PROBLÈME D'ACTIVATION DE CAPTEURS POUR SURVEILLANCE DE ZONES

I Construction des heuristiques

I.1 Explication des méthodes is_*valide()* et *is_elementaire()*

Tout d'abord, nous allons vous présenter deux méthodes utilisées dans toutes nos heuristiques. Pour l'ensemble de notre code nous avons créé un objet **Capteur** qui contient la durée de vie ainsi que les zones couvertes afin de faciliter la manipulation des données.

I**s_valide()** : La méthode **is_valide()** permet de vérifier si la solution proposée en paramètre couvre bien toutes les zones. Cette méthode parcourt tous les capteurs de la solution et ajoute dans un tableau les zones couvertes par ce capteur en enlevant les doublons. Cela nous permet de tester si la solution est valide grâce à la longueur du tableau et donc de savoir si toutes les zones sont couvertes.

is_elementaire() : La méthode **is_elementaire()** permet de savoir si il y a des capteurs inutiles dans la solution. Cette méthode utilise une double boucle sur les capteurs de la solution. L'idée de cette méthode est de vérifier si la solution couvre toutes les zones en enlevant un capteur. Ce test est réalisé pour chaque capteurs, ce qui permet de vérififier si il ya des capteurs inutiles et donc de savoir si la solution est élémentaire.

I.2 Glouton

La méthode **generer_combinaison_solution()** commence par une boucle sur le nombre de zones, en utilisant une variable i. Cette boucle permet de générer toutes les combinaisons possibles de capteurs pour couvrir les zones en utilisant la méthode **itertools.combinations()**. En effet, elle génère toutes les combinaisons possibles de i éléments parmi les capteurs fournis, sans tenir compte de l'ordre (ainsi, les combinaisons (1, 2) et (2, 1) sont considérées comme identiques). Ensuite, nous vérifions que chacune de ces combinaisons est valide puis élémentaire pour réduire le temps d'exécution. Si ces deux conditions sont respectées, la combinaison est ajoutée au tableau qui est retourné à la fin de la méthode. De plus, nous avons ajouté un timer à l'aide de la bibliothèque **time** dans l'objectif d'arrêter la méthode en cas de fichiers trop volumineux.

I.3 Récursion avec une liste taboue

Classique

Cette méthode repose sur les principes de récursion et de liste taboue. L'objectif est de générer un nombre maximum de solutions pertinentes au détriment de l'ensemble des solutions possibles.

Elle commence par boucler sur la liste de tous les capteurs. Si le capteur est dans la liste taboue nous appelons de manière récursive la méthode en augmentant l'index de départ de la première boucle pour passer au capteur suivant et éviter de prendre à nouveau des capteurs déjà visités. Dans le cas où le capteur n'est pas dans la liste taboue, nous l'ajoutons à la solution actuelle. Puis nous testons si la solution actuelle est valide et élémentaire à l'aide des méthodes présentées ci-dessus. Si c'est le cas nous ajoutons la solution au tableau de solutions final et nous appelons à nouveau de manière récursive la méthode en changeant l'index de début de la grande boucle.

De la même manière que la méthode glouton, nous avons ajouté plusieurs paramètres (**nb_itération** et **timer_depart/time_limit)** dans le but de stopper la méthode dans le cas où les fichiers à traiter sont trop volumineux.

Tri croissant du nombre de zones couvertes par les capteurs

Cette heuristique consiste dans un premier temps à trier l'ensemble des capteurs par ordre croissant du nombre de zones qu'ils couvrent. Puis, on utilise la même méthode que dans la section ci-dessus mais avec ce tableau de capteurs triés.

Tri décroissant du nombre de zones couvertes par les capteurs

De la même manière que l'heuristique précédente, nous commençons cette fois-ci par trier les capteurs par ordre décroissant avant d'exécuter la méthode classique.

II Solutions obtenues

Nom Fichier / Méthodes	Fichier-exemple			Moyen test 3			Moyen test 2			Gros test 1			Maxi test		
	Max	Nb conf	Temps (s)	Max	Nb conf	Temps (s)	Max	Nb conf	Temps (s)	Max	Nb conf	Temps (s)	Max	Nb conf	Temps (s)
Glouton	8,5	4	0,0003	395	19	0,0056	104	238	3,95	2793,8	12780	15,211	5680	7377	15,739
Récursion classique	8,5	4	0,0005	395	10	0,0009	55	13	0,0011	748	37	0,666	296	28	165
Récursion croissante	8,5	4	0,0005	345,5	7	0,0009	28	3	0,0008	1159,5	41	0,936	/	0	206,9
Récursion décroissante	8,5	4	0,0006	395	13	0,0008	51	10	0,0009	1190	141	0,463	82	199	0,231
													276	843	1,198

Case bleue : expériences réalisées en limitant le temps d'exécution à 15 sec (**Glouton**) ou à 2 sec (**Récursion décroissante**) pour cause de récursions trop nombreuses (maximum recursion depth exceeded)

III Analyse des résultats

Pour cette partie, nous allons faire une analyse de nos résultats fichier par fichier.

Pour le **fichier-exemple** toutes les méthodes ont réussi à obtenir la solution optimale. La méthode la plus performante en terme de temps d'exécution est la méthode **glouton** même si toutes les méthode sont assez proches.

Pour le fichier **moyen_test_3**, seule la méthode **récursion_croissante** n'a pas réussi à obtenir la solution optimale. Cependant, on remarque que la méthode **récursion_décroissante** est la plus rapide à s'exécuter suivi de près par la méthode de **récursion_classique**.

Pour le fichier **moyen_test_2**, on remarque que la méthode **glouton** est celle qui a obtenu le plus grand maximum, cependant elle à un temps d'exécution significativement plus long. Néanmoins, nous pensons que les autres méthodes sont trop éloignées de la valeur optimale trouvée et donc moins pertinentes dans ce contexte.

Pour le fichier **gros_test_1**, la méthode **glouton** est également la plus précise en terme de solution obtenue. Cependant, dans ce contexte la méthode à été arrêté à cause du timer pour limiter le temps d'exécution. On remarque que pour cette expérience la méthode **recursion_décroissante** est particulièrement efficace compte tenue de son temps d'exécution rapide.

Pour le fichier **maxi_test**, la méthode **glouton** est également la plus précise en terme de solution obtenue tout en étant également limité avec le timer. Pour ce test on remarque que la **récursion_croissante** a obtenu 0 configuration et donc n'a pas de valeur optimale malgré son temps d'exécution très élevé. Cela s'explique par le fait que les capteurs sont triés par ordre croissant, ce qui implique que la probabilité d'avoir une solution valide et élémentaire sur un très gros fichier est très faible au début de l'algorithme. C'est pourquoi l'exécution est arrêtée par le **nb_itération** avant que la méthode ait réussi à trouver une solution valable. Cela explique également pourquoi à l'inverse la **récursion_décroissante** arrive à trouver un certain nombre de solutions en peu de temps.

En général, nous avons trouvé que l'algorithme **glouton** est plus efficace avec un timer malgré son temps d'exécution plus long car la différence entre les valeurs obtenues est significative, en particulier pour les deux plus gros fichiers. En deuxième position, l'algorithme de **récursion_décroisssant** est plutôt performant, tout particulièrement lorsqu'il s'agit de fichiers volumineux. Son problème repose sur son trop grand nombre de récursions qui l'empêche de s'exécuter de manière optimale.