TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN
MÔN DATA MINING

ĐÀO MINH TOÀN – 1512581

ACCEPTED MANUSCRIPT

BÀI BÁO KHOA HỌC

GIÁO VIÊN HƯỚNG DẪN

Lê Hoài Bắc

KHÓA 2015-2019

Contents

I Tổng quan

Khai phá tần số của "itemset" là một nhiệm vụ cơ bản trong khai phá dữ liệu. Có rất nhiều các ứng dụng sử dụng việc khai phá như trên. Trong những năm gần đây, một số cấu trúc dữ liệu được biểu diễn dựa vào cấu trúc "node" trên cây tiền tố. Các cấu trúc dữ liệu đó chứa thông tin cần thiết về tần số của "itemset".

Trong bài báo này,chúng ta đưa ra một dạng cấu trúc dữ liệu hiệu quả – NegNodeset.Cũng giống như mọi kiểu cấu trúc dữ liệu, NegNodeset bao gồm một tập các node trong cây tiền tố dựa vào các tập "bitmap" được biểu diễn. Dựa vào cấu trúc của NegNodeSet,chúng ta đề xuất ra negFin-một thuật toán hiệu quả cho việc khai

phá tần số "itemset". Hiệu quả của việc sử dụng thuật toán negFIN được xác nhận bởi 3 lý do sau :

- Một là NegNodesets của các itemsets được trích xuất bằng cách sử dụng các toán tử bitwise
- Hai là độ phức tạp của NegNodesets và các toán tử đếm được giảm đến O(n) với n là số lượng NegNodesets
- Ba là nó sử dụng cây truy vấn để tạo ra tần số của các "itemsets" và sử dụng một phương pháp tiến bộ để loại bỏ bớt đi không gian tìm kiếm trên cây. Việc mở rộng nghiên cứu trên số lượng lớn cơ sở dữ liệu chuẩn của chúng tôi đã cho thấy rằng negFin là thuật toán nhanh nhất so với các thuật toán hiện đại trước đây.Tuy nhiên, thuật toán này chạy với tốc độ cùng bằng với tốc độ của dFIN trên một vài tập dữ liệu.

1 Giới thiệu

Việc khai phá tần số của "itemsets" có nhiều ứng dụng như : khám phá ra được một số quy tắc liên kết, nhóm và phân loại dữ liệu.Ban đầu thì nó được sử dụng đểphân tích giỏ hàng trong thị trường và lần đầu tiên được đề xuất trong (Agrawal,Imielinski, &Swami,1993). Mục đích là tìm ra các mục trong cơ sở dữ liệu giao dịch của khách hàng những món hàng thường được mua cùng với nhau.

a Một số định nghĩa

Giả sử $I = \{i_1, i_2 .. i_{nit}\}$ là tập hợp tất cả các item trong tập chuyển đổi dữ liệu, một phép biển đổi T là tập các items ($T \subseteq I$ với một TID xác định duy nhất và một cơ sở dữ liệu $DB = \{T_1, T_2, \dots . T_{nt}\}$ là tập các phép biến đổi. Với mỗi P sao cho $P \subseteq I$ được gọi là một "itemset". Khi đó P cũng được gọi là k-itemset với $|P| = k$. Một phép biến đổi T chứa một itemset P khi và chỉ khi $P \subseteq T$. Giá trị support của P(kí hiệu là support(P)) được xác định là tỉ lệ các phép biến đổi trong DB chứa P. Ngưỡng giá trị min-support do người dùng quy định. P còn được gọi là tần ố của itemset khivà chỉ khi $\min - \text{support} \leq \text{support(P)}$. Cho cơ sở dữ liệu DB và ngưỡng giá trị min-support, việc tìm ra tần số các itemset được định nghĩa như việc khai phá tất cả tần số của các itemsets với giá trị support của chúng. Số lượng của các itemsets được kiểm tra để tìm ra tần số của itemset là 2^{nit} với $nit = |I|$. Vì vậy, việc tìm ra tần số của itemset chính là NP

b Đông lực và sự đóng góp

- Khai phá tần số của itemset là một chủ đề nghiên cứu rất nóng trong lĩnh vực khai phá dữ liệu trong hai thập kỉ cuối này. Trong những năm gần đây, 4 kiểu cấu trúc dữ liệu dựa vào tập các "nodes" trong cây tiền tố được đưa ra để tăng cường hiệu quả trong việc tìm ra tần số của các itemsets. Đó là Node-list, N-list, Nodeset và DiffNodeset. Tất cả các cấu trúc dữ liệu sử dụng cây tiền tố với các nodes mã hóa

và liên kết giữa các tập nodes với mỗi itemset. Những node có trong Node-list và N-list được mã hóa bởi thứ tự sắp xếp tăng bậc của các nodes. Hai thuật toán: PPV và PrePost được đề xuất để tìm ra tần số các itemset dựa vào hai cấu trúc dữ liệu. Hai thuật toán trên làm tốt hơn những thuật toán trước đó. Tuy nhiên, chúng có một hạn chế: sử dụng quá nhiều bộ nhớ. Để khắc phục được vấn đề này, Nodeset được đề đưa ra. Không giống như N-list và Node-list, các nodes trong một Nodeset được mã hóa chỉ bởi duyệt tiền tố(hoặc duyệt hậu tố) thứ tự xếp hạng của các node. Nodeset của mỗi k-itemset ($k \geq 3$) được trích ra bởi phép giao của Nodesets của 2 tập (k-1) itemset. Thuật toán FIN được đề xuất dựa vào cấu trúc này. Tac hại của Nodeset đó chính là nhiều yếu tố trong Nodeset trở nên rất lớn trong cơ sở dữ liệu. Để khắc phục hạn chế này, một cấu trúc dữ liệu khác đã được sử dụng: DiffNodeset. Không như Nodeset, mỗi k-itemset ($3 \leq k$) trong DiffNodeset được trích ra bởi 2 (k-1)itemsets khác nhau. Và theo kết quả thực nghiệm, DiffNodeset nhỏ hơn so với Nodeset. Chính vì thế, thuật toán dFIN dựa vào cấu trúc này được đề xuất sử dụng chạy nhanh hơn so với các thuật toán trước của nó.

- Dù cho những thuận lợi mà DiffNodeset mang lại, chúng ta thấy rằng việc tính toán sự khác nhau giữa 2 DiffNodeset mất khá nhiều thời gian trên một số cơ sở dữ liệu. Để khắc phục tình trạng này, NegNodeset – thuật toán dựa vào cây tiền tố cũng như những cấu trúc dữ liệu khác được đưa ra. Tuy nhiên, NegNodeset sử dụng mô hình mã hóa nodes mới dựa vào bitmap để biểu diễn các tập, Giả sử xét tập U với n thành phần, chúng ta sẽ biểu diễn các tập con của U bởi bitmap có size n. Mỗi thành phần của U được đánh dấu là một trong các bits trong bitmap. Nếu có một phần tử của tập con S($S \subseteq U$) được đánh dấu là 1 thì các bits còn lại là 0.
- Dựa vào cấu trúc của NegNodeset, negFIN, một thuật toán tìm tần số của itemset khá nhanh được đề xuất sử dụng. Điểm mạnh của thuật toán này là: Một là NegNodeset mới được tạo ra bởi các toán tử bitwise, chính vì vậy nó sẽ chạy rất nhanh. Hai là việc tạo ra một NegNodeset mới và các phép đếm có độ phức tạp là O(n) thay vì O(m+n) trong các thuật toán trước đó, Ba là nó sử dụng tập cây đếm để tạo ra tần số của itemsets và sử dụng một số phương thức giảm bớt không gian tìm kiếm của cây.

c Biểu diễn negFIN

❖ Bằng việc thực hiện một số thực nghiệm nghiên cứu để biểu diễn thuật toán negFIn, cần phải so sánh thuật toán negFIN và dFIn. Kết quả thực nghiệm đã chỉ ra rằng negFIN biểu diễn tốt hơn và chạy nhanh hơn với mọi tập dữ liệu. Ngoài ra, với một vài tập dữ liệu khác nó chạy nhanh hơn rất nhiều so với FP-growth, dFIN.

II Các vấn đề liên quan

- Các thuật toán để tìm ta tần số của các itemsets được chia thành 2 loại chính: Một là các thuật toán sử dụng phương thức "candidate generation" và hai là các thuật toán sử dụng phương thức "pattern growth".
- Trong phương thức sử dụng candidate generation, các thành phần itemset được tạo ra đầu tiên, và sau đó, tần số của chúng được xác định từ các candidate itemsets. Phương thức này không sử dụng các thành phần đơn điệu được gọi là Apriori để giảm không gian tìm kiếm. Nếu một itemsets không có tần số thì super-itemset của nó cũng vậy.
- Không giống như candidate generation, phương thức "pattern growth" không tạo ra các thành phần itemsets và tránh quét nhiều database cùng một lúc bằng cách chứa thông tin về tần số của các itemset trong một cấu trúc dữ liệu đặc biệt. Điều cơ bản của thuật toán này đó chính là các danh mục trong thuật toán FP-growth. Nó chứa các thông tin về tần số các itemset trong một cây dữ liệu có tên là FP-tree. Tương tự như FP-growth và các thuật toán khác, việc sử dụng các phương thức mẫu để tìm ra tần số của itemsets. Mặc dù vậy chúng có những nhược điểm nhất định: Một là các dư liệu sẽ rỗng và thưa và hai là cấu trúc dữ liệu sử dụng bởi các mẫu trong thuật toán này khá phức tạp.
- Trong những năm gần đây, có 4 loại cấu trúc dữ liệu dựa vào cây tiền tố được đề xuất sử dụng để chứa thông tin về tần số của các itemset: Node-list, N-list, Nodeset và DiffNodeset. Cả Node-list và N-list đều dựa vào cấu trúc của cây PPC, một loại cấu trúc dữ liệu mã hóa các node bằng phép duyệt tiền tố và hậu tố. Node-list và N-list của một itemset là một tập các nodes trong cây PPC. N-list có 2 thuận lợi hơn so với Node-list: thứ nhất là số nhiều của N-list của một itemset nhỏ hơn rất nhiều so với số lượng lớn của Node-list và thứ hai là N-list sử dụng "single path property" để trực tiếp tìm ra tần số của các itemsets mà không cần tạo ra những thành phần khác trong một số trường hợp. Hai thuật toán PPV và PrePost được đề xuất dựa vạo Node-list và N-list. Trong những năm gần đây đã sử dụng các kĩ thuật giảm bớt/cắt tỉa nhằm tăng sư biểu diễn cho PrePost. Mặc dù có rất nhiều thuận lợi khi sử dụng Node-list và N-list, nhưng chúng phải sử dụng một không gian bộ nhớ rất lớn, vì vậy, cần phải chứa phép duyệt tiền tố và hậu tố của các nodes. Để khắc phục tình trạng này, Nodeset được đề xuất sử dụng. Thuật toán này chỉ cần lưu một trong các phép duyệt tiền tố hoặc hậu tố của các nodes.
- Năm 2014, thuật toán FIN được đề xuất sử dụng dựa trên cấu trúc của Nodesets. Tuy nhiên, Nodeset cũng có một số hạn chế: một số lượng các lớn các itemset trở nên rất lớn trong một số cơ sở dữ liệu, chính vì thế, DiffNodeset tiếp tục được đề xuất sử dụng.
- Thêm vào đó, PUN – list, một thuật toán được công bố 2018 cũng dựa vào cấu trúc dữ liệu trên được đề xuất để khai phá "high utility itemset", một mục tiêu khác trong việc khai phá dữ liệu. Với mỗi item có giá trị hữu ích và có thể được biến đổi nhiều hơn

một lần. Sự tiện lợi của một tập itemset đó chính là sự tiện dụng của nó không nhỏ hơn ngưỡng support. Hơn thế nữa, để lưu một thông tin về tần số của itemsets, PUN-list cũng chứa thông tin về các các tiện ích. Thuật toán MIP được đề xuất cho việc khai thác ra tính tiện lợi của các itemset dựa vào PUN-list.

❖ Dưới đây là một số ví dụ

Ví dụ 1: Xét một tập biến đổi trên cơ sở dữ liệu được cho trong Table 1 dưới đây, ngưỡng min-support = 0.4. Với Table 1, cột thứ 1 là ID(TID), cột thứ hai là các item trong mỗi phép biến đổi và cột thứ ba chính là tần số xuất hiện của mỗi item trong mỗi phép biến đổi đã được sắp xếp theo giá trị $support(\alpha)$ với α là một item

TID	items	Ordered frequent items
1	e, b, g, d	b, d, e
2	c, e, b, a	a, b, c, e
3	c, b, a, i	a, b, c
4	a, d, h	a, d
5	a, d, c, b, f	a, b, c, d

Định nghĩa 1: $\forall i_1, i_2 \in F_1$(tập các tần số của items), $i_2 \succ i_1$ nếu và chỉ nếu $support(i_2) \geq support(i_1)$

Định nghĩa 2: Cho F_1, L_1 là các vector không dựa trên sự sắp xếp tần số của các items, với các items được sắp xếp không giảm bởi giá trị $support(\alpha)$ – α là một item L_1 được kí hiệu $L_1 = [i_0, i_1, \ldots . i_{nf-1}]$ với $nf = |F_1|$ và $i_{nf-1} \succ \cdots \succ i_1 \succ i_0$. Một k-itemset P được kí hiệu $P_k = i_k \ldots i_2 i_1$ hoặc $P_k = i_k P_{k-1}$ với $i_k \succ \cdots \succ i_2 \succ i_1$ và $P_{k-1} = i_{k-1} \ldots i_2 i_1$

Định nghĩa 3: $P_k = i_k P_{k-1} (2 \leq k)$. P_k được xác định bởi $P_k = \lceil i_k P_{k-1}$ với $\lceil i_k$ có không có trong item i_k

Định nghĩa 4: (index(item i)) Với bất kì item i, j ϵ L_1, index(i) được xác định bằng vị trí của item i trong L_1

Support → 0.4 , 0.6 , 0.6 , 0.8 , 0.8
L_1 = [e , d , c , b , a]
index → *0 , 1 , 2 , 3 , 4*

Figure 1 The zero-based vector L_1 , and the index of each frequent item in Example 1.

Định nghĩa 5: (BMC(itemset P_k) - bitmap code của mỗi itemset) Mỗi itemset P_k có thể biểu diễn bởi một bitmap code $BMC(P_k) = b_{nf-1} \ldots b_1 b_0$ của độ lớn nf được tính như sau: j^{th} item trong vector L_1 được đánh dấu đến bit b_j trong bitmap này. Nếu mỗi

item i ($i \in L_1$) là một thành phần của P_k, nó sẽ mang giá trị 1, ngược lại mang giá trị 0.

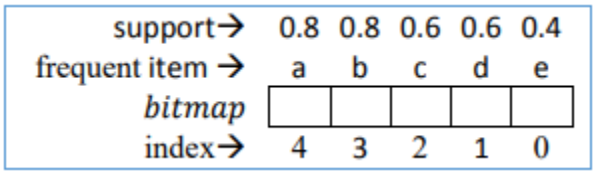


Figure 2 The bit assigned to each frequent item for Example 1.

Định nghĩa 6: (BMC-tree) một cây BMC là cây thõa mãn:

- Nó là node root rỗng và có một số cây con item tiền tố như thể con của nó.
- Mỗi node trong mỗi cây con item tiền tố lưu giữ một item i ($i \in L_1$). Nếu cha của node này biểu diễn item j thì $j \succ i$. Và đường đi đến node này biểu diễn bởi *node-path*
- Mỗi node có 4 thuộc tính: *item-name, count, bitmap-code, children-list*. *Item-name* lưu giữ item i ($i \in L_1$). *count* lưu số lượng phép biến đổi bao gồm itemset *node-path*. *bitmap-code* lưu *BMC(node-path)* và *children-list* lưu tất cả các con của node này

Định nghĩa 7: (phần chính và phân không quan tâm của BMC(node-path) *Node N* lưu giữ một item i_1 , $N.node-path = i_k \dots i_2 i_1$ với $i_k \succ \cdots \succ i_2 \succ i_1$, $BMC(node-path) = b_{nf-1} \dots b_1 b_0$ và $item\ i_1$ được đánh dấu đến $b_m (m = index(i_1))$. Các bit $b_{nf-1} \dots b_m$ được xác định bởi phần chính của $BMC(node-path)$ và các bits $b_m \dots b_1 b_0$ được xác định bởi các phần không quan tâm (*don't-care section of BMC(node-path)*)

Property 1: Giá trị bit trong *don't-care section of BMC* có giá trị *don't-care*, vì vậy chúng mang giá trị 0

Property 2: bit 0 trong phần chính của *BMC (node-path)* có nghĩa là item không tồn tại trong node-path

Property 3: Mọi bit trong *bitmap-code* có trong root của cây BMC đều là 0

Property 4: Giả sử *node N* lưu item i_1 và một node N.father là node cha của nó, *N.bitmap-code = N.father.bitmap* - $code\ V\ 2^{index(i_1)}$

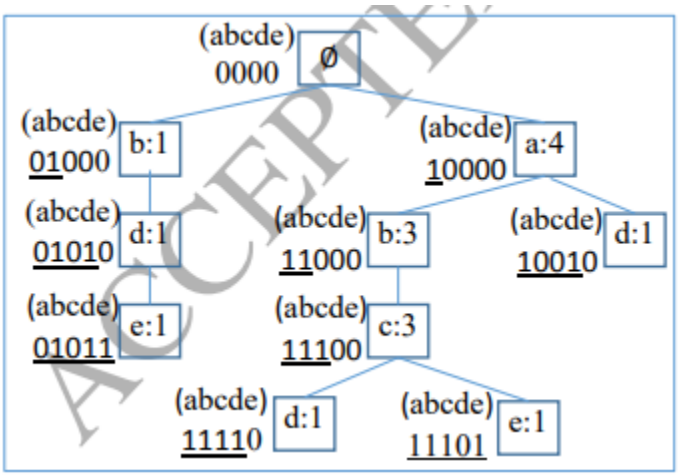


Figure 3 The BMC-tree for Example 1. Each node label represents the item-name field. The number in each node represents the count field. A binary number on the left side of each node represents the bitmap-code field. Items in parentheses represent the item assigned to each bit of bitmap-code . The underlined digits in bitmap-code represent the main section, and other bits represent the don't-care section.

Dựa vào Định nghĩa 6 và **Property 3, Property 4,** cây cấu trúc của cây BMC:

Algorithm 1 (constructing_BMC_tree)

Input: A transactional database , and a threshold .
Output: A BMC-tree (Definition 6), and (Definition 2).
1. Scan to find ;
2. =Sort the items of in non-descending order, with respect to ( ), where is an item.
3. Create as the root of a BMC-tree, and do the flowing assignments:
4. Tr.item-name = φ;
5. Tr.count = 0;
6. Tr.bitmap – code = $b_{nf-1}, \dots b_1, b_0$ where $b_i = 0$, and 0≤i≤nf-1; //(Property 3)
7.For each transaction in do:
8. Remove all infrequent items from T.
9. Sort T according to the order of items in $L_1^{reverse}$ (reverse order of L_1)
10. current – root = Tr ;
11. For each item in do:
12. Let N be a child of .current – root in such a way that N.item-name = i ;
13. If such node does not exist then:
14. Create the new node ; 15. ;
15. N.item-name = i
16. N.count = 0;
17. Add N to current-root.children-list;
18. Endif
19. N.count = N.count+1;
20. N.bitmap-code = current-root.bitmap-code V $2^{index(i)}$

21. current – root = N;
22. Endfor
23.Endfor
24. Return A BMC-tree T_r, and a zero-based vector L_1;

Định nghĩa 8(N-info) N là một node của cây *BMC*, *N-info* của *N* là cặp *bitmap-code* và *count fields(bitmap-code, count)*

Định nghĩa 9 (Nodeset(itemset P_k)) *Nodeset* của *itemset* P_k là tập tất cả các *N-info* của *node N* trong cây *BMC*.

Nodeset **(P_k) = {The N – info of N| N lưu i_1 và $\forall i_j, 1 \leq j \leq k$,**

bit được đánh dấu tới i_j trong *N.bitmap-code* là 1**}**

$$NS_a \rightarrow \{(abcde, (10000, 4))\}$$
$$NS_b \rightarrow \{(abcde, (01000, 1)), (abcde, (11000, 3))\}$$
$$NS_c \rightarrow \{(abcde, (11100, 3))\}$$
$$NS_d \rightarrow \{(abcde, (01010, 1)), (abcde, (11110, 1)), (abcde, (10010, 1))\}$$
$$NS_e \rightarrow \{(abcde, (01011, 1)), (abcde, (11101, 1))\}$$

Figure 4 The Nodeset of each frequent 1-itemset for Example 1. Here, NS is the abbreviation for Nodeset

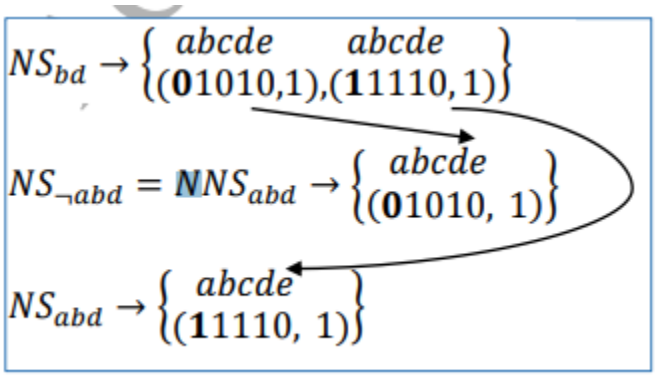


Figure 5The Nodeset of itemset bd .Furthermore, the Nodeset and NegNodeset of itemset abd, in Example 1. Here, NNS is the abbreviation for NegNodeset.

Định nghĩa 10 (NegNodeset(itemset P_k)): Với 2≤k, *NegNodeset* của itemset $\boldsymbol{P_k} = \boldsymbol{i_k P_{k-1}}$ bằng với $Nodeset(P_k = \neg \boldsymbol{i_k P_{k-1}}$**.** Vì vậy, *NegNodeset* của itemset $\boldsymbol{P_k}$ là một tập tất cả các *N-info* của *N* trong cây *BMC* giống như *N* lưu item i_1, mỗi item trong $i_{k-1} \ldots i_2$ được lưu trong các node cha của *N* và item i_k là node không được lưu trong bất kì node cha nào của *N*. Xét định nghĩa 5 và 6, vậy ta có *NegNodeset* của itemset $\boldsymbol{P_k}$ được định nghĩa như sau:

$$NegNodeset(\boldsymbol{P_k = i_k i_{k-1} \ldots i_2 i_1}) = Nodeset(\boldsymbol{P_k = \neg i_k i_{k-1} \ldots i_2 i_1})$$

$$= \{\boldsymbol{N - info\ of\ N | the\ bit\ assigned\ to\ i_j\ in\ N.bitmap - code = 1\ and\ the\ bit\ assigned\ to\ i_k = 0}\}$$

Property 5: $support(P_k) = \sum ni \in Nodeset(P_k) ni.count$

Property 6: Giả sử $2 \leq k. support(P_k) = \sum ni \in Nodeset(P_k) ni.count$

Property 7: itemset $\boldsymbol{P_k = i_k i_{k-1} P_{k-2}}$ và $\boldsymbol{Q_{k-1} = i_k P_{k-2}}$ và **3≤k**, $NegNodeset$ của *k-itemset* có thể trực tiếp trích lấy ra từ $NegNodeset$ của *(k-1)-itemset* Q_{k-1}:

$$\boldsymbol{NegNodeset(P_k = i_k i_{k-1} P_{k-2})}$$

$$= \{\boldsymbol{ni | ni \in NegNodeset(Q_{k-1} = i_k P_{k-2})}$$

$$\boldsymbol{and\ the\ bit\ assigned\ to\ i_{k-1}\ in\ ni.bitmap - code = 1}$$

Property 8: itemset $\boldsymbol{P_k = i_k P_{k-1}}$ và $\boldsymbol{P_k = \neg i_k P_{k-1}}$ và **2≤k**

$$\boldsymbol{support(P_k) = support(P_{k-1}) - support(P_k)}$$

Property 9: itemset P, Q và item I, nơi P∩Q = ∅, i ∉ P, và i ∉ Q, nếu support(P) = support(P ⋃ i), support(P ⋃ Q) = support(P ⋃ i). Trong bảng 1, cho P = ab, Q = e, và i = c. support(ab) = support(abc) = 3. Do đó, support(abe) = support(abce) = 1

Định nghĩa 11 (Set-enumeration tree) Cho L_1(Định nghĩa 2), set-enumeration tree có cấu trúc như sau:

1) Mỗi node N trong set-enumeration tree có hai trường: item-name và children-list. N.item-name giữ item i(i$\epsilon L_1 \cup \{\emptyset\}$). N.children-list giữ tất cả con của Node N. Hơn nữa , node N đại diện cho itemset N.itemset.
2) Root giữ item ∅(root.itemset = ∅) và đại diện cho itemset ∅ (root.itemset = ∅). Node con của root giữ item I, nơi I iϵL_1.
3) Cho mỗi node N, node con của N giữ item I, nơi $i \epsilon L_1 \wedge i \succ N$. item-name tương ứng. itemset của N được định nghĩa N.itemset = N.father.itemset ⋃ N. item − name, nơi Node N.father là cha của N.

Dựa trên định nghĩa 11, mã giả của thuật toán set-enumeration tree mô tả trong Algorithm 2.

Algorithm 2 (Set-enumeration tree Construction)

Input: The zero-based vector (Definition 2).
Output: A set-enumeration tree (Definition 11).

```
1.  Create the node;
2.  root.level = 0; // The root is at level 0;
3.  root.children-list = ∅;
4.  root.item-name = ∅;
5.  root.itemset = ∅;
6.  for each item i∈L1 do:
7.      Create the node child_i;
8.      child_i.level = root.level + 1;
9.      child_i.item-name =i;
10.     child_i.itemset ={i};
11.     Append child_i into root.children-list;
12.     Call constructing_set_enumeration_tree(child_i);//Line 15
13. end for
14. return for
15. procedure constructing_set_enumeration_tree(N)
16.         P  = N.itemset
```

```
17.       N.children-list = ∅
18.       for each item i∈L_1 ∧ I ≻ N.item-name do”
19.            R = P ∪ {i}
20.            Create the node child_i
21.            child_i.level = N.level + 1
22.            child_i.item − name = i
23.            child_i.itemset = R
24.            Append child_i  into N.children-list
25.            Call constructing_set_enumeration_tree(child_i);//Line 15
26.       end for
27. end procedure
```

Set-enumeration tree cho ví dụ 1 trong hình 6. Node được đánh dấu bằng dấu hoa thị chưa mục đó và đại diện cho itemset. Node được đánh dấu hoa thị chứa item và đại diện cho itemset.

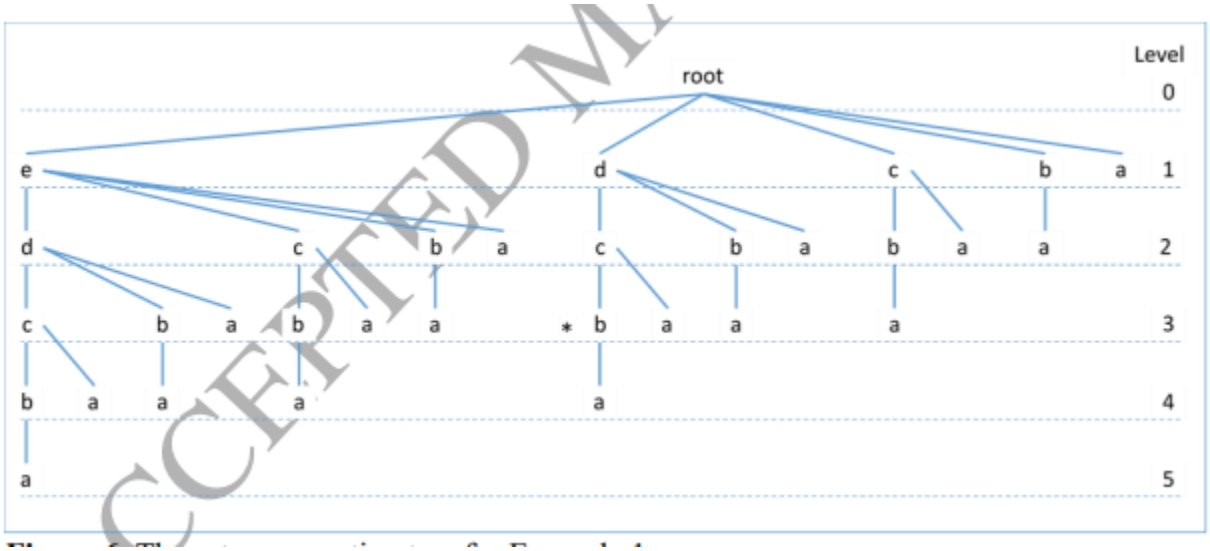


Figure 6 The set-enumeration tree for Example 1.

III negFIN: thuật toán đề xuất

negFIN sử dụng set-enumeration tree(Định nghĩa 11) để biểu diễn không gian tìm kiếm. negFIN bao gồm ba bước. Trong bước đầu tiên, BMC-tree được xây dựng, tất cả tập phổ biến thứ 1 và Nodesets xác định, và mức 1 của set-enumeration tree đã xây dựng.

Trong bước thứ 2, tất cả tập phổ biến thứ 2 và NegFINsets xác định, và mức 2 của set-enumeration tree đã xây dựng. Trong bước 3, tất cả tập phổ biến thứ k (3<=k) và NegNodesets đã xác định và mức khác của set-enumeration tree được xây dựng. negFIN sử dụng dữ liệu của property(Property 9) như một chiến lượt của riêng nó.

Thuật toán 3 hiển thị mã giả cho thuật toán negFIN. F trong dòng (1) giữ tập phổ biến và khởi tạo bởi bộ trống. Dòng(2) xây dựng BMC-tree và L1, bằng cách gọi thuật toán 1. Dòng (3) them tất cả tập phổ biến 1 bằng BMC-tree lần lượt. Dòng (7) tới (19) xây dựng "frequent itemset tree", tương tự với set-enumeration tree (Định nghĩa 11). Dòng (7) tới (11) xây dựng mức 0 của cây(root). Dòng (12) tới (17) xây dựng level k (2 $\leq$ k) của cậy và cho tất cả tập phổ biến thứ k bằng cách gọi đệ quy cho hàm **constructing_frequent_itemset_tree()** (Thuật toán 4). Hàm này tương tự với hàm **constructing_set_enumeration_tree()** được trình bày trong thuật toán 2.

Algorithm 3 (negFIN algoriths)

Input: A transactional database DB and a threshold min-support.
Output: The set of all frequent itemsets, F.

1. F = $\emptyset$
2. call constructing_BMC_tree(DB, min-support)(Algorithm 1) to construct the BMC-tree and find L_1(Definition 2);
3. $F = F \cup L_1$
4. **for each** node N in the BMC_tree do://Traverse the BMC-tree in arbitrary order.
5. Append the N-info of N into the Nodeset of item N.item-name
6. **end for**
7. Create the node root;
8. root.level=0;
9. root.children-list = $\emptyset$
10. root.item-name = $\emptyset$
11. root.itemset = $\emptyset$
12. for each item I $i \epsilon L_1$ do:
13. Create the node $child_i$;
14. $child_i$.level = root.level+1;
15. $child_i$.item-name = i;
16. $child_i$.itemset = {i};
17. Append $child_i$ into root.children-list;
18. call constructing_frequent_itemset_tree($child_i$, $\emptyset$);//Algorithm 4
19. end for
20. return root;

Hàm **constructing_frequent_itemset_tree()** có hai tham số: N và FISparent. N là node hiện tại trong cây phổ biến.FIS parent sử dụng để giữ tập phổ biến trên cha của N. P trong dòng (2) giữ itemset đại diện cho N. Dòng (5) tới 38 mở rộng P bởi bộ i. Mở rộng itemset là ký hiệu như R trong dòng(6). Dòng (8) tới (24) NegNodeset của R. Nếu R là itemset thứ 2 (N ở mức 1), sau đó NegNodeset của R mở rộng từ Nodeset của P(Định nghĩa 10), như dòng (8) tới (15) làm. Dòng (11) kiểm tra điều kiện đặc biệt trong định nghĩa 10 là true. Nếu R là itemset thứ k($3 \leq k$), sau đó NegNodeset của R mở rộng từ NegNodeset của P(Property 7)., như dòng (15) tới (24) làm. Dòng (20) kiểm tra điều kiện đặc biệt của thuộc tính 7 là true. Dòng (25) sử dụng thuộc tính 6 để tính toán độ hổ trợ của R. Dòng (26) sử dụng thuộc tính 8 để tính toán độ hổ trợ của R. Dòng (27) tới (37) tìm item có thể được sử dụng để xây dựng node con của N. Dòng (27) kiểm tra điều kiện đặc biết trong "superset equivalence property"(thuộc tính 9) là true. Nếu điều kiện này là true, item I là promoted item. A promoted item giúp trong N.equivalent_items, cho sử dụng trong tương lại, trong dòng(28); promoted items không sử dụng để xây dựng node con của N, bởi vì tất cả thông tin về tập phổ biến liên quan tới tập này được giữ trong N. Chiến lượt cắt tỉa này được gọi là promotion. Dòng (30), kiểm tra itemset R có phổ biến không.Sau đó dòng (31) tới (45) sử dụng itemset I để tạo node con của N. Dòng (39) tới (45) chỉ định tất cả tập phổ biến trong N, ký hiệu là FISn. Nếu FIS parent là trống, thì FISn tương tự với PSet. Ngược lại, FISn mở rộng từ PSet và FISparent như dòng (44) làm. Dòng (47) tới (51) mở rộng node con của N bằng cách gọi **constructing_frquent_itemset_tree()** (Thuật toán 4) đệ quy.

Phần tốn thời gian của thuật toán negFIN(Thuật toán 3) là việc xây dựng tập phổ biến. Phần đầu tiên của thuật toán negFIN là việc xây dựng cây BMC(Thuật toán 1). Trong trường hợp xấu nhất, độ phức tạp về thời gian của phần này là O(ntxnitxlognit) (độ phức tạp của vòng lặp trong dòng thứ 7 của thuật toán 1), nơi nt = |DB| và nit = |l|. Phần thứ 2 là Nodesets của tất cả các tập phổ biến thứ nhất. Trong trường hợp xấu nhất, độ phức tạp của phần này là O(2^^nit) (độ phức tạp của BMC-tree). Phần 3 là xây dựng một cây phổ biến. Mức k (2<=k) của cây này được xây dựng trong thuật toán 4. Để xây dựng mỗi node ở mức này, thứ nhất, NegNodeset của itemset được gán tới node từ một bộ các node với nhiều n, như vòng lặp trong dòng 9 và 18 của thuật toán 4. Thứ hai, độ hổ trợ của itemset gán tới node được tính. Trong trường hợp xấu nhất, độ phức tạp của toán tử này là O(n)(Độ phức tạp của dòng 25 của thuật toán 4).Thứ 3, nó kiểm tra itemset gán cho node là phổ biến. Độ phức tạp là O(1). Trong trường hợp xấu nhất độ phức tạp của

phần 3 của thuật toán negFIN là $O(2^{nit}n)$, nơi n^{nit} là số node lớn nhất trong cây phổ biến.

Độ phức tạp của thuật toán negFIN tương đương với độ phức tạp của phần thứ ba vì phần này có độ phức tạp lớn nhất trong số các phần khác. Cho l số node tại mức k ($2 \leq k$) của tập phổ biến. Độ phức tạp của negFIN là $O(ln)$. Tham số l là tương đương cho negFIN và công việc trước đó. Độ phức tạp của công việc trước đó là $O(l(x + y))$, nơi x và y là hai bộ của node và $O(x + y)$ là độ phức tạo của bộ mới.

Algorithm 4 (Procedure constructing_frequent_itemset_tree)

```
1.  Procedure constructing_frequent_itemset_tree(N, FIS_parent)
2.      P = N.itemset;
3.      N.children-list = ∅
4.      N.equivalent_items = ∅
5.      for each item i ϵ L_1 ∧ i ≻ N.item-name do:
6.          R = iP;
7.          R.NegNodeset = ∅
8.          if N.level = 1 then:
9.              for each N-info ni ϵ Nodeset(P) do:
10.                 //Checks whether the bit assigned to the item i in ni.bitmap-code is
    0?(Definition 10)
11.                 if ni.bitmap-code ∧ 2^index(i)=0 then:
12.                     R.NegNodeset = R.NegNodeset ∪ {ni};
13.                 end if
14.             end for
15.         else
16.             jX = P;
                R = ijX;
17.             Q = iX;
18.             for each N-info ni ∪ NegNodeset(Q) do:
19.                 //Checks whether the bit assigned to the item j in ni.bitmap-code is
    1?(Property 7)
20.                 if ni.bitmap-code ∧ 2^index(i) = 1 then:
21.                     R.NegNodeset = R.NegNodeset ∪ {ni}
22.                 end if
23.             end for
24.         else if
25.         R.support = ∑ niϵNegNodeset(R) ni.count; Property 6
26.         R.support  = P.support – R.support;//Property 8
27.         if R.support = P.support then:
28.             N.equivalent_items = N.equivalent_items ∪ {i};
29.         else
```

```
30.                 if  R.support ≥ |DB| x min-support then:
31.                     Create the node child_i ;
32.                     child_i.level = N.level +1;
33.                     child_i.item-name = i;
34.                     child_i.itemset = R;
35.                     Append child_i into N.children-list
36.               end if
37.             end if
38.         end for
39.     S =  the set of all subsets of N.equivalent_items;
40.     PSet ← {A|A = {N.item-name} ∪ A, A ϵSS};
41.     if FIS_parent = ∅ then:
42.         FIS_N = Pset;
43.     else
44.         FIS_N = {P|P = P_1 ∪ P_2, P_1, PSet ∧ P_2 ϵFIS_{parent}};
45.     end if
46.     F = F ϵ FIS_N
47.     if N.children-list = ∅ then:
48.         for each child_i ϵ N.children-list do:
49.             call construting_frequent_itemset_tree(child_i, FIS_N);//Algorithm 4
50.         end for
51.     end
52.           return;
53.      end if
54. end proceducre
```

IV Kết quả thử nghiệm và phân tích

Để đánh giá hiệu suất của thuật toán negFIN, chúng tôi tiến hành hai nhóm thí nghiệm. Mục đích của nhóm đầu tiên thử nghiệm là so sánh hiệu suất của thuật toán negFIN với các thuật toán sau: (1) Goethals's Eclat (Goethals & Zaki, 2004), đây là thuật toán tiên tiến trong nhánh của thuật toán khai thác dữ liệu (Z. Deng et al., 2012), and (2) FP-growth* (Grahne & Zhu, 2005), đây là thuật toán tiên tiến trong một họ của FP-trê dựa trên thuật toán FP-Growth(Z. Deng et al., 2012). Trong nhóm thử nghiệm thứ hai, chúng tôi đã tiến hành thử nghiệm toàn diện để so sánh hiệu suất của thuật toán negFIn so với thuật toán dFIN(Z.-H.Deng, 2016), do (1) cả hai thuật toán đều thuộc về cùng họ thuật toán(thuật toán dựa trên nodeset) và (2) dFIN là thuật toán nhanh nhất giữa họ các thuật

toán này với họ các thuật toán khác của thuật toán khai phá tần số của "itemset" (Z.-H Deng,2016). Kết quả được tạo ra bởi thuật toán tương tự nhau. Nhưng các thuật toán khác nhau liên quan đến thời gian chạy và bộ nhớ tiêu thụ.

1 Tập dự liệu

Chúng tôi đã chạy các thuật toán so sánh trên bảy bộ dữ liệu thực, là các tập dữ liệu chung từ các nghiên cứu khai thác mặt hang thường xuyên trước đó và một tập dữ liệu tổng hợp. Các bộ dữ liệu này có thể được tại xuống từ kho lưu trữ FIMI((http://fimi.ua.ac.be). Mô tả các bộ dữ liệu này được thể hiện trong Bảng 2. Trong bảng này, là số lượng các mục, số lượng các giao dịch và là thời lượng giao dịch trung bình. Bảy bộ dữ liệu thực này thường rất dày đặc. Bộ dữ liệu tổng hợp T10I4D100K có nhiều bộ dữ liệu hơn bộ dữ liệu thực này. Tập dữ liệu được tạo bởi trình tạo IBM, có thể tải xuống từ http://www.almaden.ibm.com/cs/quest/syndata.html. Để tạo dữ liệu này, kích thước giao dịch trung bình, kích thước tập hợp có thể xảy ra tối đa trung bình, số lượng giao dịch trong tập dữ liệu và số lượng mục khác nhau được sử dụng trong tập dữ liệu được đặt thành 10, 4, 98487 và 949 tương ứng.

Bảng 2. Mô tả tập dữ liệu sử dụng

Dataset	Type	Item	Transactions	Avg.Length
Accidents	Real	468	340.183	33.8
Chess	Real	75	3.196	37
Connect	Real	129	67.557	43
Kosarak	Real	41.270	990.002	8.1
Mushroom	Real	119	8.124	23
Pumsh	Real	2113	49.046	74
Retail	Real	16.469	88.162	10.3
T10I4D100K	Synthetic	949	98.487	10

2 Môi trường thực thi

Để cho công bằng, tất cả các thí nghiệm này được thực hiện trong cùng điều kiện phần cứng và phần mềm. Chúng tôi sử dụng máy tính có bộ nhớ 8GB và bộ vi xử lý Intel Core i5 3.0 GHz, hệ điều hành Windows 10 x64 Standard Edition. Tất cả đều được thực hiện trên C++. Thực hiện FP-growth và Goethals's Eclat có sẵn tại http://fimi.ua.ac.be/src/ and http://adrem.ua.ac.be/~goethals/software/.

3 negFIN versus FP-growth* và Goethals's Eclat

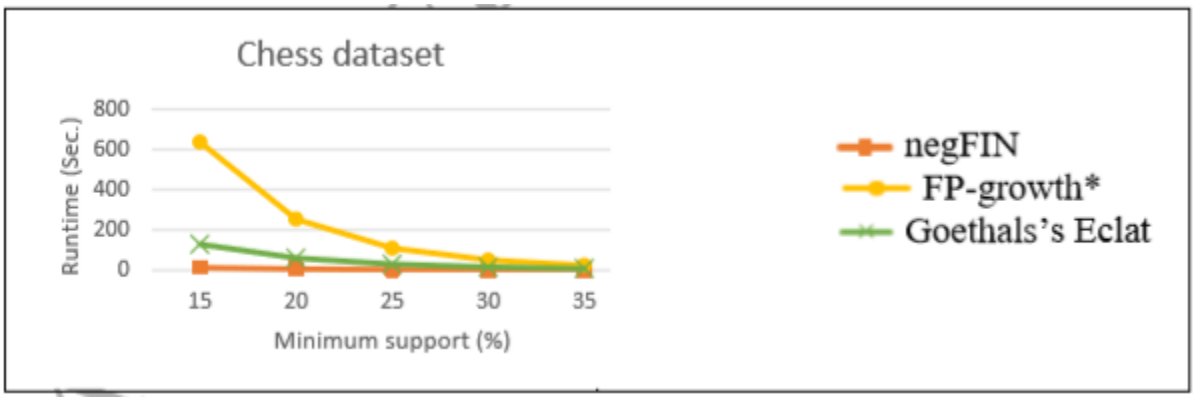
Mục đích của nhóm thử nghiệm này là so sánh thời gian chạy và mức tiêu thụ bộ nhớ của thuật toán negFIN với thuật toán FP-growth* và Goethals's Eclat. Chúng tôi đã

tiến hành các thí nghiệm này trên năm tập dữ liệu chess, pumsb, kosarak, mushroom, and T10I4D100 K với nhiều minimum support khác nhau.

a So sánh Runtime

So sanh thời gian chạy của negFIN với FP-growth* và Goethals's Eclat được thể hiện qua Hình 7. Trong hình này, X và Y là hổ trợ và thời gian chạy tương ứng. Thời gian chạy là thời gian mà thuật toán chạy.

Như chúng ta có thể thấy trong Hình 7, negFIN vượt qua FP-Growth và Goethals's Eclat trong ba tập dữ liệu: chess, pumsb, và kosarak. Mặc dù negFIN chạy nhanh hơn trên tập dữ liệu mushroom và T10I4D100K-không có sự khác biệt đáng kể giữa negFIN và hai thuật toán này.



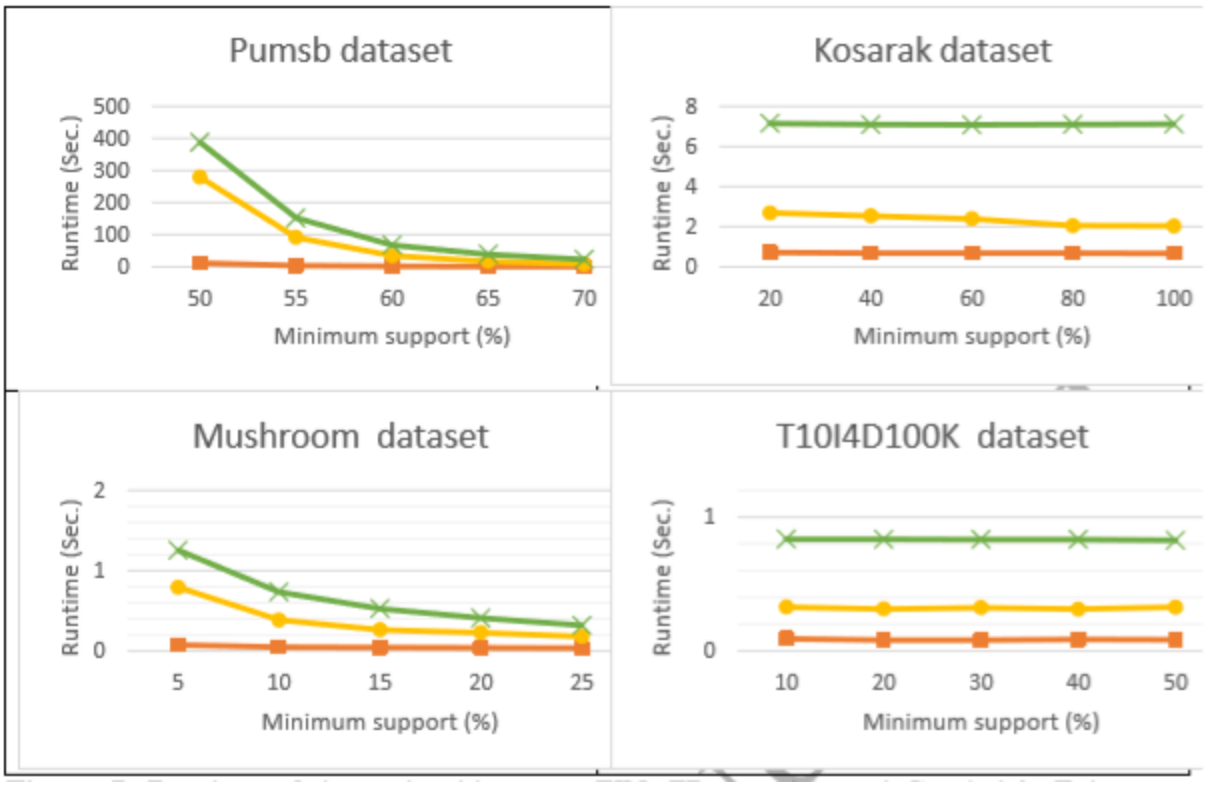


Figure 7 Runtime of three algorithms, negFIN, FP-growth, and Goethals's Eclat, on different datasets, depending on the minimum support*

b So Sánh Bộ Nhớ Tiêu Thụ

Việc so sánh mức tiêu thụ bộ nhớ của negFIN với FP-growth* và Goethals's Eclat được hiện thị trong Hình 8. Trong Hình này, trục Y là mức bộ nhớ tiêu thụ được đo bởi hàm PeekWorkingSetSize trong C/C++. negFIN tiêu thụ nhiều bộ nhớ hơn hai thuật toán kia trên bộ dữ liệu chess và pumsb khi minimum support thấp. Lý do là các phần chính của tiêu thụ bộ nhớ trong negFIN và FP-Growth là BMC-tree và FP-tree. Vì nút của cây BMC lơn hơn so với nút của cây FP, nó giữ nhiều thông tin hơn so với nút của cây FP. Do đó, cây BMC tiêu thụ nhiều bộ nhớ hơn so với cây FP. Ngoài ra, negFIN duy trì một cây BMC trong khi tạo NegNodeSets của một frequent 1-itemsets.

Trong Hình 8, chúng tôi quan sát negFIN và FPgrowth tiêu thụ gần nhưu cùng một lượng bộ nhớ để minimum support trên bộ dữ liệu chess và pumsb , và cho tất cả minimum support trên bộ dữ liệu mushroom, và T10l4D100K. Goethals's Eclat tiêu thụ nhiều bộ nhớ hơn negFIN và FP-growth cho minimum support cao trên bộ dữ liệu pumsh và mushroom, và cho tất cả minimum support trên bộ dữ liệu kosarak và T10l4D100K.

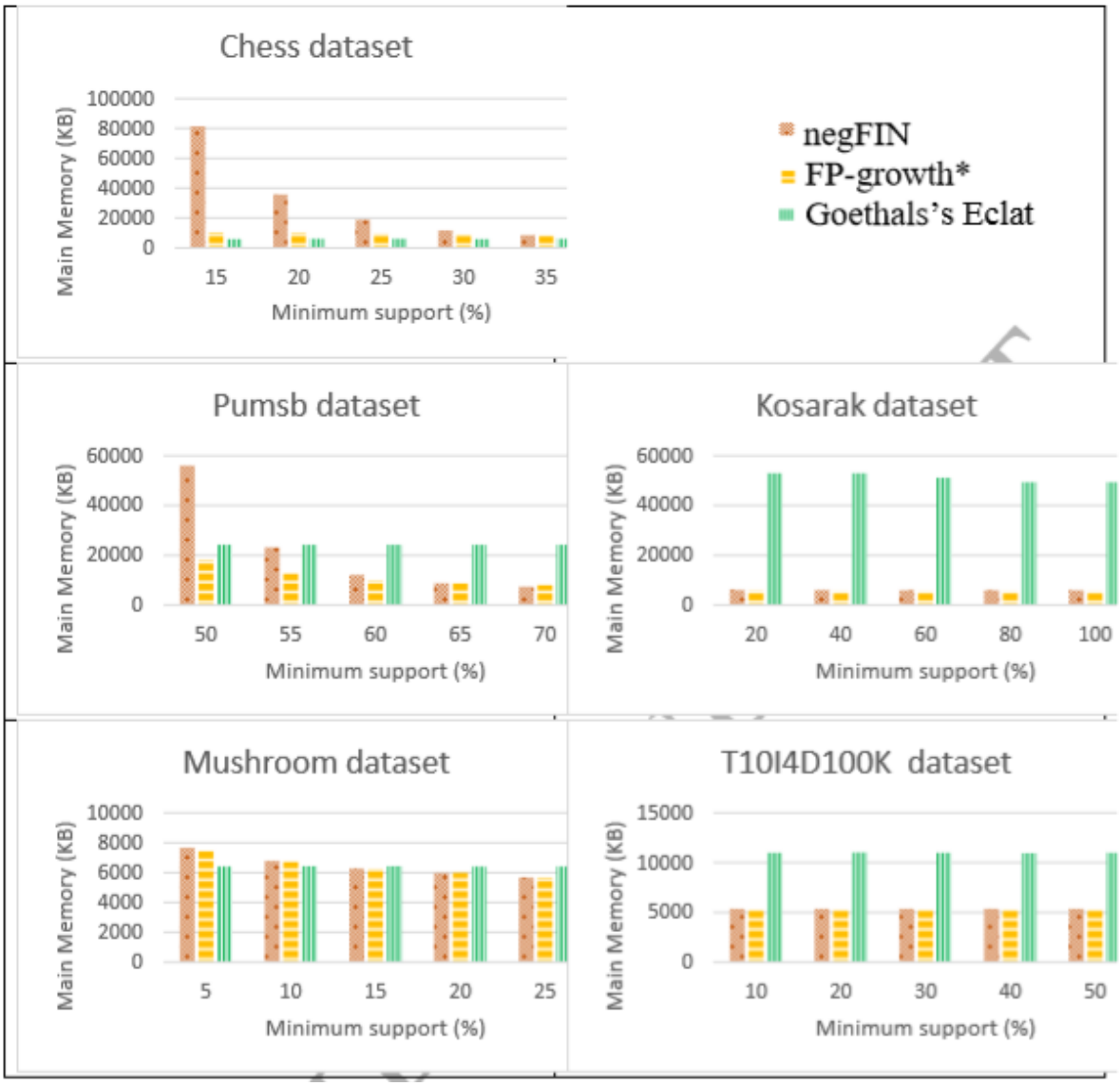


Figure 8 . Memory consumption of three algorithms, negFIN, FP-growth, and Goethals's Eclat, on different datasets, depending on the minimum support*

4 negFIN versus dFIN

Trong phần này, chúng tôi so sành negFIN và dFIN dựa trên ba khía cạnh: (1) Số khóa toán tử, (2) thời gian chạy, và (3) Bộ nhớ tiêu thụ.

a Số khóa toán tử

Trong thuật toán negFIN(dFIN), mỗi NegNodeset(DiffNodeset) của k-itemset(k≥2) có nguồn gốc từ một bộ(hai bộ) của node. Cho S1^NegNodeSet là bộ của node cũng như NegNodeset của P có nguồn gốc từ đó, và |S1^NegNodeset| = n^negFIN. Hơn nữa, cho $S1^{DiffNodeset}$ và$S2^{DiffNodeset}$ có 2 bộ của node với DiffNodeset của P có nguồn gốc từ nó, $\left|S1^{DiffNodeset}\right| = n^{dFIN}$, và$\left|S2^{DiffNodeset}\right| = m^{dFIN}$. Độ phức tạp của

NegNodeset và DiffNodeset của P tương ứng là $O(n^{negFIN})$ và $O(n^{dFIN} + m^{dFIN})$. Phần tiêu thụ thời gian của negFIN(dFIN) là dẫn xuất của $NegNodesets(DiffNodesets)$. Cho l^{negFIN} và l^{dFIN} là số lượng NegNodesets và DiffNodesets tương ứng. Như vậy, Độ phức tạp của negFIN và dFIN là $O(l^{negFIN} * n^{negFIN})$ và $O(l^{dFIN}\ (n^{dFIN} + m^{dFIN})$ tương ứng.

Trong hình 9, trung bình của $n^{negFIN}, n^{dFIN}, m^{dFIN}$ và số trung bình của khóa toán tử yêu cầu đưa tới NegNodeset(DiffNodeset) của mỗi k-itemset(k≥2) được hiển thị. Số trung bình của khóa toán tử được ký hiệu như KOD(Chữ viết tắt của key operations in each derivation). Ở đây, khóa toán tử thực hiện vòng lặp. Do đó, KOD là số lần trinh bình khi vòng lặp được thực thi.

Bằng cách kiểm tra hình 9, các kết quả sau đây thu được: (1) Số lượng trinh bình của các toán tử tới NegNodeset là bằng n^{negFIN}. Do đó, đạo hàm của NegNodeset có độ phức tạp là $O(n^{negFIN})$. (2) Số lượng trình của khóa toán tử tới DiffNodeset là khoảng n^{dFIN} và $(n^{dFIN} + m^{dFIN})$. Như vậy, đạo hàm của DiffNodeset có độ phức tạp là $O(n^{dFIN} + m^{dFIN})$. (3)$n^{dFIN} \leq m^{dFIN}$. (4)$n^{negFIN} = n^{dFIN}$. Để đơn giản, chúng tôi sử dụng ký hiệu n thay cho n^{negFIN} và n^{FIN}, và ký hiệu m thay cho m^{dFIN}. Chúng tôi kết luận từ (1) tới (4): (5) Độ phức tạp cho đạo hàm của mỗi NegNodeset là O(n), (6) độ phức tạp của đạo hàm của mỗi DiffNodeset là $O(n + m)$, và (7) n ≤ m. Sau đó, kết luận tổng thể là NegNodeset của itemset được tạo ra khoảng hai đơn vị cường độ nhanh hơn DiffNodeset.

Trong hình 10, l^{negFIN} và l^{dFIN} đại diện cho hai tập dữ liệu khác nhau. Như chúng ta có thể xem trong hình này, $l^{negFIN} = l^{dFIN}$ cho tất cả tập dữ liệu. Để đơn giản, chúng tôi sử dụng ký hiệu l thay chol^{negFIN} vàl^{dFIN}. Do đó, độ phức tạp của negFIN và dFIN là $O(ln)$ và$O\big(l(n + m)\big)$, n ≤ m, tương ứng.

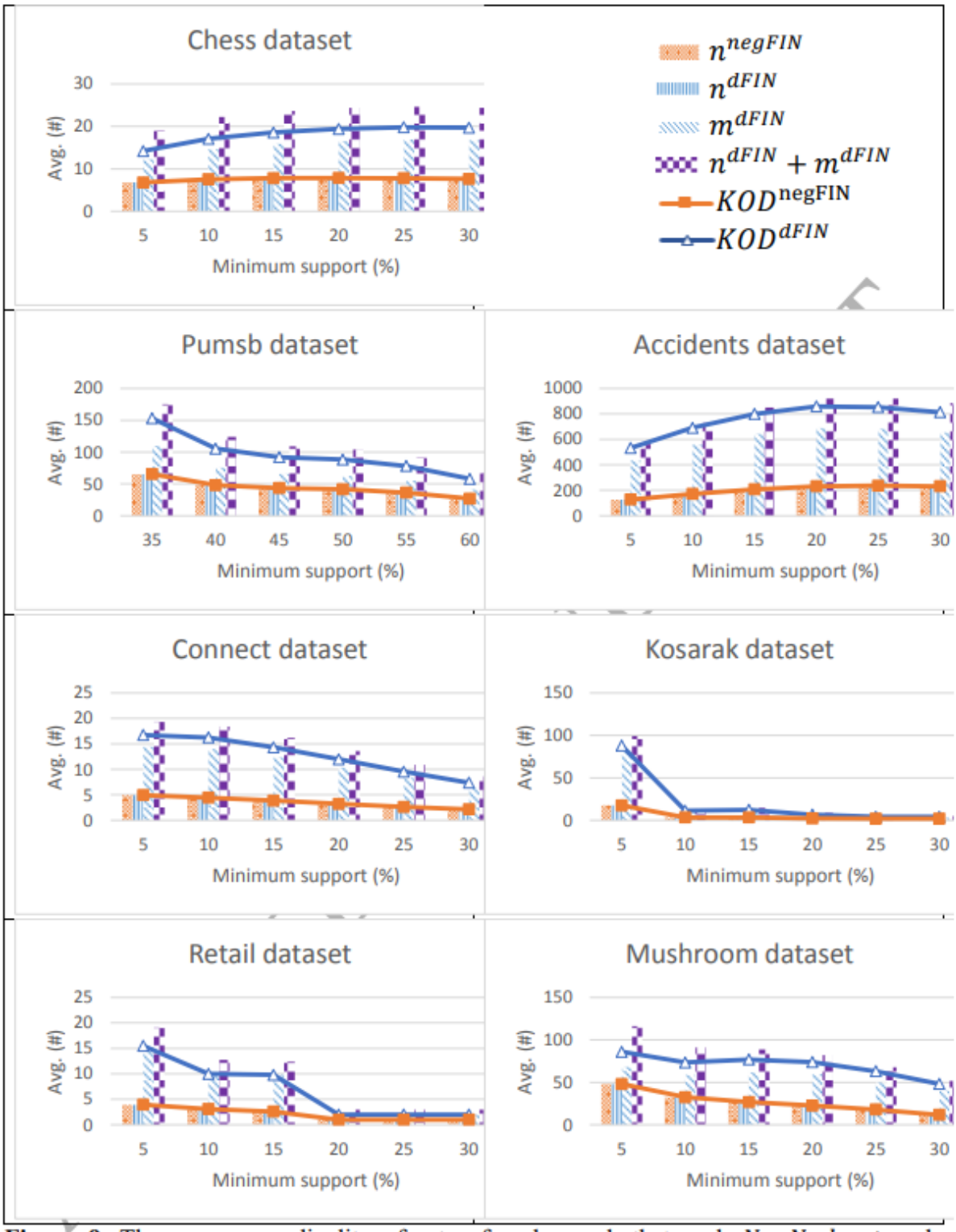


Figure 9. The average cardinality of sets of nodes such that each $NegNodeset$ and $DiffNodest$ of k-itemset $(2 \leq k)$ is derived from them and the average number of key operations required to derive each $NegNodeset$ and $DiffNodest$, which is denoted as KOD, for different datasets, depending on the minimum support. Here, KOD is the abbreviation for **k**ey **o**perations in each **d**erivation.

b So sánh thời gian chạy

Hình 11 cho thấy so sành thời gian chạy của negFIN và dFIN. Như chúng ta có thể thấy trong hình này, negFIN không chậm hơn dFIN trên tất cả các tập dữ liệu. negFIN chạy nhanh hơn dFIN trên một số tập dữ liệu, đặc biệt là minimum support thấp. Lý do như sau: độ phức tạp về thời gian của negFIN và dFIN là $O(ln) và O\big(l(n+m)\big)$ tương ứng. Như chúng ta thấy trong Hình 9, cả n và m có giá trị nhỏ. Do đó, sự khác nhau giữa ln và $l(n+m)$ là không đáng kể cho giá trị nhỏ. Một lần nữa, xem hình 10 và hình 11. Như chúng ta có thể thấy trong số liệu này, đối với các bộ dữ liệu như chess, pumsb, và accidents, trong đó có giá trị lớn, sự khác nhau giữa thời gian chạy của negFIN và dFIN là quan trọng.

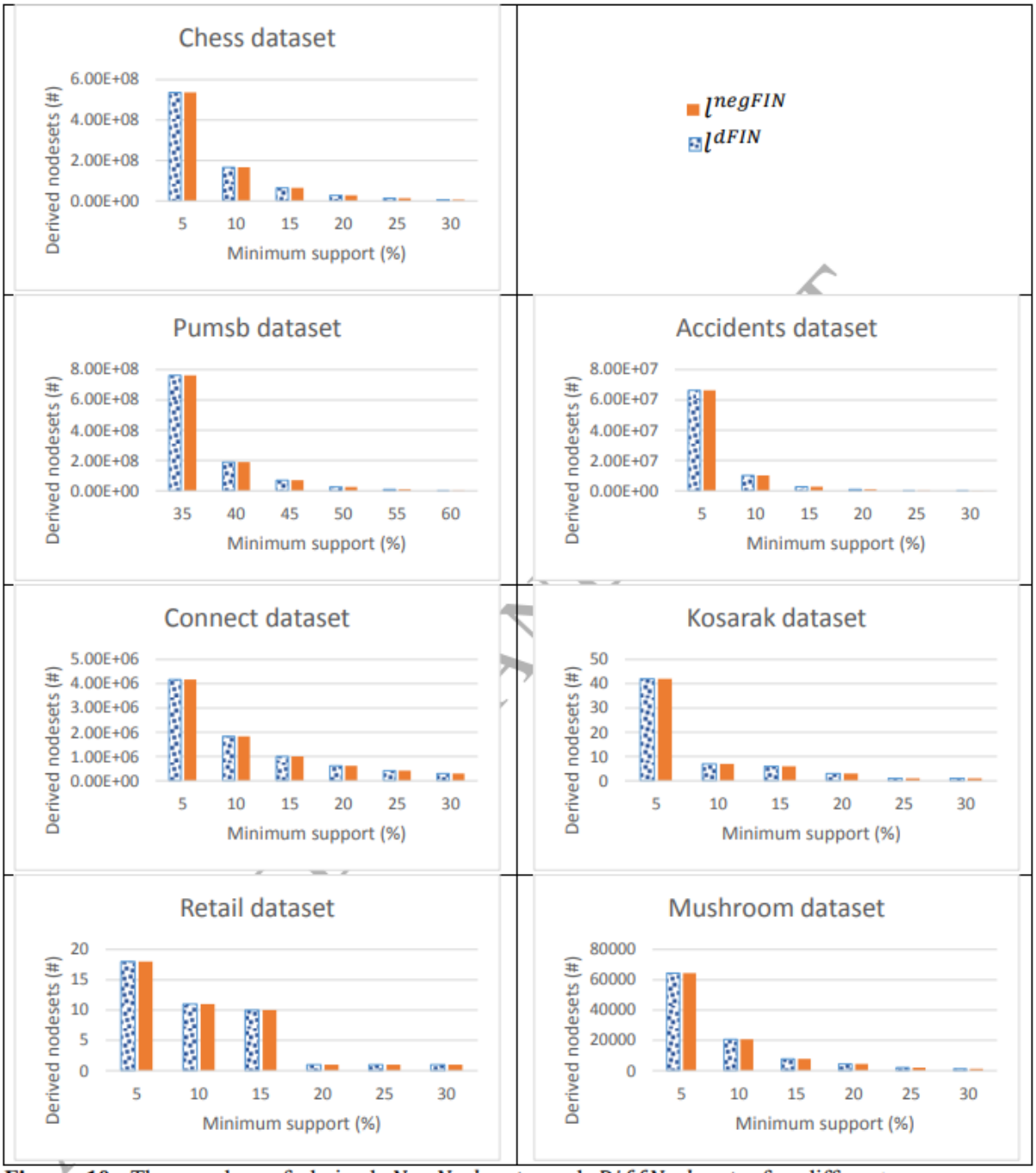


Figure 10. The number of derived *NegNodesets* and *DiffNodesets* for different datasets, depending on the minimum support.

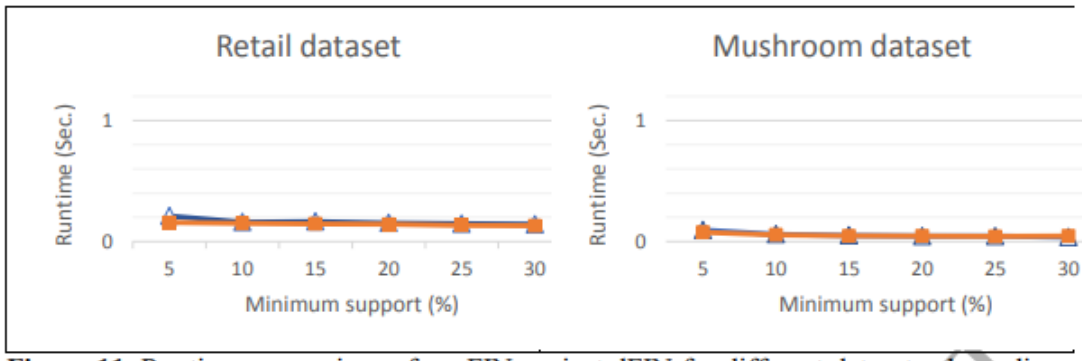
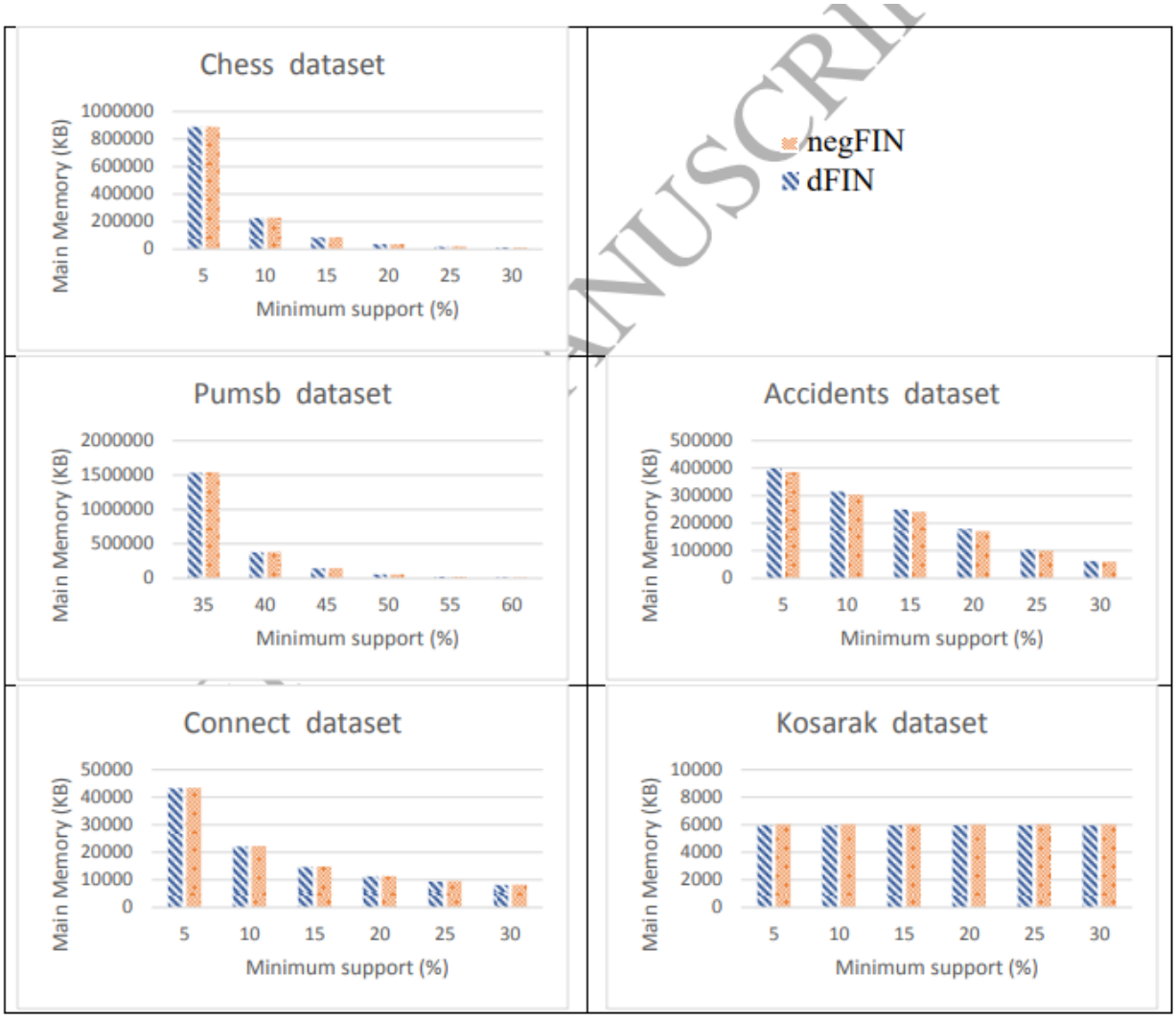


Figure 11. Runtime comparison of negFIN against dFIN for different datasets, depending on the minimum support.



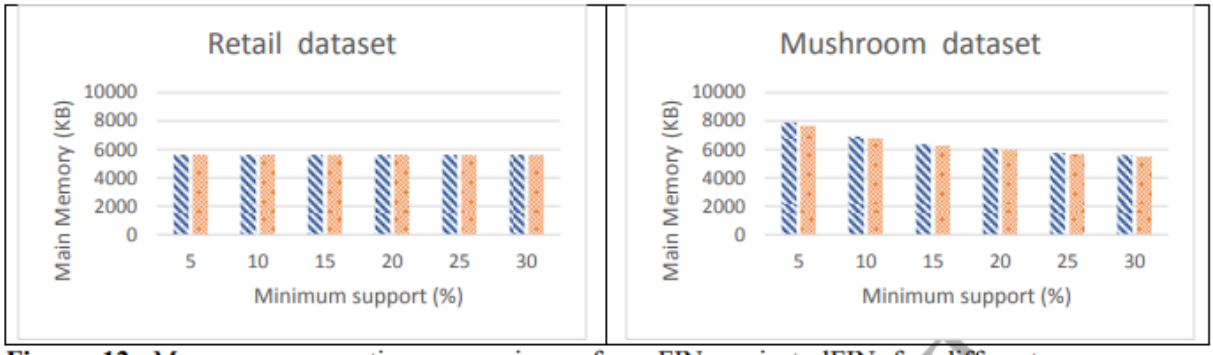


Figure 12. Memory consumption comparison of negFIN against dFIN for different datasets, depending on the minimum support.

c So sánh bộ nhớ tiêu thụ

Trong hình 12, cho thấy bộ nhớ tiêu thụ của negFIN và dFIN. Như chúng ta có thể thấy trong hình này, mức tiêu thụ bộ nhớ của cả hai thuật toán gần như nhau.

V Kết luận

Trong bài báo này, chúng tôi đã trình bày một cấu trúc dữ liệu mới, được gọi là lưu trữ thông tin về tập phổ biến. Chúng tôi trình bày một thuật toán, được gọi là negFIN, để khai phá tất cả tập phổ biến trong cơ sở dữ liệu một cách nhanh hơn. So với nFIN, các ưu điểm chính của negFIN như sau: (1) nó sử dụng các toán tử bitwise để tạo ra các nút mới. (2) Nó làm giảm độ phức tạp của việc phát hiện các tập phổ biến tới $O(ln)$, thay vì $O\big(l(m+n)\big)$, nơi m và n là hai tập hợp các nút cơ sở, n ≤ m, và l là số lượng các nút được tạo ra. Chúng tôi thực hiện các thuật toán negFIN và dFIN và tiến hành các thí

nghiệm mở rộng để so sánh hiểu suất của negFIN với một số thuật toán khai thác tập phổ biến khác. Các thử nghiệm này cho thấy thuật toán của chúng tôi là thuật toán nhanh nhất trên tất cả các tập dữ liệu với minimum support khác nhau với các thuật toán trước đây. Tuy nhiên, trên một số bộ dữ liệu với một số minimum tối thiểu, thuật toán của chúng tôi chạy với tốc độ như dFIN.