INF6102 – Métaheuristiques
Travaux pratiques n°2 – 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 – Stratégie de recherche locale

A – Caractéristiques de l'approche de recherche locale

L'approche de recherche locale employée dans ce laboratoire est caractérisée par les éléments suivants :

Configuration	Une configuration S est une matrice N*k contenant des symboles admissibles, à savoir les entiers entre 0 et v-1.
Fonction d'évaluation	La fonction f(S), qui renvoie le nombre de contraintes élémentaires non satisfaites par la configuration S. L'objectif est de minimiser f(S).
Mouvement	Un mouvement consiste à remplacer le symbole a contenu dans la matrice à la ligne l et à la colonne c par un nouveau symbole b.

B – Discussion de l'approche de recherche locale

Dans le problème de recherche locale, on fixe le nombre de lignes de la matrice à N, et on part d'une solution aléatoire ne satisfaisant probablement pas toutes les contraintes. Une configuration est donc nécessairement une matrice de taille N*k contenant les symboles admissibles, et ce à chaque tour de boucle de l'algorithme puisque N est fixé.

Puisque la taille de la matrice est fixée, N ne peut pas être le résultat de la fonction d'évaluation, car on ne complète pas la solution en y rajoutant des lignes. De ce fait, on s'expose à trouver une solution ne satisfaisant pas toutes les contraintes élémentaires, le nombre de lignes de la matrice pouvant être insuffisant pour cela. Ainsi, la fonction d'évaluation d'une configuration est tout simplement le nombre de contraintes élémentaires non satisfaites, qu'il convient donc de minimiser.

Enfin, la fonction de voisinage prend en entrée une configuration et la renvoie en ayant modifié un coefficient uniquement par un autre symbole admissible. La taille de la matrice étant fixe, la seule façon d'affecter une configuration est de changer des symboles. Or, l'opération affectant le moins possible la matrice est de remplacer un seul symbole. Voilà pourquoi le voisin d'une configuration est la même configuration, à un symbole près.

C – Optimum local non global

Une exécution de l'algorithme de descente avec les paramètres v=3, k=20 et pour N=15 renvoie la matrice suivante :

2	2	0	2	2	2	0	0	2	1	2	2	2	1	1	0	0	0	1	0
0	1	1	1	1	2	1	1	2	2	0	2	0	1	2	2	1	0	2	1
1	2	2	2	1	1	1	0	2	0	0	0	2	0	0	2	2	1	1	2
1	1	2	2	0	0	0	2	0	0	1	2	0	0	2	0	1	2	1	0
2	0	2	1	0	0	1	2	0	1	2	0	1	2	1	0	2	0	2	1
0	1	0	1	2	1	0	2	0	1	1	1	2	1	0	2	0	1	2	2
0	2	1	1	2	1	1	0	0	2	0	1	1	2	2	0	2	2	0	0
2	0	1	0	0	1	0	2	1	2	0	2	2	2	1	1	1	1	0	2
2	2	0	2	0	0	2	1	2	1	0	1	1	2	0	2	1	2	1	2
0	0	1	2	0	2	0	0	1	2	1	0	1	1	0	1	0	2	1	1
2	0	0	0	1	2	2	2	2	1	1	0	0	0	2	1	2	0	0	2
1	0	1	1	2	1	2	1	1	0	2	1	2	0	2	2	0	0	0	1
1	2	2	0	1	2	1	1	0	0	1	1	0	2	1	1	0	1	2	1
0	1	2	0	2	0	2	0	1	2	2	0	0	0	1	1	1	2	2	0
1	1	0	0	1	0	2	1	1	0	2	2	1	1	0	0	2	1	0	0

Cette configuration a un coût égal à 43 en termes d'erreurs, c'est-à-dire qu'elle est responsable de la violation de 43 contraintes. Or, il existe une configuration avec ces mêmes paramètres pour laquelle toutes les contraintes élémentaires sont satisfaites :

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	0	1	2	2	2	2	2	2	2	2	2	2	2
0	1	1	1	1	0	1	2	2	0	0	0	0	1	1	1	2	2	2	2
0	2	2	2	2	2	2	0	1	0	0	0	0	1	2	2	0	1	1	1
1	0	1	1	1	2	2	0	1	0	1	1	2	0	0	1	1	0	1	2
1	1	2	2	2	1	0	1	0	2	1	1	0	0	2	1	2	2	1	0
1	2	0	1	2	0	2	1	0	2	0	2	2	1	0	2	1	0	2	1
1	2	1	0	2	2	1	2	0	1	2	1	1	0	1	2	0	2	0	1
1	2	1	2	0	2	1	1	2	2	1	0	1	2	0	0	2	1	0	0
2	0	2	2	2	0	1	2	2	1	2	2	0	2	2	0	1	0	1	2
2	1	0	2	1	2	0	2	2	2	1	2	2	0	1	2	0	1	2	0
2	1	2	0	1	1	2	0	2	1	0	1	1	2	1	0	2	0	2	1
2	1	2	1	0	1	2	2	1	1	2	0	2	1	0	0	1	2	0	0
2	2	1	1	1	1	0	1	0	0	2	2	1	2	2	1	0	1	0	2

De ce fait, l'optimum local exhibé par l'algorithme de descente est différent de l'optimum global pour ce problème, qui est représenté par la configuration ci-dessus.

III – Description de l'algorithme

A – Description de haut niveau

Paramètres de l'algorithme

- S : configuration initiale N*k générée aléatoirement au préalable
- T : la température initiale de la procédure de recuit
- alpha : coefficient de décroissance de la température

Description

- Fixer critère d'arrêt
- TANT QUE (critère d'arrêt non atteint)
    - Générer S', un voisin de S
    - Calculer delta = f(S') - f(S)
    - Calculer le critère de Métropolis
    - SI (metropolis)
        - S = S'
        - SI (meilleureConfiguration)
            - meilleurCoût = f(S')
        - FIN SI
    - FIN SI
    - Mettre à jour T = alpha * T
- FIN TANT QUE
- RETOURNER meilleureConfiguration

B – Implémentation

La performance d'un mouvement d'une configuration S vers une configuration S' est donnée par la différence entre f(S') et f(S), où f est la fonction d'évaluation. Plus précisément, f correspond à la fonction "verifierSolution(Mouvement)" dans le code, qui renvoie le nombre de contraintes élémentaires violées par la configuration suite à un mouvement. Dans le principe, l'objectif est de vérifier si un mouvement a introduit des violations de contraintes supplémentaires, ou si au contraire il a pu en satisfaire de nouvelles. Une CA_Solution est en effet caractérisée, notamment, par l'ensemble des contraintes qu'elle a à satisfaire, chacune étant associée à un booléen de satisfaction. Un mouvement est caractérisé par la ligne et la colonne concernées, ainsi que les ancien et nouveau symboles à cet emplacement. En sauvegardant l'état des contraintes avant un mouvement, on est donc en mesure d'effectuer une comparaison suite à celui-ci. Une implémentation bas-niveau de la fonction "verifierSolution(Mouvement)" répond donc au pseudocode suivant :

verifierSolution(CA_Solution sol, Mouvement mv)

- erreursDernierMv = sol.erreurs
- POUR chaque colonne k1
    - POUR chaque symbole v1

        - copier l'état de la contrainte (k1,mv.colonne,v1,mv.ancienSymbole) dans copieContrainteAncien[k1][v1]
        - copier l'état de la contrainte (k1,mv.colonne,v1,mv.nouveauSymbole) dans copieContrainteNouveau[k1][v1]
    - FIN POUR
- FIN POUR
- POUR chaque colonne k1
    - SI (copieContrainteAncien[k1][sol[mv.ligne][k1]] ET k1 != mv.ligne)
        - copieContraintesAncien[i1][sol[mv.ligne][k1]] = false
        - erreursDernierMv++
    - FIN SI
- FIN POUR
- POUR chaque ligne l
    - SI (sol[l][mv.colonne] = mv.ancienSymbole ET l != mv.ligne)
        - POUR chaque colonne k1
            - SI (!copieContraintesAncien[k1][sol[l][k1]] E Tk1 != mv.colonne
                - copieContraintesAncien[k1][sol[l][k1]] = true
                - erreursDernierMv—
            - FIN SI
        - FIN POUR
    - FIN SI
- FIN POUR
- POUR chaque colonne k1
    - SI (!copieContraintesNouveau[k1][sol[mv.ligne][k1]] ET k1 != mv.colonne)
        - copieContraintesNouveau[k1][sol[mv.ligne][k1]] = true
        - erreursDernierMv--;
    - FIN SI
- FIN POUR
- RETOURNER erreursDernierMv

L'impact d'un mouvement de S vers S' sur la fonction d'évaluation vaut donc : d(.) = verifierSolution(S') - verifierSolution(S).

La complexité de la fonction de vérification d'une configuration se calcule en observant le nombre passages dans les boucles imbriquées dans le pseudocode précédent :

- POUR chaque colonne k1 (k passages)
    - POUR chaque symbole v1 (v passages)
    - FIN POUR
- FIN POUR
- POUR chaque colonne k1 (k passages)
- FIN POUR
- POUR chaque ligne l (N passages)

    - POUR chaque colonne k1 (k passages)
    - FIN POUR
- FIN POUR
- POUR chaque colonne k1 (k passages)
- FIN POUR

Si l'on tient compte de l'imbrication des boucles, on se retrouve avec quatre blocs indépendants. Le premier est de complexité O(kv), le deuxième en O(k), le troisième en O(Nk) et le quatrième en O(k). Finalement, l'ensemble de la fonction a donc une complexité en O(k*(v+N)).

III – Expériences

A – Données utilisées

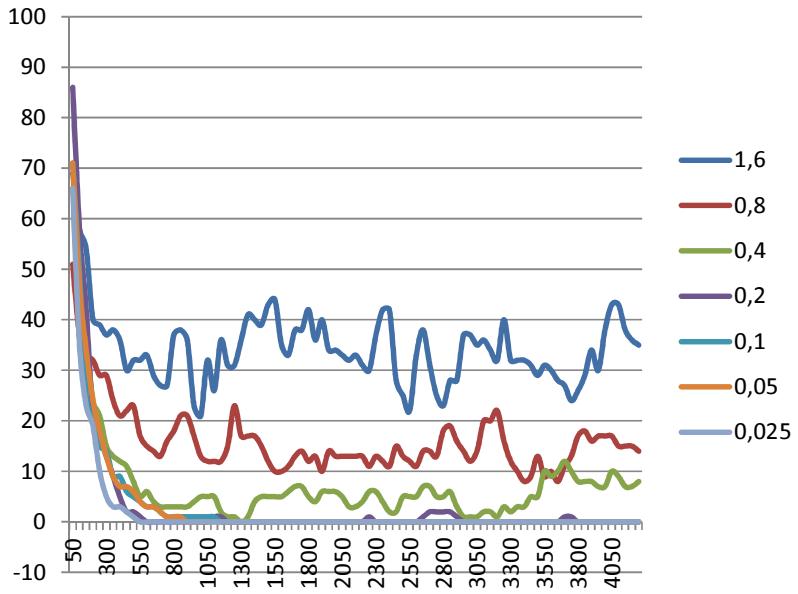
Les données utilisées à des fins de test sont constituées de 7 exemplaires, chacun correspondant à un couple (v,k) différent. Les cas traités par la suite sont les suivants :

- v = 2, k = 4 – au mieux, avec l'algorithme glouton, on a pu avoir N = 6.
- v = 3, k = 20 - au mieux, avec l'algorithme glouton, on a pu avoir N = 21.
- v = 3, k = 60 - au mieux, avec l'algorithme glouton, on a pu avoir N = 27.
- v = 5, k = 10 - au mieux, avec l'algorithme glouton, on a pu avoir N = 47.
- v = 5, k = 15 - au mieux, avec l'algorithme glouton, on a pu avoir N = 54.
- v = 8, k = 10 - au mieux, avec l'algorithme glouton, on a pu avoir N = 117.
- v = 8, k = 15 - au mieux, avec l'algorithme glouton, on a pu avoir N = 120.

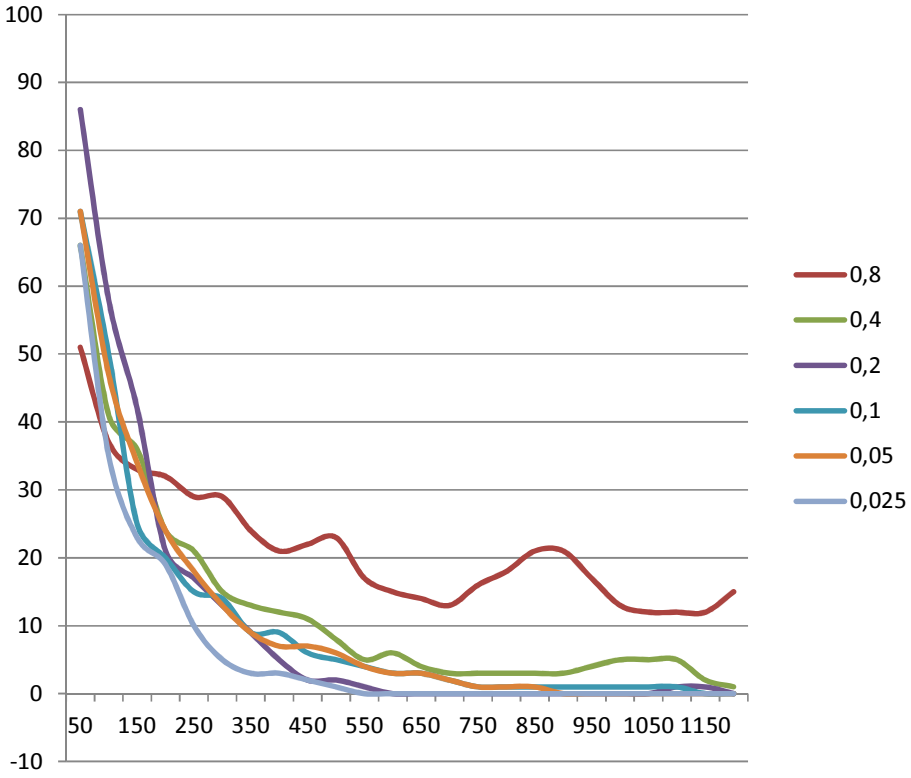
B – Calibrage des paramètres

Pour régler les paramètres du recuit simulé (température initiale, coefficient de décroissance), on commence par exécuter l'algorithme à température constante avec les sept valeurs suivantes : 3,2 ; 1,6 ; 0,8 ; 0,4 ; 0,2 ; 0,1 ; 0,05 ; 0,025.

Pour déterminer la température initiale de l'algorithme, on analyse, pour chacune des températures précédentes, la courbe de fmin en fonction du nombre d'itérations à température constante. Le résultat obtenu est le suivant :



Afin de mieux visualiser le comportement des courbes les plus basses, on effectue un zoom sur le coin inférieur gauche du graphique :



En définitive, les courbes fmin = f(itération) les plus basses sont caractérisées par des températures T=0,1 et T=0,025. On choisit donc dans notre algorithme de recuit simulé d'initialiser la température à T0 = 0,1, et on utilisera un schéma de refroidissement à décroissance géométrique par paliers avec un facteur de décroissance de ½.

Plus précisément, on partira donc d'une température égale à 0,1. Puis, au bout de 1000 itérations sans que la meilleure solution trouvée jusqu'alors n'ait été modifiée, on appliquera la division par deux de la température du recuit. L'algorithme se termine au bout de 10 millions d'itérations au total, ce qui représente environ une minute d'exécution pour les exemples les plus difficiles du panel. L'algorithme termine bien entendu avant si une configuration résolvant toutes les contraintes est trouvée.

Afin de déterminer le nombre de lignes N avec lequel l'algorithme de recuit simulé va travailler, nous avons effectué une expérimentation préalable. Grâce à la fonction « trouverMeilleure(...) », qui lance l'algorithme sur des tailles de plus en plus petites de matrice, nous déterminons à l'avance le nombre de lignes qu'il est raisonnable d'espérer de notre algorithme. Pour avoir des temps de recherche similaire, nous avons pris un nombre de lignes inférieur à cette valeur prédéterminée, car ainsi l'algorithme continuera son exécution jusqu'à ce que le critère d'arrêt soit valide (et pas quand il a trouvé une matrice valide). Ainsi, nous avons utilisé les valeurs suivantes pour nos tests :

- v = 2, k = 4, N = 4.
- v = 3, k = 20, N = 17.
- v = 3, k = 60, N = 22.
- v = 5, k = 10, N = 38.
- v = 5, k = 15, N = 43.
- v = 8, k = 10, N = 97.
- v = 8, k = 15, N = 110.

Autrement dit, notre algorithme a permis d'obtenir ces mêmes valeurs augmentées de 1 : cela constitue déjà une grande amélioration par rapport au glouton.

C – Résultats

Le tableau suivant récapitule les différents résultats obtenus pour chacun des exemples considérés. On a également consigné le nombre de contraintes non satisfaites subsistant en moyenne, le nombre total d'itérations moyen, le nombre moyen de vrais mouvements (c'est-à-dire améliorant la solution en cours), et le temps moyen d'exécution (ajusté pour être comparable avec la machine de référence). Ces quantités sont disponibles en valeur moyenne, minimum et maximum.

Résultats pour le recuit simulé (T0 = 0,1)

v	k	N	CoutMin	CoutMoyen	CoutMax	ItMin	ItMoyen	ItMax	MvMin	MvMoyen	MvMax	TempsMin	TempsMoyen	TempsMax
2	4	4	2	2.2	4	1.00E+007	1.00E+007	1.00E+007	6	8.7	15	1324.33	2153.89	1368.4
3	20	17	1	3.4	6	1.00E+007	1.01E+007	1.02E+007	1063	41090.7	94163	6308.73	10192.1	6522.14
3	60	22	2	5.6	10	1.01E+007	1.12E+007	1.45E+007	1.30E+004	6.19E+004	9.73E+004	19951.1	34848.5	28497.4
5	10	38	0	2.7	6	533594	9.36E+006	1.22E+007	1025	20669.3	4.34E+004	266.387	7096.66	5767.95
5	15	43	6	9.7	19	1.01E+007	1.08E+007	1.25E+007	5.85E+003	2.05E+004	5.23E+004	6777.76	11507	8408.99
8	10	97	0	0.1	1	2.55E+005	2.09E+006	1.06E+007	3.02E+003	9.12E+003	3.37E+004	201.631	2349.29	7360.58
8	15	110	1	5.5	14	1.12E+007	1.60E+007	2.59E+007	1.33E+004	2.78E+004	4.19E+004	11116.9	25168.6	25592.6

Résultats pour la descente (T = 0)

v	k	N	CoutMin	CoutMoyen	CoutMax	ItMin	ItMoyen	ItMax	MvMin	MvMoyen	MvMax	TempsMin	TempsMoyen	TempsMax
2	4	4	2	2.8	4	1.00E+007	1.00E+007	1.00E+007	3	5.8	11	1316.75	2124.95	1362.37
3	20	17	0	4.6	7	4.39E+004	9.05E+006	1.01E+007	868	29804.5	79454	36.8686	9121.09	6424.79
3	60	22	2	7	14	1.01E+007	1.13E+007	1.60E+007	3.05E+004	4.81E+004	8.90E+004	19580.4	34954.4	30707.3
5	10	38	0	2.6	8	2.17E+006	9.34E+006	1.04E+007	828	24876.4	7.75E+004	1021.11	7020.65	4921.91
5	15	43	4	7.7	13	1.01E+007	1.08E+007	1.29E+007	5.73E+003	1.83E+004	4.63E+004	6759.49	11476.6	8559.63
8	10	97	0	0.8	2	3.32E+005	4.64E+006	1.10E+007	2.47E+003	6.12E+003	1.65E+004	251.553	5122.84	7567
8	15	110	1	7.6	18	1.13E+007	1.53E+007	2.41E+007	9.40E+003	2.17E+004	4.15E+004	11175.7	23974.5	23698.8

D – Commentaires

En prenant un nombre de lignes inférieur à celui trouvé par le glouton, nous pouvons voir si l'algorithme considéré trouve de meilleures solutions. Si le coût minimum pour une combinaison de paramètres est nul, cela signifie que l'algorithme a trouvé une configuration qui respecte l'intégralité des contraintes : on a donc généré une matrice valide ayant un nombre de lignes inférieur à celle générée par le glouton.

Dans ce contexte, nous observons que l'algorithme de recuit simulé trouve des matrices valides pour 2 combinaisons (v, k, N) : (5, 10, 38) et (8, 10, 97). Pour les autres configurations, aucune solution valable n'est trouvée. Nous remarquons tout de même que les coûts des solutions trouvées sont proches de 0 : c'est bien normal, puisque nous avons choisi le nombre de lignes comme étant le plus petit ne donnant pas 0.

De plus, les coûts moyens sont meilleurs pour le recuit simulé que pour la descente, mais les coûts les plus bas sont atteints pour la descente. On pouvait s'y attendre, puisque la descente a tendance à s'enfermer dans des optima locaux, tandis que le recuit simulé oscille plus mais permet des relances.

E – 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