Compte-rendu mini-projet d'algorithmique

Pierre Dénès 24/05/2020

Introduction

On s'intéresse dans ce compte rendu aux manières d'obtenir une valeur N à partir de la somme de valeurs issues d'un ensemble C. Plus particulièrement on s'intéresse à la manière d'obtenir cette valeur en sommant un nombre minimal d'éléments. Un exemple pratique d'application de ce problème est le rendu de monnaie, ou la manière optimale de finir une partie de fléchettes.

On définit dans la suite les valeurs disponibles par $C = \{c_i\}_{i\in[1,n]}$. Une solution est alors $S = \{s_i\}_{i\in[1,n]}$ tel que $N = \sum_{i=1}^{n} c_i \times s_i$. Une solution optimale est alors S tel que la somme $\sum_{i=1}^{n} s_i$ est minimale.

Les algorithmes étudiés pour obtenir une solution seront les essais successifs, la programmation dynamique, et un algorithme glouton.

1 Essais successifs

On définit un candidat comme un tableau de taille n, contenant le nombre de chaque élément de C. Les sommes calculées dans les algorithmes suivants sont calculées au fur et à mesure et passées en paramètre, elle n'apparaissent pas pour alléger la liste des entrées.

Algorithme 1 : Essais successifs

```
1  Entrée:
2  C[n]   // Valeurs disponibles
3  N      // Valeur à atteindre (en plus de la somme courante)
4  S[n]   // Somme courante
5  M[n]   // Meilleure solution trouvée (peut être vide)
6  Sortie:
7  M[n]   // Meilleure solution trouvée
8  Début essais_successifs(C, N, S, M)
9      si satisfaisant(N, S, M)
10         M ← S
11     sinon
12         pour chaque C[i]
13             S[i] ← S[i] + 1
14             si non_elagage(C, N - C[i], S, M)
15                 essais_successifs(C, N - C[i], S, M)
16             fsi
17             S[i] ← S[i] - 1
18         fpour
19     fsi
20 Fin
```

Algorithme 2 : Satisfaisant

```
Entrée:
N      // Valeur à atteindre (en plus de la somme courante)
S[n]   // Somme courante
M[n]   // Meilleure solution trouvée (peut être vide)
Sortie:
Booléen
Début satisfaisant(N, S, M)
    // Dans le cas où M n'existe pas, la deuxième condition pour
    // Vrai n'est pas testée.
    Retourne
        Vrai si N = 0 et la somme des S[i] < la somme des M[i]
        Faux sinon
Fin
```

Algorithme 3 : Non elagage

```
Entrée:
C[n]   // Valeurs disponibles
N      // Valeur à atteindre (en plus de la somme courante)
S[n]   // Somme courante
M[n]   // Meilleure solution trouvée (peut être vide)
Sortie:
Booléen
Début non_elagage(C, N, S, M)
    // Dans le cas où M n'existe pas, les conditions associées ne
    // sont pas testées.
    // Les min, max, et sommes sont en réalité déjà en mémoire
    Retourne
        Vrai si (N >= min(C[i]
                et somme(S[i]) < somme(M[i])
                et (somme(M[i]) - somme(S[i])) * max(C[i]) > N
                )
                ou N = 0
        Faux sinon
Fin
```

1.1 Conditions d'élagage

On s'appuie sur plusieurs critères pour stopper les appels récursifs sans aboutir à une solution.

— Le premier évident est de ne pas faire d'appel récursif si il y a déjà autant d'éléments dans le candidat courant que dans la meilleure solution.

— Ensuite si la valeur à ajouter pour atteindre N est inférieure au plus petit c_i disponible, N ne peut plus être atteinte.

— Enfin si en ajoutant $\sum M[i] - \sum S[i]$ fois le plus grand c_i la valeur atteinte est inférieure à N, il n'est plus possible d'atteindre une meilleure solution.

Pour atteindre une première solution la plus courte possible et commencer à élaguer en utilisant la première condition, il faut lancer les appels récursifs par valeur décroissant de c_i (ie ordonner le tableau C[n] en ordre décroissant avant le premier appel). L'efficacité de cet ordonnancement des c_i est vérifiée par le test *pruning_sorting* qui utilise $C = \{1, 2, 5, 10, 20, 50\}$ et vérifie que

$$\forall N \in [55, 99],\ temps\_execution\_decroissant \leq temps\_execution\_croissant$$

L'efficacité des autres conditions d'élagage est vérifiée par le test *pruning_target* qui prend le même C et vérifie que

$$\forall N \in [300, 333],\ temps\_execution\_avec\_elagage \leq temps\_execution\_sans\_elagage$$

Les valeurs choisies sont volontairement grandes pour exacerber la différence faite par l'élagage, qui pour des petites valeurs peut être compensée par la durée des calculs dans le test d'élagage.

1.2 Complexité de l'algorithme

On utilise le nombre d'appels récursifs comme métrique. On calcule la complexité dans le pire des cas. On suppose C ordonné de manière décroissante.

Soit $n = Card(C)$. Soit N la valeur cible. Soit l la longueur de la première solution trouvée.

Dans le pire des cas la première solution trouvée est la meilleure et tous les appels suivants atteignent la même profondeur l (sans élagage). La complexité est alors n^l. Comme les c_i sont ordonnés, pour N grand devant $max(c_i)$, $l = \mathcal{O}(N \div max(c_i)) = \mathcal{O}(N)$.

La complexité temporelle pour $N \to \infty$ est donc

$$\mathcal{O}(n^N)$$

La complexité spaciale est $n \times sizeof(entier)$ où entier est le type utilisé pour stocker le nombre d'éléments pour chaque c_i.

1.3 Code Rust associé

Structures	Méthodes	Fonctions	Tests
Coins	is_better	successive_tries_init	succ_tries
	is_solution	successive_tries_single_thread	pruning_sorting
	is_viable	successive_tries_no_pruning	pruning_target
		successive_tries	

1.4 Exemple (interactive CLI)

```
enssat-algo on  master is — v0.1.0 via — v1.45.0-nightly
❯ cargo run --release
   Compiling algo v0.1.0 (/home/doune/Documents/ENSSAT/enssat-algo)
    Finished release [optimized] target(s) in 1.22s
     Running `target/release/algo`
+-------+-----+-----+-----+----+----+---+---+---+
| Coins | 500 | 200 | 100 | 50 | 20 | 10 | 5 | 2 | 1 |
+-------+-----+-----+-----+----+----+---+---+---+
q to quit, l to reload coin values, s to set target
s
enter number to set new target
999
s for successive tries, d for dynamic programming, g for greedy algorithm
s
Selected successive tries...
Time elapsed: 878ms
+--------+-----+-----+-----+----+----+---+---+---+---+-----+
| Coins  | 500 | 200 | 100 | 50 | 20 | 10 | 5 | 2 | 1 | 888 |
+--------+-----+-----+-----+----+----+---+---+---+---+-----+
| Change | 1   | 2   | 0   | 1  | 2  | 0  | 1 | 2 | 0 | 9   |
+--------+-----+-----+-----+----+----+---+---+---+---+-----+
| Total  | 500 | 400 | 0   | 50 | 40 | 0  | 5 | 4 | 0 | 999 |
+--------+-----+-----+-----+----+----+---+---+---+---+-----+
q to quit, l to reload coin values, s to set target
```

2 Programmation dynamique

2.1 Relation de récurrence

On définit les équations suivantes pour exprimer $NBP(i,j)$:

$\forall j \in \mathbb{N},\ NBP(0,j) = \infty$

$\forall j \in \mathbb{N},\ \forall i \in [1,n],\ c_i = j \implies NBP(i,j) = 1$

$\forall j \in \mathbb{N},\ \forall i \in [1,n],\ c_i > j \implies NBP(i,j) = NBP(i-1,j)$

$\forall j \in \mathbb{N},\ \forall i \in [1,n],\ c_i < j \implies NBP(i,j) = min(NBP(i-1,j),\ 1 + NBP(i,j-c_i))$

Les valeurs pour $i = 0$ ou $j = 0$ sont définies ainsi pour les calculs du minimum. Les deux dernières équations donnent la structure tabulaire pour les sous-problèmes ainsi que le sens de parcours. L'algorithme utilisera une matrice $n+1$ par $N+1$, parcourue par i croissants et j croissants. La dernière ligne donnera les solutions pour atteindre les valeurs de 1 à N. La complexité temporelle est donc $\mathcal{O}(nN)$. La complexité spatiale est nN fois celle de l'algorithme des essais successifs.

2.2 Algorithme

Algorithme 4 : Programmation dynamique

```
Entrée:
C[n]   // Valeurs disponibles
N      // Valeur à atteindre
Sortie:
M[n]   // Meilleure solution trouvée (peut être vide)
Début programmation_dynamique(C, N)
    // Initialisation de la matrice
    T ← {0} // Chaque case contient un tableau de n 0
    T[0] ← {∞} // Première équation
    pour i de 1 à n − 1
        pour j de 1 à N
            si C[i - 1] = j
                T[i][j][i] ← 1          // Deuxième équation
            sinon si C[i - 1] > j
                T[i][j] ← T[i - 1][j]   // Troisième équation
            sinon
                // Quatrième équation
                T[i][j] ← min(T[i - 1][j], 1 + T[i][j - C[i]])
                // où min retourne le tableau de somme minimale
            fsi
        fpour
    fpour
    Retourne T[n][N]
Fin
```

En pratique les sommes de valeurs de tableaux sont calculées et stockées au fur et à mesure pour être comparées pour le minimum.

2.3 Code Rust associé

Structures	Méthodes	Fonctions	Tests
Coins		dynamic_programming	dyn_prog
Matrix			
Solution			

2.4 Exemple (interactive CLI)

```
enssat-algo on  master [!] is — v0.1.0 via — v1.45.0-nightly
❯ cargo run --release
   Compiling algo v0.1.0 (/home/doune/Documents/ENSSAT/enssat-algo)
    Finished release [optimized] target(s) in 0.85s
     Running `target/release/algo`
+-------+-----+-----+-----+----+----+---+---+---+
| Coins | 500 | 200 | 100 | 50 | 20 | 10 | 5 | 2 | 1 |
+-------+-----+-----+-----+----+----+---+---+---+
q to quit, l to reload coin values, s to set target
s
enter number to set new target
999
s for successive tries, d for dynamic programming, g for greedy algorithm
d
Selected dynamic programming...
Time elaped: 808us
+--------+-----+-----+-----+----+----+----+---+---+---+-----+
| Coins  | 500 | 200 | 100 | 50 | 20 | 10 | 5 | 2 | 1 | 888 |
+--------+-----+-----+-----+----+----+----+---+---+---+-----+
| Change | 1   | 2   | 0   | 1  | 2  | 0  | 1 | 2 | 0 | 9   |
+--------+-----+-----+-----+----+----+----+---+---+---+-----+
| Total  | 500 | 400 | 0   | 50 | 40 | 0  | 5 | 4 | 0 | 999 |
+--------+-----+-----+-----+----+----+----+---+---+---+-----+
q to quit, l to reload coin values, s to set target
```

3 Algorithme glouton

Soit $C = \{6, 4, 1\}$ et $N = 8$. L'algorithme glouton donne alors pour solution $S = \{1, 0, 2\}$ alors que la solution optimale est $S = \{0, 2, 0\}$. En expérimentant avec l'application Rust, on se rend compte que la solution trouvée est bien optimale dans le cas du système de l'euro.

3.1 Algorithme glouton optimal pour l'euro ?

Le système de l'euro est construit sur un modèle 1-2-5 ($C = \{c_1 = 1, c_2 = 2, c_3 = 5, \forall k \in [\![4, n]\!],\ c_k = 10 \times c_{k-3}\}$).

Soient les m_k le maximum de pièce k dans une solution optimale $O = \{o_k\}$ (ie $m_k \geq o_k$). Comme les valeurs se "répètent" à une puissance de 10 différente, on s'intéresse seulement aux valeurs c_1 à c_4 (1, 2, 5, 10). On a alors $m_1 = 1$ (deux pièces de 1 se remplaceraient par une pièce de 2 dans une solution optimale). De même on a $m_2 = 2$ car $5 + 1 = 6$, et $m_3 = 1$. On a de plus la contrainte si $o_2 = 2$ alors $o_1 = 0$ car $2 + 2 + 1 = 5$.

Soit $N \in [\![11, 19]\!] = [\![c_4 + 1, c_5 - 1]\!]$. Supposons que la solution optimale ne contient pas $c_4 = 10$. La somme maximale pouvant être atteinte par une solution optimale est alors obtenue par $o_3 \times c_3 + o_2 \times c_2 + o_1 \times c_1 \leq 9$. Donc la solution optimale contient c_4 (la pièce donnée par l'algorithme glouton).

De même $o_2 \times c_2 + o_1 \times c_1 < c_3$ et $o_1 \times c_1 < c_2$, l'algorithme glouton est bien optimal pour N.

On peut répéter le même raisonnement pour les valeurs c_k supérieures. Pour les valeurs supérieures à $2 \times c_n$, il est évident qu'on peut se rapporter à une valeur inférieure.

3.2 Cas $C' = \{1, 3, 5, 10, 30, 50\}$

On peut calculer de la même manière la valeur maximale atteinte par une solution ne contenant la pièce de valeur maximale. On a $60 = 50 + 10 = 30 + 30$, $90 = 50 + 30 + 10 = 30 + 30 + 30$, et $120 = 50 + 50 + 10 + 10 = 30 \times 4$ et $150 = 3 \times 50 = 5 \times 30$. Même s'il existe des solutions optimales ne contenant pas 50 pour des dizaines supérieures à 50, il existe toujours une solution optimale contenant 50 équivalente. On restraint donc ici la définition des m_k au nombre maximum de pièces c_k qui ne produit pas une valeur ayant une autre solution optimale contenant c_{k+1}. On a donc ici $M = \{m_k\} = \{2, 1, 1, 2, 1\}$, la même contrainte qu'on ne peut pas avoir $o_1 = 2$ et $o_2 = 1$ (de même pour o_4 et o_5).

La valeur maximale pouvant être atteinte sans 50 est donc $30 + 10 + 5 + 3 + 1 = 49$. Sans 30 on a $10 + 10 + 5 + 3 + 1 = 29$, et ainsi de suite. L'algorithme glouton fonctionne bien pour C' (mais il existe d'autres solutions optimales).

3.3 Complexité

On considère deux cas : $N = 1$ et $N \ggg max(c_k)$. Dans le premier cas la complexité temporelle est $\mathcal{O}(n)$ (itération sur les valeurs). Dans le deuxième cas, la complexité est $\mathcal{O}(N)$.

La complexité temporelle est donc $\mathcal{O}(max(n, N))$ si les valeurs sont déjà triées, $\mathcal{O}(max(n \log(n), N))$ sinon. Pour N suffisamment grand (ou C déjà trié) c'est donc un algorithme de classe linéaire.

La complexité spatiale dans tous les cas est égale à celle des essais successifs ($n \times sizeof(entier)$).

4 Questions complémentaires

L'effort de conception de l'algorithme A est relativement simple, la partie la plus compliquée étant les conditions d'élagage. Celui de la partie B est un peu plus compliqué à mettre en place, même si ici le travail est grandement simplifié par le fait qu'on nous donne le fait que récurrence se fait sur un sous-ensemble de C.

Il serait possible de mettre en place une solution de type diviser pour reigner en fixant le nombre d'éléments c_k et en résolvant les sous-problèmes pour $C' = C \setminus \{c_k\}$ et $N' = N - x_k \times c_k$ où $x_k \in [\![0, \lfloor N/c_k \rfloor]\!]$. Cependant cette solution rencontrerait le même problème que celle des essais successifs : calculer plusieurs fois les même valeurs.

4.1 Conclusion

La solution au problème posé dépend des contraintes que l'on a pour le résoudre : si l'on cherche à obtenir une réponse juste dans tous les cas, la programmation dynamique est la meilleure solution, mais elle nécessite un espace mémoire beaucoup plus important que les autres solutions. Si l'on cherche à obtenir une réponse satisfaisante ou que l'algorithme glouton est juste pour le C que l'on s'est donné, il est alors l'algorithme à préférer. Si on veut toujours trouver la solution exacte mais que l'on est limités par la mémoire, il y a de bonne chance que la complexité temporelle des essais successifs soit également blocante, et il n'y a alors pas de solution.

L'étude réalisée dans ce projet met bien en avant l'importance de la conception d'un algorithme lors de la résolution d'un problème, notamment sa complexité : résoudre le problème n'est pas suffisant, il faut bien le résoudre. Cela est particulièrement vrai dans le cadre d'une formation où l'on se retrouve régulièrement avec une puissance de calcul très limitée (composants embarqués). Ce projet met également en avant la "puissance" d'une bonne heuristique comme celle de l'algorithme glouton utilisé (et également l'importance de la bonne conception d'un système monéraire dans ce cas d'application pratique).