Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département d'Intelligence Artificielle
et Sciences de Données

Master I Systèmes Informatiques Intelligents

Rapport du TP Méta-heuristiques pour la résolution du problème NP-COMPLET

‘’ Problèmes de Partition ‘’

Réalise par :

- DJENANE Nihad 191931040689 G1
- M'BAREK Lydia 181831064011 G1

Année universitaire
2022/2023

Table de matières

List des figures

List des tableaux

Introduction

Les problèmes NP-COMPLETS sont une classe de problèmes de la théorie de la complexité qui sont considérés comme les problèmes les plus difficiles à résoudre.

Pour de nombreux problèmes NP-COMPLETS, il n'existe pas d'algorithme connu qui puisse fournir une solution exacte en un temps raisonnable.

Le problème de partitionnement peut sembler simple à première vue, mais il est en fait NP-COMPLET, ce qui signifie il n'existe pas d'algorithme efficace pour résoudre ce problème pour des ensembles de grande taille.

Dans ce qui suit, on va étudier le problème de partitionnement avec des différentes solutions proposées et plusieurs tailles.

1. Description du problème

Le problème de partitionnement d'un ensemble est un problème mathématique fondamental qui consiste à diviser un ensemble de n éléments en deux sous-ensembles disjoints, de telle sorte que la différence entre les sommes des éléments dans les deux sous-ensembles soit minimale.

Plus formellement, supposons que nous ayons un ensemble A = {a1, a2, ..., an} de n éléments. Nous voulons diviser A en deux sous-ensembles B et C tels que
$B \cup C = A$ et taille(B) + taille(C) = taille (A)
Nous voulons minimiser la différence entre les sommes des éléments de B et de C, c'est-à-dire :
|sum(B) - sum(C)| où sum(B) et sum(C) sont les sommes des éléments de B et de C respectivement.

2. Modalisation et résolution du problème

Nous avons proposé une modélisation du problème de partitionnement sous forme de deux listes :

- Liste d'ensemble : Soit Ensemble = [a1, a2, ..., an] une liste d'entiers positifs représentant l'ensemble de départ.
- Liste de solution : Soit Solution = [1, 1, ...2..., 1, 2] une liste de n entiers représentant la solution ou la valeur 1 représente les éléments assignés à l'ensemble A, et la valeur 2 représente les éléments assignés à l'ensemble B.

Par exemple, si Ensemble = [5, 3, 2, 7, 4], alors une solution possible pourrait être [1, 2, 2, 1, 1], ce qui signifie que les éléments a1, a4 et a5 appartiennent à l'ensemble A, et les éléments a2 et a3 appartiennent à l'ensemble B.

Afin d'illustrer cette modélisation en Java, nous avons utilisé trois classes distinctes.

2.1 Class Partition

La classe Partition est utilisé pour représenter le problème. Elle contient les attributs suivants :

- Liste ensemble : représente l'ensemble à partitionner.
- Liste des listes d'entier qui représente les différentes solutions trouvée.

```
public class Partition {


    private List<Integer> ensemble  ;
    private List< List<Integer> > solution  ;
    public int difference;

    public Partition(){
        ensemble = new ArrayList();
        solution = new ArrayList< List <Integer> > ();
        difference = -1;
    }
```

Figure 1 : Attributes du class Partition

Elle contient également les méthodes suivantes :

- La méthode "generateInst" génère une instance du problème de taille n.

```
// génération d'une instance du problème
public void generateInst(int n) {
    int valAleo;
    for (int i = 0; i < n; i++) {
        valAleo = (int) (Math.random() * ((300 - 10) + 1));
        ensemble.add(valAleo);
    }
}
```

Figure 2 : Méthode pour génère une instance

- La méthode "solutionAleo" génère une solution initiale pour le problème en attribuant une valeur de 1 ou 2 à chaque élément de l'ensemble en fonction de sa parité.

```
// Solution
    public void solutionAleo() {
        int size = ensemble.size();
        for (int i = 0; i < size; i++) {
            if (ensemble.get(i) % 2 == 0) {
                solution.add(1);
            } else {
                solution.add(2);
            }
        }

    }
```

Figure 3 : Méthode pour génère une solution aléatoire

- La méthode "validSolution" vérifie si une solution donnée est valide ou non, en s'assurant que la taille de l'ensemble est égale à la liste de solutions et que chaque sous-ensemble contient au moins un élément.

```
// La vérification de la validité d'une solution
    public boolean validSolution() {
        int nb1 = 0, nb2 = 0, size = solution.size();
        if (size != ensemble.size()) {
            return false;
        }
        for (int i = 0; i < size; i++) {
            if (solution.get(i) == 1) {
                nb1++;
            } else if (solution.get(i) == 2) {
                nb2++;
            } else {
                return false;
            }
        }
        if (nb1 == 0 || nb2 == 0) {
            return false;
        }
        return true;
    }
```

Figure 4 : Méthode qui vérifier si la solution est valide

- La méthode "Evaluation" qui retourne la différence entre les sommes des ensembles A et B.

```
// L'Évaluation d'une solution
    public int Evaluation() {
        int s1 = 0, s2 = 0;
        int size = solution.size();
        for (int i = 0; i < size; i++) {
            if (solution.get(i) == 1) {
                s1 = s1 + ensemble.get(i);
            } else {
                s2 = s2 + ensemble.get(i);
            }
        }
        return abs(s1 - s2);
    }
```

- La méthode "equale" qui prend en entre deux solutions est vérifié si sont égaux.

```
public boolean equale ( List<Integer> solution1 , List<Integer> solution2){
    if ( solution1 == null && solution2 == null ) return true;

    List Ensemble11 , Ensemble12 , Ensemble21 , Ensemble22;
    Ensemble11 = new ArrayList();
    Ensemble12 = new ArrayList();
    Ensemble21 = new ArrayList();
    Ensemble22 = new ArrayList();

    for ( int i = 0; i < this.ensemble.size(); i++){

        if ( solution1.get(i) == 1 ) Ensemble11.add(this.ensemble.get(i));
        else Ensemble12.add(this.ensemble.get(i));

        if ( solution2.get(i) == 1 ) Ensemble21.add(this.ensemble.get(i));
        else Ensemble22.add(this.ensemble.get(i));

    }

    if ( Ensemble11.equals(Ensemble21) && Ensemble12.equals(Ensemble22)  ) return true;
    if ( Ensemble11.equals(Ensemble22) && Ensemble12.equals(Ensemble21) ) return true;

    return false;

}
```

Figure 5 : La méthode equale

- + Les méthodes "afficheEnsemble" et "afficheSolution" qui sont utilisées pour afficher l'ensemble initial et une solution donnée, respectivement.

2.2 Class DFSsolution

Cette classe contient une implémentation de l'algorithme de recherche en profondeur d'abord (DFS) pour résoudre le problème de la partition. Les principaux attributs de cette classe sont :

- une liste de solutions trouvées pour stocker temporairement une solution trouve (solutionTrouve).
- une liste de toutes les solutions trouvées (AllSolutions).
- une valeur de différence (difference) .
- un compteur pour le nombre de nœuds visités lors de la recherche en profondeur du 1 er solution (nouedGenre).

Les méthodes principales de cette classe est :

- La méthode "solve_Partition" prend en entrée un objet de type Partition et index initialisé à zéro représentant la position actuelle dans l'ensemble à partitionner.

Tout d'abord, la méthode vérifie si la solution n'a pas encore été trouvée, si tel est le cas, elle incrémente un compteur de nœuds visités.
Ensuite, elle vérifie si l'indice actuel est égal à la taille de l'ensemble. Si oui, elle évalue la validité de la solution et la compare à celles déjà trouvées. Si la solution est la première trouvée, elle l'ajoute au problème. Sinon, si la différence est inférieure à celle précédemment trouvée, elle efface toutes les solutions précédentes et ajoute cette nouvelle solution à la liste des solutions. Si la différence est égale à celle précédemment trouvée et que la solution n'a pas été déjà trouvée, elle est ajoutée à la liste des solutions du problème.
Si l'indice actuel n'est pas égal à la taille de l'ensemble, cela signifie qu'il reste encore des valeurs à attribuer, donc un appel récursif est effectué avec l'indice incrémenté.

- La méthode alreadyFound est une méthode qui vérifie si une solution a déjà été trouvée dans la liste de toutes les solutions.

```
public void solve_Partition(Partition problem , int index ){
    if ( problem.difference == -1 ) this.nouedGenre++;
    if ( index == problem.getEnsemble().size()){
        if( problem.validSolution(solutionTrouve) ) {
            if ( ! alreadyFound(problem , solutionTrouve))
                this.AllSolutions.add(new ArrayList(solutionTrouve));
            this.difference = problem.Evaluation(solutionTrouve);
            if ( problem.difference == -1 ){
                problem.addSolution(solutionTrouve);
                problem.difference = this.difference;
            }
            else if ( this.difference == problem.difference && ! problem.alreadyFound(solutionTrouve) )
                problem.addSolution(solutionTrouve);
            else if ( this.difference < problem.difference ){
                problem.getSolution().clear();
                problem.addSolution(solutionTrouve);
                problem.difference = this.difference;
            }
        }
    }
    else {
        for ( int i = 1; i <= 2; i++){
            this.solutionTrouve.add(index , i );
            solve_Partition(problem , index + 1 );
            this.solutionTrouve.remove(index);
        }
    }
}
```

Figure 6 : Méthode de resolution DFS

2.3 Class Main

Cette classe est le principale de programme contient les appels aux fonctions créées ainsi que les affichages des questions demandées.

```
public class main {

    public static void main(String[] args) {

        long startTime , endTime ;

        List<Integer> ensemble =  new ArrayList<>(Arrays.asList(382745, 799601, 909247, 729069, 467902,  44328,  34610, 698150, 823460,
));

        Partition problem = new Partition(ensemble);
        DFSsolution solution = new DFSsolution();

    //   problem.generateInst(15);
        System.out.println("L'ensemble est  : ");
        problem.afficheEnsemble();

        startTime = System.currentTimeMillis();

        solution.solve_Partition(problem, 0);

        endTime = System.currentTimeMillis();

        if ( problem.getSolution().isEmpty() )
            System.out.println(" Aucun solution trouve !  ");
        else {
            System.out.println(" Temps d'exeution est : " + (endTime - startTime) + "ms");
            System.out.println("\n Nombre de solutions  : " + solution.AllSolutions.size());
            System.out.println("\n Nombre de noued avant de trouve la 1 er solution  : " + solution.nouedGenre);
            System.out.println("\n Nombre de solution optimale trouve  : " + problem.getSolution().size());
            System.out.println("\n La difference minimum est : " + problem.difference);
            System.out.println(" Listes des  solutions optimale trouve : ");
            for ( int i = 0; i < problem.getSolution().size(); i++) {
                System.out.println(" Solution " + (i + 1 ) );
                problem.afficheSolution(problem.getSolution().get(i));
                System.out.println(" ");
            }
```

Figure 7 : Class main

3. Etude expérimentale

Environnement Expérimentale :

RAM : 8GO .

Processeur : Intel(R) Core™ i7-7820HQ CPU @ 2.90 GHz

Cette partie est répartie en deux sous parties :

- D'abord, on va faire une exécution de dataset envoyées, et pour chaque instance on rapporte :
  - Le temps d'exécution.
  - Le nombre de nœuds générés avant de trouver la 1ère solution.
  - Le nombre de solution trouvés.
  - La différence entre les deux sous-ensembles.
- La liste des solutions trouvées.

Instance 1 : 31, 10, 20, 19, 4, 3, 6

```
L'ensemble est  :
31 || 10 || 20 || 19 || 4 || 3 || 6 ||
 Temps d'exeution est : 13ms

 Nombre de solutions  : 63

 Nombre de noued avant de trouve la 1 er solution  : 8

 Nombre de solution optimale trouve  : 1

 La difference minimum est : 1
 Listes des  solutions optimale trouve :
 Solution 1
    L'ensemble 1 : 31 || 10 || 6 ||
    L'ensemble 2 : 20 || 19 || 4 || 3 ||
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 8 : Instance 1

Instance 2 : 25, 35, 45, 5, 25, 3, 2, 2

```
L'ensemble est  :
25 || 35 || 45 || 5 || 25 || 3 || 2 || 2 ||
 Temps d'exeution est : 23ms

 Nombre de solutions  : 95

 Nombre de noued avant de trouve la 1 er solution  : 9

 Nombre de solution optimale trouve  : 4

 La difference minimum est : 2
 Listes des  solutions optimale trouve :
 Solution 1
    L'ensemble 1 : 25 || 35 || 5 || 3 || 2 || 2 ||
    L'ensemble 2 : 45 || 25 ||
 Solution 2
    L'ensemble 1 : 25 || 35 || 5 || 3 || 2 ||
    L'ensemble 2 : 45 || 25 || 2 ||
 Solution 3
    L'ensemble 1 : 25 || 45 || 2 ||
    L'ensemble 2 : 35 || 5 || 25 || 3 || 2 ||
 Solution 4
    L'ensemble 1 : 25 || 45 ||
```

Figure 9 : Instance 2

Instance 3 : 3, 4, 3, 1, 3, 2, 3, 2, 1

```
L'ensemble est  :
3 || 4 || 3 || 1 || 3 || 2 || 3 || 2 || 1 ||
 Temps d'exeution est : 64ms

 Nombre de solutions  : 239

 Nombre de noued avant de trouve la 1 er solution  : 10

 Nombre de solution optimale trouve  : 23

 La difference minimum est : 0
 Listes des  solutions optimale trouve :
 Solution 1
    L'ensemble 1 : 3 || 4 || 3 || 1 ||
    L'ensemble 2 : 3 || 2 || 3 || 2 || 1 ||
 Solution 2
    L'ensemble 1 : 3 || 4 || 3 || 1 ||
    L'ensemble 2 : 1 || 3 || 2 || 3 || 2 ||
```

Figure 10 : Instance 3

Instance 4 : 2, 10, 3, 8, 5, 7, 9, 5, 3, 2

```
L'ensemble est  :
2 || 10 || 3 || 8 || 5 || 7 || 9 || 5 || 3 || 2 ||
 Temps d'exeution est : 225ms

 Nombre de solutions  : 511

 Nombre de noued avant de trouve la 1 er solution  : 11

 Nombre de solution optimale trouve  : 23

 La difference minimum est : 0
 Listes des  solutions optimale trouve :
 Solution 1
    L'ensemble 1 : 2 || 10 || 3 || 5 || 7 ||
    L'ensemble 2 : 8 || 9 || 5 || 3 || 2 ||
 Solution 2
    L'ensemble 1 : 2 || 10 || 3 || 5 || 5 || 2 ||
    L'ensemble 2 : 8 || 7 || 9 || 3 ||
```

Figure 11 : Instance 4

Instance 5 : 484, 114, 205, 288, 506, 503, 201, 127, 410

```
L'ensemble est  :
484 || 114 || 205 || 288 || 506 || 503 || 201 || 127 || 410 ||
 Temps d'exeution est : 66ms

 Nombre de solutions  : 255

 Nombre de noued avant de trouve la 1 er solution  : 10

 Nombre de solution optimale trouve  : 1

 La difference minimum est : 0
 Listes des  solutions optimale trouve :
 Solution 1
    L'ensemble 1 : 484 || 114 || 205 || 288 || 201 || 127 ||
    L'ensemble 2 : 506 || 503 || 410 ||
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 12 : Instance 5

Instance 6 : 23, 31, 29, 44, 53, 38, 63, 85, 89, 82

```
L'ensemble est  :
23 || 31 || 29 || 44 || 53 || 38 || 63 || 85 || 89 || 82 ||
 Temps d'exeution est : 200ms

 Nombre de solutions  : 511

 Nombre de noued avant de trouve la 1 er solution  : 11

 Nombre de solution optimale trouve  : 5

 La difference minimum est : 1
 Listes des  solutions optimale trouve :
 Solution 1
    L'ensemble 1 : 23 || 31 || 29 || 44 || 53 || 89 ||
    L'ensemble 2 : 38 || 63 || 85 || 82 ||
 Solution 2
    L'ensemble 1 : 23 || 31 || 29 || 38 || 63 || 85 ||
    L'ensemble 2 : 44 || 53 || 89 || 82 ||
 Solution 3
    L'ensemble 1 : 23 || 31 || 44 || 89 || 82 ||
    L'ensemble 2 : 29 || 53 || 38 || 63 || 85 ||
```

Figure 13 : Instance 6

Instance 7 : 771, 121, 281, 854, 885, 734, 486, 1003, 83, 62

```
771 || 121 || 281 || 854 || 885 || 734 || 486 || 1003 || 83 || 62 ||
 Temps d'exeution est : 222ms

 Nombre de solutions  : 511

 Nombre de noued avant de trouve la 1 er solution  : 11

 Nombre de solution optimale trouve  : 1

 La difference minimum est : 0
 Listes des  solutions optimale trouve :
 Solution 1
    L'ensemble 1 : 771 || 281 || 854 || 734 ||
    L'ensemble 2 : 121 || 885 || 486 || 1003 || 83 || 62 ||
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 14 : Instance 7

Instance 8 : 70, 73, 77, 80, 82, 87, 90, 94, 98, 106, 110, 113, 115, 118, 120

```
L'ensemble est  :
70 || 73 || 77 || 80 || 82 || 87 || 90 || 94 || 98 || 106 || 110 || 113 || 115 || 118 || 120 ||
 Temps d'exeution est : 92117ms

 Nombre de solutions  : 16383

 Nombre de noued avant de trouve la 1 er solution  : 16

 Nombre de solution optimale trouve  : 59

 La difference minimum est : 1
 Listes des  solutions optimale trouve :
 Solution 1
    L'ensemble 1 : 70 || 73 || 77 || 80 || 82 || 106 || 110 || 118 ||
    L'ensemble 2 : 87 || 90 || 94 || 98 || 113 || 115 || 120 ||
 Solution 2
    L'ensemble 1 : 70 || 73 || 77 || 80 || 82 || 106 || 113 || 115 ||
    L'ensemble 2 : 87 || 90 || 94 || 98 || 110 || 118 || 120 ||
 Solution 3
```

Figure 15 : Instance 8

Instance 9 : 382745, 799601, 909247, 729069, 467902, 44328, 34610, 698150, 823460, 903959, 853665, 551830, 610856, 670702, 488960, 951111, 323046, 446298, 931161, 31385, 496951, 264724, 224916, 169684 :

```
L'ensemble est  :
382745 || 799601 || 909247 || 729069 || 467902 || 44328 || 34610 || 698150 || 823460 || 903959 || 853665 || 5518
 Temps d'exeution est :  10867521 ms

 Nombre de solutions  : 16777215

 Nombre de noued avant de trouve la 1 er solution  : 25

 Nombre de solution optimale trouve  :  2

 La difference minimum est : 0
```

Figure 16 : Instance 9

- Dans cette partie, on a généré des instances aléatoires avec des tailles croissantes, et on a mesurés le temps d'exécutions pour chaque instance comme l'indique le tableau suivant

Table 1 : Tableau représente les temps d'exécution en fonction de différentes tailles de N.

N	5	10	15	20	25
Temps d'exécution (min)	0.00001	0.00331	1.52000	31.58693	108.67521

Le graphe

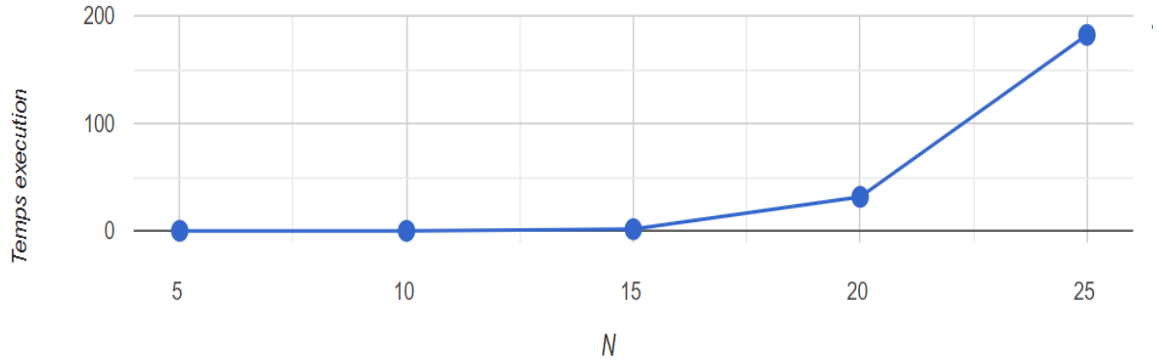

Figure 17 : Graphe represente temps d'execution par rapport à N

Résultat :

Nous avons testé notre méthode de résolution sur plusieurs instances du problème, en utilisant des données réelles (dataset) ou générées aléatoirement.

Les résultats obtenus ont montré que le temps d'exécution de l'algorithme augmente de manière très rapide avec la taille de l'ensemble de données. Cela confirme que le problème de partitionnement est très complexe et nécessite une exploration exhaustive de l'espace de recherche pour trouver une solution optimale.

Cependant, malgré cette complexité, notre méthode de résolution a été capable de trouver des solutions optimales pour toutes les instances testées. Cela montre que l'algorithme DFS est capable de résoudre efficacement le problème de partitionnement pour des ensembles de données de taille raisonnable.

Conclusion

Dans cette étude, nous avons exploré la méthode de parcours en profondeur (DFS) pour résoudre le problème de partitionnement. Nous avons démontré que cette méthode est garantie de trouver une solution valide, bien qu'elle puisse être coûteuse en termes de temps et de ressources pour des ensembles plus grands.

Cependant, il est important de noter qu'il existe des techniques pour améliorer la méthode DFS. Par exemple, l'utilisation d'heuristiques peut aider à réduire l'espace de recherche et accélérer le processus de recherche.

La méthode DFS reste une technique importante pour résoudre le problème de partitionnement, mais des améliorations peuvent être apportées en utilisant des techniques pour accélérer la recherche. Ces améliorations permettent d'obtenir des résultats plus rapidement et avec moins de ressources pour des ensembles plus grands.