Rapport TP2 INF4705

Florentin Burgeat (1809123), Elliot Sisteron (1807165) 20 novembre 2015

Introduction

Pour ce deuxième TP d'analyse et conception d'algorithmes, on se propose de résoudre un problème d'optimisation combinatoire. On dispose de n objets ayant chacun un certain volume. On voudrait faire rentrer ces objets dans plusieurs boîtes d'une capacité fixe, de manière à maximiser le volume total utilisé.

Soit $v = (v_1, v_2, ..., v_n)^T$ le vecteur d'entiers positifs représentant les volumes des objets. On pose $m \leq n$ le nombre de boîtes et c leur capacité, deux autres entiers positifs. On considère aussi que $\forall j \in [\![1, n]\!],\ v_j \leq c$, quitte à retirer des objets.

On souhaite donc résoudre ce PLNE :

$$\begin{aligned} \underset{M \in \mathcal{M}_{m,n}([\![0,1]\!])}{\text{Maximiser}} \quad & ||Mv||_1 \\ \text{Sous les conditions} \quad & \forall j \in [\![1, n]\!],\ \sum_{i=1}^{m} m_{i,j} \leq 1, \\ & \forall i \in [\![1, m]\!],\ \sum_{j=1}^{n} v_j m_{i,j} \leq c. \end{aligned}$$

$M \in \mathcal{M}_{m,n}([\![0, 1]\!])$ constitue une matrice où $m_{i,j} = 1$ si l'on met l'objet j dans la boîte i, et $m_{i,j} = 0$ si on ne le met pas. La première condition se traduit naturellement par : « une fois qu'un objet est utilisé, on ne peut pas le mettre dans une autre boîte ». Et, la seconde est équivalente à : « le volume que la boîte contient ne peut pas excéder la capacité de la boîte ». $x = Mv \in [\![0, c]\!]^m$ constitue le vecteur de la solution, où x_i est le volume utilisé dans la boîte i. De ce fait, $||x||_1$ constitue le volume total utilisé.

Pour pouvoir résoudre ce PLNE, nous utiliserons plusieurs algorithmes réalisés à l'aide de « patrons de conception » ; respectivement : le patron glouton, la programmation dynamique et le recuit simulé. À l'aide d'une analyse théorique de leur complexité et de leur efficience ; puis, grâce à des tests empiriques, nous essayerons de déterminer lequel est le plus approprié à notre problème.

Table des matières

1 Analyse théorique des algorithmes étudiés

On rappelle ici rapidement le fonctionnement des algorithmes, puis on analyse leur complexité.

1.1 Algorithme vorace

1.1.1 Présentation de l'algorithme

Un algorithme suivant un patron de conception vorace procède par amélioration successive ; en prenant l'option qui semble la meilleure à l'instant du choix. Dans notre cas, le « meilleur choix » à faire, sans considérer l'état antérieur ou postérieur de notre système, est de choisir le plus gros objet et de le mettre dans une boite. Pour ce faire, on choisira la plus petite boite ayant assez de place pour accueillir l'objet. On va donc d'abord trier les items par ordre décroissant de volume, puis on parcoure-ra les boîtes par espace restant.

Au niveau matriciel, cela se traduit en écrivant le vecteur v par ordre de volume décroissant. Ensuite, $\forall j \in [\![1, n]\!]$ il suffit d'ajouter l'objet j sur la ligne $i_{min}(j) = argmin_{i \in [\![1,m]\!]} \{x_i \mid v_j \leq c - x_i\}$ de $x = Mv$, si cette ligne existe.

Il est très important de noter qu'un tel algorithme ne sera pas forcément exact : il approximera plus ou moins bien l'extremum recherché en fonction de l'échantillon étudié. Prenons un contre exemple simple pour le démontrer :

$$\begin{array}{ll} n = 3 & m = 1 \\ v = (5, 4, 3)^T & c = 7 \end{array}$$

Du fait du choix hâtif de l'algorithme, la solution sera ici 5 alors qu'il est tout à fait possible de remplir complètement la boîte avec $4 + 3 = 7$.

1.1.2 Complexité

En utilisant le tri rapide, on a d'abord un premier bloc algorithmique d'une complexité $O(n \cdot log(n))$. Qu'en est-il du calcul de la ligne minimale ? En fait, en utilisant une structure de donnée ordonnée, on peut limiter la recherche du minimum en $O(1)$, sous condition d'avoir un ajout/retrait/modification en $O(log(m))$ dans la structure. Il n'en reste pas moins qu'il nous faudra entre n et $m \cdot n$ itérations pour calculer la solution (ajout direct vs. parcours de toutes les boites avant l'ajout). Il faut ajouter à cela le coût de l'opération de modification d'une ligne de x, qui sera donc de $O(log(m))$ avec notre structure de donnée. Il nous reste ainsi une complexité de $O(n \cdot (m \cdot log(m) + log(n)))$ dans le pire des cas.

1.1.3 Algorithme de programmation dynamique

1.1.4 Présentation de l'algorithme

Pour écrire un algorithme avec un patron de programmation dynamique, il faut appliquer un raisonnement similaire à celui « diviser pour régner » ; dans le sens où l'on cherche une relation de récurrence entre une partie de solution et une autre partie de solution appliquée à un échantillon plus important. La différence fondamentale se fait au niveau de la construction de la

solution ; là où l'algorithme « diviser pour régner » va utiliser une définition récursive, la solution utilisant la programmation dynamique calculera tout dans un tableau de manière itérative. En se « souvenant » des calculs précédent, ce schéma de programmation nous permet d'éviter plusieurs appels de fonction pour une même valeur ; en plus d'alléger la pile des appels récursifs.

En appliquant cela à notre problème, en connaissant la solution pour un agencement de boîte (i.e. le nombre de boîte de chaque capacité) et un ensemble d'objets ; il est possible de trouver la solution avec le même agencement pour ce même ensemble complété d'un autre objet. Vu que l'on teste chaque agencement possible avant d'ajouter ou de laisser de côté un objet, on est sûr d'arriver à la solution exacte.

1.1.5 Complexité

Il est intéressant d'énumérer ces agencements ; du fait du caractère combinatoire de notre problème, il y en a :

$$\binom{m+c}{c}$$

De plus, comme on a n objets, on utilisera un espace mémoire de l'ordre de $O\left(n \cdot \binom{m+c}{c}\right)$. À c fixé, d'après la formule de Stirling, on a un espace d'ordre $O\left(nm \cdot \frac{m^c}{c!}\right) = O\left(nm^{c+1}\right)$ consommé.

On itère sur ces possibilités dans un ordre précis, et à chaque itération, on fait au plus c opérations. Soit une complexité temporelle de l'ordre de $O\left(n \cdot c \cdot \binom{m+c}{c}\right)$ et $O\left(nm^{c+1}\right)$ à c fixé.

1.2 Algorithme du recuit simulé

1.2.1 Présentation de l'algorithme

L'algorithme du recuit simulé est un algorithme de recherche de solution dans un grand espace. Il s'applique souvent dans le cas de problème d'optimisation à espace discret mais de taille importante, ce qui en fait une arme de prédilection pour les problèmes combinatoires.

De la même façon que l'algorithme « Hill Climbing », le recuit simulé consiste à améliorer au hasard une solution pendant un certain nombre d'itérations fixé au préalable. Une différence majeure est toutefois la présence d'une notion de « température », permettant de cibler un extremum local en se basant sur une méta-heuristique. Plus cette température est froide, plus la recherche est spécialisée autour de cet extremum local. Là où l'algorithme « Hill Climbing » prend plusieurs milliers d'itérations avant de se rapprocher de son objectif, le recuit simulé le cible dans le but de l'atteindre beaucoup plus rapidement.

En considérant k_{max} le nombre de pas maximum, P le palier de refroidissement :

— plus on choisit un nombre de pas élevé, plus on se déplacera longtemps dans les solutions possible ;

— plus on choisit un palier de refroidissement élevé, plus on pourra se déplacer loin de cette solution.

Ce qui nous donnera, en fonction des paramètres choisis, une approximation plus ou moins correcte de l'extremum global.

1.2.2 Complexité

On a déjà $k_max \cdot P$ itérations dans l'algorithme, mais cela est dans le pire des cas. En effet, nous avons optimisé l'algorithme de sorte qu'il s'arrête si toutes les boites sont pleines ou si tous les objets ont été utilisés.

Il faut ajouter à cela un coup dépendant de la forme de la solution. Une solution est de taille $n \cdot m$, et nous sommes amenés parfois à effectuer des affectations de solution. Cela a un coup linéaire, soit $n \cdot m$.

Donc, un total de $O\left(k_{max} \cdot P \cdot m \cdot n\right)$ dans le pire des cas.

2 Analyse empirique de la consommation des algorithmes étudiés

2.1 Quelques remarques avant de commencer

2.1.1 Cadre expérimental

Nous avons principalement travaillé sur un MacBook Air mi-2011, dont la configuration est la suivante :

FIGURE 1 – Cadre expérimental

Le langage choisi est le C++ et compilateur utilisé est g++.

2.1.2 Jeux de données

Nous utiliserons les jeux de données fournit dans le cadriciel de ce deuxième TP. Nous ne travaillerons pas avec les exemplaires de grande taille pour l'algorithme de programmation dynamique car ils prendraient une durée bien trop longue pour atteindre un résultat.

2.2 Présentation du programme réalisé

Les programmes C++ réalisés se trouvent dans le dossier du rapport.

2.2.1 Installation et usage

L'ensemble des instructions à suivre pour installer, exécuter et utiliser nos programmes sur une distribution Linux ou MacOS se trouvent dans le fichier *README.txt*.

2.2.2 Explications

L'ensemble des fonctions résolvant le problème sont définies dans le fichier *solver.h*, elles sont implémentées dans le fichier *solver.cpp*

Quelques notes :

— nous avons utilisé un multiset pour optimiser la récupération du minimum dans l'algorithme vorace ;

— en utilisant un unordered_map à la place d'un map pour la programmation dynamique, nous avons réduit drastiquement le temps de calcul ;

— en coupant l'exécution du recuit simulé lorsque la solution obtenue est la meilleure pour sûr (i.e quand toutes les boîtes sont remplies ou quand tous les volumes ont été utilisé), nous observons beaucoup d'échantillon pour lesquels le temps de calcul est moindre.

2.2.3 Résultats du programme

La difficulté est ici d'effectuer notre analyse dans un cadre multidimensionnel. Sachant que dans tous les cas, l'algorithme utilisé regardera chaque item, on peut déjà considérer n comme facteur commun de difficulté de résolution. Seulement, il faut aussi considérer le nombre de boîte m et leur capacité c. Ces trois valeurs sont intimement liée au temps de calcul, et donc il faut aussi les prendre en compte. Dans nos jeux de données, c est fixé à 100, ce qui nous simplifiera la tâche.

Voici les résultats pour l'algorithme vorace, après traitement dans un tableur, et en les triant par temps d'exécution croissant :

Pour l'algorithme vorace :

Coef (n*m)	n	m	c	Vorace
20	10,00	2,00	100,00	2,17E-05
64	32,00	2,00	100,00	2,61E-05
30	10,00	3,00	100,00	2,92E-05
40	10,00	4,00	100,00	3,00E-05
60	10,00	6,00	100,00	3,12E-05
96	32,00	3,00	100,00	3,15E-05
128	32,00	4,00	100,00	3,35E-05
192	32,00	6,00	100,00	3,61E-05
200	100,00	2,00	100,00	4,15E-05
300	100,00	3,00	100,00	4,51E-05
400	100,00	4,00	100,00	5,00E-05
600	100,00	6,00	100,00	6,38E-05
640	32,00	20,00	100,00	6,56E-05
632	316,00	2,00	100,00	1,01E-04
2000	100,00	20,00	100,00	1,19E-04
948	316,00	3,00	100,00	1,21E-04
1264	316,00	4,00	100,00	1,30E-04
1896	316,00	6,00	100,00	1,55E-04
3600	100,00	36,00	100,00	2,03E-04
2000	1000,00	2,00	100,00	2,38E-04
3000	1000,00	3,00	100,00	2,99E-04
4000	1000,00	4,00	100,00	3,30E-04
6320	316,00	20,00	100,00	3,45E-04
6000	1000,00	6,00	100,00	3,98E-04
11376	316,00	36,00	100,00	4,96E-04
20000	1000,00	20,00	100,00	8,57E-04
63200	316,00	200,00	100,00	1,38E-03
36000	1000,00	36,00	100,00	1,44E-03
200000	1000,00	200,00	100,00	7,26E-03
632000	1000,00	632,00	100,00	1,12E-02

FIGURE 2 – Résultats de l'algorithme vorace

On remarque que ni n, ni m n'apparaît par ordre croissant, et c est constant (égale à 100) cependant le produit $n \cdot m$ lui est croissant, on l'utilisera donc comme valeur d'abscisse pour analyser les algorithmes.

Pour l'algorithme dynamique :

Coef (n*m)	n	m	c	Dynamique
20	10,00	2,00	100,00	3,41E+00
30	10,00	3,00	100,00	1,19E+02

FIGURE 3 – Résultat de l'algorithme dynamique

Dû au temps d'exécution extrêmement long, nous n'avons obtenu que les vingt premiers résultats, ce qui ne nous donne que très peu de points pour effectuer l'analyse.

Pour l'algorithme de recuit simulé :

Coef (n*m)	n	m	c	Recuit Simulé
2000	1000,00	2,00	100,00	9,91E-04
632	316,00	2,00	100,00	1,33E-03
3000	1000,00	3,00	100,00	3,33E-03
4000	1000,00	4,00	100,00	6,33E-03
948	316,00	3,00	100,00	2,95E-02
1264	316,00	4,00	100,00	3,51E-02
200	100,00	2,00	100,00	3,52E-02
6000	1000,00	6,00	100,00	4,30E-02
63200	316,00	200,00	100,00	6,12E-02
1896	316,00	6,00	100,00	8,81E-02
300	100,00	3,00	100,00	1,13E-01
64	32,00	2,00	100,00	1,26E-01
400	100,00	4,00	100,00	1,35E-01
60	10,00	6,00	100,00	1,55E-01
640	32,00	20,00	100,00	1,71E-01
20	10,00	2,00	100,00	1,80E-01
96	32,00	3,00	100,00	1,97E-01
20000	1000,00	20,00	100,00	2,09E-01
30	10,00	3,00	100,00	2,13E-01
128	32,00	4,00	100,00	2,15E-01
40	10,00	4,00	100,00	2,36E-01
600	100,00	6,00	100,00	2,79E-01
192	32,00	6,00	100,00	3,44E-01
632000	1000,00	632,00	100,00	5,15E-01
6320	316,00	20,00	100,00	7,54E-01
2000	100,00	20,00	100,00	8,49E-01
36000	1000,00	36,00	100,00	1,23E+00
11376	316,00	36,00	100,00	1,52E+00
3600	100,00	36,00	100,00	1,53E+00
200000	1000,00	200,00	100,00	6,90E+00

FIGURE 4 – Résultat de l'algorithme de recuit simulé

Dans ces résultats on remarque que le tri par ordre croissant de temps des temps d'exécution ne montre aucun pattern visible aussi bien sur le nombre d'item que sur le nombre de boîtes.

2.3 Algorithme vorace

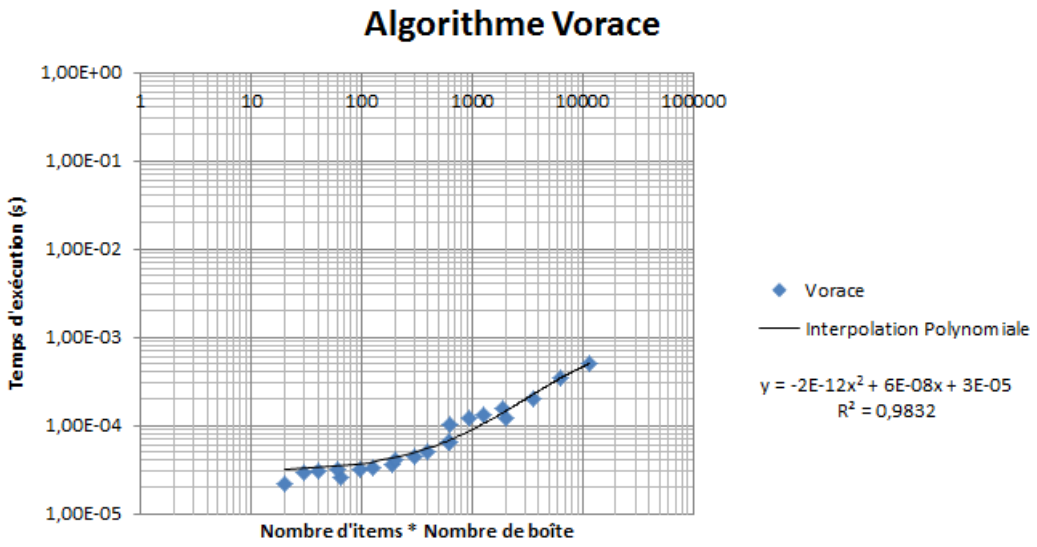

FIGURE 5 – Temps d'exécution de l'algorithme vorace en fonction de $n \cdot m$

Ce test n'est pas très concluant et ne montre rien de particulier, si ce n'est qu'une partie de la courbe dépend bien de $n \cdot m$. Nous utiliserons plutôt un test des constantes du fait du caractère bidimensionnel de l'échantillon. Nous considérons $T(n, m) = n \cdot (m \cdot log(m) + log(n))$ l'allure estimée de notre courbe.

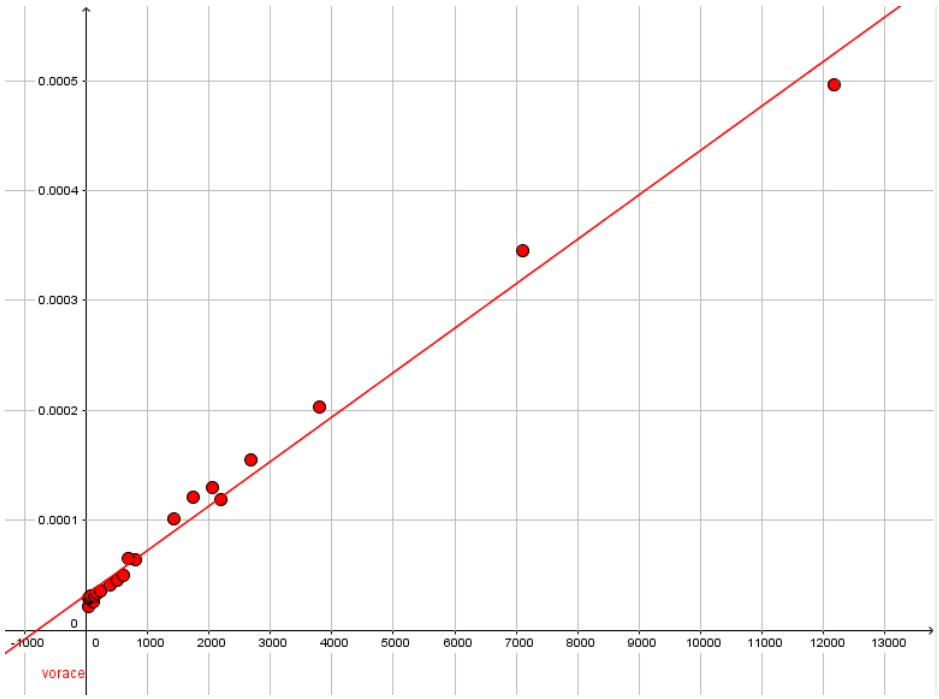

FIGURE 6 – Test des constantes sur l'algorithme vorace

La droite est d'équation $y = 4.04 \cdot 10^{-8} + 3.23 \cdot 10^{-5}$. Le résidut de cette interpolation linéaire est $R^2 = 0.988$ ce qui est plutôt satisfaisant.

On en déduit donc que la constante multiplicative de l'algorithme vorace est $4.04 \cdot 10^{-8}$ et a

un cout fixe de $3.23 \cdot 10^{-5}$.

On peut donc estimer le temps moyen de calcul de cet algorithme à $T(n, m) = 4.04 \cdot 10^{-8} n \cdot (m \cdot log(m) + log(n)) + 3.23 \cdot 10^{-5} s$.

2.4 Algorithme de programmation dynamique

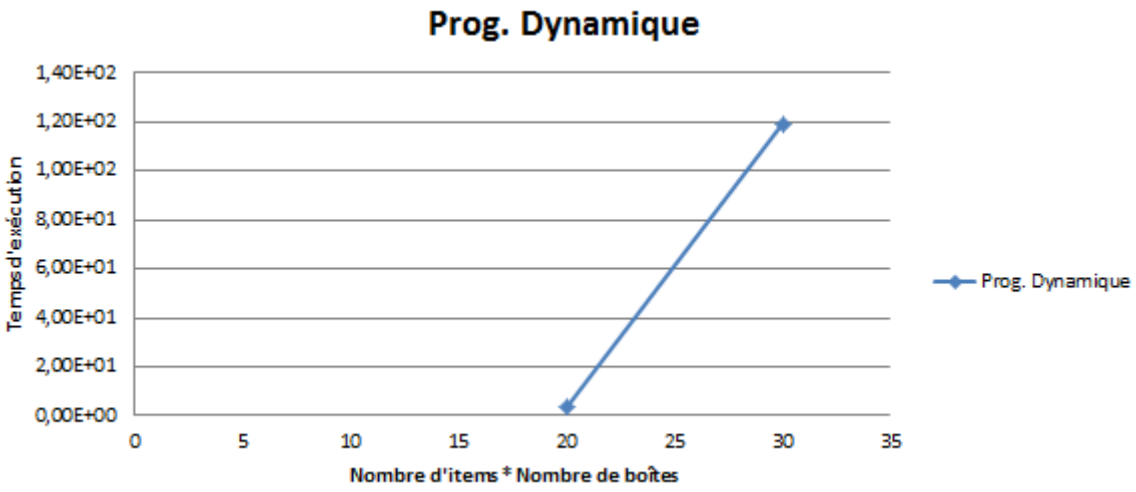

FIGURE 7 – Algorithme par programmation dynamique

Vu que nous ne disposons que de deux points, cette courbe n'est pas très intéressantes. On peut toutefois dire que la pente de la courbe est ici une pente exponentielle.

2.5 Algorithme du recuit simulé

Les valeurs des différentes variables de l'algorithme ont été fixés et sont les mêmes pour chaque exécution ($k_{max} = 1000; T = 10.0; P = 100.0; \alpha = 0.9$).

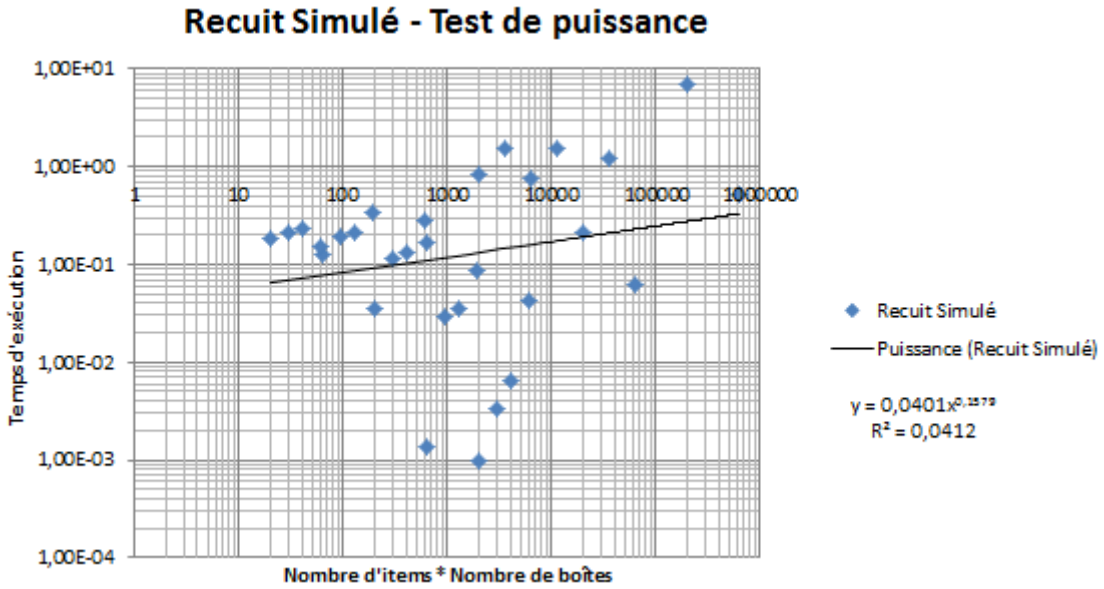

FIGURE 8 – Temps d'exécution de l'algorithme de recuit simulé en fonction de $n \cdot m$

On aurait pu s'attendre à une fonction linéaire de $n \cdot m$ mais ce n'est ici clairement pas le cas. Cela vient du caractère aléatoire de l'algorithme, auquel s'ajoute notre optimisation qui réduit de manière très visible le temps de calcul sur certains échantillon. En effet, dans le cas où l'algorithme tombe sur une solution « parfaite », il s'arrête en plein milieu d'exécution, ce qui explique ces nombreux fossés sur la courbe.

On voit que le test de puissance ne s'applique vraiment pas à ce cas là, le résidu de la fonction de puissance étant $R^2 = 0.0412$, du fait des oscillations de la courbe. Le temps moyen d'exécution est de $0,539s$.

3 Quel algorithme utiliser ?

Chacun des algorithmes utilisé a ses qualités et ses défauts.

L'algorithme vorace tout d'abord : cet algorithme plutôt simple n'est pas le plus efficace. Il fait une seule tentative de remplissage en ayant préalablement trié ses items et propose celle ci comme solution. En général cette méthode amènera sur un bon résultat, mais ne certifie pas du tout qu'il le sera toujours.

Points positifs
- Facile à implémenter
- Assez rapide
- Relativement simple de reconstruire la solution

Points négatifs
- Non optimal
- Pas de garantie sur la qualité de la solution

L'algorithme à programmation dynamique : il a la particularité d'être exact. Il parcourt toutes les possibilités afin de trouver le meilleur résultat. Cependant, ce souci du détail se paye par un temps d'exécution exponentiel ; ce qui le rend presque inutilisable pour les exemplaires de grande taille.

Points positifs
- Examine toutes les possibilités
- Optimal

Points négatifs
- Plus difficile à implémenter
- Difficile de reconstruire la solution
- Temps d'exécution, espace mémoire demandé exponentiel

L'algorithme du recuit simulé : Il améliore itérativement une solution en ciblant un extremum local, puis se déplace de cette solution pour en trouver une possiblement meilleure. Le nombre de fois que l'algorithme va se déplacer ainsi que la distance de déplacement depuis la position courante est fixé par l'utilisateur. Avec de mauvais choix de ces critères, l'algorithme peut rester coincé sur une solution locale qui n'est pas la solution générale.

Points positifs
- Plus précis que l'algorithme vorace
- Garantie une bonne solution
- Exécution rapide, même sur de gros exemplaires
- Très facile de reconstruire la solution

Points négatifs
- Pas optimal, peut rester bloquer sur un mauvais extremum local (rarement)
- Il faut trouver les bonnes constantes (T, α, P, k_{max}) et la bonne méta-heuristique

Voici les 20 premières solutions de nos trois algorithmes :

Vorace	Dynamique	Recuit	Meilleure solution
190	198	198	Recuit
198	198	198	Vorace
177	182	179	Dynamique
190	190	190	Vorace
193	198	193	Dynamique
195	200	195	Dynamique
183	194	189	Dynamique
186	194	194	Recuit
195	199	196	Dynamique
192	199	199	Recuit
291	296	291	Dynamique
267	288	286	Dynamique
286	293	287	Dynamique
257	273	273	Recuit
273	284	284	Recuit
285	286	286	Recuit
285	295	285	Dynamique
285	290	285	Dynamique
268	290	289	Dynamique
291	293	293	Recuit

FIGURE 9 – Les 20 premières solutions

Évidemment, l'algorithme dynamique apporte toujours la meilleure solution. Mais, pour des raisons de visibilité, lorsqu'un algorithme trouve la solution de programmation dynamique, nous l'affichons comme étant le « meilleur ». On peut ainsi mieux se rendre compte de l'efficience des deux algorithmes non optimal.

En regardant les résultats de nos programmes sur tous nos échantillons, nous avons élaboré un tableau comptant le nombre de fois où un des deux algorithme non optimal s'est avéré trouver une meilleure solution que l'autre. Comme on peut le constater, l'algorithme du recuit simulé l'emporte largement sur le vorace :

Vorace meilleur	Recuit meilleur	Solution égale
8	140	152

Il s'agit là du meilleur compromis lorsque la taille de l'échantillon devient trop importante. Il nous garantit une bonne solution qui, de plus, est améliorable en ajustant les variables de l'algorithme.

Conclusion

Dans ce TP nous avons pu mettre en application nos connaissance théoriques en implémentant trois algorithmes étudiés en cours. Le premier était une application directe de l'algorithme vorace (glouton), et était relativement simple à implémenter.

Le second algorithme (par programmation dynamique) s'est révélé bien plus complexe à développer que prévu et nous a posé de nombreuses difficultés. De plus, son temps d'exécution est fonction exponentielle de l'échantillon étudié, ce qui nous a bloqué pour son analyse empirique. Cependant, ses résultats nous ont permis de comparer la fiabilité des deux autres algorithmes. Nous avons ainsi remarqué qu'ils ne donnaient pas toujours le bon résultat, bien que leurs solutions soient proches du résultat.

Le dernier algorithme, par recuit simulé, nous a fait découvrir une nouvelle approche de programmation. Il a l'avantage d'être rapide et modulable et bien qu'il ne soit pas optimal, il constitue le meilleur compromis.