UNIVERSITÉ DE NGAOUNDÉRÉ
FACULTÉ DES SCIENCES

THE UNIVERSITY OF NGAOUNDERE
FACULTY OF SCIENCE

DÉPARTEMENT DE MATHEMATIQUES ET INFORMATIQUE
PARCOURS : Systèmes et Logiciels en Environnements Distribués

Technique d'optimisation

Problème du sac a dos avec Méthode de recherche locale itérée (Iterated local search)

GROUPE 2

Sous la direction de :

Pr. NDAM

Annee academique 2023 - 2024

Nom et Prénom	**Matricules**
1- KHADIDJA HINDOU	21B057FS
2- KHADIDJA SALISSOU	19B602FS
3- DJONRE NDOSSUE CHRISTIAN	19A156FS
4- RACHIDATOU MAHMOUDA	19B384FS
5- SAYNBE MO RINGBE	19B059FS
6- SOUAPIEBE EDMOND PROSPERE	19B057FS

Année : 2024

Chapitre 1

1.1 SOMMAIRE

1.2 Introduction

1.3 Méthode de Recherche Locale Itérée

1.4 Implementaion de la Recherche Locale Itérée

1.5 Etude Comparative

1.6 Conclusion

1.7 Bibliographie

Introduction

Le problème du sac à dos est un problème classique d'optimisation combinatoire où l'objectif est de maximiser la valeur des objets placés dans un sac à dos tout en respectant sa capacité maximale. Il est largement étudié dans les domaines de l'informatique, de la logistique et de l'ingénierie pour ses nombreuses applications pratiques. Dans cet exposé, nous aborderons une approche spécifique pour résoudre ce problème : **la méthode de recherche locale itérée.** en suite nous comparons les résultats obtenue avec une autre approche vue en cour celle du **Hill Climbing.**

Méthode de Recherche Locale Itérée

La recherche locale itérée (LS : Iterated Local Search) est une approche d'amélioration de la méthode de descente locale [1]. Elle vise à éviter de converger vers un optimum local en redémarrant l'algorithme à partir de nouveaux points de départ. Contrairement à la méthode de Hill Climbing qui génère aléatoirement de nouveaux points de départ, la recherche locale itérée perturbe l'optimum trouvé à l'itération précédente pour générer de nouveaux points de départ. Cette approche permet de sortir de potentiels optimum locaux et d'améliorer la qualité de la solution.

Implémentation de la Recherche Locale Itérée

Dans cette section, nous nous plongerons dans les détails de l'algorithme de la recherche locale itérée, une méthode puissante pour résoudre le problème du sac à dos. Nous examinerons les différentes étapes de cet algorithme ainsi que les choix de paramètres critiques qui influent sur son efficacité. Enfin, nous fournirons un pseudo-code illustratif pour clarifier le fonctionnement de l'algorithme.

Algorithm de la Recherche Locale Itérée

L'algorithme de la recherche locale itérée suit une approche itérative pour trouver la meilleure solution possible au problème du sac à dos. Voici les étapes principales de cet algorithme :

1. **Initialisation** : Démarrez avec une solution initiale valide. Cela peut être obtenu en utilisant une méthode heuristique simple ou en générant une solution aléatoire.

2. **Recherche Locale** : Appliquez un algorithme de recherche locale, tel que le Hill Climbing, à la solution initiale. Cela implique d'explorer les voisins de la solution actuelle pour trouver une solution améliorante.

3. **Perturbation** : Après chaque itération de la recherche locale, perturbez légèrement la solution actuelle pour échapper aux optima locaux. Cela peut être réalisé en modifiant aléatoirement quelques éléments de la solution.

4. **Critère d'Acceptation** : Déterminez si la nouvelle solution obtenue après la perturbation est meilleure que la solution actuelle. Si c'est le cas, mettez à jour la solution actuelle avec la nouvelle solution. Sinon, conservez la solution actuelle.

5. **Critère d'Arrêt** : Répétez les étapes 2 à 4 jusqu'à ce qu'un critère d'arrêt soit atteint. Ce critère peut être un nombre fixe d'itérations, l'atteinte d'une certaine valeur de fonction objectif, ou l'épuisement du temps alloué.

Choix des Paramètres

Le succès de la recherche locale itérée dépend en grande partie des choix de paramètres tels que la solution initiale, la méthode de perturbation, et les critères d'acceptation et d'arrêt. Ces paramètres doivent être soigneusement ajustés pour obtenir de bons résultats. Par exemple, une

perturbation trop agressive peut conduire à une exploration excessive de l'espace de recherche, tandis qu'une perturbation trop faible peut entraîner une convergence prématurée vers un optimum local.

Pseudo-code

Voici un pseudo-code simplifié illustrant l'algorithme de la recherche locale itérée :

Algorithme 2 Recherche locale itérée // minimisation

Entrées: $\vec{x_0}$, f
Sorties: $\vec{x^*}$

1: $\vec{x} \leftarrow recherchelocale(f, \vec{x_0})$
2: **répéter**
3: $\quad \vec{x'} \leftarrow perturbation(f, \vec{x})$
4: $\quad \vec{x} \leftarrow recherchelocale(f, \vec{x'})$
5: $\quad$ **Si** $f(\vec{x'}) < f(\vec{x})$ ou critère d'acceptation **alors**
6: $\quad\quad \vec{x} \leftarrow \vec{x'}$
7: $\quad$ **Fin Si**
8: **jusqu'à** critère d'arrêt atteint
9: $\vec{x^*} \leftarrow \vec{x}$

FIGURE 1.1 – Algorithme de la recherche locale itérée

Ce pseudo-code donne un aperçu de la structure générale de l'algorithme de la recherche locale itérée et peut être adapté selon les besoins spécifiques du problème du sac à dos.

Étude Comparative

Nous réaliserons une étude comparative entre la recherche locale itérée et la méthode de Hill Climbing en utilisant des exemples numériques du problème du sac à dos. Nous analyserons les résultats obtenus en termes de qualité des solutions.

1.7.1 Code

Nous avons écris les codes suivant :

Import des bibliothèques

In [4]:
```
import numpy as np
import random
```

Initialisation des données du problème

In [5]:
```
n = 5                 #le nombre d'objets.
capacity = 30          # capacité maximale du sac à dos
objects_values = [7, 3, 2, 6, 9]      # liste des valeurs des objets
objects_weight = [12, 16, 11, 10, 18]    # la liste des poids des objets
```

Generation des solution aléatoire

In [6]:
```
def generate_initial_solutions(n):
    solutions = [random.randint(0, 1) for _ in range(n)]
    weight = total_weight(objects_weight, solutions)
    while weight > capacity:
        solutions = generate_initial_solutions(n)
        weight = total_weight(objects_weight, solutions)
    return solutions
```

FIGURE 1.2 – Initialisation et generation des solutions

1.7.2 Resultats

Apres execution des codes, nous obtenons le resultats suivant : Apres 10 exécutions, nous obtenons les résultats suivant :

Calculs des valeurs totale et de poids total

In [7]:
```
def total_values(objects_values, solutions):
    return sum(objects_values[i] * solutions[i] for i in range(len(solutions)))

def total_weight(objects_weight, solutions):
    return sum(objects_weight[i] * solutions[i] for i in range(len(solutions)))
```

Solution initiale, hill_climbing et Iterated local search

:n [10]:
```
solutions_initiales = generate_initial_solutions(n)
values = total_values(objects_values, solutions_initiales)
weight = total_weight(objects_weight, solutions_initiales)

print("\n----------------------------------------\n")
print("\n---- Solutions initiales  ----\n")
print("Initial solution : {}".format(solutions_initiales))
print("Valeur initiale: {}".format(values))
print("Poids inital: {}".format(weight))
```

FIGURE 1.3 – Fonction de calcul des valeurs et poids totaux et affichage

```
solutions_initiales = generate_initial_solutions(n)
# Fonction hill_climbing Tente d'ameliorer la solution initiale en explorant les voisins de la solution initiale
def hill_climbing(objects_values, objects_weight, capacity, n, solutions_initiales):
    solutions = solutions_initiales
    values = total_values(objects_values, solutions)
    weight = total_weight(objects_weight, solutions)


    for _ in range(100):
        neighbors = []
        neighbor = solutions
        for i in range(n):
            neighbor[i] = 1 - neighbor[i]
            neighbors.append(neighbor)

    best_neighbor = max(neighbors, key = lambda x: total_values(objects_values, x) if total_weight(objects_weight, x) <= capacity
    best_neighbor_value = total_values(objects_values, best_neighbor)
    best_neighbor_weight = total_weight(objects_weight, best_neighbor)

    if best_neighbor_value > values and best_neighbor_weight <= capacity:
        solutions = best_neighbor
        values = best_neighbor_value
        weight = best_neighbor_weight

    return solutions, values, weight
```

FIGURE 1.4 – Fonction du Hill Climbing

```
# Fonction perturbation melange aleatoirement la liste poour perturber la solution da la recherche itérée
def perturbation(x):
    random.shuffle(x)

# Fonction recherche locale itérée
def recherhe_locale_iteree(objects_values, objects_weight, capacity, n, x0):
    x, values, weight = hill_climbing(objects_values, objects_weight, capacity, n, x0)

    for _ in range(100):
        x1 = [k for k in x]
        perturbation(x1)
        x2, values2, weight2 = hill_climbing(objects_values, objects_weight, capacity, n, x1)

        if values2 > values:
            x, values, weight = x2, values2, weight2

    return x, values, weight

solutions_hill, values_hill, weights_hill = hill_climbing(objects_values, objects_weight, capacity, n, solutions_initiales)
```

FIGURE 1.5 – Fonction de perturbation et de la recherche locale itérée

```
solutions_hill, values_hill, weights_hill = hill_climbing(objects_values, objects_weight, capacity, n, solutions_initiales)

print("\n---------------------------------------\n")
print("\n---- Recherce Locale (Hill Climbing)  ----\n")
print("Optimal solution : {}".format(solutions_hill))
print(" Values : {}".format(values_hill))
print(" Weights : {}".format(weights_hill))

solutions_rli, values_rli, weights_rli = recherhe_locale_iteree(objects_values, objects_weight, capacity, n, solutions_initiales)

print("\n---------------------------------------\n")
print("\n---- Recherce Locale Iteree  ----\n")
print("Optimal solution  : {}".format(solutions_rli))
print(" Values : {}".format(values_rli))
print(" Weights : {}".format(weights_rli))
```

FIGURE 1.6 – Fonction d'affichage

```
---- Solutions initiales  ----

Initial solution : [1, 0, 0, 1, 0]
Valeur initiale: 13
Poids inital: 22

---------------------------------------


---- Recherce Locale (Hill Climbing)  ----

Optimal solution : [1, 0, 1, 0, 0]
 Values : 9
 Weights : 23

---------------------------------------


---- Recherce Locale Iteree  ----

Optimal solution  : [1, 0, 0, 0, 1]
 Values : 16
 Weights : 30
```

FIGURE 1.7 – Solution obtenue

	solutions initiale			Hill Climbing			Recherche Iteree		
	Sol init	Val init	Poids init	H Sol optimal	H Val	H Poids	I Sol optimal	I Val	I Poids
1	1,0,0,1,0	13	22	1,0,1,0,0	9	23	1,0,0,0,1	16	30
2	1,0,0,0,0	7	12	1,1,0,0,0	10	28	1,0,0,0,1	16	30
3	0,0,1,0,0	2	11	0,0,1,0,0	2	11	0,0,0,0,1	9	18
4	1,1,0,0,0	10	28	1,0,0,0,1	16	30	1,0,0,0,1	16	30
5	0,1,0,0,0	3	16	1,0,1,0,0	9	23	1,0,0,0,1	16	30
6	1,0,1,0,0	9	23	0,0,1,0,0	2	11	0,0,0,0,1	9	18
7	0,1,0,0,0	3	16	1,1,0,0,0	10	28	1,0,0,0,1	16	30
8	0,0,0,0,1	9	18	0,1,1,0,0	5	27	1,0,0,0,1	16	30
9	0,0,0,1,0	6	10	1,1,0,0,0	10	28	1,0,0,0,1	16	30
10	0,0,0,1,1	15	28	1,0,1,0,0	9	23	1,0,0,0,1	16	30
				10%			**80%**		

FIGURE 1.8 – Analyse statistique des 10 instances

Conclusion

Dans ce travail, nous avons exploré la méthode de recherche locale itérée comme approche pour résoudre le problème du sac à dos. Nous avons commencé par comprendre en détail le problème lui-même et son importance dans divers domaines, tels que l'informatique, la logistique et l'ingénierie.

Ensuite, nous avons examiné en profondeur la méthode de recherche locale itérée, en mettant en lumière ses principes fondamentaux, ses avantages et ses inconvénients. Nous l'avons comparée à une autre approche populaire, le Hill Climbing, pour mieux comprendre ses performances et son efficacité dans la résolution du problème du sac à dos.

Nous avons également discuté de l'implémentation pratique de la recherche locale itérée, en décrivant son algorithme, ses étapes principales et les choix de paramètres importants. Un pseudo-code a été présenté pour clarifier le fonctionnement de l'algorithme.

Enfin, nous avons souligné l'importance de cette méthode,. Malgré ses limitations, la recherche locale itérée offre une approche efficace et flexible pour résoudre le problème du sac à dos et d'autres problèmes d'optimisation combinatoire.

En conclusion, la recherche locale itérée représente un outil puissant pour la résolution de problèmes complexes, offrant un équilibre entre efficacité, simplicité et adaptabilité.

Bibliographie

– Pr NDAM cours technique d'optimisation, M2 SLED 2023-2024

– Local Search Algorithms" dans Wiley Interdisciplinary Reviews : Computational Statistics

– Iterated Local Search : A Framework and its Applications" dans European Journal of Operational Research