INSTITUT POLYTECHNIQUE DE PARIS

le cnam

Institut Polytechnique de paris

Major : Master2 MPRO

Academic year : 2021-2022

Projet en métaheuristiques : K-couverture connexe

Author :
Justine De Sousa
Yue Zhang

Enseignant :
Agnès Plateau

Table des matières

1 Caractérisation du problème

Étant donné un certain nombre de cibles placées dans un plan, dont une cible particulière nommée "puits", on souhaite placer un nombre minimal de capteurs sur certaines de ces cibles qui couvrent l'ensemble du plan et forme un ensemble connexe avec le puits. Plus précisément, les capteurs ont certaines propriétés de couverture et de communication. Il s'agit alors, avec un nombre minimal de capteurs, de respecter les deux contraintes suivantes :

k - couverture : Étant donné un rayon de couverture appelé R_{capt} pour l'ensemble des capteurs. Cette contrainte se traduit par le fait que chacune des cibles placée dans le plan sera couverte par au moins k capteurs. C'est-à-dire qu'elle contiendra au moins k capteurs dans une boule centrée sur elle-même et de rayon R_{capt}.

connexité : On définit également un rayon de communication R_{com} pour l'ensemble des capteurs. On dit que deux capteurs communiquent entre eux s'ils sont distants d'au plus R_{com}. On souhaite alors que l'ensemble des capteurs forment un ensemble connexe avec le puits. C'est-à-dire que chacun des capteurs est relié par l'intermédiaire d'un certains nombre de capteurs au puits.

Pour résoudre ce problème, nous sommes munies d'un certain nombre d'instance à résoudre par le biais d'une métaheuristique. Nous disposons de deux types d'instances :

Les instances aléatoires : Les cibles de ces instances sont générées aléatoirement dans un plan. Il s'agit de résoudre ces instances pour des paires $(Rcapt, Rcom) \in \{(1,1),(1,2),(2,2),(2,3)\}$ avec $k \in \{1,2,3\}$ et le nombre de cibles varie de 150 à 1500.

Affichage des cibles aléatoires

Les instances tronquées : Les cibles de ces instances sont placées aux intersections d'une grille carrée dont on a supprimé quelques points. Il s'agit de résoudre ces instances pour des paires $(Rcapt, Rcom) \in \{(1,1),(1,2),(2,2),(2,3)\}$ avec $k = 1$ et les tailles de grilles varient de 10×10 à 40×40.

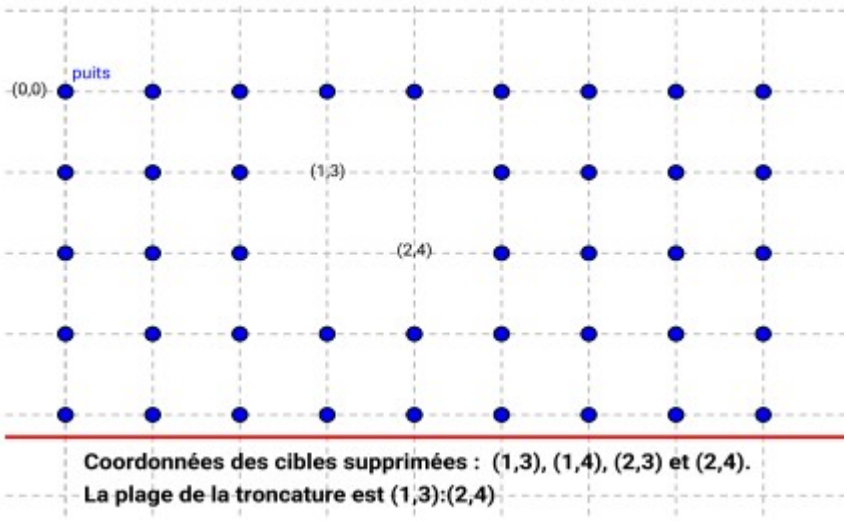


FIGURE 1 – Instance tronquée - grille 9×5

Le puits ne nécessite pas d'être couvert.

2 Heuristique

Dans un premier temps, le travail effectué consiste à écrire une heuristique qui génère des solutions réalisables de façon aléatoire. Pour ce faire, on construit deux graphes :

Graphe de captation : Une cible et un capteur sont reliés s'ils sont distants d'au plus R_{capt}. Dans une solution réalisable, chaque sommet possède au moins k capteurs voisins .

Graphe de communication : Les sommets sont les capteurs et deux sommets sont reliés s'ils sont distants d'au plus R_{com}. Dans une solution réalisable, l'ensemble des sommets qui possède un capteurs (+ le puits) forme une composante connexe du graphe.

Construire une solution réalisable est alors un exercice aisé puisqu'il suffit de poser un capteur sur chacune des cibles. Dans ce cas, on est certain que chaque cible sera couverte au moins k fois et l'ensemble des capteurs forme un ensemble connexe puisque tous les sommets sont reliés entre eux. Il s'agit alors d'améliorer cette solution qui contient des capteurs sur l'ensemble de ses cibles.

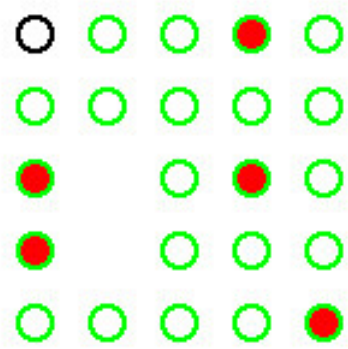
Stratégie de l'heuristique : Partant d'une solution réalisable, l'heuristique consiste à sélectionner une cible occupée par un capteur de façon aléatoire. Le capteur est retiré de la solution. Si la solution reste réalisable, alors on obtient une nouvelle solution strictement meilleure que la première. On peut ensuite itérer ce procédé jusqu'à obtenir une solution non réalisable. Dans ce cas, on replace le capteur et on obtient une solution réalisable strictement meilleure que la première.

Algorithme 1 : HEURISTIQUE(*Instance*)

Entrées : Instance = (cibles, R_{capt}, R_{com}, k).
Sorties : Une solution réalisable S.

```
1 S ← (0, 1, 1, ..., 1), i ;
2 tant que S est_réalisable faire
3 |   i ← indice aléatoire entre 1 et |S| ;
4 |   S[i] ← 0 ;
5 S[i] ← 1 ;
6 retourner S ;
```

Remarque : Les solutions générées par l'heuristique sont toujours réalisables.



Légende :

- le puits en haut à gauche,
- les capteurs en rouge,
- les cibles K(=1)-captées entourées en vert.

FIGURE 2 – Une solution générée par l'heuristique

3 Structure de voisinage

Étant donnée une solution $s \in \{0,1\}^N$ (non nécessairement réalisable), on considère le voisinage suivant :

$$\mathcal{V}(s) = \left\{ s' \in \{0,1\}^N \;\middle|\; \sum_{i=1}^{N} s_i = \sum_{i=1}^{N} s'_i, \text{ et } s \neq s' \right\}$$

C'est-à-dire l'ensemble des solutions qui contiennent le même nombre de capteurs que s (avec $s \neq s'$. Intuitivement, il s'agit d'agencer les capteurs différemment.

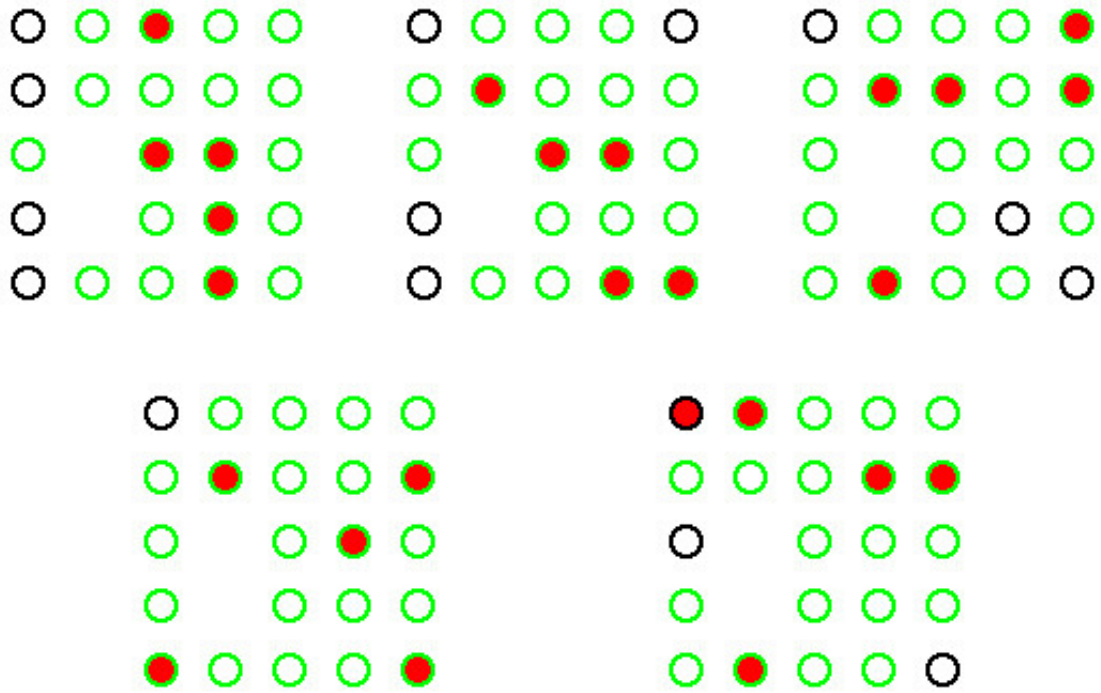


FIGURE 3 – 5 solutions dans le voisinage de la solution heuristique figure 2

Ce voisinage est utilisé dans la métaheuristique évolutionnaire lorsque la population de solutions devient trop homogène pour former de nouveaux individus intéressants. On remplace alors ces solutions identiques par des solutions du voisinage de la meilleure solution actuelle.

4 Méta-heuristique

Suite à la lecture de nombreux articles sur le sujet de la k-couverture-connexe, le choix de la métaheuristique s'est portée sur un algorithme génétique.

4.1 Algorithme Génétique

L'algorithme génétique consiste à reproduire le principe de sélection naturelle observé en biologie. En effet, en biologie, sur une échelle de temps suffisamment grande, il se trouve que les meilleures caractéristiques des individus d'une population sont conservées au fil des générations tandis que les moins bonnes caractéristiques sont éliminées naturellement soit parce que les individus survivent moins bien, soit parce qu'ils se reproduisent moins. L'algorithme utilisé est présenté Algorithm 2.

Algorithme 2 : AlgorithmeGenetique(P, S_{min})

Données : paramètres fixés N, *type_selection*, *rep_rate*, *mut_rate*, vie_{max}, $temps_{max}$
Entrées : Population initiale P, et sa meilleure solution S_{min}
Sorties : Population P et la meilleure solution rencontrée S_{min}.

```
1  temps_init ← time();
2  tant que temps < temps_max faire
3      pour chaque S ∈ P faire
4          S.vie++ ;
5      parents ← selection(P, ⌊rep_rate * N⌋, type_selection) ;
6      pour i=0,...,⌊|parents|/2⌋ faire
7          P1 ← parents[i];
8          P2 ← parents[i + ⌊|parents|/2⌋ ];
9          enfants ← enfants ∪ cross_over(P1, P2);
10         si /enfants/ = N - /parents/ alors break;
11     pour chaque S ∈ enfants faire
12         mutation(S, mut_rate) ;
13     P ← parents ∪ enfants;
14     si |P| < N alors
15         P ← P ∪ voisinage(S_min, N-|P|);
16     pour chaque S ∈ P faire
17         si S.vie ⩾ vie_max alors
18             S ← solution_voisinage(S_min);
19     si P.meilleure_solution() < S_min alors
20         S_min ← P.meilleure_solution()
21     S_min.vie = 0 ;
22     temps ← time() - temps_init;
23 retourner P, S_min
```

4.1.1 Principe général

La population : Dans ce cas précis, la population sera composée de différentes solutions (non nécessairement réalisables). Dans le cas de la population initiale, on génère N solutions par le biais de l'heuristique génératrice de solutions réalisables.

La fonction *fitness* : Les solutions se voient attribuer une fonction *fitness* qui prend en compte le nombre de capteurs posés, le nombre de capteurs manquant sur chacune des cibles et le nombre de composantes connexes (i.e : la fonction objectif et la violation des contraintes). Afin de guider l'algorithme vers des solutions réalisables, les contraintes sont pénalisées encore plus sévèrement en leur appliquant un coefficient 2. La fonction fitness d'une solution s'écrit alors :

$$\texttt{fitness(s) = nb\_capteurs + 2*(nb\_composantes\_connexes-1) + 2} * \sum_i \texttt{nb\_capteurs\_manquant(i)}$$

Lorsque la solution est réalisable, cette fonction *fitness* donne exactement le nombre de capteurs.

La sélection des meilleurs solutions : Au fil des générations, différents types de solutions seront générées. L'idée générale est de conserver les meilleures solutions afin de générer des enfants qui gardent les meilleures caractéristiques.

La diversification des solutions : L'inconvénient de sélectionner toujours les meilleurs individus est qu'on risque de se retrouver avec une population très homogène. En effet, les croisements de parents ressemblants risquent de former des enfants toujours aussi ressemblants et cela empêche d'explorer de nouvelles structures de solutions. C'est pourquoi, on a recours à divers stratagèmes pour s'assurer d'avoir toujours une certaine diversité dans la population tels que la mutation des individus et la durée de vie de certaines solutions. Tout ceci sera expliqué en détail dans la sous-sous-section 4.1.2.

4.1.2 Les étapes de l'algorithme génétique

Critère d'arrêt : On laisse l'algorithme tourner 3 minutes. Le nombre d'itérations dépend de la taille de l'instance. Moins il y de cibles dans l'instance, plus l'algorithme fait d'itérations.

Les sélections : Il y a deux moments où on souhaite sélectionner les meilleurs individus :

- La sélection des reproducteurs
- La sélection des enfants

Deux types de sélection sont testées :

- La sélection ELITE : elle garde uniquement les meilleurs individus ;
- La sélection ROULETTE : les individus sont sélectionné aléatoirement avec une probabilité qui dépend de sa fonction fitness. Son nom vient du fait qu'on peut représenter ce tirage comme étant sur une roue (cf figure 4).

Pour la solution i, on a :

$$\mathbb{P}(\text{sélectionner } i) = \frac{f_i}{\sum_i f_i}$$

FIGURE 4 – La sélection roulette

Les sélections nécessitent de comparer les individus entre eux. Cela se fait via la biais de la fonction fitness.

Le cross-over : Le passage d'une génération à une autre passe par la reproduction des parents. Cette reproduction est représentée par un cross-over (encore par analogie avec la biologie). Il s'agit alors de "mélanger" deux parents afin de former deux nouveaux individus enfants. Ici l'opérateur choisi est défini de la façon suivante.

On définit une plage de la grille (un bit-mask) de façon aléatoire (la largeur de la plage est fixée à 0.5*nb-bit et le coin supérieur gauche est pris aléatoirement). Le parent P1 donne sa structure à l'enfant E1 et le parent P2

donne sa structure à l'enfant E2. Le croisement vient du fait que le bit-mask du parent P2 se retrouvera dans l'enfant E1 et inversement. Ceci est illustré figure 5.

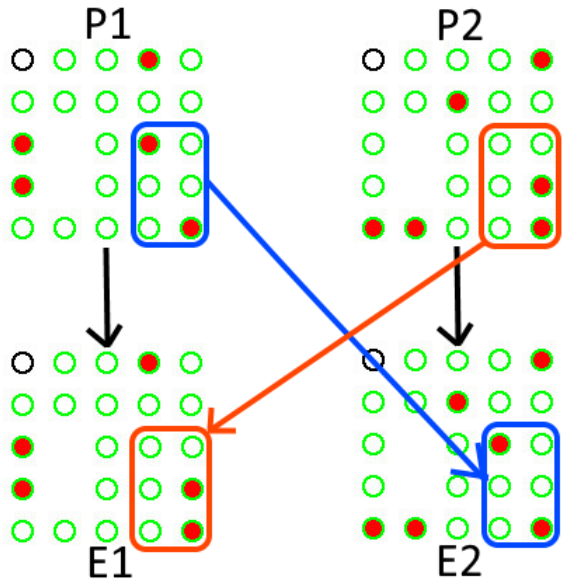


FIGURE 5 – Cross-over entre P1 et P2

Le recours au voisinage : Lors du croisement des individus, on s'assure que deux individus identiques ne se croisent pas. Ainsi, si la population est très homogène, il se peut qu'aucun enfant ne soit formé (ce qui arrive peu dès lors que la taille de la population est suffisamment grande). Si cela se produit, on choisit alors suffisamment d'individus dans le voisinage de la meilleure solution qu'on intègre à notre population afin de garder une population de taille constante.

La mutation : Une fois les enfants générés ceux-ci ont tous une certaine probabilité de muter. Ici, la mutation consiste à inverser un bit de la solution en question. Plus le taux de mutation est élevé et plus on a de chance de produire une population diversifiée. Néanmoins, nous ne souhaitons pas perdre nos bon individus. Il faut donc faire attention au réglage de ce paramètre. On notera tout de même que le meilleur individu de chaque population est conservé dans la population au fil des générations.

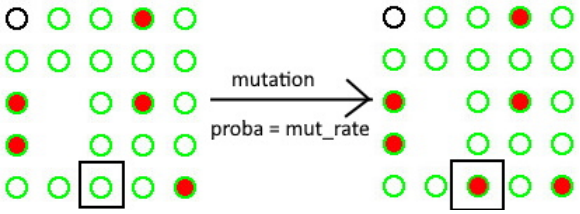


FIGURE 6 – Mutation d'une solution avec proba = mut_rate

Durée de vie des individus : On fixe également une durée de vie à chaque individu. Dès lors qu'un individu atteint l'âge maximal autorisé, celui-ci meurt, c'est-à-dire qu'il est éliminé de la population. Afin de garder un nombre suffisant d'individus dans la population, on génère ensuite un nombre suffisant d'individus dans le voisinage de la meilleure solution.

Remarque : à chaque début et fin d'itération, la population manipulée contient un nombre N d'individus. La dernière population renvoyée contient donc le même nombre d'individus que dans la première population.

4.2 Choix des paramètres

L'algorithme génétique utilisé contient de nombreux paramètres. Ces paramètres ont été fixés de manière arbitraire avec un peu de bon sens dans un premier temps. Ensuite, différents jeux de paramètres sont testés et comparés afin d'affiner au mieux les résultats. Les tests sont effectués sur l'instance `grille2525_1` et $R_{capt} = R_{com} = 1$ avec une fenêtre de temps de 3 min.

4.2.1 La taille de la population

Plusieurs tailles de population sont testées, on affiche les résultats dans tableau 1.

N	nb capteurs minimal	moyenne sur 5 exécutions	nb itérations en moyenne
50	206	208	148
100	205	207	67
150	207	207	40

TABLE 1 – Comparaison de différentes tailles de population sur l'instance grille2525_1

On observe que la meilleure solution trouvée et la moyenne des solutions renvoyées ne sont pas significativement différente en fonction de la taille de la population. Néanmoins, lorsque la population est petite, cela permet à l'algorithme de faire plus d'itérations. On sélectionne ici $N = 100$ qui est le paramètre qui a donné le meilleur résultat.

4.2.2 Le type de sélection

Les selection ELITE et ROULETTE sont testées, on affiche les résultats dans tableau 2. La sélection ELITE

Type de sélection	nb capteurs minimal	moyenne sur 5 exécutions	nb itérations en moyenne
ROULETTE	207	209	69
ELITE	196	199	71

TABLE 2 – Comparaison de la sélection ROULETTE avec la sélection ELITE sur l'instance grille2525_1

qui consiste à toujours garder les meilleurs individus semble surpasser la sélection ROULETTE de quelques points. C'est le type de sélection qui sera alors conservé pour la suite.

4.2.3 La durée de vie des individus

Différentes durées de vie sont testées, on affiche les résultats dans le tableau 2. On constate que qu'en fixant le paramètre VIE_Max à 10 ou à 50, on obtient des résultats similaires. Néanmoins, avec VIE_Max = 50, les solutions renvoyées sont plus souvent réalisables. C'est donc ce paramètre qui est conservé pour la suite.

VIE_MAX	nb capteurs minimal	moyenne sur 5 exécutions	nb itérations en moyenne
5	196	200	83
10	193	196	81
30	196	199	85
50	192	197	88
70	196	198	80

Table 3 – Comparaison de différentes durées de vie sur l'instance grille2525_1

4.2.4 Le taux de reproducteurs dans la population

On teste différents taux de reproduction, c'est-à-dire différents taux du nombre de reproducteurs et on affiche les résultats dans tableau 4. Les meilleurs résultats semblent subvenir lorsqu'on sélectionne N/2 individus

taux de reproduction	nb capteurs minimal	moyenne sur 5 exécutions	nb itérations en moyenne
0.2	201	203	91
0.5	192	196	84
0.7	197	198	89

Table 4 – Comparaison de différents taux de reproduction sur l'instance grille2525_1

reproducteurs dans la population. Cela signifie également que les cross_over génère N/2 enfants et qu'il n'y a donc aucune sélection parmi les enfants.

4.2.5 Le taux de mutation

Différents taux de mutation sont testés, on affiche les résultats dans tableau 5. Lorsqu'on fixe le taux de

taux mutation	nb capteurs minimal	moyenne sur 5 exécutions	nb itérations en moyenne
0.05	194	198	67
0.2	191	194	83
0.5	194	196	76
0.7	192	196	72

Table 5 – Comparaison des taux de mutation sur l'instance grille2525_1

mutation à 20%, on semble obtenir les meilleurs résultats. C'est également le taux qui permet de faire plus d'itérations sur la plage de temps de 3 min. On fixe dorénavant le paramètre mut_rate à 0.2.

5 Implémentation en C++

L'implémentation de l'algorithme génétique a été fait en C++. C'est la partie qui a demandé le plus de temps. Les différentes classes utilisées sont décrites brièvement dans cette section.

5.1 La classe Instance

Cette classe hérite de la classe `vector<vector<float>>` qui est en fait le stockage des distance entre chaque paire de sommets de la grille.

Deux classes filles sont également implémentées. Une classe `Instance_alea` et une classe `Instance_tronc` qui contiennent respectivement un `vector< pair<float,float> > cibles` et un `vector< pair<int,int> > cibles`. L'initialisation de ces objets se fait grâce à un constructeur qui lit un fichier d'instance.

5.2 La classe Solution

Cette classe hérite de la classe `vector<bool>` et contient un attribut statique `const Instance* instance`. Elle stocke la valeur `1` en position `i` si un capteur est posé sur la `i`-ième cible, `0` sinon. Elle contient également deux attributs de type `Graph` : `graph_capt` et `graph_com`. qui permette d'évaluer si une solution est réalisable et de calculer sa fonction fitness.

5.3 La classe Graph

La classe Graph hérite de la classe `vector<set<int>>` représentant la liste adjacence du chaque sommet. La classe Graph a un attribut `graph_type` pour distinguer les deux type de réseau (captation et communication). Dans cette classe, on effectue toutes les opérations nécessaires concernant les graphes non-orientés. Par exemple, le fait d'ajouter un capteur, de vérifier la k-couverture et de compter le nombre de composantes connexes du réseau de communication.

5.4 La classe Population

Elle hérite de la classe `vector<Solution>` et permet de faire les opérations nécessaires à l'algorithme génétique telles que les sélections.

6 Résultats

Dans cette section, on affiche les résultats obtenus pour chacune des instances proposées. Les tests sont effectués sur une machine virtuelle Linux utilisant le CPU `Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz` du 4 processors.

Le temps d'exécution a été fixé à 3 minutes et on a exécuté l'algorithme 5 fois pour chacune des instances. Sur ces 5 itérations, on retient :

- le nombre de capteurs de la meilleure solution de la population de départ (générée avec l'heuristique),
- le nombre de capteurs de la meilleure solution rencontrée lors des 5 exécutions,
- le nombre de capteurs moyen sur ces 5 exécutions,
- le nombre d'itérations moyenne de l'algorithme génétique.

6.1 Commentaire des résultats

On observe que les solutions renvoyées par l'algorithme génétique sont meilleures que les solutions renvoyées par l'heuristique de génération de solutions tout en restant proche de cette solution de départ.

Les grandes instances requièrent beaucoup plus de temps de calcul par itération. En 3 min, certaines grandes instances ne peuvent aller au delà de 20 itérations.

Par exemple, pour la petite instance `captANOR150_7_4`, la solution renvoyée par l'algorithme génétique est assez proche du minorant obtenue par résolution d'un programme linéaire relaché. En revanche, pour une grande instance telle que `captANOR1600_16_100_2021` l'algorithme fait moins de 20 itérations et la solution renvoyée est loin de la borne du PL. L'algorithme n'évolue que très peu en seulement 20 itérations.

On observe également que les résultats obtenus semblent beaucoup plus proche de la borne du PL lorsque `R_capt` et `R_com` sont grands.

6.2 Perspectives

On pourrait penser à améliorer l'heuristique de génération de solutions en supprimant les capteurs dans l'ordre croissant du nombre de cibles couvertes, intuitivement, on préfère garder les capteurs qui couvrent le plus de cibles.

L'algorithme génétique est sensible à la population initiale, on pourrait donc imaginer améliorer l'heuristique afin d'avoir une population initiale plus diversifiée.

Les paramètres ont été choisis en se basant sur les résultats obtenus sur une instance particulière (`grillle2525_1`). L'algorithme étant très sensible aux paramètres, il pourrait être intéressant d'affiner ce choix de paramètres en testant des instances différentes. Il serait peut-être intéressant également de choisir des paramètres différents en fonction de la taille de l'instance.

Étant donné que l'algorithme génétique renvoie très souvent une solution réalisable, la nécessité d'écrire une heuristique de réparation ne s'est pas présentée. C'est pourquoi, on a décidé de ne pas passer plus de temps pour finir son implémentation, une trace est tout de même disponible dans le fichier `main.cpp`.

Dans la fonction fitness, le terme correspondant à la violation des contraintes possède un coefficient 2. Cela permet d'obtenir plus souvent des solutions réalisables qu'avec un coefficient 1. Il pourrait être intéressant d'affiner plus précisément ce coefficient afin d'équilibrer les poids entre le nombre de capteurs et les termes de violation des contraintes.

Il serait aussi intéressant de faire tourner l'algorithme plus longtemps que 3 min, notamment sur les grandes instances.

Un autre objectif serait également de faire de la programmation parallèle, notamment lorsqu'il s'agit d'évaluer tous les individus d'une population et les trier (c'est le tri qui prend le plus de temps).

Instance	Rcapt	Rcom	Résultats			
			Heuristique	Meilleure	Moyenne	itérations moyenne
grille1010_1_	1	1	39	**36**	36	544
grille1010_1_	1	2	30	**29**	29	618
grille1010_1_	2	2	18	**17**	17	615
grille1010_1_	2	3	13	**12**	12	610
grille1010_2_	1	1	42	**39**	39	492
grille1010_2_	1	2	32	**30**	30	537
grille1010_2_	2	2	19	**17**	18	533
grille1010_2_	2	3	14	**13**	13	628
grille1515_1_	1	1	92	**87**	88	188
grille1515_1_	1	2	67	**65**	66	266
grille1515_1_	2	2	42	**39**	41	265
grille1515_1_	2	3	29	**28**	28	312
grille1515_2_	1	1	97	**91**	91	235
grille1515_2_	1	2	71	**68**	69	265
grille1515_2_	2	2	45	**43**	43	257
grille1515_2_	2	3	30	**30**	30	321
grille2020_1_	1	1	153	**142**	147	127
grille2020_1_	1	2	114	**112**	112	145
grille2020_1_	2	2	73	**67**	68	144
grille2020_1_	2	3	49	**46**	47	178
grille2020_2_	1	1	174	**168**	169	117
grille2020_2_	1	2	125	**122**	123	131
grille2020_2_	2	2	80	**77**	78	133
grille2020_2_	2	3	54	**52**	53	162
grille2525_1_	1	1	206	**191**	197	89
grille2525_1_	1	2	152	**150**	151	101
grille2525_1_	2	2	99	**90**	92	101
grille2525_1_	2	3	67	**64**	64	125
grille2525_2_	1	1	274	**266**	270	65
grille2525_2_	1	2	197	**195**	196	75
grille2525_2_	2	2	125	**121**	123	74
grille2525_2_	2	3	85	**82**	83	95
grille3030_1_	1	1	311	**301**	307	51
grille3030_1_	1	2	225	**224**	225	61
grille3030_1_	2	2	143	**139**	141	58
grille3030_1_	2	3	97	**95**	96	72
grille3030_2_	1	1	394	**389**	392	39
grille3030_2_	1	2	283	**279**	281	46
grille3030_2_	2	2	182	**177**	179	48
grille3030_2_	2	3	122	**119**	121	61
grille4040_1_	1	1	571	**565**	567	24
grille4040_1_	1	2	408	**407**	408	28
grille4040_1_	2	2	260	**253**	257	27
grille4040_1_	2	3	177	**172**	175	34
grille4040_2_	1	1	704	**704**	705	17
grille4040_2_	1	2	502	**500**	501	21
grille4040_2_	2	2	322	**316**	322	22
grille4040_2_	2	3	216	**214**	215	28

TABLE 6 – Résultats pour les instances de grilles tronquées

Instance	K	Rcapt	Rcom	Résultats				Borne PL
				Heuristique	Meilleure	Moyenne	itérations moyenne	
captANOR150_7_4_	1	1	1	35	**31**	32	355	20.42
captANOR150_7_4_	1	1	2	20	**20**	20	429	18.5
captANOR150_7_4_	1	2	2	9	**9**	9	424	5.74
captANOR150_7_4_	1	2	3	7	**7**	7	495	5.5
captANOR150_7_4_	2	1	1	48	**45**	46	310	39.28
captANOR150_7_4_	2	1	2	42	**40**	41	251	39
captANOR150_7_4_	2	2	2	14	**13**	13	360	11
captANOR150_7_4_	2	2	3	13	**12**	12	366	11
captANOR150_7_4_	3	1	1	65	**63**	63	237	60
captANOR150_7_4_	3	1	2	63	**62**	62	140	60
captANOR150_7_4_	3	2	2	19	**18**	18	281	17
captANOR150_7_4_	3	2	3	20	**19**	19	254	17
captANOR400_7_10_2021_	1	1	1	41	**37**	37	124	19.43
captANOR400_7_10_2021_	1	1	2	25	**23**	24	147	19.07
captANOR400_7_10_2021_	1	2	2	11	**10**	10	134	6.05
captANOR400_7_10_2021_	1	2	3	8	**7**	7	171	6
captANOR400_7_10_2021_	2	1	1	54	**53**	53	109	38.18
captANOR400_7_10_2021_	2	1	2	49	**48**	49	97	38.18
captANOR400_7_10_2021_	2	2	2	16	**15**	15	116	12
captANOR400_7_10_2021_	2	2	3	15	**14**	14	113	12
captANOR400_7_10_2021_	3	1	1	72	**71**	71	87	57.42
captANOR400_7_10_2021_	3	1	2	72	**70**	71	59	57.43
captANOR400_7_10_2021_	3	2	2	23	**22**	22	85	18
captANOR400_7_10_2021_	3	2	3	23	**22**	22	70	18
captANOR900_14_20_2021_	1	1	1	157	**146**	151	233	69.33
captANOR900_14_20_2021_	1	1	2	93	**88**	90	321	68.27
captANOR900_14_20_2021_	1	2	2	44	**39**	40	253	19.71
captANOR900_14_20_2021_	1	2	3	30	**27**	27	290	19.53
captANOR900_14_20_2021_	2	1	1	199	**193**	196	164	137.47
captANOR900_14_20_2021_	2	1	2	176	**169**	172	150	137.44
captANOR900_14_20_2021_	2	2	2	58	**54**	56	197	39.13
captANOR900_14_20_2021_	2	2	3	53	**52**	52	219	39.07
captANOR900_14_20_2021_	3	1	1	260	**256**	257	153	210.98
captANOR900_14_20_2021_	3	1	2	258	**254**	255	101	210.98
captANOR900_14_20_2021_	3	2	2	78	**75**	76	184	58.65
captANOR900_14_20_2021_	3	2	3	77	**76**	76	176	58.64
captANOR1600_16_100_2021_	1	1	1	210	**209**	210	21	89.73
captANOR1600_16_100_2021_	1	1	2	129	**127**	128	24	89.72
captANOR1600_16_100_2021_	1	2	2	57	**56**	56	21	24.90
captANOR1600_16_100_2021_	1	2	3	38	**37**	38	28	24.66
captANOR1600_16_100_2021_	2	1	1	269	**267**	269	18	178.71
captANOR1600_16_100_2021_	2	1	2	240	**239**	240	16	178.68
captANOR1600_16_100_2021_	2	2	2	76	**75**	76	19	49.37
captANOR1600_16_100_2021_	2	2	3	70	**68**	70	20	49.32
captANOR1600_16_100_2021_	3	1	1	350	**349**	350	15	269.30
captANOR1600_16_100_2021_	3	1	2	346	**345**	346	10	269.30
captANOR1600_16_100_2021_	3	2	2	102	**100**	102	16	75.05
captANOR1600_16_100_2021_	3	2	3	101	**101**	101	15	74.03

TABLE 7 – Résultats pour les instances de cibles aléatoires