Simon Modeste
simon.modeste@umontpellier.fr

Les vitres – Clôture
Quelques éléments de résolution

Nous proposons ici une (ou plusieurs) résolution(s) possible(s) du problème des vitres. Ce texte s'adresse directement aux enseignants (mais certaines parties peuvent être lues par des élèves) pour leur donner des éléments de résolution possible et les accompagner dans l'organisation de la clôture dans leur classe de la session de résolution collaborative de problème.

Ce document est une première version, qui, malgré les relectures peut encore contenir quelques erreurs. Si vous en rencontrez, n'hésitez pas à les signaler à simon.modeste@umontpellier.fr.

1. Le problème suite à la modélisation

Le problème proposé à l'issu de la fiction relancée est le suivant :

> Trouver une méthode pour minimiser la surface totale des chutes selon les commandes chaque semaine.

Et les éléments du problème que nous penons en compte sont les suivants :

- On découpe des grandes plaques rectangulaires de dimension 600 cm x 320 cm (parallèlement à leurs bords, et l'épaisseur de la découpe est négligeable)
- pour produire des vitres rectangulaires de 4 dimensions différentes :
  (a) 210 cm x 215 cm, (b) 100 cm x 215 cm, (c) 100 x 125 cm, (d) 60 x 215 cm.
- Les morceaux restants après découpe sont considérés comme perdus, et on souhaite minimiser leur surface totale.

Autrement dit, pour une commande donnée (c'est-à-dire une ensemble de rectangles de dimensions (a), (b), (c) et (d)), il faut trouver une disposition de ces rectangles dans des grands rectangles de dimensions 600 x 320, de façon à minimiser la surface non occupée par les rectangles de la commande.

Remarque 1 : Pour une commande donnée, la surface totale des rectangles de la commande peut se calculer facilement (somme des surfaces de tous les rectangles). Si l'on utilise n plaques pour les disposer, la perte sera égale à la surface totale des plaques utilisées à laquelle on soustrait la surface totale des rectangles de la commande. Autrement dit, on peut se contenter de chercher à disposer entièrement une commande sur des grandes plaques, en essayant d'utiliser le moins possible de grandes plaques.

Notation : Nous noterons une commande par son nombre de plaque de chaque dimension.
Par exemple : 4 (a), 2 (b), 5 (c), 6 (d).

Pour être sûr qu'un certain ensemble de rectangles peut être disposé dans un grand rectangle, il faut réaliser une disposition de ces rectangles et se convaincre que les rectangles ne débordent pas du grand

rectangles et ne se superposent pas entre eux. On peut s'appuyer sur des figures représentant les dispositions de rectangles, mais parfois quelques calculs sont nécessaires pour être sûr qu'il n'y a pas de problème.

Pour aider la recherche, on peut découper des rectangles des quatre dimensions possible et les disposer sur un support de la taille du grand rectangle (pour cela, on peut changer d'échelle, ça ne change pas le problème).

Pour étudier notre problème, on peut commencer par s'intéresser à différentes commandes puis voir comment on peut traiter de manière plus générale diverses commandes.

2. Quelques exemples et premiers résultats généraux

Exemple 1. Supposons que l'on ait la commande suivante : 1 (a), 4 (b), 3 (c), 2 (d). On peut essayer de chercher par tâtonnement une disposition des ces rectangles dans un grand rectangle. En n'y parvenant pas, on peut se demander si cela est possible.

On peut calculer la surface totale de la commande. On peut calculer l'aire de chaque type de rectangle et du grand rectangle :

	Rectangle (a)	Rectangle (b)	Rectangle (c)	Rectangle (d)	Grand rectangle
Largeur (m)	2,1	1	1	0,6	3,2
Longueur (m)	2,15	2,15	1,25	2,15	6
Aire (m^2)	4,515	2,15	1,25	1,29	19,2

L'aire totale de la commande est donc 1 x 4,515 + 4 x 2,15 + 3 x 1,25 + 2 x 1,29 = 19,445.

Cela excède l'aire d'un grand rectangle, il faut donc au moins deux plaques pour cette commande. On peut facilement trouver une disposition sur deux grands rectangles :

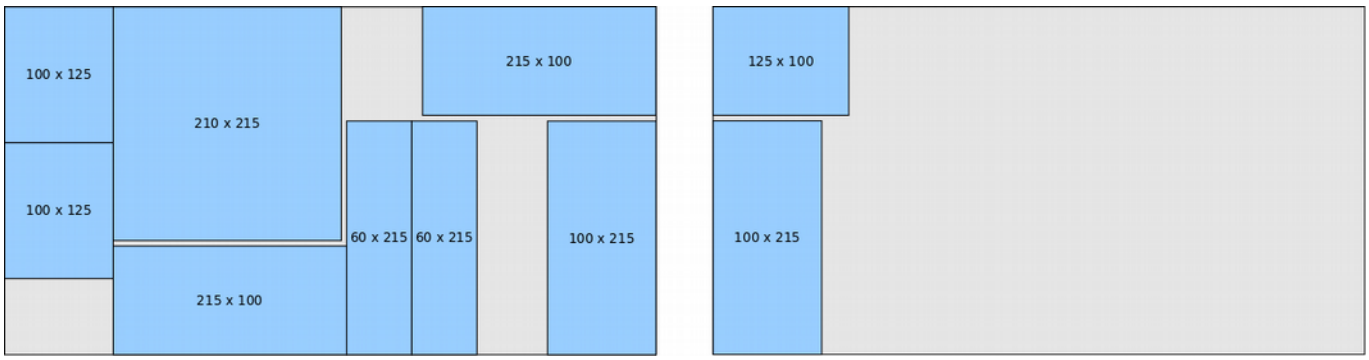

Figure 1. Une disposition pour l'exemple 1.

On peut calculer la perte liée à cette disposition : 18,96 m^2. On sait qu'on ne peut malheureusement pas faire mieux pour cette commande.

Remarque 2 : On peut se dire que cela n'est pas très optimal pour l'entreprise, et revenir sur la modélisation du problème (et la gestion par l'entreprise). On pourrait alors supposer qu'on découpe les premières plaques les plus remplies (ici la première plaque), et qu'on attend la commande de la semaine suivante pour compléter la dernière plaque et éviter de trop grandes pertes.
On pourrait alors considérer que les pertes sont celles de premières plaques seulement (ici juste la première, c'est-à-dire 3,155 m^2) et que les pertes de la dernière plaques se détermineront la semaine suivante.
Ceci dit, cela ne change pas beaucoup le problème mathématique à étudier, car pour minimiser les

pertes, il faudra toujours essayer d'utiliser le moins possible de plaques compte-tenu d'une certaine commande (même si cette commande comprend des vitres de la semaine précédente, et même si on ne découpe pas tout de suite la dernière grande plaque nécessaire).

Ce premier exemple fait aussi apparaître un premier résultat général, qui donne une condition nécessaire pour qu'une commande puisse être disposée sur un certain nombre de grandes plaques.

Théorème 1 (condition nécessaire) : La surface totale d'une commande doit toujours être inférieure que la surface totale des plaques utilisées.
Autrement dit, on peut trouver un minorant du nombre de grandes plaques nécessaire pour une commande (c'est-à-dire le nombre de plaques minimal qu'il faut envisager pour la commande). Ce nombre de plaques se détermine en divisant l'aire totale de la commande par l'aire d'une grande plaque (19,2 m²). On peut ensuite déterminer la plus petite perte qu'on peut espérer avoir, en soustrayant l'aire totale de la commande à l'aire totale associé au minorant du nombre de plaque calculé auparavant.

Comme on peut s'en douter, nous verrons juste après que l'on arrive pas toujours à trouver une disposition des rectangles de la commande dans le nombre de grands rectangles donné par le théorème 1 (autrement dit, ce minorant n'est pas toujours atteint).

Avant cela, pour s'éviter des calculs fastidieux, on anticipe les calculs d'aires et de pertes pour une commande donnée et un nombre de grandes plaques utilisées, qui seront toujours les mêmes, en s'aidant d'un tableur :

	Vitre 1	Vitre 2	Vitre 3	Vitre 4
largeur (cm)	210	100	100	60
longueur (cm)	215	215	125	215
aire (m²)	**4,515**	**2,15**	**1,25**	**1,29**

Grande plaque
320
600
19,2

					Total
Quantités	1	3	2	2	8
aire (m²)	4,515	6,45	2,5	2,58	**16,045**

Minoration simple	
Nombre min grandes plaques	**1**
Reste minimal	**3,155**

Dans une solution obtenue	
Nb grandes plaques	1
Reste obtenu	**3,155**

Figure 2. Extrait de la page de tableur.

Ainsi, pour une commande donnée (ici 1 (a), 3 (b), 2 (c), 2 (d)), on obtient alors directement l'aire totale de la commande, le minorant du nombre de plaques du théorème 1 ainsi que le reste si ce minorant est atteint (*Nombre min grandes plaques* et *Reste minimal*). Si l'on a trouvé une solution utilisant un certain nombre de plaques (*Nb grandes plaques*), on peut aussi directement calculer la perte associée à la commande (*Reste obtenu*).

Exemple 2. Supposons que l'on ait la commande suivante : 2 (a), 3 (b), 2 (c), 0 (d). L'aire totale de la commande est de 17,98 m², et si on arrive à trouver une configuration des rectangles de la commande qui utilise une seule plaque la perte sera de 1,22 m² (merci le tableur!)

Pourtant, on n'arrive pas à trouver une telle configuration ! Il n'est pas facile de prouver qu'on ne peut pas faire entrer cette commande dans une seule plaque dans ce cas particulier (vous pouvez essayer). Quoi qu'il en soit, on finit par se convaincre que deux plaques seront nécessaires, et on trouve facilement une disposition (figure 3, ci-dessous).

La perte serait alors de 20,42 m². Si l'on tient compte de la remarque 2, la configuration de la figure 3 sera sûrement la plus intéressante puisqu'en ne mettant qu'un rectangle de la plus petite aire possible dans le dernière plaque, elle laisse la plus grande aire disponible dans cette dernière plaque.

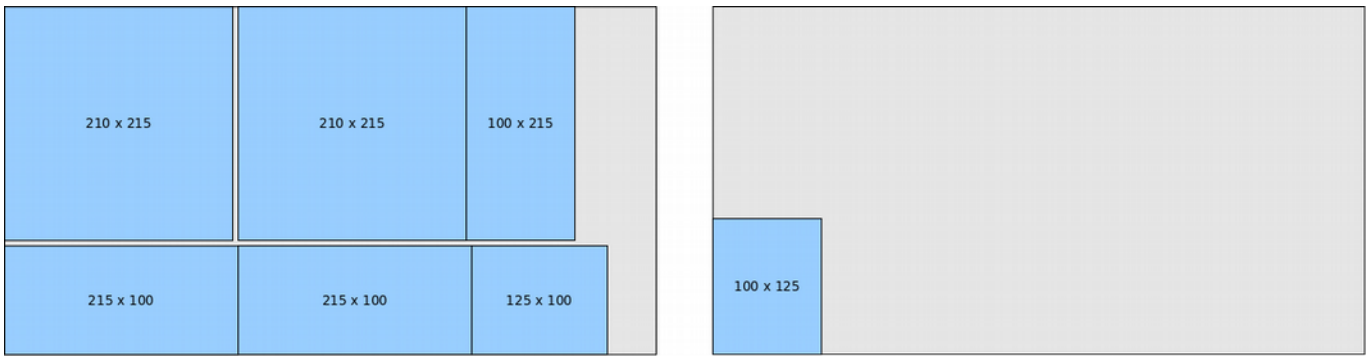

Figure 3. Une disposition pour l'exemple 2

Exemple 3. Étudions la commande suivante : 0 (a), 6 (b), 2 (c), 2 (d).

Le calcul des aires nous dit que la commande a une aire totale de 17,98 m². On peut espérer la faire entrer dans une seule plaque. Mais, ça n'est pas si simple, prenez le temps d'essayer !

Finalement, vous tomberez peut-être une de ces deux configurations, pas évidentes à trouver :

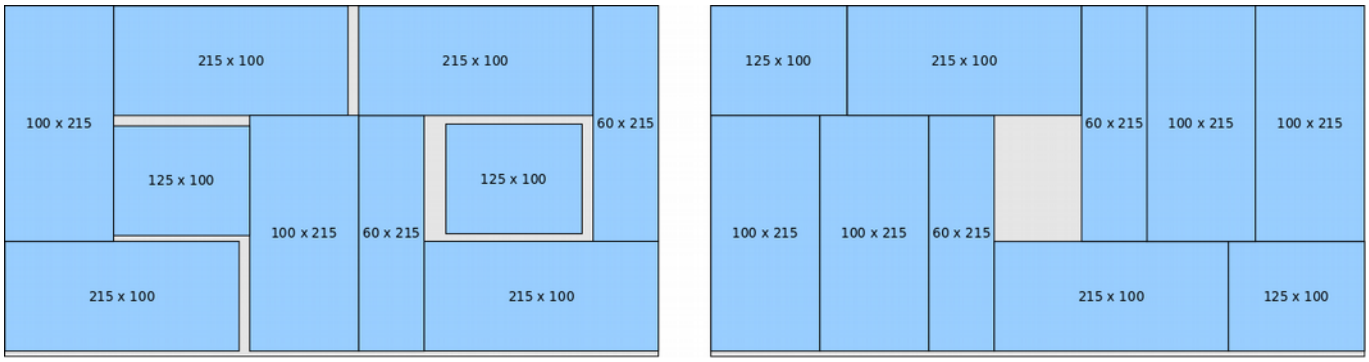

Figure 4. Deux dispositions pour l'exemple 3 (la perte est de 1,22 m²).

On voit bien à quel point il est difficile de savoir si une commande donnée peut ou non être découpée dans une seule plaque, lorsque l'on compare les exemple 2 et 3.

Remarque 3 : En observant les deux dispositions de l'exemple 3 (figure 4), en revenant au problème d'origine de découpe de vitrage, on peut se demander si ces configurations sont acceptables en réalité. En effet, on ne peut pas trouver d'ordre de découpe tel que chaque découpage se fasse d'un bord à l'autre (ou, autrement dit, en cassant à chaque étape de découpe la vitre suivant une ligne tracée – au diamant par exemple). On pourrait, revenir sur le choix de modélisation du problème, se dire que de telles dispositions ne sont pas acceptables et demander à ce que les découpes puissent se faire dans un certain ordre, toujours de « bord à bord ».

3. Des commandes simples

Il peut sembler difficile d'envisager directement une méthode pour disposer les rectangles de façon optimale, quelle que soit la commande donnée. Pour avancer dans la résolution du problème, on peut commencer par se poser la question de cas où les commandes sont simples, par exemple composées d'une seule dimension de vitre. La question de minimiser les pertes revient alors à chercher à maximiser le nombre de vitres d'un même type dans une grande plaque. Regardons de plus près le cas des quatre formats de vitre.

Commande de vitres au format (a)

Combien au maximum peut-on disposer de vitres (a) dans un grand rectangle ? En les disposant verticalement ou horizontalement, on a toujours la même chose, au maximum 2 sur une grande plaque.

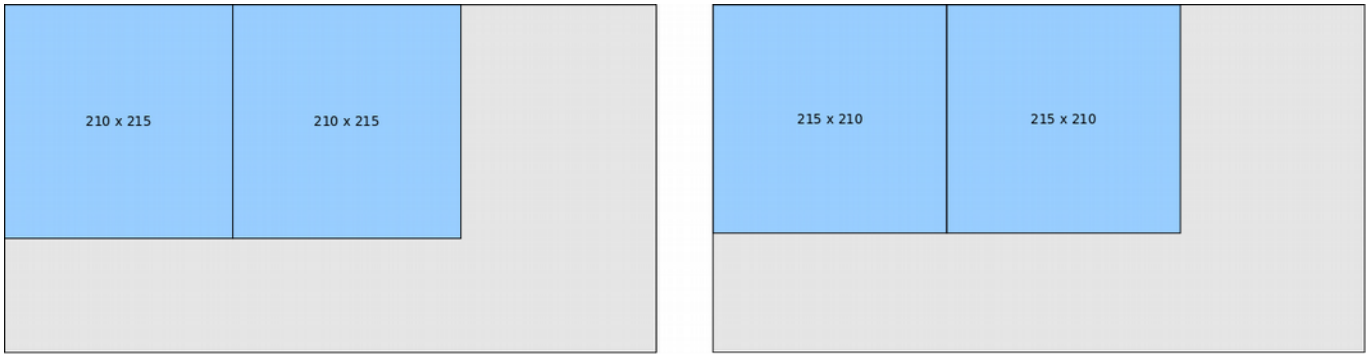

Figure 5. Deux dispositions pour placer un maximum de vitres (a) dans une plaque.

On peut facilement se convaincre qu'on ne peut pas faire mieux (dans la largeur du grand rectangle, on ne peut pas faire entrer la vitre deux fois, et on ne peut pas la faire rentrer 3 fois dans la longueur, on ne pourra pas faire mieux que deux). La perte est alors de 10,17 m² pour une plaque.

Pourtant, en terme d'aire, on peut faire entrer 4 fois l'aire du rectangle (a) dans celle du grand rectangle. On voit à nouveau que le minorant donné par le théorème 1 (pour une commande de 4 (a), il faut au moins une grande plaque) n'est pas toujours atteignable. On peut par contre utiliser le fait que seulement 2 vitres (a) peuvent entrer dans une grande plaque pour fournir un nouveau minorant :

Théorème 2 : Pour une commande donnée, si elle contient n vitres (a), il faut au moins E(n/2) grandes plaques (E étant la fonction partie entière supérieure).

Ainsi, par exemple, si la commande contient 11 (a), on sait qu'il faut au moins 6 grandes plaques. Ce minorant peut être parfois meilleur que celui du théorème 1.

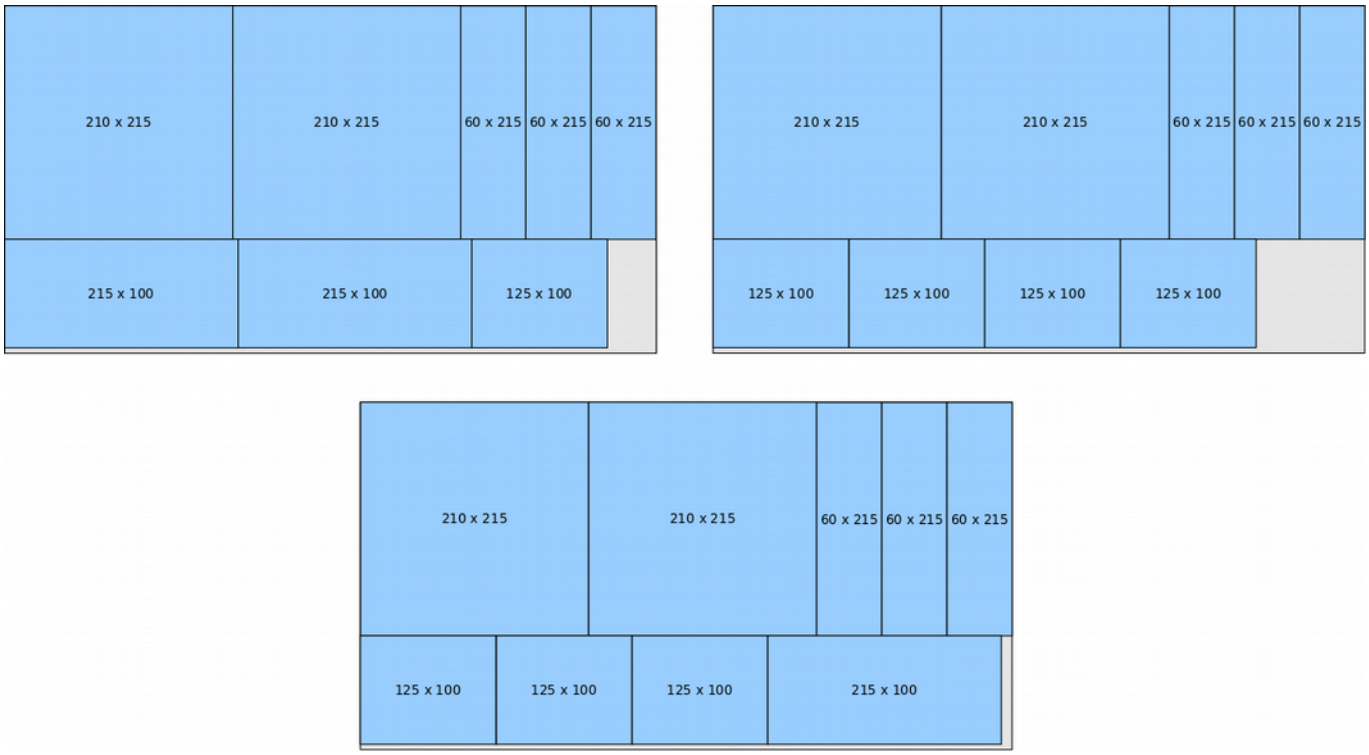

Figure 6. Dispositions de 3 commandes contenant 2 (a) en essayant choisir la commande minimisant les pertes (perte de 0,75 m² pour la première, 1,3 m² pour la deuxième et 0,4 m² pour la troisième).

Sachant qu'on ne peut pas mettre plus de 2 vitres (a) dans une plaque, on peut se demander comment on peut compléter une plaque contenant déjà deux vitres (a) avec d'autres vitres afin de minimiser les pertes. Cela peut conduire à des dispositions comme celles de la figure 6.

Remarque 4 : Pour minimiser les pertes dans une rectangle de largeur 100 cm et de longueur 600 cm, on note au passage la meilleure solution est d'utiliser 1 (b) et 3 (c).

Commande de vitres au format (b)

Combien au maximum peut-on disposer de vitres (b) dans un grand rectangle ? En disposant d'abord un maximum de vitres verticales ou horizontales et en complétant, on obtient les deux dispositions de la figure 7. La deuxième permet d'en faire entrer plus : 8 vitres (b) pour une perte de 2 m².

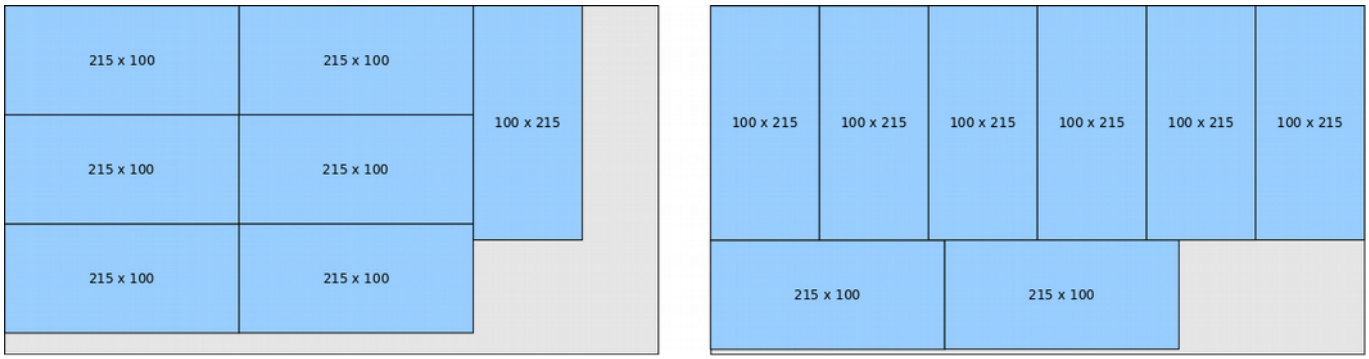

Figure 7. Deux dispositions avec uniquement des rectangles (b).

On peut noter que la seconde dispositions est optimale puisque la perte est de 2 m² et que l'aire d'un rectangle (b) est de 2,15 m². On ne peut pas envisager d'en faire entrer un autre dans le grand rectangle.

Théorème 3 : Pour une commande donnée, si elle contient n vitres (b), il faut au moins E(n/8) grandes plaques (même argument que pour le théorème 2).

De la même façon que précédemment, on peut chercher à voir comment peuvent être compléter de telles commandes (figure 7) pour minimiser la perte en n'utilisant qu'une seule plaque. La figure 8 montre de telles propositions.

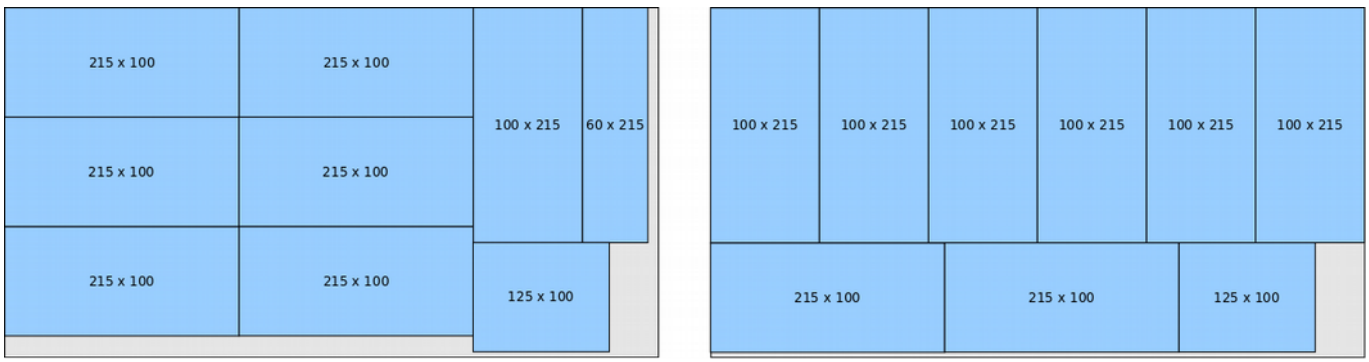

Figure 8. Des dispositions sur une plaque utilisant 7 ou 8 vitres (b) et complétée afin d'essayer de minimiser la perte (perte de 1,61 m² pour la première et de 0,75 m² pour la deuxième).

Commande de vitres au format (c)

Combien au maximum peut-on disposer de vitres (c) dans un grand rectangle ? En disposant d'abord un maximum de vitres verticales ou horizontales et en complétant, on obtient les dispositions de la figure 9.

On arrive à faire entrer 14 rectangles (c) dans un grand rectangle. Il reste encore 1,7 m² disponible et, en raisonnant sur les aires, on voit qu'il pourrait être envisageable d'essayer de faire entrer 15 rectangles

(c). Cependant, on n'y parvient pas.

Pour argumenter que cela est impossible, raisonnons sur la largeur du grand rectangle. Dans la largeur de 320 cm, parmi les deux dimensions d'un rectangle (c), on ne peut placer que 2 fois 125 cm (colonne de (c) verticaux), ou 3 fois 100 cm (colonne de (c) horizontaux), ou encore une fois 125 cm et une fois 100 cm (colonne avec un (c) vertical et un (c) horizontal). Dans chacun de ces cas, la perte sur la colonne est au moins de 0,7 m², 0,25 m² ou 0,95 m².
Si l'on faisait entrer 15 rectangles (b), on peut calculer que la perte serait de 0,45 m². Donc seule la disposition en colonnes de (c) horizontaux est envisageable pour ne pas dépasser cette perte. Or, dès qu'on fera une seconde colonne utilisant des rectangles disposés horizontalement, on aura une nouvelle perte de 0,25 m², et on dépasse déjà la perte de 0,45 m². Il est donc impossible de faire entrer 15 vitres (c) dans une grande plaque.

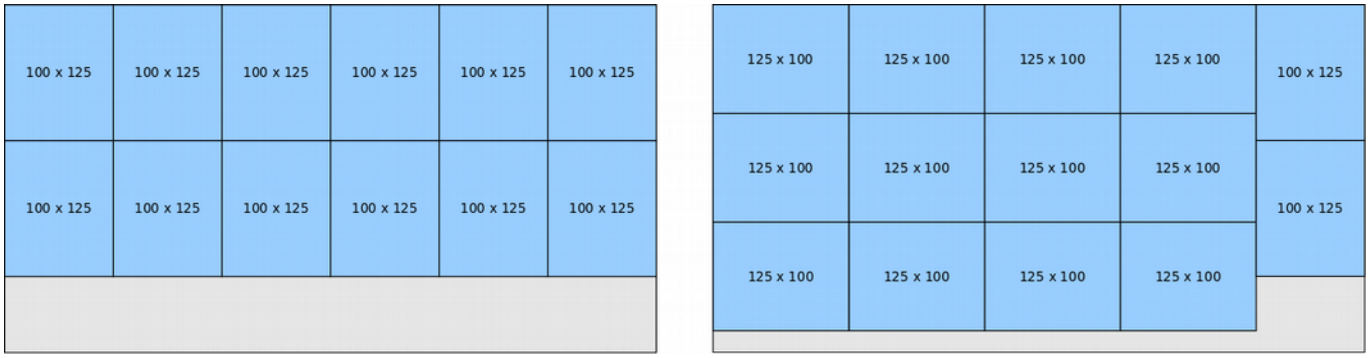

Figure 9. Deux dispositions de rectangles (c) dans un grand rectangle (12 et 14).
La perte est de 4,2 m²dans le premier cas, de 1,7 m² dans le deuxième cas.

On peut produire un théorème de minoration similaire à ceux énoncés précédemment :

Théorème 4 : Pour une commande donnée, si elle contient n vitres (c), il faut au moins E(n/14) grandes plaques (même argument que pour les théorèmes 2 et 3).

On peut essayer compléter les deux dispositions de la figure 9 avec d'autres vitres pour essayer de minimiser les pertes. On ne peut rien ajouter à la seconde disposition, mais la première peut être complétée par 2 rectangles (d) pour obtenir la disposition de la commande de la figure 10, qui a une perte de 1,62 m².

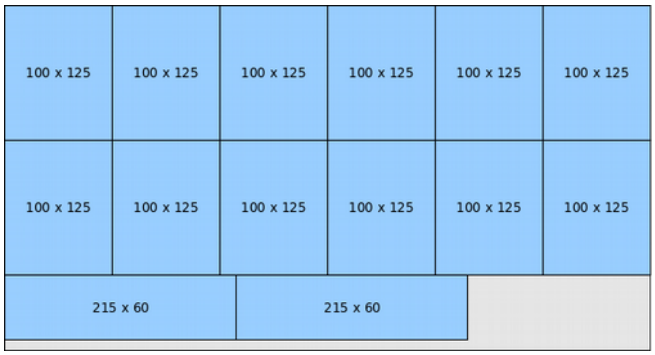

Figure 10. Disposition de 12 rectangles (c) et 2 rectangles (d).
On ne peut pas ajouter de rectangles à cette disposition.

Commande de vitres au format (d)

Combien au maximum peut-on disposer de vitres (d) dans un grand rectangle ? En disposant d'abord un maximum de vitres verticales ou horizontales et en complétant, on obtient les dispositions de la figure 11. On parvient à faire entrer dans un grand rectangle 12 rectangles (d), avec une perte de 3,72 m². En

raisonnant sur les aires, on voit que l'aire de 14 rectangles (d) n'excède pas l'aire du grand rectangle (l'aire de 15 (d) dépasserait). Comment s'assurer qu'on ne peut pas faire entrer 13, voire 14 rectangles de format (d) ? Une preuve du même type que précédemment doit pouvoir être faite, mais elle risque d'être assez longue est fastidieuse.

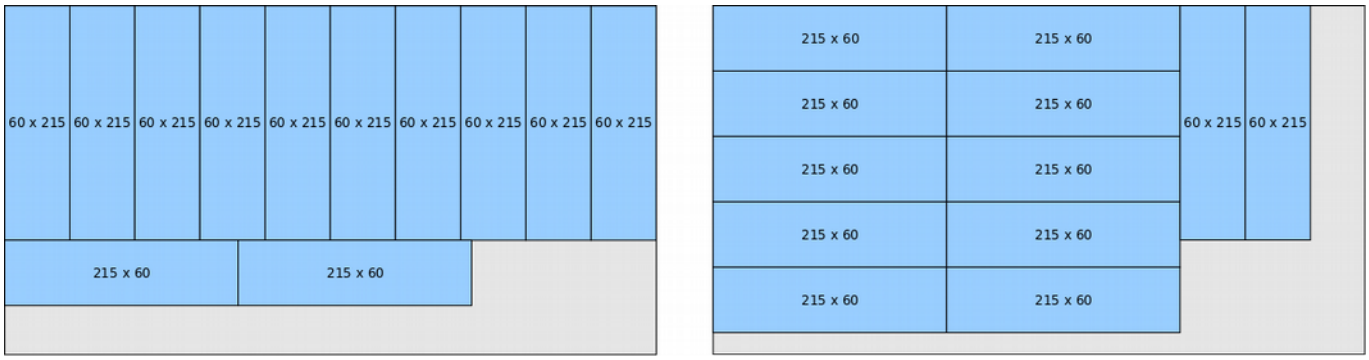

Figure 11. Une commande de 12 rectangles (d) dans deux dispositions différentes dans un grand rectangle.

En admettant qu'on ne peut faire mieux que 12, on peut à nouveau obtenir un théorème de minoration :

Théorème 5 : Pour une commande donnée, si elle contient n vitres (d), il faut au moins E(n/12) grandes plaques (même argument que pour les théorèmes 2, 3 et 4).

Les dispositions de la figure 11 peuvent être complétées comme dans le figure 12 ou complétées en supprimant préalablement un ou deux rectangles (d) comme dans la figure 13.

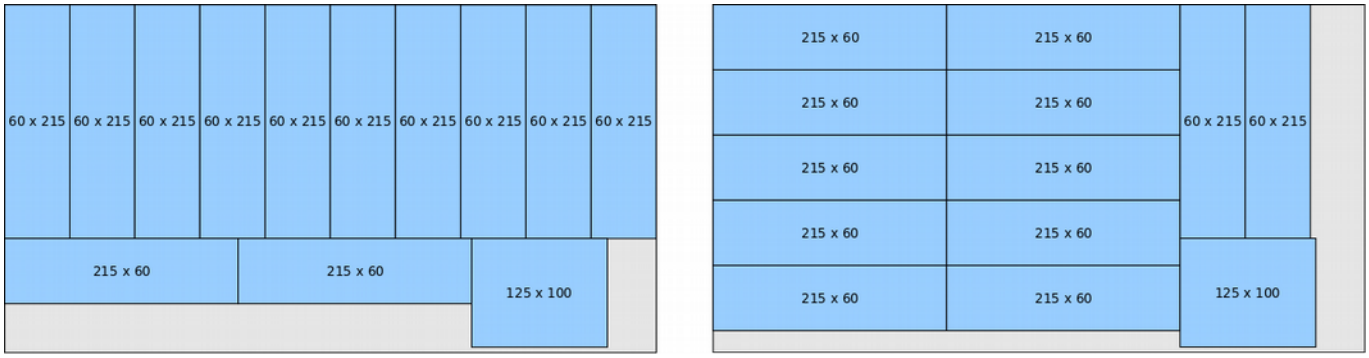

Figure 12. Deux dispositions correspondant à des commandes contenant 12 vitres (d). Chacune donne une perte de 2,47 m².

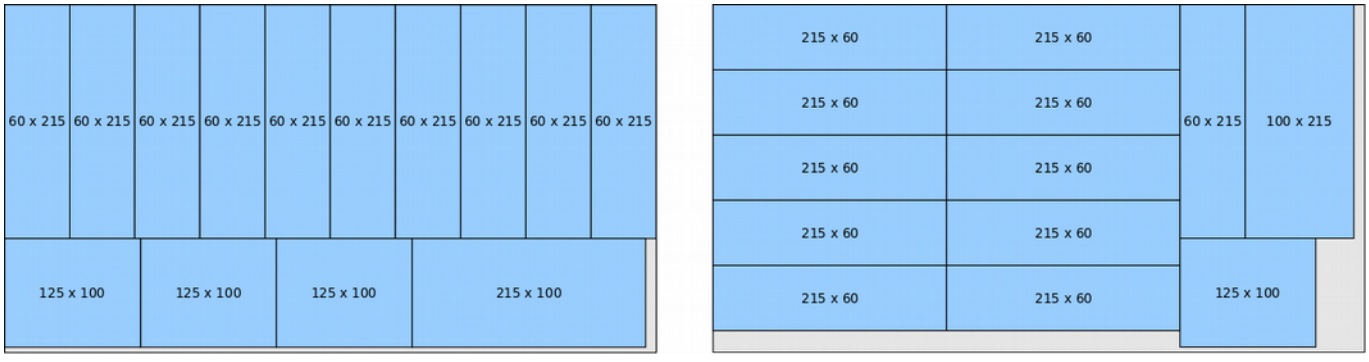

Figure 13. Deux dispositions pour des commandes contenant plus de 10 rectangles (d). La première a une perte de 0,4 m², la deuxième de 1,61 m².

4. Commandes minimisant les pertes pour une seule plaque

On peut aussi se poser la question de chercher des commandes qui produisent le moins de pertes possibles en n'excédant pas une grande plaque. Nous avons vu plusieurs dispositions très économiques dans les cas étudiés précédemment. Le meilleur cas rencontré correspond à une perte de 0,4 m² pour une plaque (figure 6 et figure 13). Peut-on trouver une commande qui aurait encore moins de pertes ?

Peut-on trouver d'autres commandes ayant des pertes petites (moins de 1 m² par exemple) ?

Note : remarquons au passage que nos deux commandes amenant à une perte de 0,4 m² sont assez similaires puisque deux rectangles (a) peuvent être remplacés par 7 rectangles (d).

En raisonnant sur les aires, on peut essayer de chercher quelles sont les plus grandes aires inférieures à l'aire d'une grande plaque que l'on peut obtenir en combinant les aires des rectangles (a), (b), (c) et (d).

Autrement dit, on ne se préoccupe pas des contraintes liées au configurations des rectangles dans le plan, on fait comme si le problème était en dimension 1.

On peut aborder ce problème de façon algorithmique, en énumérant toutes les combinaisons d'aires qui n'excèdent pas celles du grand rectangles, et chercher leur maximum. C'est ce que nous allons faire ici, avec un algorithme récursif. Formalisons d'abord le problème.

Problème 1 :
Données : une valeur maximale à ne pas dépasser V et un ensemble de valeurs $\{v_i\}_{i=1,\dots,n}$ positives.
Question : Quelle est la plus grande combinaison linéaire des v_i à coefficients entiers positifs inférieure ou égale à V ? Autrement, quelle est le maximum de l'ensemble $\{\sum_{i=1}^{n} a_i.v_i \leqslant V, a_i \in \mathbb{N}^*\}$?

La résolution algorithmique que nous allons présenter s'appuie sur le principe suivant :
Si $n = 1$, alors il suffit de faire entrer le plus de fois v_1 dans V (division euclidienne).
Sinon, on s'intéresse à la première valeur v_1 et pour toutes les valeurs possibles du coefficient a_1 (de 0 à $\left\lfloor \frac{V}{v_i} \right\rfloor$), on cherche la combinaison à coefficients entiers de valeurs $\{v_i\}_{i=2,\dots,n}$ qui est maximale inférieure à $V - a_1.v_1$. On prend ensuite le maximum parmi les solutions obtenues.

Cela peut donner lieu à l'algorithme récursif 14.

```python
# Cherche le remplissage optimal "pour une plaque" :
# ie. une combinaison entière positive des valeurs autorisées (Tab_val)
# qui se rapproche le plus possible d'un maximum donné (max)

# Optim retourne le reste minimal trouvé et la liste des coeficients correspondants à chaque valeur de Tab_val

def optim(max,Tab_val):
    nb_elt = len(Tab_val)
    reste = max
    if nb_elt == 1:
        val = Tab_val[0]
        nb_fois = reste // val
        reste = reste % val
        return (reste, [nb_fois])
    else :
        val = Tab_val[0]
        sous_tab = Tab_val[1:len(Tab_val)]
        nb_fois = reste // val
        reste_mini = max
        liste_mini = [0]*len(Tab_val)
        for i in range(nb_fois + 1):
            reste = max - (i * val)
            min_i , tab = optim(reste , sous_tab)
            if (min_i < reste_mini):
                reste_mini = min_i
                liste_mini = [i] + tab
        return (reste_mini, liste_mini)
```

Algorithme 14. Algorithme récursif pour le problème 1 (en Python).

Ainsi, optim(V, [v1,v2,v3,v4]) retourne une valeur r et une liste de coefficients a1,a2,a3,a4 tels que V = a1.v1 + a2.v2 + a3.v3 + a4.v4 + r, avec r le plus petit possible.

On peut alors évaluer optim(19200, [4515,2150,1250,1290]) pour connaître la combinaison d'aires de rectangles (a), (b), (c) et (d) la plus grande inférieure à l'aire d'un grand rectangle. Pour rester en nombres entiers, on travaille en millième de m². On obtient alors : (10, [0, 0, 4, 11]), autrement dit, si la commande 4 (c), 11 (d) entre dans une grande plaque, elle est celle qui produit le moins de perte (1 cm²). Malheureusement, on ne peut pas trouver de disposition pour cela :

> Raisonnons sur la largeur du grand rectangle (320) : avec les dimensions 100, 125, 60 et 215 on ne peut pas obtenir exactement 320 mais au mieux 315 (on pourrait utiliser optim(320, [60,100,125,215]) pour cela). Le trou obtenu dans la colonne aura au moins la largeur de la plus petite dimension d'une vitre (60) et donc une superficie de 300 cm².

On peut alors calculer la plus grande aire suivante en cherchant la combinaison d'aires la plus grande qui soit légèrement inférieure à celle qu'on vient de trouver (au moins plus petite d'une unité d'aire), c'est-à-dire en appelant (optim(19200 - 11, [4515,2150,1250,1290]), qui donne (24, [1, 1, 10, 0]) (impossible aussi en une seule plaque). Ainsi de suite, on peut compléter le tableau 15, on peut s'arrêter lorsque l'on retrouve une perte de 0,4 m², pour laquelle on avait une disposition possible (figures 6 et 13).

Commande	Retour	Perte (cm²)
optim(19200,[4515,2150,1250,1290])	(10, [0, 0, 4, 11])	1
optim(19200-11,[4515,2150,1250,1290])	(24, [1, 1, 10, 0])	3,5
optim(19200-36,[4515,2150,1250,1290])	(14, [0, 0, 5, 10])	5
optim(19200-51,[4515,2150,1250,1290])	(14, [1, 2, 0, 8])	6,5
optim(19200-66,[4515,2150,1250,1290])	(24, [0, 0, 6, 9])	9
optim(19200-91,[4515,2150,1250,1290])	(14, [1, 2, 1, 7])	10,5
optim(19200-106,[4515,2150,1250,1290])	(24, [0, 0, 7, 8])	13
optim(19200-131,[4515,2150,1250,1290])	(14, [1, 2, 2, 6])	14,5
optim(19200-146,[4515,2150,1250,1290])	(24, [0, 0, 8, 7])	17
optim(19200-171,[4515,2150,1250,1290])	(14, [1, 2, 3, 5])	18,5
optim(19200-186,[4515,2150,1250,1290])	(24, [0, 0, 9, 6])	21
Optim(19200-211,[4515,2150,1250,1290])	(14, [1, 2, 4, 4])	22,5
optim(19200-226,[4515,2150,1250,1290])	(24, [0, 0, 10, 5])	25
optim(19200-251,[4515,2150,1250,1290])	(14, [1, 2, 5, 3])	26,5
optim(19200-266,[4515,2150,1250,1290])	(14, [0, 1, 0, 13])	28
optim(19200-281,[4515,2150,1250,1290])	(9, [0, 0, 11, 4])	29
optim(19200-291,[4515,2150,1250,1290])	(14, [1, 2, 6, 2])	30,5
optim(19200-306,[4515,2150,1250,1290])	(14, [0, 1, 1, 12])	32
optim(19200-321,[4515,2150,1250,1290])	(9, [0, 0, 12, 3])	33
optim(19200-331,[4515,2150,1250,1290])	(14, [1, 2, 7, 1])	34,5
optim(19200-346,[4515,2150,1250,1290])	(14, [0, 1, 2, 11])	36
optim(19200-361,[4515,2150,1250,1290])	(9, [0, 0, 13, 2])	37
optim(19200-371,[4515,2150,1250,1290])	(14, [1, 2, 8, 0])	38,5
optim(19200-386,[4515,2150,1250,1290])	(14, [0, 1, 3, 10])	40

Tableau 15. Tableau des plus grandes aires totales de commandes inférieures à l'aire du grand rectangle. Les solutions en rouge sont connues comme impossibles (preuve au-dessus pour la première, théorème 5 pour la seconde).

Il faudrait alors tester si les commandes listées dans le tableau 15 permettent ou non une disposition des vitres dans une seule grande plaque et déterminer ainsi la commande qui entraîne la plus petite perte dans une plaque. Nous n'avons pas réussi à réaliser des dispositions dans le grand rectangle de commandes du tableau 15 (hormis la dernière). Cela ne veut pas dire que ce n'est pas possible Pour être sûr, il faudrait aussi vérifier que les combinaisons d'aires données par le programme dans le tableau 15 ne peuvent pas être obtenues avec d'autres commandes. Quoi qu'il en soit, cela permet d'engendrer des candidats de commandes optimales, ou au moins ayant peu de pertes. Ainsi, on pourrait poursuivre le tableau 15 pour essayer de trouver d'autres commandes avec de petites pertes.

Remarque 5 : On peut aussi noter certaines régularités dans le tableau 15 qui pourraient être étudiées.

Condition suffisante et perte maximale. Le théorème 1 (comme les suivants) fournit une condition nécessaire, basée sur les aires, pour qu'une commande utilise un certain nombre de grandes plaques et donc un minorant du nombre de plaques minimal. On chercher à exprimer une condition suffisante. Celle qui suit s'appuie sur le remplissage de la grande plaque en plaçant toutes les vitres de la commande verticalement côte à côte.

Théorème 6 (condition suffisante) : Si l'aire totale d'une commande est inférieure à 25/64 de l'aire de la grande plaque, alors la commande entre dans une seule grande plaque.

Preuve : Comme la longueur d'une vitre n'excède pas 215 cm on peut disposer les vitre de la commande verticalement côte à côte dans une grande plaque (vue horizontalement), comme sur la figure 16. Alors, la surface occupée par les vitres contient un rectangle dont le côté vertical mesure 125 cm et le côté horizontal la somme des largeurs des vitres de la commande (cf figure 16). Soit A_R l'aire de ce rectangle, A_C l'aire de la commande et A_P l'aire d'une grande plaque. Puisque $A_R \leq A_C \leq \frac{25}{64}A_P$, alors la dimension horizontale du rectangle ne peut excéder $\frac{1}{125}.\frac{25}{64}A_P = \frac{A_P}{320} = 600$. Dans le sens horizontal, l'ensemble des vitres disposé ainsi ne dépasse pas la longueur disponible (600 cm). Cette disposition des vitres entre bien dans une grande plaque.

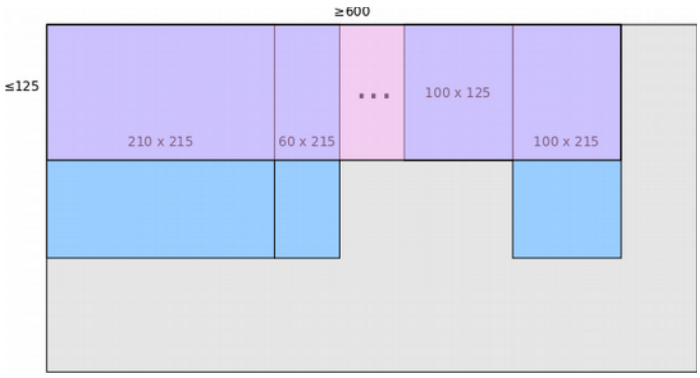

Figure 16. Configuration pour la preuve du théorème 6.

Remarque 6 : On peut améliorer ce résultat avec un coefficient de 1/2 au lieu de 25/64, voire plus, en prenant en compte le fait que l'on peut mettre de vitres de format (c) verticalement, l'une sous l'autre dans la disposition, et que les longueurs des rectangles de formats (a), (b) et (d) excèdent la moitié de 320 cm.

Remarque 7 : Le théorème 6 donne une condition suffisante pour que une commande soit possible en n plaques : il suffit qu'on puisse le regrouper en n sous-commandes dont les aires totales n'excèdent pas 25/64 de l'aire d'une grande plaque (ou la moitié dans une version optimisée du théorème 6).

Remarque 8 : Obtenir des majorants comme au théorème 6 peut permettre de majorer les pertes possibles. En effet, supposons que l'on sait, par exemple, que si une commande a une aire total inférieur à x % de l'aire d'une grande plaque alors la commande tient en entier sur une grande plaque. Si on a rempli une plaque d'une certaine façon, et qu'on peut ajouter une vitre sans que l'aire total ne dépasse ces x %, alors, on sait qu'il existe une disposition avec cette vitre en plus. Cela permet de dire qu'une commande d'aire totale inférieure à ces x % moins la plus petite aire d'une vitre peut toujours être complétée, et donc, en complétant une commande, on ne peut pas avoir une perte par plaque supérieure à une borne (ici, (100-x) % de l'aire de la grande plaque + plus petite aire de vitre).

5. Des commandes quelconques

En étudiant des commandes particulières, on comprend mieux le problème. Mais on ne répond pas directement au problème posé de « trouver une méthode pour minimiser la surface totale des chutes selon les commandes chaque semaine ».

Cependant, les cas que nous avons étudiés précédemment peuvent nous donner une idée : on pourrait découper les grandes plaques dans leur longueur pour obtenir un morceau de largeur 215 cm et un morceau de largeur 105 cm (ou 220 cm et 100 cm) pour positionner d'un côté les rectangles qui ont une dimension de 215 (formats (a), (b) et (d)) et de l'autre ceux qui ont une dimension de 100 (formats (b) et (c)), les formats (b) pouvant être utilisés d'un côté ou de l'autre. La figure 17 donne un exemple.

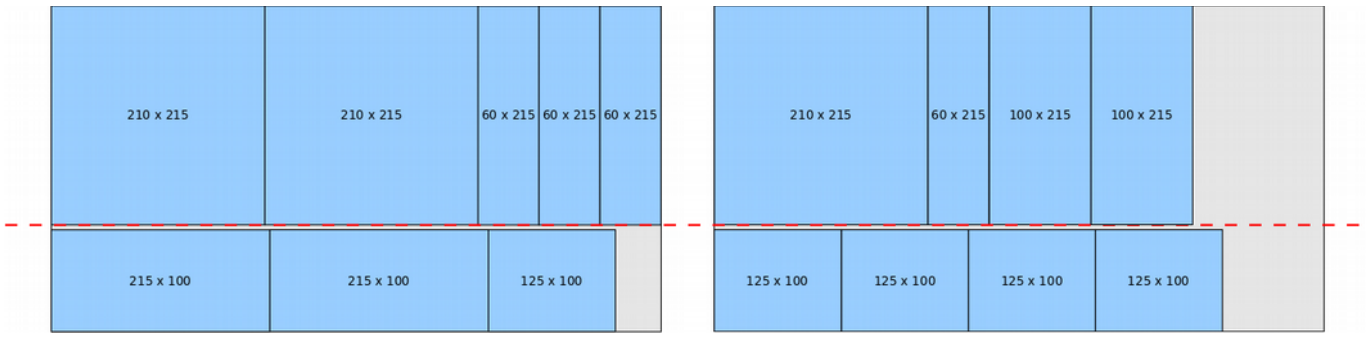

Figure 17. Une disposition de la commande 3 (a), 4 (b), 5 (c), 4 (d) sur deux grandes plaques selon la méthode proposée.

Dans la répartition de la figure 17, sur les 4 rectangles de format (b) de la commande, on a choisit d'en placer 2 dans la « bande » de largeur 215 et 2 autres dans la « bande » de largeur 100. Si l'on choisit d'en faire passer un du « haut » vers le « bas », on voit qu'une troisième grande plaque serait nécessaire. De plus, il reste à bien organiser la disposition des vitres dans chaque bande, pour minimiser le nombre de grandes plaques utilisées. Prenons l'exemple de la « bande » de largeur 100 dans laquelle on doit placer 2 (b) et 6 (c). Selon la disposition choisie, il faudra 2 ou 3 plaques (figure 18). Un problème d'optimisation sur une « bande » apparaît.

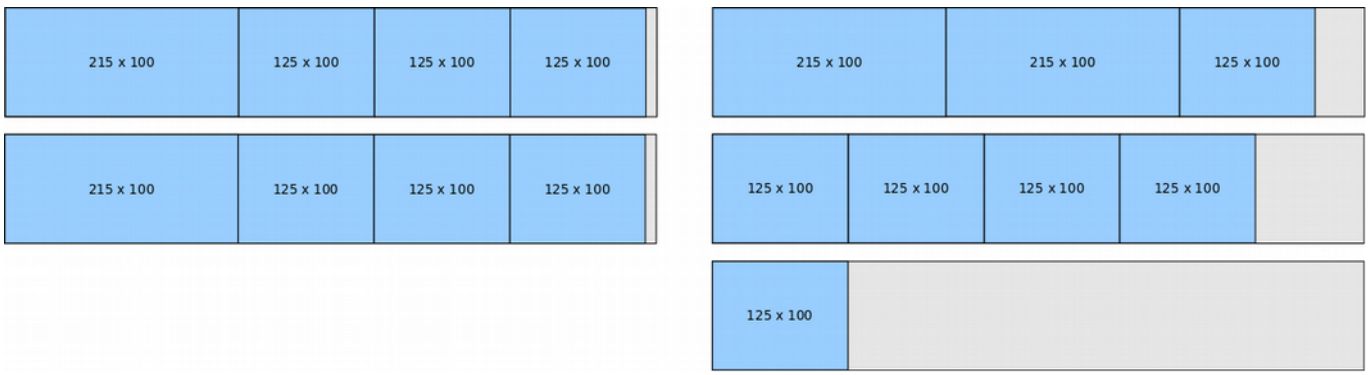

Figure 18. Deux dispositions de 2 (b) et 6 (c) dans une « bande » de 100 x 600. À gauche, utilisant 2 « bandes », à droite, trois.

On peut alors s'intéresser à ce nouveau problème, qui émerge de la méthode proposée.

On considère seulement une dimension des éléments de la commande (215 et 125 pour (b) et (c), 210, 100 et 60 pour (a), (b) et (d)) à disposer dans des « lignes » de longueur 600. À chaque fois, on veut savoir quelle est la disposition qui produit le moins de pertes, ou, de façon équivalente, celle qui utilise le moins de plaques. On peut voir ce problème comme un problème de découpe de câble : on dispose de bobines de câble d'une longueur L et on a une commande de câbles de différentes longueurs ℓ_i, comment découper tous les câbles de la commande en ayant le moins de pertes (ou en utilisant le moins de bobines) ?
Plus formellement, le problème se décrit ainsi :

Problème 2 :
Données : une dimension L et un ensemble de dimensions $\{\ell_i\}_{i=1,\dots,n}$.
Question : déterminer une répartition des ℓ_i en groupes de sommes inférieures à L avec un nombre minimal de groupes ?

Dans notre cas précis, les ℓ_i sont particuliers, puisqu'ils ne peuvent prendre qu'un nombre fini de valeurs connues à l'avance. Le problème peut alors être abordé plus simplement. Ainsi, on peut formuler le problème 2bis.

Problème 2bis : On a fixé un ensemble fini de valeurs $(\ell_i)_{i=1,\dots,n}$ (ici (210, 100, 60) ou (215, 125)).
Données : Une dimension L et une commande $(a_i)_{i=1,\dots,n}$ où a_i est le nombre d'éléments de dimension ℓ_i dans la commande.
Question : déterminer une répartition des éléments de la commande en groupes de sommes inférieures à L avec le nombre minimal de groupes ?

Étude des problèmes 2 et 2bis

Une première idée pour aborder ce problème est de commencer par répartir les éléments de plus grande taille. En parcourant les éléments dans l'ordre du plus grand au plus petit, on peut les grouper en essayant d'abord de les placer dans des groupes où il reste de la place et si c'est impossible, en créant un nouveau groupe d'éléments. Cela donne un algorithme dit « glouton » qui permet de faire un groupement des commandes. L'algorithme 19 décrit plus précisément cette méthode.

```python
# 'valeurs' est la liste des valeurs possible (l_i) et 'commande' est la liste des quantités de chaque l_i (a_i)
# 'valeurs' doit être donné dans l'ordre décroissant et 'commande' doit correspondre à cet ordre décroissant

def repartition_glouton(L, valeurs, commande):
    repartition = [] #destiné à contenir les listes de l_i regroupées (de somme < ou = à L)
    for i in range(len(valeurs)):
        val = valeurs[i]
        nb = commande[i]
        for j in range(len(repartition)):
            nb_dispo = (L - sum(repartition[j])) // val #nombre d'éléments de taille val pouvant être ajoutés à la liste
            nb_ajout = min(nb, nb_dispo) # on n'en ajoute pas plus que le nombre restant dans la commande
            if nb_ajout != 0:
                for k in range(nb_ajout):
                    repartition[j].append(val) # on les ajoute à la liste concernée
            nb = nb - nb_ajout #on décrémente nb de ce qui a été ajouté dans la liste concernée
        if nb != 0: #cette partie contruit des listes nouvelles et y place les éléments val qui n'ont pu être placés.
            nb_dans_L = L // val
            q = nb // nb_dans_L
            r = nb % nb_dans_L
            repartition = repartition + [[val]*nb_dans_L for _ in range(q)]
            if r != 0:
                repartition.append([val]*r)
    return (repartition, len(repartition))
```

Algorithme 19. Méthode « gloutonne » pour tout cas du problème 2bis (donc aussi du problème 2), en Python.

Cependant, si cet algorithme donne toujours une solution recevable, il ne fournit pas toujours une solution optimale. Pour s'en convaincre, on peut l'appliquer à l'exemple de la figure 18, il donnera une solution à 3 « bandes » (voir figure 20) alors qu'il existe une solution en utilisant seulement deux.

```
repartition_glouton(600,[215,125],[2,6])
```

```
([[215, 215, 125], [125, 125, 125, 125], [125]], 3)
```

Figure 20. Application de l'algorithme 19 à la commande de la figure 18. La solution produite utilise 3 « bandes ».

On peut choisir d'essayer de résoudre de manière exacte le problème 2bis, en utilisant une approche exhaustive, qui sera plus longue à appliquer puisqu'elle devra explorer tous les cas possible pour trouver l'optimal. Pour cela il est d'abord nécessaire d'avoir un algorithme qui liste toutes les façons de remplir au maximum une « bande » de longueur donnée, avec les différentes valeurs possibles des éléments (on entend par maximum, un remplissage auquel on ne pourrait plus ajouter aucun élément, quelle que soit sa valeur, un peu comme nous en avions cherché dans la section 3). L'algorithme 21 fait cela de façon exhaustive (il est décrit de façon récursive).

```
# Liste les remplissage "optimaux" (ie. on ne peut rien ajouter) de "plaques" (valeur donnée)
# avec des éléments (valeurs) d'une liste donnée (liste_val)

def liste_plaques_optim(plaque,liste_val): #la liste doit être triée décroissante
    val = liste_val[0]
    if len(liste_val) == 1:
        nb_max = plaque // val
        reste = plaque % val
        return ([[val]*nb_max]) # + le reste ?
    else:
        nb_max = plaque // val
        liste_repartitions = []
        for nb in range (nb_max + 1):
            reste_plaque = plaque - (nb * val)
            liste_ecourtee = liste_val[1:]
            (liste_repartition_reste) = liste_plaques_optim(reste_plaque , liste_ecourtee) # + reste ?
            for rep in liste_repartition_reste:
                nouvelle_rep = [val]*nb + rep
                liste_repartitions.append(nouvelle_rep)
        return (liste_repartitions)
```

```
liste_plaques_optim(11,[4,3])
```

```
[[3, 3, 3], [4, 3, 3], [4, 4, 3]]
```

Algorithme 21. Algorithme listant les remplissage maximum (en Python) et application pour une bande de dimension 11, avec les valeurs 3 et 4 : trois bandes maximums sont possibles.

Pour bien comprendre, on peut appliquer cet algorithme aux dimensions de notre cas particulier : 210, 100 et 60 dans une bande de 600 et 215 et 125 dans une bande de 600 (figure 22). On reconnaît des cas que l'on a pu voir précédemment.

```
liste_plaques_optim(600,[210,100,60])
```

```
[[60, 60, 60, 60, 60, 60, 60, 60, 60, 60],
 [100, 60, 60, 60, 60, 60, 60, 60, 60],
 [100, 100, 60, 60, 60, 60, 60, 60],
 [100, 100, 100, 60, 60, 60, 60, 60],
 [100, 100, 100, 100, 60, 60, 60],
 [100, 100, 100, 100, 100, 60],
 [100, 100, 100, 100, 100, 100],
 [210, 60, 60, 60, 60, 60, 60],
 [210, 100, 60, 60, 60, 60],
 [210, 100, 100, 60, 60, 60],
 [210, 100, 100, 100, 60],
 [210, 210, 60, 60, 60],
 [210, 210, 100, 60]]
```

```
liste_plaques_optim(600,[215,125])
```

```
[[125, 125, 125, 125], [215, 125, 125, 125], [215, 215, 125]]
```

Figure 22. Énumération des « bandes » optimums pour les données du problème.

Cet algorithme va être utile pour résoudre exactement le problème 2bis : il reste a essayer judicieusement toutes les combinaisons des ces « bandes » optimums de façon à couvrir la commande (quitte à avoir des éléments en plus de la commande) et à conserver la combinaisons qui utilise le moins de « bandes ». C'est ce que fait l'algorithme 23 décrit récursivement (et qui nécessite de définir quelques fonctions auxiliaires au préalable).

```
def nettoyer(valeurs, commande):
    valeurs_clean = []
    commande_clean = []
    for i in range(len(valeurs)):
        if commande[i] != 0:
            valeurs_clean.append(valeurs[i])
            commande_clean.append(commande[i])
    return(valeurs_clean, commande_clean)
```

```
def val_total_commande(valeurs, commande):
    S = 0
    for i in range(len(valeurs)):
        S += valeurs[i]*commande[i]
    return S
```

```
def diminuer(valeurs, commande, a_retirer):
    new_commande = [0]*len(commande)
    # new_valeurs =
    for i in range(len(valeurs)):
        nb = a_retirer.count(valeurs[i])
        new_commande[i] = max( 0 , commande[i] - nb)
    return (new_commande)
```

```
# Recherche de la meilleure répartition "en ligne" (minimise le nombre de plaques pour une commande donnée)
# Peux servir sur les aires, mais ne tient pas compte de dispositions impossibles...

def optim_lin(plaque, valeurs, commande):
    (valeurs,commande) = nettoyer(valeurs, commande)
    L = liste_plaques_optim(plaque, valeurs)
    total_commande = val_total_commande(valeurs, commande)
    if total_commande == 0:
        return([],0)
    elif total_commande <= plaque:
        liste_final = []
        for i in range(len(valeurs)):
            for j in range(commande[i]):
                liste_final.append(valeurs[i])
        return([liste_final],1)
    else:
        meilleur = sum(commande) +1
        for plaque_choisie in L:
            reste_commande = diminuer(valeurs, commande, plaque_choisie)
            (repartition, cout) = optim_lin(plaque, valeurs, reste_commande)
            if cout + 1 < meilleur:
                meilleure_repartition = repartition
                meilleure_repartition.append(plaque_choisie)
                meilleur = cout + 1
        return(meilleure_repartition, meilleur)
```

Algorithme 23. Recherche exhaustive de la meilleures disposition dans un bande (en Python).

On peut appliquer cet algorithme à l'exemple de la figure 18 pour voir qu'il fournit la solution optimal (là où l'algorithme glouton ne la donnait pas (figure 24). D'autres exemples montrent que l'algorithme glouton ne donne parfois mais pas toujours une solution optimale (figure 25).

```
optim_lin(600,[215,125],[2,6])
```

```
([[215, 125, 125, 125], [215, 125, 125, 125]], 2)
```

Figure 24. Solution optimale obtenue par l'algorithme 23.

Retour au problème de découpe des plaques de verre

On peut alors s'appuyer sur l'un ou l'autre des algorithmes 19 et 23 pour proposer un algorithme qui propose un remplissage optimal des deux bandes selon la méthode proposée au début de la section et schématisée par la figure 17. Comme, dans cette méthode, les rectangles de formats (b) peuvent être utilisés soit dans leur largeur soit dans leur longueur, on peut étudier toutes les répartitions de ces

rectangles (de aucun à tous du même côté). Pour chaque cas, on applique alors un algorithme de répartition dans une ligne et on regarde combien de « bandes » sont utilisées (maximum entre la solution « 215 » et la solution « 100 » pour le nombre de grandes plaques utilisées). Il suffit alors de conserver la répartition des rectangles de format (b) qui minimise le nombre de grande plaques. Cela donne les algorithmes 26 et 27 selon que l'on utilise l'algorithme exhaustif ou glouton pour remplir les « bandes ».

```
repartition_glouton(13,[7,4,3],[4,10,6])
```

```
([[7, 4], [7, 4], [7, 4], [7, 4], [4, 4, 4], [4, 4, 4], [3, 3, 3, 3], [3, 3]],
 8)
```

```
print(optim_lin(13,[7,4,3],[4,10,6]))
```

```
([[7, 4], [7, 3, 3], [7, 3, 3], [7, 3, 3], [4, 4, 4], [4, 4, 4], [4, 4, 4]], 7)
```

```
optim_lin(600,[210,100,60],[3,3,3])
```

```
([[210, 210, 100], [210, 100, 100, 60, 60, 60]], 2)
```

```
repartition_glouton(600,[210,100,60],[3,3,3])
```

```
([[210, 210, 100, 60], [210, 100, 100, 60, 60]], 2)
```

Figure 25. Répartitions obtenues par l'algorithme exhaustif et l'algorithme glouton.

```python
def optim_2_col(commande):
    L=600
    (l1,l2,l3) = (210,100,60)
    (l4,l5) = (215,125)
    [a,b,c,d] = commande
    opt=sum(commande)+1
    for nb_b_vert in range(b+1):
        nb_b_horiz = b - nb_b_vert
        (dispo_plaq_215, opt_215) = optim_lin(L,[l1,l2,l3],[a,nb_b_vert,d])
        (dispo_plaq_100, opt_100) = optim_lin(L,[l4,l5],[nb_b_horiz,c])
        nb_plaq = max(opt_215, opt_100)
        if nb_plaq < opt:
            opt = nb_plaq
            meilleure_dispo = (dispo_plaq_215, dispo_plaq_100, opt)
    return(meilleure_dispo)
```

Algorithme 26. Recherche d'une répartition de commande en « deux bandes » faisant appel à la méthode exhaustive.

```python
def optim_2_col_glouton(commande):
    L=600
    (l1,l2,l3) = (210,100,60)
    (l4,l5) = (215,125)
    [a,b,c,d] = commande
    opt = sum(commande)+1
    for nb_b_vert in range(b):
        nb_b_horiz = b - nb_b_vert
        (dispo_plaq_215, opt_215) = repartition_glouton(L,[l1,l2,l3],[a,nb_b_vert,d])
        (dispo_plaq_100, opt_100) = repartition_glouton(L,[l4,l5],[nb_b_horiz,c])
        nb_plaq = max(opt_215, opt_100)
        if nb_plaq < opt:
            opt = nb_plaq
            meilleure_dispo = (dispo_plaq_215, dispo_plaq_100, opt)
    return(meilleure_dispo)
```

Algorithme 27. Recherche d'une répartition de commande en « deux bandes » faisant appel à la méthode exhaustive.

Puisque la méthode exhaustive donne les solutions optimales sur une bande (contrairement à la méthode gloutonne), elle donne de meilleures solutions (ou au moins équivalentes) pour le découpage des grandes plaques. La figure 28 donne un exemple de commande pour laquelle la méthode exhaustive est strictement meilleure (2 (a), 5 (b), 6 (c), 8 (d)).

Non optimalité.

Sur des exemples, on peut voir que la méthode de découpe en deux « bandes » n'est pas toujours optimale (c'est normal, la méthode se restreint à certaines dispositions seulement). La figure 29 montre cela sur des exemples vus au début.

```
optim_2_col([2,5,6,8])
```

```
([[210, 210, 60, 60, 60], [100, 100, 100, 60, 60, 60, 60, 60]],
 [[215, 125, 125, 125], [215, 125, 125, 125]],
 2)
```

```
optim_2_col_glouton([2,5,6,8])
```

```
([[210, 210, 100, 60], [60, 60, 60, 60, 60, 60, 60]],
 [[215, 215, 125], [215, 215, 125], [125, 125, 125, 125]],
 3)
```

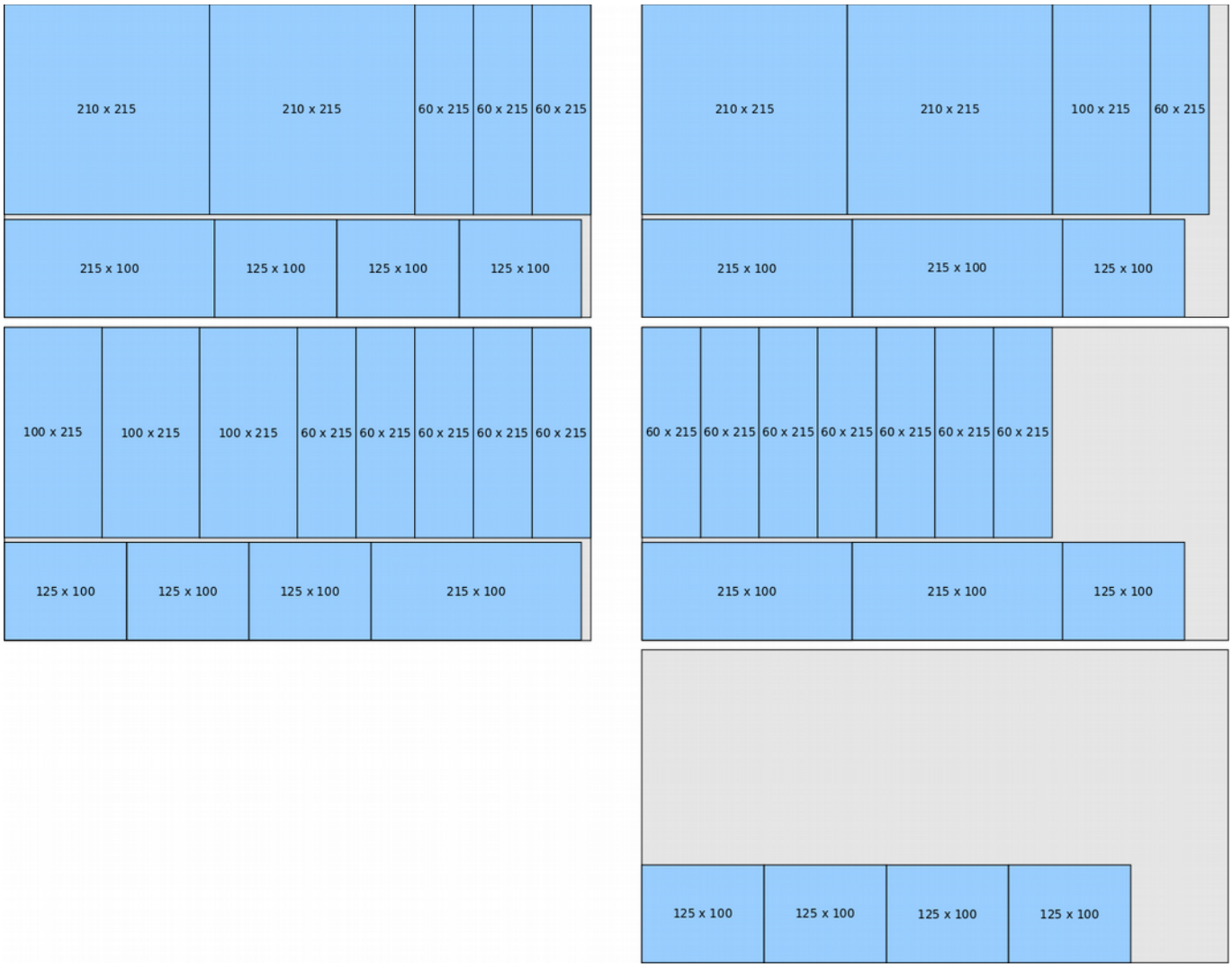

Figure 28. Comparaison des méthodes faisant appel à l'exhaustivité ou à l'approche gloutonne sur les « bandes », avec représentation des solutions obtenues (à gauche pour la première, à droite pour la seconde).

```
optim_2_col([0,1,1,11]) #Exemple de la figure 11
```

```
([[60], [60, 60, 60, 60, 60, 60, 60, 60, 60, 60]], [[215, 125]], 2)
```

```
optim_2_col([0,6,2,2]) #Exemple de la figure 4
```

```
([[100, 100, 60, 60]], [[215, 215, 125], [215, 215, 125]], 2)
```

```
optim_2_col([0,0,12,2]) #Exemple de la figure 10
```

```
([[60, 60]],
 [[125, 125, 125, 125], [125, 125, 125, 125], [125, 125, 125, 125]],
 3)
```

Figure 29. Application de l'algorithme basé sur l'exhaustivité sur les « bandes » sur 3 commandes vus précédemment qui peuvent être disposés sur une seule grande plaque (le dernier ne propose qu'une solution sur 3 plaques!).

Retour au problème de départ

Revenons à notre problème de départ. Nous avons maintenant une méthode pour produire des répartitions de commandes qui s'adaptent à toutes les commandes. En nous appuyant sur le travail

réalisé jusqu'ici, peut-on trouver d'autres méthodes qui traiteraient des commandes quelconques ?

Application de l'algorithme exhaustif 23 au problème de découpe en deux dimensions

En raisonnant sur les aires, on peut vouloir appliquer les algorithmes proposés pour la dimension 1 au cas de dimensions 2. On traite alors les aires comme on l'a fait pour les longueurs dans le cas précédent. Bien sûr, les solutions obtenues ne sont pas forcément réalisables (il faut vérifier qu'on peut trouver des dispositions associées dans de grands rectangles qui soient acceptables). Pour l'algorithme utilisant une méthode exhaustive, si les dispositions obtenues sont réalisables, la solutions produite sera optimale (ce ne sera pas forcément le cas pour la méthode gloutonne. La figure 30 montre des cas où la méthode exhaustive, donne des solutions non réalisables. La figure 31 montre un cas réalisable (exemple de la figure 28) proposé par la méthode exhaustive et la méthode gloutonne.

```
optim_lin(19200, [4515,2150,1250,1290], [3,0,0,1]) # impossible en une plaque (théorème 2)
```

```
([[4515, 4515, 4515, 1290]], 1)
```

```
optim_lin(19200, [4515,2150,1250,1290], [0,0,4,11]) #vu comme impossible à réaliser en 1 plaque (tableau 15)
```

```
([[1250,
   1250,
   1250,
   1250,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290]],
 1)
```

Figure 30. Applications de l'optimisation en dimension 1 à la dimension 2.

```
optim_lin(19200, [4515,2150,1250,1290], [2,5,6,8])
```

```
([[4515, 4515, 2150, 2150, 2150, 2150, 1250],
  [2150,
   1250,
   1250,
   1250,
   1250,
   1250,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290]],
 2)
```

```
repartition_glouton(19200, [4515,2150,1250,1290], [2,5,6,8])
```

```
([[4515, 4515, 2150, 2150, 2150, 2150, 1250],
  [2150,
   1250,
   1250,
   1250,
   1250,
   1250,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290,
   1290]],
 2)
```

Figure 31. Applications des méthodes exhaustives et gloutonnes de la dimension 1 à la dimension 2.

Rappelons que la méthode exhaustive devient inapplicable si l'on s'intéresse à des commandes trop grosses, alors que la méthode gloutonne reste plus raisonnable pour de grosses commandes.

Inspiration de la méthode en dimension 1

En dimension 1 (sur une « bande »), nous avons vu que la méthode exhaustive s'appuyait d'abord sur la construction d'une liste de toutes les commande qui entrent sur un « bande » auxquelles aucun élément ne peut être ajouté. Par ailleurs, nous avons vu que la méthode de dimension 1 appliquée sur les aires (en dimension 2) produisait des solutions non-réalisables.

Pour éviter ce problème, une idée peut être de se donner une liste de commandes de vitres qui tiennent sur une plaque sans pouvoir être augmentées. Bien sûr, on ne peut pas toutes les lister comme précédemment. La figure 32 montre que l'algorithme d'énumération des plaques précédent identifie 168 possibilités, et il faudrait pour chacune vérifier si oui ou non il existe une disposition réalisable en une plaque (et on perd par ailleurs, certaines propriétés qui garantissaient l'optimalité de la méthode exhaustive en dimension 1).

```
L=liste_plaques_optim(19200, [4515,2150,1250,1290])
len(L)
```

```
168
```

Figure 32. 168 commandes dont il faudrait vérifier la réalisabilité.

Par contre, on peut envisager de collecter quelques dispositions qui auraient des pertes faibles. On peut ensuite les utiliser comme dans l'algorithme exhaustif, en essayant de chercher la meilleure combinaison de ces « commandes » pour couvrir une commande donnée. Cela donne une méthode que l'on peut appliquer à des cas de commandes pas trop grandes. En effet, comme pour le cas précédent, sur des cas trop grands, l'algorithme devient trop gourmand en temps de calcul.

La figure 33 montre la liste de configurations que nous avons choisi d'utiliser (issues des commandes étudiées plus haut, en conservant celles qui avaient les pertes les plus faibles). Cette liste de commande est convertie sous forme de « plaque » (liste des aires des vitres utilisées).

```
Liste_config_perte_faible_nb =[
    [2,2,1,3], #figure 6
    [2,0,4,3], #figure 6
    [2,1,3,3], #figure 6
    [0,4,3,5], #figure 28 (perte 0,4 oubliée avant)
    [0,7,1,1], #figure 8
    [0,8,1,0], #figure 8
    [0,0,14,0], #figure 9
    [0,0,12,2], #figure 10
    [0,0,1,12], #figure 12
    [0,1,3,10], #figure 13
    [0,0,4,10], #figure 13 (variante)
    [0,2,1,10], #figure 13 (variante)
    [0,1,1,11] #figure 13
    #à compléter avec d'autres configurations ?
]
```

```
Liste_config_perte_faible_val = []
for [a,b,c,d] in Liste_config_perte_faible_nb:
    Liste_config_perte_faible_val.append([4515]*a+[2150]*b+[1250]*c+[1290]*d)
```

Figure 33. Liste des configurations de perte faible choisies (produite dans la liste 'Liste_config_perte_faible_val' à partir d'une liste sous forme de « commandes »)

En adaptant l'algorithme d'optimisation en dimension 1, en utilisant les fonctions auxiliaires déjà définies, et en modifiant la fonction 'nettoyer' pour qu'elle s'adapte à notre situation ('nettoyer2'), on peut écrire un algorithme qui met en place notre méthode, et trouve la combinaison d'éléments de 'Liste_config_perte_faible_val' qui couvre la commande en minimisant le nombre de plaques utilisées.

Cela est détaillé dans l'algorithme 34.

```
def nettoyer2(valeurs, commande, liste_config):
    valeurs_clean = []
    commande_clean = []
    liste_config_clean = []
    liste_config_a_garder = [0]*len(liste_config)
    for i in range(len(valeurs)):
        if commande[i] != 0:
            valeurs_clean.append(valeurs[i])
            commande_clean.append(commande[i])
            for j in range(len(liste_config)):
                if (valeurs[i] in liste_config[j]):
                    liste_config_a_garder[j]=1
    for k in range(len(liste_config)):
        if liste_config_a_garder[k] == 1:
            liste_config_clean.append(liste_config[k])
    return(valeurs_clean, commande_clean, liste_config_clean)
```

```
# Recherche de la meilleure répartition (minimise le nombre de plaques pour une commande donnée)

def optim_lin_partiel(plaque, valeurs, commande, liste_config):
    (valeurs, commande, liste_config) = nettoyer2(valeurs, commande, liste_config)
    total_commande = val_total_commande(valeurs, commande)
    if total_commande == 0:
        return([],0)
    elif total_commande <= (plaque // 2): #à justifier
        liste_final = []
        for i in range(len(valeurs)):
            for j in range(commande[i]):
                liste_final.append(valeurs[i])
        return([liste_final],1)
    else:
        meilleur = sum(commande) + 1
        for plaque_choisie in liste_config:
            reste_commande = diminuer(valeurs, commande, plaque_choisie)
            (repartition, cout) = optim_lin_partiel(plaque, valeurs, reste_commande, liste_config)
            if cout + 1 < meilleur:
                meilleure_repartition = repartition
                meilleure_repartition.append(plaque_choisie)
                meilleur = cout + 1
        return(meilleure_repartition, meilleur)

def optim_lin_partiel_2d(commande):
    plaque = 19200
    valeurs = [4515, 2150, 1250, 1290]
    return(optim_lin_partiel(plaque, valeurs, commande, Liste_config_perte_faible_val))
```

Algorithme 34. Optimisation du nombre de plaques en utilisant uniquement des configurations listées au préalable.

L'application de l'algorithme 34 sur quelques exemples est donnée dans la figure 35.

```
optim_lin_partiel_2d([2,5,6,8])
```

```
([[2150, 2150, 2150, 2150, 1250, 1250, 1250, 1290, 1290, 1290, 1290, 1290],
  [4515, 4515, 2150, 1250, 1250, 1250, 1290, 1290, 1290]],
 2)
```

```
optim_lin_partiel_2d([7,7,3,6])
```

```
([[4515, 2150],
  [4515, 4515, 2150, 2150, 1250, 1290, 1290, 1290],
  [4515, 4515, 2150, 2150, 1250, 1290, 1290, 1290],
  [4515, 4515, 2150, 2150, 1250, 1290, 1290, 1290]],
 4)
```

Figure 35. Application de l'algorithme 34 sur deux commandes.

D'autres méthodes peuvent être imaginées pour traiter les commandes. En particulier, l'algorithme 34 a une complexité algorithme telle qu'on ne peut pas traiter de commandes trop grosses. Des méthodes approchées plus efficaces pourraient être envisagées.

Pour aller plus loin...

Le problème étudié se nomme cutting stock (2d ou 1d) dans la littérature. Nous avons étudié des variantes où les commandes sont souvent limitées à une liste de dimensions fixées à l'avance (ce qui, en général, simplifie le problème. La version où il est obligatoire de trouver des découpes qui peuvent être

réalisées de « bord à bord » est appelé la version guillotine du problème cutting stock 2d (souvent utilisé dans l'industrie du verre).

Ce sont des problèmes d'optimisation combinatoire. Dans la même famille, on rencontre les problèmes de packing (dont le bin packing), de sac à dos (knapsack problem).

Généralement, ces problèmes ont une complexité algorithmique telle qu'il faut envisager de chercher des algorithmes plus efficaces qui donnent des solutions approchées (des méthodes gloutonnes par exemple, mais aussi de la programmation dynamique, des algorithmes génétiques…).