Étude de cas : k-couverture connexe minimum dans les réseaux de capteurs

Nathan Godey et Valentine Huré

3 Novembre 2020

1 Choix de représentation du problème

1.1 Rappel du problème et notations

Nous modélisons ici un problème de couverture connexe à nombre de sommets minimal dans un réseau de capteurs.

On considère un ensemble $(T_i)_{i\in[0,N]}$ de N cibles, dont un puits T_0. Chaque cible T_i est placée en un point (x_i, y_i), avec notamment $T_0 = (0, 0)$.

On pose 3 paramètres : un entier $k \in \{1, 2, 3\}$, un rayon de captation R_{capt} et un rayon de communication $R_{comm} \geq R_{capt}$. Ces rayons induisent des graphes de connexité G_{capt} et G_{comm} dans l'ensemble T des cibles.

Une solution admissible est un sous-ensemble d'indices $I_C \subset [1, N]$ définissant un ensemble de capteurs $C = (T_i)_{i\in I_C}$ tels que :

- Il existe un chemin dans G_{comm} entre tout capteur $c \in C$ et le puits T_0
- Toute cible T_i pour $i \in [1, N]$ est captée au moins k fois, à savoir au moins k de ses voisins sont contenus dans C

Nous cherchons à minimiser, dans l'espace des solutions admissibles A, le nombre de capteurs. Notre problème se résume donc à :

$$I_C^* \;=\; \underset{I_C\in A}{\operatorname{argmin}} \, |I_C|$$

1.2 Choix de la représentation et implémentation

Une approche possible pour la représentation du problème et de ses solutions est d'utiliser l'algèbre linéaire, via une représentation vectorielle utilisant des composantes binaires. Néanmoins, en étudiant les caractéristiques de ce problème, et en prenant en considération la complexité de certaines opérations, il nous est apparu clairement que cette représentation n'était pas optimale du point de vue

de la complexité théorique et informatique.

En effet, la plupart des objets considérés (matrice d'adjacence, vecteurs binaires) sont creux, ce qui nous conduirait dans certains langages à utiliser des librairies de calcul spécifiques.

Il est pourtant possible de simplement changer de paradigme et de considérer ce problème d'un point de vue ensembliste. D'un point de vue calculatoire, cette formalisation est analogue à l'utilisation d'objets creux (ou *sparse*), mais il est important de redéfinir les opérations de graphe sous ce nouvel angle.

Techniquement, les solutions sont représentées par des `std::unordered_set` en C++, contenant les indices du I_C correspondant. Cette structure de conteneur propose l'insertion, la déletion, ainsi que la recherche d'élément en $O(1)$. On pourra aussi profiter de l'assurance d'unicité des éléments dans un `unordered_set`, évitant ainsi les problèmes de maladresses, et permettant l'insertion sans vérification préalable.

De même, nous représentons les matrices d'adjacence par des listes d'ensembles (implémentées sous la forme d'`unordered_set`s), ce qui revient à considérer la liste des ensembles de voisins pour chaque cible.

On remarque que le produit scalaire entre deux vecteurs binaires devient l'intersection pour deux vecteur creux représentés comme des ensembles. C'est un avantage important puisque l'opération d'intersection entre deux `std::unordered_set`s peut être implémentée en complexité $O(min(m, n))$ où m et n représentent la taille des vecteurs d'entrée.

Dans la pratique, on observe d'une part que les algorithmes implémentés ont une complexité qui diminue à mesure que les vecteurs et matrices sont creux, ce qui fournit un avantage pour les graphes peu connexes; et d'autre part, on observe que les itérations de ces algorithmes sont de plus en plus rapides à mesure que les solutions considérées contiennent peu de capteurs, puisque les ensembles résultants deviennent plus petits au fil du temps.

2 Heuristique

2.1 Description

L'heuristique comporte deux étapes successives : une étape de construction d'une solution admissible et une étape de suppression de capteurs qui est une heuristique gloutonne.

2.1.1 Heuristique gloutonne

Le principe de cette heuristique est d'affecter un poids à chacune des cibles et de supprimer les cibles dans l'ordre croissant de leur poids, tout en s'assurant

que cette suppression nous donne une solution admissible.

Nous aurions pu choisir d'affecter à une cible un poids égal à son nombre de voisins dans le graphe de captation. Ainsi, les cibles ayant peu de voisins auraient été retirées en premier. Cependant, nous avons décidé d'affecter un poids qui comprendrait aussi une information sur le nombre de voisins des voisins de la cibles : nous affectons à une cible la somme sur ses voisins des inverses du nombre de leur voisins (excepté la cible en question). Cette affectation des poids permet de prendre en compte à la fois l'intérêt que nous devrions porter à une cible (plus une cible à de voisins, plus elle est valorisée en moyenne), mais aussi sa capacité à capter des voisins qu'elle est la seule à pouvoir capter (moins un voisin est captable, plus il valorise la cible).
Ainsi, en utilisant la matrice d'adjacence du graphe de captation, M_{capt}, nous avons la formule suivante pour le poids p_i de la cible i:

$$p_i = \sum_j \frac{M_{capt}(i,j)}{\sum_{k \neq i} M_{capt}(j,k)}$$

Cette heuristique appliquée à la solution admissible pour laquelle toutes les cibles sont des capteurs ($I_C = [1, N]$) nous donne une solution admissible assez proche du minorant (en moyenne nous obtenons une augmentation de 40% par rapport au minorant).

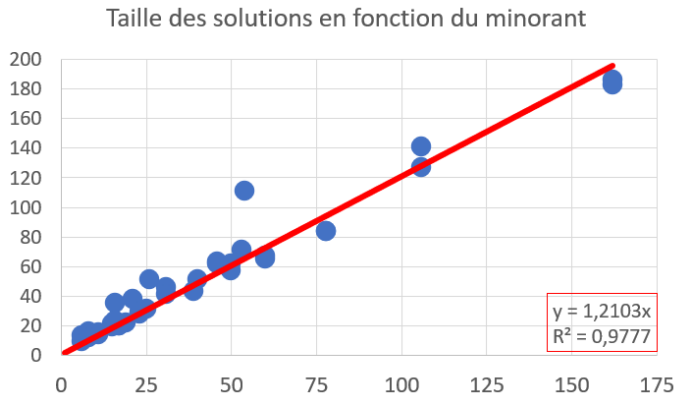

Figure 1: Taille des solutions en fonction du minorant du problème pour les instances de taille inférieure ou égal à 625

Cependant pour des solutions comportant beaucoup de capteurs, le temps de calcul d'admissibilité du vecteur est assez long. En effet, une partie de la vérification consiste à effectuer un parcours du graphe de communication engendré par les capteurs : la complexité est en $O(n + m)$ où n est le nombre de capteurs et m le nombre d'arètes dans le graphe de communication engendré par les capteurs ($m \geq n + 1$), il est donc intéressant de partir d'une solution admissible plus petite que la solution triviale.

2.1.2 Accélération de l'heuristique gloutonne

Afin d'obtenir une solution admissible rapidement, nous avons constaté qu'il suffit de générer k arbres couvrants dans le graphe de captation en faisant en sorte que certaines arêtes ne puissent pas être sélectionnées plusieurs fois. Les noeuds de ces arbres seront nos capteurs.

L'algorithme consiste en la génération successive de k arbres couvrants dans le graphe de captation qui va subir des modifications entre ces générations d'arbres. Entre la génération du premier et du deuxième arbre (si k est supérieur à 1 donc), on supprime toutes les arêtes (à l'exception de celles reliées au puits) du premier arbre couvrant : les cibles ne peuvent pas être captées par les mêmes capteurs dans le second arbre. Entre la génération du deuxième et du troisième arbre (si k est égal à 3 donc), on supprime toutes les arêtes reliant un noeud du deuxième arbre à une feuille : les cibles n'étant pas désignées comme capteurs seront reliées à un 3ème capteur et les cibles désignées comme capteurs seront au moins reliées à deux capteurs. Il se peut que certaines cibles ne puissent pas apparaître dans le dernier graphe, il suffit alors de les ajouter à la liste des cibles pour obtenir une solution admissible.

Cet algorithme fournit des solutions contenant entre 30 et 70% des cibles (à quelques exceptions près pour la plus petite instance). Utiliser la solution fournie par cet algorithme permet donc d'accélérer grandement l'heuristique gloutonne qui, grâce à l'implémentation ensembliste du problème, itère beaucoup plus vite à mesure que les instances diminuent en taille.

Ainsi, l'utilisation de cet algorithme avant l'heuristique permet de réduire en moyenne de 70%, la génération d'une solution admissible. De plus, la solution trouvée est de taille (en moyenne) similaire à ce que pouvait trouver l'heuristique gloutonne seule.

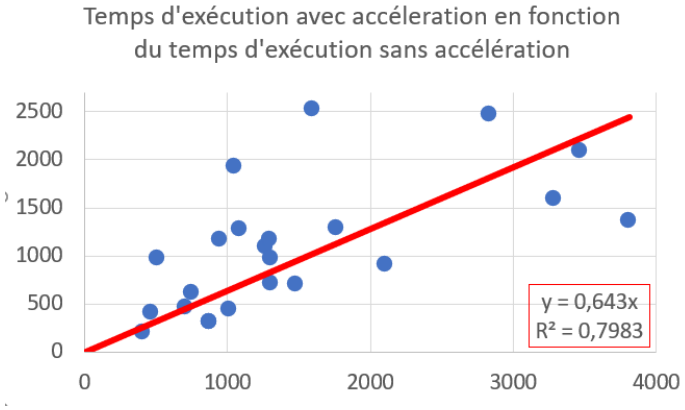

Figure 2: Temps d'exécution de l'heuristique avec accélération en fonction du temps d'exécution sans accélération (les temps sont en millisecondes)

2.2 Résultats obtenus

Voici la représentation graphique de quelques solutions obtenues. Les points verts sont les cibles, les points rouges sont les cibles désignées capteurs et le triangle bleu est le puits. Les traits rouges entre deux capteurs signifient que ces capteurs communiquent.
Pour l'instance de 150 cibles avec $K = 2$, $R_{comm} = 3$,$R_{capt} = 2$, notre heuristique trouve un solution à 13 capteurs (pour un minorant à 11).

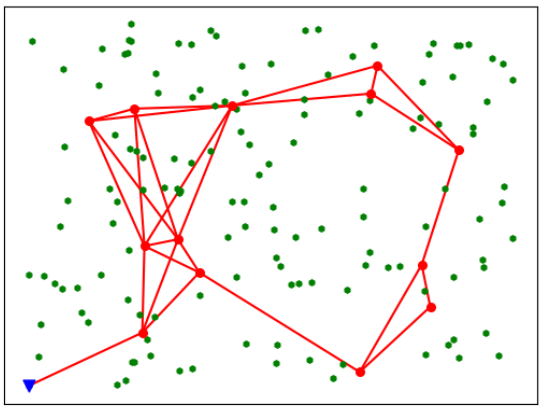

Figure 3: Instance de 150 cibles avec $K = 2$, $R_{comm} = 3$,$R_{capt} = 2$

Pour l'instance de 625 cibles avec $K = 2$, $R_{comm} = 2$,$R_{capt} = 1$, notre heuristique trouve un solution à 126 capteurs (pour un minorant à 105.88).

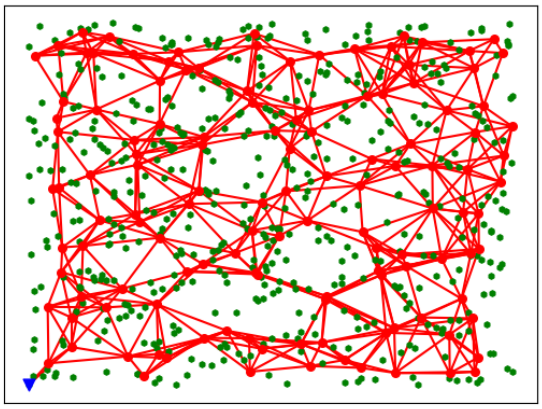

Figure 4: Instance de 625 cibles avec $K = 2$, $R_{comm} = 2$,$R_{capt} = 1$

Pour l'instance de 1500 cibles avec $K = 3$, $R_{comm} = 2$, $R_{capt} = 1$, notre heuristique trouve un solution à 407 capteurs (pour un minorant à 340.52).

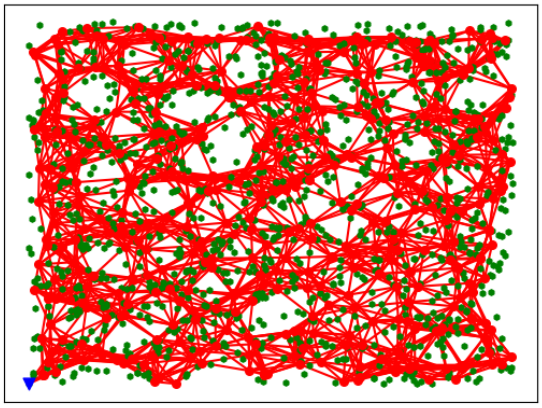

Figure 5: Instance de 1500 cibles avec $K = 3$, $R_{comm} = 2$, $R_{capt} = 1$

3 Algorithme génétique

Pour affiner les résultats de l'heuristique, nous avons décidé d'adapter une métaheuristique évolutionnaire.

3.1 Population initiale

Nous créons notre population initiale en ajoutant aléatoirement des capteurs (au nombre de 10% de la taille de l'instance) à la solution générée par l'heuristique. De cette manière nous sommes sûrs de partir d'une population de solutions admissibles mais que la population est assez diverse.

3.2 Évolution des populations

Dans une première partie la population évolue au gré des cross-overs (de taille maîtrisée) et mutations (de taille et de fréquence maîtrisée) puis elle est réduite avec une étape de sélection naturelle qui élimine les individus les moins bons (dans une proportion maîtrisée).

Pour les cross-overs, nous effectuons des petits échanges sur quelques gènes afin de profiter de la rapidité de calcul pour des petits changements (par exemple pour l'instance de 150 cibles, nous en faisons 3).

Pour les mutations mêmes idées, nous faisons des petits changements : quelques délétions et un peu moins d'insertions de gènes (par exemple pour l'instance de cibles, nous faisons 3 délétions et une insertion). La proportion de mutation était de 1%.

3.3 Pénalisation

Pour pénaliser les solution non-admissibles, nous avons décidé d'estimer l'écart de celles-ci sur chacune des conditions d'admissibilité que sont la k-captation et la connexité du graphe de communication engendré par les capteurs.

Pour estimer l'écart à la condition de k-captation, nous avons deux cas de figure :

- l'individu est issu d'une solution admissible : il nous suffit de compter, dans les capteurs que l'on veut supprimer, le nombre de capteurs dont un ou plus des voisins n'est pas k-captés
- l'individu n'est pas issu d'une solution admissible, on compte le nombre de cibles n'étant pas k-captées (si une cible n'est captée qu'une fois pour la 3-captation on augmente de 2 la pénalisation)

Cette estimation majore l'écart à l'admissibilité.

Pour estimer l'écart à la condition de connexité du graphe de communication engendré par les capteurs, on calcule le nombre de composantes connexes de ce graphe. Pour des solutions non-admissibles mais proche de l'admissibilité, cette estimation est sensée. Cependant si l'on s'écarte trop de l'admissibilité, cette estimation sera bien trop petite.

La pénalisation totale est une fonction linéaire des deux quantités présentées précédemment, des coefficients λ_{capt}, λ_{comm} sont associés aux deux quantités qui mesurent l'écart aux conditions d'admissibilité. Nous avons choisi fixé ces coefficients à $\lambda_{capt} = 2$ pour pénaliser la non satisfaction de la k-captation (nous voulions que coefficient soit plus grand que 1 mais pas trop grand car l'estimation d'écart est un majorant parfois très grand) et $\lambda_{capt} = 4$ pour la non-satisfaction de communication (nous voulions aussi ce coefficient plus grand que 1 et comme l'estimation de l'écart est un minorant, nous voulions être sûrs de favoriser les individus admissibles de la population).

Comme nous avons remarqué que la population convergeait rapidement et que peu de solutions non admissibles étaient gardées, nous avons rendus les coefficients λ_{capt}, λ_{comm} dynamiques en fonction de la proportion de solutions non-admissibles dans la population, en les multipliant par l'exponentielle du taux de solutions non admissibles à chaque génération.

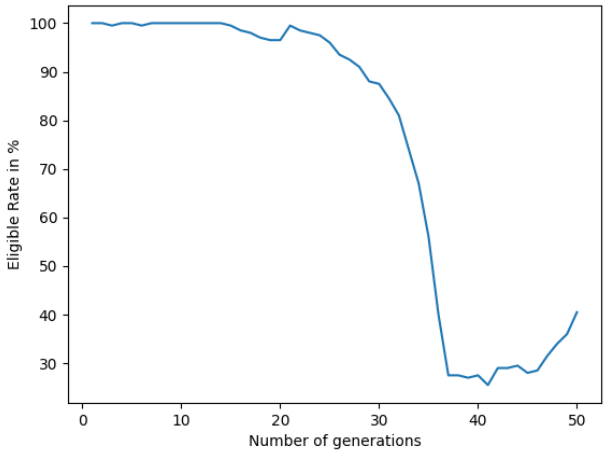

Figure 6: Évolution du taux d'individus admissibles avec les générations

3.4 *Social Disaster Technique*

Comme nous avions toujours un problème de convergence rapide des solutions, nous avons mis en place la *Social Disaster Technique* (une technique introduite par Kureichick[1] en 1996).

Le principe de cette technique est de déterminer quand il y a une perte de diversité dans la population et dans ce cas deux possibilités sont envisagées pour réintroduire de la diversité :

- le *packing* : sur les individus de même valeur de fitness, seul un est inchangé et le reste subit une mutation
- le *judgment day* : seul l'individu de meilleur valeur fitness ne subit pas de mutation

Ci-dessous il est possible de voir l'évolution des métriques de la population au cours des itérations de l'algorithme génétique, les applications du *packing* peuvent être observés : ce sont les pics dans la fonction fitness. Les mutations faites pour le *packing* sont les mêmes que celles réalisées dans le processus habituel d'évolution de la population.

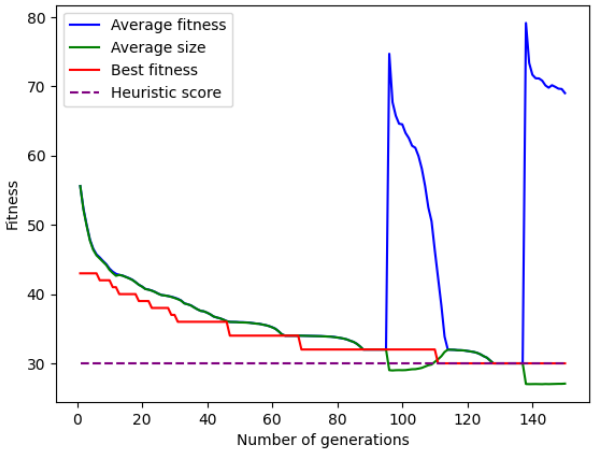

Figure 7: Évolution de diverses métriques au cours des évolutions

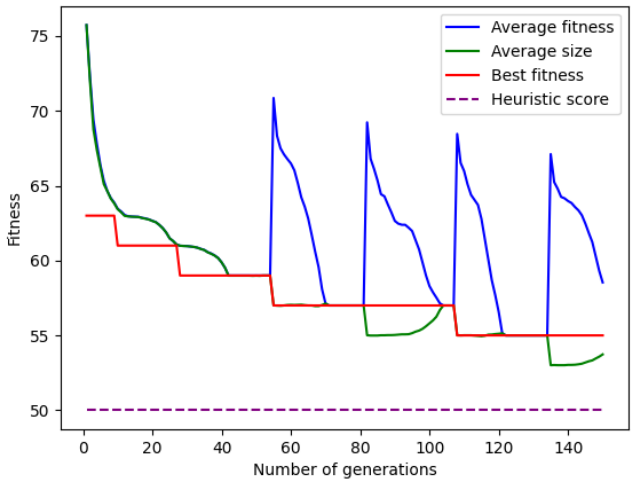

Figure 8: Évolution de diverses métriques au cours des évolutions pour une autre instance

4 Résultats (heuristique seulement)

4.1 Tableaux de résultats

Pour l'instance captANOR150_7_4 :

K	R_{comm}	R_{capt}	Taille de la solution trouvée	Temps d'exécution (en s)
1	1	1	36	0.151
1	2	1	21	0.207
1	2	2	12	0.225
1	3	2	8	0.299
2	1	1	50	0.447
2	2	1	43	0.665
2	2	2	14	0.399
2	3	2	13	0.565
3	1	1	63	0.501
3	2	1	62	0.801
3	2	2	19	0.536
3	3	2	19	0.739

Pour l'instance captANOR225_8_10 :

K	R_{comm}	R_{capt}	Taille de la solution trouvée	Temps d'exécution (en s)
1	1	1	50	0.489
1	2	1	30	0.73
1	2	2	15	0.703
1	3	2	11	0.993
2	1	1	61	1.251
2	2	1	56	1.817
2	2	2	20	1.145
2	3	2	18	1.571
3	1	1	83	1.104
3	2	1	83	1.59
3	2	2	28	1.275
3	3	2	28	1.767

Pour l'instance captANOR625_12_100 :

K	R_{comm}	R_{capt}	Taille de la solution trouvée	Temps d'exécution (en s)
1	1	1	110	6.477
1	2	1	70	10.968
1	2	2	35	7.153
1	3	2	23	9.726
2	1	1	140	15.429
2	2	1	126	18.324
2	2	2	46	18.241
2	3	2	40	21.073
3	1	1	185	16.832
3	2	1	180	20.346
3	2	2	62	16.189
3	3	2	61	21.693

Pour l'instance captANOR900_15_20 :

K	R_{comm}	R_{capt}	Taille de la solution trouvée	Temps d'exécution (en s)
1	1	1	171	20.379
1	2	1	105	24.817
1	2	2	45	19.444
1	3	2	36	29.24
2	1	1	209	36.668
2	2	1	186	47.752
2	2	2	65	35.722
2	3	2	61	46.819
3	1	1	277	44.7
3	2	1	276	53.622
3	2	2	87	39.608
3	3	2	87	51.597

Pour l'instance captANOR1500_15_100 :

K	R_{comm}	R_{capt}	Taille de la solution trouvée	Temps d'exécution
1	1	1	177	74.08
1	2	1	122	100.598
1	2	2	50	90.986
1	3	2	36	116.965
2	1	1	224	154.423
2	2	1	207	210.781
2	2	2	74	180.781
2	3	2	69	229.056
3	1	1	301	174.839
3	2	1	302	211.439
3	2	2	98	192.857
3	3	2	98	250.162

Pour l'instance captANOR1500_18_100 :

K	R_{comm}	R_{capt}	Taille de la solution trouvée	Temps d'exécution
1	1	1	239	71.053
1	2	1	150	81.358
1	2	2	68	76.912
1	3	2	52	99.155
2	1	1	310	158.879
2	2	1	281	237.468
2	2	2	97	169.102
2	3	2	90	199.353
3	1	1	413	201.801
3	2	1	407	250.444
3	2	2	136	169.073
3	3	2	133	215.415

4.2 Éléments d'analyses

Nous avons remarqué que pour les minorants, la valeur pour $R_{comm} = R_{capt}$ est quasiment égale à $R_{comm} = R_{capt} + 1$ alors que cela variait grandement pour nos résultats. Nous avons donc décidé de présenter les résultats séparément :

- pour $R_{comm} = R_{capt}$, nous obtenons des résultats en moyenne égaux à 155% du minorant
- pour $R_{comm} = R_{capt} + 1$, nous obtenons des résultats en moyenne égaux à 130% du minorant

5 Références

[1] V.Kureichick, A.N.Melikhov, V.V.Miaghick, O.V.Savelev and A.P.Topchy, Some New Features in the Genetic Solution of the Traveling Salesman Problem. In Ian Parmee and M.J.Denham eds. *Adaptive Computing in Engineering Design and Control 96(ACEDC'96), 2nd International Conference of the Integration of Genetic Algorithms and Neural Network Computing and Related Adaptive Computing with Current Engineering Practice*, Plymouth, UK, March 1996.