

Rapport de projet informatique

Problème du set-cover et algorithme de branchement

Réalisé par

Badr Agrad

Table des matières

1	Introduction	3
1.1	Contexte	3
1.2	Motivation	3
2	Algorithme de branchement	4
3	Conception de l'algorithme	5
3.1	Code	5
3.2	Structure du projet	6
4	Résultat obtenu et conclusions	8
5	Manuel d'utilisation	9

1 Introduction

1.1 Contexte

Le problème du set cover est un défi mathématique fondamental en informatique. En effet, il pose une question cruciale : Comment sélectionner le moins de sous-ensembles possible parmi un ensemble donné afin de couvrir tous les éléments de manière optimale ? Ce problème trouve des applications dans de nombreux domaines tels que la logistique, la planification ou encore les réseaux de communication. Formellement, considérons U un ensemble d'éléments et $S = \{S_1, S_2, \dots, S_k\}$ un ensemble de sous-ensembles de U . L'objectif est de trouver C , un sous-ensemble minimal de S , tel que l'union de tous les ensembles dans C soit égale à U .

Prenons un exemple, soit un ensemble $U = \{1, 2, 3, 4, 5, 6, 7\}$ et soit $S = \{A, B, C, D, E, F\}$ une collection de sous-ensembles tel que :

- $A = \{1, 4, 7\}$
- $B = \{1, 4\}$
- $C = \{4, 5, 7\}$
- $D = \{3, 5, 6\}$
- $E = \{2, 3, 6, 7\}$
- $F = \{2, 7\}$

Alors, la sous-collection $\{B, D, F\}$ est une couverture car elle couvre l'ensemble des éléments de U en une taille minimale de sous-ensembles.

1.2 Motivation

Le Problème du Set Cover est reconnu pour sa grande complexité. Sa nature combinatoire en fait un défi redoutable pour les algorithmes d'optimisation. La difficulté principale réside dans la recherche de la meilleure combinaison de sous-ensembles permettant de couvrir l'ensemble. En effet, le nombre de sous-ensembles possibles pour un ensemble donné croît exponentiellement avec la taille de cet ensemble. Plus précisément, pour un ensemble de taille n , le nombre de sous-ensembles est de 2^n . Cette croissance exponentielle signifie qu'explorer systématiquement chaque combinaison possible de sous-ensembles devient rapidement impraticable, voire impossible.

Cela rend inefficace la stratégie naïve consistant à tester toutes les combinaisons pour trouver la solution optimale. Ainsi, la complexité exponentielle du Problème du Set Cover impose l'utilisation d'approches heuristiques et d'algorithmes efficaces pour obtenir des résultats acceptables dans un délai raisonnable.

2 Algorithme de branchement

L'optimisation de problèmes complexes tels que le Problème du Set Cover a suscité l'intérêt pour d'autres approches algorithmiques. Parmi celles-ci, les algorithmes de branchement se sont révélés être des outils puissants pour aborder ce problème.

L'algorithme de branchement explore de manière exhaustive l'espace de recherche en utilisant une approche de type "diviser pour régner". Il procède par itérations successives, choisissant de manière sélective les sous-ensembles à inclure dans la solution, en évaluant à chaque étape toutes les possibilités pour construire la couverture de l'ensemble U . L'algorithme élimine les branches non prometteuses dès qu'il est évident qu'elles ne mèneront pas à une solution optimale.

Dans le cadre du Problème du Set Cover, l'algorithme de branchement propose une approche systématique pour énumérer et évaluer toutes les combinaisons possibles de sous-ensembles, en visant à trouver la solution optimale ou une solution proche de l'optimalité.

Soit un vecteur de 0 et de 1 représentant les sous ensembles sélectionnées durant l'exécution de notre algorithme.

1. **Initialisation** : Définir un vecteur représentant l'ensemble universel et initialiser un vecteur de 0 et de 1 pour représenter les sous-ensembles sélectionnés.
2. **Sélection séquentielle** : Parcourir séquentiellement les sous-ensembles disponibles et sélectionner un sous-ensemble à chaque itération.
3. **Évaluation de la solution** : À chaque étape, évaluer si la sélection actuelle de sous-ensembles permet de couvrir tous les éléments de l'ensemble universel.
4. **Branchement** : Si la solution actuelle n'est pas complète, créer des branches pour explorer toutes les possibilités. Pour chaque branche, sélectionner un nouveau sous-ensemble pour compléter la solution partielle.
5. **Élimination des branches non-prometteuses** : À mesure que les branches sont explorées, éliminer les branches qui ne peuvent pas conduire à une solution optimale ou qui sont moins prometteuses en termes de couverture.
6. **Répétition** : Répéter les étapes précédentes jusqu'à ce qu'une solution complète soit trouvée ou jusqu'à l'exploration de toutes les branches possibles.

L'algorithme de branchement procède ainsi de manière itérative, explorant systématiquement les différentes combinaisons de sous-ensembles pour trouver une solution qui couvre efficacement tous les éléments de l'ensemble universel.

3 Conception de l'algorithme

3.1 Code

D'une part, il est essentiel d'initialiser la collection de sous-ensembles de l'exemple plus haut par une matrice d'entiers.

	1	2	3	4	5	6	7
<i>A</i>	1	0	0	1	0	0	1
<i>B</i>	1	0	0	1	0	0	0
<i>C</i>	0	0	0	1	1	0	1
<i>D</i>	0	0	1	0	1	1	0
<i>E</i>	0	1	1	0	0	1	1
<i>F</i>	0	1	0	0	0	0	1

Cette matrice nous permettra de connaître le score du vecteur courant. En effet, en le parcourant, nous avons la possibilité pour chaque indice sélectionné (soit 1) de nous référer au sous ensembles correspondant. La combinaisons de ces sous-ensembles permet de savoir si c'est une solution ou non. L'algorithme de branchement s'arrête lorsque nous avons trouvé un score optimal. Dans le cas échéant, le vecteur courant nous permet de connaître la combinaisons de sous-ensembles.

Dans le pseudo-code ci-dessus, les fonctions :

- *compter_uns*(*int* * *tab*, *int* *taille*) permet de compter le nombre de 1 dans un vecteur.
- *is_solution*(*int* * *tab*) permet de déterminer si un vecteur est solution du Set Cover
- *can_continue*(*int* * *tab*, *int* *i*) permet de déterminer si il est judicieux ou non de sélectionner le prochain sous-ensemble. Ainsi, cela permet d'éliminer les branches inutiles.

Algorithm 1 Fonction Branchement

```
1: function BRANCHEMENT(current, index, best)
2:   occurrence  $\leftarrow$  occurrence + 1
3:   if compter_uns(current, nombreSousEnsembles) > best then
4:     return best
5:   end if
6:   if is_solution(current) then
7:     return compter_uns(current, nombreSousEnsembles)
8:   end if
9:   if index  $\geq$  nombreSousEnsembles then
10:    return best
11:  end if
12:  current[index]  $\leftarrow$  1
13:  best  $\leftarrow$  BRANCHEMENT(current, index + 1, best)
14:  current[index]  $\leftarrow$  0
15:  if can_continue(current, index + 1) then
16:    best  $\leftarrow$  BRANCHEMENT(current, index + 1, best)
17:  end if
18:  return best
19: end function
```

3.2 Structure du projet

Afin de rendre l'algorithme adaptable, nous allons définir une structure de benchmark qui permettra de récupérer les paramètres du programme depuis un fichier texte.

```
typedef struct {  
    int nombreSousEnsembles;  
    int tailleEnsemble;  
    int **matriceEnsembles;  
} BenchmarkParameters;
```

FIGURE 1 – Structure benchmark

Ainsi, voici comment est agencé le fichier texte :

```
≡ benchmark.txt  
1  NombreSousEnsembles 6  
2  TailleEnsemble 7  
3  1 0 0 1 0 0 1  
4  1 0 0 1 0 0 0  
5  0 0 0 1 1 0 1  
6  0 0 1 0 1 1 0  
7  0 1 1 0 0 1 1  
8  0 1 0 0 0 0 1
```

FIGURE 2 – Caption

La fonction *readBenchmarkParameters()* nous permettra de lire et de stocker ces informations.

```
BenchmarkParameters readBenchmarkParameters() {
    FILE *file = fopen("benchmark.txt", "r");
    if (file == NULL) {
        perror("Erreur lors de l'ouverture du fichier de benchmark");
        exit(1);
    }

    fscanf(file, "NombreSousEnsembles %d\n", &params.nombreSousEnsembles);
    fscanf(file, "TailleEnsemble %d\n", &params.tailleEnsemble);

    params.matriceEnsembles = malloc(params.nombreSousEnsembles * sizeof(int *));

    for (int i = 0; i < params.nombreSousEnsembles; i++) {
        params.matriceEnsembles[i] = malloc(params.tailleEnsemble * sizeof(int));
        for (int j = 0; j < params.tailleEnsemble; j++) {
            fscanf(file, "%d", &params.matriceEnsembles[i][j]);
        }
    }
    fclose(file);
    return params;
}
```

FIGURE 3 – Fonction *readBenchmarkParameters()*

Les fichiers sont séparés entre les fonctions liés à l’algorithme génétique, le benchmarking, ainsi que la fonction *main* permettant d’utiliser l’algorithme. Voici l’arborescence de fichiers du projet :

```
/ (racine)
├── Makefile
├── benchmark.txt
├── bin
│   └── main
├── include
│   ├── benchmark.h
│   └── branchement.h
├── obj
│   ├── branchement.o
│   └── main.o
└── src
    ├── branchement.c
    └── main.c
```

4 Résultat obtenu et conclusions

L'utilisation de l'algorithme de branchement pour résoudre le problème du Set Cover s'est révélée particulièrement efficace. En seulement une quarantaine d'appels récursifs à la fonction de branchement, nous avons obtenu une solution optimale.

L'algorithme de branchement a démontré sa capacité à explorer de manière systématique et efficace l'espace de recherche des combinaisons de sous-ensembles. En éliminant sélectivement les branches inintéressantes, l'algorithme a pu converger rapidement vers une solution optimale sans avoir besoin d'explorer toutes les possibilités. Voici ce que l'algorithme propose comme solutions :

```
---Solution Minimale---
**3** 1 | 0 | 0 | 1 | 0 | 1

---Solution Minimale---
**3** 0 | 1 | 1 | 0 | 1 | 0

---Solution Minimale---
**3** 0 | 1 | 0 | 1 | 1 | 0

---Solution Minimale---
**3** 0 | 1 | 0 | 1 | 0 | 1

** La nombre de sous-ensembles optimal est : 3
** La fonction a réalisé 42 appels récursifs
```

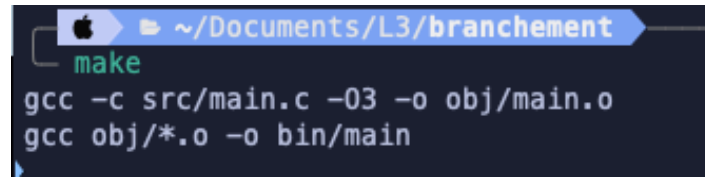
FIGURE 4 – solutions

Cette rapidité dans la convergence vers la solution optimale souligne la puissance de l'algorithme de branchement pour le Problème du Set Cover. Comparé à d'autres approches, l'algorithme de branchement s'avère être une méthode efficace pour trouver une solution optimale dans un temps relativement court.

En conclusion, l'algorithme de branchement offre une approche robuste et efficace pour résoudre le Problème du Set Cover. Sa capacité à trouver rapidement des solutions optimales en explorant sélectivement l'espace de recherche en fait une méthode de choix pour des problèmes similaires nécessitant des solutions optimales dans des délais raisonnables.

5 Manuel d'utilisation

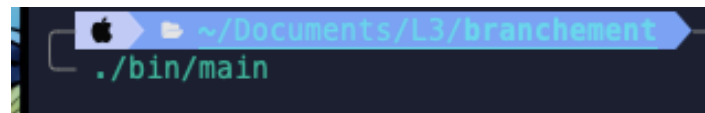
Une MakeFile a été créé afin de pouvoir compiler l'ensemble des fichiers. Ainsi, il vous suffit d'entrer la commande *make* dans le terminal pour compiler le projet :

A terminal window with a dark background and a light blue header bar. The header bar contains an Apple logo, a folder icon, and the text "~/Documents/L3/branchement". The terminal shows the command "make" being entered, followed by the output "gcc -c src/main.c -O3 -o obj/main.o" and "gcc obj/*.o -o bin/main".

```
~/Documents/L3/branchement  
make  
gcc -c src/main.c -O3 -o obj/main.o  
gcc obj/*.o -o bin/main
```

FIGURE 5 – compilation

Ensuite, il vous faudra entrer la commande *./bin/main* dans le terminal afin d'exécuter le projet.

A terminal window with a dark background and a light blue header bar. The header bar contains an Apple logo, a folder icon, and the text "~/Documents/L3/branchement". The terminal shows the command "./bin/main" being entered.

```
~/Documents/L3/branchement  
./bin/main
```

FIGURE 6 – exécution