Data mining and Machine learning

Part 8. Itemset Mining

Frequent Itemset Mining

In many applications one is interested in how often two or more objects of interest co-occur, the so-called *itemsets*.

The prototypical application was *market basket analysis*, that is, to mine the sets of items that are frequently bought together at a supermarket by analyzing the customer shopping carts (the so-called "market baskets").

Frequent itemset mining is a basic exploratory mining task, since the since the basic operation is to find the co-occurrence, which gives an estimate for the joint probability mass function.

Once we mine the frequent sets, they allow us to extract *association rules* among the itemsets, where we make some statement about how likely are two sets of items to co-occur or to conditionally occur.

Frequent Itemsets: Terminology

Itemsets: Let $\mathcal{I} = \{x_1, x_2, \ldots, x_m\}$ be a set of elements called *items*. A set $X \subseteq \mathcal{I}$ is called an *itemset*. An itemset of cardinality (or size) k is called a k-itemset. Further, we denote by $\mathcal{I}^{(k)}$ the set of all k-itemsets, that is, subsets of $\mathcal{I}$ with size k.

Tidsets: Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be another set of elements called transaction identifiers or *tids*. A set $T \subseteq \mathcal{T}$ is called a *tidset*. Itemsets and tidsets are kept sorted in lexicographic order.

Transactions: A *transaction* is a tuple of the form $\langle t, X \rangle$, where $t \in \mathcal{T}$ is a unique transaction identifier, and X is an itemset.

Database: A binary database $\boldsymbol{D}$ is a binary relation on the set of tids and items, that is, $\boldsymbol{D} \subseteq \mathcal{T} \times \mathcal{I}$. We say that tid $t \in \mathcal{T}$ *contains* item $x \in \mathcal{I}$ iff $(t, x) \in \boldsymbol{D}$. In other words, $(t, x) \in \boldsymbol{D}$ iff $x \in X$ in the tuple $\langle t, X \rangle$. We say that tid t *contains* itemset $X = \{x_1, x_2, \ldots, x_k\}$ iff $(t, x_i) \in \boldsymbol{D}$ for all $i = 1, 2, \ldots, k$.

Database Representation

Let 2^X denote the powerset of X, that is, the set of all subsets of X. Let $\boldsymbol{i} : 2^{\mathcal{T}} \rightarrow 2^{\mathcal{I}}$ be a function, defined as follows:

$$\boldsymbol{i}(T) = \{x \mid \forall t \in T,\ t \text{ contains } x\}$$

where $T \subseteq \mathcal{T}$, and $\boldsymbol{i}(T)$ is the set of items that are common to *all* the transactions in the tidset T. In particular, $\boldsymbol{i}(t)$ is the set of items contained in tid $t \in \mathcal{T}$.

Let $\boldsymbol{t} : 2^{\mathcal{I}} \rightarrow 2^{\mathcal{T}}$ be a function, defined as follows:

$$\boldsymbol{t}(X) = \{t \mid t \in \mathcal{T} \text{ and } t \text{ contains } X\} \tag{1}$$

where $X \subseteq \mathcal{I}$, and $\boldsymbol{t}(X)$ is the set of tids that contain *all* the items in the itemset X. In particular, $\boldsymbol{t}(x)$ is the set of tids that contain the single item $x \in \mathcal{I}$.

The binary database $\boldsymbol{D}$ can be represented as a *horizontal* or *transaction database* consisting of tuples of the form $\langle t, \boldsymbol{i}(t) \rangle$, with $t \in \mathcal{T}$.

The binary database $\boldsymbol{D}$ can also be represented as a *vertical* or *tidset database* containing a collection of tuples of the form $\langle x, \boldsymbol{t}(x) \rangle$, with $x \in \mathcal{I}$.

Binary Database: Transaction and Vertical Format

$\boldsymbol{D}$	A	B	C	D	E
1	1	1	0	1	1
2	0	1	1	0	1
3	1	1	0	1	1
4	1	1	1	0	1
5	1	1	1	1	1
6	0	1	1	1	0

Binary Database

t	$\boldsymbol{i}(t)$
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

Transaction Database

$\boldsymbol{t}(x)$				
A	B	C	D	E
1	1	2	1	1
3	2	4	3	2
4	3	5	5	3
5	4	6	6	4
	5			5
	6			

Vertical Database

This dataset $\boldsymbol{D}$ has 5 items, $\mathcal{I} = \{A, B, C, D, E\}$ and 6 tids $\mathcal{T} = \{1, 2, 3, 4, 5, 6\}$.

The the first transaction is $\langle 1, \{A, B, D, E\}\rangle$, where we omit item C since $(1, C) \notin \boldsymbol{D}$. Henceforth, for convenience, we drop the set notation for itemsets and tidsets. Thus, we write $\langle 1, \{A, B, D, E\}\rangle$ as $\langle 1, ABDE\rangle$.

Support and Frequent Itemsets

The *support* of an itemset X in a dataset $\boldsymbol{D}$, denoted $sup(X)$, is the number of transactions in $\boldsymbol{D}$ that contain X:

$$sup(X) = \left|\{t \mid \langle t, \boldsymbol{i}(t)\rangle \in \boldsymbol{D} \text{ and } X \subseteq \boldsymbol{i}(t)\}\right| = |\boldsymbol{t}(X)|$$

The *relative support* of X is the fraction of transactions that contain X:

$$rsup(X) = \frac{sup(X)}{|\boldsymbol{D}|}$$

It is an estimate of the *joint probability* of the items comprising X.

An itemset X is said to be *frequent* in $\boldsymbol{D}$ if $sup(X) \geq minsup$, where *minsup* is a user defined *minimum support threshold*.

The set $\mathcal{F}$ to denotes the set of all frequent itemsets, and $\mathcal{F}^{(k)}$ denotes the set of frequent k-itemsets.

Frequent Itemsets

Minimum support: $minsup = 3$

t	$\boldsymbol{i}(t)$
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

Transaction Database

sup	itemsets
6	B
5	E, BE
4	$A, C, D, AB, AE, BC, BD, ABE$
3	$AD, CE, DE, ABD, ADE, BCE, BDE, ABDE$

Frequent Itemsets

The 19 frequent itemsets shown in the table comprise the set $\mathcal{F}$. The sets of all frequent k-itemsets are

$$\mathcal{F}^{(1)} = \{A, B, C, D, E\}$$
$$\mathcal{F}^{(2)} = \{AB, AD, AE, BC, BD, BE, CE, DE\}$$
$$\mathcal{F}^{(3)} = \{ABD, ABE, ADE, BCE, BDE\}$$
$$\mathcal{F}^{(4)} = \{ABDE\}$$

Itemset Mining Algorithms: Brute Force

The brute-force algorithm enumerates all the possible itemsets $X \subseteq \mathcal{I}$, and for each such subset determines its support in the input dataset $\boldsymbol{D}$. The method comprises two main steps: (1) candidate generation and (2) support computation.

Candidate Generation: This step generates all the subsets of $\mathcal{I}$, which are called *candidates*, as each itemset is potentially a candidate frequent pattern. The candidate itemset search space is clearly exponential because there are $2^{|\mathcal{I}|}$ potentially frequent itemsets.

Support Computation: This step computes the support of each candidate pattern X and determines if it is frequent. For each transaction $\langle t, \boldsymbol{i}(t) \rangle$ in the database, we determine if X is a subset of $\boldsymbol{i}(t)$. If so, we increment the support of X.

Computational Complexity: Support computation takes time $O(|\mathcal{I}| \cdot |\boldsymbol{D}|)$ in the worst case, and because there are $O(2^{|\mathcal{I}|})$ possible candidates, the computational complexity of the brute-force method is $O(|\mathcal{I}| \cdot |\boldsymbol{D}| \cdot 2^{|\mathcal{I}|})$.

Brute Force Algorithm

BruteForce ($\boldsymbol{D}$, $\mathcal{I}$, *minsup*):

```
1 F ← ∅ // set of frequent itemsets
2
3 foreach X ⊆ I do
4 |   sup(X) ← ComputeSupport (X, D)
5 |   if sup(X) ≥ minsup then
6 |   |   F ← F ∪ {(X, sup(X))}
7 return F
```

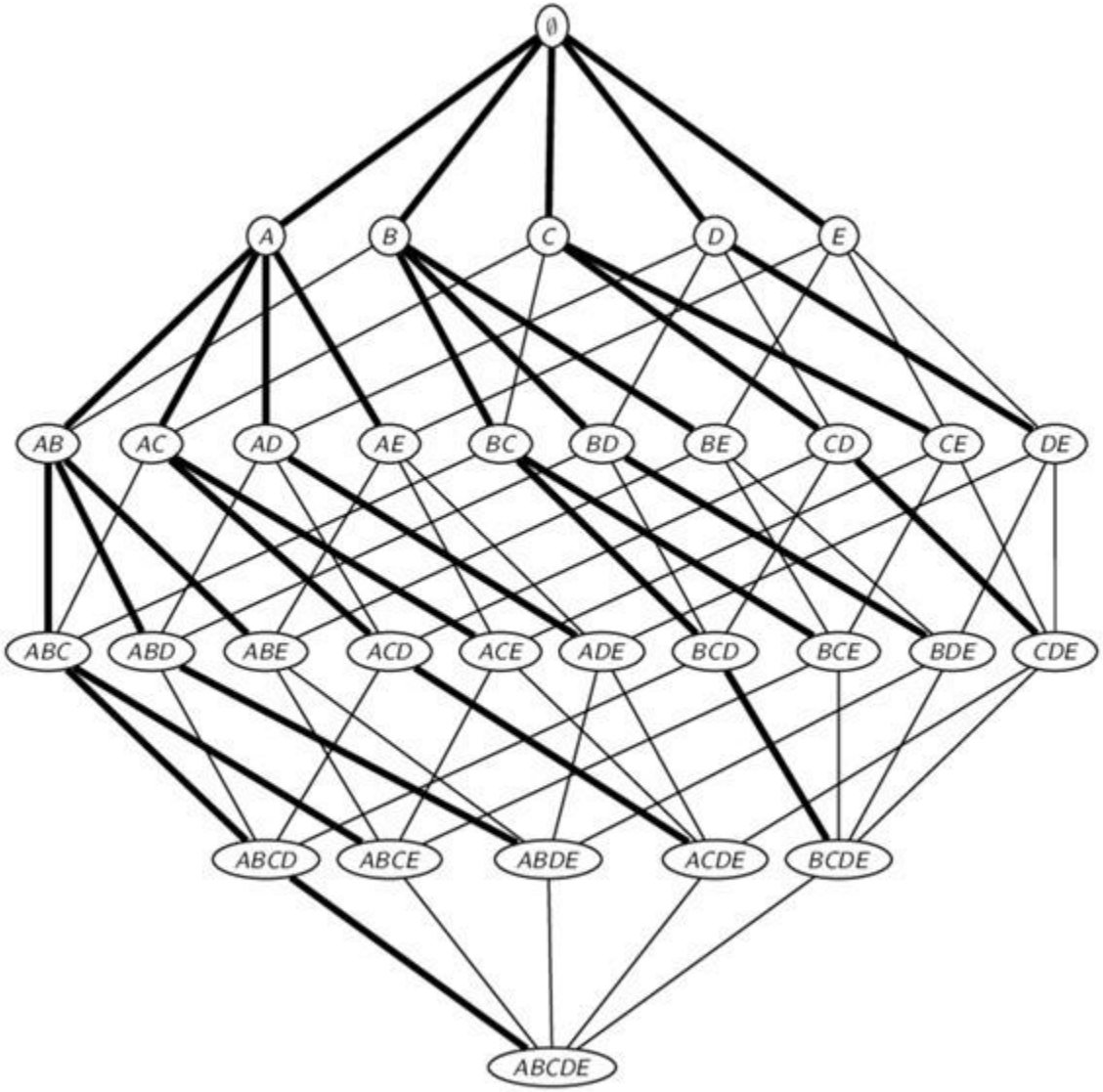
ComputeSupport ($X, \boldsymbol{D}$):

```
1 sup(X) ← 0
2 foreach ⟨t, i(t)⟩ ∈ D do
3 |   if X ⊆ i(t) then
4 |   |   sup(X) ← sup(X) + 1
5 return sup(X)
```

Itemset lattice and prefix-based search tree

Itemset search space is a lattice where any two itemsets X and Y are connected by a link iff X is an *immediate subset* of Y, that is, $X \subseteq Y$ and $|X| = |Y| - 1$.

Frequent itemsets can enumerated using either a BFS or DFS search on the *prefix tree*, where two itemsets X, Y are connected by a link iff X is an immediate subset and prefix of Y. This allows one to enumerate itemsets starting with an empty set, and adding one more item at a time.



Level-wise Approach: Apriori Algorithm

If $X \subseteq Y$, then $sup(X) \geq sup(Y)$, which leads to the following two observations: (1) if X is frequent, then any subset $Y \subseteq X$ is also frequent, and (2) if X is not frequent, then any superset $Y \supseteq X$ cannot be frequent.

The *Apriori algorithm* utilizes these two properties to significantly improve the brute-force approach. It employs a level-wise or breadth-first exploration of the itemset search space, and prunes all supersets of any infrequent candidate, as no superset of an infrequent itemset can be frequent. It also avoids generating any candidate that has an infrequent subset.

In addition to improving the candidate generation step via itemset pruning, the Apriori method also significantly improves the I/O complexity. Instead of counting the support for a single itemset, it explores the prefix tree in a breadth-first manner, and computes the support of all the valid candidates of size k that comprise level k in the prefix tree.

The Apriori Algorithm

Apriori ($\boldsymbol{D}$, $\mathcal{I}$, *minsup*):

1 $\mathcal{F} \leftarrow \emptyset$

2 $\mathcal{C}^{(1)} \leftarrow \{\emptyset\}$ // Initial prefix tree with single items

3 **foreach** $i \in \mathcal{I}$ **do** Add i as child of $\emptyset$ in $\mathcal{C}^{(1)}$ with $sup(i) \leftarrow 0$

4 $k \leftarrow 1$ // k denotes the level

5 **while** $\mathcal{C}^{(k)} \neq \emptyset$ **do**

6 ComputeSupport ($\mathcal{C}^{(k)}, \boldsymbol{D}$)

7 **foreach** *leaf* $X \in \mathcal{C}^{(k)}$ **do**

8 **if** $sup(X) \geq minsup$ **then** $\mathcal{F} \leftarrow \mathcal{F} \cup \{(X, sup(X))\}$

9 **else** remove X from $\mathcal{C}^{(k)}$

10 $\mathcal{C}^{(k+1)} \leftarrow$ ExtendPrefixTree ($\mathcal{C}^{(k)}$)

11 $k \leftarrow k + 1$

12 **return** $\mathcal{F}^{(k)}$

Apriori Algorithm

ComputeSupport $(\mathcal{C}^{(k)}, \boldsymbol{D})$**:**

foreach $\langle t, \boldsymbol{i}(t) \rangle \in \boldsymbol{D}$ **do**
  foreach *k-subset* $X \subseteq \boldsymbol{i}(t)$ **do**
    if $X \in \mathcal{C}^{(k)}$ **then** $sup(X) \leftarrow sup(X) + 1$

ExtendPrefixTree $(\mathcal{C}^{(k)})$**:**

foreach *leaf* $X_a \in \mathcal{C}^{(k)}$ **do**
  foreach *leaf* $X_b \in \text{sibling}(X_a)$, *such that* $b > a$ **do**
    $X_{ab} \leftarrow X_a \cup X_b$
    `// prune if there are any infrequent subsets`
    if $X_j \in \mathcal{C}^{(k)}$, ***for all*** $X_j \subset X_{ab}$, *such that* $|X_j| = |X_{ab}| - 1$ **then**
      Add X_{ab} as child of X_a with $sup(X_{ab}) \leftarrow 0$
  if *no extensions from* X_a **then**
    remove X_a and its ancestors with no extensions from $\mathcal{C}^{(k)}$
return $\mathcal{C}^{(k)}$

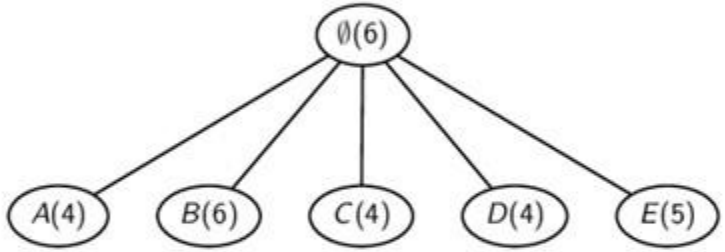
Itemset Mining: Apriori Algorithm

Infrequent itemsets in gray

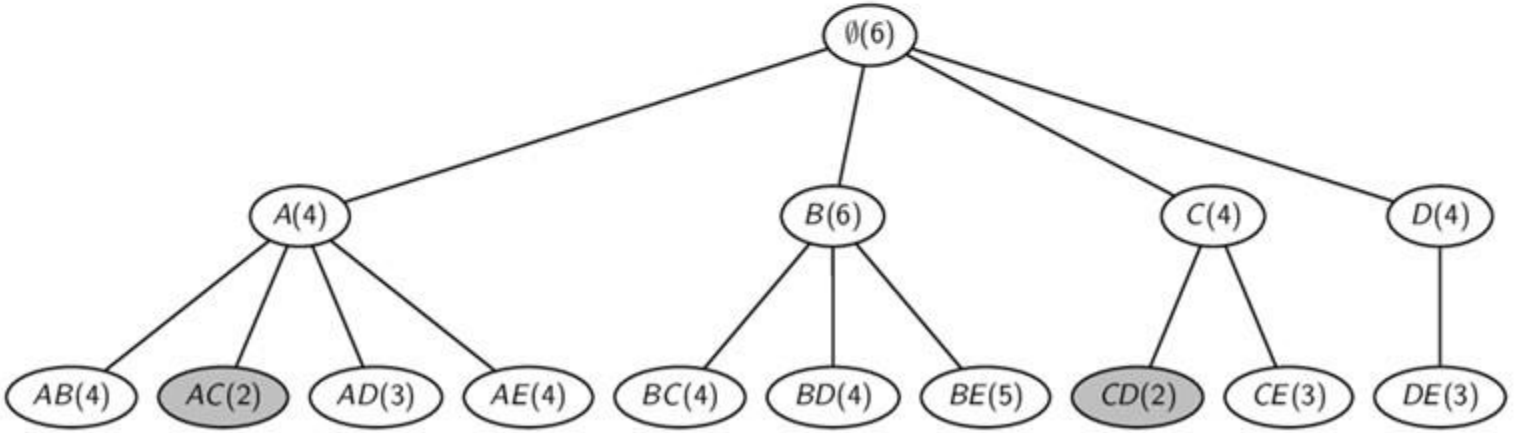
D

t	$i(t)$
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

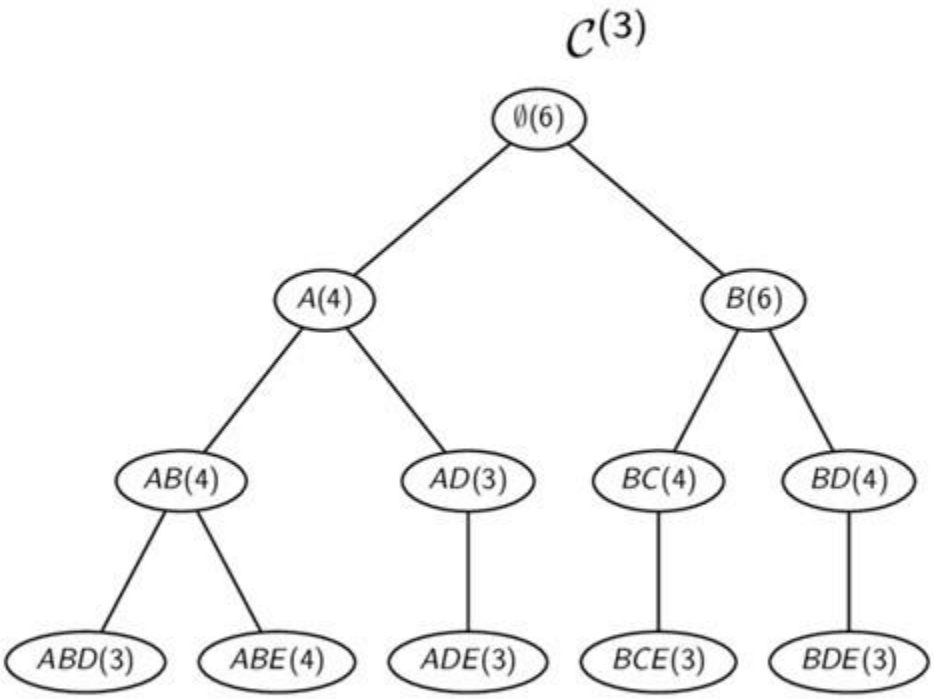
$\mathcal{C}^{(1)}$

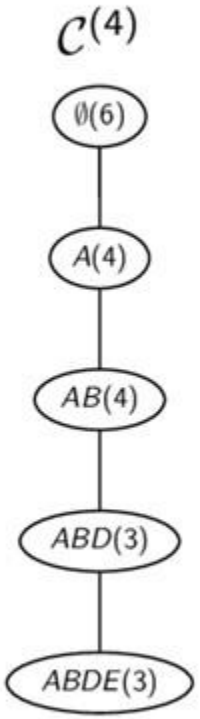


$\mathcal{C}^{(2)}$

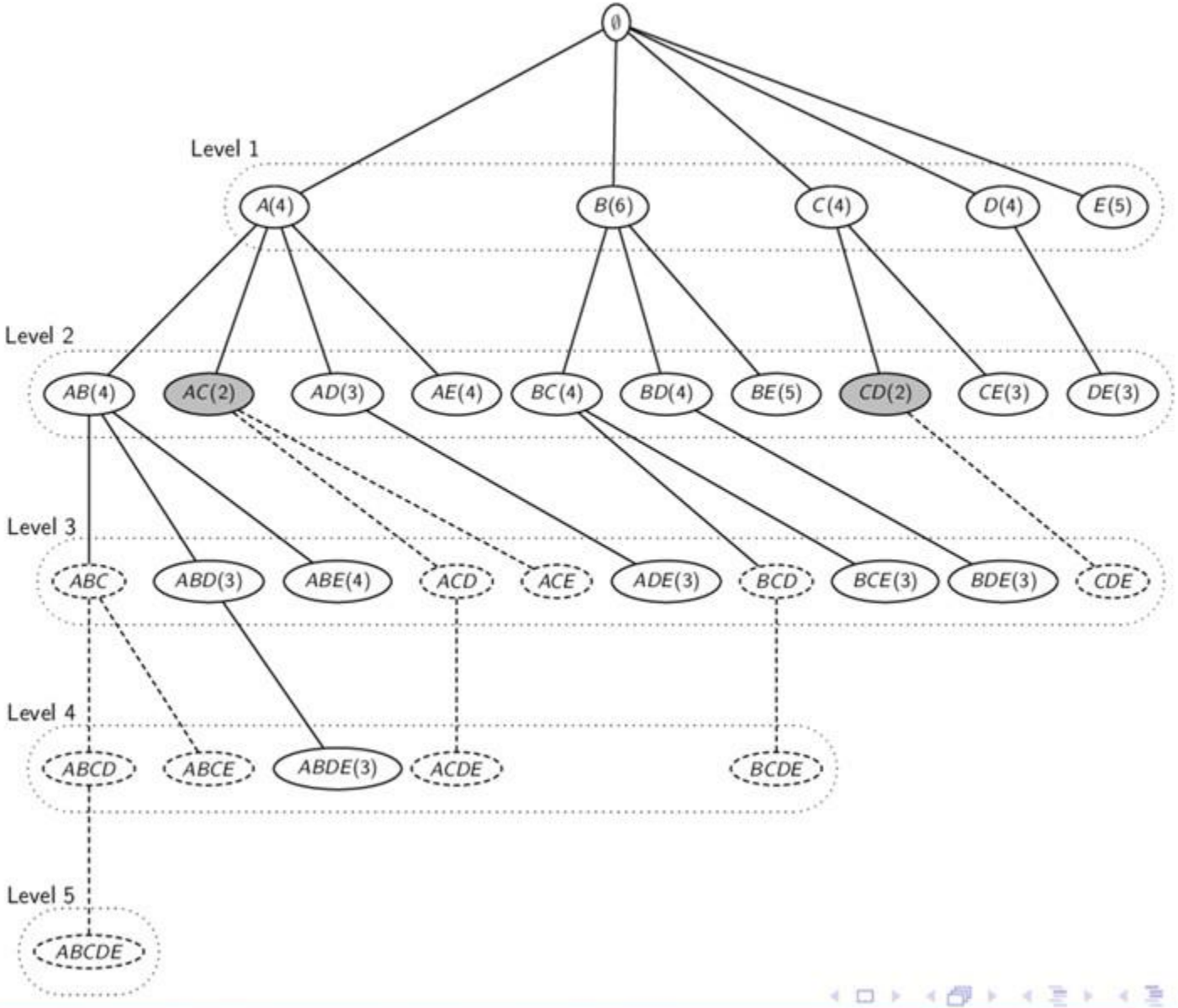


Itemset Mining: Apriori Algorithm





Apriori Algorithm: Prefix Search Tree and Pruning



Apriori Algorithm: Details

Let $\mathcal{C}^{(k)}$ denote the prefix tree comprising all the candidate k-itemsets.

Apriori begins by inserting the single items into an initially empty prefix tree to populate $\mathcal{C}^{(1)}$.

The support for the current candidates is obtained via ComputeSupport procedure that generates k-subsets of each transaction in the database $\boldsymbol{D}$, and for each such subset it increments the support of the corresponding candidate in $\mathcal{C}^{(k)}$ if it exists. Next, we remove any infrequent candidate.

The leaves of the prefix tree that survive comprise the set of frequent k-itemsets $\mathcal{F}^{(k)}$, which are used to generate the candidate $(k+1)$-itemsets for the next level. The ExtendPrefixTree procedure employs prefix-based extension for candidate generation. Given two frequent k-itemsets X_a and X_b with a common $k-1$ length prefix, that is, given two sibling leaf nodes with a common parent, we generate the $(k+1)$-length candidate $X_{ab} = X_a \cup X_b$. This candidate is retained only if it has no infrequent subset. Finally, if a k-itemset X_a has no extension, it is pruned from the prefix tree, and we recursively prune any of its ancestors with no k-itemset extension, so that in $\mathcal{C}^{(k)}$ all leaves are at level k.

If new candidates were added, the whole process is repeated for the next level. This process continues until no new candidates are added.

Tidset Intersection Approach: Eclat Algorithm

The support counting step can be improved significantly if we can index the database in such a way that it allows fast frequency computations.

The Eclat algorithm leverages the tidsets directly for support computation. The basic idea is that the support of a candidate itemset can be computed by intersecting the tidsets of suitably chosen subsets. In general, given $\boldsymbol{t}(X)$ and $\boldsymbol{t}(Y)$ for any two frequent itemsets X and Y, we have

$$\boldsymbol{t}(XY) = \boldsymbol{t}(X) \cap \boldsymbol{t}(Y)$$

The support of candidate XY is simply the cardinality of $\boldsymbol{t}(XY)$, that is, $sup(XY) = |\boldsymbol{t}(XY)|$.

Eclat intersects the tidsets only if the frequent itemsets share a common prefix, and it traverses the prefix search tree in a DFS-like manner, processing a group of itemsets that have the same prefix, also called a *pref ix equivalence class*.

Eclat Algorithm

// Initial Call: $\mathcal{F} \leftarrow \emptyset, P \leftarrow \{\langle i, \boldsymbol{t}(i)\rangle \mid i \in \mathcal{I}, |\boldsymbol{t}(i)| \geq minsup\}$

Eclat (P, *minsup*, $\mathcal{F}$):

1 **foreach** $\langle X_a, \boldsymbol{t}(X_a)\rangle \in P$ **do**

2 $\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{(X_a, sup(X_a))\}$

3 $\quad P_a \leftarrow \emptyset$

4 $\quad$ **foreach** $\langle X_b, \boldsymbol{t}(X_b)\rangle \in P$, *with* $X_b > X_a$ **do**

5 $\quad\quad X_{ab} = X_a \cup X_b$

6 $\quad\quad \boldsymbol{t}(X_{ab}) = \boldsymbol{t}(X_a) \cap \boldsymbol{t}(X_b)$

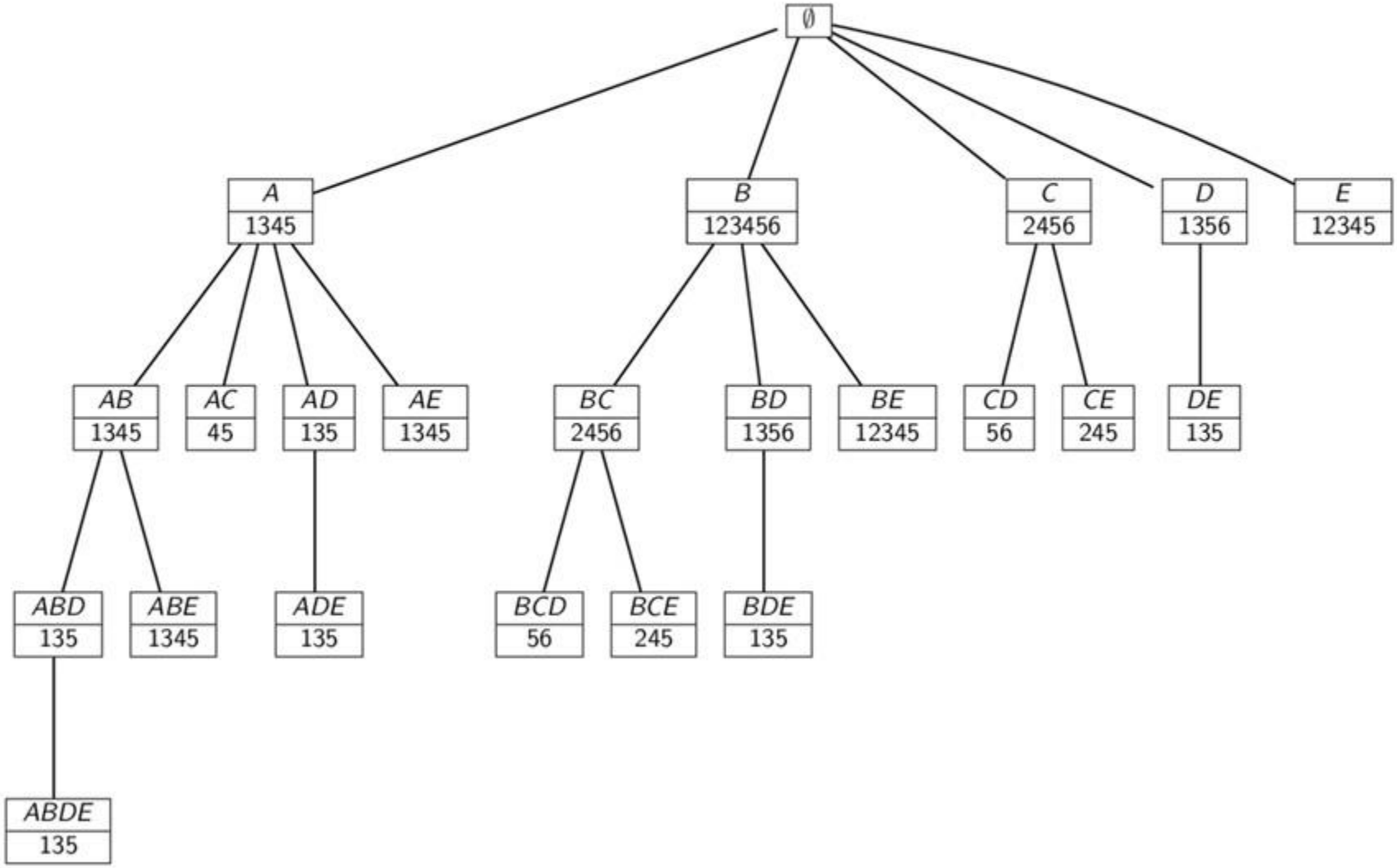
7 $\quad\quad$ **if** $sup(X_{ab}) \geq minsup$ **then**

8 $\quad\quad\quad P_a \leftarrow P_a \cup \{\langle X_{ab}, \boldsymbol{t}(X_{ab})\rangle\}$

9 $\quad$ **if** $P_a \neq \emptyset$ **then** Eclat (P_a, *minsup*, $\mathcal{F}$)

Eclat Algorithm: Tidlist Intersections

Infrequent itemsets in gray



Diffsets: Difference of Tidsets

The Eclat algorithm can be significantly improved if we can shrink the size of the intermediate tidsets. This can be achieved by keeping track of the differences in the tidsets as opposed to the full tidsets.

Let $X_a = \{x_1, \ldots, x_{k-1}, x_a\}$ and $X_b = \{x_1, \ldots, x_{k-1}, x_b\}$, so that $X_{ab} = X_a \cup X_b = \{x_1, \ldots, x_{k-1}, x_a, x_b\}$.

The *diffset* of X_{ab} is the set of tids that contain the prefix X_a, but not the item X_b

$$\boldsymbol{d}(X_{ab}) = \boldsymbol{t}(X_a) \setminus \boldsymbol{t}(X_{ab}) = \boldsymbol{t}(X_a) \setminus \boldsymbol{t}(X_b)$$

We can obtain an expression for $\boldsymbol{d}(X_{ab})$ in terms of $\boldsymbol{d}(X_a)$ and $\boldsymbol{d}(X_b)$ as follows:

$$\boldsymbol{d}(X_{ab}) = \boldsymbol{d}(X_b) \setminus \boldsymbol{d}(X_a)$$

which means that we can replace all intersection operations in Eclat with diffset operations.

Diffsets: Difference of Tidsets

Given $\mathcal{T} = 123456$:

$$\boldsymbol{d}(A) = \mathcal{T} \setminus 1345 = 26 \qquad \boldsymbol{d}(B) = \mathcal{T} \setminus 123456 = \emptyset$$
$$\boldsymbol{d}(C) = \mathcal{T} \setminus 2456 = 13 \qquad \boldsymbol{d}(D) = \mathcal{T} \setminus 1356 = 24$$
$$\boldsymbol{d}(E) = \mathcal{T} \setminus 12345 = 6$$

For instance, the diffsets of AB and AC are given as

$$\boldsymbol{d}(AB) = \boldsymbol{d}(B) \setminus \boldsymbol{d}(A) = \emptyset \setminus \{2,6\} = \emptyset$$
$$\boldsymbol{d}(AC) = \boldsymbol{d}(C) \setminus \boldsymbol{d}(A) = \{1,3\} \setminus \{2,6\} = 13$$

and their support values are

$$sup(AB) = sup(A) - |\boldsymbol{d}(AB)| = 4 - 0 = 4$$
$$sup(AC) = sup(A) - |\boldsymbol{d}(AC)| = 4 - 2 = 2$$

The new prefix equivalence class P_A is:

$$P_A = \{\langle AB, \emptyset, 4\rangle, \langle AD, 4, 3\rangle, \langle AE, \emptyset, 4\rangle\}$$

Algorithm dEclat

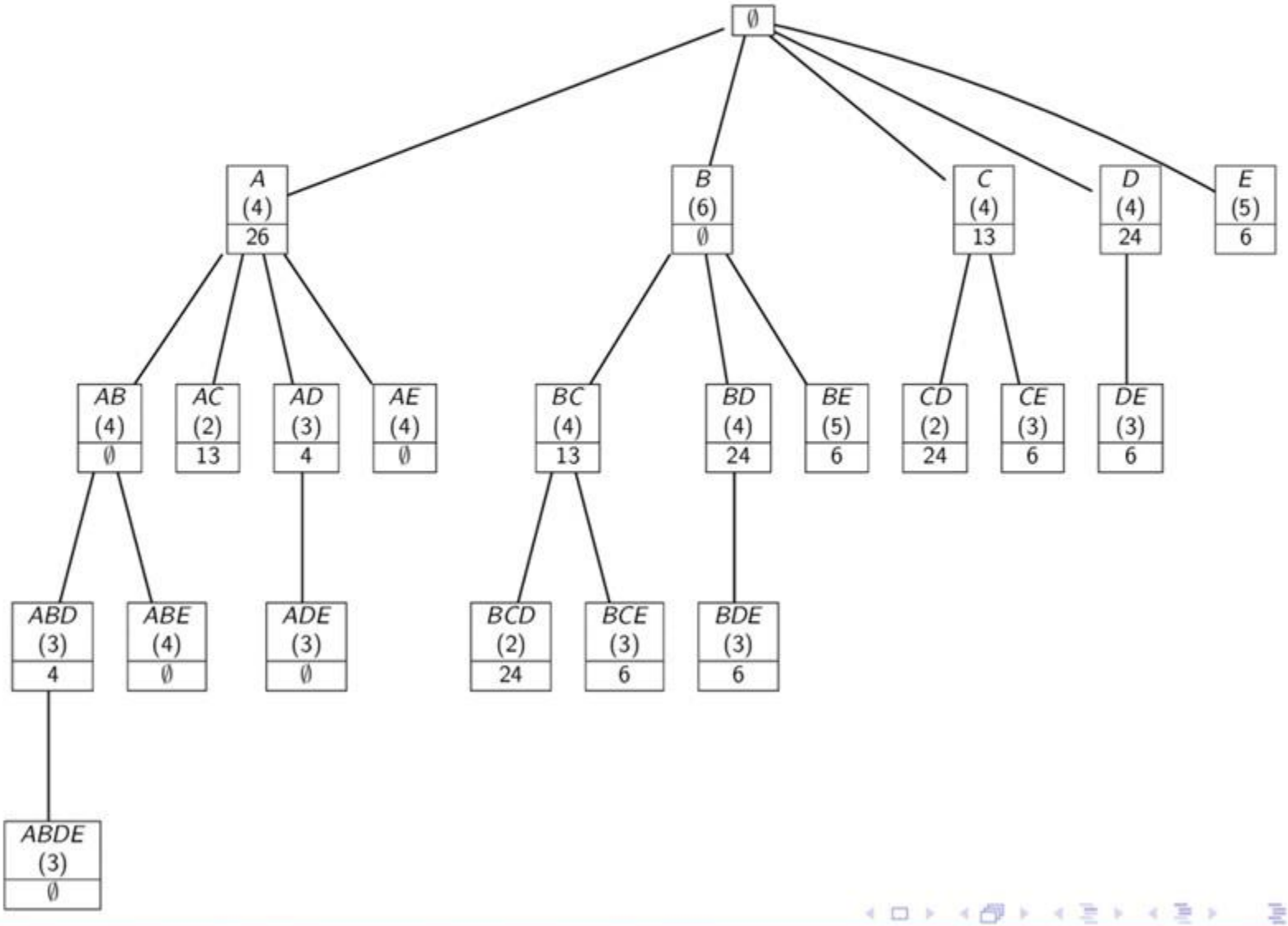
// Initial Call: $\mathcal{F} \leftarrow \emptyset$,
$P \leftarrow \{\langle i, \boldsymbol{d}(i), sup(i)\rangle \mid i \in \mathcal{I}, \boldsymbol{d}(i) = \mathcal{T} \setminus \boldsymbol{t}(i), sup(i) \geq minsup\}$

dEclat (P, *minsup*, $\mathcal{F}$)**:**

1 **foreach** $\langle X_a, \boldsymbol{d}(X_a), sup(X_a)\rangle \in P$ **do**
2 $\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{(X_a, sup(X_a))\}$
3 $\quad P_a \leftarrow \emptyset$
4 $\quad$ **foreach** $\langle X_b, \boldsymbol{d}(X_b), sup(X_b)\rangle \in P$, *with* $X_b > X_a$ **do**
5 $\quad\quad X_{ab} = X_a \cup X_b$
6 $\quad\quad \boldsymbol{d}(X_{ab}) = \boldsymbol{d}(X_b) \setminus \boldsymbol{d}(X_a)$
7 $\quad\quad sup(X_{ab}) = sup(X_a) - |\boldsymbol{d}(X_{ab})|$
8 $\quad\quad$ **if** $sup(X_{ab}) \geq minsup$ **then**
9 $\quad\quad\quad P_a \leftarrow P_a \cup \{\langle X_{ab}, \boldsymbol{d}(X_{ab}), sup(X_{ab})\rangle\}$
10 $\quad$ **if** $P_a \neq \emptyset$ **then** dEclat (P_a, *minsup*, $\mathcal{F}$)

dEclat Algorithm: Diffsets

support shown within brackets; infrequent itemsets in gray



Frequent Pattern Tree Approach: FPGrowth Algorithm

The FPGrowth method indexes the database for fast support computation via the use of an augmented prefix tree called the *frequent pattern tree* (FP-tree).

Each node in the tree is labeled with a single item, and each child node represents a different item. Each node also stores the support information for the itemset comprising the items on the path from the root to that node.

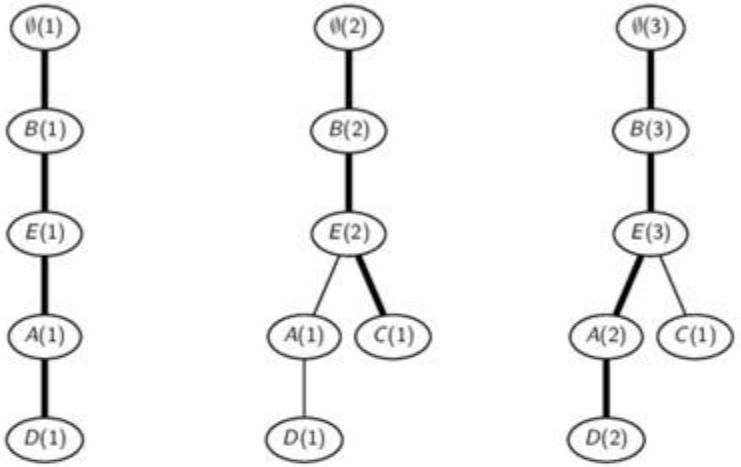
The FP-tree is constructed as follows. Initially the tree contains as root the null item $\emptyset$. Next, for each tuple $\langle t, X \rangle \in \boldsymbol{D}$, where $X = \boldsymbol{i}(t)$, we insert the itemset X into the FP-tree, incrementing the count of all nodes along the path that represents X.

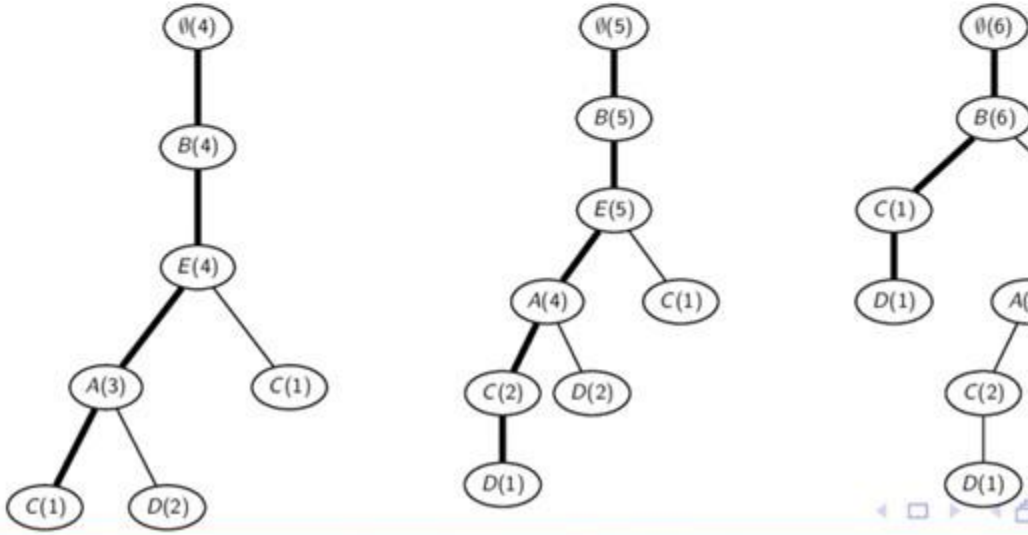
If X shares a prefix with some previously inserted transaction, then X will follow the same path until the common prefix. For the remaining items in X, new nodes are created under the common prefix, with counts initialized to 1. The FP-tree is complete when all transactions have been inserted.

Frequent Pattern Tree

The FP-tree is a prefix compressed representation of $\boldsymbol{D}$. For most compression items are sorted in descending order of support.

Transactions
BEAD
BEC
BEAD
BEAC
BEACD
BCD





FPGrowth Algorithm: Rationale

Given an FP-tree R, projected FP-trees are built for each frequent item i in R in increasing order of support in a recursive manner.

To project R on item i, we find all the occurrences of i in the tree, and for each occurrence, we determine the corresponding path from the root to i. The count of item i on a given path is recorded in $cnt(i)$ and the path is inserted into the new projected tree R_X, where X is the itemset obtained by extending the prefix P with the item i. While inserting the path, the count of each node in R_X along the given path is incremented by the path count $cnt(i)$.

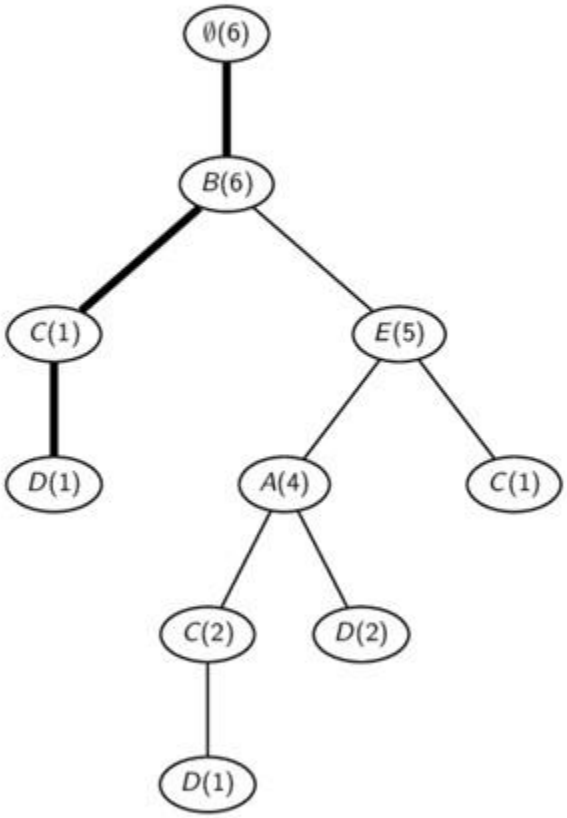
The base case for the recursion happens when the input FP-tree R is a single path. FP-trees that are paths are handled by enumerating all itemsets that are subsets of the path, with the support of each such itemset being given by the least frequent item in it.

FPGrowth Algorithm

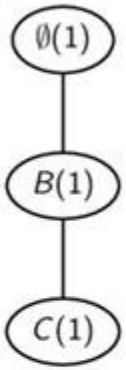
```
// Initial Call: R ← FP-tree(D), P ← ∅, F ← ∅
FPGrowth (R, P, F, minsup):
1  Remove infrequent items from R
2  if IsPath(R) then // insert subsets of R into F
3      foreach Y ⊆ R do
4          X ← P ∪ Y
5          sup(X) ← min_{x∈Y}{cnt(x)}
6          F ← F ∪ {(X, sup(X))}
7  else    // process projected FP-trees for each frequent item i
8      foreach i ∈ R in increasing order of sup(i) do
9          X ← P ∪ {i}
10         sup(X) ← sup(i) // sum of cnt(i) for all nodes labeled i
11
12         F ← F ∪ {(X, sup(X))}
13         R_X ← ∅ // projected FP-tree for X
14
15         foreach path ∈ PathFromRoot(i) do
16             cnt(i) ← count of i in path
17             Insert path, excluding i, into FP-tree R_X with count cnt(i)
18         if R_X ≠ ∅ then FPGrowth (R_X, X, F, minsup)
```

Projected Frequent Pattern Tree for D

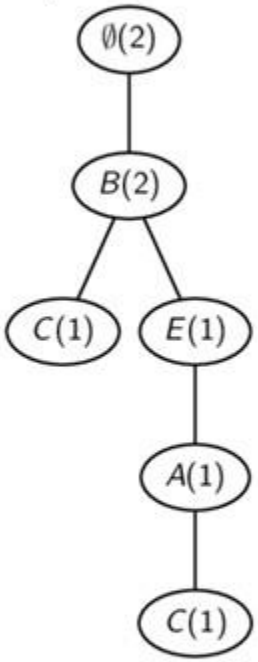
FP-Tree



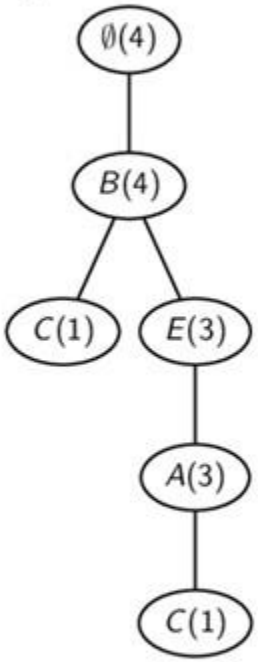
Add $BC, cnt = 1$



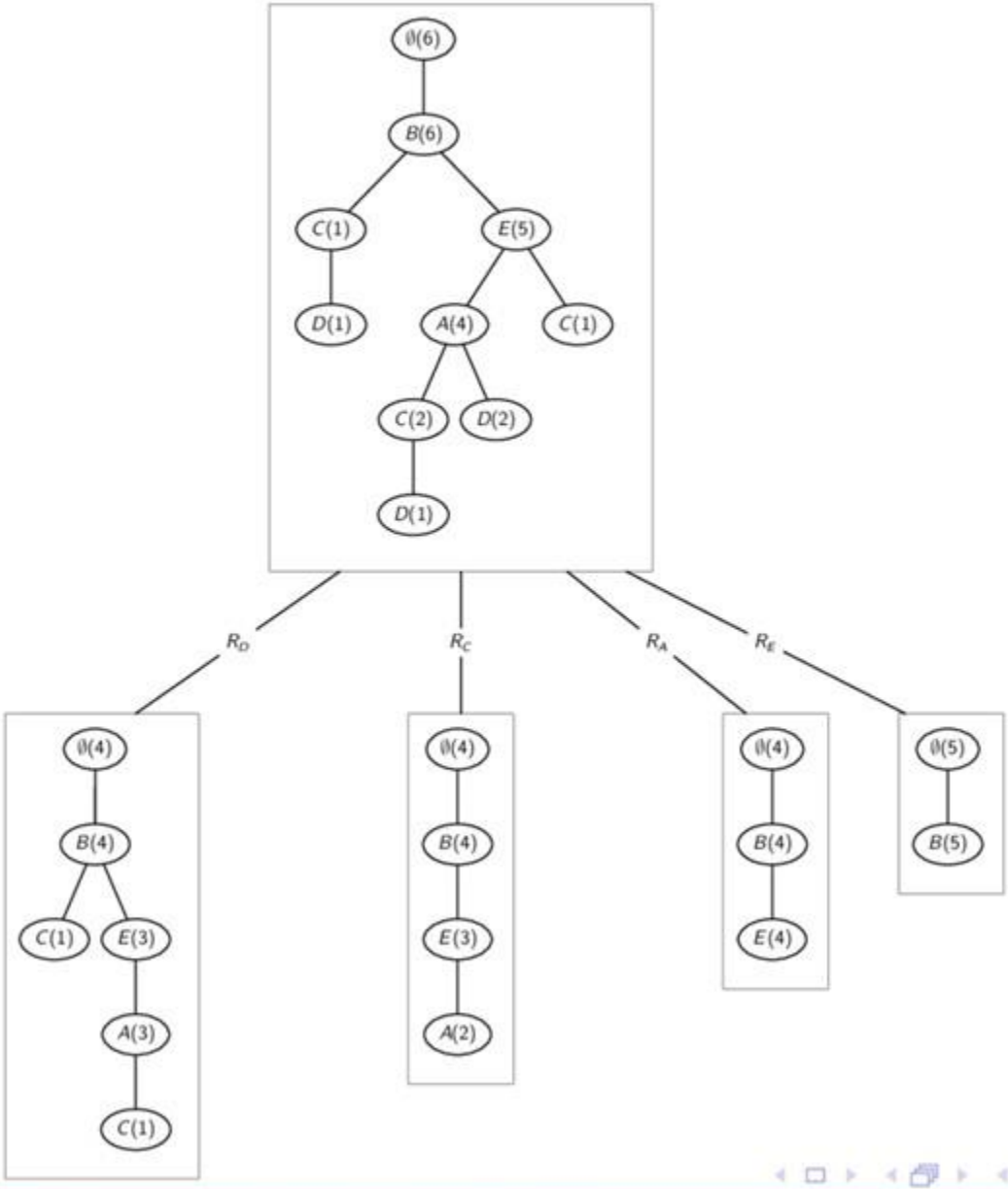
Add $BEAC, cnt = 1$



Add $BEA, cnt = 2$



FPGrowth Algorithm: Frequent Pattern Tree Projection



Association Rules

An *association rule* is an expression

$$X \xrightarrow{s,c} Y$$

where X and Y are itemsets and they are disjoint, that is, $X, Y \subseteq \mathcal{I}$, and $X \cap Y = \emptyset$. Let the itemset $X \cup Y$ be denoted as XY.

The *support* of the rule is the number of transactions in which both X and Y co-occur as subsets:

$$s = sup(X \longrightarrow Y) = |\boldsymbol{t}(XY)| = sup(XY)$$

The *relative support* of the rule is defined as the fraction of transactions where X and Y co-occur, and it provides an estimate of the joint probability of X and Y:

$$rsup(X \longrightarrow Y) = \frac{sup(XY)}{|\boldsymbol{D}|} = P(X \wedge Y)$$

The *confidence* of a rule is the conditional probability that a transaction contains Y given that it contains X:

$$c = conf(X \longrightarrow Y) = P(Y|X) = \frac{P(X \wedge Y)}{P(X)} = \frac{sup(XY)}{sup(X)}$$

Generating Association Rules

Given a frequent itemset $Z \in \mathcal{F}$, we look at all proper subsets $X \subset Z$ to compute rules of the form

$$X \xrightarrow{s,c} Y, \text{ where } Y = Z \setminus X$$

where $Z \setminus X = Z - X$.

The rule must be frequent because

$$s = sup(XY) = sup(Z) \geq minsup$$

We compute the confidence as follows:

$$c = \frac{sup(X \cup Y)}{sup(X)} = \frac{sup(Z)}{sup(X)}$$

If $c \geq minconf$, then the rule is a strong rule. On the other hand, if $conf(X \longrightarrow Y) < c$, then $conf(W \longrightarrow Z \setminus W) < c$ for all subsets $W \subset X$, as $sup(W) \geq sup(X)$. We can thus avoid checking subsets of X.

Association Rule Mining Algorithm

AssociationRules ($\mathcal{F}$, *minconf*):

1 **foreach** $Z \in \mathcal{F}$, *such that* $|Z| \geq 2$ **do**
2 $\mathcal{A} \leftarrow \{X \mid X \subset Z, X \neq \emptyset\}$
3 **while** $\mathcal{A} \neq \emptyset$ **do**
4 $X \leftarrow$ maximal element in $\mathcal{A}$
5 $\mathcal{A} \leftarrow \mathcal{A} \setminus X$ `// remove` X `from` $\mathcal{A}$
6 $c \leftarrow sup(Z)/sup(X)$
7 **if** $c \geq minconf$ **then**
8 print $X \longrightarrow Y$, $sup(Z)$, c
9 **else**
10 $\mathcal{A} \leftarrow \mathcal{A} \setminus \{W \mid W \subset X\}$
 `//` `remove all subsets of` X `from` $\mathcal{A}$

Data mining and Machine learning

Part 9. Summarizing Itemset

An Example Database

Transaction database

Tid	Itemset
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

Frequent itemsets ($minsup = 3$)

sup	Itemsets
6	B
5	E, BE
4	$A, C, D, AB, AE, BC, BD, ABE$
3	$AD, CE, DE, ABD, ADE, BCE, BDE, ABDE$

Maximal Frequent Itemsets

Given a binary database $\boldsymbol{D} \subseteq \mathcal{T} \times \mathcal{I}$, over the tids $\mathcal{T}$ and items $\mathcal{I}$, let $\mathcal{F}$ denote the set of all frequent itemsets, that is,

$$\mathcal{F} = \{X \mid X \subseteq \mathcal{I} \text{ and } sup(X) \geq minsup\}$$

A frequent itemset $X \in \mathcal{F}$ is called *maximal* if it has no frequent supersets. Let $\mathcal{M}$ be the set of all maximal frequent itemsets, given as

$$\mathcal{M} = \{X \mid X \in \mathcal{F} \text{ and } \nexists Y \supset X, \text{ such that } Y \in \mathcal{F}\}$$

The set $\mathcal{M}$ is a condensed representation of the set of all frequent itemset $\mathcal{F}$, because we can determine whether any itemset X is frequent or not using $\mathcal{M}$. If there exists a maximal itemset Z such that $X \subseteq Z$, then X must be frequent; otherwise X cannot be frequent.

Closed Frequent Itemsets

Given $T \subseteq \mathcal{T}$, and $X \subseteq \mathcal{I}$, define

$$\begin{aligned} \boldsymbol{t}(X) &= \{t \in \mathcal{T} \mid t \text{ contains } X\} \\ \boldsymbol{i}(T) &= \{x \in \mathcal{I} \mid \forall t \in T,\ t \text{ contains } x\} \\ \boldsymbol{c}(X) &= \boldsymbol{i} \circ \boldsymbol{t}(X) = \boldsymbol{i}(\boldsymbol{t}(X)) \end{aligned}$$

The function $\boldsymbol{c}$ is a *closure operator* and an itemset X is called *closed* if $\boldsymbol{c}(X) = X$. It follows that $\boldsymbol{t}(\boldsymbol{c}(X)) = \boldsymbol{t}(X)$. The set of all closed frequent itemsets is thus defined as

$$\mathcal{C} = \{X \mid X \in \mathcal{F} \text{ and } \nexists Y \supset X \text{ such that } sup(X) = sup(Y)\}$$

X is closed if all supersets of X have strictly less support, that is, $sup(X) > sup(Y)$, for all $Y \supset X$.

The set of all closed frequent itemsets $\mathcal{C}$ is a condensed representation, as we can determine whether an itemset X is frequent, as well as the exact support of X using $\mathcal{C}$ alone.

Minimal Generators

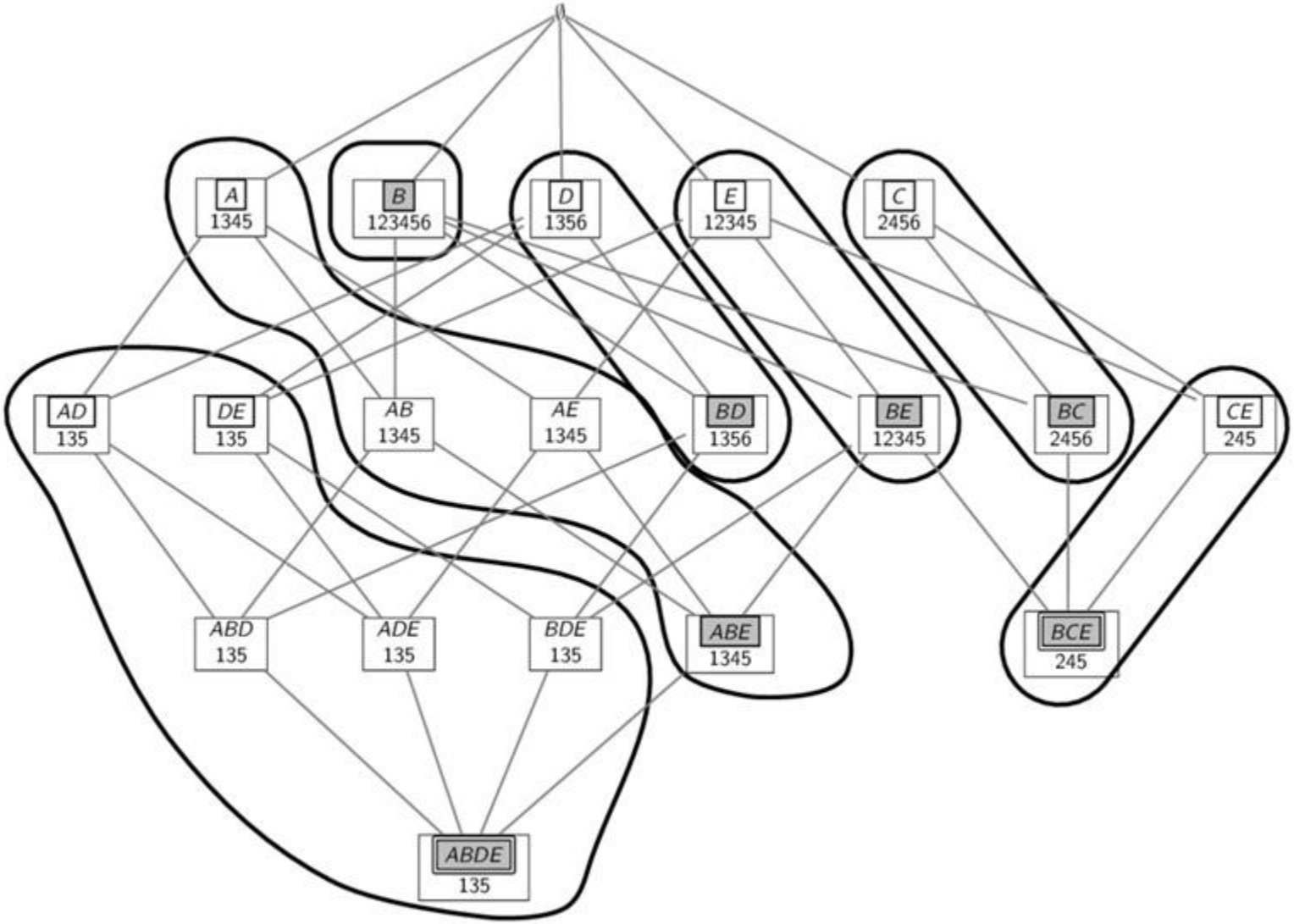
A frequent itemset X is a *minimal generator* if it has no subsets with the same support:

$$\mathcal{G} = \{X \mid X \in \mathcal{F} \text{ and } \nexists Y \subset X, \text{ such that } sup(X) = sup(Y)\}$$

In other words, all subsets of X have strictly higher support, that is, $sup(X) < sup(Y)$, for all $Y \subset X$.

Given an equivalence class of itemsets that have the same tidset, a closed itemset is the unique maximum element of the class, whereas the minimal generators are the minimal elements of the class.

Frequent Itemsets: Closed, Minimal Generators and Maximal



Itemsets boxed and shaded are closed, double boxed are maximal, and those boxed are minimal generators

Mining Maximal Frequent Itemsets: GenMax Algorithm

Mining maximal itemsets requires additional steps beyond simply determining the frequent itemsets. Assuming that the set of maximal frequent itemsets is initially empty, that is, $\mathcal{M} = \emptyset$, each time we generate a new frequent itemset X, we have to perform the following maximality checks

- **Subset Check:** $\nexists Y \in \mathcal{M}$, such that $X \subset Y$. If such a Y exists, then clearly X is not maximal. Otherwise, we add X to $\mathcal{M}$, as a potentially maximal itemset.
- **Superset Check:** $\nexists Y \in \mathcal{M}$, such that $Y \subset X$. If such a Y exists, then Y cannot be maximal, and we have to remove it from $\mathcal{M}$.

GenMax Algorithm: Maximal Itemsets

GenMax is based on dEclat, i.e., it uses diffset intersections for support computation. The initial call takes as input the set of frequent items along with their tidsets, $\langle i, \boldsymbol{t}(i) \rangle$, and the initially empty set of maximal itemsets, $\mathcal{M}$. Given a set of itemset–tidset pairs, called IT-pairs, of the form $\langle X, \boldsymbol{t}(X) \rangle$, the recursive GenMax method works as follows.

If the union of all the itemsets, $Y = \bigcup X_i$, is already subsumed by (or contained in) some maximal pattern $Z \in \mathcal{M}$, then no maximal itemset can be generated from the current branch, and it is pruned. Otherwise, we intersect each IT-pair $\langle X_i, \boldsymbol{t}(X_i) \rangle$ with all the other IT-pairs $\langle X_j, \boldsymbol{t}(X_j) \rangle$, with $j > i$, to generate new candidates X_{ij}, which are added to the IT-pair set P_i.

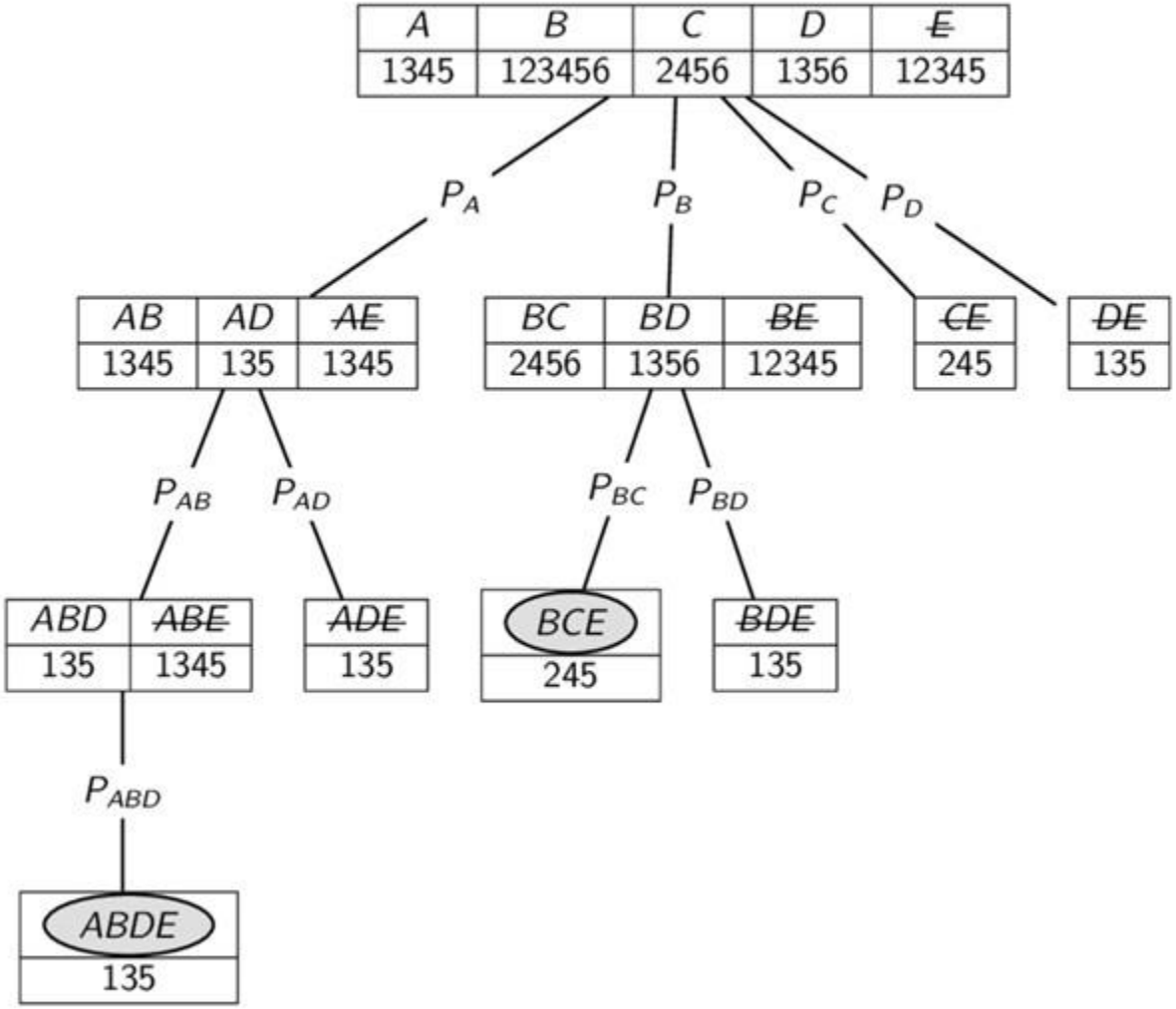
If P_i is not empty, a recursive call to GenMax is made to find other potentially frequent extensions of X_i. On the other hand, if P_i is empty, it means that X_i cannot be extended, and it is potentially maximal. In this case, we add X_i to the set $\mathcal{M}$, provided that X_i is not contained in any previously added maximal set $Z \in \mathcal{M}$.

GenMax Algorithm

// Initial Call: $\mathcal{M} \leftarrow \emptyset$,
$P \leftarrow \{\langle i, \boldsymbol{t}(i)\rangle \mid i \in \mathcal{I}, sup(i) \geq minsup\}$
GenMax (P, *minsup*, $\mathcal{M}$):

```
1  Y ← ⋃ X_i
2  if ∃Z ∈ M, such that Y ⊆ Z then
3      return // prune entire branch
4  foreach ⟨X_i, t(X_i)⟩ ∈ P do
5      P_i ← ∅
6      foreach ⟨X_j, t(X_j)⟩ ∈ P, with j > i do
7          X_ij ← X_i ∪ X_j
8          t(X_ij) = t(X_i) ∩ t(X_j)
9          if sup(X_ij) ≥ minsup then P_i ← P_i ∪ {⟨X_ij, t(X_ij)⟩}
10     if P_i ≠ ∅ then GenMax (P_i, minsup, M)
11     else if ∄Z ∈ M, X_i ⊆ Z then
12         M = M ∪ X_i    // add X_i to maximal set
```

Mining Maximal Frequent Itemsets



Mining Closed Frequent Itemsets: Charm Algorithm

Mining closed frequent itemsets requires that we perform closure checks, that is, whether $X = \boldsymbol{c}(X)$. Direct closure checking can be very expensive.
Given a collection of IT-pairs $\{\langle X_i, \boldsymbol{t}(X_i)\rangle\}$, Charm uses the following three properties:

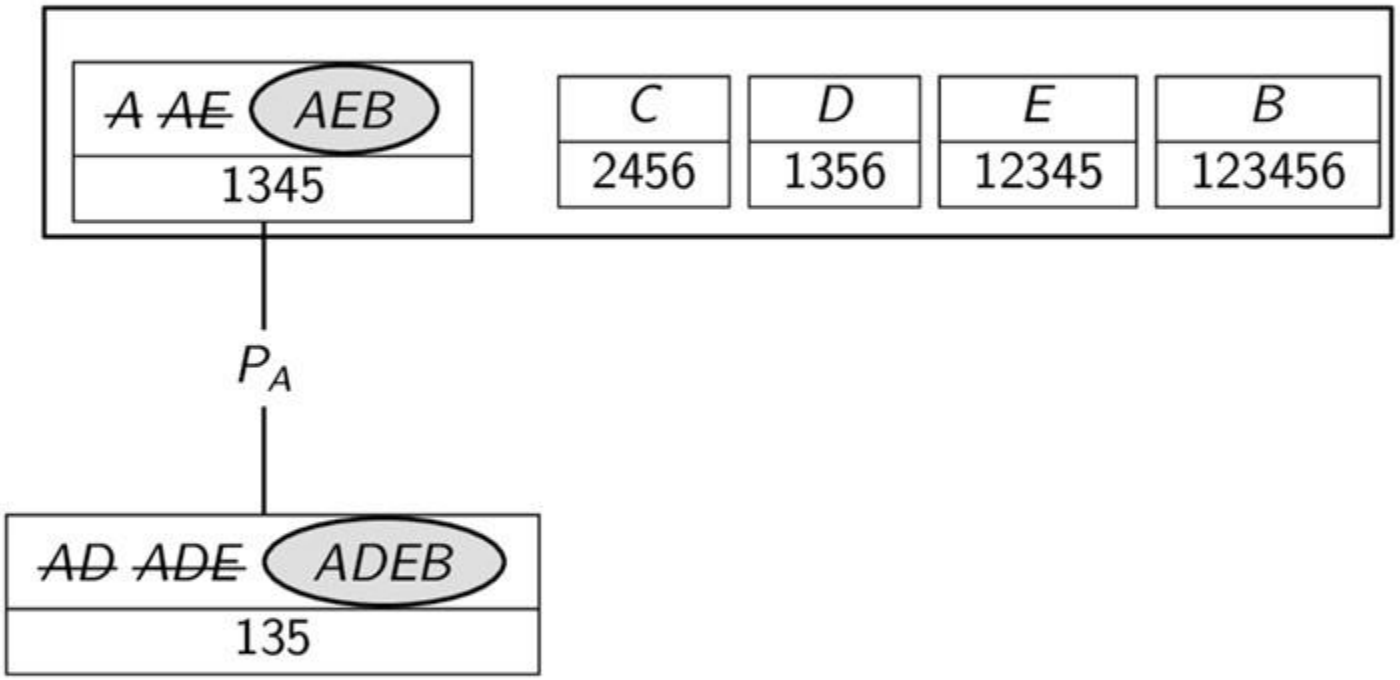
- If $\boldsymbol{t}(X_i) = \boldsymbol{t}(X_j)$, then $\boldsymbol{c}(X_i) = \boldsymbol{c}(X_j) = \boldsymbol{c}(X_i \cup X_j)$, which implies that we can replace every occurrence of X_i with $X_i \cup X_j$ and prune the branch under X_j because its closure is identical to the closure of $X_i \cup X_j$.
- If $\boldsymbol{t}(X_i) \subset \boldsymbol{t}(X_j)$, then $\boldsymbol{c}(X_i) \neq \boldsymbol{c}(X_j)$ but $\boldsymbol{c}(X_i) = \boldsymbol{c}(X_i \cup X_j)$, which means that we can replace every occurrence of X_i with $X_i \cup X_j$, but we cannot prune X_j because it generates a different closure. Note that if $\boldsymbol{t}(X_i) \supset \boldsymbol{t}(X_j)$ then we simply interchange the role of X_i and X_j.
- If $\boldsymbol{t}(X_i) \neq \boldsymbol{t}(X_j)$, then $\boldsymbol{c}(X_i) \neq \boldsymbol{c}(X_j) \neq \boldsymbol{c}(X_i \cup X_j)$. In this case we cannot remove either X_i or X_j, as each of them generates a different closure.

Charm Algorithm: Closed Itemsets

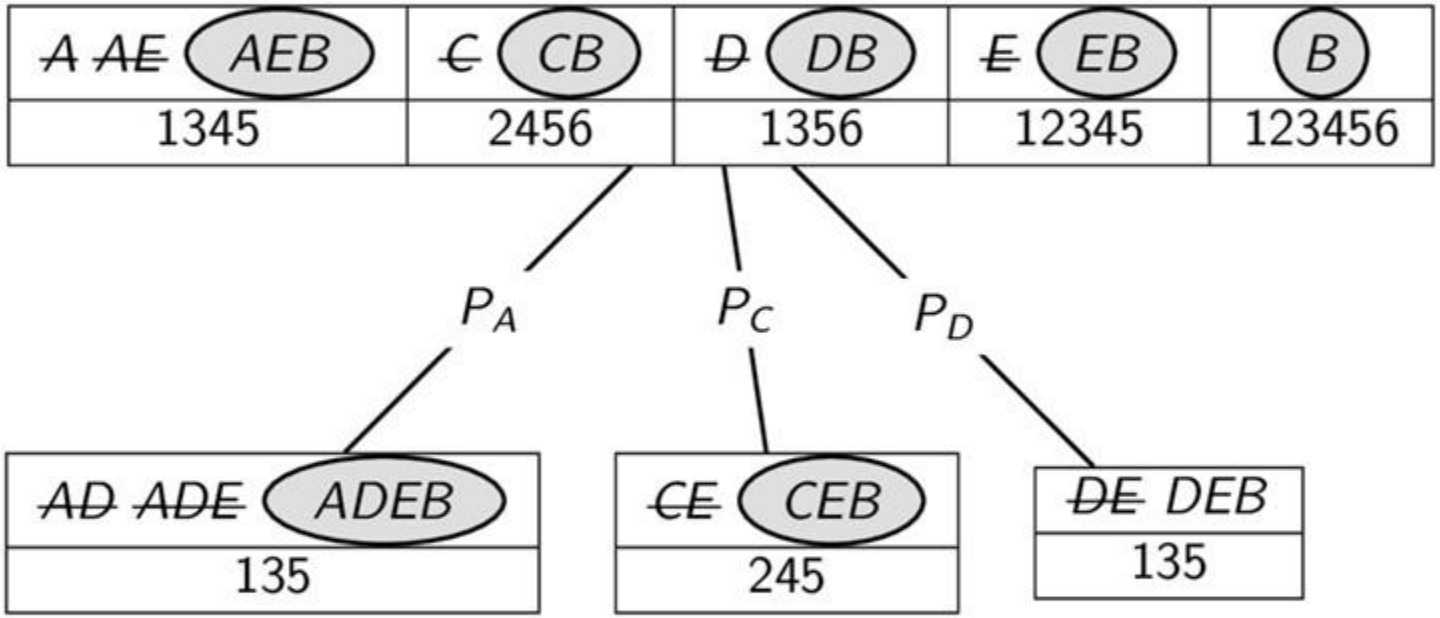
```
// Initial Call: C ← ∅, P ← {⟨i, t(i)⟩ : i ∈ I, sup(i) ≥ minsup}
Charm (P, minsup, C):
1  Sort P in increasing order of support (i.e., by increasing |t(X_i)|)
2  foreach ⟨X_i, t(X_i)⟩ ∈ P do
3      P_i ← ∅
4      foreach ⟨X_j, t(X_j)⟩ ∈ P, with j > i do
5          X_ij = X_i ∪ X_j
6          t(X_ij) = t(X_i) ∩ t(X_j)
7          if sup(X_ij) ≥ minsup then
8              if t(X_i) = t(X_j) then // Property 1
9                  Replace X_i with X_ij in P and P_i
10                 Remove ⟨X_j, t(X_j)⟩ from P
11             else
12                 if t(X_i) ⊂ t(X_j) then // Property 2
13                     Replace X_i with X_ij in P and P_i
14                 else // Property 3
15                     P_i ← P_i ∪ {⟨X_ij, t(X_ij)⟩}
16     if P_i ≠ ∅ then Charm (P_i, minsup, C)
17     if ∄Z ∈ C, such that X_i ⊆ Z and t(X_i) = t(Z) then
18         C = C ∪ X_i    // Add X_i to closed set
```

Mining Frequent Closed Itemsets: Charm

Process A



Mining Frequent Closed Itemsets: Charm



Nonderivable Itemsets

An itemset is called *nonderivable* if its support cannot be deduced from the supports of its subsets. The set of all frequent nonderivable itemsets is a summary or condensed representation of the set of all frequent itemsets. Further, it is lossless with respect to support, that is, the exact support of all other frequent itemsets can be deduced from it.

Generalized Itemsets: Let X be a k-itemset, that is, $X = \{x_1, x_2, \ldots, x_k\}$. The k tidsets $\boldsymbol{t}(x_i)$ for each item $x_i \in X$ induce a partitioning of the set of all tids into 2^k regions, where each partition contains the tids for some subset of items $Y \subseteq X$, but for none of the remaining items $Z = X \setminus Y$.

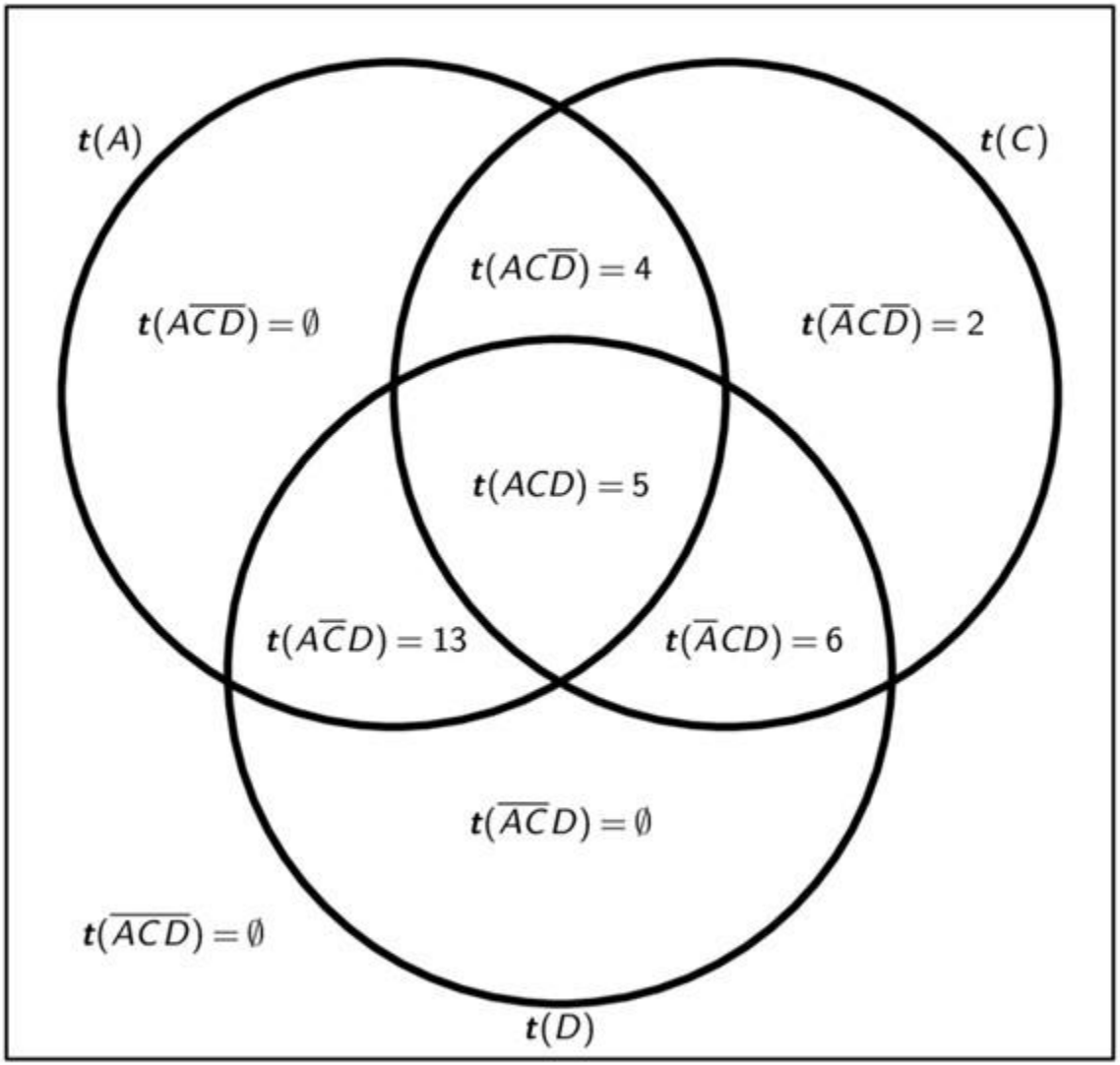
Each partition is therefore the tidset of a *generalized itemset* $Y\overline{Z}$, where Y consists of regular items and Z consists of negated items.

Define the support of a generalized itemset $Y\overline{Z}$ as the number of transactions that contain all items in Y but no item in Z:

$$sup(Y\overline{Z}) = \left|\{t \in \mathcal{T} \mid Y \subseteq \boldsymbol{i}(t) \text{ and } Z \cap \boldsymbol{i}(t) = \emptyset\}\right|$$

Tidset Partitioning Induced by $\boldsymbol{t}(A)$,$\boldsymbol{t}(C)$, and $\boldsymbol{t}(D)$

Tid	Itemset
1	*ABDE*
2	*BCE*
3	*ABDE*
4	*ABCE*
5	*ABCDE*
6	*BCD*



Inclusion–Exclusion Principle: Support Bounds

Let $Y\overline{Z}$ be a generalized itemset, and let $X = Y \cup Z = YZ$. The inclusion–exclusion principle allows one to directly compute the support of $Y\overline{Z}$ as a combination of the supports for all itemsets W, such that $Y \subseteq W \subseteq X$:

$$sup(Y\overline{Z}) = \sum_{Y \subseteq W \subseteq X} -1^{|W \setminus Y|} \cdot sup(W)$$

Inclusion–Exclusion for Support

Consider the generalized itemset $\overline{A}C\overline{D} = C\overline{AD}$, where $Y = C$, $Z = AD$ and $X = YZ = ACD$. In the Venn diagram, we start with all the tids in $\boldsymbol{t}(C)$, and remove the tids contained in $\boldsymbol{t}(AC)$ and $\boldsymbol{t}(CD)$. However, we realize that in terms of support this removes $sup(ACD)$ twice, so we need to add it back. In other words, the support of $C\overline{AD}$ is given as

$$\begin{aligned} sup(C\overline{AD}) &= sup(C) - sup(AC) - sup(CD) + sup(ACD) \\ &= 4 - 2 - 2 + 1 = 1 \end{aligned}$$

But, this is precisely what the inclusion–exclusion formula gives:

$$\begin{aligned} sup(C\overline{AD}) = \; & (-1)^0\, sup(C) + && W = C, |W \setminus Y| = 0 \\ & (-1)^1\, sup(AC) + && W = AC, |W \setminus Y| = 1 \\ & (-1)^1\, sup(CD) + && W = CD, |W \setminus Y| = 1 \\ & (-1)^2\, sup(ACD) && W = ACD, |W \setminus Y| = 2 \\ = \; & sup(C) - sup(AC) - \\ & sup(CD) + sup(ACD) \end{aligned}$$

Support Bounds

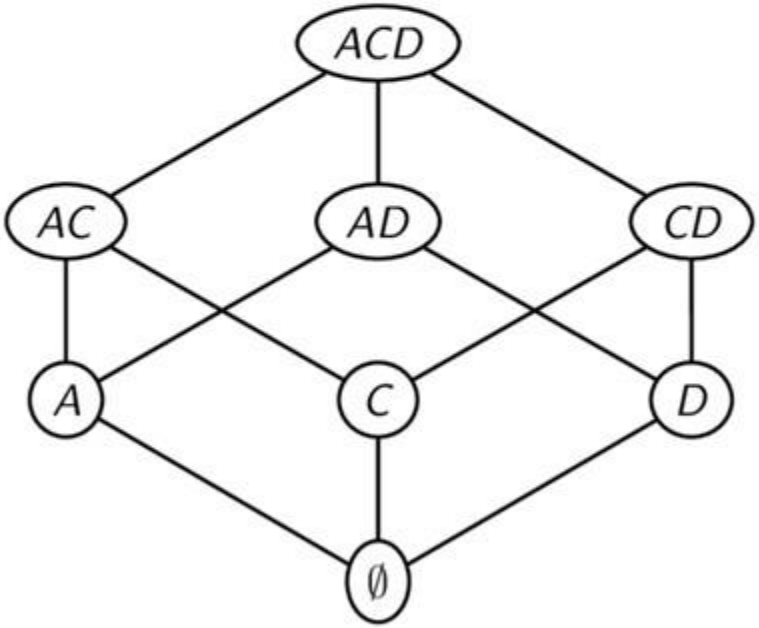
Because the support of any (generalized) itemset must be non-negative, we can derive a bound on the support of X from each of the 2^k generalized itemsets by setting $sup(Y\overline{Z}) \geq 0$.
Thus, from the 2^k possible subsets $Y \subseteq X$, we derive 2^{k-1} lower bounds and 2^{k-1} upper bounds for $sup(X)$, obtained after setting $sup(Y\overline{Z}) \geq 0$:

Upper Bounds ($|X \setminus Y|$ is odd): $$sup(X) \leq \sum_{Y \subseteq W \subset X} -1^{(|X \setminus W|+1)} sup(W) \quad (1)$$

Lower Bounds ($|X \setminus Y|$ is even): $$sup(X) \geq \sum_{Y \subseteq W \subset X} -1^{(|X \setminus W|+1)} sup(W) \quad (2)$$

Support Bounds for Subsets

subset lattice



sign	inequality	level
1	$\leq$	1
-1	$\geq$	2
1	$\leq$	3

Support Bounds

Example

For example, if $Y = C$, then the inclusion-exclusion principle gives us

$$sup(C\overline{AD}) = sup(C) - sup(AC) - sup(CD) + sup(ACD)$$

Setting $sup(C\overline{AD}) \geq 0$, we obtain

$$sup(ACD) \geq -sup(C) + sup(AC) + sup(CD)$$

From each of the partitions, we get one bound, and out of the eight possible regions, exactly four give upper bounds and the other four give lower bounds for the support of ACD:

$$\begin{array}{lll}
sup(ACD) & \geq 0 & \text{when } Y = ACD \\
& \leq sup(AC) & \text{when } Y = AC \\
& \leq sup(AD) & \text{when } Y = AD \\
& \leq sup(CD) & \text{when } Y = CD \\
& \geq sup(AC) + sup(AD) - sup(A) & \text{when } Y = A \\
& \geq sup(AC) + sup(CD) - sup(C) & \text{when } Y = C \\
& \geq sup(AD) + sup(CD) - sup(D) & \text{when } Y = D \\
& \leq sup(AC) + sup(AD) + sup(CD) - & \\
& \quad sup(A) - sup(C) - sup(D) + sup(\emptyset) & \text{when } Y = \emptyset
\end{array}$$

Nonderivable Itemsets

Given an itemset X, and $Y \subseteq X$, let $IE(Y)$ denote the summation

$$IE(Y) = \sum_{Y \subseteq W \subset X} -1^{(|X \setminus Y|+1)} \cdot sup(W)$$

Then, the sets of all upper and lower bounds for $sup(X)$ are given as

$$UB(X) = \Big\{ IE(Y) \big|\ Y \subseteq X, |X \setminus Y| \text{ is odd} \Big\}$$
$$LB(X) = \Big\{ IE(Y) \big|\ Y \subseteq X, |X \setminus Y| \text{ is even} \Big\}$$

An itemset X is called *nonderivable* if $\max\{LB(X)\} \neq \min\{UB(X)\}$, and we know only the range of possible values, that is,

$$sup(X) \in \Big[\max\{LB(X)\}, \min\{UB(X)\}\Big]$$

X is derivable if $sup(X) = \max\{LB(X)\} = \min\{UB(X)\}$ Thus, the set of all frequent nonderivable itemsets is given as

$$\mathcal{N} = \{X \in \mathcal{F} \mid \max\{LB(X)\} \neq \min\{UB(X)\}\}$$

where $\mathcal{F}$ is the set of all frequent itemsets.

Nonderivable Itemsets: Example

Consider the support bound formulas for $sup(ACD)$. The lower bounds are

$$\begin{aligned} sup(ACD) &\geq 0 \\ &\geq sup(AC) + sup(AD) - sup(A) = 2+3-4 = 1 \\ &\geq sup(AC) + sup(CD) - sup(C) = 2+2-4 = 0 \\ &\geq sup(AD) + sup(CD) - sup(D) = 3+2-4 = 0 \end{aligned}$$

and the upper bounds are

$$\begin{aligned} sup(ACD) &\leq sup(AC) = 2 \\ &\leq sup(AD) = 3 \\ &\leq sup(CD) = 2 \\ &\leq sup(AC) + sup(AD) + sup(CD) - sup(A) - sup(C) - \\ &\quad sup(D) + sup(\emptyset) = 2+3+2-4-4-4+6 = 1 \end{aligned}$$

Thus, we have

$$LB(ACD) = \{0,1\} \qquad \max\{LB(ACD)\} = 1$$
$$UB(ACD) = \{1,2,3\} \qquad \min\{UB(ACD)\} = 1$$

Because $\max\{LB(ACD)\} = \min\{UB(ACD)\}$ we conclude that ACD is derivable.

Data mining and Machine learning

Part 10. Sequence Mining

Sequence Mining: Terminology

Let Σ be the *alphabet*, a set of symbols. A *sequence* or a *string* is defined as an ordered list of symbols, and is written as $\boldsymbol{s} = s_1 s_2 \ldots s_k$, where $s_i \in \Sigma$ is a symbol at position i, also denoted as $\boldsymbol{s}[i]$. $|\boldsymbol{s}| = k$ denotes the *length* of the sequence.

The notation $\boldsymbol{s}[i:j] = s_i s_{i+1} \cdots s_{j-1} s_j$ denotes the *substring* or sequence of consecutive symbols in positions i through j, where $j > i$.

Define the *prefix* of a sequence $\boldsymbol{s}$ as any substring of the form $\boldsymbol{s}[1:i] = s_1 s_2 \ldots s_i$, with $0 \leq i \leq n$.

Define the *suffix* of $\boldsymbol{s}$ as any substring of the form $\boldsymbol{s}[i:n] = s_i s_{i+1} \ldots s_n$, with $1 \leq i \leq n+1$.

$\boldsymbol{s}[1:0]$ is the empty prefix, and $\boldsymbol{s}[n+1:n]$ is the empty suffix. Let $\Sigma^\star$ be the set of all possible sequences that can be constructed using the symbols in Σ, including the empty sequence $\emptyset$ (which has length zero).

Sequence Mining: Terminology

Let $\boldsymbol{s} = s_1 s_2 \ldots s_n$ and $\boldsymbol{r} = r_1 r_2 \ldots r_m$ be two sequences over Σ. We say that $\boldsymbol{r}$ is a *subsequence* of $\boldsymbol{s}$ denoted $\boldsymbol{r} \subseteq \boldsymbol{s}$, if there exists a one-to-one mapping $\phi : [1, m] \rightarrow [1, n]$, such that $\boldsymbol{r}[i] = \boldsymbol{s}[\phi(i)]$ and for any two positions i, j in $\boldsymbol{r}$, $i < j \implies \phi(i) < \phi(j)$. In If $\boldsymbol{r} \subseteq \boldsymbol{s}$, we also say that $\boldsymbol{s}$ *contains* $\boldsymbol{r}$.

The sequence $\boldsymbol{r}$ is called a *consecutive subsequence* or substring of $\boldsymbol{s}$ provided $r_1 r_2 \ldots r_m = s_j s_{j+1} \ldots s_{j+m-1}$, i.e., $\boldsymbol{r}[1 : m] = \boldsymbol{s}[j : j + m - 1]$, with $1 \leq j \leq n - m + 1$.

Let $\Sigma = \{A, C, G, T\}$, and let $\boldsymbol{s} = ACTGAACG$.
Then $\boldsymbol{r}_1 = CGAAG$ is a subsequence of $\boldsymbol{s}$, and $\boldsymbol{r}_2 = CTGA$ is a substring of $\boldsymbol{s}$. The sequence $\boldsymbol{r}_3 = ACT$ is a prefix of $\boldsymbol{s}$, and so is $\boldsymbol{r}_4 = ACTGA$, whereas $\boldsymbol{r}_5 = GAACG$ is one of the suffixes of $\boldsymbol{s}$.

Frequent Sequences

Given a database $\boldsymbol{D} = \{\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_N\}$ of N sequences, and given some sequence $\boldsymbol{r}$, the *support* of $\boldsymbol{r}$ in the database $\boldsymbol{D}$ is defined as the total number of sequences in $\boldsymbol{D}$ that contain $\boldsymbol{r}$

$$sup(\boldsymbol{r}) = \left| \left\{ \boldsymbol{s}_i \in \boldsymbol{D} | \boldsymbol{r} \subseteq \boldsymbol{s}_i \right\} \right|$$

The *relative support* of $\boldsymbol{r}$ is the fraction of sequences that contain $\boldsymbol{r}$

$$rsup(\boldsymbol{r}) = sup(\boldsymbol{r})/N$$

Given a user-specified *minsup* threshold, we say that a sequence $\boldsymbol{r}$ is *frequent* in database $\boldsymbol{D}$ if $sup(\boldsymbol{r}) \geq minsup$. A frequent sequence is *maximal* if it is not a subsequence of any other frequent sequence, and a frequent sequence is *closed* if it is not a subsequence of any other frequent sequence with the same support.

Mining Frequent Sequences

For sequence mining the order of the symbols matters, and thus we have to consider all possible *permutations* of the symbols as the possible frequent candidates. Contrast this with itemset mining, where we had only to consider *combinations* of the items.

The sequence search space can be organized in a prefix search tree. The root of the tree, at level 0, contains the empty sequence, with each symbol $x \in \Sigma$ as one of its children. As such, a node labeled with the sequence $\boldsymbol{s} = s_1 s_2 \ldots s_k$ at level k has children of the form $\boldsymbol{s}' = s_1 s_2 \ldots s_k s_{k+1}$ at level $k+1$. In other words, $\boldsymbol{s}$ is a prefix of each child $\boldsymbol{s}'$, which is also called an *extension* of $\boldsymbol{s}$.

Example Sequence Database

Id	Sequence
s_1	$CAGAAGT$
s_2	$TGACAG$
s_3	$GAAGT$

Using $minsup = 3$, the set of frequent subsequences is given as:

$$\mathcal{F}^{(1)} = A(3), G(3), T(3)$$
$$\mathcal{F}^{(2)} = AA(3), AG(3), GA(3), GG(3)$$
$$\mathcal{F}^{(3)} = AAG(3), GAA(3), GAG(3)$$
$$\mathcal{F}^{(4)} = GAAG(3)$$

Level-wise Sequence Mining: GSP Algorithm

The GSP algorithm searches the sequence prefix tree using a level-wise or breadth-first search. Given the set of frequent sequences at level k, we generate all possible sequence extensions or *candidates* at level $k+1$. We next compute the support of each candidate and prune those that are not frequent. The search stops when no more frequent extensions are possible.

The prefix search tree at level k is denoted $\mathcal{C}^{(k)}$. Initially $\mathcal{C}^{(1)}$ comprises all the symbols in Σ. Given the current set of candidate k-sequences $\mathcal{C}^{(k)}$, the method first computes their support.

For each database sequence $\boldsymbol{s}_i \in \boldsymbol{D}$, we check whether a candidate sequence $\boldsymbol{r} \in \mathcal{C}^{(k)}$ is a subsequence of $\boldsymbol{s}_i$. If so, we increment the support of $\boldsymbol{r}$. Once the frequent sequences at level k have been found, we generate the candidates for level $k+1$.

For the extension, each leaf $\boldsymbol{r}_a$ is extended with the last symbol of any other leaf $\boldsymbol{r}_b$ that shares the same prefix (i.e., has the same parent), to obtain the new candidate $(k+1)$-sequence $\boldsymbol{r}_{ab} = \boldsymbol{r}_a + \boldsymbol{r}_b[k]$. If the new candidate $\boldsymbol{r}_{ab}$ contains any infrequent k-sequence, we prune it.

Algorithm GSP

GSP ($\boldsymbol{D}$, Σ, *minsup*)**:**

1 $\mathcal{F} \leftarrow \emptyset$
2 $\mathcal{C}^{(1)} \leftarrow \{\emptyset\}$ // Initial prefix tree with single symbols
3 **foreach** $s \in \Sigma$ **do** Add s as child of $\emptyset$ in $\mathcal{C}^{(1)}$ with $sup(s) \leftarrow 0$
4 $k \leftarrow 1$ // k denotes the level
5 **while** $\mathcal{C}^{(k)} \neq \emptyset$ **do**
6 ComputeSupport $(\mathcal{C}^{(k)}, \boldsymbol{D})$
7 **foreach** *leaf* $\boldsymbol{s} \in \mathcal{C}^{(k)}$ **do**
8 **if** $sup(\boldsymbol{r}) \geq minsup$ **then** $\mathcal{F} \leftarrow \mathcal{F} \cup \{(\boldsymbol{r}, sup(\boldsymbol{r}))\}$
9 **else** remove $\boldsymbol{s}$ from $\mathcal{C}^{(k)}$
10 $\mathcal{C}^{(k+1)} \leftarrow$ ExtendPrefixTree $(\mathcal{C}^{(k)})$
11 $k \leftarrow k + 1$
12 **return** $\mathcal{F}^{(k)}$

Algorithm ComputeSupport

ComputeSupport $(\mathcal{C}^{(k)}, \boldsymbol{D})$**:**

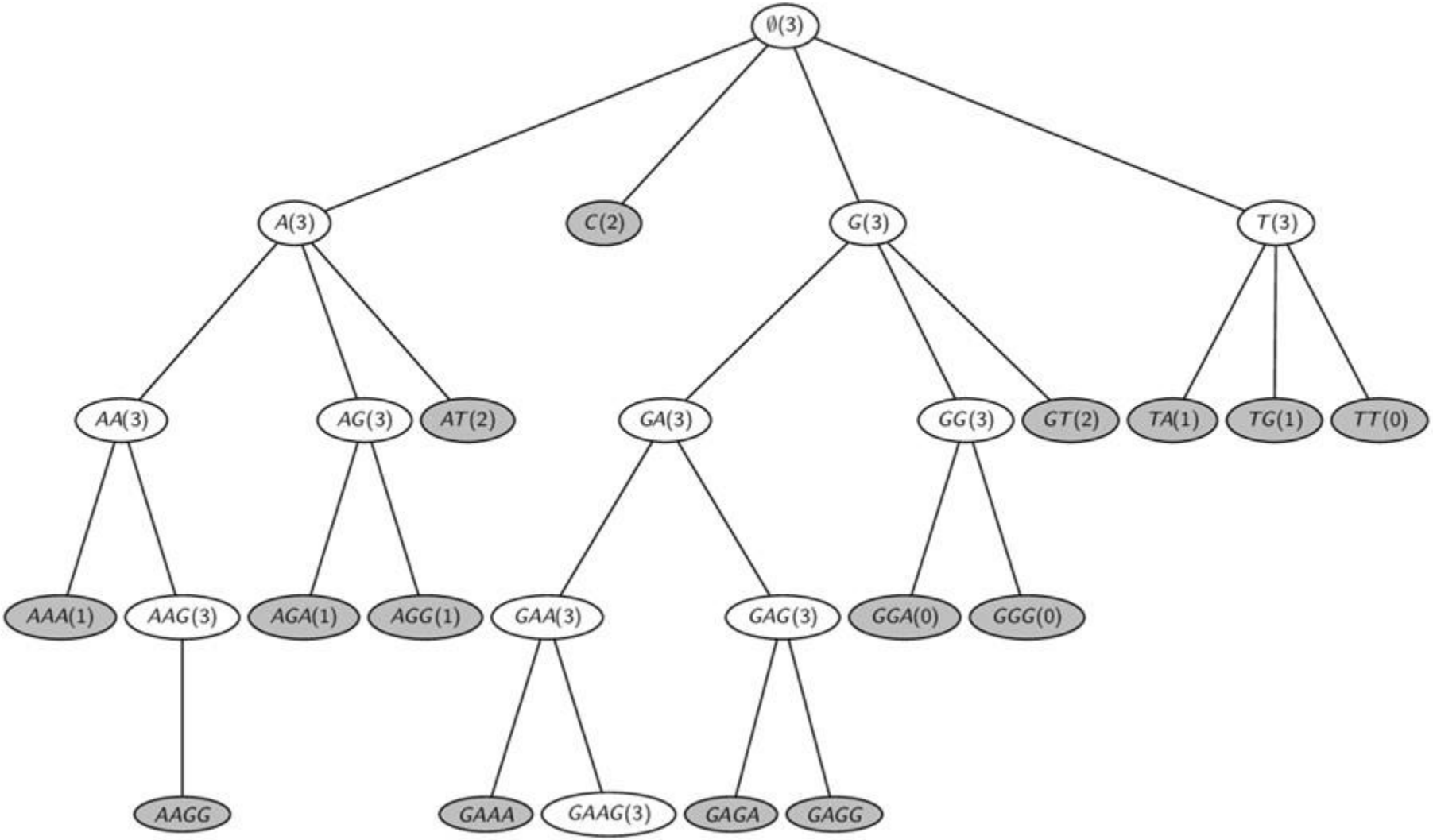
foreach $\boldsymbol{s}_i \in \boldsymbol{D}$ **do**
  foreach $\boldsymbol{r} \in \mathcal{C}^{(k)}$ **do**
    if $\boldsymbol{r} \subseteq \boldsymbol{s}_i$ **then** $sup(\boldsymbol{r}) \leftarrow sup(\boldsymbol{r}) + 1$

ExtendPrefixTree $(\mathcal{C}^{(k)})$**:**

foreach *leaf* $\boldsymbol{r}_a \in \mathcal{C}^{(k)}$ **do**
  foreach *leaf* $\boldsymbol{r}_b \in \text{Children}(\text{Parent}(\boldsymbol{r}_a))$ **do**
    $\boldsymbol{r}_{ab} \leftarrow \boldsymbol{r}_a + \boldsymbol{r}_b[k]$ `// extend` $\boldsymbol{r}_a$ `with last item of` $\boldsymbol{r}_b$
    `// prune if there are any infrequent subsequences`
    if $\boldsymbol{r}_c \in \mathcal{C}^{(k)}$, ***for all*** $\boldsymbol{r}_c \subset \boldsymbol{r}_{ab}$, *such that* $|\boldsymbol{r}_c| = |\boldsymbol{r}_{ab}| - 1$ **then**
      Add $\boldsymbol{r}_{ab}$ as child of $\boldsymbol{r}_a$ with $sup(\boldsymbol{r}_{ab}) \leftarrow 0$
  if *no extensions from* $\boldsymbol{r}_a$ **then**
    remove $\boldsymbol{r}_a$, and all ancestors of $\boldsymbol{r}_a$ with no extensions, from $\mathcal{C}^{(k)}$
return $\mathcal{C}^{(k)}$

Sequence Search Space

shaded ovals are infrequent sequences



Vertical Sequence Mining: Spade

The Spade algorithm uses a vertical database representation for sequence mining. For each symbol $s \in \Sigma$, we keep a set of tuples of the form $\langle i, pos(s) \rangle$, where $pos(s)$ is the set of positions in the database sequence $\boldsymbol{s}_i \in \boldsymbol{D}$ where symbol s appears.

Let $\mathcal{L}(s)$ denote the set of such sequence-position tuples for symbol s, which we refer to as the *poslist*. The set of poslists for each symbol $s \in \Sigma$ thus constitutes a vertical representation of the input database.

Given k-sequence $\boldsymbol{r}$, its poslist $\mathcal{L}(\boldsymbol{r})$ maintains the list of positions for the occurrences of the last symbol $\boldsymbol{r}[k]$ in each database sequence $\boldsymbol{s}_i$, provided $\boldsymbol{r} \subseteq \boldsymbol{s}_i$. The support of sequence $\boldsymbol{r}$ is simply the number of distinct sequences in which $\boldsymbol{r}$ occurs, that is, $sup(\boldsymbol{r}) = |\mathcal{L}(\boldsymbol{r})|$.

Spade Algorithm

Support computation in Spade is done via *sequential join* operations.

Given the poslists for any two k-sequences $\boldsymbol{r}_a$ and $\boldsymbol{r}_b$ that share the same $(k-1)$ length prefix, a sequential join on the poslists is used to compute the support for the new $(k+1)$ length candidate sequence $\boldsymbol{r}_{ab} = \boldsymbol{r}_a + \boldsymbol{r}_b[k]$.

Given a tuple $\langle i, pos(\boldsymbol{r}_b[k]) \rangle \in \mathcal{L}(\boldsymbol{r}_b)$, we first check if there exists a tuple $\langle i, pos(\boldsymbol{r}_a[k]) \rangle \in \mathcal{L}(\boldsymbol{r}_a)$, that is, both sequences must occur in the same database sequence $\boldsymbol{s}_i$.

Next, for each position $p \in pos(\boldsymbol{r}_b[k])$, we check whether there exists a position $q \in pos(\boldsymbol{r}_a[k])$ such that $q < p$. If yes, this means that the symbol $\boldsymbol{r}_b[k]$ occurs after the last position of $\boldsymbol{r}_a$ and thus we retain p as a valid occurrence of $\boldsymbol{r}_{ab}$. The poslist $\mathcal{L}(\boldsymbol{r}_{ab})$ comprises all such valid occurrences.

We keep track of positions only for the last symbol in the candidate sequence since we extend sequences from a common prefix, and so there is no need to keep track of all the occurrences of the symbols in the prefix.

We denote the sequential join as $\mathcal{L}(\boldsymbol{r}_{ab}) = \mathcal{L}(\boldsymbol{r}_a) \cap \mathcal{L}(\boldsymbol{r}_b)$.

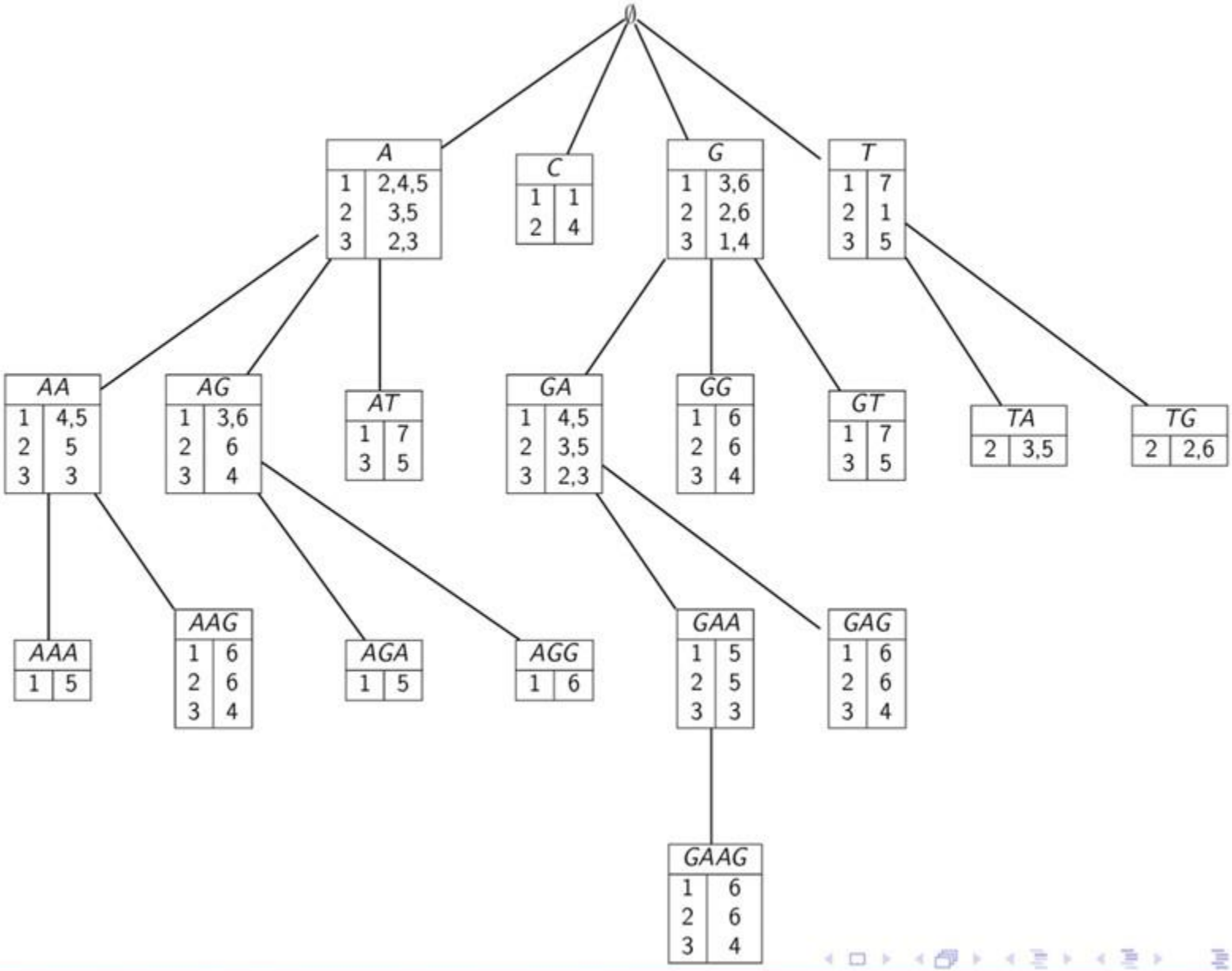
Spade Algorithm

// Initial Call: $\mathcal{F} \leftarrow \emptyset,\ k \leftarrow 0,$
$P \leftarrow \{\langle s, \mathcal{L}(s)\rangle \mid s \in \Sigma, sup(s) \geq minsup\}$

Spade (P, *minsup*, $\mathcal{F}$, k):

1 **foreach** $\boldsymbol{r}_a \in P$ **do**
2 $\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{(\boldsymbol{r}_a, sup(\boldsymbol{r}_a))\}$
3 $\quad P_a \leftarrow \emptyset$
4 $\quad$ **foreach** $\boldsymbol{r}_b \in P$ **do**
5 $\quad\quad \boldsymbol{r}_{ab} = \boldsymbol{r}_a + \boldsymbol{r}_b[k]$
6 $\quad\quad \mathcal{L}(\boldsymbol{r}_{ab}) = \mathcal{L}(\boldsymbol{r}_a) \cap \mathcal{L}(\boldsymbol{r}_b)$
7 $\quad\quad$ **if** $sup(\boldsymbol{r}_{ab}) \geq minsup$ **then**
8 $\quad\quad\quad P_a \leftarrow P_a \cup \{\langle \boldsymbol{r}_{ab}, \mathcal{L}(\boldsymbol{r}_{ab})\rangle\}$
9 $\quad$ **if** $P_a \neq \emptyset$ **then** Spade (P_a, *minsup*, $\mathcal{F}$, $k+1$)

Sequence Mining via Spade



Projection-Based Sequence Mining: PrefixSpan

Let $\boldsymbol{D}$ denote a database, and let $s \in \Sigma$ be any symbol. The *projected database* with respect to s, denoted $\boldsymbol{D}_s$, is obtained by finding the first occurrence of s in $\boldsymbol{s}_i$, say at position p. Next, we retain in $\boldsymbol{D}_s$ only the suffix of $\boldsymbol{s}_i$ starting at position $p+1$. Further, any infrequent symbols are removed from the suffix. This is done for each sequence $\boldsymbol{s}_i \in \boldsymbol{D}$.

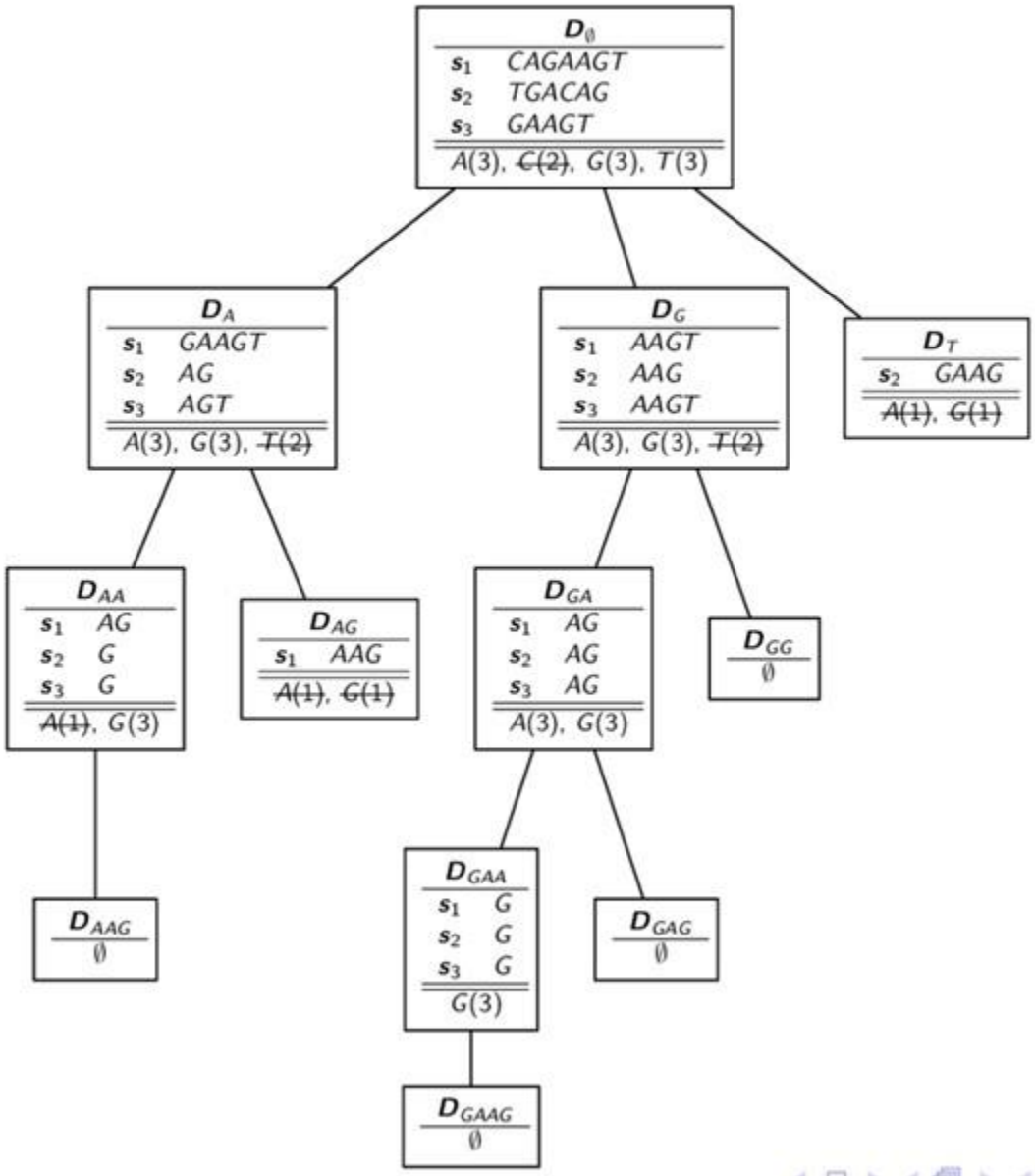
PrefixSpan computes the support for only the individual symbols in the projected database $\boldsymbol{D}_s$; it then performs recursive projections on the frequent symbols in a depth-first manner.

Given a frequent subsequence $\boldsymbol{r}$, let $\boldsymbol{D}_{\boldsymbol{r}}$ be the projected dataset for $\boldsymbol{r}$. Initially $\boldsymbol{r}$ is empty and $\boldsymbol{D}_{\boldsymbol{r}}$ is the entire input dataset $\boldsymbol{D}$. Given a database of (projected) sequences $\boldsymbol{D}_{\boldsymbol{r}}$, PrefixSpan first finds all the frequent symbols in the projected dataset. For each such symbol s, we extend $\boldsymbol{r}$ by appending s to obtain the new frequent subsequence $\boldsymbol{r}_s$. Next, we create the projected dataset $\boldsymbol{D}_s$ by projecting $\boldsymbol{D}_{\boldsymbol{r}}$ on symbol s. A recursive call to PrefixSpan is then made with $\boldsymbol{r}_s$ and $\boldsymbol{D}_s$.

PrefixSpan Algorithm

```
// Initial Call:  D_r ← D,  r ← ∅,  F ← ∅
PrefixSpan (D_r, r, minsup, F):
1 foreach s ∈ Σ such that sup(s, D_r) ≥ minsup do
2     r_s = r + s // extend r by symbol s
3     F ← F ∪ {(r_s, sup(s, D_r))}
4     D_s ← ∅ // create projected data for symbol s
5     foreach s_i ∈ D_r do
6         s'_i ← projection of s_i w.r.t symbol s
7         Remove any infrequent symbols from s'_i
8         Add s'_i to D_s if s'_i ≠ ∅
9     if D_s ≠ ∅ then PrefixSpan (D_s, r_s, minsup, F)
```

Projection-based Sequence Mining: PrefixSpan



Substring Mining via Suffix Trees

Let $\boldsymbol{s}$ be a sequence having length n, then there are at most $O(n^2)$ possible distinct substrings contained in $\boldsymbol{s}$. This is a much smaller search space compared to subsequences, and consequently we can design more efficient algorithms for solving the frequent substring mining task.

Naively, we can mine all the frequent substrings in worst case $O(Nn^2)$ time for a dataset $\boldsymbol{D} = \{\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_N\}$ with N sequences.

We will show that all sequences can be mined in $O(Nn)$ time via Suffix Trees.

Suffix Tree

Given a sequence $\boldsymbol{s}$, we append a terminal character $\$ \notin \Sigma$ so that $\boldsymbol{s} = s_1 s_2 \ldots s_n s_{n+1}$, where $s_{n+1} = \$$, and the jth suffix of $\boldsymbol{s}$ is given as $\boldsymbol{s}[j : n+1] = s_j s_{j+1} \ldots s_{n+1}$.

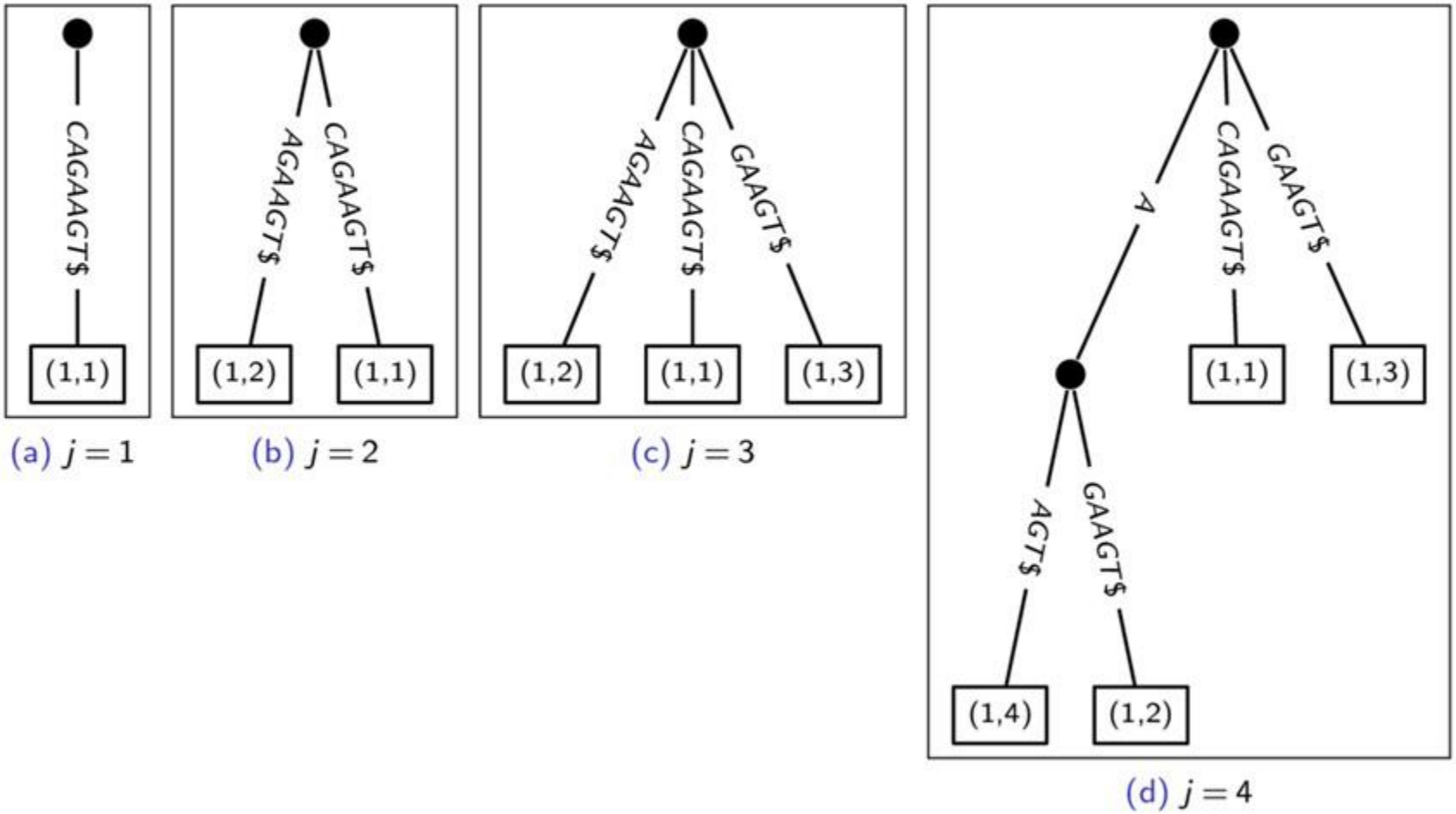
The *suffix tree* of the sequences in the database $\boldsymbol{D}$, denoted $\mathcal{T}$, stores all the suffixes for each $\boldsymbol{s}_i \in \boldsymbol{D}$ in a tree structure, where suffixes that share a common prefix lie on the same path from the root of the tree.

The substring obtained by concatenating all the symbols from the root node to a node v is called the *node label* of v, and is denoted as $L(v)$. The substring that appears on an edge (v_a, v_b) is called an *edge label*, and is denoted as $L(v_a, v_b)$.

A suffix tree has two kinds of nodes: internal and leaf nodes. An internal node in the suffix tree (except for the root) has at least two children, where each edge label to a child begins with a different symbol. Because the terminal character is unique, there are as many leaves in the suffix tree as there are unique suffixes over all the sequences. Each leaf node corresponds to a suffix shared by one or more sequences in $\boldsymbol{D}$.

Suffix Tree Construction for $s = CAGAAGT\$$

Insert each suffix j per step

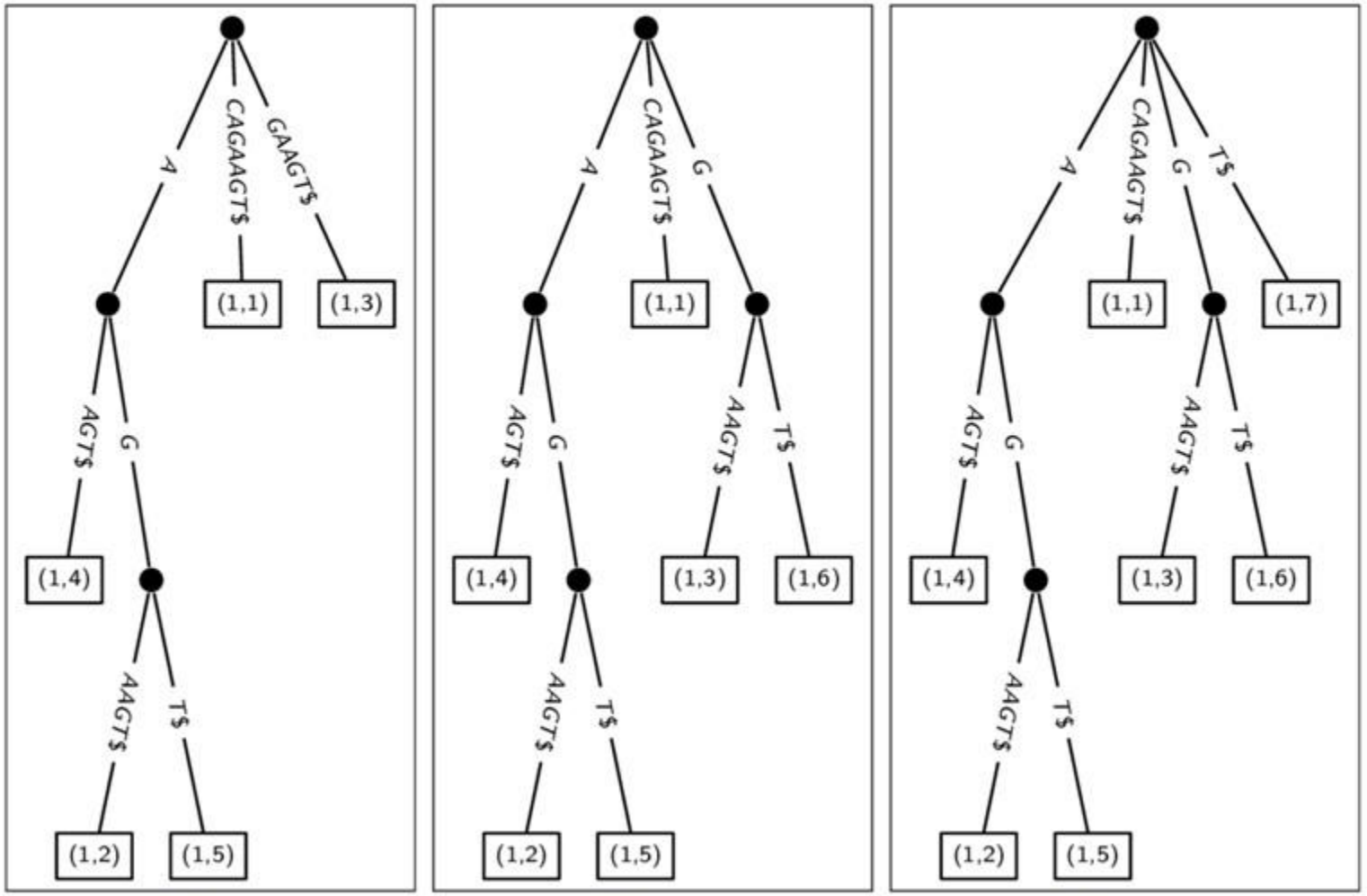


(a) $j = 1$ (b) $j = 2$ (c) $j = 3$

(d) $j = 4$

Suffix Tree Construction for $s = CAGAAGT\$$

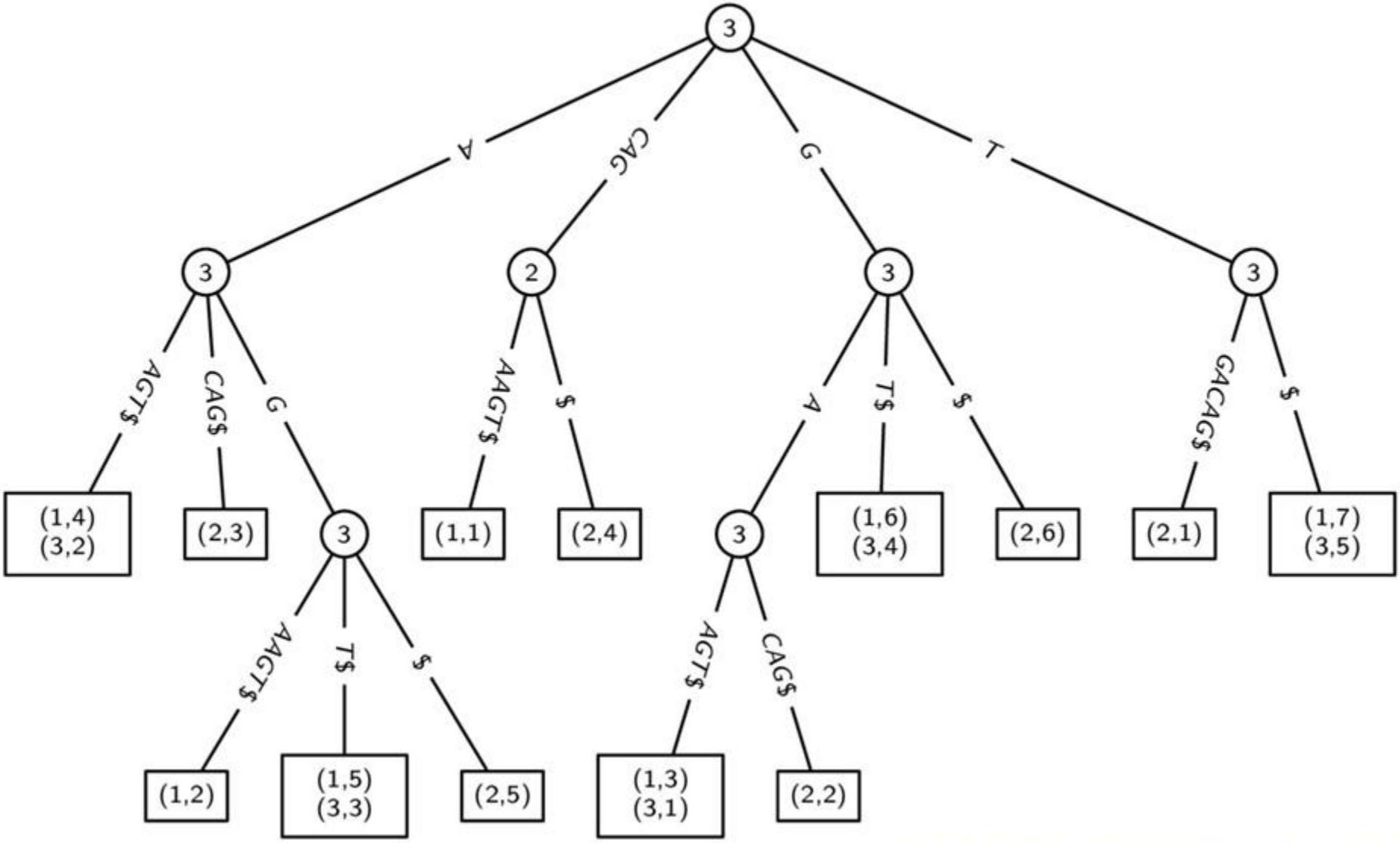
Insert each suffix j per step



(e) $j = 5$ (f) $j = 6$ (g) $j = 7$

Suffix Tree for Entire Database

$D = \{s_1 = CAGAAGT, s_2 = TGACAG, s_3 = GAAGT\}$



Frequent Substrings

Once the suffix tree is built, we can compute all the frequent substrings by checking how many different sequences appear in a leaf node or under an internal node.

The node labels for the nodes with support at least *minsup* yield the set of frequent substrings; all the prefixes of such node labels are also frequent.

The suffix tree can also support ad hoc queries for finding all the occurrences in the database for any query substring $\boldsymbol{q}$. For each symbol in $\boldsymbol{q}$, we follow the path from the root until all symbols in $\boldsymbol{q}$ have been seen, or until there is a mismatch at any position. If $\boldsymbol{q}$ is found, then the set of leaves under that path is the list of occurrences of the query $\boldsymbol{q}$. On the other hand, if there is mismatch that means the query does not occur in the database.

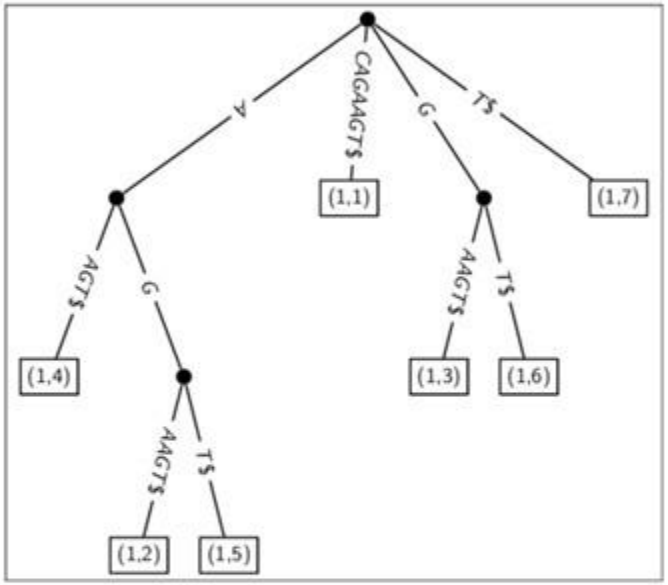
Because we have to match each character in $\boldsymbol{q}$, we immediately get $O(|\boldsymbol{q}|)$ as the time bound (assuming that $|\Sigma|$ is a constant), which is *independent* of the size of the database. Listing all the matches takes additional time, for a total time complexity of $O(|\boldsymbol{q}| + k)$, if there are k matches.

Ukkonen's Linear Time Suffix Tree Algorithm

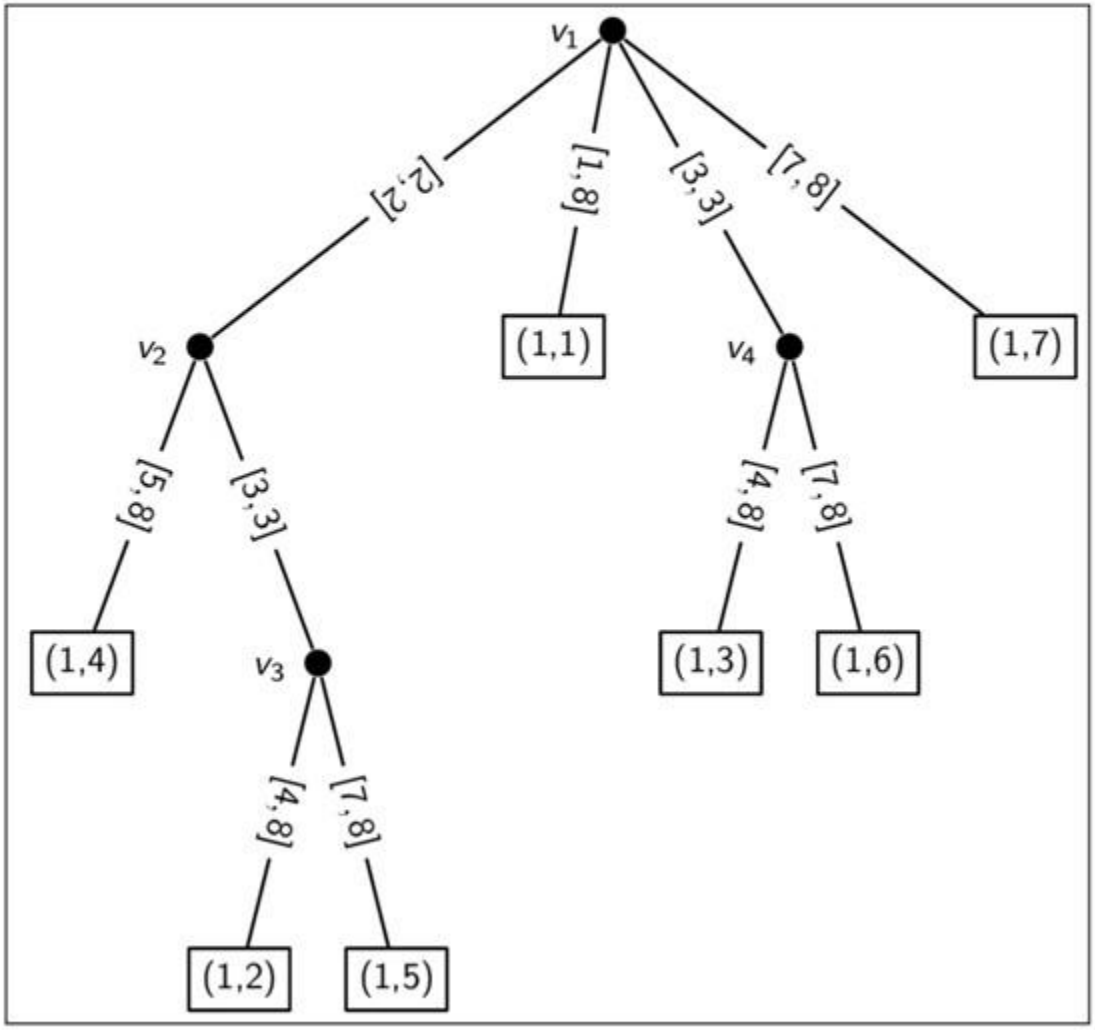
Achieving Linear Space: If an algorithm stores all the symbols on each edge label, then the space complexity is $O(n^2)$, and we cannot achieve linear time construction either.

The trick is to not explicitly store all the edge labels, but rather to use an *edge-compression* technique, where we store only the starting and ending positions of the edge label in the input string $\boldsymbol{s}$. That is, if an edge label is given as $\boldsymbol{s}[i:j]$, then we represent is as the interval $[i,j]$.

Suffix Tree using Edge-compression: $s = CAGAAGT\$$



(a) Full Tree



(b) Compressed Tree

Ukkonen Algorithm: Achieving Linear Time

Ukkonen's method is an *online* algorithm, that is, given a string $\boldsymbol{s} = s_1 s_2 \ldots s_n\$$ it constructs the full suffix tree in phases.

Phase i builds the tree up to the i-th symbol in $\boldsymbol{s}$. Let $\mathcal{T}_i$ denote the suffix tree up to the ith prefix $\boldsymbol{s}[1:i]$, with $1 \leq i \leq n$. Ukkonen's algorithm constructs $\mathcal{T}_i$ from $\mathcal{T}_{i-1}$, by making sure that all suffixes including the *current* character s_i are in the new intermediate tree $\mathcal{T}_i$.

In other words, in the ith phase, it inserts all the suffixes $\boldsymbol{s}[j:i]$ from $j = 1$ to $j = i$ into the tree $\mathcal{T}_i$. Each such insertion is called the jth *extension* of the ith *phase*.

Once we process the terminal character at position $n+1$ we obtain the final suffix tree $\mathcal{T}$ for $\boldsymbol{s}$.

However, this naive Ukkonen method has cubic time complexity because to obtain $\mathcal{T}_i$ from $\mathcal{T}_{i-1}$ takes $O(i^2)$ time, with the last phase requiring $O(n^2)$ time. With n phases, the total time is $O(n^3)$. We will show that this time can be reduced to $O(n)$.

Algorithm NaiveUkkonen

NaiveUkkonen (s):

```
1 n ← |s|
2 s[n+1] ← $ // append terminal character
3 T ← ∅ // add empty string as root
4 foreach i = 1,...,n+1 do // phase i - construct T_i
5 |   foreach j = 1,...,i do // extension j for phase i
  |   |   // Insert s[j:i] into the suffix tree
6 |   |   Find end of the path with label s[j:i-1] in T
7 |   |   Insert s_i at end of path;
8 return T
```

Ukkonen's Linear Time Algorithm: Implicit Suffixes

This optimization states that, in phase i, if the jth extension $\boldsymbol{s}[j:i]$ is found in the tree, then any subsequent extensions will also be found, and consequently there is no need to process further extensions in phase i.

Thus, the suffix tree $\mathcal{T}_i$ at the end of phase i has *implicit suffixes* corresponding to extensions $j+1$ through i.

It is important to note that all suffixes will become explicit the first time we encounter a new substring that does not already exist in the tree. This will surely happen in phase $n+1$ when we process the terminal character \$, as it cannot occur anywhere else in $\boldsymbol{s}$ (after all, $\$ \notin \Sigma$).

Ukkonen's Algorithm: Implicit Extensions

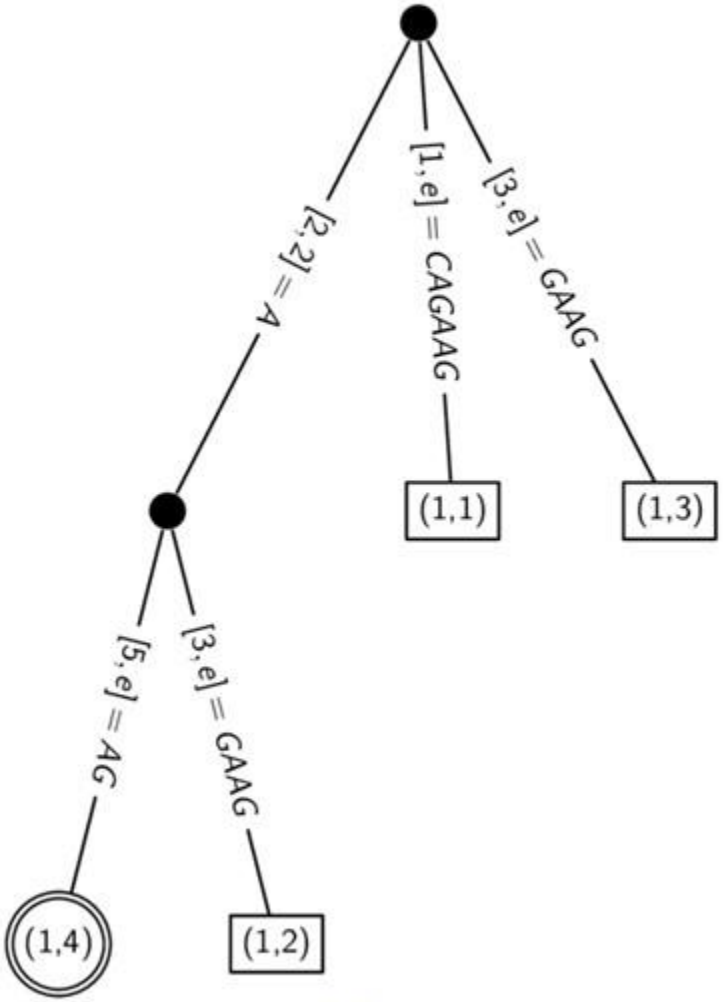
Let the current phase be i, and let $l \leq i-1$ be the last explicit suffix in the previous tree $\mathcal{T}_{i-1}$.

All explicit suffixes in $\mathcal{T}_{i-1}$ have edge labels of the form $[x, i-1]$ leading to the corresponding leaf nodes, where the starting position x is node specific, but the ending position must be $i-1$ because s_{i-1} was added to the end of these paths in phase $i-1$.

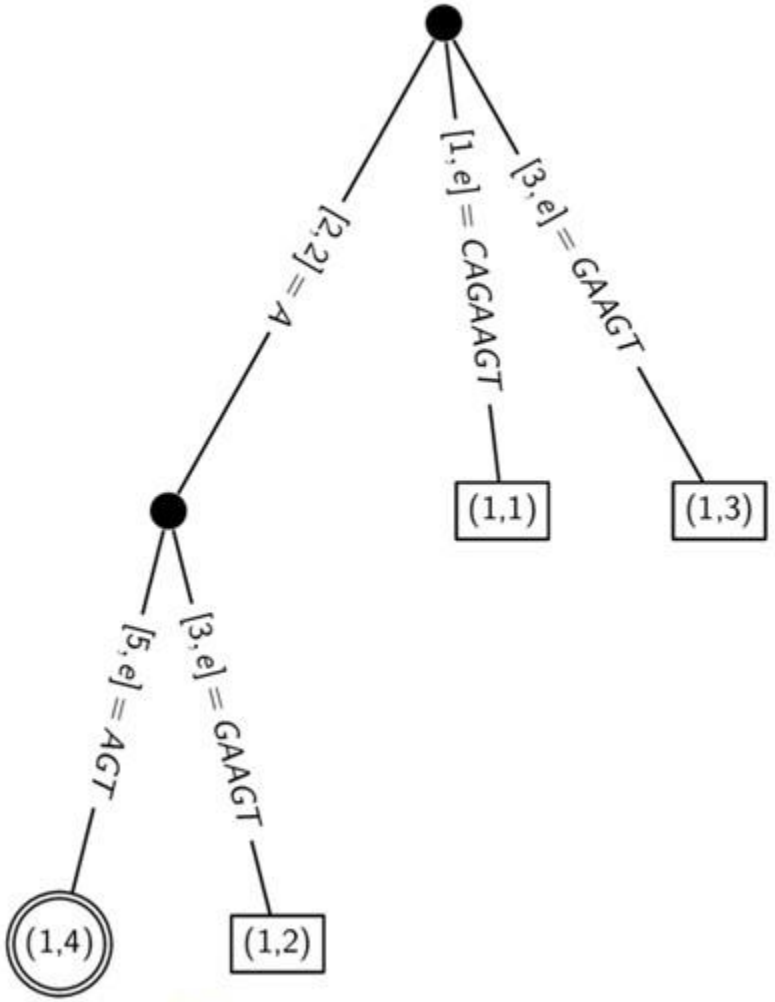
In the current phase i, we would have to extend these paths by adding s_i at the end. However, instead of explicitly incrementing all the ending positions, we can replace the ending position by a pointer e which keeps track of the current phase being processed.

If we replace $[x, i-1]$ with $[x, e]$, then in phase i, if we set $e = i$, then immediately all the l existing suffixes get *implicitly* extended to $[x, i]$. Thus, in one operation of incrementing e we have, in effect, taken care of extensions 1 through l for phase i.

Implicit Extensions: $s = CAGAAGT\$$, Phase $i = 7$



(a) $\mathcal{T}_6$



(b) $\mathcal{T}_7$, extensions $j = 1, \ldots, 4$

Ukkonen's Algorithm: Skip/count Trick

For the jth extension of phase i, we have to search for the substring $\boldsymbol{s}[j : i-1]$ so that we can add s_i at the end.

Note that this string must exist in $\mathcal{T}_{i-1}$ because we have already processed symbol s_{i-1} in the previous phase. Thus, instead of searching for each character in $\boldsymbol{s}[j : i-1]$ starting from the root, we first *count* the number of symbols on the edge beginning with character s_j; let this length be m. If m is longer than the length of the substring (i.e., if $m > i-j$), then the substring must end on this edge, so we simply jump to position $i-j$ and insert s_i.

On the other hand, if $m \leq i-j$, then we can *skip* directly to the child node, say v_c, and search for the remaining string $\boldsymbol{s}[j+m : i-1]$ from v_c using the same skip/count technique.

With this optimization, the cost of an extension becomes proportional to the number of nodes on the path, as opposed to the number of characters in $\boldsymbol{s}[j : i-1]$.

Ukkonen's Algorithm: Suffix Links

We can avoid searching for the substring $\boldsymbol{s}[j : i-1]$ from the root via the use of *suffix links*.

For each internal node v_a we maintain a link to the internal node v_b, where $L(v_b)$ is the immediate suffix of $L(v_a)$.

In extension $j-1$, let v_p denote the internal node under which we find $\boldsymbol{s}[j-1 : i]$, and let m be the length of the node label of v_p. To insert the jth extension $\boldsymbol{s}[j : i]$, we follow the suffix link from v_p to another node, say v_s, and search for the remaining substring $\boldsymbol{s}[j+m-1 : i-1]$ from v_s.

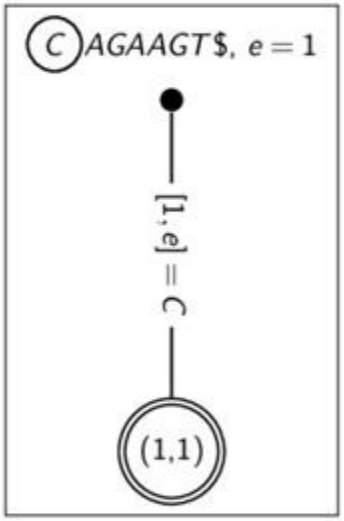
The use of suffix links allows us to jump internally within the tree for different extensions, as opposed to searching from the root each time.

Linear Time Ukkonen Algorithm

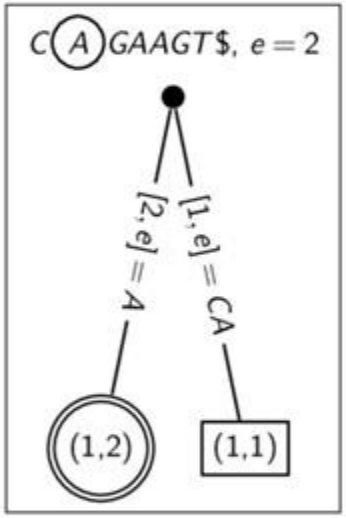
Ukkonen (s):

```
1  n ← |s|
2  s[n+1] ← $ // append terminal character
3  T ← ∅ // add empty string as root
4  l ← 0 // last explicit suffix
5  foreach i = 1,...,n+1 do // phase i - construct T_i
6  |  e ← i // implicit extensions
7  |  foreach j = l+1,...,i do // extension j for phase i
   |  |  // Insert s[j:i] into the suffix tree
8  |  |  Find end of s[j:i-1] in T via skip/count and suffix links
9  |  |  if s_i ∈ T then // implicit suffixes
10 |  |  |  break
11 |  |  else
12 |  |  |  Insert s_i at end of path
13 |  |  |  Set last explicit suffix l if needed
14 return T
```

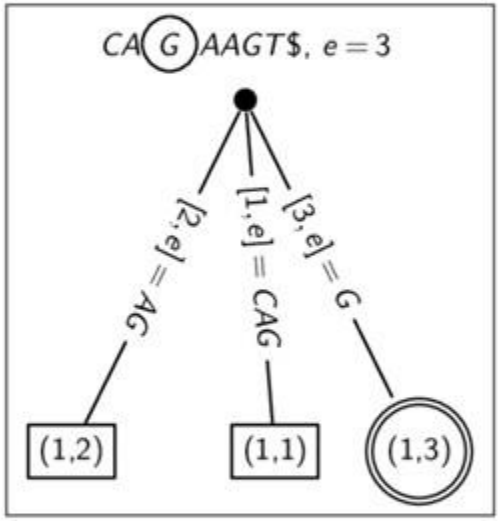
Ukkonen's Suffix Tree Construction: $s = CAGAAGT\$$



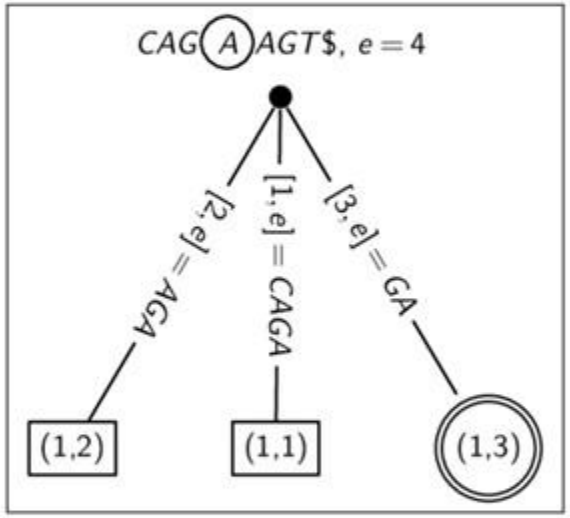
(a) $\mathcal{T}_1$



(b) $\mathcal{T}_2$

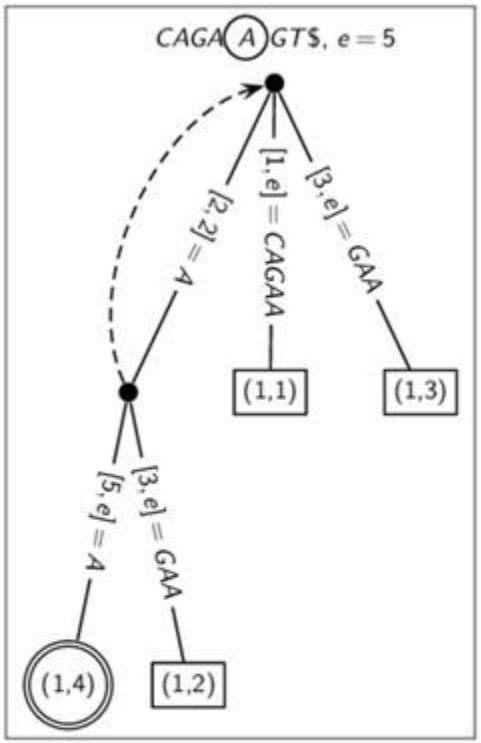


(c) $\mathcal{T}_3$

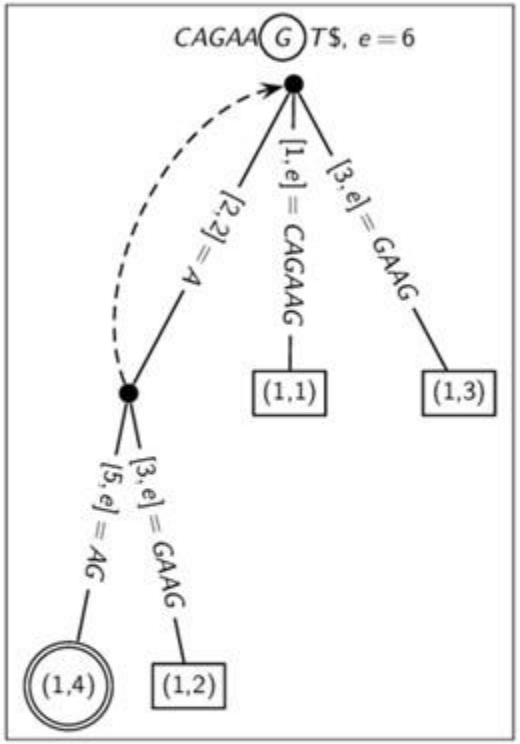


(d) $\mathcal{T}_4$

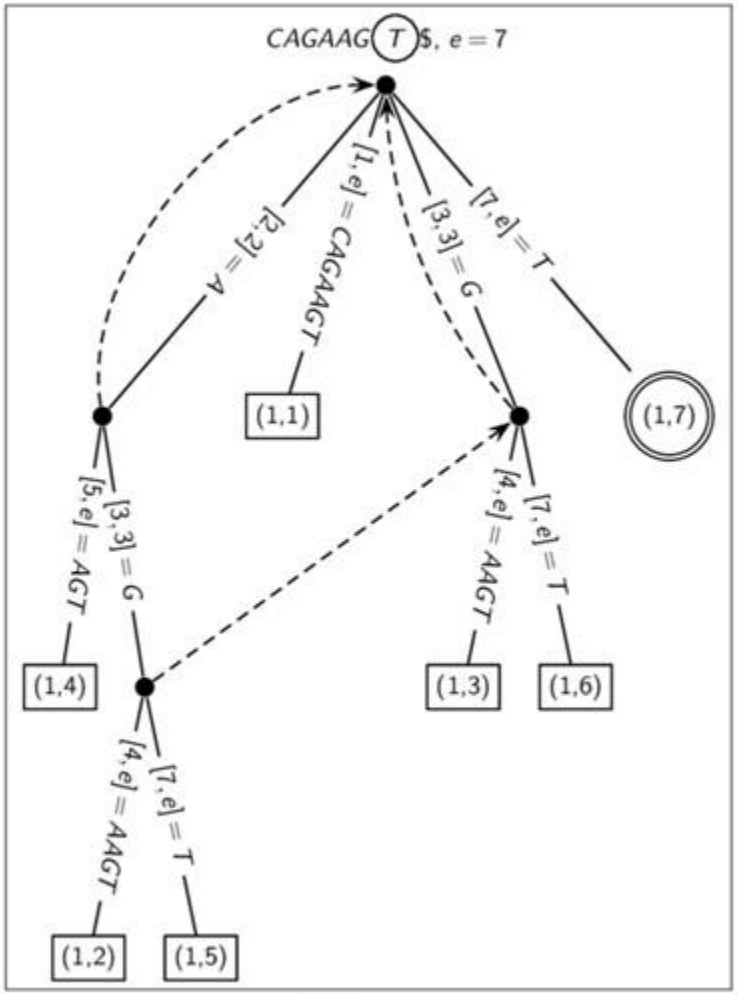
Ukkonen's Suffix Tree Construction: $s = CAGAAGT\$$



(e) $\mathcal{T}_5$

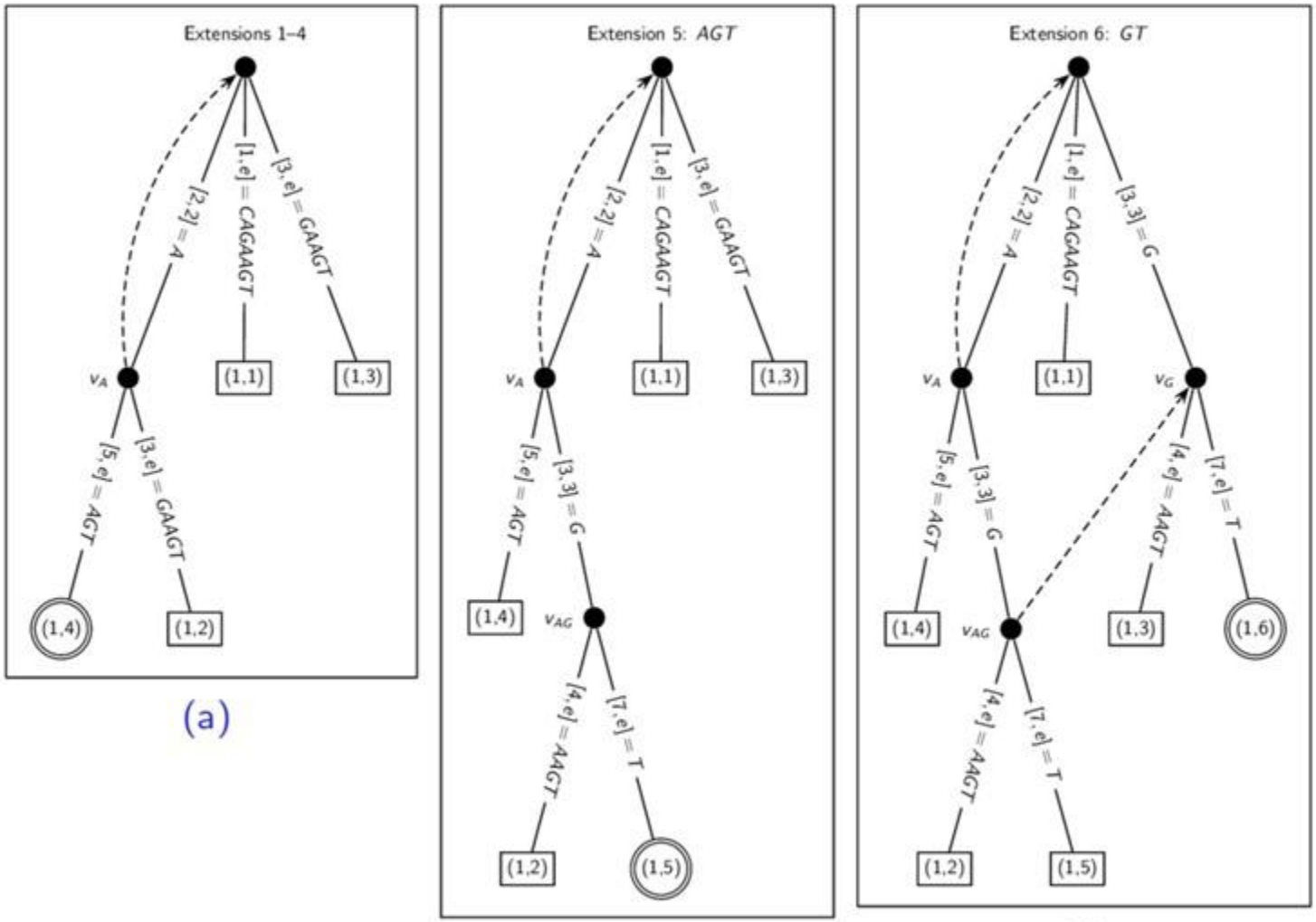


(f) $\mathcal{T}_6$



(g) $\mathcal{T}_7$

Extensions in Phase $i = 7$



Data mining and Machine learning

Part 11. Graph Pattern Mining

Unlabeled and Labeled Graphs

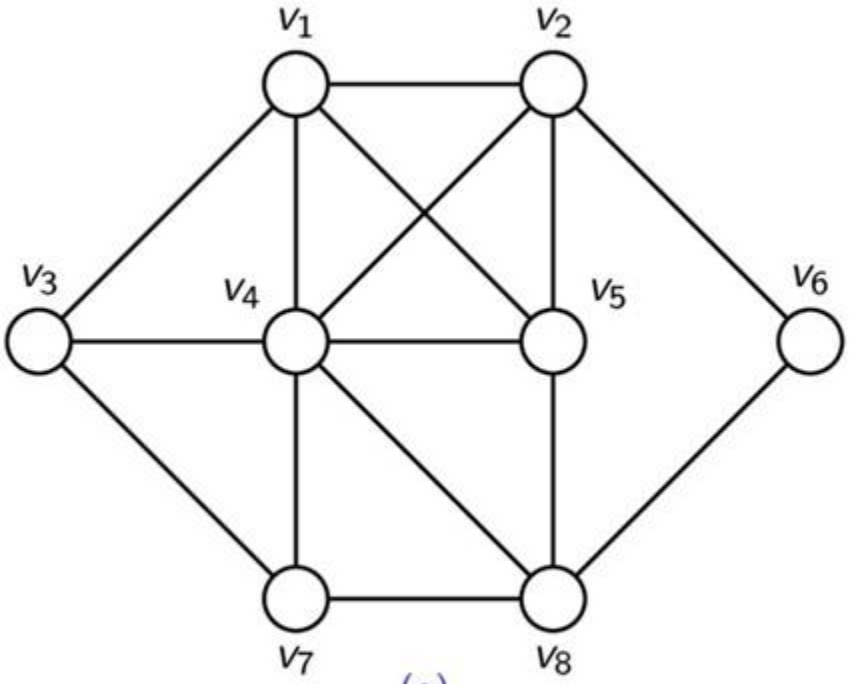
The goal of graph mining is to extract interesting subgraphs from a single large graph (e.g., a social network), or from a database of many graphs.

A graph is a pair $G = (V, E)$ where V is a set of vertices, and $E \subseteq V \times V$ is a set of edges.

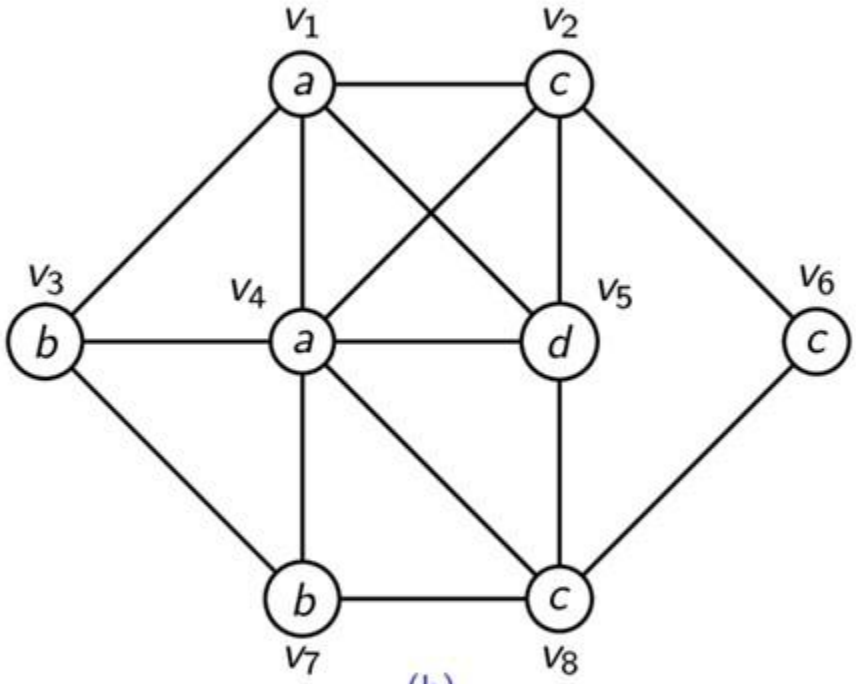
A *labeled graph* has labels associated with its vertices as well as edges. We use $L(u)$ to denote the label of the vertex u, and $L(u, v)$ to denote the label of the edge (u, v), with the set of vertex labels denoted as Σ_V and the set of edge labels as Σ_E. Given an edge $(u, v) \in G$, the tuple $\langle u, v, L(u), L(v), L(u, v) \rangle$ that augments the edge with the node and edge labels is called an *extended edge*.

A graph $G' = (V', E')$ is said to be a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *connected subgraph* is defined as a subgraph G' such that $V' \subseteq V$, $E' \subseteq E$, and for any two nodes $u, v \in V'$, there exists a *path* from u to v in G'.

Unlabeled and Labeled Graphs

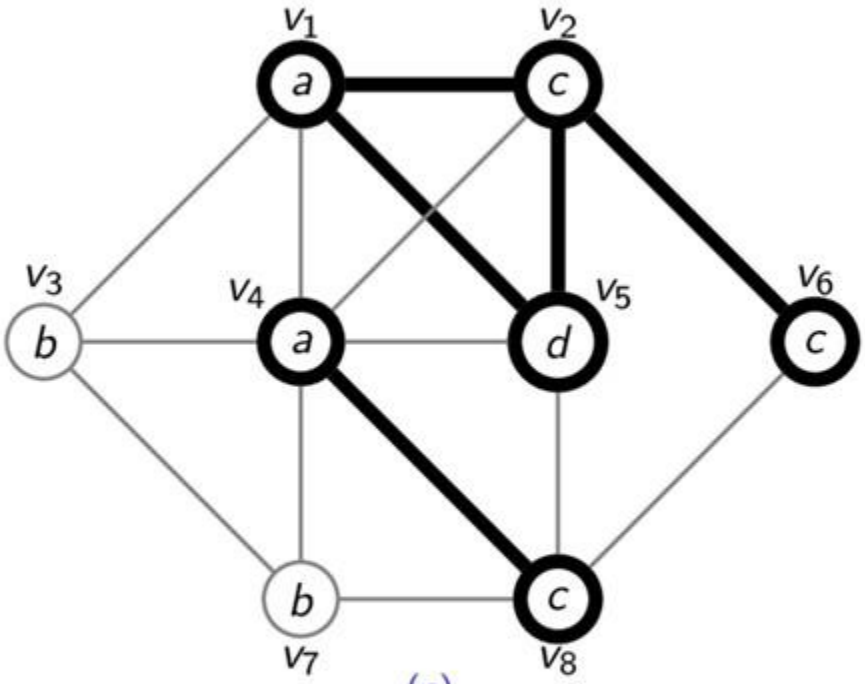


(a)

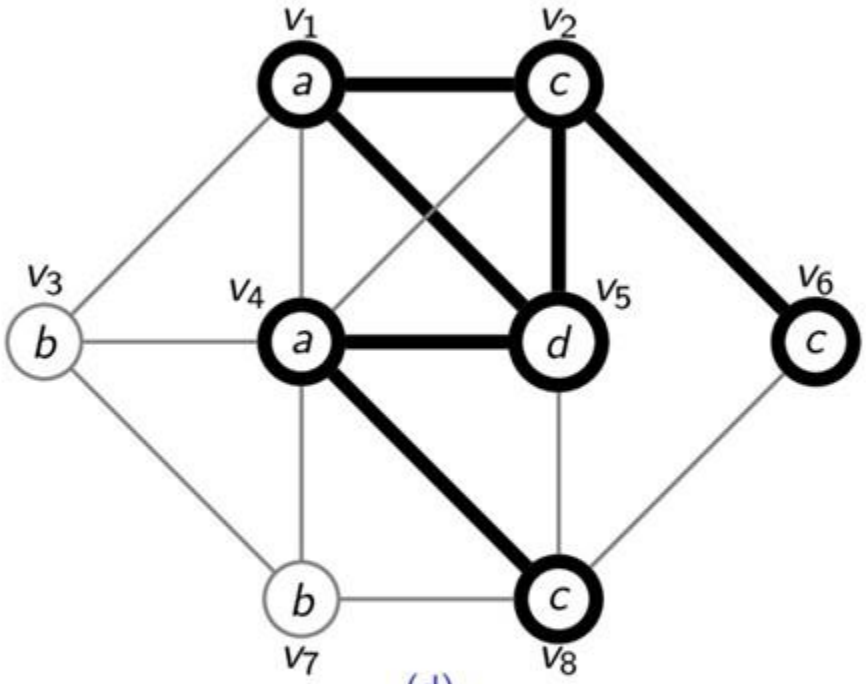


(b)

Subgraph and Connected Subgraph



(c)



(d)

Mining Frequent Subgraph

Given a database of graphs, $\boldsymbol{D} = \{G_1, G_2, \ldots, G_n\}$, and given some graph G, the support of G in $\boldsymbol{D}$ is defined as follows:

$$sup(G) = \left| \{ G_i \in \boldsymbol{D} \mid G \subseteq G_i \} \right|$$

The support is simply the number of graphs in the database that contain G. Given a *minsup* threshold, the goal of graph mining is to mine all frequent connected subgraphs with $sup(G) \geq minsup$.

If we consider subgraphs with m vertices, then there are $\binom{m}{2} = O(m^2)$ possible edges. The number of possible subgraphs with m nodes is then $O(2^{m^2})$ because we may decide either to include or exclude each of the edges. Many of these subgraphs will not be connected, but $O(2^{m^2})$ is a convenient upper bound. When we add labels to the vertices and edges, the number of labeled graphs will be even more.

Graph Pattern Mining

There are two main challenges in frequent subgraph mining.

The first is to systematically generate nonredundant candidate subgraphs. We use *edge-growth* as the basic mechanism for extending the candidates.

The second challenge is to count the support of a graph in the database. This involves subgraph isomorphism checking, as we have to find the set of graphs that contain a given candidate.

Candidate Generation

An effective strategy to enumerate subgraph patterns is *rightmost path extension*.

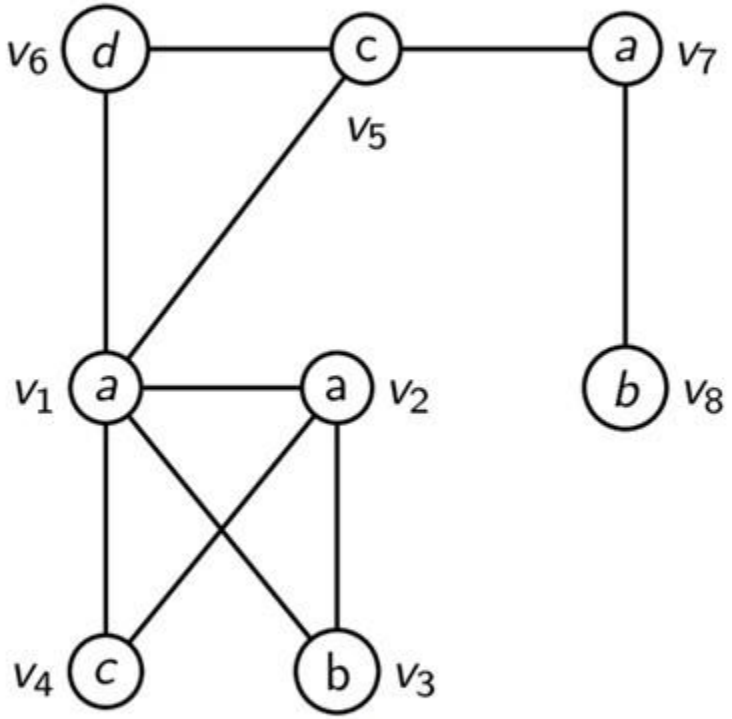
Given a graph G, we perform a depth-first search (DFS) over its vertices, and create a DFS spanning tree, that is, one that covers or spans all the vertices.

Edges that are included in the DFS tree are called *forward* edges, and all other edges are called *backward* edges. Backward edges create cycles in the graph.

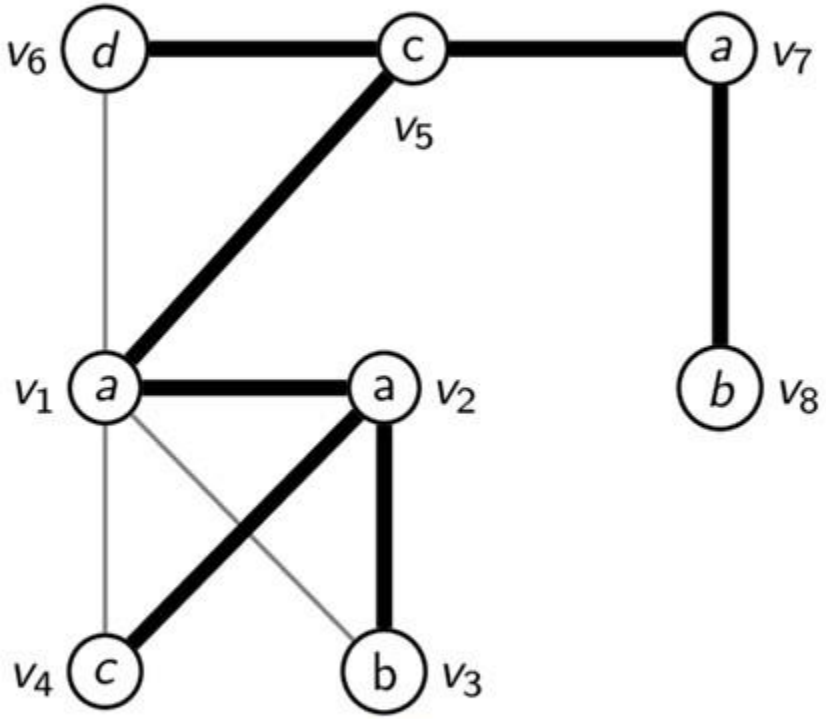
Once we have a DFS tree, define the *rightmost* path as the path from the root to the rightmost leaf, that is, to the leaf with the highest index in the DFS order.

Depth-first Spanning Tree

Starting at v_1, each DFS step chooses the vertex with smallest index



(e)



(f)

DFS Code

For systematic enumeration we rank the set of isomorphic graphs and pick one member as the *canonical* representative.

Let G be a graph and let T_G be a DFS spanning tree for G. The DFS tree T_G defines an ordering of both the nodes and edges in G. The DFS node ordering is obtained by numbering the nodes consecutively in the order they are visited in the DFS walk.

Assume that for a pattern graph G the nodes are numbered according to their position in the DFS ordering, so that $i < j$ implies that v_i comes before v_j in the DFS walk.

The DFS edge ordering is obtained by following the edges between consecutive nodes in DFS order, with the condition that all the backward edges incident with vertex v_i are listed before any of the forward edges incident with it.

The *DFS code* for a graph G, for a given DFS tree T_G, denoted DFScode(G), is defined as the sequence of extended edge tuples of the form $\langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j) \rangle$ listed in the DFS edge order.

Canonical DFS Code

A subgraph is *canonical* if it has the smallest DFS code among all possible isomorphic graphs.

Let t_1 and t_2 be any two DFS code tuples:

$$t_1 = \langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j) \rangle$$
$$t_2 = \langle v_x, v_y, L(v_x), L(v_y), L(v_x, v_y) \rangle$$

We say that t_1 is smaller than t_2, written $t_1 < t_2$, iff

i) $(v_i, v_j) <_e (v_x, v_y)$, or

ii) $(v_i, v_j) = (v_x, v_y)$ and $\langle L(v_i), L(v_j), L(v_i, v_j) \rangle <_l \langle L(v_x), L(v_y), L(v_x, v_y) \rangle$

where $<_e$ is an ordering on the edges and $<_l$ is an ordering on the vertex and edge labels.

The *label order* $<_l$ is the standard lexicographic order on the vertex and edge labels.

The *edge order* $<_e$ is derived from the rules for rightmost path extension, namely that all of a node's backward extensions must be considered before any forward edge from that node, and deep DFS trees are preferred over bushy DFS trees.

Canonical DFS Code: Edge Ordering

Let $e_{ij} = (v_i, v_j)$ and $e_{xy} = (v_x, v_y)$ be any two edges. We say that $e_{ij} <_e e_{xy}$ iff

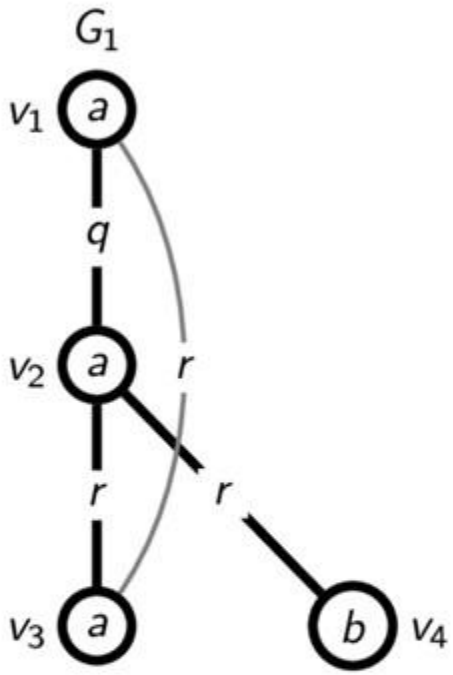
- If e_{ij} and e_{xy} are both forward edges, then (a) $j < y$, or (b) $j = y$ and $i > x$.
- If e_{ij} and e_{xy} are both backward edges, then (a) $i < x$, or (b) $i = x$ and $j < y$.
- If e_{ij} is a forward and e_{xy} is a backward edge, then $j \leq x$.
- If e_{ij} is a backward and e_{xy} is a forward edge, then $i < y$.

The *canonical DFS code* for a graph G is defined as follows:

$$\mathcal{C} = \min_{G'} \left\{ \text{DFScode}(G') \mid G' \text{ is isomorphic to } G \right\}$$

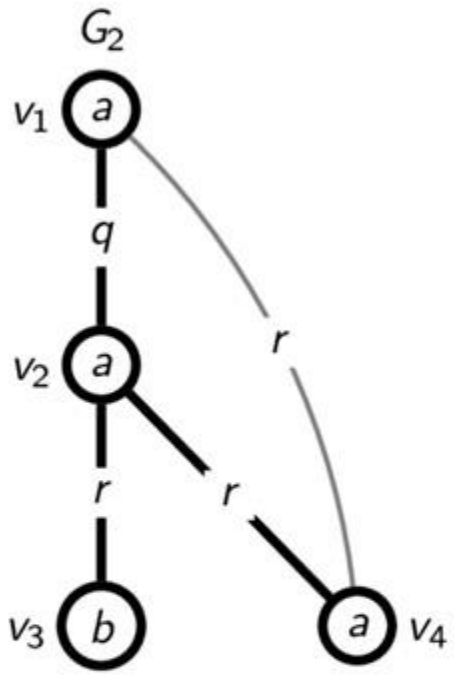
Canonical DFS Code

G_1 has the canonical or minimal DFS code



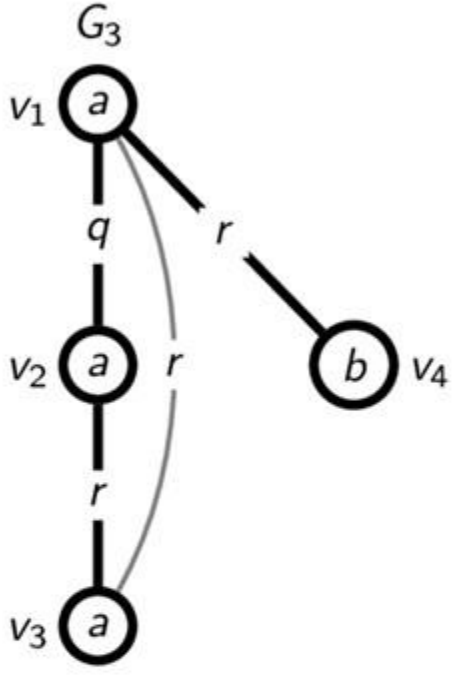
$$t_{11} = \langle v_1, v_2, a, a, q \rangle$$
$$t_{12} = \langle v_2, v_3, a, a, r \rangle$$
$$t_{13} = \langle v_3, v_1, a, a, r \rangle$$
$$t_{14} = \langle v_2, v_4, a, b, r \rangle$$

DFScode(G_1)



$$t_{21} = \langle v_1, v_2, a, a, q \rangle$$
$$t_{22} = \langle v_2, v_3, a, b, r \rangle$$
$$t_{23} = \langle v_2, v_4, a, a, r \rangle$$
$$t_{24} = \langle v_4, v_1, a, a, r \rangle$$

DFScode(G_2)



$$t_{31} = \langle v_1, v_2, a, a, q \rangle$$
$$t_{32} = \langle v_2, v_3, a, a, r \rangle$$
$$t_{33} = \langle v_3, v_1, a, a, r \rangle$$
$$t_{34} = \langle v_1, v_4, a, b, r \rangle$$

DFScode(G_3)

Canonicality Checking

Given a DFS code $C = \{t_1, t_2, \ldots, t_k\}$ comprising k extended edge tuples and the corresponding graph $G(C)$, the task is to check whether the code C is canonical.

This can be accomplished by trying to reconstruct the canonical code C^* for $G(C)$ in an iterative manner starting from the empty code and selecting the least rightmost path extension at each step, where the least edge extension is based on the extended tuple comparison operator.

If at any step the current (partial) canonical DFS code C^* is smaller than C, then we know that C cannot be canonical and can thus be pruned. On the other hand, if no smaller code is found after k extensions then C must be canonical.

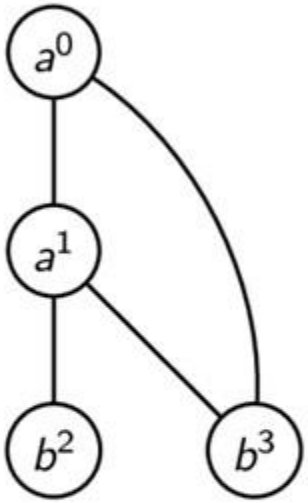
Algorithm IsCanonical: Canonicality Checking

IsCanonical (C):

1 $\boldsymbol{D}_C \leftarrow \{G(C)\}$ // graph corresponding to code C
2 $C^* \leftarrow \emptyset$ // initialize canonical DFScode
3 **for** $i = 1 \cdots k$ **do**
4 $\quad \mathcal{E} = \text{RightMostPath-Extensions}(C^*, \boldsymbol{D}_C)$ // extensions of C^*
5 $\quad (s_i, sup(s_i)) \leftarrow \min\{\mathcal{E}\}$ // least rightmost edge extension of C^*
6 $\quad$ **if** $s_i < t_i$ **then**
7 $\quad\quad$ **return** *false* // C^* is smaller, thus C is not canonical
8 $\quad C^* \leftarrow C^* \cup s_i$
9 **return** *true* // no smaller code exists; C is canonical

Canonicality Checking

G



C

$t_1 = \langle 0, 1, a, a \rangle$
$t_2 = \langle 1, 2, a, b \rangle$
$t_3 = \langle 1, 3, a, b \rangle$
$t_4 = \langle 3, 0, b, a \rangle$

Step 1

G^*

C^*

$s_1 = \langle 0, 1, a, a \rangle$

Step 2

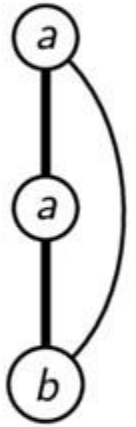
G^*

C^*

$s_1 = \langle 0, 1, a, a \rangle$
$s_2 = \langle 1, 2, a, b \rangle$

Step 3

G^*



C^*

$s_1 = \langle 0, 1, a, a \rangle$
$s_2 = \langle 1, 2, a, b \rangle$
$s_3 = \langle 2, 0, b, a \rangle$

Graph and Subgraph Isomorphism

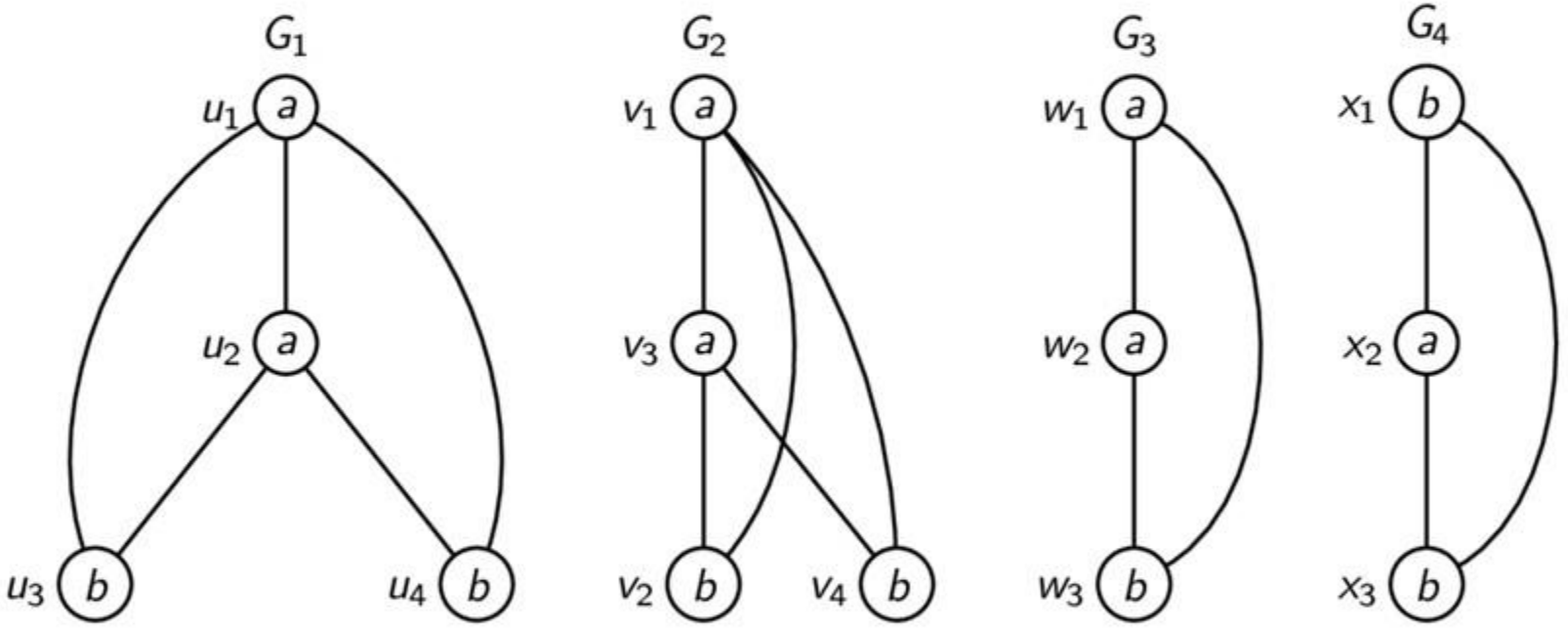
A graph $G' = (V', E')$ is said to be *isomorphic* to another graph $G = (V, E)$ if there exists a bijective function $\phi : V' \to V$, i.e., both injective (into) and surjective (onto), such that

- $(u, v) \in E' \iff (\phi(u), \phi(v)) \in E$
- $\forall u \in V'$, $L(u) = L(\phi(u))$
- $\forall (u, v) \in E'$, $L(u, v) = L(\phi(u), \phi(v))$

In other words, the *isomorphism* ϕ preserves the edge adjacencies as well as the vertex and edge labels. Put differently, the extended tuple $\langle u, v, L(u), L(v), L(u, v)\rangle \in G'$ if and only if $\langle \phi(u), \phi(v), L(\phi(u)), L(\phi(v)), L(\phi(u), \phi(v))\rangle \in G$.

If the function ϕ is only injective but not surjective, we say that the mapping ϕ is a *subgraph isomorphism* from G' to G. In this case, we say that G' is isomorphic to a subgraph of G, that is, G' is *subgraph isomorphic* to G, denoted $G' \subseteq G$; we also say that G *contains* G'.

Graph and Subgraph Isomorphism



Graph Isomorphism

G_1 and G_2 are isomorphic graphs. There are several possible isomorphisms between G_1 and G_2. An example of an isomorphism $\phi : V_2 \to V_1$ is

$$\phi(v_1) = u_1 \qquad \phi(v_2) = u_3 \qquad \phi(v_3) = u_2 \qquad \phi(v_4) = u_4$$

The inverse mapping ϕ^{-1} specifies the isomorphism from G_1 to G_2. For example, $\phi^{-1}(u_1) = v_1$, $\phi^{-1}(u_2) = v_3$, and so on.

The set of all possible isomorphisms from G_2 to G_1 are as follows:

	v_1	v_2	v_3	v_4
ϕ_1	u_1	u_3	u_2	u_4
ϕ_2	u_1	u_4	u_2	u_3
ϕ_3	u_2	u_3	u_1	u_4
ϕ_4	u_2	u_4	u_1	u_3

Subgraph Isomorphism

The graph G_3 is subgraph isomorphic to both G_1 and G_2. The set of all possible subgraph isomorphisms from G_3 to G_1 are as follows:

	w_1	w_2	w_3
ϕ_1	u_1	u_2	u_3
ϕ_2	u_1	u_2	u_4
ϕ_3	u_2	u_1	u_3
ϕ_4	u_2	u_1	u_4

The graph G_4 is **not** subgraph isomorphic to either G_1 or G_2, since x_1 and x_3 are directly connected, and u_3 and u_4, from G_1, are not. The same applies to v_2 and v_4 from G_2.

Subgraph Isomorphisms

To enumerate all the possible isomorphisms from C to each graph $G_i \in \boldsymbol{D}$ the function SubgraphIsomorphisms, accepts a code C and a graph G, and returns the set of all isomorphisms between C and G.

The set of isomorphisms Φ is initialized by mapping vertex 0 in C to each vertex x in G that shares the same label as 0, that is, if $L(x) = L(0)$.

The method considers each tuple t_i in C and extends the current set of partial isomorphisms. Let $t_i = \langle u, v, L(u), L(v), L(u,v) \rangle$. We have to check if each isomorphism $\phi \in \Phi$ can be extended in G using the information from t_i

If t_i is a forward edge, then we seek a neighbor x of $\phi(u)$ in G such that x has not already been mapped to some vertex in C, that is, $\phi^{-1}(x)$ should not exist, and the node and edge labels should match, that is, $L(x) = L(v)$, and $L(\phi(u), x) = L(u, v)$. If so, ϕ can be extended with the mapping $\phi(v) \rightarrow x$. The new extended isomorphism, denoted ϕ', is added to the initially empty set of isomorphisms Φ'.

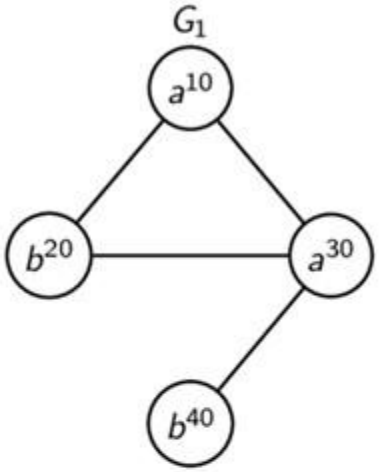
If t_i is a backward edge, we have to check if $\phi(v)$ is a neighbor of $\phi(u)$ in G. If so, we add the current isomorphism ϕ to Φ'.

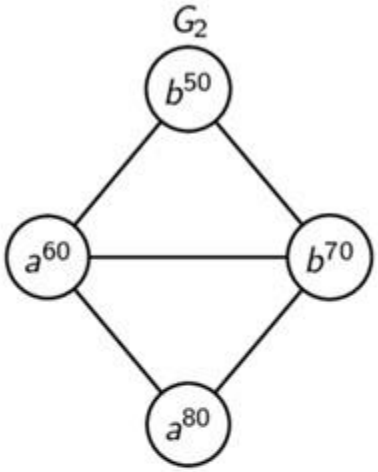
Algorithm SubgraphIsomorphisms

SubgraphIsomorphisms ($C = \{t_1, t_2, \ldots, t_k\}$, G):

1 $\Phi \leftarrow \{\phi(0) \rightarrow x \mid x \in G \text{ and } L(x) = L(0)\}$
2 **foreach** $t_i \in C$, $i = 1, \ldots, k$ **do**
3 $\quad \langle u, v, L(u), L(v), L(u,v) \rangle \leftarrow t_i$ // expand extended edge t_i
4 $\quad \Phi' \leftarrow \emptyset$ // partial isomorphisms including t_i
5 $\quad$ **foreach** *partial isomorphism* $\phi \in \Phi$ **do**
6 $\quad\quad$ **if** $v > u$ **then**
$\quad\quad\quad$ // forward edge
7 $\quad\quad\quad$ **foreach** $x \in N_G(\phi(u))$ **do**
8 $\quad\quad\quad\quad$ **if** $\nexists \phi^{-1}(x)$ *and* $L(x) = L(v)$ *and* $L(\phi(u), x) = L(u, v)$ **then**
9 $\quad\quad\quad\quad\quad \phi' \leftarrow \phi \cup \{\phi(v) \rightarrow x\}$
10 $\quad\quad\quad\quad\quad$ Add ϕ' to Φ'
11 $\quad\quad$ **else**
$\quad\quad\quad$ // backward edge
12 $\quad\quad\quad$ **if** $\phi(v) \in N_{G_j}(\phi(u))$ **then** Add ϕ to Φ' // valid isomorphism
13
14 $\quad \Phi \leftarrow \Phi'$ // update partial isomorphisms
15 **return** Φ

Subgraph Isomorphisms

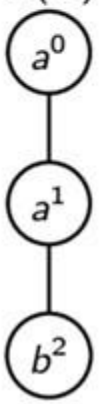




C

$t_1 : \langle 0, 1, a, a \rangle$
$t_2 : \langle 1, 2, a, b \rangle$

$G(C)$



Initial Φ

id	ϕ	0
G_1	ϕ_1	10
	ϕ_2	30
G_2	ϕ_3	60
	ϕ_4	80

Add t_1

id	ϕ	0, 1
G_1	ϕ_1	10, 30
	ϕ_2	30, 10
G_2	ϕ_3	60, 80
	ϕ_4	80, 60

Add t_2

id	ϕ	0, 1, 2
G_1	ϕ'_1	10, 30, 20
	ϕ''_1	10, 30, 40
	ϕ_2	30, 10, 20
G_2	ϕ_3	60, 80, 70
	ϕ'_4	80, 60, 50
	ϕ''_4	80, 60, 70

Rightmost Path Extensions

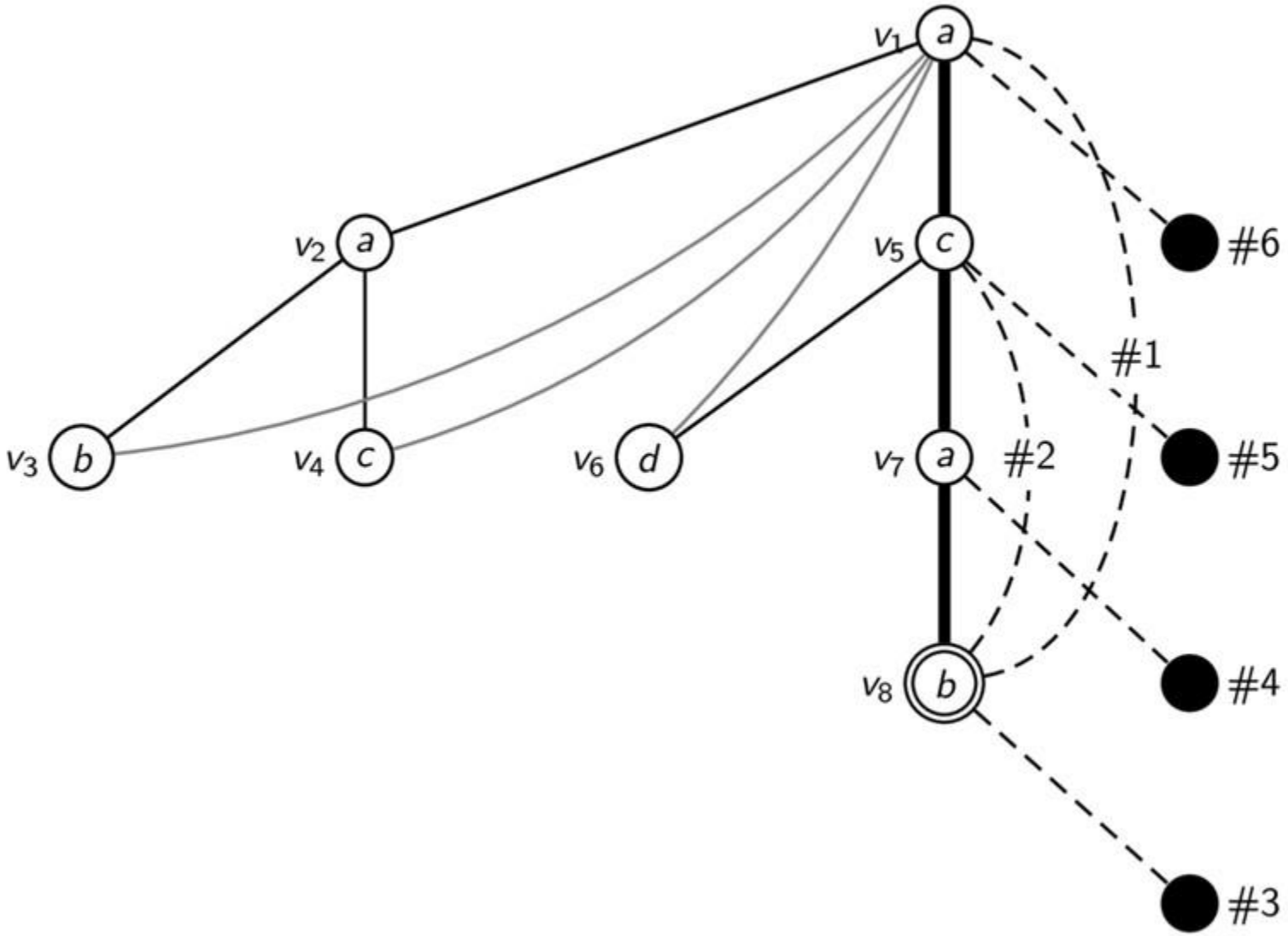
For generating new candidates from a given graph G, we extend it by adding a new edge to vertices only on the rightmost path. We can either extend G by adding backward edges from the *rightmost vertex* to some other vertex on the rightmost path (disallowing self-loops or multi-edges), or we can extend G by adding forward edges from any of the vertices on the rightmost path. A backward extension does not add a new vertex, whereas a forward extension adds a new vertex.

For systematic candidate generation we impose a total order on the extensions, as follows: First, we try all backward extensions from the rightmost vertex, and then we try forward extensions from vertices on the rightmost path.

Among the backward edge extensions, if u_r is the rightmost vertex, the extension (u_r, v_i) is tried before (u_r, v_j) if $i < j$. In other words, backward extensions closer to the root are considered before those farther away from the root along the rightmost path.

Among the forward edge extensions, if v_x is the new vertex to be added, the extension (v_i, v_x) is tried before (v_j, v_x) if $i > j$. In other words, the vertices farther from the root (those at greater depth) are extended before those closer to the root. Also note that the new vertex will be numbered $x = r + 1$, as it will become the new rightmost vertex after the extension.

Rightmost Path Extensions



Extension and Support Computation

The support computation task is to find the number of graphs in the database $\boldsymbol{D}$ that contain a candidate subgraph, which is very expensive because it involves subgraph isomorphism checks. gSpan combines the tasks of enumerating candidate extensions and support computation.

Assume that $\boldsymbol{D} = \{G_1, G_2, \ldots, G_n\}$ comprises n graphs. Let $C = \{t_1, t_2, \ldots, t_k\}$ denote a frequent canonical DFS code comprising k edges, and let $G(C)$ denote the graph corresponding to code C. The task is to compute the set of possible rightmost path extensions from C, along with their support values.

Given code C, gSpan first records the nodes on the rightmost path (R), and the rightmost child (u_r). Next, gSpan considers each graph $G_i \in \boldsymbol{D}$. If $C = \emptyset$, then each distinct label tuple of the form $\langle L(x), L(y), L(x,y)\rangle$ for adjacent nodes x and y in G_i contributes a forward extension $\langle 0, 1, L(x), L(y), L(x,y)\rangle$ On the other hand, if C is not empty, then gSpan enumerates all possible subgraph isomorphisms Φ_i between the code C and graph G_i. Given subgraph isomorphism $\phi \in \Phi_i$, gSpan finds all possible forward and backward edge extensions, and stores them in the extension set $\mathcal{E}$.

Forward and Backward Extensions

Backward extensions are allowed only from the rightmost child u_r in C to some other node on the rightmost path R.

The method considers each neighbor x of $\phi(u_r)$ in G_i and checks whether it is a mapping for some vertex $v = \phi^{-1}(x)$ along the rightmost path R in C. If the edge (u_r, v) does not already exist in C, it is a new extension, and the extended tuple $b = \langle u_r, v, L(u_r), L(v), L(u_r, v) \rangle$ is added to the set of extensions $\mathcal{E}$, along with the graph id i that contributed to that extension.

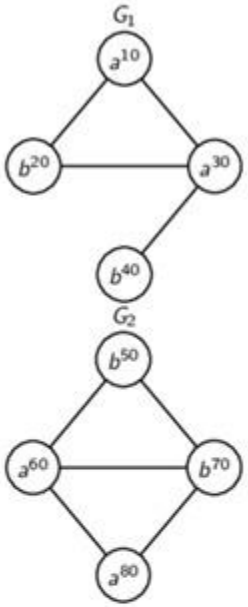
Forward extensions are allowed only from nodes on the rightmost path R to new nodes. For each node u in R, the algorithm finds a neighbor x in G_i that is not in a mapping from some node in C. For each such node x, the forward extension $f = \langle u, u_{r+1}, L(\phi(u)), L(x), L(\phi(u), x) \rangle$ is added to $\mathcal{E}$, along with the graph id i. Because a forward extension adds a new vertex to the graph $G(C)$, the id of the new node in C must be u_{r+1}, that is, one more than the highest numbered node in C, which by definition is the rightmost child u_r.

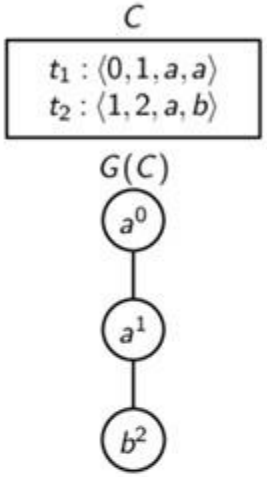
Algorithm RightMostPath-Extensions

RightMostPath-Extensions (C, D):

1 $R \leftarrow$ nodes on the rightmost path in C
2 $u_r \leftarrow$ rightmost child in C // `dfs number`
3 $\mathcal{E} \leftarrow \emptyset$ // `set of extensions from` C
4 **foreach** $G_i \in D$, $i = 1, \ldots, n$ **do**
5 **if** $C = \emptyset$ **then**
6 **foreach** *distinct* $\langle L(x), L(y), L(x,y) \rangle \in G_i$ **do**
7 $f = \langle 0, 1, L(x), L(y), L(x,y) \rangle$
8 Add tuple f to $\mathcal{E}$ along with graph id i
9 **else**
10 $\Phi_i = \text{SubgraphIsomorphisms}(C, G_i)$
11 **foreach** *isomorphism* $\phi \in \Phi_i$ **do**
12 **foreach** $x \in N_{G_i}(\phi(u_r))$ *such that* $\exists v \leftarrow \phi^{-1}(x)$ **do**
13 **if** $v \in R$ *and* $(u_r, v) \notin G(C)$ **then**
14 $b = \langle u_r, v, L(u_r), L(v), L(u_r, v) \rangle$
15 Add tuple b to $\mathcal{E}$ along with graph id i
16 **foreach** $u \in R$ **do**
17 **foreach** $x \in N_{G_i}(\phi(u))$ *and* $\nexists \phi^{-1}(x)$ **do**
18 $f = \langle u, u_{r+1}, L(\phi(u)), L(x), L(\phi(u), x) \rangle$
19 Add tuple f to $\mathcal{E}$ along with graph id i
20 **foreach** *distinct extension* $s \in \mathcal{E}$ **do**
21 $sup(s) =$ number of distinct graph ids that support tuple s
22 **return** *set of pairs* $\langle s, sup(s) \rangle$ *for extensions* $s \in \mathcal{E}$, *in tuple sorted order*

Righmost Path Extensions





(a) G_1, G_2, Code C and graph $G(C)$

Φ	ϕ	0	1	2
Φ_1	ϕ_1	10	30	20
	ϕ_2	10	30	40
	ϕ_3	30	10	20
Φ_2	ϕ_4	60	80	70
	ϕ_5	80	60	50
	ϕ_6	80	60	70

(b) Subgraph isomorphisms

Id	ϕ	Extensions
G_1	ϕ_1	$\{\langle 2,0,b,a \rangle, \langle 1,3,a,b \rangle\}$
	ϕ_2	$\{\langle 1,3,a,b \rangle, \langle 0,3,a,b \rangle\}$
	ϕ_3	$\{\langle 2,0,b,a \rangle, \langle 0,3,a,b \rangle\}$
G_2	ϕ_4	$\{\langle 2,0,b,a \rangle, \langle 2,3,b,b \rangle, \langle 0,3,a,b \rangle\}$
	ϕ_5	$\{\langle 2,3,b,b \rangle, \langle 1,3,a,b \rangle\}$
	ϕ_6	$\{\langle 2,0,b,a \rangle, \langle 2,3,b,b \rangle, \langle 1,3,a,b \rangle\}$

(c) Edge extensions

Extension	Support
$\langle 2,0,b,a \rangle$	2
$\langle 2,3,b,b \rangle$	1
$\langle 1,3,a,b \rangle$	2
$\langle 0,3,a,b \rangle$	2

(d) Extensions (sorted) and supports

gSpan Graph Mining Algorithm

gSpan enumerates patterns in a depth-first manner, starting with the empty code. Given a canonical and frequent code C, gSpan first determines the set of possible edge extensions along the rightmost path.

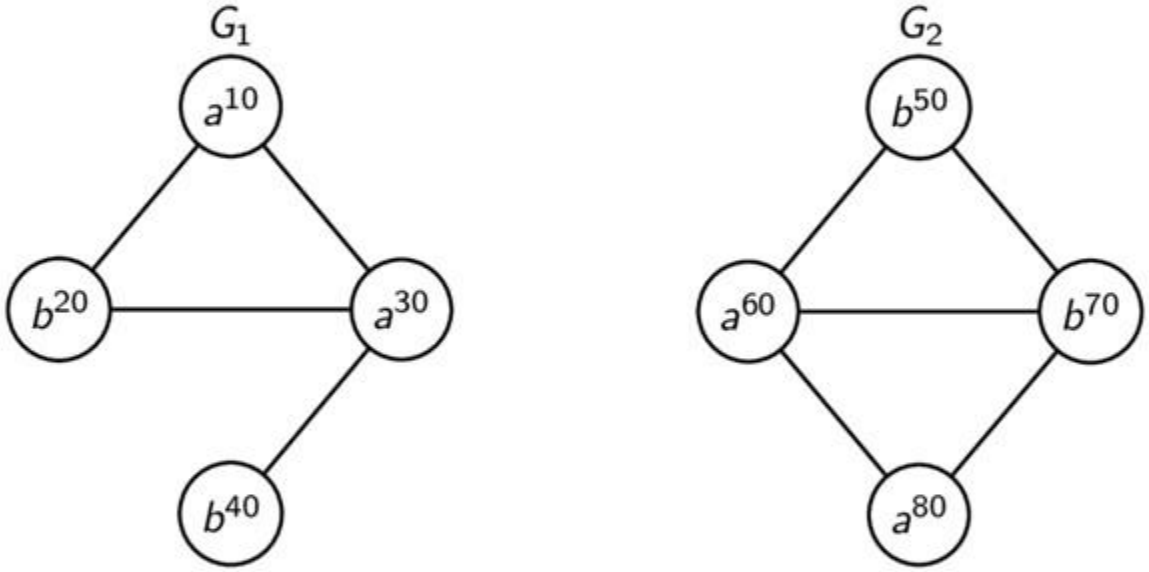
The function RightMostPath-Extensions returns the set of edge extensions along with their support values, $\mathcal{E}$. Each extended edge t in $\mathcal{E}$ leads to a new candidate DFS code $C' = C \cup \{t\}$, with support $sup(C) = sup(t)$.

For each new candidate code, gSpan checks whether it is frequent and canonical, and if so gSpan recursively extends C' The algorithm stops when there are no more frequent and canonical extensions possible.

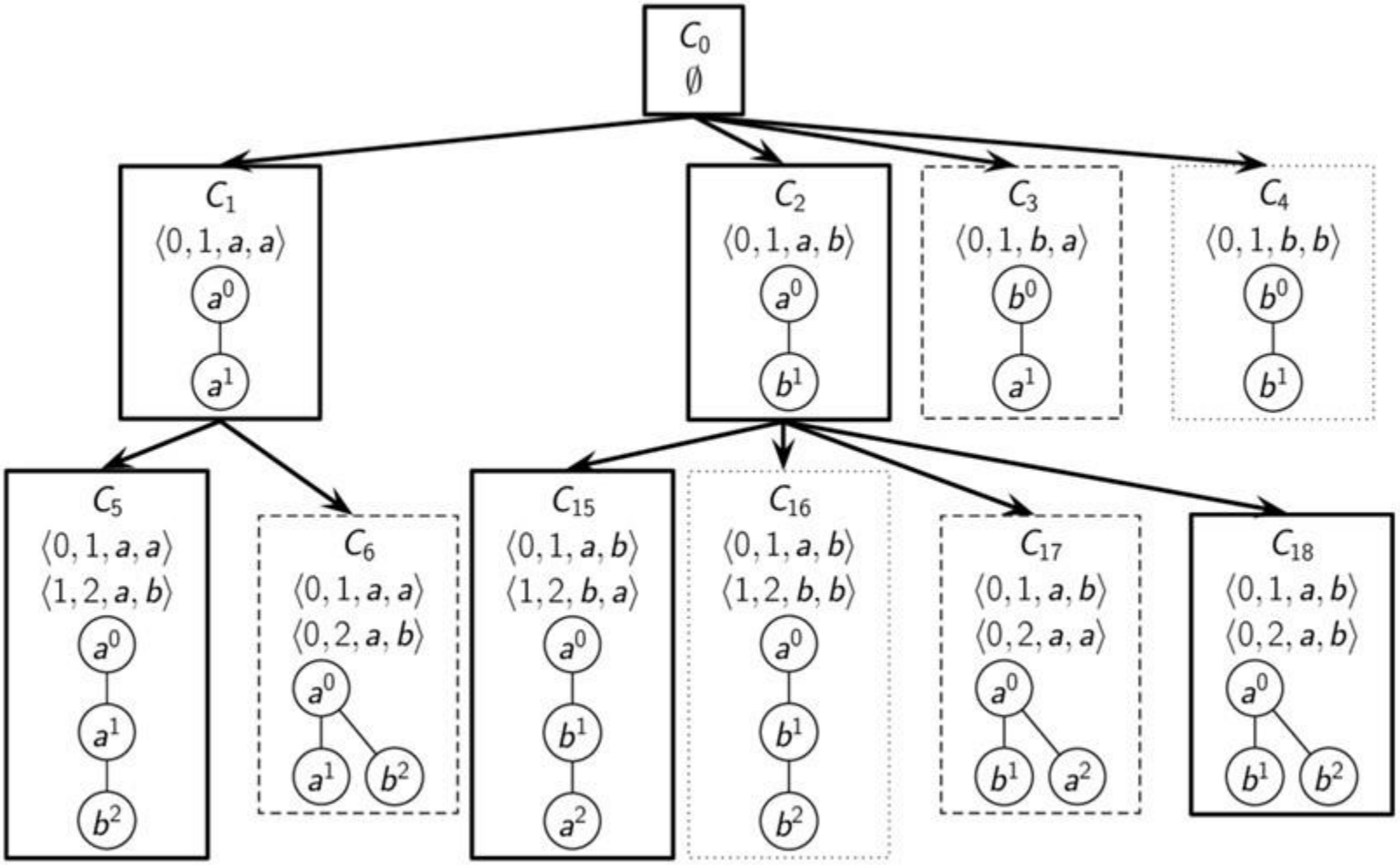
Algorithm gSpan

```
// Initial Call:  C ← ∅
gSpan (C, D, minsup):
1 E ← RightMostPath-Extensions(C, D) // extensions and
      supports
2 foreach (t, sup(t)) ∈ E do
3     C' ← C ∪ t // extend code with extended edge tuple t
4     sup(C') ← sup(t) // record the support of new extension
      // recursively call gSpan if code is frequent and
         canonical
5     if sup(C') ≥ minsup and IsCanonical (C') then
6         gSpan (C', D, minsup)
```

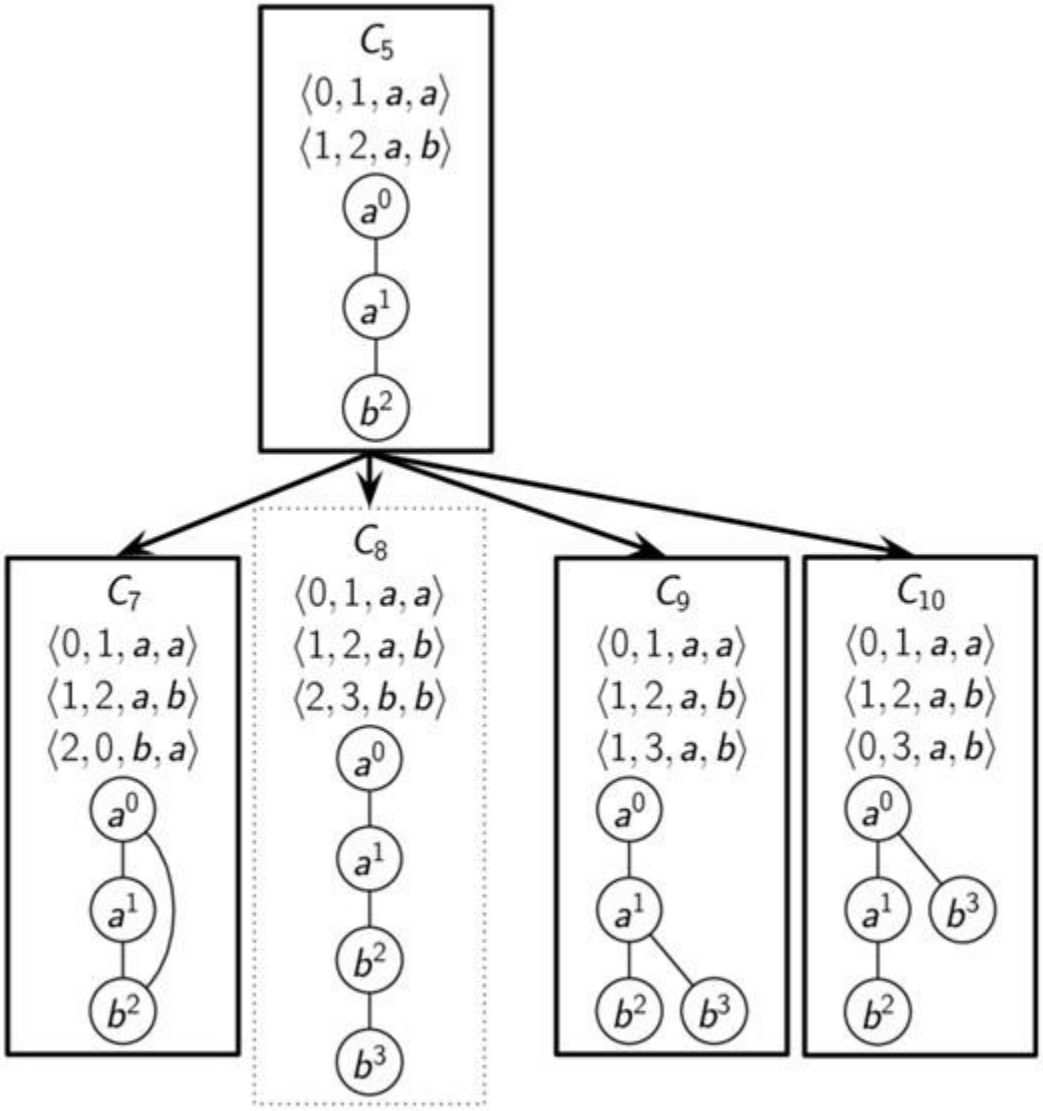
Example Graph Database: gSpan



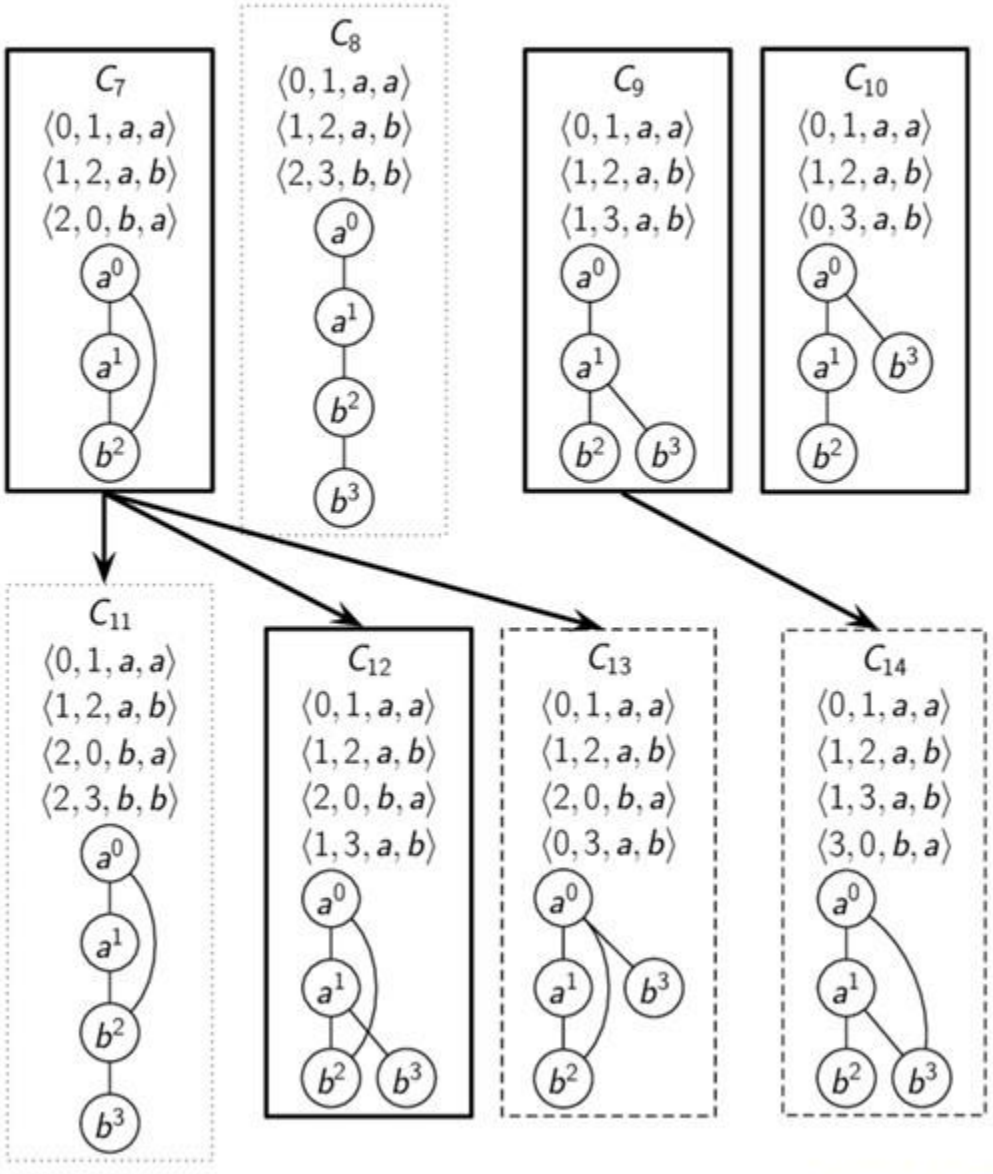
Frequent Graph Mining: gSpan



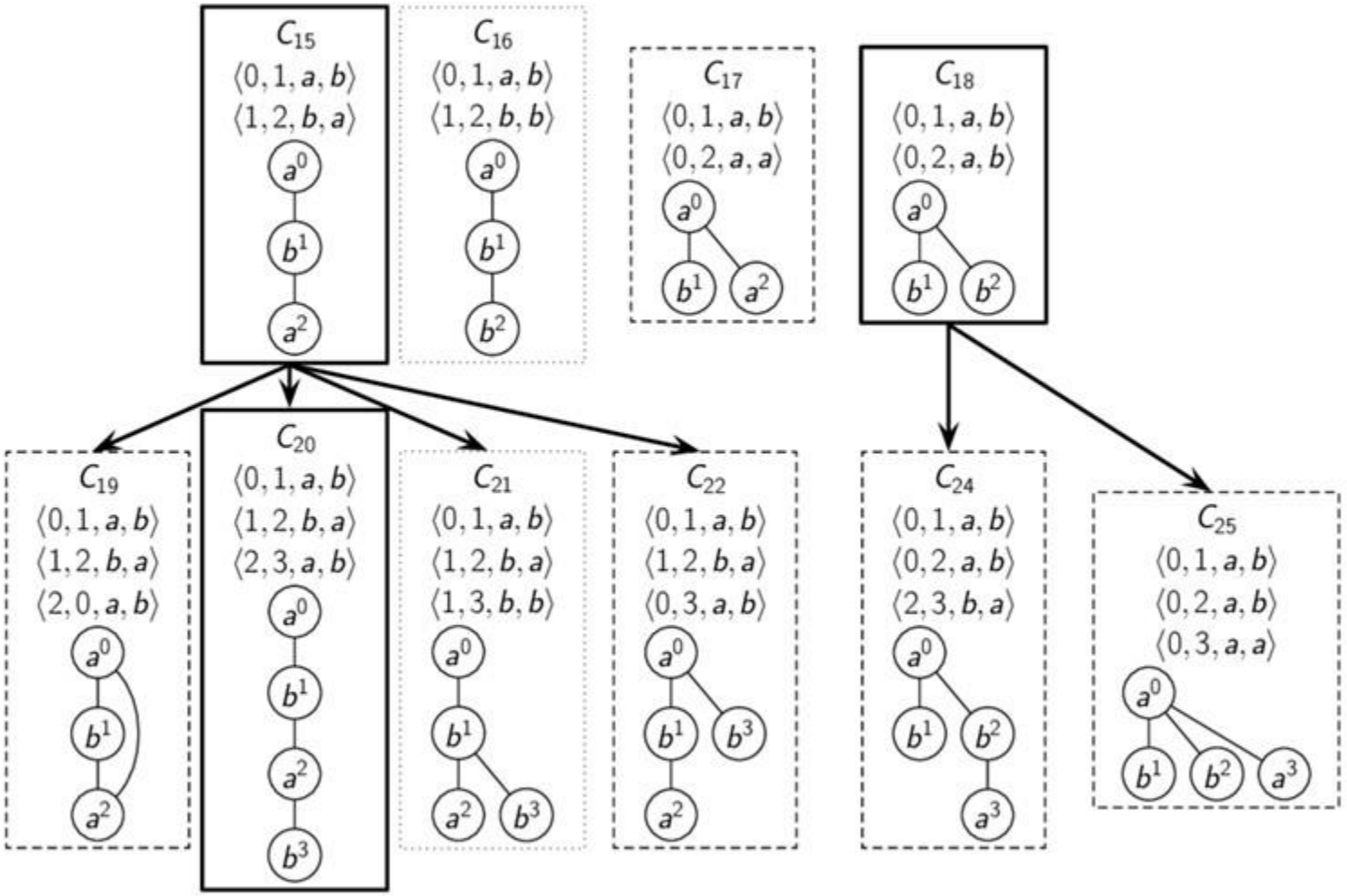
Frequent Graph Mining: gSpan



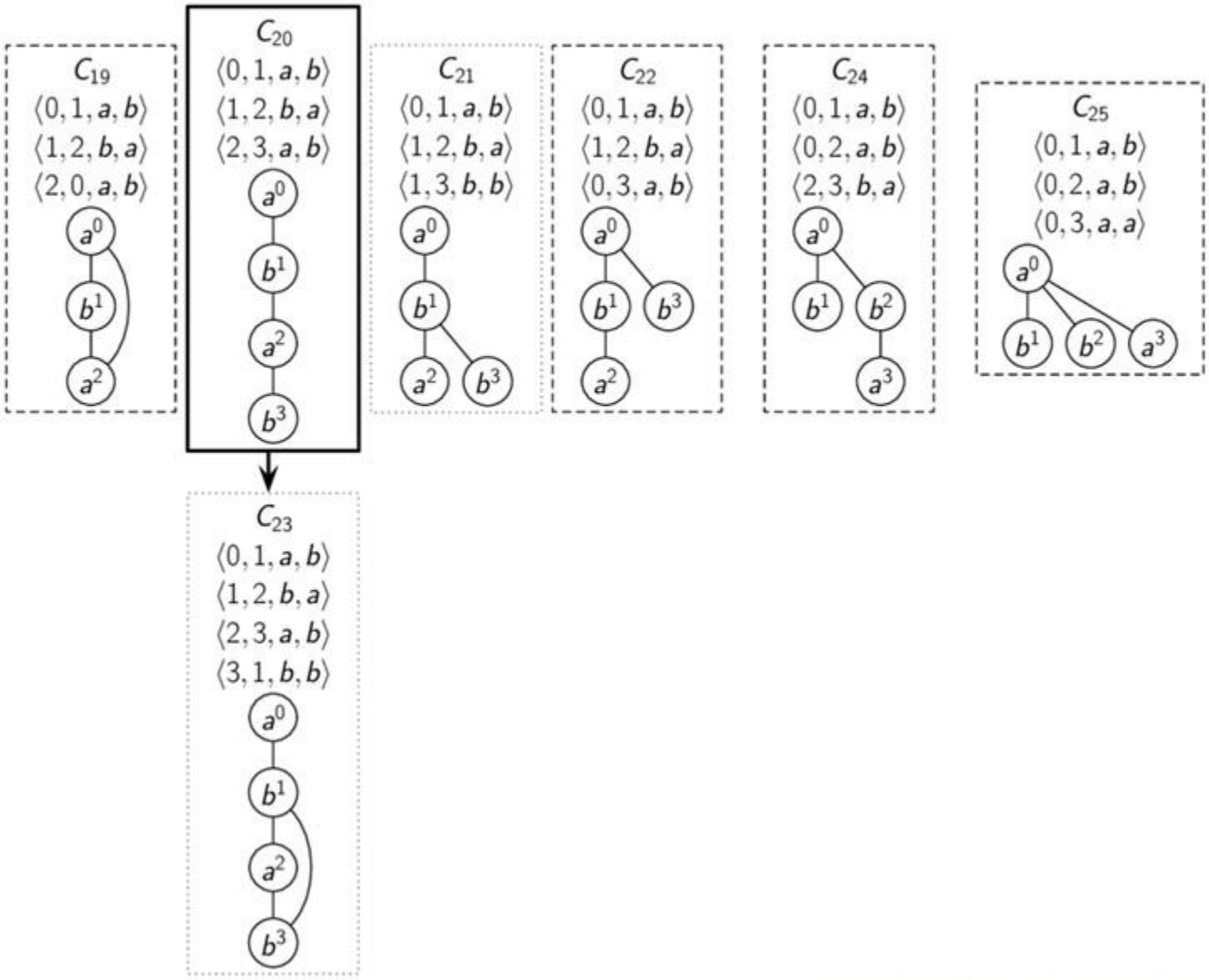
Frequent Graph Mining: gSpan



Frequent Graph Mining: gSpan



Frequent Graph Mining: gSpan



Data mining and Machine learning

Part 12. Pattern and Rule Assessment

Rule Assessment Measures: Support and Confidence

Support: The *support* of the rule is defined as the number of transactions that contain both X and Y, that is,

$$sup(X \longrightarrow Y) = sup(XY) = |\boldsymbol{t}(XY)|$$

The *relative support* is the fraction of transactions that contain both X and Y, that is, the empirical joint probability of the items comprising the rule

$$rsup(X \longrightarrow Y) = P(XY) = rsup(XY) = \frac{sup(XY)}{|\boldsymbol{D}|}$$

Confidence: The *conf idence* of a rule is the conditional probability that a transaction contains the consequent Y given that it contains the antecedent X:

$$conf(X \longrightarrow Y) = P(Y|X) = \frac{P(XY)}{P(X)} = \frac{rsup(XY)}{rsup(X)} = \frac{sup(XY)}{sup(X)}$$

Example Dataset: Support and Confidence

Tid	Items
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

Frequent itemsets: $minsup = 3$

sup	$rsup$	Itemsets
3	0.5	ABD, $ABDE$, AD, ADE BCE, BDE, CE, DE
4	0.67	A, C, D, AB, ABE, AE, BC, BD
5	0.83	E, BE
6	1.0	B

Rule confidence

Rule	$conf$
$A \longrightarrow E$	1.00
$E \longrightarrow A$	0.80
$B \longrightarrow E$	0.83
$E \longrightarrow B$	1.00
$E \longrightarrow BC$	0.60
$BC \longrightarrow E$	0.75

Confidence should be evaluated considering the support of the rule components. For instance, since $P(BC) = 0.67$, the rule $E \longrightarrow BC$, with a 60% confidence, has a deleterious effect on BC.

Rule Assessment Measures: Lift, Leverage and Jaccard

Lift: Lift is defined as the ratio of the observed joint probability of X and Y to the expected joint probability if they were statistically independent, that is,

$$lift(X \longrightarrow Y) = \frac{P(XY)}{P(X) \cdot P(Y)} = \frac{rsup(XY)}{rsup(X) \cdot rsup(Y)} = \frac{conf(X \longrightarrow Y)}{rsup(Y)}$$

Leverage: Leverage measures the difference between the observed and expected joint probability of XY assuming that X and Y are independent

$$leverage(X \longrightarrow Y) = P(XY) - P(X) \cdot P(Y) = rsup(XY) - rsup(X) \cdot rsup(Y)$$

Jaccard: The Jaccard coefficient measures the similarity between two sets. When applied as a rule assessment measure it computes the similarity between the tidsets of X and Y:

$$\begin{aligned} jaccard(X \longrightarrow Y) &= \frac{|\boldsymbol{t}(X) \cap \boldsymbol{t}(Y)|}{|\boldsymbol{t}(X) \cup \boldsymbol{t}(Y)|} \\ &= \frac{P(XY)}{P(X) + P(Y) - P(XY)} \end{aligned}$$

Lift and Leverage

Rule	*lift*
$AE \longrightarrow BC$	0.75
$CE \longrightarrow AB$	1.00
$BE \longrightarrow AC$	1.20

lift < 1 indicates the rule support is smaller than expected, while *lift* > 1 means the reverse.

Rule	*rsup*	*lift*	*leverage*
$ACD \longrightarrow E$	0.17	1.20	0.03
$AC \longrightarrow E$	0.33	1.20	0.06
$AB \longrightarrow D$	0.50	1.12	0.06
$A \longrightarrow E$	0.67	1.20	0.11

Lift and leverage must be evaluated together, since the same lift may refer to significantly different leverages.

Lift, Jaccard, and Confidence

Rule	*rsup*	*conf*	*lift*
$E \longrightarrow AC$	0.33	0.40	1.20
$E \longrightarrow AB$	0.67	0.80	1.20
$B \longrightarrow E$	0.83	0.83	1.00

Lift and confidence must be evaluated together, to avoid either weak rules or rules where the antecedent and consequent are independent ($lift = 1$).

Rule	*rsup*	*lift*	*jaccard*
$A \longrightarrow C$	0.33	0.75	0.33
$A \longrightarrow E$	0.67	1.20	0.80
$A \longrightarrow B$	0.67	1.00	0.67

Jaccard and Lift provide similar information, but Jaccard is bounded to the interval $[0, 1]$.

Contingency Table for X and Y

We may also define the contingency table for X and Y, and exploit their absence, represented by $\neg X$ and $\neg Y$.

	Y	$\neg Y$	
X	$sup(XY)$	$sup(X\neg Y)$	$sup(X)$
$\neg X$	$sup(\neg XY)$	$sup(\neg X\neg Y)$	$sup(\neg X)$
	$sup(Y)$	$sup(\neg Y)$	$\|\boldsymbol{D}\|$

Rule Assessment Measures: Conviction

Define $\neg X$ to be the event that X is not contained in a transaction, that is, $X \not\subseteq t \in \mathcal{T}$, and likewise for $\neg Y$. There are, in general, four possible events depending on the occurrence or non-occurrence of the itemsets X and Y as depicted in the contingency table.

Conviction measures the expected error of the rule, that is, how often X occurs in a transaction where Y does not. It is thus a measure of the strength of a rule with respect to the complement of the consequent, defined as

$$conv(X \longrightarrow Y) = \frac{P(X) \cdot P(\neg Y)}{P(X \neg Y)} = \frac{1}{lift(X \longrightarrow \neg Y)} = \frac{1 - P(Y)}{1 - conf(X \longrightarrow Y)}$$

If the joint probability of $X \neg Y$ is less than that expected under independence of X and $\neg Y$, then conviction is high, and vice versa.

Rule Conviction

Rule			*rsup*	*conf*	*lift*	*conv*
A	$\longrightarrow$	DE	0.50	0.75	1.50	2.00
DE	$\longrightarrow$	A	0.50	1.00	1.50	∞
E	$\longrightarrow$	C	0.50	0.60	0.90	0.83
C	$\longrightarrow$	E	0.50	0.75	0.90	0.68

$A \longrightarrow DE$ is a strong rule, confirmed by high values of both lift and conviction.
$DE \longrightarrow A$ has 100% confidence, being a trivial rule.
$E \longrightarrow C$ and $C \longrightarrow E$ are weak, but, despite the same support and lift, conviction indicates that the $E \longrightarrow C$ is stronger than $C \longrightarrow E$, while confidence indicates the reverse.

Rule Assessment Measures: Odds Ratio

The odds ratio utilizes all four entries from the contingency table. Let us divide the dataset into two groups of transactions – those that contain X and those that do not contain X. Define the odds of Y in these two groups as follows:

$$odds(Y|X) = \frac{P(XY)/P(X)}{P(X\neg Y)/P(X)} = \frac{P(XY)}{P(X\neg Y)}$$

$$odds(Y|\neg X) = \frac{P(\neg XY)/P(\neg X)}{P(\neg X\neg Y)/P(\neg X)} = \frac{P(\neg XY)}{P(\neg X\neg Y)}$$

The odds ratio is then defined as the ratio of these two odds:

$$oddsratio(X \longrightarrow Y) = \frac{odds(Y|X)}{odds(Y|\neg X)} = \frac{P(XY)\cdot P(\neg X\neg Y)}{P(X\neg Y)\cdot P(\neg XY)}$$

$$= \frac{sup(XY)\cdot sup(\neg X\neg Y)}{sup(X\neg Y)\cdot sup(\neg XY)}$$

If X and Y are independent, then odds ratio has value 1.

Odds Ratio

Let us compare the odds ratio for two rules, $C \longrightarrow A$ and $D \longrightarrow A$. The contingency tables for A and C, and for A and D, are given below:

	C	$\neg C$
A	2	2
$\neg A$	2	0

	D	$\neg D$
A	3	1
$\neg A$	1	1

The odds ratio values for the two rules are given as

$$oddsratio(C \longrightarrow A) = \frac{sup(AC) \cdot sup(\neg A \neg C)}{sup(A \neg C) \cdot sup(\neg AC)} = \frac{2 \times 0}{2 \times 2} = 0$$

$$oddsratio(D \longrightarrow A) = \frac{sup(AD) \cdot sup(\neg A \neg D)}{sup(A \neg D) \cdot sup(\neg AD)} = \frac{3 \times 1}{1 \times 1} = 3$$

Thus $D \longrightarrow A$ is stronger than $C \longrightarrow A$, which is also confirmed by lift and confidence.

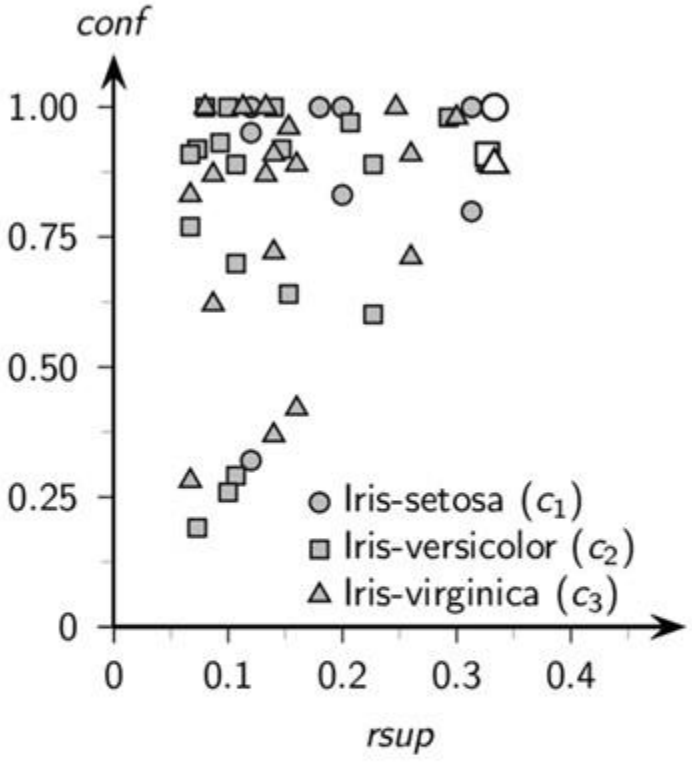
Example: Association Rules from Iris Data

Discretization of Iris Data

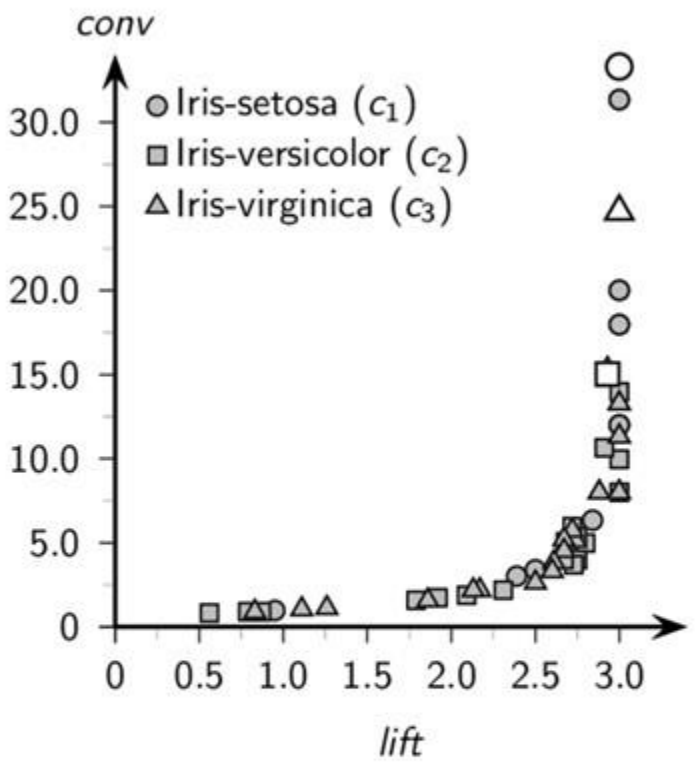
Attribute	Range or value	Label
Sepal length	4.30–5.55	sl_1
	5.55–6.15	sl_2
	6.15–7.90	sl_3
Sepal width	2.00–2.95	sw_1
	2.95–3.35	sw_2
	3.35–4.40	sw_3
Petal length	1.00–2.45	pl_1
	2.45–4.75	pl_2
	4.75–6.90	pl_3
Petal width	0.10–0.80	pw_1
	0.80–1.75	pw_2
	1.75–2.50	pw_3
Class	Iris-setosa	c_1
	Iris-versicolor	c_2
	Iris-virginica	c_3

Iris: Support vs. Confidence, and Conviction vs. Lift

minsup = 10 and *minlift* = 0.1 results in 79 rules



(a) Support vs. confidence



(b) Lift vs. conviction

For each class we select the most specific (i.e., with maximal antecedent) rule with the highest relative support and then confidence, and also those with the highest conviction and then lift.

Iris Data: Best Class-specific Rules

Best Rules by Support and Confidence

Rule	*rsup*	*conf*	*lift*	*conv*
$\{pl_1, pw_1\} \longrightarrow c_1$	0.333	1.00	3.00	∞
$pw_2 \longrightarrow c_2$	0.327	0.91	2.72	6.00
$pl_3 \longrightarrow c_3$	0.327	0.89	2.67	5.24

Best Rules by Lift and Conviction

Rule	*rsup*	*conf*	*lift*	*conv*
$\{pl_1, pw_1\} \longrightarrow c_1$	0.33	1.00	3.00	∞
$\{pl_2, pw_2\} \longrightarrow c_2$	0.29	0.98	2.93	15.00
$\{sl_3, pl_3, pw_3\} \longrightarrow c_3$	0.25	1.00	3.00	∞

Comparing the rules for each criterion, we verify that the best rule for c_1 is the same, but the comparison between rules for c_2 and c_3 suggests a trade-off between support and novelty, represented by lift and conviction.

Pattern Assessment Measures: Support and Lift

Support: The most basic measures are support and relative support, giving the number and fraction of transactions in $\boldsymbol{D}$ that contain the itemset X:

$$sup(X) = |\boldsymbol{t}(X)| \qquad rsup(X) = \frac{sup(X)}{|\boldsymbol{D}|}$$

Lift: The *lift* of a k-itemset $X = \{x_1, x_2, \ldots, x_k\}$ is defined as

$$lift(X, \boldsymbol{D}) = \frac{P(X)}{\prod_{i=1}^{k} P(x_i)} = \frac{rsup(X)}{\prod_{i=1}^{k} rsup(x_i)}$$

Generalized Lift: Assume that $\{X_1, X_2, \ldots, X_q\}$ is a q-partition of X, i.e., a partitioning of X into q nonempty and disjoint itemsets X_i. Define the generalized lift of X over partitions of size q as follows:

$$lift_q(X) = \min_{X_1, \ldots, X_q} \left\{ \frac{P(X)}{\prod_{i=1}^{q} P(X_i)} \right\}$$

This is, the least value of lift over all q-partitions X.

Pattern Assessment Measures: Rule-based Measures

Let Θ be some rule assessment measure. We generate all possible rules from X of the form $X_1 \longrightarrow X_2$ and $X_2 \longrightarrow X_1$, where the set $\{X_1, X_2\}$ is a 2-partition, or a bipartition, of X.

We then compute the measure Θ for each such rule, and use summary statistics such as the mean, maximum, and minimum to characterize X.

For example, if Θ is rule lift, then we can define the average, maximum, and minimum lift values for X as follows:

$$AvgLift(X) = \operatorname*{avg}_{X_1, X_2} \Big\{ lift(X_1 \longrightarrow X_2) \Big\}$$

$$MaxLift(X) = \max_{X_1, X_2} \Big\{ lift(X_1 \longrightarrow X_2) \Big\}$$

$$MinLift(X) = \min_{X_1, X_2} \Big\{ lift(X_1 \longrightarrow X_2) \Big\}$$

Iris Data: Support Values for $\{pl_2, pw_2, c_2\}$ and its Subsets

Consider the support and relative support of itemset $X = \{pl_2, pw_2, c_2\}$ and its subsets.

Itemset	*sup*	*rsup*
$\{pl_2, pw_2, c_2\}$	44	0.293
$\{pl_2, pw_2\}$	45	0.300
$\{pl_2, c_2\}$	44	0.293
$\{pw_2, c_2\}$	49	0.327
$\{pl_2\}$	45	0.300
$\{pw_2\}$	54	0.360
$\{c_2\}$	50	0.333

$$lift(X) = \frac{rsup(X)}{rsup(pl_2)rsup(pw_2)rsup(c_2)} = \frac{0.293}{0.3 * 0.36 * 0.333} = 8.16$$

Rules Generated from $X = \{pl_2, pw_2, c_2\}$

Consider all rules that may be generated from X:

Bipartition	Rule	*lift*	*leverage*	*conf*
$\{\{pl_2\}, \{pw_2, c_2\}\}$	$pl_2 \longrightarrow \{pw_2, c_2\}$	2.993	0.195	0.978
	$\{pw_2, c_2\} \longrightarrow pl_2$	2.993	0.195	0.898
$\{\{pw_2\}, \{pl_2, c_2\}\}$	$pw_2 \longrightarrow \{pl_2, c_2\}$	2.778	0.188	0.815
	$\{pl_2, c_2\} \longrightarrow pw_2$	2.778	0.188	1.000
$\{\{c_2\}, \{pl_2, pw_2\}\}$	$c_2 \longrightarrow \{pl_2, pw_2\}$	2.933	0.193	0.880
	$\{pl_2, pw_2\} \longrightarrow c_2$	2.933	0.193	0.978

We may then calculate $AvgLift(X)$:

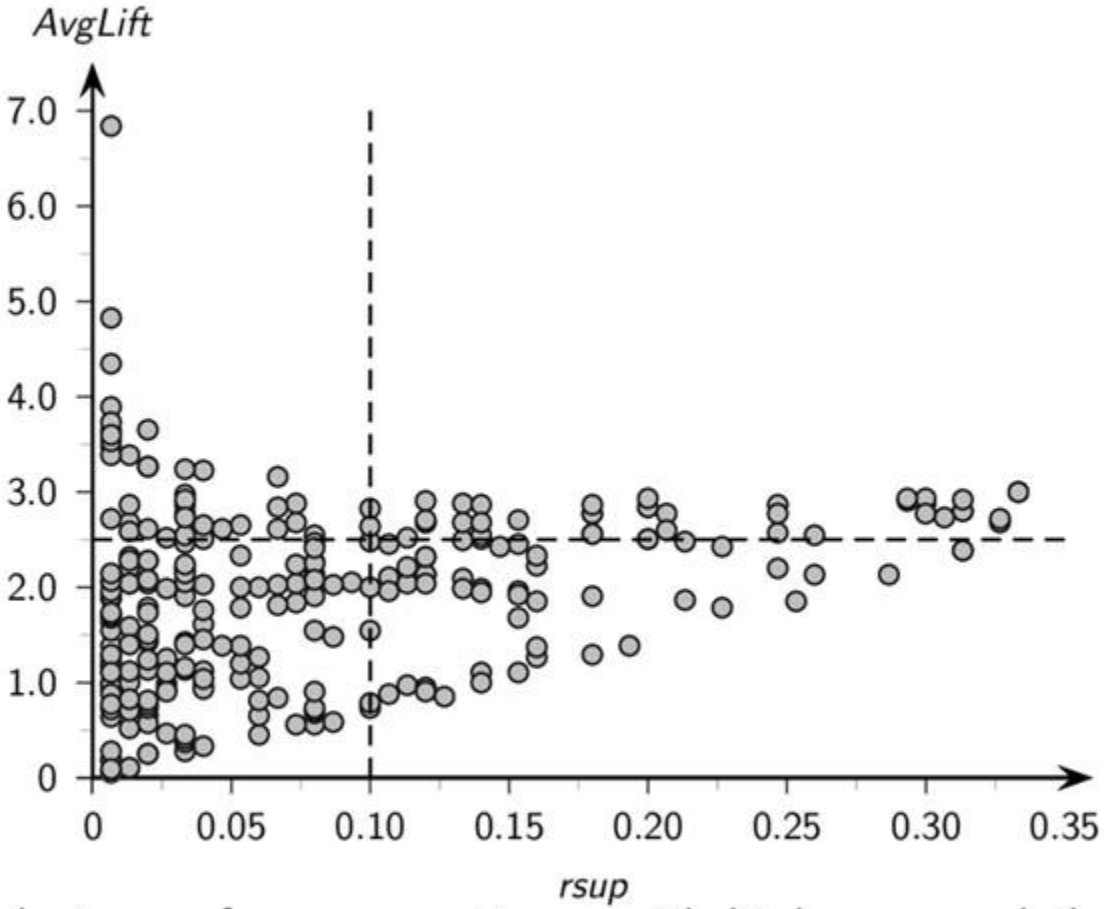
$AvgLift(X) = avg\{2.993, 2.778, 2.933\} = 2.901$

And also $AvgConf(X)$:

$AvgConf(X) = avg\{0.978, 0.898, 0.815, 1.0, 0.88, 0.978\} = 0.925$

Iris: Relative Support and Average Lift of Patterns

306 frequent itemsets with $minsup = 1$ and $k \geq 2$



For sake of analysis, we focus on patterns with high *rsup* and then high *AvgLift*, such as $X = \{pl_1, pw_1, c_1\}$.

Comparing Itemsets: Maximal Itemsets

An frequent itemset X is *maximal* if all of its supersets are not frequent, that is, X is maximal iff

$$sup(X) \geq minsup, \text{ and for all } Y \supset X, sup(Y) < minsup$$

Given a collection of frequent itemsets, we may choose to retain only the maximal ones, especially among those that already satisfy some other constraints on pattern assessment measures like lift or leverage.

Iris: Maximal Patterns for Average Lift

We focus on the 37 class-specific itemsets that present $rsup > 0.1$ and $AvgLift > 2.5$ and select the maximal ones:

Pattern	Avg. lift
$\{sl_1, sw_2, pl_1, pw_1, c_1\}$	2.90
$\{sl_1, sw_3, pl_1, pw_1, c_1\}$	2.86
$\{sl_2, sw_1, pl_2, pw_2, c_2\}$	2.83
$\{sl_3, sw_2, pl_3, pw_3, c_3\}$	2.88
$\{sw_1, pl_3, pw_3, c_3\}$	2.52

For instance, for c_1, the essential items are sl_1, pl_1, pw_1 and either sw_2 or sw_3.

Closed Itemsets and Minimal Generators

An itemset X is *closed* if all of its supersets have strictly less support, that is,

$$sup(X) > sup(Y), \text{ for all } Y \supset X$$

An itemset X is a *minimal generator* if all its subsets have strictly higher support, that is,

$$sup(X) < sup(Y), \text{ for all } Y \subset X$$

If an itemset X is not a minimal generator, then it implies that it has some redundant items, that is, we can find some subset $Y \subset X$, which can be replaced with an even smaller subset $W \subset Y$ without changing the support of X, that is, there exists a $W \subset Y$, such that

$$sup(X) = sup(Y \cup (X \setminus Y)) = sup(W \cup (X \setminus Y))$$

One can show that all subsets of a minimal generator must themselves be minimal generators.

Closed Itemsets and Minimal Generators

The support of an itemset X is

- the maximum support among all closed itemsets that contain X.
- the minimum support among all minimal generators that are subsets of X.

sup	Closed Itemset	Minimal Generators
3	*ABDE*	*AD*, *DE*
3	*BCE*	*CE*
4	*ABE*	*A*
4	*BC*	*C*
4	*BD*	*D*
5	*BE*	*E*
6	*B*	*B*

Consider itemset AE:

$$sup(AE) = max\{sup(ABE), sup(ABDE)\} = 4$$

$$sup(AE) = min\{sup(A), sup(E)\} = 4$$

Comparing Itemsets: Productive Itemsets

An itemset X is *productive* if its relative support is higher than the expected relative support over all of its bipartitions, assuming they are independent. More formally, let $|X| \geq 2$, and let $\{X_1, X_2\}$ be a bipartition of X. We say that X is productive provided

$$rsup(X) > rsup(X_1) \times rsup(X_2), \text{ for all bipartitions } \{X_1, X_2\} \text{ of } X$$

This immediately implies that X is productive if its minimum lift is greater than one, as

$$MinLift(X) = \min_{X_1, X_2} \left\{ \frac{rsup(X)}{rsup(X_1) \cdot rsup(X_2)} \right\} > 1$$

In terms of leverage, X is productive if its minimum leverage is above zero because

$$MinLeverage(X) = \min_{X_1, X_2} \left\{ rsup(X) - rsup(X_1) \times rsup(X_2) \right\} > 0$$

Comparing Itemsets: Productive Itemsets

$ABDE$ is not productive because there is at least a bipartition with $lift = 1$. For instance, the bipartition $\{B, ADE\}$:

$$lift(B \longrightarrow ADE) = \frac{rsup(ABDE)}{rsup(B) \cdot rsup(ADE)} = \frac{3/6}{6/6 \cdot 3/6} = 1$$

ADE, on the other hand, is productive:

$$lift(A \longrightarrow DE) = \frac{rsup(ADE)}{rsup(A) \cdot rsup(DE)} = \frac{3/6}{4/6 \cdot 3/6} = 1.5$$

$$lift(D \longrightarrow AE) = \frac{rsup(ADE)}{rsup(D) \cdot rsup(AE)} = \frac{3/6}{4/6 \cdot 4/6} = 1.125$$

$$lift(E \longrightarrow AD) = \frac{rsup(ADE)}{rsup(E) \cdot rsup(AD)} = \frac{3/6}{5/6 \cdot 3/6} = 1.2$$

Comparing Rules

Given two rules $R : X \longrightarrow Y$ and $R' : W \longrightarrow Y$ that have the same consequent, we say that R is *more specific* than R', or equivalently, that R' is *more general* than R provided $W \subset X$.

Nonredundant Rules: We say that a rule $R : X \longrightarrow Y$ is *redundant* provided there exists a more general rule $R' : W \longrightarrow Y$ that has the same support, that is, $W \subset X$ and $sup(R) = sup(R')$.

Improvement and Productive Rules: Define the *improvement* of a rule $X \longrightarrow Y$ as follows:

$$imp(X \longrightarrow Y) = conf(X \longrightarrow Y) - \max_{W \subset X} \Big\{ conf(W \longrightarrow Y) \Big\}$$

A rule $R : X \longrightarrow Y$ is *productive* if its improvement is greater than zero, which implies that for all more general rules $R' : W \longrightarrow Y$ we have $conf(R) > conf(R')$.

Comparing Rules

Consider rule $R : BE \longrightarrow C$, which has support 3, and confidence $3/5 = 0.60$.
It has two generalizations, namely

$$R_1' : E \longrightarrow C, \quad sup = 3, conf = 3/5 = 0.6$$
$$R_2' : B \longrightarrow C, \quad sup = 4, conf = 4/6 = 0.67$$

Thus, $BE \longrightarrow C$ is redundant w.r.t. $E \longrightarrow C$ because they have the same support, that is, $sup(BCE) = sup(BC)$.

$BE \longrightarrow C$ is also unproductive, since

$imp(BE \longrightarrow C) = 0.6 - \max\{0.6, 0.67\} = -0.07.$

It has a more general rule, namely R_2', with higher confidence.

Fisher Exact Test for Productive Rules

Let $R : X \longrightarrow Y$ be an association rule. Consider its generalization $R' : W \longrightarrow Y$, where $W = X \setminus Z$ is the new antecedent formed by removing from X the subset $Z \subseteq X$.

Given an input dataset $\boldsymbol{D}$, conditional on the fact that W occurs, we can create a 2×2 contingency table between Z and the consequent Y

W	Y	$\neg Y$	
Z	a	b	$a+b$
$\neg Z$	c	d	$c+d$
	$a+c$	$b+d$	$n = sup(W)$

where

$$a = sup(WZY) = sup(XY) \qquad b = sup(WZ\neg Y) = sup(X\neg Y)$$
$$c = sup(W\neg ZY) \qquad d = sup(W\neg Z\neg Y)$$

Fisher Exact Test for Productive Rules

Given a contingency table conditional on W, we are interested in the odds ratio obtained by comparing the presence and absence of Z, that is,

$$oddsratio = \frac{a/(a+b)}{b/(a+b)} \Big/ \frac{c/(c+d)}{d/(c+d)} = \frac{ad}{bc}$$

Under the null hypothesis H_0 that Z and Y are independent given W the odds ratio is 1. If we further assume that the row and column marginals are fixed, then a uniquely determines the other three values b, c, and d, and the probability mass function of observing the value a in the contingency table is given by the hypergeometric distribution.

$$P\Big(a \big| (a+c),(a+b),n\Big) = \frac{(a+b)!\,(c+d)!\,(a+c)!\,(b+d)!}{n!\,a!\,b!\,c!\,d!}$$

Fisher Exact Test: P-value

Our aim is to contrast the null hypothesis H_0 that $oddsratio = 1$ with the alternative hypothesis H_a that $oddsratio > 1$.
The *p-value* for a is given as

$$p\text{-}value(a) = \sum_{i=0}^{\min(b,c)} P(a+i \mid (a+c),(a+b),n)$$

$$= \sum_{i=0}^{\min(b,c)} \frac{(a+b)!\,(c+d)!\,(a+c)!\,(b+d)!}{n!\,(a+i)!\,(b-i)!\,(c-i)!\,(d+i)!}$$

which follows from the fact that when we increase the count of a by i, then because the row and column marginals are fixed, b and c must decrease by i, and d must increase by i, as shown in the table below:

W	Y	$\neg Y$	
Z	$a+i$	$b-i$	$a+b$
$\neg Z$	$c-i$	$d+i$	$c+d$
	$a+c$	$b+d$	$n = sup(W)$

Fisher Exact Test: Example

Consider the rule $R: pw_2 \longrightarrow c_2$ obtained from the discretized Iris dataset. To test if it is productive, because there is only a single item in the antecedent, we compare it only with the default rule $\emptyset \longrightarrow c_2$. We have

$$a = sup(pw_2, c_2) = 49 \qquad b = sup(pw_2, \neg c_2) = 5$$
$$c = sup(\neg pw_2, c_2) = 1 \qquad d = sup(\neg pw_2, \neg c_2) = 95$$

with the contingency table given as

	c_2	$\neg c_2$	
pw_2	49	5	54
$\neg pw_2$	1	95	96
	50	100	150

Thus the *p-value* is given as
$p\text{-}value = \sum_{i=0}^{\min(b,c)} P(a+i \mid (a+c),(a+b),n) = 1.51 \times 10^{-32}$

Since the *p-value* is extremely small, we can safely reject the null hypothesis that the odds ratio is 1. Instead, there is a strong relationship between $X = pw_2$ and $Y = c_2$, and we conclude that $R: pw_2 \longrightarrow c_2$ is a productive rule.

Fisher Exact Test: Example

Consider another rule $\{sw_1, pw_2\} \longrightarrow c_2$, with $X = \{sw_1, pw_2\}$ and $Y = c_2$. Consider its three generalizations, and the corresponding contingency tables and p-values:

$R_1' : pw_2 \longrightarrow c_2$

$Z = \{sw_1\}$

$W = X \setminus Z = \{pw_2\}$

$p\text{-}value = 0.84$

$W = pw_2$	c_2	$\neg c_2$	
sw_1	34	4	38
$\neg sw_1$	15	1	16
	49	5	54

$R_2' : sw_1 \longrightarrow c_2$

$Z = \{pw_2\}$

$W = X \setminus Z = \{sw_1\}$

$p\text{-}value = 1.39 \times 10^{-11}$

$W = sw_1$	c_2	$\neg c_2$	
pw_2	34	4	38
$\neg pw_2$	0	19	19
	34	23	57

Fisher Exact Test: Example

$R'_3 : \emptyset \longrightarrow c_2$

$Z = \{sw_1, pw_2\}$

$W = X \setminus Z = \emptyset$

$p\text{-}value = 3.55 \times 10^{-17}$

$W = \emptyset$	c_2	$\neg c_2$	
$\{sw_1, pw_2\}$	34	4	38
$\neg\{sw_1, pw_2\}$	16	96	112
	50	100	150

We can see that whereas the *p-value* with respect to R'_2 and R'_3 is small, for R'_1 we have $p\text{-}value = 0.84$, which is too high and thus we cannot reject the null hypothesis. We conclude that $R : \{sw_1, pw_2\} \longrightarrow c_2$ is not productive. In fact, its generalization R'_1 is the one that is productive.

Multiple Hypothesis Testing

There can be an exponentially large number of rules that need to be tested to check whether they are productive or not.

Multiple hypothesis testing problem: The sheer number of hypothesis tests leads to some unproductive rules passing the *p-value* $\leq \alpha$ threshold by random chance.

Bonferroni correction: takes into account the number of experiments performed during the hypothesis testing process.

$$\alpha' = \frac{\alpha}{\#r}$$

where $\#r$ is the number of rules to be tested or its estimate.

The rule false discovery rate becomes bounded by α, where a false discovery is to claim that a rule is productive when it is not.

Multiple Hypothesis Testing

Given the class-specific rules of discretized Iris dataset, the maximum number of class-specific rules is given as

$$\#r = c \times \left(\sum_{i=1}^{4} \binom{4}{i} b^i \right)$$

where c is the number of Iris classes, b is the maximum number of bins for any other attribute, i is the antecedent size, and there are b^i possible combinations for the chosen set of i attributes.

Since $c = 3$ and $b = 3$, the number of possible rules is:

$$\#r = 3 \times \left(\sum_{i=1}^{4} \binom{4}{i} 3^i \right) = 3(12 + 54 + 108 + 81) = 3 \cdot 255 = 765$$

Given $\alpha = 0.01$, $\alpha' = \alpha / \#r = 0.01/765 = 1.31 \times 10^{-5}$.

The rule $pw_2 \longrightarrow c_2$ has $p\text{-}value = 1.51 \times 10^{-32}$, and thus it remains productive even when we use α'.

Permutation Test for Significance: Swap Randomization

A *permutation* or *randomization* test determines the distribution of a given test statistic Θ by randomly modifying the observed data several times to obtain a random sample of datasets, which can in turn be used for significance testing.

The *swap randomization* approach maintains as invariant the column and row margins for a given dataset, that is, the permuted datasets preserve the support of each item (the column margin) as well as the number of items in each transaction (the row margin).

Given a dataset $\boldsymbol{D}$, we randomly create k datasets that have the same row and column margins. We then mine frequent patterns in $\boldsymbol{D}$ and check whether the pattern statistics are different from those obtained using the randomized datasets. If the differences are not significant, we may conclude that the patterns arise solely from the row and column margins, and not from any interesting properties of the data.

Swap Randomization

Given a binary matrix $\boldsymbol{D} \subseteq \mathcal{T} \times \mathcal{I}$, the swap randomization method exchanges two nonzero cells of the matrix via a *swap* that leaves the row and column margins unchanged.

Consider any two transactions $t_a, t_b \in \mathcal{T}$ and any two items $i_a, i_b \in \mathcal{I}$ such that $(t_a, i_a), (t_b, i_b) \in \boldsymbol{D}$ and $(t_a, i_b), (t_b, i_a) \notin \boldsymbol{D}$, which corresponds to the 2×2 submatrix in $\boldsymbol{D}$, given as

$$\boldsymbol{D}(t_a, i_a; t_b, i_b) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

After a swap operation we obtain the new submatrix

$$\boldsymbol{D}(t_a, i_b; t_b, i_a) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

where we exchange the elements in $\boldsymbol{D}$ so that $(t_a, i_b), (t_b, i_a) \in \boldsymbol{D}$, and $(t_a, i_a), (t_b, i_b) \notin \boldsymbol{D}$. We denote this operation as $\mathit{Swap}(t_a, i_a; t_b, i_b)$.

Algorithm SwapRandomization

SwapRandomization(t, $\boldsymbol{D} \subseteq \mathcal{T} \times \mathcal{I}$):

1 **while** $t > 0$ **do**
2 Select pairs $(t_a, i_a), (t_b, i_b) \in \boldsymbol{D}$ randomly
3 **if** $(t_a, i_b) \notin \boldsymbol{D}$ *and* $(t_b, i_a) \notin \boldsymbol{D}$ **then**
4 $\boldsymbol{D} \leftarrow \boldsymbol{D} \setminus \{(t_a, i_a), (t_b, i_b)\} \cup \{(t_a, i_b), (t_b, i_a)\}$
5 $t = t - 1$
6 **return** $\boldsymbol{D}$

Swap Randomization Example

Tid	Items					Sum
	A	*B*	*C*	*D*	*E*	
1	1	1	0	1	1	4
2	0	1	1	0	1	3
3	1	1	0	1	1	4
4	1	1	1	0	1	4
5	1	1	1	1	1	5
6	0	1	1	1	0	3
Sum	4	6	4	4	5	

(a) Input binary data $\boldsymbol{D}$

Tid	Items					Sum
	A	*B*	*C*	*D*	*E*	
1	1	1	1	0	1	4
2	0	1	1	0	1	3
3	1	1	0	1	1	4
4	1	1	0	1	1	4
5	1	1	1	1	1	5
6	0	1	1	1	0	3
Sum	4	6	4	4	5	

(b) $Swap(1, D; 4, C)$

Tid	Items					Sum
	A	*B*	*C*	*D*	*E*	
1	1	1	1	0	1	4
2	1	1	0	0	1	3
3	1	1	0	1	1	4
4	0	1	1	1	1	4
5	1	1	1	1	1	5
6	0	1	1	1	0	3
Sum	4	6	4	4	5	

(c) $Swap(2, C; 4, A)$

Swap Randomization Example

We generated $k = 100$ swap randomized datasets (150 swaps).

Let the test statistic be the total number of frequent itemsets using $minsup = 3$. For $\boldsymbol{D}$, we have $|\mathcal{F}| = 19$, and for the $k = 100$ permuted datasets we find:

$$P\big(|\mathcal{F}| = 19\big) = 0.67 \qquad P\big(|\mathcal{F}| = 17\big) = 0.33$$

Because $p\text{-}value(19) = 0.67$, we may conclude that the set of frequent itemsets is essentially determined by the row and column marginals.

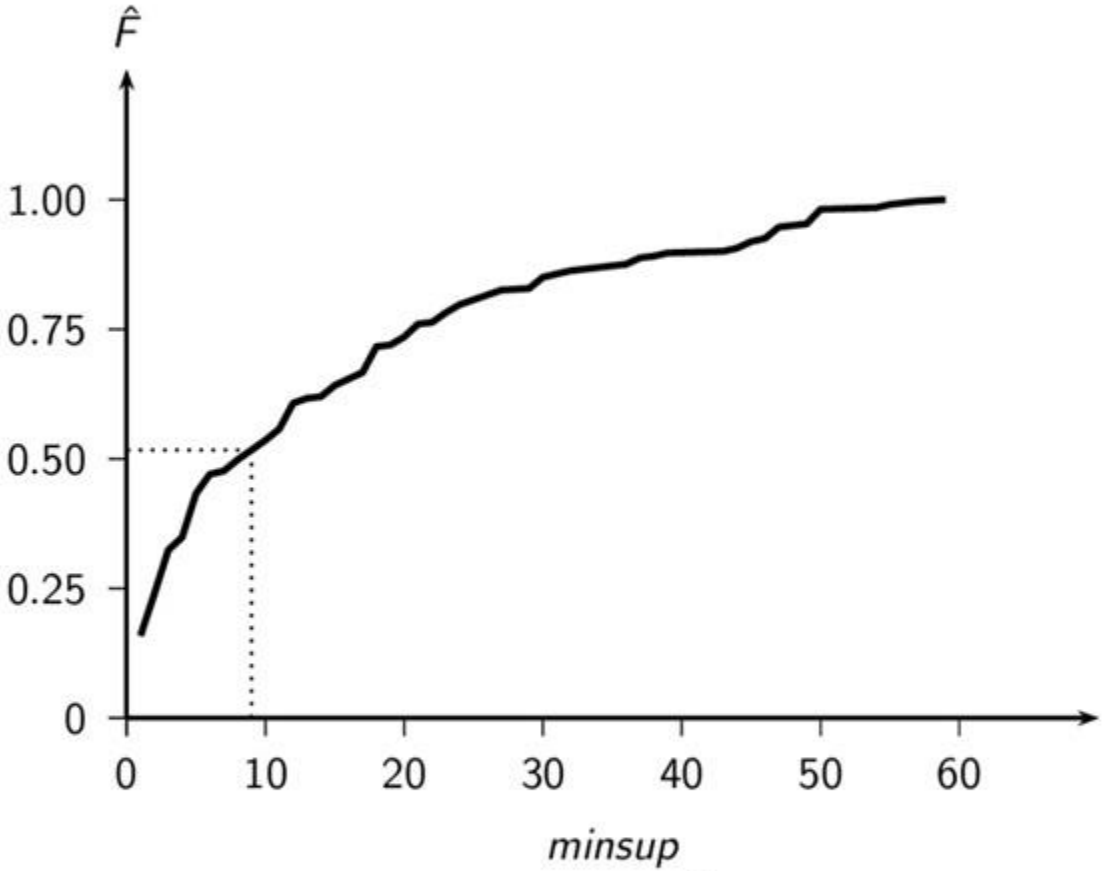
Consider $ABDE$, where $sup(ABDE) = 3$ and the probability that $ABDE$ is frequent is $17/100 = 0.17$. As this probability is not very low, $ABDE$ is not a statistically significant pattern.

Consider BCD, where $sup(BCD) = 2$. The empirical PMF is given as

$$P(sup = 2) = 0.54 \qquad P(sup = 3) = 0.44 \qquad P(sup = 4) = 0.02$$

Since 54% indicates BCD is infrequent, we may assume it.

CDF for Number of Frequent Itemsets: Iris



We choose $minsup = 10$, for which we have $\hat{F}(10) = P(sup < 10) = 0.517$, that is, 48.3% of the itemsets that occur at least once are frequent.

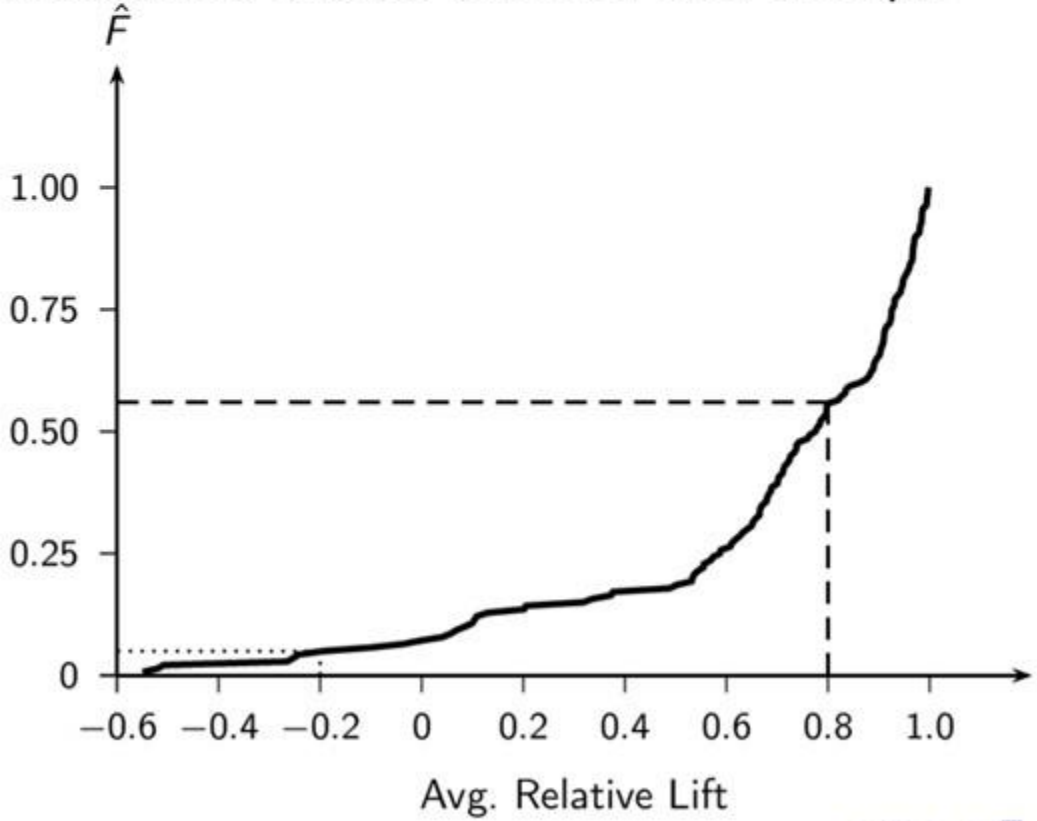
CDF for Average Relative Lift: Iris

$k = 100$ swap randomization steps, 140 frequent itemsets

The relative lift statistic is

$$rlift(X, \boldsymbol{D}, \boldsymbol{D}_i) = \frac{sup(X, \boldsymbol{D}) - sup(X, \boldsymbol{D}_i)}{sup(X, \boldsymbol{D})} = 1 - \frac{sup(X, \boldsymbol{D}_i)}{sup(X, \boldsymbol{D})}$$

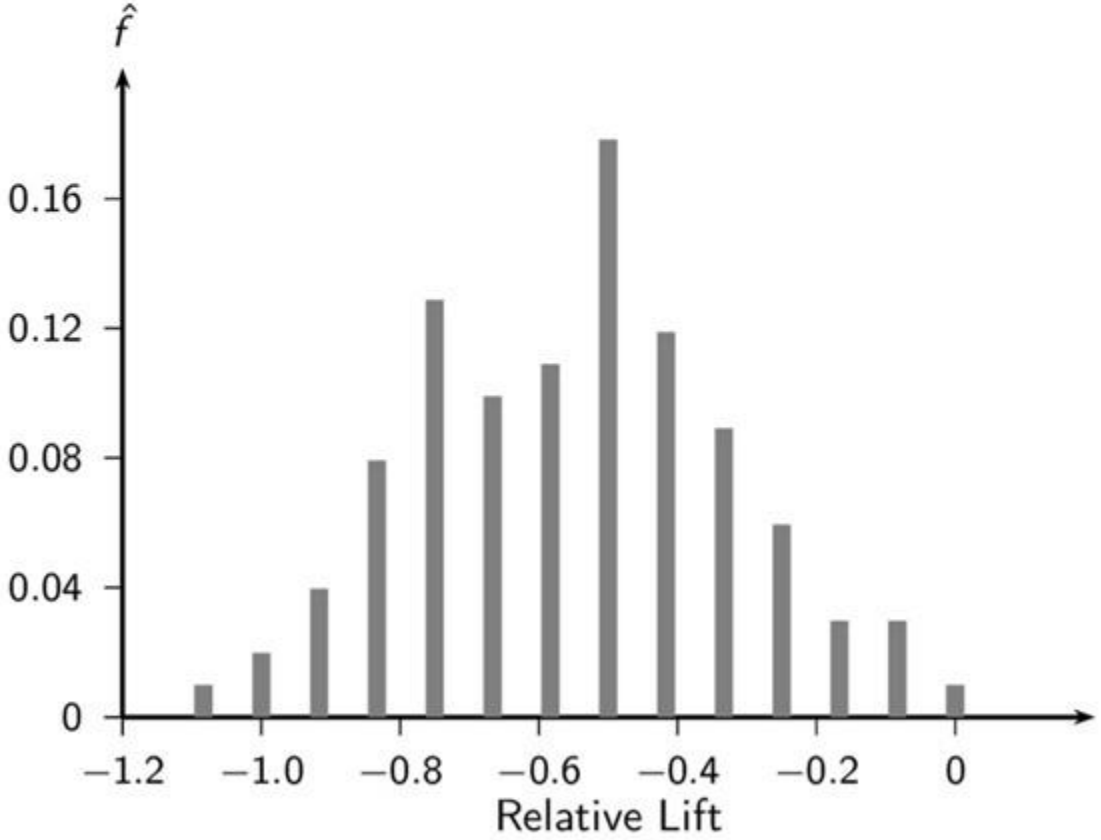
$\boldsymbol{D}_i$ is ith swap randomized dataset obtained after k steps.



PMF for Relative Lift: $\{sl_1, pw_2\}$

$k = 100$ swap randomization steps

Its average relative lift value is -0.55, and $p\text{-}value(-0.2) = 0.069$, which indicates that the itemset is likely to be disassociative.



Bootstrap Sampling for Confidence Interval

We can generate k bootstrap samples from $\boldsymbol{D}$ using sampling *with replacement*. Given pattern X or rule $R : X \longrightarrow Y$, we can obtain the value of the test statistic in each of the bootstrap samples; let θ_i denote the value in sample $\boldsymbol{D}_i$.

From these values we can generate the empirical cumulative distribution function for the statistic

$$\hat{F}(x) = \hat{P}(\Theta \leq x) = \frac{1}{k} \sum_{i=1}^{k} I(\theta_i \leq x)$$

where I is an indicator variable that takes on the value 1 when its argument is true, and 0 otherwise.

Given a desired confidence level α (e.g., $\alpha = 0.95$) we can compute the interval for the test statistic by discarding values from the tail ends of $\hat{F}$ on both sides that encompass $(1-\alpha)/2$ of the probability mass. In other words, the interval $\left[v_{1-\alpha/2}, v_{\alpha/2}\right]$ encompasses $1-\alpha$ fraction of the probability mass, and therefore it is called the $100(1-\alpha)\%$ confidence interval for the chosen test statistic Θ.

Bootstrap Confidence Interval Algorithm

Bootstrap-ConfidenceInterval(X, α, k, $\boldsymbol{D}$):

1 **for** $i \in [1, k]$ **do**

2 $\quad \boldsymbol{D}_i \leftarrow$ sample of size n with replacement from $\boldsymbol{D}$

3 $\quad \theta_i \leftarrow$ compute test statistic for X on $\boldsymbol{D}_i$

4 $\hat{F}(x) = P(\Theta \leq x) = \frac{1}{k} \sum_{i=1}^{k} I(\theta_i \leq x)$

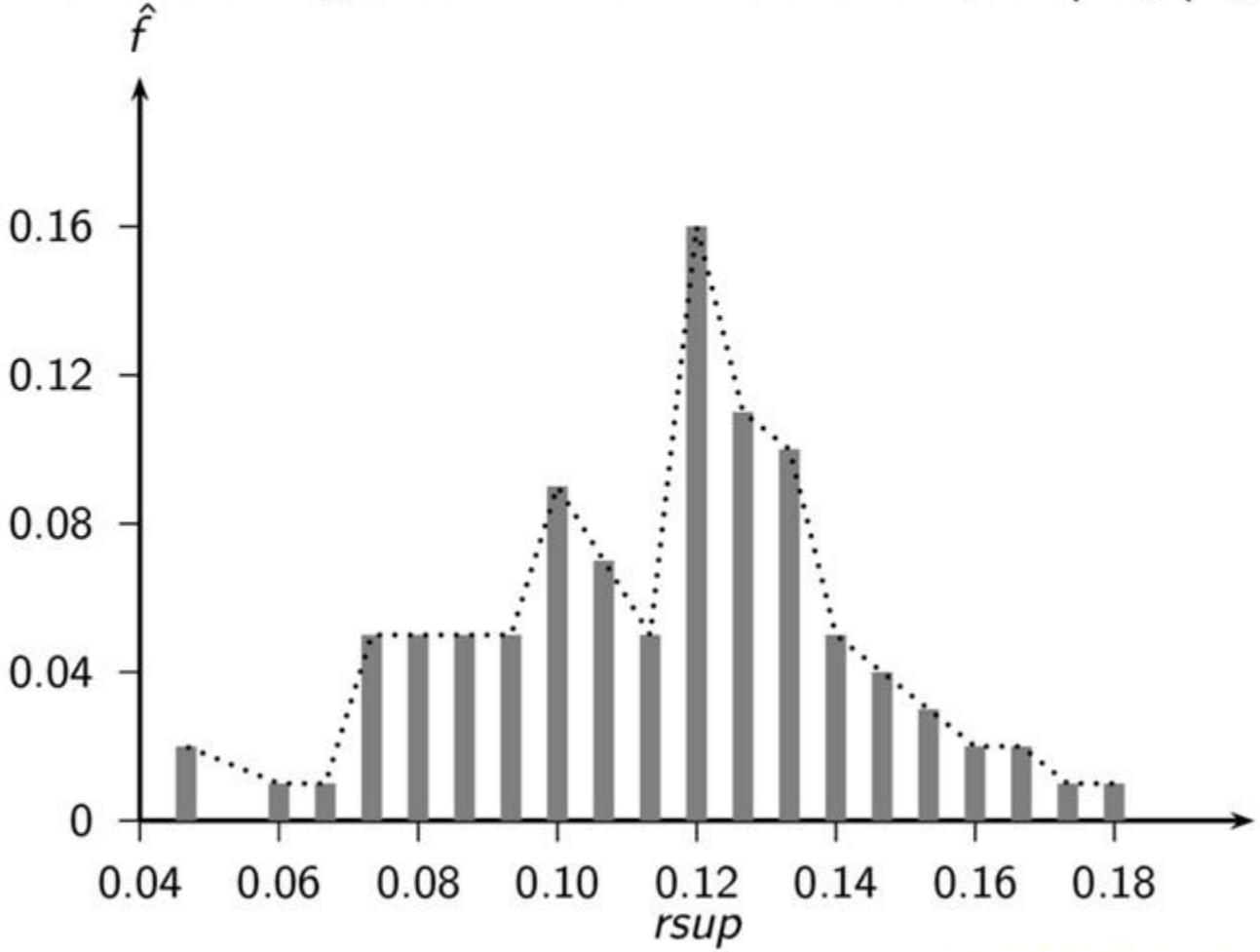
5 $v_{(1-\alpha)/2} = \hat{F}^{-1}((1-\alpha)/2)$

6 $v_{(1+\alpha)/2} = \hat{F}^{-1}((1+\alpha)/2)$

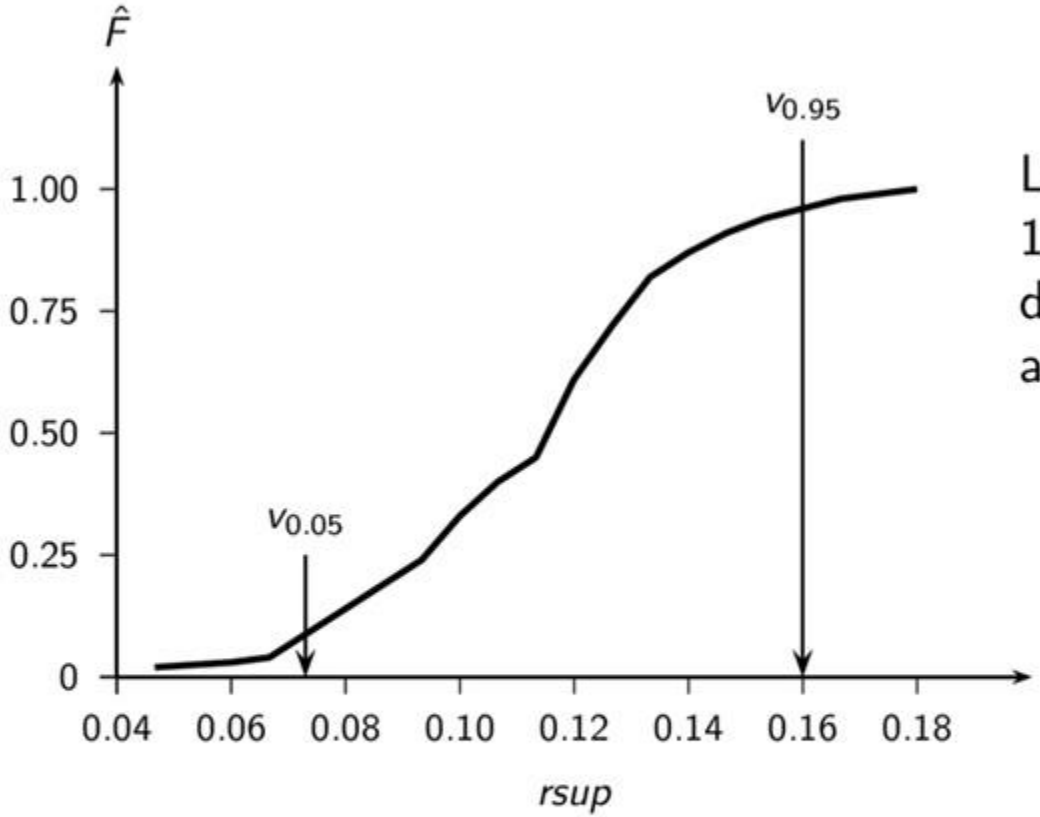
7 **return** $[v_{(1-\alpha)/2}, v_{(1+\alpha)/2}]$

Empirical PMF for RelSupport: $X = \{sw_1, pl_3, pw_3, cl_3\}$

Given $rsup(X, \boldsymbol{D}) = 0.113$ (or $sup(X, \boldsymbol{D}) = 17$) and $k = 100$ bootstrap samples, we compute the relative support of X in each of the samples ($rsup(X, \boldsymbol{D}_i)$).



Empirical CDF for RelSupport: $X = \{sw_1, pl_3, pw_3, cl_3\}$



Let the confidence level be $1 - \alpha = 0.9$, thus $\alpha = 0.1$, discarding the values that account for $\alpha/2 = 0.05$:

$$v_{1-\alpha/2} = v_{0.95} = 0.073$$
$$v_{\alpha/2} = v_{0.05} = 0.16$$

The 90% confidence interval for $rsup(X) \in [0.073, 0.16]$, i.e., $sup \in [11, 24]$.

Note that $rsup(X, \boldsymbol{D}) = 0.113$, with $p\text{-}value(0.113) = 0.45$, and $\mu_{rsup(X)} = 0.115$.

Q&A