voisin faisable n'améliore la solution).

6 Affichage de la meilleure solution

— Après la fin de l'algorithme, la **meilleure valeur obtenue** pour la fonction objectif est affichée. Cette valeur correspond à la meilleure solution trouvée après avoir exploré les voisins et appliqué la recherche locale.

7 Résumé en étapes

1. **Initialisation** : Définir la solution initiale et initialiser la liste tabou.
2. **Recherche des voisins** : Générer tous les voisins en inversant un élément et appliquer la recherche locale pour améliorer chaque voisin.
3. **Sélection du meilleur voisin** : Choisir le voisin faisable avec la meilleure valeur de la fonction objectif.
4. **Mise à jour de la liste tabou** : Ajouter la solution courante à la liste tabou et retirer l'élément le plus ancien si nécessaire.
5. **Arrêt** : L'algorithme s'arrête après un nombre maximal d'itérations ou si un critère de convergence est atteint.
6. **Affichage du résultat** : Afficher la meilleure valeur obtenue à la fin des itérations.

Exemple didactic :

Initialisation :
— Le nombre d'itérations = 100.
— Le nombre d'itérations de la recherche locale = 100.
— Taille de la liste Tabou = 7
On appelle la fonction suivante :

```
x_best, z_best = tabu_search(x0, C, A, max_iter, tabu_size, max_local_iter)
```

Pour afficher les résultats, on utilise :

println("Meilleure solution trouvée avec la recherche tabou : ", x_best)
println("Meilleure valeur trouvée avec la recherche tabou est : ", z_best)

Les résultats sont les suivants :

```
Running Tabusearch...
Meilleure solution trouvée avec la recherche tabou : [0, 0, 0, 1, 0, 1, 1,
0, 0]
Meilleure valeur trouvée avec la recherche tabou est : 30
```

Expérimentation numérique comparative GRASP vs métaheuristique concurrente

Présentation de l'environnement machine sur lequel les algorithmes vont tourner (référence) :
L'environnement machine est un PC Lenovo Thinkpad T460 avec le système d'exploitation Windows 10 Professionnel.

Référence complète : LENOVO_MT_20FM_BU_Think_FM_ThinkpadT460

Budget de calcul : 60 secondes

Condition d'arrêt : Après 100 itérations

Réglage des paramètres : Longueur de la liste Tabou = 7

Présenter sous forme de tableau les résultats finaux obtenus pour les 10 instances sélectionnées.

Instances	**Construct_Heuristic**	**Local_Search**	**GLPK**	**GRASP**	**Tabu_Search**
pb_100rnd0100.dat	342	353	372	368	351
pb_100rnd0200.dat	29	32	34	32	32
pb_100rnd0300.dat	193	194	203	195	195
pb_100rnd0400.dat	13	14	16	15	15
pb_100rnd0500.dat	512	622	639	630	630
pb_100rnd0600.dat	45	63	64	64	63
pb_100rnd0700.dat	460	498	503	499	499
pb_100rnd0800.dat	32	38	39	38	38
pb_100rnd0900.dat	442	452	463	453	455
pb_100rnd1000.dat	36	38	40	39	38

TABLE 4 – Tableau des résultats pour différentes instances

Discussion

Tirer des conclusions en comparant les résultats collectés avec vos deux métaheuristiques :

En comparant les résultats des deux métaheuristiques, il est évident que **ReactiveGRASP** surpasse **Tabu_Search** dans la majorité des cas, obtenant des meilleures solutions sur presque toutes les instances testées. Toutefois, dans certains cas, notamment pour les instances plus petites comme `pb_100rnd0200.dat` et `pb_100rnd0800.dat`, **Tabu_Search** présente des résultats égaux ou très proches de ceux de **GRASP**, montrant ainsi que la recherche tabou peut parfois être compétitive. En général, **ReactiveGRASP** semble offrir des solutions de meilleure qualité et plus robustes, confirmant son efficacité supérieure par rapport à la méthode **Tabu_Search** pour la majorité des scénarios testés.

Recommandations à l'issue de l'étude

À l'issue de cette étude, plusieurs recommandations peuvent être faites pour améliorer la performance des métaheuristiques utilisées et leur application à des problèmes similaires :

- **Affiner les paramètres des algorithmes :** Les performances des deux métaheuristiques (ReactiveGRASP et Tabu_Search) peuvent être améliorées en ajustant les paramètres comme la taille de la liste tabou, le nombre d'itérations, et d'autres paramètres spécifiques à chaque méthode. Une étude systématique de ces paramètres pourrait permettre d'optimiser les résultats.
- **Tester sur des instances de plus grande taille :** Bien que les résultats pour les petites instances soient satisfaisants, il serait intéressant de tester les algorithmes sur des instances de plus grande taille pour mieux évaluer leur robustesse et leur efficacité en conditions réelles. Cela permettrait aussi d'observer comment chaque méthode évolue face à des problèmes plus complexes.
- **Utiliser des stratégies d'adaptation dynamique :** En combinant des éléments de différentes métaheuristiques ou en adaptant dynamiquement les paramètres de recherche, il est possible d'améliorer la flexibilité des algorithmes pour résoudre une plus large gamme de problèmes d'optimisation combinatoire.