JOURDAN Leane, BAUSSAY Lucas
681C

Université de Nantes
2019/2020

RECHERCHE OPERATIONNELLE

Une méthode simple pour la résolution exacte du problème du sac à dos unidimensionnel en variables binaires

UNIVERSITÉ DE NANTES

X32I030
Licence Maths - Info 3

Table des matières

1 Introduction

On souhaite résoudre le problème du sac à dos unidimentionnel en variables binaires grâce à un programme conçu spécifiquement pour ce problème dans l'optique que sa résolution soit plus rapide et efficace que celle effectuée par un solveur général (exemple : GLPK).
Ainsi, nous ferons une première résolution avec GLPK qui nous servira de "base" pour comparer nos résultats et notre rapidité d'execution.
Nous implémenterons en julia 1.4.0 (avec les mises à jour de JuMP et GLPK) les deux algorithmes de programmations dynamique (version basique et moins basique), l'algorithme de prétraitement ainsi que la partie "pour aller plus loin" présentés dans le sujet.

Le code fourni étant dûment commenté il peut être intéressant de le lire en parallèle de ce rapport.
Il est également important de lire le $README.md$ avant de compiler.

2 Structures de données

2.1 Structure créée en Julia

— Tout d'abord il y a la structure Objet qui contient :

- profit : Correspond au profit de l'objet
- cout : Correspond au poids de l'objet
- id : Son identifiant unique qui est son indice dans la liste d'objets initiale
- ratio : Le ratio du profit de l'objet par rapport à son poids.

On a créé la structure Objet pour simplifier le code et pouvoir acceder librement à chacun de ses paramètres sans avoir à transmettre une liste pour chaque paramètre et trier toutes les listes à chaque fois.

— Puis, la structure Etat qui contient :

- pere : Un autre état qui est le prédécesseur de cet état (On expliquera ça dans les paragraphes suivants).
- prendPere : Un booléen qui vaut $vrai$ si on a prit l'objet correspondant à l'état précedent, $faux$ sinon.
- k : Comme vu dans le sujet, k correspond au numéro de la colonne dans le "graphe"
- c : Le profit cumulé depuis le début jusqu'à cet état
- d : Le poids cumulé depuis le début jusqu'à cet état

— Et il y a les autres structures créées par Anthony PRZYBYLSKI comme le Parser, etc ...

Ces structures de données sont disponibles dans le fichier : structKP.jl

2.2 Programmation dynamique

2.2.1 Programmation Dynamique Basique

Pour la programmation dynamique, on a opté pour une structure de données sous forme de tableau, c'est à dire un $Vector\{Vector\{Etat\}\}$.
Ainsi la grosse partie du calcul revient à créer cette structure que l'on appellera tableau dans la suite de l'explication.

Pour créer ce tableau on va tout d'abord l'initialiser avec un état vide : Il ne comporte pas de père (pere = $nothing$), ainsi prendPere = $nothing$ et k = 1, c = d = 0.

Une fois cet état initial créé, il suffit de créer la nouvelle colonne par rapport à la précedente d'après les formules données dans le Polycopié du projet.

2.2.2 Programmation dynamique moins basique

Dans cette partie nous créons nos 2 itérateurs *avec* et *sans* qui se chargeront, comme indiqué dans le sujet, de créer les différentes colonnes de notre tableau, triées par ordre croissant suivant les poids cumulés de chaque état.

Ainsi on simplifie le calcul pour vérifier la domination des états puisqu'on a simplement besoin de tester si notre état domine l'état juste au dessus de lui.
Cette implémentation est faite pour simplifier énormément les calculs. On verra la qualité des 2 implémentations dans la partie "Analyse des résultats".

3 Calcul des bornes supérieures

Les calculs de bornes sont fait dans le fichier *pretraitement*1.*jl*.

3.1 Bornes supérieures sur la valeur optimale

Nous nous intéressons désormais au calcul des bornes supérieures sur la valeur optimale.
On décide de ne pas utiliser la borne supérieure issue de la relaxation continue : $U_0 = \sum_{j+1}^{s-1} p_j + \left\lfloor \overline{\omega} \frac{p_s}{w_s} \right\rfloor$.

On commence donc par implémenter les fonctions permettant d'obtenir la borne de Martello et Toth :

- La fonction *calculBornesSup*() appliquant les formules données dans le sujet et retournant les valeurs de $U^0 = \sum_{j+1}^{s-1} p_j + \left\lfloor \omega \frac{p_{s+1}}{w_{s+1}} \right\rfloor$ et de $U^1 = \sum_{j=1}^{s-1} p_j + \left\lfloor p_s - (w_s - \omega) \frac{p_{s-1}}{w_{s-1}} \right\rfloor$.
- La fonction $UB(U^0, U^1)$ retournant $U_1 = max\{U^0, U^1\}$, la borne supérieure de Marthello et Toth.

3.2 Utilisation des bornes pour prétraiter le problème

On détermine une première solution admissible pour le problème du sac à dos en utilisant l'algorithme glouton, elle nous fournit une première borne inférieure LB sur la valeur optimale.
On considère les $2n$ sous-problèmes fixant chacun une variable x_i soit à 0, soit à 1 pour $i \in \{1, \ldots, n\}$.
On souhaite désormais calculer la borne supérieure de Martello et Toth sur la valeur optimale de chaque sous-problème.
On commence par calculer la borne de Martello et Toth UB sur la valeur optimale du problème de base.
On garde en mémoire la valeur de la borne inférieure (issue de l'application de l'algorithme glouton), les caractéristiques de la solution *dantzig* (sa valeur et la place qu'elle occupe dans le sac) ainsi que l'indice de l'objet cassé.
Cela permet de les fournir à chaque sous-problème pour pouvoir les utiliser directement dans les formules et ainsi d'éviter de les recalculer à chaque fois et donc de grandement réduire le nombre de calculs à effectuer.

3.2.1 UB0

Soit $UB0(i)$ la borne supérieure obtenue pour le sous problème fixant la variable x_i à 0.

Pour calculer $UB0(i)$ on utilise la méthode donnée dans le sujet en appliquant les formules sur les paramètres pré-calculés(valeur de la solution Dantzig...).
La fonction calcule si besoin les valeurs de la nouvelle borne U^0 et/ou de la nouvelle borne U^1 et renvoie leur maximum.

3.2.2 UB1

Soit $UB1(i)$ la borne supérieure obtenue pour le sous problème fixant la variable x_i à 1.

Voici la méthode que nous avons trouvé pour calculer le plus efficacement possible les bornes $UB1(i)$:

Soit s l'indice de l'objet cassé.

Il y a 5 cas à considérer :

— Si $i <= s-2$,
l'objet qu'on force à être dans le sac fait forcément partie de ceux composant la solution *dantzig*. Cela signifie que l'objet est déjà présent dans le sac. Donc, il n'y a pas de changement pour la solution *dantzig* et l'objet cassé. De plus les variables d'indice inférieur ou égal à $s+2$ n'impacte pas le calcul de U^0 et U^1. La borne supérieure de Martello et Toth reste la même, $UB1(i) = UB$, on renvoie donc la valeur UB qui était passée en paramètre.

— Si $i = s-1$,
l'objet à l'indice $s-1$ fait partie de la solution *dantzig*, cela signifie qu'il se trouve déjà à l'intérieur du sac. La solution *dantzig* n'est pas impactée, pas plus que le terme U^0 car l'objet situé à l'indice $s-1$ n'intervient pas dans la deuxième partie du calcul de sa valeur.
Toutefois, le terme U^1 est impacté, en effet lors du calcul de sa valeur, on doit retirer un fragment de l'objet d'indice $s-1$ afin que l'objet cassé d'indice s puisse être ajouté dans le sac. Mais ici on veut que l'objet d'indice $s-1$ soit ajouter dans le sac dans son intégralité. Il faut donc lui trouver un remplacant pour pouvoir mettre l'objet cassé dans le sac. On le remplace donc par l'objet précedent, c'est à dire l'objet d'indice $s-2$ car il s'agit de l'objet déjà dans le sac ayant le ratio le plus faible dans ce sous-problème.
On a alors : $U^1 = \sum_{j=1}^{s-1} p_j + \left\lfloor p_s - (w_s - \overline{\omega}) \frac{p_{s-2}}{w_{s-2}} \right\rfloor$

— Si $i = s$,
on force l'objet cassé à être dans le sac.
Dans le calcul du terme U^1, l'objet cassé se trouve déjà à l'intérieur du sac donc U^1 est inchangé.
Cela n'est toutefois pas le cas pour le terme U^0, sa valeur va donc devoir être modifiée.
On veut trouver une façon de pouvoir calculer sa nouvelle valeur sans avoir à recalculer entièrement la solution *dantzig* sur un sac dont la capacité serais $poidsMax - poidsObjetCasse$.
Pour cela on définie dans notre code une nouvelle fonction appelée $calculRetrograde()$.
Le but de cette fonction est de trouver une nouvelle "solution *dantzig*" dans le cas où on veut forcer un objet en plus à être dans le sac si cet objet ne fait pas partie de la solution *dantzig* initiale.
On commence donc par "ajouter" cet objet dans le sac. La capacité étant dépassé on va chercher à retirer des objets jusqu'à ce que la somme de leur poids soit inférieure ou égale à la capacité ω du sac (si le nouvel objet peut directement être mis dans le sac on entre pas dans la boucle).
(On considèrera ici que le poids de l'objet forcé est inférieur à ω sinon le problème n'aurait pas d'intêret et on aurait un sac vide.)
On doit cependant prendre en compte le fait qu'on veut avoir le meilleur profit possible et que l'objet forcé lui doit rester à l'intérieur du sac. On s'intéresse donc à l'ensemble des objets formant la solution *dantzig* initiale, étant triés par ratio on va donc pouvoir les enlever un par un en commencant par celui ayant le ratio le plus faible jusqu'à ce que le poids cumulé soit inférieur ou égal à ω. On en tire l'indice du nouvel objet cassé, qu'on notera s^*. La fonction retourne l'indice du nouvel objet cassé et les caractéristiques de la nouvelle "solution *dantzig*" associée.
Grâce à ces données on peut calculer la nouvelle valeur de U^0 :
$U^0 = \sum_{j=1}^{s^*-1} p_j + p_i + \left\lfloor \overline{\omega} \frac{p_{s^*+1}}{w_{s^*+1}} \right\rfloor$ dans la fonction $calculRetrogradeU0()$.

— Si $i = s+1$,
on force l'objet d'indice $s+1$ à être dans le sac.
Or cet objet ne fait pas partie de la solution *dantzig*, cela signifie qu'il ne s'y trouve pas déjà et doit être rajouté. Cet objet est impliqué dans le calcul du terme U^0, on cherche à savoir si le forcer à être dans le sac impactera la valeur de U^0.
On distingue alors 2 sous cas :

  — Si $\frac{\omega}{w_{s=1}} >= 1$,
  cela signifie que le "fragment" de l'objet d'indice $s+1$ qu'on a mit dans le sac est plus grand que 1, donc

l'objet se trouve déjà au moins une fois dans son intégralité à l'intérieur du sac. On a donc pas besoin de le rajouter, c'est à dire que le terme U^0 n'est pas modifié.

— Si $\frac{\omega}{w_{s=1}} < 1$,
cela signifie que le "fragment" de l'objet d'indice $s+1$ qu'on a mit dans le sac est strictement plus petit que 1. On utilise donc les fonctions *calculRetrograde*() et *calculRetrogradeU*0() expliquées ci-dessus pour l'ajouter au sac, trouver le nouvel objet cassé et calculer la nouvelle valeur de U^0.

En ce qui concerne U^1, l'objet d'indice $s+1$ n'intervient pas dans la formule du calcul de ce terme. L'objet n'étant pas déjà dans le sac il faut donc le forcer à y être et modifier U^1 en conséquence. Grâce à *calculRetrograde*() on récupère l'indice du nouvel objet cassé s^* et les caractéristiques de la nouvelle "solution *dantzig*" associée.
Grâce à ces données on peut calculer la nouvelle valeur de U^1 :
$U^1 = \sum j = 1^{s^*-1} p_j + p_i + \left\lfloor \overline{\omega} \frac{p_{s^*+1}}{w_{s^*+1}} \right\rfloor$ dans la fonction *calculRetrogradeU*1().

— Si $i > s+1$,
l'objet à ajouter ne fait donc pas partie de la solution *dantzig* initiale. Il n'intervient pas non plus dans le calcul des bornes U^0 et U^1. En forçant l'objet à être dans le sac on a donc besoin d'effectuer un calcul retrograde des bornes U^0 et U^1 respectivement à l'aide des fonctions *calculRetrogradeU*0() et *calculRetrogradeU*1() faisant appel à la fonction *calculRetrograde*() (expliquées toutes trois ci-dessus).

En pratique et pour simplifier le code on fusionne le cas 4.2 avec le cas 5. En effet, dans les deux cas on doit effectuer les même actions, c'est à dire un calcul retrograde pour les deux bornes U^0 et U^1.

4 Difficultés rencontrées

— Les fonctions *calculLb*0() et *calculLb*1() appelant *BorneInfAlgoGlouton*() à chaque nouvel appel pour recalculer entièrement la solution gloutonne du sous-problème associé à x_i, on a cherché à optimiser au mieux ce calcul. Ainsi en se basant sur l'idée du calcul efficace des bornes $UB1(i)$ et $UB0(i)$ nous avons tenté de créer les fonctions *calculLB*1*bis*() et *calculLB*0*bis*(). Le but étant de réfléchir à comment avec une disjonction de cas on pouvait pour chaque i renvoyer la valeur $LB0$ ou $LB1$ uniquement en modifiant les caractéristiques de la borne inférieure ou de la solution *dantzig* fournie et non en recalculant tout.

Une fois les fonctions finies, nous avons voulu les tester sur quelques instances pour comparer leur résultats avec ceux de *calculLb*0() et *calculLb*1(). Pour l' instance $KP50-1.dat$ aucun problème, les deux versions s'exécutent rapidement et ont les mêmes résultats. Seulement pour les instances à 100 variables ou plus la compléxité devient très mauvaise les fonctions *calculLB*1*bis*() et *calculLB*0*bis*() prennent plusieurs minutes à s'exécuter avec 100 objets, et quand à l'exécution avec 500 objets elle était si longue que nous avons fini par choisir de l'interrompre.
Nous avons toutefois laissé ces fonctions dans notre code et mis en commentaires les parties les concernant.

— Nous avons aussi eu des problèmes avec la macro @timed. En effet, lors du codage du fichier pour faire l'affichage graphique des courbes temporelles, la macro @timed calcule le temps de compilation de tout le programme, ainsi toutes les courbes avaient les mêmes allures, et nous avons mis un peu plus d'une journée à surmonter ce problème.

5 Limites du programme et améliorations possible

5.1 Limites

— Certaines fonctions ont beaucoup de paramètres (les différentes fonctions de calcul retrograde en ont 6,*calculUB*0() en a 8,*calculUB*1() en a 9...) ce qui peut rendre leur signature un peu illisible. De plus leur paramètres ont tendances à être associés les un aux autres et à être les mêmes d'une fonction à l'autre.
Par exemple on a souvent besoin de passer en paramètre à ces fonctions les caractéristiques du solution *dantzig* ou d'une Borne Inf issue de l'algorithme glouton. Une solution pourrait être de créer une nouvelle structure

de données associée à ces solution (avec l'indice de l'objet cassé, la somme $\sum_{j=1}^{s-1} p_j$ et la somme $\sum_{j=1}^{s-1} w_j$ pour une solution *dantzig* par exemple).

— Une autre limite que l'on peut trouver à notre implémentation est qu'elle n'arrive pas à égaler JuMP.
En effet d'après les graphiques disponibles dans la partie : Analyse des résultats, JuMP s'effectue instantanément contrairement à notre meilleure implémentation (de l'ordre de 0.008 sec pour une instance à 500 objets). Cela paraît négligeable mais pour de plus grosses instances cette différence pourrait être certainement plus remarquable.
Malgré tout, on constate que JuMP, et notamment les modules qu'elle importe (comme MOI) met un temps assez long (∼1 sec) pour effectuer une première compilation, contrairement à nos implémentations.

5.2 Améliorations possibles du programme

Il serait possible d'apporter des améliorations dans le calcul de $UB0$ et $UB1$ qui impacteraient ainsi la partie "Pour aller plus loin", mais qui nécessiteraient de refaire entièrement les calculs de $UB0$ et $UB1$ ce qui est impossible par manque de temps.
Cette amélioration a pour but ne plus calculer tout les $UB0(i)$ et tout les $UB1(i)$ dans une boucle au préalable, pour les stocker dans des listes, puis d'effectuer le prétraitement dessus ensuite dans une seconde boucle(ce qui est actuellement le cas). L'idée serait plutôt qu'après chaque calcul de $UB0(i)$ et $UB1(i)$ on fixe cette variable à 0 ou à 1 si c'est possible. Ceci nous permettrait ainsi d'en tenir compte dans les calculs des $UB0$ et $UB1$ suivants et ainsi de pouvoir affiner nos résultats quand aux variables fixées à 1 et celles à 0 au fur et à mesure.

6 Pour aller plus loin : Utilisation des bornes dans la programmation dynamique

Utiliser les bornes supérieures et inférieures, non pas dans le prétraitement, mais lors de la programmation dynamique consiste en une fusion de la partie programmation dynamique moins basique et du prétraitement.

Dans un premier temps, lors de l'appel au programme *pourAllerPlusLoin.jl* il nous faut, comme pour le prétraitement, travailler sur une liste d'objets, non pas quelconque mais qui soit triée par ordre des ration $profit/poids$ décroissant. Ensuite nous aurons aussi besoin des différentes bornes du problème (plus particulièrement de la borne inférieure pour la comparaison et de la borne supérieure pour les différents calculs), on fait donc appel aux fonctions de *pretraitement*1.*jl* pour les calculer.

Ensuite les calculs suivants reviennent exactement à faire l'algorithme de programmation dynamique moins basique : On initialise nos deux itérateurs *avec* et *sans* puis un par un on détermine qui de l'un ou de l'autre avance. Si on applique l'itérateur *sans*, on va calculer la borne superieur du sous problème : l'objet associé à l'état sur lequel pointe l'itérateur *sans* n'est pas mis dans le sac (est mis à 0), cela revient donc à calculer $UB0$. De même, pour l'itérateur *avec* on calcule $UB1$.

Ainsi, lors du filtrage des résultats par le test de domination par l'état précédent on rajoute un test identique à celui du prétraitement c'est à dire qu'on identifie les états pour lesquels la borne superieur calculée (que ce soit $UB0$ ou $UB1$) est inférieure à la borne inférieure du problème calculée au tout début. Si on se retrouve dans ce cas alors l'état peut être filtré puisque quoi qu'il arrive la borne supérieure du profit obtenable depuis cet état sera inférieure au résultat attendu.

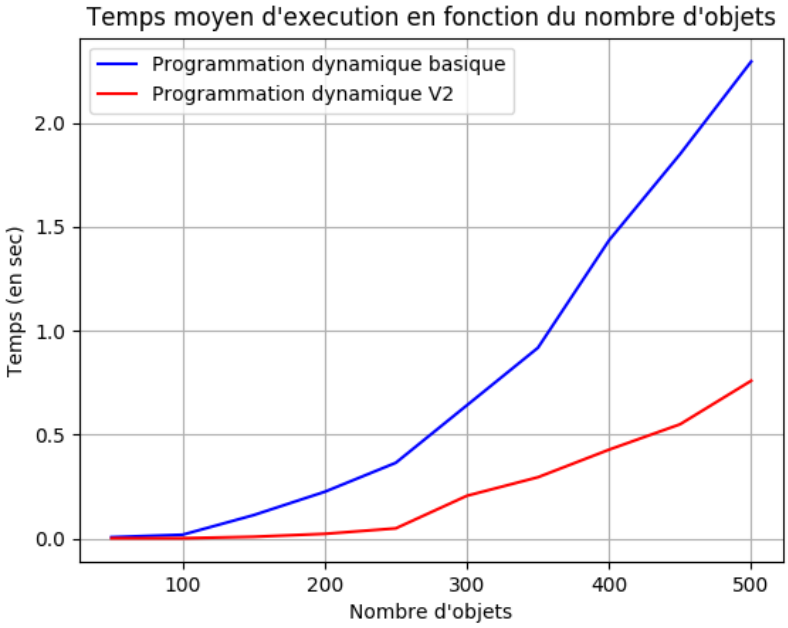
Ainsi dans cette partie, on effectue une programmation dynamique qui filtre beaucoup plus d'états que les précédentes et qui ainsi permet d'avoir une bien meilleure efficacité par rapport aux autres, ce qu'on pourra voir dans la prochaine partie : l'analyse des résultats.

7 Analyse des résultats

Une fois les différentes méthodes de résolution implémentées, on cherche à les comparer les unes aux autres sur l'ensemble des instances fournies afin de se faire une meilleure idée de leur efficacité à la fois temporelle et spaciale.

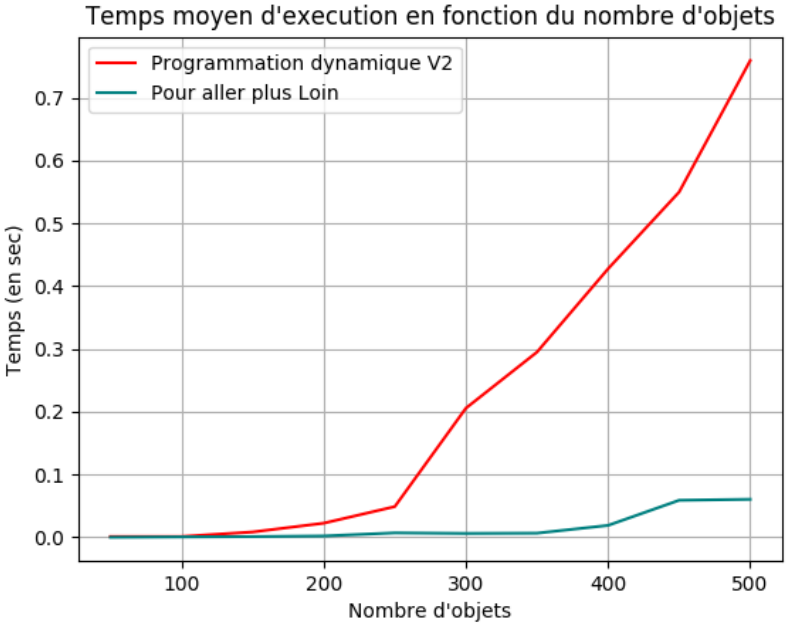
7.1 Temps de résolution

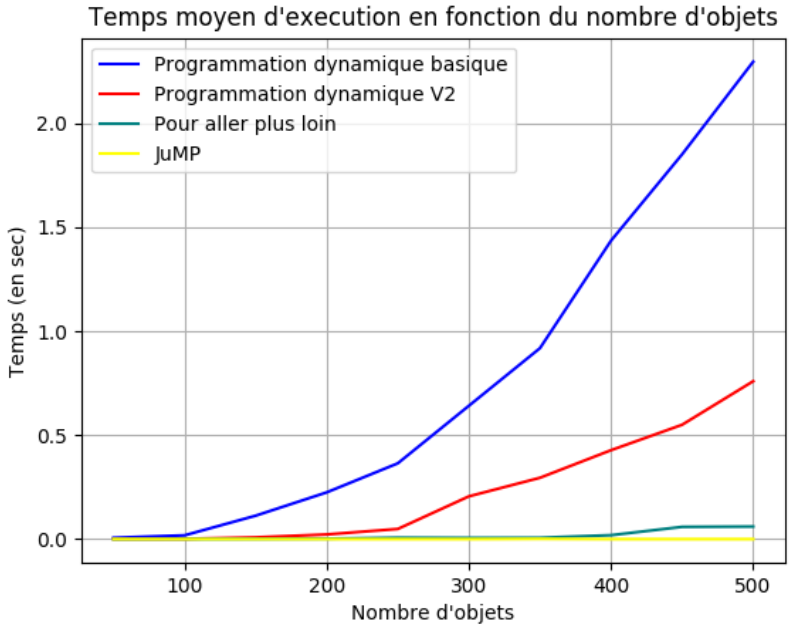
On commence par s'intéresser à la rapidité avec laquelle elles effectuent la résolution sur l'ensemble des instances sans prétraitement. Pour cela on utilisera la macro @timed ainsi que le package PyPlot pour tracer les différents graphs. On calcule pour chaque taille d'instance (50,100,150..) une moyenne des résultats obtenus sur les 10 problèmes.



On commence par les deux versions de programmation dynamique. Sur les petites instance (50 ou 100 objets) elles sont presque équivalentes avec une résolution quasiment instantanée. Cependant, une fois les 500 objets atteints, la version basique (en bleu) mettra en moyenne plus de 2sec à effectuer la résolution sur l'instance tandis que la version moins basique (en rouge) reste sous la barre de 1sec. Cela vient du fait que la 2ème version génère beaucoup moins d'états et ne les reparcourt pas à chaque colonne créée pour verifier leur dominance mais le fait au fur et à mesure.

On compare cette fois la programmation dynamique moins basique (toujours en rouge) à la partie "Pour aller plus loin" (en bleu canard). Comme précédement les deux versions sont comparables et presque instantanées sur de petites instances mais une fois arrivé à 500 objets, la partie "Pour aller plus loin" est plus de 7 fois plus rapide que la version moins basique de programmation dynamique. Cela s'explique par le fait que cette nouvelle version filtre les mêmes états que la deuxième version de programmation dynamique mais en filtre également d'autres ce qui diminue fortement le nombres d'états stockés et générés et donc le temps de calcul.(voir partie suivante : 7.2 Complexité spatiale)

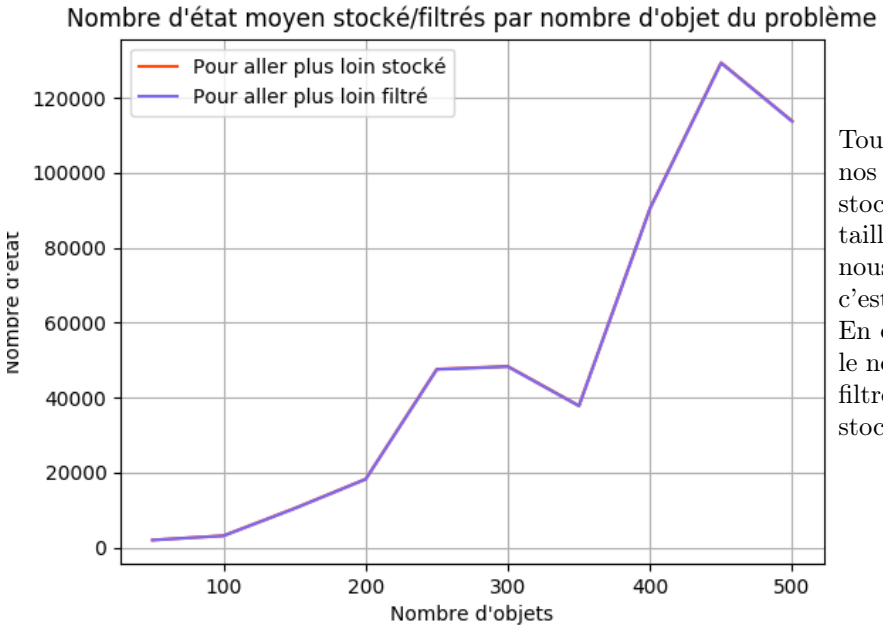




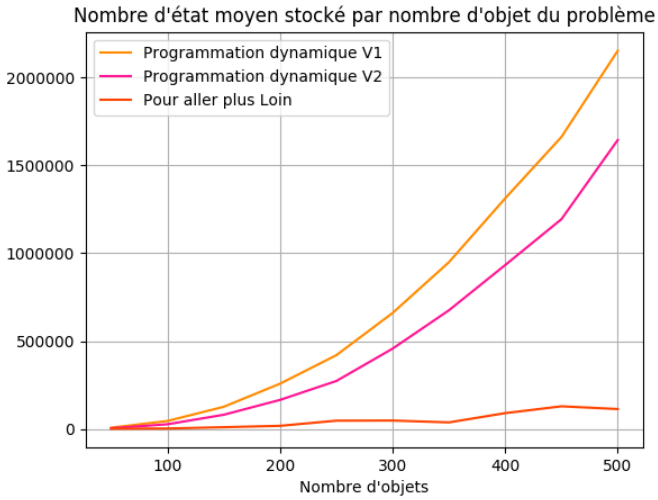
On mesure cette fois ci nos différentes implémentations à JuMP et GLPK. Bien qu'au fur et à mesure des implémentations la courbe de temps se rapproche de celle de GLPK et malgré une nette amélioration depuis la programmation dynamique basique, notre meilleure implémentation est encore bien plus lente et la résolution par GLPK d'une instance de 500 objets est quasiment instantanée.

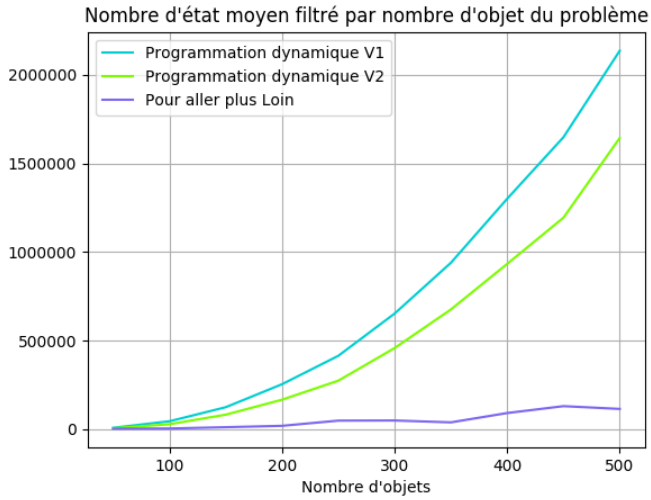
7.2 Complexité spatiale

On s'intéresse cette fois ci au nombre d'états générés que chaque version va stocker en mémoire ou filtrer. Pour les compter, on rajoute deux compteurs dans chacune des versions (qu'on supprime par la suite), on fait exécuter le programme sur l'ensemble des instance et on récupère ces informations dans un fichier .txt (fichier : *testNbEtat.jl*).



Tout d'abord on a souhaité comparer pour nos trois versions le nombre moyen d'états stockés au nombre moyen d'état filtrés par taille d'instance. Seulement dans les trois cas nous avons obtenu un graph comme celui-ci, c'est à dire des courbes qui se superposent. En effet, à chaque fois, et bien que différents, le nombre d'états stockés et le nombre d'états filtrés sont très proches. En moyenne on stocke donc un état généré sur deux.

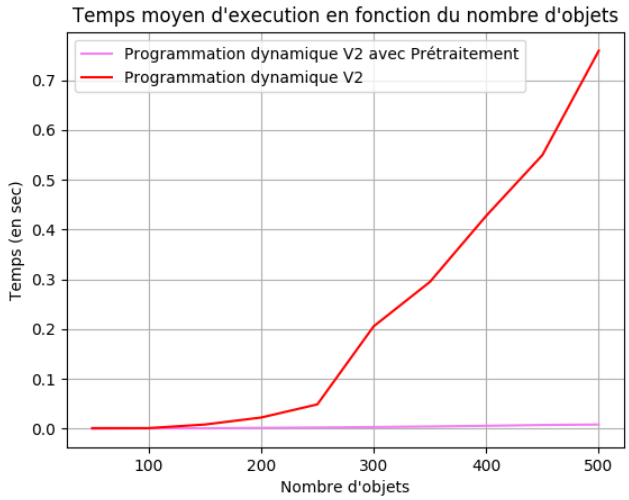


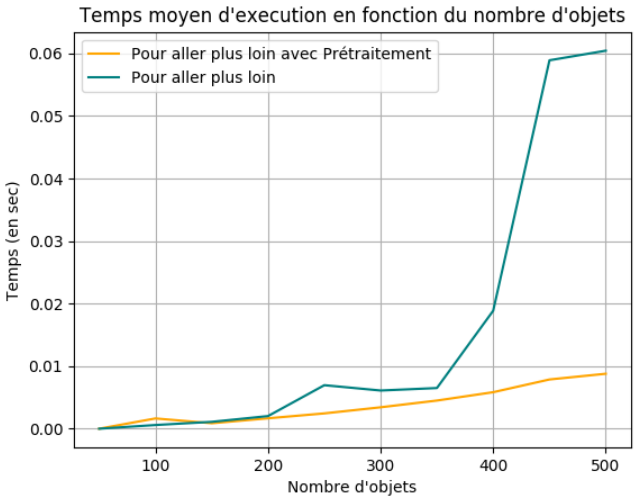


On souhaite désormais comparer nos trois versions entre elles, tout d'abord sur le nombre d'états stockés puis sur le nombre d'états filtrés (même si cela revient au même). On voit bien ici que plus un programme est rapide moins il stocke et filtre des états.
Théoriquement, si on ne filtrait aucun état il y aurait 2^{n+1} états générés. Pour une instance de 500 objets cela reviendrait à générer 2^{500} états. Hors, ici on en génère environ 4,5 millions dans la version basique de programmation dynamique, 3,2 millions pour la version moins basique et seulement 200 000 pour la partie "Pour aller plus loin". Donc, plus un programme filtre les états, moins il en génère de nouveaux et plus il sera rapide.

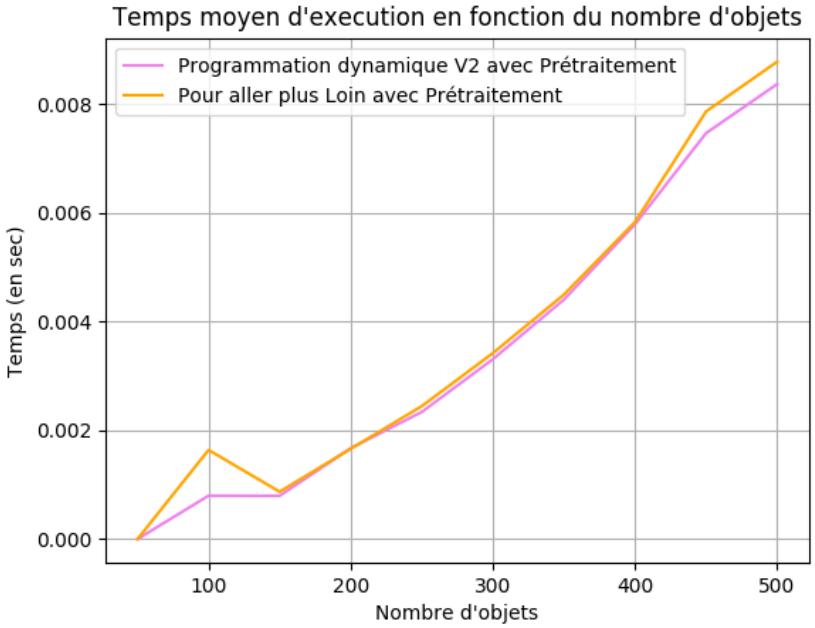
7.3 Efficacité du prétraitement

Pour finir, après avoir implémenté le prétraitement on veut savoir à quel point il est efficace et rend la résolution plus rapide. Mais surtout, si une fois ajouté à notre meilleure méthode de résolution, il nous permet d'égaler JuMP et GLPK. Pour cela on réutilise la même méthode de mesure que dans la partie 7.1 Temps de résolution.

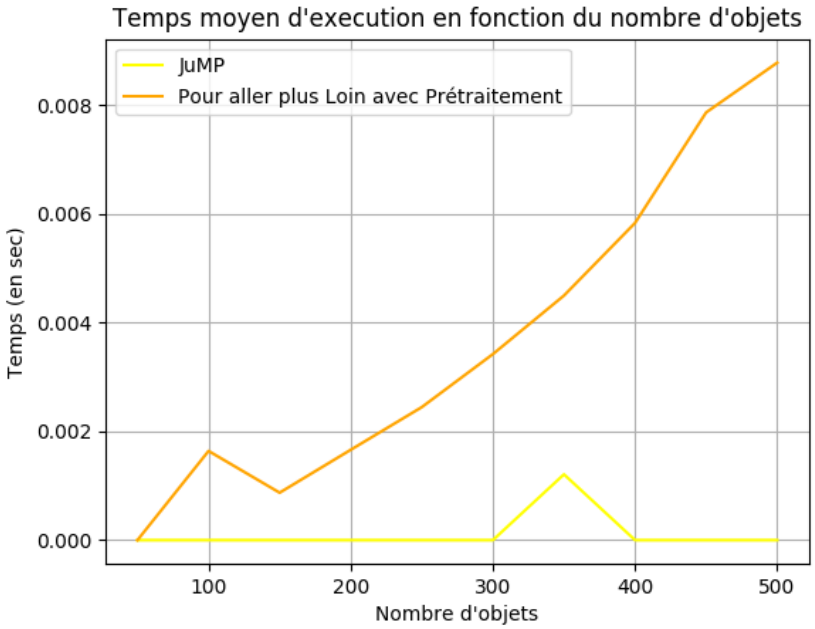




Ci-dessus, on compare pour la version moins basique de programmation dynamique (à gauche) et pour la partie "Pour aller plus loin" (à droite) la vitesse moyenne d'exécution pour chaque taille d'instance sans prétraitement (respectivement en rouge et en turquoise) et avec prétraitement (respectivement en rose et en orange). La différence est ici considérable, pour une instance de 500 objets le temps est divisé par presque 75 pour la programmation dynamique et par plus de 6 pour la version améliorée.



En effet, une fois qu'on compare les courbes des deux versions avec prétraitement on se rend compte qu'il élimine tellement de variables du problème, et en laisse si peu à traiter, que la différence entre les deux versions, auparavant très importante, devient négligeable. Il serait ici intéressant d'appliquer cela à des instances plus grandes (en milliers d'objets) pour voir l'évolution des courbes.



Avec un prétraitement aussi efficace on se demande donc si on a (enfin) réussi à égaler JuMP/GLPK (en jaune). On décide donc de le comparer à la partie "Pour aller plus loin" avec prétraitement (en orange). Seulement, il semble que ça ne soit toujours pas le cas, JuMP/GLPK étant instantané pour presque toutes les instances (même avec 500 objets).