Étude de cas : Publicité sur la lune

Dimitri Watel

Optimisation 2, Semestre 5, 2019-2020

Déroulement du projet

Ce projet consiste en l'étude d'un problème via toutes les techniques de recherche opérationnelle que l'on vous a enseigné jusqu'à présent ou que vous connaissez. Il propose des étapes d'étude de l'aspect théorique ou appliqué.

Il s'effectue en binôme. Vous devez rendre un **code**, un **rapport** et effectuer une **soutenance**. Le code et le rapport doivent être rendus en temps et la soutenance doit être effectuée sans quoi la note que le binôme se verra attribuée sera 0. En cas d'absence d'un des deux membres à la soutenance, la personne présente se verra le droit de soutenir seule. Si l'absence est justifiée, votre chargé de projet peut s'arranger pour faire soutenir l'autre personne un autre jour. Sinon elle se verra seule attribuée la note de 0.

Le rapport

Vous devez rendre le **rapport** dans une archive `rapport.tar.gz`. Le rapport en lui même doit être rédigé dans un fichier `rapport.pdf`. Si vous le souhaitez, vous pouvez ajouter des documents annexes dans un dossier `Annexe`, mais ces documents ne seront pas nécessairement relus par votre chargé de projet.

Le fichier `rapport.pdf` doit avoir une dizaine de pages **maximum** (images comprises). Il n'y a pas de minimum de page autre que celui que votre conscience vous impose. Bref, le rapport ne doit être ni insuffisant ni du remplissage inutile. Petit rappel : un rapport comporte une introduction rappelant rapidement le sujet et présentant rapidement la contribution, une conclusion, une page de garde, un sommaire, ... **Ce rapport se réfèrera uniquement à la partie 2 (Étude commune).**

Les annexes servent à mettre les preuves trop longues, les tableaux trop grands, ... Les annexes ne comptent pas dans le nombre de page du rapport. Dans tous les cas, le rapport doit se suffire à lui même. Par exemple, si vous décidez de placer une preuve en annexe, il peut être bon de laisser une idée de la preuve dans le rapport.

Le rapport sera noté sur le fond et la forme. Le fond regroupe la justesse des preuves, la validité/pertinence des modèles de programmation linéaire et des algorithmes (pourquoi pensez vous que ce sont de bons modèles/algorithmes) et, enfin, la pertinence des évaluations numériques que vous effectuerez. La forme regroupe votre qualité à synthétiser, la qualité des images et des exemples, la clarté des propos, les fautes d'orthographe, ...

Le code

Vous devez rendre le **code** dans une archive nommée `code.tar.gz`.

Vous ne devez pas créer l'archive `code.tar.gz` vous-même. Une commande vous est fournie pour ce faire. Tout est expliqué dans la partie suivante.

Vous devez travailler avec le langage Julia. Il s'agit d'un langage récent, simple à manipuler, et incluant notamment une bibliothèque d'algèbre linéaire et une bibliothèque de modélisation de programmes mathématiques (linéaires ou non). Un tutoriel complet est disponible à `http://dimitri.watel.free.fr/teaching/resources/julia_tutorial.pdf` pour installer et utiliser le langage. Ce langage est facile à installer quelle que soit la plateforme. On utilisera la version `1.2.0`.

Une fois Julia installé, **vous devez installer le paquet Distributions de Julia.** Pour cela, ouvrez Julia en ligne de commande :

```
julia> using Pkg
julia> Pkg.add("Distributions")
```

Solveur de programmation linéaire Pour programmer des modèles de programmation linéaire en Julia, vous n'avez pas besoin d'un solveur. Mais vous en avez besoin pour faire tourner votre programme. Ce dernier ne change pas quel que soit le solveur que vous souhaitez utiliser (à 2-3 détails près). Vous pouvez donc utiliser le solveur de votre choix pour ce projet, il n'influencera pas la correction. Bien entendu, la qualité du solveur fera varier les temps de calculs de vos algorithmes.

Voici 3 solveurs que vous pouvez installer pour le projet. (Vous pouvez en choisir un autre.)

— GLPK : gratuit, libre et très facile à installer ; mais le plus lent de tous ;

— SCIP : gratuit pour les étudiants, moyennement simple à installer, rapide ;

— CPLEX : gratuit pour les étudiants à condition de demander une licence à IBM. Moyennement simple à installer, très rapide.

Conseil : si vous n'avez ni SCIP ni CPLEX d'installé, commencez par GLPK, on verra pour les autres solveurs plus tard.

Comment ranger votre code Votre code doit être commenté et il doit pouvoir être exécuté sans erreur. Il devra pouvoir fonctionner uniquement avec les fichiers présents dans l'archive à l'exception de possibles fichiers d'entrée. Si une de ces conditions n'est pas respectée, la moyenne ne peut être attribuée à ce projet : il vaut mieux ne rien rendre que rendre un code qui ne s'exécute pas ! Testez votre code régulièrement, et faites des sauvegardes au fur et à mesure.

Vous devez télécharger et décompresser l'archive `opti2.tar.gz` créant ainsi normalement un dossier `opti2` qui contient tous les fichiers nécessaires au démarrage du projet.

Cette archive contient de nombreux fichiers. Ceux qui vous intéressent sont :

— Le fichier `algorithm_0.jl` contient un modèle vide de code d'algorithmes.

— Les fichiers `algorithm_example.jl` et `algorithm_lpexample.jl` contiennent des exemples d'algorithmes.

— Un dossier `unit_test` contenant des fichiers `.input`, chacun représentant une instance du problème (voir la section **Fichier d'entrée et de sortie des codes**, page 5), afin de faire des tests unitaires.

— Un fichier `helpers/generate.jl` pour générer des instances aléatoires.

— Un fichier `helpers/tester.jl` pour lancer une batterie de tests.

— Un dossier `customlibs` où vous pourrez mettre des fichiers .jl indirectement liés au projet.

— Un dossier `others` où vous pourrez mettre tout autre fichier (par exemple, des fichiers pour générer des tests).

Deux exemples vous sont donnés dans les fichiers `algorithm_example.jl` et `algorithm_lpexample.jl`. Vous pouvez déjà tester le premier. Pour pouvoir tester le second, qui vous donne un exemple d'utilisation de la bibliothèque `JuMP` de programmation linéaire de Julia, il faudra installer cette bibliothèque avec le solveur GPLK (voir le tutoriel Julia). L'exécution de ce second programme est longue, c'est normal.

```
$ julia algorithm_example.jl example.input
$ julia algorithm_lpexample.jl example.input
```

Tous les algorithmes que vous coderez seront placés dans le dossier racine `opti2`, sous le nom `algorithm_S.py` où S est le nom de votre algorithme. Votre fichier `algorithm_S.py` doit respecter quelques contraintes afin de faciliter votre travail et la correction du chargé de projet. Pour créer un nouvel algorithme, vous êtes invité à suivre la procédure indiquée dans la partie `Coder un nouvel algorithme`, page 4. **Votre code doit posséder au moins 2 algorithmes** : `algorithm_exact1.py` et `algorithm_pl1.py` correspondant à la partie 2 de ce sujet.

Vous aurez peut-être besoin de fichiers .jl pour factoriser votre code, par exemple une bibliothèque décrivant des fonctions ou des modules particuliers. Vous devez placer tous ces fichiers dans le dossier `customlibs` et les importer dans vos algorithmes. Un exemple vous est donné avec la bibliothèque `customlibs/example.jl` importée dans `algorithm_example.jl`.

Tout autre fichier que vous créez et que vous jugez utile de mettre dans l'archive `code.tar.gz` que vous rendrez peut être placé dans le dossier `others`. Par exemple, si vous générez des tests, si vous créez un fichier contenant le résultat des tests, ...

L'archive `code.tar.gz` que vous devez rendre ne doit contenir que :

— vos algorithmes ;

— le dossier customLibs ;

— le dossier others ;

Vous ne devez rien mettre d'autre dans votre archive.

Sous linux et OSX, vous pouvez pas créer l'archive `code.tar.gz` vous-même. Une commande vous est fournie pour ce faire.

```
$ make archive
```

Cette commande ajoute l'archive `code.tar.gz` à la racine de `opti2`. Vérifiez votre archive avant de rendre votre projet.

Notation du code

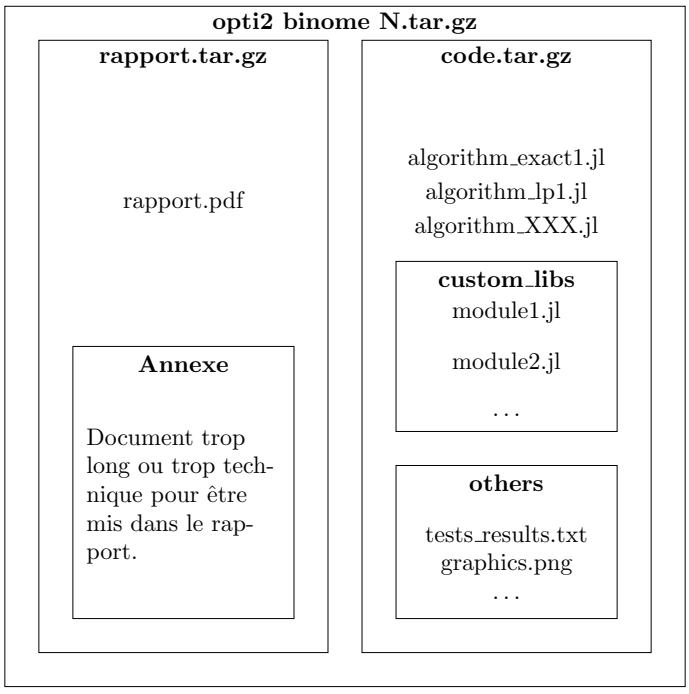
Chacun de vos fichiers algorithmes sera relu puis testé et évalué sur un ensemble d'instances de petite taille. Ces instances ne sont pas exactement celles dont vous disposez dans l'archive `opti2.tar.gz`, mais seront générées de la même façon. Ces tests permettront uniquement de vérifier si les algorithmes répondent correctement aux entrées par rapport à ce qui est indiqué dans le rapport et s'ils sont assez rapides pour résoudre les instances les plus simples.

La soutenance

Vous devez enfin effectuer une **soutenance** lors de la dernière séance. Sa durée est de 15 minutes + 5 minutes de questions. La soutenance n'a pas vocation à répéter oralement le rapport mais à évaluer vos qualités de présentation. Lors des 15 minutes de présentation, vous devez **présenter le sujet et vos résultats de la partie 3 (Développement)**. Remarque : vous êtes deux sur le travail, vous devez être deux à parler et à répondre aux questions pendant la soutenance.

L'archive à renvoyer

Les archives `rapport.tar.gz` et `code.tar.gz` doivent être toutes les deux insérées dans une archive nommée `opti2_binome_N.tar.gz` où N est votre numéro de binôme attribué à la première séance. Cette archive sera rendue seule sur `exam.ensiie.fr`.



Dates importantes

- Soutenance : vendredi 20 décembre (désolé !). Toute absence non justifiée à votre soutenance entraînera la note de 0.
- Rapport et code à rendre au plus tard le vendredi 10 janvier à 23h59. Tout rapport/code non rendu entraînera la note de 0.

1 Problème étudié

Une entreprise possède la surface de la Lune. Il désire louer des parties de cette surfaces pour y poser des annonces publicitaires. Chaque client annonce la quantité de surface qu'il souhaite acheter et le prix qu'il est prêt à payer. Le vendeur doit décider à qui il vend de la surface et quelle surface il lui vend.

On rappelle que la Lune est sphérique mais qu'elle montre toujours la même face à la Terre. Un éclairage spécial a été mis en place pour que la surface soit visible toutes les nuits de l'année. La surface de la Lune est coupée en $h+1$ parallèles (cercles horizontaux, parallèles à l'équateurs) et en $w+1$ méridiens (les demis-cercles reliant le pôle nord et le pôle sud). Autrement dit la lune est coupée en $h \times w$ cases. Ces cases sont arrondies du fait de la courbure de la lune. Chaque case r est associée à un poids $\omega(r)$. Vendre cette case à un annonceur lui coûtera ce prix.

Chaque annonceur a, lorsqu'il propose un contrat, indique 3 valeurs : le nombre de cases minimum qu'il souhaite acheter sous forme d'une largeur $W(a)$ et d'une hauteur $H(a)$ et le montant maximum $M(a)$ qu'il est prêt à payer. L'annonceur indique qu'il souhaite acheter $W(a) \cdot H(a)$ cases de la Lune organisées sous forme d'un rectangle de largeur $W(a)$ et de hauteur $H(a)$. Par exemple, supposons que $W(a) = 1$; que $H(a) = 2$, il faut lui vendre 2 cases, l'une au dessus de l'autre. Ensuite, le montant $M(a)$ indique que l'annonceur ne paiera pas plus cher que $M(a)$; quelles que soient les cases qu'on lui vend. Dans l'exemple précédent, si $M(a) = 100$ et qu'on lui donne deux cases de coûts 30 et 50 ; il paiera 80. Si on lui donne deux cases de coûts 50 et 50 ; il paiera 100. Si on lui donne deux cases de coûts 70 et 60 ; il paiera 100.

Une case ne peut pas être vendue à 2 annonceurs. On n'est pas contraint de vendre toutes les cases de la Lune ; il peut donc y avoir des cases invendues. (Et donc il peut y avoir plus de cases à vendre que demandées par les annonceurs.) Inversement, il n'est pas nécessaire de vendre à tous les annonceurs. (Et donc il peut y avoir plus de cases à demandées par les annonceurs que de cases à vendre.)

L'**objectif** est de maximiser les gains de ventes des cases de la Lune.

1.1 Manipuler les instances et les solutions : fichiers `problem.jl`

Pour vous simplifier la vie, des types ont déjà été définies pour représenter les instances et les solutions du problème. Tous les types et les fonctions sont décrits dans le fichier `helpers/problem.jl`.

Le type `Instance` représente une instance du problème. Vous pouvez appeler plusieurs fonctions et paramètres de l'instance comme le nombre d'annonceurs, le prix de chaque case, ... Le type `Solution` représente une solution du problème. Vous pouvez appeler plusieurs fonctions pour modifier et accéder aux différents paramètres de la solution comme à qui est vendue quelle case, le prix de la solution ...

Pour utiliser ces types et les fonctions associées, dans un fichier julia situé dans le dossier contenant `helper`, ou en ligne de commande depuis ce même dossier, tapez :

```
include("./helpers/problem.jl")
using .Moon
```

Pour avoir de la documentation, dans la ligne de commande Julia, après avoir tapé les 2 commandes précédentes, tapez :

```
?Moon
```

Vous pouvez aussi ouvrir les fichiers pour lire la documentation dedans directement.

Un exemple (partiel) d'utilisation de ces fonctions est donné dans les fichiers `algorithm_example.jl` et `algorithm_lpexample.jl`.

1.2 Coder un nouvel algorithme

Pour créer un nouvel algorithme dont le nom est S, vous devez copier le fichier `algorithm_0.py` en `algorithm_S.py`. Vous pouvez ensuite remplir deux fonctions :

- `run(inst::Instance, sol::Solution)` : le coeur de votre algorithme, qui décide d'une solution à apporter. inst décrit l'instance du problème et sol la solution à calculer. La solution est initialement vide. Cette fonction peut renvoyer une variable, qui sera alors transmise à la fonction `post_process` décrite ci-après.
- `post_process(cpu_time::Float64, inst::Instance, sol::Solution, others)` où vous pouvez afficher des informations dans une fichier ou dans la console. Cette fonction est appelée après `run`. Le paramètre `cpu_time` est le temps de calcul de cette dernière fonction. Les valeurs de `inst` et `sol` sont les mêmes qu'à la sortie de la fonction `run`. Enfin, `others` est ce qui est renvoyé par la fonction `run`. Vous pouvez ainsi effectuer des tests et afficher des résultats sans affecter le temps de calcul.

Pour exécuter votre algorithme, dans un console, tapez

```
$ julia algorithm_S.jl fichier.input
```

où `fichier.input` est un fichier d'entrée, décrit dans la partie ci-après.

1.3 Fichiers d'entrée et de sortie des codes

Votre algorithme prendra en entrée le nom d'un fichier d'entrée `.input` (variable `input_file`). Vous n'aurez pas à manipuler ces fichiers directement, le fichier `main.jl`, automatiquement appelé avec votre algorithme, s'en charge pour vous. Toutefois savoir comment sont organisés ces fichiers peut vous intéresser, entre autres si vous voulez générer vos propres entrées.

Ces deux types de fichiers commencent pas une ligne `# Commentaire`, suivie par une ligne vide. Le commentaire permet d'expliquer par exemple comment le fichier est généré. Il n'est pas possible de le faire sur plus d'une ligne. Dans ces fichiers d'entrée (fichiers `.input`) vous sont donnés toutes les informations de l'entrée :

— une première ligne contenant les nombres w et h.

— h lignes contenant w entiers. Le i^{e} entier de la j^{e} ligne indique le prix $\omega(r)$ de la i^{e} case de la j^{e} ligne de la Lune.

— un ligne contenant le nombre n

— n lignes contenant trois entiers correspondant aux valeurs $W(a), H(a)$ et $M(a)$ pour chaque annonceur a.

2 Étude commune (Cours + Projet + Rapport)

Rappel : la soutenance ne concerne pas cette partie, uniquement le rapport.

2.1 Modélisation (Cours)

Décrivez ce problème de manière formelle. On attend de vous que vous l'écriviez sous forme d'un problème de décision en détaillant l'instance et la sortie. On nommera ce problème (MOON).

2.2 Étude de la complexité (Cours)

On souhaite prouver que ce problème est NP-Complet dans le cas général. Puis définir quelques restrictions et vérifier s'il reste NP-Complet avec ces restrictions.

2.3 Algorithme exact (Projet)

Proposez un algorithme exact pour résoudre une instance de (MOON). Votre algorithme doit être constructif. Ce qui signifie qu'il doit décrire en sortie la solution. Vous décrirez l'algorithme dans le rapport, ainsi que sa complexité. Une preuve de l'exactitude de l'algorithme est un plus.

Inutile de proposer un algorithme compliqué.

Vous coderez ensuite cet algorithme dans un fichier `algorithm_exact1.jl`.

2.4 Programmation linéaire (Cours + Projet)

Proposez un programme linéaire pour modéliser une instance de (MOON). Vous décrirez le programme dans le rapport, ainsi que sa taille. Une preuve de la validité du modèle est un plus.

Vous coderez ensuite un algorithme construisant ce modèle, l'exécutant et déduisant une solution si elle existe dans un fichier `algorithm_pl1.jl`.

Pour manipuler un programme linéaire en Julia, vous êtes invité à utiliser la bibliothèque JuMP. Un exemple d'utilisation de programme linéaire vous est donné dans le fichier `algorithm_lpexample.jl`.

2.5 Évaluation des performances (Cours + Projet)

Proposez une évaluation de vos algorithmes `algorithm_exact1` et `algorithm_pl1`. Vous êtes invités à générer vos propres instances.

— Attention, le dossier `unit_tests` contient des instances de petite taille ou très simples pour un humain pour vous aider à débugger. ⚠ Vous **NE DEVEZ PAS** utiliser ces instances pour faire une évaluation. Ce dossier ne sert que de **TESTS UNITAIRES**.

2.5.1 Générer vos instances

Vous pouvez générer des instances avec le fichier `helpers/generate.jl`. Vous pouvez le modifier si vous le voulez. Pour utiliser les fonctions de ce fichier, vous devez installer le paquet **Distributions**, comme expliqué au début du sujet. Ensuite, dans un fichier julia situé dans le dossier contenant `helper`, ou en ligne de commande depuis ce même dossier, tapez :

```
include("./helpers/generate.jl")
using .Generator
```

Pour avoir de la documentation, dans la ligne de commande Julia, après avoir tapé les 2 commandes précédentes, tapez :

```
?Generator
```

Vous pouvez aussi ouvrir les fichiers pour lire la documentation dedans directement.

2.5.2 Tester de nombreuses instances facilement

Enfin, le script `helpers/tester.jl` vous permets de lancer des batteries de tests facilement. Vous pouvez l'utiliser et le modifier si vous le souhaitez. Le script permet d'exécuter un même algorithme sur l'ensemble des fichier `.input` d'un dossier. Ouvrez le fichier pour lire la documentation.

2 remarques :

— Lorsque vous lancez un programme Julia, il y a toujours un long moment de chargement des bibliothèques (notamment `JuMP`). Ce temps n'est pas inclu dans le temps de calcul de vos algorithmes mais si vous devez charger les bibliothèques à chaque test, il va ralentir fortement vos batteries de tests. Utiliser `helper/tester.jl` vous permet de ne faire ce chargement qu'une seule fois par batterie et de gagner du temps.

— Lorsque vous exécutez un programme Julia qui construit et résout un programme linéaire avec `JuMP`, la bibliothèque doit *chauffer*. Ça signifie que si, dans le même script Julia, vous construisez et résolvez plusieurs programmes linéaires, le premier programme prendra toujours un certain temps d'initialisation (entre 5 et 10 secondes). Ce temps est réduit mais existe toujours dans `helper/tester.jl`. En particulier, si vous exécutez un algorithme contenant un programme linéaire avec `helper/tester.jl`, le temps de calcul de la première instance sera un peu plus long que les autres (entre 0.5 et 1 secondes). Vous devez donc ignorer ce premier résultat. Une technique pour contrer ça consiste à rajouter une fausse instance dans votre dossier et à faire en sorte qu'elle soit exécutée en premier.

Vous devez décrire dans votre rapport l'évaluation de vos algorithmes. Dans un premier temps, et avant de faire les tests, réfléchissez à quels tests vous allez effectuer. Donnez des détails sur vos tests : les conditions du test (descriptif de la machine utilisée), les instances sur lesquels vos tests ont été effectués (en particulier, précisez tous les paramètres de l'instance que vous jugez pertinent pour comprendre les résultats), précisez la sortie obtenue, combien de fois avez vous fait les tests, ...

3 Développement (Projet + Soutenance)

Rappel : le rapport ne concerne pas cette partie, uniquement la soutenance.

Dans cette section choisissez un des développements proposés et ne traiterez que celui ci. Au plus 2 groupes peuvent choisir le même développement. Vous devrez présenter les résultats de cette partie lors d'une soutenance. Notez que, pour les résultats théoriques, votre chargé de projet peut vous aider.

1. **Branch and bound 1.** Codez et évaluez un algorithme exact utilisant la méthode de Branch and Bound pour résoudre (MOON).

   Pour cela, vous pouvez démontrer les résultats suivant :

   (a) On relâche la contrainte d'achat de rectangle pour un annonceur : au lieu de vouloir acheter $W(a) \times H(a)$ cases organisées sous forme d'un rectangle ayant $W(a)$ lignes et $H(a)$ colonnes, il achète **au plus** $W(a) \times H(a)$ cases. On relâche donc 2 contraintes : les cases achetées peuvent être n'importe où, sans forcément les organiser sous forme d'un rectangle ; et on peut vendre moins que $W(a) \times H(a)$ cases à un annonceur. Le prix payé est toujours le même, il paie le minimum entre $M(a)$ et la somme des poids des cases vendues.

   (b) Pourquoi cette version du problème a un gain de vente supérieur à la version originale ?

   (c) Montrer que ce problème est polynomial.

   (d) En déduire une borne utilisable par un algorithme de Branch and bound.

   Testez une exploration en profondeur.

2. **Branch and bound 2.** Codez et évaluez un algorithme exact utilisant la méthode de Branch and Bound pour résoudre (MOON). Pour la borne, utilisez la relaxation continue du programme linéaire de la section 2.

   Testez une exploration en profondeur et par le meilleur d'abord.

3. **Génération de colonne.**

   (a) Codez l'algorithme suivant : Trier les annonceurs par ordre décroissant de $\frac{M(a)}{W(a) \cdot H(a)}$. On note $\mathcal{I}_i$, pour $i \leq n$ l'instance où on a conservé que les i premier annonceurs.

   En utilisant le programme linéaire de la section 2, résoudre tous les programmes $\mathcal{I}_i$.

   (b) Evaluer cet algorithme en mesurant : comment augmente la valeur de la solution optimale quand i augmente ; comment varie le temps de calcul avec i.

4. **Inégalités valides.** On veut tester des inégalités valides de type sac à dos pour le problème.
   (a) On considère une solution réalisable. Soit une ligne $l \in [\![1; h]\!]$ de la grille de cases de la Lune et $A' \subset A$ l'ensemble des annonceurs ayant acheté un rectangle de cases qui contient des cases de la ligne L. Montrer que $\sum_{a \in A'} W(a) \leq w$. Faites de même pour les colonnes.
   (b) Modifier le programme linéaire de la section 2 en rajoutant 2 types variables binaires : $CL(a \in A, l \in [\![1; h]\!])$ et $CC(a \in A, c \in [\![1; w]\!])$. Rajoutez des contraintes pour que $CL(a \in A, l \in [\![1; h]\!])$ soit égal à 1 si et seulement si a achète au moins une case de la ligne l ; et, de même pour les colonnes.
   (c) Montrer que pour toute ligne l et tout $A' \subset A$ tels que $\sum_{a \in A'} W(a) > w$ on a $\sum_{a \in A'} CL(a, l) \leq |A'| - 1$.
   On note $IVL(l, A')$ cette inégalité. De même on note $IVC(c, A')$ l'inégalité relative à une colonne c et $A' \subset A$. Combien existe-t-il de couple (l, A') ? En déduire qu'on ne peut pas ajouter toutes ces inégalités au programme.
   (d) Sur une instance de taille raisonnable ($w = h > 30$ et $n \geq 10$) comparez la solution optimale de la relaxation continue
      — du programme linéaire de la section 2
      — du programme linéaire de la section 2 où on a ajouté toutes les inégalités précédentes.
   (e) Codez et testez un algorithme qui, connaissant une solution de la relaxation continue du programme linéaire de la section 2 cherche et renvoie un couple (l, A') ne vérifiant pas $IVL(l, A')$ ou un couple (c, A') ne vérifiant pas $IVC(c, A')$.
5. **Complexité.**
   (a) Soit k_n un entier fixé. Quelle est la complexité du problème si on se restreint au cas où le nombre n d'annonceurs est fixé égal à k_n ?
   (b) Soit k_w et k_h Quelle est la complexité du problème si on se restreint au cas où w est fixé égal à k_w ? Dans le cas où k_h est fixé égal à k_h ? Et dans le cas où w et h sont tous les deux fixés égaux à k_w et k_h ?
   (c) Quelle est la complexité du problème si $\omega(r)$ et $M(a)$ sont codés en unaires ?
   (d) On considère l'algorithme suivant : tant que c'est possible, trouver le triple (a, l, c) tel qu'il est possible de placer l'annonceur a sur la case (l, c) et qui maximise le gain de cet annonceur ; effectuer le placement et recommencer. Montrer qu'il est possible que cet algorithme renvoie une solution dont la valeur est environ n fois en dessous de celle de la solution optimale.
   (e) Supposons qu'il existe un algorithme capable de placer les annonceurs de sorte à couvrir le plus cases possible de la grille de la Lune, sans tenir compte des poids ou des montant $M(a)$. Montrer qu'il est possible qu'un tel algorithme renverrai une solution dont la valeur est environ n fois en dessous de celle de la solution optimale.
6. **Recuit-simulé** On veut coder un algorithme de recuit simulé pour résoudre le problème (MOON).
   (a) Connaissant une solution S, décrire un algorithme qui permet de modifier faiblement cette solution pour tomber une solution réalisable voisine.
   (b) Un algorithme de recuit fonctionne ainsi : on définie un paramètre de température $0 < T < 1$. On cherche une première solution S. On modifie avec l'algorithme précédent la solution. Si la solution est meilleure, elle remplace S. Sinon, elle replace S avec une probabilité T. On recommence tant que la solution S est remplacée. Quand S n'est pas remplacé, on diminue T. Et on recommence (en partant de cette nouvelle solution S).
   On s'arrête quand $T = 0$. On renvoie la meilleure solution qu'on a trouvé.
   Coder et tester cet algorithme.