TP:

Résolution de problèmes de sac à dos

Guo Xuewen

Question 1:

MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports)

2.3 GHz Intel Core i5

gcc version 9.3.0

Question 3 :

KpSolverGreedy::solveUpperBound() de la classe KpSolverGreedy.

Cette fonction est la façon continue, c’est-à-dire que nous sommes autorisées de prendre une moitié d’objet. Donc si les objets ont déjà trié suivant l’ordre ci/mi décroissant, on va tester à partie de le premier objet, parce que cet objet est celui avec la value plus haut pour chaque masse et on fait ce test jusqu’à le sac il n’a plus de place ou on a déjà testé le dernier objet. Si la reste de place du sac peut charger cet objet en entier, alors la upperBoundOPT va ajouter la valeur de cet objet. Si le sac a encore la place mais ne peut pas charger l’objet en entier. Alors on va remplir notre sac par cet objet, et la upperBoundOPT est la value de cet objet en entier fois la place restée sur la masse d’objet.

Question 4:

KpSolverGreedy::solveLowerBound() de la classe KpSolverGreedy. Cette fonction est la façon discrète, c’est-à-dire nous ne pouvons pas prendre une moitié d’objet, on doit choisir prendre un entier d’objet ou ne prendre pas. Alors on a un tableau de solution, qui a la même taille de nombres d’objets. Et pour chaque case il va avoir un résultat de vrai ou false pour décider est ce qu’on va mettre cet objet dans le sac ou pas. Donc à partie du premier objet jusqu’à le dernier objet ou il n’a plus de place. Si la place du sac peut charger l’objet, alors on le prendre, on ajoute cette valeur et donner vrai à cette case de tableau solution, si la place du sac ne peut pas charger , on donne false à cette case de tableau solution et on passe au prochain.

Question 5:

On a ajouté d’abord une fonction getgap() dans la classe kpSolver, et plus dans le fichier testIterHeur .cpp, on ajoute afficher le gap, alors le fichier sortie iterHeur.csv va contenir le gap dans la colonne D. Et après, on peut conclure que la upperBoundOPT a la possibilité d’égale à la costSolution

# 3 . Algorithmes de programmation dynamique exacts

Question 6 :

On voudrait initialiser la matrice par la méthode récurrence sur la fonction solveIer() .D’abord par les contraintes pour tout i sur [0,nbItems],la case [i][0]avoir la valeur 0 car si la capacite de sac est 0 alors on peut prendre aucun objet, et même choses pour tout m sur [0, knapsackBound], la case [0][m]avoir la valeur 0. Puis pour les restes cases on va tester est-ce que la masse de cet objet est plus grand que la capacité de sac, si dans ce cas, on ne prend pas cet objet, sinon on va tester si on ne prend pas ou si on prend, quel cas on va avoir la plus grand de valeur.

Après on a initialisé la matrice, on va utiliser backtrack pour avoir le tableau de la solution. D’abord, on suppose qu’on a tous les objets disponibles , et on va faire la teste pour chaque fois on réduit un objet ,on va arrêter jusqu’à on a aucun objet. Chaque fois on va regarder si on ne prend pas cet objet on va avoir la même valeur, si les valeurs sont différents alors on va prendre cet objet , et donc on va réduire la taille du sac à dos.

Question7

D’abord, pour le fichier avec 1000 instances, on peut résoudre les problèmes à l’optimalité par programmation dynamique, mais pour 10000 , on ne peut plus.

Donc pour avoir la taille plus précise, on a utilisé extractSubInstance pour tester , chaque fois on ajoute 500 instances, et on a trouvé que quand instances est 4500, c’est hyper lent pour résoudre .Et l’optimal est plus près des bornes de relaxation discrète.

Question8

On a comparé les temps, et stocké le fichier dans csv, qui s’appelle Question8 .

·D’abord on voudrais voir la changement de temps quand on change la capacité de sac. Alors on a ajouté les fonction setKpBound() et getKpBound(), on a essayé de grandir 1000g chaque fois , et on va trouver que si la capacité est plus grande alors pour résoudre il va utiliser moins de temps.

Et pour chaque fois, le temps de la version programmation dynamique mémorisé est plus lente que la version de programmation dynamique itérative sans parallélisation.

Et la compilation la moins optimisée (flag -O0) est plus lente que la compilation la plus optimisée (flag -O3)

* Et pour la N , si la N est plus grande , alors il va utiliser plus de temps

# 4 Heuristique de programmation dynamique

Question10 :

*KpSolverHeurDP::solveUpperBound()* cette fonction est comme la fonction *KpSolverGreedy::solveUpperBound(),* avec stockage de l’indice critique *lastIndex*, qui est l’indice du premier objet non sélectionné alors on définir cet indice quand la première fois, la reste de taille n’est pas plus grand que 0.

Question11 :

D’abord, on rajoute deux fonctions sur KpSolver, elles sont getExtractedSubInstance et getSolution. La fonction getExtractedSubInstance définissent les caractéristiques du sac à dos, par exemples le nombre d’objet disponible, les masses et les valeurs des objets, et la taille du sac à dos. La fonction getSolution qui donne le tableau de booléen pour dire chaque objet est pris ou pas.

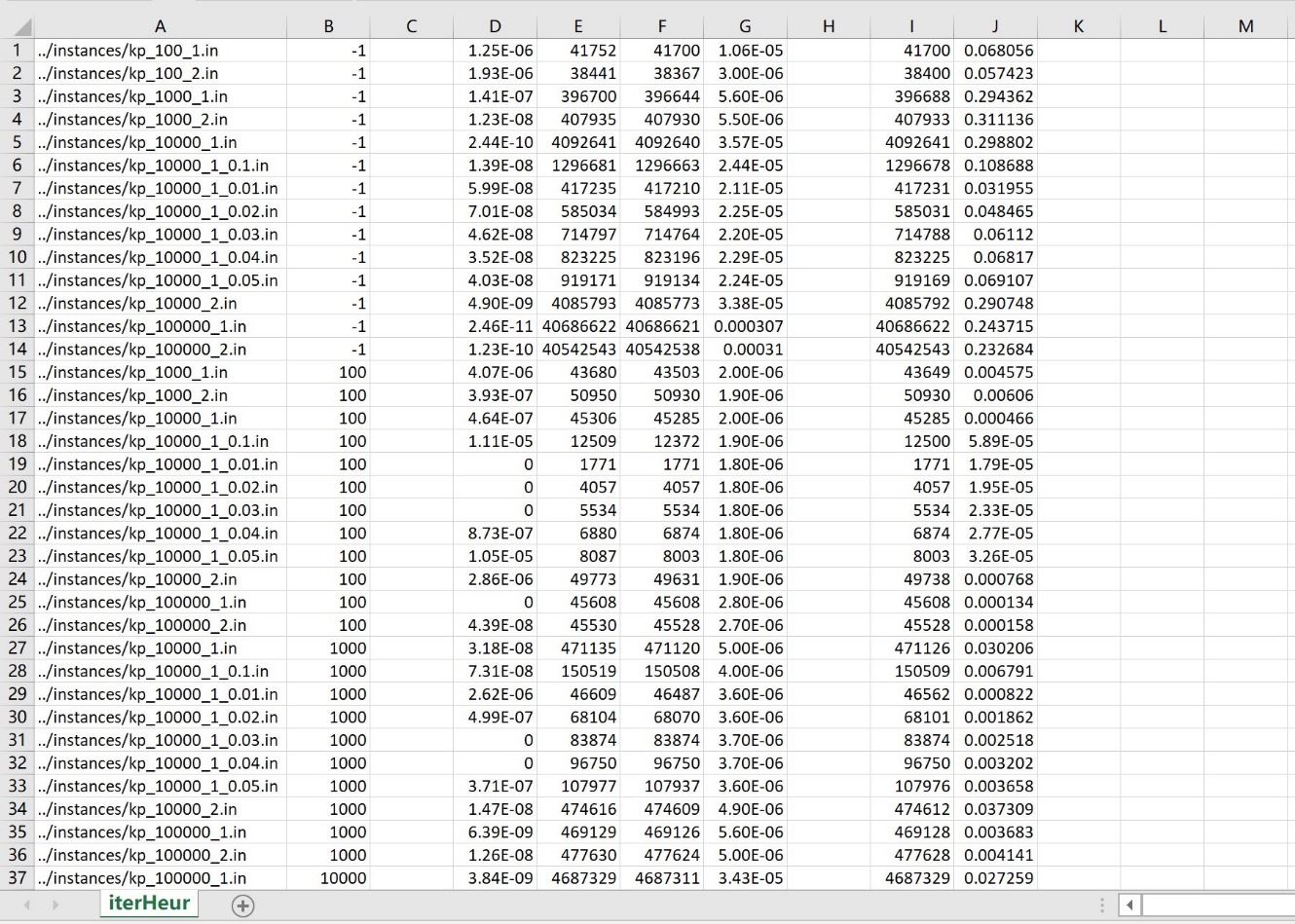
On a une valeur *nbSelectedReopt*, qui est un entier combien d’objet avant le premier objet non sélectionné on veut re optimiser. Alors on suppose de prendre tous les objets qui est dans l’intervalle [0, lastIndex - nbSelectedReopt].

Ensuit, pour re optimiser les objets dans l’intervalle[lastIndex-nbSelectedReopt,lastIndex+nbUnselectedReopt], on va d’abord générer ces jeux de données en cette intervalle par appeler la fonction de Question2. Plus on fait l’optimisation de façon dynamique dans cet intervalle par appeler la fonction dans KpSolverDP.

Question12

Si on utilise des heuristiques de programmation dynamique, on va trouver que on peut faire beaucoup plus d’instances. Exécuter le fichier qui contient 100000 instances ne pose pas le problème, et c’est vite, il utilisé presque 0.000402s pour résoudre. Et si on change le rayon, on peut comme même de résoudre cette fonction.

Conclusion :   
Dans le fichier de sortie iterHeur.csv par la tetestIterHeur, on a obtenue un tableau avec la colonne A est le nom de fichier d’instances, et la colonne B est le rayon, la colonne D est le gap , la colonne E est la borne supérieur par l’algorithme glouton continue , la colonne F est la borne inférieur par l’algorithme glouton discrète , la colonne G est le temps, et la colonne I est la valeur par l’algorithme de dynamique d’heuristique, et la colonne J est sa temps d’exécuter.



# 5 Algorithme de séparation et évaluation

Question13 :

D’abord, pour tous les objets, on regarde si cet objet est imposé dans le sac à dos par vérifier isFixed de cette case d’objet est vrai ou pas, et si est vrai, alors on va actualise la masse et la localUpperBound.

Puis si après ajouter ces objets, la masse reste du sac à dos est négative alors c’est le cas on a pris trop des objets, on va alors arrêter ici et annoncer il y a une erreur.

Si on peut continuer, alors on va tester par algo glouton des objets n’ont pas traité avant, ce sont les objets qui ni isFixed[i]= true ni isRemoved[i]= true. Et on va définir fractionalVariable la masse fractionnaire d’objet qui n’est peut pas prendre entièrement.

Question14 :

C’est la même méthode avec la question précédente, sauf c’est plus autorisé de prendre un objet partiellement. Et on va remplir le tableau primalSolution par booléen pour dire on prend cet objet ou pas.

Question15 :

J’ai ajouté les commentaires dans le fichier de code pour expliquer.

Question18 :

Utiliser le glouton continue et le random qui est le plus optimale façon.

Question19 :

On a essayé de changer le nombre de node, et on trouve que quand le nombre de node est plus grand, alors c’est plus optimale.