HDminer: Efficient Mining of High Dimensional Frequent Closed Patterns from Dense Data

Jianpeng Xu
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI
Email: xujianpe@msu.edu

Shufan Ji*
School of Computer Science and Engineering
Beijing University of Aeronautics and Astronautics
Beijing, China
Email: jishufan@buaa.edu.cn

***Abstract*—Frequent closed pattern mining has been developed for decades, mostly on a two dimensional matrix. This paper addresses the problem of mining high (≥ 3) dimensional frequent closed patterns (nFCPs) from dense binary dataset, where the dataset is represented by a high dimensional cube. As existing FP-tree or enumeration tree based algorithms do not suit for n-dimensional dense data, we are motivated to propose a novel algorithm called HDminer for nFCPs mining. HDminer employs effective search space partition and pruning strategies to enhance the mining efficiency. We have implemented HDminer, and the performance studies on synthetic data and real microarray data show its superiority over existing algorithms.**

***Keywords*—HDminer; Frequent Pattern Mining; High Dimensional Data**

I. Introduction

Frequent pattern mining [1], [2], [3], [4] has a wide application to many data mining tasks, including association analysis, correlation analysis, causality analysis, association-based classification and clustering. However, the number of frequent patterns are too large for users to digest. To reduce the number of frequent patterns (FPs), frequent closed pattern (FCP) mining [5] has been proposed to deliver the same information as the FPs, which has been successfully adopted for many data analysis tasks.

Given a 2D boolean Matrix $O = R \times C$ (shown in Table I), where the row set $R = \{r_1, r_2, \ldots, r_n\}$ and the column set $C = \{c_1, c_2, \ldots, c_m\}$, a *true* value $O_{ij} = 1$ denotes the "containing/contained" relationship between row r_i and column c_j; and a *false* value otherwise. A pattern $f = (R' \times C') \subseteq O$, where $R' \subseteq R$ and $C' \subseteq C$, is defined as a FCP if (1) $\forall O_{ij} \in f, O_{ij} = 1$; (2) $\forall c_j \notin C', \exists r_i \in R', O_{ij} = 0$; (3) $\forall r_i \notin R', \exists c_j \in C', O_{ij} = 0$; and (4) $|R'| \geqslant minR$ and $|C'| \geqslant minC$. In these four conditions,(1)(2)(3) ensure the pattern f is closed while (4) ensures f is frequent. For example, in Table I, given that $minR = minC = 2$, the pattern $f = \{r_1, r_2\} \times \{c_2, c_3, c_4\}$ is a FCP. However, $f' = \{r_1, r_2\} \times \{c_2, c_3\}$ is not a FCP in that condition (2) is not satisfied.

There are many notable FCP mining algorithms in the literature. CLOSET [6], CLOSET+ [7], and CIMNC [8] adopt the FP-tree as the fundamental data structure, while CHARM [9], CARPENTER [10], and REPT [11] employ the enumeration tree. Although these algorithms perform well on sparse data, they are not suited for data with high density(in this paper, we will consider a matrix with high density if 30% or more matrix cells are ones). D-Miner [12] was the first to employ the space partition tree to process 2D dense data. Then C-Miner [13] proposed a compressed strategy for large dataset mining, while CubeMiner [14] was the first to mine 3D frequent closed patterns from a 3 dimensional cube data, based on the space partition tree. However, all above algorithms are limited to process low dimensional (2D/3D) data. With the emergence of high dimensional data, Data-Peeler [15] has recently been proposed to mine n-dimensional frequent closed patterns (nFCP) utilizing the enumeration tree structure. Although Data-Peeler works well for n-dimensional sparse data, it does not suit for dense data - it is either inefficient (i.e., take hours or even days to produce patterns), or may even fail (i.e., run out of memory). With the emergence of many high dimensional biological data of high density, we are motivated to propose an efficient nFCP mining algorithm for dense data.

R/C	c_1	c_2	c_3	c_4
r_1	0	1	1	1
r_2	1	1	1	1
r_3	0	0	1	1

TABLE I: Data Matrix O ($minR = minC = 2$).

In this paper, we propose a novel algorithm called HDminer to mine nFCPs from dense datasets. HDminer employs effective search space partitioning and pruning strategies to enhance the mining efficiency. Rather than accumulating the *true* valued cells as the FP-tree or enumeration tree based methods, HDminer progressively narrows down the search space by pruning off the *false* valued cells, based on the space partition tree. Since the amount of the *false* valued cells are much less than that of the *true* valued ones for dense datasets, HDminer would work much more efficient than the FP-tree or enumeration tree based methods. We have implemented HDminer, and the performance study on synthetic data and real dense microarray data shows its superiority over the most recent algorithm Data-Peeler, even on datasets that are relatively sparse.

The rest of this paper is organized as follows. In the next section, we review notable existing works on FCP mining. In Section III, we provide some preliminaries. Section IV presents the principles behind HDminer, while Section V proposes the HDminer algorithm. In Section VI, we report experimental results obtained from comparing HDminer against Data-Peeler

* Corresponding author

with synthetic and real biological data. Finally, we conclude in Section VII.

II. Related Work

There are many notable algorithms for FCP mining, which can be classified into three categories according to the data structures used in their algorithms. **FP-tree.** The FP-tree structure is widely adopted by CLOSET [6], CLOSET+ [7], and CIMNC [8]. CLOSET uses a FP-tree for a compressed representation of the dataset. CLOSET+, an enhanced version of CLOSET, uses a hybrid tree-projection method to build conditional projected table in two different ways according to the density of the dataset. CIMNC maintains additional information in the FP-tree to avoid closure checking during FCP mining. Although these algorithms are efficient to process sparse data, they are limited in mining low dimensional 2D FCPs.

Enumeration Tree. While CHARM [9] enumerates the columns for the "large rows small columns" market data, CARPENTER [10] changes to enumerate rows for the "small rows large columns" biological data with some efficient search pruning techniques. In [16], COBBLER dynamically switches between column enumeration and row enumeration depending on the data characteristic in the mining process. However, the decision to switch the enumeration strategies at runtime is costly. Yet another more efficient algorithm REPT [11] is proposed by traversing the row enumeration tree using a projected transposed table represented by a prefix tree. All of these algorithms are limited in mining 2D FCPs. More recently, Data-Peeler [15] has been proposed to process high dimensional data, which is the first algorithm to mine nFCPs. The enumeration tree based algorithms are efficient to process sparse data.

Space Partition Tree. D-Miner [12] was the first algorithm to employ the space partition tree for 2D dense data processing. Then C-Miner [13] proposed a compressed strategy for large dataset mining. Further, CubeMiner [14] was proposed to mine 3D FCPs. Although these algorithms work efficiently on dense data, they could process only low dimensional (2D/3D) FCPs.

In summary, algorithms with the FP-tree or enumeration tree structure are efficient in processing sparse data, in that their mining strategy is to accumulate the *true* valued cells gradually. However, when the data are dense with large number of *true* valued cells, too many tree branches are generated, leading to long processing time and memory bottleneck. On the contrary, algorithms adopting the space partition tree progressively narrow down the search space by pruning off the *false* valued cells. The denser the data, the less the space partition tree branches. Hence, the space partition tree structure is suitable for dense data processing.

As Data-Peeler adopts the enumeration tree structure, it is inefficient for dense data processing. However, some real biological data that we used in the performance study are very dense. Thus, we are motivated to design an algorithm based on the space partition tree structure for high dimensional dense data.

$D_1^4 D_1^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	1	1	1	0
D_2^2	1	1	1	0
D_3^2	1	1	1	1
$D_1^4 D_2^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	1	1	1	1
D_2^2	0	1	1	1
D_3^2	1	1	1	1
$D_1^4 D_3^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	1	1	1	0
D_2^2	1	1	1	0
D_3^2	1	1	1	1
$D_2^4 D_1^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	0	0	1	1
D_2^2	1	1	1	0
D_3^2	1	1	0	1
$D_2^4 D_2^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	1	1	1	0
D_2^2	1	0	1	0
D_3^2	1	0	1	1
$D_2^4 D_3^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	1	1	1	1
D_2^2	1	0	1	0
D_3^2	0	1	1	1
$D_3^4 D_1^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	0	1	1	1
D_2^2	0	1	1	1
D_3^2	0	0	0	1
$D_3^4 D_2^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	1	1	1	0
D_2^2	1	1	1	1
D_3^2	0	0	1	1
$D_3^4 D_3^3$	D_1^1	D_2^1	D_3^1	D_4^1
D_1^2	1	1	1	1
D_2^2	1	1	1	0
D_3^2	0	1	0	0

TABLE II: An Example Data Matrix O (size $= 3 \times 3 \times 3 \times 4$).

III. Preliminaries

Suppose that the dataset O is composed by n types of binary features. Each type of features is a feature set of size $d_i, i = 1, ..., n$. Imagining each type of features as one dimension of the dataset, then the dataset can be regarded as a high dimensional cube of size $d_1 \times ... \times d_n$. Note that the features in different dimensions are different. In this paper, we will call those features as items. Let $D^i = \{D_1^i, D_2^i, \ldots, D_{d_i}^i\}$ denote the set of items in the *i-th* dimension, where the superscript i represents the dimension ID while the subscript represents the item ID, and hence D_m^i represents the *m-th* item on the *i-th* dimension. An n-dimensional dataset can be represented by a boolean matrix $O = D^1 \times D^2 \times \ldots \times D^n$. Table II shows a 4-dimensional example matrix O of size $3 \times 3 \times 3 \times 4$.

*Definition 1 (**Cutter**):* Let $C^i \subseteq D^i$ be a subset of the item set in the *i-th* dimension. For $z = C^1 \times ... \times C^n$, if all cells in O indexed by z are all valued 0, then z is called a cutter for O. We simplify the notation of z as $z\langle C^n, \ldots, C^i, \ldots, C^1\rangle$ to imply the item sets from different dimensions. Z is the whole cutter set for O. By setting the size of C^n to C^1 to be 1, the cutters can be easily found by checking the 0 cells in O.

As for matrix O in Table II, we aggregate the cutters on D^1 in Table III for illustration simplicity. For example, Cutter 6 $< D_2^4, D_1^3, D_1^2, D_1^1 D_2^1 >$ is composed by two cutters: $< D_2^4, D_1^3, D_1^2, D_1^1 >$ and $< D_2^4, D_1^3, D_1^2, D_2^1 >$.

$ID/Cutter$	$C^n, \dots, C^i, \dots, C^1$
1	$D_1^4, D_1^3, D_1^2, D_4^1$
2	$D_1^4, D_1^3, D_2^2, D_4^1$
3	$D_1^4, D_2^3, D_2^2, D_1^1$
4	$D_1^4, D_3^3, D_1^2, D_4^1$
5	$D_1^4, D_3^3, D_2^2, D_4^1$
6	$D_2^4, D_1^3, D_1^2, D_1^1 D_2^1$
7	$D_2^4, D_1^3, D_2^2, D_4^1$
8	$D_2^4, D_1^3, D_3^2, D_3^1$
9	$D_2^4, D_2^3, D_1^2, D_4^1$
10	$D_2^4, D_2^3, D_2^2, D_2^1 D_4^1$
11	$D_2^4, D_2^3, D_3^2, D_2^1$
12	$D_2^4, D_3^3, D_2^2, D_2^1 D_4^1$
13	$D_2^4, D_3^3, D_3^2, D_1^1$
14	$D_3^4, D_1^3, D_1^2, D_1^1$
15	$D_3^4, D_1^3, D_2^2, D_1^1$
16	$D_3^4, D_1^3, D_3^2, D_1^1 D_2^1 D_3^1$
17	$D_3^4, D_2^3, D_1^2, D_4^1$
18	$D_3^4, D_2^3, D_3^2, D_1^1 D_2^1$
19	$D_3^4, D_3^3, D_2^2, D_4^1$
20	$D_3^4, D_3^3, D_3^2, D_1^1 D_3^1 D_4^1$

TABLE III: Cutters of O.

*Definition 2 (**nFCP**):* A pattern $f\langle F^1, \dots, F^i, \dots, F^n\rangle \subseteq O$ is defined as high dimensional frequent closed pattern (nFCP) if (1) all cells in f are valued "1"; (2) $\forall D_k^i \in D^i \setminus F^i$, there exist cells valued "0" in the pattern $f'\langle F^1, \dots, F^i \cup D_k^i, \dots, F^n\rangle$; and (3) $|F^i| \geq minSup^i$, where $minSup^i$ is the user specified minimum support threshold of dimension i.

f is called "closed" pattern if satisfying condition (1) and (2), and "frequent" pattern if satisfying condition (3). Note that a closed pattern cannot be further extended in any dimension, and hence is maximal/closed in respective dimensions. For example, $f_1\langle D_3^4, D_2^3 D_3^3, D_1^2 D_2^2, D_1^1 D_2^1\rangle$ (in Table II) is not closed as it can be extended in dimension D^1 with D_3^1, resulting in $f_2\langle D_3^4, D_2^3 D_3^3, D_1^2 D_2^2, D_1^1 D_2^1 D_3^1\rangle$. Here, f_2 is closed as it cannot be extended in any dimension. And given $minSup^i = 1$, f_2 is an nFCP. Note that the cutters and the patterns in this paper are denoted in the same fashion, but it can be told from the context if not specifically mentioned.

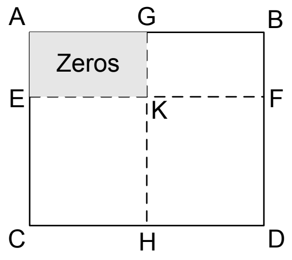
Problem Definition: Given an n-dimensional dataset O, our problem is to discover all high dimensional frequent closed patterns (nFCPs) with the user specified minimum support thresholds.

IV. HDMINER PRINCIPLE

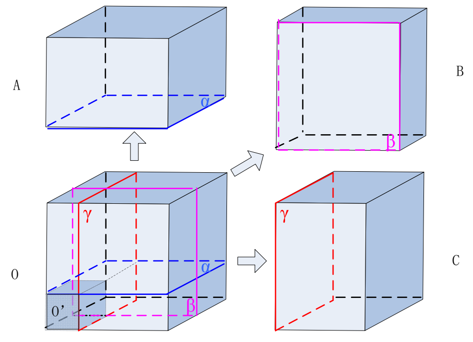
In this section, we will present the principles behind the HDminer algorithm. The space partition tree structure of HDminer is inspired and extended from CubeMiner [14]. CubeMiner is a special case of HDminer when the dimension equals to 3. By removing *false* valued cells from the boolean dataset, HDminer recursively reduces the search space and partitions it into subspaces. For efficiency, HDminer prunes useless subspaces as early as possible by three pruning s-trategies. Note that the "search space" here refers to the high dimensional space of the data presentation, instead of the feasible set of a problem (e.g., convex optimization problem, etc.).

A. Search Space Partition

To reduce the search space, *false* ("0") valued cells are removed recursively, resulting in search space partition. To illustrate the space partition principle, we take the 2D and 3D search spaces in Figure 1 as examples. Given the rectangle $(ABDC)$ as the 2D boolean matrix (search space), the 2D closed patterns are actually the maximal rectangles with cells all valued "1". Hence, if we could remove the useless "0-cells" from the 2D matrix, we would narrow the search space dramat-ically. Let rectangle $(AGKE)$ represent the useless "0-cells" to be removed. From the edge of $(AGKE)$, two lines EF and GH are derived, which split the rectangle $(ABDC)$ into two new subspaces: $(CDFE)$ and $(HDBG)$. Each subspace is still a rectangle and the equation $(CDFE) \cup (HDBG) = (ABDC) \setminus (AGKE)$ is satisfied. Similarly, the 3D closed patterns are the maximal cubes with cells all valued "1" in the 3D search space. Let cube O represent the 3D search space and cube O' represent the useless "0-cells". By the face α, β, γ of O', O is split into subspace A, B, C respectively, satisfying $\alpha \cup \beta \cup \gamma = O \setminus O'$. In any of the new subspace, there may still exist "0-cells". The same partition principle can be applied until all "0-cells" are removed. We try to remove as many "0-cells" as possible in each splitting. Hence, we group "0-cells" together on the largest dimension for efficiency. For example, "0-cells" of O (in Table II) shall be grouped on dimension D^1 in that the number of items on D^1 is the largest, resulting in the cutters in Table III. In the following presentation, we will assume D^1 as the largest dimension and will be treated differently.



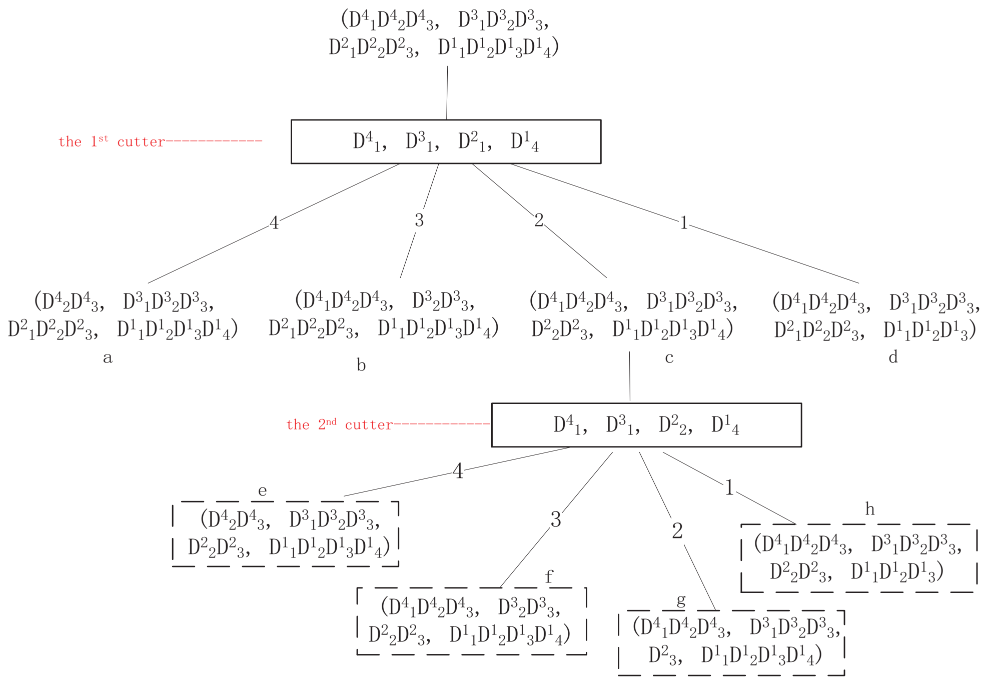
(a) 2D Search Space



(b) 3D Search Space

Fig. 1: Space Partition Principle.

According to the space partition principle, HDminer re-cursively splits the dataset $O\langle D^n, \dots, D^i, \dots, D^1\rangle$ using the cutters in Z until all cutters are used and consequently all cells in each resulting patterns have the value "1". A cutter $\langle C^n, \dots, C^i, \dots, C^1\rangle$ in Z can be used to cut the search space O if $\forall i \in [1, n], C^i \cap D^i \neq \emptyset$. After the cutter is applied, the search space O is split into n subspaces: the *i-th* subspace is denoted as $\langle D^n, \dots, D^i \setminus C^i, \dots, D^1\rangle, i \in [1, n]$. The recursive splitting procedure could be illustrated as a *space partition*



(D⁴₁D⁴₂D⁴₃, D³₁D³₂D³₃, D²₁D²₂D²₃, D¹₁D¹₂D¹₃D¹₄)
the 1st cutter
D⁴₁, D³₁, D²₁, D¹₄
4
3
2
1
(D⁴₂D⁴₃, D³₁D³₂D³₃, D²₁D²₂D²₃, D¹₁D¹₂D¹₃D¹₄)
a
(D⁴₁D⁴₂D⁴₃, D³₂D³₃, D²₁D²₂D²₃, D¹₁D¹₂D¹₃D¹₄)
b
(D⁴₁D⁴₂D⁴₃, D³₁D³₂D³₃, D²₂D²₃, D¹₁D¹₂D¹₃D¹₄)
c
(D⁴₁D⁴₂D⁴₃, D³₁D³₂D³₃, D²₁D²₂D²₃, D¹₁D¹₂D¹₃)
d
the 2nd cutter
D⁴₁, D³₁, D²₂, D¹₄
e
(D⁴₂D⁴₃, D³₁D³₂D³₃, D²₂D²₃, D¹₁D¹₂D¹₃D¹₄)
f
(D⁴₁D⁴₂D⁴₃, D³₂D³₃, D²₂D²₃, D¹₁D¹₂D¹₃D¹₄)
g
(D⁴₁D⁴₂D⁴₃, D³₁D³₂D³₃, D²₃, D¹₁D¹₂D¹₃D¹₄)
h
(D⁴₁D⁴₂D⁴₃, D³₁D³₂D³₃, D²₂D²₃, D¹₁D¹₂D¹₃)

Fig. 2: Part of the Space Partition Tree of Matrix O.

tree, in which a node denotes a subspace of the dataset, and an edge denotes a cutting process by a particular cutter. In Figure 2, we show an example of space partition tree processed by cutters in Table II with $minSup^i = 2$. We only show two levels of the partition tree in this paper as it is too large to be filled into the page.

B. Subspace Pruning

Each space split step leads to n subspaces (nodes) of the space partition tree, but not all subspaces generated in one split are useful for further splitting. We propose several pruning strategies to ensure that only the useful subspaces in each level will be kept in the tree. We define the branches from the root to a node as the node's "path". There are three categories of useless nodes, referring to Figure 2 as an example:

(a) *Nodes that do not satisfy the i-th dimension minimum support threshold.* For example, node g is to be pruned off as not satisfying $minSup^2$. Nodes of type (a) can be easily removed by support threshold checking.

(b) *The i-th ($i \in [2, n]$) child from Branch k($k \in [1, i-1]$) by the cutter whose i-th item has cut the node's path before.* For example, D_1^3 appears in both Cutter 1 and Cutter 2, that is, the 3-*rd* item D_1^3 of Cutter 2 has cut node f's path before, at the position of Cutter 1, resulting in node b. As such, node f, the 3-*rd* child from Branch 2, can be pruned off as it is the subset of node b. Similarly, node e is to be pruned off as the subset of node a.

To remove useless nodes of type (b), we maintain a track set for each node to keep track of the cutter items that have cut its path. Let $T_W = \{T_W^2, \ldots, T_W^i, \ldots, T_W^n\}$ be the track set of node W, where $T_W^i \subseteq D^i$ records the i-*th* item of the cutters that have cut node W's path. When the i-*th* ($i \in [1, n-1]$) child of W is generated by the cutter $z\langle C^n, \ldots, C^i, \ldots, C^1\rangle$, its track set:

$$T^j_{child_i} = \begin{cases} T^j_W, j \in [2, i]; \\ T^j_W \cup C^j, j \in [i+1, n]. \end{cases}$$

That is, the parent's track set is propagated to the i-*th* child with a few updates only on items from T_W^{i+1} to T_W^n. In the initial status, $T_O^i = \emptyset (i \in [2, n])$ for the root node O. For example, when node c, the 2-*nd* child of root O, is generated by Cutter 1 $\langle D_1^4, D_1^3, D_1^2, D_4^1\rangle$, O's track set $T_O^i = \emptyset (i \in [2, 4])$ is propagated to $T_c^2 = \emptyset$, with updates on $T_c^3 = \emptyset \cup D_1^3$ and $T_c^4 = \emptyset \cup D_1^4$, resulting in $T_c = \{\emptyset, D_1^3, D_1^4\}$. Based on the track set of the parent, we develop the i-*th Track Checking* in Theorem 4.1 to remove useless children of type (b).

Theorem 4.1: i-th Track Checking ($i \in [2, n]$). Let $I\langle D_X^n, \ldots, D_X^i \setminus C^i, \ldots, D_X^1\rangle$ be the i-th child of $X\langle D_X^n, \ldots, D_X^i, \ldots, D_X^1\rangle$ generated by cutter $z\langle C^n, \ldots, C^i, \ldots, C^1\rangle$. If $C^i \cap T_X^i \neq \emptyset$, I can be pruned off.

Proof: Since $C^i \cap T_X^i \neq \emptyset$, $C^i \subseteq T_X^i$, hence $\exists z'\langle C'^n, \ldots, C^i, \ldots, C'^1\rangle$ cuts X's ancestor $A\langle D_A^n, \ldots, D_A^i, \ldots, D_A^1\rangle$. Let $S\langle D_S^n, \ldots, D_S^i, \ldots, D_S^1\rangle$ be the i-*th* left sibling of A by cutter z'. Then, we get $D_S^i = D_A^i \setminus C^i$ and $D_A^j \subseteq D_S^j (j \in [1, n], j \neq i)$. Hence, after the application of cutters between z' and z, the offspring of S, say $X'\langle D_{X'}^n, \ldots, D_{X'}^i, \ldots, D_{X'}^1\rangle$, satisfies the condition that $D_{X'}^i = D_X^i \setminus C^i$ and $D_X^j \subseteq D_{X'}^j (j \in [1, n], j \neq i)$. After z is applied, C^i is removed from the i-*th* child of X, resulting in I. Thus, $I \subseteq X'$, and I should be pruned off as a subset of X'. □

For example, node f is the 3-*rd* child of node c by Cutter 2 $\langle D_1^4, D_1^3, D_2^2, D_4^1\rangle$. As $T_c^3 \cap D_1^3 \neq \emptyset$, node f (subset of node

b) is to be pruned off. Similarly, node e (subset of node a) is also pruned off by the *i-th Track Checking* in Theorem 4.1.

(c) *Nodes that are not closed in the i-th dimension ($i \in [2,n]$).* For example, node h is not closed in the 2-*nd* dimension as it is the subset of node d. To remove useless nodes of type (c), we develop the *i-th Dimension Close Checking* in Theorem 4.2.

Theorem 4.2: i-th Dimension Close Checking ($i \in [2,n]$). As for node $X\langle D_X^n, \ldots, D_X^i, \ldots, D_X^1\rangle$, $\exists D_k^i \in D^i \setminus D_X^i$, such that $\forall z\langle C^n, \ldots, C^i, \ldots, C^1\rangle \in Z$ from root to X, where $C^j \subseteq D_X^j (j \in [2,n], j \neq i), C^i = D_k^i$, if $C^1 \cap D_X^1 = \emptyset$, X is not closed in the i-th dimension and can be pruned off. During tree splitting, the i-th branch never satisfies above conditions, so only the other branches need this checking.

Proof: $\exists D_k^i \in D^i \setminus D_X^i$, such that $\forall z\langle C^n, \ldots, C^i, \ldots, C^1\rangle \in Z$ from root to X, where $C^j \subseteq D_X^j (j \in [2,n], j \neq i), C^i = D_k^i$, $C^1 \cap D_X^1 = \emptyset$, then there exists $Y\langle D_X^n, \ldots, D_X^i \cup D_k^i, \ldots, D_X^1\rangle$, which is the superset of X. Hence, X is not closed in the *i-th* dimension and can be pruned off. □

For example, as for node h, $\forall z\langle C^4, C^3, D_1^2, C^1\rangle \in Z$ from root to h, where $C^4 \subseteq D_1^4 D_2^4 D_3^4$, and $C^3 \subseteq D_1^3 D_2^3 D_3^3$, $C^1 \cap D_1^1 D_2^1 D_3^1 = \emptyset$ is satisfied. Hence, h is not closed in the 2-*nd* dimension and can be pruned off as a subset of node d.

V. ALGORITHM HDMINER

In this section, we present HDminer algorithmically. HDminer is a depth-first method to mine high dimensional nFCPs.

Algorithm 1 contains the pseudo-code of HDminer. First, each item $T_O^i (i \in [2,n])$ in the track set T_O is initialized with empty set and the cutter set Z is computed, and then the recursive function *cut*() in Algorithm 2 is called to partition the dataset as well as generate the nFCPs.

Algorithm 1 HDminer

```
1: **HDminer()**
2: Global variables: the set of items $D^i$, monotonic constraint $minSup^i$, $i \in [1,n]$.
3: Input: Matrix $O$
4: Output: $\xi$ the set of nFCPs.
5: $T_O^i \leftarrow empty(), i \in [2,n]$;
6: $Z$ and $|Z|$ are computed from $O$;
7: $\xi \leftarrow cut(O, Z, 1, |Z|, T_o^n, \ldots, T_O^i, \ldots, T_O^2)$;
```

Algorithm 2 shows how a node $O'\langle D'^n, \ldots, D'^1\rangle$ is cut into n branches. It constructs the sets $D^i (i \in [1,n])$, and uses monotonic support threshold constraints simultaneously on each dimension to reduce the search space. When a cutter in Z is applied to cut a node, the cutter must satisfy the condition that each item of the cutter has a non empty intersection with the corresponding item set of the node; otherwise, *cut*() is called with the next cutter (line 6-7).

To build the *i-th* child $O'\langle D'^n, \ldots, D'^i \setminus C^i, \ldots, D'^1\rangle$, three checks are required: monotonic constraint check $minSup^i$ $(D'^i \setminus C^i)$, the *i-th* track check (line 10), and the *i-th* dimension close check ($CloseCheck()$ in Algorithm 3). If O' is not pruned off by the three checks, *cut*() is called to process the newly generated O' recursively with the next cutter, until all cutters are applied. At the same time, the track set of O' is updated.

Algorithm 2 Cutting

```
1: $cut(O', Z, t, |Z|, T_{O'}^n, \ldots, T_{O'}^i, \ldots, T_{O'}^2)$
2: Input: node $O'$, cutter list $Z$, iteration number $t$, $|Z|$ the size of $Z$, $T_{O'}^i (i \in [2,n])$ the track set of $O'$
3: Output: $\xi$ the set of nFCPs.
4: $(C^n, \ldots, C^1) \leftarrow Z[t]$;
5: **if** $t \leq |Z|$ **then**
6:   **if** $C^i \cap D'^i = \emptyset, i \in [1,n]$ **then**
7:     $\xi \leftarrow \xi \cup cut(O', Z, t+1, |Z|, T_{O'}^n, \ldots, T_{O'}^i, \ldots, T_{O'}^2)$;
8:   **else**
9:     **for all** branch $i$, $i \in [1,n]$ **do**
10:      **if** $minSup^i(D'^i \setminus C^i)$ satisfied and $C^i \cap T_{O'}^i = \emptyset$ **then**
11:        $\alpha \leftarrow CloseCheck(O' \setminus C^i, i, Z)$;
12:        **if** $\alpha = 1$ **then**
13:          $\xi \leftarrow \xi \cup cut((O' \setminus C^i, Z, t+1, |Z|, T_{O'}^n \cup C^i, \ldots, T_{O'}^{i+1} \cup C^i, T_{O'}^i, \ldots, T_{O'}^2))$;
14:        **end if**
15:      **end if**
16:    **end for**
17:  **end if**
18: **else**
19:   $\xi \leftarrow O'$;
20: **end if**
21: **return** $\xi$;
```

Algorithm 3 Close Check

```
1: **CloseCheck**$(O'\langle D'^n, \ldots, D'^i, \ldots, D'^1\rangle, i, Z)$
2: Input: node $O'\langle D'^n, \ldots, D'^i, \ldots, D'^1\rangle$, cutting dimension $i$, cutter list $Z$.
3: Output: flag $\alpha$.
4: **for** $t = 2 \ldots n, t \neq i$ **do**
5:   **if** $\exists D_k^i \in D^i \setminus D_X^i$, such that $\forall z\langle C^n, \ldots, C^i, \ldots, C^1\rangle \in Z$, where $C^j \subseteq D_X^j (j \in [2,n], j \neq i), C^i = D_k^i$, $C^1 \cap D_X^1 = \emptyset$ **then**
6:     $\alpha \leftarrow 0$;
7:   **else**
8:     $\alpha \leftarrow 1$;
9:   **end if**
10: **end for**
11: **return** $\alpha$;
```

Theorem 5.1: Let $nFCPs$ be the set of frequent closed patterns of an n-dimensional dataset. Let ξ be the set of leaf nodes derived from HDminer. Then $nFCPs = \xi$. In other words, HDminer can correctly generate all and only all nFCPs.

Proof: First, we prove that $nFCPs \subseteq \xi$. Let $O\langle D^n, \ldots, D^i, \ldots, D^1\rangle$ be the original database, Z be the whole cutter set and P be the set of pruned nodes. Since $nFCPs \subseteq O$, and in the mining tree building process, only cells valued '0' are removed off by cutters and only useless nodes (subsets of other nodes) are pruned off (verified by *Theorem 4.1* and *Theorem 4.2*), hence, $nFCPs \subseteq O \setminus Z \setminus P$, that is, $nFCPs \subseteq \xi$. Second, we prove $\xi \subseteq nFCPs$ by contradiction. Assume there exists a leaf $A \in \xi$ but $A \notin nFCPs$. That is to say, A is a leaf of the mining tree, and it is either not satisfied by the

Density	0.01%	0.1%	1%	5%	10%
min_sup	0.005%	0.05%	0.5%	2.5%	5%
HDminer (sec)	0.2	0.18	0.17	0.31	0.41
Data-Peeler (sec)	675779.34	1658.35	2112.43	2070.59	1942.15
Density	20%	30%	40%	50%	60%
min_sup	10%	15%	20%	25%	30%
HDminer (sec)	0.72	0.82	3.33	17.85	5420.33
Data-Peeler (sec)	1941.4	692.49	1272.76	459.61	11424.12

TABLE IV: Experimental Results for Datasets with Different Densities (running time unit is second)

monotonic support constraints or not closed. During the tree building process, each time a new node is to be generated, it is checked by the monotonic support constraints. If A is not satisfied by the monotonic support constraints, it will be pruned off. Hence, we gather that A is not closed.

Suppose that $A\langle D_A^n, \ldots, D_A^i, \ldots, D_A^1\rangle$ is not closed in the D_A^i set ($i \in [1,n]$), then there exists $A' = \langle D_A^n, \ldots, D_A^i \cup D_A'^i, \ldots, D_A^1\rangle$, where all cells in A' are valued by 1. From the root to A, there should exist a set of cutters Z to cut off $D_A'^i$, and $\forall\langle C^n, \ldots, C^i, \ldots, C^1\rangle \in Z$, $\exists C^j \cap D_A^j = \emptyset$ ($j \in [1,n]$ and $j \neq i$). Given any of A's ancestor $B = \langle D_B^n, \ldots, D_B^j, \ldots, D_B^1\rangle$ derived from a cutter $\langle C^n, \ldots, C^j, \ldots, C^1\rangle \in Z$, $C^j \cap D_B^j = \alpha \neq \emptyset$ ($j \neq i$). As such, from B to A, there must exist cutters (whose *j-th* item contains α) to remove α from D_B^j, resulting in D_A^j. As C^j has cut B's path before, $C^j \subseteq T_B^j$, where T_B^j is B's *j-th* track set. Since $\alpha \subseteq C^j$, $\alpha \subseteq T_B^j$, then the j-th branch child is pruned off and hence no A will be generated, which is contrary to the previous assumption. Hence, we conclude that A is closed.

Now, we have concluded that A is closed and satisfies all monotonic constraints. Hence, $A \in nFCPs$ and our assumption that there exists a leaf $A \in \xi$ but $A \notin nFCPs$ is wrong. That is, $\xi \subseteq nFCPs$. So, our algorithm for mining $nFCPs$ is correct in that $\xi = nFCPs$. □

VI. Experimental Results of HDminer

We have implemented the HDminer in C++, and conducted a performance study to evaluate the efficiency of HDminer against Data-Peeler[15], which is the most recent and popular FCP mining algorithm for high dimensional data. To study the effect of the proposed algorithm on density and high dimensionality, we use synthetic datasets generated by the IBM data generator[1]. Since IBM data generator can only produce 2D dataset, we do a small trick to transform the 2D dataset into the high dimensional dataset we need. For example, if we want to generate a dataset with size of $3 \times 3 \times 3 \times 4$, which is a 4D dataset, we can first run the IBM data generator to generate a 2D data with size of 27×4, then we can slice the 2D dataset into a 4D dataset. Table II gives a good example of how the 2D dataset is sliced into a 4D dataset. We also conducted an experiment on a real 4D arabidopsis gene expression dataset. All experiments are run on a Intel Xeon Server with 16 4-core CPUs of 2.40 GHZ and 24 GB RAM.

A. Varying Data Density

First, we examine the running time of mining nFCP from datasets with different densities (percentage of "1"s in the boolean matrix).

[1]http://www.cs.umbc.edu/c̃giannel/assoc_gen.html

Data Description: The dataset we use to test the scalability of HDminer is IBM synthetic data. We generate ten Boolean datasets with size of $10 \times 10 \times 10 \times 10000$, and the densities range from 0.001% to 60% (with different step size).

Experimental Results: We know that the denser the dataset is, the more patterns the dataset has, given the same minimum support parameters, and hence, the longer the experiment will take. In the experiments, we tune the minimum supports in such a way that the experimental results can be achieved in reasonable time.

Since the minimum supports are different between these ten datasets, we do not study the running time between these datasets for the same algorithm. Instead, we only compare the running time between two algorithms for the same dataset.

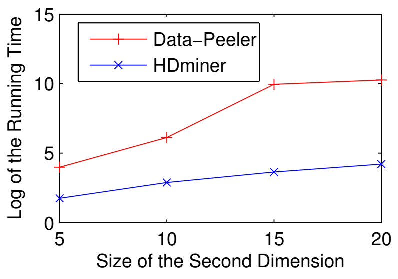
The experimental results are shown in Table IV. We can see that HDminer runs much faster than Data-Peeler with the same minimum support, especially for dense data. We are intended to find a density point before which Data-Peeler works better than HDminer and after which HDminer works better than Data-Peeler. However, even for sparse data (as sparse as we tested in the experiments), HDminer is still much faster then Data-Peeler. For Data-Peeler, notice that for density being 30% and 50%, the running time decreases. This is because there is no pattern found for this configuration.

B. Varying Data Size

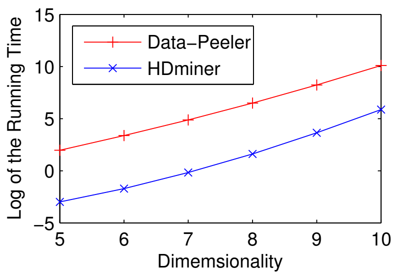
In this section, we will test the efficiency of HDminer when the dataset size varies. The way we perform this test is fixing the density and varying the length of one dimension of the synthetic data. The density of the dataset we generated is 50%. The dataset has an original size of $10 \times 5 \times 10 \times 10000$. We change the second dimension from 5 to 20 by a step of 5. The minimum support for each dimension is set to be 25%. The running time of this experiment is shown in Figure 3a. We can see that the running time increases exponentially when the dataset size increases, and the HDminer takes much less time to process the data than Data-Peeler.

C. Varying Dimensionality

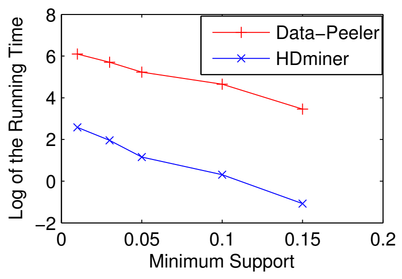
In this section, we will test the efficiency of HDminer when the dataset dimensionality varies. We generated 5 sets of data with the number of dimensions to be 5, 6, 7, 8, 9 and 10. The size of each dimension is 3, except for the last dimension, which is 10000. The density is set to be 30%. The running times of HDminer and Data-Peeler are shown in Figure 3b. Still, we can see that both the running times of HDminer and Data-Peeler increase exponentially when the number of dimension increases, the running time of Data-Peeler is much higher than that of HDminer.



(a) Varying Data Size



(b) Varying Dimensionality



(c) Varying Minimum Support

Fig. 3: Experimental results for varying data size, dimensionality and minimum support (running time unit is second, and the base of log operation is 2.)

Dimension	Itemset
Data source	shoots, roots
Treatment	cold, drought, genotoxic, heat, osmotic, oxidative, salt, UVB, wound
Time point	15min, 30min, 1h, 3h, 6h, 12h, 24h
Genes	7161 genes

TABLE V: Description of Arabidopsis Gene Expression Itemsets for Each Dimension

D. Parameter Study

In this section, we are going to do a parameter study on a real world dataset the Arabidopsis Gene Expression Data to test how the minimum support affects the running time of HDminer and Data-Peeler.

Data Description: The Arabidopsis Gene Expression Data are downloaded from Tair[2], NASC[3] and NCBI[4], preprocessed to form a gene up-regulation data matrix of size $2\times9\times7\times7161$. The density of this dataset is 31.55%. The description of the itemsets for each dimension is shown in Table V.

Experimental Results: We set the minimum support of sources, treatments and time points to be 1, 1, 3 to ensure enough FCPs in the results. The minimum support for gene dimension is the parameter we focus on. We vary the min_sup for gene dimension from 1% to 15%. The result is shown in Figure 3c. The running times for both of the Data-Peeler and HDminer decrease when the minimum support increases, but the running time for HDminer is much shorter than that for Data-Peeler.

VII. Conclusion

In this paper, we have proposed a novel algorithm HDminer for mining high dimensional frequent closed patterns (nFCPs) on dense data. We have employed effective search space partition and pruning strategies for HDminer to enhance the mining efficiency. Our performance study on synthetic data and real dense microarray data shows that HDminer outperforms the state-of-art algorithm Data-Peeler[15] on dense data scenarios.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pages 487-499, 1994.

[2] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *KDD'94*, pages 181-192, 1994.

[3] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *KDD'97*, pages 283-286, 1997.

[4] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *SIGMOD'00*, pages 22-23, 2000.

[5] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT'99*, pages 398-416, 1999.

[6] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD Int. Workshop Data Mining and Knowledge Discovery*, pages 21-30, 2000.

[7] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *KDD'03*, pages 236-245, 2003.

[8] L.C. Hsien, Y.D. Lin, W. Jungpin, and Y.K. Cheng. Efficient mining of frequent closed itemsets without closure checking. In *Eighth International Conference on Intelligent Systems Design and Applications*, pages 269-274, 2008.

[9] M. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed association rule mining. In *SDM'02*, pages 457-473, 2002.

[10] F. Pan, G. Cong, and A. K. H. Tung. Carpenter: Finding closed patterns in long biological datasets. In *KDD'03*, pages 673-642, 2003.

[11] G. Cong, K.L. Tan, A. K. H. Tung, and F. Pan. Mining frequent closed patterns in microarray data. In *ICDM'04*, pages 363-366, 2004.

[12] J. Besson, C. Robardet, and J.-F. Boulicaut. Constraint-based mining of formal concepts in transactional data. In *PaKDD'04*, pages 615-624, 2004.

[13] L. Ji, K.L. Tan, and A.K.H. Tung. Compressed hierarchical mining of frequent closed patterns from dense datasets. In *IEEE Transactions on Knowledge and Data Engineering (TKDE'07)*, volume 19, No. 9, pages 1175-1187, 2007.

[14] L. Ji, K.L. Tan, and A.K.H. Tung. Mining frequent closed cubes in 3D datasets. In *VLDB'06*, pages 811-822, 2006.

[15] L. Cerf, J. Besson, C. Robardet, and J. Boulicaut. Closed patterns meet n-ary relations. In *TKDD'09*, Volume 3, Issue 1, pages 1-33, 2009.

[16] F. Pang, A. K. H. Tung, G. Cong, and X. Xu. Cobbler: Combining column and row enumeration for closed pattern discovery. In *SSDBMS'04*, pages 21-30, 2004.

[2]http://www.arabidopsis.org

[3]http://affymetrix.arabidopsis.info/narrays/experimentbrowse.pl

[4]http://http://www.ncbi.nlm.nih.gov