INF6102 – Métaheuristiques

Travaux pratiques n°4 – Covering Array

*Kévin Baumann – 1647505  
Florian Korsakissok - 1628087*

# I – Présentation du problème

Le problème traité est celui du « Covering Array », ou autrement dit d’une matrice de couverture. La résolution d’une instance d’un tel problème dépend de deux paramètres notés v et k. L’objectif est de remplir une matrice contenant k colonnes avec un minimum de lignes, à l’aide d’entiers compris entre 0 et v-1, de telle sorte que pour chaque paire de symboles, et pour chaque paire de colonnes, au moins une ligne de la matrice contienne cette paire de symboles sur cette paire de colonnes.

Plus formellement, le problème peut être défini comme suit :

* Données d’entrée : v, k
* Sortie de l’algorithme : une matrice M à coefficients dans [0 ; v-1] de taille N\*k
* Contraintes : pour tout i, j dans [0 ; k-1], pour tout a, b dans [0 ; v-1], il existe une ligne l telle que M[l][i] = a et M[l][j] = b
* Objectif : minimiser N

N est la grandeur à minimiser : on veut se retrouver avec la matrice la plus petite possible, c’est-à-dire écrire aussi peu de lignes que possible dans le résultat. De ce fait, N peut être vu comme un coût dans le cadre de ce problème.

Dans l’implémentation de l’algorithme proposé, on utilisera souvent la notion de « contrainte élémentaire ». Une contrainte élémentaire est un quadruplet (i,j,a,b), et on dira que cette contrainte est satisfaite si il existe dans la matrice en cours de construction une ligne l telle que M[l][i] = a et M[l][j] = b.

# II – Opérateurs de variation

## A – Description de l’opérateur de mutation

L’opérateur de mutation implémenté travaille sur les symboles présents dans la solution. Pour faire muter un individu, il suffit de sélectionner au hasard quelques coefficients de la matrice puis de le remplacer par des symboles différents tirés aléatoirement.

La mutation doit garder un faible impact sur la qualité de la solution, c’est pourquoi le nombre de symbole modifié ne peut être trop important. De ce fait, on choisit lors d’une phase de mutation de modifier 1% des symboles de la configuration.

## B – Description de l’opérateur de croisement de base

L’opérateur de croisement de base construit un enfant à partir de deux parents en transmettant les symboles de l’un ou l’autre individuellement. Concrètement, lors de la génération de l’enfant, pour chaque symbole, un tirage au sort décidera si le coefficient est hérité du premier ou du second parent.

Ce processus est explicité dans le pseudo-code suivant :

* POUR i entre 1 et N
  + POUR j entre 1 et k
    - a = parent1[i][j]
    - b = parent2[i][j]
    - Affectation c = a ou c = b avec une probabilité de 50%
    - enfant[i][j] = c
  + FIN POUR
* FIN POUR

## C – Description de l’opérateur de croisement spécialisé

L’opérateur de croisement spécialisé s’adapte aux caractéristiques du problème et à la structure des données puisqu’il travaille sur les lignes des configurations, en évaluant leur performance au sens de la fonction de coût. En d’autres termes, il va s’agir d’un opérateur de croisement heuristique.

Comme précédemment, cet opérateur utilise deux parents pour générer un enfant unique. La construction de l’enfant se fait alors ligne par ligne. La première ligne est choisie aléatoirement en provenance de l’un ou l’autre des deux parents. Par la suite, pour chaque ligne, on sélectionne celle entre les deux parents qui résout le plus de nouvelles contraintes élémentaire.

La construction de l’enfant suit donc une logique gloutonne. Mais contrairement à l’algorithme glouton développé dans le premier laboratoire, la solution est ici construite ligne par ligne, avec un choix entre deux attributs différents uniquement. Cette implémentation répond au pseudo-code suivant :

* Générer solution vide enfant
* Choisir aléatoirement (uniformément) parentX = parent1 ou parent2
* enfant[1][\*] = parentX[1][\*] (M[i][\*] désigne toute la ligne i de la matrice M)
* POUR l entre 2 et N
  + candidat1 = parent1[l][\*]
  + candidat2 = parent2[l][\*]
  + enfant[l][\*] = candidat1
  + cout1 = verifierSolution(enfant)
  + enfant[l][\*] = candidat2
  + cout2 = verifierSolution(enfant)
  + SI cout1 < cout2
    - enfant[l][\*] = candidat1
  + FIN SI
* FIN POUR

# III – Description de l’algorithme évolutionniste

## A – Pseudo-code

Blabla

## B – Caractéristiques

L’algorithme génétique proposé se rapproche du schéma d’évolution (μ+λ)-ES. Un ensemble de test préliminaires (détaillés ci-après) a permis de fixer le nombre de parents m=100 et le nombre d’enfants n=200. En effet, puisque l’implémentation donnée..

## C – Liste des paramètres

Blabla

# IV – Résultats expérimentaux

## A – Présentation des données utilisées

Les données utilisées à des fins de test sont constituées de 7 exemplaires, chacun correspondant à un couple (v,k) différent. Les cas traités par la suite sont les suivants :

* v = 2, k = 4 - au mieux, avec l’algorithme tabou, on a pu avoir N = 5.
* v = 3, k = 20 - au mieux, avec l’algorithme tabou, on a pu avoir N = 17.
* v = 3, k = 60 - au mieux, avec l’algorithme tabou, on a pu avoir N = 22.
* v = 5, k = 10 - au mieux, avec l’algorithme tabou, on a pu avoir N = 37.
* v = 5, k = 15 - au mieux, avec l’algorithme tabou, on a pu avoir N = 43.
* v = 8, k = 10 - au mieux, avec l’algorithme tabou, on a pu avoir N = 94.
* v = 8, k = 15 - au mieux, avec l’algorithme tabou, on a pu avoir N = 109.

## B – Spécifications de la machine de test

Coefficient trouvé par dfmax : 5.40

Coefficient d'ajustement : 8.6/5.4 = 1.59

|  |  |
| --- | --- |
| **Système d’exploitation** | Ubuntu 13.10 64 bits |
| **Mémoire vive** | 7.7 GiB |
| **Processeur** | Intel® CoreTM i7-3610QM CPU @ 2.30GHz × 8 |

## C – Présentation des tests préliminaires

Blabla

## D – Résultats obtenus

Blabla

## E – Commentaires

Blabla