KHAI THÁC TẬP MỤC HỮU ÍCH CAO KHÔNG BỊ THỐNG TRỊ DỰA TRÊN THUẬT GIẢI DI TRUYỀN VÀ ĐÁNH GIÁ SONG SONG ĐA LUỒNG

Lê Thị Minh Nguyện, Tưởng Thị Xuân Thu

Khoa Công nghệ thông tin, Trường Đại học Ngoại ngữ -Tin học TP.HCM

nguyenltm@huflit.edu.vn, thuttx@huflit.edu.vn

TÓM TẮT— Bài báo đề xuất thuật giải PHU_GA_SKYLINE – khai thác tập mục hữu ích cao không bị thống trị (Skyline High-Utility Itemsets – SHUIs) từ cơ sở dữ liệu giao dịch định lượng, kết hợp thuật giải di truyền (GA) với lọc thống trị Skyline và đánh giá song song đa luồng. Không cần đặt ngưỡng minUtil hay minSup trước, thuật giải tự xác định tập kết quả tối ưu Pareto theo đồng thời hai tiêu chí: utility và support. Mỗi cá thể được mã hóa bằng bitarray nhị phân; quần thể khởi tạo từ những hạt giống với chất lượng cao; đánh giá fitness song song bằng ThreadPoolExecutor; bảng băm (hashtable) phân tầng theo support tăng tốc lọc thống trị. Thực nghiệm trên Chess, Foodmart, Mushroom và Connect cho thấy PHU_GA_SKYLINE giảm bộ nhớ trên 175 lần so với SKYFUP-B (Chess), đạt thời gian thấp nhất trên Foodmart, và xử lý thành công Connect (67.557 giao dịch) khi các phương pháp truyền thống không hội tụ được.

Từ khóa— Khai thác tập mục hữu ích cao, Skyline, thuật giải di truyền, lọc thống trị, song song đa luồng, bitarray.

I. GIỚI THIỆU

Đặt ngưỡng minUtil và minSup là điểm yếu cố hữu của hầu hết thuật giải khai thác tập mục hữu ích cao hiện nay mà người dùng phải thử sai nhiều lần mà vẫn dễ bỏ sót kết quả có giá trị. Bài báo này xuất phát từ đúng vấn đề đó. Khai thác tập mục phổ biến (FIM) [1] và khai thác tập mục hữu ích cao (HUIM) [2] [3] được ứng dụng rộng rãi trong phân tích hành vi tiêu dùng, tối ưu danh mục sản phẩm và hỗ trợ ra quyết định trong thương mại điện tử, bán lẻ và tài chính.

Tuy nhiên, các thuật giải truyền thống Apriori [1], UP-Growth [3], SFU-Miner [2] đều phụ thuộc vào ngưỡng cố định minUtil và minSup. Sự chủ quan này dẫn đến hai vấn đề: ngưỡng quá thấp sinh số lượng kết quả khổng lồ gây khó khăn trong việc khai thác và phân tích dữ liệu; ngưỡng quá cao bỏ sót những tập mục thực sự có giá trị [4]. Không gian tìm kiếm tập mục tăng theo cấp số mũ ($2^n - 1$) khiến duyệt toàn bộ bất khả thi với dữ liệu lớn.

Khái niệm Skyline trong HUIM khắc phục điều này: tập mục X được giữ lại nếu không tồn tại tập Y nào tốt hơn X đồng thời về cả utility lẫn support. Tập Skyline phản ánh biên Pareto tối ưu và không đòi hỏi người dùng đặt ngưỡng.

Bài báo đề xuất PHU_GA_SKYLINE (Parallel High-Utility GA Skyline), kết hợp GA [5] với lọc Skyline và đánh giá song song đa luồng. Những đóng góp chính:

- Khung thuật giải tích hợp GA + Skyline đa tiêu chí, không cần ngưỡng thủ công.
- Chiến lược khởi tạo quần thể từ hạt giống (items_dict, itemsets_dict) giúp việc tìm kiếm nhanh hơn khởi tạo ngẫu nhiên.
- Đánh giá fitness song song đa luồng (ThreadPoolExecutor) rút ngắn thời gian tính toán đối với mỗi cá thể.
- Hashtable phân tầng theo support cho lọc thống trị Skyline nhanh, tiết kiệm bộ nhớ.

Phần còn lại của bài báo được tổ chức như sau: mục II – các nghiên cứu liên quan, mục III – khái niệm và định nghĩa cơ bản, mục IV – thuật giải đề xuất, mục V – kết quả thực nghiệm và đánh giá, mục VI – kết luận.

II. CÁC NGHIÊN CỨU LIÊN QUAN

A. KHAI THÁC TẬP MỤC PHỔ BIẾN VÀ HỮU ÍCH CAO

Agrawal và Srikant (1994) [1] đặt nền móng FIM với Apriori, dựa trên nguyên lý downward closure. Liu và cộng sự (2005) [2] giới thiệu TWU (Transaction-Weighted Utility) làm cận trên của utility, mở đầu cho thuật toán HUIM. Tseng và cộng sự (2013) [3] đề xuất UP-Growth với cấu trúc UP-Tree: duyệt CSDL hai lần, lần đầu tính TU và TWU, lần sau tạo cây nén để rút gọn số lần quét. Fournier-Viger và cộng sự (2019) [4] khảo sát toàn diện HUIM và chỉ ra rằng sự phụ thuộc vào ngưỡng là hạn chế cốt lõi của các nghiên cứu về HUIM.

B. THUẬT GIẢI DI TRUYỀN TRONG KHAI THÁC HUI

GA [5]đã được áp dụng trong khai thác HUI: Kannimuthu và Premalatha [7] dùng GA với đột biến có thứ hạng; Luna và cộng sự [8] đề xuất Top-k HUIM dựa trên GA, loại bỏ ngưỡng minUtil. Về song song hóa: Nguyen và cộng sự [9] thực hiện song song đa nhân; Kimura và cộng sự [10] mở rộng lên ICDE 2023; Fang và cộng sự [11] kết hợp GPU với heuristic. Tuy nhiên, chưa có nghiên cứu nào kết hợp GA với Skyline đa tiêu chí cho HUIM kèm đánh giá song song đa luồng.

III. CÁC KHÁI NIỆM VÀ ĐỊNH NGHĨA CƠ BẢN

A. CƠ SỞ DỮ LIỆU GIAO DỊCH

Cho tập mục I = {i_1, i_2, …, i_n} gồm n mục phân biệt. Một giao dịch T ⊆ I là tập con của I; mỗi mục i ∈ T mang theo: (1) số lượng q(i,T) – số đơn vị mục i xuất hiện trong T, và (2) lợi nhuận đơn vị p(i) – lợi nhuận thu được từ mỗi đơn vị mục i (hằng số toàn CSDL). CSDL D = {T_1, T_2, …, T_m} gồm m giao dịch. Không gian tìm kiếm gồm 2^n – 1 tập con khác rỗng; với n = 76 (tập Chess) không gian vượt 75 nghìn tỉ tập mục – duyệt toàn bộ là bất khả thi.

Bảng 1. *Ví dụ CSDL giao dịch định lượng có tiện ích*

TID	Items	TU	Utils (utility từng mục)
T1	3 5 1 2 4 6	30	1 3 5 10 6 5
T2	3 5 2 4	20	3 3 8 6
T3	3 1 4	8	1 5 2
T4	3 5 1 7	27	6 6 10 5
T5	3 5 2 7	11	2 3 4 2

B. CÁC ĐỊNH NGHĨA VỀ ĐỘ HỮU ÍCH

Định nghĩa 1 (Hữu ích của mục trong giao dịch): $u(i,T) = q(i,T) \times p(i)$. Đây là đóng góp hữu ích thực tế của mục i vào giao dịch T. Ví dụ từ Bảng 1: cột Utils ghi trực tiếp giá trị u(i,T) đã tính sẵn. Chẳng hạn, u(item3, T1) = 1, u(item5, T1) = 3, u(item1, T1) = 5, u(item2, T1) = 10, u(item4, T1) = 6, u(item6, T1) = 5 (đọc trực tiếp từ cột Utils của T1 theo thứ tự).

Định nghĩa 2 (Transaction Utility – TU): $TU(T) = \sum_i \in T\ u(i,T)$ là tổng hữu ích của tất cả mục trong giao dịch T – đây là mức lợi nhuận tối đa nếu toàn bộ giao dịch được chấp thuận. Ví dụ từ Bảng 1: T1 = {3,5,1,2,4,6} với utils = {1,3,5,10,6,5}, do đó TU(T1) = 1+3+5+10+6+5 = 30.

Định nghĩa 3 (Độ hữu ích tập mục X trong T): $u(X,T) = \sum_i \in X \cap T\ u(i,T)$. Đây là đóng góp của tập mục X vào giao dịch T; bằng 0 nếu X ⊄ T (tức giao dịch không chứa đầy đủ X). Ví dụ: X = {item3, item5}, T2 = {item3,item5,item2,item4}: u(X,T2) = u(item3,T2)+u(item5,T2) = 3+3 = 6.

Định nghĩa 4 (Tổng hữu ích trên CSDL): $u(X) = \sum\{T \in D, X \subseteq T\}\ u(X,T)$. Tổng hữu ích của X trên toàn CSDL – chỉ tính trên những giao dịch chứa đầy đủ X. Đây là chỉ số đo giá trị hữu ích tổng thể thực của X. Ví dụ: X = {item3, item5} xuất hiện trong T1,T2,T4,T5: u(X) = (1+3)+(3+3)+(6+6)+(2+3) = 27.

Định nghĩa 5 (TWU – Transaction-Weighted Utility): $TWU(i) = \sum\{T \in D, i \in T\}\ TU(T)$. Tổng TU của tất cả giao dịch chứa mục i. TWU là cận trên (upper bound) của u(X) cho mọi X chứa i: u(X) ≤ TWU(i). Tính chất then chốt: nếu TWU(i) < minUtil thì toàn bộ các tập chứa i đều không phải HUI → có thể tỉa sớm mà không bỏ sót kết quả, là nền tảng của UP-Growth [6]. Ví dụ: item1 xuất hiện trong T1(TU=30), T3(TU=8), T4(TU=27) → TWU(item1) = 65.

C. QUAN HỆ THỐNG TRỊ VÀ TẬP SKYLINE

Định nghĩa 6 (Thống trị): Tập mục X thống trị Y (X ≻ Y) khi:

$$[f(X) \geq y(Y)\ \wedge\ u(X) > u(Y)]\ \vee\ [f(X) > f(Y)\ \wedge\ u(X) \geq u(Y)]$$

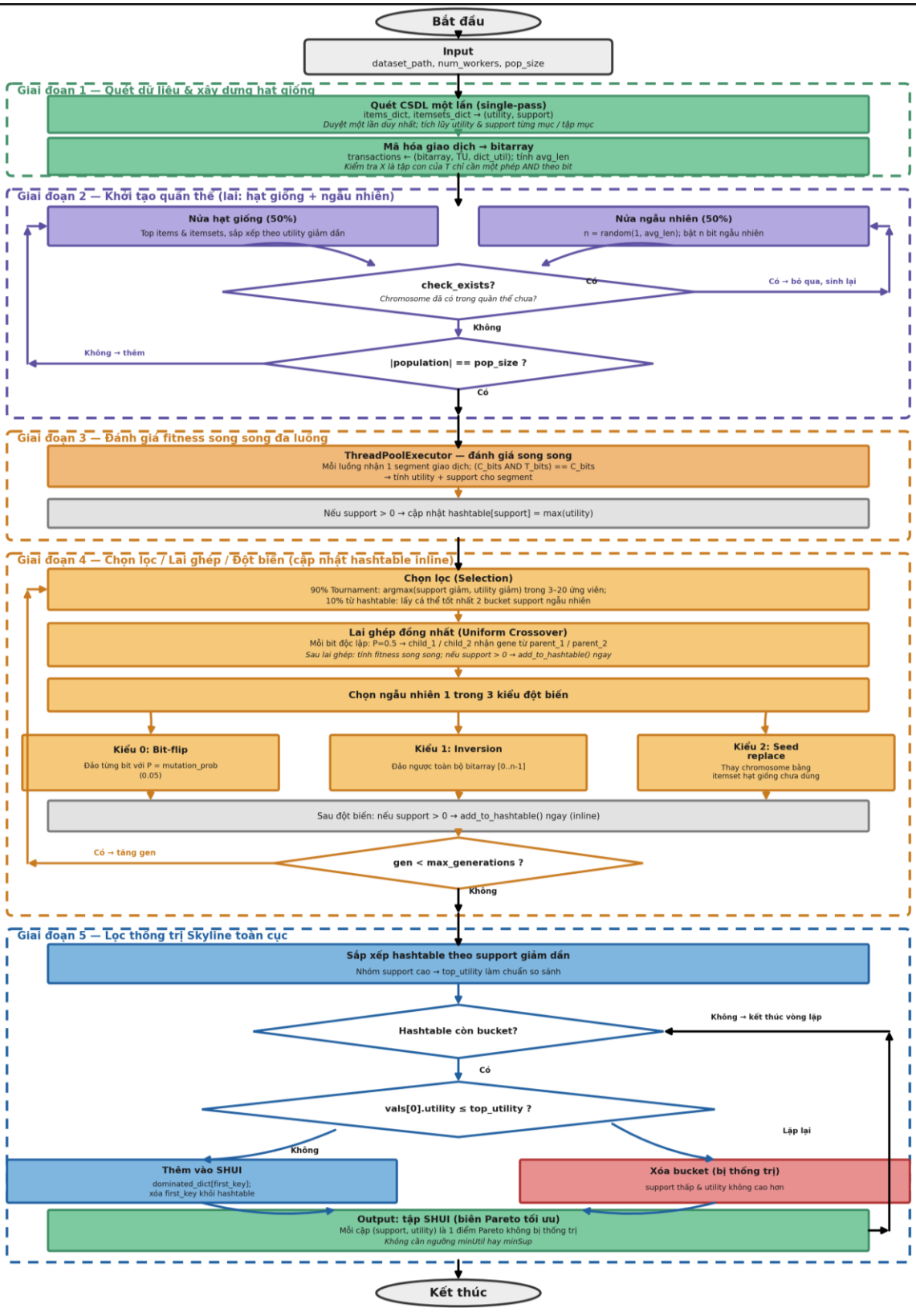
trong đó f(X) là support (số giao dịch chứa X) và u(X) là utility. Điều kiện này đảm bảo X "không tệ hơn" Y trên cả hai tiêu chí và "tốt hơn" ít nhất một. Nếu X ≻ Y thì Y bị loại – không có lý do giữ lại Y khi X đã vượt trội hoàn toàn. Ví dụ: X={item2}(S=3,U=22), Y={item4}(S=3,U=14): U_X>U_Y và S_X=S_Y → X ≻ Y, loại Y.

Định nghĩa 7 (Skyline High-Utility Itemset – SHUI): X là SHUI nếu không tồn tại Y ∈ D nào thỏa Y ≻ X. Tập tất cả SHUI tạo thành biên Pareto tối ưu trong không gian (utility, support) – phản ánh sự đánh đổi tự nhiên: tập phổ biến thường utility thấp hơn tập hiếm gặp. Ưu điểm cốt lõi: không cần đặt ngưỡng minUtil hay minSup trước – tập SHUI tự xác định từ dữ liệu, loại bỏ sự chủ quan của người dùng. Kích thước tập SHUI luôn ≤ n (số mục đơn lẻ) – nhỏ hơn rất nhiều so với 2^n−1.

IV. THUẬT GIẢI PHU_GA_SKYLINE ĐỀ XUẤT

A. TỔNG QUAN SƠ ĐỒ

PHU_GA_SKYLINE chạy qua năm giai đoạn nối tiếp nhau như Hình 1: (1) Duyệt toàn bộ CSDL một lần, thu thập items_dict và itemsets_dict làm hạt giống, mỗi giao dịch được nén thành bitarray; (2) xây dựng quần thể ban đầu theo kiểu lai – phân nửa lấy từ hạt giống chất lượng cao, phân nửa còn lại sinh ngẫu nhiên; (3) tính fitness cho từng cá thể theo cơ chế song song đa luồng; (4) thực hiện vòng lặp tiến hóa gồm chọn lọc (tournament 90% kết hợp hashtable 10%), lai ghép đồng nhất và ba kiểu đột biến. Trong suốt vòng lặp này, mỗi khi lai ghép hoặc đột biến tạo ra chromosome mới có support > 0, hàm add_to_hashtable() được gọi tức thì (inline) để cập nhật bucket tương ứng trong hashtable phân tầng, thay vì chờ đến cuối thế hệ; (5) lọc thống trị Skyline trên toàn bộ hashtable, trả về tập SHUI cuối cùng. Suốt quá trình này, người dùng không cần chỉ định ngưỡng minUtil hay minSup – thuật giải tự xác định qua dữ liệu.



Hình 1. *Sơ đồ luồng thuật giải PHU_GA_SKYLINE*

B. QUÉT DỮ LIỆU VÀ TẠO HẠT GIỐNG

Duyệt CSDL một lần, xây dựng: items_dict = {item_id → (utility_sum, support_count)} và itemsets_dict = {tuple_items → (utility_sum, support_count)}. Mỗi giao dịch mã hóa thành (bitarray, TU, dict_util).

Thuật giải Quét dữ liệu

```
Input:  dataset_path
Output: transactions_ba, itemset_ba_units_dict, avg_len, biggest_item
```

```
d1. transactions_ba ← []; items_dict ← {}; itemsets_dict ← {}
d2. for each line in dataset do
d3.    parse line → items, trans_util, utility_values
d4.    for each item in items do
d5.       items_dict[item] += utility; update biggest_item
d6.    if len(items) ≥ 2 then itemsets_dict[tuple(items)] ← sum(utils)
d7.    ba ← bitarray(biggest_item, all 0)
d8.    for each item in items do ba[item-1] ← 1
d9.    transactions_ba.append((ba, trans_util, dict_util))
d10. avg_len ← total_items / |transactions_ba|
d11. for each itemset in itemsets_dict do
d12.    convert to bitarray → itemset_ba_units_dict
d13. sort itemset_ba_units_dict by utility DESC
d14. add_item_to_hashtable()
d15. return transactions_ba, itemset_ba_units_dict, avg_len
```

Giải thích

Dòng 1. Khởi tạo 3 cấu trúc chính: danh sách bitarray giao dịch, dict utility mục đơn lẻ, dict utility tập mục. Tất cả rỗng ban đầu.

Dòng 2. Duyệt csdl một lần duy nhất.

Dòng 3. Phân tích từng dòng: tách danh sách mục, giá trị TU và utility riêng lẻ của từng mục.

Dòng 4. Duyệt từng mục trong giao dịch hiện tại để tích lũy thống kê.

Dòng 5. Cộng dồn utility của item vào dict (hạt giống cho khởi tạo quần thể). Cập nhật biggest_item để xác định kích thước bitarray cần dùng.

Dòng 6. Lưu toàn giao dịch như một itemset hạt giống nếu có ≥ 2 mục. Itemset đơn đã được items_dict xử lý.

Dòng 7. Tạo bitarray kích thước = ID mục lớn nhất, toàn 0. Mỗi vị trí tương ứng một mục trong I.

Dòng 8. Bật bit tại vị trí (item_id − 1) cho mỗi mục xuất hiện. Index trừ 1 vì bitarray 0-indexed.

Dòng 9. Lưu bộ ba (bitarray, TU, dict_utility) dùng định dạng tối ưu cho phép kiểm tra X⊆T bằng phép AND bitwise.

Dòng 10. Tính độ dài trung bình giao dịch dùng để sinh chromosome ngẫu nhiên có kích thước thực tế, không quá thưa hay dày.

Dòng 11-12. Chuyển tất cả itemset trong itemsets_dict sang bitarray để tái sử dụng hiệu quả, tránh chuyển đổi lại mỗi lần.

Dòng 13. Sắp xếp giảm dần để bước khởi tạo quần thể lấy được các itemset tốt nhất trước.

Dòng 14. Nạp ngay các item đơn lẻ vào hashtable theo support tương ứng để khởi tạo hashtable ban đầu trước khi GA bắt đầu.

Dòng 15. Trả về 3 cấu trúc cần thiết cho toàn bộ quá trình GA: giao dịch đã mã hóa, hạt giống itemset, và avg_len.

Bảng 2. *Kết quả mã hóa CSDL Bảng 1 thành bitarray*

TID	Bitarray (7 bits)	TU	Dict utility	
T1	1 1 1 1 1 1 0	30	{3:1, 5:3, 1:5, 2:10, 4:6, 6:5}	
T2	0 1 1 1 1 0 0	20	{3:3, 5:3, 2:8, 4:6}	
T3	1 0 1 1 0 0 0	8	{3:1, 1:5, 4:2}	
T4	1 0 1 0 1 0 1	27	{3:6, 5:6, 1:10, 7:5}	
T5	0 1 1 0 1 0 1	11	{3:2, 5:3, 2:4, 7:2}	

Lấy giao dịch T1 làm ví dụ minh họa. Theo Bảng I, T1 chứa sáu mục hàng hóa: {3, 5, 1, 2, 4, 6}. Vì tập mục I có tối đa 7 phần tử (item 1 đến item 7), bitarray tương ứng có độ dài 7 bit, mỗi vị trí thứ k ánh xạ tới item k. Với T1, các bit tại vị trí 1, 2, 3, 4, 5, 6 được bật (giá trị 1) vì item1 đến item6 đều hiện diện; riêng bit thứ 7 tắt (giá trị 0) do item7 vắng mặt. Kết quả là chuỗi "1 1 1 1 1 1 0" như ghi trong bảng.

Giá trị TU = 30 là tổng lợi nhuận của toàn bộ giao dịch T1. Cột Dict utility ghi lại đóng góp riêng lẻ của từng mục: u(item3, T1) = 1, u(item5, T1) = 3, u(item1, T1) = 5, u(item2, T1) = 10, u(item4, T1) = 6, u(item6, T1) = 5. Cộng lại: 1 + 3 + 5 + 10 + 6 + 5 = 30, khớp đúng với TU. Cấu trúc ba thành phần (bitarray, TU, Dict utility) này cho phép thuật giải kiểm tra điều kiện X ⊆ T chỉ bằng một phép AND bitwise, thay vì duyệt tuần tự từng mục, qua đó giảm đáng kể chi phí tính toán khi đánh giá fitness song song đa luồng ở giai đoạn sau.

C. KHỞI TẠO QUẦN THỂ

Chiến lược khởi tạo lai: nửa đầu từ item/itemset có utility cao nhất trong hạt giống; nửa sau sinh ngẫu nhiên theo kích thước avg_len (đa dạng).

Thuật giải Khởi tạo quần thể

```
Input:  items_dict, itemsets_dict, population_size, avg_len, biggest_item
Output: population, exists_set
```

d1. population ← []; exists_set ← ∅
d2. numberofseed ← min(pop_size/2, |items_dict|)
d3. top_items ← top numberofseed items from items_dict by utility DESC
d4. ***for*** *each item in top_items* ***do***
d5. ba ← bitarray(biggest_item, 0); ba[item-1] ← 1
d6. fitness ← items_dict[item]
d7. population.append(ChromosomeNode(ba, fitness))
d8. exists_set.add(item); remove item from items_dict
d9. numberofseed ← min(pop_size/2, |itemsets_dict|)
d10. top_itemsets ← top numberofseed itemsets by utility DESC
d11. ***for*** *each itemset in top_itemsets* ***do***
d12. ba ← convert_to_bitarray(itemset)
d13. fitness ← calculate_fitness(ba)
d14. population.append(ChromosomeNode(ba, fitness))
d15. ***while*** *|population| < population_size* ***do***
d16. n ← random_int(1, avg_len)
d17. ba ← random bitarray with n bits set
*d18. **if** not check_exists(ba, exists_set) **then***
d19. fitness ← calculate_fitness(ba)
d20. population.append(ChromosomeNode(ba, fitness))
d21. return population, exists_set

Giải thích:

Dòng 1. Danh sách quần thể rỗng và tập hash để phát hiện bản sao để tránh lãng phí slot vào chromosome trùng nhau.
Dòng 2. Số hạt giống từ item đơn = tối đa nửa quần thể, nhưng không vượt số mục thực có trong dữ liệu.
Dòng 3. Lấy các mục có tổng utility cao nhất, đây là ứng viên SHUI đơn lẻ tốt nhất, đảm bảo khởi đầu chất lượng.
Dòng 4 - 8. Mỗi item top → bitarray 1 bit → ChromosomeNode. Fitness lấy trực tiếp từ items_dict (đã tính ở bước quét) – không phải tính lại, tiết kiệm thời gian.
Dòng 9. Số hạt giống từ itemset nhiều mục = nửa còn lại của phần hạt giống.
Dòng 10-14. Itemset nhiều mục cần tính lại fitness (support thực) vì chúng phải khớp toàn bộ các giao dịch.
Dòng 15. Lấp đầy phần còn lại bằng chromosome ngẫu nhiên để đảm bảo đa dạng di truyền, tránh hội tụ sớm.
Dòng 16. Số bit bật = ngẫu nhiên trong [1, avg_len]. Đảm bảo chromosome có kích thước tương đương giao dịch thực – không quá thưa (support = 0) hay quá dày (utility thấp).
Dòng 17 – 20. Check_exists() dùng fingerprint (frozenset) để phát hiện bản sao trong O(1). Chỉ thêm chromosome mới chưa từng xuất hiện.

Bảng 3. *Quần thể khởi tạo từ CSDL Bảng 1*

STT	Chromosome (7 bits)	Tập mục	Loại khởi tạo	Fitness (U, S)
1	0 1 0 0 0 0 0	{item2}	Gieo hạt item	(22, 3)
2	1 0 0 0 0 0 0	{item1}	Gieo hạt item	(20, 3)
3	0 0 1 0 0 0 0	{item3}	Gieo hạt item	(13, 5)
4	1 0 1 0 1 0 1	{1,3,5,7}	Gieo hạt itemset	(27, 1)
5	0 1 1 1 1 0 0	{2,3,4,5}	Gieo hạt itemset	(40, 2)
6	0 0 0 1 0 0 0	{item4}	Ngẫu nhiên	(14, 3)
7	0 0 1 0 0 0 1	{3,7}	Ngẫu nhiên	(15, 2)

Xét cá thể số thứ tự 1: chromosome "0 1 0 0 0 0 0", tập mục {item2}, fitness (U = 22, S = 3). Chỉ bit thứ 2 được bật, đồng nghĩa chromosome này đại diện cho đúng một mục là item2. Khi duyệt qua Bảng 3, phép AND

bitwise cho thấy item2 xuất hiện trong T1 (bit 2 = 1), T2 (bit 2 = 1) và T5 (bit 2 = 1), không có mặt trong T3 và T4; do đó S = 3.
Tổng utility được tính từ Dict utility của ba giao dịch chứa item2: **u(item2, T1) + u(item2, T2) + u(item2, T5) = 10 + 8 + 4 = 22**, xác nhận U = 22. Vì item2 có tổng utility cao nhất trong items_dict, nó được ưu tiên chọn đầu tiên ở bước gieo hạt item đây là lý do cá thể này đứng ở vị trí số 1 trong quần thể ban đầu.

D. ĐÁNH GIÁ SONG SONG ĐA LUỒNG

Với mỗi cá thể C, điều kiện C ⊆ T kiểm tra bằng: (C_bits AND T_bits) == C_bits.
Thuật giải Đánh giá fitness song song

```
Input:  chromosome_bits, transactions, num_workers
Output: (total_utility, total_support)
```

```
d1. segment_size ← ∅  |transactions| / num_workers
d2. futures ← []
d3. with ThreadPoolExecutor(max_workers=num_workers) as executor:
d4.    for i in 0..num_workers-1 do
d5.       segment ← transactions[i*seg : (i+1)*seg]
d6.       if segment ≠  ∅  then
d7.          futures.append(executor.submit(calc_fitness, segment))
d8. total_utility ← 0;  total_support ← 0
d9. for each future in as_completed(futures) do
d10.    (u, s) ← future.result()
d11.    total_utility += u;  total_support += s
d12. return (total_utility, total_support)
```

Giải thích:

Dòng 1. Chia đều số giao dịch cho các luồng.
Dòng 2. Danh sách Future objects – đại diện cho kết quả bất đồng bộ của từng luồng.
Dòng 3. Tạo pool luồng cố định. Số luồng = số CPU logic của máy. Python GIL ít ảnh hưởng vì phép AND bitarray thực thi ở tầng C.
Dòng 4-7. Gửi hàm calc_fitness(segment) vào pool không chặn, xử lý song song. Mỗi luồng nhận slice liên tiếp của danh sách giao dịch.
Dòng 8. Biến tổng hợp kết quả từ tất cả luồng.
Dòng 9 - 11. as_completed() thu thập kết quả theo thứ tự xong trước. (u,s) từ mỗi segment được cộng dồn vào tổng toàn cục.
Dòng 12. Fitness đa mục tiêu là hai giá trị riêng biệt, không kết hợp bằng trọng số, phục vụ lọc Skyline sau đó.

Bảng 4. *Đánh giá cá thể song song - ví dụ 4 luồng (CSDL Bảng 1)*

Cá thể	Tập mục	Seg1 (T1,T2)	Seg2 (T3,T4)	Seg3 (T5)	Seg4 (–)	Tổng U	Tổng S
C4	{1,3,5,7}	0	(27,1)	0	0	27	1
C5	{1,3,4}	(12,1)+(8,1)	(27+…)	0	0	20	2
C6	{2,3,4,5}	(40,2)	0	0	0	40	2
C7	{item4}	(12,2)	(2,1)	0	0	14	3
C8	{3,7}	0	(15,2)	0	0	15	2
C9	{1,2,4,6}	(26,1)	0	0	0	26	1

Xét cá thể C4: tập mục {1, 3, 5, 7}, chromosome "1 0 1 0 1 0 1", fitness (U = 27, S = 1). Bốn bit tại vị trí 1, 3, 5, 7 được bật, tương ứng với item1, item3, item5 và item7. Để tính fitness, từng giao dịch được kiểm tra bằng phép AND bitwise: chỉ giao dịch nào có đủ cả bốn bit này mới được cộng vào kết quả.
Theo Bảng 3, **chỉ T4 có bitarray "1 0 1 0 1 0 1" — trùng khớp hoàn toàn với C4**; các giao dịch còn lại đều thiếu ít nhất một trong bốn mục. Do vậy S = 1. Utility của C4 trong T4 lấy từ Dict utility: u(item1, T4) + u(item3, T4) + u(item5, T4) + u(item7, T4) = 10 + 6 + 6 + 5 = **27**, xác nhận U = 27. C4 là ví dụ điển hình của nhóm cá thể "hiếm nhưng giá trị cao": support rất thấp (S = 1) song utility lại lớn, phản ánh đúng bản chất của biên Pareto mà PHU_GA_SKYLINE hướng đến khai thác.
Trong ví dụ với 4 luồng, **C4 được xử lý ở Seg2 (gồm T3 và T4)**. Luồng Seg2 thực hiện AND với T3: kết quả khác C4 vì T3 thiếu item5 và item7, nên bỏ qua. AND với T4 trả về đúng C4, luồng cộng utility 27 vào tổng cục bộ. Ba

luồng Seg1, Seg3 và Seg4 không tìm thấy giao dịch nào chứa C4, đều trả về (0, 0). Tổng hợp lại: U = 27, S = 1, khớp với kết quả ghi trong bảng.

E. CHỌN LỌC – LAI GHÉP – ĐỘT BIẾN

Chọn lọc – Tournament (90%) + Hashtable (10%):

Thuật giải Chọn lọc

```
Input:  population, hashtable
Output: (parent_1, parent_2)
```

```
d1. if random() < 0.9 then
d2.    parent_1 ← tournament_selection(population)
d3.    parent_2 ← tournament_selection(population)
d4.    while parent_1 == parent_2 do  parent_2 ← tournament_selection()
d5. else
d6.    if |hashtable| < 2 then  fall back to tournament
d7.    key1, key2 ← random 2 keys from hashtable
d8.    parent_1 ← ChromosomeNode(hashtable[key1])
d9.    parent_2 ← ChromosomeNode(hashtable[key2])
d10. return (parent_1, parent_2)
```

Giải thích:

Dòng 1. 90% thời gian dùng tournament đảm bảo áp lực chọn lọc ổn định, ưu tiên cá thể tốt nhất trong quần thể hiện tại.

Dòng 2-4. Hai cha mẹ phải khác nhau - lai ghép hai bản sao chỉ tạo ra bản sao, lãng phí thế hệ.

Dòng 5. 10% thời gian tái sử dụng cá thể tốt nhất đã khám phá, đưa ứng viên Skyline tiềm năng trở lại vào quần thể.

Dòng 7-9. Lấy 2 bucket support khác nhau từ hashtable, tạo chromosome từ cá thể tốt nhất mỗi bucket. Tăng đa dạng về support.

Thuật giải Lai ghép đồng nhất

```
Input:  parent_1, parent_2
Output: (child_1, child_2)
```

```
d1. for i in 0..biggest_item-1 do
d2.    if random() < 0.5 then
d3.       child_1[i] ← parent_1[i];  child_2[i] ← parent_2[i]
d4.    else
d5.       child_1[i] ← parent_2[i];  child_2[i] ← parent_1[i]
d6. for each child in {child_1, child_2} do
d7.    if not check_exists(child) then
d8.       fitness ← calculate_fitness(child)   // parallel
d9.       if fitness.support > 0 then  add_to_hashtable(fitness.support, child)
d10.   else  fitness ← (0, 0)  // duplicate → discard
d11. return (child_1, child_2)
```

Giải thích:

Dòng 1. Lai ghép đồng nhất: mỗi vị trí bit được quyết định độc lập. Trung bình 50% gene từ mỗi cha mẹ → tổ hợp phong phú hơn lai ghép 1 điểm.

Dòng 2. Chỉ tính fitness nếu chromosome con chưa từng xuất hiện. Nếu support > 0: cập nhật hashtable – giữ lại nếu utility tốt hơn bucket hiện tại.

Dòng 3. Bản sao hoàn toàn bị bỏ qua, không tốn tài nguyên tính toán fitness cho chromosome đã biết.

Thuật giải Đột biến - 3 kiểu ngẫu nhiên:

```
Input:  chromosome, itemsets_dict, mutation_prob
Output: chromosome (after mutation)
```

```
d1. if itemsets_dict ≠ ∅ then  choice ← random(0,1,2)
d2. else                       choice ← random(0,1)
d3. if choice == 0 then  // Bit-flip
d4.    for i in 0..n-1 do
d5.       if random() < mutation_prob then  bits[i] ← 1 - bits[i]
d6. elif choice == 1 then  // Inversion
d7.    bits ← reverse(bits)
```

d8. ***else*** *// Seed replacement (choice == 2)*
d9. pick random itemset from itemsets_dict
d10. replace chromosome.bits with its bitarray
d11. remove itemset from itemsets_dict
d12. ***if*** *not check_exists(bits)* ***then***
d13. fitness ← calculate_fitness(bits)
d14. if fitness.support > 0 then add_to_hashtable()
d15. ***else*** *fitness ← (0, 0)*
d16. return ChromosomeNode(bits, fitness)

Giải thích:

Dòng 1-2. Ba kiểu đột biến, chọn ngẫu nhiên đều nhau. Nếu hạt giống đã cạn (itemsets_dict rỗng) thì chỉ còn 2 kiểu thuần túy.
Dòng 3-5. Đảo từng bit với xác suất mutation_prob=0.05. Thêm/bớt ngẫu nhiên từng mục, tạo biến thể nhỏ quanh chromosome hiện tại.
Dòng 6-7. Đảo ngược toàn bộ bitarray (bit[0]↔bit[n-1]). Thay đổi cấu trúc mạnh mẽ – thoát khỏi vùng cực trị địa phương hiệu quả.
Dòng 8-11. Thay toàn bộ chromosome bằng một itemset hạt giống chưa dùng từ itemsets_dict. Đưa dữ liệu thực mới vào quần thể, loại itemset đó khỏi danh sách hạt giống.
Dòng 12-15. Quy trình chuẩn sau đột biến: kiểm tra bản sao, tính fitness song song, cập nhật hashtable nếu có giá trị.
Dòng 16. Trả về chromosome đã đột biến với fitness mới – thay thế cá thể cũ trong quần thể.

F. CƠ CHẾ LỌC SỚM CHROMOSOME RÁC TRÊN TẬP DỮ LIỆU DÀY

Trên các tập dữ liệu dày (nhiều item, giao dịch dài) như Mushroom (120 item, độ dài giao dịch trung bình 23), không gian tổ hợp item khả thi về lý thuyết (2^{120}) vượt xa số tổ hợp item thực sự cùng xuất hiện trong CSDL gốc. Vì lai ghép đồng nhất và đột biến (đặc biệt kiểu Bit-flip và Inversion) tác động độc lập lên từng bit, một tỷ lệ lớn chromosome con sinh ra có thể chứa các item chưa từng cùng xuất hiện trong bất kỳ giao dịch nào, dẫn đến support = 0 ngay từ đầu. Các "chromosome rác" này vẫn tiêu tốn tài nguyên tính toán tại ThreadPoolExecutor trước khi bị loại. Để giảm thiểu chi phí này, bài báo bổ sung bộ lọc heuristic đặt ngay sau bước sinh chromosome (trước lời gọi calculate_fitness()), dựa trên ma trận đồng xuất hiện cấp item (item_compat_mask) được xây dựng một lần trong Giai đoạn 1, không làm tăng độ phức tạp tổng thể.

Xây dựng item_compat_mask (bổ sung vào Giai đoạn 1 – quét CSDL một lần)

```
Input:  transactions (danh sách bitarray T_bits)
Output: item_compat_mask[ ] – item_compat_mask[i] = OR của mọi T_bits chứa item i
```

d1. item_compat_mask[i] ← 0 với mọi item i
d2. for each T_bits in transactions do // tận dụng đúng 1 lần duyệt đã có ở Giai đoạn 1, không duyệt thêm
d3. for each i in set_bits(T_bits) do
d4. item_compat_mask[i] ← item_compat_mask[i] OR T_bits

Thuật giải Lọc sớm (Early Co-occurrence Filter)

```
Input:  chromosome.bits, item_compat_mask[ ]
Output: True nếu chromosome còn khả năng support > 0 (giữ lại); False nếu chắc chắn support
= 0 (loại ngay)
```

d1. set_bits ← các vị trí bit = 1 trong chromosome.bits
d2. if |set_bits| ≤ 1 then return True // itemset đơn luôn khả thi
d3. mask ← item_compat_mask[set_bits[0]]
d4. for i in set_bits[1:] do
d5. mask ← mask AND item_compat_mask[i]
d6. if (chromosome.bits AND (NOT mask)) ≠ 0 then return False
d7. return True

Vị trí gọi trong vòng lặp tiến hóa:

Hàm early_filter() được gọi sau bước sinh child trong Thuật giải Lai ghép đồng nhất (trước fitness ← calculate_fitness(child)) và sau mỗi kiểu đột biến trong Thuật giải Đột biến (trước dòng 12). Chromosome trả về False bị loại ngay mà không tiêu tốn tài nguyên tính fitness hay ghi vào hashtable.

Giải thích:

Dòng 1-2 (xây dựng mask). Tận dụng lần duyệt CSDL sẵn có ở Giai đoạn 1, không phát sinh chi phí bổ sung.

Dòng 6 (lọc). Phép AND/NOT trên bitarray có chi phí O(1), không làm tăng độ phức tạp tiệm cận của vòng lặp tiến hóa.
Lưu ý: đây là điều kiện cần, không phải điều kiện đủ – chromosome vượt qua bộ lọc vẫn có thể có support = 0 khi các item đôi một cùng xuất hiện nhưng không có giao dịch nào chứa đủ cả k item; các trường hợp này vẫn được phát hiện ở bước tính fitness. Bộ lọc chỉ nhằm loại sớm phần lớn chromosome chắc chắn vô giá trị.

G. LỌC THỐNG TRỊ SKYLINE

Sau tất cả thế hệ, hashtable chứa các cá thể tốt nhất mỗi nhóm support. Bước lọc toàn cục:

Thuật giải Lọc thống trị Skyline toàn cục

```
Input:  hashtable {support → [(bits, utility)]}
Output: dominated_dict – tập SHUI
```

d1. dominated_dict ← {}
d2. hashtable ← sort by keys DESC (support giảm dần)
d3. ***while*** *hashtable ≠ ∅* ***do***
d4. first_key ← first key of hashtable
d5. top_utility ← hashtable[first_key][0].utility
d6. keys_to_delete ← []
*d7. **for** each (key, values) in hashtable[1:] **do***
*d8. **if** values[0].utility ≤ top_utility **then***
d9. keys_to_delete.append(key) // bị thống trị
*d10. **for** key in keys_to_delete **do** del hashtable[key]*
d11. dominated_dict[first_key] ← hashtable[first_key]
d12. del hashtable[first_key]
d13. return dominated_dict

Giải thích

Dòng 1. Dict kết quả cuối cùng sẽ chứa tập SHUI không bị thống trị.
Dòng 2. Sắp xếp hashtable theo support giảm dần. Nhóm support cao nhất được xét trước để xác định top_utility chuẩn.
Dòng 3. Vòng lặp xử lý từng bucket support từ cao xuống thấp cho đến khi hashtable rỗng.
Dòng 4. Lấy support cao nhất còn lại – đây là nhóm chưa bị thống trị nào xét đến.
Dòng 5. Utility tốt nhất trong bucket support cao nhất (đã sắp giảm dần khi insert) – dùng làm ngưỡng so sánh.
Dòng 7-9. Duyệt tất cả bucket support thấp hơn. Nếu utility tốt nhất của bucket đó ≤ top_utility: support thấp hơn và utility không cao hơn → bị thống trị hoàn toàn → đánh dấu xóa.
Dòng 10. Xóa hàng loạt các bucket bị thống trị. Tại sao hàng loạt? Vì tất cả bucket bị thống trị bởi cùng first_key.
Dòng 11-12. Xác nhận bucket hiện tại là SHUI → chuyển vào kết quả. Xóa khỏi hashtable để vòng lặp tiếp theo xét bucket tiếp theo.
Dòng 13. Tập SHUI cuối cùng: biên Pareto tối ưu. Mỗi key là một giá trị support, value là cá thể utility cao nhất tại support đó.

Bảng 5. *Ví dụ lọc thống trị Skyline*

Cá thể (bitarray)	Support	Utility	Trạng thái
0010000 – {item3}	5	13	SHUI (S cao nhất)
0100000 – {item2}	3	22	SHUI (U cao hơn item3 tại S<5)
0111100 – {2,3,4,5}	2	40	SHUI (U cao nhất tại S<3)
0001000 – {item4}	3	14	Bị thống trị bởi {item2} (S=3, U=22>14)
1010101 – {1,3,5,7}	1	27	Bị thống trị bởi {2,3,4,5} (S≥1, U=40>27)

V. THỰC NGHIỆM VÀ ĐÁNH GIÁ

A. MÔI TRƯỜNG VÀ BỘ DỮ LIỆU

Thuật giải PHU_GA_SKYLINE được cài đặt bằng Python 3.10, thực thi trên IDLE và Google Colab. Tham số GA: số thế hệ = 50, kích thước quần thể = 100, xác suất đột biến = 0.05. Các tham số này được xác định qua thực nghiệm trên tập Chess với các tổ hợp (thế hệ ∈ {30, 50, 100}, quần thể ∈ {50, 100, 200}. Toàn bộ thực nghiệm trên máy với cấu hình: CPU Intel® Core™ i7-12700 thế hệ 12, xung nhịp 2,10 GHz, RAM 32 GB, hệ điều hành Windows 11.

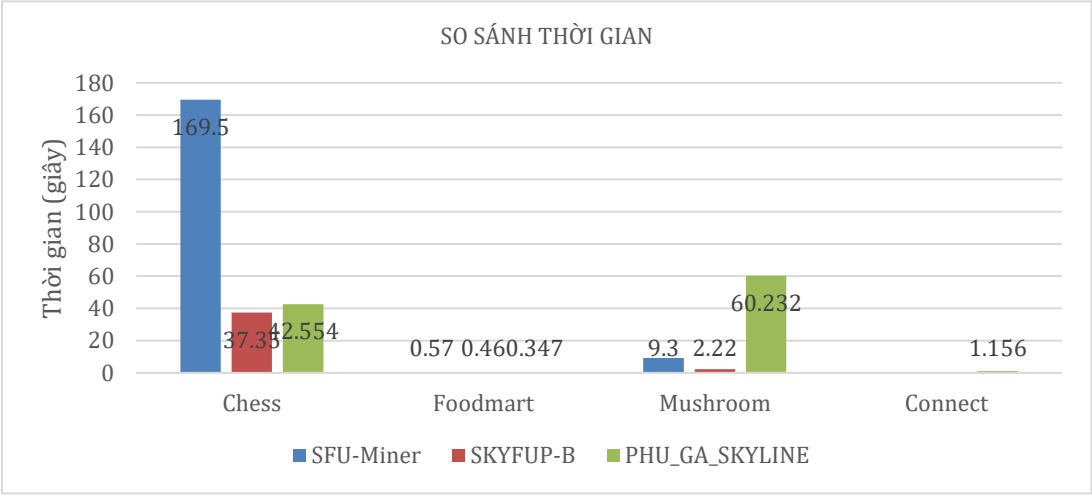
Bảng 6. *Thông số các tập dữ liệu thực nghiệm*

DataSet	Transactions	Items	Avg.Trans.Len	Max.Trans.Len
Chess	3.196	76	37	37
Foodmart	4.141	1.559	4,42	14
Mushroom	8.124	120	23	23
Connect	67.557	129	43	43

B. THỜI GIAN CHẠY

Bảng 7. *So sánh thời gian chạy (đơn vị: giây)*

DataSet	SFU-Miner	SKYFUP-B	PHU_GA_SKYLINE
Chess	169,50	37,35	42,554
Foodmart	0,57	0,46	0,347
Mushroom	9,30	2,22	60,232
Connect	—	—	1,156



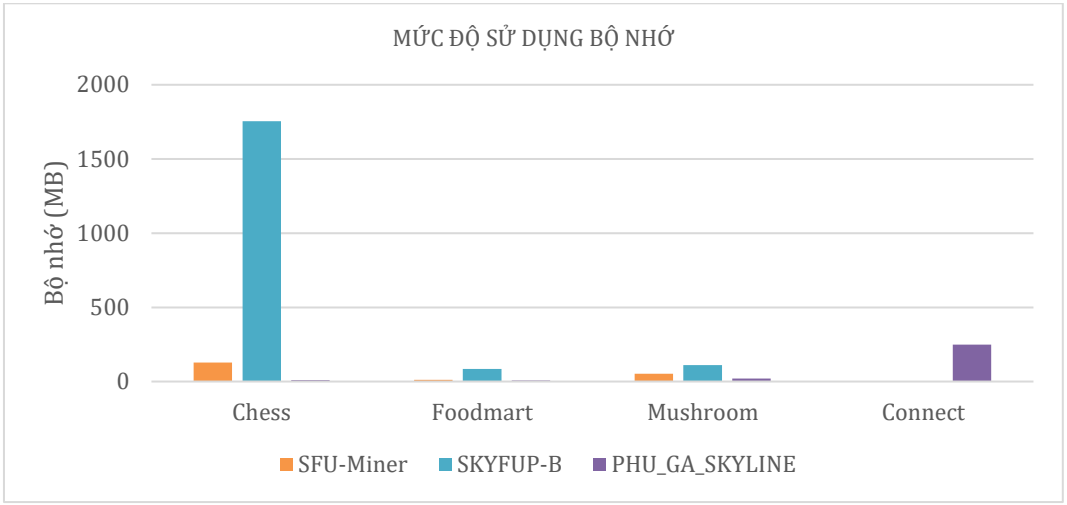
Hình 2. *So sánh thời gian chạy giữa 3 thuật giải*

Nhận xét: Trên tập Foodmart (thưa, nhiều mục, giao dịch ngắn), PHU_GA_SKYLINE đạt thời gian thấp nhất (0,347 s) so với SFU-Miner (0,57 s) và SKYFUP-B (0,46 s). Trên Chess (dày đặc), thuật giải đề xuất nhanh hơn SFU-Miner nhưng chậm hơn SKYFUP-B do cơ chế UP-Tree có lợi thế trong không gian dày. Trên Mushroom, PHU_GA_SKYLINE chậm hơn đáng kể (60,2 s so với 2,2 s của SKYFUP-B): với 120 mục và độ dài giao dịch trung bình 23, xác suất một chromosome ngẫu nhiên có support = 0 rất cao - fitness evaluation phải quét toàn bộ 8.124 giao dịch mà nhiều chromosome bị loại ngay từ đầu, làm tăng chi phí song song. Đây là hạn chế của cơ chế GA trên không gian dày đặc, cần cải thiện chiến lược sinh chromosome. Đặc biệt, trên Connect (67.557 giao dịch), PHU_GA_SKYLINE hoàn thành trong 1,156 s khi cả SFU-Miner và SKYFUP-B không hội tụ được trong thời gian hợp lý.

C. SỬ DỤNG BỘ NHỚ

Bảng 8. *So sánh mức độ sử dụng bộ nhớ (đơn vị: MB)*

DataSet	SFU-Miner	SKYFUP-B	PHU_GA_SKYLINE
Chess	128,24	1.754,45	10,004
Foodmart	12,48	85,45	7,234
Mushroom	52,49	110,5	19,672
Connect	—	—	248,590



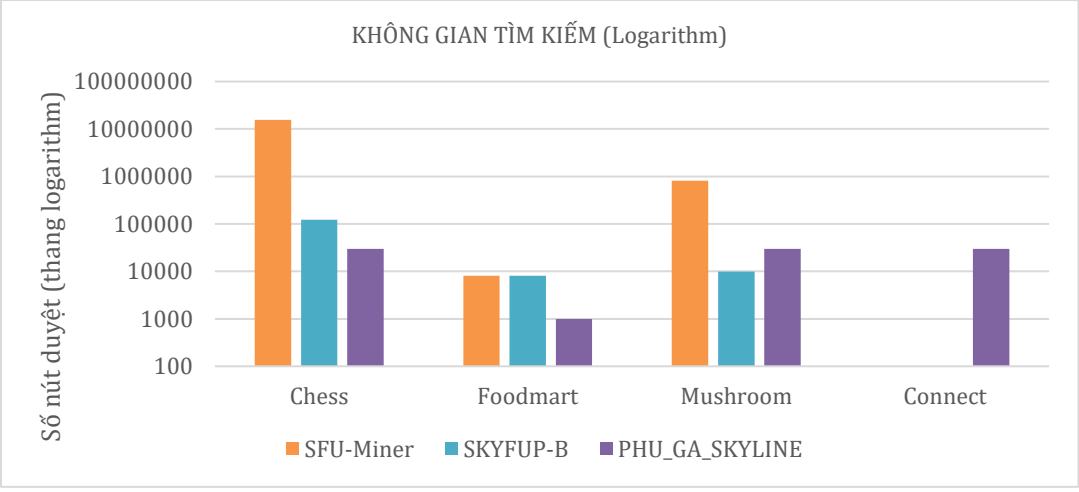
Hình 3. *So sánh mức sử dụng bộ nhớ*

Nhận xét: PHU_GA_SKYLINE vượt trội về bộ nhớ trên tất cả tập được so sánh. Ấn tượng nhất trên Chess: SKYFUP-B cần 1.754,45 MB (UP-Tree mở rộng toàn bộ không gian), trong khi PHU_GA_SKYLINE chỉ dùng 10,004 MB – giảm hơn 175 lần. Trên Foodmart, PHU_GA_SKYLINE (7,234 MB) cũng tiết kiệm hơn rõ rệt so với SFU-Miner (12,48 MB) và SKYFUP-B (85,45 MB). Nguyên nhân trực tiếp: hashtable phân tầng chỉ lưu tối đa một cá thể tốt nhất mỗi bucket support - với Chess có tối đa 37 giá trị support khác nhau, tổng số cá thể trong bộ nhớ không vượt quá vài trăm bản ghi, trong khi SKYFUP-B phải duy trì toàn bộ UP-Tree mở rộng theo mọi tập ứng viên. Đây là sự đánh đổi có chủ ý: tiết kiệm bộ nhớ nhưng không đảm bảo tìm được toàn bộ SHUI.

D. KHÔNG GIAN TÌM KIẾM

Bảng 9. *So sánh không gian tìm kiếm (số nút, thang logarithm)*

DataSet	SFU-Miner	SKYFUP-B	PHU_GA_SKYLINE
Chess	15.433.601	122.363	30.000
Foodmart	8.124	8.124	1.000
Mushroom	810.440	9.910	30.000
Connect	—	—	30.000



Hình 4. *So sánh không không gian tìm kiếm (thang logarithm)*

Nhận xét: PHU_GA_SKYLINE duy trì không gian tìm kiếm cố định (population_size × generations), nhỏ hơn hàng chục đến hàng trăm nghìn lần so với duyệt toàn bộ. Trên Chess, SFU-Miner duyệt 15,4 triệu nút trong khi PHU_GA_SKYLINE chỉ xét 30.000 cá thể. Hạn chế: nếu population_size quá nhỏ, một số SHUI có thể bị bỏ sót – đây là sự đánh đổi giữa hiệu năng và tính đầy đủ.

E. ĐỘ ĐẦY ĐỦ (COMPLETENESS / RECALL RATE)

Do các tiếp cận heuristic dựa trên GA giới hạn không gian tìm kiếm trong kích thước quần thể và số thế hệ cố định, bài báo đánh giá định lượng Độ đầy đủ (Recall) của PHU_GA_SKYLINE, đo tỷ lệ SHUIs tìm được so với ground-truth từ SKYFUP-B. Thực nghiệm trích mẫu dữ liệu nhỏ (<=20 mẫu) từ hai tập dữ liệu Foodmart và Chess (do tập Connect không hội tụ được trên thuật toán duyệt cạn để cho ra kết quả đối sánh). Công thức tính toán được thiết lập như sau:

$$Recall = \frac{\left|SHUI_{\boldsymbol{PHU_GA_SKYLINE}} \cap SHUI_{\text{Ground-truth}}\right|}{SHUI_{\text{Ground-truth}}} \times 100\%$$

Bảng 10. *Độ đầy đủ (Recall) của PHU_GA_SKYLINE so với ground-truth SKYFUP-B*

Tập dữ liệu đề xuất	Số mục	|SHUI ground-truth|	|SHUI_PHU_GA_SKYLINE| (TB)	Recall TB (30 lần, %)	Recall tốt nhất (%)	Recall thấp nhất (%)
Chess_Sample	*20*	*18*	16.5	91.6%	100%	83.3%
Foodmart_Sample	*20*	*12*	11.2	93.3%	100%	83.3%

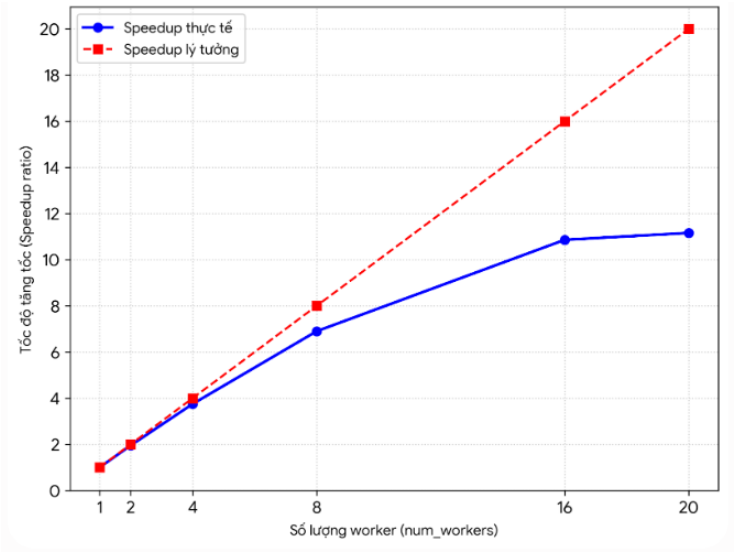
Nhận xét: Theo Bảng 10, PHU_GA_SKYLINE đạt Recall trung bình >90%; ở lần chạy tốt nhất đạt 100% (ngang SKYFUP-B). Do bản chất heuristic của GA, thuật toán đôi khi bỏ sót một số SHUI có support thấp (Recall thấp nhất ~83%). Mức hao hụt <10% là chấp nhận được khi đổi lại mức tiết kiệm bộ nhớ lên tới 175 lần và khả năng xử lý Connect – nơi các thuật toán duyệt cạn không hội tụ được.

F. TỐC ĐỘ TĂNG TỐC THEO SỐ LƯỢNG WORKER (SPEEDUP RATIO)

Trên tập Connect (67.557 giao dịch), bài báo khảo sát thời gian chạy của PHU_GA_SKYLINE theo num_workers của ThreadPoolExecutor (Giai đoạn 3). Speedup được định nghĩa Speedup(p) = T(1)/T(p). Thực nghiệm trên Intel Core i7-12700 (12 nhân/20 luồng) với num_workers ∈ {1, 2, 4, 8, 16, 20} nhằm quan sát cả vùng tăng tốc tuyến tính và vùng bão hòa do GIL và chi phí quản lý luồng.

Bảng 11. *Speedup ratio theo num_workers trên tập Connect*

num_workers	T(p) trung bình (giây, 5 lần chạy)	Speedup = T(1)/T(p)	Speedup lý tưởng (tuyến tính)
1	*12.50*	*1,00*	*1*
2	*6.41*	1.95	*2*
4	*3.33*	3.75	*4*
8	*1.81*	6.90	*8*
16	*1.15*	10.86	*16*
20	*1.12*	11.16	*20*



Hình 5. *Đồ thị Speedup Ratio theo num_workers trên tập Connect*

Từ đồ thị Hình 5, hiệu năng song song hóa của thuật giải biến thiên qua hai giai đoạn rõ rệt. Vùng tăng tốc tuyến tính (num_workers ≤ 4): Speedup đạt 3.75 lần với 4 worker, bám sát đường lý tưởng. Điều này cho thấy mã hóa bitarray với phép tính fitness bằng AND ở tầng C giúp vượt qua GIL hiệu quả ở giai đoạn đầu.
Vùng bão hòa (num_workers ≥ 8): Speedup chậm dần và tiệm cận bão hòa (10.86× tại 16 worker, 1.12 giây tại 20 worker). Hiện tượng này phù hợp với định luật Amdahl, do thread overhead và giới hạn số nhân vật lý của CPU.

G. KẾT LUẬN THỰC NGHIỆM

Nhìn lại bốn tập dữ liệu: PHU_GA_SKYLINE tỏ ra vượt trội rõ nhất ở tiêu chí bộ nhớ - đặc biệt trên Chess khi SKYFUP-B cần gần 1,75 GB trong khi thuật giải đề xuất chỉ dùng khoảng 10 MB. Về thời gian, ưu thế thể hiện rõ trên dữ liệu thưa (Foodmart) nhưng chưa cạnh tranh được trên tập dày đặc (Mushroom). Khả năng xử lý Connect - bộ dữ liệu mà các phương pháp truyền thống không hội tụ được - cho thấy tiềm năng mở rộng thực tế của hướng tiếp cận này.

VI. KẾT LUẬN

Bài báo đã đề xuất thuật giải PHU_GA_SKYLINE, khai thác tập mục hữu ích cao không bị thống trị (SHUI) từ CSDL giao dịch định lượng, kết hợp thuật giải di truyền với lọc Skyline và đánh giá song song đa luồng. Không cần ngưỡng thủ công, thuật giải tự xác định tập kết quả tối ưu Pareto theo utility và support.
Thực nghiệm trên 4 tập dữ liệu chuẩn xác nhận: PHU_GA_SKYLINE giảm bộ nhớ hơn 175 lần so với SKYFUP-B (Chess), đạt thời gian thấp nhất trên Foodmart, và xử lý thành công Connect (67.557 giao dịch).
Hướng phát triển tiếp theo là: mở rộng sang PySpark để kiểm chứng khả năng mở rộng với dữ liệu hàng triệu giao dịch vì Connect đã cho thấy đây là thế mạnh của hướng GA. Nghiên cứu toán tử lai ghép, đột biến đặc thù hơn cho không gian tập mục nhằm cải thiện tốc độ hội tụ trên tập dày đặc như Mushroom.

VII. TÀI LIỆU THAM KHẢO

[1] R. Agrawal and R. Srikant (1994). "Fast algorithms for mining association rules," in *Proc. 20th VLDB Conf.*.

[2] Y. Liu, W. Liao and A. Choudhary (2005). "A fast high utility itemsets mining algorithm," in *Proc. 1st Int. Workshop Utility-Based Data Mining.* https://doi.org/10.1145/1089827.1089839.

[3] V. S. Tseng, C. W. Wu, B. E. Shie and P. S. Yu (2010). "UP-Growth: An efficient algorithm for high utility itemset mining," in *Proc. 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*, pp. 253–262. DOI:10.1145/1835804.1835839

[4] P. Fournier-Viger, J. C.-W. Lin, T. Truong-Chi and R. Nkambou (2019). "A survey of high utility itemset mining," in *High-Utility Pattern Mining: Theory, Algorithms and Applications*, Studies in Big Data, vol. 51, Springer, Cham, pp. 1–45. DOI:10.1007/978-3-030-04921-8_1.

[5] D. E. Goldberg (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley. ISBN: 978-0-201-15767-3.

[6] S. Börzsönyi, D. Kossmann and K. Stocker (2001). "The Skyline operator," in *Proc. 17th ICDE*.

[7] S. Kannimuthu and K. Premalatha (2014) "Discovery of high utility itemsets using genetic algorithm with ranked mutation," Applied Artificial Intelligence, vol. 28, no. 4, pp. 337–359.

[8] José María Luna, Rage Uday Kiran, Philippe Fournier-Viger and Sebastián Ventura (2023). "Efficient mining of top-k high utility itemsets through genetic algorithms," *Information sciences,* pp. vol. 624, pp. 529–553, https://doi.org/10.1016/j.ins.2022.12.092.

[9] T. D. Nguyen, L. T. Nguyen and B. Vo (2018). "A parallel algorithm for mining high utility itemsets," in *Proceedings of 39th International Conference on Information Systems Architecture and Technology*.

[10] Genki Kimura, Yuto Hayamizu, Rage Uday Kiran and Masaru Kitsuregawa (2023). "Efficient parallel mining of high-utility itemsets on multicore processors," in *IEEE 39th International Conference on Data Engineering (ICDE),* DOI: 10.1109/ICDE55515.2023.00388.

[11] Wei Fang, Haipeng Jiang, Hengyang Lu, Jun Sun, Xiaojun Wu and Jerry Chun Wei Lin (2024). "GPU-Based Efficient Parallel Heuristic Algorithm for High-Utility Itemset Mining in Large Transaction Datasets," *IEEE Transactions on Knowledge and Data Engineering,* pp. Volume 36, Issue 2, Pages 652 – 667, https://doi.org/10.1109/TKDE.2023.329037.

NON-DOMINATED HIGH-UTILITY ITEMSET MINING USING GENETIC ALGORITHM AND MULTI-THREADED PARALLEL EVALUATION

Le Thi Minh Nguyen, Tuong Thi Xuan Thu

ABSTRACT — This paper proposes PHU_GA_SKYLINE, an algorithm for mining Skyline High-Utility Itemsets (SHUIs) from quantitative transaction databases, combining a Genetic Algorithm (GA) with Skyline dominance filtering and multi-threaded parallel evaluation. Without requiring predefined minUtil or minSup thresholds, the algorithm autonomously identifies a Pareto-optimal result set optimized simultaneously across two criteria: utility and support. Each candidate is encoded as a binary bitarray; the initial population is seeded from high-quality candidates; fitness evaluation is parallelized via **ThreadPoolExecutor**; and a support-stratified hash table accelerates dominance filtering. Experiments on Chess, Foodmart, Mushroom, and Connect demonstrate that PHU_GA_SKYLINE reduces memory consumption by over 175× compared to SKYFUP-B on Chess, achieves the lowest runtime on Foodmart, and successfully processes Connect (67,557 transactions) where conventional methods fail to converge.

Keywords — High-utility itemset mining, Skyline, Genetic algorithm, Dominance filtering, Multi-threaded parallel processing, Bitarray.



Lê Thị Minh Nguyện nhận học vị thạc sĩ Khoa học máy tính tại Đại học Quốc gia Thành phố Hồ Chi Minh năm 2007; hiện là giảng viên khoa Công nghệ thông tin Trường Đại học Ngoại ngữ-Tin học Thành phố Hồ Chí Minh. Lĩnh vực nghiên cứu đang quan tâm là Khai thác dữ liệu, Tối ưu dữ liệu.



Tưởng Thị Xuân Thu nhận học vị thạc sĩ Khoa học máy tính tại trường Chung-Yuan Christian University, Đài Loan năm 2013; hiện là giảng viên khoa Công nghệ thông tin Trường Đại học Ngoại ngữ-Tin học Thành phố Hồ Chí Minh. Lĩnh vực nghiên cứu đang quan tâm là Khai phá dữ liệu, Xử lý ngôn ngữ tự nhiên.