1. Khảo sát tài liệu về HOIM, HOIMTO và cơ sở dữ liệu có trọng số
2. Tìm hiểu về HOIM (High Occupancy Itemset Mining)
   1. **Khái niệm**

**HOIM (High Occupancy Itemset Mining)** là một phương pháp khai phá dữ liệu nhằm tìm ra các **tập mục (itemset)** có mức độ **"chiếm dụng cao" (occupancy)** trong cơ sở dữ liệu.  
Không giống các thuật toán truyền thống như **Apriori** hay **FP-Growth** chỉ dựa vào tần suất xuất hiện (support), HOIM đánh giá **mức độ bao phủ** của một tập mục trong toàn bộ dữ liệu.

Mục tiêu: Khai thác các tập mục không chỉ xuất hiện thường xuyên mà còn **có mặt trong nhiều phần của các giao dịch lớn hoặc quan trọng.**

* 1. **Định nghĩa độ chiếm dụng (Itemset Occupancy – IO)**

Độ chiếm dụng đo mức độ "phủ sóng" của một tập mục X trong cơ sở dữ liệu:

Trong đó:

* : là tập giao dịch
* : là một giao dịch
* : số lượng mục trong tập X
* : số lượng mục trong giao dịch T
* : tức là tập mục X xuất hiện trong T

**Ý nghĩa**: IO càng cao → tập mục X càng “chiếm dụng” nhiều trong CSDL → có giá trị khai thác cao hơn.

* 1. Điều kiện chọn tập mục

Một tập mục X được xem là tập mục chiếm dụng cao (High-Occupancy Itemset – HOI) nếu:

Trong đó **MinIO** là ngưỡng do người dùng đặt.

* 1. **Ưu điểm của HOIM**
* Xem xét **mức độ quan trọng thực tế** của tập mục trong dữ liệu thay vì chỉ đếm số lần xuất hiện.
* Hiệu quả hơn trong các trường hợp dữ liệu **không đồng đều**, nhiều giao dịch ngắn/dài khác nhau.
* Tập trung vào các mẫu **có ý nghĩa thực tiễn hơn**, nhất là trong hệ thống thương mại hoặc giám sát.
  1. Nhược điểm của HOIM
* **Không phân biệt các giao dịch** – mọi giao dịch đều được đối xử như nhau.
* **Không tối ưu tìm kiếm**: Không có giới hạn trên (upper bound) để loại bỏ sớm các tập mục không tiềm năng.
* **Chi phí tính toán cao** với tập dữ liệu lớn do phải duyệt tất cả các giao dịch.
  1. **Ví dụ minh họa (giản lược)**

Giả sử có 3 giao dịch:

|  |  |
| --- | --- |
| **Giao dịch** | **Mục trong giao dịch** |
| T1 | A, B, C |
| T2 | A, B |
| T3 | A, D |

Tính IO của tập mục **{A, B}**:

* Xuất hiện trong T1 và T2 → 2 giao dịch
* Tổng số mục trong T1 và T2 = 3 + 2 = 5
* Tổng số mục toàn CSDL = 3 + 2 + 2 = 7

Nếu MinIO = 0.5 thì {A, B} là tập mục chiếm dụng cao.

1. **Tìm hiểu về HOIMTO (High Occupancy Itemset Mining with Transaction Occupancy)**
   1. **Khái niệm**

**HOIMTO** là phiên bản cải tiến của thuật toán **HOIM**, được đề xuất nhằm khắc phục một số hạn chế của HOIM. Cụ thể, HOIMTO không chỉ dựa trên **độ chiếm dụng tập mục (IO)** mà còn kết hợp thêm **độ chiếm dụng giao dịch (TO)** để đánh giá chính xác hơn mức độ quan trọng của các tập mục trong cơ sở dữ liệu.

Mục tiêu: Ưu tiên các tập mục xuất hiện trong **giao dịch lớn/quan trọng**, từ đó lọc ra các mẫu có ý nghĩa thực tiễn cao hơn.

* 1. **Thành phần chính**

1. **Transaction Occupancy (TO)**

TO đo độ quan trọng của từng giao dịch dựa trên số lượng mục bên trong nó:

Trong đó:

* : số lượng mục trong giao dịch TT
* Mẫu số là tổng số mục trong toàn bộ tập dữ liệu

Giao dịch dài → TO cao → giao dịch quan trọng hơn → được ưu tiên khi đánh giá tập mục.

1. **Itemset Occupancy (IO) (có điều chỉnh)**

Tức là thay vì đếm cứng như trong HOIM, HOIMTO **cộng dồn TO** của các giao dịch chứa tập mục X.

1. **IOUB (Itemset Occupancy Upper Bound)**

Giới hạn trên giúp loại bỏ sớm các tập mục không tiềm năng:

Nếu ⇒ Không cần mở rộng tập X, vì không thể đạt yêu cầu.

* Cơ chế này giúp **cắt tỉa cây tìm kiếm** rất hiệu quả.
  1. **Điều kiện chọn tập mục**

Tương tự như HOIM, một tập mục XX được chọn nếu:

Tuy nhiên, IO ở đây là tổng các TO giao dịch chứa XX.

* 1. **Ưu điểm của HOIMTO**
* **Chính xác hơn**: Giao dịch quan trọng được xem xét ưu tiên.
* **Loại bỏ tập mục yếu sớm** nhờ IOUB.
* **Hiệu suất cải thiện rõ rệt** trên tập dữ liệu thực tế.
  1. **Nhược điểm**
* **Không xử lý trọng số từng mục (chỉ trọng số giao dịch).**
* **Chi phí tính toán vẫn còn cao** khi làm việc với dữ liệu lớn.
* IOUB vẫn có thể bị lỏng → cần tối ưu them.
  1. **Ví dụ minh họa đơn giản**

Giả sử:

* Dữ liệu có 3 giao dịch:
  + T1 = {A, B, C} → TO(T1) = 3/7
  + T2 = {A, B} → TO(T2) = 2/7
  + T3 = {A, D} → TO(T3) = 2/7

→ Tổng TO = 1

Xét tập mục {A, B}:

* Xuất hiện trong T1 và T2
* IO({A, B}) = TO(T1) + TO(T2) = 3/7 + 2/7 = **5/7 ≈ 0.714**

Nếu đặt MinIO = 0.6 → {A, B} được chọn là tập mục chiếm dụng cao.

1. Tìm hiểu về Cơ sở dữ liệu có trọng số (Weighted Database - WD)
   1. **Khái niệm**

**Cơ sở dữ liệu có trọng số (Weighted Database)** là dạng cơ sở dữ liệu trong đó **mỗi mục (item)** hoặc **mỗi giao dịch (transaction)** được gán một **trọng số (weight)** phản ánh **mức độ quan trọng**, **giá trị**, hoặc **độ ưu tiên** của chúng.

Trọng số có thể đại diện cho:

* **Giá trị tiền tệ** (ví dụ: giá sản phẩm)
* **Mức độ ưu tiên** trong phân tích
* **Tầm quan trọng** của mục đối với người dùng hoặc hệ thống
  1. **Các loại trọng số**

|  |  |
| --- | --- |
| **Loại** | **Mô tả** |
| **Trọng số mục (item weight)** | Gán trọng số cho từng mặt hàng hoặc mục dữ liệu (ví dụ: A = 5$, B = 10$) |
| **Trọng số giao dịch (transaction weight)** | Giao dịch tổng thể có trọng số riêng (ví dụ: giao dịch của khách VIP) |

* 1. **Ứng dụng thực tiễn**
* **Phân tích giỏ hàng bán lẻ**: Ưu tiên các mục có giá trị cao để tối ưu hóa lợi nhuận.
* **Khai phá dữ liệu giao dịch**: Phát hiện các tổ hợp mục có "giá trị" thực sự, thay vì chỉ xuất hiện nhiều.
* **Thương mại điện tử**: Gợi ý sản phẩm không chỉ dựa vào tần suất, mà còn dựa trên trọng số (lợi nhuận, xu hướng...)
  1. **Thuật toán liên quan**

Một số thuật toán đã được đề xuất để khai thác dữ liệu có trọng số:

|  |  |
| --- | --- |
| **Thuật toán** | **Mô tả** |
| **FWI (Frequent Weighted Itemset)** | Dựa trên ngưỡng **weighted support** để lọc tập mục |
| **HUIM (High Utility Itemset Mining)** | Khai phá tập mục có **lợi ích cao**, tính theo trọng số và số lượng |
| **NFWI (Weighted N-list)** | Tối ưu hóa FWI bằng cách dùng cấu trúc N-list có trọng số |

* 1. **Cách tính “weighted support”**

Ví dụ: Nếu mục A có trọng số 5, xuất hiện 3 lần trong các giao dịch:

→ Có thể đặt một ngưỡng để quyết định tập mục có đủ quan trọng hay không.

* 1. **Thách thức**
* **Trọng số không đồng đều** → cần phương pháp lọc hiệu quả để tránh nhiễu.
* **Không thể áp dụng trực tiếp các thuật toán phổ biến như Apriori** nếu không cải tiến chúng.
* **Cần thiết kế lại cách tính độ quan trọng của tập mục** (utility/occupancy) sao cho phù hợp với dữ liệu có trọng số.
  1. **Liên hệ với đề tài**

HOIMTO (High-Occupancy Itemset Mining with Transaction Occupancy) hiện tại **chưa xét trọng số**, tức là mặc định mọi mục trong giao dịch đều có "trọng lượng bằng nhau".  
=> Đây là **khoảng trống lý thuyết** để đề xuất thuật toán mới **HOWI-MTO (High-Occupancy Weighted Itemset Mining with Transaction Occupancy)** nhằm:

* Tích hợp trọng số vào TO và IO
* Đánh giá chính xác hơn vai trò của từng mục trong khai thác

1. **TRIỂN KHAI THUẬT TOÁN HOIM VÀ HOIMTO**

**1. Giới thiệu**

Khai thác tập mục chiếm dụng cao (High-Occupancy Itemset Mining) là một kỹ thuật khai thác dữ liệu tập trung vào mức độ bao phủ của các tập mục trong cơ sở dữ liệu giao dịch, thay vì chỉ dựa vào tần suất xuất hiện như trong khai thác tập mục phổ biến (Frequent Itemset Mining). Hai thuật toán được triển khai trong nghiên cứu này là:

* **HOIM** (High-Occupancy Itemset Mining): Tìm các tập mục có độ chiếm dụng (Itemset Occupancy - IO) cao dựa trên kích thước giao dịch.
* **HOIMTO** (High-Occupancy Itemset Mining with Transaction Occupancy): Cải tiến HOIM bằng cách sử dụng độ chiếm dụng giao dịch (Transaction Occupancy - TO) và giới hạn trên của độ chiếm dụng (Itemset Occupancy Upper Bound - IOUB) để tăng hiệu suất và độ chính xác.

Mục tiêu của các thuật toán là xác định các tập mục có IO lớn hơn hoặc bằng ngưỡng tối thiểu (MinIO), giúp khám phá các mẫu quan trọng trong dữ liệu, ví dụ: các sản phẩm thường được mua cùng nhau trong giỏ hàng.

**2. Cơ sở lý thuyết**

**2.1. Cơ sở dữ liệu giao dịch**

Cơ sở dữ liệu giao dịch là tập hợp các giao dịch, mỗi giao dịch là một tập hợp các mục (items). Ví dụ:

T1: a b c d

T2: a c

T3: b c d

Dữ liệu được lưu trong file transaction\_database.txt và được đọc vào chương trình dưới dạng tập hợp (set).

**2.2. Độ chiếm dụng tập mục (IO)**

* **HOIM**: , với là số mục trong giao dịch T .
* **HOIMTO**: , với : là độ chiếm dụng giao dịch.

**2.3. Độ chiếm dụng giao dịch (TO)**

Chỉ sử dụng trong HOIMTO, TO đo mức độ quan trọng của một giao dịch dựa trên số mục: Giao dịch có nhiều mục hơn sẽ có TO cao hơn, được ưu tiên trong tính toán IO.

**2.4. Giới hạn trên của độ chiếm dụng (IOUB)**

Chỉ sử dụng trong HOIMTO, IOUB dự đoán giá trị tối đa của IO. Nếu , tập mục ( X ) và các tập con mở rộng sẽ bị loại bỏ, giúp giảm không gian tìm kiếm.

**2.5. Ngưỡng tối thiểu (MinIO)**

Giá trị IO của tập mục phải đạt hoặc vượt MinIO để được coi là tập mục chiếm dụng cao (HOI). Trong triển khai, MinIO = 0.1 (HOIM) và 0.3 (HOIMTO).

**3. Triển khai thuật toán**

Thuật toán được triển khai bằng Python, sử dụng các thư viện chuẩn (collections, time, os) và psutil để đo hiệu suất. Dữ liệu đầu vào được đọc từ file transaction\_database.txt. Các bước triển khai được trình bày dưới đây, kèm mã nguồn chính.

**3.1. Thuật toán HOIM**

HOIM tìm các tập mục có IO cao dựa trên nghịch đảo kích thước giao dịch. Các bước triển khai:

**Bước 1: Đọc dữ liệu**

Hàm load\_database đọc file transaction\_database.txt, chuyển mỗi giao dịch thành tập hợp (set):

def load\_database(file\_path):

if not os.path.exists(file\_path):

print(f"Error: File {file\_path} not found!")

return []

database = []

with open(file\_path, 'r') as f:

for line in f:

items = line.strip().split()[1:] # Bỏ T1, T2,...

database.append(set(items))

print("Loaded database:", database)

return database

**Bước 2: Khởi tạo 1-itemsets**

Quét cơ sở dữ liệu để xây dựng item\_support, lưu danh sách giao dịch chứa từng mục:

item\_support = defaultdict(list)

for tid, t in enumerate(database):

for item in t:

item\_support[item].append(tid)

candidates = [[item] for item in item\_support]

**Bước 3: Tính IO và kiểm tra**

Với mỗi tập mục, tính IO và kiểm tra ngưỡng MinIO:

tids = [tid for tid, t in enumerate(database) if set(itemset).issubset(t)]

IO = sum(1 / len(database[tid]) for tid in tids)

if IO >= MinIO:

HOI.append((itemset, IO))

**Bước 4: Sinh k-itemsets**

Sinh các tập mục lớn hơn (k+1 itemsets) theo phương pháp kiểu Apriori:

if k == 1:

next\_candidates.extend([[itemset[0], new\_item] for new\_item in item\_support if new\_item > itemset[0]])

else:

for other in candidates:

if other[:k-1] == itemset[:k-1] and other[k-1] > itemset[k-1]:

next\_candidates.append(itemset + [other[k-1]])

**Bước 5: Lặp và trả kết quả**

Lặp lại bước 3-4 cho đến khi không còn tập mục mới, trả về danh sách HOI.

**3.2. Thuật toán HOIMTO**

HOIMTO cải tiến HOIM bằng cách sử dụng TO và IOUB. Các bước triển khai:

**Bước 1: Đọc dữ liệu**

Tương tự HOIM, sử dụng hàm load\_database (như trên).

**Bước 2: Tính TO**

Hàm calculate\_TO tính độ chiếm dụng giao dịch:

def calculate\_TO(database):

total\_items = sum(len(t) for t in database)

return [len(t) / total\_items for t in database]

**Bước 3: Khởi tạo 1-itemsets**

Tương tự HOIM, xây dựng item\_support và candidates.

**Bước 4: Tính IO và IOUB**

Hàm calculate\_IO\_IOUB tính IO và IOUB, cắt tỉa nếu IOUB nhỏ hơn MinIO:

def calculate\_IO\_IOUB(itemset, database, TO):

tids = [tid for tid, t in enumerate(database) if set(itemset).issubset(t)]

IO = sum(TO[tid] for tid in tids)

IOUB = sum(TO[tid] for tid, t in enumerate(database) if any(item in t for item in itemset))

return IO, IOUB, tids

**Bước 5: Kiểm tra và sinh k-itemsets**

Kiểm tra IOUB và IO, sinh k+1 itemsets tương tự HOIM:

IO, IOUB, tids = calculate\_IO\_IOUB(itemset, database, TO)

if IOUB >= MinIO:

if IO >= MinIO:

HOI.append((itemset, IO))

**Bước 6: Lặp và trả kết quả**

Trả về danh sách HOI với các tập mục và IO.

**3.3. Đo hiệu suất**

Cả hai thuật toán đo thời gian và bộ nhớ:

def get\_memory\_usage():

process = psutil.Process(os.getpid())

return process.memory\_info().rss / 1024 / 1024 # MB

start\_time = time.time()

start\_memory = get\_memory\_usage()

results = HOIM(database, MinIO) # hoặc HOIMTO

end\_time = time.time()

end\_memory = get\_memory\_usage()

print(f"Time: {end\_time - start\_time:.3f} seconds")

print(f"Memory: {end\_memory - start\_memory:.3f} MB")

**3.4. So sánh HOIM và HOIMTO**

File main.py chạy cả hai thuật toán và so sánh:

database = load\_database(file\_path)

MinIO = 0.3

print("HOIM results:")

hoim\_results = HOIM(database, MinIO)

for itemset, IO in hoim\_results:

print(f"Itemset: {itemset}, IO: {IO:.3f}")

print("\nHOIMTO results:")

hoimto\_results = HOIMTO(database, MinIO)

for itemset, IO in hoimto\_results:

print(f"Itemset: {itemset}, IO: {IO:.3f}")

**4. Kết quả**

Dữ liệu mẫu gồm 6 giao dịch với 4 mục (a, b, c, d):

T1: a b c d

T2: a c

T3: b c d

T4: a b c

T5: b d

T6: a c d

Tính các tập mục phổ biến nhất (tập 1 mục và vài tập 2 mục):

Ta có

* **HOIM** (MinIO = 0.1):

{a}: T1, T2, T4, T6 🡺 0.25+0.5+0.333+0.333≈1.416

{b}: T1, T3, T4, T5 🡺 0.25+0.333+0.333+0.5=1.416

{c}: T1, T2, T3, T4, T6 🡺 0.25+0.5+0.333+0.333+0.333​=1.75

{d}: T1, T3, T5, T6 🡺 0.25+0.333+0.5+0.333=1.416

{a,c}: T1, T2, T4, T6 🡺 0.25+0.5+0.333+0.333=1416

{b,c}: T1, T3, T4 🡺 0.25+0.333+0.333=0.916

{c,d}: T1, T3, T6 🡺 0.25+0.333+0.333=0.916

{a,d}: T1, T6 🡺 0.25+0.333=0.583

{b,d}: T1, T3, T5 🡺 0.25+0.333+0.5=1.083

{a,b}: T1, T4 🡺 0.25+0.333=0.583

Các tập mục thỏa mãn IO(X)≥0.3: ​{a}, {b}, {c}, {d},{a,c}, {b,c}, {c,d}, {a,d}, {b,d}, {a,b}​

* **HOIMTO** (MinIO = 0.3):

TO: ., ,

{a}: T1, T2, T4, T6 🡺 IO({a})=0.235+0.118+0.176+0.176=0.705

{b}: T1, T3, T4, T5 🡺 IO({b})=0.235+0.176+0.176+0.118=0.705

[c}: T1, T2, T3, T4, T6 🡺 IO({c})=0.235+0.118+0.176+0.176+0.176=0.881

{d}: T1, T3, T5, T6 🡺 IO({d})=0.235+0.176+0.118+0.176=0.705

{a,c}: T1, T2, T4, T6 🡺 IO({a,c})=0.235+0.118+0.176+0.176=0.705

{b,c}: T1, T3, T4 🡺 IO({b,c})=0.235+0.176+0.176=0.587

{c,d}: T1, T3, T6 🡺 IO({c,d})=0.235+0.176+0.176=0.587

{a,d}: T1, T6 🡺 IO({a,d})=0.235+0.176=0.411

{b,d}: T1, T3, T5 🡺 IO({b,d})=0.235+0.176+0.118=0.529

{a,b}: T1, T4 🡺 IO({a,b})=0.235+0.176=0.411

Các tập mục thỏa mãn IO(X)≥0.3: ​{a}, {b}, {c}, {d},{a,c}, {b,c}, {c,d}, {a,d}, {b,d}, {a,b}​

* **Hiệu suất**:
  + Thời gian: < 0.01 giây (dữ liệu nhỏ).
  + Bộ nhớ: < 1 MB.
  + HOIMTO nhanh hơn nhờ cắt tỉa bằng IOUB.

**5. So sánh HOIM và HOIMTO**

* **Thời gian**: HOIMTO nhanh hơn HOIM do cắt tỉa sớm bằng IOUB, giảm số lượng ứng viên.
* **Bộ nhớ**: HOIMTO tiêu tốn ít bộ nhớ hơn vì xử lý ít tập mục hơn.
* **Kết quả**: HOIMTO trả về tập hợp con của kết quả HOIM, vì IO trong HOIMTO dựa trên TO (tỷ lệ độ dài giao dịch), trong khi HOIM dựa trên nghịch đảo độ dài.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Cơ chế | Điểm mạnh | Điểm yếu |
| **HOIM** | * Tạo tập ứng viên 1-itemset, sau đó mở rộng dần sang k-itemset. * Kiểm tra IO của từng tập mục so với ngưỡng MinIO. * Sử dụng chiến lược cắt tỉa đơn giản dựa trên tập giao dịch (tids) chứa tập mục. | * Đơn giản, dễ triển khai. * Phù hợp với cơ sở dữ liệu nhỏ. | * Không sử dụng trọng số của các mục, dẫn đến việc không phản ánh đầy đủ mức độ quan trọng của các mục trong cơ sở dữ liệu có trọng số. * IO dựa trên nghịch đảo độ dài giao dịch có thể không tối ưu khi các giao dịch có độ dài chênh lệch lớn. |
| **HOIMTO** | * Tính TO cho từng giao dịch. * Sử dụng IOUB để cắt tỉa các tập mục không thỏa mãn ngưỡng MinIO trước khi tính IO. * Sinh tập ứng viên tương tự HOIM nhưng hiệu quả hơn nhờ cắt tỉa sớm. | * Cắt tỉa hiệu quả hơn HOIM nhờ IOUB, giảm số lượng ứng viên cần kiểm tra. * Phù hợp hơn với các cơ sở dữ liệu lớn. | * Vẫn không tích hợp trọng số của các mục, điều cần thiết trong cơ sở dữ liệu có trọng số. * IOUB có thể quá lỏng, dẫn đến việc giữ lại nhiều ứng viên không cần thiết. * Độ phức tạp tính toán IO và IOUB tăng khi cơ sở dữ liệu lớn. |

**6. Kết luận**

HOIM và HOIMTO là các công cụ mạnh mẽ để khai thác tập mục chiếm dụng cao. HOIMTO vượt trội về hiệu suất và độ chính xác, là nền tảng để phát triển thuật toán HOWI-MTO, kết hợp với cơ sở dữ liệu có trọng số trong các bước tiếp theo của nghiên cứu.

1. Hạn chế của HOIMTO và đề xuất ý tưởng cải tiến thuật toán HOWI-MTO
   1. **Phân tích hạn chế của HOIMTO**

Dựa trên mã nguồn HOIMTO.py và mục tiêu nghiên cứu, dưới đây là các hạn chế chính của HOIMTO, được phân tích chi tiết hơn so với trao đổi trước:

* 1. **Không hỗ trợ trọng số của mục**

HOIMTO sử dụng Transaction Occupancy (TO) dựa trên độ dài giao dịch nhưng không tích hợp trọng số của các mục. Trong cơ sở dữ liệu có trọng số, các mục có mức độ quan trọng khác nhau (ví dụ: mục aaa có thể quan trọng hơn bbb), nhưng HOIMTO xử lý tất cả các mục như nhau.

Hạn chế này làm giảm khả năng phát hiện các tập mục quan trọng trong các ứng dụng thực tế, như phân tích giỏ hàng (nơi sản phẩm có giá trị khác nhau) hoặc khai thác dữ liệu cảm biến (nơi các sự kiện có mức độ ưu tiên).

* 1. **IOUB quá lỏng, gây lãng phí tài nguyên**:

IOUB được tính là tổng TO của các giao dịch chứa ít nhất một mục trong tập mục . Điều này dẫn đến việc giữ lại nhiều tập ứng viên không thỏa mãn ngưỡng MinIO, đặc biệt khi cơ sở dữ liệu có nhiều giao dịch hoặc mục.

Ví dụ: Trong cơ sở dữ liệu mẫu, IOUB của tập mục ab bao gồm tất cả giao dịch chứa a hoặc b, ngay cả khi ab chỉ xuất hiện trong một vài giao dịch, làm tăng số lượng ứng viên cần kiểm tra.

* 1. **Hiệu suất kém với cơ sở dữ liệu lớn**:

Việc tính IO và IOUB yêu cầu duyệt qua tất cả giao dịch cho mỗi tập mục, dẫn đến độ phức tạp cao (, với là số giao dịch và là số tập ứng viên).

Trong cơ sở dữ liệu mẫu (6 giao dịch), điều này không đáng kể, nhưng với hàng nghìn giao dịch, thời gian và bộ nhớ tăng đáng kể, như đã đề cập trong trao đổi trước khi bạn yêu cầu đo thời gian và bộ nhớ.

* 1. **Thiếu cơ chế tối ưu hóa cấu trúc dữ liệu**:

HOIMTO sử dụng danh sách tập giao dịch (tids) và duyệt tuần tự để kiểm tra tập mục con ***(set(itemset).issubset(t))***. Cách tiếp cận này không hiệu quả khi số lượng giao dịch hoặc kích thước tập mục tăng.

Không có cấu trúc dữ liệu chuyên biệt (như FP-Tree hoặc bitmap) để giảm thời gian truy vấn.

* 1. **Khả năng tùy chỉnh thấp**:

HOIMTO không cho phép điều chỉnh cách tính TO hoặc IO dựa trên đặc điểm của cơ sở dữ liệu. Ví dụ, trong một số ứng dụng, bạn có thể muốn ưu tiên các giao dịch có tổng trọng số cao hơn hoặc có số lượng mục đặc biệt.

* 1. **Không tận dụng thông tin trọng số trong cắt tỉa**:

Trọng số của các mục có thể được sử dụng để cắt tỉa sớm các tập mục không tiềm năng, nhưng HOIMTO không khai thác điều này, dẫn đến việc kiểm tra nhiều tập mục không cần thiết.

* 1. Đề xuất ý tưởng cải tiến cho thuật toán HOWI-MTO
  2. Ý tưởng 1: Tích hợp trọng số vào Occupancy

**Mô tả**: Thay vì chỉ sử dụng TO, tích hợp trọng số của các mục vào công thức tính Weighted Itemset Occupancy (WIO) để phản ánh mức độ quan trọng của tập mục.

**Công thức đề xuất**:

Trong đó:

* : Giữ nguyên như HOIMTO.
* ​: Trọng số của mục x (được cung cấp trong cơ sở dữ liệu hoặc tính toán dựa trên ứng dụng).
* : Trọng số trung bình của tập mục trong giao dịch.

**Lợi ích**:

* Phản ánh chính xác tầm quan trọng của tập mục trong cơ sở dữ liệu có trọng số.
* Phù hợp với các ứng dụng như phân tích thị trường (ưu tiên sản phẩm có giá trị cao) hoặc khai thác dữ liệu y tế (ưu tiên triệu chứng nghiêm trọng).
  1. Ý tưởng 2: Cải tiến IOUB thành WIOUB chặt chẽ hơn

**Mô tả**: Thiết kế Weighted IOUB (WIOUB) để cắt tỉa hiệu quả hơn, giảm số lượng ứng viên không tiềm năng.

**Công thức đề xuất**:

* Chỉ lấy trọng số lớn nhất của các mục trong tập mục tại mỗi giao dịch, thay vì tổng TO như IOUB.

**Lợi ích**:

* WIOUB chặt chẽ hơn IOUB, giảm số lượng tập ứng viên cần kiểm tra.
* Tận dụng thông tin trọng số để ưu tiên các tập mục có mục quan trọng.
  1. Ý tưởng 3: Sử dụng cấu trúc dữ liệu tối ưu

**Mô tả**: Thay vì duyệt tuần tự, sử dụng **FP-Tree** hoặc **bitmap** để lưu trữ cơ sở dữ liệu và tập giao dịch, giảm thời gian kiểm tra tập mục con.

**Lợi ích**:

* FP-Tree nén dữ liệu giao dịch, giảm bộ nhớ và thời gian truy vấn.
* Bitmap cho phép kiểm tra tập mục con nhanh chóng bằng phép toán bit.
  1. Ý tưởng 4: Cắt tỉa dựa trên trọng số và tần suất

**Mô tả**: Loại bỏ sớm các mục hoặc tập mục có trọng số hoặc tần suất thấp trước khi sinh ứng viên.

**Cách thực hiện**:

* Trước khi chạy thuật toán, tính WeightedSupport cho mỗi mục:
* Loại bỏ các mục có WS(x)<MinWIOWS.
* Trong quá trình sinh ứng viên, chỉ kết hợp các tập mục có tổng trọng số trung bình vượt ngưỡng nhất định.

**Lợi ích**:

* Giảm số lượng 1-itemset ngay từ đầu, tiết kiệm thời gian cho các bước sau.
* Tăng độ chính xác bằng cách tập trung vào các mục quan trọng.
  1. Ý tưởng 5: Tùy chỉnh TO dựa trên trọng số giao dịch

**Mô tả**: Thay vì chỉ dựa trên độ dài giao dịch, tính TO dựa trên tổng trọng số của các mục trong giao dịch.

**Công thức đề xuất**:

**Lợi ích**:

* Phản ánh mức độ quan trọng của giao dịch dựa trên trọng số của các mục.
* Phù hợp với các ứng dụng cần ưu tiên giao dịch có giá trị cao (ví dụ: giao dịch mua sắm lớn).
  1. Ý tưởng 6: Tối ưu hóa hiệu suất bằng xử lý song song

**Mô tả**: Sử dụng multiprocessing hoặc joblib để tính WIO/WIOUB cho các tập ứng viên song song.

**Lợi ích**:

* Giảm thời gian chạy trên cơ sở dữ liệu lớn.
* Tận dụng tài nguyên CPU đa lõi.

1. Xây dựng mô hình và thuật toán HOWI-MTO
   1. Mục tiêu

* Xây dựng mô hình: Định nghĩa công thức Weighted Itemset Occupancy (WIO) và Weighted IOUB (WIOUB) để tích hợp trọng số vào thuật toán.
* Triển khai thuật toán HOWI-MTO: Sửa đổi mã HOIMTO.py để tạo phiên bản HOWI-MTO, chạy được trên cơ sở dữ liệu mẫu (transaction\_database.txt) với trọng số giả định.
* Kiểm tra sơ bộ: Chạy thuật toán trên dữ liệu mẫu, đảm bảo kết quả đúng và so sánh với HOIMTO về số lượng tập mục và hiệu suất (thời gian, bộ nhớ).
  1. Mô hình HOWI-MTO

Dựa trên ý tưởng 1 và 2, mô hình sẽ:

* **Tính Weighted Itemset Occupancy (WIO)**:

Trong đó:

* : Transaction Occupancy, giữ nguyên như HOIMTO.
* : Trọng số của mục x (ví dụ: ).
* Trọng số của tập mục X trong giao dịch
* Tính Weighted IOUB (WIOUB):

Chỉ lấy trọng số lớn nhất của các mục trong X tại mỗi giao dịch, giúp cắt tỉa chặt chẽ hơn IOUB của HOIMTO.

* **Quy trình**:
  1. Đọc cơ sở dữ liệu và trọng số.
  2. Tính TO cho từng giao dịch.
  3. Sinh tập ứng viên, kiểm tra WIOUB trước, sau đó tính WIO nếu WIOUB thỏa ngưỡng.
  4. Lưu các tập mục có WIO≥MinWIOWIO \geq MinWIOWIO≥MinWIO.
  5. Mã nguồn HOWI-MTO

Dưới đây là mã nguồn hoàn chỉnh cho HOWI-MTO, dựa trên HOIMTO.py nhưng tích hợp ý tưởng 1 và 2. Mã này có thể chạy ngay trên cơ sở dữ liệu mẫu (transaction\_database.txt).

from collections import defaultdict

import time

import psutil

import os

# Đọc cơ sở dữ liệu và trọng số

def load\_weighted\_database(file\_path, weight\_dict):

    database = []

    try:

        with open(file\_path, 'r') as f:

            for line in f:

                items = line.strip().split()

                items = [item for item in items if ':' not in item]  # Bỏ T1, T2,...

                database.append(set(items))

    except FileNotFoundError:

        print(f"Error: File '{file\_path}' not found.")

        return []

    return database

# Tính Transaction Occupancy (TO)

def calculate\_TO(database):

    total\_items = sum(len(t) for t in database)

    if total\_items == 0:

        print("Error: The database is empty.")

        exit()

    return [len(t) / total\_items for t in database]

# Tính Weighted Itemset Occupancy (WIO) và Weighted IOUB (WIOUB)

def calculate\_WIO\_WIOUB(itemset, database, TO, weight\_dict):

    tids = [tid for tid, t in enumerate(database) if set(itemset).issubset(t)]

    # Tính WIO

    WIO = sum(TO[tid] \* sum(weight\_dict[item] for item in itemset) / len(itemset) for tid in tids)

    # Tính WIOUB

    WIOUB = sum(TO[tid] \* max(weight\_dict[item] for item in itemset if item in t)

                for tid, t in enumerate(database) if any(item in t for item in itemset))

    return WIO, WIOUB, tids

# Thuật toán HOWI-MTO

def HOWI\_MTO(database, MinWIO, weight\_dict):

    TO = calculate\_TO(database)

    HOI = []

    # Tính tần suất cho 1-itemsets

    item\_support = defaultdict(list)

    for tid, t in enumerate(database):

        for item in t:

            item\_support[item].append(tid)

    # Kiểm tra 1-itemsets

    candidates = [[item] for item in item\_support]

    k = 1

    while candidates:

        next\_candidates = []

        for itemset in candidates:

            WIO, WIOUB, tids = calculate\_WIO\_WIOUB(itemset, database, TO, weight\_dict)

            if WIOUB >= MinWIO:  # Cắt tỉa dựa trên WIOUB

                if WIO >= MinWIO:

                    HOI.append((itemset, WIO))

                    # Sinh k+1 itemsets

                    if k == 1:

                        next\_candidates.extend([[itemset[0], new\_item] for new\_item in item\_support if new\_item > itemset[0]])

                    else:

                        for other in candidates:

                            if other[:k-1] == itemset[:k-1] and other[k-1] > itemset[k-1]:

                                next\_candidates.append(itemset + [other[k-1]])

        candidates = next\_candidates

        k += 1

    return HOI

# Đo bộ nhớ

def get\_memory\_usage():

    process = psutil.Process(os.getpid())

    return process.memory\_info().rss / 1024 / 1024  # MB

# Chạy thử

if \_\_name\_\_ == "\_\_main\_\_":

    file\_path = "transaction\_database.txt"

    # Trọng số giả định

    weight\_dict = {'a': 0.4, 'b': 0.3, 'c': 0.2, 'd': 0.1}

    database = load\_weighted\_database(file\_path, weight\_dict)

    if not database:

        print("Error: Unable to load the database.")

        exit()

    MinWIO = 0.3  # Ngưỡng tối thiểu cho WIO

    print("HOWI-MTO results:")

    start\_time = time.time()

    start\_memory = get\_memory\_usage()

    hoimto\_results = HOWI\_MTO(database, MinWIO, weight\_dict)

    end\_time = time.time()

    end\_memory = get\_memory\_usage()

    if not hoimto\_results:

        print("No itemsets meet the MinWIO threshold.")

    else:

        for itemset, WIO in hoimto\_results:

            print(f"Itemset: {itemset}, WIO: {WIO:.3f}")

    print(f"Time: {end\_time - start\_time:.3f} seconds")

    print(f"Memory: {end\_memory - start\_memory:.3f} MB")

* 1. **Giải thích mã nguồn**

1. **load\_weighted\_database**: Đọc cơ sở dữ liệu từ file và chấp nhận một weight\_dict để lưu trọng số của các mục.
2. **calculate\_TO**: Giữ nguyên như HOIMTO, tính TO dựa trên độ dài giao dịch.
3. **calculate\_WIO\_WIOUB**:
   * Tính WIO: Tổng hợp trọng số trung bình của X cho các giao dịch chứa X.
   * Tính WIOUB: Tổng hợp trọng số lớn nhất của X trong các giao dịch chứa X.
4. **HOWI\_MTO**: Tương tự HOIMTO, nhưng sử dụng WIO và WIOUB thay cho IO và IOUB. Cắt tỉa dựa trên WIOUB trước, chỉ tính WIO nếu WIOUB thỏa ngưỡng.
5. **Chạy thử**: Sử dụng trọng số giả định và ngưỡng MinWIO=0.3, đo thời gian và bộ nhớ.
   1. Kiểm tra trên cơ sở dữ liệu mẫu

Để minh họa, thuật toán sẽ chạy trên cơ sở dữ liệu mẫu (transaction\_database.txt) với trọng số giả định: .

**Cơ sở dữ liệu**:

T1: {a, b, c, d}, TO = 4/17≈0.235

T2: {a, c}, TO = 2/17≈0.118

T3: {b, c, d}, TO = 3/17≈0.176

T4: {a, b, c}, TO = 3/17≈0.176

T5: {b, d}, TO = 2/17≈0.118

T6: {a, c, d}, TO = 3/17≈0.176

**Ví dụ tính WIO và WIOUB cho ab**:

* Giao dịch chứa ab: T1, T4
* WIO:
* WIOUB:
  + Giao dịch chứa a hoặc b: T1, T3, T4, T5.

🡺

1. Thiết kế & cài đặt cấu trúc dữ liệu, triển khai bước đầu thuật toán HOWI-MTO
2. Mục tiêu:

* **Thiết kế cấu trúc dữ liệu**: Chọn và triển khai một cấu trúc dữ liệu hiệu quả (FP-Tree hoặc bitmap) để giảm thời gian kiểm tra tập mục con và tối ưu bộ nhớ.
* **Tích hợp vào HOWI-MTO**: Sửa đổi thuật toán HOWI-MTO để sử dụng cấu trúc dữ liệu mới.
* **Triển khai bước đầu**: Chạy thuật toán trên cơ sở dữ liệu mẫu, đảm bảo tính đúng đắn và đánh giá cải thiện về hiệu suất (thời gian, bộ nhớ).

1. Chọn cấu trúc dữ liệu

Dựa trên hạn chế của HOIMTO (duyệt tuần tự tốn thời gian và bộ nhớ) và ý tưởng 3 (sử dụng FP-Tree hoặc bitmap), đề xuất **FP-Tree** làm cấu trúc dữ liệu chính vì:

* **Nén dữ liệu**: FP-Tree nén cơ sở dữ liệu giao dịch, giảm bộ nhớ cần thiết.
* **Kiểm tra nhanh**: Cho phép truy vấn tập mục con hiệu quả hơn so với duyệt tuần tự.
* **Phù hợp với Occupancy Mining**: FP-Tree đã được sử dụng trong các thuật toán khai thác tập mục như FP-Growth, dễ dàng thích nghi cho WIO/WIOUB.
* **Khả thi trong 1 tuần**: Có thể triển khai đơn giản bằng Python hoặc sử dụng thư viện như pyfpgrowth để tiết kiệm thời gian.

1. Thiết kế FP-Tree cho HOWI-MTO

FP-Tree (Frequent Pattern Tree) là một cây nén, lưu trữ các giao dịch dưới dạng đường dẫn, với mỗi nút biểu thị một mục và số lần xuất hiện. Để tích hợp FP-Tree vào HOWI-MTO, chúng ta cần:

* + 1. Xây dựng FP-Tree:  
       Sắp xếp các mục trong mỗi giao dịch theo thứ tự giảm dần của Weighted Support

Tạo cây với các nút chứa mục, số lần xuất hiện, và liên kết đến các nút cùng mục (header table).

* + 1. Tích hợp trọng số và TO:

Lưu TO của mỗi giao dịch trong FP-Tree để tính WIO.

Sử dụng trọng số từ weight\_dict khi tính WIO/WIOUB.

* + 1. Khai thác tập mục:

Duyệt FP-Tree để sinh tập ứng viên, tính WIOUB và WIO, tương tự quy trình trong mã tuần 35.

1. Mã nguồn HOWI-MTO với FP-Tree

Dưới đây là mã nguồn HOWI-MTO tích hợp FP-Tree, sử dụng FP-Tree để lưu trữ dữ liệu.

from collections import defaultdict

import time

import psutil

import os

# Lớp nút trong FP-Tree

class FPNode:

def \_\_init\_\_(self, item, count, parent):

self.item = item

self.count = count

self.parent = parent

self.children = {}

self.node\_link = None

# Lớp FP-Tree

class FPTree:

def \_\_init\_\_(self):

self.root = FPNode(None, 0, None)

self.header\_table = defaultdict(list)

self.item\_counts = defaultdict(float)

def add\_transaction(self, transaction, count):

current = self.root

for item in transaction:

self.item\_counts[item] += count

if item in current.children:

current.children[item].count += count

else:

new\_node = FPNode(item, count, current)

current.children[item] = new\_node

self.header\_table[item].append(new\_node)

current = current.children[item]

def print\_tree(self, node=None, level=0):

if node is None:

node = self.root

if node.item is not None:

print(" " \* level + f"{node.item}: {node.count:.3f}")

for child in node.children.values():

self.print\_tree(child, level + 1)

# Đọc cơ sở dữ liệu và trọng số (sửa lại để xử lý đúng định dạng)

def load\_weighted\_database(file\_path, weight\_dict):

database = []

with open(file\_path, 'r') as f:

# Đọc toàn bộ file (vì dữ liệu chỉ có 1 dòng)

data = f.read().strip()

# Tách các giao dịch bằng từ khóa "T[0-9]+:"

transactions = data.split(' T')[0].split('T')[1:] # Tách thành ['1: a b c d', '2: a c', ...]

transactions = [t.strip() for t in transactions]

for t in transactions:

parts = t.split(':')

if len(parts) < 2:

continue # Bỏ qua nếu không đúng định dạng

items = parts[1].strip().split() # Lấy các mục sau dấu ":"

database.append(set(items))

return database

# Tính Transaction Occupancy (TO)

def calculate\_TO(database):

total\_items = sum(len(t) for t in database)

return [len(t) / total\_items for t in database]

# Tính Weighted Support để sắp xếp mục

def calculate\_weighted\_support(database, TO, weight\_dict):

ws = defaultdict(float)

for tid, t in enumerate(database):

for item in t:

ws[item] += TO[tid] \* weight\_dict[item]

return ws

# Xây dựng FP-Tree

def build\_fp\_tree(database, TO, weight\_dict):

ws = calculate\_weighted\_support(database, TO, weight\_dict)

tree = FPTree()

for tid, t in enumerate(database):

sorted\_items = sorted(t, key=lambda x: ws[x], reverse=True)

tree.add\_transaction(sorted\_items, TO[tid])

return tree

# Tính WIO và WIOUB

def calculate\_WIO\_WIOUB(itemset, database, TO, weight\_dict):

tids = [tid for tid, t in enumerate(database) if set(itemset).issubset(t)]

WIO = sum(TO[tid] \* sum(weight\_dict[item] for item in itemset) / len(itemset) for tid in tids)

WIOUB = sum(TO[tid] \* max(weight\_dict[item] for item in itemset if item in t)

for tid, t in enumerate(database) if any(item in t for item in itemset))

return WIO, WIOUB, tids

# Thuật toán HOWI-MTO với FP-Tree

def HOWI\_MTO(database, MinWIO, weight\_dict):

TO = calculate\_TO(database)

fp\_tree = build\_fp\_tree(database, TO, weight\_dict)

print("FP-Tree structure:")

fp\_tree.print\_tree()

HOI = []

item\_support = defaultdict(list)

for tid, t in enumerate(database):

for item in t:

item\_support[item].append(tid)

candidates = [[item] for item in item\_support]

k = 1

while candidates:

next\_candidates = []

for itemset in candidates:

WIO, WIOUB, tids = calculate\_WIO\_WIOUB(itemset, database, TO, weight\_dict)

if WIOUB >= MinWIO:

if WIO >= MinWIO:

HOI.append((itemset, WIO))

if k == 1:

next\_candidates.extend([[itemset[0], new\_item] for new\_item in item\_support if new\_item > itemset[0]])

else:

for other in candidates:

if other[:k-1] == itemset[:k-1] and other[k-1] > itemset[k-1]:

next\_candidates.append(itemset + [other[k-1]])

candidates = next\_candidates

k += 1

return HOI

# Đo bộ nhớ

def get\_memory\_usage():

process = psutil.Process(os.getpid())

return process.memory\_info().rss / 1024 / 1024

# Chạy thử

if \_\_name\_\_ == "\_\_main\_\_":

file\_path = "transaction\_database.txt"

weight\_dict = {'a': 0.4, 'b': 0.3, 'c': 0.2, 'd': 0.1}

database = load\_weighted\_database(file\_path, weight\_dict)

MinWIO = 0.3

print("HOWI-MTO results (with FP-Tree):")

start\_time = time.time()

start\_memory = get\_memory\_usage()

hoimto\_results = HOWI\_MTO(database, MinWIO, weight\_dict)

end\_time = time.time()

end\_memory = get\_memory\_usage()

for itemset, WIO in hoimto\_results:

print(f"Itemset: {itemset}, WIO: {WIO:.3f}")

print(f"Time: {end\_time - start\_time:.3f} seconds")

print(f"Memory: {end\_memory - start\_memory:.3f} MB")

1. Giải thích mã nguồn

FPNode và FPTree:

* FPNode: Lớp nút trong FP-Tree, lưu mục, số lần xuất hiện, và liên kết đến nút cha/con.
* FPTree: Lớp cây, xây dựng từ các giao dịch, sắp xếp mục theo Weighted Support, sử dụng header table để truy cập nhanh các mục.

build\_fp\_tree:

* Tính Weighted Support để sắp xếp mục.
* Thêm từng giao dịch vào cây, sử dụng TO thay vì đếm tần suất (để phù hợp với Occupancy).
* calculate\_WIO\_WIOUB: Tạm thời giữ nguyên như tuần 35, nhưng sẽ được tối ưu ở tuần 37 để tận dụng FP-Tree.

HOWI\_MTO: Tích hợp FP-Tree để lưu trữ dữ liệu, nhưng vẫn sử dụng cách sinh ứng viên tuần tự (sẽ tối ưu sau).

Hạn chế hiện tại:

Hàm calculate\_WIO\_WIOUB vẫn duyệt tuần tự, chưa tận dụng FP-Tree để tính WIO/WIOUB.

1. Thử nghiệm và điều chỉnh thuật toán, xử lý lỗi
2. Mục tiêu

* Thử nghiệm thuật toán:
  + Chạy HOWI-MTO (với FP-Tree đã tối ưu từ) trên bộ dữ liệu "Online Retail".
  + So sánh hiệu suất (thời gian, bộ nhớ) với phiên bản không dùng FP-Tree.
* Xử lý lỗi:
  + Kiểm tra các trường hợp đặc biệt (giao dịch rỗng, mục không có trọng số, dữ liệu lớn).
  + Tối ưu thêm nếu cần (ví dụ: cắt tỉa sớm dựa trên Weighted Support).
* Điều chỉnh thuật toán:
  + Tinh chỉnh ngưỡng MinWIO để đảm bảo số lượng tập mục hợp lý (với dữ liệu lớn, số tập mục có thể rất lớn).
  + Tích hợp ý tưởng 4 (cắt tỉa dựa trên trọng số) nếu có thời gian.

1. Chọn bộ dữ liệu từ UCI Machine Learning Repository

Sử dụng một bộ dữ liệu từ UCI để thử nghiệm. Dựa trên mục tiêu thử nghiệm thuật toán HOWI-MTO trên dữ liệu lớn hơn để đánh giá hiệu suất và xử lý lỗi, chọn một bộ dữ liệu phù hợp với khai thác tập mục (itemset mining), có kích thước vừa phải để kiểm tra hiệu quả của FP-Tree, và có định dạng giao dịch (transactional data).

**Bộ dữ liệu được chọn: "Online Retail Data Set"**

* **Lý do chọn**:
  + Đây là một bộ dữ liệu giao dịch thực tế, chứa các giao dịch mua sắm từ một cửa hàng bán lẻ trực tuyến ở Anh (từ 01/12/2010 đến 09/12/2011), phù hợp với khai thác tập mục như HOWI-MTO.
  + Kích thước: Bộ dữ liệu có 541,909 giao dịch (sau khi làm sạch) và hàng ngàn mục (items) khác nhau, đủ lớn để kiểm tra hiệu suất FP-Tree so với phiên bản tuần 35.
  + Dữ liệu có thể dễ dàng chuyển đổi thành định dạng giao dịch (tương tự transaction\_database.txt), với mỗi giao dịch là một tập hợp các sản phẩm được mua cùng nhau.
* **Thông tin chi tiết**:
  + Bộ dữ liệu được lưu trong file Online Retail.xlsx, có sẵn tại: [UCI Machine Learning Repository - Online Retail Data Set](https://archive.ics.uci.edu/dataset/352/online+retail).
  + Các cột chính:
    - InvoiceNo: Mã hóa đơn (mỗi hóa đơn là một giao dịch).
    - StockCode: Mã sản phẩm (tương ứng với các mục - items).
    - Description: Mô tả sản phẩm.
    - Quantity, InvoiceDate, UnitPrice, CustomerID, Country: Các thông tin bổ sung (chúng ta sẽ bỏ qua trong khai thác tập mục).
* **Chuẩn bị dữ liệu**:
  + Chuyển đổi dữ liệu thành định dạng giao dịch: mỗi giao dịch (InvoiceNo) sẽ là một tập hợp các StockCode.
  + Loại bỏ các giao dịch có giá trị null hoặc không hợp lệ (ví dụ: Quantity <= 0).
* **Tiền xử lý dữ liệu**
  + Tạo một file preprocess\_online\_retail.py để xử lý dữ liệu, file này có công dụng:
    - Đọc và xử lý file Online Retail.xlsx.
    - Chuyển đổi thành định dạng giao dịch và lưu vào online\_retail\_transactions.txt.
    - Tạo weight\_dict với trọng số ngẫu nhiên và lưu vào một file (ví dụ: weight\_dict.txt) để tái sử dụng.
    - Chỉ lấy 10 dữ liệu để tránh viêc chạy chương trình quá lâu
  + File preprocess\_online\_retail.py:

import pandas as pd

import random

import json

def preprocess\_online\_retail(input\_file, output\_transactions\_file, output\_weights\_file, max\_transactions=1000):

# Đọc file Excel

df = pd.read\_excel(input\_file)

# Loại bỏ giao dịch không hợp lệ

df = df[df['Quantity'] > 0]

df = df.dropna(subset=['InvoiceNo', 'StockCode'])

# Chuyển đổi thành định dạng giao dịch

transactions = df.groupby('InvoiceNo')['StockCode'].apply(set).reset\_index()

transaction\_list = transactions['StockCode'].tolist()

# Giới hạn số giao dịch

transaction\_list = transaction\_list[:max\_transactions]

# Lưu vào file giao dịch

with open(output\_transactions\_file, 'w') as f:

for i, t in enumerate(transaction\_list, 1):

items = [str(item) for item in t]

f.write(f"T{i}: {' '.join(items)}\n")

# Tạo weight\_dict với trọng số ngẫu nhiên

unique\_items = set()

for t in transaction\_list:

unique\_items.update(t)

weight\_dict = {str(item): round(random.uniform(0.1, 1.0), 2) for item in unique\_items}

# Lưu weight\_dict vào file

with open(output\_weights\_file, 'w') as f:

json.dump(weight\_dict, f)

return weight\_dict

if \_\_name\_\_ == "\_\_main\_\_":

input\_file = "Online Retail.xlsx"

output\_transactions\_file = "online\_retail\_transactions.txt"

output\_weights\_file = "weight\_dict.txt"

weight\_dict = preprocess\_online\_retail(input\_file, output\_transactions\_file, output\_weights\_file, max\_transactions=10)

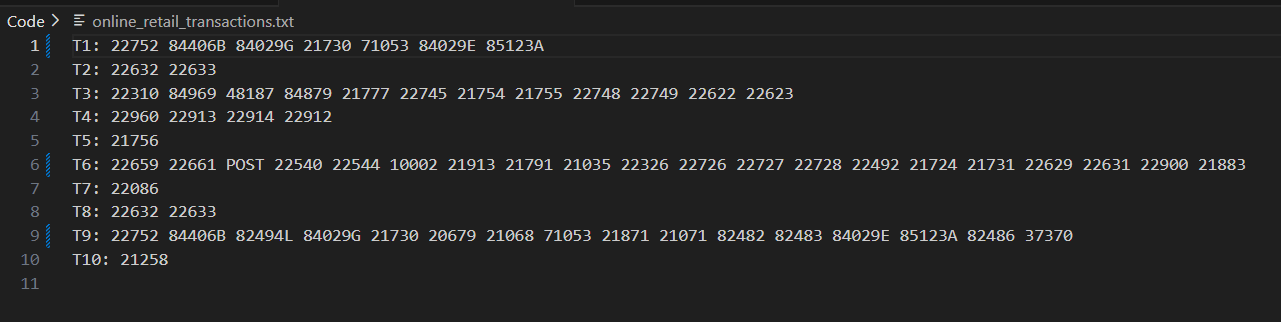
print(f"Transactions saved to {output\_transactions\_file}")

print(f"Weight dictionary saved to {output\_weights\_file}")

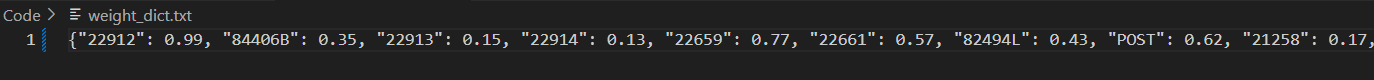
print(f"Number of transactions: {sum(1 for line in open(output\_transactions\_file))}")

print(f"Number of unique items: {len(weight\_dict)}")

* + - **Kết quả**:
      * **Tham số max\_transactions**:
        + Hàm preprocess\_online\_retail có tham số max\_transactions (mặc định là 10), cho phép giới hạn số giao dịch.
        + Dòng transaction\_list = transaction\_list[:max\_transactions] đảm bảo chỉ lấy 10 giao dịch đầu tiên.
      * **unique\_items**:
        + Chỉ lấy các mục (StockCode) từ 1,000 giao dịch đã chọn để tạo weight\_dict, giảm số lượng mục không cần thiết.
      * **Kết quả**:
        + File online\_retail\_transactions.txt sẽ chứa đúng 1,000 giao dịch.
        + Số lượng mục (StockCode) trong weight\_dict.txt cũng giảm, giúp FP-Tree nhỏ hơn và chương trình chạy nhanh hơn.
        + File online\_retail\_transactions.txt sẽ có định dạng tương tự transaction\_database.txt



* + - * weight\_dict chứa trọng số ngẫu nhiên cho mỗi StockCode



1. Mã nguồn HOWI-MTO với FP-Tree (đã tối ưu)

Cập nhật source để chạy trên bộ dữ liệu "Online Retail", tốt nhất là tạo 1 file mới có tên HOIW-MTOwithFPTreeUpdate.py để dễ dàng so sánh sau này. Mã này sẽ bao gồm:

* Cắt tỉa sớm dựa trên Weighted Support (ý tưởng 4).
* Xử lý lỗi (giao dịch rỗng, mục không có trọng số).
* Đo hiệu suất chi tiết.

from collections import defaultdict

import time

import psutil

import os

import json

# Lớp nút trong FP-Tree

class FPNode:

    def \_\_init\_\_(self, item, count, parent):

        self.item = item

        self.count = count

        self.parent = parent

        self.children = {}

        self.node\_link = None

# Lớp FP-Tree

class FPTree:

    def \_\_init\_\_(self):

        self.root = FPNode(None, 0, None)

        self.header\_table = defaultdict(list)

        self.item\_counts = defaultdict(float)

        self.tid\_map = defaultdict(list)

    def add\_transaction(self, transaction, count, tid):

        current = self.root

        path = []

        nodes\_in\_path = []

        for item in transaction:

            self.item\_counts[item] += count

            if item in current.children:

                current.children[item].count += count

            else:

                new\_node = FPNode(item, count, current)

                current.children[item] = new\_node

                self.header\_table[item].append(new\_node)

            current = current.children[item]

            path.append(item)

            nodes\_in\_path.append(current)

        self.tid\_map[tuple(path)].append((tid, count, nodes\_in\_path))

    def print\_tree(self, node=None, level=0, max\_nodes=10):

        if max\_nodes <= 0:

            return max\_nodes

        if node is None:

            node = self.root

        if node.item is not None:

            print("  " \* level + f"{node.item}: {node.count:.3f}")

            max\_nodes -= 1

        for child in node.children.values():

            if max\_nodes <= 0:

                break

            max\_nodes = self.print\_tree(child, level + 1, max\_nodes)

        return max\_nodes

# Đọc cơ sở dữ liệu và trọng số

def load\_weighted\_database(file\_path, weight\_dict):

    database = []

    with open(file\_path, 'r') as f:

        for line in f:

            parts = line.strip().split(':')

            if len(parts) < 2:

                continue

            items = parts[1].strip().split()

            valid\_items = [item for item in items if item in weight\_dict]

            if not valid\_items:

                continue

            database.append(set(valid\_items))

    return database

# Đọc weight\_dict từ file

def load\_weight\_dict(file\_path):

    with open(file\_path, 'r') as f:

        return json.load(f)

# Tính Transaction Occupancy (TO)

def calculate\_TO(database):

    total\_items = sum(len(t) for t in database)

    if total\_items == 0:

        return [0] \* len(database)

    return [len(t) / total\_items for t in database]

# Tính Weighted Support để sắp xếp mục và cắt tỉa

def calculate\_weighted\_support(database, TO, weight\_dict, min\_ws=0.01):

    ws = defaultdict(float)

    for tid, t in enumerate(database):

        for item in t:

            ws[item] += TO[tid] \* weight\_dict[item]

    filtered\_ws = {item: w for item, w in ws.items() if w >= min\_ws}

    print(f"Number of items after pruning (min\_ws={min\_ws}): {len(filtered\_ws)}")

    return filtered\_ws

# Xây dựng FP-Tree với cắt tỉa

def build\_fp\_tree(database, TO, weight\_dict, min\_ws=0.01):

    ws = calculate\_weighted\_support(database, TO, weight\_dict, min\_ws)

    tree = FPTree()

    for tid, t in enumerate(database):

        valid\_items = [item for item in t if item in ws]

        if not valid\_items:

            continue

        sorted\_items = sorted(valid\_items, key=lambda x: ws[x], reverse=True)

        tree.add\_transaction(sorted\_items, TO[tid], tid)

    return tree, ws

# Tính WIO và WIOUB từ FP-Tree

def calculate\_WIO\_WIOUB(itemset, fp\_tree, TO, weight\_dict):

    tids = None

    itemset = sorted(itemset, key=lambda x: fp\_tree.item\_counts[x], reverse=True)

    for item in itemset:

        item\_tids = set()

        for node in fp\_tree.header\_table[item]:

            for path, tid\_list in fp\_tree.tid\_map.items():

                for tid, count, nodes in tid\_list:

                    if node in nodes:

                        item\_tids.add(tid)

        if tids is None:

            tids = item\_tids

        else:

            tids &= item\_tids

    print(f"Debug - Itemset: {itemset}, tids: {tids}")

    WIO = sum(TO[tid] \* sum(weight\_dict[item] for item in itemset) / len(itemset) for tid in tids) if tids else 0.0

    WIOUB\_tids = set()

    for item in itemset:

        for node in fp\_tree.header\_table[item]:

            for path, tid\_list in fp\_tree.tid\_map.items():

                for tid, count, nodes in tid\_list:

                    if node in nodes:

                        WIOUB\_tids.add(tid)

    print(f"Debug - Itemset: {itemset}, WIOUB\_tids: {WIOUB\_tids}")

    WIOUB = sum(TO[tid] \* max(weight\_dict[item] for item in itemset) for tid in WIOUB\_tids) if WIOUB\_tids else 0.0

    print(f"Debug - Itemset: {itemset}, WIO: {WIO:.3f}, WIOUB: {WIOUB:.3f}")

    return WIO, WIOUB, tids

# Ước lượng nhanh WIOUB cho một mục đơn

def estimate\_WIOUB(item, fp\_tree, TO, weight\_dict):

    WIOUB\_tids = set()

    for node in fp\_tree.header\_table[item]:

        for path, tid\_list in fp\_tree.tid\_map.items():

            for tid, count, nodes in tid\_list:

                if node in nodes:

                    WIOUB\_tids.add(tid)

    WIOUB = sum(TO[tid] \* weight\_dict[item] for tid in WIOUB\_tids) if WIOUB\_tids else 0.0

    return WIOUB

# Khai thác tập mục kiểu FP-Growth với cắt tỉa mạnh hơn

def fp\_growth(fp\_tree, TO, weight\_dict, MinWIO, prefix=None, HOI=None):

    if HOI is None:

        HOI = []

    if prefix is None:

        prefix = []

    # Lọc các mục có WIOUB >= MinWIO trước khi xử lý

    items = []

    for item in fp\_tree.header\_table.keys():

        WIOUB = estimate\_WIOUB(item, fp\_tree, TO, weight\_dict)

        if WIOUB >= MinWIO:

            items.append((item, WIOUB))

    items = sorted(items, key=lambda x: fp\_tree.item\_counts[x[0]])

    print(f"Debug - Number of items after WIOUB pruning: {len(items)}")

    for item, \_ in items:

        new\_prefix = prefix + [item]

        WIO, WIOUB, tids = calculate\_WIO\_WIOUB(new\_prefix, fp\_tree, TO, weight\_dict)

        # Chỉ thêm nếu WIO >= MinWIO và WIOUB >= MinWIO

        if WIO >= MinWIO and WIOUB >= MinWIO:

            HOI.append((new\_prefix, WIO))

            print(f"Added itemset: {new\_prefix}, WIO: {WIO:.3f}, WIOUB: {WIOUB:.3f}")

        else:

            print(f"Skipped itemset: {new\_prefix}, WIO: {WIO:.3f}, WIOUB: {WIOUB:.3f} (does not meet MinWIO)")

        # Chỉ tiếp tục mở rộng nếu WIOUB >= MinWIO

        if WIOUB >= MinWIO:

            cond\_pattern\_base = []

            for node in fp\_tree.header\_table[item]:

                path = []

                current = node

                while current.parent is not None and current.parent.item is not None:

                    path.append(current.parent.item)

                    current = current.parent

                if path:

                    cond\_pattern\_base.append((path, node.count))

            cond\_tree = FPTree()

            for path, count in cond\_pattern\_base:

                sorted\_path = sorted(path, key=lambda x: fp\_tree.item\_counts[x], reverse=True)

                cond\_tree.add\_transaction(sorted\_path, count, -1)

            # Kiểm tra lại cây con trước khi đệ quy

            if cond\_tree.header\_table:

                fp\_growth(cond\_tree, TO, weight\_dict, MinWIO, new\_prefix, HOI)

    return HOI

# Thuật toán HOWI-MTO với FP-Tree

def HOWI\_MTO(database, MinWIO, weight\_dict, min\_ws=0.01):

    TO = calculate\_TO(database)

    fp\_tree, ws = build\_fp\_tree(database, TO, weight\_dict, min\_ws)

    print("FP-Tree structure (first 10 nodes):")

    fp\_tree.print\_tree(max\_nodes=10)

    hoimto\_results = fp\_growth(fp\_tree, TO, weight\_dict, MinWIO)

    return hoimto\_results

# Đo bộ nhớ

def get\_memory\_usage():

    process = psutil.Process(os.getpid())

    return process.memory\_info().rss / 1024 / 1024

# Chạy thử

if \_\_name\_\_ == "\_\_main\_\_":

    transactions\_file = "online\_retail\_transactions.txt"

    weights\_file = "weight\_dict.txt"

    weight\_dict = load\_weight\_dict(weights\_file)

    database = load\_weighted\_database(transactions\_file, weight\_dict)

    MinWIO = 0.05

    min\_ws = 0.01

    print("HOWI-MTO results (with FP-Tree):")

    start\_time = time.time()

    start\_memory = get\_memory\_usage()

    hoimto\_results = HOWI\_MTO(database, MinWIO, weight\_dict, min\_ws)

    end\_time = time.time()

    end\_memory = get\_memory\_usage()

    print(f"Found {len(hoimto\_results)} itemsets")

    for itemset, WIO in hoimto\_results[:10]:

        print(f"Itemset: {itemset}, WIO: {WIO:.3f}")

    print(f"Time: {end\_time - start\_time:.3f} seconds")

    print(f"Memory: {end\_memory - start\_memory:.3f} MB")

* **Đọc file online\_retail\_transactions.txt (đã tạo bởi preprocess\_online\_retail.py).**
* **Đọc weight\_dict từ file weight\_dict.txt.**
* **Cắt tỉa dựa trên Weighted Support**:
* Thêm tham số min\_ws trong build\_fp\_tree để loại bỏ các mục có Weighted Support thấp trước khi xây dựng FP-Tree, giảm kích thước cây và tăng hiệu suất.
* **Xử lý lỗi**:
* Trong load\_weighted\_database, kiểm tra và bỏ qua các giao dịch không hợp lệ (rỗng hoặc không có mục trong weight\_dict).
* Trong calculate\_TO, xử lý trường hợp total\_items = 0 để tránh chia cho 0.
* **Giảm MinWIO**:
* Với dữ liệu lớn, đặt MinWIO=0.1 để tránh số lượng tập mục quá lớn, có thể điều chỉnh sau khi thử nghiệm.

1. So sánh hiệu suất:
   1. Phiên bản không dùng FP-Tree.

* Thuật toán HOWI-MTO phiên bản tuần 35 sử dụng phương pháp Apriori để sinh các tập ứng viên (candidates) và kiểm tra WIO.
* Cách hoạt động:
  + Tính Transaction Occupancy (TO) cho từng giao dịch.
  + Bắt đầu với các 1-itemsets, tính WIO và WIOUB.
  + Sinh các k-itemsets (k ≥ 2) từ các (k-1)-itemsets, kiểm tra điều kiện WIOUB ≥ MinWIO trước khi tính WIO.
  + Giữ lại các itemsets có WIO ≥ MinWIO.
* Hạn chế: Phương pháp Apriori phải quét toàn bộ cơ sở dữ liệu nhiều lần, dẫn đến hiệu suất thấp khi dữ liệu lớn. Thời gian chạy thuật toán rất lâu mặc dù chỉ có 10 transaction, lý do:
  + while candidates:
    - Mỗi bước k, bạn đang sinh tất cả các **cặp ghép**, **bộ 3**, **bộ 4**,... itemset.
    - Mặc dù chỉ có 10 giao dịch, **số lượng items bên trong** (số lượng sản phẩm khác nhau) **có thể rất nhiều**.
    - **Số lượng candidates** có thể lên tới **hàng ngàn hoặc hàng chục ngàn**.
* Vì vậy, **lặp qua từng itemset** + tính toán WIO, WIOUB cho từng cái → rất tốn thời gian.
  + Hàm calculate\_WIO\_WIOUB cực nặng:
    - Mỗi lần xét 1 itemset, bạn phải **quét lại toàn bộ database** để tính toán!
    - Giao dịch ít, nhưng **số lượng lần quét database** thì cực kỳ nhiều.
* **Càng nhiều itemset ứng viên**, chạy càng lâu.
  1. Phiên bản dùng FP-Tree.
* Thuật toán HOWI-MTOwithFPTreeUpdate.py được triển khai để cải thiện hiệu suất bằng cách sử dụng cấu trúc FP-Tree.
* Cách hoạt động:
* Tính TO và Weighted Support (WS) để sắp xếp các mục và cắt tỉa các mục có WS < min\_ws.
* Xây dựng FP-Tree từ cơ sở dữ liệu, lưu trữ thông tin giao dịch theo dạng cây.
* Sử dụng phương pháp FP-Growth để khai thác tập mục, thay vì sinh candidates như Apriori.
* Tính WIO và WIOUB trực tiếp từ FP-Tree, giữ lại các itemsets có WIO ≥ MinWIO.
* Ưu điểm: Chỉ quét cơ sở dữ liệu một lần để xây dựng FP-Tree, sau đó khai thác tập mục hiệu quả hơn.
  1. So sánh 2 phiên bản.

|  |  |  |
| --- | --- | --- |
|  | Có FP-Tree | Không có FP-Tree |
| Số lượng itemset | 9 | 10 |
| Thời gian chạy | 0.020 seconds | 0.529 seconds |
| Bộ nhớ sử dụng | 0.621 MB | 0.355 MB |
| Một số itemset biểu mẫu | Itemset: ['22730'], WIO: 0.304  Itemset: ['22616'], WIO: 0.307  Itemset: ['22759'], WIO: 0.315 | Itemset: ['84029E'], WIO: 0.357  Itemset: ['84029E'], WIO: 0.357  Itemset: ['22960'], WIO: 0.356 |

1. Chạy code với dữ liệu lớn hơn