Université Paris Nanterre

Licence MIAGE troisième année

# Rapport de projet informatique

## Problème du set-cover et algorithme génétique

**Réalisé par**

**Badr Agrad**

# Table des matières

# 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 $S = \{S_1, S_2, \ldots, S_k\}$ 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 génétique

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 génétiques se sont révélés être des outils puissants pour aborder ce problème. Les algorithmes génétiques, inspirés du processus de sélection naturelle, évoluent des solutions potentielles au fil des générations pour converger vers des solutions de meilleure qualité. Cette approche repose sur des concepts de sélection, de croisement et de mutation, simulant le processus de sélection naturelle des gènes pour obtenir des solutions efficaces.

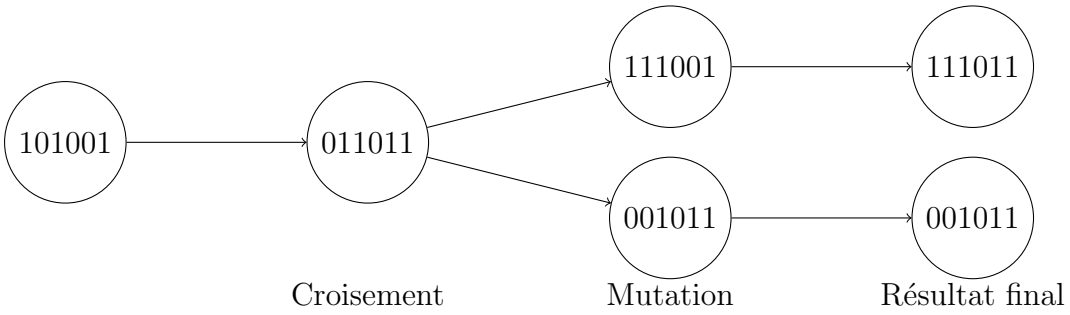
Supposons que les collections de sous-ensembles sont des individus représentés par un vecteur de 0 et de 1 de taille nombre de sous-ensembles. Le 1 indique qu'un sous-ensembles est sélectionné, 0 sinon. La sélection d'un sous-ensemble est aléatoire. Ces individus vont s'accoupler entre eux et chaque couple va engendrer deux enfants.
Par exemple, les individus :

- $\{1, 0, 0, 1, 1, 0, 0\}$
- $\{0, 0, 1, 0, 0, 1, 0\}$

Engendreront les enfants :

- enfant 1 = $\{1, 0, 0, 1, 0, 1, 0\}$
- enfant 2 = $\{0, 0, 1, 0, 1, 0, 0\}$

Ensuite, chaque enfant aura la possibilité de muter selon une certaine probabilité. Si un enfant est muté, chacun des indices du vecteur représentant cet enfant sera soumis à une probabilité de mutation afin de raréfier le processus.
Parmi toute cette population, on sélectionnera les individus ayant le meilleur score : ceux couvrant l'ensemble $U$ dans une taille minimale de sous-ensembles. La réitération de cette expérience nous permettra d'avoir un résultat convergeant vers les meilleures solutions possibles.



# 3 Conception de l'algorithme

## 3.1 Code

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

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $A$ | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| $B$ | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| $C$ | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| $D$ | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| $E$ | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $F$ | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

Cette matrice nous sera utile pour notre algorithme.
Au départ de notre code, nous allons générer aléatoirement vingts individus, modélisés par des tableau d'entiers. Chacun de ces individus va s'accoupler avec un autre individus. On considère qu'un couple génère deux enfants. Ces-derniers récupèrent les trois premiers indices de leurs pères ainsi que les derniers indices de leurs mères ou inversement. Ensuite, nous allons attribuer un score à chaque individu :

— INT MIN, dans le cas où l'individu ne couvre pas l'ensemble des éléments

— le nombre de 0, dans le cas où l'individu est solution

Afin de s'assurer qu'un individu est solution, nous allons comparer le vecteur de sous-ensembles avec la matrice initialisée au départ afin de voir si la combinaisons de sous-ensembles couvre tous les éléments. Enfin, nous trions les éléments selon leur score à l'aide d'un tri à bulle, puis nous ne gardons que les 20 meilleurs individus.
L'expérience sera réitérée 40 fois.

**Algorithm 1** Algorithme Génétique pour le Set Cover

```
1: int[20] individus
2: int[40] population
3: int[20] enfants
4: while nombre d'itérations < 40 do
5:     for individu do
6:         Sélectionner un partenaire parmi individus
7:         Générer deux enfants en combinant les indices des parents
8:         mutation(enfant)
9:         getScore(individu)
10:    end for
11:    bubbleSort(population, taille, score)
12:    Garder les 20 meilleurs individus pour la prochaine itération
13: end while
```

## 3.2 Structure du projet

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

```c
typedef struct {
  int tailleIndividu;
  int tailleEnsemble;
  int populationSize;
  int **matriceEnsembles;
} BenchmarkParameters;
```

FIGURE 1 – Structure benchmark

Ainsi, voici comment est agencé le fichier texte :

```
TailleIndividu 6
TailleEnsemble 7
TaillePopulation 20
1 0 0 1 0 0 1
1 0 0 1 0 0 0
0 0 0 1 1 0 1
0 0 1 0 1 1 0
0 1 1 0 0 1 1
0 1 0 0 0 0 1
```

FIGURE 2 – Fichier txt

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

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

  // BenchmarkParameters params;
  fscanf(file, "TailleIndividu %d\n", &params.tailleIndividu);
  fscanf(file, "TailleEnsemble %d\n", &params.tailleEnsemble);
  fscanf(file, "TaillePopulation %d\n", &params.populationSize);

  // Allouer de la mémoire pour matriceEnsembles et la remplir
  params.matriceEnsembles = malloc(params.tailleIndividu * sizeof(int *));

  for (int i = 0; i < params.tailleIndividu; i++) {
    params.matriceEnsembles[i] =
        (int *)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'argorithme. Voici l'arborescence de fichier du projet :

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

# 4 Résultat obtenu et conclusions

L'exécution de l'algorithme génétique a donné des résultats prometteurs en un temps record, parvenant à atteindre un score optimal de trois pour la couverture des éléments. La représentation des résultats montre que chaque solution trouvée par l'algorithme couvre efficacement tous les éléments requis, ce qui correspond au score optimal attendu.

Cette performance rapide et efficace de l'algorithme génétique souligne sa capacité à trouver des solutions de qualité dans un laps de temps relativement court. En comparaison avec d'autres approches algorithmiques, notamment une recherche exhaustive qui aurait dû explorer un espace de recherche immense, l'utilisation de l'algorithme génétique s'avère être une solution efficace pour résoudre le Problème du Set Cover.

Sans l'algorithme génétique, il aurait fallu un temps considérable pour parcourir toutes les combinaisons possibles de sous-ensembles afin d'atteindre le score optimal de trois. L'approche génétique a permis une exploration plus intelligente de l'espace de recherche, réduisant ainsi drastiquement le temps nécessaire pour obtenir une solution proche de l'optimalité.

En conclusion, les résultats obtenus et la rapidité de l'algorithme génétique démontrent son intérêt et son efficacité pour résoudre le Problème du Set Cover, offrant une approche puissante et efficace pour des problèmes similaires dans divers domaines. Voici quelques résultats obtenus :

```
   -  A  B  C  D  E  F  -
* 00: X        X     X
* 01:    X     X     X
* 02: X        X     X
* 03: X        X     X
* 04: X        X     X
* 05:    X     X     X
* 06: X        X     X
* 07: X        X     X
* 08: X        X     X
* 09: X        X     X
* 10:    X     X     X
* 11: X        X     X
* 12: X        X     X
* 13: X        X     X
* 14: X        X     X
* 15:    X     X     X
* 16: X        X     X
* 17:    X     X     X
* 18: X        X     X
* 19:    X     X     X
```

```
   -  A  B  C  D  E  F  -
* 00: X        X     X
* 01: X        X     X
* 02:    X     X  X
* 03: X        X     X
* 04:    X     X     X
* 05: X        X  X
* 06:    X     X     X
* 07:    X     X  X
* 08: X        X     X
* 09: X        X  X
* 10:    X     X  X
* 11: X        X  X
* 12: X        X  X
* 13: X        X  X
* 14: X        X  X
* 15: X        X  X
* 16: X        X     X
* 17: X        X  X
* 18:    X     X     X
* 19: X        X     X
```

# 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 :

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

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