Hugo PUC
Charly BEC

M1 Informatique
07/03/2022

Rendu TP1 Bin Packing

Heuristiques

Le problème du bin packing consiste à ranger des objets dans des boîtes en utilisant le moins de boîtes possibles. Chaque objet a une taille et chaque boîte a une capacité limité.

Nous avons tout d'abord rangé notre tableau d'objet dans l'ordre décroissant puis appliqué les différents algorithmes.

```c
// nombre objets
int n = dsptr->n;

// taille boîtes
int V = dsptr->V;

// tableau de poids des objets
int *array = malloc(n * sizeof(int));

// structure de boîtes
typedef struct bin {
    int *array;

    // addition des éléments de la boîte
    int size;

    // nombre d'éléments de notre boîte
    int nbelt;
} bin;

bin* bins = malloc(n * sizeof(bins));
```

On récupère les données du tableau, on va ensuite utiliser une structure bin qui simulera nos boîtes, pour chaque boîte on aura donc un tableau d'objets, la taille qui correspond à l'addition des objets et le nombre d'objets dans notre boîte.

Puis on part du principe qu'au pire des cas on aura une boîte par objet donc on crée le tableau "bins" de taille n objets.

```
// fonction pour qsort (a-b: croissant, b-a: décroissant)
int compare (const void * a, const void * b)
{
    return ( *(int*)b - *(int*)a );
}

// copie des valeurs dans notre tableau
for (int i=0; i<n; i++) {
    array[i] =  dsptr->size[i];
}

// tri le tableau dans l'ordre décroissant
qsort (array, n, sizeof(int), compare);
```

Pour trier le tableau des objets, on crée la fonction compare pour pouvoir utiliser le “qsort” et avoir nos objets triés.

```
// stocke le nombre de boîtes nécessaires
int nbBin = 0;
```

Cette variable va stocker le nombre de boîtes que l’on va utiliser à chaque heuristique pour l’affichage.

First Fit (decreasing)

Dans cette heuristique, on va parcourir les objets, puis on va placer chaque objet dans la boîte pouvant l’accepter.
Quand une boîte ne peut pas accepter un objet, on va utiliser la suivante, donc on va revenir à la première boîte pour chaque objet et tester si l’objet peut entrer en itérant sur les boîtes déjà créées.

```
// parcours les valeurs une par une
for (int i = 0; i < n; i++) {

    // parcours les struct de boîtes une par une
    for (int j=0; j<n; j++) {

        // si on peut ajouter une valeur dans une boîte
        if (array[i] + bins[j].size <= V) {

            // on ajoute la valeur dans le tableau de la boîte
            bins[j].array[bins[j].nbelt] = array[i];

            // on ajoute la valeur au total de la boîte
            bins[j].size += array[i];

            // on incrémente le nombre d'éléments pour
            // pouvoir ajouter d'autres valeurs en itérant
            // à partir de cette valeur
            bins[j].nbelt ++;

            // mets à jour le nombre de boîtes
            if (j >= nbBin) {
                nbBin = j+1;
            }

            // sort de la boucle pour itérer à la prochaine
            // valeur et reparcourir les struct de boîtes
            break;
        }
    }
}
```

Pour l’affichage du résultat, on utilise une boucle simple.

```
// Affichage du résultats

// s'arrête au nombre de boîte que l'on a
for (int i=0; i<nbBin; i++) {
    printf("boîte n° %d (taille: %d): ", i+1, bins[i].size);

    // s'arrête au nombre d'élément pour chaque
    // tableau de chaque structure de boîte
    for (int j=0; j<bins[i].nbelt; j++) {
        printf("%d ", bins[i].array[j]);
    }
    printf("\n");
}
```

Next Fit (decreasing)

Dans celle-ci, nous allons bêtement tester si un objet rentre dans la première boîte, sinon on utilise la boîte suivante etc. On va donc se placer sur la dernière boîte remplie au lieu de se placer à la première boîte comme sur l'algo précédent.

```
// cette fois on part de la boîte précédente
for (int j=nbBin; j<n; j++) {
```

On change juste la deuxième boucle en la faisant commencer par le nombre de boîte que l'on a utilisé pour le moment.

```
// mets à jour le nombre de boîtes
if (j >= nbBin) {
    nbBin = j;
}
```

On va aussi cette fois donner la valeur j et non j+1 à nbBin car sinon on aurait un souci dans la boucle.

```
// on rajoute la valeur manquante car on commence à 0
nbBin += 1;
```

A la fin de l'exécution de l'algo, on ajoute 1 à nbBin pour avoir un bon nombre de boîtes.

L'affichage du résultat est le même que l'algo first fit.

Best Fit (decreasing)

Cette heuristique ressemble au First fit à la différence que l'on va regarder la meilleure boîte pour stocker chaque objet au lieu de prendre la première boîte pouvant l'accueillir.

```
// parcours les valeurs une par une
for (int i = 0; i < n; i++) {

    // meilleur boîte
    int bestBin = 0;

    // meilleur différence valeur/tailleBoite
    int bestFit = 0;

    // parcours les struct de boîtes une par une
    for (int j=0; j<n; j++) {

        // si on peut ajouter une valeur dans une boîte
        if (array[i] + bins3[j].size <= V) {

            // si on trouve un bestFit
            if (bins3[j].size + array[i] >= bestFit) {
                bestBin = j;
                bestFit = bins3[j].size + array[i];
            }
        }
    }

    // on ajoute la valeur dans le tableau de la boîte
    bins3[bestBin].array[bins3[bestBin].nbelt] = array[i];

    // on ajoute la valeur au total de la boîte
    bins3[bestBin].size += array[i];

    // on incrémente le nombre d'éléments pour
    // pouvoir ajouter d'autres valeurs en itérant
    // à partir de cette valeur
    bins3[bestBin].nbelt ++;

    // mets à jour le nombre de boîtes
    if (bestBin >= nbBin) {
        nbBin = bestBin+1;
    }
}
```

Pour afficher le résultat, on va devoir enlever les boîtes nulles.

```
// Affichage du résultats

// Pour enlever les boîtes vides.
int cpt = 0;

// s'arrête au nombre de boîte que l'on a
for (int i=0; i<nbBin; i++) {

    if (bins3[i].size != 0) {

        printf("boîte n° %d (taille: %d): ", i+1-cpt, bins3[i].size);

        // s'arrête au nombre d'élément pour chaque
        // tableau de chaque structure de boîte
        for (int j=0; j<bins3[i].nbelt; j++) {
            printf("%d ", bins3[i].array[j]);
        }
        printf("\n");

    } else {
        cpt++;
    }
}
```

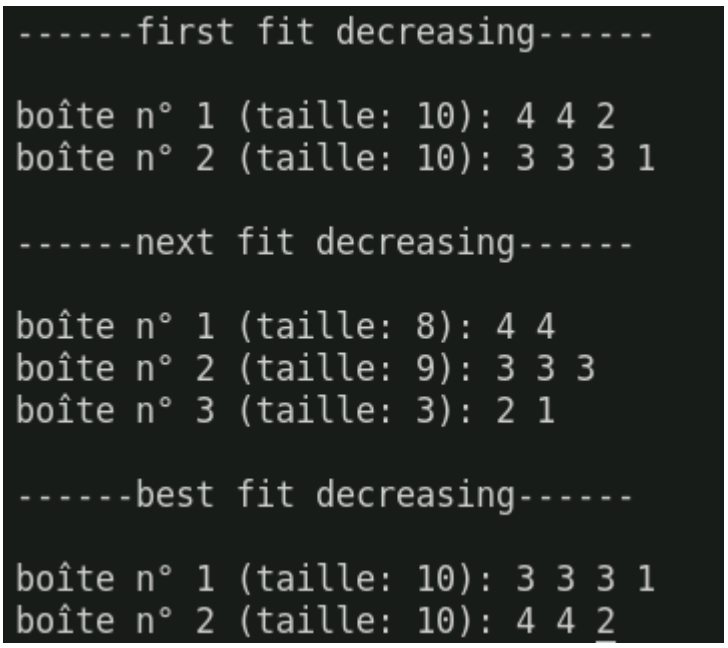
Résultats

On va pouvoir comparer les résultats sur le premier jeu de données.

```
------heuristic------

nombre objets: 7
tailles boîtes: 10
poids objets: 4 4 3 3 3 2 1
```

On va donc utiliser nos algos sur ce tableau d'objets triés au préalable.



Le first fit decreasing va donc utilisé la taille minimale de boîte possibles pour ce problème tout comme le best fit decreasing.

Le next fit decreasing va quant à lui utiliser une boîte de plus car il va aller directement vers une solution rapide et moins travaillée.

Regardons pour le deuxième jeu de données.

```
boîte n° 1 (taille: 149): 100 49
boîte n° 2 (taille: 149): 100 49
boîte n° 3 (taille: 149): 100 49
boîte n° 4 (taille: 150): 98 52
boîte n° 5 (taille: 150): 98 52
boîte n° 6 (taille: 150): 98 52
boîte n° 7 (taille: 150): 97 53
boîte n° 8 (taille: 148): 97 51
boîte n° 9 (taille: 148): 97 51
boîte n° 10 (taille: 150): 96 54
boîte n° 11 (taille: 149): 95 54
boîte n° 12 (taille: 143): 95 48
boîte n° 13 (taille: 141): 94 47
boîte n° 14 (taille: 140): 94 46
boîte n° 15 (taille: 150): 92 58
boîte n° 16 (taille: 150): 92 58
boîte n° 17 (taille: 150): 91 59
boîte n° 18 (taille: 149): 91 58
boîte n° 19 (taille: 149): 91 58
boîte n° 20 (taille: 148): 91 57
boîte n° 21 (taille: 150): 89 61
boîte n° 22 (taille: 149): 89 60
boîte n° 23 (taille: 149): 89 60
boîte n° 24 (taille: 150): 88 62
boîte n° 25 (taille: 145): 88 57
boîte n° 26 (taille: 150): 87 63
boîte n° 27 (taille: 150): 86 64
boîte n° 28 (taille: 150): 85 65
boîte n° 29 (taille: 150): 85 65
boîte n° 30 (taille: 149): 85 64
boîte n° 31 (taille: 141): 84 57
boîte n° 32 (taille: 149): 82 67
boîte n° 33 (taille: 149): 82 67
boîte n° 34 (taille: 150): 81 69
boîte n° 35 (taille: 150): 81 69
boîte n° 36 (taille: 150): 80 70
boîte n° 37 (taille: 150): 79 71
boîte n° 38 (taille: 149): 79 70
boîte n° 39 (taille: 150): 77 73
boîte n° 40 (taille: 150): 76 74
boîte n° 41 (taille: 150): 75 75
boîte n° 42 (taille: 141): 72 69
boîte n° 43 (taille: 150): 67 57 26
boîte n° 44 (taille: 135): 45 45 45
boîte n° 45 (taille: 150): 44 43 42 21
boîte n° 46 (taille: 149): 40 40 39 30
boîte n° 47 (taille: 150): 39 38 37 36
boîte n° 48 (taille: 140): 37 35 34 34
boîte n° 49 (taille: 150): 33 33 32 29 23
```

```
boîte n° 50 (taille: 136): 29 29 27 26 25
boîte n° 51 (taille: 85): 23 22 20 20
```

Voici le résultat de first fit decreasing, next fit decreasing va utiliser 71 boîtes, best fit decreasing va utiliser 51 boîtes.

Maintenant on va tester en enlevant le tri décroissant du tableau sur une instance particulière.

```
------heuristic------

4 4 3 3 3 1 2
```

Sur cette instance, best fit devrait trouver un meilleur résultat que first fit car le tableau n'est pas bien trié.

First fit:

```
boîte n° 1 (taille: 9): 4 4 1
boîte n° 2 (taille: 9): 3 3 3
boîte n° 3 (taille: 2): 2
```

Best fit:

```
boîte n° 1 (taille: 10): 3 3 3 1
boîte n° 2 (taille: 10): 4 4 2
```

On va relancer notre programme sur la deuxième instance sans trier le tableau et voir les différences des algos.

First fit:

```
boîte n° 53 (taille: 125): 85 40
boîte n° 54 (taille: 57): 57
```

On trouve 54 boîtes (moins bien que le first fit decreasing).

Next fit:

```
boîte n° 67 (taille: 97): 40 57
```

On trouve 67 boîtes (mieux que le next fit decreasing).

Best fit:

```
boîte n° 53 (taille: 147): 100 22 25
```

On trouve 53 boîtes (moins bien que le best fit decreasing mais mieux que le first fit).

Donc on peut remarquer que sur la deuxième instance non triée, le best fit s’en sort mieux que le first fit.

CPLEX

```
Elapsed time = 484.31 sec. (341514.75 ticks, tree = 532.91 MB, solutions = 22)
 449189 39989        cutoff             51.0000       50.0000 1.02e+08    1.96%
 455131 40386       50.0000    37       51.0000       50.0000 1.04e+08    1.96%
 460658 40744    infeasible             51.0000       50.0000 1.05e+08    1.96%
 465779 41020        cutoff             51.0000       50.0000 1.06e+08    1.96%
 470758 41510       50.0000    47       51.0000       50.0000 1.09e+08    1.96%
 475673 41853       50.0000    34       51.0000       50.0000 1.10e+08    1.96%
```

Le cplex nous trouve une solution à 51 boîtes avec une tolérance de 1.96%. On remarque que quand il y a beaucoup de valeurs, le cplex prend beaucoup de temps pour résoudre le problème. Pour l'exemple difficile le cplex a mis 484 secondes pour une solution à 1.96% de tolérances. Avec une tolérance aussi faible il est cependant très probable que cette solution soit la solution exacte.
De plus, on peut voir que la plupart des objectifs sont infeasibles ou cutoff.

Le fait que le cplex peine à trouver la solution exacte peut être généralisé à tous les problèmes de bin packing comprenant beaucoup de valeurs.