INF6102 – Métaheuristiques
Travaux pratiques n°3 – 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. C'est nécessairement une colonne contenant encore des contraintes non résolues.

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 – Implémentation de la recherche locale

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 : verifierSolution(S, mv) = 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 – Description de l'algorithme tabou

A – Description de la liste taboue

Le liste tabou implémentée dans l'algorithme interdit des états de la matrice, et non pas un mouvement, pendant un certain nombre d'itérations.

Cette liste se présente sous la forme d'une matrice cubique dont les dimensions sont N pour le nombre de lignes de la matrice, k pour le nombre de colonnes, et v pour le nombre de symboles possibles. En considérant que chaque itération de l'algorithme est numérotée, un coefficient i tel que Tabou[l][c][w] = i traduit le fait que le symbole w est interdit dans la solution à la ligne l, colonne c, jusqu'à l'itération i.

En d'autres termes, lors d'un mouvement d'une configuration contenant un symbole a à la ligne l, colonne j, vers une configuration contenant un symbole b à ces mêmes coordonnées, on va bannir l'état que l'on vient de quitter. C'est-à-dire que si ce mouvement a lieu à l'itération j, et que l'on veut bannir l'état initial pendant q itérations, on verra dans la liste taboue : Tabou[l][c][a] = j+q.

B – Description de haut niveau de l'algorithme tabou

Paramètres

- **CA_Solution* configInit** : la configuration aléatoire de laquelle on part pour construire une solution valide avec l'algorithme tabou.
- **int longueurListe** : nombre d'itérations pendant lesquelles un mouvement est placé sur la liste taboue.

Algorithme

- Déclaration de **configTestee** (configuration en cours de test) et de **meilleureConfig** (meilleure des configurations testées jusqu'ici)

- Initialisation de la **listeTtaboue** : création d'un tableau à trois dimensions de taille N*k*v où N est le nombre de lignes de la matrice de configuration, k le nombre de colonnes, et v le nombre de symboles admissibles.
- Pour tout i1 dans [0 ;N-1], i2 dans [0 ;k-1], i3 dans [0 ;v-1], **listeTaboue[i1][i2][i3]** = 0 : aucun mouvement n'est encore interdit dans la liste.
- TANT QUE **tempsExecution** < 60s ET **meilleureConfig** a un coût **coutMeilleure** non nul
    - TANT QUE on n'a pas testé tous les mouvements possibles
        - Sélection du **mouvement** à tester (choisi selon un ordre lexicographique des mouvements possibles)
        - Vérification de **coutTest**, le coût de la configuration à laquelle le **mouvement** a été appliqué
        - Calcul de **delta = coutTest – coutActuelle**
        - SI **mouvement** n'est pas tabou pour l'itération en cours, OU si le **mouvement** testé est meilleur que les précédents (**delta** est minimisé)
            - SI le **mouvement** testé est strictement meilleur que les précédents
                - Remettre à zéro la liste des meilleurs mouvements
            - FIN SI
            - Ajouter le **mouvement** à la liste des meilleurs mouvements
        - FIN SI
    - FIN TANT QUE
    - Choix du **meilleurMouvement**, ayant pour coût **coutMin**, parmi la liste des meilleurs mouvements selon un tirage au sort à distribution uniforme. Disons que ce mouvement a pour origine **Matrice[l][c]** = a et rend **Matrice[l][c]** = b.
    - Application du **meilleurMouvement** à la **configTestee**
    - **listeTaboue[l][c][a]** = **numIteration** + **longueurListe**
    - SI **coutMin** < **coutMeilleure**
        - **meilleureConfig** = **configTestee**
        - **coutMeilleure** = **coutMin**
    - FIN SI
    - **numIteration++**
- FIN TANT QUE
- RETOURNER **meilleureConfig**

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 de recuit simulé, on a pu avoir N = 5.
- v = 3, k = 20 - au mieux, avec l'algorithme de recuit simulé, on a pu avoir N = 18.
- v = 3, k = 60 - au mieux, avec l'algorithme de recuit simulé, on a pu avoir N = 23.
- v = 5, k = 10 - au mieux, avec l'algorithme de recuit simulé, on a pu avoir N = 38.
- v = 5, k = 15 - au mieux, avec l'algorithme de recuit simulé, on a pu avoir N = 44.
- v = 8, k = 10 - au mieux, avec l'algorithme de recuit simulé, on a pu avoir N = 97.
- v = 8, k = 15 - au mieux, avec l'algorithme de recuit simulé, on a pu avoir N = 111

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

L'exécution de l'algorithme tabou dépend de deux paramètres à fixer au préalable. D'une part le nombre d'itérations dans l'algorithme, et d'autre part la longueur de la liste taboue.

Plutôt que de donner un nombre d'itérations à effectuer au maximum, il a été préférable de travailler sur les temps de calcul. Ainsi, pour chaque exécution de l'algorithme, une minute (en temps standardisé) au maximum sera accordée à la résolution du problème. Bien entendu, pour un N donné, si l'algorithme tabou parvient à trouver une couverture valable en moins d'une minute, le calcul termine avant soixante secondes.

Par ailleurs, la longueur de la liste taboue a été déterminée grâce à quelques expériences préliminaires qui ont permis, pour chaque exemplaire du problème, de trouver une valeur favorisant l'obtention de meilleurs résultats. La longueur de la liste taboue n'est pas absolue : pour chaque exemplaire du problème, sa valeur optimale sera différente.

Sans entrer dans les détails quant à l'obtention de ces meilleures valeurs, voici les longueurs de liste taboue qui ont été retenues pour chacun des problèmes :

v	k	Longueur liste taboue
2	4	5
3	20	20
3	60	30
5	10	15
5	15	30
8	10	30
8	15	40

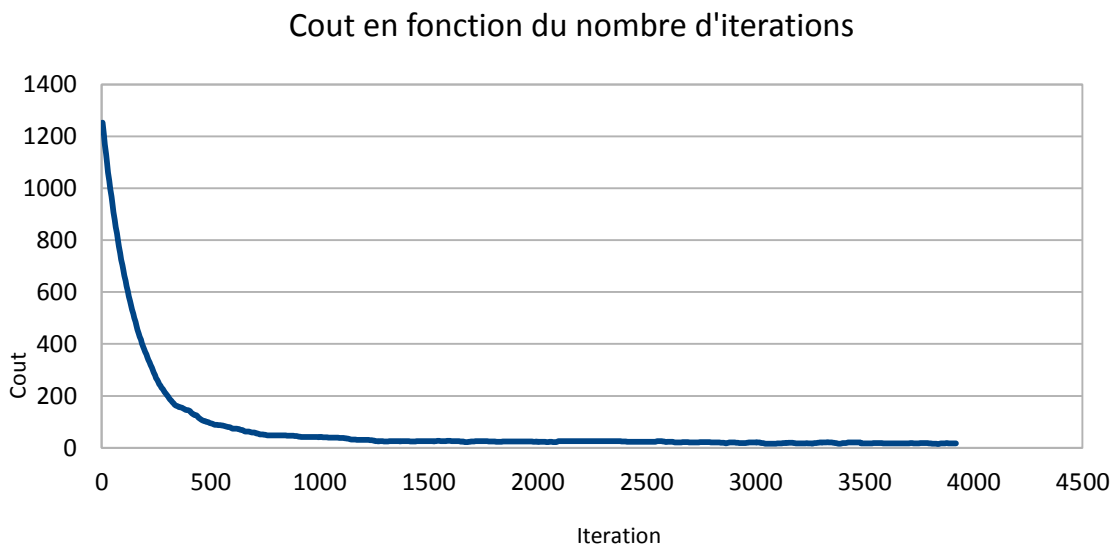
D – Résultats obtenus

Les résultats présentés font état, pour chaque exemplaire du problème, du coût de la solution (nombre d'erreurs subsistant pour le premier N tel que la solution n'est pas valide), le nombre d'itérations de l'algorithme, et le temps d'exécution. Une exécution de l'algorithme ne parvenant pas à trouver de solution satisfaisante durera toujours une minute, puisqu'il s'agit là d'un des critères d'arrêt du calcul. Cependant, il peut arriver qu'une solution valable soit exposée, réduisant ainsi le temps d'exécution de l'algorithme à moins d'une minute.

Pour chacune de ces trois grandeurs, les valeurs minimale, moyenne, et maximale sur 10 exécutions sont données.

			Coût			Nombre d'itérations			Temps (s)		
v	k	N	Min	Moy	Max	Min	Moy	Max	Min	Moy	Max
2	4	4	2	2	2	1,76E+007	1,77E+007	1,78E+007	60	60	60
3	20	16	4	6,8	8	105780	107742	108796	60	60	60
3	60	21	18	20,9	24	8841	8936,4	9007	60	60	60
5	10	36	5	6	7	67521	67717,7	68143	60	60	60
5	15	43	0	2,2	3	15445	24125,7	25267	36,9	57,7	60
8	10	94	0	0,8	2	3502	7749,5	10356	20,3	45	60
8	15	108	9	13,9	19	4008	4019,7	4026	60	60	60

Par ailleurs, on livre le profil de la courbe de l'évolution du coût en fonction du nombre d'itérations effectuées. La courbe ci-dessous reflète donc le profil d'exécution du programme pour l'exemple v=8, k=15 :



E – Commentaires

Tous les résultats obtenus révèlent de meilleures performances que ceux de l'algorithme de recuit simulé. Par ailleurs, alors que la conception du recuit simulé donnait lieu à des temps d'exécution variables, la présente implémentation de l'algorithme tabou garantit l'obtention de résultats en une minute maximum. Le recuit simulé était certes plus rapide en général, ce qui peut laisser à penser que la comparaison des résultats est légèrement biaisée. Cependant, il paraît plus pertinent de commencer à parler de temps de calcul plutôt que de nombre d'itérations qui ne constitue pas une grandeur analysable et comparable dans l'absolu.

Par ailleurs, le profil d'exécution exposé montre que la chute du coût en fonction du nombre d'itérations est essentiellement supportée lors du premier dixième.

V – Techniques de diversification

1 – Présentation

La technique de diversification retenue consiste en une approche continue. Dans le principe, sachant qu'une exécution de l'algorithme tabou est supposée prendre une durée de l'ordre de la minute, on fixe deux seuils de temps K et K' régissant la répartition des phases de diversification.

Pendant K secondes, l'algorithme tabou s'exécute normalement, après quoi une phase de diversification de K' prend le relais. Ces deux phases s'enchaînent à tour de rôle jusqu'à la fin de l'algorithme. En pratique, on a choisi de fixer K à 9 secondes, et K' à 1 seconde. Cela signifie qu'au début de l'algorithme, la phase taboue classique s'exécutera pendant 9

secondes, avant de laisser sa place à 1 seconde de diversification. Ce cycle s'exécutera alors au maximum 6 fois, étant donné que le temps de calcul maximum autorisé est d'une minute.

Lors d'une phase de diversification, la fonction de calcul du delta de coût d'un mouvement testé est modifiée. En effet, on prend désormais en compte la fréquence d'utilisation du nouveau symbole dans la configuration en cours dans l'optique de favoriser les symboles les moins présents. De cette façon, on espère maximiser les chances de voir apparaître un symbole au bon endroit pour résoudre un maximum de nouvelles contraintes.

Plus concrètement, la phase de diversification fait intervenir trois tableaux de mémoire à long terme concernant la présence de chaque symbole, leur dernière présence, ainsi que leur fréquence d'apparition. La fonction *choixDiversification* prend alors en compte ces trois paramètres ainsi que la configuration testée pour ajuster la liste des meilleurs mouvements.

Les fréquences des nouveau et ancien symboles sont ajustées au sein de cette fonction et la fonction δ' d'un mouvement faisant intervenir un symbole de départ a vers un symbole d'arrivée b est donnée par :

$\delta'(a,b) = \delta(a,b) + h(b) - h(a)$; où $h(x)$ est la fréquence d'apparition du symbole x.

2 – Résultats

Les résultats de la diversification sont présentés de la même façon que précédemment, et pour les mêmes valeurs de N.

			Coût			Nombre d'itérations			Temps		
v	**k**	**N**	**Min**	**Moy**	**Max**	**Min**	**Moy**	**Max**	**Min**	**Moy**	**Max**
2	4	4	2	2	2	1,82E+007	1,86E+007	1,87E+007	60	60	60
3	20	16	6	7,3	8	105163	106544	108071	60	60	60
3	60	21	21	25	29	8484	8879,7	8982	60	60	60
5	10	36	5	6,2	7	67250	67397,5	67509	60	60	60
5	15	43	1	2,1	3	25137	25181,3	25225	60	60	60
8	10	94	0	2,2	4	7378	9923,5	10327	43	58,3	60
8	15	108	10	15,9	20	3915	3983,3	4010	60	60	60

Il se trouve qu'en moyenne, la phase de diversification détériore les solutions par rapport à l'algorithme tabou de base. En effet, d'une part, les soixante secondes de calcul sont requises pour beaucoup plus d'exemples que précédemment. D'autre part, le coût moyen des solutions trouvées est toujours légèrement supérieur à ceux obtenus lors des premières expériences.