République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole nationale Supérieure d'Informatique
ex : INI (Institut National de formation en Informatique)

Rapport du projet d'optimisation combinatoire

Résolution du problème du sac à dos.

PARTIE I

- AMIEUR Nardjes
- AOUADJ Iheb Nassim
- ASSENINE Mohamed Sami
- AZIZ Rezak
- BOUMENDIL Anais
- MEDJADJI Chaimaa

GROUPE : SIQ3

Table des matières

1 INTRODUCTION **4**

2 ETAT DE L'ART **5**
2.1 ETUDE DU PROBLÈME . . . 5
2.1.1 DÉFINITION DU PROBLÈME (KNAPSACKS PROBLEM) . . . 5
2.2 VARIANTES DU PROBLÈME L . . . 5
2.2.1 PROBLÈME DU SAC À DOS BORNÉ (BKP) . . . 5
2.2.2 PROBLÈME DU SAC À DOS NON-BORNÉ (UKP) . . . 5
2.3 ETUDE DES MÉTHODES IMPLÉMENTÉES . . . 5
2.3.1 MÉTHODES EXACTES . . . 5
2.3.1.1 BRANCH AND BOUND . . . 5
2.3.1.2 PROGRAMMATION DYNAMIQUE . . . 6
2.3.2 HEURISTIQUES . . . 7
2.3.2.1 HEURISTIQUE PAR CONSTRUCTION . . . 7
2.3.2.2 HEURISTIQUE PAR VOISINAGE (PAR AMÉLIORATION) . . . 7
2.3.3 MÉTA-HEURISTIQUE . . . 8
2.3.3.1 MÉTHODES ITÉRATIVES DE VOISINAGE . . . 8
2.3.3.2 MÉTHODES ÉVOLUTIVES . . . 8

3 CONCEPTION **10**
3.1 ALGORITHMES MÉTHODES EXACTES . . . 10
3.1.1 BRANCH AND BOUND . . . 10
3.1.2 PROGRAMMATION DYNAMIQUE . . . 14
3.2 ALGORITHMES HEURISTIQUES . . . 17
3.2.1 HEURISTIQUES PAR CONSTRUCTION . . . 17
3.2.1.1 HEURISTIQUE PAR ARRONDI . . . 17
3.2.1.2 DENSITY ORDERED GREEDY . . . 19
3.2.1.3 WEIGHTED ORDERED GREEDY . . . 19
3.2.2 HEURISTIQUES PAR AMÉLIORATION . . . 20
3.2.2.1 RECHERCHE LOCALE . . . 20
3.3 ALGORITHMES MÉTA-HEURISTIQUES . . . 24
3.3.1 MÉTA-HEURISTIQUE ITÉRATIVE . . . 24
3.3.1.1 LE RECUIT SIMULÉ . . . 24
3.3.2 MÉTA-HEURISTIQUE ÉVOLUTIVES . . . 32
3.3.2.1 ALGORITHME GÉNÉTIQUE . . . 32
3.3.2.2 COLONIE DE FOURMIS . . . 38

4 TESTS ET RÉSULTATS **45**
4.1 INSTANCES UTILISÉES . . . 45
4.2 CONFIGURATION DE LA MACHINE UTILISÉE (COLAB) . . . 45
4.3 RÉSULTATS DES MÉTHODES EXACTES . . . 46
4.3.1 PROGRAMMATION DYNAMIQUE . . . 46
4.3.2 BRANCH AND BOUND . . . 47
4.4 ETUDE COMPARATIVE . . . 48
4.4.1 LES MÉTHODES EXACTES . . . 48
4.4.2 LES MÉTHODES APPROCHÉES . . . 49
4.4.2.1 L'HEURISTIQUE PAR AMÉLIORATION . . . 51
4.4.3 MÉTA-HEURISTIQUE ITÉRATIVE . . . 52
4.4.4 MÉTA-HEURISTIQUE ÉVOLUTIVES . . . 54
4.5 ETUDE EMPIRIQUE . . . 55

4.5.1 L'HEURISTIQUE PAR AMÉLIORATION 55
4.5.2 LE RECUIT SIMULÉ 56
4.5.3 ALGORITHME GÉNÉTIQUE 59
4.5.4 COLONIE DE FOURMIS 61

5 INTERFACE GRAPHIQUE **63**
5.1 PAGE D'ACCUEIL 63
5.2 PAGE D'ÉXCUTION D'UN ALGORITHME 63
5.3 PAGE DE COMPARAISON 64

6 CONCLUSION **66**

7 BIBLIOGRAPHIE **67**

Table des figures

1 Organigramme de l'algorithme "Branch & Bound" partie 1 10
2 Organigramme de l'heuristique par amélioration (recherche locale) 21
3 Organigramme de l'algorithme du recuit simulé 25
4 Organigramme de l'algorithme Génétique 33
5 Evolution du temps d'exécution du DP sur des instances de différentes tailles 40
6 Temps d'exécution de l'algorithme DP (en secondes) en fonction du nombre d'objets des instances 46
7 Evolution du temps d'exécution du DP sur des instances de différentes tailles 47
8 Temps d'exécution du Branch and bound (en secondes) en fonction du nombre d'objets des instances 47
9 Comparaison du temps d'exécution (en secondes) du B&B et DP 48
10 Comparaison des gains obtenus par les heuristiques spécifiques implémentées 49
11 Comparaison des temps d'exécution des heuristiques spécifiques implémentées 50
12 Comparaison gain de l'heuristique par amélioration avec les autres méthodes 52
13 Comparaison gain du recuit simulé avec les autres méthodes 52
14 Comparaison du temps d'exécution du recuit simulé avec les autres méthode 53
15 Comparaison des résultats entre AG, ACO et Branch and Bound 55
16 Etude de l'influence du paramètre "nombre d'itérations" sur le gain donné par l'heuristique par amélioration 56
17 Etude de l'influence du paramètre "facteur de refroidissement" sur le gain du recuit simulé . . 57
18 Etude de l'influence du paramètre "nombre de paliers" sur le gain du recuit simulé 57
19 Etude de l'influence du paramètre "température initiale" sur le gain du recuit simulé 58
20 Etude de l'influence du choix de la "solution initiale" sur le gain du recuit simulé 58
21 Comparaison des gains donnés par "recherche locale" et "recuit simulé" 59
22 Comparaison de l'influence de différentes combinaisons de paramètres sur le gain du recuit simulé 59
23 Etude de l'influence de nombre d'itération sur le temps d'exécution et sur la qualité de la solution 60
24 Etude de l'influence de la taille de la population sur le temps d'exécution et sur la qualité de la solution 60
25 Etude de l'influence de nombre de fourmis sur le temps d'exécution et sur la qualité de la solution 61
26 Etude de l'influence de nombre d'itération sur le temps d'exécution et sur la qualité de la solutions 61
27 Interface graphique de la page d'accueil 63
28 Interface graphique d'un algorithme 63

29 Interface graphique du graphe des resultats en terms de temps et de gain de l'algorithme B&B 64
30 Interface graphique pour changer les paramètres d'algorithme 64
31 Interface graphique pour sélectionner les algorithmes à comparer 64
32 Interface graphique pour sélectionner les instances 65
33 Interface graphique de la comparaison entre les algorithmes 65

1 INTRODUCTION

Un aspect important dans la vie humaine est le concept de décision. En effet, de nos jours on est toujours amené à prendre des décisions. L'impact de ces dernières dans un milieu professionnel a une importance cruciale, pour cela on s'intéresse à prendre des décisions qui auront un meilleur impact. On parle alors d'Optimisation. L'optimisation est essentiellement un outil d'aide à la décision. Et le problème d'optimiser se retrouve dans plusieurs domaines économique, politique, scientifique et social.

L'optimisation possède donc une part assez importante dans le domaine de recherche. Pour avoir une idée sur le but de cette branche, la citation suivante tirée du livre "Foundations of optimisation" est un bon résumé : " le désir de perfection trouve son expression dans la théorie de l'optimisation. Elle étudie comment décrire et atteindre ce qui est meilleur, une fois que l'on connaît comment mesurer et modifier ce qui est bon et ce qui est mauvais... La théorie de l'optimisation comprend l'étude des optimums et les méthodes pour les trouver."

Dans notre cas, on s'intéresse à un problème bien particulier : le problème de sac à dos. Ce problème est un problème qui est au centre de l'optimisation combinatoire. Il est tant rencontré dans la vie réelle que dans le monde de la recherche :

- Étant donné un certain montant d'investissement dans certains projets, on se pose la question de savoir quels sont les projets à choisir pour que l'ensemble rapporte le plus d'argent possible.
- Lors de la découpe de matériaux, on cherche toujours à savoir comment minimiser les pertes dues aux chutes.
- Durant les voyages, comment faire pour que le chargement de tous les bagages de bateau ou d'avion soit amené sans être en surcharge en fonction de plusieurs critères : poids, volume, destination, marchandises spécifiques. Plus généralement, ce problème peut modéliser toutes les situations où l'on veut sélectionner un sous ensemble optimal, de poids total borné, d'un ensemble d'éléments ayant chacun un poids et un profit associés.

  Dans un souci d'organisation, nous avons organisé notre rapport en 3 Chapitres : une étude littérature dans laquelle nous nous intéresserons à l'étude du problème de sac à dos et les méthodes existantes pour le résoudre en passant par les méthodes exactes, les heuristiques et métaheuristiques. Dans le second chapitre on s'intéresse à la conception des différentes méthodes citées et en dernier lieu nous ferons des tests tout en interprétant les résultats.

2 ETAT DE L'ART

2.1 ETUDE DU PROBLÈME

2.1.1 DÉFINITION DU PROBLÈME (KNAPSACKS PROBLEM)

Le problème de sac à dos, noté KP (Knapsack Problem) est un problème d'optimisation combinatoire. Il tient son nom du fait qu'il est analogue à la situation de remplissage d'un sac à dos pour un randonneur qui doit décider quoi prendre avec lui. On considère qu'il a un grand nombre d'objets utiles, chaque objet a un poids w_j et un bénéfice p_j. Pour des contraintes bien précises, le poids maximum que le sac peut supporter est limité.

Le Problème de sac à dos peut être formulé donc de la façon suivante : Étant donné une instance de sac à dos avec une capacité C et un ensemble N, contenant n objets $(x_1, x_2, ..., x_n)$ ayant chacun un profit $(p_1, p_2, ..., p_n)$ et un poid $(w_1, w_2, ..., w_n)$. L'objectif est de sélectionner un sous-ensemble de N qui maximise le bénéfice total des objets sélectionnés et dont la somme des poids ne dépasse pas la capacité C.

$$max \sum_{i=1}^{n} x_i p_i \quad sous \quad la \quad contrainte \quad \sum_{i=1}^{n} x_i w_i \leq C$$

2.2 VARIANTES DU PROBLÈME L

2.2.1 PROBLÈME DU SAC À DOS BORNÉ (BKP)

Le nombre d'exemplaires est limité, Dans ce cas le xi indiquera si le type d'objet est pris ou non $(x_i \in \{0, 1\})$.

2.2.2 PROBLÈME DU SAC À DOS NON-BORNÉ (UKP)

Le nombre d'exemplaires est illimité, Dans ce cas le xi indiquera le nombre d'exemplaires pris de chaque type d'objet $(x_i \geq 0)$

Dans le cadre de ce projet, on s'intéressera à cette variante précise du problème du sac à dos.

2.3 ETUDE DES MÉTHODES IMPLÉMENTÉES

2.3.1 MÉTHODES EXACTES

2.3.1.1 BRANCH AND BOUND

Pour trouver la solution optimale, et être certain qu'il n'y a pas mieux, il faut utiliser une méthode exacte, qui demande un temps de calcul beaucoup plus long par rapport aux méthodes approchées. Il n'existe pas une méthode exacte universellement plus rapide que toutes les autres. Chaque problème possède des méthodes mieux adaptées que d'autres. Nous allons présenter un exemple d'algorithme de ce type, nommé procédure par séparation et évaluation (PSE) ou en anglais Branch and Bound.

Une PSE est un algorithme qui permet d'énumérer intelligemment toutes les solutions possibles. En pratique, seules les solutions potentiellement de bonne qualité seront énumérées, les solutions ne pouvant pas conduire à améliorer la solution courante ne sont pas explorées.

Pour représenter une PSE, nous utilisons une structure d'arbre de recherche constitué :

- De nœuds ou sommets, où un nœud représente une étape de construction de la solution.
- D'arcs pour indiquer certains choix faits pour construire la solution.

Dans le cas du sac à dos, un nœud de niveau i représente le nombre d'exemplaires de l'objet i qui ont été pris. L'ensemble des arcs, reliant les nœuds de la racine à un nœud feuille, constitue une solution réalisable.

Trouver la solution optimale revient à calculer la valeur "gain" du sac à dos pour chaque nœud feuille acceptable et prendre la solution ayant le plus grand gain. Cependant la taille de l'arbre de recherche est exponentielle en le nombre d'objets.

Les procédures par séparation et évaluation (PSE) permettent d'élaguer quelques branches de l'arbre pour ne laisser que les branches dont il est susceptible qu'ils améliorent la solution courante. Ce mécanisme d'élagage utilise la notion de borne inférieure et supérieure de la fonction objective :

- Une borne supérieure est une valeur maximale de la fonction objective. Autrement dit, nous savons que la meilleure solution a nécessairement une valeur plus petite. Un majorant de toutes les solutions d'un nœud donné peut s'obtenir en remplissant le volume restant avec l'objet qui rapporte le plus par unité de volume. Ce processus de calcul du majorant est appelé **évaluation d'un nœud**.

À partir d'un nœud et de son évaluation, nous savons que les solutions descendantes de ce nœud ne pourront pas avoir une valeur plus grande que cette borne supérieure. Si jamais, à un nœud donné, la valeur de l'évaluation est inférieure à la valeur de la borne inférieure, alors il est inutile d'explorer les nœuds descendants de celui-ci. On dit qu'on coupe ou on **élague l'arbre de recherche**.

Pour le critère de séparation et à partir d'un nœud donné correspondant à l'objet b_i, on génère k nœuds fils tel que $k = int\left(\frac{V}{v_{i+1}}\right)$ (v_{i+1} : volume de l'objet b_{i+1} et V est le volume restant dans le sac à dos), donc on fixe la valeur de la variable x_{i+1} à ses valeurs possibles de $x_{i+1} = k$ jusqu'à $x_{i+1} = 0$. Les objets sont triés par ordre décroissant de leur utilité. Le produit le plus intéressant étant celui qui rapporte le plus par unité de volume (utilité de l'objet $b_i = \frac{g_i}{v_i}$ tel que g_i est le gain de l'objet b_i et v_i et le volume de cet objet).

2.3.1.2 Programmation Dynamique

La programmation dynamique est une méthode algorithmique visant à résoudre des problèmes d'optimisation. Cette résolution consiste en la décomposition du problème en sous problèmes de plus en plus petits en sauvegardant les résultats intermédiaires. Cette sauvegarde permet d'éviter de calculer deux fois la même chose.

La programmation dynamique est une méthode ascendante puisqu'elle consiste à d'abord résoudre les sous problèmes les plus petits puis remonter vers le problème initial.

Cette méthode est particulièrement utilisée pour résoudre les problèmes satisfaisant le principe d'optimalité dont voici l'énoncé : "Dans une séquence optimale, chaque sous-séquence doit aussi être optimale". Cela signifie que la solution optimale d'un problème peut être calculée à partir des solutions optimales des sous problèmes associés.

La programmation dynamique réalise un bon compromis entre la complexité en temps et en espace. En effet, le stockage des résultats intermédiaires offre un gain de temps considérable et permet d'éviter le basculement dans une complexité exponentielle. Cependant il faut toujours veiller à réduire intelligemment l'espace mémoire occupé.

La programmation dynamique est donc une approche efficace et puissante tout en restant simple. C'est donc une méthode très utilisée dans la résolution des problèmes d'optimisation.

2.3.2 HEURISTIQUES

Une heuristique est une méthode de calcul (algorithme) qui fournit rapidement une solution réalisable, qui n'est pas nécessairement optimale ou exacte, pour un problème d'optimisation combinatoire difficile.

Généralement, une heuristique est conçue pour un problème particulier en s'appuyant sur sa structure propre, ces heuristiques sont appelées **heuristiques spécifiques**, mais les approches peuvent contenir des principes plus généraux.

Dans la famille des heuristiques spécifiques, on trouve les heuristiques gloutonnes (par construction) et les heuristiques par amélioration (de voisinage).

2.3.2.1 HEURISTIQUE PAR CONSTRUCTION

Une heuristique gloutonne est un paradigme algorithmique qui consiste à construire une solution étape par étape en choisissant à chaque étape l'option qui est localement la meilleure sans égard à conséquences futures.
Voici ci-dessous trois heuristiques gloutonnes :

- Weighted Ordered Greedy

  Cette approche consiste à trier les objets selon l'ordre croissant ou décroissant de leurs poids (ou volumes). La liste des objets ainsi triée est parcourue et pour chaque type d'objet, on prend autant de fois que possible d'exemplaires de ce dernier et ainsi de suite jusqu'au parcours complet de la liste d'objets.
  Cette heuristique donne très rarement de bons résultats. Souvent, il est meilleur de trier les objets selon l'ordre décroissant de leurs poids car généralement les objets, à petits volumes, ont un petit gain.

- Density Ordered Greedy

  Dans cet algorithme, les objets sont triés selon l'ordre décroissant de leurs utilités (rapport entre bénéfices et poids). On parcourt la liste triée et pour chaque objet, on prend le nombre maximal possible d'unités de ce dernier.
  C'est la méthode la plus utilisée parmi les méthodes par construction, car elle donne souvent de bons résultats, et dans certains cas elle donne même des solutions optimales alors que dans le pire des cas, la solution obtenue est la moitié de la solution optimale.

- Arrondi

  Cette méthode a le même concept que density ordered greedy (les deux méthodes se base sur l'utilité des objets), la différence est que cette dernière construit la solution pièce par pièce, en choisissant toujours la pièce suivante qui offre l'avantage le plus évident et le plus immédiat sans égard aux conséquences futures. Elle donne de bons résultats qui sont proches de l'optimalité.

2.3.2.2 HEURISTIQUE PAR VOISINAGE (PAR AMÉLIORATION)

Est un processus itératif fondé sur deux éléments essentiels : un voisinage qui est une transformation élémentaire appliquée aux éléments d'une solution S ,et une procédure exploitant le voisinage qui débute avec une solution initiale, et réalise ensuite un processus itératif qui consiste à remplacer la configuration courante par l'un de ses voisins en tenant compte de la fonction coût , le processus s'arrête et retourne la meilleure configuration trouvée quand la condition d'arrêt est satisfaite. Cette dernière concerne généralement un nombre d'itérations ou un objectif à réaliser.

- Recherche locale

  Est une méthode itérative qui explore l'espace de recherche en partant d'une solution initiale S0 et en se déplaçant pas à pas d'une solution à une solution voisine dans le but d'améliorer la solution courante. A chaque itération, la RL cherche une solution meilleure que la solution courante dans le voisinage de cette dernière. Si une telle solution existe alors celle-ci remplace la solution courante et on passe à une autre itération. Sinon, la RL cherche une solution de même coût que la solution courante et relance la recherche à partir de cette nouvelle solution. Cette opération permet une légère diversification. Le processus de recherche s'arrête quand une solution optimale a été trouvée ou en cas de stagnation.

2.3.3 MÉTA-HEURISTIQUE

Un métaheuristique (heuristique générique) est un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficile (souvent issus des domaines de la recherche opérationnelle, de l'ingénierie ou de l'intelligence artificielle) pour lesquels on ne connaît pas de méthode classique plus efficace.

Les métaheuristiques sont généralement des algorithmes stochastiques itératifs, qui progressent vers un optimum global, Elles se comportent comme des algorithmes de recherche, tentant d'apprendre les caractéristiques d'un problème afin d'en trouver une approximation de la meilleure solution.

2.3.3.1 MÉTHODES ITÉRATIVES DE VOISINAGE

Ces méthodes manipulent une seule solution telles que le recuit simulé et la recherche tabou

- Le recuit simulé

  Est une procédure itérative qui cherche des configurations de coût plus faible tout en acceptant de manière contrôlée des configurations qui dégradent la fonction de coût.
  À chaque étape, l'heuristique de recuit simulé considère un état s^* de l'état courant s, et décide de manière probabiliste entre déplacer le système vers l'état s^* ou rester dans l'état s. Ces probabilités conduisent finalement le système à passer à des états de coût inférieur. En général, cette étape est répétée jusqu'à ce que le système atteigne un état suffisamment bon pour l'application, ou jusqu'à ce qu'un budget de calcul donné soit épuisé

2.3.3.2 MÉTHODES ÉVOLUTIVES

Ces méthodes manipulent une population de solutions et se basent en général sur l'évolution naturelle ou sur l'observation des comportements sociaux des individus ou des communautés. On retrouve dans cette classe les algorithmes génétiques, les méthodes basées sur les colonies de fourmis, les essaims d'abeilles et les systèmes immunitaires.

- Les algorithmes génétiques

  Ils sont inspirés des mécanismes de la sélection naturelle et des phénomènes d'évolution génétique : sélection, croisement, mutation et remplacement.
  Son principe est le suivant : pour un problème pour lequel une solution est inconnue, un ensemble de solutions possibles est créé aléatoirement. On appelle cet ensemble la population. Les caractéristiques (ou variables à déterminer) sont alors utilisées dans des séquences de gènes qui seront combinées avec d'autres gènes pour former des chromosomes. Chaque

solution est associée à un individu, et cet individu est évalué et classifié selon sa ressemblance avec la meilleure, mais encore inconnue, solution au problème.
Comme dans les systèmes biologiques soumis à des contraintes, les meilleurs individus de la population sont ceux qui ont une meilleure chance de se reproduire et de transmettre une partie de leur héritage génétique à la prochaine génération. Une nouvelle génération est alors créée en combinant les gènes des parents. On s'attend à ce que certains individus de la nouvelle génération possèdent les meilleures caractéristiques de leurs deux parents, et donc qu'ils seront meilleurs et seront une meilleure solution au problème. La nouvelle génération est alors soumise aux mêmes critères de sélection. Ce processus est répété plusieurs fois, jusqu'à ce que tous les individus possèdent le même héritage génétique. Les membres de cette dernière génération possèdent de l'information génétique qui correspond à la meilleure solution trouvée au problème.

- Les algorithmes par colonie de fourmis

  Ce type d'algorithmes est inspiré par le comportement des fourmis en partant de leurs fourmilières vers une source de nourriture et vice versa. Les fourmis aveugle déposent au sol une substance chimique (qu'on appelle la phéromone) attractive et volatile et forment ainsi des pistes odorantes qui sont exploitées par leurs congénères. dans le cas de l'optimisation par colonie de fourmis, on essaie d'imiter le modèle de colonie de fourmis et de l'améliorer utilement pour résoudre des problèmes complexes. Pour cela on utilise des fourmis artificielles. ces dernières sont caractérisé par :

  - Comme dans les colonies de fourmis réelles, les algorithmes sont composés d'une population de fourmis.
  - Les fourmis artificielles modifient quelques aspects de leurs environnement comme le cas des fourmis réelles. les fourmis réelles disposent de la phéromone, par contre les fourmis artificielles modifient des informations numériques.
  - Pour se déplacer d'un état vers un autre les fourmis artificielles utilisent une règle de décision probabiliste. cette règle utilisent des informations localement disponible et des informations spécifique au problème.
  - le mécanisme d'évaporation de l'information phéromone permet au fourmis artificielles de de diriger la recherche vers de nouvelles directions pour explorer l'espace de recherche.
  - les fourmis artificielles peuvent être enrichies par des capacités supplémentaires pour s'adapter à des problèmes bien spécifiques.

3 Conception

3.1 Algorithmes méthodes exactes

3.1.1 Branch and Bound

- **Organigramme de l'algorithme :**

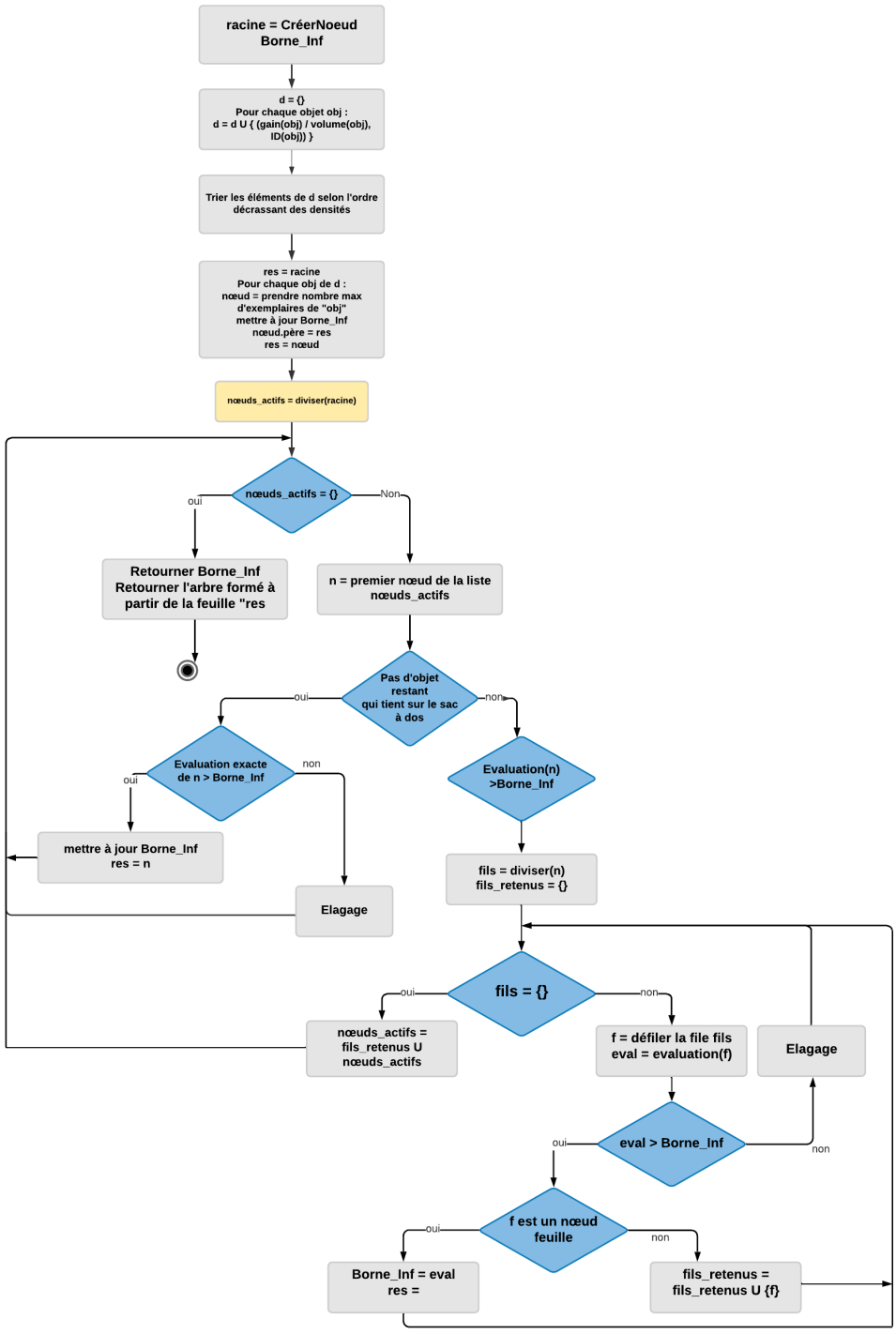


FIGURE 1 – Organigramme de l'algorithme "Branch & Bound" partie 1

- **Implémentation**

Le Branch and Bound est basé sur le concept d'exploration intelligente d'un arbre à plusieurs niveaux et une branche représente une solution réalisable. Résoudre le problème revient à trouver la branche la plus optimale.
Voici dans ce qui suit, la structure d'un nœud de notre arbre :

```
class noeud:
    def __init__(self, objet, val, pere, poids, m):
        #objet : ID de l'objet et qui correspond aussi au niveau du
        #nœud dans l'arborescence
        self.objet = objet
        #val : nombre d'exemplaires pris pour cet objet
        self.val = val
        #pere : pointeur vers le nœud père. Lorsqu'on trouve une
        #solution, on remonte
        #en utilisant cet attribut jusqu'à arriver à la racine
        self.pere = pere
        #poids : c'est le volume restant dans le sac à dos après
        # l'ajout des "val" exemplaires
        #de l'objet "objet"
        self.poids = poids
        #m : c'est le gain cumulé des objets dans le sac à dos jusqu'à
        #cet objet
        self.m = m
```

Avant d'entamer les fonctions et le programme principale du Branch and Bound, jetant un coup d'œil sur les variables globales qui sont utilisées fréquemment sur notre solution :

- La liste "b" : c'est une liste de bénéfices (gains) de chaque objet. Exemple : $b = [5,4,6,2]$, alors l'objet 1 a un gain de 5, l'objet 2 a un gain de 4, etc.
- La liste "v" : c'est une liste des volumes de chaque objet. Exemple : $v = [49,33,60,32]$ le volume (poids) de l'objet 3 est 60.
- La variable "w" contiendra le volume du sac à dos.
- La variable "n" contiendra le nombre d'objets existants.
- La liste "d" : c'est une liste des densités (ou utilités) de chaque objet. En effet, elle contient des couple (a,b) tel que : a est la densité (bénéfice / volume) de l'objet b. On aura besoin de l'ID de l'objet parce que la liste "d" va être triée et donc les objets vont être mélangés. Mais en gardant les ID on pourra, en un temps $o(1)$, de trouver dans la liste "b" et "v" le gain et le poids de l'objet respectivement.

La méthode Branch and Bound a besoin d'une solution initiale pour initialiser la borne inférieure. Voici la fonction python qui implémente l'heuristique Density ordered Greedy :

```
def first_solution(d,b,v,w,racine):
    #La première solution est trouvée grace à l'heuristique : Density
    #Ordered Greedy
    #La variable "M" stockera le gain de la solution proposée
    #par cette heuristique
```

```
    M = 0
    n = racine
    #On fait un parcours complet des objets
    for i in range(len(d)):
        #Si la taille restante du sac à dos est égale à 0, on arrete
        #le parcours
        if w==0:
            break
        #Sinon, s'il reste de l'espace dans le sac à dos :
        #On calcule dans "nb" le nombre d'exemplaires possibles de
        #l'objet d[i][1]
        nb = int(w/v[d[i][1]])
        #On met à jour le gain "M" après l'ajout de "nb" exemplaires
        #de l'objet d[i][1]
        M += nb * b[d[i][1]]
        #On met à jour le poids restant dans le sac à dos "w"
        w -= nb * v[d[i][1]]
        #On construit notre branche au fur à mesure, elle correspond
        #à la branche la plus à droite dans l'arbre du branch and bound
        n = noeud(i+1,nb,n,w,M)
    #On retourne le gain du sac à dos et le nœud feuille qui nous
    #permet de trouver tous les nœuds de la branche à la fin
    return M,n
```

La fonction de séparation est simple à implémenter grâce aux structures de données utilisées. Voici le code python correspondant :

```
def diviser(n,b,v,d):
    #Trouver l'ID de l'objet qui suit l'objet du nœud "n"
    ind = d[n.objet][1]
    #Etant donné un nœud "n", on calcule dans "nb" combien
    #d'exemplaires de l'objet suivant sont possibles à mettre dans
    #le sac à dos
    nb = int(n.poids/v[ind])
    #On retourne une liste de "nb+1" nœuds, le premier ajoute au sac
    #à dos "nb" exemplaires le deuxième "nb-1" exemplaires et ainsi de
    #suite jusqu'à le dernier qui n'ajoute rien
    return [noeud(n.objet+1,i,n,n.poids-i*v[ind],n.m+i*b[ind])
    for i in range(nb,-1,-1)]
```

La fonction d'évaluation est similaire à celle du cours :

```
def evaluer(n,d):
    if n.objet==len(d):
        #Si le nœud correspond à une feuille donc on retourne la
        #solution exacte qui se trouve dans l'attribut "m"
        return n.m
    #Sinon, on utilise la meme fonction d'évaluation du cours
    return n.m+n.poids*d[n.objet][0]
```

Après avoir implémenter les fonctions de séparation, d'évaluation et d'initialisation de la borne inférieure, arrive le temps de parler de la fonction principale. Le code suivant implémente la fonction du branch and bound :

```
def branch_and_bound_ukp(b, v, w):
    #Initialisation du nœud racine
    racine = noeud(0,-1,None,w,0)
    #Construction de la liste des densités (utilités des objets)
    d = [(b[i]/v[i],i) for i in range(len(v))]
    #Ordonner la liste des densités par ordre décroissant
    d.sort(key=lambda x:x[0], reverse=True)
    ############################
    #Ce bloc permet de retourner une liste "min_w" tel que min_w[i]
    #est le volume minimal  des objets de la sous liste d[i :: nb_objets].
    #Cette liste va nous servir dans  l'optimisation et l'anticipation
    #de l'élagage dans certains cas
    mini = v[d[-1][1]]
    min_w = []
    for i in range(len(d)-1,-1,-1):
        mini = min(mini, v[d[i][1]])
        min_w.insert(0, mini)
    ############################
    #Trouver une première solution. "M" c'est la borne inférieure et
    # "res" c'est la solution actuelle
    M,res = first_solution(d,b,v,w,racine)
    #Initialisation de la liste "na", liste des nœuds actifs, avec
    #le résulat de la séparation du nœud racine
    na = diviser(racine,b,v,d)
    #Tant qu'il y aura de nœuds actifs, on termine notre exploration
    #de l'arbre
    while len(na)!=0:
        #On récupère le premier nœud actif dans "n" (on prend le
        #premier nœud pour  assurer une exploration en profondeur
        #de l'arbre)
        n = na.pop(0)
        #S'il y a aucun autre objet qui peut tenir dans le volume
        #restant dans le sac à dos, alors on élage et on modifie
        #la borne inférieure si notre branche donne un résultat
        # meilleur (on remarque l'utilisation de la liste "min_w"
        #pour  optimiser la vérification)
        if n.poids<min_w[n.objet-1]:
            if n.m>M:
                M = n.m
                res = n
        #Dans le cas contraire, si l'évaluation de notre nœud donne
        #une valeure inférieure à "M" on élague. Sinon, on sépare le
        #nœud et pour chaque nœud fils "f", si son évaluation peut
        #améliorer la solution on le garde sinon on élague.
        #Si le nœud fils "f" est une feuille et en plus il améliore
```

```
        # la solution courante, on le prend comme nouvelle solution.
        #PS : on remarque l'utilisation de la fonction python int()
        #après chaque évaluation, le but c'est de retourner une
        #approximation à l'entier qui est juste inférieure à
        # l'évaluation et ça nous permet d'élaguer le plutot possible.
        elif int(evaluer(n,d))>M:
            fils = diviser(n,b,v,d)
            fils_retenus = []
            for f in fils:
                evaluation = evaluer(f,d)
                if int(evaluation)>M: #int(eval)>M : pour optimiser
                    if f.objet==len(d):
                        M = evaluation
                        res = f
                    else:
                        fils_retenus.append(f)
            na = fils_retenus + na
    #Remonter dans l'arborescence en utilisant l'attribut "pere" afin
    #de retourner la solution exacte trouvée
    sol = [0 for _ in range(len(b))]
    M = res.m
    while res.val!=-1:
        sol[d[res.objet-1][1]] = res.val
        res = res.pere
    return M,sol
```

3.1.2 PROGRAMMATION DYNAMIQUE

La résolution du problème du sac à dos dans sa version non bornée à l'aide de la programmation dynamique repose sur l'utilisation d'un tableau que l'on notera K. Ce dernier contiendra le gain maximal qui peut être obtenu pour un poids variant de 0 à W (W étant la capacité du sac). Pour chacun de ces poids, il existe une itération correspondante dans l'algorithme. Cette itération consiste à appliquer les traitements suivants sur chaque objet.
Vérifier si le poids de l'objet ne dépasse pas le poids considéré dans l'itération.

- Si l'objet améliore le gain alors il est inclus dans la solution, sinon il est ignoré.
- Le gain maximal correspond à la valeur de $K[W]$. Il est possible de récupérer la séquence d'objets générant ce bénéfice optimal en mettant à jour une liste "items" à chaque itération de l'algorithme. Cette dernière contiendra donc la liste des objets générant le gain maximal pour un poids variant de 0 à W. La séquence d'objets optimale est donc donnée par items $[W]$.

La démarche décrite peut être résumée par le pseudo algorithme suivant :

```
K [0] =0
Items [0]= Ø
Pour x variant de 1 à W
   K [x] =0
   Pour i variant de 1 à n (n étant le nombre de type d'objet)
```

```
      Si (poids objet i <= x)
         K [x] = max { K [x], K [x- poids objet i] + gain objet i}
         Si k [x] a été modifié
            items [x] = items [x- poids objet i] U { objet i}
         fin si
      fin si
   fin pour
fin pour
```

La programmation dynamique s'applique très facilement au problème du sac à dos. La complexité correspondante est $o(nW)$, n étant le nombre de types d'objet et W la capacité du sac.

- **Implémentation**

  Nous avons implémenté le pseudo algorithme précédent en utilisant le langage Python. Nous allons décrire dans ce qui suit le programme résultant et nous allons expliquer chaque étape de ce dernier.
  Pour commencer nous avons défini une fonction dp ukp qui englobe le pseudo code précédent. Elle possède les paramètres suivants :

  – w : la capacité du sac
  – n : le nombre de types d'objets
  – b : la liste des gains de chaque objet
  – v : la liste des poids ou volumes de chaque objet

  Au début de cette fonction, nous avons déclaré deux variables :

  – K : une liste qui contiendra le gain maximal associé à des poids variant de 0 à w. Au départ, ces gains étaient initialement à 0. Cette liste nous permettra d'appliquer le principe de la programmation dynamique. En effet, nous obtiendrons le gain maximal associé au poids w à travers les gains associés aux poids qui lui sont inférieurs.
  – Items : il s'agit d'une liste de listes. Items contient donc la liste des objets permettant d'obtenir le gain maximal associé à des poids allant de 0 à w.

```

def dp_ukp(w,n,b,v):
    # k contient le gain maximal associé aux poids allant de 0 à w (poids
    #max)
    k=[0 for i in range(w+1)]

    # items contient la liste des objets choisis pour obtenir le gain
    # maximal associé aux poids allant de 0 à w
    items=[[] for i in range(w+1)]

```

Le programme consiste ensuite en une boucle parcourant des poids variant de 0 à w. Pour chacun de ces poids, nous allons parcourir tous les objets, nous obtenons donc les deux boucles imbriquées suivantes :

```
for wi in range(w+1):
        for j in range(n):
```

Pour chaque objet d'indice j, nous commençons par vérifier si son poids $v[j]$ ne dépasse pas le poids en cours w_i. Dans le cas où il le dépasse, cet objet n'est pas pris en compte pour le poids w_i et nous passons à l'objet suivant. Dans le cas contraire, cet Objet peut potentiellement améliorer la solution. Nous commençons par sauvegarder le gain courant associé au poids w_i ($k[w_i]$) dans une variable tmp. Puis on choisit d'intégrer l'objet d'indice j à la solution dans le cas où il améliore le gain sinon l'objet n'est pas choisi. Il s'agit donc de prendre le maximum entre $k[w_i]$ qui est le gain courant et $(k[w_i - v[j]] + b[j])$ qui n'est autre que le gain obtenu en ajoutant l'objet d'indice j au sac. Cela signifie que la capacité du sac sera réduite de $v[j]$ (le poids de l'objet ajouté). Cette démarche est réalisée par le code suivant :

```
if (v[j]<=wi): #si le poids de l'objet est inférieur au poids considéré
                tmp=k[wi] # tmp sera utilise pour savoir si k[wi] a été
                          # modifié (pour modifier items en conséquences)
                k[wi]=max(k[wi],k[wi-v[j]]+b[j])
```

Après avoir défini si l'objet sera choisi ou pas, il est nécessaire de mettre à jour la liste des objets choisis. En effet, si l'objet n'est pas pris la liste reste inchangée sinon il faut y ajouter l'exemplaire de l'objet choisi. Pour savoir si l'objet a été pris, il suffit de vérifier si le gain $k[w_i]$ a été modifié. Voilà pourquoi nous avons défini la variable tmp qui sauvegarde le gain de l'itération précédente. Si $k[w_i]$ est égal à tmp alors le gain n'a pas été amélioré et la liste d'objets est de ce fait inchangée. Dans le cas contraire, il faut mettre à jour cette dernière. Elle contiendra donc les objets constituant la solution du poids égal à $(w_i - v[j])$ ce qui correspond à items $[w_i - v[j]]$ auquel on ajoute l'objet j. Voici donc le code correspondant :

```
if (k[wi]>tmp): # si k[wi] a changé (donc on a trouvé une val supérieur à
#la valeur précédente sauvegardée dans tmp), on met à jour les objets pris
if (k[wi]>tmp): # si k[wi] a changé (donc on a trouvé une val supérieur
                    # à la valeur précedente sauvgardée dans tmp), on met
                    #  à jour les objets pris
                     items[wi]=[]
                     for l in range(len(items[wi-v[j]])):
                         items[wi].append(items[wi-v[j]][l])
                     items[wi].append(j+1)
```

Pour terminer la fonction retourne $k[w]$ qui correspond au gain maximal. La fonction complète obtenue est donc la suivante :

```
def dp_ukp(w,n,b,v):
    # k contient le gain maximal associé aux poids allant de 0 à w (poids
    # max)
    k=[0 for i in range(w+1)]
```

```
    # items contient la liste des objets choisis pour obtenir le gain
    # maximal associé aux poids allant de 0 à w
    items=[[] for i in range(w+1)]

    for wi in range(w+1):
        for j in range(n):
            if (v[j]<=wi): #si le poids de l'objet est inférieur au poids
                           #considéré
                tmp=k[wi] # tmp sera utilise pour savoir si k[wi] a été
                 # modifié (pour modifier items en conséquences)
                k[wi]=max(k[wi],k[wi-v[j]]+b[j])
                if (k[wi]>tmp): # si k[wi] a changé (donc on a trouvé une
# val supérieur à la valeur précedente sauvgardée dans tmp),
# on met à jour les objets pris
                    items[wi]=[]
                    for l in range(len(items[wi-v[j]])):
                        items[wi].append(items[wi-v[j]][l])
                    items[wi].append(j+1)
                    # donc la liste des objets pris est la liste des objets
                 # de items[wi-wt[j] en plus de l'objet j ajouté
    return k[w],items[w]

```

3.2 Algorithmes heuristiques

3.2.1 Heuristiques par construction

3.2.1.1 Heuristique par arrondi

Cette heuristique, basée sur l'approche Density ordre greedy, construit une solution pièce par pièce, en choisissant toujours la pièce suivante qui offre l'avantage le plus évident et le plus immédiat sans égard aux conséquences futures. Elle donne des bons résultats qui sont proches de l'optimalité, voici le pseudo algorithme représentant cette méthode :

- Calculer le rapport $\left(\frac{v_i}{p_i}\right)$ pour chaque objet i ;
- Trier tous les objets par ordre décroissant de cette valeur ;
- Mettre à 0 toutes les variables représentant chacune, dans un tableau, un type d'objet ;
- Sélectionner les objets un à un dans l'ordre du tri et ajouter l'objet sélectionné dans le sac si le poids maximal reste respecté et mettre à 1 la variable qui représente cet objet ;
- Faire jusqu'à l'obtention de différence minimum entre la capacité du sac et la capacité des objets ajoutés :
  - Sélectionner les objets déjà ajoutés dans le sac un à un dans l'ordre du tri et ajouter l'objet sélectionné dans le sac si le poids maximal reste respecté ;
  - Incrémenter la variable représentant cet objet ;
  - Calculer la différence entre la capacité du sac et la capacité des objets ajoutés ;

L'implémentation de cet algorithme est faite par la construction de trois procédures sous python :

- **Ratios()** : calcule l'utilité des objets et retourne la liste triée par ordre descendant.
- **Intermediate solution()** : qui calcule la première solution pour un seul exemplaire d'objet.
- **Heuristic arrondi()** : qui fait le calcul de la solution pour N exemplaires et retourne la liste, le nombre des objets ainsi que la capacité pris par l'ensemble des objets sélectionnés.

```
def ratios(b,v,V_t):
    ratios = [(b[i]/v[i],i) for i in range(len(b))]
    ratios.sort(reverse=True)
    return ratios


def intermediate_solution(b,v,V_t):
    x = [0 for i in range(len(b))]
    ratio = ratios(b,v,V_t)
    cap_act = 0
    capacite = V_t
    cap = cap_act
    i = 0
    benefice = 0
    while( capacite >= cap_act + v[ratio[i][1]]):
        if(i==len(b)):
            i = 0
        benefice += b[ratio[i][1]]
        cap_act += v[ratio[i][1]]
        x[ratio[i][1]] = x[ratio[i][1]]+1
        i += 1
    return x,cap_act,benefice,i

def heuristic_arrondi(b,v,V_t):
    x,capacite,benefice,i_max = intermediate_solution(b,v,V_t)
    ratio = ratios(b,v,V_t)
    difference = V_t - capacite
    difference_min = 100000000
    i = 0
    while(difference_min != difference):
        if(i == i_max):
            i = 0
        if(x[ratio[i][1]] == 0):
            i += 1
        else:
            if(capacite + b[ratio[i][1]] <= V_t):
                benefice += b[ratio[i][1]]
                capacite += v[ratio[i][1]]
                x[ratio[i][1]] = x[ratio[i][1]]+1
            i += 1
            difference = V_t - capacite
            difference_min = min(difference , difference_min)
```

```
    return capacite,x,benefice

```

3.2.1.2 DENSITY ORDERED GREEDY

Le code python suivant montre l'implémentation de l'heuristique Density Ordered Greedy (vu en cours), pour plus de détails sur la méthode consulter la partie état de l'art.

```
def density_ordered_greedy_ukp(b, v, w):
    #Construction de la liste des densités (utilités des objets)
    d = [(b[i]/v[i],i) for i in range(len(v))]
    #Ordonner la liste des densités par ordre décroissant
    d.sort(key=lambda x:x[0], reverse = True)
    #La variable "M" stockera le gain de la solution proposée par cette
    #heuristique
    M = 0
    #La liste "res" contiendra la solution à la fin. res[i] est le nombre
    #d'exemplaires de l'objet "i"
    res = [0 for _ in range(len(d))]
    #On fait un parcours complet des objets
    for i in range(len(d)):
        #Si la taille restante du sac à dos est égale à 0, on arrete
        #le parcours
        if w==0:
            break
        #Sinon, s'il reste de l'espace dans le sac à dos :
        #On calcule dans "nb" le nombre d'exemplaires possibles de
        #l'objet d[i][1]
        nb = int(w/v[d[i][1]])
        #On met à jour le gain "M" après l'ajout de "nb" exemplaires
        #de l'objet d[i][1]
        M += nb * b[d[i][1]]
        #On met à jour le poids restant dans le sac à dos "w"
        w -= nb * v[d[i][1]]
        #On sauvegarde le résultat "nb" dans la liste "res" à la position
        #d[i][1] (ID de l'objet)
        res[d[i][1]] = nb
    #On retourne le gain du sac à dos "M" et la liste des nombres
    # d'exemplaires de chaque objet "res"
    return M,res


```

3.2.1.3 WEIGHTED ORDERED GREEDY

L'heuristique Weighted Ordered Greedy est similaire à la Density Ordered Greedy en termes d'implémentation sauf qu'on manipule les volumes des objets et pas les densités de ces derniers. Le détail du programme est montré ci-dessous :

```python
def weighted_ordered_greedy_ukp(b, v, w, ordre_croissant=False):
    #Construction de la liste "d" contenant les couples (a,b) tel que a est
    #le volume de l'objet b. Cette structure nous permet de trouver le
    #gain de l'objet  rapidement après le trie de cette liste.
    d = [(v[i],i) for i in range(len(v))]
    #Ordonner la liste "d" selon l'ordre croissant ou décroissant selon
    # le paramètre ordre_croissant
    d.sort(key=lambda x:x[0], reverse=not ordre_croissant)
    #La variable "M" stockera le gain de la solution proposée par cette
    #heuristique
    M = 0
    #La liste "res" contiendra la solution à la fin. res[i] est le nombre
    #d'exemplaires de l'objet "i"
    res = [0 for _ in range(len(d))]
    #On fait la meme boucle que celle du Density Ordered Greedy
    for i in range(len(d)):
        if w==0:
            break
        nb = int(w/v[d[i][1]])
        M += nb * b[d[i][1]]
        w -= nb * v[d[i][1]]
        res[d[i][1]] = nb
    #On retourne le gain du sac à dos "M" et la liste des nombres
    #d'exemplaires de chaque objet "res"
    return M,res
```

3.2.2 Heuristiques par amélioration

3.2.2.1 Recherche locale

- **Principe de l'algorithme**

1. **L'initialisation :**
   - Solution initiale
   - Nombre d'itérations
2. **Génération du voisin :**
   - Choisir aléatoirement s' dans $V(s)$
3. **Calcul delta :**
   - $\Delta F = F(s) - F(s_i)$
4. **Critère d'arrêt :**
   - Atteindre un nombre maximal d'itérations

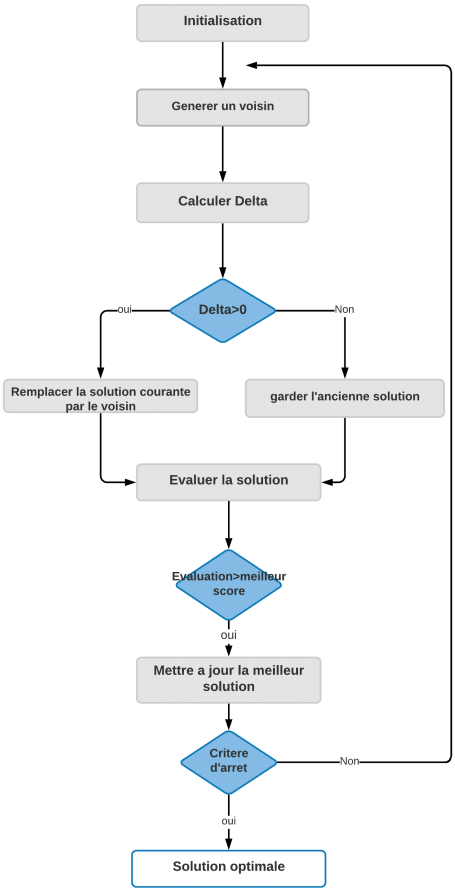


FIGURE 2 – Organigramme de l'heuristique par amélioration (recherche locale)

- **Pseudo algorithme**

```
Début
        initialisation nbr_iterations=nombre d itérations
        générer une solution initiale
        poser s*=s
        Tant que nbr_iterations>0 faire:
            choisir aléatoirement s' dans v(s)
            si F(s')<F(s) poser s=s'
               nbr_iterations=nbr_iterations-1
        Fin Tant que
Fin
```

- **Pseudo algorithme**
  La solution Initiale : générée soit aléatoirement en faisant appel à la fonction gen-random-sol().
  Le nombre d'itérations : Argument en entré de la méthode

- **Implémentation de l'algorithme**

– **Passage d'une solution à une autre**
Cette méthode assure le passage d'une configuration à une autre commence par générer une solution voisine l'évaluer si cette dernière donne un gain supérieur à celui de la solution courante on met à jour la solution (on accepte que des solutions voisines meilleurs que la solution courante)

```
def getNextState_heuristic( solution,taille,tab_poids_new,
                                          tab_gain_new, capacity):
        # generer une solution voisine
        newSolution =
        getNeighbour(solution, taille, tab_poids_new, capacity)
        # generer evaluer la solution generée
        evalNewSol= eval_solution(newSolution,tab_gain_new)
        # evaluation l'ancienne solution
        evalOldSol= eval_solution(solution,tab_gain_new)
        delta = evalNewSol - evalOldSol
        # solution ameliorante acceptée sinon garder l'ancienne
        if (delta > 0):
            return newSolution
        else :
            return solution
```

– **Fonction principale**

* La fonction reçoit en entrée un tableau contenant les différents objets, leurs poids et leurs gains, la capacité du sac, la solution initiale et le nombre d'itérations.
* A l'entrée de la méthode on trie la liste des objets par ordre décroissant de leurs utilités.
* On calcule le nombre maximal que peut contenir le sac de chaque objet et on stocke le résultat dans un tableau.
  $[[7,20],[2,5],[5,10],[3,2]]$ représente le tableau des objets en entrée (7 représente le poids du premier objet et 20 représente son gain).
  $[2,10,4,6]$ 2 représente le nombre maximal d'unités de l'objet 1 (de poids 7) que peut contenir le sac (de capacité 20).
* Puis en transforme la solution initiale n aire en une solution binaire
  $[1,1,0,2]$ devient $[1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1]$
  En utilisant le tableau représentant le nombre maximal que peut contenir le sac de chaque objet $[10,6,4,2]$ déjà calculé
  $[1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1]$
  Si au niveau de la solution n aire on a une m unités de l'objet on met les m premiers bits à 1 et le reste a 0.
* En utilisant ce tableau $[2,10,4,6]$ on calcule les deux tableaux gain par unité et poids par unité $[[7,20],[2,5],[5,10],[3,2]]$
  On duplique le gain autant de fois qu'au nombre d'exemplaire s que peut contenir le sac de chaque objet $[20,20,5,5,5,5,5,5,5,5,5,5,10,10,10,10,2,2,2,2,2,2]$
* Pour le nombre d'itérations précisées en entrée de la méthode on génère une solution voisine
  Pour générer la solution voisine on manipule la solution sous sa forme binaire.
  $[1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1]$
  $[1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1]$

On générer un indice aléatoirement si sa valeur est à 1 on l'altère a 0 sinon on cherche des indices qu'on peut altérer de 0 à 1 (tout en tenant compte de la contrainte du poids) et ont choisi un parmi aléatoirement si aucun n'est trouvé on cherche des indices qu'on peut altérer de 1 à 0 et ont choisi un parmi aléatoirement.
Puis on recherche des indices qu'on peut altérer de 0 à 1 (tout en tenant compte de la contrainte du poids) et on choisit un parmi eux aléatoirement.

* On l'évalue si elle améliore le gain on met à jour la meilleure solution.
* A la sortie de la méthode on convertit la solution binaire en une solution n aire.

```
def amelioration_method(itemsIn,capacity,solinit,iterations):
    items=itemsIn.copy()
    for i in range(len(items)):
        items[i].append(solinit[i])
    # trier les objets par utilié
    items_sorted=trier_objet_utility(items)
    # reordonner la solution
    solinitsorted=[]
    for i in range(len(items_sorted)):
        solinitsorted.append(items_sorted[i][2])
    # recupere le tabeau contenant le nombre max d'exemplaires
    # de chaque objet
    tab_max_nb,taille= get_max_number_item(items_sorted,
    capacity)
    tab_poids_new= get_tab_poid_new(items_sorted, tab_max_nb)
    tab_gain_new= get_tab_gain_new(items_sorted,tab_max_nb)
    # convertir la solution en une solution binaire
    solCurrent= ntobinary(solinitsorted, tab_max_nb)
   # evaluer la solution
    evalsol= eval_solution(solCurrent,tab_gain_new)
    # initialiser la meilleur solution
    bestSol= solCurrent.copy()
    bestEval= evalsol

    while (iterations>0):

            # passage a une nouvelle configuration
            solCurrent =getNextState_heuristic(solCurrent,
        taille,tab_poids_new, tab_gain_new, capacity)
            # evaluer la nouvelle configuation
            evalCurrent=eval_solution(solCurrent, tab_gain_new);
            # si meilleur MAJ de la meilleur solution
            if evalCurrent > bestEval:
                bestSol= solCurrent.copy()
                bestEval=evalCurrent

            iterations=iterations-1

    objects=[]
    solution=[]
    #convertir la solution binaire trouver en une solution en n
```

```
    Nsol= binaryToNsolution(bestSol, tab_max_nb)
    for i,item in enumerate(Nsol):
        if item!=0:
            objects.append(items[i])
            solution.append(item)
    poids=0
    for i,obj in enumerate(objects):
        poids+=obj[0]*solution[i]
    # retourne la solution son gain et son poids
    return objects,solution, Nsol, bestEval,poids
```

3.3 ALGORITHMES MÉTA-HEURISTIQUES

3.3.1 MÉTA-HEURISTIQUE ITÉRATIVE

3.3.1.1 LE RECUIT SIMULÉ

Le recuit simulé est basé sur les deux principes exploration (accepter des solutions voisine de coût inférieur à la solution courante) et exploitation de l'ensemble des solutions à chaque fois mettre à jour la meilleure solution et son coût jusqu'à atteindre le critère d'arrêt (température minimale ou nombre d'itérations).

- **Principe de l'algorithme**

  1. **L'initialisation :**
     - Solution initiale.
     - Facteur de refroidissement.
     - Température initiale.
     - Le nombre de paliers pour chaque température.
  2. **Génération du voisin :**
     - Choisir aléatoirement s' dans $V(s)$
  3. **Calcul delta :**
     - $\Delta F = F(s) - F(s_i)$
  4. **Calcul proba d'acceptation :**
     - $proba = eF(s) - \frac{F(s')}{T}$
     - Comparer la proba d'acceptation et le réel r.
  5. **Mettre à jour la température :**
     - $T = \alpha T$
  6. **Critère d'arrêt :**
     - Atteindre un nombre maximal d'itérations ou une température inférieure.

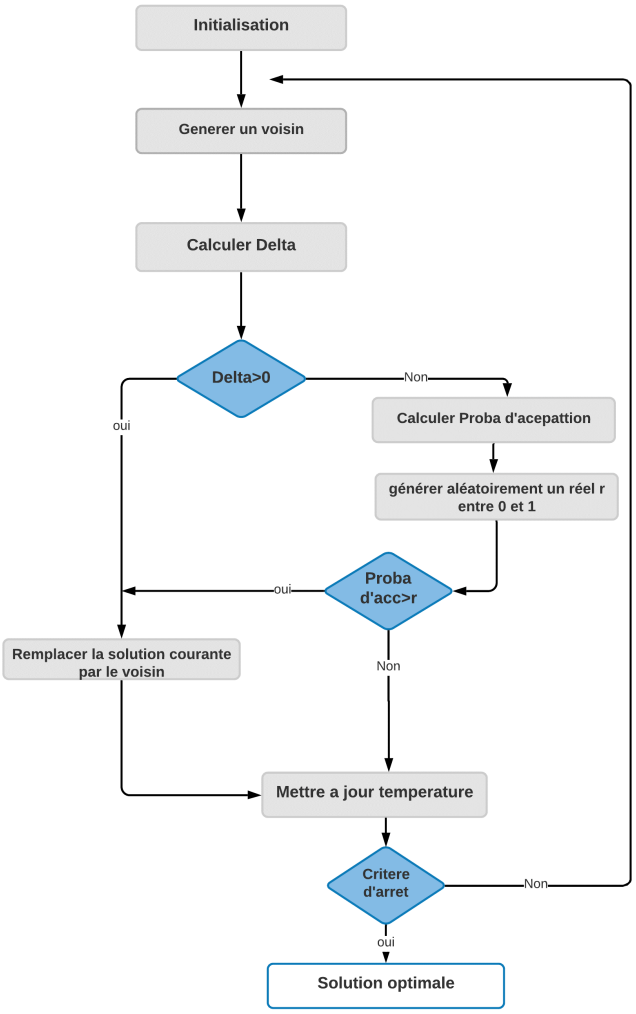


FIGURE 3 – Organigramme de l'algorithme du recuit simulé

- **Pseudo algorithme**

```
Début
        initialisation T=température initiale
        générer une solution initiale
        poser s*=s ,choisir T
        choisir $\alpha \in$(0,1)
        Tant que critère d arrêt non satisfait faire:
            Répéter R fois :
               choisir aléatoirement s' dans v(s)
               si F(s')<F(s) poser s=s'
               sinon
                  générer un réel r entre 0 et 1
               si r<e(F(s)-F(s')/T) poser s=s'
               Fin si

            Fin répéter
        poser T= $\alpha$T
```

```
        Fin Tant que
Fin
```

- **Paramètres de l'algorithme**
  La solution Initiale : générée soit aléatoirement, en faisant appel à la fonction gen-random-sol() soit avec une méthode constructive(nous allons utiliser les résultats de l'heuristique constructive density ordered greedy).
  La température initiale : Argument en entré de la méthode.
  Facteur de refroidissement : Argument en entré de la méthode.
  Le nombre d'itérations pour chaque température : Stratégie de refroidissement par palier.
  La stratégie du voisinage : passage d'une solution a une solution voisine en faisant appel à la fonction getneighbor().
  Le critère d'arrêt de l'algorithme : une température minimale.

- **Implémentation de l'algorithme**

  - **Les fonctions nécessaires**

    * **La fonction de refroidissement**
      Cette fonction met à jour la température.

```python
# Mise a jour de la temperature
def cool(temprature, coolingFactor):
    return temprature* coolingFactor
```

    * **Evaluation d'une solution obtenue**
      Cette fonction calcule le gain total d'une solution en multipliant le nombre d'exemplaires des objets avec leurs gains.

```python
# evaluation d'une solution obtenue
def eval_solution(solution, tab_gain_new):
    gain_total=0
    for i in range(len(solution)):
        gain_total= gain_total + solution[i]*tab_gain_new[i]

    return gain_total

```

    * **Définir le nombre maximal que peut contenir le sac de chaque objet**
      Cette fonction calcule le nombre maximal que peut contenir le sac de chaque objet et stocke le résultat dans un tableau.

```python
# le nombre maximal que peut contenir le sac de chaque objet
def get_max_number_item(items, capacity=0):
    tab_number= [capacity//item[0] for item in items]
    return tab_number, sum(tab_number)
```

* **Définir la table des gains pour chaque unité des objets**
Cette fonction retourne un tableau contenant le gain de chaque unité de chaque objet

```
1
2  #definir une table de gain correspondante a l'écriture
3  #binaire d'une solution
4  def get_tab_gain_new(items_sorted, tab_max_nb):
5      tab_gain = []
6      for i in range(len(tab_max_nb)):
7          tab= [items_sorted[i][1]]*tab_max_nb[i]
8          tab_gain= tab_gain + tab
9
10     return tab_gain
```

* **Définir la table des poids pour chaque unité des objets**
Cette fonction retourne un tableau contenant le poids de chaque unité de chaque objet.

```
1
2  #definir une table de poids  correspondante a l'ecriture
3  #binaire d'une solution
4  def get_tab_poid_new(items_sorted, tab_max_nb):
5      tab_poid = []
6      for i in range(len(tab_max_nb)):
7          tab= [items_sorted[i][0]]*tab_max_nb[i]
8          tab_poid= tab_poid + tab
9      return tab_poid
10
```

* **Convertir une solution n en une solution binaire**
Cette fonction convertit une solution n aires en une solution binaire.

```
1
2  # convertir une solution en n en une forme binaire
3  def ntobinary(nsol, max_num_tab):
4      bsol=[]
5      for i in range(len(max_num_tab)):
6          for p in range(nsol[i]):
7              bsol.append(1)
8          for p in range(nsol[i], max_num_tab[i]):
9              bsol.append(0)
10     return bsol
11
```

* **Convertir une solution n en une solution binaire**
Cette fonction convertit une solution binaire en une solution n aire, Si au niveau de la solution n aire on a une m unités de l'objet on met les m premiers bits à 1 et le reste a 0 (le reste jusqu'au nombre maximal d'exemplaires de cet objet que peut contenir le sac).

```
# convertir une solution binaire en une solution en n
def binaryToNsolution(solution, tab_max_nb):
    solN= []
    indMin=0
    for i in range(len(tab_max_nb)):
        indMax= indMin+tab_max_nb[i]
        solN.append(sum(solution[indMin:indMax]))
        indMin = indMax
    return solN

```

* **Définir le poids pris par une solution écrite sous sa forme binaire**
  Cette fonction calcule le poids total d'une solution en multipliant le nombre d'exemplaires des objets avec leurs poids.

```
#le poids obtenue par une solution ecrite sous sa forme binaire
def get_poids_total(bsol,tab_poid_new ):
    poid_total = 0
    for i in range(len(bsol)):
        poid_total = poid_total+ bsol[i]*tab_poid_new[i]
    return poid_total
```

* **Trier les objets par utilité**

  Cette fonction trie les objets par ordre décroissant de leurs utilités.

```
# trier par utilité
def trier_objet_utility(items ):
    items.sort(key=lambda x: x[1]/x[0], reverse=True)
    return items

```

* **Définir le voisinage d'une solution**
  Cette fonction retourne la première solution voisine trouvée.
  On choisit aléatoirement un indice de la solution binaire si sa valeur est à 1 on enlève cet exemplaire sinon on parcourt la solution pour trouver un exemplaire aléatoire non pris de poids inferieur a la capacité restante et l'altérer a 1 si tous les objets restants ont un poids supérieur à la capacité restante on parcourt la solution pour trouver un exemplaire aléatoire déjà pris et l'enlever.

```
# Definir le voisinage d'une solution
def  getNeighbour(solution,taille, tab_poids_new, capacity):
    np.random.seed()
    sol= solution.copy()
    i=0;
    # generer un indice aleatoire
    x = np.random.randint(taille)
    # alterer la valeur de la solution correspondante a
```

```
    # l'indice de 1 a 0
    if sol[x] == 1:
        sol[x]=0
    # essayer d'alterer une des valeurs de la solution de 0 a 1
    else:
        capacityRest = capacity -
        get_poids_total(sol,tab_poids_new)
        listItemCanEnter=  []
        for i in range( len(sol)):
            if capacityRest>tab_poids_new[i] and sol[i]==0:
                listItemCanEnter.append(i)
        if len(listItemCanEnter) != 0:
            ind= np.random.randint(len(listItemCanEnter))
            sol[listItemCanEnter[ind]]=1
      # essayer d'alterer une des valeurs de la solution de
      #1 a 0
        else:
            listItemPris = []
            for i in range( len(sol)):
                if sol[i]==1:
                    listItemPris.append(i)
            if len(listItemPris) != 0:
                ind= np.random.randint(len(listItemPris))
                sol[listItemPris[ind]]=0
     # essayer d'alterer une des valeurs de la solution de
     # 0 a 1
            capacityRest =
         capacity - get_poids_total(sol,tab_poids_new)
            listItemCanEnter=  []
            for i in range( len(sol)):
                if capacityRest>tab_poids_new[i] and sol[i]==0:
                    listItemCanEnter.append(i)
            if len(listItemCanEnter) != 0:
                ind= np.random.randint(len(listItemCanEnter))
                sol[listItemCanEnter[ind]]=1
    return sol

```

* **Passage d'une solution à une autre**
Cette fonction assure le passage d'une configuration à une autre commence par générer une solution voisine en faisant appel à getneighbor () évalue cette solution si cette dernière donne un gain meilleur en remplace la solution par la solution voisine sinon on accepte une solution qui détériore les résultats à un degré près.

```
#passage d'une solution a une autre
def getNextState( solution,taille,tab_poids_new, tab_gain_new,
                        capacity, temperature):

        # generer le voisin
        newSolution =
```

```
        getNeighbour(solution, taille, tab_poids_new, capacity);
        # evaluer le voisin
        evalNewSol= eval_solution(newSolution,tab_gain_new)
        # evaluer l'ancienne solution
        evalOldSol= eval_solution(solution,tab_gain_new)
        # calculer delta
        delta = evalNewSol - evalOldSol

        if (delta > 0):
            return newSolution  # solution meilleur => accepée
        else :
            x = np.random.rand() # generer un nombre aleatoire

            # critere d'acceptation de la solution
            if (x < math.exp(delta / temperature)) :
                return newSolution
                # verifié => accepter nouvelle solution
            else :
                return solution
                # non verifié => garder l'ancienne solution

```

* **Générer une solution initiale aléatoire**
  Cette fonction est invoquée si on veut commencer la recherche à partir d'une solution aléatoire. La solution est obtenue en ajoutant, à chaque itération, un nombre aléatoire d'exemplaires d'un objet choisi aléatoirement jusqu'au remplissage du sac à dos tout en tenant compte de l'espace restant dans le sac.

```
# generer une solution aleatoire
def gen_random_sol(tab,n,capacity):
    weight=[]
    profits=[]
    capacityleft=capacity
    sol=[]
    # initialiser la solution avec des 0
    for k in range(0,n):
        sol.append(0)
    for i in range(0, n):
        weight.append(tab[i][0])
        profits.append(tab[i][1])
    j=0
    # TQ capacité max non atteinte
    while(j<n and capacityleft>0):
        # generer un indice aleatoire
        index=np.random.randint(0,n-1)
        # calculer le nombre maximale d'exemplaires qu'on peut
        # rajouter
        maxQuantity = int(capacityleft / weight[index]) +1
        if (maxQuantity==0):
            nbItems=0
```

```
        else:# si maxQuantity>0 generer un nombre aleatoire
             # d'exemplaires inferieurs a maxQuantity
            nbItems=np.random.randint(0,maxQuantity)
        sol[index]=nbItems
        capacityleft=capacityleft-weight[index]*sol[index]
        j=j+1

    gain_out=0 # calculer le gain obtenu
    for i in range(n):
        gain_out=gain_out+profits[i]*sol[i]

    return gain_out,capacityleft,sol
```

– **Fonction principale du recuit simulé**
La fonction reçoit entrée la liste des objets (poids et gains des objets), la capacité du sac, la solution initiale, le nombre de palier pour chaque température, la température initiale, le facteur de refroidissement et la température finale

* La méthode trie la liste des objets en entrée par ordre décroissant de leur utilité et ordonne la solution initiale suivant le même ordre.
* Calculer le nombre maximal d'unités que peut contenir le sac de chaque objet ainsi que les deux tables : poids par unité et gain par unité.
* Convertir la solution initiale en une solution binaire et l'évaluer pour définir la borne supérieure du gain.
* Génère la configuration voisine.
* Évaluer la solution voisine si cette dernière est supérieure au meilleur gain met à jour la meilleure solution et la meilleure évaluation.
* Mettre à jour la température et répéter ce processus jusqu'à atteindre la température minimale
* Pour chaque température on répète le processus le nombre de palier fois .

```
# la fonction principale du recuit simulé
def simulatedAnnealing(itemsIn,capacity,solinit,samplingSize,
        temperatureInit,coolingFactor, endingTemperature):

    items=itemsIn.copy()
    for i in range(len(items)):
        items[i].append(solinit[i])
    # trier objets par utilité
    items_sorted=trier_objet_utility(items)
    # reordonner la solution
    solinitsorted=[]
    for i in range(len(items_sorted)):
        solinitsorted.append(items_sorted[i][2])
    # recupere le tabeau contenant le nombre max d'exemplaires
    # de chaque objet
    tab_max_nb,taille=
    get_max_number_item(items_sorted, capacity)
    tab_poids_new= get_tab_poid_new(items_sorted, tab_max_nb)
```

```
    tab_gain_new= get_tab_gain_new(items_sorted,tab_max_nb)
    # convertir la solution en une solution binaire
    solCurrent= ntobinary(solinitsorted, tab_max_nb)
   # evaluer la solution
    evalsol= eval_solution(solCurrent,tab_gain_new)
    # recuperer la temperature initaile
    temperature= temperatureInit
    # initialiser la meilleur solution
    bestSol= solCurrent.copy()
    bestEval= evalsol
    while (temperature > endingTemperature):

        for i in range(samplingSize):
            # passage a une nouvelle configuration
            solCurrent = getNextState(solCurrent,taille,
        tab_poids_new, tab_gain_new, capacity, temperature)
            # evaluer la nouvelle configuation
            evalCurrent=eval_solution(solCurrent, tab_gain_new);
            # si meilleur MAJ de la meilleur solution
            if evalCurrent > bestEval:
                bestSol= solCurrent.copy()
                bestEval=evalCurrent
        # MAJ la temperature
        temperature= cool(temperature, coolingFactor)

    objects=[]
    solution=[]
    #convertir la solution binaire trouver en une solution en n
    Nsol= binaryToNsolution(bestSol, tab_max_nb)
    for i,item in enumerate(Nsol):
        if item!=0:
            objects.append(items[i])
            solution.append(item)
    poids=0
    for i,obj in enumerate(objects):
        poids+=obj[0]*solution[i]
    # retourne la solution son gain et son poids
    return objects,solution, Nsol, bestEval,poids


```

3.3.2 MÉTA-HEURISTIQUE ÉVOLUTIVES

3.3.2.1 ALGORITHME GÉNÉTIQUE

- **Principe de l'algorithme**

1. **Génération de la population initiale :**elle consiste à générer aléatoirement N individus (Chromosomes). Chaque individu représente une solution réalisable du problème de sac à dos.

2. **La sélection :**elle consiste à sélectionner certaines solutions pour former une solution intermédiaire pour lui appliquer la mutation et le croisement. Il existe plusieurs méthodes de sélection : la sélection élitiste, ordonnancement, la sélection par roue de loterie, ...
3. **Le croisement :**le principe est de prendre deux individus aléatoirement parmi les individus élus, les couper en un ou plusieurs points puis interchanger les parties. Il est appliqué selon une probabilité Pc.
4. **La mutation :**Elle consiste à choisir un ou deux bits aléatoirement, puis les inverser. Cette opération s'applique selon un taux de mutation Pm.
5. **Mise à jour de la population :**Il s'agit de l'opérateur de remplacement. En effet, après le croisement et la mutation, la taille de la population augmente. Cet opérateur est appliqué pour garder une taille constante de la population. Pour cela, plusieurs stratégies existent.
6. **Critère d'arrêt :**Le processus s'arrête selon deux critères : atteindre le nombre maximum des itérations ou bien lorsque la population cesse d'évoluer (stagnation).

Il est à noter qu'après le croisement et la mutation des solutions non réalisables peuvent être générées pour cela nous avons besoin d'**un mécanisme de correction des chromosomes**. De plus, **une fonction d'évaluation** des solutions est nécessaire pour attribuer à chaque individu un coût pour déterminer s'il est apte ou non à se reproduire.

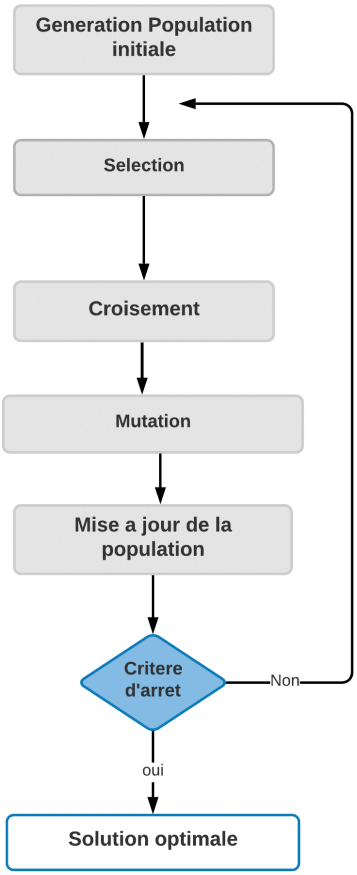


FIGURE 4 – Organigramme de l'algorithme Génétique

- **Implémentation**

– **Algorithme Principale**
Il suit le schéma précédent en utilisant la sélection élitiste. Il possède plusieurs paramètres et retourne la meilleure solution trouvée :

* La liste des objets à considérer (b : bénéfices, v : Volumes) et sa taille n.
* La capacité du Sac w.
* La taille de la population N et la taille de la population intermédiaire NI.
* La probabilité de croisement Pc et de mutation Pm.
* Le nombre maximum d'itération maxi.
* Un booléen pour prendre en compte la stagnation ou non avec un nombre maximum de même solution pour dire qu'il s'agit d'une stagnation.

```
def AG(n, w, b, v, N, NI, Pc, Pm, max_it, max_n, stagnation):
    #Gébération de la population
    population = generation_population(n, w, b, v, N)
    #Calcul de la liste des densités (utilités) de chaque objet
    #et l'ordonner d'une manière décroissante  par rapport
    #à la densité
    d = [(b[i]/v[i],i) for i in range(len(v))]
    d.sort(key=lambda x:x[0], reverse=True)
    t = 1
    # "mem" : une liste pour savegarder les "max_n" derniers gains
    # donnés par les "max_n" dernières itérations
    mem = []
    while not arreter(t, max_it, mem, max_n, stagnation):
        #Tant que les conditions d'arret ne sont pas satisfaites,
        #on génère un pool de chromosomes selon
        #la sélection élisite en prenant les "NI" meilleures
        #solutions de la population initiale
        pool = selection_elitiste(population, NI)
        #Générer un ensemble d'enfants à partir du pool en
        #utilisant le processus de croisement
        enfants1 = croisement(pool, Pc, n, w, b, v, d)
        #Générer un ensemble d'enfants à partir du pool en
        #utilisant le processus de mutation
        enfants2 = mutation(pool, Pm, n, w, b, v, d)
        #Mettre à jour la population pour ne garder que les "N"
        #meilleures solutions de la nouvelle population
        #après l'ajout des chromosomes ajoutés par le croisement
        #et la mutation
        population =
        maj_population(population, enfants1, enfants2, N)
        #Mettre à jour la liste "mem" dans le cas où on a
        #choisi d'arreter lors d'une stagnation du
        #résultat au bout de "max_n" itérations
        if stagnation:
            if len(mem)==max_n:
                mem.pop(0)
            mem.append(population[0].gain)
        t += 1
```

```
    return population[0].gain,
            population[0].poids,
            population[0].solution

```

– **Critère d'arrêt** Il teste le nombre d'itérations et retourne Vrai si on a atteint au nombre maximum d'itérations, sinon il teste si la population évolue encore.

```
def arreter(t, max_it, mem, max_n, stagnation):
    #Si nombre d'itérations max est atteint
    if t==max_it:
        return True
    #Sinon, si la stagnation est activée, on vérifie si les max_n
    #dernières valeurs sont toutes égales.
    #Si c'est le cas, on arrete sinon on continue l'exploration
    if stagnation:
        if len(mem)==max_n:
            val = mem[0]
            for elt in mem:
                if elt!=val:
                    return False
            return True
    return False

```

– **Procédure de sélection** Il s'agit de la méthode de la sélection élitiste. Cette procédure prend en paramètres la population courante et le nombre d'individus dans la population intermédiaire (NI). Elle retourne les NI meilleurs individus.

```
def selection_roue_loterie(population, NI):
    pop = list(population)
    pool = []
    for _ in range(min(len(pop), NI)):
        gain_t = 0
        probas = []
        for chromosome in pop:
            gain_t += chromosome.gain
            probas.append(gain_t)
        probas = np.array(probas)/gain_t
        rand = random.uniform(0,1)
        ind = recherche_dico(probas,rand)
        pool.append(pop[ind])
        pop.pop(ind)
    return pool

```

– **Procédure de croisement** Pour faciliter le croisement des individus sélectionnés par l'étape précédente, deux fonctions sont créées :

* Fonction croisement-1-point : cette fonction effectue le croisement entre deux parents (parent1, parent2) à un point et produit deux fils. Elle prend en paramètres aussi la liste des objets du sac, leur nombre et la capacité du sac. Ces derniers sont utilisés pendant la correction des chromosomes en cas de solution non réalisable.
* Fonction croisement : Elle prend en paramètre la population intermédiaire (pool), la probabilité de croisement Pc ainsi que la liste des objets et la capacité du sac. Parmi la population intermédiaire, elle choisit ceux qui vont subir un croisement puis applique la première fonction.

```
def croisement_1_point(parent1, parent2, n, w, b, v, d):
    #Choisir une position "k" aléatoire entre 1 et n-1
    #(n : nombre de types d'objets)
    k = random.randint(1,n-1)
    #Faire le croisement à un point en concaténant les "k"
    #premiers éléments du premier parent avec les (n-k) derniers
    #éléments du deuxième parent, et puis faire une éventuelle
    #correction de la solution trouvée pour la rendre réalisable
    enfant1 = corriger(parent1.solution[0:k]
        + parent2.solution[k::], n, w, b, v, d)
    #Faire la meme chose pour générer le deuxième enfant cette
    #fois-ci en concaténant les "k" premiers éléments du
    #deuxième parent avec les (n-k) derniers éléments du premier
    #parent (et le corriger bien sur)
    enfant2 = corriger(parent2.solution[0:k]
         + parent1.solution[k::], n, w, b, v, d)
    return enfant1, enfant2

def croisement(pool, Pc, n, w, b, v, d):
    selected = []
    #On sélectionne aléatoirement selon la probabilité "Pc"
    #un ensemble de chromosomes
    for chromosome in pool:
        r = random.uniform(0,1)
        if r<=Pc:
            selected.append(chromosome)
    enfants = []
    for i in range(0,len(selected)-1,2):
        #Pour chaque pair de chromosome, on fait un croisement à
        #un point et on ajoute les deux enfants générés
        #dans la liste des enfants
        enfant1, enfant2 = croisement_1_point(selected[i],
                                    selected[i+1], n, w, b, v, d)
        enfants.append(enfant1)
        enfants.append(enfant2)
    return enfants
```

– **Procédure de mutation** Elle prend en paramètre la population intermédiaire (pool), la probabilité de mutation Pm ainsi que la liste des objets, sa taille et la capacité du sac qui vont servir pour la correction des individus après mutation. Pour la mutation, elle est effectuée au niveau binaire en changeant la valeur d'un bit.

```
def mutation(pool, Pm, n, w, b, v, d):
    enfants = []
    for chromosome in pool:
        r = random.uniform(0,1)
        #Pour chaque chromosome, on génère un nombre aléatoire "r"
        #selon une loi uniforme sur [0,1]
        #Si r<=Pm alors l'individu est choisi pour subir une
        #mutation à fin de produire un nouveau chromosome
        if r<=Pm:
            liste = list(chromosome.solution)
            r = random.randint(0,n-1)
            #Choisir aléatoirement une position "r" sur la liste
            #des nombres d'exemplaires de chaque objet de
         #solution du chromosome
            #Convertir le nombre d'exemplaires de l'objet "r" en
         #binaire dans la variable "val_s"
            val_s = str(bin(liste[r]))
            rand = random.randint(2,len(val_s)-1)
            #Choisir une position aléatoirement sur la chaine
         #binaire "val_s", et basculer le bit correspondant
            if val_s[rand]=="0":
                val_s = val_s[0:rand] + '1' + val_s[rand+1:]
            else:
                val_s = val_s[0:rand] + '0' + val_s[rand+1:]
            #Après le basculement du bit, on remet le nouveau
         #nombre d'exemplaires dans la solution
            liste[r] = int(val_s,2)
            #Correction, éventuelle, de la solution trouvée pour
         #la rendre réalisable si elle ne l'est pas
            #Ajouter ce nouveau chromosome dans la liste
         #des enfants
            enfants.append(corriger(liste, n, w, b, v, d))
    return enfants
```

– **Procédure de correction des chromosomes** Procédure de correction des chromosomes : Cette procédure prend en paramètre la solution générée par mutation ou croisement sous forme d'une liste d'exemplaire de chaque objet.
On calcule d'abord le poids et le gain des objets de liste, tant que le gain dépasse la capacité du sac à dos, on enlève le maximum d'exemplaires de l'objet le moins important au sens densité pour transformer la solution en une solution réalisable.

```
def corriger(liste, n, w, b, v, d):
    poids = 0
    gain = 0
    #Calcul du poids et gain totaux des objets de la liste "liste"
    #en entrée
    for i in range(n):
        poids += liste[i] * v[i]
        gain += liste[i] * b[i]
```

```
    r = len(d)-1
    #Tant que le gain dépasse la capacité du sac à dos, on enlève
    #le maximum d'exemplaires
    #de l'objet qui a la plus petite densité pour transformer
    #la solution en une solution réalisable
    while poids>w:
        i = d[r][1]
        nb = int((poids-w)/v[i])
        if nb*v[i]!=poids-w:
          nb += 1
        if nb<=liste[i]:
            liste[i] -= nb
            poids -= nb * v[i]
            gain -= nb * b[i]
        else:
            poids -= liste[i] * v[i]
            gain -= liste[i] * b[i]
            liste[i] = 0
        r -= 1
    #Retourner un nouveau chromosome corrigé
    return chromosome(gain, poids, liste)

```

– **Evaluation des solutions** L'évaluation des solutions se fait en temps réel des manipulations grâce à la structure qui contient la solution. En effet, le chromosome contient la solution, son coût et son poids.

```
class chromosome:
def __init__(self, gain, poids, solution):
self.gain = gain
self.poids = poids
self.solution = solution

```

3.3.2.2 COLONIE DE FOURMIS

- **Principe de l'algorithme**
  L'algorithme utilise une approche par construction. Chaque fourmis est positionnée sur un nœud (objet). La fourmis ensuite construit la solution en choisissant le prochain objet à prendre parmi les objets d'une liste candidate.

  1. **Générer Solution initiale :** Elle se fait à travers une heuristique spécifique. et la considérer comme la meilleure solution au début pour qu' au pire des cas on aura cette solution comme la meilleure solution.
  2. **Générer des fourmis et construire les solutions :** Chaque fourmis construit une solution en partant initialement par une solution vide et en choisissant des objets parmi

une liste candidate. Le choix du prochain objet à prendre se fait selon la probabilité :

$$P_{i,j}^{k}(t) = \begin{cases} \frac{(\tau_{ij}(t))^{\alpha}(\eta_{ij})^{\beta}}{\Sigma_{I \in J_i^k}[\tau_{il(t)}]^{\alpha}[\eta_{il}]^{\beta}} \; si \; j \in J_i^k \\ 0 \; si \; j \notin J_i^k \end{cases} \tag{1}$$

τ : la quantité de phéromone posé sur l'objet
n : l'utilité de l'objet
alpha et beta sont deux paramètres contrôlant l'importance de l'utilité et de la quantité du pherom.

3. **Déposer le phéromone :** On utilise une stratégie élitiste telle que seuls les n meilleurs fourmis déposent du phéromone sur leur chemin. pour le dépôt de phéromone il se fait selon la formule :

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{1}{L^k(t)} \; si \; (i,j) \in T^k(t) \\ 0 \; sinon \end{cases} \tag{2}$$

   L : est le gain de la solution construite

4. **Mise à jour de la meilleure solution :** Parmi toutes les solutions construites par les fourmis, on teste si elles sont meilleures que la meilleure solution obtenue jusque là et on met à jour la solution.
5. **Evaporation de phéromone :** décrémenter le taux de phéromone pour diversifier le nombre de solution et choisir d'autres chemins non emprunté.
6. **Critère d'arrêt :** on choisit que le critère d'arrêt est le nombre d'itération spécifié.

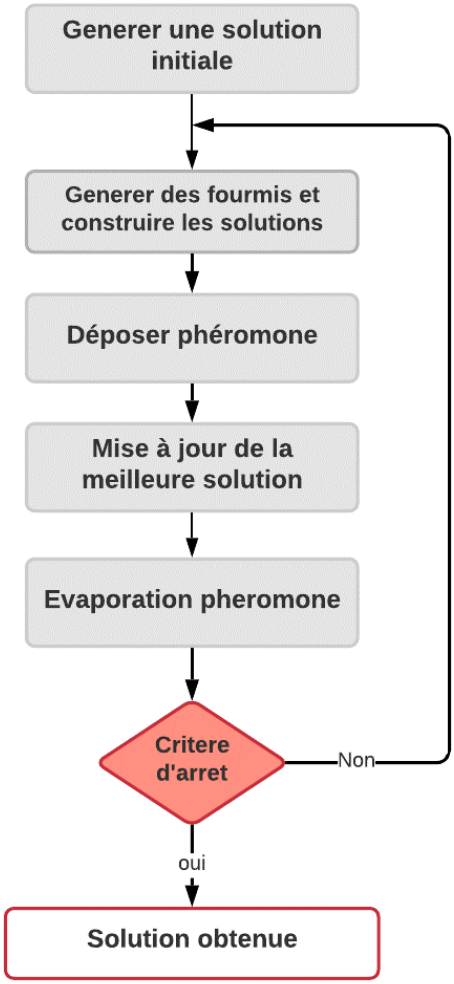


FIGURE 5 – Evolution du temps d'exécution du DP sur des instances de différentes tailles

- **Implementation**
  Pour l'implémentation de la solution, on a fait une implémentation orientée objet en créant une classe AntColony. Cette classe a comme attribut les éléments montré dans la figures ci dessus :

```
    def __init__(self,benifices,poids, utilites,n_objets, W,densitySol
        ,n_ants, n_best, n_iterations, decay, alpha=1, beta=1):
        """ benifices (1D numpy.array): Les benifices de chaque objet.
            poids (1D numpy.array): Les poids de chaque objet.
            poids (1D numpy.array): L'utilité d'un objet de chaque
         objet.
            n_objets (int): Nombre d'objets
            W (int): La capacité du sac
            densitySol (liste): Solution generé par heuristique
         spécifique
            n_ants (int): Nombre de fourmis par iterations
            n_best (int): Nombre de meilleures fourmis qui déposent
         le pheromone
            n_iteration (int): nombre d'iteration
            decay (float): 1-Taux d'evaporation de pheromone
```

```
            alpha (int or float): Exposant dans le pheromone, Alpha
         grand donne plus de poid au pheromone
            beta (int or float): Exposant sur l'utilité, Beta grand
         donne plus de poid a l'utilité
        """
        self.utilites  = utilites
        self.W=W
        self.n_objets=n_objets
        self.poids=poids
        self.benifices=benifices
        self.pheromone = np.ones(n_objets)
        for i,s in enumerate(densitySol): #ajouter du pheromone aux
                                  #objets generé par heuristique
          if s>0:
            self.pheromone[i]+=s*0.1
        self.all_inds = range(len(utilites))
        self.n_ants = n_ants
        self.n_best = n_best
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta

```

La méthode principale à exécuter pour résoudre le problème est la méthode run. Cette méthode implémente le schémas expliqué dans l'organigramme précédent.

```
def run(self,n_candidats,densitySol):
        """
        Args:
            n_candidats (int): Nombre de candidats pour construire
         les solutions
            densitySol (Gain:int,sol:liste,Poid:int): Solution generé
         par heuristique spécifique
        Example:
            best_sol = ant_colony.run(n_candidats,densitySol)
        """
        best_solution =
         (densitySol[1], densitySol[0],self.W-densitySol[2])
        best_solution_all_time =
         (densitySol[1], densitySol[0],self.W-densitySol[2])
        for i in range(self.n_iterations):
            #generer toutes les solutions par les fourmis
            all_solutions = self.gen_all_solutions(n_candidats)
            #mise a jours des pistes pheromones
            self.spread_pheronome(all_solutions, self.n_best,
                                  best_solution=best_solution)
            #Choisir meilleure solution dans l'iteration actuelle
            best_solution = max(all_solutions, key=lambda x: x[1])
```

```
            #print (best_solution)
            #Mettre a jour la meilleure solution globale
            if best_solution[1] > best_solution_all_time[1]:
                best_solution_all_time = best_solution

            #evaporation de pheromone
            self.pheromone= self.pheromone * self.decay
            self.pheromone[self.pheromone<1]=1
        return best_solution_all_time
```

La méthode spread_pheromone :

```
def spread_pheronome(self, solutions, n_best, best_solution):
      """
        Dépose le pheromone sur les n_best meilleures solutions

      """
      sorted_solution = sorted(solutions, key=lambda x: x[1]
                                                        ,reverse=True)
      for sol, gain,poid in sorted_solution[:n_best]:
            for i in sol:
              self.pheromone+= 1/gain

```

La methode gen_all_sollutions :

```
def gen_all_solutions(self,n_candidats):
        """
          Generer toutes les solutions par les fourmis

        """
        all_solutions = []
        for i in range(self.n_ants):
          #Positionner la fourmis sur un objets de départ aleatoirement
          n=rn.randint(0,self.n_objets-1)
          #generation de la solution par la fourmis en utilisant
        #n_candidats
          solution = self.gen_sol(n,n_candidats)
          #print("solution",solution[2])
          #ajouter la solution a la liste de toute les solutions
          all_solutions.append((solution[0],solution[1], solution[2]))
        return all_solutions


```

la méthode liste_candidate :

```
def listeCandidate(self,phero,visited,n_candidats):
      """
        retourne La liste des candidats pour une solution

      """
      pheromone=phero.copy()

      pheromone[list(visited)]= 0
       #rn.choices returns a list with the randomly selected element
       #from the list.
       #weights to affect a probability for each element
      c=rn.choices(self.all_inds,weights=[p for p in pheromone],
                k=n_candidats)
      c=list(set(c))#pour eleminer les doublons
      i=len(c)
      nb_candidats=len(c)
      return c ,pheromone

```

la methode gen_sol : construire une solution par une fourmis en partant par un objet initial et un nombre de candidats

```
  #generer solution c'est bon
    def gen_sol(self, start,n_candidats):
        sol = np.zeros(self.n_objets)
        poidrestant=self.W
        visited = set()#liste des objets visité
        #ajouter le premier objet
        r=rn.randint(1,poidrestant//poids[start])
        sol[start]=r
        poidrestant-=poids[start]*r
        gain=r*benifices[start]
        visited.add(start)#ajouter le debut a la liste civité
        #la liste candidates avec les pheromones mis a jours localement
        #(0 sur les visited)
        candidats,pheromones=
        self.listeCandidate(self.pheromone,visited,n_candidats)
        for i in candidats :
            #Choisir le prochain objets parmi les candidats ainsi que
         #le nombre
            move,nb =
        self.pick_move(pheromones, candidats,
                n_candidats,self.utilites, visited)
            toPop=candidats.index(move)
            candidats.pop(toPop)
            n_candidats-=1
            np.delete(pheromones,toPop)#rendre le pheromone à 0 pour
                                  #indiquer qu'il a été visité
            #Mise a jour poidRestant et gain de la solution
```

```
            poidrestant-=poids[move]*nb
            while poidrestant < 0:
              nb-=1
              poidrestant+=poids[move]
            sol[move]=nb
            gain+=nb*benifices[move]
            #ajouter l'objet a visited
            visited.add(move)
        return sol,gain,self.W-poidrestant

```

la methode pick_move : cette méthode retourne le prochain objet à prendre parmi la liste des candidats en se basant sur la formule déjà cité

```
def pick_move(self, pheromone,liste_cand,
                        n_candidats, utilite, visited):

        pheromone=pheromone.copy()[liste_cand]
      #generer le regle de déplacement sur les candidat
        numerateurs=(pheromone**self.alpha)*
                ((1.0 / (utilite[liste_cand]))**self.beta)
      #formule vu en cours
        P = (numerateurs / numerateurs.sum())

        #choisir l'objet suivant en utilisant les probabilité P
        move = np_choice(liste_cand, 1, p=P)[0]
        #nombre d'objet a prendre
        nb=self.W//self.poids[move]
        #nb=rn.randint(0,self.W//self.poids[move])

        return (move,nb)

```

4 TESTS ET RÉSULTATS

4.1 INSTANCES UTILISÉES

Pour les tests des méthodes exactes, nous avons utilisé un ensemble de 41 instances. La taille de ces dernières varie de 5 à 205 objets augmentant avec un pas de 5 objets pour chaque instance. Ces instances sont appelées **generated** et elles suivent le format suivant pour le nommage de leurs fichiers : **nombre_objets.txt**.

Pour aller plus loin et pousser les méthodes exactes à leur limite, nous avons utilisé le dossier **EDUK2000** contenant 6 instances de 2000 objets chacune.

Nous avons aussi eu recours aux instances Knapsack_problem_01KP qui comporte notamment des instances de taille 10, 100, 200, 500, 1000 et enfin 2000 objets.

Pour les tests des méthodes approchées en l'occurrence les heuristiques et les méta-heuristiques implémentées, nous avons utilisé les instances qui sont décrites dans le tableau suivant :

Instance	Nombre d'objets	Poids du sac à dos
`Instance 1 cap591952_-5000_facile.csv`	5000	591952
`Instance 2 cap7547243_-10000_facile.csv`	10000	7547243
`Instance 3 cap3897377_-5000_moy.csv`	5000	3897377
`Instance 4 cap52926330_-10000_moy.csv`	10000	52926330
`Instance 5 cap1596642_-5000_diff.csv`	5000	1596642
`Instance 6 cap1596642_-10000_diff.csv`	10000	1596642

TABLE 1 – Tableau descriptif des instances utilisées pour les heuristiques

PS : ces instances se trouvent dans le dossier Datasets.

4.2 CONFIGURATION DE LA MACHINE UTILISÉE (COLAB)

On a effectué les tests des méthodes implémentées sur une machine ,dans le cloud ,qui a la configuration matérielle suivante :

- Modèle CPU : Intel(R)Xeon(R)
- Fréquence de l'horloge CPU : 2.30GHz
- Coeurs CPU : 2 coeurs
- RAM disponible : 12 Go(peut arriver jusqu'à 26.75 Go)
- Espace disque : environ 25 Go

4.3 RÉSULTATS DES MÉTHODES EXACTES

Nous allons dans ce qui suit effectuer un certain nombre de tests sur plusieurs instances du problème UKP. Nous observerons à chaque test, le temps d'exécution obtenu avec la méthode du Branch and Bound et la méthode de programmation dynamique. Étant des méthodes exactes, ces derniers retourneront toujours la meilleure solution. Le seul critère d'évaluation de la qualité des deux méthodes est donc le temps d'exécution.

4.3.1 PROGRAMMATION DYNAMIQUE

Nous allons commencer par présenter les résultats obtenus sur un ensemble de 41 instances. La taille de ces dernières varie de 5 à 205 objets augmentant avec un pas de 5. Etant donné le nombre élevé d'instances testées nous avons préféré résumer les résultats obtenus à travers le graphe suivant :

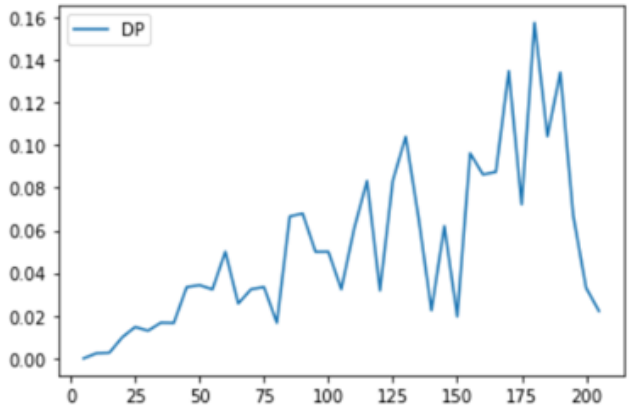


FIGURE 6 – Temps d'exécution de l'algorithme DP (en secondes) en fonction du nombre d'objets des instances

D'après l'allure générale de la courbe, le temps d'exécution augmente logiquement en augmentant la taille de l'instance. On peut voir que pour les instances de taille 140, 150, 200 et 205 le temps d'exécution est meilleur comparé à des instances de taille inférieure. Cela est justifié par le fait que la taille de l'instance n'est pas le seul paramètre influant. En effet, la complexité de l'instance joue aussi un rôle important et a une forte répercussion sur le temps d'exécution. Ceci est l'information principale à retenir de ce premier test. On peut cependant rajouter, que le temps d'exécution maximal a été enregistré pour l'instance de taille 180 et qu'il vaut environ 0.15727 secondes.

Nous avons par la suite effectué d'autres tests sur des instances de taille : 10, 100, 200, 500 et enfin 1000 objets. Voici un tableau résumant les temps d'exécution obtenus :

Taille de l'instance	**Temps d'exécution (secondes)**
10	0.00099
100	0.12987
200	0.18311
500	1.64930
1000	45.52428

TABLE 2 – Evolution du temps d'exécution du DP sur des instances de différentes tailles

Nous pouvons aussi visualiser les résultats grâce au graphe suivant :

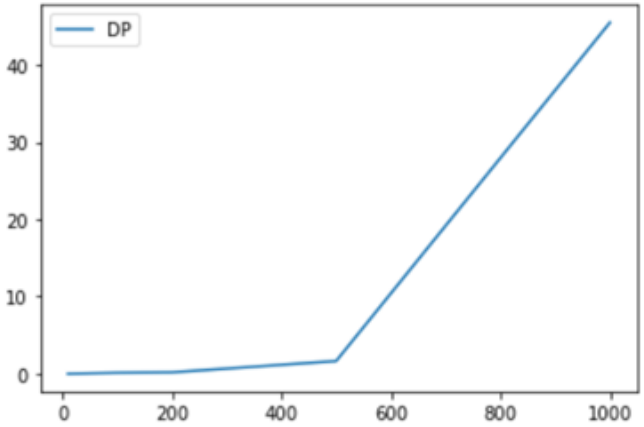


FIGURE 7 – Evolution du temps d'exécution du DP sur des instances de différentes tailles

On peut voir que sur les instances de taille 10, 100 et 200 le temps d'exécution est assez correct. Il commence à devenir significatif à partir de l'instance de taille 500. On peut de nouveau indiquer qu'il dépend de la complexité de l'instance puisque nous avons obtenu un temps d'environ 3 secondes pour une instance plus complexe de taille 500.

Le temps d'exécution explose pour une instance de taille 1000 puisqu'il est de 45 secondes. Nous avons même essayé une instance de taille 2000 qui nous a donné un temps de 188.86 secondes. On voit donc qu'à partir d'une certaine taille d'instance le temps d'exécution augmente très vite et de façon brutale.

4.3.2 BRANCH AND BOUND

Nous allons maintenant montrer les résultats donnés par la méthode exacte du Branch and Bound sur les mêmes instances utilisées avec la programmation dynamique.
Dans un premier temps, nous affichons la courbe d'évolution du temps d'exécution par rapport au nombre d'objets des 41 instances qui évolue avec un pas de 5 objets d'une instance à une autre. Le graphe obtenu est le suivant :

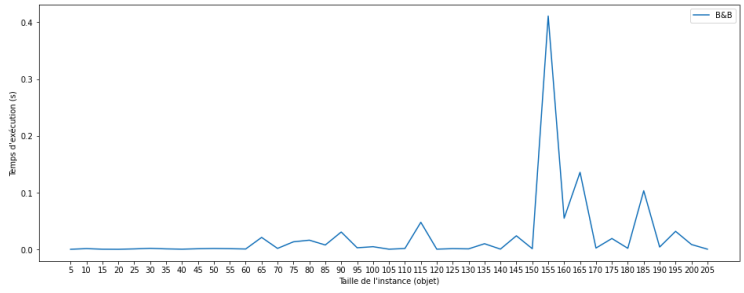


FIGURE 8 – Temps d'exécution du Branch and bound (en secondes) en fonction du nombre d'objets des instances

Avec des instances de petite taille (moins de 60 objets) les résultats sont très proches de quelques dizaines de millisecondes. A partir de 65 objets, le temps d'exécution commence à augmenter et à donner des résultats différents entre deux instances qui se suivent. On remarque que le temps maximal atteint, sauf pour l'instance avec 155 objets, est 0.12 secondes. L'instance de taille 155 est

un très bon exemple qui montre la dépendance de cette méthode aux valeurs des instances et pas qu'à leurs tailles. Avec l'instance 155, notre méthode n'a pas pu élaguer trop tôt comme avec les autres instances.

Nous avons voulu, par la suite, pousser les limites de la méthode Branch and Bound en l'essayant sur des instances de grandes tailles. Le tableau suivant résume les résultats obtenus :

Taille de l'instance	**Temps d'exécution (secondes)**
200	0.0007
500	0.0025
1000	0.015
2000	0.037
5000(facile)	0.0284
5000(difficile)	31.4

TABLE 3 – Evolution du temps d'exécution du B&B sur des grandes instances

A partir d'un certain nombre d'objets, il devient trop compliqué pour les méthodes exactes de donner le résultat en un temps raisonnable ce qui rappelle la particularité de ces méthodes quand elles sont appliquées à des problèmes NP-difficiles tels qu'UKP.

Leur complexité est exponentielle, ce qui fait qu'elles atteignent très vite leur limite et deviennent inutilisables. Ce constat nous pousse à renoncer à la solution exacte du problème au profit d'un temps d'exécution réduit à travers l'utilisation des heuristiques et les méta-heuristiques.

4.4 ETUDE COMPARATIVE

4.4.1 LES MÉTHODES EXACTES

Il peut être intéressant de comparer les deux méthodes exactes implémentées. Le Branch & Bound et la programmation dynamique sont des méthodes suivant des principes différents mais dont le point commun est de toujours donner la meilleure solution. Donc ici seule la comparaison en termes de temps d'exécution est viable.

Nous avons superposé les résultats obtenus pour les deux méthodes sur différentes instances dont la taille varie de 5 à 205 objets augmentant avec un pas de 5. Le résultat est montré dans la figure suivante :

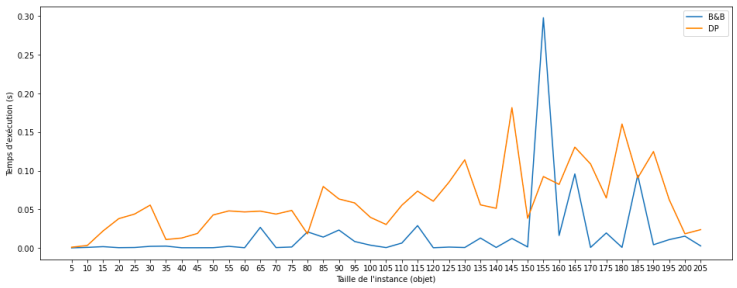


FIGURE 9 – Comparaison du temps d'exécution (en secondes) du B&B et DP

Selon le graphe précédent et selon l'analyse du temps d'exécution obtenu pour chaque méthode nous pouvons tirer les conclusions suivantes :

- Pour la majorité des instances, le Branch & Bound donne de meilleurs résultats par rapport à la programmation dynamique. Cela s'explique par le fait que le Branch & Bound utilise le principe de l'élagage qui permet de réduire considérablement l'espace de recherche et par conséquent d'améliorer la performance.

- Pour des instances comme 155, la programmation dynamique offre une meilleure performance. Cela est relatif à la complexité de cette instance. En effet, le Branch & Bound fait appel à une fonction d'évaluation des nœuds de l'arbre de recherche. Cette dernière donne des estimations grossières avec des instances un peu compliquées, ce qui impacte le principe d'élagage qui permet cependant au Branch & Bound de prendre le devant par rapport à la programmation dynamique dans la majorité des cas.

La performance des deux méthodes varie donc beaucoup en fonction de l'instance utilisée. Il serait donc impossible de dire quelle méthode est la plus performante. On peut en revanche souligner que ces deux méthodes exactes atteignent leur limite pour des instances de grande taille ou de complexité élevée. Pour le Branch & Bound, la complexité est l'élément ralentisseur de cet algorithme alors que l'ennemi principal de la programmation dynamique est la taille des instances (il est exponentielle par rapport au nombre d'objets) voilà pourquoi on renonce à leur utilisation et on se contente des solutions données par les heuristiques.

4.4.2 Les méthodes approchées

Abréviations utilisées : Sol OPT : solution exacte.
DOG : Density Ordered Greedy.
WOG desc : Weighted Ordered Greedy avec ordre décroissant.
WOG asc : Weighted Ordered Greedy avec ordre croissant.
Arrondi : Heuristique avec arrondi.

Dans un premier temps, nous allons comparer les quatre heuristiques spécifiques implémentées à savoir DOG, WOG desc, WOG asc et Arrondi. Étant donné que ces méthodes sont très rapides, on va nous intéresser au début au gain trouvé par chaque algorithme. La figure suivante montre le gain obtenu par chaque méthode pour chacune des 41 instances (la taille de ces instances varie entre 5 et 205 objets) :

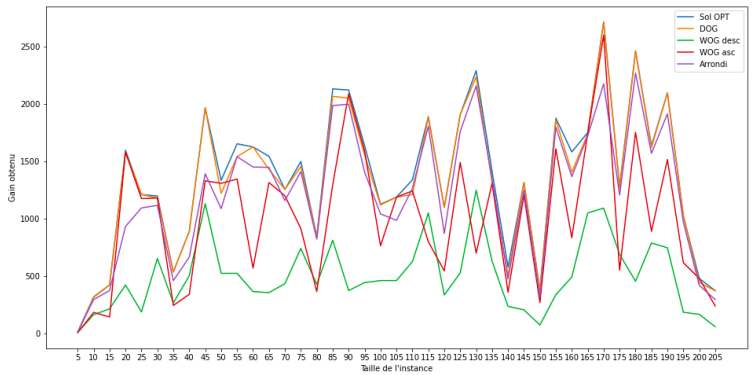


FIGURE 10 – Comparaison des gains obtenus par les heuristiques spécifiques implémentées

En analysant les différentes courbes, on remarque clairement que la méthode WOG avec ordre décroissant des poids (courbe verte) est très mauvaise par rapport aux autres alors que la même méthode avec un ordre croissant (courbe rouge) s'en sort beaucoup mieux et c'est le contraire de ce qu'on a observé avec les instances utilisées dans la partie tests empiriques. Les méthodes DOG et Arrondi donnent des résultats très proches avec un petit avantage pour la densité mais elles

restent de très bonnes méthodes parce qu'elles sont restées stables sur toutes les instances et elles suivent toujours l'allure de la courbe du gain exacte.

La méthode DOG, à travers la notion de densité, essaye de trouver un compromis entre le volume et le bénéfice d'un objet alors que la méthode WOG ne prend pas en considération les gains des objets c'est pour cela que cette méthode dépend fortement des instances, et pour choisir entre un ordre croissant ou décroissant des poids on devrait d'abord voir la relation entre le gain et le poids de chaque objet. Si la relation est linéaire alors l'utilisation du WOG avec un ordre convenable donnera de bonnes performances, sinon, si ce n'est pas linéaire alors il est préférable d'utiliser DOG parce qu'elle est plus flexible.

Il est clair que le temps n'est pas un souci pour les méthodes heuristiques mais nous avons quand même prévu une étude comparative du temps d'exécution. La figure suivante montre les temps d'exécution en secondes de chaque algorithme pour chacune des 41 instances :

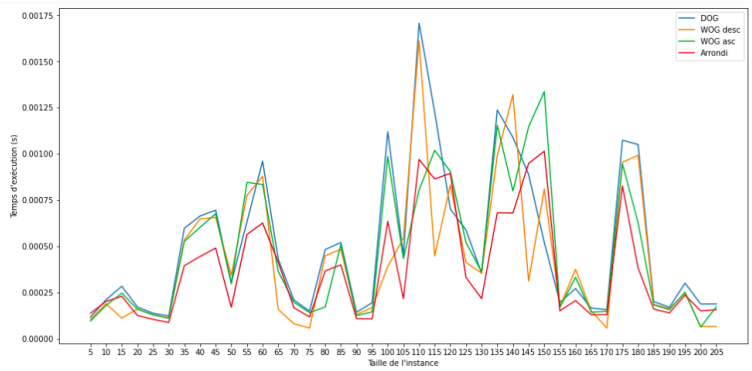


FIGURE 11 – Comparaison des temps d'exécution des heuristiques spécifiques implémentées

Puisque les deux algorithmes DOG et WOG (desc et asc) ont la même complexité n*log(n) (tri d'une liste et la parcourir par la suite) alors il est logique qu'ils soient très proches. Pour l'heuristique par arrondissement, on remarque qu'elle suit la même allure que les autres mais en faisant moins de temps car elle prend les objets avec utilités maximales qui respectent la contrainte de capacité du sac en essayant d'ajouter des exemplaires de ces objets.elle ne parcourt pas toute la table des utilités des objets (la contrainte d'arrêt de cet heuristique est la stagnation de la capacité calculée à partir d'objets choisis avec leurs nombres d'exemplaires). Dans cette partie, nous allons tester les méthodes approchées implémentées (heuristiques). Puisque ces méthodes ne sont pas gourmandes en termes de temps d'exécution puisqu'elles sont conçues d'une manière pour favoriser l'obtention des résultats approchés en un temps très réduit en quelques secondes voire des millisecondes même avec des instances trop grandes et difficiles.

Nous allons maintenant tester nos heuristiques, heuristiques Density Ordered Greedy, Weighted Ordered Greedy (avec ordre croissant et décroissant), heuristique Arrondi et l'heuristique par amélioration sur les 6 instances de grandes tailles citées dans la partie instances. Les résultats sont résumés sur le tableau ci-dessous :

Instance	Density Ordered Greedy	Weighted Ordered Greedy (Ordre décroissant)	Weighted Ordered Greedy (Ordre croissant)	Heuristique Arrondi	Heuristique par amélioration
Instance 1	590295 (5.86 ms)	585156 (6.31 ms)	590295 (5.86 ms)	592006 (7.06 ms)	592185 (16.55 s)
Instance 2	7543770 (12.2 ms)	7541508 (12.5 ms)	7543770 (11.9 ms)	7548268 (13.5 s)	7017970 (9 min 38s)
Instance 3	202816936 (5.86 ms)	202816936 (6.92 ms)	3914475 (4.97 ms)	201870786 (8.42 ms)	65310332 (30 s)
Instance 4	5337839522 (12.4 ms)	5337839522 (12.5 ms)	53105220 (12.6 ms)	5280383469 (13.9 ms)	470327213 (2 min 39s)
Instance 5	1994693 (11.1 ms)	2005645 (6.87 ms)	1613654 (7.1 ms)	1281496 (5.68 ms)	1962823 (1.32 s)
Instance 6	1994860 (13 ms)	2005701 (11.9 ms)	1613654 (12.6 ms)	1281496 (14.5 ms)	1973899 (3.47 s)

TABLE 4 – Tableau comparatifs des résultats d'exécution des quatre heuristiques implémentées

A partir des résultats ci-dessus, il en ressort que pour l'ensemble des instances utilisées, la Density Ordered Greedy trouve toujours un bon compromis entre le temps d'exécution et le gain trouvé. En examinant les résultats de plus près, on remarque que :

- L'heuristique arrondie est plus adaptée pour les deux premières instances (instances faciles).
- La Density Ordered Greedy et Weighted Ordered Greedy avec ordre décroissant donnent de meilleurs résultats pour les instances 3 et 4 (instances moyennes).
- L'algorithme Weighted Ordered Greedy (ordre décroissant) est le meilleur pour les instances difficiles (instances 5 et 6).
- L'heuristique par amélioration est plus adapté aux deux dernières instances (instances difficile) prend moins de temps en la comparant avec les instances faciles et moyennes et les gains trouvés sont meilleur que L'heuristique arrondi et Weighted Ordered Greedy croissant.

Pour la méthode Weighted Ordered Greedy avec ordre croissant des poids des objets les résultats ne sont globalement pas bons, ce qui est justifiable par le fait que la solution est constituée par des objets de petite taille et en générale leurs bénéfices correspondants sont très petits aussi.Et pour approcher de mieux en mieux les solutions exactes, d'autres méthodes ont été implémentées dans le principe est de commencer par des résultats assez bons donnés par des heuristiques (Density Ordered Greedy généralement) et essayer de construire d'autres solutions meilleures.

4.4.2.1 L'HEURISTIQUE PAR AMÉLIORATION

Cette méthode commence à partir d'une solution initiale générée aléatoirement et le nombre d'itérations est à 200.

Dans un premier temps, nous allons comparer les résultats de cette heuristique avec l'ensemble des méthodes déjà vues. On va nous intéresser au gain trouvé par chaque algorithme. La

figure suivante montre le gain obtenu par chaque méthode pour chacune des 41 instances (la taille de ces instances varie entre 5 et 205 objets) :

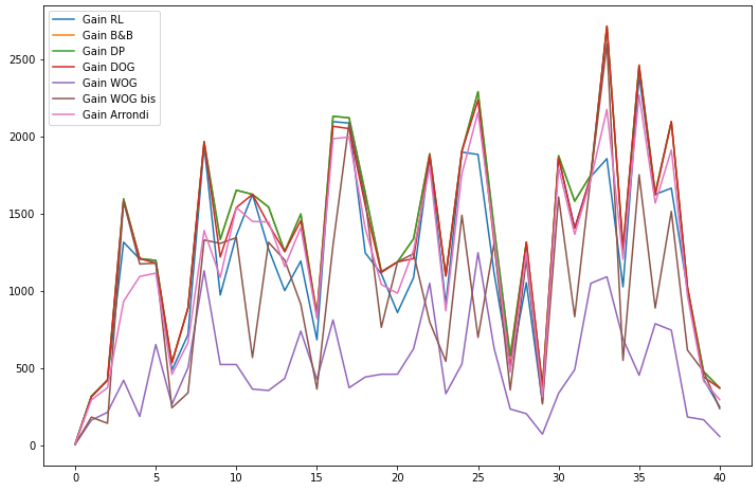


FIGURE 12 – Comparaison gain de l'heuristique par amélioration avec les autres méthodes

Selon le graphe précédent et selon l'analyse du gain obtenu pour chaque instance nous pouvons tirer les conclusions suivantes :

- Pour la majorité des instances, la recherche locale donne des résultats assez proches de Density ordered greedy, meilleur que weighted ordered greedy et Arrondi.
- Pour des instances comme entre 15 et 20 cette méthode donne des résultats meilleurs que density ordered, assez proches des résultats de la programmation dynamique.

La performance de cette méthode varie beaucoup en fonction de l'instance utilisée et c'est dû à l'aspect aléatoire dans la solution initiale ainsi que le voisinage qui sont générées aléatoirement, la complexité de l'instance et sa taille.

4.4.3 MÉTA-HEURISTIQUE ITÉRATIVE

Dans un premier temps, nous allons comparer les résultats du recuit simulé avec l'ensemble des méthodes déjà vues. On va nous intéresser au gain trouvé par chaque algorithme. La figure suivante montre le gain obtenu par chaque méthode pour chacune des 41 instances (la taille de ces instances varie entre 5 et 205 objets) :

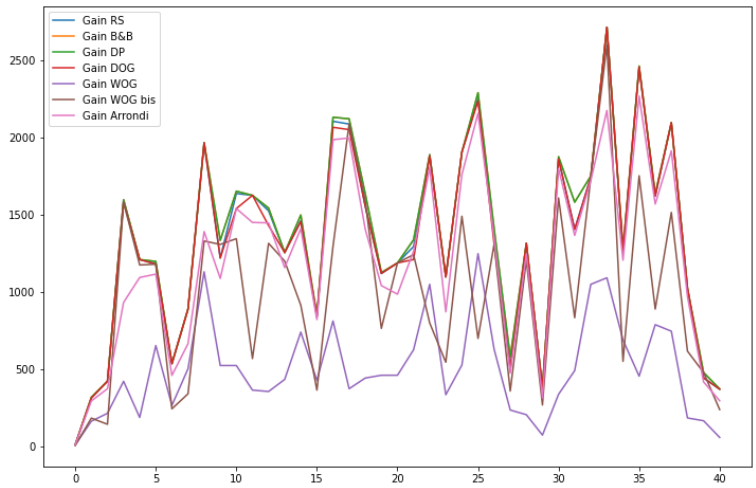


FIGURE 13 – Comparaison gain du recuit simulé avec les autres méthodes

Selon le graphe précédent et selon l'analyse du gain obtenu pour chaque instance nous pouvons tirer les conclusions suivantes :

- En comparant le gain obtenu par la méthode du recuit simulé avec l'heuristique Density ordered greedy pour la plupart des instances le résultat est le même mais pour quelques instances on remarque que la méthode du recuit simulé arrive à améliorer la solution et se rapproche de la solution exacte trouvé par la programmation dynamique.

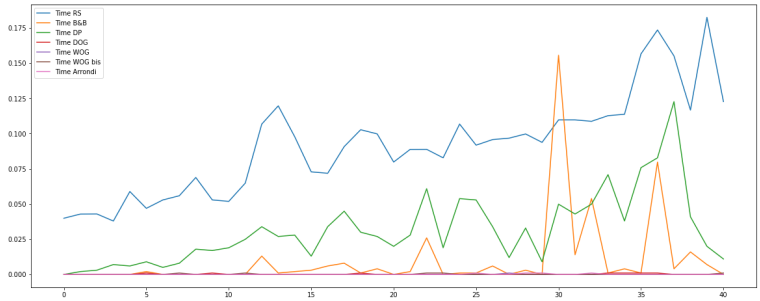


FIGURE 14 – Comparaison du temps d'exécution du recuit simulé avec les autres méthode

Selon le graphe précédent et selon l'analyse des temps, pour chaque instance nous pouvons tirer les conclusions suivantes :

- Le recuit simulé prend plus de temps que la plupart des méthodes.
- Pour l'instance 30 le recuit simulé prend moins de temps que Branch & Bound.
- Plus les instances deviennent grandes et complexes plus le recuit simulé performe bien en termes de temps.

Nous allons dans ce qui suit effectuer un certain nombre de tests sur plusieurs instances du problème UKP. Nous observerons à chaque test, le gain et le temps d'exécution obtenu par les méta-heuristiques., ces derniers retourneront des solutions approchées. Donc le critère d'évaluation de la qualité des méthodes est le gain obtenu.

Les résultats d'exécution des méta-heuristiques de voisinage et évolutionnaires sur les 6 instances citées au-dessus sont résumés sur le tableau ci-dessous :

Instance	Métaheuristique de voisinage Recuit simulé(Solution initiale DOG)	Density Ordered Greedy	Métaheuristique de voisinage Recuit simulé (Solution initiale aléatoire)
Instance 1	592246 (53 s)	590295 (5.86 ms)	592177 (59 s)
Instance 2	7551012 (22 min 59s)	7543770 (12.2 ms)	6704149 (21 min 6s)
Instance 3	202816936 (2 min 6s)	202816936 (5.86 ms)	99277957 (2 min 6s)
Instance 4	5337839522 (9 min 3s)	5337839522 (12.4 ms)	2438802564 (10 min)
Instance 5	2001168 (6 s)	1994693 (11.1 ms)	1994280 (7 s)
Instance 6	2004743 (15 s)	1994860 (13 ms)	1945018 (14 s)

TABLE 5 – Tableau comparatifs des résultats d'exécution des deux variantes implémentées du recuit simulé avec la méthode DOG

- La méta-heuristique du recuit simulé (solution initiale DOG) utilise le résultat de l'heuristique Density ordered greedy comme solution initiale, une température initiale de 1000, le facteur de refroidissement est à 0.9, la température finale est de 5 et le nombre de paliers est de 5.
- La méta-heuristique du recuit simulé (solution initiale aléatoire) utilise une solution générée aléatoirement comme solution initiale, une température initiale de 1000, le facteur de refroidissement est à 0.9, la température finale est de 5 et le nombre de paliers est de 5.

A partir d'un certain nombre d'objets et pour des instances complexe et grande, il devient trop facile pour la méthode du recuit de donner de meilleurs résultats en un temps raisonnable quelques secondes.

- La méta-heuristique du recuit simulé est plus adaptée aux deux dernières instances (instances difficiles).
- Pour les instances difficiles la méthode du recuit simulé trouve un bon compromis entre le temps d'exécution et le gain trouvé.
- Pour ces deux dernières instances la méthode du recuit simulé a donné de meilleurs résultats que la méthode Density ordered greedy.
- La méthode du recuit simulé qui part d'une solution initiale trouvé par l'heuristiques Density oredered Grey donne toujours des résultats meilleurs que celle qui démarre d'une solution initiale aléatoire.

4.4.4 MÉTA-HEURISTIQUE ÉVOLUTIVES

Dans ce qui suit, on effectue une étude comparative entre les résultats données par l'algorithme génétique, l'algorithme par colonie de fourmis et les solutions exactes.

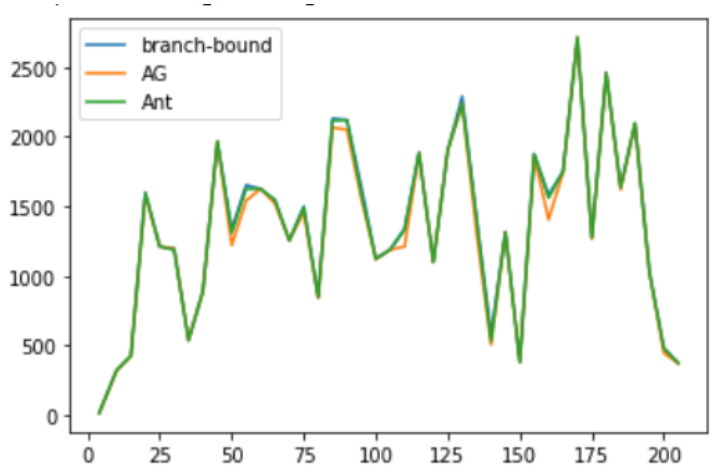


FIGURE 15 – Comparaison des résultats entre AG, ACO et Branch and Bound

Observation :

On remarque que dans les instances prises (41 instances) les deux algorithmes arrivent à donner de très bons résultats. En effet dans la plupart des cas les deux algorithmes arrivent à trouver les solutions exactes surtout en ce qui concerne l'algorithme par colonie de fourmis qu'on voit superposé sur la solution exacte(en vert).
Dans ce qui suit on fixe les paramètres des algorithmes en se basant sur les résultats précédent comme suit : Pour AG : max iteration= 500, taille mémoire = 10, taille population = 200,taille de la population intermédiaire = 100, Pc = 0.8,Pm = 0.1,stagnation = True Pour l'algorithme par colonie de fourmis : nombre de fourmis = 100, nombre de meilleures fourmis qui déposeront la phéromone = 10, nombre d'itérations= 10, taux d'évaporation de la phéromone= 0.8,alpha = 1, beta = 1

	AG	**Colonie de fourmis**
instance 1	592161 (3.03s)	592189 (4.33s)
instance 2	7549618 (7.18s)	7551009 (12.4s)
instance 3	202816936 (4.34s)	202816936 (4.66s)
instance 4	5337839522 (7.88s)	5337839522 (9.16s
instance 5	1994693 (4.15s))	2004598 (4.04)
instance 6	2004657 (19s)	2004302 (8.03s)

TABLE 6 – Evolution du temps d'exécution du B&B sur des grandes instances

Conclusion :

On peut dire qu' aucun algorithme ne se démarque vraiment en termes de qualité de solution. Ce qu'un algorithme perd en termes de gain, il le récupère en temps d'exécution et vice-versa.

4.5 ETUDE EMPIRIQUE

4.5.1 L'HEURISTIQUE PAR AMÉLIORATION

- **Le nombre des itérations** Nous avons superposé les résultats obtenus pour cette méthode avec différents nombres d'itérations, sur différentes instances dont la taille varie de 5 à 205 objets augmentant avec un pas de 5. Le résultat est montré dans la figure suivante :

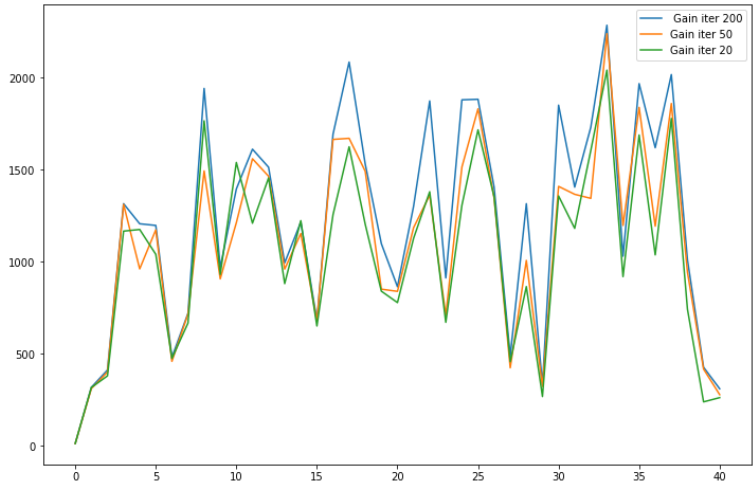


FIGURE 16 – Etude de l'influence du paramètre "nombre d'itérations" sur le gain donné par l'heuristique par amélioration

Selon le graphe précédent et selon l'analyse du gain obtenu pour chaque instance nous pouvons tirer les conclusions suivantes :

- Le nombre d'itérations de 200 donne de meilleurs résultats pour toutes les instances.
- En augmentant le nombre d'itérations la méthode explore plus de voisin et améliore sa fonction gain.

4.5.2 LE RECUIT SIMULÉ

Cette méthode donne des résultats variables sur les différentes instances qui dépendent de la solution initiale (qu'elle soit générée aléatoirement ou à partir des résultats d'une heuristique constructive) la méthode du voisinage (l'aspect aléatoire dedans) les températures initiales affectée par le type de l'instance et sa taille.

Pour certaines instances et on en prenant une solution initiale précise l'augmentation de la température initiale n'améliore pas les résultats (instances petites et faciles), pour d'autres instances (grandes difficiles il faut des températures assez élevée).

Pour certaines instances et on en prenant une solution initiale précise l'augmentation de la température initiale n'améliore pas les résultats (instances petites et faciles), pour d'autres instances (grandes difficiles il faut des températures assez élevée).

- **Le facteur de refroidissement**
  Nous avons superposé les résultats obtenus pour cette méthode avec différentes valeurs du facteur de refroidissement, sur différentes instances dont la taille varie de 5 à 205 objets augmentant avec un pas de 5. Le résultat est montré dans la figure suivante :

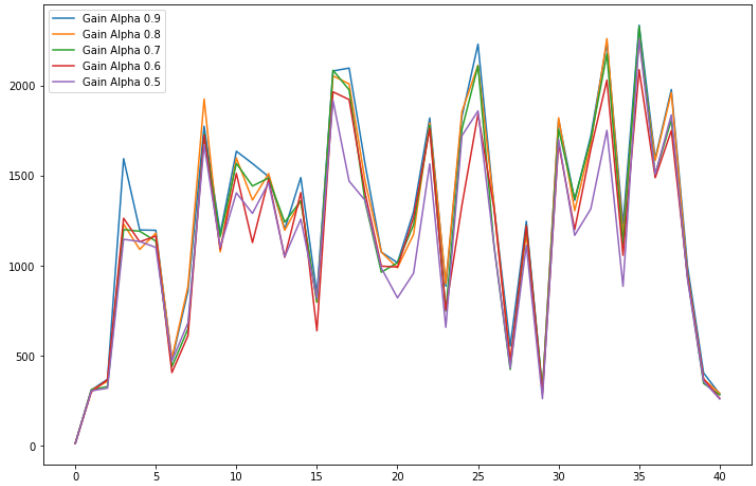


FIGURE 17 – Etude de l'influence du paramètre "facteur de refroidissement" sur le gain du recuit simulé

En faisant des tests sur un ensemble d'instances nous remarquons que le facteur de refroidissement 0.9 donne toujours de meilleurs résultats.

- **Le nombre de paliers**
  Nous avons superposé les résultats obtenus pour cette méthode avec différentes valeurs du nombre des paliers, sur différentes instances dont la taille varie de 5 à 205 objets augmentant avec un pas de 5. Le résultat est montré dans la figure suivante :

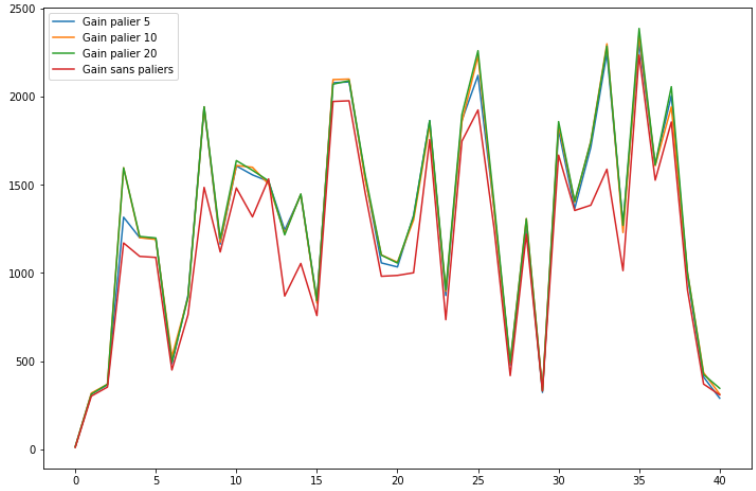


FIGURE 18 – Etude de l'influence du paramètre "nombre de paliers" sur le gain du recuit simulé

  – Le recuit simulé avec un nombre de paliers 20 donne toujours de meilleurs résultats, ces résultats sont assez proches des résultats avec le nombre de paliers 10.
  – Le recuit simulé sans paliers donne les résultats les plus faibles.

  Pour chaque température explorer et exploiter plus de voisins élevé la possibilité de tomber sur des solutions meilleures.

- **La température initiale**
  Nous avons superposé les résultats obtenus pour cette méthode avec différentes valeurs de la température initiale, sur différentes instances dont la taille varie de 5 à 205 objets augmentant avec un pas de 5. Le résultat est montré dans la figure suivante :

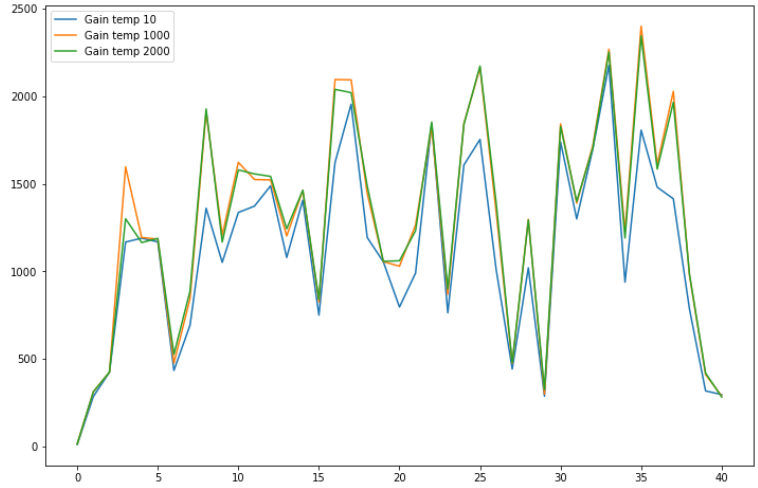


FIGURE 19 – Etude de l'influence du paramètre "température initiale" sur le gain du recuit simulé

  - La température initiale 1000 donne les meilleurs résultats car elle induit à une proba d'acceptation assez élevé et amené la méthode à élargir l'espace des solutions visité donc induit à explorer des solutions meilleures .

- **La solution initiale**
  Nous avons superposé les résultats obtenus pour cette méthode en se basant sur une solution initiale aléatoire ou en se basant sur une solution Density oredered greedy, sur différentes instances dont la taille varie de 5 à 205 objets augmentant avec un pas de 5. Le résultat est montré dans la figure suivante

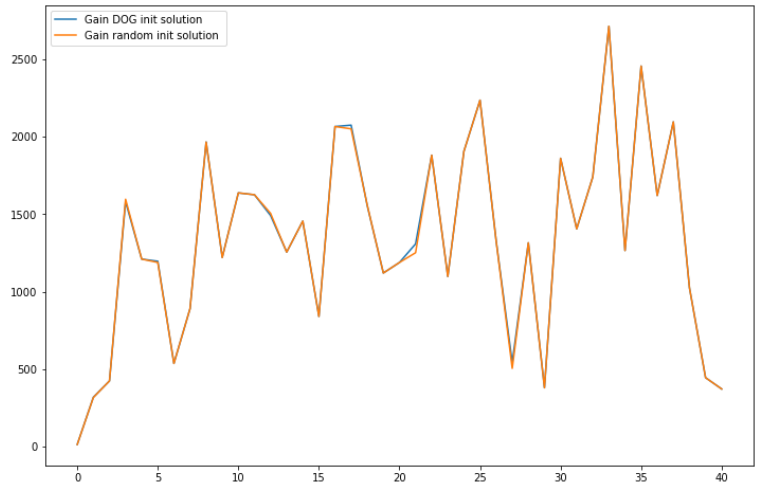


FIGURE 20 – Etude de l'influence du choix de la "solution initiale" sur le gain du recuit simulé

  - Les résultats sont assez proches mais la solution basée sur Density oredred greedy est toujours meilleur que la solution aléatoire.

- **Comparaison recuit simulé et recherche locale**
  Il peut être intéressant de comparer les deux méthodes recuit simulé et la recherche locale. Ce sont des méthodes suivant des principes différents le recuit simulé autorise la diversification (explorer de nouvelles solutions voisines même si ces dernières donnent un gain inferieur a la solution courante qui peut conduire à explorer des solutions qui peuvent par la suite donner un gain élevé).

Et part d'une solution déjà trouvé par l'heuristique Density ordre greedy par contre la recherche réduit l'espace des solutions explorées et part d'une solution aléatoire.

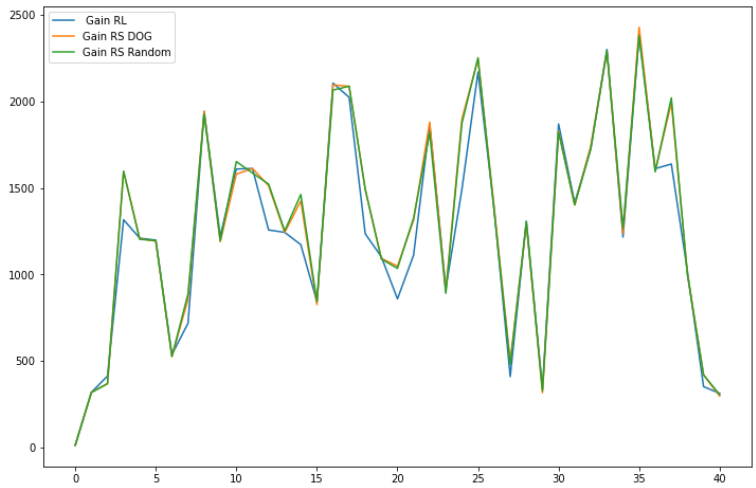


FIGURE 21 – Comparaison des gains donnés par "recherche locale" et "recuit simulé"

- Le recuit simulé donne toujours de meilleurs résultats ceci est dû à la diversification au niveau du recuit simulé (l'espace des solutions exploité par le recuit simulé est plus grand que celui de la recherche locale) ce qui induit à explorer des solutions qui peuvent donner des gains meilleurs.

- **Comparaison différentes combinaisons de paramètres**
  La combinaison température initiale de 1000 le nombre de paliers de 10 et le facteur de refroidissement de 0.9 donne toujours de meilleurs résultats

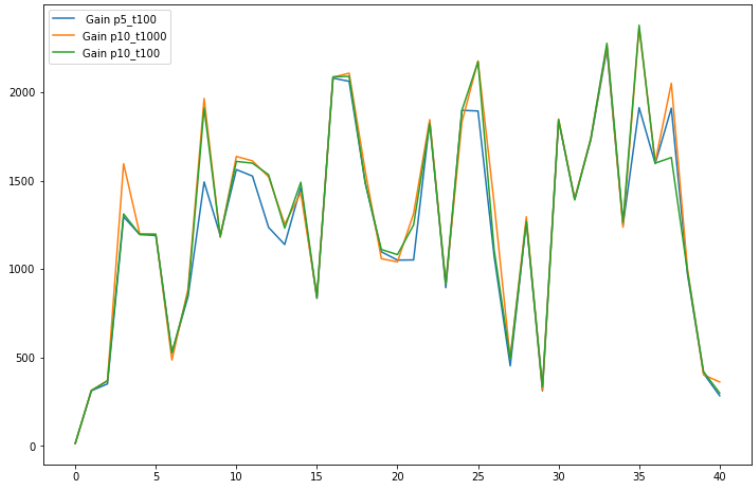
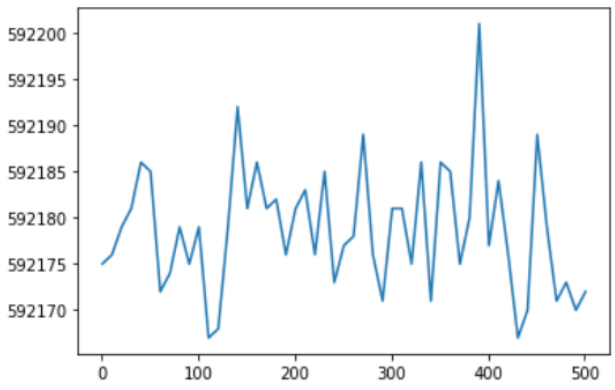


FIGURE 22 – Comparaison de l'influence de différentes combinaisons de paramètres sur le gain du recuit simulé

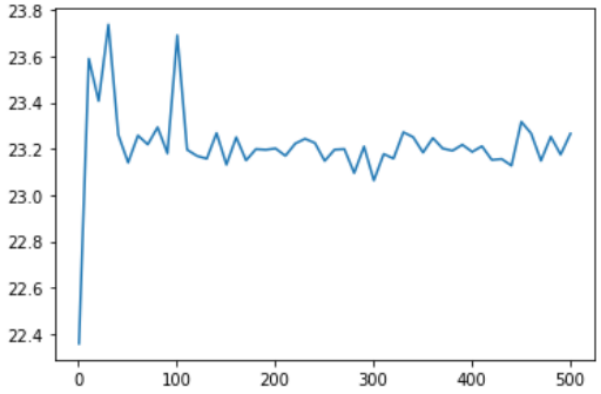
4.5.3 ALGORITHME GÉNÉTIQUE

Pour effectuer une étude empirique sur l'AG, on s'intéresse à l'effet de la variation des paramètres sur la qualité des solutions et le temps d'exécution :

- **Temps d'exécution et gain en fonction du nombre d'itérations :**



(a) Gain en fonction de nombre d'itération

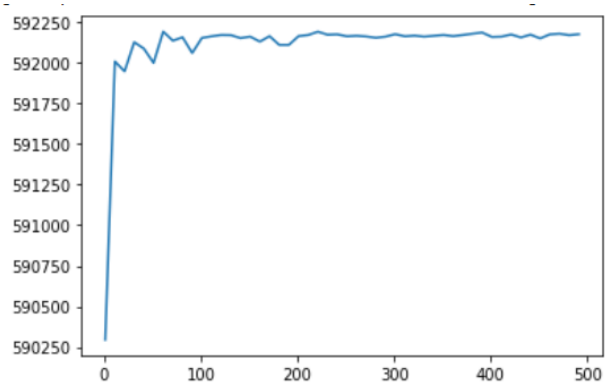


(b) Temps d'exécution en fonction de nombre d'itération

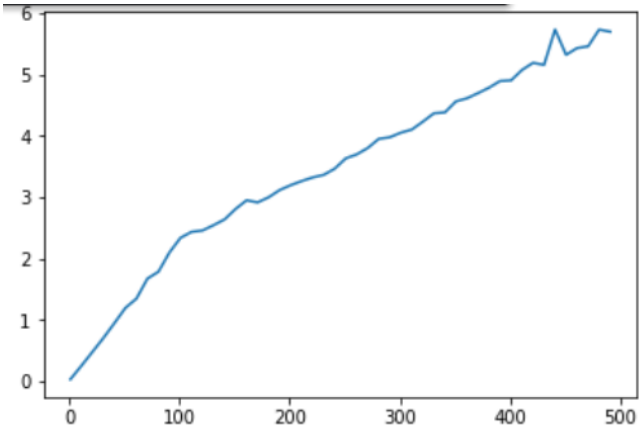
FIGURE 23 – Etude de l'influence de nombre d'itération sur le temps d'exécution et sur la qualité de la solution

Analyse et interprétation : On remarque que cet algorithme ne garantit pas une convergence vers la solution optimale en augmentant le nombre d'itération. Ceci peut s'expliquer par le fait que l'algorithme a peu de fondement théorique de plus son fonctionnement aléatoire pendant les mutations et le croisement. Quant au temps d'exécution, il est plus ou moins constant et cela peut s'expliquer par le fait que l'algorithme stagne à partir d'un certain nombre d'itération.

- **Temps d'exécution et gain en fonction de la taille de la population globale :**



(a) Gain en fonction de la taille de la population



(b) Temps d'exécution en fonction de la taille de la population

FIGURE 24 – Etude de l'influence de la taille de la population sur le temps d'exécution et sur la qualité de la solution

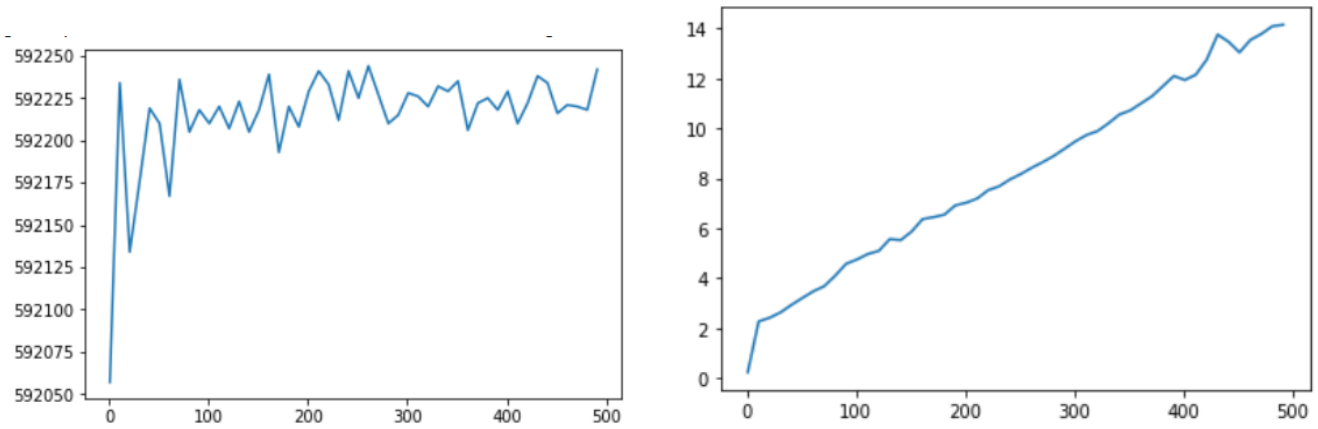
Analyse et interprétation : L'utilisation de grande population n'améliore pas les résultats mais on remarque que a partir d'un certain seuil (à partir de 100 individus) le gain ne s'améliore pas. Alors que, le temps d'exécution augmente linéairement en augmentant la taille de la population ce qui montre qu'un nombre important d'itérations ne donne pas forcément un meilleur gain parce que généralement au bout de quelques centaines de boucles l'algorithme converge.

- **Variation de la probabilité de mutation et probabilité de croisement :** La variation de la probabilité de mutation et de la probabilité de croisement participe principalement à la di-

versification de la population. Plus ces taux sont élevés plus la population subit des changements donc plus le temps d'exécution augmente. Des études expérimentales considèrent qu'il faut prendre un taux de mutation faible car elle est considérée comme un mécanisme d'adaptation secondaire pour les algorithmes génétiques.

4.5.4 COLONIE DE FOURMIS

- **Gain et temps d'exécution en fonction de nombre de fourmis :**

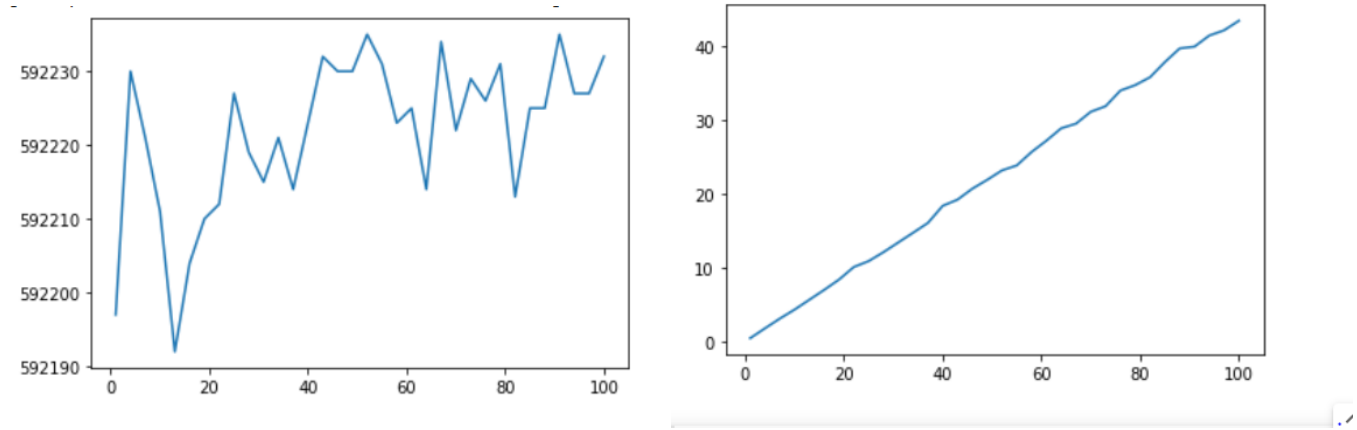


(a) Gain en fonction de la taille de nombre de fourmis (b) Temps d'exécution en fonction de la taille de nombre de fourmis

FIGURE 25 – Etude de l'influence de nombre de fourmis sur le temps d'exécution et sur la qualité de la solution

Analyse et interprétation : On remarque que l'utilisation d'un nombre important de fourmis n'assure pas une convergence absolu vers une solution optimale car cela dépend de la position initiale des fourmis et des choix effectués au hasard au moment de l'exécution de plus le temps d'exécution augmente en augmentant le nombre de fourmis.

- **Gain et temps d'exécution en fonction de nombre d'itération :**



(a) Gain en fonction de nombre d'itération (b) Temps d'exécution en fonction de nombre d'itération

FIGURE 26 – Etude de l'influence de nombre d'itération sur le temps d'exécution et sur la qualité de la solutions

Analyse et interprétation : On remarque qu' utiliser un grand nombre d'itérations peut améliorer le résultat initial. Cependant il faut trouver un bon compromis entre le nombre d'itération et le temps d'exécution car ce dernier augmente de façon considérable quand on augmente le nombre d'itérations.

- **Variation des paramètres alpha et beta :** Ces deux paramètres indiquent l'importance à donner pour les pistes de phéromone et l'utilité de l'objet pendant le choix du prochain objet. Dans la plupart des cas, on les fixe à 1 pour donner la même importance.

5 INTERFACE GRAPHIQUE

5.1 PAGE D'ACCUEIL

La page d'accueil contient un presentation du probleme, un descriptif bref et la liste des méthodes utilisées :

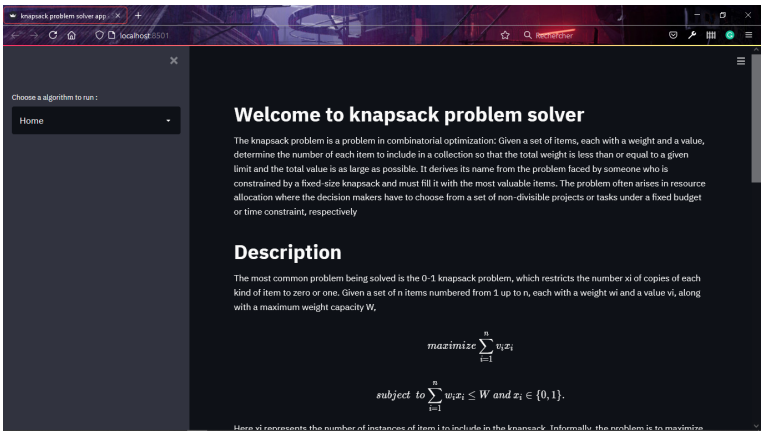


FIGURE 27 – Interface graphique de la page d'accueil

5.2 PAGE D'ÉXCUTION D'UN ALGORITHME

L'application contient une page pour chaque algorithme traité, cette page est composé d'un description de la méthode, la possibilité d'exécuter l'algorithme sur une seule instance ou plusieurs instance pour générer un graphe de temps et de gain pour chaque nombre d'élément dans les instances.

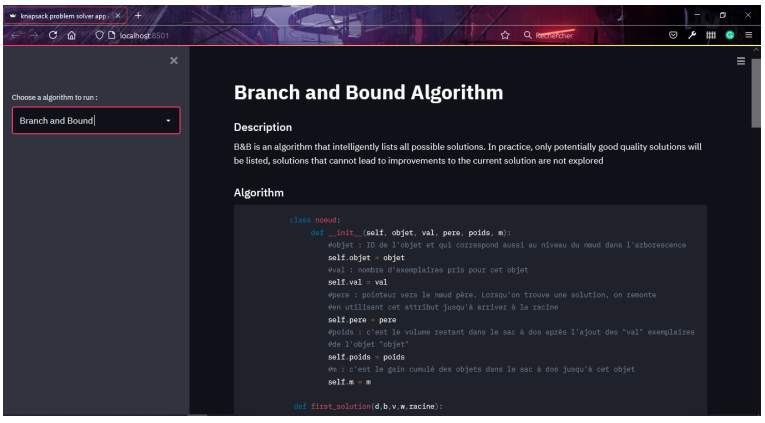


FIGURE 28 – Interface graphique d'un algorithme

FIGURE 29 – Interface graphique du graphe des resultats en terms de temps et de gain de l'algorithme B&B

L'application offre la possibilité de changer les paramètres de l'algorithme (si l'algorithme a besoin des paramètres).

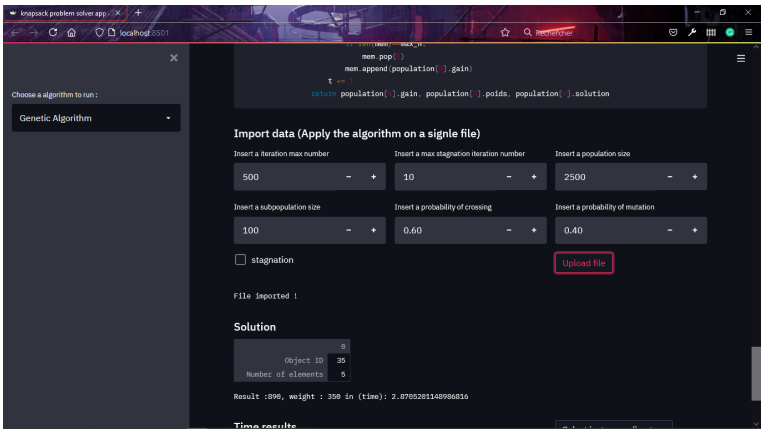


FIGURE 30 – Interface graphique pour changer les paramètres d'algorithme

5.3 PAGE DE COMPARAISON

L'application offre la possibilité de faire une comparaison entre les différents algorithmes en termes de temps et de gain après sélection du répertoire des instances.

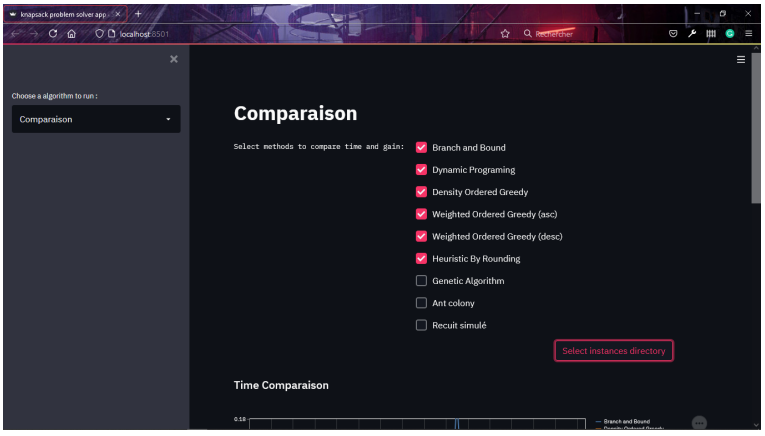


FIGURE 31 – Interface graphique pour sélectionner les algorithmes à comparer

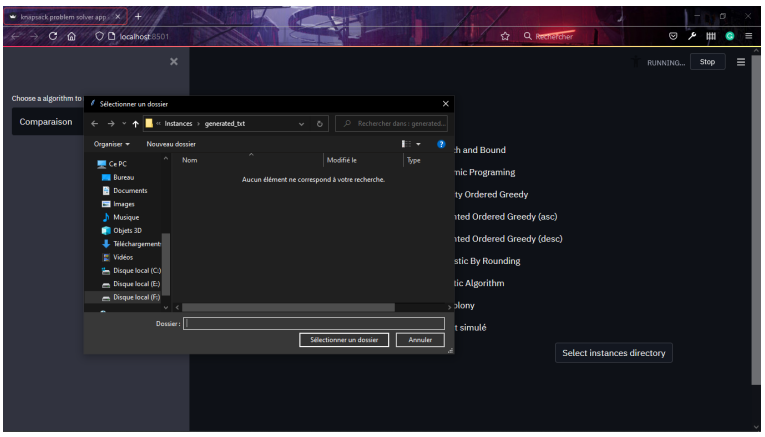


FIGURE 32 – Interface graphique pour sélectionner les instances

FIGURE 33 – Interface graphique de la comparaison entre les algorithmes

6 CONCLUSION

A travers cette première partie du projet nous avons implémenté plusieurs méthodes de résolution du problème du sac à dos dans sa version non bornée. Nous avons commencé par les méthodes exactes pour lesquelles nous avons rapidement observé les limites. Leur temps d'exécution augmente brusquement avec des instances de grande taille. Voilà pourquoi nous sommes passés à des méthodes approchées. Nous avons donc implémenté quelques heuristiques qui permettent de réduire considérablement le temps d'exécution mais qui ne se rapprochent pas toujours de la solution optimale. Pour améliorer la qualité de la solution tout en gardant un temps d'exécution convenable nous avons vu que les métaheuristiques sont une bonne alternative. Cependant ces méthodes dépendent d'un ensemble de paramètres qu'il faut ajuster afin d'obtenir de bonnes performances.

La résolution du problème du sac à dos reste une voie de recherche ouverte puisque l'on cherche toujours à mettre en place des méthodes de plus en plus évoluées qui nous permettraient de se rapprocher de la meilleure solution tout en gardant un temps d'exécution convenable. On pourrait se demander si on trouvera un jour la méthode parfaite qui donne la solution optimale dans des délais corrects mais cette question reste en suspend. En effet, y répondre signifierait qu'on aurait résolu un des sept problèmes du millénaire en répondant à la fameuse question P est-t-il égal à NP..

7 BIBLIOGRAPHIE

- Bernhard Korte, Jens Vygen. “Optimisation combinatoire : Théorie et algorithmes”, édition Springer, 2010.
- Article sur le problème du sac à dos du Laboratoire Bordelais de Recherche en Informatique (LaBRI) : https://dept-info.labri.fr/ENSEIGNEMENT/projet2/supports/Sac-a-dos/Le-probleme-du-sac-a-dos.pdf
- Article de Gabriel Cormier sur les algorithmes génétiques : http://www8.umoncton.ca/umcm-cormier_gabriel/SystemesIntelligents/AG.pdf
- Les cours d’optimisation combinatoire de Mme. Bessedik Malika, Ecole nationale Supérieure d’Informatique (ESI).
- Ines Alaya, Christine Solnon, Khaled Ghedira. Ant algorithm for the multidimensional knapsack problem. International conference on Bioinspired Methods and their Applications (BIOMA 2004) : https://hal.archives-ouvertes.fr/hal-01541529/document
- Cours de l’université de Stanford sur la programmation dynamique : http://web.stanford.edu/class/archive/cs/cs161/cs161.1194/Lectures/Lecture13/Lecture13-compressed.pdf
- https://fr.wikipedia.org/wiki/Programmation_dynamique