Sorbonne Université

Rapport de Projet

Algorithmique, Confitures

César Mathéus
Anatole Vercelloni

Février 2021 — Juillet 2021

Table des matières

1 Introduction

Nous allons nous pencher dans ce rapport sur un problème d'optimisation combinatoire, le remplissage d'un système de pots de confiture. Pour cela, nous allons nous pencher sur 3 algorithmes avec des approches différentes : un algorithme récursif, un algorithme itératif et un algorithme glouton. Nous analysons les différentes complexité ainsi que l'optimalité de ces algorithmes. Nous supposons le lecteur déjà familiarisé avec le contexte ainsi que les différents notions du sujet. Ainsi, en général nous de reviendrons pas sur la signification des différents symboles dès lors que ceux-ci sont définis dans le sujet du projet.

Nous n'avons pas indiqué les numéros des questions pour une meilleure présentation du rapport cependant l'ordre y est respecté .

2 Partie théorique

Avant tout travail théorique, on remarque que la valeur $m(S)$ peut s'écrire de la manière suivante en fonction des $m(s,i)$:

$$m(S) = min_{1 \leq i \leq k}(m(s,i))$$

2.1 Montrons la relation de récurrence suivante :

$$m(s,i) = \begin{cases} 0 \text{ si } s = 0 \\ min\{m(s,i-1), m(s-V[i],i)+1\} \text{ sinon} \end{cases} \quad (1)$$

cas de base : $i = 1$

Si $i = 1$, alors $m(s,i-1) = m(s,0) = \infty$.
Par ailleurs, comme $V[1] = 1$ on a bien que :

$$m(s,1) = m(s-1,1) + 1 \leq \infty$$

En effet si on a uniquement des pots de capacité 1, $m(S) = S$, on aura besoin de S pots.Le cas de base vérifie donc bien la propriété précédente.

Hypothèse de récurrence et héréditée : Supposons que la propriété est vérifiée pour un i quelconque, montrons que $m(s,i+1)$ vérifie aussi l'égalité précédente.

Passer de i à i+1 revient à se donner une nouvelle catégorie de pots de capacité V[i+1]. Deux cas sont alors possibles. Soit l'assortiment de pots permettant d'atteindre $m(s,i+1)$, soit les pots de capacité V[i+1] ne sont pas utilisés. Etudions ces deux cas :

1) Si on utilise pas de pots de capacités V[i+1], on utilise les pots de capacités V[j] avec $j \in \{1,..,i\}$.
Alors $m(s,i+1) = m(s,i)$

2 Si on utilise au moins un pot de capacité V[i+1] alors calculer $m(s,i+1)$ revient à cacluler le nombre de pots minimum pour une quantité s-V[i+1] de confitures. Nombre auquel on ajoute 1, le pot de capacité V[i+1] utilisé.
Alors $m(s,i+1) = m(s-V[i+1],i+1)+1$

On cherche la configuration utilisant le moins de pots possible,donc on a bien que :

$$m(s,i+1) = \begin{cases} 0 \text{ si } s = 0 \\ min\{m(s,i), m(s-V[i+1],i+1)+1\} \text{ sinon} \end{cases} \quad (2)$$

Si la propriété est vérifiée pour i, elle est vérifiée pour i+1. Comme elle est vérifiée pour $i = 1$ elle est vérifiée pour tout $i \in \{1,...,k\}$

2.2 Un permier algorithme

On propose alors un algorithme permettant de trouver m(S) :
Où S est la quantité de confiture à stocker, V la liste des capacités des pots et i un entier indiquant que l'on peut utiliser les i premiers pots dans V. L'appel intial pour cet algorithme serait alors **Algo1(S,V,k)** où k est la taille de V.

Algorithm 1 Algo1

Require: S, V, i
  if $S = 0$ **then**
    return 0
  end if
  if $i = 0$ **then**
    return ∞
  end if
  return $min\{Algo1(S, V, i-1), Algo1(S - V[i], V, i) + 1\}$

2.3 Arbre d'appels récursifs

On propose ci-dessous la liste des appels récursifs avec à chaque fois le début, la fin ainsi que la valeur retournée lors de l'exécution de notre algorithme pour S = 5, k = 2, V[1] = 1 et V[2] = 2.

appel initial : Algo1(5,V,2)
Appel 2 : Algo1(5,V,1)
Appel 3 : Algo1(5,V,0)
Fin appel 3 : Algo1(5,V,0) renvoie ∞
Appel 4 : Algo1(4,V,1)
Appel 5 : Algo1(4,V, 0)
Fin appel 5 : Algo1(4,V,0) renvoie ∞
Appel 6 : Algo1(3,V,1)
Appel 7 : Algo1(3,V, 0)
Fin appel 7 : Algo1(3,V,0) renvoie ∞
Appel 8 : Algo1(2,V,1)
Appel 9 : Algo1(2,V, 0)
Fin appel 9 : Algo1(2,V,0) renvoie ∞
Appel 10 : Algo1(1,V,1)
Appel 11 : Algo1(1,V, 0)
Fin appel 11 : Algo1(1,V,0) renvoie ∞
Appel 12 : Algo1(0,V,1)
Fin appel 12 : Algo1(0,V,1) renvoie 0
Fin appel 10 : Algo1(1,V,1) renvoie $min(\infty, 0+1) = 1$
Fin appel 8 : Algo1(2,V,1) renvoie $min(\infty, 1+1) = 2$
Fin appel 6 : Algo1(3,V,1) renvoie $min(\infty, 2+1) = 3$
Fin appel 4 : Algo1(4,V,1) renvoie $min(\infty, 3+1) = 4$
Fin appel 2 : Algo1(5,V,1) renvoie $min(\infty, 4+1) = 5$
Appel 13 : Algo1(3,V,2)
Appel 14 Algo1(3,V,1)
Appel 15 : Algo1(3,V, 0)
Fin appel 15 : Algo1(3,V,0) renvoie ∞
Appel 16 : Algo1(2,V,1)
Appel 17 : Algo1(2,V, 0)
Fin appel 17 : Algo1(2,V,0) renvoie ∞
Appel 18 : Algo1(1,V,1)
Appel 19 : Algo1(1,V, 0)
Fin appel 19 : Algo1(1,V,0) renvoie ∞
Appel 20 : Algo1(0,V,1)
Fin appel 20 : Algo1(0,V,1) renvoie 0
Fin appel 18 : Algo1(1,V,1) renvoie $min(\infty, 0+1) = 1$
Fin appel 16 : Algo1(2,V,1) renvoie $min(\infty, 1+1) = 2$
Fin appel 14 : Algo1(3,V,1) renvoie $min(\infty, 2+1) = 3$
Appel 21 Algo1(1,V,2)
Appel 22 Algo1(1,V,1)
Appel 23 Algo1(1,V,0)
Fin appel 23 : Algo1(1,V,0) renvoie ∞
Appel 24 Algo1(0,V,1)
Fin appel 24 : Algo1(0,V,1) renvoie 0
Fin appel 22 : Algo1(1,V,1) renvoie $min(\infty, 0+1) = 1$
Appel 25 Algo1(0,V,2)

Fin appel 25 : Algo1(0,V,2) renvoie 0
Fin appel 21 : Algo1(1,V,2) renvoie $min(1,1) = 1$
Fin appel 13 : Algo1(3,V,2) renvoie $min(3,1+1) = 2$
Fin appel Initial : Algo1(5,V,2) renvoie $min(5,2+1) = 3$

Par soucis de lisibilité on représente aussi l'abre des appels récursifs pour les mêmes données d'entrée. On ne note pas les valeurs de retour pour chaque appel mais elles sont déjà indiques dans la liste précédente.

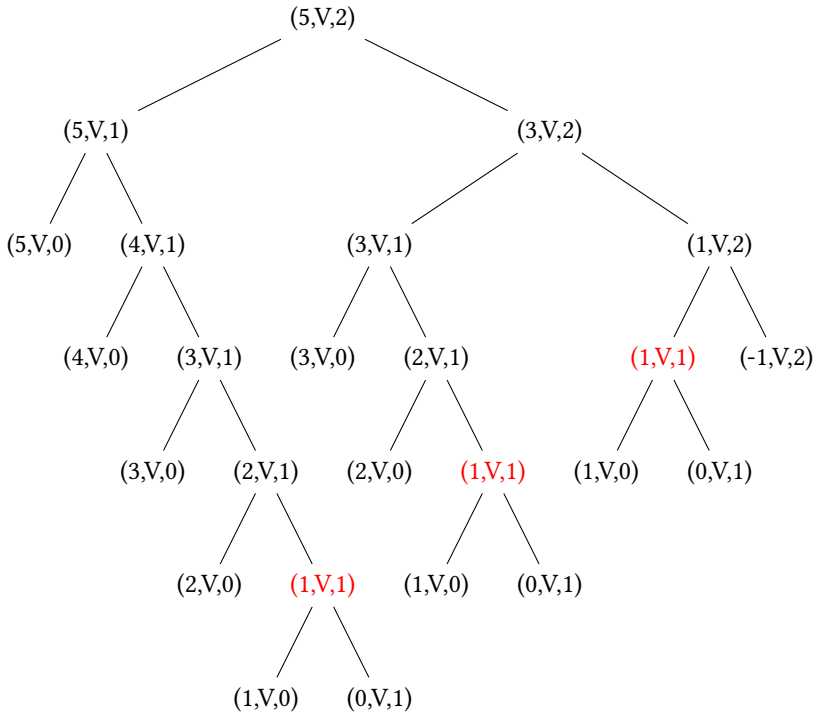

FIGURE 1 – Arbre des appels récursifs pour l'appel Alog1(5,[1,2],2)

On remarque que $m(1,1)$ est calculé trois fois. De manière plus générale, si S est impaire, $m(1,1)$,dont les appels ont été notés en rouge dans l'arbre précédent, sera calculé $m(S)$ fois.

2.4 Un second algorithme

On remarque que dans le tableau M doublement indicé contenant les valeur $m(s,i)$, la première colonne contient $m(s,0)$ pour $s \in \{0,...,S\}$. Cet première colonne contient alors ∞ en chaque case. De même la première ligne contient les $m(0,i)$ pour $i \in \{0,...,k\}$ et peut donc être remplie avec des 0 et cela sans calculs. On peut alors calculer $m(1,1)$ puisque qu'on connaît $m(0,1)$ et $m(1,0)$. De la même manière, on pourra remplir le tableau en suivant l'ordre du parcours classique d'un tableau, ligne par ligne de gauche à droite. A ceci près que la première ligne et la première colonne sont déjà remplies.

On présente ci-dessous le pseudo-code d'un nouvel algorithme possible :
Ici S, V et k sont les même que dans le premier algorithme. On va s'intérresser à la complexité spatiale et temporelle de cet algorithme.

Pour la complexité spatiale, on note n le nombre de bit sur lequel sont codés les entiers que l'on manipule. Nous faisons le choix pratique d'associer ∞ à la plus grande valeur représentable avec n bit. A la fin de l'exécution de l'algorithme, un tableau de taille $S \times k$ contenant des entiers a été systématiquement alloué. La complexité spatiale est donc en $\Theta(S \times k \times n)$. La valeur indiquée dans le Θ est alors en nombre de bit.

Etudions maintenant la complexité temporelle. L'opération élémentaire que nous choisissons pour établir cette complexité est l'affectation d'une valeur à une case de M. On constate qu'une affectation est effectuée à chaque itération de la double boucle imbriquée indexée sur S et sur k. On en déduit que la complexité temporelle de l'algorithme est en $\Theta(S \times k)$ opérations élémentaires.

Algorithm 2 AlgoOptimisé

Require: S, V, k
  $M \leftarrow$ une matrice $S \times k$ initialisée à 0
  for s from 0 to S **do**
    for i from 0 to k **do**
      if $s = 0$ **then**
        $M[s, i] \leftarrow 0$
      end if
      if $i = 0$ **then**
        $M[s, i] \leftarrow \infty$
      end if
    else
      $M[s, i] \leftarrow min\{M[s, i-1], M[s - V[i], i] + 1\}$
    end for
  end for
  return $M[s, k]$

On peut modifier notre algorithme pour que chaque case contienne l'assortiment des bocaux utilisés. Cet assortiment est représentés par une liste L de taille k. $L[i]$ est un entier indiquant le nombre de bocaux de capacités $V[i]$ utilisé pour atteindre $m(s, i)$. L'algorithme modifié est présenté ci-dessous.

Algorithm 3 AlgoOptimisé modifié

Require: S, V, k
  $M \leftarrow$ une matrice $S \times k$ initialisée à $(0, L_{s,i})$ avec $L_{s,i}$ une liste de taille k initialisée à 0
  for s from 0 à S **do**
    for i from 0 à k **do**
      if $i = 0$ **then**
        $M[s, i] \leftarrow (\infty, L_{s,i})$
      end if
    else
      if $min\{M[s, i-1], M[s - V[i], i] + 1 = M[s - V[i], i] + 1$ **then**
        $L_{s,i}[i] \leftarrow \mathrm{L}_{s,i}[i] + 1$
        $M[s, i] \leftarrow ((M[s - V[i], i] + 1), L_{s,i})$
      else
        $M[s, i] \leftarrow (M[s, i-1], L_{s,i-1})$
      end if
    end for
  end for
  return $(M[S, k], L_{S,k})$

La complexité temporelle de cet algorithme est alors en $\Theta(S \times k^2 \times n)$ bit, ce qui peut sembler excessif. On peut cependant se passer d'une telle dégradation de la complexité spatiale en utilisant une autre approche.

Algorithm 4 Backward

Require: $m(S), S, V, k, M$
  $s \leftarrow S$
  $i \leftarrow k$
  $m(s,i) \leftarrow m(S)$
  while $s > 0$ **do**
    if $m(s,i) = M[s, i-1]$ **then**
      $i \leftarrow i - 1$
      $m(s,i) \leftarrow M[s,i]$
    else
      $s \leftarrow s - V[i]$
      $L[i] \leftarrow L[i] + 1$
      $m(s,i) \leftarrow M[s,i]$
    end if
  end while
  return $M[s,k], L$

Etudions la compexité de cet algorithme. On va pour cela s'intéresser au nombre de comparaions entre deux valeurs. Ces comparaisons ont lieu à chaque tour de boucle while. Etudions le pire cas, c'est à dire le cas donnant lieu au nombre de boucle le plus élevé. Les itérations où l'on vérifie la condition $m(s,i) = M[s, i-1]$ ne font pas diminiuer s. Par ailleurs s dimininue le plus lentement lorsque la capacité V[i] que l'on déduit de s est petite. Ainsi le pire cas va avoir lieu lorsque notre jeu de pots sera constitué uniquement de pots de capacités 1. L'algorithme réalisera alors k+m(S) itérations.

On en déduit que la complexité spatiale sera un $O(k \times S + k + m(S)) = O(k(S+1) + m(S))$. En effet, en exécutant AlgoOptimisé puis backward successivement le nombre d'opérations élémentaires sera égale à la somme des nombres d' opérations effectuées dans chaque algorithme. De plus, nous avons établi uniquement une compexité pire-cas pour l'algorithme backward, cela justifie que la complexité temporelle donnée précédemment soit elle aussi une complexité pire-cas. La complexité spatiale sera en $O(k \times S + k) = O(k(S+1))$. Il s'agit de la complexité spatiale d'alogoOptimisé à laquelle on il faut ajouter l'espace mémoire occupé par la liste L renvoyée à la fin de backward. Cette complexité est bien plus intéressante que la complexité spatiale de la version naïve de l'algorithme précédemment évoqué, en particulier lorsque k devient grand. Dans la suite de ce rapport, on désigne la "concaténation" de algoOptimisé et de backward par "algorithme II".

2.5 Montrons que l'alogirthme II est de complexité pseudo-polynomiale

On rappelle que la complexité temporelle de l'algorithme II est $O(kS + m(S))$. Il est alors judicieux de s'intérresser en détail à ce qui se cache derrière k,S et m(S). k représente la taille du tableau de capacités V tandis que S représente la quantité de confitures manipulée. Lors de l'exécution de notre algorithme des opérations sont éfectués sur des quantités indexées sur S, notamment des soustractions. Le temps d'exécution de ces soustractions dépends alors de la taille mémoire de S. S va nécessiter $log_2(S)$ bits pour être stocké en mémoire. Notons p = $log_2(S)$. Alors notre complexité peut se réécrire $O(k(2^p) + m(2^p))$. Sous cette écriture, S est représenté selon sa taille mémoire et non selon sa valeur numérique.

On remarque que si p augmente d'une unité, le nombre d'opérations élémentaires double. IL en résulte que notre algorithme est de complexité exponentielle si l'on considère la taille en mémoire des données d'entrées. On a bien montré que l'algorithme II est de complexité pseudo-polynomiale.

2.6 Algorithme glouton

On peut s'intérresser à une nouvelle approche pour résoudre ce problème en utilisant notamment un algorithme glouton. On propose ci-dessous un pseudo code pour cet algorithme.

Etudions la complexité temporelle d'un tel algorithme. On prend la division euclidienne comme opération élémentaire. Le pire cas a lieu lorsqu'un pot de taille 1 est utilisé. Dans cette configuration, on réalise k itérations de la boucle. Alors on aura i = 1 et donc $s\%1 = 0$ ce qui nous fera sortir de la boucle while. Le nombre d'itérations et donc de divisions euclidiennes est alors borné par k. L'algorithme Glouton est en $\Omega(1), O(k)$.

Montrons qu'il existe des systèmes de capacités qui ne sont pas gloutons compatibles. Considérons le système 1, 20, 30. Soit une quantité S = 100 ml de confiture. l'exécution de notre algorithme glouton nous donne comme résultat 13, 3 pots de 30 ml et 10 pots de 1 ml. Cependant 13 ne peut pas valoir m(S). En effet on peut construire un assortiment de 4 pots, 2 pots de 20 ml et 2 pots de 30 ml. Ainsi la solution proposée par l'algorithme glouton pour S = 100 et V = 1, 20, 30

Algorithm 5 Glouton

Require: S, V, k
  $s \leftarrow S$
  $i \leftarrow k$
  $res \leftarrow 0$
  while $s > 0$ **do**
    $res \leftarrow res + s/V[i]$: le quotient pour la division euclidienne
    $s \leftarrow s\%V[i]$ le reste pour la division euclidienne
    $i \leftarrow i - 1$
  end while
  return res

n'est pas la solution minimale. On en déduit que V = 1,20,30 n'est pas Glouton-compatible.

Montrons cependant que tout système V tel que k = 2 est glouton-compatible. Soit S fixé et m(S) tel que défini dans le sujet. Considérons le cas où S < V[2]. Alors m(S) = S, or c'est bien le résultat renvoyé par l'algorithme glouton (car alors, S/V[2] = 0 et S%V[2] = S).

Considérons le cas où S > V[2]. L'algorithme glouton propose une configuration avec S/V[2] pots de capacités V[2] et S%V[2] pots de capcités 1. Notons $m(S)_1$ le nombre de pots dans cette configuartion . La configuration donnée par glouton contient le maxmimum de pots de taille V[2] possible, tout autre configuration contient donc moins de pots de taille V[2]. Posons c = V[2]. Considérons une nouvelle configuration différente, on note n le nombre de pots de capacité V[2] qu'elle contient en moins. Alors elle contient $c \times n$ pots de capcictés 1 en plus,On a placé la confiture de ces n pots retirés dans des pots de capacités 1, comme k = 2 c'est la seule alternative possible. Or $c > 1$, donc $c \times n > n$ et on en déduit que le nombre total de pots a augmenté par rapport à $m(S)_1$. Comme le raisonnement est fait sur n quelconque non nul, on a que toute configuration de pots valide et différente de celle proposée par glouton se compose de plus de pots que le résultat de gloutons. On en déduit que Glouton renvoit bien m(S).

Ainsi pour tout S > 0, si k = 2 alors glouton renvoie bien m(S). Donc si k = 2, le système de capacité est glouton-compatible.

2.7 Etude de l'algorithme de test de compatibilité

Calculons la complexité temporelle de cet algorithme. Pour des soucis de lisibilité, on pose $T = V[k-1] + V[k] - 1 - V[3] - 2$. La première boucle réalise T itérations, la seconde en réalise k et à chaque itération de la seconde boucle on réalise deux appels à algoGlouton qui est en $\Omega(1), O(k)$. Il en résulte une complexité en $\Omega(T \times k), O(T \times k^2)$. Montrons que ce résultat est bien polynomial (et non pas speudo-polynomial). Comme indiqué précedemment, k représente la taille de V. T quand à lui est paramétré par V. Plus précisément, la taille en mémoire de T est bornée par la taille en mémoire des éléments V[k], v[k-1] et v[3]. Notons n le nombre de bits sur lequel on écrit les valeurs de V. La taille en mémoire de V est $k \times n$ et la taille en mémoire de T vaut n. Alors notre complexité en fonction de la taille mémoire des entrées s'écrit $O(n \times k \times n) = O(kn^2)$, on retrouve bien une complexité polynomiale.

Nous sommes maintenant munis de plusieurs algorithmes pour résoudre notre problème. Il est temps de les implémenter, de les tester puis de comparer leurs performances.

3 Mise en oeuvre et analyses

3.1 Choix du langage et implémentation

Nous avons fait le choix d'utiliser Python 3 pour implémenter nos algorithmes. Ce choix se justifie par la simplicité du langage, notamment en ce qui concerne l'utilisation de la bibliothèque graphique matplotlib. Nous allons ainsi présenter le jeu de tests que nous avons effectué sur nos différents algorithmes. Veuillez noter la présence d'une erreur commune à l'ensemble de nos graphiques. Le temps y est indiqué en milliseconde alors que l'unité exacte est la seconde. Nous vous prions de bien vouloir nous excuser pour ces erreurs.

3.2 Evolution du temps de calcul en fonction de S

Dans cette première partie nous étudions le temps d'exécution de nos algorithmes lorsque S devient très grand. On présente ci-dessous un premier graphique concernant uniquement l'algorithme I :

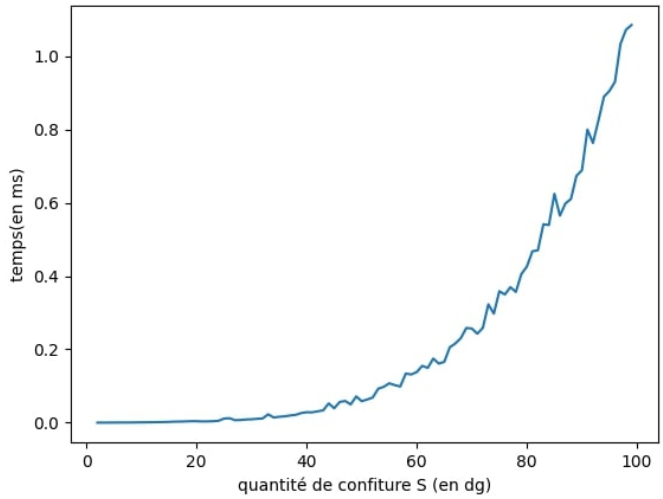

FIGURE 2 – Temps d'exécution de l'algorithme I en fonction de S. K = 7, d = 2

La quantité de confiture considérée est petite mais suffisante pour qu'il en découle des temps de calcul significatifs. Nous souhaitons comparer les performances de nos algorithmes, aussi est-il plus intéressant d'afficher simultanément les performances des Algorithmes I,II et III sur les mêmes données d'entrées.

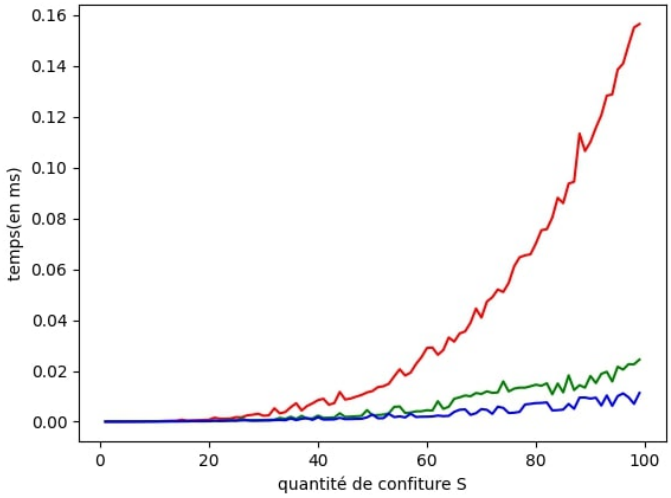

FIGURE 3 – Temps d'exécution de l'algorithme I(rouge),II(vert) et III(bleu) en fonction de S. K = 7, d = 2

On constate rapidement que l'algorithme I, en rouge, voit son temps d'exécution évoluer très rapidement alors que S ne dépasse pas 100 dl. Les différences de performances entre l'algorithme II et l'algorithme III est moins flagrante, aussi nous avons représenté le même graphique mais sous une échelle logarithmique.

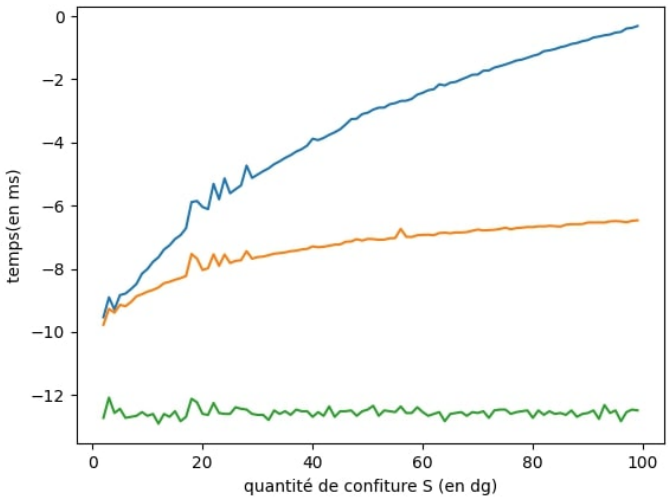

FIGURE 4 – Temps d'exécution de l'algorithme I(bleu),II(orange) et III(vert) en fonction de S. K = 7, d = 2

nb : le temps en abscisse est ici en log(s) et non en ms, veuillez nous excuser pour cette coquille.

Sur ce graphique, la différence de performance entre l'algorithme II et l'algorithme III apparaît clairement et ce sans nécessité de prendre de grandes valeurs pour S (ce qui permet de les comparer sur les mêmes données avec l'algorithme I). Cette différence de performance s'explique par le fait que le temps d'exécution de l'algorithme III est paramétré avant tout par k et T une donnée dependante de V et non pas S. Ainsi il est logique d'observer un temps d'exécution quasi constant pour l'algorithme III quand S varie, contrairerment à l'algorithme II dont le nombre d'opérations élémentaire est directement indexé sur S. En effet rappelons que l'aglorithme II a une complexité en $O(kS + m(S))$,pseudo-polynomial en S, tandis que l'algorithme III est en $O(Tk^2)$ où T ne dépend pas de S. Etudions à présent le comportement de nos algorithmes lorsque S est fixé (cette fois-ci à une valeur plus conséquente) et lorsque k varie.

3.3 Evolution du temps de calcul en fonction de k et de d

Le travail d'analyse de nos algorithmes n'est pas terminé. En effet le nombre d'opération effectués ne dépend pas que de S mais aussi de k, la taille du tableau en entrée. Nous allons alors étudier L'évolution du temps de calcul en fonction de k. On utilise, comme dans les tests précédents, le système de capacité glouton-compatible présenté dans l'énoncé. On va surtout chercher à comparer l'algorithme II et l'algorithme III. A priori les deux algorithmes sont en complexité polynomiale par rapport à k (et même de complexité linéaire). Pour minimiser le rôle joué par S, on décide de fixer S à une valeur petite, ici 42. On fait alors varier k et on obtient le graphique suivant :

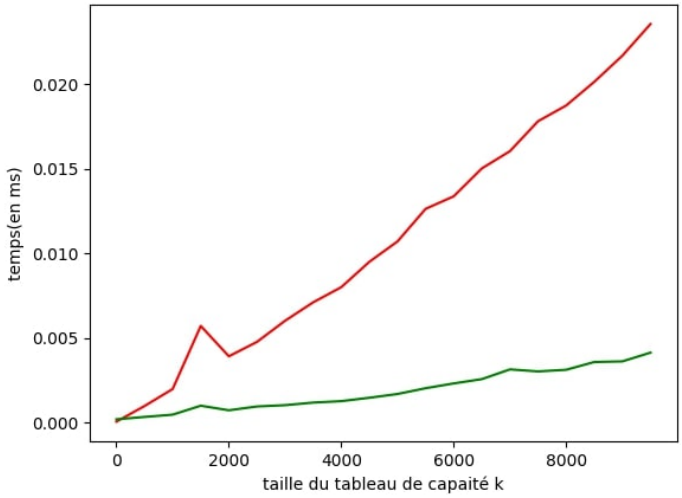

FIGURE 5 – Temps d'exécution de l'algorithme II(rouge) et III(vert) en fonction de k. S = 42, d = 2

Utiliser ici une échelle polynomiale ne semblait pas nécessaire. On constate une évolution quasi linéaire du temps de calcule lorsque k augmente. Cela concorde avec nos hypothèses. La différence de temps d'exécution (quand k varie uniquement) entre l'algorithme I et l'algorithme II est de l'ordre d'une constante qui n'apparaît alors pas dans nos formules de complexités car elles sont exprimées en utilisant des O.

Enfin on peut s'intérresser au comportement de nos algorithmes lorsque d varie. On a fait varier d en observant l'algorithme II uniquement puis l'algotithme III et illustré le résultat dans les graphiques-dessous.

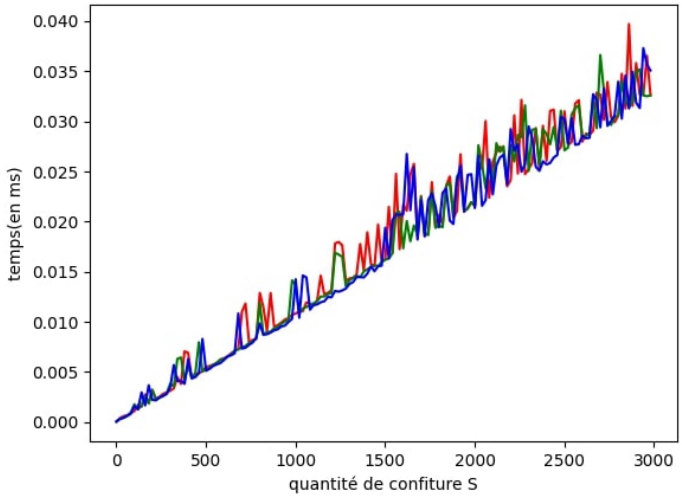

FIGURE 6 – Temps d'exécution de l'algorithme II en fonction de k. S = 500. En bleu d = 2, en rouge d = 3 en vert d = 4

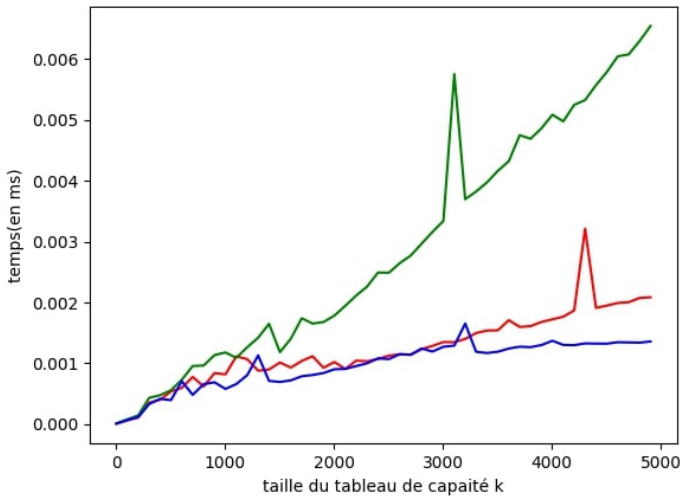

FIGURE 7 – Temps d'exécution de l'algorithme III en fonction de k. S = 500. En bleu d = 2, en rouge d = 3 en vert d = 4

On observe aucune différence significative pour l'algorithme II. En revanche on constate un écart de temps de calcul quand d = 4 et quand d = 2 par exemple pour l'algorithme III. Nous n'avons à ce jour pas d'explications abouties à fournir concernant ce résultat. Nous pouvons cependant formuler une hypothèse. Lorsque d = 4, et lorsque k devient grand, les capacités deviennent bien plus grandes. Si k = 101, $V[k] = 4^{100}$. Or glouton effectue des opérations, notamment des divisions sur les V[i]. Soit d = 2, 3, 4.On note $V[k]_d$ la dernière capacité dans notre système particulier défini dans l'énoncé. On a alors que :

$$V[k]_4 = (V[k]_2)^2$$

Faire une division sur $V[k]_4$ est alors plus long que de faire une divison sur $V[k]_2$, car ce dernier est quadratiquement plus petit. Il est possible que cela soit en partie à l'origine de l'écart observer sur ces graphiques.

Nous allons à présent nous intéresser à la probabilité de trouver un système de capacité glouton-compatible est à l'erreur engendrée par l'exécution de l'algorithme III sur une entrée qui n'est pas glouton-compatible.

3.4 Etude des systèmes de capacités générés aléatoirement

Est-ce qu'un système de capcité glouton-compatible est quelque chose de rare ? Et si oui, quels sont les risques à utiliser notre algorithme glouton lorsque le système n'est pas glouton-compatible ? Nous allons essayer de répondre à ces deux questions dans cette partie.

Pour tenter de répondre à la première question, nous allons fixer S puis faire varier k, pour chaque valeur de k nous allons créer 100 systèmes et mesurer la proportion de ces systèmes qui est glouton-compatible. Pour S fixé à 100 on obtient le graphique suivant :

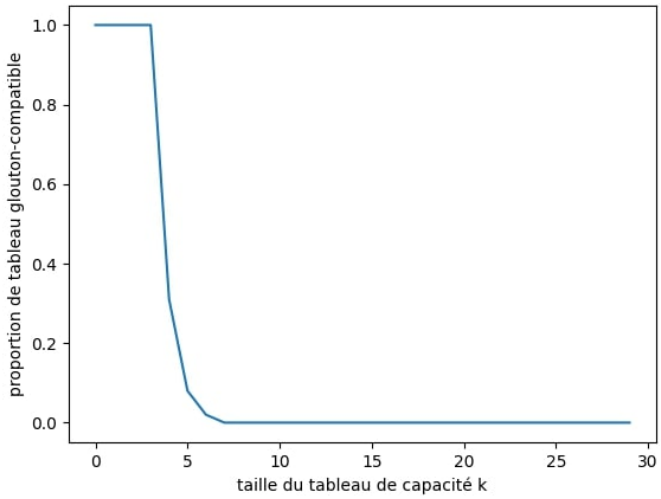

FIGURE 8 – Proportion de systèmes glouton-compatible en fonction de k, S fixé à 100

On remarque que lorsque k = 1 et k = 2 tout les sytèmes créés sont glouton-compatibles. Le cas k = 1 est trivial et le cas k = 2 concorde bien avec le résultat démontré précédemment selon lequel tout système de capacité tel que k = 2 est glouton-compatible. Cette proportion chute ensuite drastiquement jusqu'à devenir proche de 0 (et presque systématiquement sous 1 %) lorsque k dépasse 7. Il semble, de manière expérimentale, que les systèmes gloutons-compatibles soient très rares dès que k dépasse 10. Ainsi dans la pratique il n'est pas judicieux de considérer qu'un système de capacité choisi de manière aléatoire est glouton-compatible. A noter que nous avons fixé S à 100 pour mener ce test. On pourrait se demander ce qu'il en est lorsque S varie. Nous avons effectué des tests similaires pour S = 10,1000 et 5000. Les graphiques obtenus étaient sensiblement identiques au graphique ci-dessous. Ainsi nous présentons un seul graphique afin de ne pas surchager ce rapport de graphiques.

Ainsi, a priori, les systèmes glouton-compatibles se font rares. On peut être alors invité à se demander si l'exécution de l'algorithme glouton sur un système non compatible entraîne un écart significatif avec la solution exacte et si oui, alors quantifier cet écart. Dans un premier temps on étudie l'évolution en faisant varier k, puis pour chaque valeur de k on calcule la moyenne des écarts pour S variant entre p_{max} et $f \times p_{max}$. On obtient le graphique suivant :

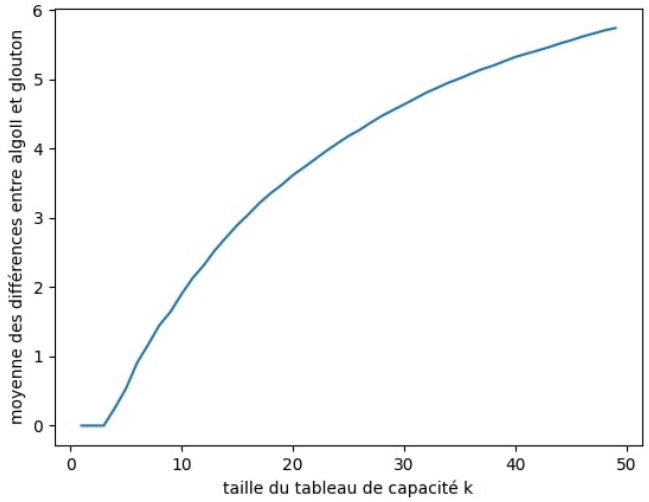

FIGURE 9 – Ecart moyen entre l'algorithme II et l'algorithme III pour S allant de p_{max} à $f \times p_{max}$ en fonction de k avec f = 20 et p_{max} = 10

On observe que lorsque k = 1 ou 2, l'écart est nul, car tout les systèmes sont gloutons compatibles. L'écart moyen augmente avec régularité lorsque k augmente jusqu'à atteindre 5.8 lorsque k vaut 50. Il pourrait être intérressant d'utiliser des outils statistiques pour quantifier la gravité d'un tel écart moyen sachant que S varie entre 10 et 200. Sans ces outils on peut se limiter à dire que cet écart ne semble pas trop important par rapport à S (presque 2 fois inférieur à la plus grande valeur prise par S). On réalise par la suite un test similaire, mais où cette fois on fait varier S puis pour chaque valeur de S entre p_{max} et $f \times p_{max}$ on mesure l'écart moyen pour 100 systèmes de capcités générés aléatoirement et non compatibles avec une taille k fixée (ici 20). On affiche alors l'écart moyen en fonction de S et on obtient le graphique ci-dessous :

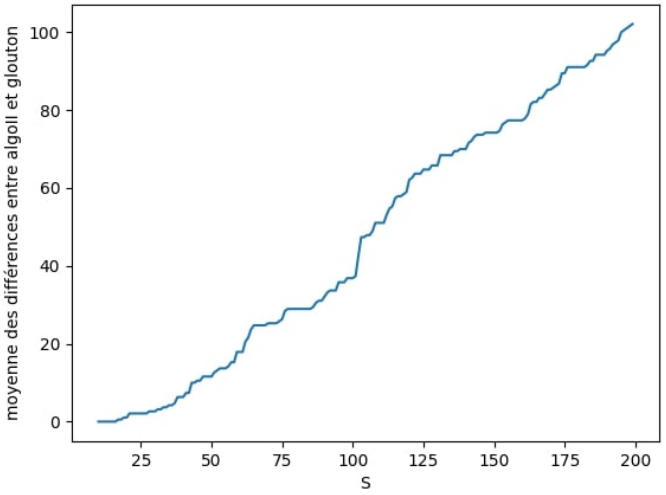

FIGURE 10 – Ecart moyen entre l'algorithme II et l'algorithme III sur 100 systèmes de capacités aléatoires en fonction de S avec k fixé à 10

On constate que faire augmenter S fait augmenter l'erreur moyenne sur 100 système de capacités. On constate par ailleurs qu'en général, l'ordre de grandeur du nombre d'erreur moyenne est inférieur à l'ordre de grandeur de S lorsque k = 25.

Enfin on trace les pires écarts pour k fixé à 10 et S allant de p_{max} à $f \times p_{max}$:

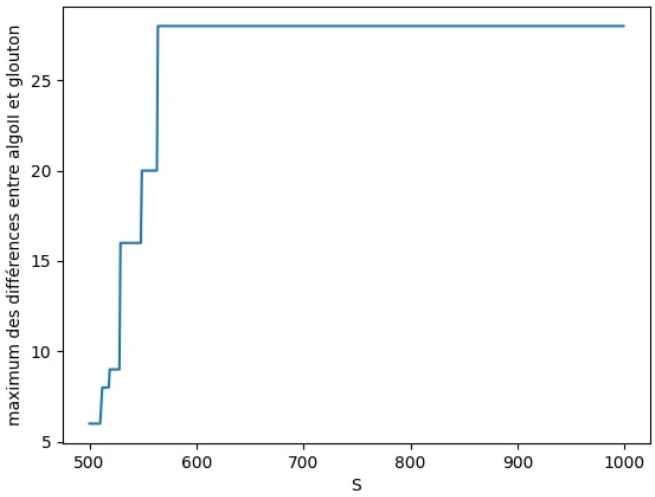

FIGURE 11 – Erreur maximum entre l'algorithme II et l'algorithme III en fonction de S, k = 10

Pour chaque valeur de S , on réalise le test d'écart entre les deux algorithmes sur 100 systèmes de capacités de taille 10. On observe un phénomène particulier, l'erreur maximum finit par atteindre un palier qui ne va pas être dépassé : 28. Nous ne sommes pas en mesure d'expliquer ce phénomène, cependant on constate avec intérêt que cette écart devient rapidement petit devant S. Ce résultat nous sembe cependant si curieux que nous préférons ne pas en tirer trop de conclusion, il pourrait être dû notamment à la manière dont nous générons nos systèmes de capacités même si cela nous semble peu probable.

In fine, il existe en pratique peu, très peu de systèmes de capacités glouton-compatible. Utiliser l'algorithme III sur des systèmes non-gloutons-compatibles entraine une erreur qui augmente lorsque S augmente et lorsque k augmente. Cette erreur reste cependant, en ordre de grandeur, inférieure à l'ordre de grandeur de S et de k. C'est alors à l'utilisateur de choisir si il préfère une solution exacte mais coûteuse en temps et en espace lorsque k et S deviennent grand : L'algorithme II ; ou bien un algorithme qui propose une solution en très peu de temps devant S et k mais une solution qui sera très rarement une solution exacte, mais une approximation acceptable : l'algorithme III. Cela conclut ainsi notre rapport. Ce projet nous a permis de traiter un problème d'optimisation combinatoire en nous sensibilisant à la problèmatique des algorithmes de complexité pseudo-polynomiale. Il nous a aussi permis de développer un algorithme glouton et de comparer ses performances et son utilité pratique. Nous espérons que le travail que nous proposons est clair. Nous reviendrons au besoin sur les éventuelles questions suscités par la lecture de ce rapport avec plaisir lors de la soutenance de ce projet.