Rapport du projet de métaheuristique

Couverture convexe minimum dans les réseaux de capteurs

Adrien BLASSIAU
Corentin JUVIGNY

MPRO - 5 novembre 2019

SOMMAIRE

1. Introduction

L'objectif du projet est de déterminer une couverture connexe minimum dans un réseau de capteurs. C'est un problème d'optimisation :

- une **instance du problème** correspond à une grille carrée de côté n sur laquelle chaque noeud est une cible à capter, excepté le noeud en haut à gauche qui est un puits.
- une **solution réalisable** du problème est une grille dans laquelle chaque capteur est relié par une chaîne de communication au puits et où les cibles sont toutes détectées par au moins un capteur.
- on **mesure** le nombre de capteurs placés sur la grille.
- l'**objectif** est de minimiser ce nombre de capteurs.

De nombreuses informations liées à l'installation de différents modules et à la construction des différents exécutables sont disponibles dans le fichier `README.md`, à la racine du projet. Le fichier `Makefile.options` permet de paramétrer ces exécutables.

2. Langage de programmation, structures de données et outils utilisés

2.1. Langage de programmation

Nous avons tout d'abord réalisé une ébauche du projet en **python** afin d'obtenir rapidement des résultats et tester les différentes méthodes que nous voulions mettre en place. Nous voulions un code optimisé et plus rapide en utilisant des mécanismes bas niveau, ce que ne permet pas le python. C'est pour cela que nous avons choisi le **C**.

2.2. Outils principaux utilisés

2.2.1 Doxygen

C'est un générateur de documentation sous licence libre qui peut notamment analyser des fichiers en langage C.
Toute la documentation du projet a été générée avec cet outil et est disponible dans le fichier `/doc/html/index.html` que nous vous invitons à ouvrir sur votre navigateur web favori.

2.2.2 CUnit

CUnit est une bibliothèque de tests unitaires pour C. Elle permet de programmer des tests, de les exécuter, et d'afficher un résumé des tests réussis ou échoués. Un test unitaire ne teste qu'une partie atomique des spécifications sous des conditions précises qui, bien généralement, ne couvrent pas tous les cas.
Les fonctions les plus critiques du projet ont été testées avec cet outil. Une partie du `README.md` est chargée de vous expliquer comment exécuter ces tests unitaires.

2.3. Structures de données utilisées

2.3.1 L'arbre k-d, pour une recherche rapide des voisins d'un point dans un rayon donné

Nous avons décidé de stocker les points dans un arbre binaire, l'**arbre k-d** (ou k-d tree en anglais), sur lequel nous avons pu effectuer des recherches des plus proches voisins dans un rayon donné, avec une complexité en $\mathcal{O}(\log n)$. Cela permet de ne plus avoir une complexité classique en $\mathcal{O}(n)$ qui correspond au parcours d'une liste de points.

Expliquons rapidement sa construction. L'arbre k-d est un arbre binaire assez simple à construire. Il repose sur le principe de partitionnement de l'espace. Ici, on travaille en deux dimensions donc l'espace est le plan. Chaque noeud de l'arbre contient un point. Chaque noeud non terminal divise l'espace en deux sous espaces. Ainsi les sous branches droite et gauche d'un noeud représentent deux demi espaces. Si le point d'un noeud divise l'espace selon l'axe (Ox) par exemple, il correspond à la médiane de l'ensemble des points dont il fait partie. Ainsi tous les points de coordonnée x inférieure à la coordonnée du point du noeud père se retrouve dans la sous branche gauche. Les autres se retrouvent dans la sous branche droite. Pour chaque noeud,

on alterne l'axe selon lequel la division se fait. L'exemple suivant montre une partitionnement de l'espace effectué sur un ensemble de 10 points et son stockage dans l'arbre (Figure 1 et Figure 2).

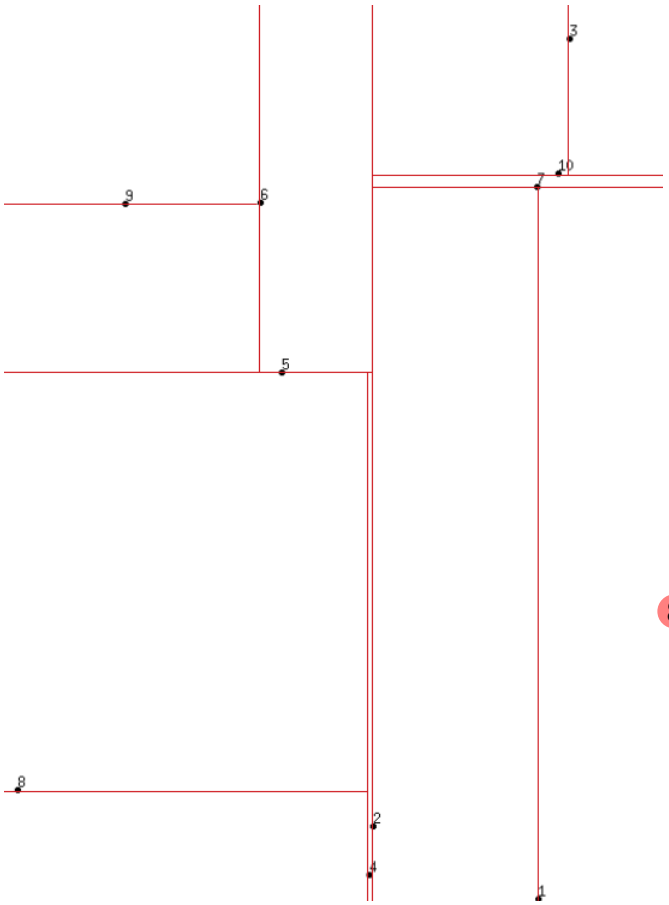

FIGURE 1 – Partitionnement de l'espace

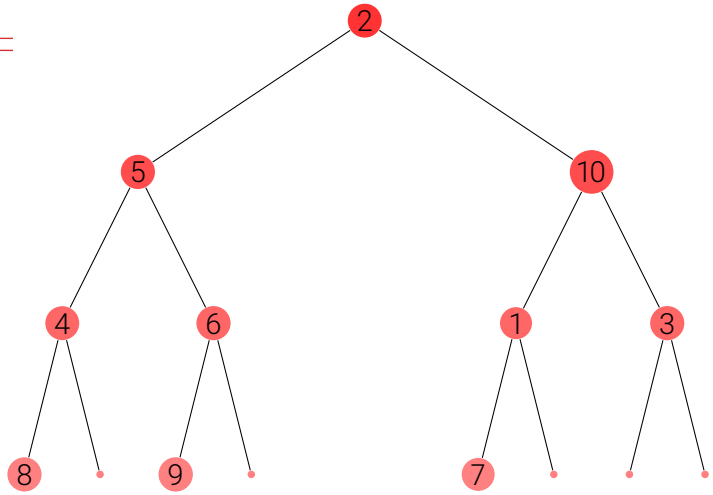

FIGURE 2 – Stockage dans l'arbre k-d

De par cette construction très particulière, l'accès aux voisins d'un point dans un cercle de rayon donné est très rapide.

2.3.2 Des AVL pour un accès et une modification rapide des données

Nous avons utilisés des **AVL** comme structure de données quand les opérations d'accès, d'ajout et de retrait étaient fréquentes. Dans cette structure d'arbre binaire de recherche équilibrée, ces opérations ont une complexité moyenne en $\mathcal{O}(\log n)$.

2.3.3 Des listes chaînées de taille arbitraire pour itérer sur nos données

Enfin, pour stocker les données qui n'étaient destinées qu'à être itérées, nous avons choisi d'utiliser des listes chaînées car leur taille peux être modifiée dynamiquement.

3. DÉMARCHES ET MÉTHODES UTILISÉES

3.1. Heuristique choisie

Afin de déterminer une **solution réalisable**, nous avons décidé de mettre en place une **heuristique gloutonne** par insertion.
Cette heuristique construit progressivement une solution en rajoutant à chaque itération un capteur. Lorsque toutes les cibles sont couvertes, on obtient une **solution réalisable**. Le choix du capteur à rajouter à la solution en cours de construction est important.
Au début de l'algorithme, on ne dispose que du puits. On cherche donc une cible à remplacer par un capteur qui possède dans son rayon de communication le puits et dans son rayon de captation le plus de cibles possibles. On construit ce que l'on appelle la **solution courante**.
Au tour suivant, on cherche là encore une cible, à remplacer par un capteur, qui communique avec un capteur de la solution courante et qui capte le plus de cibles. On réitère jusqu'à ce que l'on aie capté toutes les cibles.

Cette algorithme nécessite d'**avoir constamment accès aux voisins d'un point**, c'est-à-dire aux points situés dans son rayon de captation ou de communication. Pour accélérer cette partie, nous avons effectuer en amont un pré traitement des points de la grille en les stockant dans la **structure d'arbre kd** présentée plus haut.

3.2. Voisinages utilisés

Nous avons décidé de construire deux structures de voisinages. On rappelle que l'on cherche à minimiser le nombre de capteurs présents sur la grille. Soit X l'ensemble finit des solutions réalisables de notre problème.

- Notre première structure de voisinage est la fonction suivante :

$$\begin{array}{rcccc} Add\_node & : & X & \rightarrow & \mathscr{P}(X) \\ & & x & \mapsto & Add\_node(x) \end{array}$$

  Soit une solution réalisable $x \in X$, $Add\_node(x)$ est un ensemble de solutions réalisables construites à partir de x. Soit $v \in Add\_node(x)$, c'est une solution réalisable construite à partir de x auquel on a rajouté un capteur qui communique avec au moins un capteur déjà présent. Ce voisinage correspond donc à une **dégradation d'une solution réalisable**, à un bruitage.

- Notre deuxième structure de voisinage est la fonction suivante :

$$\begin{array}{rcccc} Remove\_node & : & X & \rightarrow & \mathscr{P}(X) \\ & & x & \mapsto & Remove\_node(x) \end{array}$$

  Soit une solution réalisable $x \in X$, $Remove\_node(x)$ est un ensemble de solutions réalisables construites à partir de x. Soit $v \in Remove\_node(x)$, c'est une solution réalisable construite à partir de x auquel on a enlevé un capteur. Pour que v soit réalisable, on ne peut par retirer n'importe quel capteur. Il faut :

- que le capteur retiré soit dans le rayon de captation d'un capteur encore présent
- que l'ensemble des capteurs encore présents ainsi que le puits forment une seule et même composante connexe, ce qui est facilement vérifiable avec un parcours en profondeur.
- que les cibles couvertes par le capteur retiré le soient encore par au moins un capteur encore présent.

Ce voisinage correspond donc à une **amélioration d'une solution réalisable**.

Illustrons cette structure sur l'exemple suivant (Figure 3).

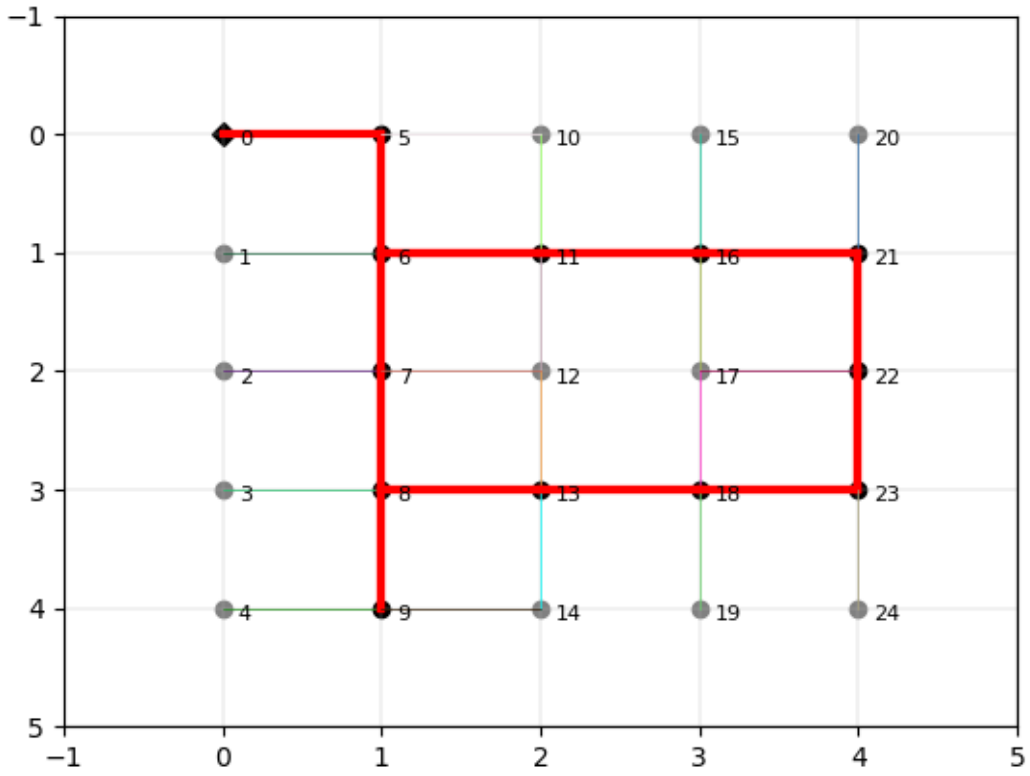

FIGURE 3 – Exemple sur une grille de taille 5.

Notre **première structure de voisinage** nous propose de rajouter les capteurs suivants : 1, 2, 3, 4, 10,12, 14, 15,17, 19, 20 et 24, donc **12 nouvelles solutions réalisables dégradantes**.
Notre **seconde structure de voisinage** nous propose d'enlever les capteurs suivants : 11, 13 et 22, donc **3 nouvelles solutions réalisables améliorantes**.

3.3. Métaheuristique appliquée

Nous avons choisi de mettre en place la métaheuristique du **recuit simulé** présentée en cours. Le choix des paramètres s'est fait de manière expérimentale :

- pour la **solution initiale**, c'est celle donnée par l'algorithme glouton présenté plus haut.
- On choisit au hasard un **voisin** parmi toutes les solutions réalisables générées par les structures de voisinages définies plus haut.
- on choisit une **température initiale** T_ini pas trop élevée autour de 40(°C).
- on choisit un **modèle de décroissance linéaire**, avec $\phi = 0.99995$ afin que la température décroisse très lentement.
- on choisit entre **100 000 et 200 000 itérations** afin de laisser le temps à l'algorithme de passer par les trois phases classiques du recuit simulé : le déplacement aléatoire, l'extraction d'un minimum local et le refus de toute dégradation.
- on choisit des **paliers de température** de tailles faibles, de 2 itérations.

4. Résultats obtenus

4.1. Illustration sur un exemple

Pour l'observation des résultats, nous avons mis en place un affichage graphique (désactivable dans le fichier `Makefile.options`) qui permet de rapidement visualiser la structure des solutions et aussi de déboguer. Aussi, nous ne voulions pas présenter uniquement une liste de points en sortie du programme car c'est peu parlant pour un utilisateur.

Avant de présenter le tableau des résultats, illustrons les différentes étapes de l'algorithme sur un petit exemple, une grille tronquée de 199 points avec $R\_com = R\_sat = 2$.

Après l'algorithme glouton, on obtient en moins d'une seconde le résultat présenté en Figure 4, **38 capteurs ont été placés**. L'intuition nous amène à penser que pour améliorer cette solution, moins de capteurs doivent longer les bords de la grille ou des trous. Regardons si l'on obtient une solution qui vérifie cette propriété.

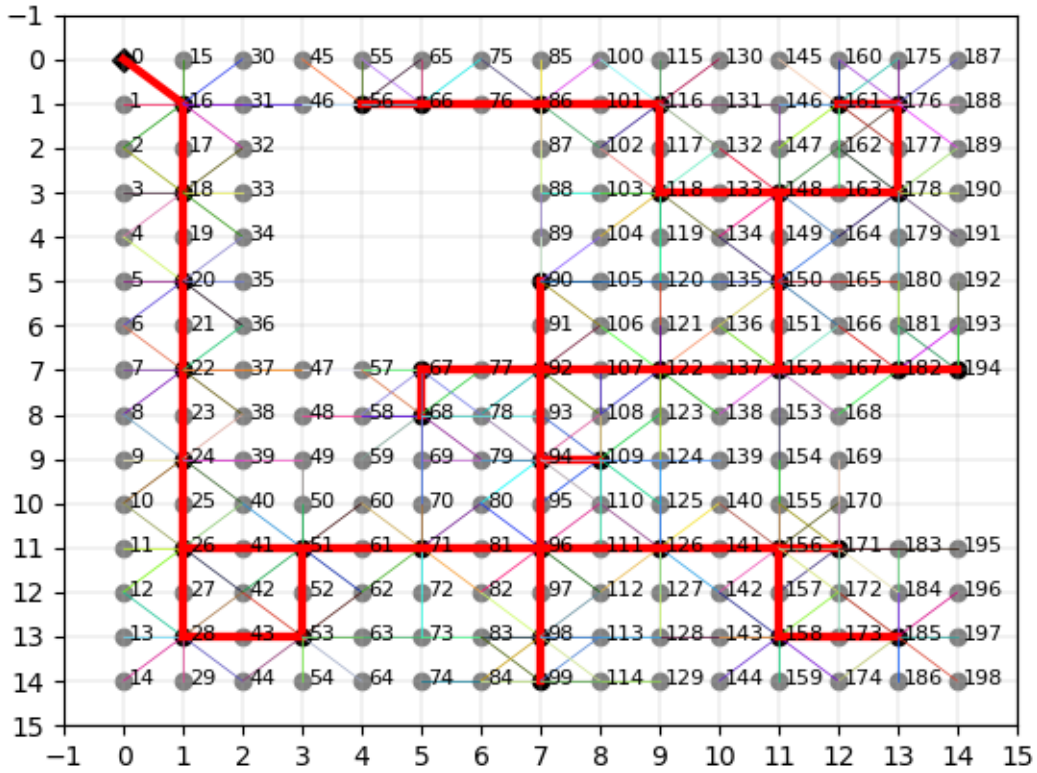

Figure 4 – Solution réalisable obtenue après construction gloutonne

Après l'application d'un recuit simulé avec les paramètres présentés plus haut, on obtient une **meilleure solution réalisable de 32 capteurs** en 20 secondes (Figure 5). Comme prévu, dans cette solution, les capteurs longent bien moins de bords !

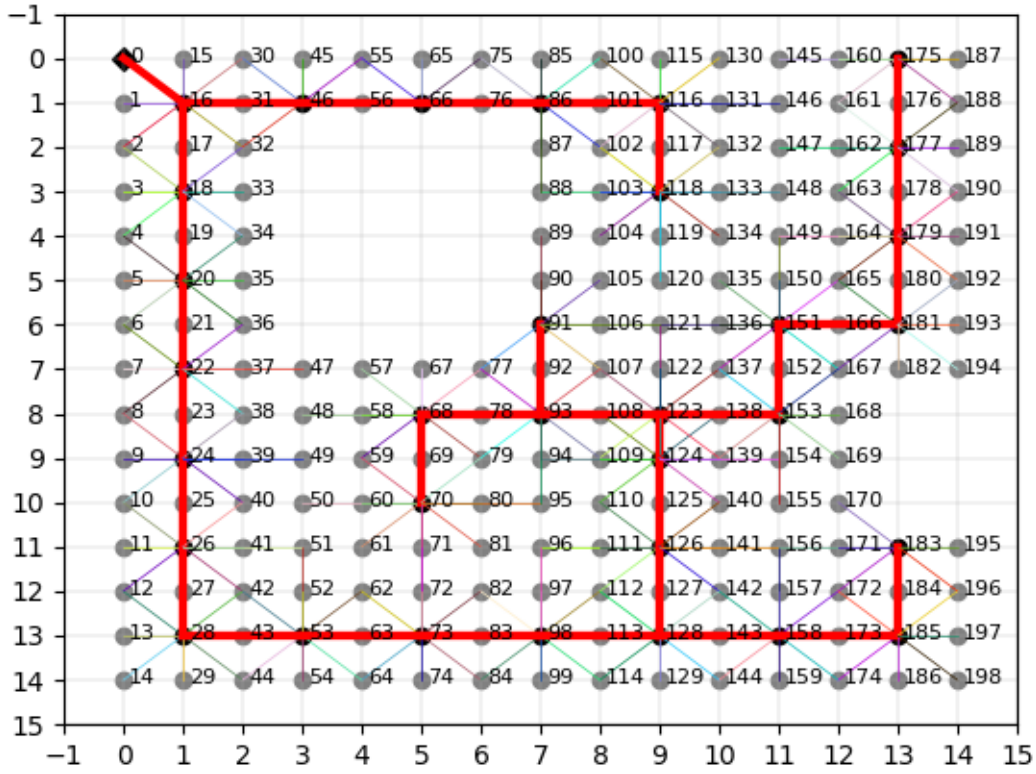

FIGURE 5 – Solution réalisable obtenue après recuit simulé

4.2. Tableau des résultats obtenus

On obtient les résultats suivants sur les différentes instances. On remarque que les résultats sont assez éloignés des minorants donnés pour les grilles avec des points aléatoires.
On remarque également que pour les grilles de petites tailles, la méthode gloutonne donne des résultats très satisfaisants puisqu'ils ne sont que très peu améliorés par le recuit simulé.

	après glouton	majorant	minorant	temps (s)
10*10 (1,1)	41	39		23
10*10 (1,2)	32	30		21
10*10 (2,2)	19	17		17
10*10 (2,3)	14	12		16
15*15 (1,1)	89	80		90
15*15 (1,2)	71	66		88
15*15 (2,2)	41	35		69
15*15 (2,3)	30	25		56
20*20 (1,1)	155	146		314
20*20 (1,2)	121	115		236
20*20 (2,2)	72	62		162
20*20 (2,3)	55	47		133
25*25 (1,1)	239	224		527
25*25 (1,2)	184	177		384
25*25 (2,2)	111	94		246
25*25 (2,3)	78	73		212
30*30 (1,1)	338	313		744
30*30 (1,2)	258	249		594
30*30 (2,2)	159	137		594
30*30 (2,3)	118	102		307
40*40 (1,1)	590	569		1018
40*40 (1,2)	443	427		762
40*40 (2,2)	281	241		551
40*40 (2,3)	194	183		381

FIGURE 6 – Tableau de résultats obtenus sur les grilles simples

	après glouton	majorant	minorant	temps (s)
captANOR225_9_20.dat (1,1)	59	52	30.5	63
captANOR225_9_20.dat (1,2)	35	29	28.5	29
captANOR225_9_20.dat (2,2)	16	13	9.2	13
captANOR225_9_20.dat (2,3)	12	10	9	17
captANOR400_10_80.dat (1,1)	66	61	36.7	38
captANOR400_10_80.dat (1,2)	46	37	34.9	31
captANOR400_10_80.dat (2,2)	18	16	10.87	27
captANOR400_10_80.dat (2,3)	17	12	10.62	11
captANOR625_15_100.dat (1,1)	158	148	84.9	620
captANOR625_15_100.dat (1,2)	99	84	82	184
captANOR625_15_100.dat (2,2)	43	38	22.3	87
captANOR625_15_100.dat (2,3)	33	27	21.9	49
captANOR900_15_20.dat (1,1)	157	140	81.5	81
captANOR900_15_20.dat (1,2)	102	84	79.9	87
captANOR900_15_20.dat (2,2)	42	38	22.1	93
captANOR900_15_20.dat (2,3)	37	27	21.8	68
captANOR1500_21_500.dat (1,1)	295	275	154.86	1371
captANOR1500_21_500.dat (1,2)	188	163	152.07	528
captANOR1500_21_500.dat (2,2)	78	74	41.69	261
captANOR1500_21_500.dat (2,3)	67	53	41.46	104
captANOR1500_15_100.dat (1,1)	157	141	79.54	539
captANOR1500_15_100.dat (1,2)	112	90	79.05	174
captANOR1500_15_100.dat (2,2)	42	39	22.37	121
captANOR1500_15_100.dat (2,3)	37	28	22.16	98

FIGURE 7 – Tableau de résultats obtenus sur les grilles avec points aléatoires

	après glouton	majorant	minorant	temps (s)
captTRUNC87_10_10.dat (1,1)	38	36		22
captTRUNC87_10_10.dat (1,2)	29	27		18
captTRUNC87_10_10.dat (2,2)	18	16		16
captTRUNC87_10_10.dat (2,3)	14	11		15
captTRUNC90_10_10.dat (1,1)	44	39		26
captTRUNC90_10_10.dat (1,2)	31	29		15
captTRUNC90_10_10.dat (2,2)	19	17		13
captTRUNC90_10_10.dat (2,3)	13	11		11
captTRUNC199_15_15.dat (1,1)	83	76		77
captTRUNC199_15_15.dat (1,2)	66	60		66
captTRUNC199_15_15.dat (2,2)	38	33		39
captTRUNC199_15_15.dat (2,3)	27	24		26
captTRUNC200_15_15.dat (1,1)	95	83		73
captTRUNC200_15_15.dat (1,2)	71	62		56
captTRUNC200_15_15.dat (2,2)	40	36		39
captTRUNC200_15_15.dat (2,3)	31	26		27
captTRUNC332_20_20.dat (1,1)	134	122		115
captTRUNC332_20_20.dat (1,2)	105	102		105
captTRUNC332_20_20.dat (2,2)	62	54		67
captTRUNC332_20_20.dat (2,3)	50	41		31
captTRUNC351_20_20.dat (1,1)	161	146		206
captTRUNC351_20_20.dat (1,2)	127	110		153
captTRUNC351_20_20.dat (2,2)	71	61		101
captTRUNC351_20_20.dat (2,3)	52	45		51
captTRUNC436_25_25.dat (1,1)	183	167		227
captTRUNC436_25_25.dat (1,2)	148	134		180
captTRUNC436_25_25.dat (2,2)	80	75		102
captTRUNC436_25_25.dat (2,3)	59	54		80
captTRUNC557_25_25.dat (1,1)	236	230		595
captTRUNC557_25_25.dat (1,2)	188	172		529
captTRUNC557_25_25.dat (2,2)	114	99		429
captTRUNC557_25_25.dat (2,3)	73	70		150
captTRUNC669_30_30.dat (1,1)	273	252		738
captTRUNC669_30_30.dat (1,2)	205	197		348
captTRUNC669_30_30.dat (2,2)	122	107		344
captTRUNC669_30_30.dat (2,3)	89	81		109
captTRUNC800_30_30.dat (1,1)	344	318		1019
captTRUNC800_30_30.dat (1,2)	271	245		886
captTRUNC800_30_30.dat (2,2)	160	141		538
captTRUNC800_30_30.dat (2,3)	109	104		213
captTRUNC1223_40_40.dat (1,1)	501	457		1018
captTRUNC1223_40_40.dat (1,2)	363	356		829
captTRUNC1223_40_40.dat (2,2)	220	202		793
captTRUNC1223_40_40.dat (2,3)	156	145		239
captTRUNC1425_40_40.dat (1,1)	592	572		3788
captTRUNC1425_40_40.dat (1,2)	481	435		2711
captTRUNC1425_40_40.dat (2,2)	279	248		1715
captTRUNC1425_40_40.dat (2,3)	183	177		344

FIGURE 8 – Tableau de résultats obtenus sur les grilles tronquées

5. Conclusion

Pour conclure, nous avons mis en place une heuristique gloutonne puis un recuit simulé pour obtenir des solutions de qualité satisfaisante. Enfin, le langage et les structures de données utilises ont permis d'optimiser la complexité des différents algorithmes associés aux méthodes mises en place.