Etude de cas : Couverture connexe minimum dans des réseaux de capteurs

Matthieu Roux et Pierre Laplaize

MPRO, Métaheuristiques, 2019

1 Introduction

La disposition de capteurs en réseau nécessite la satisfaction de deux contraintes de nature raltivement différentes. L'une, que nous appellerons contrainte de captation, impose que chaque cible soit dans le bassin de captation d'au moins un capteur. La seconde, que nous appellerons contrainte de communication, impose que le réseau de capteurs soit connexe et puisse communiquer avec le puits. Il semblerait d'ailleurs que ce soit la contrainte de connexité qui ajoute beaucoup de complexité au problème, bien que nous verrons que nous pourrons distinguer les instances avec $R_{capt} = R_{com}$ et $R_{capt} < R_{com}$.

Notre angle d'approche pour, sinon résoudre, au moins proposer une solution approchée de bonne qualité à ce problème a été structuré autour de deux métaheuristiques. D'une part nous avons mis en place une VNS, qui a donc nécessité de proposer un certain nombre de voisinages à explorer, et d'autre part un algorithme de colonies de fourmis, l'intuition sous-jacente étant qu'il pourrait permettre d'identifier des "noeuds charnières", ayant beaucoup/peu de chances de figurer dans une solution optimale.

Dans ce rapport synthétique, nous commencerons par présenter une heuristique simple permettant de générer des solutions réalisables. Puis nous détaillerons la VNS, avec les voisinages étudiés et les succès et difficultés rencontrées. Puis dans un second temps nous exposerons l'aglorithme de colonies de fourmis mis en oeuvre, en précisant notamment les fonctions de phéromones et les voisinages choisis pour faire se déplacer les fourmis.

2 Une première heuristique gloutonne

Une première manière de construire une solution réalisable est de considérer un algorithme glouton. En effet, trouver une solution n'étant pas difficile (poser des capteurs sur tous les sommets est réalisable), il est intéressant d'envisager un algorithme glouton.

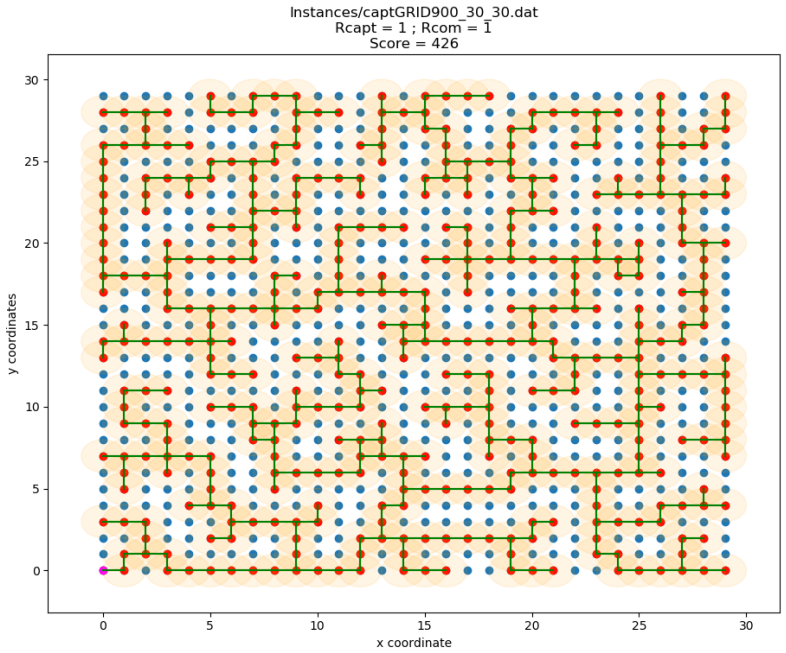
L'approche retenue a été la suivante. Pour commencer on pose un capteur sur toutes les cibles. Puis, tant que l'on peut on retire un capteur au hasard (à condition que la solution produite soit toujours réalisable). En reproduisant le processus éventuellement plusieurs fois et en sélectionnant la meilleure solution obtenue, on obtient ainsi aisément une première solution réalisable de qualité "moyenne".

Cette première solution pourra ensuite être utilisée comme point de départ pour les métaheuristiques suivantes.

Table 1. Performance moyenne de cette première heuristique gloutonne (moyennée sur 100 appels)

Instances	R_{capt}, R_{com}	Score	Temps de calcul moyen
Grille 10x10	(1,1)	48.79 ± 2.00	0.04s
	(1,2)	35.23 ± 1.01	0.05s
Grille 20x20	(1,1)	189.87 ± 3.40	0.76s
	(1,2)	134.92 ± 2.47	0.92s
Grille 40x40	(1,1)	742.68 ± 2.53	15.05s
	(1,1)	526.35 ± 3.99	14.08s

Figure 1. Structure d'une solution obtenue avec *greedyDelete*



3 VNS

3.1 Description de l'étude

Ayant déjà travaillé sur le recuit simulé, nous avions envie de mettre à profit ce projet afin d'explorer des nouvelles métaheuristiques vues dans le cours. Pour ce faire, il nous a semblé que la méthode VNS, reposant tout comme le recuit sur une sélection soigneuse des voisinages, permettrait de bien s'adapter au problème.

Nous présenterons donc dans cette partie la structure général du code, puis détaillerons quelques voisinages, avant de terminer par dire quelques mots sur les succès et échecs rencontrés.

3.2 Structure du code

Le code, intitulé VNS.py, est structuré autour de la notion centrale de voisinage. Nous avons donc construit un ensemble de voisinages, certains étroits, d'autres plus larges, et tentons d'améliorer une solution courante en utilisant ces structures de voisinages.

Ainsi, l'algorithme explore successivement les différents voisinages V_0, V_1, V_2, etc... Lorsqu'il parvient à améliorer strictement une solution, il revient à V_0. L'algorithme s'arrête alors lorsque le dernier voisinage ne permet pas d'améliorer la solution courante.

Par la suite, nous avons essayé de tirer profit de la structure multi-coeurs de nos ordinateurs, et pour ce faire avons parallélisé le code. Ceci n'a pas été évident, et il semblerait que ce ne soit pas la manière la plus adéquate de tirer profit de cet atout, mais cette méthode a au moins le mérite d'être simple à mettre en place : lancer 6 calculs en parallèle, partant de 6 solutions différentes.

En pratique, nous avons essayé d'introduire quelques synchronisations entre les différents processus parallèles :

1. Chaque processus ne gère pas une solution mais plutôt une liste de solutions. Cela permet de diminuer la dépendance à la solution initiale en apportant de la diversité.
2. Chaque processus, lorsqu'il tente d'améliorer ses solutions courantes, explore les voisinages en procédant sur les solutions successivement par ordre croissant de score. Ainsi, lorsque l'on passe au voisinage V_1, un processus tentera en premier d'améliorer sa meilleure solution courante.
3. A chaque fois qu'un voisinage V_j s'avère bloqué dans un optimum local et que l'on doit passer à V_{j+1}, les processus communiquent de la façon suivante : on trie l'ensemble des solutions par ordre croissant de score. Puis on affecte les solutions aux différents processus (le processus $P1$ recevra donc dans sa liste courante une très bonne solution, une bonne solution, une moyenne et une mauvaise par exemple ; idem pour $P2$ etc...)
4. Cette communication permet de faire en sorte que l'on tente toujours d'améliorer en premier des bonnes solutions. (Ceci permet ensuite d'envisager d'écrémer les solutions courantes que l'on maintient dans la liste au fur et à mesure que l'on explore des voisinages de plus en plus grands).

L'algorithme VNS renvoie enfin le score de la meilleure solution rencontrée.

3.3 Implémentation pratique

En pratique, l'algorithme peut être décrit schématiquement de la façon suivante :

1. Des voisinages, que l'on notera pour le moment v_0, v_1, v_2, ..., v_k. Ils sont ordonnés par taille croissante (nombre de voisins) et par ordre d'importance décroissant (ceux qui améliorent vite et beaucoup une solution sont placés en premier)
2. Des ensembles de voisinages V_0, V_1, V_2, ..., V_k définis ainsi : $V_j = \{v_0, v_1, ..., v_j\}$ $\forall j \in \{0, ..., k\}$. Lorsque plus aucun voisinage de V_j ne peut améliorer la solution, on passe donc à l'ensemble suivant V_{j+1}
3. Des fonctions $runParallelVj$: lance plusieurs descentes locales en parallèle en explorant les voisinages de V_j. Prend en argument une liste de solutions courantes à améliorer.

Ainsi, le corps du script consiste donc à générer une liste de solutions initiales avec $runParallelV0$. Puis, lancer $runParallelV1$ sur la liste ainsi générée et répéter le processus jusqu'à $runParallelVk$.

3.4 Notions de voisinage

Dans cette étude, nous nous sommes concentrés sur les voisinages d'un seul type, faisant appel à l'ajouts de ce que nous appellerons sommets pivots. Lorsqu'il n'est plus possible de supprimer un sommet sans violer l'une des deux contraintes, on doit trouver un autre moyen de diminuer le score. L'idée est donc d'essayer d'ajouter 1 sommet (dit pivot) puis d'en supprimer 2 autres. Cette opération permet donc de diminuer de 1 le score, sous réserve que la solution reste réalisable.

On généralise ensuite aisément cette notion avec k pivots, nécessitant la suppression de $k + 1$ autres sommets.

La question qui se pose ensuite est de savoir parmi quels ensembles on peut choisir :

1. Les *pivots* : sommets que l'on va tenter d'insérer
2. les *candidats* : sommets que l'on va tenter de supprimer

A priori, les pivots potentiels sont tous les sommets vides (ie n'ayant pas reçu de capteur).

En revanche, le sujet mérite plus d'attention pour les *candidats*. En effet, s'il n'y avait que la contrainte de captation, il n'y aurait que les sommets autour du pivot à considérer (autour signifiant $2R_{capt}$). Cependant, la contrainte de connexité fait tomber cette hypothèse simplificatrice et il faudrait considérer tous les sommets pleins, ce qui en fait un voisinage tout de suite beaucoup plus large.

Par la suite, nous ferons donc référence à 2 sous-catégories de voisinages : 'small' et 'large'. 'Small' désignera le fait que l'on a restreint le voisinage en ne considérant que les candidats "proches" du pivot (ou des pivots). Par opposition, 'large' désignera un voisinage plus exhaustif, mais donc aussi plus grand.

3.5 Description des voisinages

— *greedyDelete* : tente de supprimer un sommet au hasard
— *greedyPivot*1 : tente d'insérer 1 pivot puis de supprimer 2 candidats. Existe en version 'small' et 'large'
— *greedyPivot*2 : tente d'insérer 2 pivots puis de supprimer 3 candidats. Existe en version 'small' et 'large'

— $greedyPivot3$: tente d'insérer 3 pivots puis de supprimer 4 candidats. Commence à devenir un assez gros voisinage, que l'on a du abandonner pour des raions de temps de calcul trop important (il était illusoire de pouvoir effectuer un parcours exhaustif de ce visinage)
— $localPathPivot1$: tentative pour sortir des minima locaux. Au lieu de chercher une structure voisine strictement améliorante, construit une succession de solutions de score identique (en insérerant 1 pivot puis supprimant 1 candidat), jusqu'à pouvoir améliorer strictement le score. En pratique fonctionne avec une liste tabou qui empêche de cycler en insérant puis supprimant les mêmes éléments. Ce voisinage semblait être une bonne idée mais a du être mis de côté car il ne donnait pas de résultat. Cependant, nous restons convaincus qu'il peut y avoir des pistes d'amélioration de ce côté-là...

3.6 Le critère d'arrêt

Pour des raisons de praticité, nous avons choisi un critère d'arrêt déterminé par un temps de calcul maximum. Ainsi, on est sûr de pouvoir obtenir une solution en temps raisonnable.

En pratique, cela a nécessité quelques ajustement dans la VNS :

1. $dtMax$: durée maximum de calcul, spécifiée au début de l'alorithme
2. Pour chaque voisinage V_i, on attribue une fraction du temps total $dtMax$. En pratique ce paramétrage est difficile à calibrer, mais il permet de répartir les calculs en fonction des largeurs des voisinages
3. Nous avons du insérer un critère d'arrêt également lorsque l'on examine un voisinage. En effet, pour certaine instance le simple parcours d'un voisinage peut requérir plusieurs dizaines de secondes. Par conséquent, il était important d'insérer dans ce parcours également une durée maximum d'exécution

3.7 Calibrage des paramètres

Cette implémentation nécessite de calibrer deux types de paramétrages, tous les deux liés au temps de calcul. En effet, on impose un critère pratique de temps maximum de calcul, afin d'éviter d'avoir des voisinages tournant pendant des heures sur les grosses instances et bloquant le reste des calculs. Par conséquent, il faut déterminer quel $dtMax$ imposer en fonction de la taille de l'instance. Une bonne idée semble d'utiliser la génération initiale de solutions comme étalons : on souhaite donner suffisamment de temps pour que plusieurs solutions initiales puissent être générées (typiquement 4 à 10). Ensuite, nous devons nous assurer que tous les voisinages seront explorés, au moins partiellement. Il nous faut donc répartir un "budget temps" à tous les ensembles de voisinages V_j, le tout en veillant à ce que pour chacun de ces ensembles, on ait le temps d'examiner, sinon toutes les solutions courantes de la liste, au moins une proportion significative.

Sur la figure ci-dessous, on représente les données issues de la calibration. On retrouve bien le temps de calcul global, ainsi que le nombre de solutions générées pour V_0 ainsi que la proportion de solutions courantes examinées dans chanque V_j, $j \geq 1$

Table 2. Calibrage des temps alloués par instance et par type de voisnage pour $(R_{capt}, R_{com}) = (1, 1)$

Instances	V_0 : solutions générées	V_1 : %	V_2 : %	V_3 : %	V_4 : %	Temps de calcul global
Grille 10x10	7	6.2/7	7/7	5/7	5/7	dt = 24s
Grille 20x20	7.5	7.5/7.5	7/7.5	7/7.5	7/7.5	dt = 2*600s
Grille 40x40	5.2	4/5.2	3/5.2	2.5/5.2	2.5/5.2	dt = 25*60s

FIGURE 2. Illustration de la calibration

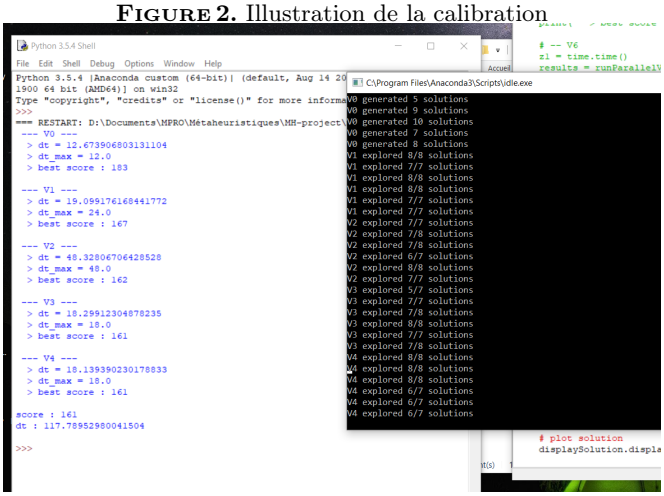
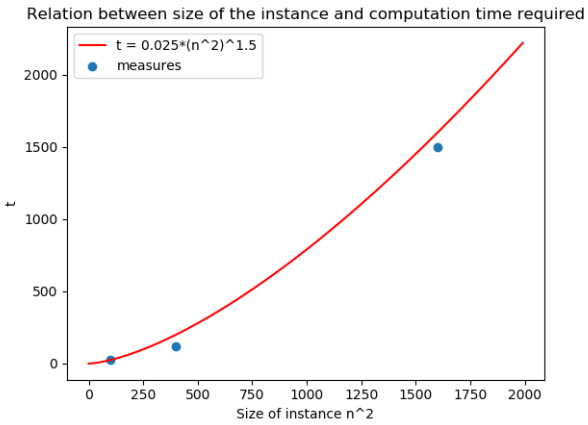


FIGURE 3. Illustration de la calibration de $dtMax$



3.8 Echecs

Parmi les échecs rencontrés, nous avons déjà évoqué les voisinages $greedyPivot3$ et $localPathPivot1$ qui n'ont pas donné les résultats escomptés (en pratique, ils n'ont pas réussi à améliorer la solution courante en temps raisonnable).

A présent nous aimerions dire quelques mots sur les voisinages 'large' et les difficultés qu'ils ont engendrées. Un voisinage 'large' est en pratique trop grand pour pouvoir être parcouru de manière exhaustive. Par conséquent, nous avons tenté d'introduire une version stochastique. Au lieu de parcourir tout le voisinage, on n'en parcourt qu'un sous-échantillon. Cependant, une question se posait : comment choisir la taille d'un sous-échantillon ? La réponse pouvait être de deux types, mais engendrait à chaque fois des insatisfactions :

— Dans un cas on pouvait imposer un temps de calcul limite. Problème : cela ne s'insérait pas du tout dans notre code, qui était malheureusement architecturé pour parcourir tous les voisins d'une solution courante puis enchainer sur une autre solution courante. Il nous aurait alors fallu plus de temps.

— Imposer un nombre maximum d'itérations (ie la taille du sous-échantillon). Mais alors comment définir la taille ? Il faut qu'elle ne soit ni trop grande pour bloquer les calculs trop longtemps et empêcher l'exploration d'autres solutions courantes, mais ni trop faibles pour garantir une certaine exploration tout de même.

4 Colonie de fourmis

Le choix de l'algorithme de la colonie de fourmis s'est fait à partir de ce constat simple : cette métaheuristique a des défauts et des avantages bien différents de l'autre métaheuristique que nous avons choisi de développé. En réalité elle s'est révélée très intéressante puisqu'elle permettait de comparer plusieurs solutions entre elles et de toujours utiliser les meilleures d'entre elles.

Certains choix de paramètres dans l'algorithme se sont révélés épineux et ont requis des concertations. Par exemple : quel est le rapport à fixer entre les probabilités de se déplacer sur une cible qu'on a déjà visité par rapport à une cible sur laquelle on est jamais allé ? La réponse sur laquelle nous nous somme mis d'accord a été 0. Si on autorise une fourmi à revenir sur ses pas alors il y a une chance beaucoup trop importante pour créer des boucles et des cycles inutiles. Le revers de la médaille de cette décision est que beaucoup de fourmis, dans un premier temps et sans l'aide des phéromones peuvent se perdre et ne jamais réussir à créer une solution réalisable. Nous avons essayé aussi de pénaliser plus lourdement et de favoriser plus largement certaines fourmis sur un essai mais le résultat ne changeait guère.

Figure 4. Scores moyens obtenus pour une pénalisation simple des phéromones

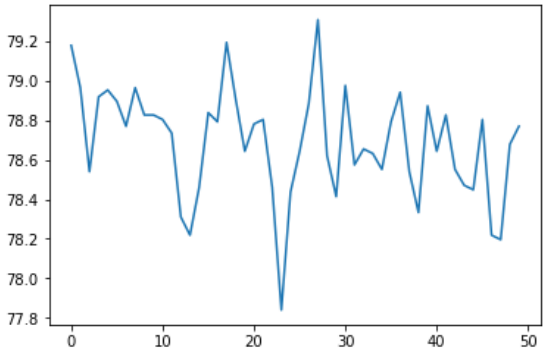
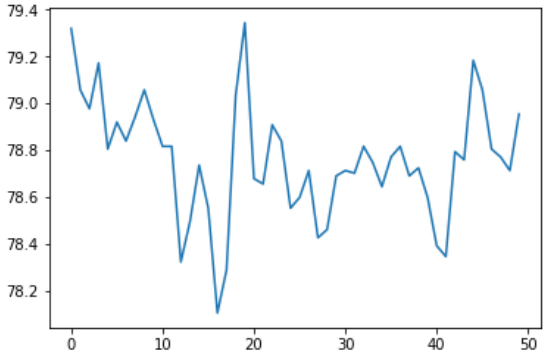


Figure 5. Scores moyens obtenus pour une pénalisation cinq fois plus forte des phéromones



On peut observer ci dessus que la puissance de la pénalisation n'influait pas beaucoup sur l'évolution du score moyen.

On peut aussi noter que ces figures font preuve d'efficacité de l'algorithme, grâce aux phéromones, les solutions se font bien de plus en plus petites.

Pour réussir à faire une différence plus significative. Nous nous sommes dit que la probabilité qu'il soit sélectionné était déjà considérable grâce au travail des phéromones. Forcer la main à l'algorithme ne nous a semblé ni pertinent ni efficace, puisque le temps gagné n'était pas suffisamment intéressant, et que en soi il était probable que l'algorithme fasse ça de lui même. Mon algorithme était relativement lent, c'était inévitable dans la mesure où il devait faire beaucoup de test et qu'il s'agit de méta heuristique mais j'ai préféré concentrer mes efforts sur la précision de mon programme plutôt que sur sa complexité temporelle. Nous ne regrettons pas cette décision même si cela nous a handicapé notamment au moment où nous avons décidé de faire les tests finaux pour noter nos résultats.

Il ne nous a pas paru pertinent de chambouler complètement les pics de phéromones quand il y en a pour sortir d'un éventuel optimum local. Le risque de l'optimum local est le suivant : une première fourmi tombe dans cet optimum et trouve un meilleur résultat que ses collègues qui n'ont pas trouvé d'optimum. On met donc des phéromones sur cet optimum local, les prochaines fourmis tombent facilement dans ce piège et ainsi de suite. Déjà il faut souligner que l'on fait parcourir un nombre de fourmi égal au nombre de cible, ainsi il faut que l'optimum soit très intéressant ou être très malchanceux pour tomber dedans. Ensuite comme on fait voyager beaucoup de fourmis et qu'on fait un grand nombre de générations de fourmis nous avons fait des tests et il est apparu qu'il valais mieux utiliser peu de fourmi mais beaucoup de générations de manière à optimiser) nous savons qu'il est très improbable de rester dans cet optimum local avec autant de fourmi.

5 Résultats

Table 3. Grilles simples

Instances	R_{capt}, R_{com}	Score VNS	Score colonie de fourmis
Grille 10x10	(1,1)	39	-
	(1,2)	29	60
	(2,2)	17	-
	(2,3)	12	-
Grille 15x15	(1,1)	84	-
	(1,2)	66	-
	(2,2)	35	-
	(2,3)	27	-
Grille 20x20	(1,1)	156	-
	(1,2)	119	-
	(2,2)	65	-
	(2,3)	47	-
Grille 25x25	(1,1)	253	-
	(1,2)	185	-
	(2,2)	102	-
	(2,3)	73	-
Grille 30x30	(1,1)	363	-
	(1,2)	267	-
	(2,2)	144	-
	(2,3)	108	-
Grille 40x40	(1,1)	699	-
	(1,2)	485	-
	(2,2)	295	-
	(2,3)	195	-

Table 4. Grilles tronquées

Instances	R_{capt}, R_{com}	Score VNS	Score colonie de fourmis
TRUNC87_10_10	(1,1)	36	-
	(1,2)	28	-
	(2,2)	16	-
	(2,3)	11	-
TRUNC90_10_10	(1,1)	39	56
	(1,2)	29	-
	(2,2)	17	-
	(2,3)	11	-
TRUNC199_15_15	(1,1)	78	-
	(1,2)	60	-
	(2,2)	34	-
	(2,3)	24	-
TRUNC200_15_15	(1,1)	87	-
	(1,2)	64	-
	(2,2)	36	-
	(2,3)	27	-
TRUNC332_20_20	(1,1)	131	-
	(1,2)	103	-
	(2,2)	57	-
	(2,3)	41	-
TRUNC351_20_20	(1,1)	156	-
	(1,2)	114	-
	(2,2)	66	-
	(2,3)	46	-
TRUNC436_25_25	(1,1)	177	-
	(1,2)	139	-
	(2,2)	80	-
	(2,3)	56	-
TRUNC557_25_25	(1,1)	247	-
	(1,2)	178	-
	(2,2)	102	-
	(2,3)	72	-
TRUNC669_30_30	(1,1)	277	-
	(1,2)	206	-
	(2,2)	117	-
	(2,3)	83	-
TRUNC800_30_30	(1,1)	-	-
	(1,2)	-	-
	(2,2)	-	-
	(2,3)	-	-
TRUNC1223_40_40	(1,1)	-	-
	(1,2)	-	-
	(2,2)	-	-
	(2,3)	-	-
TRUNC1425_40_40	(1,1)	-	-
	(1,2)	-	-
	(2,2)	-	-
	(2,3)	-	-

Table 5. Cibles aléatoirement réparties

Instances	R_{capt}, R_{com}	Score VNS	Score colonie de fourmis
ANOR225_9_20	(1,1)	52	-
	(1,2)	29	105
	(2,2)	13	-
	(2,3)	10	-
ANOR400_10_80	(1,1)	62	-
	(1,2)	38	180
	(2,2)	16	-
	(2,3)	13	-
ANOR625_15_100	(1,1)	152	-
	(1,2)	86	-
	(2,2)	39	-
	(2,3)	29	-
ANOR900_15_20	(1,1)	150	-
	(1,2)	89	-
	(2,2)	40	-
	(2,3)	28	-
ANOR1500_15_100	(1,1)	152	-
	(1,2)	97	-
	(2,2)	42	-
	(2,3)	30	-
ANOR1500_21_500	(1,1)	322	-
	(1,2)	177	-
	(2,2)	85	-
	(2,3)	61	-

6 Annexe - Quelques méthodes pour vérifier la pertinence des algorithmes mis en place

6.1 readData

Premier module pour lire les données et créer les premières fonctions qui seront utiles quelque soit la manière par laquelle on choisit d'aborder le problème.

Par exemple, il faut nécessairement créer les matrices Acom et Acapt ne serait ce que pour pouvoir créer des solutions et voir si elles sont recevables.

6.2 ParserInstance

Ce module reprend et optimise le module précédent en créant Acom et Acapt automatiquement et qui crée de nouvelles instances particulièrement utiles. Par exemple NeighCom qui permet de renvoyer en temps constant les sommets accessibles à partir d'un sommet quelconque pour le rayon Rcom.

6.3 Constraints

Ce module est de loin le plus important de tous les modules créés en amont.

Ce module permet de vérifier que pour une solution donnée, toutes les contraintes sont bien vérifiées, et en cas de non satisfiabilité de la solution savoir d'où vient le problème et qu'est ce qu'il manque à cette instance pour qu'elle soit valide.

6.4 displaySolution

displaySolution donne une représentation des solutions obtenues. Ce module n'est pas fondamental en soi mais permet de faire une présentation beaucoup plus efficace de nos résultats. Qui plus est il nous a été utile dans la visualisation de ce que nous avons réussi à produire et de ce qu'il nous manque pour arriver à améliorer la solution.

Par exemple on peut remarquer beaucoup plus facilement ce que nous avons appelé un "point charnière", à savoir un point incontournable pour pouvoir couvrir toutes les cibles.