Lucas BRIATTE VATEL
Antonin DEPREISSAT
Rémy THIEFFRY
Tri-Man William VU

Optimisation Stochastique

Compte rendu de projet de recherche et développement

Sujet : Détection de fraudes dans les réseaux de grandes tailles

Table des matières

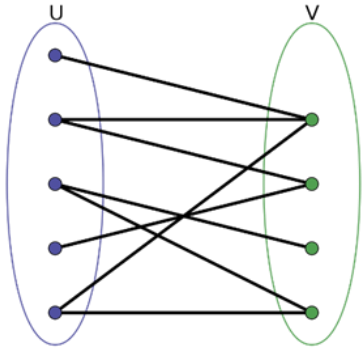
Approche FRAUDAR

Compréhension de l'algorithme

Fraudar a été conçu pour détecter les blocs frauduleux dans un graphe. Autrement dit, de déterminer le sous-graphe le plus suspicieux.

Il faut tout d'abord savoir que dans les études de l'algorithme Fraudar, le graphe sur lequel cet algorithme s'appuie s'agit par exemple, d'un graphe d'utilisateurs et de produits notés par ces utilisateurs.

Leur approche est différente de la nôtre puisqu'elle fonctionne selon un graphe biparti utilisateur-produit :



Les résultats obtenus lors de l'exécution de cet algorithme donnent séparément les nœuds du graphe correspondants aux utilisateurs et ceux correspondants aux produits, l'objectif étant d'identifier les sous-graphes denses qui sont caractéristiques de fraudes.

De plus, l'algorithme Fraudar se repose sur la modification du poids des arêtes pour améliorer leur précision.

Le document étudié sur Fraudar spécifie que les fraudes se résultent par l'ajout de nombreuses arêtes, créant ainsi de larges et denses blocs dans la matrice d'adjacence du graphe.

Les différences notables entre nos implémentations et celle de Fraudar sont que Fraudar est à l'état de l'art. Cet algorithme traite également les fraudes susceptibles d'être camouflées tout en étant scalable. Sa complexité est quasi-linéaire au nombre d'arêtes.

Fraudar garantit de trouver une solution au moins équivalente à une moitié de la solution optimale.

Détails de l'algorithme

Discutons maintenant de l'algorithme en lui-même.

Fraudar instaure une métrique de mesure qui servira à évaluer la densité d'un sous-graphe. L'algorithme cherche donc à maximiser celle-ci. Voici la forme de cette métrique nommée **g** :

$$g(\mathcal{S}) = \frac{f(\mathcal{S})}{|\mathcal{S}|} \qquad \begin{aligned} f(\mathcal{S}) &= f_{\mathcal{V}}(\mathcal{S}) + f_{\mathcal{E}}(\mathcal{S}) \\ &= \sum_{i \in \mathcal{S}} a_i + \sum_{i,j \in \mathcal{S} \wedge (i,j) \in \mathcal{E}} c_{ij} \end{aligned} \qquad a_i \geq 0 \; c_{ij} > 0$$

Ici, $\sum_{i \in \mathcal{S}} a_i$ représente la somme des nœuds du graphe. Cela peut être interprété comme le degré de suspicion d'un utilisateur ou d'un produit. On remarque que dans nos propres études réalisées, nous n'incluons pas cette variable puisque nous n'avons pas considéré le même modèle de graphe, ici étant un graphe biparti utilisateur-produit avec chaque nœud pouvant être lui-même suspicieux du fait que l'utilisateur puisse être un fraudeur ou que le produit puisse être frauduleux.

La variable $\sum_{i,j \in \mathcal{S} \wedge (i,j) \in \mathcal{E}} c_{ij}$ représente la somme des arêtes du graphe. Cela peut être interprété comme le degré de suspicion d'une arête.

L'algorithme Fraudar optimise la métrique de densité **g** par une approche d'un algorithme glouton. En effet, à partir du graphe entier, l'algorithme supprime à chaque itération un nœud parmi ceux restants, pour maximiser **g**. On a ainsi à chaque itération un ensemble de nœuds (un sous-graphe) qui rétrécie à chaque nouvelle itération. A la fin de la boucle d'itérations, l'algorithme renvoie l'ensemble de nœuds (le sous-graphe) qui maximise la métrique de densité **g** parmi tous ceux construits à chaque itération.

De plus, afin de trouver le nœud à supprimer, chaque nœud possède une priorité qui sera modifiée au cours de l'exécution de l'algorithme. Ces priorités sont stockées dans un arbre binaire de priorité. L'intérêt de cet arbre de priorité est de pouvoir trouver rapidement l'élément de priorité maximale, ainsi que de pouvoir mettre à jour rapidement ces priorités.

Leur approche ne traite pas toutes les arêtes e(i,j) de manière égale, mais affecte un poids (degré de suspicion) d'arête en fonction du poids du nœud j correspondant au produit noté par un utilisateur i. Fraudar possède ainsi un système de pondération en fonction du poids d'un nœud.

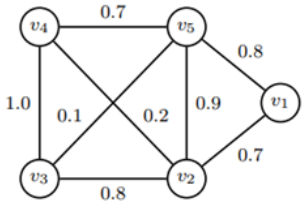
Mais encore, Fraudar optimise leurs résultats en pondérant en fonction des nœuds correspondants au produit. En effet, en considérant la matrice d'adjacence telle que les lignes représentent les utilisateurs et les colonnes représentent les produits, Fraudar effectue une pondération sur les nœuds en réduisant ceux dont la somme des poids de chacun des nœuds représentés par cette colonne dépasse un seuil.

Approche ZOU

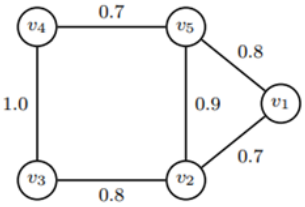
L'algorithme Zou permet de trouver le sous-graphe le plus dense pour un graphe incertain donné.

Il est important de savoir que les propriétés pour un graphe incertain sont différentes de celles pour un graphe qui sera au contraire, dit exact. On va alors ici déterminer la densité prévue pour un graphe incertain.

Pour cela, on définit un graphe incertain par $\hat{G}$ = (V, E, P) tel que chaque arête a une probabilité P(e) d'exister. On considère également que pour un graphe exact G = (V, E), la densité de ce graphe est définie par le ratio du nombre d'arêtes sur le nombre de nœuds : ρ(G) = |E|/|V|



La différence d'un graphe incertain par rapport à un graphe exact s'aperçoit notamment lors du calcul de la densité de ce graphe si on la calcule de la même façon. En effet, pour que les calculs soient pertinents, il serait préférable de considérer le graphe incertain ci-dessus de la manière suivante :



Les arêtes dont la probabilité d'exister est très faible faussent le calcul de la densité. C'est la raison pour laquelle l'algorithme Zou va introduire une nouvelle approche plus adaptée afin de calculer la densité d'un graphe incertain.

Cet algorithme s'appliquant sur un graphe incertain, on peut d'avance exclure les propriétés qui résultent de cette étude sur l'algorithme de Zou en ce qui nous concerne, puisque ne nous traitons pas de graphes incertains dans nos implémentations. Néanmoins, nous continuerons d'expliquer le principe global de cet algorithme dans cette partie.

En pratique, un graphe incertain $\hat{G}$ = (V, E, P) existe comme un graphe exact G = (V, E'). On dit alors que $\hat{G}$ implique G. Il existe un ensemble de graphes G pouvant exister à partir de $\hat{G}$. La probabilité que $\hat{G}$ implique un certain graphe G parmi ceux possibles, c'est-à-dire la probabilité de trouver un graphe exact G à partir du graphe incertain $\hat{G}$ est définie par :

$$\Pr[\mathcal{G} \Rightarrow G] = \prod_{e \in E'} P(e) \cdot \prod_{e \in E \setminus E'} (1 - P(e))$$

Cette probabilité doit respecter la contrainte marginale suivante :

$$\sum_{G=(V,E') \in \Omega(\mathcal{G}), e \in E'} \Pr[\mathcal{G} \Rightarrow G] = P(e)$$

La densité prévue de $\hat{G}$ est donc déduite en fonction de la probabilité de trouver un graphe exact G à partir du graphe incertain $\hat{G}$, et la densité ρ(G) du graphe G en question :

$$\bar{\rho}(\mathcal{G}) = \sum_{G \in \Omega(\mathcal{G})} \rho(G) \Pr[\mathcal{G} \Rightarrow G]$$

L'algorithme va donc chercher à trouver le sous-graphe ayant la densité prévue la plus grande possible.

Le cas considéré ici est le cas où l'on considère un sous-ensemble R ⊆ V et que l'on trouve un sous-graphe ayant la densité prévue maximale, telle que R ⊆ V '. Ce sous-ensemble R est un paramètre de l'algorithme qui induit sur le sous-graphe obtenu. Ici, R = ∅.

Ce problème étudié avec un graphe incertain est équivalent au problème de trouver le sous-graphe le plus dense pour un graphe exact pondéré.

Approche par programmation linéaire déterministe

Dans cette partie, on suppose que le graphe étudié est déterministe. Le programme linéaire est le suivant où la formule (3) est la fonction objectif à maximiser. Les variables de décisions sont les x et y. Les variables $x_{i,j}$ correspondent au poids des arrêtes entre les nœuds i et j. Tandis que les variables y_i correspondent au poids du nœud i.

$$\max \sum_{(ij)\in E} x_{ij} \tag{3}$$

$$x_{i,j} \leq y_i, \forall (i,j) \in E \tag{4}$$

$$x_{i,j} \leq y_j, \forall (i,j) \in E \tag{5}$$

$$\sum_{i\in V} y_i \leq 1, \tag{6}$$

$$x, y \geq 0, \tag{7}$$

L'objectif de cet algorithme est de détecter les cliques d'un graphe donné, c'est-à-dire les sous-graphes de plus fortes densités par rapport au reste du graphe. Ces cliques représentent les zones suspectes du graphe, c'est-à-dire les zones avec où il y a de forte chance qu'il y ait de la fraude.

Pour réaliser cette approche, nous avons utilisé l'API Python de CPLEX proposé par IBM. Cet API nous permet d'adapter directement la fonction objectif ainsi que les contraintes à partir d'un fichier texte représentant les arêtes entre les différents nœuds d'un graphe. Nous avons réalisé cette approche au sein de la fonction *run_linear()* au sein du script *algorithms.py*.

La première partie de ce script consiste à récupérer un graphe entré en paramètre sous format *.txt*. On convertit ensuite ce fichier en un ensemble de variables exploitables. Ensuite, on crée les contraintes et la fonction objectif à partir des variables créées. On exécute le solveur d'optimisation CPLEX. Enfin, on sauvegarde la valeur des variables de décisions au sein du fichier *out/decision_variables_values.out* et on affiche le temps d'exécution ainsi que la valeur de la fonction objectif.

Approche par programmation linéaire stochastique

Dans cette partie, nous étudions les contraintes probabilistes de notre précédente approche qui était une approche par programmation linéaire déterministe.

Soit le programme linéaire déterministe étudié :

$$\max f(x) = \sum_{(ij)\in E} x_{ij}$$

S.T :

$$x_{ij} \leq y_i \text{ , } \forall (ij) \in E$$

$$x_{ij} \leq y_j \text{ , } \forall (ij) \in E$$

$$\sum_{i\in V} y_i \leq 1$$

$$y_i \geq 0 \text{ , } \forall i \in V$$

$$x_{ij} \geq 0 \text{ , } \forall (ij) \in E$$

Avec la méthode des contraintes probabilistes, on a alors la formulation déterministe suivante :

$$\max f(x) = \sum_{(ij)\in E} x_{ij}$$

S.T :

$$x_{ij} \leq y_i \text{ , } \forall (ij) \in E$$

$$x_{ij} \leq y_j \text{ , } \forall (ij) \in E$$

$$P(\sum_{i\in V} y_i \leq 1) \geq p_i$$

$$y_i \geq 0 \text{ , } \forall i \in V$$

$$x_{ij} \geq 0 \text{ , } \forall (ij) \in E$$

avec p_i la probabilité de confiance que l'on fixe à 0,95

On suppose que y_i est une variable aléatoire qui suit une loi Normale $N(\mu,\sigma^2)$.
Calculons alors l'espérance et la variance de f(x) :

$$\mathbb{E}(f(x)) = \mathbb{E}(\sum_{(ij)\in E} x_{ij}) = \sum_{(ij)\in E} x_{ij}$$

$$\mathrm{Var}(f(x)) = \mathbb{E}(f(x)^2) - \mathbb{E}^2(f(x)) = \mathbb{E}((\sum_{(ij)\in E} x_{ij})^2) - (\sum_{(ij)\in E} x_{ij})^2$$
$$= (\sum_{(ij)\in E} x_{ij})^2 - (\sum_{(ij)\in E} x_{ij})^2 = 0$$

On trouve ainsi une nouvelle fonction objective déterministe :

$$F(x) = k1 * \mathbb{E}(f(x)) + k2 * \sqrt{\mathrm{Var}(f(x))}$$

avec les constantes de proportions $k1, k2 \geq 0$

On considère $P(\sum_{i\in V} y_i \leq 1) \geq p_i$ et on pose alors $h_i = \sum_{i\in V} y_i - 1$

On calcule l'espérance et la variance de h_i :

$$\mathbb{E}(h_i) = \mathbb{E}(\sum_{i\in V} y_i - 1) = \sum_{i\in V} \mathbb{E}(y_i) - 1 = \sum_{i\in V} y_i - 1$$

$$\mathrm{Var}(h_i) = \mathbb{E}(h_i^2) - \mathbb{E}^2(h_i) = \mathbb{E}((\sum_{i\in V} y_i - 1)^2) - (\sum_{i\in V} y_i - 1)^2 = (\sum_{i\in V} \mathbb{E}(y_i) - 1)^2 - (\sum_{i\in V} y_i - 1)^2$$
$$= (\sum_{i\in V} y_i - 1)^2 - (\sum_{i\in V} y_i - 1)^2 = 0$$

Comme la loi Normale qu'on considère est centrée et réduite N(0,1), on a alors :

$$P(h_i \leq 0) \geq p_i$$

$$\Leftrightarrow P(\frac{h_i - \mathbb{E}(h_i)}{\sqrt{\mathrm{Var}(h_i)}} \leq \frac{0 - \mathbb{E}(h_i)}{\sqrt{\mathrm{Var}(h_i)}}) \geq p_i$$

$$\Leftrightarrow \phi(\frac{-\mathbb{E}(h_i)}{\sqrt{\mathrm{Var}(h_i)}}) \geq \phi(s) \quad \text{avec } \phi(s) = p_i$$

$$\Leftrightarrow \frac{-\mathbb{E}(h_i)}{\sqrt{\mathrm{Var}(h_i)}}) \geq s$$

avec s = 1,65 d'après la table de la loi Normale centrée réduite N(0,1)

$$\Leftrightarrow \mathbb{E}(h_i) + s * \sqrt{\mathrm{Var}(h_i)} \leq 0$$

On obtient alors la formule déterministe de notre programme linéaire stochastique :

$$\max F(x) = k1 * \mathbb{E}(f(x)) + k2 * \sqrt{\mathrm{Var}(f(x))} = k1 * \sum_{(ij)\in E} x_{ij} + k2 * 0$$
$$= \sum_{(ij)\in E} x_{ij} \quad \text{avec } k1 = 1.00$$

S.T :

$$x_{ij} \leq y_i \text{ , } \forall (ij) \in E$$

$$x_{ij} \leq y_j \text{ , } \forall (ij) \in E$$

$$\mathbb{E}(h_i) + s * \sqrt{\mathrm{Var}(h_i)} \leq 0 \Leftrightarrow (\sum_{i\in V} y_i - 1) + 1{,}65 * 0 \leq 0$$
$$\Leftrightarrow \sum_{i\in V} y_i - 1 \leq 0$$
$$\Leftrightarrow \sum_{i\in V} y_i \leq 1$$

$$y_i \geq 0 \text{ , } \forall i \in V$$

$$x_{ij} \geq 0 \text{ , } \forall (ij) \in E$$

Ainsi, notre approche par programmation linéaire stochastique est identique à notre approche par programmation linéaire déterministe obtenue précédemment.

Description du dataset

Nous avons utilisé un dataset d'Amazon pour réaliser l'étude de détection de fraude. Ce dataset provient de https://snap.stanford.edu/data/amazon0302.html . Nous avons retiré les commentaires

en début de fichier afin que Fraudar puisse être exécuté sans erreur. Nous pouvons voir ci-dessous un extrait du dataset d'Amazon :

```
# Directed graph (each unordered pair of nodes is saved once): Amazon0302.txt
# Amazon product co-purchaisng network from March 02 2003
# Nodes: 262111 Edges: 1234877
# FromNodeId    ToNodeId
0   1
0   2
0   3
0   4
0   5
1   0
1   2
1   4
1   5
```

Ce fichier correspond à un graphe où chaque ligne représente une arête. Pour chacune des lignes, le premier nombre correspond à l'identifiant du nœud d'origine. Tandis que le second correspond au nœud de destination. Sachant que notre graphe est non orienté, chaque arête est en double : "noeud1, nœud2" et "noeud2, noeud1" correspondent à la même arête.

Notre objectif est d'étudier les différents algorithmes de détection de fraudes sur ce dataset.

Implémentation du code

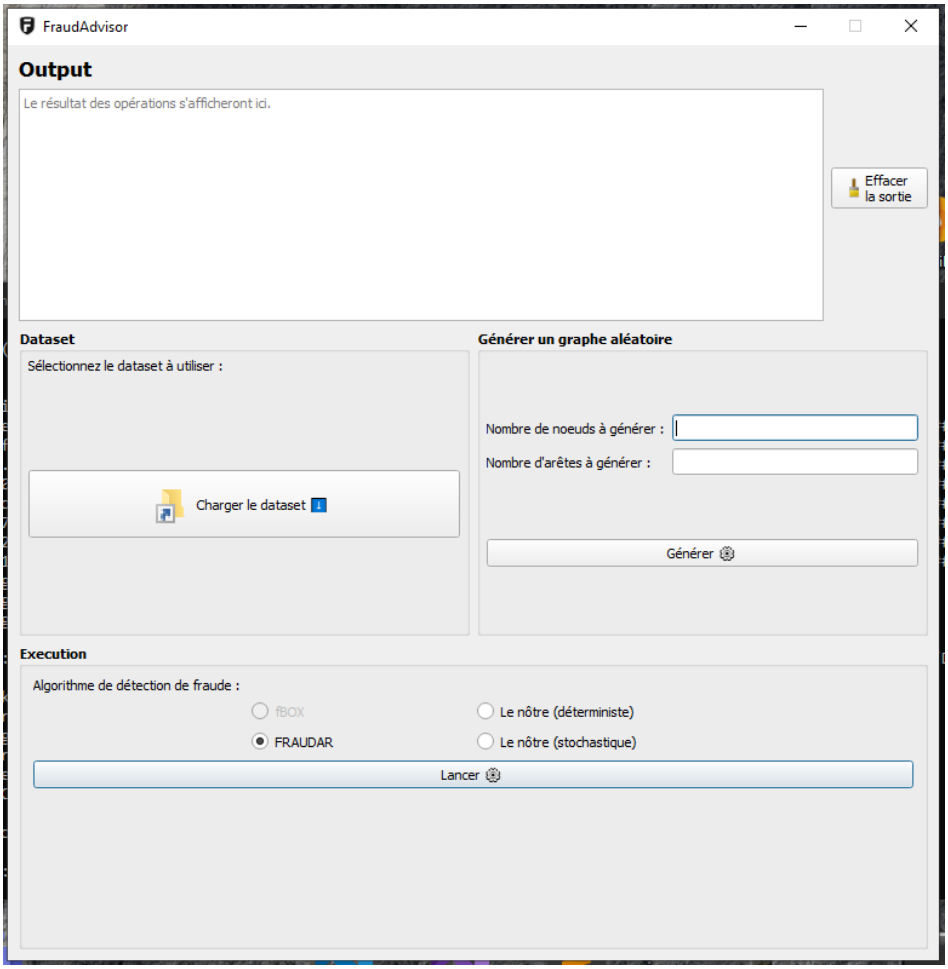
Interface graphique

Pour ouvrir l'interface graphique, il faut exécuter le script python *main.py*. L'interface utilisateur de l'application a été réalisée avec PyQt5. Elle permet d'exécuter l'algorithme désiré à partir d'un dataset fourni. Il est également possible de générer aléatoirement un graphe en fournissant le nombre de nœuds et d'arêtes désirés.

En fonction de ces paramètres, les résultats d'exécution sont affichés dans la zone 'Output' qui se charge de transcrire les sorties de Fraudar ou du modèle mathématique d'optimisation stochastique.

Dans le cas où on génère aléatoirement un graphe, on appelle la fonction *graphGenerator()* présente dans le script *graph_generation.py*, ce qui créé un fichier .txt dans le dossier */dataset*. Tout comme le dataset d'Amazon, chaque ligne du fichier créé correspond à une arête les nombres correspondent aux identifiants des nœuds ce cette arête.

Aperçu de l'interface graphique :

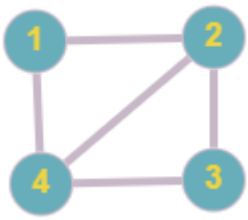


Création de la matrice d'adjacence à partir d'un dataset

Nous sommes capables de lire un dataset au format textuel afin d'en déduire une demi-matrice d'adjacence qui servira par la suite dans les différentes approches linéaires déterministes et stochastiques.

Chaque ligne du dataset correspond à une arrête du graphe. Une ligne est composée des identificateurs entiers séparées par un espace.

Prenons l'exemple du graphe suivant :



Le fichier texte ci-dessous représente une traduction possible du graphe ci-dessus. Chaque ligne correspond à une arête et chaque nombre correspond à l'identifiant d'un nœud de l'arête :

```
# COMMENTAIRE
1   2
1   4
4   1
4   2
4   3
2   1
2   4
2   3
3   2
3   4
```

L'algorithme de lecture du dataset va convertir ce dernier sous la forme d'une matrice d'adjacence :

$$M = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Le graphe que nous analysons est un graphe non directionnel, une arrête d'un nœud a vers un nœud b serait équivalent à une arrête du nœud b vers le nœud a.

De même, nous ne considérerons pas les nœuds ayant une arrête sur eux-mêmes. De ce fait, pour éviter de considérer plusieurs fois la même arrête, nous avons choisis de considérer uniquement la matrice :

$$M_{demi} = \forall i \leq j,\ M_{i,j} = 0$$

Ici, dans notre exemple, on obtient alors :

$$M_{demi} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Génération aléatoire d'un graphe

Nous avons créé une fonction permettant de générer aléatoirement un graphe, cette fonction est *graphGenerator* qui est présente dans le script *graph_generation.py*. Cette fonction dispose de deux arguments, $nodeNumber$ correspondant au nombre de nœuds et $edgeNumber$ correspondant au nombre d'arrêtes souhaitées du graphe.

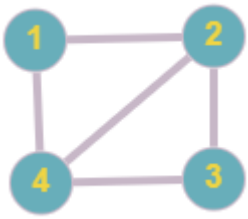
La fonction affecte aléatoirement chaque arrête à un couple de nœud $(a, b)\ avec\ a \neq b$ et affecte le résultat dans une demi-matrice d'adjacence comme définie dans la partie précédente.

Lorsque toutes les arrêtes ont été positionnées aléatoirement, la matrice d'adjacence est lue ligne par ligne et convertie au format texte sous le modèle des datasets.

De cette manière, le fichier ainsi créé est directement exploitable par notre programme et ne nécessite pas de pré-traitement additionnel.

Approche par programmation linéaire

Dans un premier temps, nous avons vérifié que nous obtenions bien les mêmes résultats avec le solveur d'optimisation CPLEX de base et avec son API en Python. Nous avons utilisé le graphe suivant:



Vérification avec l'API Python du solveur d'optimisation CPLEX

Le fichier texte ci-dessous représente une traduction possible du graphe ci-dessus. Chaque ligne correspond à une arête et chaque nombre correspond à l'identifiant d'un nœud de l'arête :

```
# COMMENTAIRE
1   2
1   4
4   1
4   2
4   3
2   1
2   4
2   3
3   2
3   4
```

On entre en paramètre ce dataset : *examples/example.txt*.

Output

```
---------------------------------------------
Vous avez sélectionné le dataset : C:/Users/33750/Desktop/Programmation Stochastique/Projet/FakeAdvisor/examples/example.txt

=============================================

Lancement de l'algorithme determinist...

Le résultat obtenu est 1.25.

Les résultats sont dans le dossier out.
```

En résultat, on obtient une fonction objectif de 1.25 et des variables de décisions à 0.25 (fichier *out/decision_variables_values.out*) :

```
x1_2 = 0.25
x1_4 = 0.25
x2_4 = 0.25
x2_3 = 0.25
x3_4 = 0.25
y1 = 0.25
y2 = 0.25
y3 = 0.25
y4 = 0.25
```

Les valeurs obtenues respectent bien les contraintes fixées. De plus, ces résultats sont cohérents, la clique obtenue correspond au graphe d'origine.

Vérification avec le solveur d'optimisation CPLEX et un fichier .lp

Le fichier .lp suivant est une transcription possible du problème ci-dessus :

```
Maximize
    obj1 : x1 + x2 + x3 + x4 + x5

Subject To
    \ Conditions
    c1 : x1 - x6 <= 0
    c2 : x2 - x6 <= 0
    c3 : x3 - x7 <= 0
    c4 : x4 - x7 <= 0
    c5 : x5 - x8 <= 0
    c6 : x1 - x7 <= 0
    c7 : x2 - x9 <= 0
    c8 : x3 - x8 <= 0
    c9 : x4 - x9 <= 0
    c10 : x5 - x9 <= 0
    c11 : x6 + x7 + x8 + x9 <= 1

Bounds
    x1 >= 0
    x2 >= 0
    x3 >= 0
    x4 >= 0
    x5 >= 0
    x6 >= 0
    x7 >= 0
    x8 >= 0
    x9 >= 0

End
```

Les variables x1 à x5 représentent les variables x du problème. Tandis que les variables x6 à x9 représentent les variables y du problème.

On obtient bien les mêmes résultats qu'avec l'API Python :

```
Objective =  1.2500000000e+00
Display values of which variable(s): x1-x9
Variable Name           Solution Value
x1                            0.250000
x2                            0.250000
x3                            0.250000
x4                            0.250000
x5                            0.250000
x6                            0.250000
x7                            0.250000
x8                            0.250000
x9                            0.250000
```

Comparaison des approches par les résultats

L'algorithme Fraudar et notre algorithme par programmation linéaire ne cherchent pas à maximiser la même fonction. En effet, ces deux approches ont une fonction objectif différente.

Fraudar prend en compte le degré de suspicion des nœuds en plus du degré de suspicion des arêtes, le tout divisé par le nombre de nœuds. Tandis que notre algorithme par programmation linéaire ne prend qu'en compte le degré de suspicion des arêtes.

Formules du calcul de degré de suspicion pour Fraudar :

$$g(\mathcal{S}) = \frac{f(\mathcal{S})}{|\mathcal{S}|} \qquad \begin{aligned} f(\mathcal{S}) &= f_{\mathcal{V}}(\mathcal{S}) + f_{\mathcal{E}}(\mathcal{S}) \\ &= \sum_{i \in \mathcal{S}} a_i + \sum_{i,j \in \mathcal{S} \wedge (i,j) \in \mathcal{E}} c_{ij} \end{aligned} \qquad a_i \geq 0 \; c_{ij} > 0$$

De plus, l'algorithme Fraudar va utiliser cette information telle une métrique de densité, autrement dit, une unité de mesure sur la densité, afin de faire tourner un algorithme glouton créant un ensemble de sous-graphes. Le sous graphe retourné par l'algorithme est celui parmi lequel la métrique de densité **g** mesurée est la plus élevée.

La méthode de maximisation de la valeur objective n'est donc également pas identique.

Formule du calcul de degré de suspicion pour notre algorithme par programmation linéaire :

$$\max \sum_{(ij) \in E} x_{ij} \tag{3}$$

$$x_{i,j} \leq y_i, \forall (i,j) \in E \tag{4}$$

$$x_{i,j} \leq y_j, \forall (i,j) \in E \tag{5}$$

$$\sum_{i \in \mathcal{V}} y_i \leq 1, \tag{6}$$

$$x, y \geq 0, \tag{7}$$

Ainsi, il ne semble pas vraiment qualitatif de comparer les deux valeurs obtenues pour la fonction objectif.

Exécution de Fraudar avec le dataset Amazon

Output

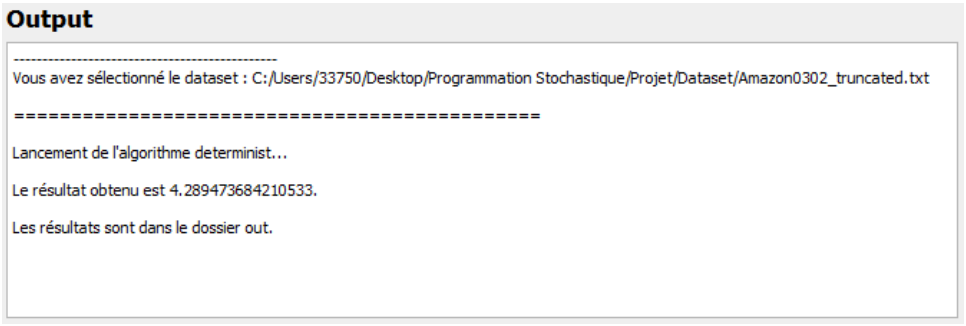
```
---------------------------------------------
Vous avez sélectionné le dataset : C:/Users/33750/Desktop/Programmation Stochastique/Projet/Dataset/Amazon0302.txt
=============================================

Lancement de l'algorithme fraudar...

Le score obtenu est 1.0823084191717245.

Les résultats sont dans le dossier out.
```

L'avantage de Fraudar est qu'il est très rapide, seulement 1 à 2 secondes suffisent pour l'exécuter avec le dataset Amazon. En résultat, nous obtenons un score de 1.08 pour la fonction objectif et respectivement 535 et 547 nœuds au sein des fichiers *out/out.col* et *out/out.rows*. Cela signifie que ces nœuds sont suspects et probablement à l'origine de fraude.

Exécution du solveur d'optimisation CPLEX avec l'API Python sur le dataset Amazon

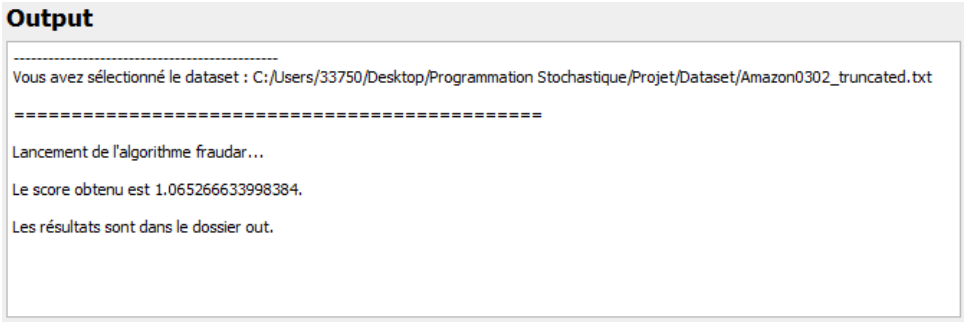
Lorsqu'on exécute notre approche par programmation linéaire sur le dataset d'Amazon, le temps d'exécution est trop long et ne nous ne permet pas d'obtenir des résultats car ce dataset est trop volumineux. En effet, ce dernier possède plus de 250 000 nœuds et plus d'un million d'arêtes. Nous avons étudié les temps d'exécution des différentes parties de notre script et nous nous sommes aperçus que les parties antérieures à la résolution du problème étaient celles qui prenaient le plus de temps. Nous avons donc optimisé la partie de notre script dédiée à la conversion du fichier texte en variables exploitables mais le temps d'exécution est toujours trop long.

En tronquant le dataset d'Amazon à 50 005 arêtes, le temps d'exécution lié à la conversion du fichier texte et à la création des variables du problème est de l'ordre de 2-3 minutes. Tandis que le temps d'exécution lié à la résolution du problème est de l'ordre de 8 secondes. Concernant les résultats avec le dataset tronqué, nous obtenons une fonction d'objectif à 4.29 et des variables aléatoires à valeurs cohérentes (fichier *out/decision_variable_values.txt*). Au total, nous obtenons plus de 600 variables de décision ayant une valeur différente de 0. Parmi ces variables, 489 concernent des poids d'arêtes différents de 0.

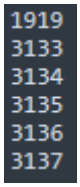


Exécution de Fraudar avec le dataset Amazon tronqué

Afin de comparer les performances de notre algorithme avec Fraudar, nous avons décidé d'éxécuté Fraudar sur le dataset d'Amazon tronqué. Le temps d'exécution de ce dataset tronqué avec Fraudar est de l'ordre de 1.5 s.



Nous obtenons un score de 1.07 pour la fonction objectif avec 6 nœuds au sein des fichiers *out.cols* et out.rows :



En ne prenant en compte que les 50 005 premières arêtes du dataset d'Amazon, on en déduit que Fraudar considère ces 6 nœuds comme suspects et probablement à l'origine de fraude.

Conclusion

Nous avons vu que l'algorithme Fraudar est très rapide, même lorsque le dataset est très volumineux. A la différence de notre algorithme par approche par programmation linéaire qui ne permet pas de trouver une solution avec le dataset Amazon. Cependant, nous pouvons voir que, dans le cas où nous prenons le dataset tronqué, notre algorithme fournit une liste relativement grande de nœuds susceptibles d'être frauduleux. A la différence de Fraudar qui fournit une liste de seulement 6 nœuds lorsqu'on l'exécute sur le dataset tronqué.

Nous décidons donc de comparer Fraudar et notre algorithme en nous basant sur le dataset d'Amazon tronqué. Tout d'abord, nous avons conservé les identifiants des 6 nœuds suspects obtenus avec Fraudar. Puis nous avons analysé la valeur des variables liées à ces nœuds au niveau des résultats obtenus avec notre approche par programmation linéaire, c'est-à-dire le poids des arêtes liées à ces nœuds et les poids de ces nœuds. Une unique variable de décision est différente de 0 : `y3136 = 0.008771929824561403`Cela montre bien que ce nœud est suspect. Cependant, les autres variables associées aux 6 nœuds sont à valeur nulle.

Pour un grand jeu de données, on remarque que l'approche par programmation linéaire trouve un sous-graphe plus grand que celui de l'algorithme Fraudar. Pour un petit jeu de données, les deux algorithmes trouvent chacun un sous-graphe qui a, en général, des nœuds en commun avec celui de l'autre algorithme. On en conclut donc qu'en général, l'approche par Fraudar serait alors plus précise pour déterminer le sous-graphe le plus dense.

Quant à la valeur de la fonction objectif, celle-ci est nettement plus élevée pour notre programme linéaire que pour l'approche Fraudar. Cela s'explique par la différence des fonctions objectifs à maximiser. De plus, notre algorithme ne permet pas d'étudier les datasets volumineux. A la différence de Fraudar qui permet d'étudier des datasets importants dans un intervalle de temps très court.

Limites

Après avoir terminé l'étude de l'approche FRAUDAR, nous avions décidé d'étudier l'approche spectrale FBox. En revanche, sachant que ce code est déployé sous Matlab, nous avons préféré nous focaliser sur l'approche par programmation linéaire. Nous comptions convertir le code Matlab de FBox en Python pour pouvoir l'inclure à notre projet mais nous n'avons pas pu par manque de temps.