Rapport du Projet d'ORO

Bouton Nicolas

Avril 2021

Contents

1 But du projet

Le but du projet d'Optimisation et de Recherche Opérationnelle est d'implémenter en **C** un algorithme de séparation et d'évaluation (Branch and Bound) de notre choix.

1.1 Choix du problème à résoudre

J'ai personnellement choisis le problème du sac à dos pour sa simplicité et du fait que j'ai tout de suite pensé à faire un lecteur de format csv pour ce problème, que je décrirais plus tard.

2 Problème du sac à dos

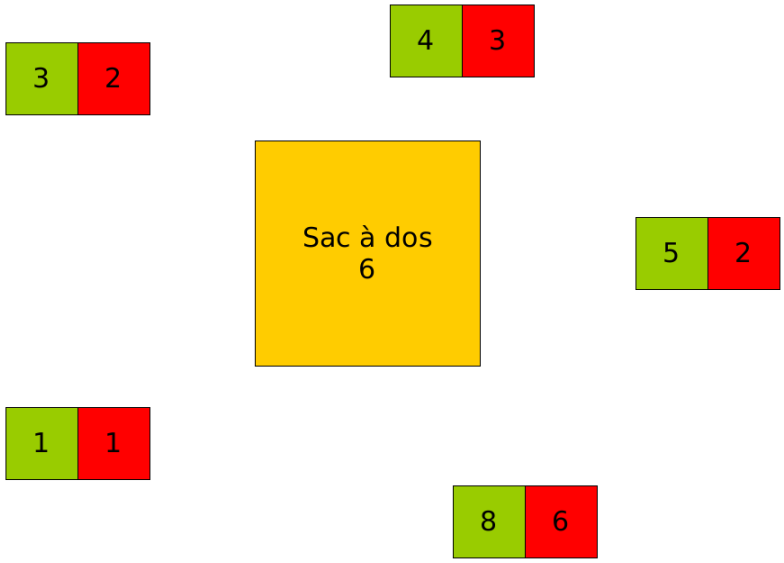

Figure 1: Problème du sac à dos

Le problème du sac à dos est de le remplir afin d'avoir une valeur maximum sans exéder son poids maximum (ici 6). Pour ce faire il y a des pairs d'objets avec des valeurs (ici en vertes) et des poids (ici en rouges).

2.1 Démonstration de l'algorithme implémenté

2.1.1 Tri de valeur par poid

Mon algorithme tri d'abord les paires par valeur par poid. C'est à dire qu'il tri en ordre décroissant les paires suivant leurs taux valeur/poids.

1. Exemple

   Imaginons que nous ayons l'entrée suivante:

```
"value", "weight"
3, 2
4, 3
1, 1
8, 6
5, 2
```

Dans ce cas nous aurions des taux:

```
"rate"
1.5
1.333333
1.0
1.333333
2.5
```

L'algorithme de tri mettra la pair (5, 2) en premier, et déplacera la pair (8, 6) d'un cran (car on ne change de place que si c'est strictement supérieur):

```
"value", "weight"
5, 2
3, 2
4, 3
8, 6
1, 1
```

2.1.2 Résolution par récursion

Le système de résolution par récursion est simple, à **gauche** de l'arbre on prend la pair, à **droite** on ne prend pas la pair. **Si on prend la pair**, alors il ne faut pas que le sac déborde sinon on met la valeur à -1 pour dire que le sous arbre n'est pas réalisable (c'est à dire que pour toute les noeuds suivant la solution reste iréalisable). Si le sac ne déborde pas on continue. Si on arrive à la fin alors on calcul la valeur du sac et on le stock dans un tableau suivant la position de la feuille dans l'arbre.

Sur les arrêtes on retrouve la valeur actuelle des éléments pris. A la fin on vois un tableau qui contient les valeurs de chaque chemin ce qui nous permettra ensuite de savoir quelle est la valeur maximal réalisable.

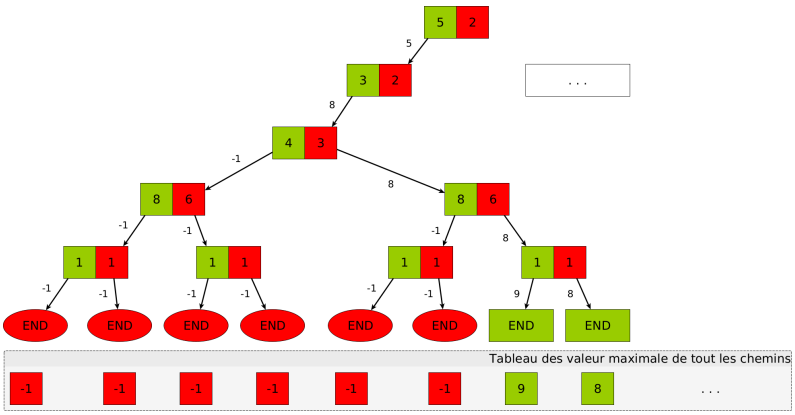

Figure 2: Démonstration de l'arbre

2.1.3 Trouve le chemin de la valeur maximale possible

Maintenant qu'on a le tableaux contenant la valeur maximale de tout les chemins. On peut trouver la valeur maximale réalisable qui est en fait la valeur maximale car toutes les valeurs non réalisables sont à -1. Ensuite pour savoir quel chemin à été parcourus on se base sur le fait que c'est un arbre binaire et donc on a la propriété suivante:

- gauche (on prends) égale 0 modulo 2
- droite (on ne prends pas) égale 1 modulo 2

Dans ce cas on pars des feuilles et on remonte l'arbre pour savoir quelle pair on a pris.

3 Choix d'implémentaion

3.1 Language

J'ai choisis le language **C** car c'est le language que j'ai le plus utilisé. Le **C++** permet de manipuler les listes plus facilement mais il faut faire un peu d'orienté objet que je ne maîtrise pas complètement. Et en réalité le language **C** est plus simple car on manipule dès objets très spécifiques.

3.2 Commentaire

Normalement le code est bien commenté (en anglais) donc vous devriez comprendre le code.

3.3 Lecteur csv

Comme je l'ai dis dans l'intro, la première chose qui m'est venu à l'esprit en voulant faire le problème du sac à dos était de faire un lecteur de fichier csv. Afin de pouvoir mettre des donnée sur **exel** par exemple et les exporter en format **csv**. Je n'ai pas eu trop de problème avec cette partis étant donné que j'ai trouvé une fonction qui me permet de séparé une ligne par un mot avec **strtok**. Ensuite il suffisait juste de capturer le bon élément.

3.4 Problème du sac à dos

Ma méthode était juste de faire un algorithme récursif naïf qui parcourais toutes les branches et ne s'arrêter que lorsque toute les branches étaient parcourus. Je voulais utiliser un tableau **taken** pour savoir si la pair était prise mais j'ai vite compris que le résultat était faux en regardant la sortie. Car à chaque nouveau parcour on éditait le tableau.

Donc pour savoir quel élément était pris, j'ai ajouté un tableau à la base de taille **le nombre d'entrée**. Mais j'ai très vite remarqué que mon code ne fonctionner plus correctement car j'aivais des problèmes de corruptions de donnée (**free**). Le problème venait du fait qu'évidemenent il fallait mettre comme taille de talbeau **le nombre de feuille** de l'arbre qui est 2 puissance **le nombre de feuille** (ici 2 car on a un arbre binaire) car sinon on dépassait dans la mémoire du tableau et donc on écrivait dans une autre case mémoire (ce qui causait la corruption de donnée).

Ensuite lorsque l'on a ce tableau qui nous indique la valeur de chaque chemin, comme dis en haut il suffit de trouver la valeur maximale car toutes les valeurs positives sont réalisables. Et avec la technique que j'ai décris en haut on peut retrouver le chemin.

Sinon il n'y a pas grand chose à dire. L'algorithme décris en haut donne les grandes étapes de mon implémentations. Les commentaires peuvent le compléter aussi.

3.5 Affichage

Rien de particulier à dire. J'utilise la sortie d'erreur pour la sortie car elle n'est pas bufferizé. J'affiche d'abord les entrées et ensuite j'aiffiche la sortie qui est composé de 3 champs:

- la valeur maximale réalisable
- la liste des (valeur, poid) pris

- le chemin dans l'arbre de la valeur maximale réalisable

Pour afficher l'arbre, j'ai juste jouer sur le fait de décaler de **n espace** à gauche ou à droite suivant si la pair était pris ou non.

4 Démonstation du code

Pour l'entrée suivante:

```
"value", "weight"
1.0, 0.1
4.0, 1.1
5.0, 1.9
2.0, 2.2
3.0, 4.0
```

Et la ligne de commande suivante:

```
$ ./ks --input test.csv 5.2
```

Nous obtenons:

```
Knapsack problem input:
 - max weight: 5.200000
 - file input:
     "value", "weight"
     1.000000, 0.100000
     4.000000, 1.100000
     5.000000, 1.900000
     2.000000, 2.200000
     3.000000, 4.000000

Result:
 - max value: 11.000000
 - list of pair taken:
     "value", "weight"
     4.000000, 1.100000
     5.000000, 1.900000
     2.000000, 2.200000
 - tree:
                                1.0, 0.1
```

```
                    \
                 4.0, 1.1
                  /
               5.0, 1.9
                /
            2.0, 2.2
             /
         3.0, 4.0
              \
               END
Time in second(s): 0.000001
```

5 Conclusion

Pour conclure, le travail fait pour résoudre ce problème peut être améliorer étant donné que l'algorithme pour le résoudre parcours tout les chemins, contrairement à l'algorithme que l'on a vu en cours qui parcourrait changeait le chenmin à parsourir à chaque étape.

Liste des fonctionnalités implémentés:

- lecteur de fichier **csv**
- résolution du problème du sac à dos
- affichage du chemin de l'arbre donnant la valeur maximale réalisable