Algorithmes Arithmétiques II – TP 3 – Algèbre linéaire creuse

Julien Lavauzelle

16 octobre 2024

L'objectif de ce TP est d'implanter des opérations élémentaires sur des **matrices creuses**, afin d'obtenir (à l'aide des TP précédents) un algorithme de résolution de systèmes linéaires creux à grande échelle, de complexité sous-cubique. Nous étudierons la complexité pratique de ces algorithmes. Par simplicité, nos matrices seront à coefficients dans le corps fini premier $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$, où p est donc un nombre premier quelconque.

Pour ce TP, vous aurez besoin du fichier `sparse_matrix.py`, disponible sur la page web du cours

`https://www.lvzl.fr/teaching/2024-25/aa.html`

— Le fichier `sparse_matrix.py` contient une classe `SparseMatrix`, que vous devrez compléter, et qui permettra de représenter des matrices creuses de manière compacte.

Vous aurez également besoin des fichiers `matrix.py`, `timer.py`, `polynomial.py` et `sequence.py` utilisés lors des TP1 et TP2.

Représentation d'une matrice creuse

Ces premières questions ont pour objetif de vous faire comprendre la structure choisie pour la repréentation d'une matrice creuse via la classe `SparseMatrix`.

Question 1.– Lire attentivement la classe `SparseMatrix`, jusqu'à la méthode `random_matrix(...)` comprise. Bien prêter attention à la structure employée pour stocker les coefficients de la matrice. Puis :

1. créer une matrice creuse aléatoire de taille 10×10, ayant $t = 12$ coefficients non-nuls sur $\mathbb{F}_5$;
2. observer l'affichage particulier des éléments nuls de la matrice ;
3. convertir la matrice creuse créée en une matrice dense, à l'aide de la méthode `to_dense()` ;
4. afficher la version dense de la matrice.

Maintenant que l'on comprend comment une matrice creuse est repréentée en machine, essayons d'implanter une matrice creuse très classique, la matrice identité de taille n. Les seuls éléments non-nuls de cette matrice se trouvent sur sa diagonale, où ils valent tous 1. Le dictionnaire représentant cette matrice doit donc être :

```
{ (i,i): 1 for i in range(n) }
```

Question 2.– Complétez la méthode `identity_matrix(n, mod)` qui doit retourner une version creuse de la matrice identité de taille `n` définie sur les entiers réduits modulo `mod`. Puis, testez votre implémentation et affichez le résultat.

On souhaite maintenant effectuer la somme de deux matrices creuses $\boldsymbol{A}$ et $\boldsymbol{B}$. Pour cela, l'idée est de ne parcourir **que** les coefficients $A_{i,j}$ et $B_{i,j}$ qui sont non-nuls, et de faire les sommes correspondantes. L'Algorithme 1 détaille cette méthode de calcul.

Algorithme 1 : Somme de deux matrices creuses $\boldsymbol{A}$ et $\boldsymbol{B}$.

Entrée : Deux matrices creuses $\boldsymbol{A}$ et $\boldsymbol{B}$.
Sortie : La somme $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$ sous la forme d'une matrice creuse.

```
Créer un dictionnaire vide dico.
Pour tout (i,j) tel que A_{i,j} ≠ 0 faire
    Si B_{i,j} ≠ 0
        Calculer c ← A_{i,j} + B_{i,j}
        Si c ≠ 0
            Placer dans dico[i,j] la valeur c.
    Sinon
        Placer dans dico[i,j] la valeur A_{i,j}.
Pour tout (i,j) tel que B_{i,j} ≠ 0 faire
    Si A_{i,j} = 0
        Placer dans dico[i,j] la valeur B_{i,j}.
Retourner la matrice associée au dictionnaire dico.
```

Question 3.– Complétez la méthode `__add__(self, other)` de la classe, afin de pouvoir retourner la somme de deux matrices creuses sous la forme d'une matrice creuse. Puis, testez votre implémentation.

Question 4.– Donnez une conjecture sur la complexité algorithmique de la somme de deux matrices creuses de taille $n \times n$ comprenant t coefficients non-nuls. Cette complexité sera exprimée en fonction de t et n. Puis, vérifiez expérimentalement votre conjecture en calculant des temps d'exécution moyens.

Question 5.– Mêmes questions pour le produit scalaire :

1. Complétez la méthode `inner_product(self, other)` qui doit retourner le produit scalaire entre les matrices données. Plus explicitement, si $\boldsymbol{A}$ et $\boldsymbol{B}$ représentent ces deux matrices, la méthode doit retourner $\sum_{i,j} A_{i,j}B_{i,j}$.
2. Trouver la complexité du calcul en fonction de t et n.

Remarque : vous pouvez obtenir une fonction donc la complexité est grossièrement en $O(t)$.

La méthode représentant la **multiplication matricielle** est un peu plus difficile à implanter ; elle vous est donc fournie dans le fichier `sparse_matrix.py`.

Question 6.– Testez la multiplication entre :

1. deux matrices creuses carrées de même taille n ;
2. une matrice creuse carrée de taille n et un vecteur de taille n (représenté comme une matrice colonne).

Puis, donnez la complexité de l'opération de multiplication en fonction des grandeurs caractéristiques de ces matrices/vecteurs.

Résolution d'un système linéaire creux

Passons maintenant à l'objectif final de ce chapitre : la résolution de systèmes linéaires de la forme $\boldsymbol{Ax} = \boldsymbol{b}$, où $\boldsymbol{A}$ est une matrice creuse de taille n assez importante. Rappelons qu'en résolvant naïvement le système avec les notions d'algèbre linéaire vues au TP1, le temps d'exécution pour résoudre ce problème est en $O(n^3)$.

Pour faire mieux, nous avons vu en cours différents algorithmes qui, mis bout à bout, nous permettent de résoudre ce problème en temps $O(nt)$, où t est le nombre de coefficients non-nuls dans la matrice $\boldsymbol{A}$.

Ces algorithmes sont :

— *(Algorithme 2)* Le calcul efficace des premiers termes d'une suite récurrente scalaire $\boldsymbol{u}$ dite "itérée". Précisément, cette suite a la forme $u_i = \langle \boldsymbol{y}, \boldsymbol{A}^i\boldsymbol{b} \rangle$, où $\boldsymbol{y}$ est un vecteur non-nul quelconque.

— *(Algorithme 3)* Le calcul efficace du polynôme de connexion de la suite vectorielle itérée $(\boldsymbol{A}^i\boldsymbol{b})_i$. Ce calcul repose en partie sur l'algorithme de Berlekamp–Massey vu en TP2.

— *(Algorithme 4)* La résolution final du système linéaire, qui consiste en le calcul d'un polynôme de connexion et une évaluation rapide de la forme $Q(\boldsymbol{A})\boldsymbol{b}$ par une méthode de Horner (aussi vue lors du TP2).

Question 7.– Implanter l'1lgorithme 2 dans la méthode `iterated_sequence(self, b, x, length)` de la classe `SparseMatrix`. Tester ensuite la validité de l'implantation ; on pourra s'aider des valeurs de test suivantes :

— sur $\mathbb{F}_5$, pour

$$\boldsymbol{A} = \begin{pmatrix} 1 & 3 \\ 2 & 0 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{et} \quad \boldsymbol{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

les 8 premiers termes de la suite sont $(1, 3, 4, 2, 1, 3, 4, 2)$.

— sur $\mathbb{F}_2$, pour

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad \text{et} \quad \boldsymbol{x} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix},$$

les 12 premiers termes de la suite sont $(1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0)$.

Algorithme 2 : Calcul de k termes de la suite itérée $\boldsymbol{u}$ donnée par $u_i = \langle \boldsymbol{x}, \boldsymbol{A}^i\boldsymbol{b} \rangle$, où $\boldsymbol{A}$, $\boldsymbol{x}$ et $\boldsymbol{b}$ sont fixés.

Entrée : Une matrice creuse $\boldsymbol{A}$ et deux vecteurs $\boldsymbol{x}$ et $\boldsymbol{b}$ fixés. Un entier k.
Sortie : Les k premiers termes de la suite $\boldsymbol{u}$ définie par $u_i = \langle \boldsymbol{x}, \boldsymbol{A}^i\boldsymbol{b} \rangle$.

1 Stocker dans `tmp` une copie de `b`.
2 Initialiser une liste `u` vide.
3 **Pour tout** $i = 0, \ldots, k-1$ **faire**
4 Ajouter à `u` le produit scalaire (calculé efficacement) entre `x` et `tmp`.
5 Mettre à jour `tmp` avec le produit (calculé efficacement) entre $\boldsymbol{A}$ et `tmp`.
6 **Retourner** la liste `u`.

Question 8.– Dans la méthode `connexion_polynomial_iterated_sequence(self, b)` de la classe `SparseMatrix`, implanter l'Algorithme 3. Tester ensuite la validité de l'implantation ; on pourra s'aider des valeurs de test suivantes :

— sur $\mathbb{F}_5$, pour

$$\boldsymbol{A} = \begin{pmatrix} 1 & 3 \\ 2 & 0 \end{pmatrix} \quad \text{et} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

le polynôme de connexion est $P(x) = 1 + 4x + 4x^2$.

— sur $\mathbb{F}_2$, pour

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad \text{et} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

le polynôme de connexion est $P(x) = 1 + x^2 + x^3 + x^4$.

Algorithme 3 : Calcul du polynôme de connexion de la suite itérée $(\boldsymbol{A}^i\boldsymbol{b})_i$.

Entrée : Une matrice creuse $\boldsymbol{A}$ **inversible** et un vecteur $\boldsymbol{b}$.
Sortie : Le polynôme de connexion minimal de la suite $(\boldsymbol{A}^i\boldsymbol{b})$.

Si $\boldsymbol{b} = \boldsymbol{0}$
  Retourner le polynôme 1
Définir $P(X) \leftarrow 1$ et $\boldsymbol{b}' \leftarrow \boldsymbol{b}$.
Tant que $\boldsymbol{b}' \neq 0$ **faire**
  Faire
    Tirer $\boldsymbol{x}$ un vecteur aléatoire.
    Calculer $k = 2(n - \deg(P)) + 2$ termes de la suite itérée $\boldsymbol{u}$ définie comme $u_i = \langle \boldsymbol{x}, \boldsymbol{A}^i\boldsymbol{b} \rangle$.
  tant que $\boldsymbol{u} = \boldsymbol{0}$;
  Calculer $Q(X)$, le polynôme de connexion du LFSR minimal qui engendre $\boldsymbol{u}$.
  Mettre à jour $P(X) \leftarrow P(X) \times Q^{\perp}(X)$, où $Q^{\perp}(X)$ est le réciproque de $Q(X)$.
  Mettre à jour $\boldsymbol{b}' \leftarrow \boldsymbol{A} \cdot \boldsymbol{b}'$
Retourner le polynôme P.

Algorithme 4 : Calcul d'une solution $\boldsymbol{x}$ du système creux $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$.

Entrée : Une matrice creuse $\boldsymbol{A}$ **inversible** et un vecteur $\boldsymbol{b}$.
Sortie : Une solution $\boldsymbol{x}$ au système creux $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$.

Calculer $P(X)$, le polynôme de connexion de la suite itérée $(\boldsymbol{A}^i\boldsymbol{b})_i$.
Calculer $Q(X) = -\frac{P(X) - P(0)}{P(0)X}$.
Retourner $Q(\boldsymbol{A})\boldsymbol{b}$.

Question 9.– Implanter l'Algorithme 4 dans la méthode `solve_right(self, b)` de la classe `SparseMatrix`. Tester ensuite la validité de l'implantation ; on pourra s'aider des valeurs de test suivantes :

— sur $\mathbb{F}_5$, pour

$$\boldsymbol{A} = \begin{pmatrix} 1 & 3 \\ 2 & 0 \end{pmatrix} \quad \text{et} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

la solution est

$$\boldsymbol{x} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}.$$

— sur $\mathbb{F}_2$, pour

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad \text{et} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

la solution est

$$\boldsymbol{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Avec la même matrice $\boldsymbol{A}$, mais en considérant le vecteur

$$\boldsymbol{b}' = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

à la place de $\boldsymbol{b}$, la solution devient

$$\boldsymbol{x} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

Analyse de complexité

Une méthode statique `somewhat_random_invertible_matrix(n, mod, t)` vous permet de générer de grandes matrices creuses et inversible, tirées "aléatoirement"[1], qui sont (1) particulièrement difficiles à résoudre avec de l'algèbre linéaire dense, et (2) ressemblantes à des matrices qui apparaissent dans les algorithmes de calcul d'indice ou de crible quadratique (que nous verrons en fin d'année pour la factorisation d'entiers ou le logarithme discret).

Pour cette méthode, la valeur de t correspond au nombre de coefficients non-nuls **par ligne**, pour la plupart des lignes de la matrice creuse.

1. attention : la distribution est loin d'être uniforme

Question 10.– Exécutez l'appel

```
A = SparseMatrix.somewhat_random_invertible_matrix(20, 2, 3)
```

Puis,

1. vérifiez que la matrice $\boldsymbol{A}$ possède $20 - 3 = 17$ lignes comprenant 3 coefficients non-nuls, et 3 lignes comprenant un seul coefficient non-nul ;
2. vérifiez également que la matrice $\boldsymbol{A}$ est inversible ; pour cela, on peut vérifier, après transformation en matrice dense, que la matrice échelonnée associée à $\boldsymbol{A}$ n'a pas de coefficient non-nul sur sa diagonale.

Question 11.– Créez un vecteur $\boldsymbol{x}$ aléatoire de la forme suivante :

```
x = SparseMatrix.random_matrix(20, 1, 2, 10)
```

puis construisez le vecteur $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}$. Résolvez ensuite l'équation $\boldsymbol{A}\boldsymbol{y} = \boldsymbol{b}$ d'inconnue le vecteur $\boldsymbol{y}$, et vérifiez que le vecteur $\boldsymbol{y}$ obtenu est bien solution de l'équation.

Question 12.– En s'inspirant des questions précédentes, comparez les temps d'exécution de la résolution de systèmes linéaires creux et dense, pour des matrices $\boldsymbol{A}$ issues de la méthode `somewhat_random_invertible_matrix`, avec les paramètres suivants :

— $n = 100$ et $t = 10$, sur $\mathbb{F}_2$;
— $n = 800$ et $t = 10$, sur $\mathbb{F}_2$;
— $n = 800$ et $t = 100$, sur $\mathbb{F}_2$;
— si c'est raisonnable sur votre machine, $n = 3200$ et $t = 10$, sur $\mathbb{F}_2$ (pour la résolution "dense", cela devrait être très, très long).

Question 13.– Difficile. Vérifiez expérimentalement que la complexité algorithmique de la résolution de systèmes linéaires inversibles creux est en $O(nt)$, où t est le nombre de coefficients non-nuls dans la matrice du système de taille n.