Sorbonne Université

Projet STL
Extraction de motifs graduels fermés fréquents sous contrainte de temporalité

Composition du trinôme :
Muhannad Alabdullah
Lea El Abboud
Esma Hocini

Encadrants :
Mehdi Naima
Lionel Tabourier

12 mai 2024

Table des matières

1 Introduction générale

1.1 Présentation du projet

Pendant leurs opérations quotidiennes, de nombreuses entreprises commerciales collectent une quantité considérable d'informations pour étudier les comportements de leurs clients. Les supermarchés, en particulier, accumulent des données sur les achats en évaluant le contenu des paniers des consommateurs. Cette démarche vise non seulement à améliorer les stocks et à optimiser la disposition des produits en rayon mais également de développer un système de recommandations pour les boutiques en ligne.

Par exemple, si une analyse des paniers de clients révèle que les couches et les bières sont souvent achetées ensemble, cela peut guider les supermarchés dans l'agencement de leurs produits.

Dans le cadre de notre projet, nous visons ainsi à développer des algorithmes efficaces pour l'extraction de motifs graduels fermés fréquents.
En fouille de données, un motif fréquent désigne un ensemble d'éléments d'un jeu de données qui apparaissent en même temps de façon cohérente et dont le nombre d'occurrences (le support) dépasse un seuil donné. Un motif graduel identifie des relations telles que "plus/moins X, plus/moins Y", capturant la manière dont les variables évoluent ensemble dans le temps. Un motif fermé, quant à lui, est un groupe d'éléments qui apparaissent fréquemment ensemble dans les données et qui ne peut pas être agrandie sans réduire sa fréquence d'apparition.

Ce travail, réalisé en langage C, nécessitera la mise en place de structures de données pour traiter efficacement les ensembles de données de grande taille et complexité. Notre démarche sera structurée en plusieurs étapes clés. Tout d'abord, nous définirons les concepts et termes essentiels pour établir une base solide de compréhension. Puis, nous aborderons le développement des algorithmes Apriori et Apriori TID, tout en discutant de leur complexité et des structures de données ainsi que des fonctions utilisées. Enfin, nous procéderons à une étude expérimentale, générant des données artificielles et traçant des courbes expérimentales, afin de comparer l'efficacité de ces algorithmes.

1.2 Définitions

Avant de détailler notre approche, il est primordial de se familiariser avec certaines définitions fondamentales. Nos exemples seront issus de la base de données $\mathcal{D}$ suivante :

TID	Items
10	1 2 5
20	1 2
30	3 4 6
40	1 2 3 5
50	3 6

TABLE 1.1 – Base de données $\mathcal{D}$ des transactions : la colonne de gauche affiche les identifiants de transactions et celle de droite liste les items contenus dans chaque transaction.

Définition 1.2.1. *Un item fait référence à tout objet appartenant à un ensemble fini d'éléments distincts* $\mathcal{I} = \{x_1, x_2, \ldots, x_m\}$.

Les articles en vente dans un magasin sont des items. Dans la base $\mathcal{D}$, les élements de $\mathcal{I} = \{1, 2, 3, 4, 5, 6\}$ sont des items.

Définition 1.2.2. *Un ensemble d'éléments de* $\mathcal{I}$ *est désigné comme un itemset. Plus précisément, un itemset composé de* k *éléments est nommé un* k*-itemset.*

Une combinaison d'items. Un exemple de $\mathcal{D}$ est {1, 2, 5}, qui est un 3-itemset de la transaction TID 40.

Définition 1.2.3. *Une transaction est un ensemble de données identifié par un identificateur unique Tid.*

Un ensemble d'items achetés ensemble. Un panier au supermarché. Par exemple, l'ensemble d'items {1, 2} avec TID 20 est une transaction.

Définition 1.2.4. *La fréquence (Support) d'un itemset désigne le nombre de fois que cet itemset apparaît dans un ensemble de transactions donnée. Nous noterons* $s(i)$ *pour la fréquence de* i *dans la base de donnée* $\mathcal{D}$.

Si nous prenons l'item 1 dans notre base de données $\mathcal{D}$, il a un support de 3 car il apparaît dans trois transactions : TID 10, 20 et 40.

Définition 1.2.5. *Le support minimum* S *représente la fréquence minimale requise pour qu'un itemset soit considéré comme fréquent.*

Si le support minimum était de 2 dans notre base de données $\mathcal{D}$, alors l'itemset {1, 2}, apparaissant dans les transactions TID 10 et 40, serait considéré comme fréquent.

Définition 1.2.6. *Un itemset est dit fréquent si son support est au-delà du support minimum fixé à priori.*

Dans $\mathcal{D}$, en prenant un support minimum de 2, l'item 1 serait considéré comme faisant partie d'un itemset fréquent car il apparaît dans trois transactions.

Définition 1.2.7. *Un itemset est considéré comme un motif fermé s'il n'existe aucun sur-ensemble ayant le même support que cet itemset.*

Considérons l'itemset {1, 2}, qui apparaît dans les transactions TID 10, TID 20, et TID 40 de $\mathcal{D}$, lui accordant un support de 3. Aucun sur-ensemble de {1, 2} (par exemple, {1, 2, 5} ou {1, 2, 3}) n'apparaît dans exactement ces trois transactions, ce qui fait que leur support est inférieur à 3. Par conséquent, {1, 2} est un motif fermé car il n'existe aucun sur-ensemble ayant le même support de 3, faisant de {1, 2} le plus grand ensemble avec ce niveau de support parmi ses extensions dans la base de données $\mathcal{D}$.

2 Algorithme d'extraction des itemsets fréquents

2.1 Algorithme Apriori

Pour résoudre ce type de problème, nous définissons une base de données $\mathcal{D}$ constituée de transactions, avec un support minimal S. Notre objectif est d'identifier et de lister tous les motifs fréquents fermés de $\mathcal{D}$. Nous nous appuyons pour cela sur l'algorithme Apriori proposé par Agrawal et Srikant en 1994 [1]. Il est reconnu pour sa capacité à identifier des propriétés qui reviennent fréquemment dans un ensemble de données.

2.1.1 Principe de base

Dans le contexte de l'extraction des itemsets fréquents, la propriété d'antimonotonie joue un rôle essentiel dans l'efficacité de la recherche.

Remarque 2.1.1. Pour tout itemset I qui est un sous-ensemble de J (noté $I \subseteq J$), $s(I) \geq s(J)$, indépendamment de la base de données $\mathcal{D}$ [2].

L'approche naïve consiste à explorer tous les itemsets possibles pour déterminer ceux qui sont fréquents, un processus souvent inefficace en raison du nombre élevé de combinaisons à évaluer. En revanche, l'approche d'Apriori, en exploitant la propriété d'antimonotonie, permet une exploration itérative en étendant progressivement les motifs actuels.

En effet, lorsqu'un itemset est étendu en ajoutant un autre élément, son support ne peut augmenter. Cela constitue la base de la propriété Apriori, qui précise que si $s(I) < S$, alors tout sur-ensemble J de I sera également peu fréquent, $s(J) < S$. Par conséquent, on peut réduire l'espace de recherche de manière significative. Cette stratégie de réduction est connue sous le nom d'élagage basé sur le support. Prenons la Figure 2.1. Par exemple, si l'itemset {c, d, e} est fréquent, toute transaction le contenant inclura également ses sous-ensembles {c, d}, {c, e}, et {d, e} comme le montre la Figure 2.2. En revanche, pour un itemset comme a, b qui n'est pas fréquent, tous ses sur-ensembles seront également non fréquents, permettant ainsi l'élagage de l'intégralité du sous-graphe qui les contient, comme l'indique la Figure 2.3.

Notation 2.1.1. Soient

L_k, l'ensemble des k-itemsets fréquents de taille k.

C_k, l'ensemble de candidats k-itemsets. $C_k \subset L_k$. Un candidat désigne un itemset potentiellement fréquent que l'algorithme évalue pour déterminer si son support atteint ou dépasse le support minimal.

L'algorithme Apriori implémente une approche itérative pour trouver les itemsets fréquents. Tout d'abord, il détermine le support des 1-itemsets en effectuant une première passe sur la base de données. Grâce à la propriété d'antimonotonie, l'algorithme élimine les itemsets non fréquents à chaque étape. Par la suite, un nouvel ensemble des 2-itemset, désignés comme les itemsets candidats, est formé. Après un autre passage sur la base de données, les supports de ces 2-itemset candidats sont calculés. Ceux non fréquents sont rejetés. Le processus décrit précédemment est maintenu jusqu'à ce que l'on ne puisse plus générer les itemsets fréquents.

Dans l'algorithme principale, il existe une fonction `Apriori-Gen` qui prend L_{k-1} comme paramètre. Celle-ci est utilisée pour générer des candidats, produisant ainsi un ensemble de tous les k-itemsets candidats. Cette

opération se déroule en trois étapes : l'extraction de tous les éléments possibles de l'ensemble L_{k-1}, puis parcourir cet ensemble d'éléments extraits et L_{k-1} et construire toutes les combinaisons possibles de taille k en essayant d'ajouter à chaque itération, un élément extrait à un ensemble de taille $k-1$ dans L_{k-1}. Ensuite, lors de l'étape d'élagage, tous les itemsets c dans C_k tels qu'il existe un sous-ensemble de c de taille *k-1* qui n'est pas dans L_{k-1} sont supprimés.

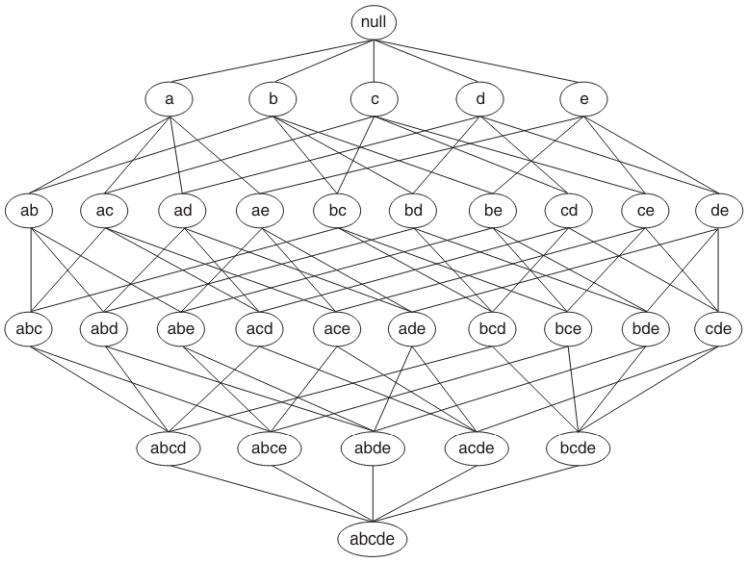

FIGURE 2.1 – Diagramme de Hasse d'inclusion sur un ensemble de N éléments. Chaque arêtes montre qu'un ensemble est un sous-ensemble direct de l'autre, organisés par niveaux croissants de taille de sous-ensemble, du plus petit (en haut) au plus grand (en bas) [3].

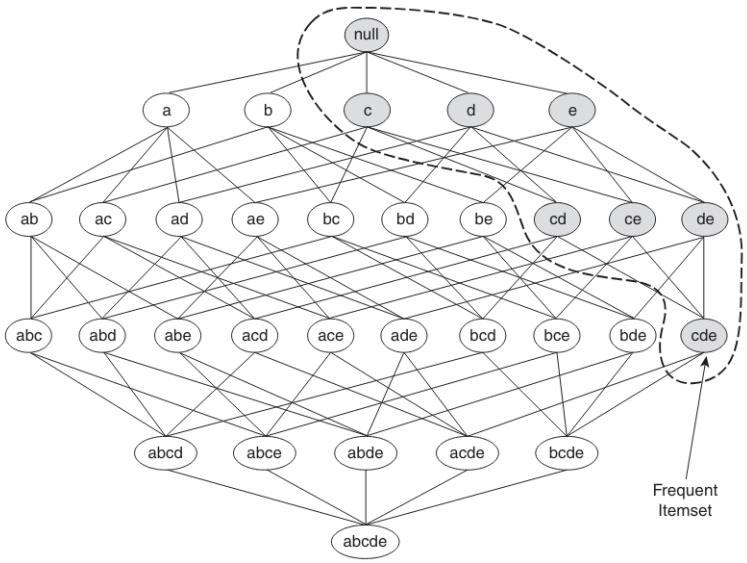

FIGURE 2.2 – Illustration du principe d'Apriori. Si {c,d,e} est fréquent, alors tout sous-ensembles de cet itemset est fréquent [3].

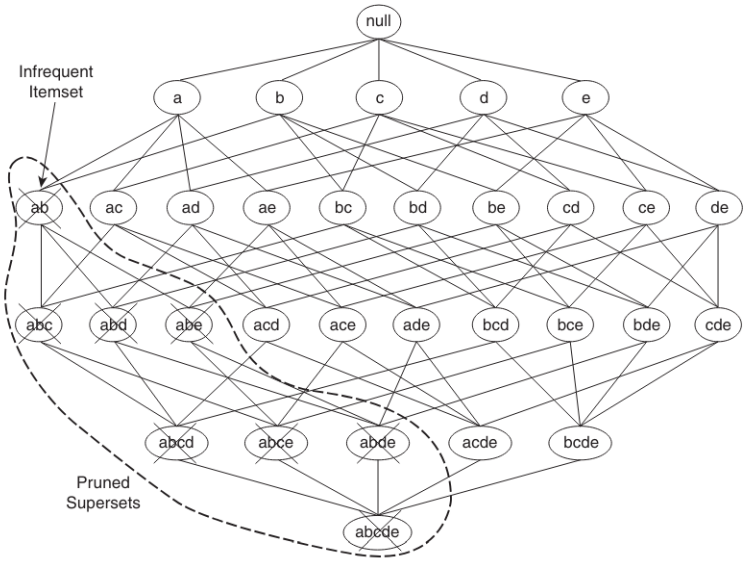

FIGURE 2.3 – Illustration de l'élagage basé sur le support minimum. Si {a,b} n'est pas fréquent, alors tout les sur-ensembles de cet itemset ne sont pas fréquents [3].

2.1.2 Pseudo-code

Algorithm 1 Algorithme Apriori

function ALGORITHME_APRIORI($ficher, support, colonne\_id\_article, colonne\_id\_panier, taille$)
  $res \leftarrow$ `construire_l1`$(f, support, colonne\_id\_article, taille)$
  $lk\_1 \leftarrow$ `construire_l1`$(f, support, colonne\_id\_article, taille)$
  $fin \leftarrow$ vrai
  while fin **do**
    $fin \leftarrow$ faux
    $ck \leftarrow$ `Apriori_gen`(lk_1)
    `libérer_table_hachage`(lk_1)
    $lk\_1 \leftarrow$ `table_de_hachage`()
    $t \leftarrow$ `obtenir_items_du_fichier`$(f, colonne\_id\_article, taille)$
    $indexes \leftarrow$ `construire_c1`$(t, *taille)$
    $fichier \leftarrow$ `ouvrir`$(ficher)$
    while `lire_ligne_suivante`$(fichier)$ **do**
      `traiter_ligne`$(ligne, indexes, ck, colonne\_id\_panier, bitmap)$
    end while
    `mettre_a_jour`$(ck, support, lk\_1, res)$
  end while
  return res
end function

Cet algorithme est une implémentation simplifiée de l'algorithme Apriori pour l'exploration des motifs fréquents dans les ensembles de données transactionnels. Voici une description de ce qui se passe dans cet algorithme :

— **Initialisation :** L'algorithme commence par construire deux tables de hachage : res et lk_1. Ces tables de hachage sont utilisées pour stocker les ensembles d'articles fréquents découverts jusqu'à présent. La variable taille représente la taille de la table de hachage.

— **Boucle Principale :** L'algorithme itère jusqu'à ce qu'il ne soit plus possible de générer de nouveaux ensembles d'articles fréquents.

— **Génération de nouveaux candidats :** À chaque itération de la boucle principale, l'algorithme génère de nouveaux ensembles candidats ck à partir des ensembles fréquents précédents lk_1. Cela est fait en utilisant une fonction appelée Apriori_gen.

— **Traitement des transactions :** Ensuite, l'algorithme parcourt le fichier de transactions ligne par ligne. Pour chaque transaction, il traite la ligne en identifiant les articles présents dans la transaction et met à

jour la table de hachage ck en conséquence. Cela implique de vérifier quels ensembles candidats de ck sont présents dans la transaction et d'incrémenter leur compteur.

— **Mise à jour des ensembles fréquents :** Une fois toutes les transactions traitées, l'algorithme met à jour les ensembles fréquents lk_1 en se basant sur les ensembles candidats ck et en appliquant un seuil de support.

— **Fin de l'itération :** Si de nouveaux ensembles fréquents ont été découverts lors de cette itération, la variable fin reste à false, indiquant que l'algorithme doit continuer. Sinon, la variable fin est définie à true, ce qui signifie que l'algorithme a terminé et peut renvoyer les ensembles fréquents finaux.

Algorithm 2 Apriori-Gen , fonction pour la construction de *ck*

```
1: taille_element_possible ← 0
2: elements_possibles ← obtenir_elements_de_table(lk_1, &taille_element_possible)
3: ck ← table_de_hachage()
4: for i de 0 à TAILLE_TABLEAU do
5:     courant ← lk_1 → pointeurs[i]
6:     while courant n'est pas NULL do
7:         for j de 0 à taille_element_possible do
8:             candidat_possible ← copier_elements(courant → itemset)
9:             ajouter_element_a_itemset(elements_possibles[j], candidat_possible)
10:            if candidat_possible → k ≠ courant → itemset → k then
11:                if tous les itemsets sont dans la table de hachage then
12:                    ajouter_itemset(candidat_possible, ck)
13:                end if
14:            end if
15:        end for
16:        courant ← courant → suivant
17:    end while
18: end for
19: return ck
```

Voici la description de chaque fonction auxiliaire utilisée :

— **Obtenir éléments de table**(*lk_ 1*, *&taille_ element_ possible*) : Cette fonction récupère les éléments de la table de hachage *lk_ 1* et met à jour la variable *taille_ element_ possible* avec le nombre d'éléments possibles.

— **Table de hachage**() : Cette fonction initialise une nouvelle table de hachage et la retourne. Elle est utilisée pour créer la table de hachage *ck*.

— **Copier éléments**(*courant → itemset*) : Cette fonction effectue une copie de l'ensemble d'éléments de l'élément courant et retourne cette copie. Cela garantit que les manipulations ultérieures n'affectent pas les données originales.

— **Ajouter élément à itemset**(*elements_ possibles[j], candidat_ possible*) : Cette fonction ajoute un élément à l'ensemble d'éléments *elements_ possibles[j]*. Elle est utilisée pour générer de nouveaux candidats en ajoutant un élément à l'ensemble d'éléments actuel.

— **Ajouter itemset**(*candidat_ possible, ck*) : Cette fonction ajoute l'ensemble d'éléments candidat *candidat_ possible* à la table de hachage *ck*. Elle vérifie également si tous les itemsets sont déjà présents dans la table de hachage avant d'ajouter le candidat. Cela garantit que seuls les itemsets qui respectent les critères de support minimum sont ajoutés à *ck*.

2.1.3 Exemple

Pour décrire les étapes successives de l'algorithme, nous exécuterons Apriori sur l'exemple de la Figure 1.1 en choisissant un support minimum $S = 2$.

Étape 1.

Nous commençons par une première passe sur la base de données pour identifier les candidats C_1, qui sont les 1-itemsets, et calculons leur support. Les items {1}, {2}, {3}, {5}, et {6} se qualifient comme itemsets fréquents formant l'ensemble L_1, puisqu'ils apparaissent au moins deux fois. Nous éliminons l'item {4}, car son support est inférieur à S.

C_1	
Itemsets	Support
{1}	3
{2}	3
{3}	3
{4}	1
{5}	2
{6}	2

L_1	
Itemsets	Support
{1}	3
{2}	3
{3}	2
{5}	2
{6}	2

TABLE 2.1 – Premier passage sur la base de données pour trouver des ensembles d'éléments uniques

Étape 2.
Avec L_1 établi, nous faisons une deuxième passe pour générer C_2, les 2-itemsets candidats, et calculons à nouveau le support. De cette étape, les paires {1 2}, {1 5}, {2 5}, et {3 6} s'avèrent fréquentes, constituant ainsi l'ensemble L_2.

C_2	
Itemsets	Support
{1 2}	3
{1 3}	1
{1 5}	2
{1 6}	0
{2 3}	1
{2 5}	2
{2 6}	0
{3 5}	1
{3 6}	2
{5 6}	0

L_2	
Itemsets	Support
{1 2}	3
{1 5}	2
{2 5}	2
{3 6}	2

TABLE 2.2 – Deuxième parcours pour trouver des ensembles d'éléments de taille 2

Étape 3.
En progressant vers C_3, les 3-itemsets candidats, une troisième passe sur la base de données révèle que seul l'itemset {1 2 5} a un support qui satisfait S, ce qui nous donne l'ensemble L_3.

C_3	
Itemsets	Support
{1 2 5}	2

L_3	
Itemsets	Support
{1 2 5}	2

TABLE 2.3 – Troisième parcours pour trouver des ensembles d'éléments de taille 3

Puisqu'aucun itemset de taille supérieure à trois ne peut être formé avec un support suffisant, le processus itératif s'achève, laissant comme résultat final l'union des ensembles L_1 : {{1}, {2}, {3}, {5}, {6}}, L_2 : {{1 2}, {1 5}, {2 5}, {3 6}}, et L_3 : {{1 2 5}}.

Ainsi, le résultat final obtenu est le suivant :
{{1}, {2}, {3}, {5}, {6}, {1 2}, {1 5}, {2 5}, {3 6}, {1 2 5}}

2.2 Les structures de données

Pour optimiser l'extraction de motifs fréquents à partir de grandes bases de données transactionnelles, nous avons conçu des structures de données spécifiques pour soutenir les algorithmes que nous avons implémentés.

— **Structure des Itemsets (`item_set`)**

La structure `item_set` est utilisée pour représenter les itemsets identifiés lors du processus de fouille de données. Elle contient un tableau dynamique pour stocker les items de cet ensemble, leur nombre (k) et un compteur d'occurrences (*count*) utilisé pour enregistrer le nombre d'apparition de cet itemset dans l'ensemble de données.

— **Table de hachage**

Afin de gérer efficacement les grandes quantités d'itemsets et de faciliter les opérations fréquentes, nous avons intégré une structure de table de hachage, une composante clé pour notre algorithme.

La table de hachage consiste à contenir des itemsets, alors elle représente les ensembles de type L_k et C_k.

— *Structure et fonctionnement de la table de hachage*

La table de hachage est composée d'un tableau de pointeurs `pointers`, où chaque entrée pointe vers un `hash_node`, une structure de donnée qui sert de conteneur pour les `item_set`, offrant ainsi un accès aux itemsets et leur données associées.

— *Fonctions de hachage*

Dans notre algorithme, il y a deux fonctions de hachage essentielles, une fonction qui donne une valeur de hachage à un élément `hash_item_id(int id)`, et une autre fonction qui donne une valeur de hachage à un itemset `hash_item_set(item_set s)`. Soit `val(item i)` une fonction qui prend un élément en paramètre et qui renvoie un entier v qui correspond à cet élément, la fonction `hash_item_id(id)` renvoie id mod 100 et la fonction `hash_item_set(s)` renvoie $\left(\sum_{i \in s} \texttt{val(i)}\right)$ mod 100. Notez que dans notre implémentation nous considérons que les éléments sont par défaut représentés par des entiers.

— *Gestion des collisions*

La gestion des collisions est un aspect important de la performance de la table de hachage. Lors de l'insertion d'un nouvel itemset, la fonction de hachage calcule un indice qui détermine où le `hash_node` correspondant devrait être placé dans le tableau de pointeurs. L'indice calculé sera également conservé dans le `hash_node` pour une récupération efficace. En cas de collision, quand plusieurs itemsets sont attribués le même indice de hachage, les `hash_node` sont reliés en liste chaînée grâce au pointeur `next`. Chaque nœud ajouté est connecté au suivant, permettant ainsi de résoudre les collisions tout en préservant l'accès rapide aux itemsets.

— **Avantages de la table de hachage**

L'utilisation de la table de hachage offre une amélioration des performances de notre algorithme, surtout lorsqu'il est appliqué à de grandes bases de données transactionnelles. Cette structure de données facilite les opérations fréquentes, comme les insertions et les recherches, grace à sa complexité temporelle moyenne qui est constante $\mathcal{O}(1)$.

2.3 Fonctions et Opérations

Plusieurs opérations sont implémentées afin de manipuler les itemsets dans l'algorithme. Les fonctions clés incluent l'initialisation d'itemsets en créant un nouvel itemset vide (`init_item_set`), l'ajout d'éléments à un itemset existant tout en assurant qu'il n'y a pas de doublons (`add_item_to_item_set`), et la vérification de la présence d'un item dans un itemset (`item_in_item_set`) ou dans la table de hachage (`item_set_in_hash_table`)

Opération	Complexité
`init_item_set`	$\mathcal{O}(1)$
`add_item_to_item_set`	$\mathcal{O}(k)$
`item_in_item_set`	$\mathcal{O}(k)$
`item_set_in_hash_table`	$\mathcal{O}(1)$

TABLE 2.4 – Complexité des opérations, k désigne le nombre d'élément dans l'itemset.

2.4 Complexité

Bien que théoriquement la complexité de l'algorithme Apriori puisse atteindre une forme exponentielle dans le pire des cas, son application pratique sur des bases transactionnelles réelles est souvent bien plus gérable, car le pire des cas est très éloigné de la réalité.

2.5 Algorithme AprioriTID

N.B. : L'algorithme AprioriTID est une optimisation de l'algorithme Apriori standard. Son implémentation est en cours de développement et sera donc détaillée dans une version prochaine de ce rapport.

2.5.1 Principe de base

Afin d'optimiser les performances de l'algorithme standard, les mêmes auteurs ont proposé l'algorithme AprioriTID. Basé sur un principe identique à celui d'Apriori, dans AprioriTID à partir de la deuxième passe, la base de données n'est plus utilisée pour calculer les supports des itemsets candidats. Un ensemble $\overline{C_k}$ composé des identifiants de transaction (TID) et des k-itemsets candidats présents dans cette transaction est utilisé. Pour $k = 1$, C_1 correspond à la base de transaction D. Par rapport à l'algorithme Apriori, le principal avantage de AprioriTID réside dans sa capacité à mémoriser les identifiants de transactions qui contiennent les itemsets fréquents au sein de l'ensemble $\overline{C_k}$, permettant ainsi de réduire le nombre de parcours de la base D.

2.5.2 Pseudo-code

Notation 2.5.1. $\overline{C_k}$. Contient des paires de l'identifiant de transaction et du k-itemset candidat présent dans cette transaction, c'est à dire (Tid, C_k)

Algorithm 3 AprioriTID

1: $L_1 \leftarrow \{1\text{-itemsets fréquents}\}$
2: $C_1 \leftarrow$ Base de donnée D;
3: **for** $(k \leftarrow 2; L_{k-1} \neq \emptyset; k++)$ **do**
4: $C_k \leftarrow$ Apriori-Gen(L_{k-1}); // Nouveaux candidats
5: $\overline{C_k} \leftarrow \emptyset$;
6: **for all** entrée $t \in \overline{C}_{k-1}$ **do**
7: $\overline{C}_t \leftarrow \{c \in C_k | (c - c[k]) \in t.\text{set-of-itemsets} \wedge$
8: $(c - c[k-1]) \in t.\text{set-of-itemsets}\}$;
9: **for all** candidats $c \in \overline{C}_t$ **do**
10: $c.\text{count}++$;
11: **if** $(\overline{C}_t \neq \emptyset)$ **then**
12: $\overline{C}_k \leftarrow \overline{C}_k \cup \{t.TID, \overline{C}_t\}$;
13: **end if**
14: **end for**
15: **end for**
16: $L_k \leftarrow \{c \in \overline{C}_k | c.\text{count} \geq \text{S}\}$
17: **end for**
18: Answer $\leftarrow \bigcup_k L_k$;

2.5.3 Étapes de génération avec Apriori TID

Étape 1 (Génération de L_1)

— Nous utilisons les ensembles d'articles uniques de la base de données pour générer L_1.

— Par exemple, les ensembles {10}, {20}, {30} et {40} sont tous des articles uniques qui apparaissent dans la base de données.

Étape 2 (Génération de C_2 à partir de L_1)

— Nous utilisons les ensembles d'articles uniques dans L_1 pour former des paires uniques d'articles, ce qui génère C_2.

— Par exemple, à partir de L_1, nous formons des paires comme $\{10, 20\}$, $\{10, 30\}$, $\{10, 40\}$, $\{20, 30\}$, $\{20, 40\}$ et $\{30, 40\}$ pour former C_2.

Étape 3 (Génération de L_2 à partir de C_2)

— Nous utilisons les candidats de paires d'articles dans C_2 pour rechercher leur occurrence dans la base de données et calculer leur support.

— Par exemple, nous examinons chaque transaction pour voir si elle contient les paires d'articles dans $\mathcal{C}_2$ et comptons leur occurrence. Les paires qui dépassent le seuil minimum de support deviennent les ensembles d'articles fréquents dans $\mathcal{L}_2$.

Étapes suivantes (Génération de C_k à partir de L_{k-1} et de L_k à partir de C_k)

— À chaque étape, nous générons C_k à partir de L_{k-1} en formant des ensembles candidats d'articles.

— Ensuite, nous utilisons C_k pour générer $\bar{C}_k$ en recherchant les occurrences de ces ensembles candidats dans la base de données.

— $\bar{C}_k$ est défini comme l'ensemble des éléments c de C_k tels que $(c - c[k])(c - c[k])$ appartient à l'ensemble de k-ensembles et $(c - c[k-1])(c - c[k-1])$ appartient à l'ensemble d'ensembles.

— Nous utilisons ensuite $\bar{C}_k$ pour générer L_k en calculant le support de chaque ensemble candidat et en le comparant au support minimum. Seuls les ensembles candidats qui dépassent le seuil minimum de support deviennent des ensembles d'articles fréquents dans L_k.

— En résumé, $\bar{C}_k$ est utilisé pour représenter les ensembles d'articles candidats à chaque étape, ce qui nous permet de générer efficacement les ensembles d'articles fréquents correspondants L_k dans la base de données transactionnelle.

Arrivés à la fin de notre exemple, nous aurons identifié tous les ensembles d'articles fréquents dans les transactions fournies. Dans ce jeu de données, il existe un ensemble d'articles fréquents comprenant plus de 3 articles, qui est {10, 20, 30, 40}. Cependant, celui-ci ne dépasse pas le support défini, qui est de 2.

Base de données

TID	Items
100	10 20 30
200	10 20 30 40
300	20 30 40
400	10 30 40
500	20 30
600	10 20
700	10 30
800	10 40

$\overline{C}_1$

TID	Set-of-Itemsets
100	{ {10}, {20}, {30} }
200	{ {10}, {20}, {30}, {40} }
300	{ {20}, {30}, {40} }
400	{ {10}, {30}, {40} }
500	{ {20}, {30} }
600	{ {10}, {20} }
700	{ {10}, {30} }
800	{ {10}, {40} }

L_1

Itemset	Support
{10}	5
{20}	5
{30}	6
{40}	4

$\overline{C}_2$

C_2

Itemset
{10, 20}
{10, 30}
{10, 40}
{20, 30}
{20, 40}
{30, 40}

TID	Set-of-Itemsets
100	{ {10, 20}, {10, 30}, {20, 30} }
200	{ {10, 20}, {10, 30}, {10, 40}, {20, 30}, {20, 40}, {30, 40} }
300	{ {20, 30}, {20, 40}, {30, 40} }
400	{ {10, 30}, {10, 40}, {30, 40} }
500	{ {20, 30} }
600	{ {10, 20} }
700	{ {10, 30} }
800	{ {10, 40} }

L_2

Itemset	Support
{10, 20}	3
{10, 30}	4
{10, 40}	3
{20, 30}	4
{20, 40}	2
{30, 40}	3

$\overline{C}_3$

C_3

Itemset
{10, 20, 30}
{10, 20, 40}
{10, 30, 40}
{20, 30, 40}

TID	Set-of-Itemsets
100	{ {10, 20, 30} }
200	{ {10, 20, 30} , {10, 20, 40}, {10, 30, 40}, {20, 30, 40} }
300	{ {20, 30, 40} }
400	{ {10, 30, 40} }

L_3

Itemset	Support
{10, 20, 30}	2
{10, 30, 40}	2
{20, 30, 40}	2

C_4

Itemset
{10, 20, 30, 40}

$\overline{C}_4$

TID	Set-of-Itemsets
200	{ {10, 20, 30, 40} }

L_4

Itemset	Support
$\emptyset$	$\emptyset$

FIGURE 2.4 – Application d'AprioriTID sur une base de données

3 Étude expérimentale

Pour une évaluation approfondie du temps de calcul de l'algorithme Apriori, l'analyse de courbes générées à partir d'un ensemble de données crée aléatoirement se révèle particulièrement utile.

3.1 Paramètres

Définissons d'abord quelques paramètres utiles pour notre étude :

Symbole	**Description**
N	Nombre totale d'items
$\|D\|$	Nombre de transactions
$\|T\|$	Taille moyenne des transactions

TABLE 3.1 – Paramètres

3.2 Génération de données artificielles

Dans le cadre de notre projet, la génération de données artificiels de transactions est essentielle pour analyser la performance de notre algorithme Apriori sur un large ensemble de données. Cette approche est réalisée par la fonction `generate_baskets_randomly(D,N,P(i))` avec $P(i)$ la probabilité d'appartenance d'un item i dans une transaction.

Pour chaque ensemble de données généré, nous avons appliqué l'algorithme Apriori pour identifier les motifs fréquents. Le temps d'exécution de l'algorithme pour chaque ensemble a été enregistré.

Probabilité d'appartenance des items La probabilité $P(i)$ joue un rôle essentiel dans la détermination de l'absence ou de la présence d'un item dans une transaction donnée. La fonction `generate_0_or_1` décide, en fonction d'une probabilité donnée, si un item spécifique est inclus (représenté par 1) ou non (représenté par 0) dans une transaction, en générant un nombre aléatoire et en le comparant à cette probabilité. Cette logique est appliquée à travers la fonction `generate_baskets_randomly`.

Fichier obtenu. Ces données synthétiques sont enregistrés dans un fichier CSV "DataRand.csv" facilitant le traitement ultérieur. Chaque ligne du fichier représente un item présent dans une transaction, identifié par un identifiant unique (`Transaction ID`).

3.3 Temps d'exécution

Pour évaluer la performance de l'algorithme Apriori, nous adoptons la méthode suivante : un seul paramètre de notre générateur de données est varié à la fois tout en fixant les autres. Cela nous aide à comprendre comment chaque paramètre affecte la performance de l'algorithme.

3.3.1 Variation de la probabilité

En variant le paramètre de probabilité, nous pouvons simuler des scénarios de transactions plus ou moins denses, ce qui nous permet de tester l'efficacité de l'algorithme Apriori.

Paramètres utilisées :

— Support minimum = 80
— $|D| = 50$
— $|N| = 200$

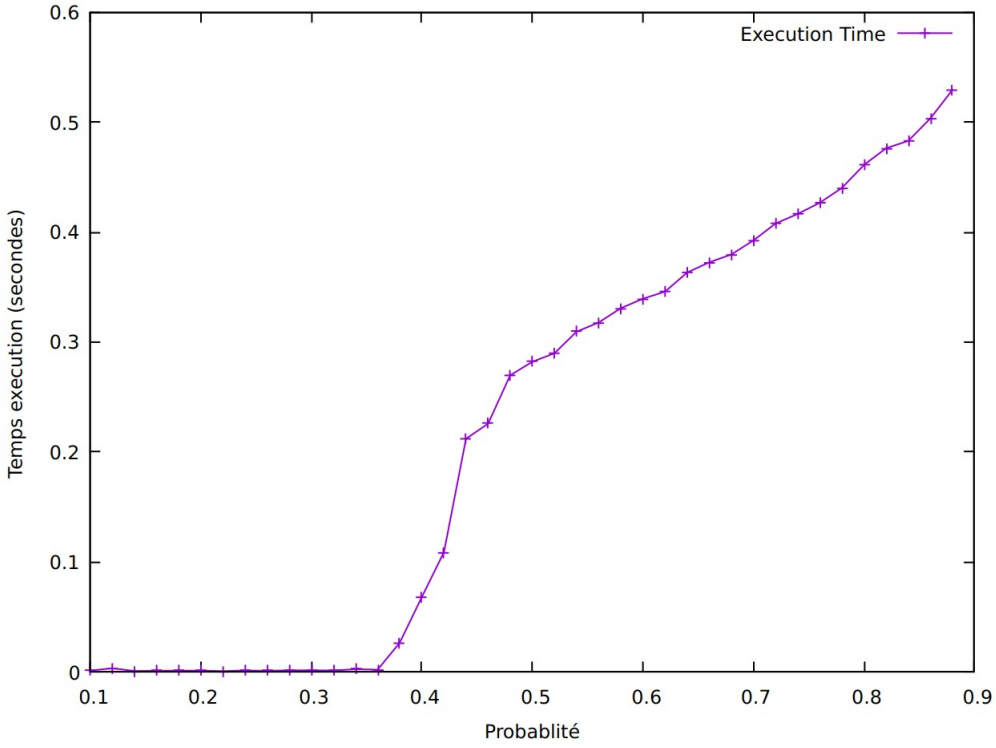

FIGURE 3.1 – Variation du temps d'exécution en secondes de l'algorithme Apriori en fonction de la probabilité d'apparition des items.

Nous pouvons observer que le temps d'exécution varie significativement avec $P(i)$. Une probabilité plus élevée entraîne des transactions contenant un plus grand nombre d'items. Autrement dit, si les items sont très souvent présents dans les transactions, l'algorithme doit examiner beaucoup plus de combinaisons possibles, ce qui augmente le temps nécessaire pour que l'algorithme Apriori exécute son processus de fouille de données.

3.3.2 Variation du support minimum

Le support minimum constitue un critère important pour l'efficacité de l'algorithme Apriori. Dans cette section, nous étudierons l'influence de la variation de ce paramètre sur l'exécution de l'algorithme.

Paramètres utilisées :

— $P(i) = 0.9$
— $|D| = 50$
— $|N| = 200$

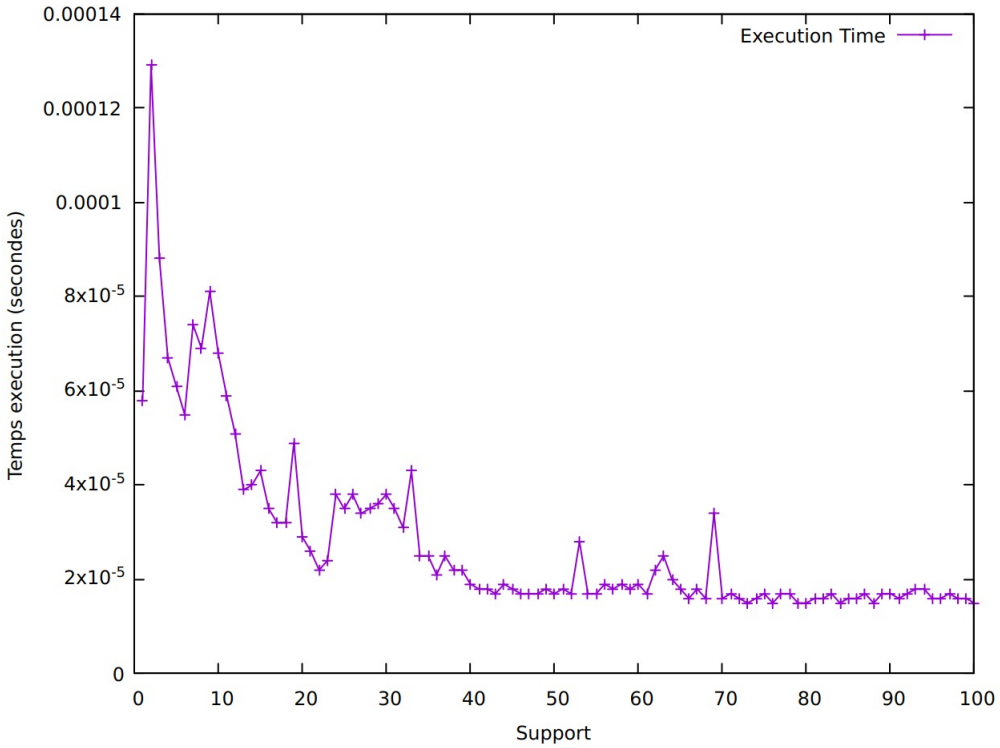

FIGURE 3.2 – Variation du temps d'exécution en secondes de l'algorithme Apriori en fonction du support minimum.

En analysant la courbe, nous remarquons que le temps d'exécution est élevé pour les supports minimaux très faibles et diminue rapidement à mesure que le support minimum augmente, jusqu'à un certain point où le temps se stabilise. Cela est du au fait qu'un support minimum plus élevé réduit le nombre d'ensembles d'items à considérer, car seuls les ensembles d'items dépassant le support minimum spécifié sont pris en compte.

3.3.3 Variation du nombre d'items

Paramètres utilisées :

— Support minimum = 80
— $P(i) = 0.9$
— $|D| = 50$

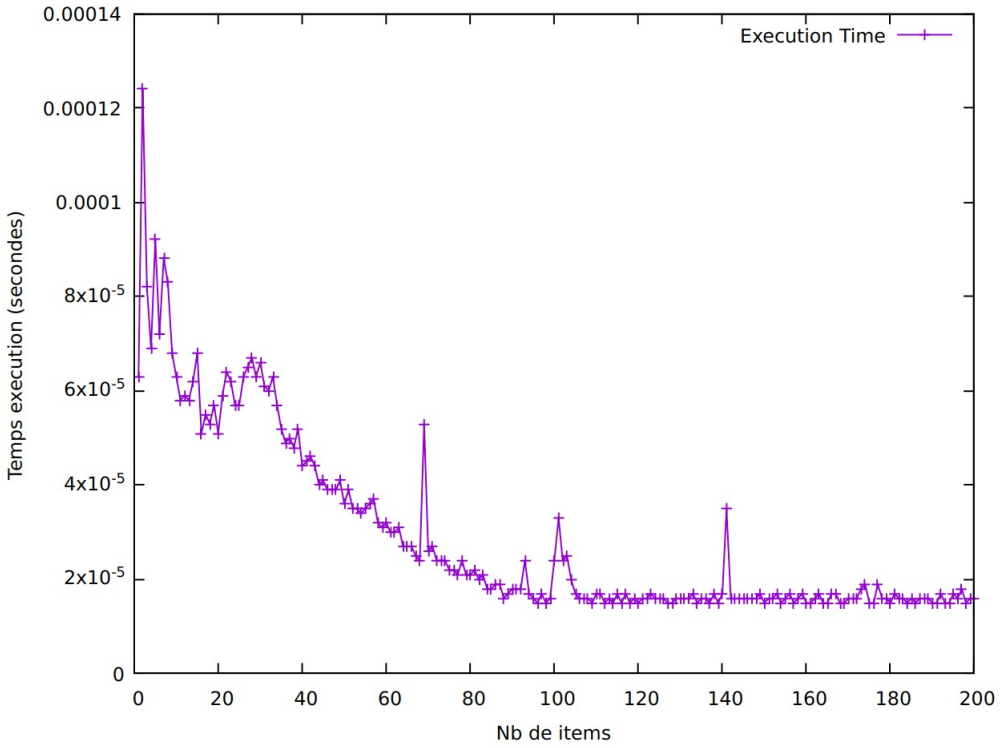

FIGURE 3.3 – Variation du temps d'exécution en secondes de l'algorithme Apriori en fonction du nombre d'items.

Ce graphe illustre le temps d'exécution de l'algorithme Apriori en fonction du nombre d'items, allant de 0 à 200. Il montre un pic très élevé pour les premiers items, avant que le temps ne diminue rapidement et se stabilise malgré quelques fluctuations.

3.3.4 Variation du nombre de transactions

Paramètres utilisées :

— Support minimum = 80

— $P(i) = 0.9$

— $|N| = 200$

Les données présentées ci-dessous sont le résultat de la moyenne de 100 tests effectués pour chaque nombre de transactions représenté sur l'axe des x.

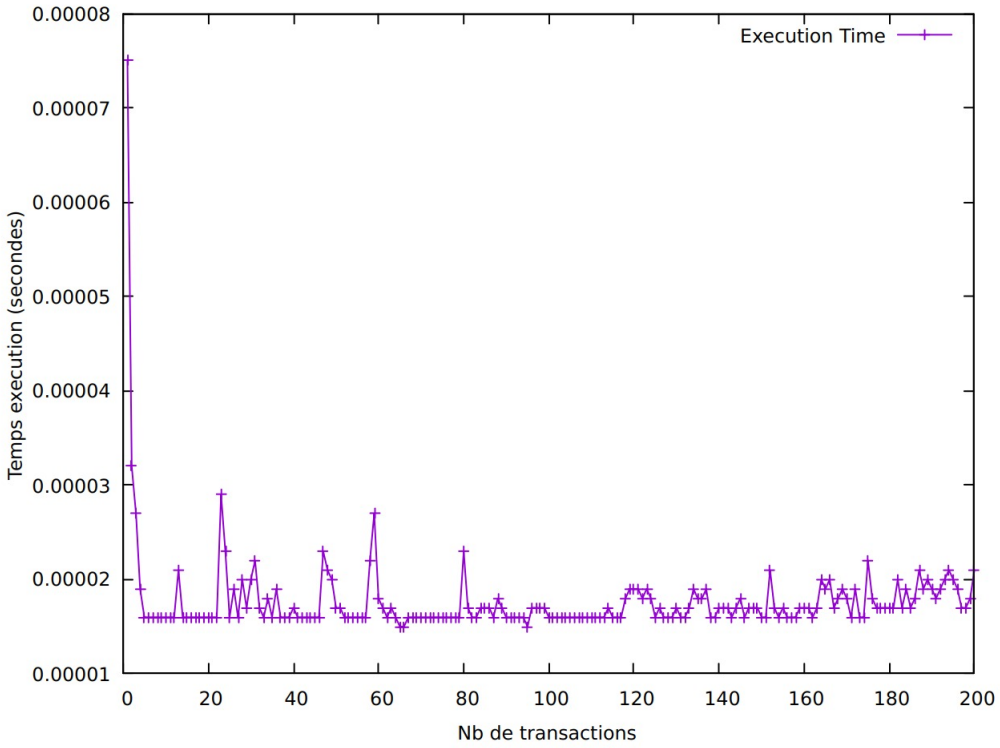

FIGURE 3.4 – Variation du temps d'exécution en secondes de l'algorithme Apriori en fonction du nombre de transaction.

Nous pouvons remarquer que ce graphe commence par un pic de temps d'exécution extrêmement élevé pour un petit nombre de transactions, qui diminue rapidement et devient plus stable avec des fluctuations mineures à mesure que le nombre de transactions augmente.

3.4 Perspectives

Le modèle de génération de données aléatoires est considéré comme irréaliste car il ignore deux aspects importants : d'abord, il suppose que la probabilité qu'un article soit présent dans une transaction est indépendante de celle des autres, ce qui est contraire à la réalité commerciale où les articles sont souvent achetés en fonction de leurs associations dues aux préférences des clients ou d'autre raisons. Ensuite, ce modèle échoue à refléter le fait que la probabilité d'achat varie d'un article à l'autre, étant influencée par la popularité de l'article, et d'autres facteurs. Ainsi, ces simplifications empêchent le modèle de refléter fidèlement la complexité des données transactionnelles dans le monde réel.

4 Bilan et conclusion

Pour conclure, la première phase de notre étude s'est concentrée sur l'application de l'algorithme Apriori, la génération d'un jeu de données artificiel, et l'analyse du temps d'exécution à travers divers graphiques. Cependant, la mise en œuvre de l'algorithme AprioriTID lors de la seconde phase a rencontré des difficultés inattendues, laissant cette partie inachevée. Parmi les autres tâches prévues lors ce projet, figure l'utilisation le forage de données pour détecter des séquences temporelles dans de vastes ensembles de données, une méthode fréquemment employée en data mining. Notre objectif était d'identifier des motifs temporels récurrents dans des données environnementales, en mettant en lumière des variations corrélées. Malgré ces défis, nous avons non seulement enrichi nos connaissances dans l'algorithmique, mais nous avons également fait connaissance avec l'algorithme Apriori et exploré le monde du data mining.

Bibliographie

[1] Rakesh Agrawal and Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules*, 1994.

[2] Christian Borgelt. *Frequent item set mining*, 2012.

[3] Anuj Karpatne Vipin Kumar Pang-Ning Tan, Michael Steinbach. *Introduction to Data Mining* , 2018.