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 pseudocode 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 pseudocode 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

## D – Evaluation de la diversité

Afin de mesurer la diversité et son évolution, à titre indicatif, au sein d’une population d’individu, il a été nécessaire de définir la notion de distance entre deux configurations. Puisque ce concept n’a rien d’évident concernant les solutions du problème de covering array, il a fallu tâcher d’implémenter quelque chose d’arbitraire mais intuitif.

De façon assez basique, la distance entre deux configurations correspond au nombre de symboles différents aux mêmes emplacements. La diversité d’une population est alors la somme des distances entre tous les individus pris deux à deux.

Cette fonction est décrite pour une population « pop » de taille M par le pseudocode suivant :

* d = 0
* POUR i entre 0 et M-2
  + POUR j entre i+1 et M-1
    - POUR c entre 0 et k\*N-1
      * SI pop[i]->matrice[c] != pop[j]->matrice[c]
        + d++
      * FIN SI
    - FIN POUR
  + FIN POUR
* FIN POUR
* RETOURNER d

# III – Description de l’algorithme évolutionniste

## A – Pseudocode

* Créer le tableau population de taille tailleParents+tailleEnfants
* POUR i entre 0 et tailleParents-1
  + population[i] = configurationAleatoire(v, k, N)
* FIN POUR
* meilleureSolution = population[0]
* coutMeilleure = verifierSolution(meilleureSolution)
* TANT QUE ((coutMeilleure > 0) ET (tempsCalcul < 60s))
  + POUR e entre 0 et tailleEnfants-1
    - Choisir aléatoirement parent1 dans la population parmi les parents admissibles (tailleParents premiers indices)
    - Choisir aléatoirement parent2 dans la population parmi les parents admissibles (tailleParents premiers indices)
    - Générer l’enfant suite au croisement de parent1 et parent2
    - Appliquer l’opérateur de mutation à l’enfant
    - population[tailleParents+e] = enfant
  + FIN POUR
  + POUR i entre 0 et tailleParents+tailleEnfants-1
    - couts[i] = verifierSolution(population[i])
  + FIN POUR
  + Trier population et couts par ordre croissant de coût (les tailleParents premiers sont les meilleures configurations et pourront être parents à l’itération suivante)
  + Calculer la diversité de la population
  + meilleureSolution = population[0]
  + coutMeilleure = couts[0]
* FIN TANT QUE
* RETOURNER meilleureSolution

## 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=20 et le nombre d’enfants n=20. En effet, puisque l’implémentation donnée est purement génétique, et n’utilise donc pas d’opérateur de recherche locale, il est possible et préférable, dans l’optique d’obtenir de meilleurs résultats, de travailler sur des populations assez élargies.

Lors d’une itération génétique, les parents admissibles pour générer la population suivante sont les m meilleures configurations, c’est-à-dire les individus violant le moins de contraintes élémentaires. On génère alors n enfants par croisement de parents sélectionnés aléatoirement deux à deux avant de faire muter les enfants sur 1% de leurs symboles. La population subsistant à cette étape est alors la réunion des parents et des enfants, ce qui implique que la technique implémentée utilise le recouvrement.

La diversité de la population est alors calculée à titre indicatif, mais on ne cherche pas particulièrement à la favoriser, d’où la non utilisation d’une technique de fitness sharing par exemple lors de la sélection des individus survivant.

L’algorithme implémenté n’impose pas de forte pression sélective. Certes, seuls m individus parmi les m+n configurations de la population peuvent prétendre à être parents pour la création de la génération suivante, mais le tirage des parents pour un croisement est totalement uniforme lors de cette étape.

## C – Liste des paramètres

* v : nombre de symboles admissibles dans la matrice de configuration
* k : nombre de colonnes de la matrice
* N : nombre de lignes de la matrice
* tailleParents : nombre de parents sélectionnés dans une population
* tailleEnfants : nombre d’enfants générés lors du croisement
* pourcentMutation : pourcentage du nombre de symboles changeant lors d’une mutation

# 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