Frequent Pattern Mining

Christian Borgelt

Dept. of Artificial Intelligence & Human Interfaces
Paris-Lodron-University of Salzburg
Jakob-Haringer-Straße 2, 5020 Salzburg, Austria

christian.borgelt@plus.ac.at
christian@borgelt.net

http://www.borgelt.net/

https://meet.google.com/scx-waqy-esr

Overview

Frequent Pattern Mining comprises

- Frequent Item Set Mining and Association Rule Induction
- Frequent Sequence Mining
- Frequent Tree Mining
- Frequent Graph Mining

Application Areas of Frequent Pattern Mining include

- Market Basket Analysis
- Click Stream Analysis
- Web Link Analysis
- Genome Analysis
- Drug Design (Molecular Fragment Mining)

Frequent Item Set Mining

Frequent Item Set Mining: Motivation

- Frequent Item Set Mining is a method for **market basket analysis**.

- It aims at finding regularities in the shopping behavior of customers of supermarkets, mail-order companies, on-line shops etc.

- More specifically:
  Find sets of products that are frequently bought together.

- Possible applications of found frequent item sets:

  ○ Improve arrangement of products in shelves, on a catalog's pages etc.

  ○ Support cross-selling (suggestion of other products), product bundling.

  ○ Fraud detection, technical dependence analysis etc.

- Often found patterns are expressed as **association rules**, for example:

  If a customer buys **bread** and **wine**,
  then she/he will probably also buy **cheese**.

Frequent Item Set Mining: Basic Notions

- Let $B = \{i_1, \ldots, i_m\}$ be a set of **items**. This set is called the **item base**.

  Items may be products, special equipment items, service options etc.

- Any subset $I \subseteq B$ is called an **item set**.

  An item set may be any set of products that can be bought (together).

- Let $T = (t_1, \ldots, t_n)$ with $\forall k, 1 \leq k \leq n : t_k \subseteq B$ be a tuple of **transactions** over B. This tuple is called the **transaction database**.

  A transaction database can list, for example, the sets of products bought by the customers of a supermarket in a given period of time.

  Every transaction is an item set, but some item sets may not appear in T.

  Transactions need not be pairwise different: it may be $t_j = t_k$ for $j \neq k$.

  T may also be defined as a *multiset* or *bag* of transactions.

  The item base B may not be given explicitly, but only implicitly as $B = \bigcup_{k=1}^{n} t_k$.

Frequent Item Set Mining: Basic Notions

Let $I \subseteq B$ be an item set and T a transaction database over B.

- A transaction $t \in T$ **covers** the item set I or the item set I is **contained in** a transaction $t \in T$ iff $I \subseteq t$.
- The set $K_T(I) = \{k \in \{1, \ldots, n\} \mid I \subseteq t_k\}$ is called the **cover** of I w.r.t. T.

  The cover of an item set is the index set of the transactions that cover it.

  It may also be defined as a tuple of all transactions that cover it (which, however, is complicated to write in a formally correct way).
- The value $s_T(I) = |K_T(I)|$ is called the **(absolute) support** of I w.r.t. T.

  The value $\sigma_T(I) = \frac{1}{n}|K_T(I)|$ is called the **relative support** of I w.r.t. T.

  The support of I is the number or fraction of transactions that contain it.

  Sometimes $\sigma_T(I)$ is also called the *(relative) frequency* of I w.r.t. T.

Frequent Item Set Mining: Basic Notions

Alternative Definition of Transactions

- A **transaction** over an item base B is a pair $t = (\text{tid}, J)$, where
  - tid is a unique **transaction identifier** and
  - $J \subseteq B$ is an item set.

- A **transaction database** $T = \{t_1, \ldots, t_n\}$ is a *set* of transactions.

  A simple set can be used, because transactions differ at least in their identifier.

- A transaction $t = (\text{tid}, J)$ **covers** an item set I iff $I \subseteq J$.

- The set $K_T(I) = \{\text{tid} \mid \exists J \subseteq B : \exists t \in T : t = (\text{tid}, J) \land I \subseteq J\}$ is the **cover** of I w.r.t. T.

Remark: If the transaction database is defined as a tuple, there is an implicit transaction identifier, namely the position/index of the transaction in the tuple.

Frequent Item Set Mining: Formal Definition

Given:

- a set $B = \{i_1, \dots, i_m\}$ of items, the **item base**,
- a tuple $T = (t_1, \dots, t_n)$ of transactions over B, the **transaction database**,
- a number $s_{\text{min}} \in \mathbb{N}, 1 \leq s_{\text{min}} \leq n,$ or (equivalently)
  a number $\sigma_{\text{min}} \in \mathbb{R}, 0 < \sigma_{\text{min}} \leq 1,$ the **minimum support**.

Desired:

- the set of **frequent item sets**, that is,
  the set $F_T(s_{\text{min}}) = \{I \subseteq B \mid s_T(I) \geq s_{\text{min}}\}$ or (equivalently)
  the set $\Phi_T(\sigma_{\text{min}}) = \{I \subseteq B \mid \sigma_T(I) \geq \sigma_{\text{min}}\}$.

Note that with the relations $s_{\text{min}} = \lceil n\sigma_{\text{min}} \rceil$ and $\sigma_{\text{min}} = \frac{1}{n} s_{\text{min}}$ the two versions can easily be transformed into each other.

Frequent Item Sets: Example

transaction database

1:	$\{a, d, e\}$
2:	$\{b, c, d\}$
3:	$\{a, c, e\}$
4:	$\{a, c, d, e\}$
5:	$\{a, e\}$
6:	$\{a, c, d\}$
7:	$\{b, c\}$
8:	$\{a, c, d, e\}$
9:	$\{b, c, e\}$
10:	$\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7 $\{b\}$: 3 $\{c\}$: 7 $\{d\}$: 6 $\{e\}$: 7	$\{a, c\}$: 4 $\{a, d\}$: 5 $\{a, e\}$: 6 $\{b, c\}$: 3 $\{c, d\}$: 4 $\{c, e\}$: 4 $\{d, e\}$: 4	$\{a, c, d\}$: 3 $\{a, c, e\}$: 3 $\{a, d, e\}$: 4

- In this example, the minimum support is $s_{\min} = 3$ or $\sigma_{\min} = 0.3 = 30\%$.
- There are $2^5 = 32$ possible item sets over $B = \{a, b, c, d, e\}$.
- There are 16 frequent item sets (but only 10 transactions).

Searching for Frequent Item Sets

Properties of the Support of Item Sets

- A **brute force approach** that traverses all possible item sets, determines their support, and discards infrequent item sets is usually **infeasible**:

  The number of possible item sets grows exponentially with the number of items. A typical supermarket offers (tens of) thousands of different products.

- **Idea:** Consider the properties of an item set's cover and support, in particular:

  $$\forall I : \forall J \supseteq I : \quad K_T(J) \subseteq K_T(I).$$

  This property holds, since $\forall t : \forall I : \forall J \supseteq I : \quad J \subseteq t \Rightarrow I \subseteq t$.

  Each additional item is another condition that a transaction has to satisfy. Transactions that do not satisfy this condition are removed from the cover.

- It follows: $\qquad \forall I : \forall J \supseteq I : \quad s_T(J) \leq s_T(I).$

  That is: **If an item set is extended, its support cannot increase.**

  One also says that support is **anti-monotone** or **downward closed**.

Properties of the Support of Item Sets

- From $\forall I : \forall J \supseteq I : s_T(J) \leq s_T(I)$ it follows immediately

  $$\forall s_{\min} : \forall I : \forall J \supseteq I : \quad s_T(I) < s_{\min} \;\Rightarrow\; s_T(J) < s_{\min}.$$

  That is: **No superset of an infrequent item set can be frequent.**

- This property is often referred to as the **Apriori Property**.

  Rationale: Sometimes we can know *a priori*, that is, before checking its support by accessing the given transaction database, that an item set cannot be frequent.

- Of course, the contraposition of this implication also holds:

  $$\forall s_{\min} : \forall I : \forall J \subseteq I : \quad s_T(I) \geq s_{\min} \;\Rightarrow\; s_T(J) \geq s_{\min}.$$

  That is: **All subsets of a frequent item set are frequent.**

- This suggests a compressed representation of the set of frequent item sets (which will be explored later: maximal and closed frequent item sets).

Reminder: Implication and Contraposition

- A formula $\varphi \mathrel{\widehat{=}} p \Rightarrow q$ is called an **implication**.
  (An implication is an if-then-statement; here: "if p, then q.")

- The if-part (formula before "$\Rightarrow$", here: p) is called **antecedent**,
  the then-part (formula after "$\Rightarrow$", here: q) is called **consequent**.

- The formula $\psi \mathrel{\widehat{=}} \neg q \Rightarrow \neg p$ is called the **contraposition** of $\varphi \mathrel{\widehat{=}} p \Rightarrow q$.

- The two formulae φ and ψ are logically equivalent.

Semantic Equivalence

Var.		φ	ψ
p	q	$p \Rightarrow q$	$\neg q \Rightarrow \neg p$
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	1

Syntactic Equivalence

$$\begin{aligned} p \Rightarrow q &= \neg p \vee q && \text{(def. implication)} \\ &= q \vee \neg p && \text{(commutativity)} \\ &= \neg\neg q \vee \neg p && \text{(involution)} \\ &= \neg q \Rightarrow \neg p && \text{(def. implication)} \end{aligned}$$

Reminder: Partially Ordered Sets

- A **partial order** is a binary relation $\leq$ over a set S which satisfies $\forall a, b, c \in S$:
  - $a \leq a$ (reflexivity)
  - $a \leq b \wedge b \leq a \;\Rightarrow\; a = b$ (anti-symmetry)
  - $a \leq b \wedge b \leq c \;\Rightarrow\; a \leq c$ (transitivity)
- A set with a partial order is called a **partially ordered set** (or **poset** for short).
- Let a and b be two distinct elements of a partially ordered set $(S, \leq)$.
  - if $a \leq b$ or $b \leq a$, then a and b are called **comparable**.
  - if neither $a \leq b$ nor $b \leq a$, then a and b are called **incomparable**.
- If all pairs of elements of the underlying set S are comparable, the order $\leq$ is called a **total order** or a **linear order**.
- In a total order the reflexivity axiom is replaced by the stronger axiom:
  - $a \leq b \vee b \leq a$ (totality)

Properties of the Support of Item Sets

Monotonicity in Calculus and Mathematical Analysis

- A function $f : \mathbb{R} \to \mathbb{R}$ is called **monotonically non-decreasing** if $\forall x, y \in \mathbb{R} : \; x \le y \;\Rightarrow\; f(x) \le f(y)$.
- A function $f : \mathbb{R} \to \mathbb{R}$ is called **monotonically non-increasing** if $\forall x, y \in \mathbb{R} : \; x \le y \;\Rightarrow\; f(x) \ge f(y)$.

Monotonicity in Order Theory

- Order theory is concerned with arbitrary (partially) ordered sets. The terms *increasing* and *decreasing* are avoided, because they lose their pictorial motivation as soon as sets are considered that are not totally ordered.
- A function $f : S \to R$, where S and R are two partially ordered sets, is called **monotone** or **order-preserving** if $\forall x, y \in S : \; x \le_S y \Rightarrow f(x) \le_R f(y)$.
- A function $f : S \to R$ is called **anti-monotone** or **order-reversing** if $\forall x, y \in S : \; x \le_S y \Rightarrow f(x) \ge_R f(y)$.
- In this sense the **support** of item sets **is anti-monotone**.

Properties of Frequent Item Sets

- A subset R of a partially ordered set $(S, \leq)$ is called **downward closed** if for any element of the set all smaller elements are also in it:

  $$\forall x \in R: \ \forall y \in S: \quad y \leq x \ \Rightarrow \ y \in R$$

  In this case the subset R is also called a **lower set**.

- The notions of **upward closed** and **upper set** are defined analogously.

- For every $s_{\min}$ the set of frequent item sets $F_T(s_{\min})$ is downward closed w.r.t. the partially ordered set $(2^B, \subseteq)$, where 2^B denotes the powerset of B:

  $$\forall s_{\min}: \ \forall X \in F_T(s_{\min}): \ \forall Y \subseteq B: \quad Y \subseteq X \ \Rightarrow \ Y \in F_T(s_{\min}).$$

- Since the set of frequent item sets is induced by the support function, the notions of **up-** or **downward closed** are transferred to this function:

  Any set of item sets w.r.t. a support threshold $s_{\min}$ is up- or downward closed.

  $F_T(s_{\min}) = \{S \subseteq B \mid s_T(S) \geq s_{\min}\}$ (frequent item sets) is downward closed,
  $G_T(s_{\min}) = \{S \subseteq B \mid s_T(S) < s_{\min}\}$ (infrequent item sets) is upward   closed.

Reminder: Partially Ordered Sets and Hasse Diagrams

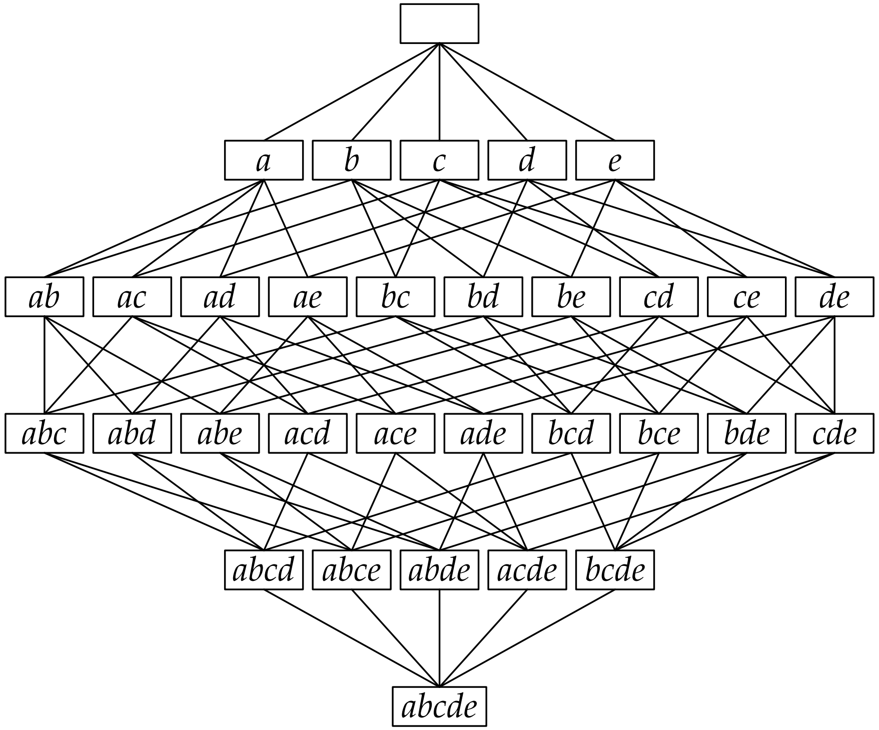
- A finite partially ordered set $(S, \leq)$ can be depicted as a (directed) acyclic graph G, which is called **Hasse diagram**.

- G has the elements of S as vertices. The edges are selected according to:

  If x and y are elements of S with $x < y$ (that is, $x \leq y$ and not $x = y$) and there is no element between x and y (that is, no $z \in S$ with $x < z < y$), then there is an edge from x to y.

- Since the graph is acyclic (there is no directed cycle), the graph can always be depicted such that all edges lead downward.

- The Hasse diagram of a total order (or linear order) is a chain.



Hasse diagram of $(2^{\{a,b,c,d,e\}}, \subseteq)$.

(Edge directions are omitted; all edges lead downward.)

Searching for Frequent Item Sets

- The standard search procedure is an **enumeration approach**, that enumerates candidate item sets and checks their support.

- It improves over the brute force approach by exploiting the **apriori property** to skip item sets that cannot be frequent because they have an infrequent subset.

- The **search space** is the **partially ordered set** $(2^B, \subseteq)$.

- The structure of the partially ordered set $(2^B, \subseteq)$ helps to identify those item sets that can be skipped due to the apriori property.
  $\Rightarrow$ **top-down search** (from empty set/one-element sets to larger sets)

- Since a partially ordered set can conveniently be depicted by a **Hasse diagram**, we will use such diagrams to illustrate the search.

- Note that the search may have to visit an exponential number of item sets.
  In practice, however, the search times are often bearable,
  at least if the minimum support is not chosen too low.

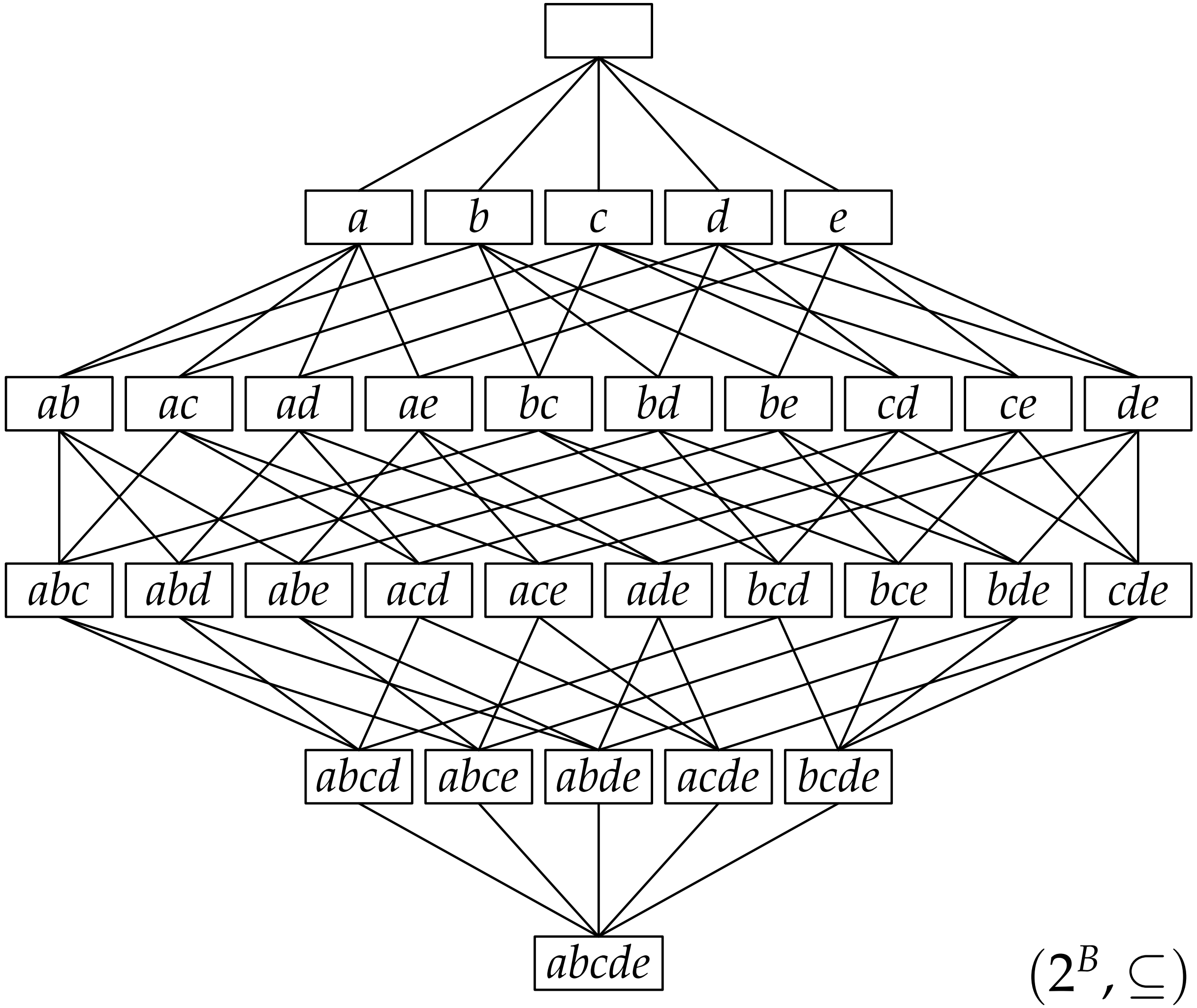
Searching for Frequent Item Sets

Idea: Use the properties of the support to organize the search for all frequent item sets, especially the **apriori property**:

$$\forall I : \forall J \supset I : \quad s_T(I) < s_{\min} \;\Rightarrow\; s_T(J) < s_{\min}.$$

Since these properties relate the support of an item set to the support of its **subsets** and **supersets**, it is reasonable to organize the search based on the structure of the **partially ordered set** $(2^B, \subseteq)$.

Hasse diagram for five items $\{a, b, c, d, e\} = B$:



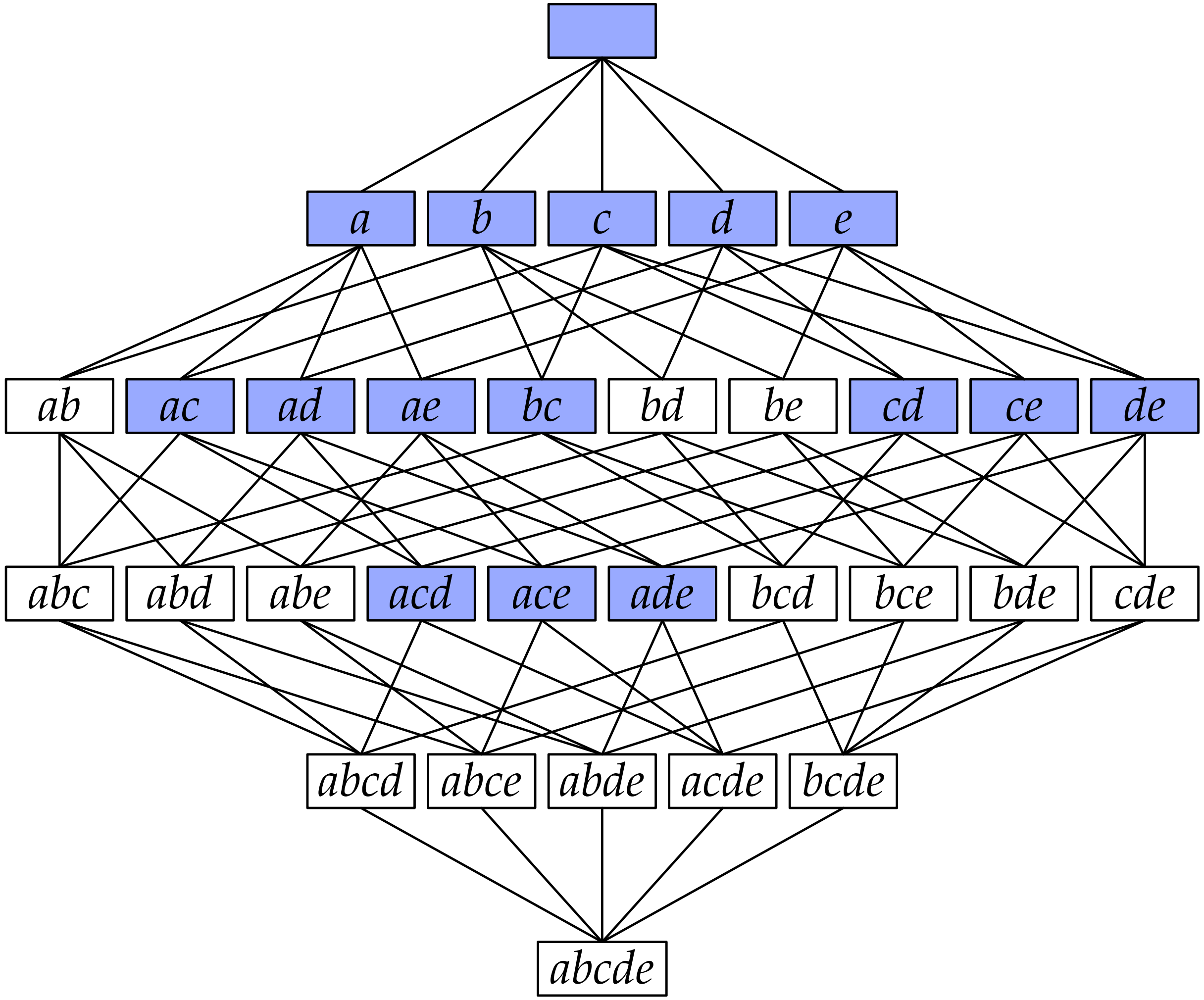
$(2^B, \subseteq)$

Hasse Diagrams and Frequent Item Sets

transaction database

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

Hasse diagram with frequent item sets ($s_{\min} = 3$):



Blue boxes are frequent item sets, white boxes infrequent item sets.

The Apriori Algorithm

[Agrawal and Srikant 1994]

Searching for Frequent Item Sets

Possible scheme for the search:

- Determine the support of the one-element item sets (a.k.a. singletons) and discard the infrequent items / item sets.
- Form candidate item sets with 2 items (both items must be frequent), determine their support, and discard the infrequent item sets.
- Form candidate item sets with 3 items (all contained pairs must be frequent), determine their support, and discard the infrequent item sets.
- Continue by forming candidate item sets with 4, 5, 6 etc. items until no candidate item set is frequent.

This is the general scheme of the **Apriori Algorithm**.

It is based on two main steps: **candidate generation** and **pruning**.

All enumeration algorithms are based on these two steps in some form.

The Apriori Algorithm 1

```
function apriori $(B, T, s_{\min})$
begin                                                 (∗ — Apriori algorithm ∗)
    $k \quad := 1;$                                   (∗ initialize the item set size ∗)
    $E_k := \bigcup_{i \in B} \{\{i\}\};$             (∗ start with single element sets ∗)
    $F_k := \text{prune}(E_k, T, s_{\min});$          (∗ and determine the frequent ones ∗)
    while $F_k \neq \emptyset$ do begin               (∗ while there are frequent item sets ∗)
        $E_{k+1} := \text{candidates}(F_k);$          (∗ create candidates with one item more ∗)
        $F_{k+1} := \text{prune}(E_{k+1}, T, s_{\min});$  (∗ and determine the frequent item sets ∗)
        $k \quad := k + 1;$                           (∗ increment the item counter ∗)
    end;
    return $\bigcup_{j=1}^{k} F_j;$                   (∗ return the frequent item sets ∗)
end (∗ apriori ∗)
```

E_j: candidate item sets of size j, F_j: frequent item sets of size j.

The Apriori Algorithm 2

```
function candidates ($F_k$)
begin                                      (* — generate candidates with $k+1$ items *)
  $E := \emptyset$;                        (* initialize the set of candidates *)
  forall $f_1, f_2 \in F_k$                (* traverse all pairs of frequent item sets *)
  with  $f_1 = \{i_1, \ldots, i_{k-1}, i_k\}$     (* that differ only in one item and *)
  and   $f_2 = \{i_1, \ldots, i_{k-1}, i'_k\}$    (* are in a lexicographic order *)
  and   $i_k < i'_k$ do begin              (* (this order is arbitrary, but fixed) *)
    $f := f_1 \cup f_2 = \{i_1, \ldots, i_{k-1}, i_k, i'_k\}$;   (* union has $k+1$ items *)
    if $\forall i \in f : f - \{i\} \in F_k$   (* if all subsets with $k$ items are frequent, *)
    then $E := E \cup \{f\}$;              (* add the new item set to the candidates *)
  end;                                     (* (otherwise it cannot be frequent) *)
  return $E$;                              (* return the generated candidates *)
end (* candidates *)
```

The Apriori Algorithm 3

```
function prune ($E, T, s_{\min}$)
begin                                        (* — prune infrequent candidates *)
  forall $e \in E$ do                        (* initialize the support counters *)
    $s_T(e) := 0$;                           (* of all candidates to be checked *)
  forall $t \in T$ do                        (* traverse the transactions *)
    forall $e \in E$ do                      (* traverse the candidates *)
      if $e \subseteq t$                     (* if the transaction contains the candidate, *)
      then $s_T(e) := s_T(e) + 1$;           (* increment the support counter *)
  $F := \emptyset$;                          (* initialize the set of frequent candidates *)
  forall $e \in E$ do                        (* traverse the candidates *)
    if $s_T(e) \geq s_{\min}$                (* if a candidate is frequent, *)
    then $F := F \cup \{e\}$;                (* add it to the set of frequent item sets *)
  return $F$;                                (* return the pruned set of candidates *)
end (* prune *)
```

Improving the Candidate Generation

Searching for Frequent Item Sets

- The Apriori algorithm searches the partial order top-down level by level.

- Collecting the frequent item sets of size k in a *set* F_k has drawbacks:
  A frequent item set of size $k+1$ can be formed in

  $$j = \frac{k(k+1)}{2}$$

  possible ways. (For infrequent item sets the number may be smaller.)

  As a consequence, the candidate generation step may carry out a lot of redundant work, since it suffices to generate each candidate item set once.

- **Question:** Can we reduce or even eliminate this redundant work?

  More generally:
  How can we make sure that any candidate item set is generated at most once?

- **Idea:** Assign to each item set a unique parent item set,
  from which this item set is to be generated.

Searching for Frequent Item Sets

- A core problem is that an item set of size k (that is, with k items) can be generated in $k!$ different ways (on $k!$ paths in the Hasse diagram), because in principle the items may be added in any order.

- If we consider an item-by-item process of building an item set (which can be imagined as a levelwise traversal of the partial order), there are k possible ways of forming an item set of size k from item sets of size $k - 1$ by adding the remaining item.

- It is obvious that it suffices to consider each item set at most once in order to find the frequent ones (infrequent item sets need not be generated at all).

- **Question:** Can we reduce or even eliminate this redundant work?

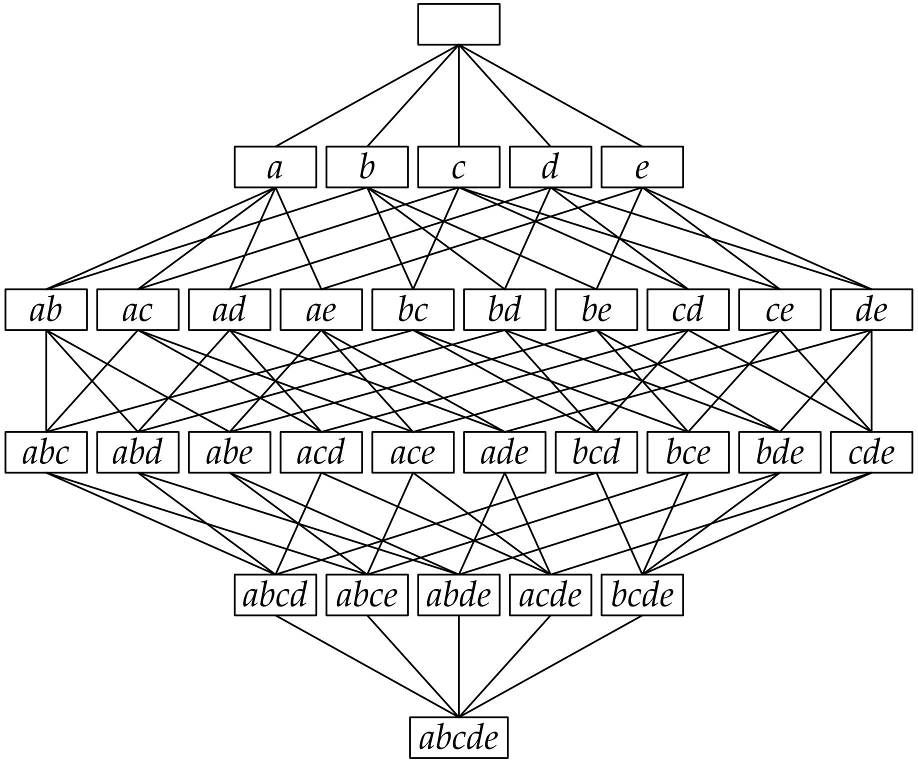
  More generally:
  How can we make sure that any candidate item set is generated at most once?

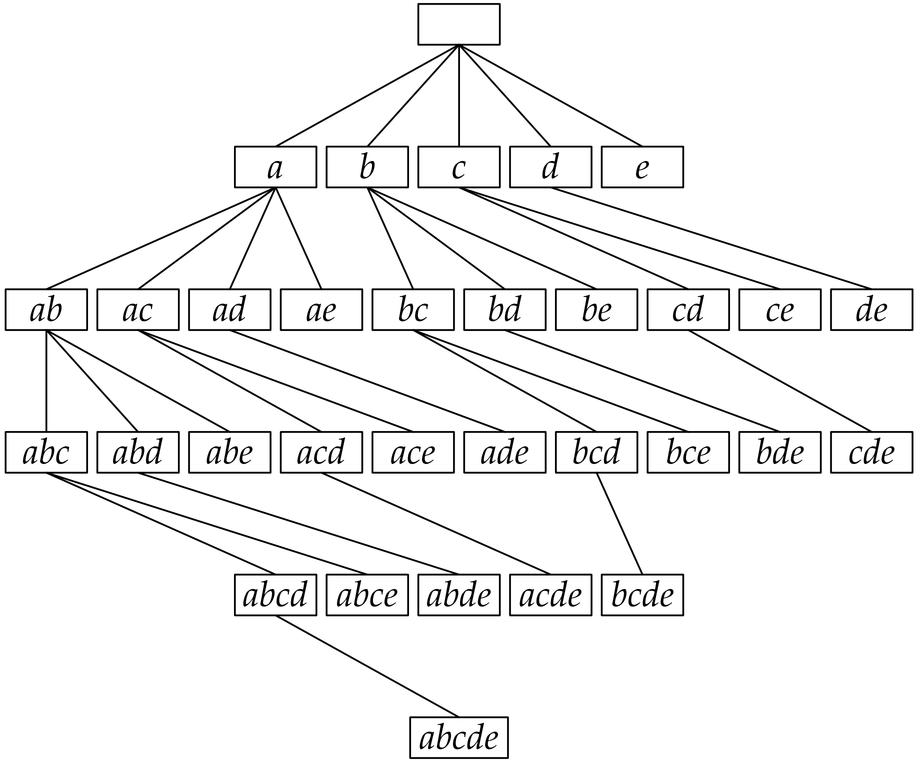
- **Idea:** Assign to each item set a unique parent item set,
  from which this item set is to be generated.

Searching for Frequent Item Sets

- We have to search the partially ordered set $(2^B, \subseteq)$ or its Hasse diagram.
- Assigning **unique parents** turns the Hasse diagram into a **tree**.
- Traversing the resulting tree explores each item set exactly once.

Hasse diagram and a possible tree for five items:





Searching with Unique Parents

Principle of a Search Algorithm based on Unique Parents:

- **Base Loop:**
  - Traverse all one-element item sets (their unique parent is the empty set).
  - Recursively process all one-element item sets that are frequent.
- **Recursive Processing:**

  For a given frequent item set I:
  - Generate all extensions J of I by one item (that is, $J \supset I$, $|J| = |I| + 1$) for which the item set I is the chosen unique parent.
  - For all J: if J is frequent, process J recursively, otherwise discard J.
- **Questions:**
  - How can we formally assign unique parents?
  - How can we make sure that we generate only those extensions for which the item set that is extended is the chosen unique parent?

Assigning Unique Parents

- Formally, the set of all **possible/candidate parents** of an item set I is

  $$\Pi(I) = \{J \subset I \mid \nexists K : J \subset K \subset I\}.$$

  In other words, the possible parents of I are its *maximal proper subsets*.

- In order to single out one element of $\Pi(I)$, the **canonical parent** $\pi_c(I)$, we can simply define an (arbitrary, but fixed) global order of the items:

  $$i_1 < i_2 < i_3 < \cdots < i_m.$$

  Then the canonical parent of an item set I can be defined as the item set

  $$\pi_c(I) = I - \{\max_{i \in I} i\} \qquad \left(\text{or} \quad \pi_c(I) = I - \{\min_{i \in I} i\}\right),$$

  where the maximum (or minimum) is taken w.r.t. the chosen order of the items.

- Even though this approach is straightforward and simple, we reformulate it now in terms of a **canonical form** of an item set, in order to lay the foundations for the study of frequent (sub)graph mining.

Canonical Forms of Item Sets

Canonical Forms

The meaning of the word "canonical":

(source: Oxford Advanced Learner's Dictionary — Encyclopedic Edition)

canon /ˈkænən/ *n* **1** general rule, standard or principle, by which sth is judged: *This film offends against all the canons of good taste.* …

canonical /kəˈnɒnɪkl/ *adj* … **3** standard; accepted. …

- A **canonical form** of something is a standard representation of it.
- The canonical form must be unique (otherwise it could not be standard).

  Nevertheless there are often several possible choices for a canonical form. However, one must fix one of them for a given application.
- In the following we will define a standard representation of an item set, and later standard representations of a graph, a sequence, a tree etc.
- This canonical form will be used to assign unique parents to all item sets.

A Canonical Form for Item Sets

- An item set is represented by a **code word**; each letter represents an item.

  The code word is a word over the alphabet B, the item base.

- There are $k!$ possible code words for an item set of size k, because the items may be listed in any order.

- By introducing an (arbitrary, but fixed) **order of the items**, and by comparing code words lexicographically w.r.t. this order, we can define an order on these code words.

  Example: $abc < bac < bca < cab$ etc. for the item set $\{a, b, c\}$ and $a < b < c$.

- The lexicographically smallest (or, alternatively, greatest) code word for an item set is defined to be its **canonical code word**.

  Obviously the canonical code word lists the items in the chosen, fixed order.

Remark: These concepts may appear obfuscated, since the core idea and the result are very simple. However, the view developed here will help us a lot when we turn to frequent (sub)graph mining.

Canonical Forms and Canonical Parents

- Let I be an item set and $w_c(I)$ its canonical code word.

  The **canonical parent** $\pi_c(I)$ of the item set I is the item set described by the **longest proper prefix** of the code word $w_c(I)$.

- Since the canonical code word of an item set lists its items in the chosen order, this definition is equivalent to

$$\pi_c(I) = I - \{\max_{i \in I} i\}.$$

- **General Recursive Processing with Canonical Forms:**

  For a given frequent item set I:

  - Generate all possible extensions J of I by one item ($J \supset I$, $|J| = |I| + 1$).
  - Form the canonical code word $w_c(J)$ of each extended item set J.
  - For each J: if the last letter of $w_c(J)$ is the item added to I to form J and J is frequent, process J recursively, otherwise discard J.

The Prefix Property

- Note that the considered item set coding scheme has the **prefix property**:

  *The longest proper prefix of the canonical code word of any item set
  is a canonical code word itself.*

⇒ With the longest proper prefix of the canonical code word of an item set I we not only know the canonical parent of I, but also its canonical code word.

- Example: Consider the item set $I = \{a, b, d, e\}$:
  - The canonical code word of I is $abde$.
  - The longest proper prefix of $abde$ is abd.
  - The code word abd is the canonical code word of $\pi_c(I) = \{a, b, d\}$.

- Note that the prefix property immediately implies:

  Every prefix of a canonical code word is a canonical code word itself.

(In the following both statements are called the **prefix property**, since they are obviously equivalent.)

Searching with the Prefix Property

The prefix property allows us to **simplify the search scheme**:

- The general recursive processing scheme with canonical forms requires to construct the **canonical code word** of each created item set in order to decide whether it has to be processed recursively or not.

⇒ We know canonical code words of all item sets that are processed recursively.

- With this code word we know, due to the **prefix property**, the canonical code words of all child item sets that have to be explored in the recursion *with the exception of the last letter* (that is, the added item).

⇒ We only have to check whether the code word that results from appending the added item to the given canonical code word is canonical or not.

- **Advantage:**
  Checking whether a given code word is canonical can be simpler/faster than constructing a canonical code word from scratch.

Searching with the Prefix Property

Principle of a Search Algorithm based on the Prefix Property:

- **Base Loop:**
  - Traverse all possible items, that is,
    the canonical code words of all one-element item sets.
  - Recursively process each code word that describes a frequent item set.

- **Recursive Processing:**

  For a given (canonical) code word of a frequent item set:

  - Generate all possible extensions by one item.
    This is done by simply **appending the item** to the code word.
  - Check whether the extended code word is the **canonical code word**
    of the item set that is described by the extended code word
    (and, of course, whether the described item set is frequent).

    If it is, process the extended code word recursively, otherwise discard it.

Searching with the Prefix Property: Examples

- Suppose the item base is $B = \{a, b, c, d, e\}$ and let us assume that we simply use the alphabetical order to define a canonical form (as before).

- Consider the recursive processing of the code word *acd* (this code word is canonical, because its letters are in alphabetical order):
  - Since *acd* contains neither *b* nor *e*, its extensions are *acdb* and *acde*.
  - The code word *acdb* is not canonical and thus it is discarded (because $d > b$ — note that it suffices to compare the last two letters)
  - The code word *acde* is canonical and therefore it is processed recursively.

- Consider the recursive processing of the code word *bc*:
  - The extended code words are *bca*, *bcd* and *bce*.
  - *bca* is not canonical and thus discarded.
    bcd and *bce* are canonical and therefore processed recursively.

Searching with the Prefix Property

Exhaustive Search

- The **prefix property** is a necessary condition for ensuring that all canonical code words can be constructed in the search by appending extensions (items) to visited canonical code words.

- Suppose the prefix property would not hold. Then:
  - ○ There exist a canonical code word w and a (proper) prefix v of w, such that v is not a canonical code word.
  - ○ Forming w by repeatedly appending items must form v first (otherwise the prefix would differ).
  - ○ When v is constructed in the search, it is discarded, because it is not canonical.
  - ○ As a consequence, the canonical code word w can never be reached.

⇒ The simplified search scheme is exhaustive only if the prefix property holds.

Searching with Canonical Forms

Straightforward Improvement of the Extension Step:

- The considered canonical form lists the items in the chosen item order.

$\Rightarrow$ If the added item succeeds all already present items in the chosen order, the result is in canonical form.

$\wedge$ If the added item precedes any of the already present items in the chosen order, the result is not in canonical form.

- As a consequence, we have a very simple **canonical extension rule** (that is, a rule that generates all children and only canonical code words).

- Applied to the Apriori algorithm, this means that we generate candidates of size $k+1$ by combining two frequent item sets $f_1 = \{i_1, \ldots, i_{k-1}, i_k\}$ and $f_2 = \{i_1, \ldots, i_{k-1}, i'_k\}$ only if $i_k < i'_k$ and $\forall j, 1 \leq j < k : i_j < i_{j+1}$.

  Note that it suffices to compare the last letters/items i_k and i'_k if all frequent item sets are represented by canonical code words.

Searching with Canonical Forms

Final Search Algorithm based on Canonical Forms:

- **Base Loop:**
  - Traverse all possible items, that is,
    the canonical code words of all one-element item sets.
  - Recursively process each code word that describes a frequent item set.

- **Recursive Processing:**

  For a given (canonical) code word of a frequent item set:

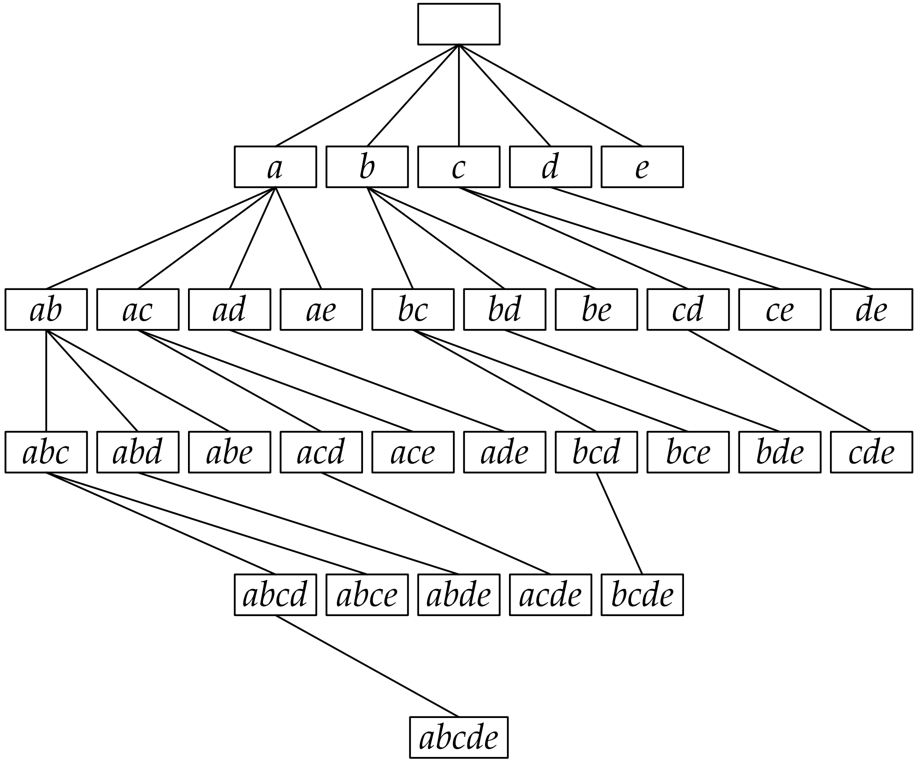
  - Generate all possible extensions by a single item,
    where this item succeeds the last letter (item) of the given code word.
    This is done by simply **appending the item** to the code word.
  - If the item set described by the resulting extended code word is frequent,
    process the code word recursively, otherwise discard it.

- This search scheme generates each candidate item set **at most once**.

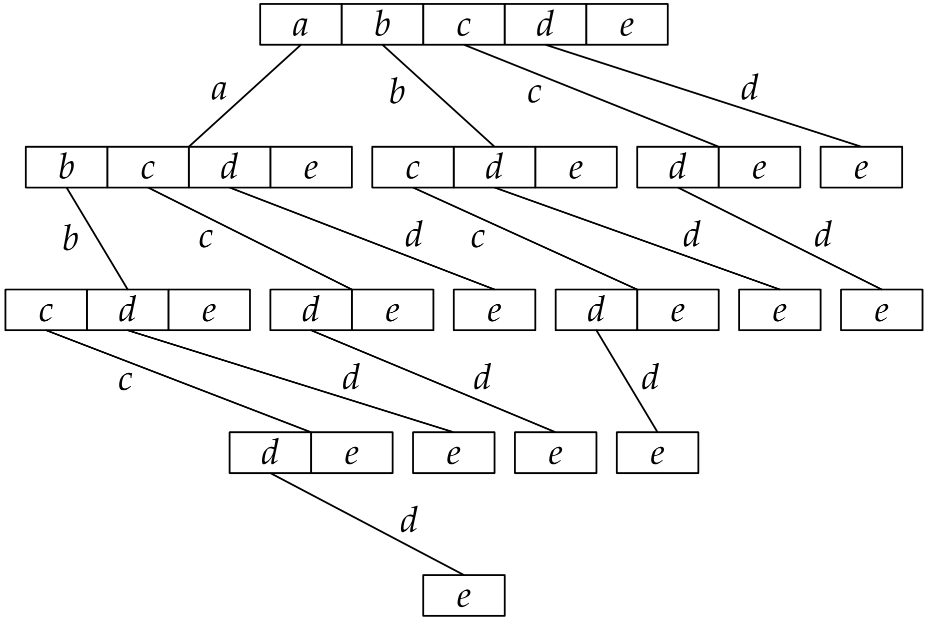
Canonical Parents and Prefix Trees

- Item sets, whose canonical code words share the same longest proper prefix are siblings, because they have (by definition) the same canonical parent.
- This allows us to represent the canonical parent tree as a **prefix tree** or **trie**.

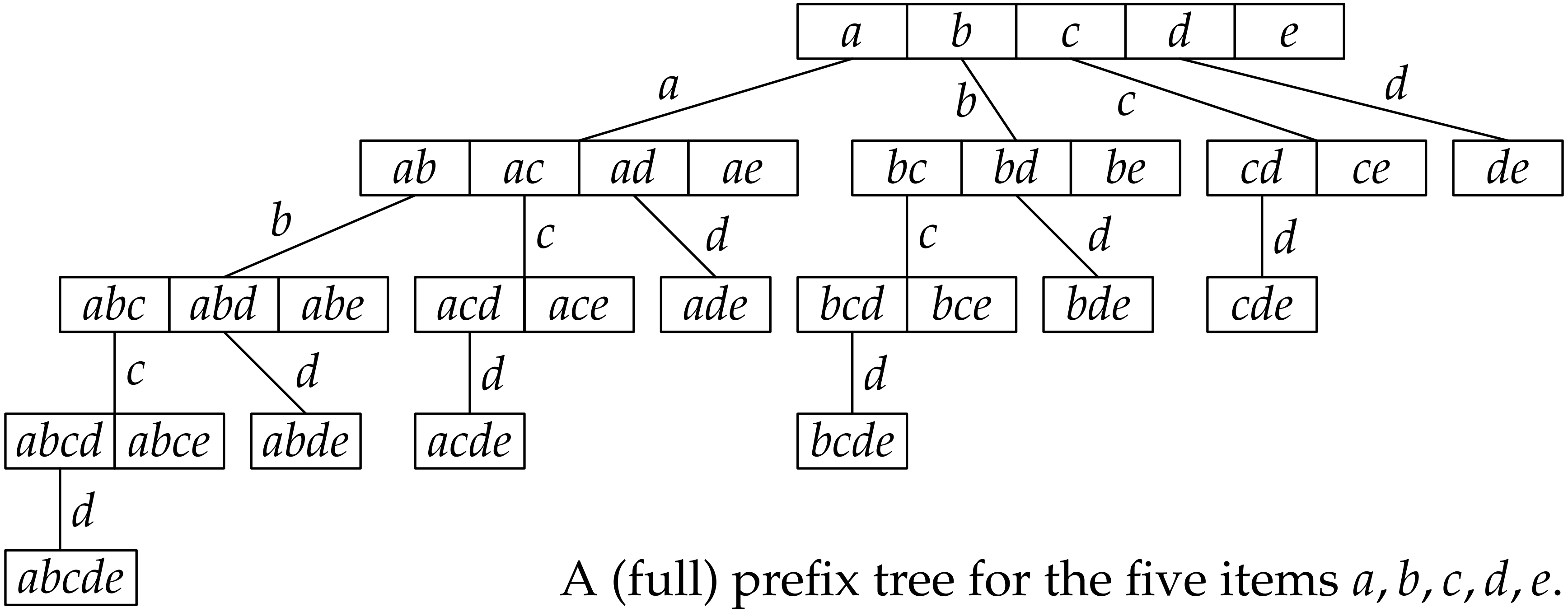
Canonical parent tree/prefix tree and prefix tree with merged siblings for five items:



a *b* *c* *d* *e*
ab *ac* *ad* *ae* *bc* *bd* *be* *cd* *ce* *de*
abc *abd* *abe* *acd* *ace* *ade* *bcd* *bce* *bde* *cde*
abcd *abce* *abde* *acde* *bcde*
abcde



Canonical Parents and Prefix Trees



A (full) prefix tree for the five items a, b, c, d, e.

- Based on a global order of the items (which can be arbitrary).
- The item sets counted in a node consist of
  - all items labeling the edges to the node (common prefix) and
  - one item following the last edge label in the item order.

Search Tree Pruning

In applications the search tree tends to get very large, so pruning is needed.

- **Structural Pruning:**
  - Extensions based on canonical code words remove superfluous paths.
  - Explains the unbalanced structure of the full prefix tree.

- **Support Based Pruning:**
  - **No superset of an infrequent item set can be frequent.**
    (*apriori property*)
  - No counters for item sets having an infrequent subset are needed.

- **Size Based Pruning:**
  - Prune the tree if a certain depth (a certain size of the item sets) is reached.
  - Idea: Sets with too many items can be difficult to interpret.

The Order of the Items

- The structure of the (structurally pruned) prefix tree obviously depends on the chosen order of the items.

- In principle, the order is arbitrary (that is, any order can be used).

  However, the number and the size of the nodes that are visited in the search differs considerably depending on the order.

  As a consequence, the execution times of frequent item set mining algorithms can differ considerably depending on the item order.

- Which order of the items is best (leads to the fastest search) can depend on the frequent item set mining algorithm used.

  Advanced methods even adapt the order of the items during the search (that is, use different, but “compatible” orders in different branches).

- Heuristics for choosing an item order are usually based on (conditional) independence assumptions.

The Order of the Items

Heuristics for Choosing the Item Order

- **Basic Idea: independence assumption**

  It is plausible that frequent item sets consist of frequent items.

  ○ Sort the items w.r.t. their support (frequency of occurrence).

  ○ Sort descendingly: Prefix tree has fewer, but larger nodes.

  ○ Sort ascendingly: Prefix tree has more, but smaller nodes.

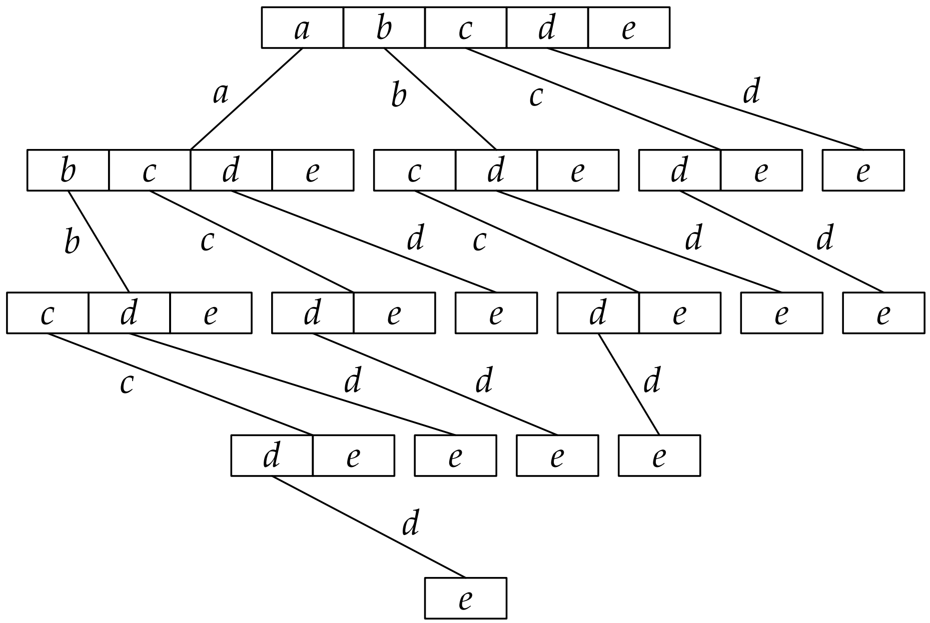
- **Extension of this Idea:**

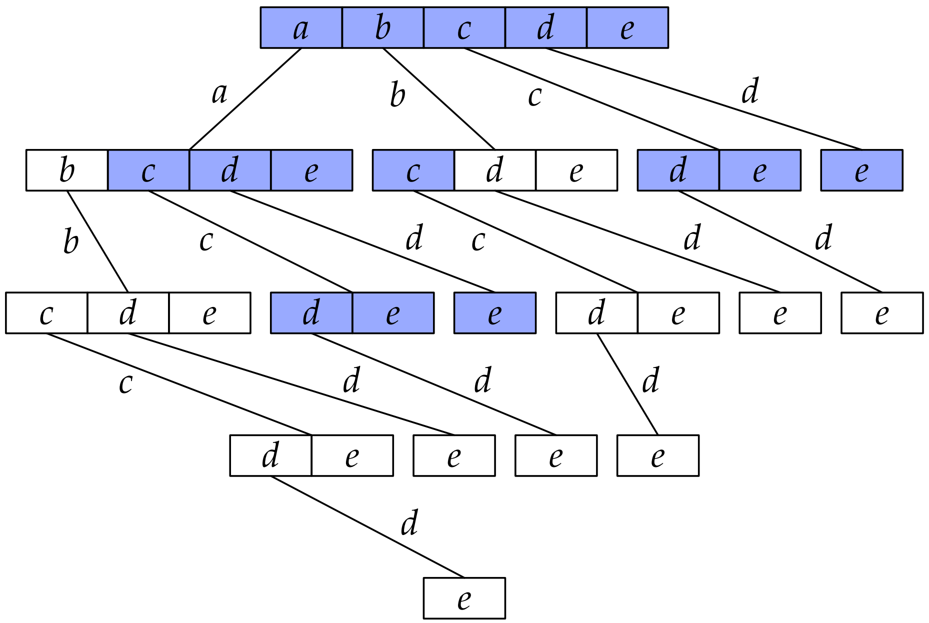
  Sort items w.r.t. the sum of the sizes of the transactions that cover them.

  ○ Idea: the sum of transaction sizes also captures implicitly the frequency of pairs, triplets etc. (though, of course, only to some degree).

  ○ Empirical evidence: better performance than simple frequency sorting.

Searching the Prefix Tree





- **Apriori**
  - Breadth-first/levelwise search (item sets of same size).
  - Subset tests on transactions to find the support of item sets.
- **Eclat**
  - Depth-first search (item sets with same prefix).
  - Intersection of transaction lists to find the support of item sets.

Searching the Prefix Tree Levelwise

(Apriori Algorithm Revisited)

Apriori: Basic Ideas

- The item sets are checked in the **order of increasing size** (**breadth-first/levelwise traversal** of the prefix tree).

- The canonical form of item sets and the induced prefix tree are used to ensure that each candidate item set is generated at most once.

- The already generated levels are used to execute *a priori* pruning of the candidate item sets (using the **apriori property**).

  (*a priori:* before accessing the transaction database to determine the support)

- Transactions are represented as simple arrays of items (so-called **horizontal transaction representation**, see also below).

- The support of a candidate item set is computed
  by checking whether they are subsets of a transaction or
  by generating subsets of a transaction and finding them among the candidates.

Apriori: Levelwise Search

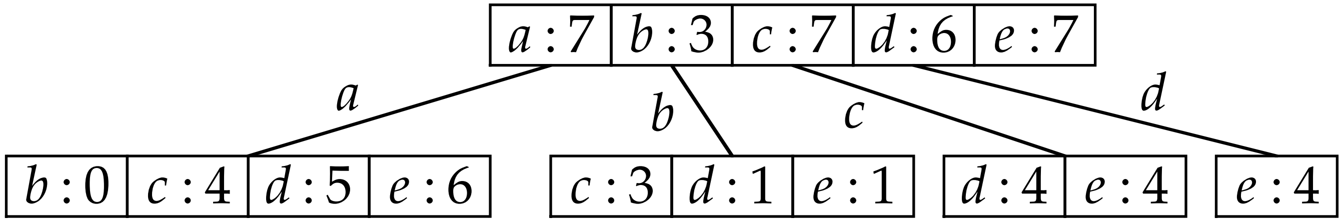
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

$a:7$	$b:3$	$c:7$	$d:6$	$e:7$

- Example transaction database with 5 items and 10 transactions.
- Minimum support: 30%, i.e., at least 3 transactions must contain the item set.
- All sets with one item (singletons) are frequent ⇒ full second level is needed.

Apriori: Levelwise Search

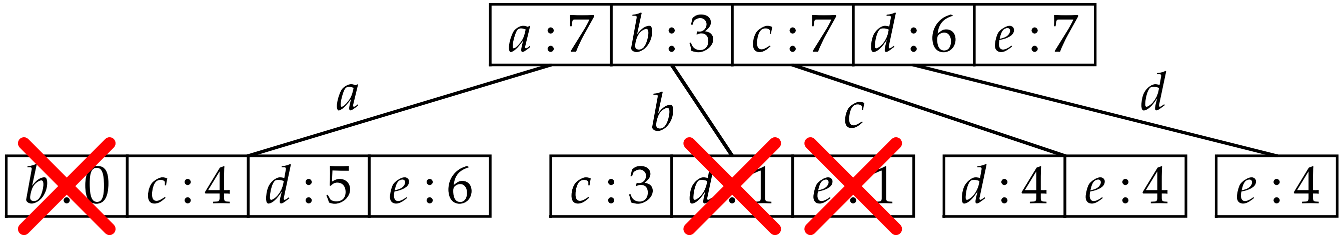
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Determining the support of item sets: For each item set traverse the database and count the transactions that contain it (highly inefficient).
- Better: Traverse the tree for each transaction and find the item sets it contains (efficient: can be implemented as a simple (doubly) recursive procedure).

Apriori: Levelwise Search

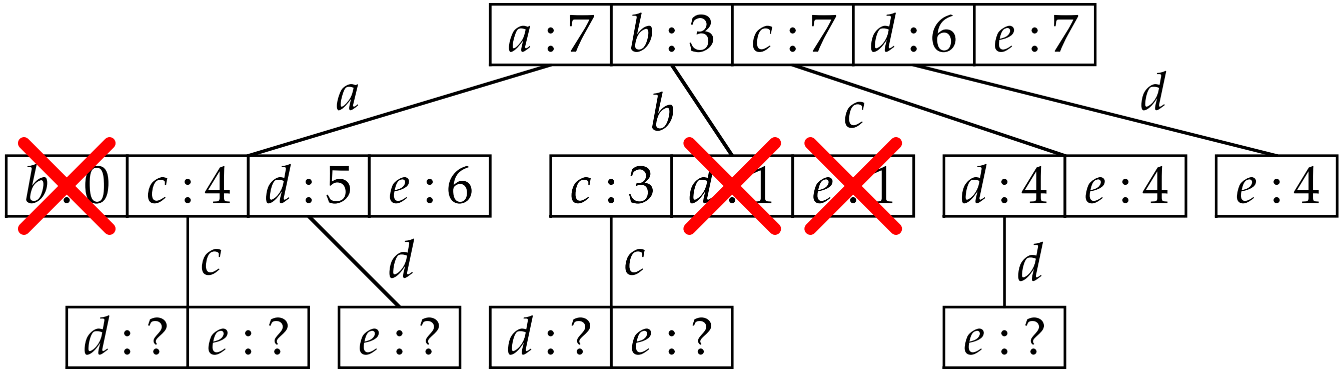
1: $\{a,d,e\}$
2: $\{b,c,d\}$
3: $\{a,c,e\}$
4: $\{a,c,d,e\}$
5: $\{a,e\}$
6: $\{a,c,d\}$
7: $\{b,c\}$
8: $\{a,c,d,e\}$
9: $\{b,c,e\}$
10: $\{a,d,e\}$



- Minimum support: 30%, i.e., at least 3 transactions must contain the item set.
- Infrequent item sets: $\{a,b\}$, $\{b,d\}$, $\{b,e\}$.
- The subtrees starting at these item sets can be pruned.
  (*a posteriori*: after accessing the transaction database to determine the support)

Apriori: Levelwise Search

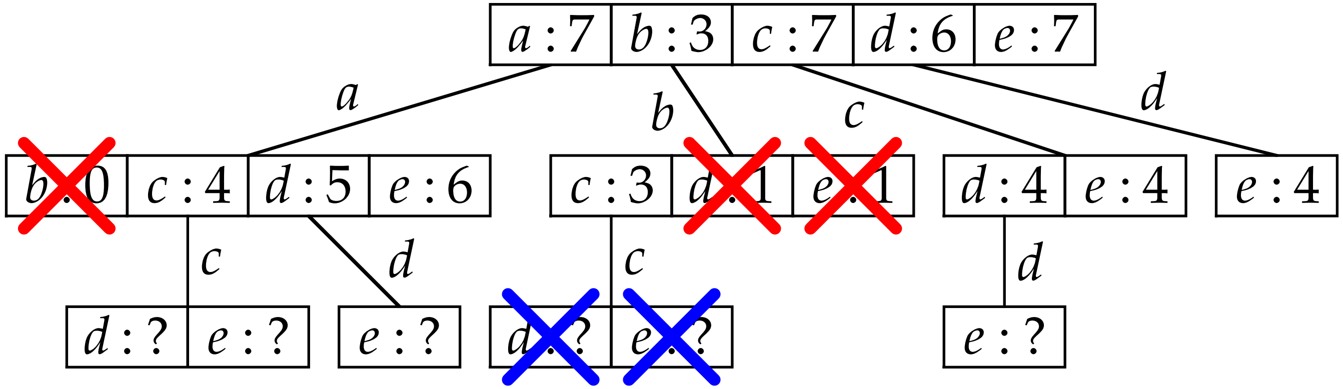
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Generate candidate item sets with 3 items (parents must be frequent).
- Before counting, check whether the candidates contain an infrequent item set.
  - An item set with k items has k subsets of size $k - 1$.
  - The parent item set is only one of these subsets.

Apriori: Levelwise Search

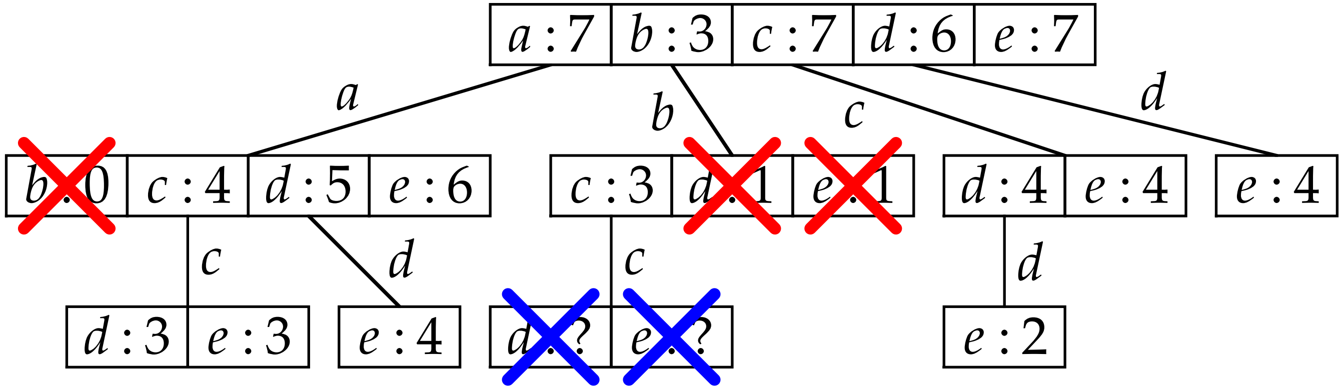
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The item sets $\{b, c, d\}$ and $\{b, c, e\}$ can be pruned, because
  - $\{b, c, d\}$ contains the infrequent item set $\{b, d\}$ and
  - $\{b, c, e\}$ contains the infrequent item set $\{b, e\}$.
- *a priori*: before accessing the transaction database to determine the support

Apriori: Levelwise Search

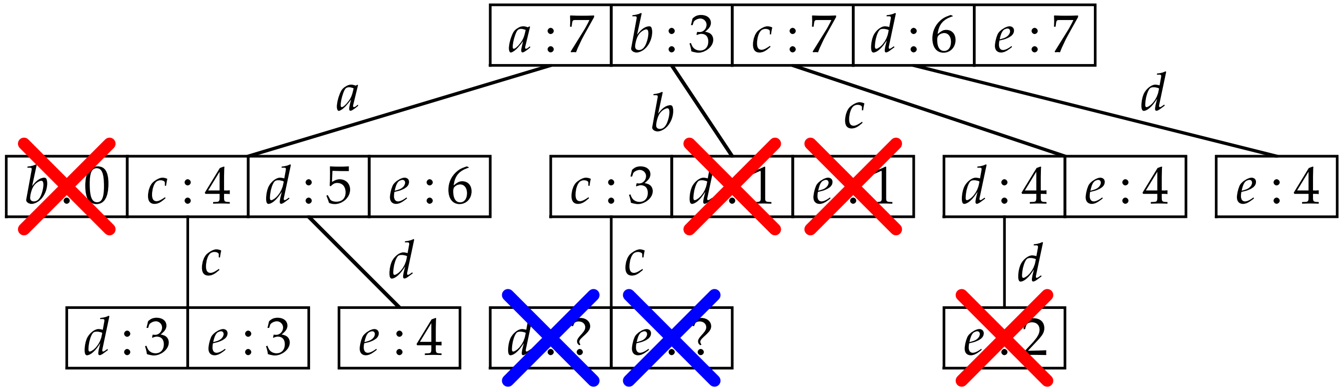
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Only the remaining four item sets of size 3 are evaluated.
- No other item sets of size 3 can be frequent.
- The transaction database is accessed to determine the support.

Apriori: Levelwise Search

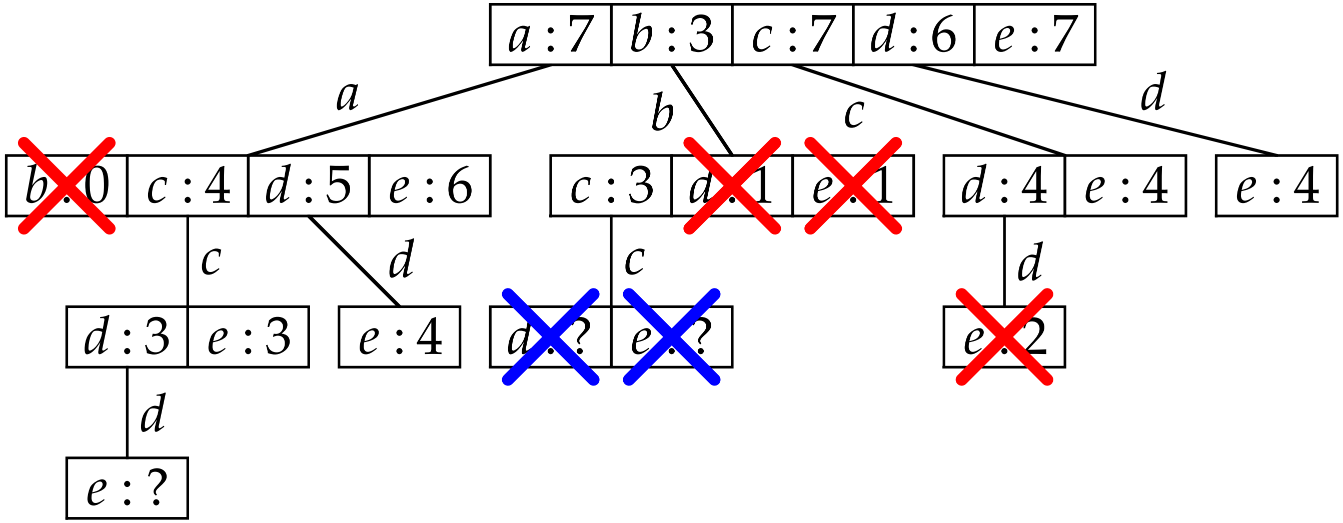
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Minimum support: 30%, i.e.; at least 3 transactions must contain the item set.
- The infrequent item set $\{c, d, e\}$ is pruned.
  (*a posteriori*: after accessing the transaction database to determine the support)
- Blue: *a priori* pruning, Red: *a posteriori* pruning.

Apriori: Levelwise Search

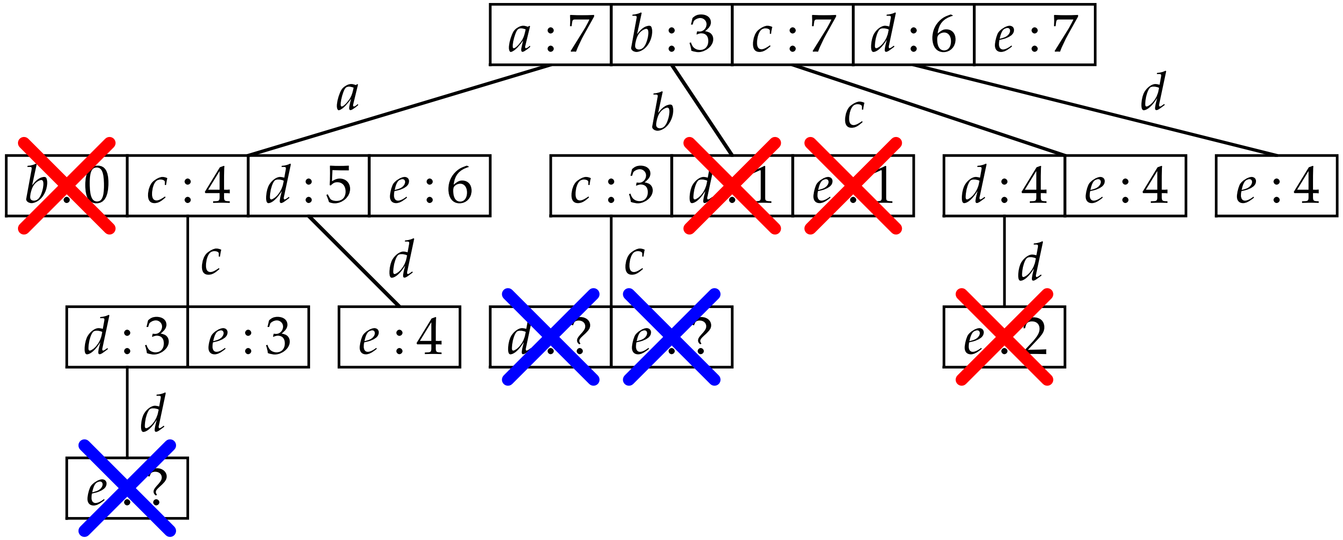
1: $\{a,d,e\}$
2: $\{b,c,d\}$
3: $\{a,c,e\}$
4: $\{a,c,d,e\}$
5: $\{a,e\}$
6: $\{a,c,d\}$
7: $\{b,c\}$
8: $\{a,c,d,e\}$
9: $\{b,c,e\}$
10: $\{a,d,e\}$



- Generate candidate item sets with 4 items (parents must be frequent).
- Before counting, check whether the candidates contain an infrequent item set. (*a priori* pruning)

Apriori: Levelwise Search

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The item set $\{a, c, d, e\}$ can be pruned, because it contains the infrequent item set $\{c, d, e\}$.
- Consequence: No candidate item sets with four items.
- Fourth access to the transaction database is not necessary.

Apriori: Node Organization 1

Idea: Optimize the organization of the counters and the child pointers.

Direct Indexing:

- Each node is a simple array of counters.
- An item is used as a direct index to find the counter.
- Advantage: Counter access is extremely fast.
- Disadvantage: Memory usage can be high due to “gaps” in the index space.

Sorted Vectors:

- Each node is a (sorted) array of item/counter pairs.
- A binary search is necessary to find the counter for an item.
- Advantage: Memory usage may be smaller, no unnecessary counters.
- Disadvantage: Counter access is slower due to the binary search.

Apriori: Node Organization 2

Hash Tables:

- Each node is a array of item/counter pairs (closed hashing).
- The index of a counter is computed from the item code.
- Advantage: Faster counter access than with binary search.
- Disadvantage: Higher memory usage than sorted arrays (pairs, fill rate).
  The order of the items cannot be exploited.

Child Pointers:

- The deepest level of the item set tree does not need child pointers.
- Fewer child pointers than counters are needed.

  ⇒ It pays to represent the child pointers in a separate array.
- The sorted array of item/counter pairs can be reused for a binary search.

Apriori: Item Coding

- Items are coded as consecutive integers starting with 0
  (needed for the direct indexing approach).

- The size and the number of the "gaps" in the index space
  depend on how the items are coded.

- Idea: It is plausible that frequent item sets consist of frequent items.
  - Sort the items w.r.t. their frequency (group frequent items).
  - Sort descendingly: prefix tree has fewer nodes.
  - Sort ascendingly: there are fewer and smaller index "gaps".
  - Empirical evidence: sorting ascendingly is better.

- Extension: Sort items w.r.t. the sum of the sizes
  of the transactions that cover them.
  - Empirical evidence: better than simple item frequencies.

Apriori: Recursive Counting

- The items in a transaction are sorted (ascending item codes).

- Processing a transaction is a **(doubly) recursive procedure**.
  To process a transaction for a node of the item set tree:

  - Go to the child corresponding to the first item in the transaction and count the suffix of the transaction recursively for that child.

    (In the currently deepest level of the tree we increment the counter corresponding to the item instead of going to the child node.)

  - Discard the first item of the transaction and
    process the remaining suffix recursively for the node itself.

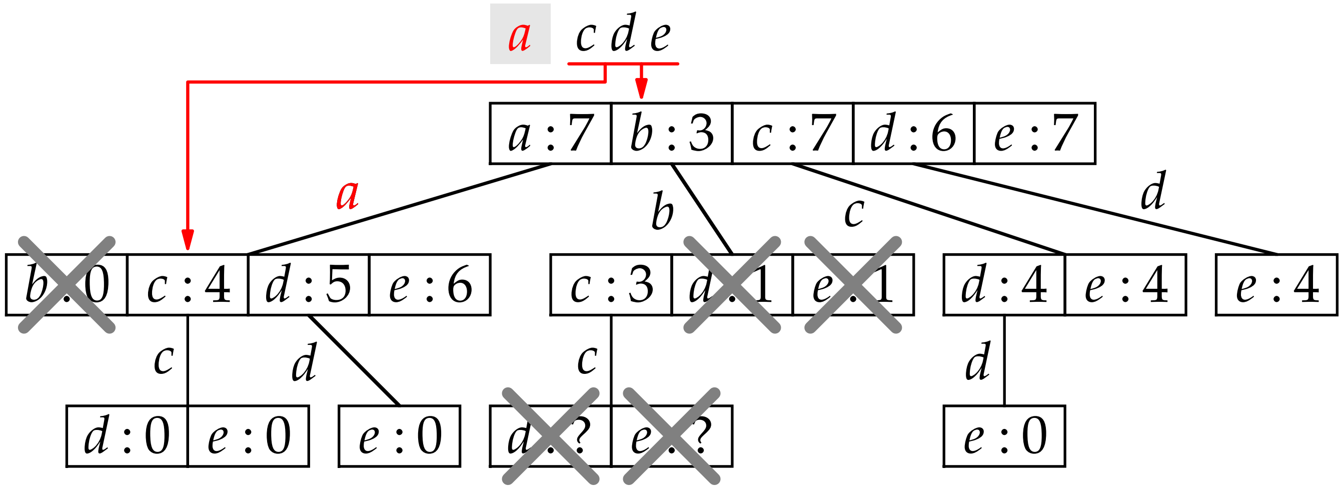
- Optimizations:

  - Directly skip all items preceding the first item in the node.
  - Abort the recursion if the first item is beyond the last one in the node.
  - Abort the recursion if a transaction is too short to reach the deepest level.

Apriori: Recursive Counting

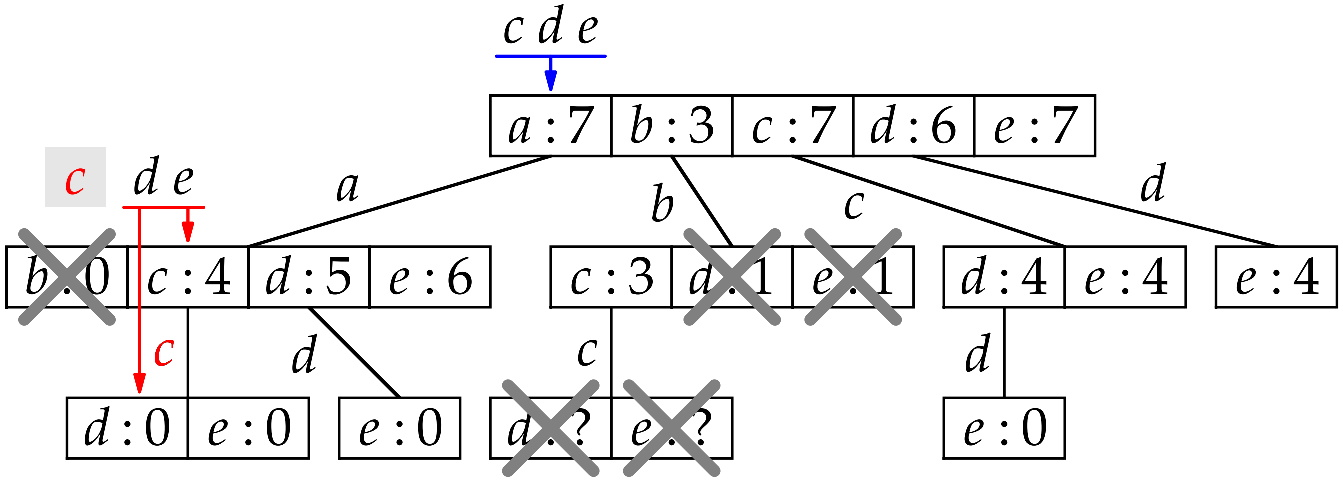
transaction
to count:
$\{a, c, d, e\}$

current
item set size: 3



processing: a

processing: c

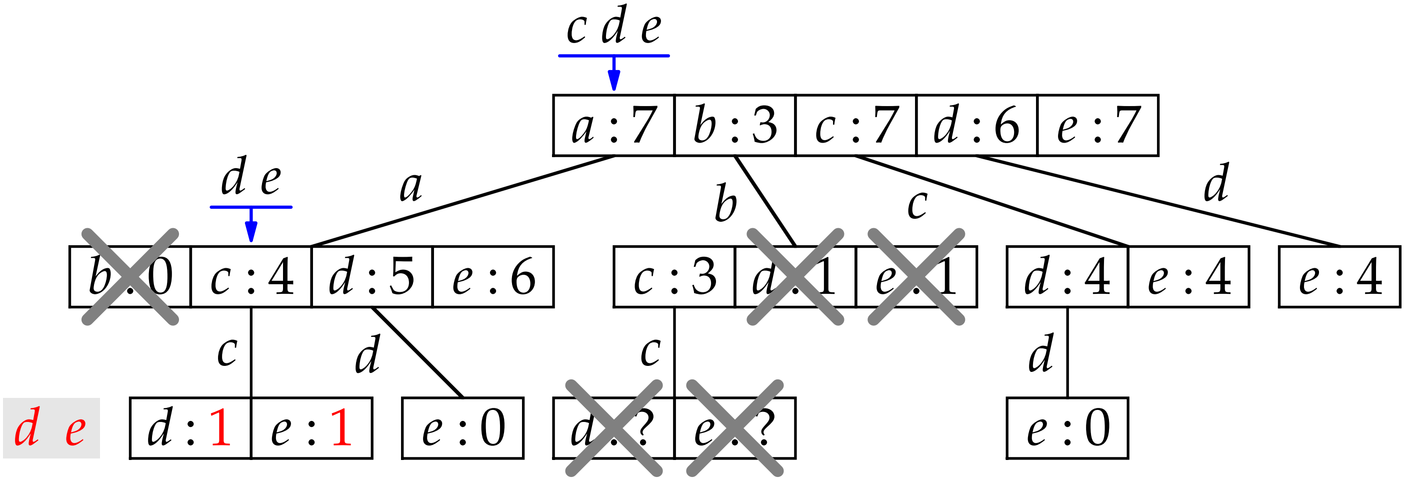


Apriori: Recursive Counting

processing: *a*

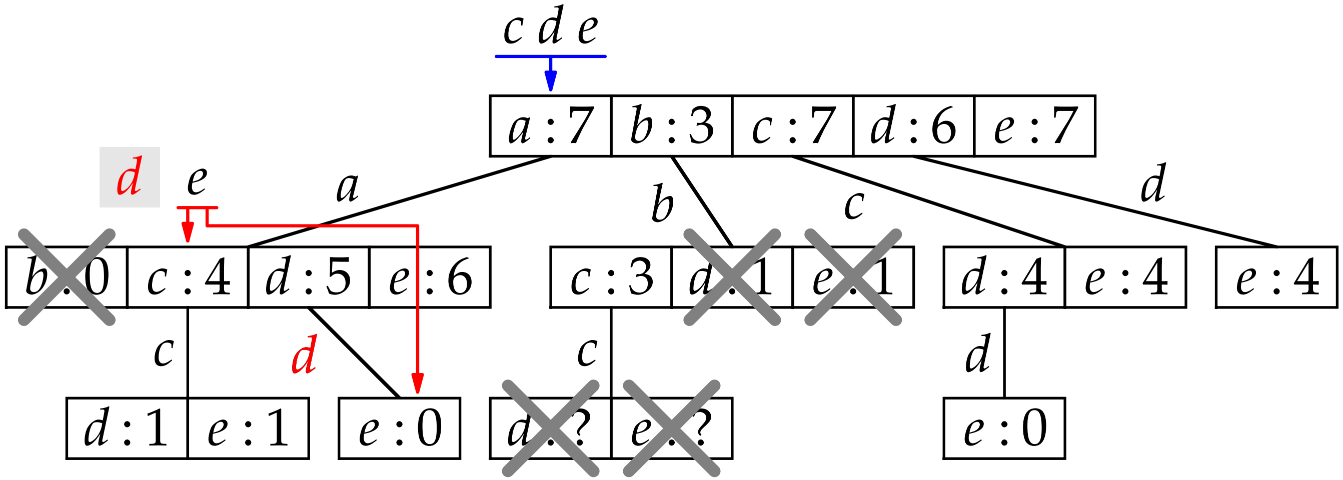
processing: *c*

processing: *d e*



processing: *a*

processing: *d*

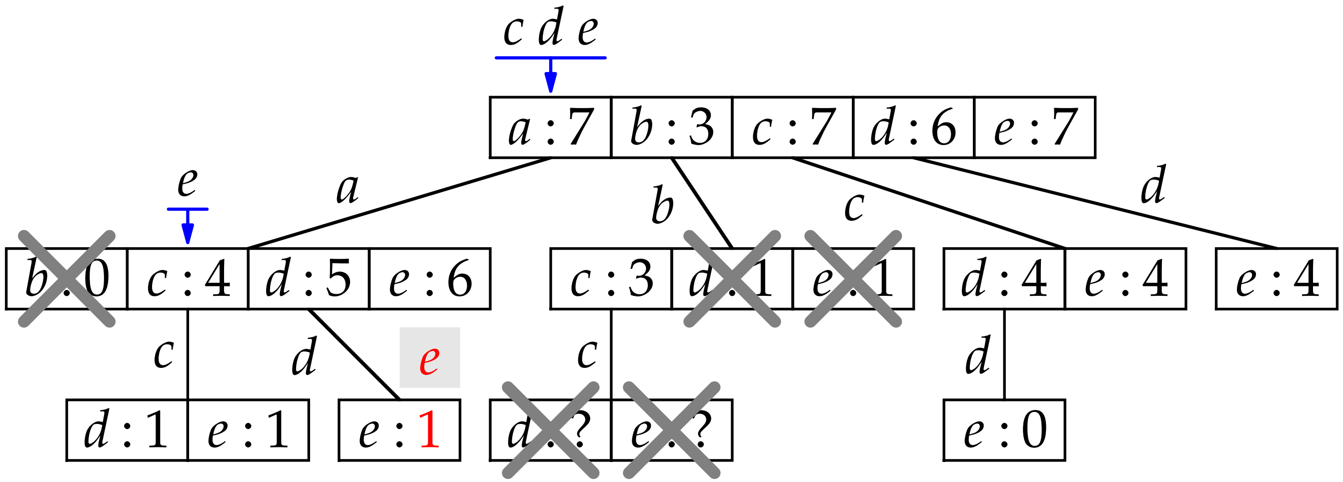


Apriori: Recursive Counting

processing: *a*

processing: *d*

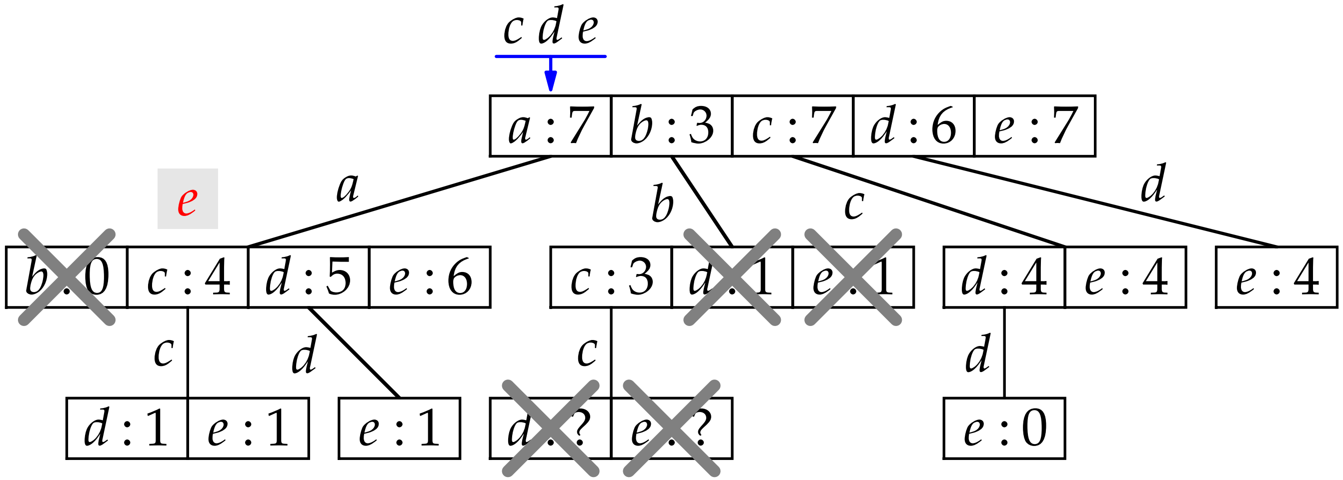
processing: *e*



processing: *a*

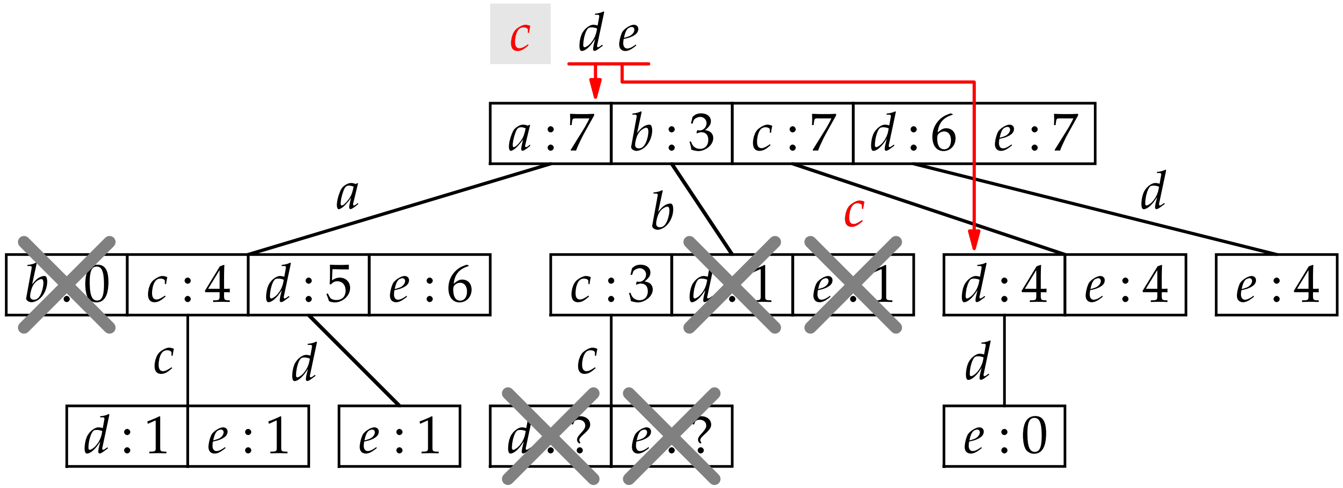
processing: *e*

(skipped:
too few items)



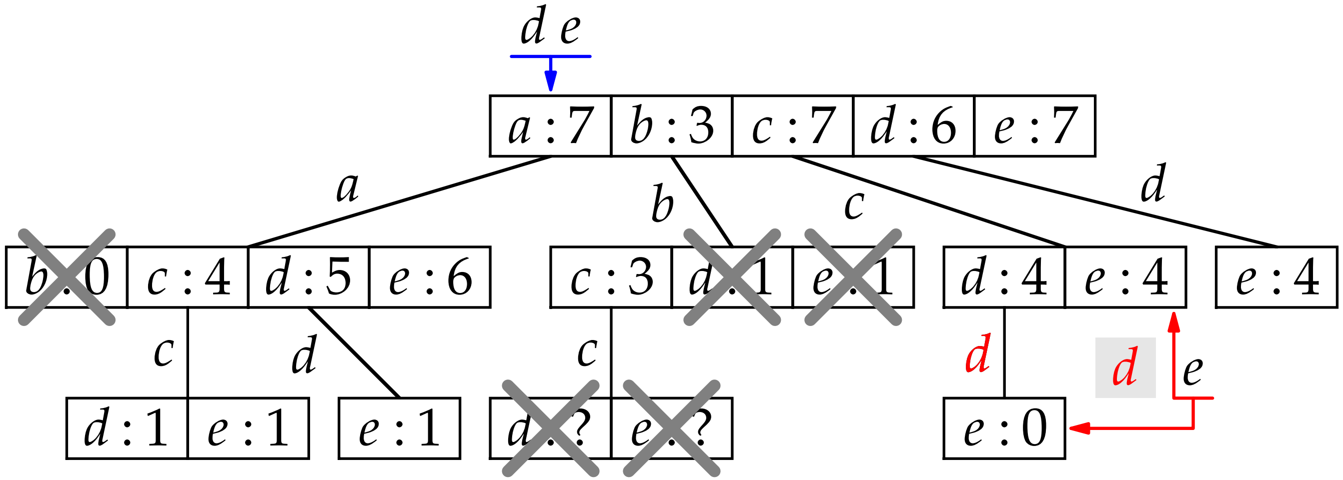
Apriori: Recursive Counting

processing: *c*



processing: *c*

processing: *d*

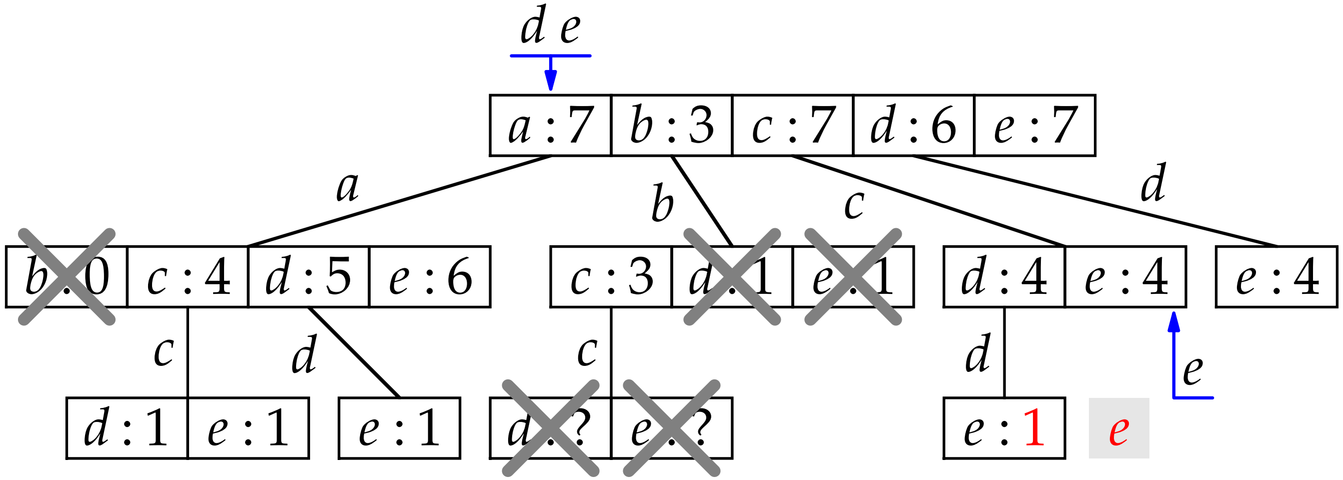


Apriori: Recursive Counting

processing: *c*

processing: *d*

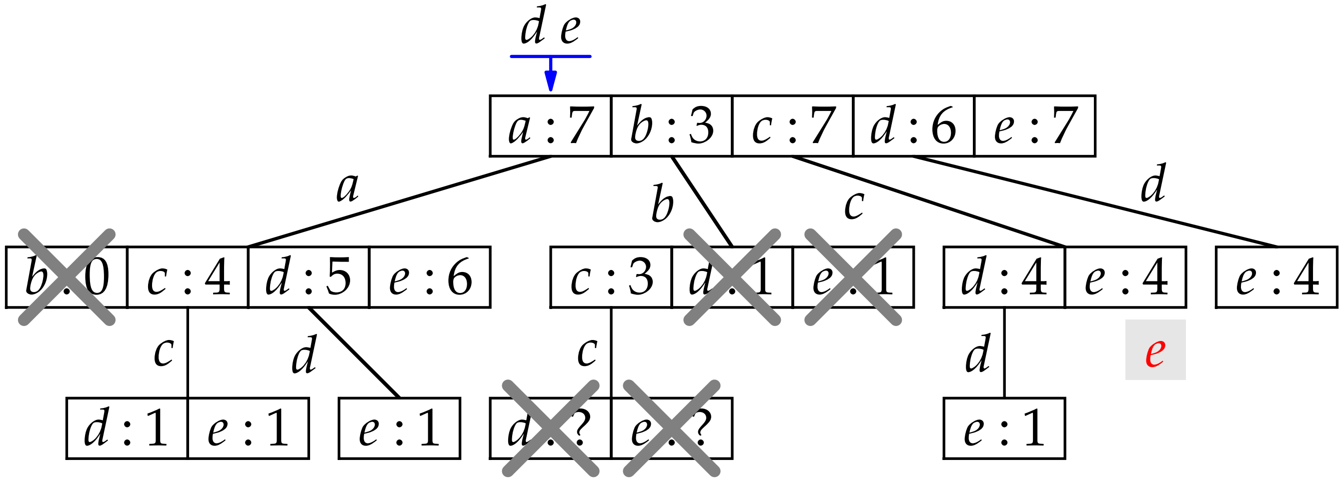
processing: *e*



processing: *c*

processing: *e*

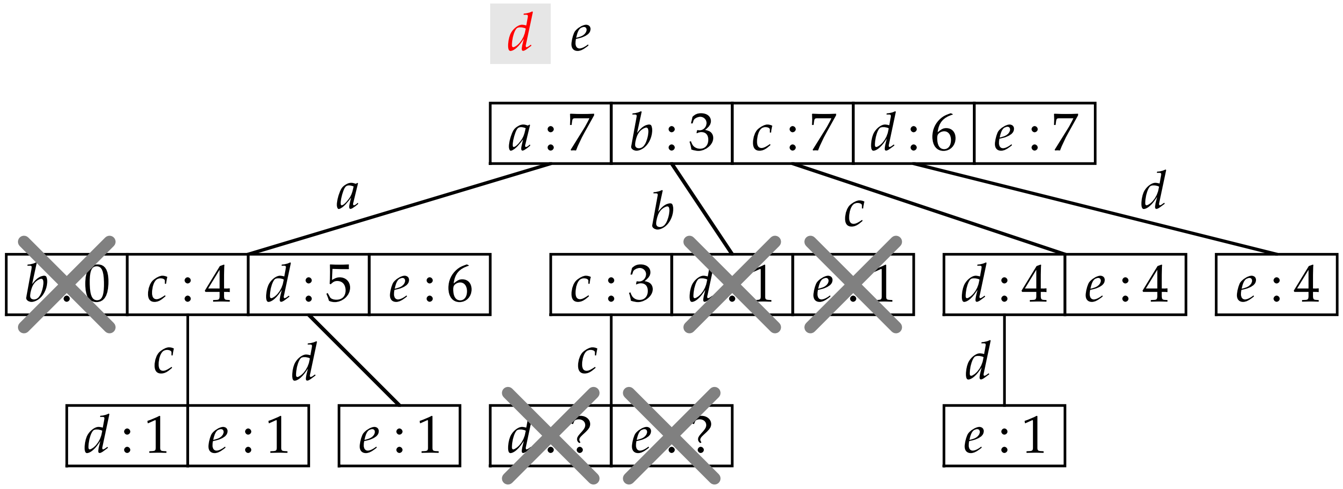
(skipped:
too few items)



Apriori: Recursive Counting

processing: *d*

(skipped:
too few items)



- Processing a transaction (suffix) in a node is implemented as a simple loop.
- For each item the remaining suffix is processed in the corresponding child.
- If the (currently) deepest tree level is reached, counters are incremented for each item in the transaction (suffix).
- If the remaining transaction (suffix) is too short to reach the (currently) deepest level, the recursion is terminated.

Apriori: Transaction Representation

Direct Representation:

- Each transaction is represented as an array of items.
- The transactions are stored in a simple list or array.

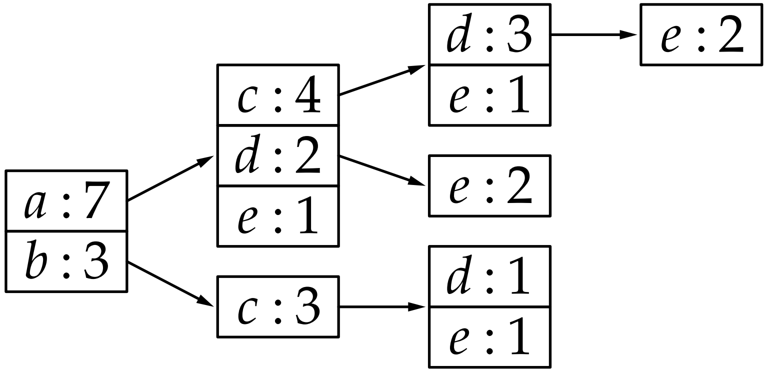
Organization as a Prefix Tree:

- The items in each transaction are sorted (arbitrary, but fixed order).
- Transactions with the same prefix are grouped together.
- Advantage: a common prefix is processed only once in the support counting.
- Gains from this organization depend on how the items are coded:
  - Common transaction prefixes are more likely
    if the items are sorted with descending frequency.
  - However: an ascending order is better for the search and
    this dominates the execution time (empirical evidence).

Apriori: Transactions as a Prefix Tree

transaction database	lexicographically sorted
a, d, e	a, c, d
b, c, d	a, c, d, e
a, c, e	a, c, d, e
a, c, d, e	a, c, e
a, e	a, d, e
a, c, d	a, d, e
b, c	a, e
a, c, d, e	b, c
b, c, e	b, c, d
a, d, e	b, c, e

prefix tree representation



- Items in transactions are sorted w.r.t. some arbitrary order, transactions are sorted lexicographically, then a prefix tree is constructed.
- **Advantage:** identical transaction prefixes are processed only once.

Summary Apriori

Basic Processing Scheme

- Breadth-first/levelwise traversal of the partially ordered set $(2^B, \subseteq)$.
- Candidates are formed by merging item sets that differ in only one item.
- Support counting can be done with a (doubly) recursive procedure.

Advantages

- "Perfect" pruning of infrequent candidate item sets (with infrequent subsets).

Disadvantages

- Can require a lot of memory (since all frequent item sets are represented).
- Support counting takes very long for large transactions.

Software

- `http://www.borgelt.net/apriori.html`

Searching the Prefix Tree Depth-First

(Eclat, FP-growth and other algorithms)

Depth-First Search and Conditional Databases

- A depth-first search can also be seen as a **divide-and-conquer scheme**:

  First find all frequent item sets that contain a chosen item,
  then all frequent item sets that do not contain it.

- General search procedure:

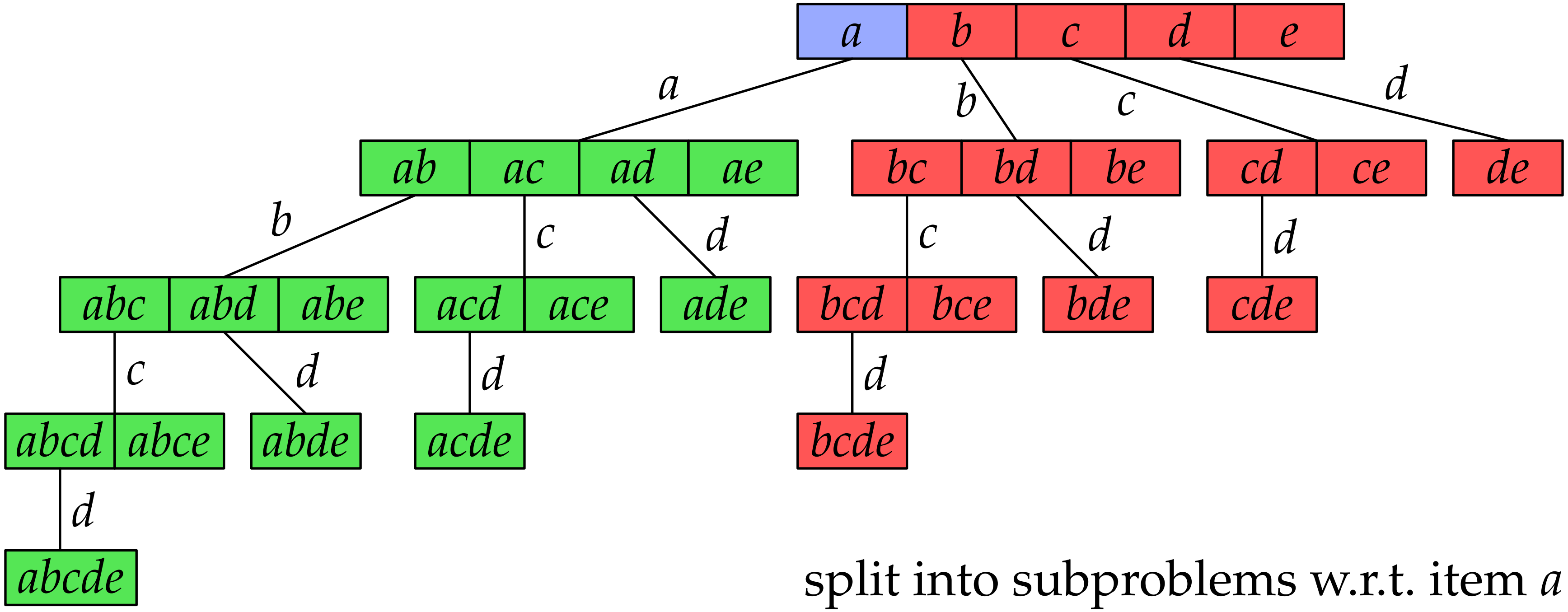
  - Let the item order be $a < b < c < \cdots$.
  - Restrict the transaction database to those transactions that contain a.
    This is the **conditional database for the prefix** a.

    Recursively search this conditional database for frequent item sets
    and add the prefix a to all frequent item sets found in the recursion.
  - Remove the item a from the transactions in the *full* transaction database.
    This is the **conditional database for item sets without** a.

    Recursively search this conditional database for frequent item sets.

- With this scheme only frequent one-element item sets have to be determined.
  Larger item sets result from adding possible prefixes.

Depth-First Search and Conditional Databases

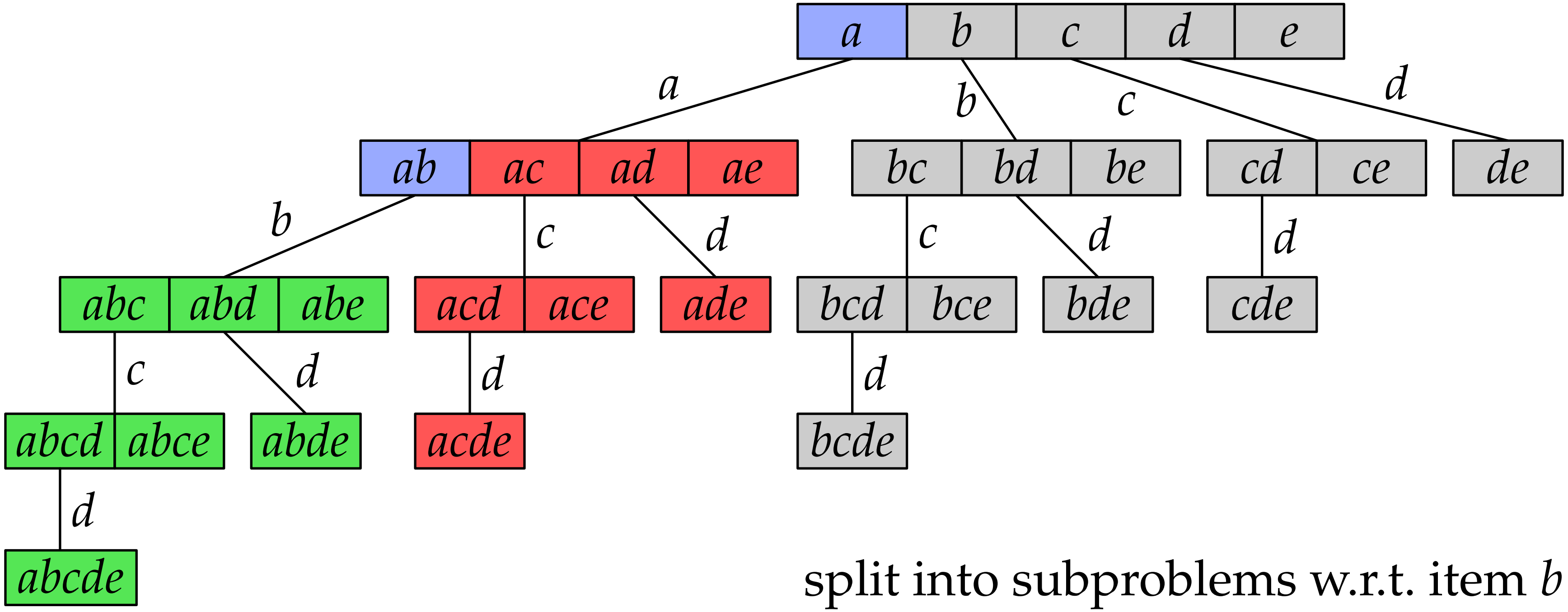


split into subproblems w.r.t. item *a*

- blue : item set containing *only* item *a*.
  green: item sets containing item *a* (and at least one other item).
  red : item sets not containing item *a* (but at least one other item).

- green: needs cond. database with transactions containing item *a*.
  red : needs cond. database with *all* transactions, but with item *a* removed.

Depth-First Search and Conditional Databases

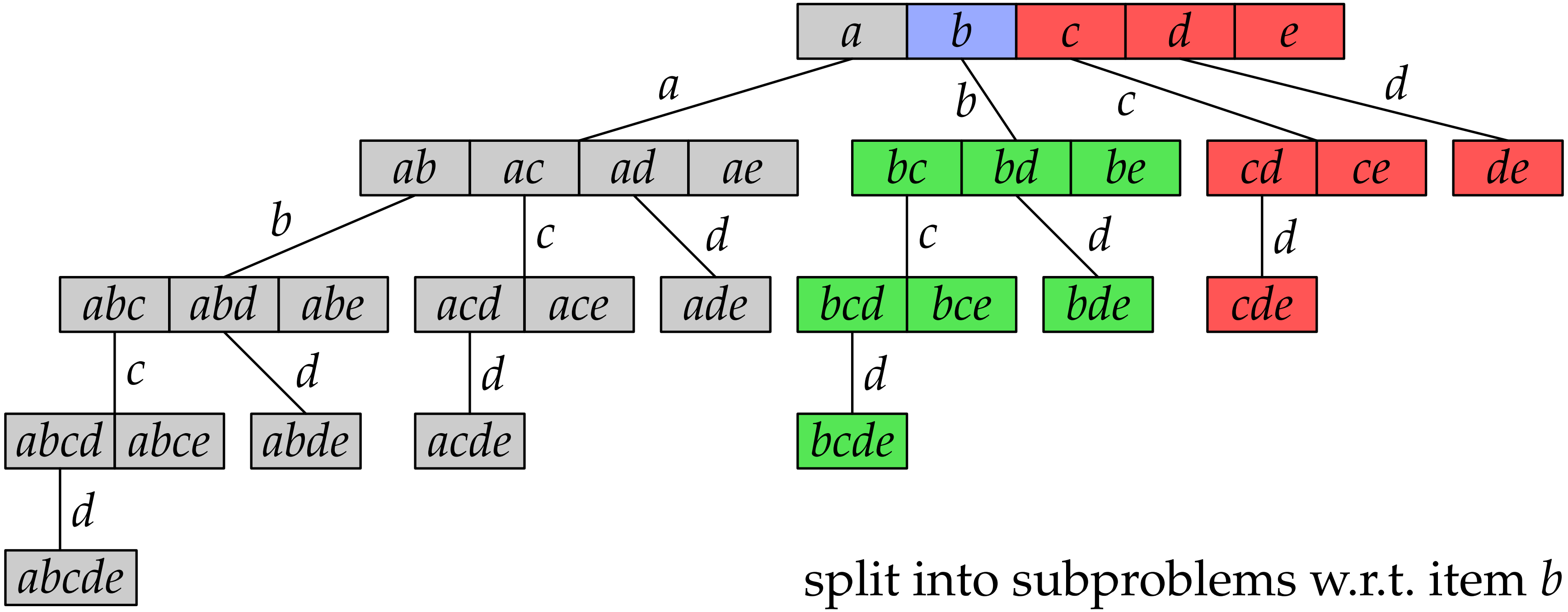


split into subproblems w.r.t. item b

- blue : item sets $\{a\}$ and $\{a, b\}$.
  green: item sets containing both items a and b (and at least one other item).
  red : item sets containing item a (and at least one other item), but not item b.

- green: needs database with trans. containing both items a and b.
  red : needs database with trans. containing item a, but with item b removed.

Depth-First Search and Conditional Databases



split into subproblems w.r.t. item *b*

- blue : item set containing *only* item *b*.
  green: item sets containing item *b* (and at least one other item), but not item *a*.
  red : item sets containing neither item *a* nor *b* (but at least one other item).

- green: needs database with trans. containing item *b*, but with item *a* removed.
  red : needs database with *all* trans., but with both items *a* and *b* removed.

Formal Description of the Divide-and-Conquer Scheme

- A divide-and-conquer scheme can be described as a set of (sub)problems.
  - The initial (sub)problem is the actual problem to solve.
  - A subproblem is processed by splitting it into smaller subproblems, which are then processed recursively.

- All subproblems that occur in frequent item set mining can be defined by
  - a **conditional transaction database** and
  - a **prefix** (of items).

  The prefix is a set of items that has to be added to all frequent item sets that are discovered in the conditional transaction database.

- Formally, all subproblems are tuples $S = (T_*, P)$, where T_* is a conditional transaction database and $P \subseteq B$ is a prefix.

- The initial problem, with which the recursion is started, is $S = (T, \emptyset)$, where T is the transaction database to mine and the prefix is empty.

Formal Description of the Divide-and-Conquer Scheme

A subproblem $S_0 = (T_0, P_0)$ is processed as follows:

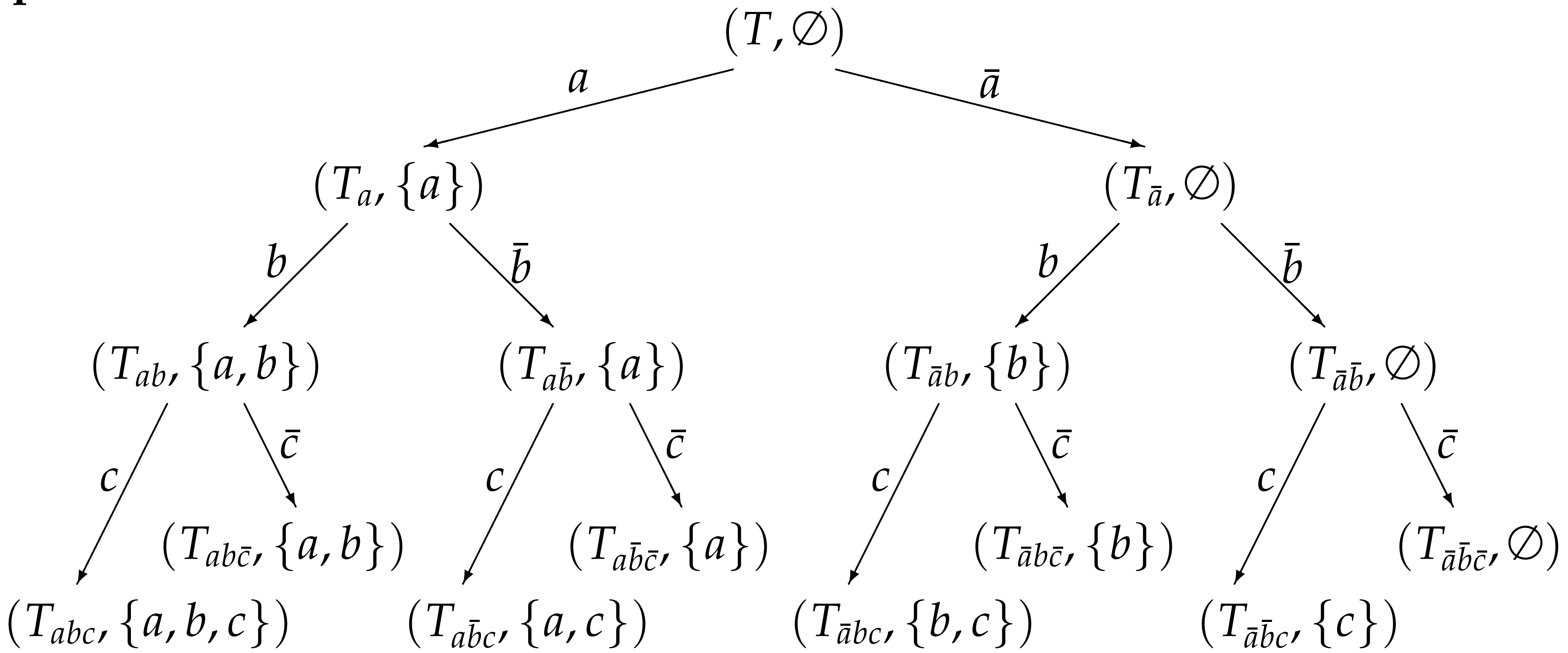
- Choose an item $i \in B_0$, where B_0 is the set of items occurring in T_0.
- If $s_{T_0}(i) \geq s_{\min}$ (where $s_{T_0}(i)$ is the support of the item i in T_0):
  - Report the item set $P_0 \cup \{i\}$ as frequent with the support $s_{T_0}(i)$.
  - Form the subproblem $S_1 = (T_1, P_1)$ with $P_1 = P_0 \cup \{i\}$.

    T_1 comprises all transactions in T_0 that contain the item i, but with the item i removed (and empty transactions removed).
  - If T_1 is not empty, process S_1 recursively.
- In any case (that is, regardless of whether $s_{T_0}(i) \geq s_{\min}$ or not):
  - Form the subproblem $S_2 = (T_2, P_2)$, where $P_2 = P_0$.

    T_2 comprises all transactions in T_0 (whether they contain i or not), but again with the item i removed (and empty transactions removed).
  - If T_2 is not empty, process S_2 recursively.

Divide-and-Conquer Recursion

Subproblem Tree



$(T, \emptyset)$
a
$\bar{a}$
$(T_a, \{a\})$
$(T_{\bar{a}}, \emptyset)$
b
$\bar{b}$
b
$\bar{b}$
$(T_{ab}, \{a, b\})$
$(T_{a\bar{b}}, \{a\})$
$(T_{\bar{a}b}, \{b\})$
$(T_{\bar{a}\bar{b}}, \emptyset)$
c
$\bar{c}$
c
$\bar{c}$
c
$\bar{c}$
c
$\bar{c}$
$(T_{ab\bar{c}}, \{a, b\})$
$(T_{a\bar{b}\bar{c}}, \{a\})$
$(T_{\bar{a}b\bar{c}}, \{b\})$
$(T_{\bar{a}\bar{b}\bar{c}}, \emptyset)$
$(T_{abc}, \{a, b, c\})$
$(T_{a\bar{b}c}, \{a, c\})$
$(T_{\bar{a}bc}, \{b, c\})$
$(T_{\bar{a}\bar{b}c}, \{c\})$

- Branch to the left: include an item (first subproblem)
- Branch to the right: exclude an item (second subproblem)

(Items in the indices of the conditional transaction databases T have been removed from them.)

Reminder: Searching with the Prefix Property

Principle of a Search Algorithm based on the Prefix Property:

- **Base Loop:**
  - Traverse all possible items, that is, the canonical code words of all one-element item sets.
  - Recursively process each code word that describes a frequent item set.
- **Recursive Processing:**

  For a given (canonical) code word of a frequent item set:

  - Generate all possible extensions by one item. This is done by simply **appending the item** to the code word.
  - Check whether the extended code word is the **canonical code word** of the item set that is described by the extended code word (and, of course, whether the described item set is frequent).

    If it is, process the extended code word recursively, otherwise discard it.

Perfect Extensions

The search can easily be improved with so-called **perfect extension pruning**.

- Let T be a transaction database over an item base B.
  Given an item set I, an item $i \notin I$ is called a **perfect extension** of I w.r.t. T, iff the item sets I and $I \cup \{i\}$ have the same support: $s_T(I) = s_T(I \cup \{i\})$ (that is, if all transactions containing the item set I also contain the item i).

- Perfect extensions have the following properties:

  ◦ If the item i is a perfect extension of an item set I, then i is also a perfect extension of any item set $J \supseteq I$ (provided $i \notin J$).

    This can most easily be seen by considering that $K_T(I) \subseteq K_T(\{i\})$ and hence $K_T(J) \subseteq K_T(\{i\})$, since $K_T(J) \subseteq K_T(I)$.

  ◦ If $X_T(I)$ is the set of all perfect extensions of an item set I w.r.t. T
    (that is, if $X_T(I) = \{i \in B - I \mid s_T(I \cup \{i\}) = s_T(I)\}$),
    then all sets $I \cup J$ with $J \in 2^{X_T(I)}$ have the same support as I
    (where 2^M denotes the power set of a set M).

Perfect Extensions: Examples

transaction database

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

- c is a perfect extension of $\{b\}$ since $\{b\}$ and $\{b, c\}$ both have support 3.
- a is a perfect extension of $\{d, e\}$ since $\{d, e\}$ and $\{a, d, e\}$ both have support 4.
- There are no other perfect extensions in this example for a minimum support of $s_{\text{min}} = 3$.

Perfect Extension Pruning

- Consider again the original **divide-and-conquer scheme**:
  A subproblem $S_0 = (T_0, P_0)$ is split into
  - a subproblem $S_1 = (T_1, P_1)$ to find all frequent item sets that *do* contain an item $i \in B_0$ and
  - a subproblem $S_2 = (T_2, P_2)$ to find all frequent item sets that *do not* contain the item i.
- Suppose the item i is a **perfect extension** of the prefix P_0.
  - Let F_1 and F_2 be the sets of frequent item sets that are reported when processing S_1 and S_2, respectively.
  - It is $\quad I \cup \{i\} \in F_1 \quad \Leftrightarrow \quad I \in F_2.$
  - The reason is that generally $P_1 = P_2 \cup \{i\}$ and in this case $T_1 = T_2$, because all transactions in T_0 contain item i (as i is a perfect extension).
- Therefore it suffices to solve one subproblem (namely S_2).
  The solution of the other subproblem (S_1) is constructed by adding item i.

Perfect Extension Pruning

- Perfect extensions can be exploited by collecting these items in the recursion, in a third element of a subproblem description.

- Formally, a subproblem is a triplet $S = (T_*, P, X)$, where
  - ○ T_* is a **conditional transaction database**,
  - ○ P is the set of **prefix items** for T_*,
  - ○ X is the set of **perfect extension items**.

- Once identified, perfect extensions are no longer processed in the recursion, but are only used to generate supersets of the prefix having the same support. Consequently, they are removed from the conditional transaction databases. This technique is also known as **hypercube decomposition**.

- The divide-and-conquer scheme has basically the same structure as without perfect extension pruning.

  However, the exact way in which perfect extensions are collected can depend on the specific algorithm used.

Reporting Frequent Item Sets

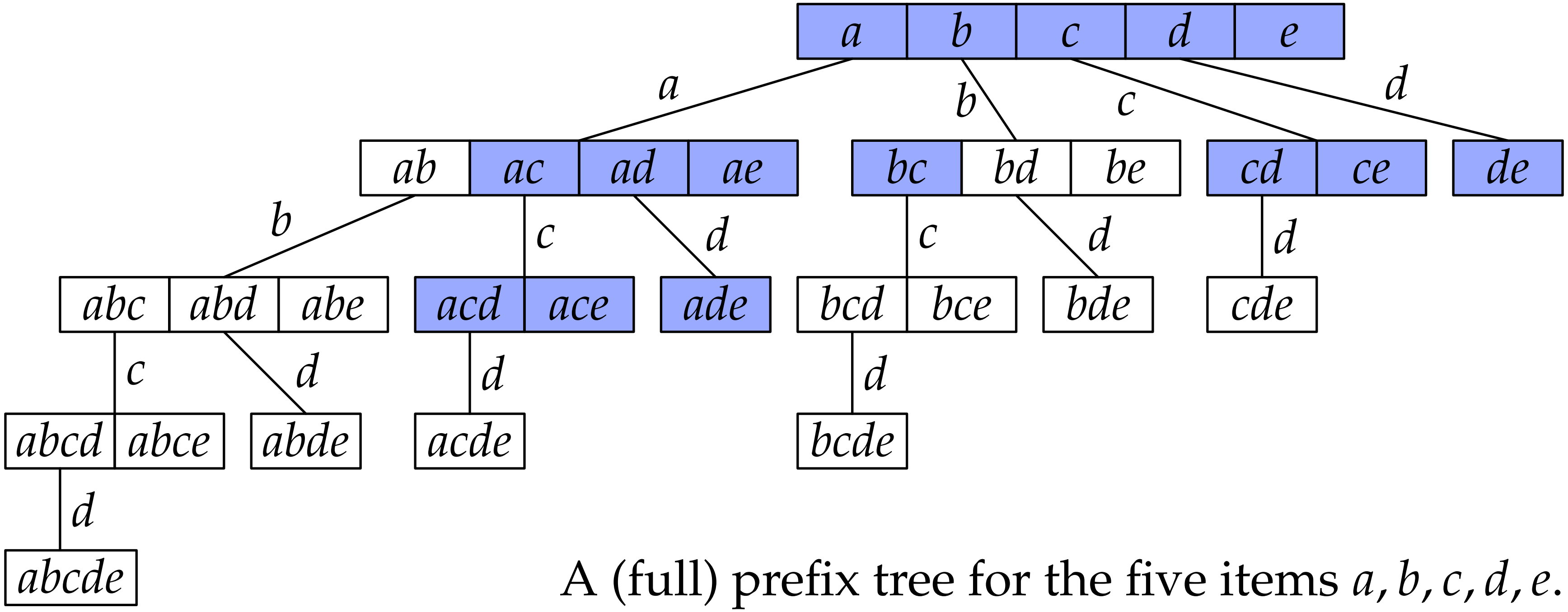
- With the described divide-and-conquer scheme, item sets are reported in **lexicographic order**.

- This can be exploited for **efficient item set reporting**:
  - The prefix P is a string, which is extended when an item is added to P.
  - Thus only one item needs to be formatted per reported frequent item set, the prefix is already formatted in the string.
  - Backtracking the search (return from recursion) removes an item from the prefix string.
  - This scheme can speed up the output considerably.

Example:

a	(7)
$a\ c$	(4)
$a\ c\ d$	(3)
$a\ c\ e$	(3)
$a\ d$	(5)
$a\ d\ e$	(4)
$a\ e$	(6)
b	(3)
$b\ c$	(3)
c	(7)
$c\ d$	(4)
$c\ e$	(4)
d	(6)
$d\ e$	(4)
e	(7)

Reporting Frequent Item Sets



A (full) prefix tree for the five items a, b, c, d, e.

Example:

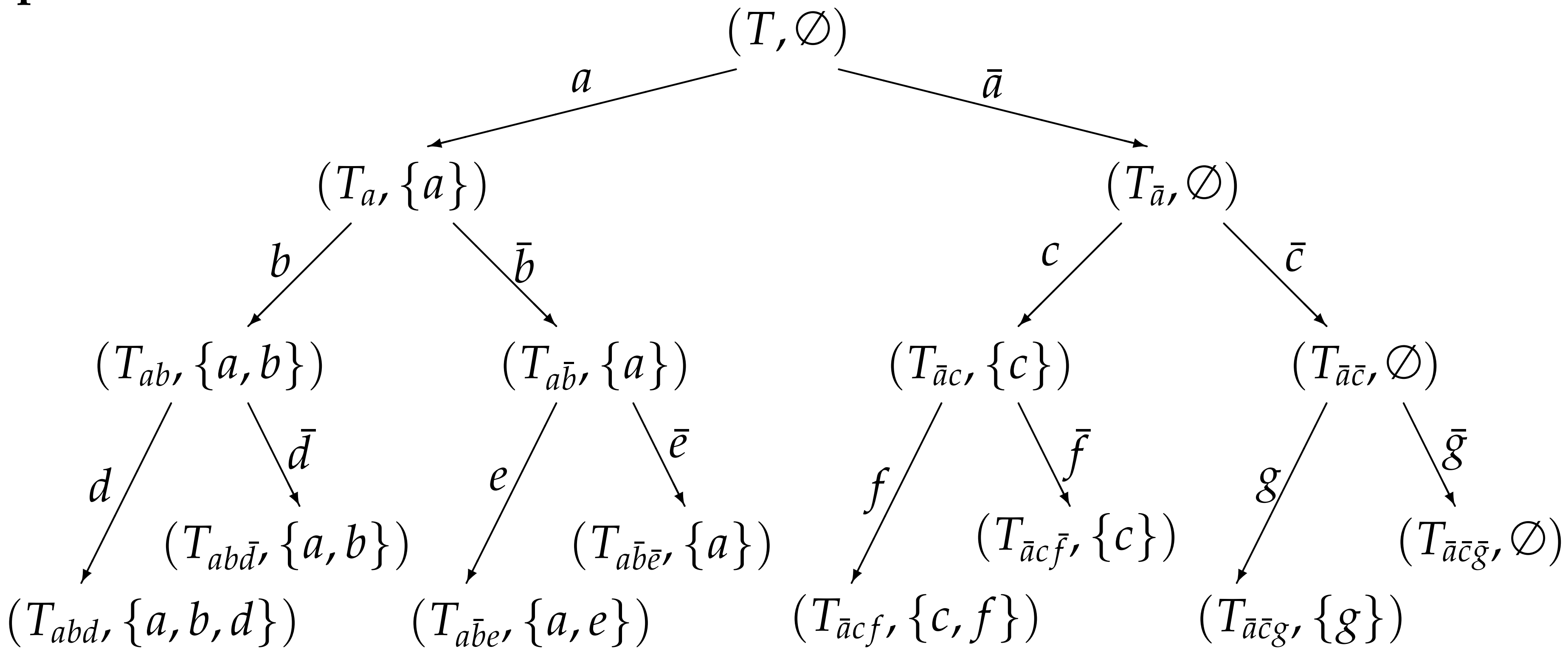
a	(7)
$a\ c$	(4)
$a\ c\ d$	(3)
$a\ c\ e$	(3)
$a\ d$	(5)
$a\ d\ e$	(4)
$a\ e$	(6)
b	(3)
$b\ c$	(3)
c	(7)
$c\ d$	(4)
$c\ e$	(4)
d	(6)
$d\ e$	(4)
e	(7)

Global and Local Item Order

- Up to now we assumed that the item order is (globally) fixed, and determined at the very beginning based on heuristics.
- However, the described divide-and-conquer scheme shows that a globally fixed item order is more restrictive than necessary:
  - The item used to split the current subproblem can be any item that occurs in the conditional transaction database of the subproblem.
  - There is no need to choose the same item for splitting sibling subproblems (as a global item order would require us to do).
  - The same heuristics used for determining a global item order suggest that the split item for a given subproblem should be selected from the (conditionally) least frequent item(s).
- As a consequence, the item orders may differ for every branch of the search tree.
  - However, two subproblems must share the item order that is fixed by the common part of their paths from the root (initial subproblem).

Item Order: Divide-and-Conquer Recursion

Subproblem Tree


$(T, \emptyset)$
a
$\bar{a}$
$(T_a, \{a\})$
$(T_{\bar{a}}, \emptyset)$
b
$\bar{b}$
c
$\bar{c}$
$(T_{ab}, \{a, b\})$
$(T_{a\bar{b}}, \{a\})$
$(T_{\bar{a}c}, \{c\})$
$(T_{\bar{a}\bar{c}}, \emptyset)$
d
$\bar{d}$
e
$\bar{e}$
f
$\bar{f}$
g
$\bar{g}$
$(T_{abd}, \{a, b, d\})$
$(T_{ab\bar{d}}, \{a, b\})$
$(T_{a\bar{b}e}, \{a, e\})$
$(T_{a\bar{b}\bar{e}}, \{a\})$
$(T_{\bar{a}cf}, \{c, f\})$
$(T_{\bar{a}c\bar{f}}, \{c\})$
$(T_{\bar{a}\bar{c}g}, \{g\})$
$(T_{\bar{a}\bar{c}\bar{g}}, \emptyset)$

- All local item orders start with $a < \ldots$
- All subproblems on the left share $a < b < \ldots$,
  All subproblems on the right share $a < c < \ldots$.

Global and Local Item Order

Local item orders have advantages and disadvantages:

- **Advantage**
  - In some data sets the order of the conditional item frequencies differs considerably from the global order.
  - Such data sets can sometimes be processed significantly faster with local item orders (depending on the algorithm).
- **Disadvantage**
  - The data structure of the conditional databases must allow us to determine conditional item frequencies quickly.
  - Not having a globally fixed item order can make it more difficult to determine conditional transaction databases w.r.t. split items (depending on the employed data structure).
  - The gains from the better item order may be lost again due to the more complex processing / conditioning scheme.

Transaction Database Representation

Transaction Database Representation

- Eclat, FP-growth and several other frequent item set mining algorithms rely on the described basic divide-and-conquer scheme.

  They differ mainly in how they represent the conditional transaction databases.

- The main approaches are horizontal and vertical representations:
  - In a **horizontal representation**, the database is stored as a list (or array) of transactions, each of which is a list (or array) of the items contained in it.
  - In a **vertical representation**, a database is represented by first referring with a list (or array) to the different items. For each item a list (or array) of identifiers is stored, which indicate the transactions that contain the item.

- However, this distinction is not pure, since there are many algorithms that use a combination of the two forms of representing a transaction database.

- Frequent item set mining algorithms also differ in how they construct new conditional transaction databases from a given one.

Transaction Database Representation

- The Apriori algorithm uses a **horizontal transaction representation**: each transaction is an array of the contained items.
  - Note that the alternative prefix tree organization is still an essentially *horizontal* representation.

- The alternative is a **vertical transaction representation**:
  - For each item a **transaction (index/identifier) list** is created.
  - The transaction list of an item i indicates the transactions that contain it, that is, it represents its **cover** $K_T(\{i\})$.
  - Advantage: the transaction list for a pair of items can be computed by intersecting the transaction lists of the individual items.
  - Generally, a vertical transaction representation can exploit

$$\forall I, J \subseteq B: \quad K_T(I \cup J) = K_T(I) \cap K_T(J).$$

- A combined representation is the **frequent pattern tree** (to be discussed later).

Transaction Database Representation

- **Horizontal Representation:** List items for each transaction
- **Vertical Representation:** List transactions for each item

1:	a, d, e
2:	b, c, d
3:	a, c, e
4:	a, c, d, e
5:	a, e
6:	a, c, d
7:	b, c
8:	a, c, d, e
9:	b, c, e
10:	a, d, e

horizontal representation

a	b	c	d	e
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

vertical representation

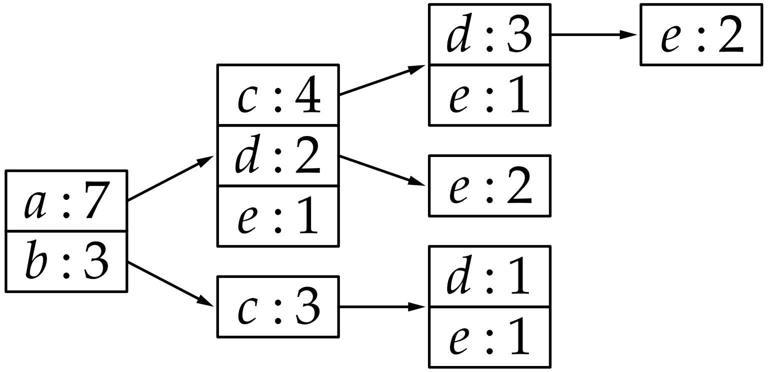
	a	b	c	d	e
1:	**1**	0	0	**1**	**1**
2:	0	**1**	**1**	**1**	0
3:	**1**	0	**1**	0	**1**
4:	**1**	0	**1**	**1**	**1**
5:	**1**	0	0	0	**1**
6:	**1**	0	**1**	**1**	0
7:	0	**1**	**1**	0	0
8:	**1**	0	**1**	**1**	**1**
9:	0	**1**	**1**	0	**1**
10:	**1**	0	0	**1**	**1**

matrix representation

Transaction Database Representation

transaction database	lexicographically sorted
a, d, e	a, c, d
b, c, d	a, c, d, e
a, c, e	a, c, d, e
a, c, d, e	a, c, e
a, e	a, d, e
a, c, d	a, d, e
b, c	a, e
a, c, d, e	b, c
b, c, e	b, c, d
a, d, e	b, c, e

prefix tree representation



- Note that a prefix tree is a "compressed" horizontal representation.
- Principle: **equal prefixes of transactions are merged.**
- This is most effective if the items are sorted descendingly w.r.t. their support.

The Eclat Algorithm

[Zaki, Parthasarathy, Ogihara, and Li 1997]

Eclat: Basic Ideas

- The item sets are checked in **lexicographic order** (**depth-first traversal** of the prefix tree).

- The search scheme is the same as the general scheme for searching with canonical forms having the prefix property and possessing a perfect extension rule (generate only canonical extensions).

- Eclat generates more candidate item sets than Apriori, because it (usually) does not store the support of all visited item sets.*

  As a consequence it cannot fully exploit the Apriori property for pruning.

- Eclat uses a purely **vertical transaction representation**.

- No subset tests and no subset generation are needed to compute the support.

  The support of item sets is rather determined by intersecting transaction lists.

* Note that Eclat cannot fully exploit the Apriori property, because it does not *store* the support of all explored item sets, not because it cannot *know* it. If all computed support values were stored, it could be implemented in such a way that all support values needed for full *a priori* pruning are available.

Eclat: Subproblem Split

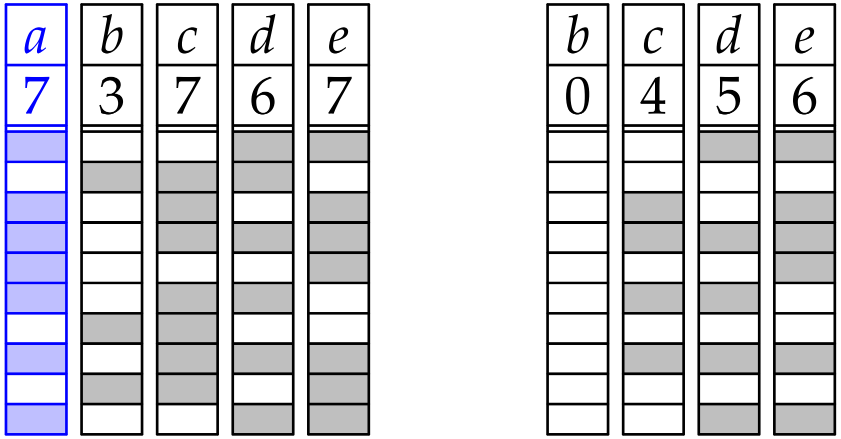
a	*b*	*c*	*d*	*e*
7	3	7	6	7
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

b	*c*	*d*	*e*
0	4	5	6
	3	1	1
	4	4	3
	6	6	4
	8	8	5
		10	8
			10

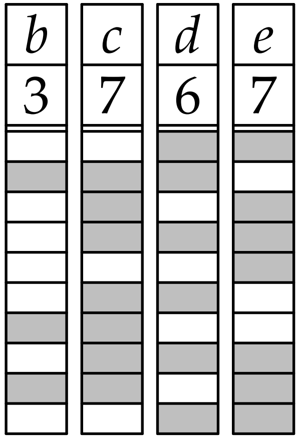
↑ Conditional database for prefix *a* (1st subproblem)

b	*c*	*d*	*e*
3	7	6	7
2	2	1	1
7	3	2	3
9	4	4	4
	6	6	5
	7	8	8
	8	10	9
	9		10

← Conditional database with item *a* removed (2nd subproblem)



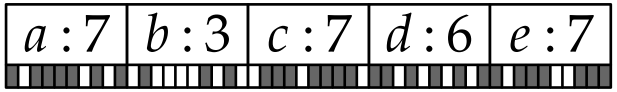
↑ Conditional database for prefix *a* (1st subproblem)



← Conditional database with item *a* removed (2nd subproblem)

Eclat: Depth-First Search

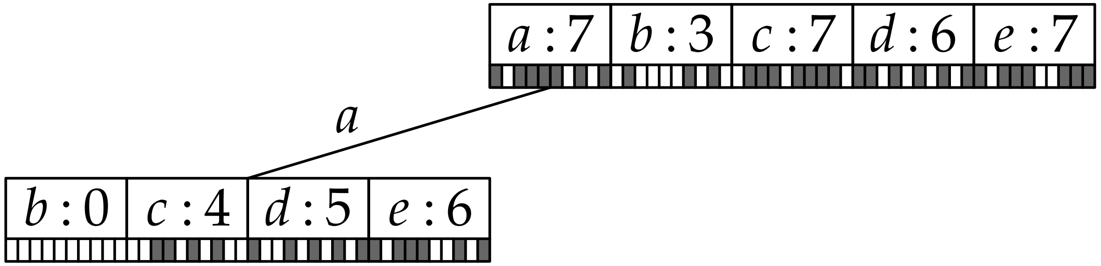
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Form a transaction list for each item. Here: bit array representation.
  - gray: item is contained in transaction
  - white: item is not contained in transaction
- Transaction database is needed only once (for the single item transaction lists).

Eclat: Depth-First Search

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

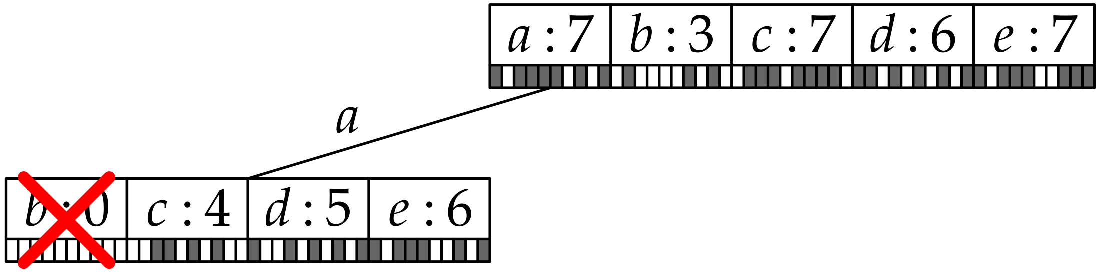


- Intersect the transaction list for item *a* with the transaction lists of all other items (*conditional database* for item *a*).

- Count the number of bits that are set (number of containing transactions). This yields the support of all item sets with the prefix *a*.

Eclat: Depth-First Search

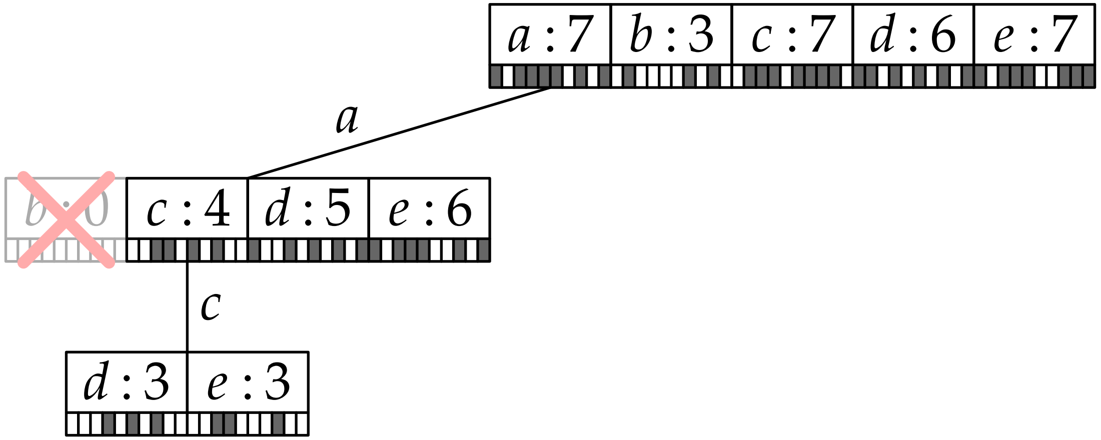
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The item set $\{a, b\}$ is infrequent and can be pruned.
- All other item sets with the prefix a are frequent and are therefore kept and processed recursively.

Eclat: Depth-First Search

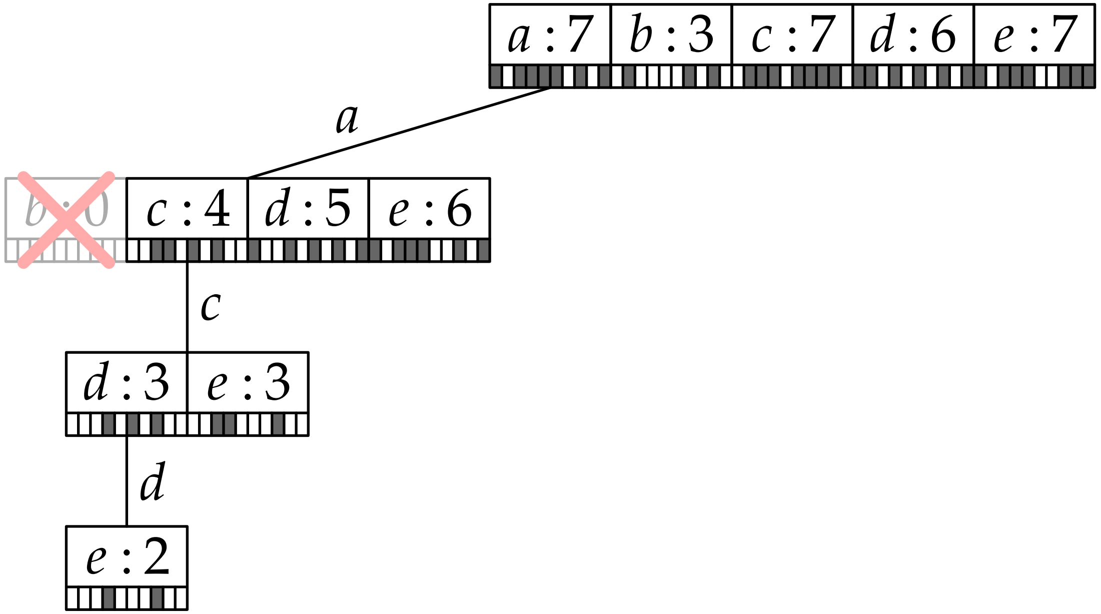
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Intersect the transaction list for the item set $\{a, c\}$ with the transaction lists of the item sets $\{a, x\}$, $x \in \{d, e\}$.
- Result: Transaction lists for the item sets $\{a, c, d\}$ and $\{a, c, e\}$.
- Count the number of bits that are set (number of containing transactions). This yields the support of all item sets with the prefix *ac*.

Eclat: Depth-First Search

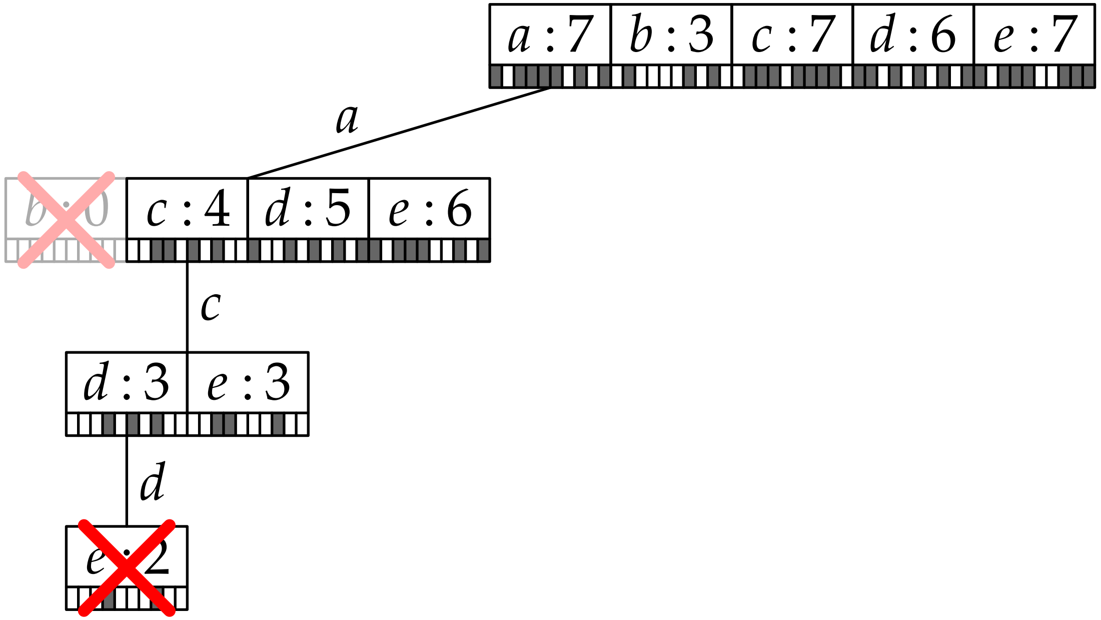
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Intersect the transaction lists for the item sets $\{a, c, d\}$ and $\{a, c, e\}$.
- Result: Transaction list for the item set $\{a, c, d, e\}$.
- With Apriori this item set could be pruned before counting, because it was known that $\{c, d, e\}$ is infrequent.

Eclat: Depth-First Search

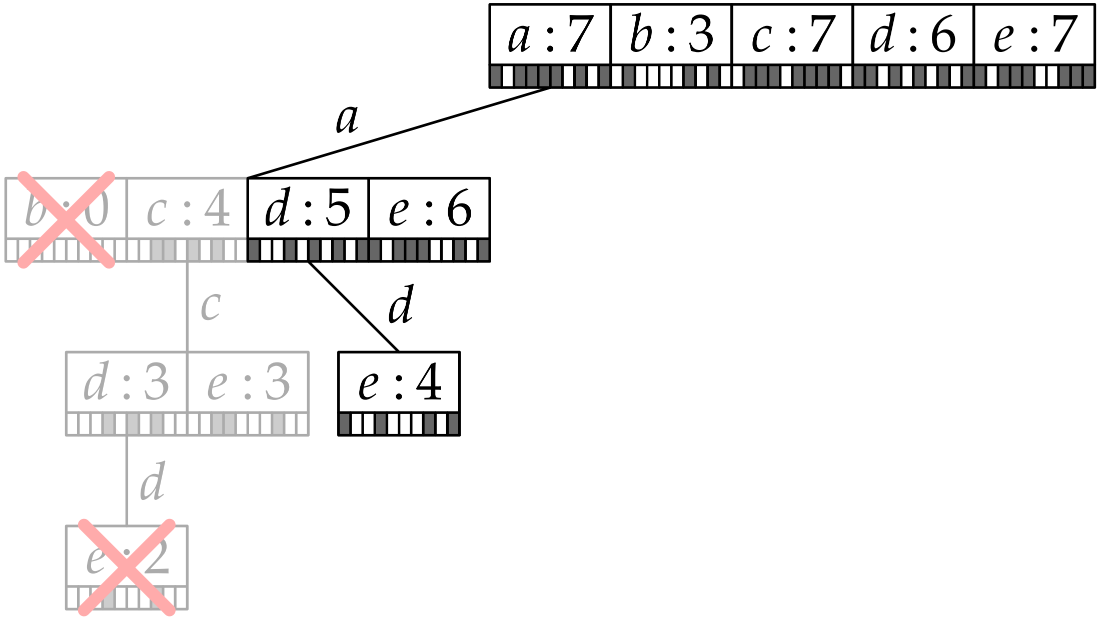
1: $\{a,d,e\}$
2: $\{b,c,d\}$
3: $\{a,c,e\}$
4: $\{a,c,d,e\}$
5: $\{a,e\}$
6: $\{a,c,d\}$
7: $\{b,c\}$
8: $\{a,c,d,e\}$
9: $\{b,c,e\}$
10: $\{a,d,e\}$



- The item set $\{a,c,d,e\}$ is not frequent (support 2/20%) and therefore pruned.
- Since there is no transaction list left (and thus no intersection possible), the recursion is terminated and the search backtracks.

Eclat: Depth-First Search

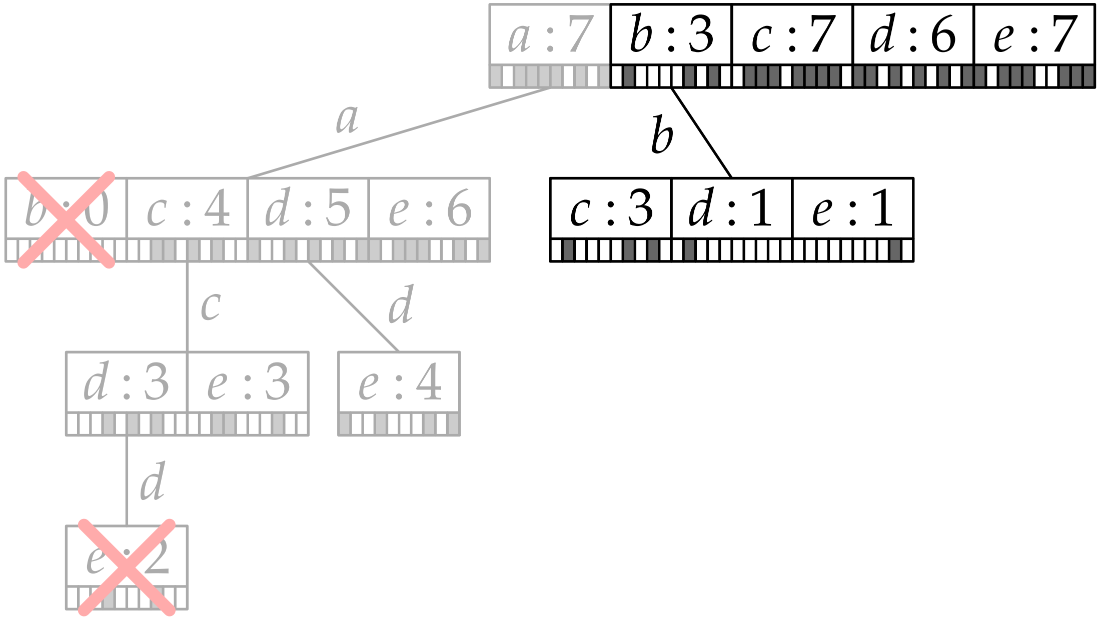
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The search backtracks to the second level of the search tree and intersects the transaction list for the item sets $\{a, d\}$ and $\{a, e\}$.
- Result: Transaction list for the item set $\{a, d, e\}$.
- Since there is only one transaction list left (and thus no intersection possible), the recursion is terminated and the search backtracks again.

Eclat: Depth-First Search

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

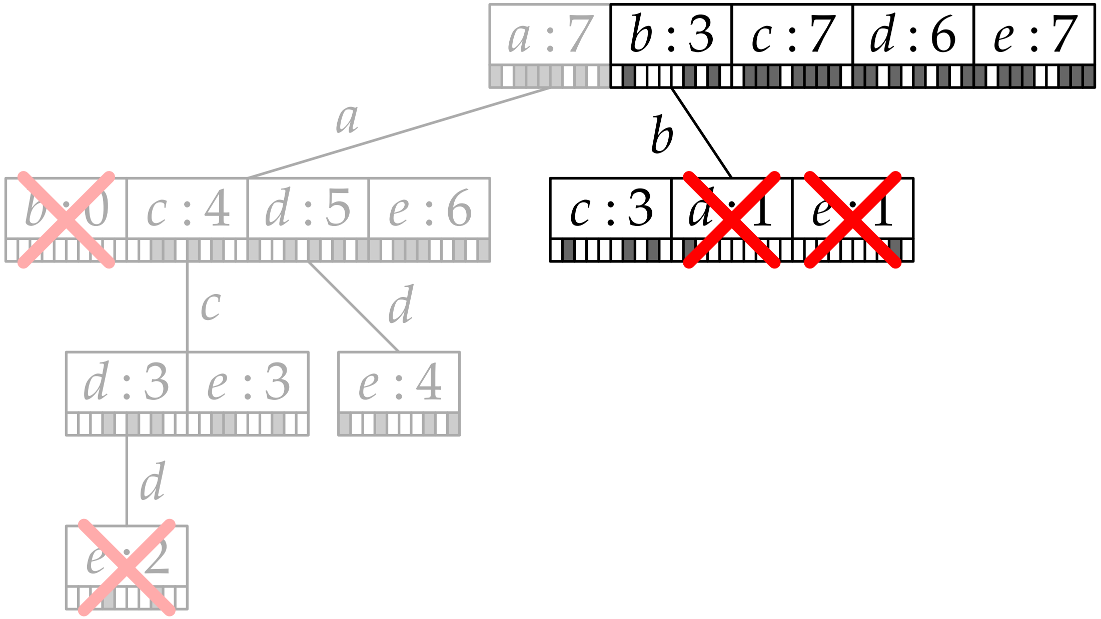


- The search backtracks to the first level of the search tree and intersects the transaction list for b with the transaction lists for c, d, and e.

- Result: Transaction lists for the item sets $\{b, c\}$, $\{b, d\}$, and $\{b, e\}$.

Eclat: Depth-First Search

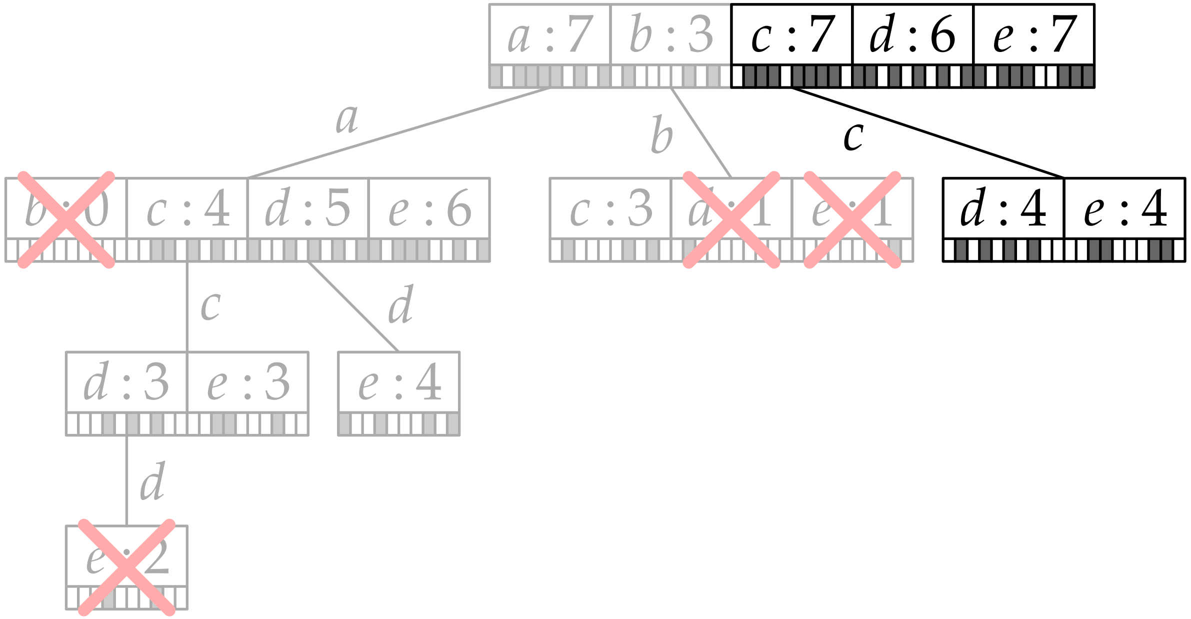
1: $\{a,d,e\}$
2: $\{b,c,d\}$
3: $\{a,c,e\}$
4: $\{a,c,d,e\}$
5: $\{a,e\}$
6: $\{a,c,d\}$
7: $\{b,c\}$
8: $\{a,c,d,e\}$
9: $\{b,c,e\}$
10: $\{a,d,e\}$



- Only one item set has sufficient support ⇒ prune all subtrees.
- Since there is only one transaction list left (and thus no intersection possible), the recursion is terminated and the search backtracks again.

Eclat: Depth-First Search

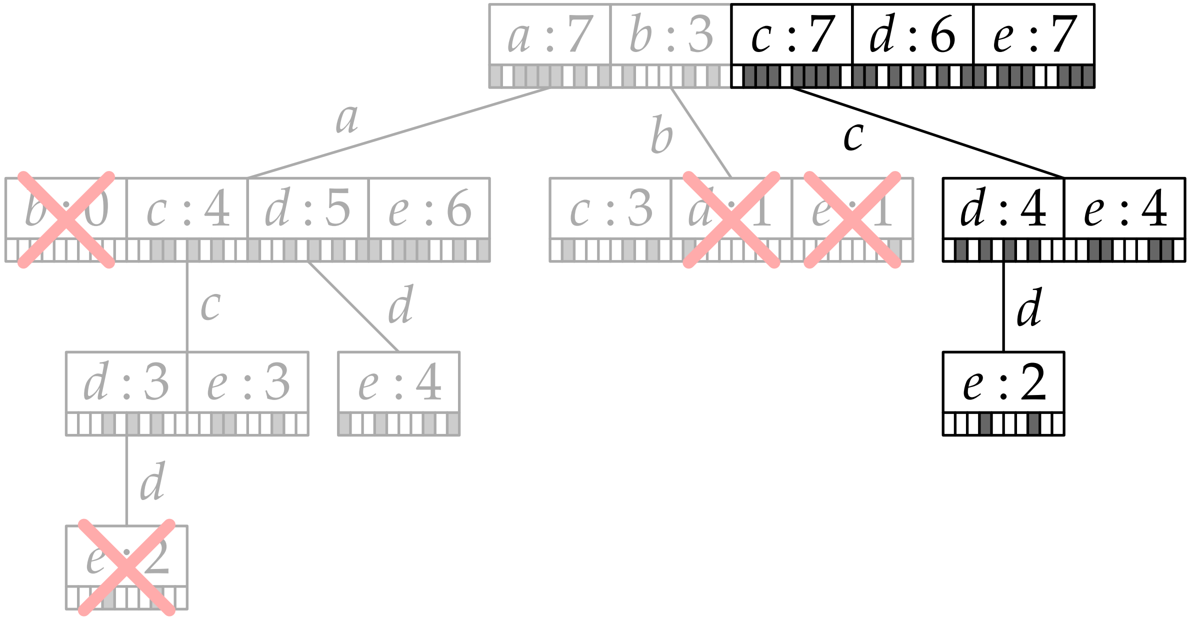
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Backtrack to the first level of the search tree and intersect the transaction list for c with the transaction lists for d and e.
- Result: Transaction lists for the item sets $\{c, d\}$ and $\{c, e\}$.

Eclat: Depth-First Search

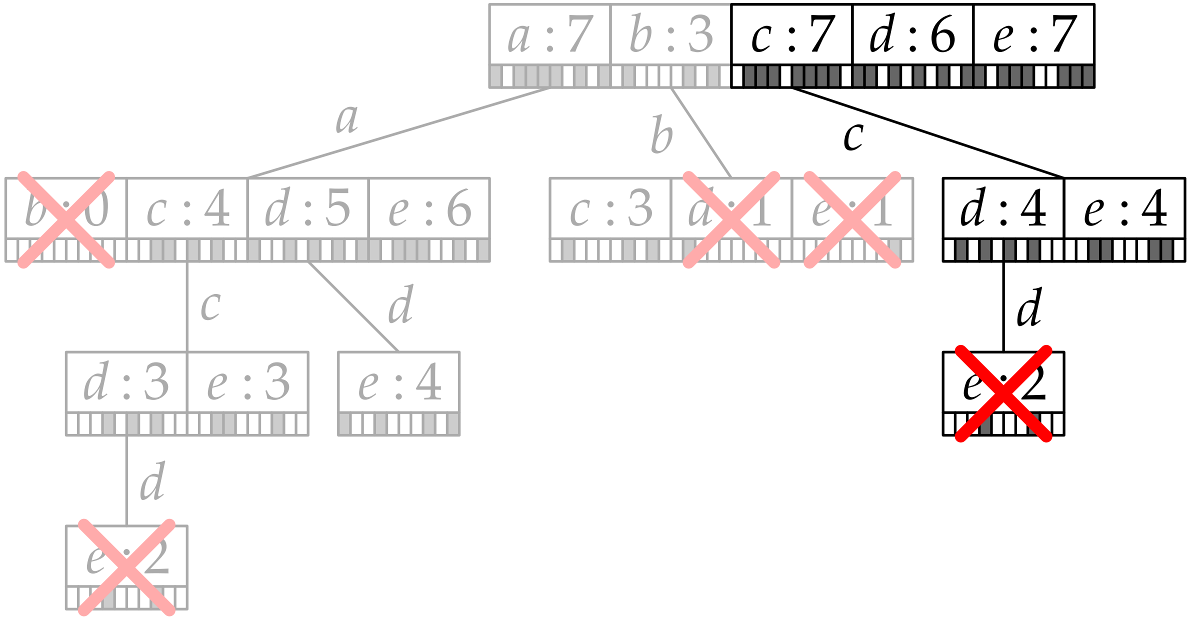
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Intersect the transaction list for the item sets $\{c, d\}$ and $\{c, e\}$.
- Result: Transaction list for the item set $\{c, d, e\}$.

Eclat: Depth-First Search

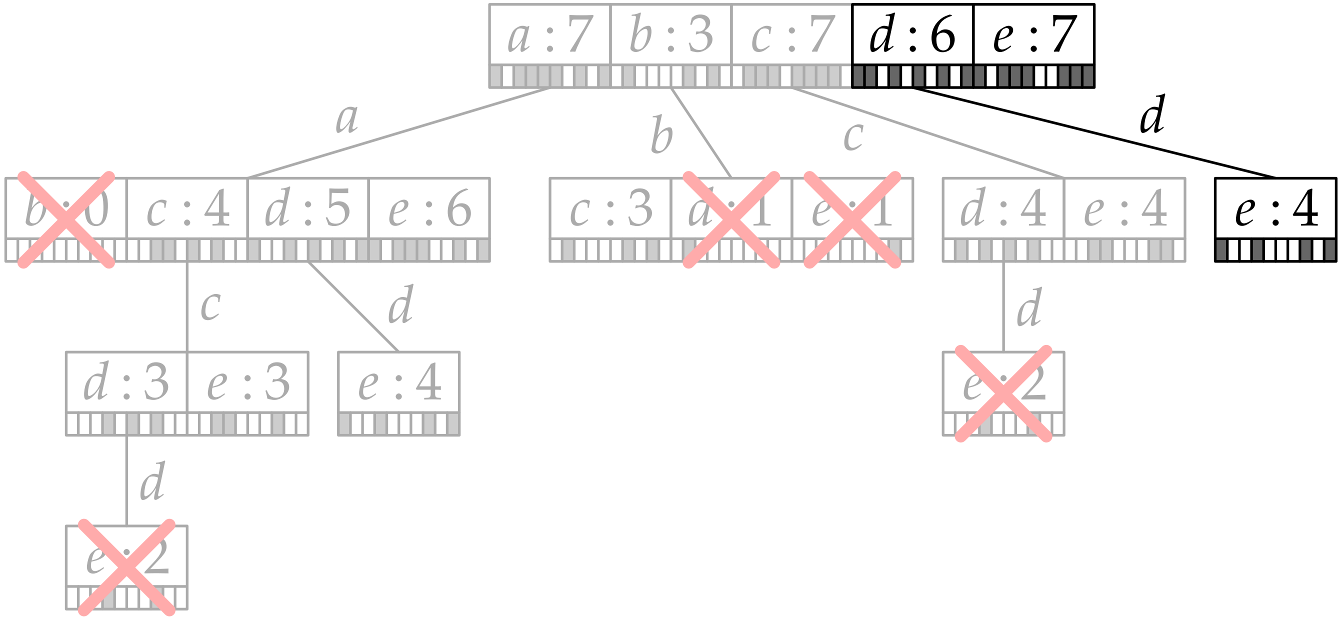
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The item set $\{c, d, e\}$ is not frequent (support 2/20%) and therefore pruned.
- Since there is no transaction list left (and thus no intersection possible), the recursion is terminated and the search backtracks.

Eclat: Depth-First Search

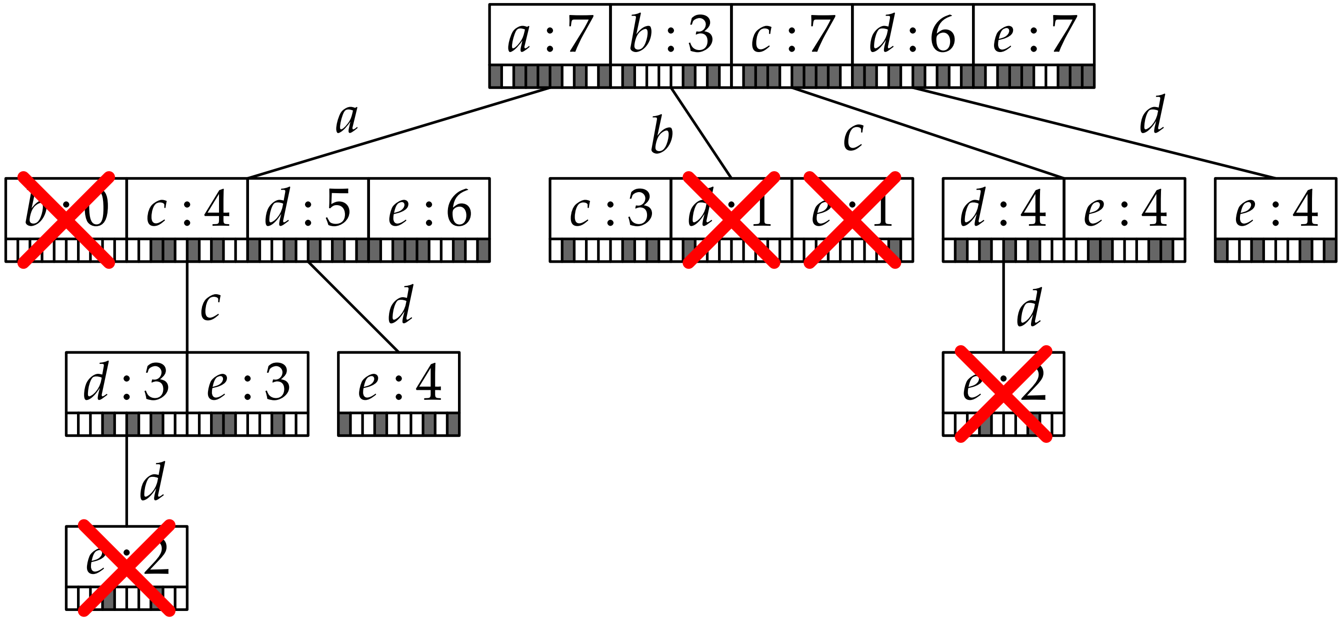
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The search backtracks to the first level of the search tree and intersects the transaction list for d with the transaction list for e.
- Result: Transaction list for the item set $\{d, e\}$.
- With this step the search is completed.

Eclat: Depth-First Search

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

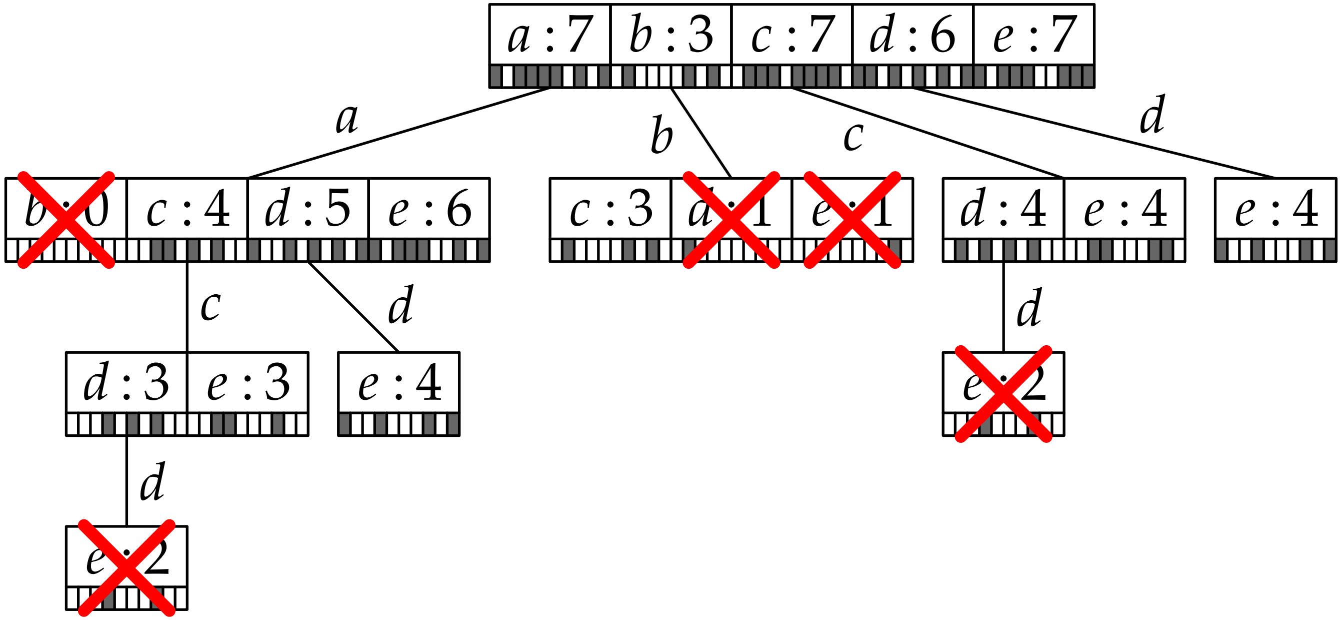


- The found frequent item sets coincide, of course, with those found by the Apriori algorithm.

- However, a fundamental difference is that Eclat usually only writes found frequent item sets to an output file, while Apriori keeps the whole search tree in main memory.

Eclat: Depth-First Search

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Note that the item set $\{a, c, d, e\}$ could be pruned by Apriori without computing its support, because the item set $\{c, d, e\}$ is infrequent.
- The same can be achieved with Eclat if the depth-first traversal of the prefix tree is carried out from right to left *and* computed support values are stored.
  It is debatable whether the potential gains justify the memory requirement.

Eclat: Representing Transaction Identifier Lists

Bit Matrix Representations

- Represent transactions as a bit matrix:
  - Each column corresponds to an item.
  - Each row corresponds to a transaction.
- Normal and sparse representation of bit matrices:
  - Normal: one memory bit per matrix bit
    (zeros are represented).
  - Sparse : lists of row indices of set bits (transaction identifier lists).
    (zeros are not represented)
- Which representation is preferable depends on
  the ratio of set bits to cleared bits.
- In most cases a sparse representation is preferable,
  because the intersections clear more and more bits.

Eclat: Intersecting Transaction Lists

```
function isect (src1, src2 : tidlist)
begin                                    (∗ — intersect two transaction id lists ∗)
  var dst : tidlist;                     (∗ created intersection ∗)
  while both src1 and src2 are not empty do begin
    if     head(src1) < head(src2)       (∗ skip transaction identifiers that are ∗)
    then   src1 = tail(src1);            (∗ unique to the first source list ∗)
    elseif head(src1) > head(src2)       (∗ skip transaction identifiers that are ∗)
    then   src2 = tail(src2);            (∗ unique to the second source list ∗)
    else begin                           (∗ if transaction id is in both sources ∗)
      dst.append(head(src1));            (∗ append it to the output list ∗)
      src1 = tail(src1); src2 = tail(src2);
    end;                                 (∗ remove the transferred transaction id ∗)
  end;                                   (∗ from both source lists ∗)
  return dst;                            (∗ return the created intersection ∗)
end;   (∗ function isect() ∗)
```

Eclat: Filtering Transaction Lists

```
function filter (transdb : list of tidlist)
begin                                   (∗ — filter a transaction database ∗)
  var condb : list of tidlist;          (∗ created conditional transaction database ∗)
      out : tidlist;                    (∗ filtered tidlist of other item ∗)
  for tid in head(transdb) do           (∗ traverse the tidlist of the split item ∗)
    contained[tid] := true;             (∗ and set flags for contained tids ∗)
  for inp in tail(transdb) do begin     (∗ traverse tidlists of the other items ∗)
    out := new tidlist;                 (∗ create an output tidlist and ∗)
    condb.append(out);                  (∗ append it to the conditional database ∗)
    for tid in inp do                   (∗ collect tids shared with split item ∗)
      if contained[tid] then out.append(tid);
  end                                   (∗ (“contained” is a global boolean array) ∗)
  for tid in head(transdb) do           (∗ traverse the tidlist of the split item ∗)
    contained[tid] := false;            (∗ and clear flags for contained tids ∗)
  return condb;                         (∗ return the created conditional database ∗)
end;     (∗ function filter() ∗)
```

Eclat: Item Order

Consider **Eclat with transaction identifier lists** (sparse representation):

- Each computation of a conditional transaction database intersects the transaction list for an item (let this be list L) with all transaction lists for items following in the item order.

- The lists resulting from the intersections cannot be longer than the list L. (This is another form of the fact that support is anti-monotone.)

- If the **items are processed in the order of increasing frequency** (that is, if they are chosen as split items in this order):

  - Short lists (less frequent items) are intersected with many other lists, creating a conditional transaction database with many short lists.

  - Longer lists (more frequent items) are intersected with few other lists, creating a conditional transaction database with few long lists.

- Consequence: The average size of conditional transaction databases is reduced, which leads to **faster processing / search**.

Eclat: Item Order

a	*b*	*c*	*d*	*e*
7	3	7	6	7
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

b	*c*	*d*	*e*
0	4	5	6
	3	1	1
	4	4	3
	6	6	4
	8	8	5
		10	8
			10

↑ Conditional database for prefix *a* (1st subproblem)

b	*c*	*d*	*e*
3	7	6	7
2	2	1	1
7	3	2	3
9	4	4	4
	6	6	5
	7	8	8
	8	10	9
	9		10

← Conditional database with item *a* removed (2nd subproblem)

b	*d*	*a*	*c*	*e*
3	6	7	7	7
2	1	1	2	1
7	2	3	3	3
9	4	4	4	4
	6	5	6	5
	8	6	7	8
	10	8	8	9
		10	9	10

d	*a*	*c*	*e*
1	0	3	1
2		2	9
		7	
		9	

↑ Conditional database for prefix *b* (1st subproblem)

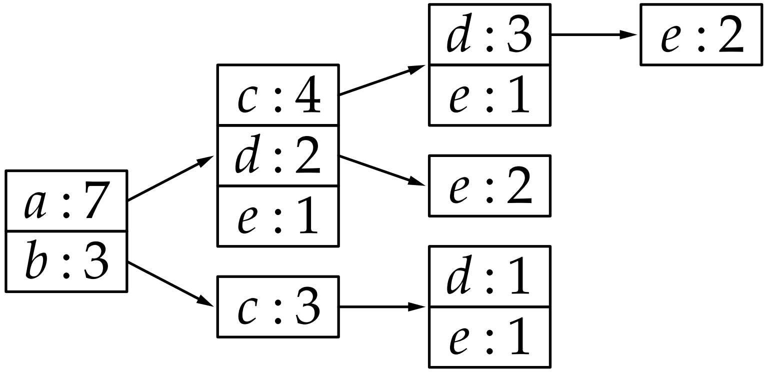
d	*a*	*c*	*e*
6	7	7	7
1	1	2	1
2	3	3	3
4	4	4	4
6	5	6	5
8	6	7	8
10	8	8	9
	10	9	10

← Conditional database with item *b* removed (2nd subproblem)

Reminder (Apriori): Transactions as a Prefix Tree

transaction database	lexicographically sorted
a, d, e	a, c, d
b, c, d	a, c, d, e
a, c, e	a, c, d, e
a, c, d, e	a, c, e
a, e	a, d, e
a, c, d	a, d, e
b, c	a, e
a, c, d, e	b, c
b, c, e	b, c, d
a, d, e	b, c, e

prefix tree representation

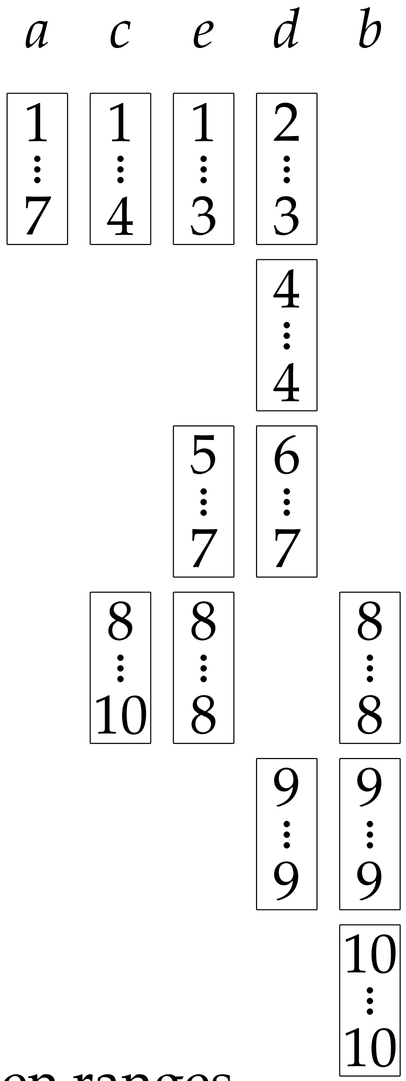


- Items in transactions are sorted w.r.t. some arbitrary order, transactions are sorted lexicographically, then a prefix tree is constructed.

- **Advantage:** identical transaction prefixes are processed only once.

Eclat: Transaction Ranges

transaction database	item frequencies	sorted by frequency	lexicographically sorted
a, d, e	a: 7	a, e, d	1: a, c, e
b, c, d	b: 3	c, d, b	2: a, c, e, d
a, c, e	c: 7	a, c, e	3: a, c, e, d
a, c, d, e	d: 6	a, c, e, d	4: a, c, d
a, e	e: 7	a, e	5: a, e
a, c, d		a, c, d	6: a, e, d
b, c		c, b	7: a, e, d
a, c, d, e		a, c, e, d	8: c, e, b
b, c, e		c, e, b	9: c, d, b
a, d, e		a, e, d	10: c, b

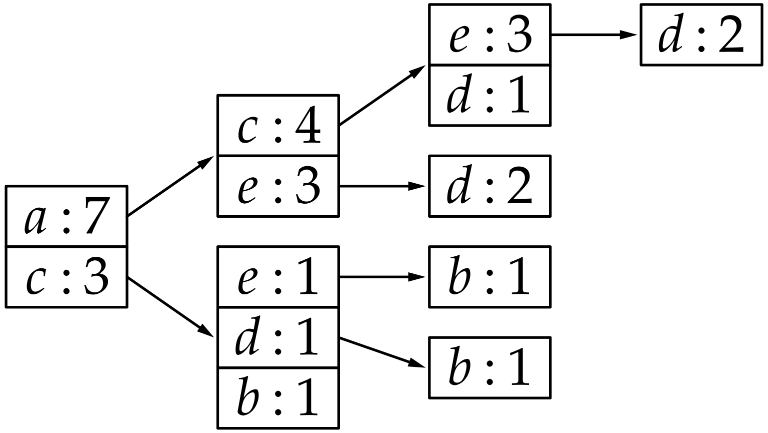


- The transaction lists can be compressed by combining consecutive transaction identifiers into ranges.
- Exploit item frequencies and ensure subset relations between ranges from lower to higher frequencies, so that intersecting the lists is easy.

Eclat: Transaction Ranges / Prefix Tree

transaction database	sorted by frequency	lexicographically sorted
a, d, e	a, e, d	1: a, c, e
b, c, d	c, d, b	2: a, c, e, d
a, c, e	a, c, e	3: a, c, e, d
a, c, d, e	a, c, e, d	4: a, c, d
a, e	a, e	5: a, e
a, c, d	a, c, d	6: a, e, d
b, c	c, b	7: a, e, d
a, c, d, e	a, c, e, d	8: c, e, b
b, c, e	c, e, b	9: c, d, b
a, d, e	a, e, d	10: c, b

prefix tree representation



- Items in transactions are sorted by descending frequency, transactions are sorted lexicographically, then a prefix tree is constructed.
- The transaction ranges reflect the structure of this prefix tree.

Eclat: Difference sets (Diffsets)

- In a conditional database, all transaction lists are “filtered” by the prefix: Only transactions contained in the transaction identifier list for the prefix can be in the transaction identifier lists of the conditional database.

- This suggests the idea to use **diffsets** to represent conditional databases:

$$\forall I : \forall a \notin I : \quad D_T(a \mid I) = K_T(I) - K_T(I \cup \{a\})$$

  $D_T(a \mid I)$ contains the identifiers of the transactions that contain I but not a.

- The support of direct supersets of I can now be computed as

$$\forall I : \forall a \notin I : \quad s_T(I \cup \{a\}) = s_T(I) - |D_T(a \mid I)|.$$

  The diffsets for the next level can be computed by

$$\forall I : \forall a, b \notin I, a \neq b : \quad D_T(b \mid I \cup \{a\}) = D_T(b \mid I) - D_T(a \mid I)$$

- For some transaction databases, diffsets speed up the search considerably.

Eclat: Diffsets

Proof of the Formula for the Next Level:

$$\begin{aligned}
D_T(b \mid I \cup \{a\}) &= K_T(I \cup \{a\}) - K_T(I \cup \{a,b\}) \\
&= \{k \mid I \cup \{a\} \subseteq t_k\} - \{k \mid I \cup \{a,b\} \subseteq t_k\} \\
&= \{k \mid I \subseteq t_k \wedge a \in t_k\} \\
&\qquad -\{k \mid I \subseteq t_k \wedge a \in t_k \wedge b \in t_k\} \\
&= \{k \mid I \subseteq t_k \wedge a \in t_k \wedge b \notin t_k\} \\
&= \{k \mid I \subseteq t_k \wedge b \notin t_k\} \\
&\qquad -\{k \mid I \subseteq t_k \wedge b \notin t_k \wedge a \notin t_k\} \\
&= \{k \mid I \subseteq t_k \wedge b \notin t_k\} \\
&\qquad -\{k \mid I \subseteq t_k \wedge a \notin t_k\} \\
&= (\{k \mid I \subseteq t_k\} - \{k \mid I \cup \{b\} \subseteq t_k\}) \\
&\qquad -(\{k \mid I \subseteq t_k\} - \{k \mid I \cup \{a\} \subseteq t_k\}) \\
&= (K_T(I) - K_T(I \cup \{b\}) \\
&\qquad -(K_T(I) - K_T(I \cup \{a\}) \\
&= D(b \mid I) - D(a \mid I)
\end{aligned}$$

Summary Eclat

Basic Processing Scheme

- Depth-first traversal of the prefix tree (divide-and-conquer scheme).
- Data is represented as lists of transaction identifiers (one per item).
- Support counting is done by intersecting lists of transaction identifiers.

Advantages

- Depth-first search reduces memory requirements.
- Usually (considerably) faster than Apriori.

Disadvantages

- With a sparse transaction list representation (row indices) intersections are difficult to execute for modern processors (branch prediction).

Software

- `http://www.borgelt.net/eclat.html`

The LCM Algorithm

Linear Closed Item Set Miner
[Uno, Asai, Uchida, and Arimura 2003] (version 1)
[Uno, Kiyomi and Arimura 2004, 2005] (versions 2 & 3)

LCM: Basic Ideas

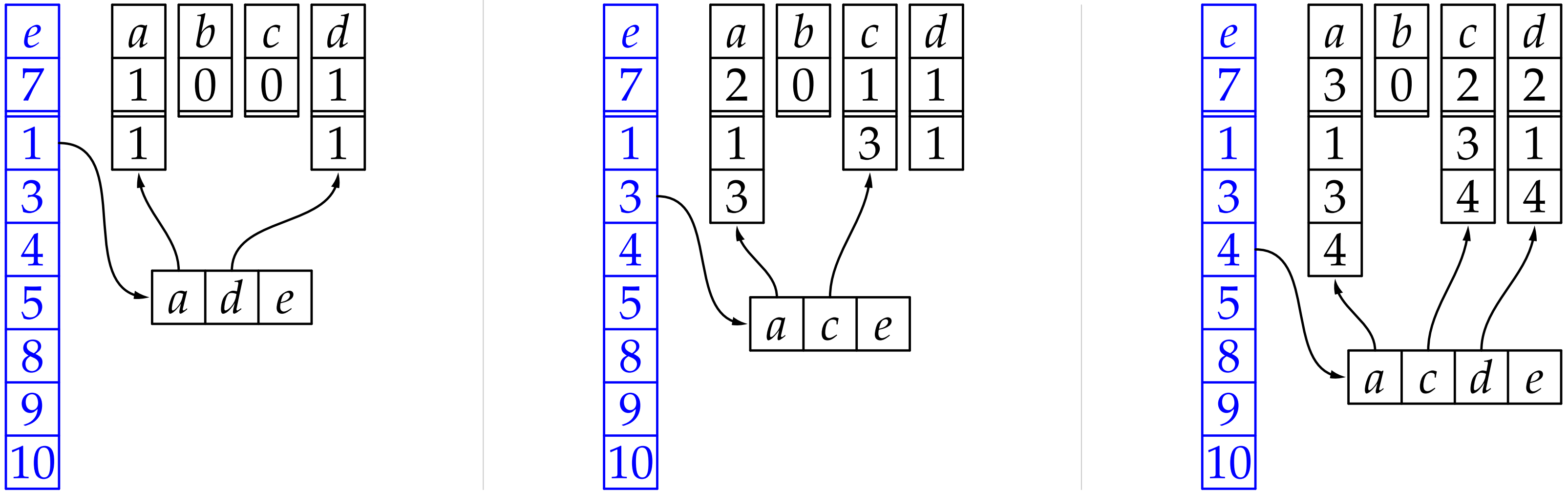
- The item sets are checked in **lexicographic order** (**depth-first traversal** of the prefix tree).
- Standard divide-and-conquer scheme (include/exclude items); recursive processing of the conditional transaction databases.
- Closely related to the Eclat algorithm.
- Maintains **both a horizontal and a vertical representation** of the transaction database in parallel.
  - Uses the vertical representation to filter the transactions with the chosen split item.
  - Uses the horizontal representation to fill the vertical representation for the next recursion step (no intersection as in Eclat).
- Usually traverses the search tree from **right to left** in order to reuse the memory for the vertical representation (fixed memory requirement, proportional to database size).

LCM: Occurrence Deliver

1:	*a*	*d*	*e*	
2:	*b*	*c*	*d*	
3:	*a*	*c*	*e*	
4:	*a*	*c*	*d*	*e*
5:	*a*	*e*		
6:	*a*	*c*	*d*	
7:	*b*	*c*		
8:	*a*	*c*	*d*	*e*
9:	*b*	*c*	*e*	
10:	*a*	*d*	*e*	

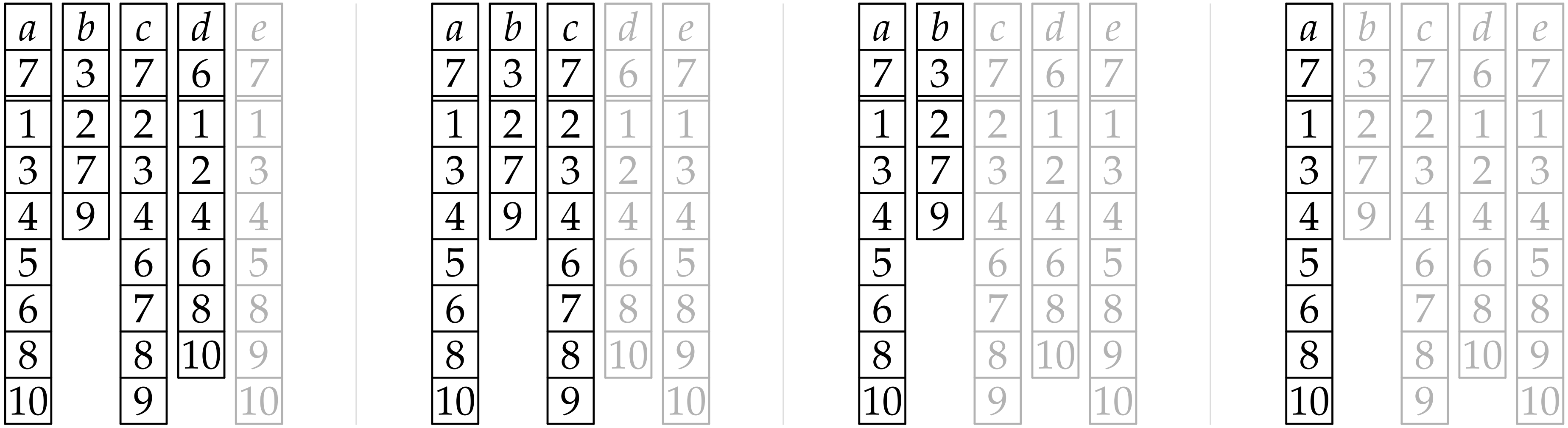
a	*b*	*c*	*d*	*e*
7	3	7	6	7
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

Occurrence deliver scheme used by LCM to find the conditional transaction database for the first subproblem (needs a horizontal representation in parallel).



etc.

LCM: Solve 2nd Subproblem before 1st



a	*b*	*c*	*d*	*e*
7	3	7	6	7
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

a	*b*	*c*	*d*	*e*
7	3	7	6	7
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

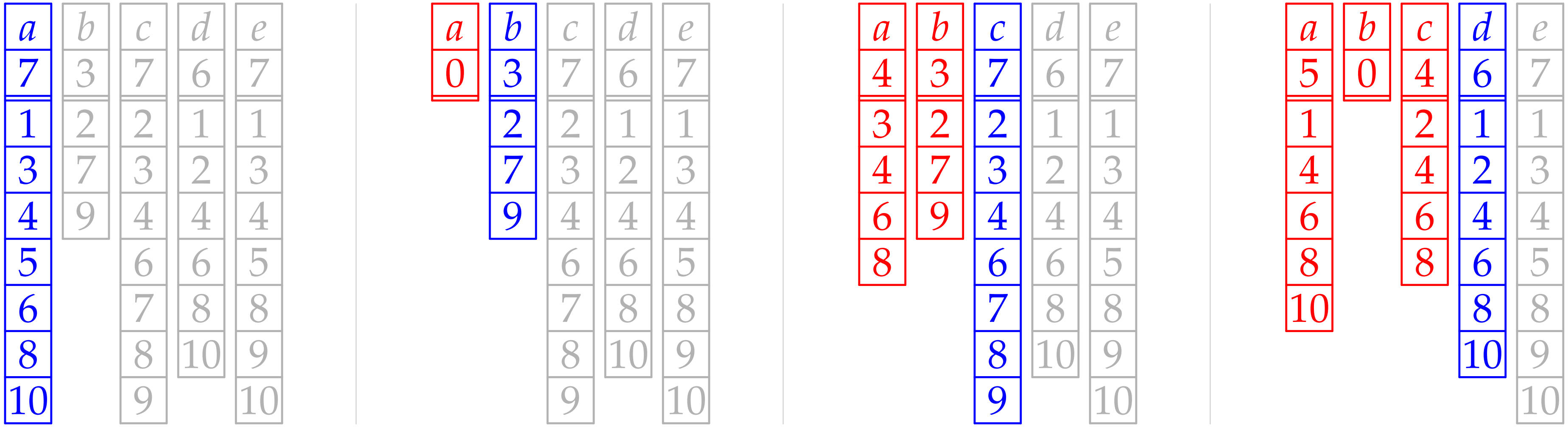
a	*b*	*c*	*d*	*e*
7	3	7	6	7
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

a	*b*	*c*	*d*	*e*
7	3	7	6	7
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

gray: excluded item (2nd subproblem first) black: data needed for 2nd subproblem

- The second subproblem (exclude split item) is solved before the first subproblem (include split item).
- The algorithm is executed only on the memory that stores the initial vertical representation (plus the horizontal representation).
- If the transaction database can be loaded, the frequent item sets can be found.

LCM: Solve 2nd Subproblem before 1st



gray: unprocessed part　　blue: split item　　red: conditional database

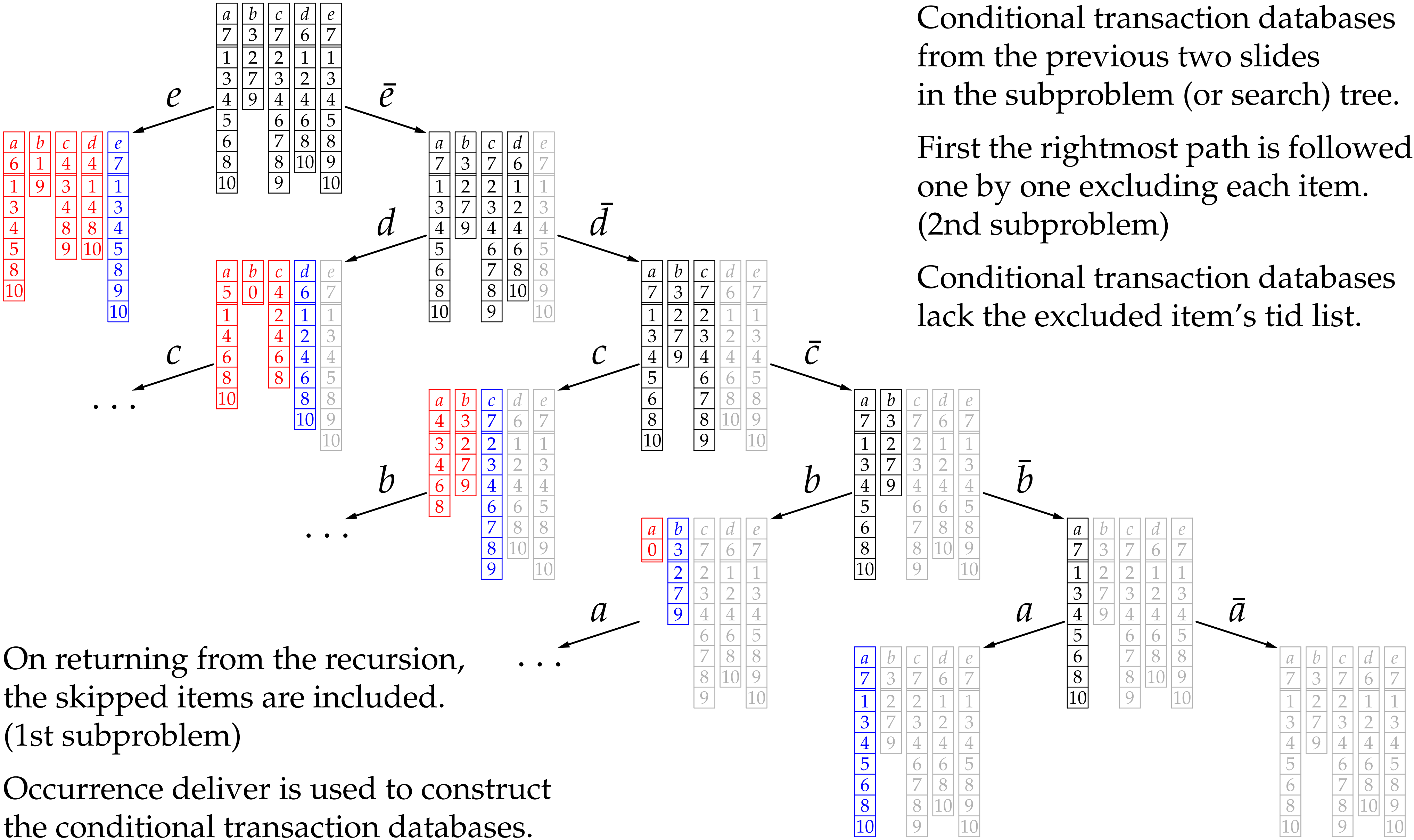
- The second subproblem (exclude split item) is solved before the first subproblem (include split item).
- The algorithm is executed only on the memory that stores the initial vertical representation (plus the horizontal representation).
- If the transaction database can be loaded, the frequent item sets can be found.

LCM: Solve 2nd Subproblem before 1st

Conditional transaction databases
from the previous two slides
in the subproblem (or search) tree.

First the rightmost path is followed,
one by one excluding each item.
(2nd subproblem)

Conditional transaction databases
lack the excluded item’s tid list.



On returning from the recursion,
the skipped items are included.
(1st subproblem)

Occurrence deliver is used to construct
the conditional transaction databases.

Summary LCM

Basic Processing Scheme

- Depth-first traversal of the prefix tree (divide-and-conquer scheme).
- Parallel horizontal and vertical transaction representation.
- Support counting is done during the occurrence deliver process.

Advantages

- Fairly simple data structure and processing scheme.
- Very fast if implemented properly (and with additional tricks).

Disadvantages

- Simple, straightforward implementation is relatively slow.

Software

- `http://www.borgelt.net/eclat.html` (option `-Ao`)

The SaM Algorithm

Split and Merge Algorithm [Borgelt 2008]

SaM: Basic Ideas

- The item sets are checked in **lexicographic order** (**depth-first traversal** of the prefix tree).
- Standard divide-and-conquer scheme (include/exclude items).
- Recursive processing of the conditional transaction databases.
- While Eclat uses a purely vertical transaction representation, SaM uses a purely **horizontal transaction representation**.

  This demonstrates that the traversal order for the prefix tree and the representation form of the transaction database can be combined freely.
- The data structure used is a simply array of transactions.
- The two conditional databases for the two subproblems formed in each step are created with a **split step** and a **merge step**.

  Due to these steps the algorithm is called Split and Merge (SaM).

SaM: Preprocessing the Transaction Database

① $a\ d$
$a\ c\ d\ e$
$b\ d$
$b\ c\ d\ g$
$b\ c\ f$
$a\ b\ d$
$b\ d\ e$
$b\ c\ d\ e$
$b\ c$
$a\ b\ d\ f$

② g: 1
f: 2

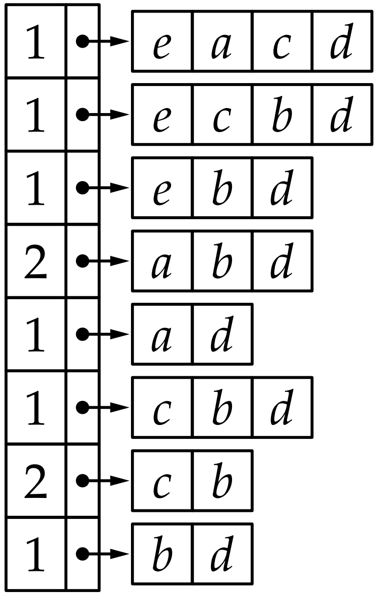
e: 3
a: 4
c: 5
b: 8
d: 8

$s_{\min} = 3$

③ $a\ d$
$e\ a\ c\ d$
$b\ d$
$c\ b\ d$
$c\ b$
$a\ b\ d$
$e\ b\ d$
$e\ c\ b\ d$
$c\ b$
$a\ b\ d$

④ $e\ a\ c\ d$
$e\ c\ b\ d$
$e\ b\ d$
$a\ b\ d$
$a\ b\ d$
$a\ d$
$c\ b\ d$
$c\ b$
$c\ b$
$b\ d$

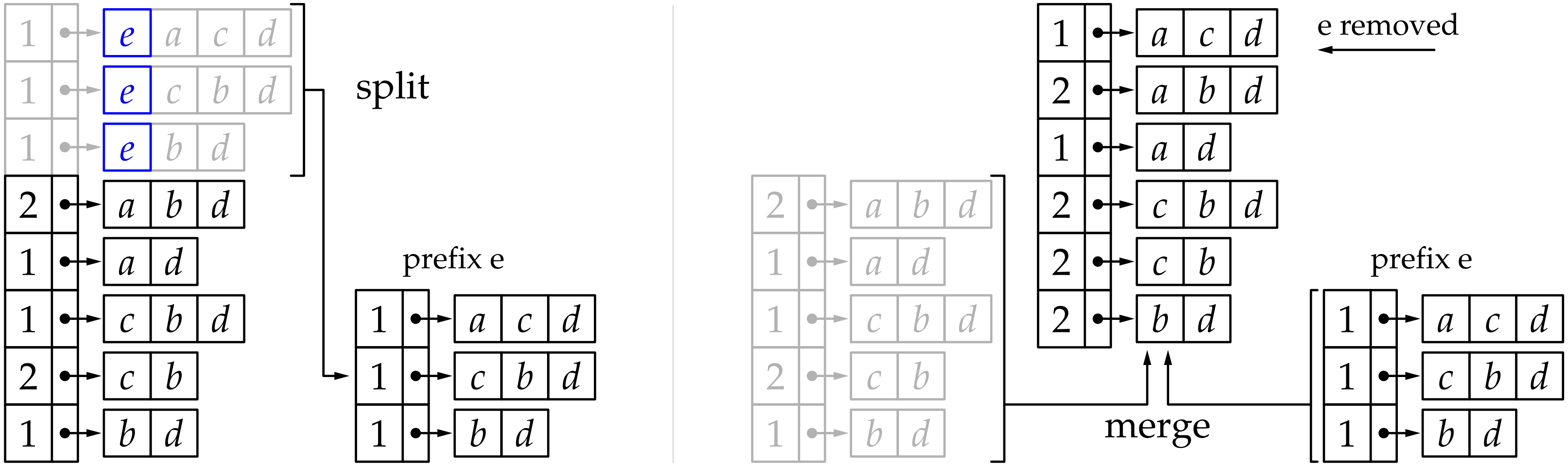
⑤



1 → $e\ a\ c\ d$
1 → $e\ c\ b\ d$
1 → $e\ b\ d$
2 → $a\ b\ d$
1 → $a\ d$
1 → $c\ b\ d$
2 → $c\ b$
1 → $b\ d$

1. Original transaction database.
2. Frequency of individual items.
3. Items in transactions sorted ascendingly w.r.t. their frequency.
4. Transactions sorted lexicographically in descending order (comparison of items inverted w.r.t. preceding step).
5. Data structure used by the algorithm.

SaM: Basic Operations



- **Split Step:** (on the left; for first subproblem)
  - Move all transactions starting with the same item to a new array.
  - Remove the common leading item (advance pointer into transaction).
- **Merge Step:** (on the right; for second subproblem)
  - Merge the remainder of the transaction array and the copied transactions.
  - The merge operation is similar to a *mergesort* phase.

SaM: Pseudo-Code

```
function SaM (a: array of transactions,        (∗ conditional database to process ∗)
              p: set of items,                 (∗ prefix of the conditional database a ∗)
              $s_{\min}$: int)                 (∗ minimum support of an item set ∗)
var i: item;                                   (∗ buffer for the split item ∗)
    b: array of transactions;                  (∗ split result ∗)
begin                                          (∗ — split and merge recursion — ∗)
  while a is not empty do                      (∗ while the database is not empty ∗)
    i := a[0].items[0];                        (∗ get leading item of first transaction ∗)
    move transactions starting with i to b;    (∗ split   step: first    subproblem ∗)
    merge b and the remainder of a into a;     (∗ merge step: second subproblem ∗)
    if $s(i) \geq s_{\min}$ then               (∗ if the split item is frequent: ∗)
      $p := p \cup \{i\}$;                     (∗ extend the prefix item set and ∗)
      report p with support $s(i)$;            (∗ report the found frequent item set ∗)
      SaM($b, p, s_{\min}$);                   (∗ process the split result recursively, ∗)
      $p := p - \{i\}$;                        (∗ then restore the original prefix ∗)
    end;
  end;
end;   (∗ function SaM() ∗)
```

SaM: Pseudo-Code — Split Step

```
var $i$: item;                                   (* buffer for the split item *)
    $s$: int;                                    (* support of the split item *)
    $b$: array of transactions;                  (* split result *)
begin                                            (* — split step *)
  $b$ := empty; $s$ := 0;                        (* initialize split result and item support *)
  $i$ := $a[0]$.items[0];                        (* get leading item of first transaction *)
  while $a$ is not empty                         (* while database is not empty and *)
  and   $a[0]$.items[0] = $i$ do                 (* next transaction starts with same item *)
    $s := s + a[0]$.wgt;                         (* sum occurrences (compute support) *)
    remove $i$ from $a[0]$.items;                (* remove split item from transaction *)
    if    $a[0]$.items is not empty              (* if transaction has not become empty *)
    then remove $a[0]$ from $a$ and append it to $b$;
    else  remove $a[0]$ from $a$; end;           (* move it to the conditional database, *)
  end;                                           (* otherwise simply remove it: *)
end;                                             (* empty transactions are eliminated *)
```

- Note that the split step also determines the support of the item i.

SaM: Pseudo-Code — Merge Step

```
var c: array of transactions;                  (* buffer for remainder of source array *)
begin                                          (* — merge step *)
  c := a; a := empty;                          (* initialize the output array *)
  while b and c are both not empty do          (* merge split and remainder of database *)
    if      c[0].items > b[0].items            (* copy lex. smaller transaction from c *)
    then    remove c[0] from c and append it to a;
    else if c[0].items < b[0].items            (* copy lex. smaller transaction from b *)
    then    remove b[0] from b and append it to a;
    else    b[0].wgt := b[0].wgt +c[0].wgt;        (* sum the occurrences/weights *)
            remove b[0] from b and append it to a;
            remove c[0] from c;                (* move combined transaction and *)
    end;                                       (* delete the other, equal transaction: *)
  end;                                         (* keep only one copy per transaction *)
  while c is not empty do                      (* copy remaining transactions in c *)
    remove c[0] from c and append it to a; end;
  while b is not empty do                      (* copy remaining transactions in b *)
    remove b[0] from b and append it to a; end;
end;                                           (* second recursion: executed by loop *)
```

SaM: Optimization

- If the transaction database is sparse,
  the two transaction arrays to merge can differ substantially in size.

- In this case SaM can become fairly slow,
  because the merge step processes many more transactions than the split step.

- Intuitive explanation (extreme case):

  - Suppose *mergesort* always merged a single element
    with the recursively sorted remainder of the array (or list).

  - This version of mergesort would be equivalent to *insertion sort*.

  - As a consequence the time complexity worsens from $O(n \log n)$ to $O(n^2)$.

- Possible optimization:

  - Modify the merge step if the arrays to merge differ significantly in size.

  - Idea: use the same optimization as in **binary search** based *insertion sort*.

SaM: Pseudo-Code — Binary Search Based Merge

```
function merge ($a, b$: array of transactions) : array of transactions
var $l, m, r$: int;                                 (* binary search variables *)
    $c$: array of transactions;                     (* output transaction array *)
begin                                               (* — binary search based merge — *)
  $c$ := empty;                                     (* initialize the output array *)
  while $a$ and $b$ are both not empty do           (* merge the two transaction arrays *)
    $l := 0$; $r :=$ length($a$);                   (* initialize the binary search range *)
    while $l < r$ do                                (* while the search range is not empty *)
      $m := \lfloor \frac{l+r}{2} \rfloor$;         (* compute the middle index *)
      if    $a[m] < b[0]$                           (* compare the transaction to insert *)
      then $l := m + 1$; else $r := m$;             (* and adapt the binary search range *)
    end;                                            (* according to the comparison result *)
    while $l > 0$ do                                (* while still before insertion position *)
      remove $a[0]$ from $a$ and append it to $c$;
      $l := l - 1$;                                 (* copy lex. larger transaction and *)
    end;                                            (* decrement the transaction counter *)
    ...
```

SaM: Pseudo-Code — Binary Search Based Merge

```
    ...
    remove $b[0]$ from $b$ and append it to $c$;  (* copy the transaction to insert and *)
    $i$ := length($c$) − 1;                       (* get its index in the output array *)
    if   $a$ is not empty and $a[0]$.items = $c[i]$.items
    then $c[i]$.wgt = $c[i]$.wgt +$a[0]$.wgt;    (* if there is another transaction *)
         remove $a[0]$ from $a$;                 (* that is equal to the one just copied, *)
    end;                                         (* then sum the transaction weights *)
  end;                                           (* and remove trans. from the array *)
  while $a$ is not empty do                      (* copy remainder of transactions in $a$ *)
    remove $a[0]$ from $a$ and append it to $c$; end;
  while $b$ is not empty do                      (* copy remainder of transactions in $b$ *)
    remove $b[0]$ from $b$ and append it to $c$; end;
  return $c$;                                    (* return the merge result *)
end;  (* function merge() *)
```

- Applying this merge procedure if the length ratio of the transaction arrays exceeds 16:1 accelerates the execution on sparse data sets.

SaM: Optimization and External Storage

- Accepting a slightly more complicated processing scheme, one may work with **double source buffering**:
  - Initially, one source is the input database and the other source is empty.
  - A split result, which has to be created by moving and merging transactions from both sources, is always merged to the smaller source.
  - If both sources have become large, they may be merged in order to empty one source.
- Note that SaM can easily be implemented to work on **external storage**:
  - In principle, the transactions need not be loaded into main memory.
  - Even the transaction array can easily be stored on external storage or as a relational database table.
  - The fact that the transaction array is processed linearly is advantageous for external storage operations.

Summary SaM

Basic Processing Scheme

- Depth-first traversal of the prefix tree (divide-and-conquer scheme).
- Data represented as an array of transactions (purely horizontal representation).
- Support counting is done implicitly in the split step.

Advantages

- Very simple data structure and processing scheme.
- Easy to implement for operation on external storage / relational databases.

Disadvantages

- Can be slow on sparse transaction databases due to the merge step.

Software

- `http://www.borgelt.net/sam.html`

The RElim Algorithm

Recursive Elimination Algorithm [Borgelt 2005]

Recursive Elimination: Basic Ideas

- The item sets are checked in **lexicographic order** (**depth-first traversal** of the prefix tree).
- Standard divide-and-conquer scheme (include/exclude items).
- Recursive processing of the conditional transaction databases.
- Avoids the main problem of the SaM algorithm:
  does not use a merge operation to group transactions with same leading item.
- RElim rather maintains **one list of transactions per item**,
  thus employing the core idea of *radix sort*.

  However, only transactions starting with an item are in the corresponding list.
- After an item has been processed, transactions are reassigned to other lists (based on the next item in the transaction).
- RElim is in several respects similar to the LCM algorithm (as discussed before) and closely related to the H-mine algorithm (not covered in this lecture).

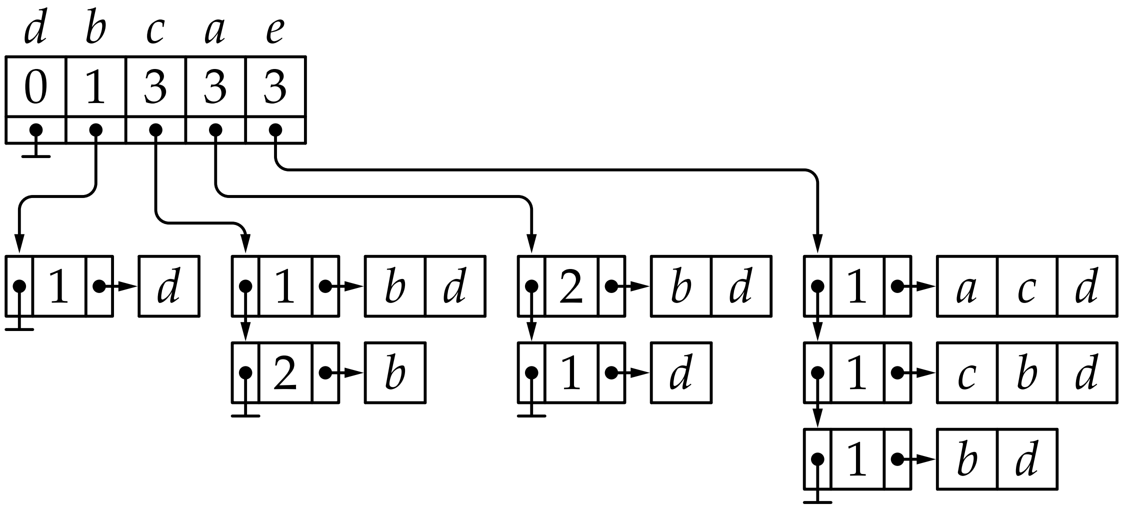
RElim: Preprocessing the Transaction Database

① · · · ③

same
as for
SaM

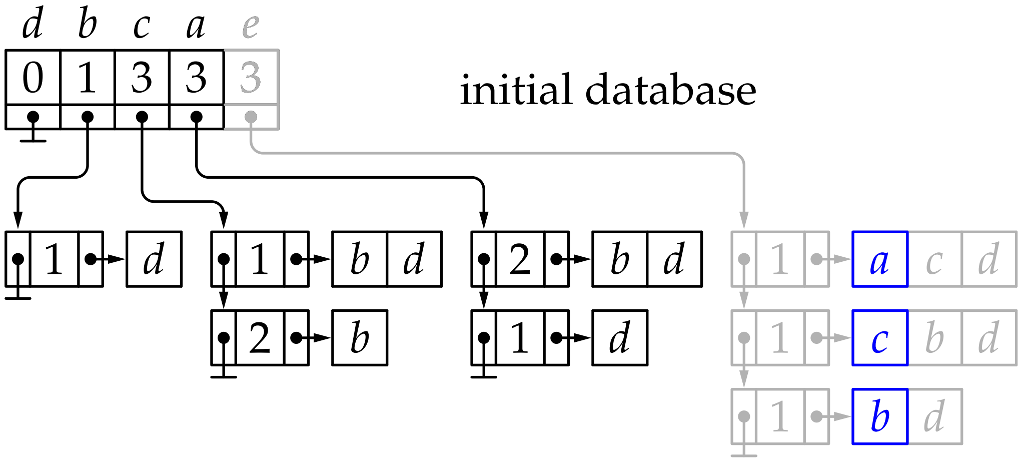
④ *e a c d*
e c b d
e b d
a b d
a b d
a d
c b d
c b
c b
b d

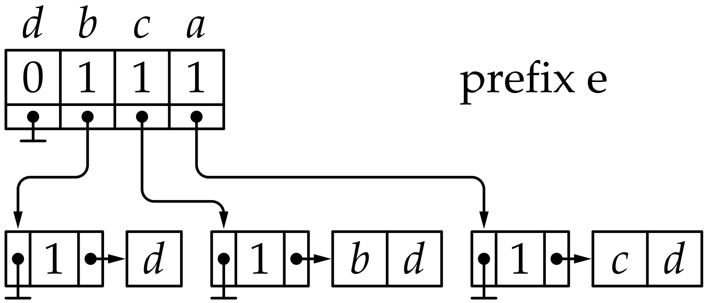
⑤

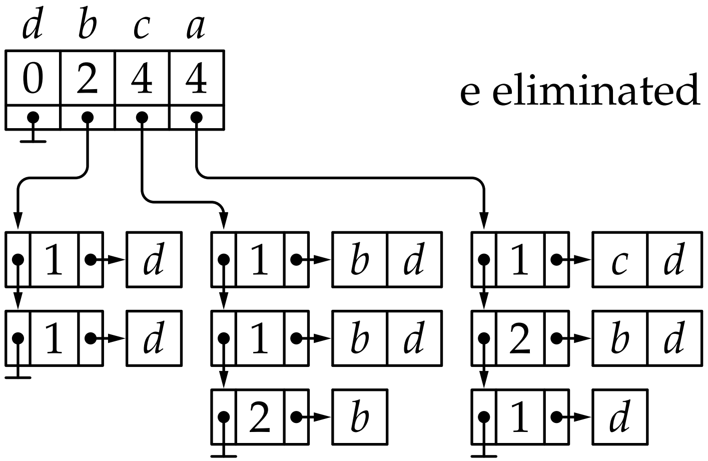


1. Original transaction database.
2. Frequency of individual items.
3. Items in transactions sorted ascendingly w.r.t. their frequency.
4. Transactions sorted lexicographically in descending order (comparison of items inverted w.r.t. preceding step).
5. Data structure used by the algorithm (leading items implicit in list).

RElim: Subproblem Split


initial database


prefix e


e eliminated

The subproblem split of the RElim algorithm. The rightmost list is traversed and reassigned: once to an initially empty list array (conditional database for the prefix *e*, see top right) and once to the original list array (eliminating item *e*, see bottom left). These two databases are then both processed recursively.

- Note that after a simple reassignment there may be duplicate list elements.

RElim: Pseudo-Code

```
function RElim (a: array of transaction lists,   (* cond. database to process *)
                p: set of items,                 (* prefix of the conditional database a *)
                $s_{\min}$: int) : int                (* minimum support of an item set *)
var i, k: item;                                  (* buffer for the current item *)
    s: int;                                      (* support of the current item *)
    n: int;                                      (* number of found frequent item sets *)
    b: array of transaction lists;               (* conditional database for current item *)
    t, u: transaction list element;              (* to traverse the transaction lists *)
begin                                            (* — recursive elimination — *)
  n := 0;                                        (* initialize the number of found item sets *)
  while a is not empty do                        (* while conditional database is not empty *)
    i := last item of a; s := a[i].wgt;          (* get the next item to process *)
    if $s \geq s_{\min}$ then                         (* if the current item is frequent: *)
      $p := p \cup \{i\}$;                            (* extend the prefix item set and *)
      report p with support s;                   (* report the found frequent item set *)
      ...                                        (* create conditional database for i *)
      $p := p - \{i\}$;                               (* and process it recursively, *)
    end;                                         (* then restore the original prefix *)
```

RElim: Pseudo-Code

```
if $s \geq s_{\min}$ then                         (∗ if the current item is frequent: ∗)
    ...                                           (∗ report the found frequent item set ∗)
    $b$ := array of transaction lists;            (∗ create an empty list array ∗)
    $t := a[i]$.head;                             (∗ get the list associated with the item ∗)
    while $t \neq$ nil do                         (∗ while not at the end of the list ∗)
        $u$ := copy of $t$; $t := t$.succ;        (∗ copy the transaction list element, ∗)
        $k := u$.items[0];                        (∗ go to the next list element, and ∗)
        remove $k$ from $u$.items;                (∗ remove the leading item from the copy ∗)
        if $u$.items is not empty                 (∗ add the copy to the conditional database ∗)
        then $u$.succ = $b[k]$.head; $b[k]$.head = $u$; end;
        $b[k]$.wgt := $b[k]$.wgt +$u$.wgt;        (∗ sum the transaction weight ∗)
    end;                                          (∗ in the list weight/transaction counter ∗)
    $n := n + 1 + \text{RElim}(b, p, s_{\min})$;  (∗ process the created database recursively ∗)
    ...                                           (∗ and sum the found frequent item sets, ∗)
end;                                              (∗ then restore the original item set prefix ∗)
...                                               (∗ go on by reassigning ∗)
                                                  (∗ the processed transactions ∗)
```

RElim: Pseudo-Code

```
      ...
      $t$ := $a[i]$.head;                   (∗ get the list associated with the item ∗)
      while $t \neq$ nil do                 (∗ while not at the end of the list ∗)
        $u$ := $t$; $t$ := $t$.succ;        (∗ note the current list element, ∗)
        $k$ := $u$.items[0];                (∗ go to the next list element, and ∗)
        remove $k$ from $u$.items;          (∗ remove the leading item from current ∗)
        if $u$.items is not empty           (∗ reassign the noted list element ∗)
        then $u$.succ = $a[k]$.head; $a[k]$.head = $u$; end;
        $a[k]$.wgt := $a[k]$.wgt +$u$.wgt;  (∗ sum the transaction weight ∗)
      end;                                  (∗ in the list weight/transaction counter ∗)
      remove $a[i]$ from $a$;               (∗ remove the processed list ∗)
    end;
    return $n$;                             (∗ return the number of frequent item sets ∗)
  end;  (∗ function RElim() ∗)
```

- In order to remove duplicate elements, it is usually advisable to sort and compress the next transaction list before it is processed.

The *k*-Items Machine

- Introduced with the LCM algorithm to combine equal transaction suffixes.

- Idea: If the number of items is small, a *bucket/bin sort scheme* can be used to perfectly combine equal transaction suffixes.

- This scheme leads to the ***k*-items machine** (for small *k*). [Uno *et al.* 2004]
  - All possible transaction suffixes are represented as bit patterns; one bucket/bin is created for each possible bit pattern.
  - A RElim-like processing scheme is employed (on a fixed data structure).
  - Leading items are extracted with a table that is indexed with the bit pattern.
  - Items are eliminated with a bit mask.

Table of highest set bits for a 4-items machine (special instructions: `bsr` / `lzcount`):

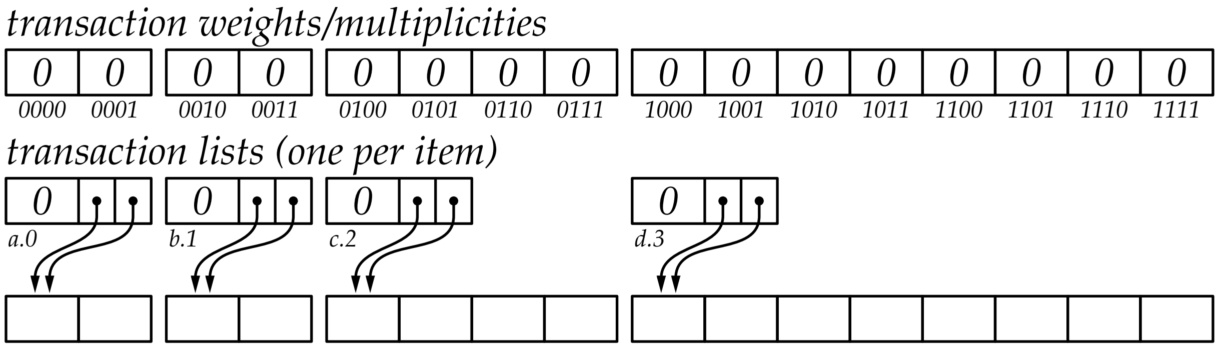
highest items/set bits of transactions (constant)

.	*a.0*	*b.1*	*b.1*	*c.2*	*c.2*	*c.2*	*c.2*	*d.3*	*d.3*	*d.3*	*d.3*	*d.3*	*d.3*	*d.3*	*d.3*
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
____	___a	__b_	__ba	_c__	_c_a	_cb_	_cba	d___	d__a	d_b_	d_ba	dc__	dc_a	dcb_	dcba

The k-items Machine

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

Empty 4-items machine (no transactions)



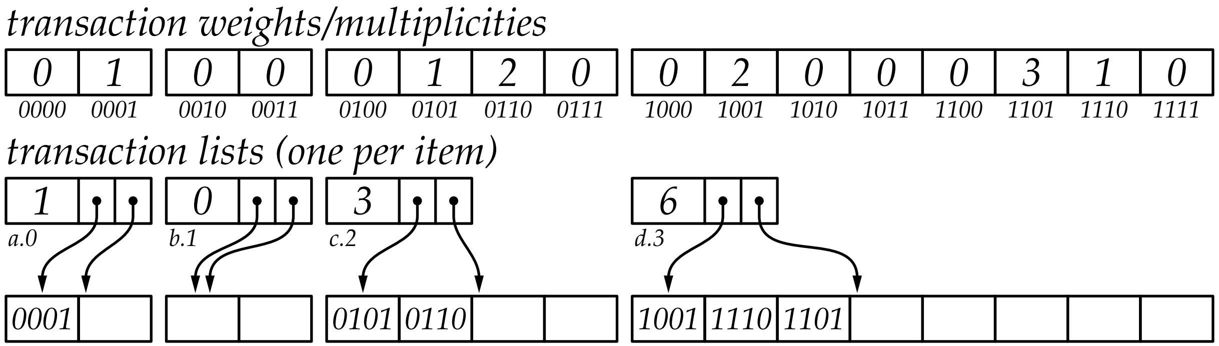
transaction weights/multiplicities

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

transaction lists (one per item)

a.0	b.1	c.2	d.3
0	0	0	0

4-items machine after inserting the transactions



transaction weights/multiplicities

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	0	0	0	1	2	0	0	2	0	0	0	3	1	0

transaction lists (one per item)

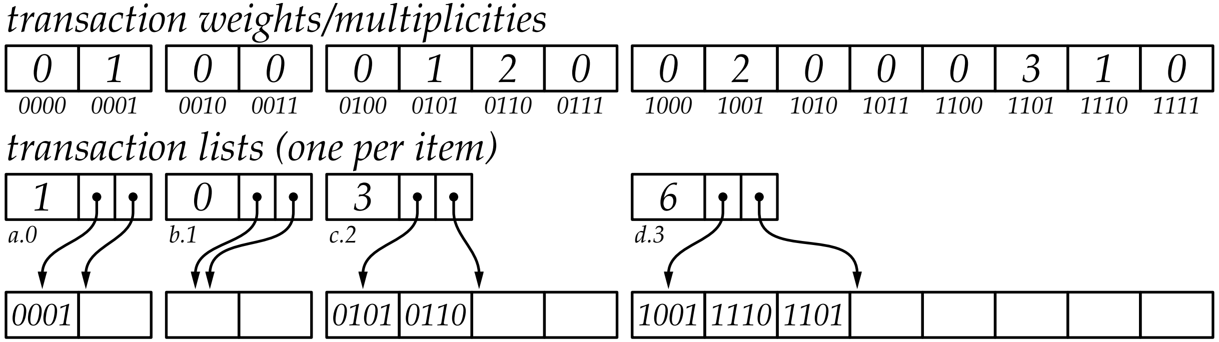
a.0	b.1	c.2	d.3
1	0	3	6
0001		0101 0110	1001 1110 1101

- In this state the 4-items machine represents a special form of the initial transaction database of the RElim algorithm.

The *k*-items Machine

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

4-items machine after inserting the transactions



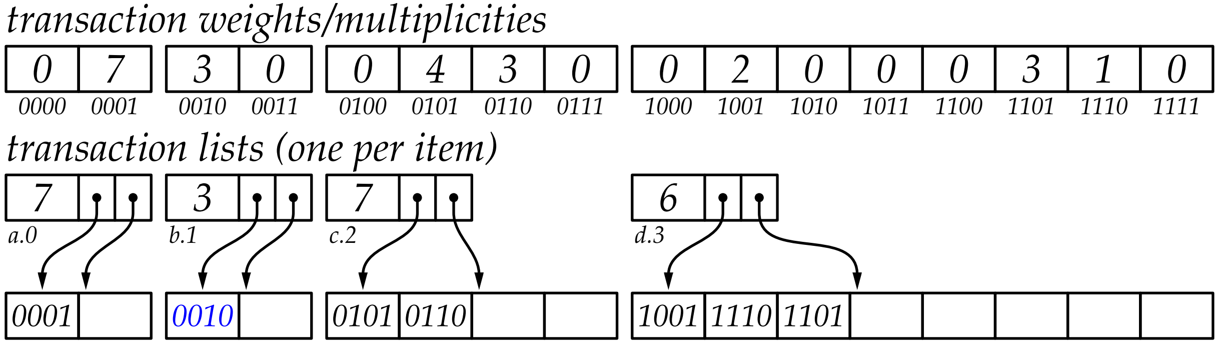
transaction weights/multiplicities

0000	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
0	*1*	*0*	*0*	*0*	*1*	*2*	*0*	*0*	*2*	*0*	*0*	*0*	*3*	*1*	*0*

transaction lists (one per item)

a.0	*b.1*	*c.2*	*d.3*
1	*0*	*3*	*6*
0001		*0101* *0110*	*1001* *1110* *1101*

After propagating the transaction lists



transaction weights/multiplicities

0000	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
0	*7*	*3*	*0*	*0*	*4*	*3*	*0*	*0*	*2*	*0*	*0*	*0*	*3*	*1*	*0*

transaction lists (one per item)

a.0	*b.1*	*c.2*	*d.3*
7	*3*	*7*	*6*
0001	*0010*	*0101* *0110*	*1001* *1110* *1101*

- Propagating the transactions lists is equivalent to occurrence deliver.
- Conditional transaction databases are created as in RElim plus propagation.

Summary RElim

Basic Processing Scheme

- Depth-first traversal of the prefix tree (divide-and-conquer scheme).
- Data is represented as lists of transactions (one per item).
- Support counting is implicit in the (re)assignment step.

Advantages

- Fairly simple data structures and processing scheme.
- Competitive with the fastest algorithms despite this simplicity.

Disadvantages

- RElim is usually outperformed by LCM and FP-growth (discussed later).

Software

- `http://www.borgelt.net/relim.html`

The FP-Growth Algorithm

Frequent Pattern Growth Algorithm [Han, Pei, and Yin 2000]

FP-Growth: Basic Ideas

- FP-Growth means **Frequent Pattern Growth**.
- The item sets are checked in **lexicographic order**
  (**depth-first traversal** of the prefix tree).
- Standard divide-and-conquer scheme (include/exclude items).
- Recursive processing of the conditional transaction databases.
- The transaction database is represented as an **FP-tree**.
  An FP-tree is basically a **prefix tree** with additional structure:
  nodes of this tree that correspond to the same item are linked into lists.

  This **combines a horizontal and a vertical database representation**.
- This data structure is used to compute conditional databases efficiently.

  All transactions containing a given item can easily be found
  by the links between the nodes corresponding to this item.

FP-Growth: Preprocessing the Transaction Database

①	②	③	④	⑤
$a\ d\ f$	d: 8	$d\ a$	$d\ b$	
$a\ c\ d\ e$	b: 7	$d\ c\ a\ e$	$d\ b\ c$	
$b\ d$	c: 5	$d\ b$	$d\ b\ a$	FP-tree
$b\ c\ d$	a: 4	$d\ b\ c$	$d\ b\ a$	(see next slide)
$b\ c$	e: 3	$b\ c$	$d\ b\ e$	
$a\ b\ d$	f: 2	$d\ b\ a$	$d\ c$	
$b\ d\ e$	g: 1	$d\ b\ e$	$d\ c\ a\ e$	
$b\ c\ e\ g$		$b\ c\ e$	$d\ a$	
$c\ d\ f$		$d\ c$	$b\ c$	
$a\ b\ d$	$s_{\min} = 3$	$d\ b\ a$	$b\ c\ e$	

1. Original transaction database.
2. Frequency of individual items.
3. Items in transactions sorted descendingly w.r.t. their frequency and infrequent items removed.
4. Transactions sorted lexicographically in ascending order (comparison of items as in preceding step).
5. Data structure used by the algorithm (details on next slide).

Transaction Representation: FP-Tree

- Build a **frequent pattern tree (FP-tree)** from the transactions (basically a prefix tree with *links between the branches* that link nodes with the same item and a *header table* for the resulting item lists).
- Frequent single item sets can be read directly from the FP-tree.

Simple Example Database

① *a d f*

a c d e

b d

b c d

b c

a b d

b d e

b c e g

c d f

a b d

④ *d b*

d b c

d b a

d b a

d b e

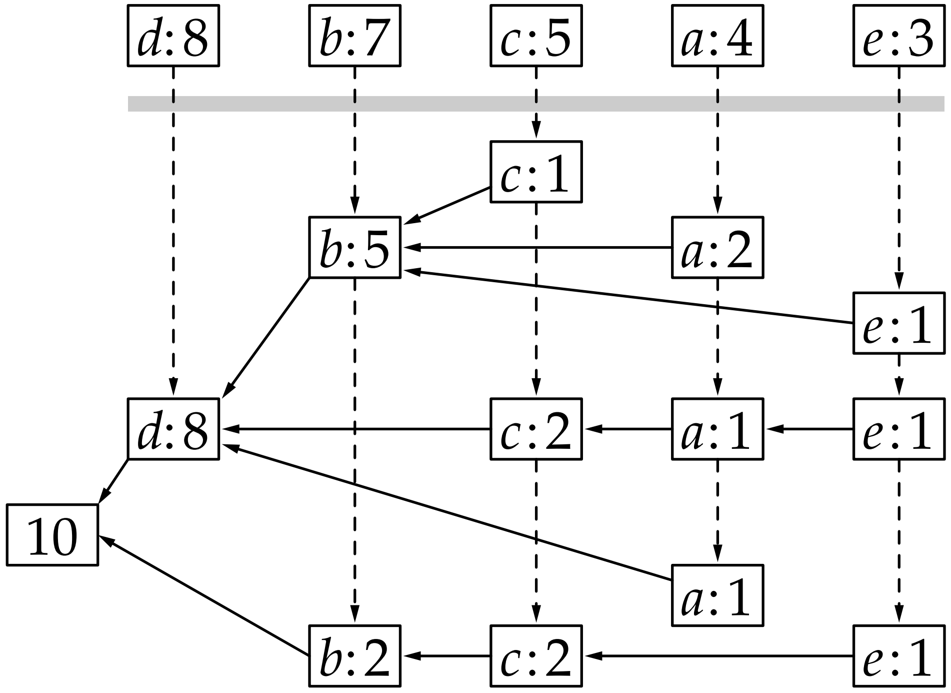
d c

d c a e

d a

b c

b c e



frequent pattern tree

Transaction Representation: FP-Tree

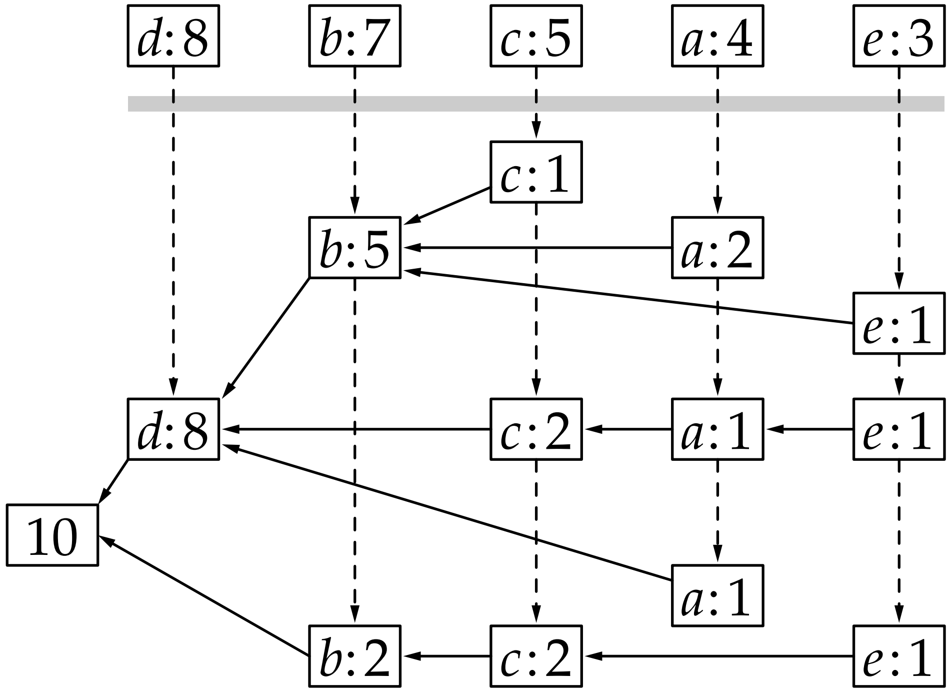
- An FP-tree combines a horizontal and a vertical transaction representation.
- **Horizontal Representation:** prefix tree of transactions

  Vertical Representation: links between the prefix tree branches

Note: the prefix tree is inverted, i.e. there are only parent pointers.

Child pointers are not needed due to the processing scheme (to be discussed).

In principle, all nodes referring to the same item can be stored in an array rather than a list.



frequent pattern tree

Recursive Processing

- The initial FP-tree is **projected** w.r.t. the item corresponding to the rightmost level in the tree (let this item be *i*).

- This yields an FP-tree of the **conditional transaction database** (database of transactions containing the item *i*, but with this item removed — it is implicit in the FP-tree and recorded as a common prefix).

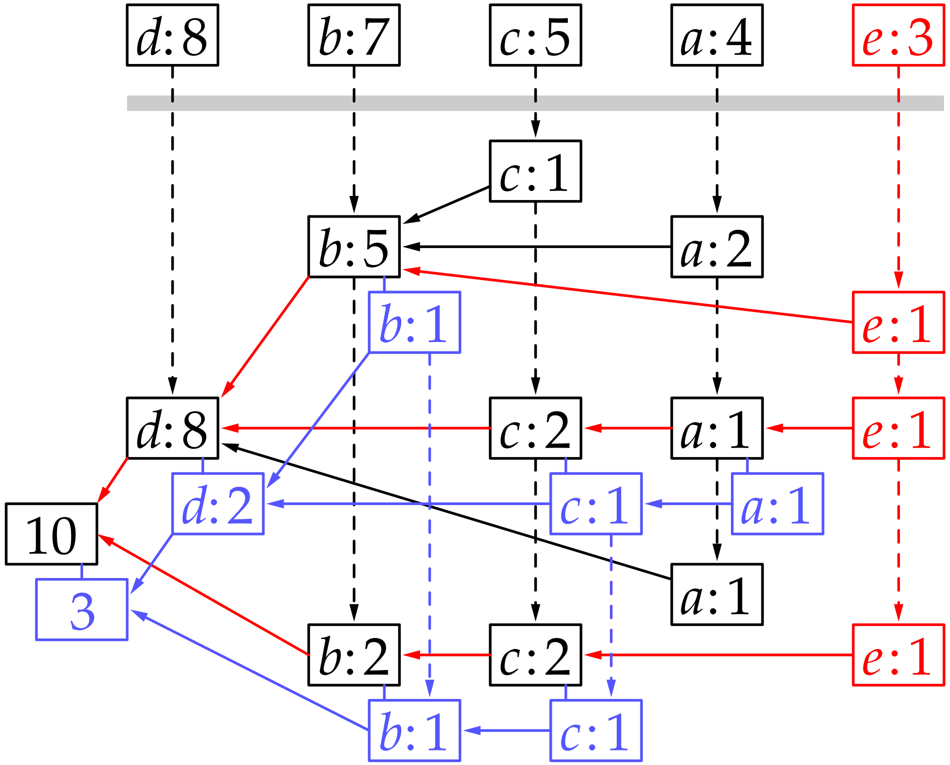
- From the projected FP-tree the frequent item sets containing item *i* can be read directly.

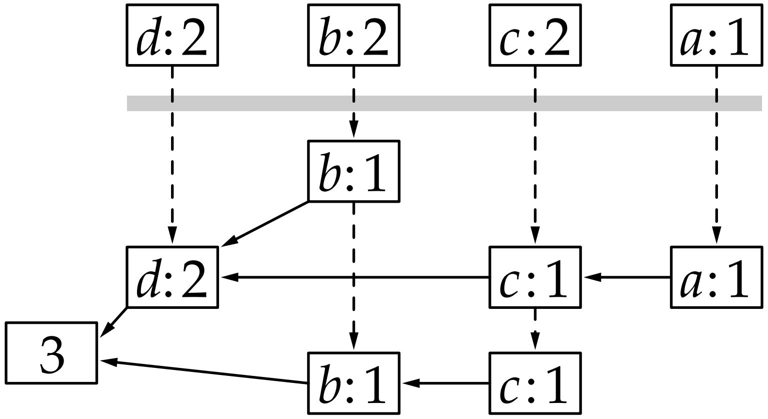
- The **rightmost level** of the original (unprojected) FP-tree is **removed** (the item *i* is removed from the database — exclude split item).

- The projected FP-tree is processed recursively; the item *i* is noted as a prefix that is to be added in deeper levels of the recursion.

- Afterward the reduced original FP-tree is further processed by working on the next level leftward.

Projecting an FP-Tree





↑

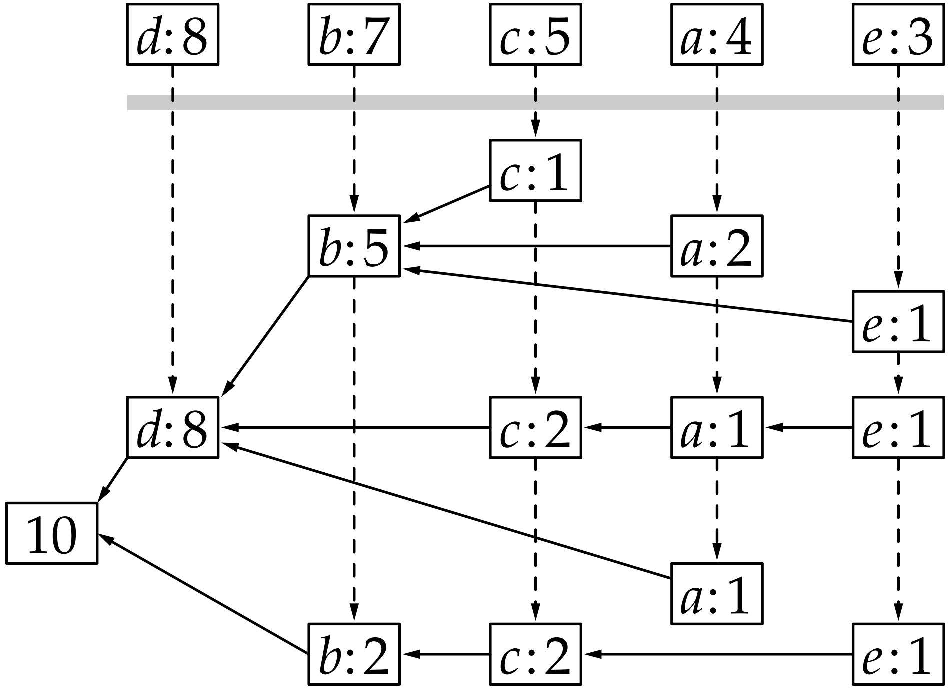
detached projection (prefix e)

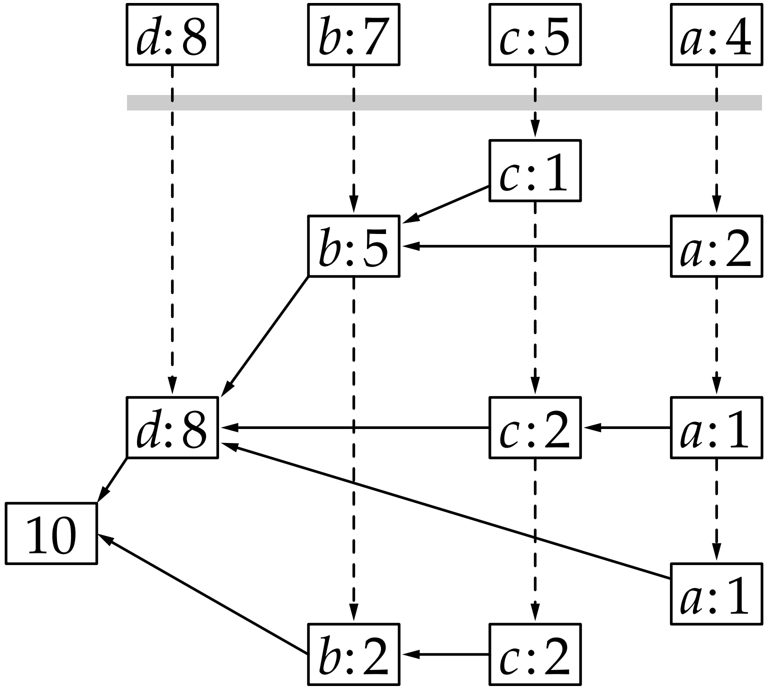
← FP-tree with attached projection

- By traversing the node list for the rightmost item, all transactions containing this item can be found.

- The FP-tree of the conditional database for this item is created by copying the nodes on the paths to the root.

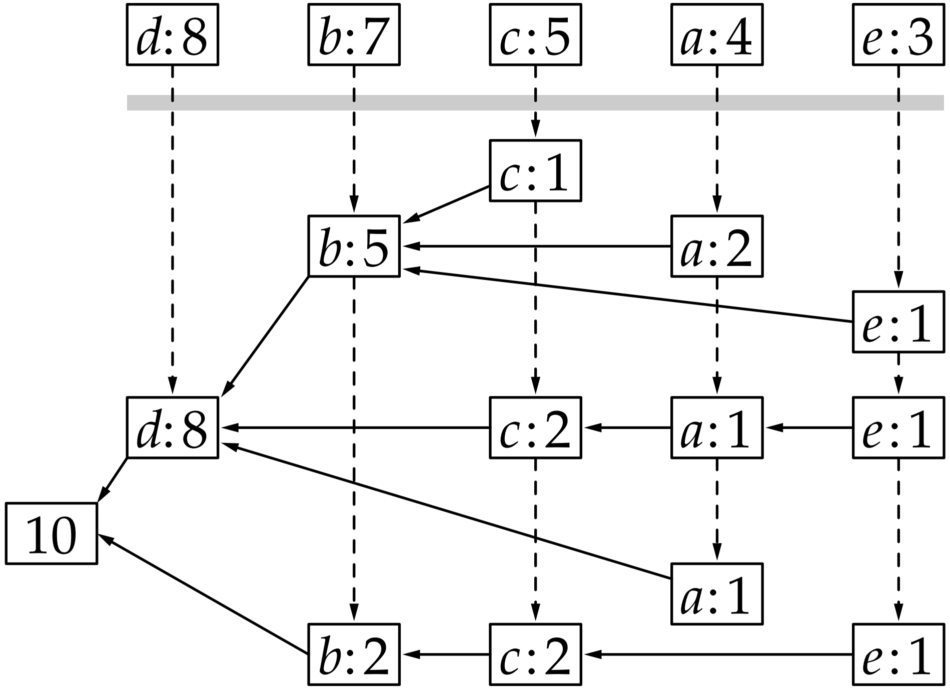
Reducing the Original FP-Tree

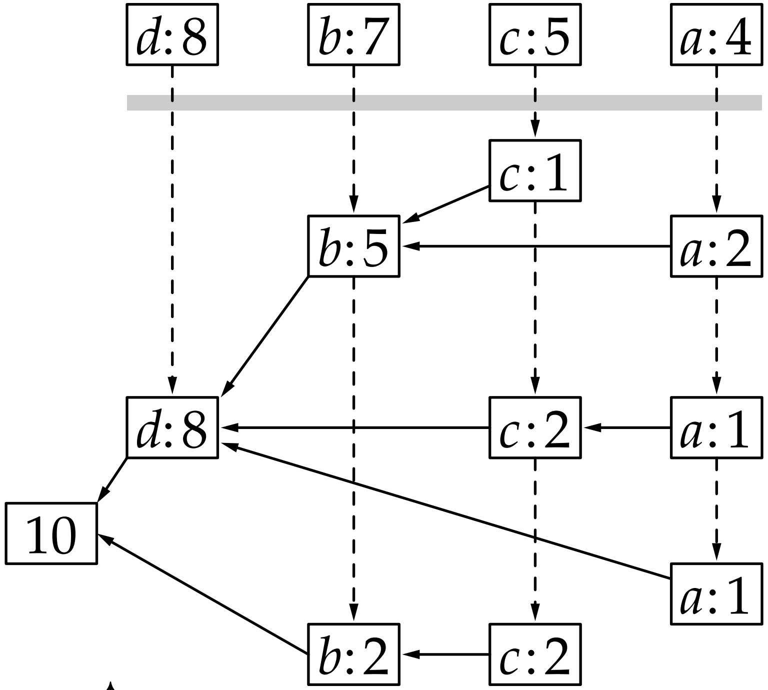




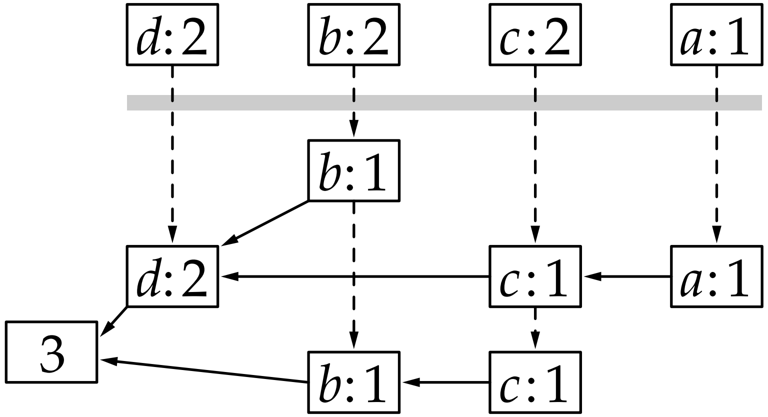
- The original FP-tree is reduced by removing the rightmost level.
- This yields the conditional database for item sets *not* containing the item corresponding to the rightmost level.

FP-growth: Divide-and-Conquer





↑
Conditional database
with item *e* removed
(second subproblem)



← Conditional database for prefix *e*
(first subproblem)

Projecting an FP-Tree

- A simpler, but equally efficient projection scheme (compared to node copying) is to extract a path to the root as a (reduced) transaction (into a global buffer) and to insert this transaction into a new, initially empty FP-tree.

- For the insertion into the new FP-tree, there are two approaches:

  - Apart from a parent pointer (which is needed for the path extraction), each node possesses a pointer to its **first child** and **right sibling**.
    These pointers allow to insert a new transaction top-down.

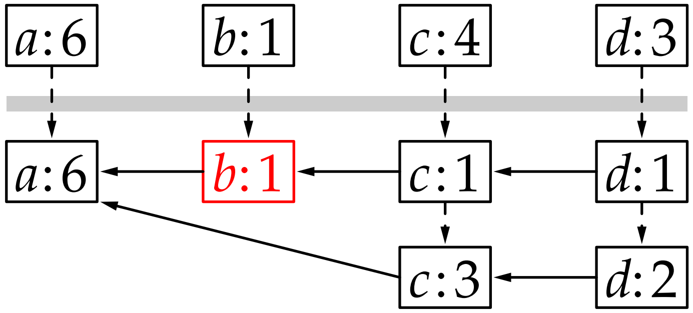
  - If the initial FP-tree has been built from a lexicographically sorted transaction database, the traversal of the item lists yields the (reduced) transactions in lexicographical order.
    This can be exploited to insert a transaction using only the **header table**.

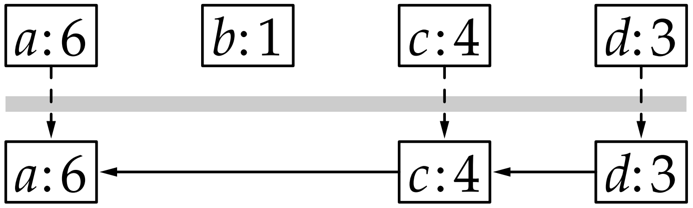
- By processing an FP-tree from **left to right** (or from **top to bottom** w.r.t. the prefix tree), the projection may even reuse the already present nodes and the already processed part of the header table (**top-down FP-growth**).
  In this way the algorithm can be executed on a fixed amount of memory.

Pruning a Projected FP-Tree

- **Trivial case:** If the item corresponding to the rightmost level is infrequent, the item and the FP-tree level are removed without projection.
- **More interesting case:** An item corresponding to a middle level is infrequent, but an item on a level further to the right is frequent.

Example FP-Tree with an infrequent item on a middle level:





- So-called **ff**-pruning or Bonsai pruning of a (projected) FP-tree.
- Implemented by left-to-right levelwise merging of nodes with same parents.
- Not needed if projection works by extraction, support filtering, and insertion.

FP-growth: Implementation Issues

- **Rebuilding the FP-tree:**

  An FP-tree may be projected by extracting the (reduced) transactions described by the paths to the root and inserting them into a new FP-tree.

  The transaction extraction uses a single global buffer of sufficient size.

  This makes it possible to change the item order, with the following **advantages**:

  - No need for α- or Bonsai pruning, since the items can be reordered so that all conditionally frequent items appear on the left.
  - No need for perfect extension pruning, because the perfect extensions can be moved to the left and are processed at the end with chain optimization.

    (Chain optimization is explained on the next slide.)

  However, there are also **disadvantages**:

  - Either the FP-tree has to be traversed twice or pair frequencies have to be determined to reorder the items according to their conditional frequency (for this the resulting item frequencies need to be known.)

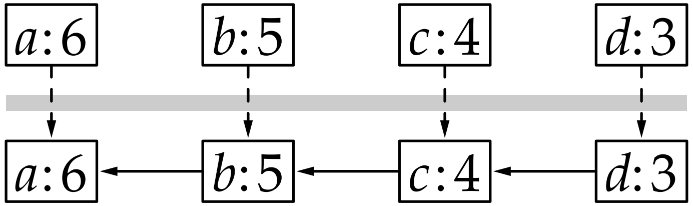
FP-growth: Implementation Issues

- **Chains:**

  If an FP-tree has been reduced to a chain, no projections are computed anymore. Rather all subsets of the set of items in the chain are formed and reported.

- Example of chain processing, exploiting **hypercube decomposition**.

  Suppose we have the following conditional database with prefix P:



  - $P \cup \{d\}$ has support 3 and c, b and d as perfect extensions.
  - $P \cup \{c\}$ has support 4 and b and d as perfect extensions.
  - $P \cup \{b\}$ has support 5 and d as a perfect extension.
  - $P \cup \{a\}$ has support 6.

- Local item order and chain processing implicitly do perfect extension pruning.

FP-growth: Implementation Issues

- The initial FP-tree is built from an array-based main memory representation of the transaction database (eliminates the need for child pointers).

- This has the disadvantage that the memory savings often resulting from an FP-tree representation cannot be fully exploited.

- However, it has the advantage that no child and sibling pointers are needed and the transactions can be inserted in lexicographic order.

- Each FP-tree node has a constant size of 16/24 bytes (2 integers, 2 pointers). Allocating these through the standard memory management is wasteful. (Allocating many small memory objects is highly inefficient.)

- Solution: The nodes are allocated in one large array per FP-tree.

- As a consequence, each FP-tree resides in a single memory block. There is no allocation and deallocation of individual nodes. (This may waste some memory, but is highly efficient.)

FP-growth: Implementation Issues

- An FP-tree can be implemented with only **two integer arrays** [Rasz 2004]:
  - one array contains the transaction counters (support values) and
  - one array contains the parent pointers (as the indices of array elements).

  This reduces the memory requirements to 8 bytes per node.

- Such a memory structure has **advantages**
  due the way in which modern processors access the main memory:

  Linear memory accesses are faster than random accesses.
  - Main memory is organized as a "table" with rows and columns.
  - First the row is addressed and then, after some delay, the column.
  - Accesses to different columns in the same row can skip the row addressing.

- However, there are also **disadvantages**:
  - Programming projection and α- or Bonsai pruning becomes more complex, because less structure is available.
  - Reordering the items is virtually ruled out.

Summary FP-Growth

Basic Processing Scheme

- The transaction database is represented as a frequent pattern tree.
- An FP-tree is projected to obtain a conditional database.
- Recursive processing of the conditional database.

Advantages

- Often the fastest algorithm or among the fastest algorithms.

Disadvantages

- More difficult to implement than other approaches, complex data structure.
- An FP-tree can need more memory than a list or array of transactions.

Software

- `http://www.borgelt.net/fpgrowth.html`

Experimental Comparison

Experiments: Data Sets

- **Chess**
  A data set listing chess end game positions for king vs. king and rook.
  This data set is part of the UCI machine learning repository.

  75 items, 3196 transactions
  (average) transaction size: 37, density: ≈ 0.5

- **Census** (a.k.a. **Adult**)
  A data set derived from an extract of the US census bureau data of 1994, which was preprocessed by discretizing numeric attributes.
  This data set is part of the UCI machine learning repository.

  135 items, 48842 transactions
  (average) transaction size: 14, density: ≈ 0.1

The **density** of a transaction database is the average fraction of all items occurring per transaction: density = average transaction size / number of items.

Experiments: Data Sets

- **T10I4D100K**
  An artificial data set generated with IBM's data generator.
  The name is formed from the parameters given to the generator
  (for example: 100K = 100000 transactions, T10 = 10 items per transaction).

  870 items, 100000 transactions
  average transaction size: ≈ 10.1, density: ≈ 0.012

- **BMS-Webview-1**
  A web click stream from a leg-care company that no longer exists.
  It has been used in the KDD cup 2000 and is a popular benchmark.

  497 items, 59602 transactions
  average transaction size: ≈ 2.5, density: ≈ 0.005

The **density** of a transaction database is the average fraction of all items occurring per transaction: density = average transaction size / number of items

Experiments: Programs and Test System

- All programs are my own implementations.

  All use the same code for reading the transaction database and for writing the found frequent item sets.

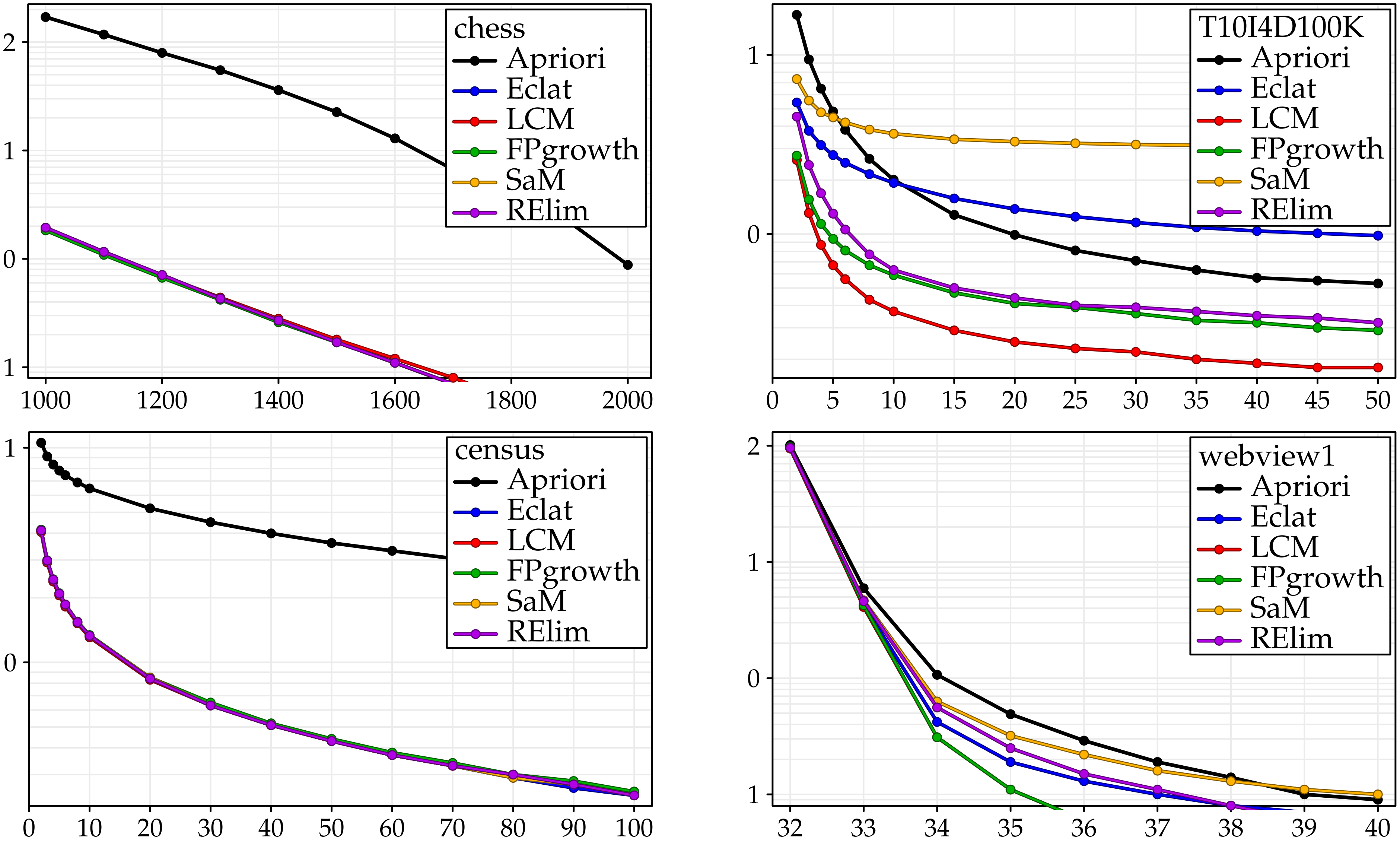
  Therefore differences in speed can only be the effect of the processing schemes.

- These programs and their source code can be found on my web site: `http://www.borgelt.net/fpm.html`

○ Apriori	`http://www.borgelt.net/apriori.html`
○ Eclat & LCM	`http://www.borgelt.net/eclat.html`
○ FP-Growth	`http://www.borgelt.net/fpgrowth.html`
○ RElim	`http://www.borgelt.net/relim.html`
○ SaM	`http://www.borgelt.net/sam.html`

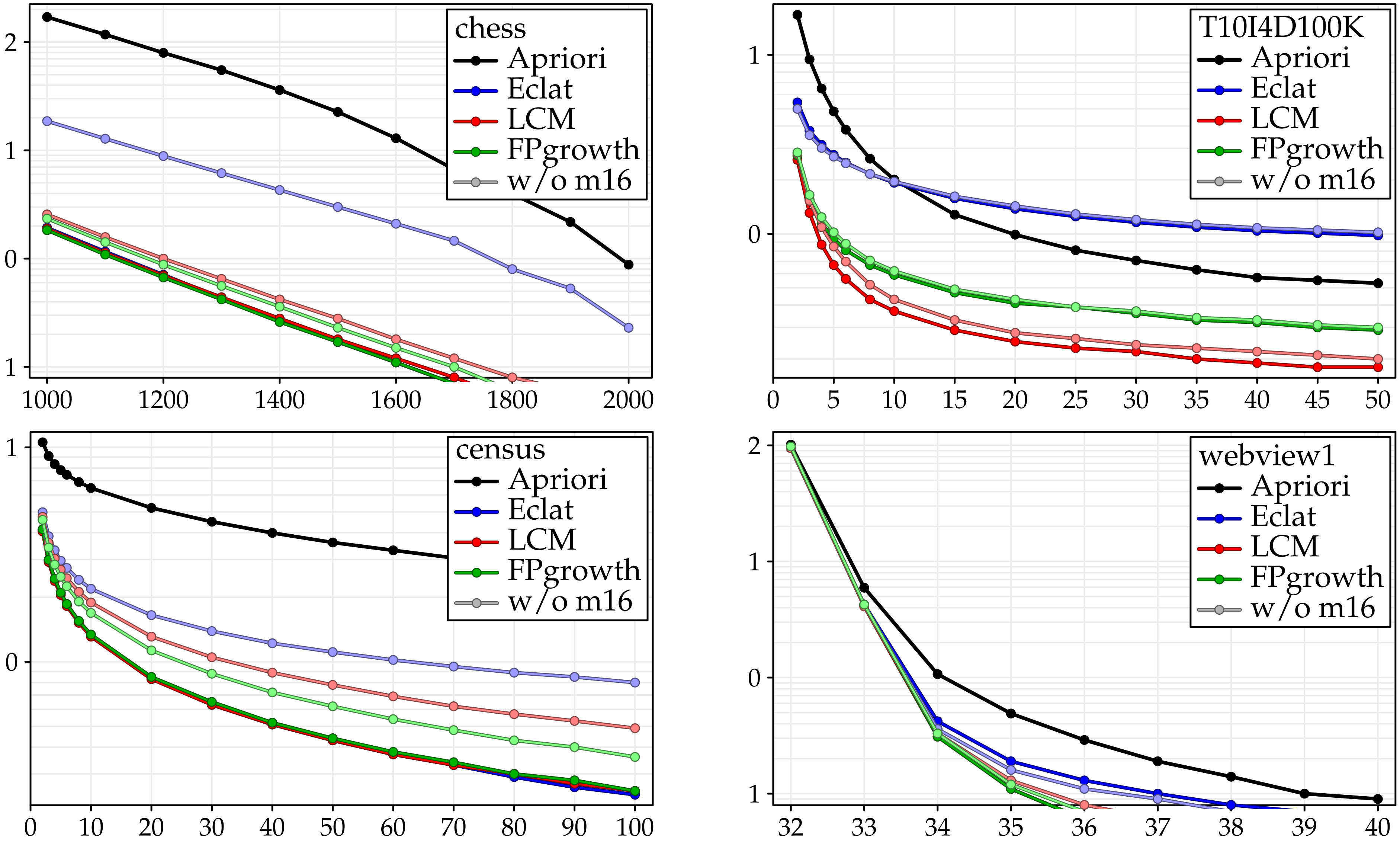
- All tests were run on an Intel Core2 Quad Q9650@3GHz with 8GB memory running Ubuntu Linux 14.04 LTS (64 bit); programs were compiled with GCC 4.8.2.

Experiments: Execution Times



Decimal logarithm of execution time in seconds over absolute minimum support.

Experiments: k-items Machine (here: $k = 16$)

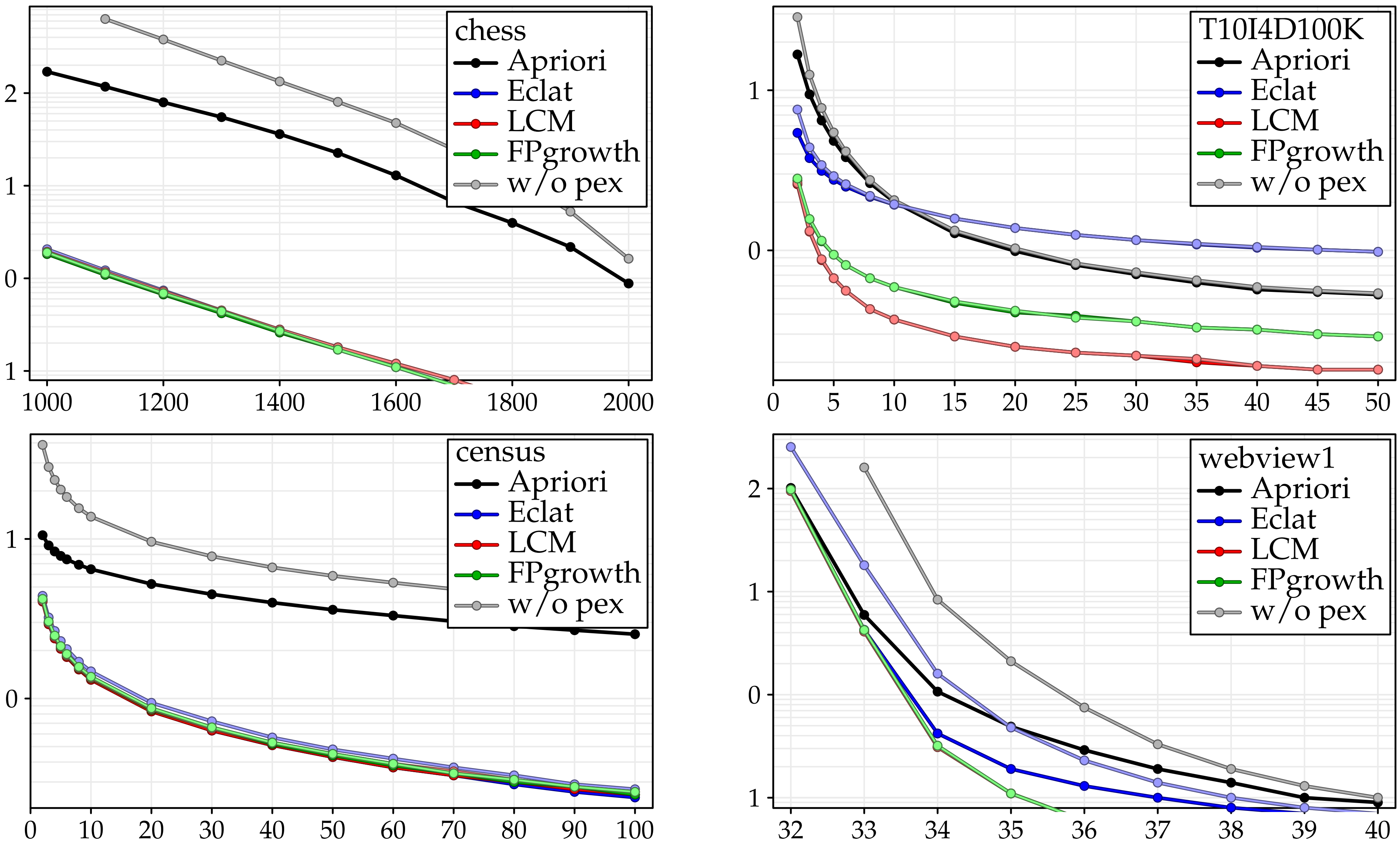


Decimal logarithm of execution time in seconds over absolute minimum support.

Reminder: Perfect Extensions

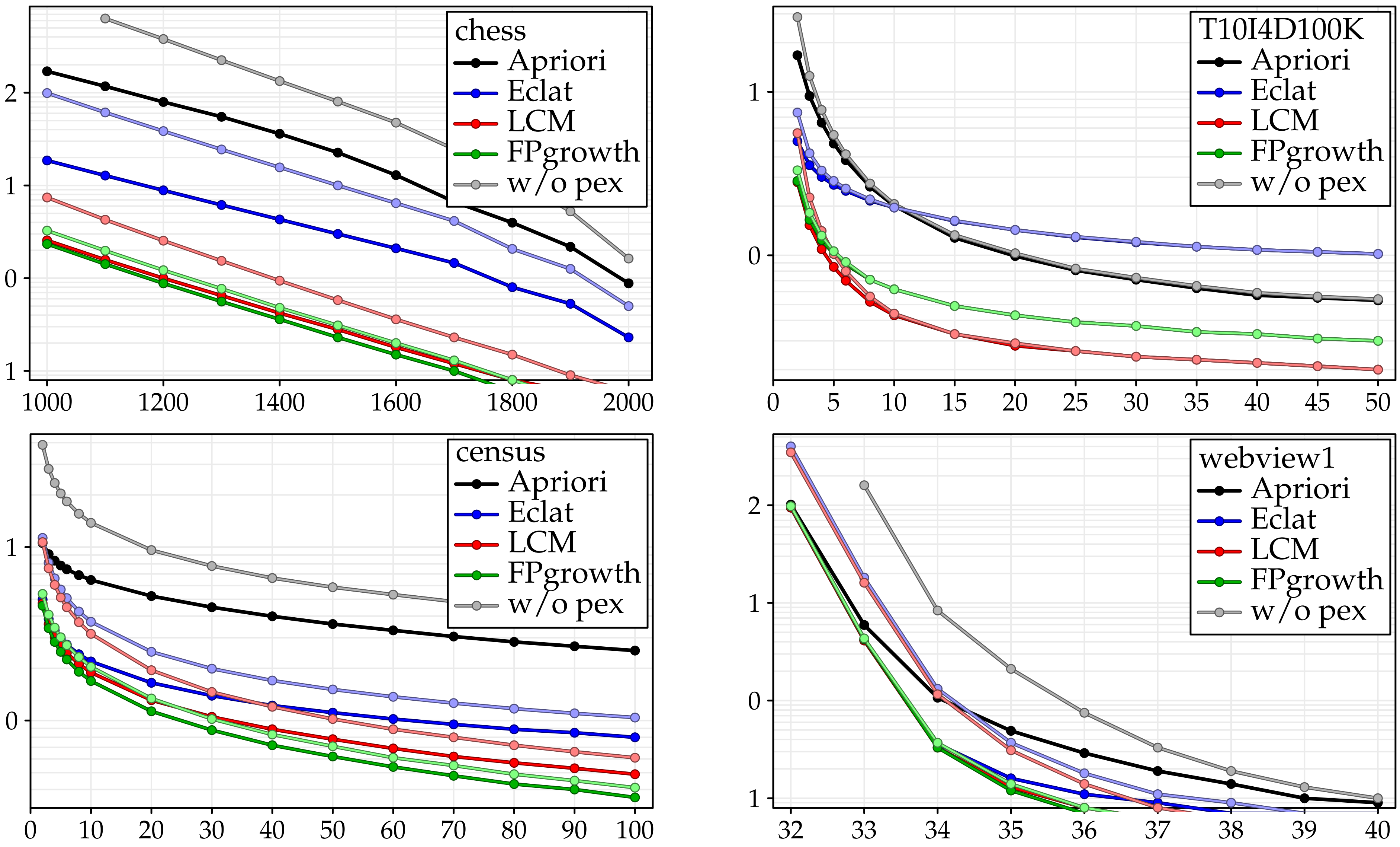
- The search can be improved with so-called **perfect extension pruning**.
- Given an item set I, an item $i \notin I$ is called a **perfect extension** of I, iff I and $I \cup \{i\}$ have the same support (all transactions containing I contain i).
- Perfect extensions have the following properties:
  - If the item i is a perfect extension of an item set I, then i is also a perfect extension of any item set $J \supseteq I$ (as long as $i \notin J$).
  - If I is a frequent item set and X is the set of all perfect extensions of I, then all sets $I \cup J$ with $J \in 2^X$ (where 2^X denotes the power set of X) are also frequent and have the same support as I.
- This can be exploited by collecting perfect extension items in the recursion, in a third element of a subproblem description: $S = (T_*, P, X)$.
- Once identified, perfect extensions are no longer processed in the recursion, but are only used to generate supersets of the prefix having the same support.

Experiments: Perfect Extension Pruning (with m16)



Decimal logarithm of execution time in seconds over absolute minimum support.

Experiments: Perfect Extension Pruning (w/o m16)



Decimal logarithm of execution time in seconds over absolute minimum support.

Reducing the Output:

Closed and Maximal Item Sets

Maximal Item Sets

- Consider the set of **maximal (frequent) item sets**:

$$M_T(s_{\min}) = \{I \subseteq B \mid s_T(I) \geq s_{\min} \wedge \forall J \supset I : s_T(J) < s_{\min}\}.$$

That is: **An item set is maximal if it is frequent,**
but none of its proper supersets is frequent.

- Since with this definition we know that

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad I \in M_T(s_{\min}) \;\vee\; \exists J \supset I : s_T(J) \geq s_{\min}$$

it follows (can easily be proven by successively extending the item set I)

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \exists J \in M_T(s_{\min}) : \quad I \subseteq J.$$

That is: **Every frequent item set has a maximal superset.**

- Therefore: $$\forall s_{\min} : \quad F_T(s_{\min}) = \bigcup_{I \in M_T(s_{\min})} 2^I$$

Mathematical Excursion: Maximal Elements

- Let R be a subset of a partially ordered set $(S, \leq)$.

  An element $x \in R$ is called **maximal** or a **maximal element** of R if

  $$\forall y \in R : \quad y \geq x \quad \Rightarrow \quad y = x.$$

- The notions **minimal** and **minimal element** are defined analogously.

- Maximal elements need not be unique,
  because there may be elements $x, y \in R$ with neither $x \leq y$ nor $y \leq x$.

- Infinite partially ordered sets need not possess a maximal/minimal element.

- Here we consider the set $F_T(s_{\min})$ as a subset of the partially ordered set $(2^B, \subseteq)$:

  The **maximal (frequent) item sets** are the maximal elements of $F_T(s_{\min})$:

  $$M_T(s_{\min}) = \{I \in F_T(s_{\min}) \mid \forall J \in F_T(s_{\min}) : \; J \supseteq I \; \Rightarrow \; J = I\}.$$

  That is, no superset of a maximal (frequent) item set is frequent.

Maximal Item Sets: Example

transaction database

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

- The maximal item sets are:

$$\{b, c\}, \quad \{a, c, d\}, \quad \{a, c, e\}, \quad \{a, d, e\}.$$

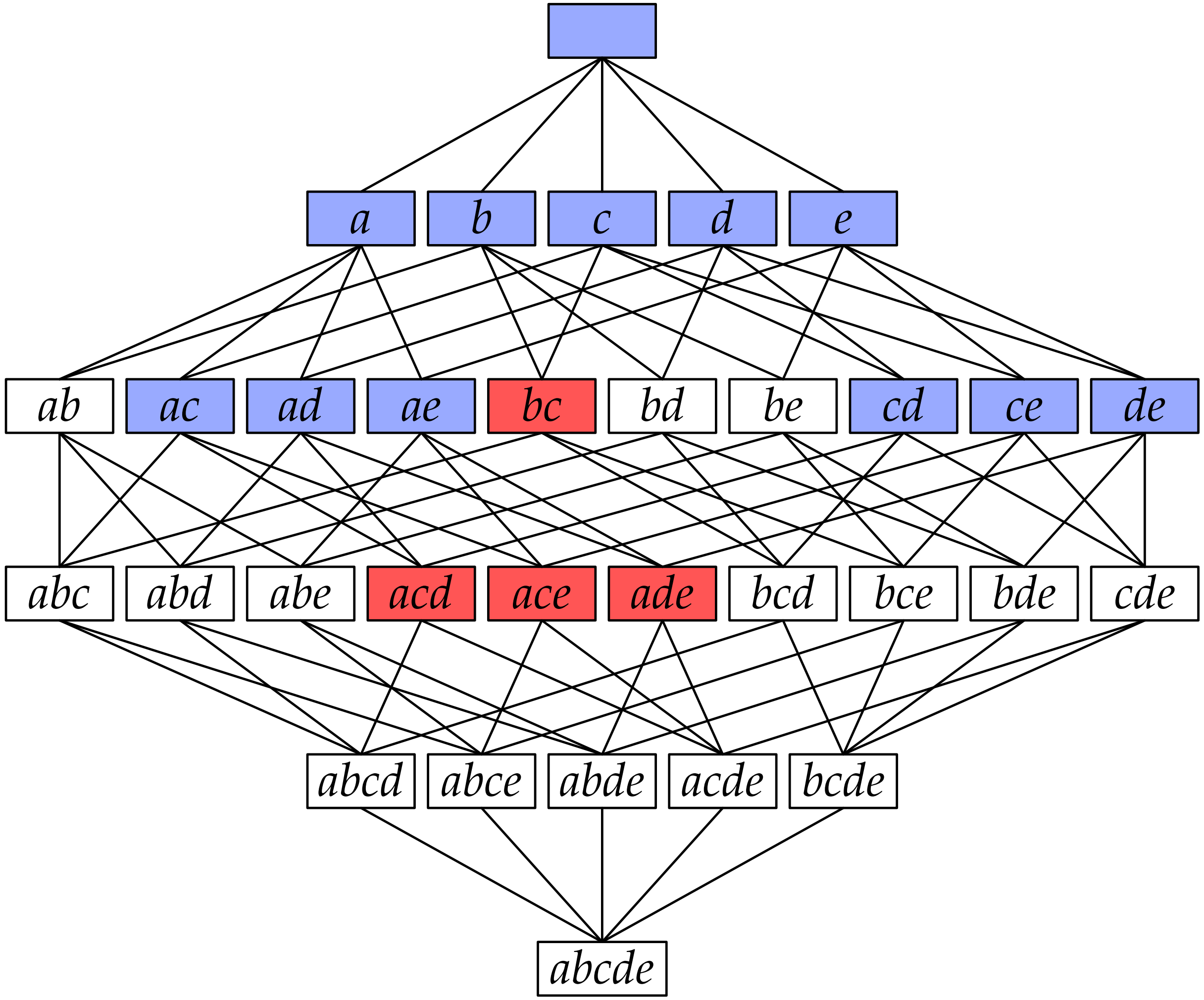
- Every frequent item set is a subset of at least one of these sets.

Hasse Diagram and Maximal Item Sets

transaction database

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

Hasse diagram with maximal item sets ($s_{\min} = 3$):



Red boxes are maximal item sets, white boxes infrequent item sets.

Limits of Maximal Item Sets

- The set of maximal item sets captures the set of all frequent item sets, but then we know at most the support of the maximal item sets exactly.

- About the support of a non-maximal frequent item set we only know:

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) - M_T(s_{\min}) : \quad s_T(I) \geq \max_{J \in M_T(s_{\min}), J \supset I} s_T(J).$$

  This relation follows immediately from $\forall I : \forall J \supseteq I : \; s_T(I) \geq s_T(J)$, that is, an item set cannot have a lower support than any of its supersets.

- Note that we have generally

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad s_T(I) \geq \max_{J \in M_T(s_{\min}), J \supseteq I} s_T(J).$$

- **Question:** Can we find a subset of the set of all frequent item sets, which also preserves knowledge of all support values?

Closed Item Sets

- Consider the set of **closed (frequent) item sets**:

$$C_T(s_{\min}) = \{I \subseteq B \mid s_T(I) \geq s_{\min} \land \forall J \supset I : s_T(J) < s_T(I)\}.$$

That is: **An item set is closed if it is frequent,**
but none of its proper supersets has the same support.

- Since with this definition we know that

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad I \in C_T(s_{\min}) \ \lor \ \exists J \supset I : s_T(J) = s_T(I)$$

it follows (can easily be proven by successively extending the item set I)

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \exists J \in C_T(s_{\min}) : \quad I \subseteq J.$$

That is: **Every frequent item set has a closed superset.**

- Therefore: $$\forall s_{\min} : \quad F_T(s_{\min}) = \bigcup_{I \in C_T(s_{\min})} 2^I$$

Closed Item Sets

- However, not only has every frequent item set a closed superset, but it has a **closed superset with the same support**:

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \exists J \supseteq I : \quad J \in C_T(s_{\min}) \ \wedge \ s_T(J) = s_T(I).$$

  (Proof: see (also) the considerations on the next slide)

- The set of all closed item sets preserves knowledge of all support values:

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad s_T(I) = \max_{J \in C_T(s_{\min}), J \supseteq I} s_T(J).$$

- Note that the weaker statement

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad s_T(I) \geq \max_{J \in C_T(s_{\min}), J \supseteq I} s_T(J)$$

  follows immediately from $\forall I : \forall J \supseteq I : s_T(I) \geq s_T(J)$, that is, an item set cannot have a lower support than any of its supersets.

Closed Item Sets

- **Alternative characterization of closed (frequent) item sets:**

$$I \text{ is closed} \quad \Leftrightarrow \quad s_T(I) \geq s_{\min} \quad \wedge \quad I = \bigcap_{k \in K_T(I)} t_k.$$

Reminder: $K_T(I) = \{k \in \{1, \ldots, n\} \mid I \subseteq t_k\}$ is the *cover* of I w.r.t. T.

- This is derived as follows: since $\forall k \in K_T(I) : I \subseteq t_k$, it is obvious that

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad I \subseteq \bigcap_{k \in K_T(I)} t_k,$$

If $I \subset \bigcap_{k \in K_T(I)} t_k$, it is not closed, since $\bigcap_{k \in K_T(I)} t_k$ has the same support.
On the other hand, no superset of $\bigcap_{k \in K_T(I)} t_k$ has the cover $K_T(I)$.

- Note that the above characterization allows us to construct for any item set the (uniquely determined) closed superset that has the same support.

Closed Item Sets: Example

transaction database

1:	$\{a, d, e\}$
2:	$\{b, c, d\}$
3:	$\{a, c, e\}$
4:	$\{a, c, d, e\}$
5:	$\{a, e\}$
6:	$\{a, c, d\}$
7:	$\{b, c\}$
8:	$\{a, c, d, e\}$
9:	$\{b, c, e\}$
10:	$\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7 $\{b\}$: 3 $\{c\}$: 7 $\{d\}$: 6 $\{e\}$: 7	$\{a, c\}$: 4 $\{a, d\}$: 5 $\{a, e\}$: 6 $\{b, c\}$: 3 $\{c, d\}$: 4 $\{c, e\}$: 4 $\{d, e\}$: 4	$\{a, c, d\}$: 3 $\{a, c, e\}$: 3 $\{a, d, e\}$: 4

- All frequent item sets are closed with the exception of $\{b\}$ and $\{d, e\}$.
- $\{b\}$ is a subset of $\{b, c\}$, both have a support of $3 \mathrel{\hat{=}} 30\%$.

  $\{d, e\}$ is a subset of $\{a, d, e\}$, both have a support of $4 \mathrel{\hat{=}} 40\%$.

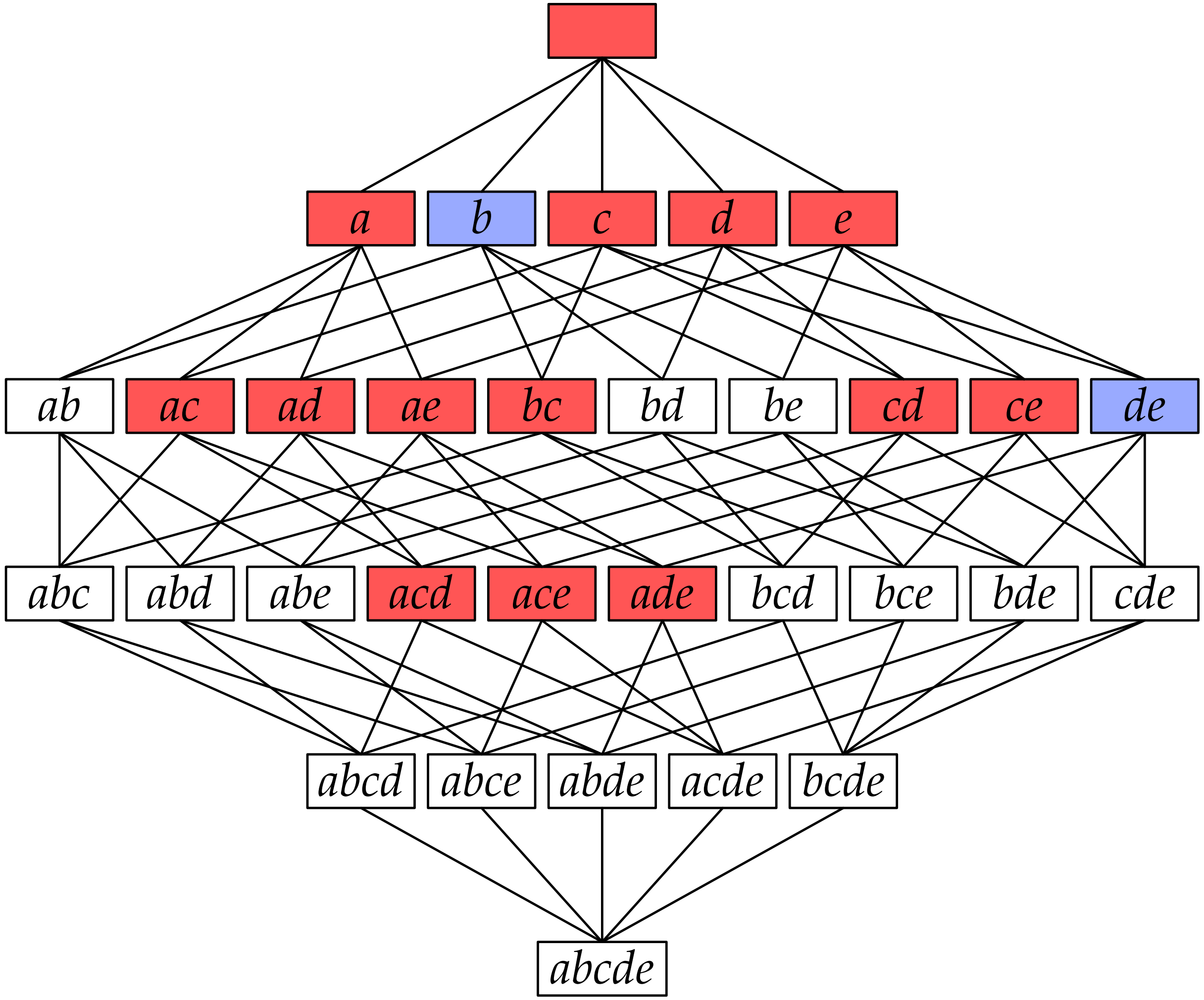
Hasse diagram and Closed Item Sets

transaction database

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

Red boxes are closed item sets, white boxes infrequent item sets.

Hasse diagram with closed item sets ($s_{\min} = 3$):



Reminder: Perfect Extensions

- The search can be improved with so-called **perfect extension pruning**.

- Given an item set I, an item $i \notin I$ is called a **perfect extension** of I, iff I and $I \cup \{i\}$ have the same support (all transactions containing I contain i).

- Perfect extensions have the following properties:

  - If the item i is a perfect extension of an item set I, then i is also a perfect extension of any item set $J \supseteq I$ (as long as $i \notin J$).

  - If I is a frequent item set and X is the set of all perfect extensions of I, then all sets $I \cup J$ with $J \in 2^X$ (where 2^X denotes the power set of X) are also frequent and have the same support as I.

- This can be exploited by collecting perfect extension items in the recursion, in a third element of a subproblem description: $S = (T_*, P, X)$.

- Once identified, perfect extensions are no longer processed in the recursion, but are only used to generate supersets of the prefix having the same support.

Closed Item Sets and Perfect Extensions

transaction database

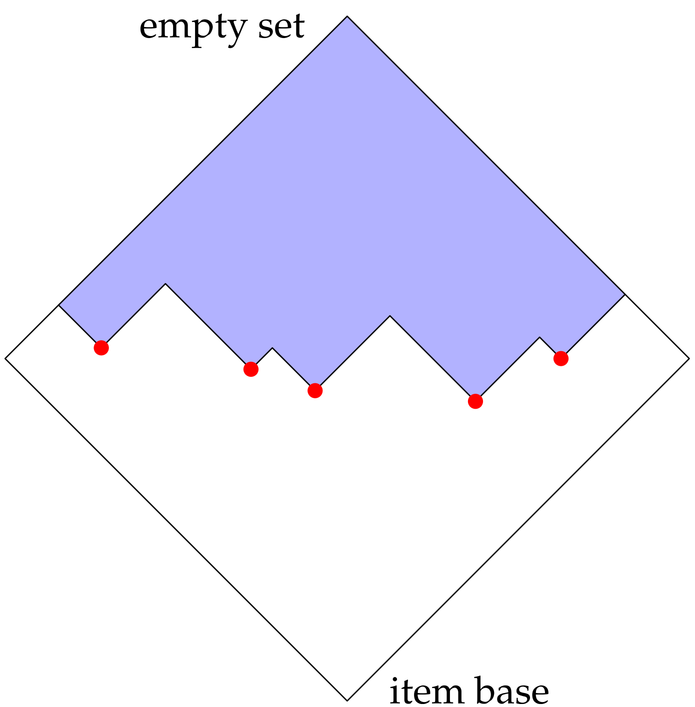
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

frequent item sets

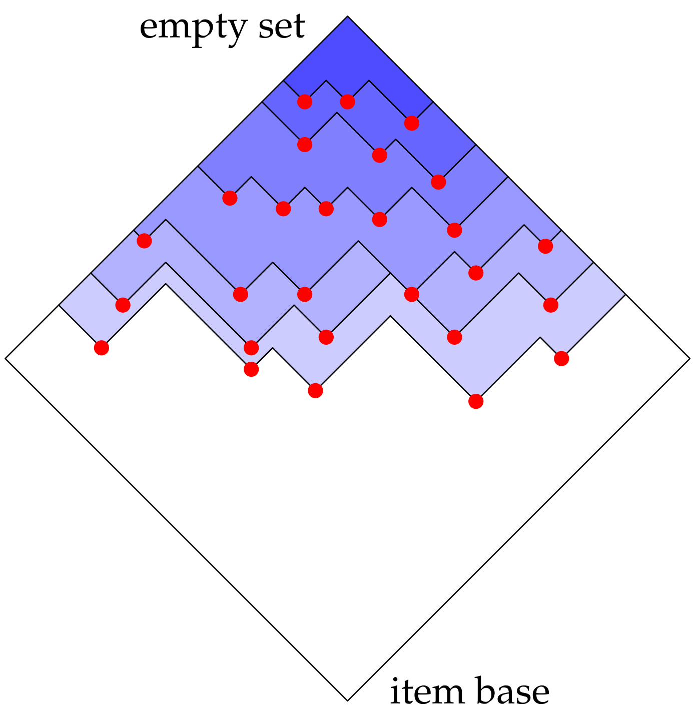
0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

- c is a perfect extension of $\{b\}$ as $\{b\}$ and $\{b, c\}$ both have support 3.
- a is a perfect extension of $\{d, e\}$ as $\{d, e\}$ and $\{a, d, e\}$ both have support 4.
- Non-closed item sets possess at least one perfect extension, closed item sets do not possess any perfect extension.

Relation of Maximal and Closed Item Sets



maximal (frequent) item sets



closed (frequent) item sets

- The set of closed item sets is the union of the sets of maximal item sets for all minimum support values at least as large as $s_{\min}$:

$$C_T(s_{\min}) = \bigcup_{s \in \{s_{\min}, s_{\min}+1, \ldots, n-1, n\}} M_T(s)$$

Mathematical Excursion: Closure Operators

- A **closure operator** on a set S is a function $cl : 2^S \to 2^S$ that satisfies the following conditions $\forall X, Y \subseteq S$:
  - $X \subseteq cl(X)$ (cl is *extensive*)
  - $X \subseteq Y \;\Rightarrow\; cl(X) \subseteq cl(Y)$ (cl is *increasing or monotone*)
  - $cl(cl(X)) = cl(X)$ (cl is *idempotent*)
- A set $R \subseteq S$ is called **closed** if it is equal to its closure:

$$R \text{ is closed} \qquad \Leftrightarrow \qquad R = cl(R).$$

- The **closed (frequent) item sets** are induced by the closure operator

$$cl(I) = \bigcap_{k \in K_T(I)} t_k.$$

restricted to the set of frequent item sets:

$$C_T(s_{\min}) = \{I \in F_T(s_{\min}) \mid I = cl(I)\}$$

Mathematical Excursion: Galois Connections

- Let $(X, \preceq_X)$ and $(Y, \preceq_Y)$ be two partially ordered sets.

- A function pair (f_1, f_2) with $f_1 : X \to Y$ and $f_2 : Y \to X$ is called a **(monotone) Galois connection** iff
  - $\forall A_1, A_2 \in X : \qquad A_1 \preceq_X A_2 \quad \Rightarrow \quad f_1(A_1) \preceq_Y f_1(A_2)$,
  - $\forall B_1, B_2 \in Y : \qquad B_1 \preceq_Y B_2 \quad \Rightarrow \quad f_2(B_1) \preceq_X f_2(B_2)$,
  - $\forall A \in X : \forall B \in Y : \qquad A \preceq_X f_2(B) \quad \Leftrightarrow \quad B \preceq_Y f_1(A)$.

- A function pair (f_1, f_2) with $f_1 : X \to Y$ and $f_2 : Y \to X$ is called an **anti-monotone Galois connection** iff
  - $\forall A_1, A_2 \in X : \qquad A_1 \preceq_X A_2 \quad \Rightarrow \quad f_1(A_1) \succeq_Y f_1(A_2)$,
  - $\forall B_1, B_2 \in Y : \qquad B_1 \preceq_Y B_2 \quad \Rightarrow \quad f_2(B_1) \succeq_X f_2(B_2)$,
  - $\forall A \in X : \forall B \in Y : \qquad A \preceq_X f_2(B) \quad \Leftrightarrow \quad B \preceq_Y f_1(A)$.

- In a monotone Galois connection, both f_1 and f_2 are monotone, in an anti-monotone Galois connection, both f_1 and f_2 are anti-monotone.

Mathematical Excursion: Galois Connections

- Let the two sets X and Y be power sets of some sets U and V, respectively, and let the partial orders be the subset relations on these power sets, that is, let

$$(X, \preceq_X) = (2^U, \subseteq) \qquad \text{and} \qquad (Y, \preceq_Y) = (2^V, \subseteq).$$

- Then the combination $f_1 \circ f_2 : X \to X$ of the functions of a Galois connection is a **closure operator** (as well as the combination $f_2 \circ f_1 : Y \to Y$).

(i) $\forall A \subseteq U: \quad A \subseteq f_2(f_1(A))$ (a closure operator is **extensive**):

- ◦ Since (f_1, f_2) is a Galois connection, we know

$$\forall A \subseteq U : \forall B \subseteq V : \quad A \subseteq f_2(B) \Leftrightarrow B \subseteq f_1(A).$$

- ◦ Choose $B = f_1(A)$:

$$\forall A \subseteq U : \quad A \subseteq f_2(f_1(A)) \Leftrightarrow \underbrace{f_1(A) \subseteq f_1(A)}_{=\text{true}}.$$

- ◦ Choose $A = f_2(B)$:

$$\forall B \subseteq V : \quad \underbrace{f_2(B) \subseteq f_2(B)}_{=\text{true}} \Leftrightarrow B \subseteq f_1(f_2(B)).$$

Mathematical Excursion: Galois Connections

(ii) $\forall A_1, A_2 \subseteq U: \quad A_1 \subseteq A_2 \Rightarrow f_2(f_1(A_1)) \subseteq f_2(f_1(A_2))$

(a closure operator is **increasing** or **monotone**):

- This property follows immediately from the fact that the functions f_1 and f_2 are both (anti-)monotone.

- If f_1 and f_2 are both monotone, we have

$$\begin{aligned} & \forall A_1, A_2 \subseteq U: \quad A_1 \subseteq A_2 \\ \Rightarrow\ & \forall A_1, A_2 \subseteq U: \quad f_1(A_1) \subseteq f_1(A_2) \\ \Rightarrow\ & \forall A_1, A_2 \subseteq U: \quad f_2(f_1(A_1)) \subseteq f_2(f_1(A_2)). \end{aligned}$$

- If f_1 and f_2 are both anti-monotone, we have

$$\begin{aligned} & \forall A_1, A_2 \subseteq U: \quad A_1 \subseteq A_2 \\ \Rightarrow\ & \forall A_1, A_2 \subseteq U: \quad f_1(A_1) \supseteq f_1(A_2) \\ \Rightarrow\ & \forall A_1, A_2 \subseteq U: \quad f_2(f_1(A_1)) \subseteq f_2(f_1(A_2)). \end{aligned}$$

Mathematical Excursion: Galois Connections

(iii) $\forall A \subseteq U\colon\ f_2(f_1(f_2(f_1(A)))) = f_2(f_1(A))$ (a closure operator is **idempotent**):

- Since both $f_1 \circ f_2$ and $f_2 \circ f_1$ are extensive (see above), we know

$$\begin{aligned} \forall A \subseteq V: &\quad A \subseteq f_2(f_1(A)) \subseteq f_2(f_1(f_2(f_1(A)))) \\ \forall B \subseteq V: &\quad B \subseteq f_1(f_2(B)) \subseteq f_1(f_2(f_1(f_2(B)))) \end{aligned}$$

- Choosing $B = f_1(A')$ with $A' \subseteq U$, we obtain

$$\forall A' \subseteq U: \quad f_1(A') \subseteq f_1(f_2(f_1(f_2(f_1(A'))))).$$

- Since (f_1, f_2) is a Galois connection, we know

$$\forall A \subseteq U: \forall B \subseteq V: \quad A \subseteq f_2(B) \Leftrightarrow B \subseteq f_1(A).$$

- Choosing $A = f_2(f_1(f_2(f_1(A'))))$ and $B = f_1(A')$, we obtain

$$\begin{aligned} \forall A' \subseteq U: \quad & f_2(f_1(f_2(f_1(A')))) \subseteq f_2(f_1(A')) \\ & \Leftrightarrow \underbrace{f_1(A') \subseteq f_1(f_2(f_1(f_2(f_1(A')))))}_{=\text{true (see above)}}. \end{aligned}$$

Galois Connections in Frequent Item Set Mining

- Consider the partially ordered sets $(2^B, \subseteq)$ and $(2^{\{1,\ldots,n\}}, \subseteq)$.

  Let $f_1: \quad 2^B \to 2^{\{1,\ldots,n\}}, \quad I \mapsto K_T(I) = \{k \in \{1,\ldots,n\} \mid I \subseteq t_k\}$

  and $f_2: \quad 2^{\{1,\ldots,n\}} \to 2^B, \quad J \mapsto \bigcap_{j \in J} t_j = \{i \in B \mid \forall j \in J : i \in t_j\}$.

- The function pair (f_1, f_2) is an **anti-monotone Galois connection**:
  - $\forall I_1, I_2 \in 2^B$:

    $I_1 \subseteq I_2 \quad \Rightarrow \quad f_1(I_1) = K_T(I_1) \supseteq K_T(I_2) = f_1(I_2)$,
  - $\forall J_1, J_2 \in 2^{\{1,\ldots,n\}}$:

    $J_1 \subseteq J_2 \quad \Rightarrow \quad f_2(J_1) = \bigcap_{k \in J_1} t_k \supseteq \bigcap_{k \in J_2} t_k = f_2(J_2)$,
  - $\forall I \in 2^B : \forall J \in 2^{\{1,\ldots,n\}}$:

    $I \subseteq f_2(J) = \bigcap_{j \in J} t_j \quad \Leftrightarrow \quad J \subseteq f_1(I) = K_T(I)$.

- As a consequence $f_1 \circ f_2 : 2^B \to 2^B, I \mapsto \bigcap_{k \in K_T(I)} t_k$ is a **closure operator**.

Galois Connections in Frequent Item Set Mining

- Likewise $f_2 \circ f_1 : 2^{\{1,\ldots,n\}} \to 2^{\{1,\ldots,n\}}$, $J \mapsto K_T(\bigcap_{j \in J} t_j)$ is also a **closure operator**.

- Furthermore, if we restrict our considerations to the respective sets of closed sets in both domains, that is, to the sets

  $\mathcal{C}_B = \{I \subseteq B \mid I = f_2(f_1(I)) = \bigcap_{k \in K_T(I)} t_k\}$ and

  $\mathcal{C}_T = \{J \subseteq \{1, \ldots, n\} \mid J = f_1(f_2(J)) = K_T(\bigcap_{j \in J} t_j)\}$,

  there exists a **1-to-1 relationship** between these two sets, which is described by the Galois connection:

  $f'_1 = f_1|_{\mathcal{C}_B}$ is a **bijection** with $f'^{-1}_1 = f'_2 = f_2|_{\mathcal{C}_T}$.

  (This follows immediately from the facts that the Galois connection describes closure operators and that a closure operator is idempotent.)

- Therefore finding closed item sets with a given **minimum support** is equivalent to finding closed sets of transaction indices of a given **minimum size**.

Closed Item Sets / Transaction Index Sets

- Finding closed item sets with a given **minimum support** is equivalent to finding closed sets of transaction indices of a given **minimum size**.

Closed in the item set domain 2^B: an item set I is closed if

- adding an item to I reduces the support compared to I;
- adding an item to I loses at least one trans. in $K_T(I) = \{k \in \{1, \ldots, n\} | I \subseteq t_k\}$;
- there is no perfect extension, that is, no (other) item that is contained in all transactions t_k, $k \in K_T(I)$.

Closed in the transaction index set domain $2^{\{1,\ldots,n\}}$:
a transaction index set K is closed if

- adding a transaction index to K reduces the size of the transaction intersection $I_K = \bigcap_{k \in K} t_k$ compared to K;
- adding a transaction index to K loses at least one item in $I_K = \bigcap_{k \in K} t_k$;
- there is no perfect extension, that is, no (other) transaction that contains all items in $I_K = \bigcap_{k \in K} t_k$.

Types of Frequent Item Sets: Summary

- **Frequent Item Set**

  Any frequent item set (support is higher than the minimal support):

  I frequent $\quad\Leftrightarrow\quad s_T(I) \geq s_{\min}$

- **Closed (Frequent) Item Set**

  A frequent item set is called *closed* if no superset has the same support:

  I closed $\quad\Leftrightarrow\quad s_T(I) \geq s_{\min} \quad\wedge\quad \forall J \supset I : s_T(J) < s_T(I)$

- **Maximal (Frequent) Item Set**

  A frequent item set is called *maximal* if no superset is frequent:

  I maximal $\quad\Leftrightarrow\quad s_T(I) \geq s_{\min} \quad\wedge\quad \forall J \supset I : s_T(J) < s_{\min}$

- Obvious relations between these types of item sets:
  - All maximal item sets and all closed item sets are frequent.
  - All maximal item sets are closed.

Types of Frequent Item Sets: Summary

0 items	1 item	2 items	3 items
$\emptyset^{+}$: 10	$\{a\}^{+}$: 7	$\{a,c\}^{+}$: 4	$\{a,c,d\}^{+*}$: 3
	$\{b\}$: 3	$\{a,d\}^{+}$: 5	$\{a,c,e\}^{+*}$: 3
	$\{c\}^{+}$: 7	$\{a,e\}^{+}$: 6	$\{a,d,e\}^{+*}$: 4
	$\{d\}^{+}$: 6	$\{b,c\}^{+*}$: 3	
	$\{e\}^{+}$: 7	$\{c,d\}^{+}$: 4	
		$\{c,e\}^{+}$: 4	
		$\{d,e\}$: 4	

- **Frequent Item Set**
  Any frequent item set (support is higher than the minimal support).

- **Closed (Frequent) Item Set** (marked with $^{+}$)
  A frequent item set is called *closed* if no superset has the same support.

- **Maximal (Frequent) Item Set** (marked with *)
  A frequent item set is called *maximal* if no superset is frequent.

Searching for Closed and Maximal Item Sets

Searching for Closed Frequent Item Sets

- We know that it suffices to find the closed item sets together with their support: from them all frequent item sets and their support can be retrieved.

- The characterization of closed item sets by

$$I \text{ closed} \quad \Leftrightarrow \quad s_T(I) \geq s_{\min} \quad \wedge \quad I = \bigcap_{k \in K_T(I)} t_k$$

  suggests to find them by **forming all possible intersections** of the transactions (of at least $s_{\min}$ transactions).

- However, on standard data sets, approaches using this idea are rarely competitive with other methods.

- Special cases in which they are competitive are domains with few transactions and very many items.
  Examples of such a domains are **gene expression analysis** and the **analysis of document collections**.

Carpenter

[Pan, Cong, Tung, Yang, and Zaki 2003]

Carpenter: Enumerating Transaction Sets

- The **Carpenter** algorithm implements the intersection method by enumerating sets of transactions (or, equivalently, sets of trans. indices), intersecting them, and removing possible duplicates (ensuring closed transaction index sets).

- This is done with basically the same **divide-and-conquer scheme** as for the item set enumeration approaches, only that it is applied to transactions (that is, items and transactions exchange their meaning [Rioult *et al.* 2003]).

- The task to enumerate all transaction index sets is split into two sub-tasks:
  - enumerate all transaction index sets that contain the index 1
  - enumerate all transaction index sets that do *not* contain the index 1.

- These sub-tasks are then further divided w.r.t. the transaction index 2: enumerate all transaction index sets containing
  - both indices 1 and 2,
  - index 1, but not index 2,
  - index 2, but not index 1,
  - neither index 1 nor index 2,

  and so on recursively.

Carpenter: Enumerating Transaction Sets

- All subproblems in the recursion can be described by triplets $S = (I, K, k)$.
  - ○ $K \subseteq \{1, \ldots, n\}$ is a set of transaction indices,
  - ○ $I = \bigcap_{k \in K} t_k$ is their intersection, and
  - ○ k is a transaction index, namely the index of the next transaction to consider.
- The initial problem, with which the recursion is started, is $S = (B, \emptyset, 1)$, where B is the item base and no transactions have been intersected yet.
- A subproblem $S_0 = (I_0, K_0, k_0)$ is processed as follows:
  - ○ Let $K_1 = K_0 \cup \{k_0\}$ and form the intersection $I_1 = I_0 \cap t_{k_0}$.
  - ○ If $I_1 = \emptyset$, do nothing (return from recursion).
  - ○ If $|K_1| \geq s_{\min}$, and there is no transaction t_j with $j \in \{1, \ldots, n\} - K_1$ such that $I_1 \subseteq t_j$ (that is, K_1 is closed), report I_1 with support $s_T(I_1) = |K_1|$.
  - ○ Let $k_1 = k_0 + 1$. If $k_1 \leq n$, then form the subproblems $S_1 = (I_1, K_1, k_1)$ and $S_2 = (I_0, K_0, k_1)$ and process them recursively.

Carpenter: List-based Implementation

- **Transaction identifier lists** are used to represent the current item set I (vertical transaction representation, as in the Eclat algorithm).
- The intersection consists in collecting all lists with the next transaction index k.
- Example:

transaction database

t_1	a	b	c	
t_2	a	d	e	
t_3	b	c	d	
t_4	a	b	c	d
t_5	b	c		
t_6	a	b	d	
t_7	d	e		
t_8	c	d	e	

transaction identifier lists

a	b	c	d	e
1	1	1	2	2
2	3	3	3	7
4	4	4	4	8
6	5	5	6	
	6	8	7	
			8	

collection for $K = \{1\}$

a	b	c
2	3	3
4	4	4
6	5	5
	6	8

for $K = \{1,2\}, \{1,3\}$

a
4
6

b	c
4	4
5	5
6	8

Carpenter: Table-/Matrix-based Implementation

- Represent the data set by a $n \times |B|$ matrix M as follows [Borgelt *et al.* 2011]

$$m_{ki} = \begin{cases} 0, & \text{if item } i \notin t_k, \\ |\{j \in \{k, \ldots, n\} \mid i \in t_j\}|, & \text{otherwise.} \end{cases}$$

- Example: transaction database / matrix representation

transaction database

t_1	a	b	c	
t_2	a	d	e	
t_3	b	c	d	
t_4	a	b	c	d
t_5	b	c		
t_6	a	b	d	
t_7	d	e		
t_8	c	d	e	

matrix representation

	a	b	c	d	e
t_1	4	5	5	0	0
t_2	3	0	0	6	3
t_3	0	4	4	5	0
t_4	2	3	3	4	0
t_5	0	2	2	0	0
t_6	1	1	0	3	0
t_7	0	0	0	2	2
t_8	0	0	1	1	1

- The current item set I is simply represented by the contained items.
  An intersection collects all items $i \in I$ with $m_{ki} > \max\{0, s_{\min} - |K| - 1\}$.

Carpenter: Duplicate Removal/Closedness Check

- The intersection of several transaction index sets can yield the same item set.

- The **support** of the item set is the size of the **largest transaction index set** that yields the item set; smaller transaction index sets can be skipped/ignored.

  This is the reason for the check whether there exists a transaction t_j with $j \in \{1, \ldots, n\} - K_1$ such that $I_1 \subseteq t_j$.

- This check is split into the two checks whether there exists such a transaction t_j
  - with $j > k_0$ and
  - with $j \in \{1, \ldots, k_0 - 1\} - K_0$.

- The **first check** is easy, because such transactions are considered in the **recursive processing** which can return whether one exists.

- The problematic **second check** is solved by maintaining a **repository of already found closed frequent item sets**.

- In order to make the look-up in the repository efficient, it is laid out as a **prefix tree** with a flat array top level.

Summary Carpenter

Basic Processing Scheme

- Enumeration of transactions sets (transaction identifier sets).
- Intersection of the transactions in any set yields a closed item set.
- Duplicate removal/closedness check is done with a repository (prefix tree).

Advantages

- Effectively linear in the number of items.
- Very fast for transaction databases with many more items than transactions.

Disadvantages

- Exponential in the number of transactions.
- Very slow for transaction databases with many more transactions than items.

Software

- `http://www.borgelt.net/carpenter.html`

IsTa

Intersecting Transactions
[Mielikäinen 2003] (simple repository, no prefix tree)
[Borgelt, Yang, Nogales-Cadenas, Carmona-Saez, and Pascual-Montano 2011]

Ista: Cumulative Transaction Intersections

- Alternative approach: maintain a repository of all closed item sets, which is updated by intersecting it with the next transaction [Mielikainen 2003].

- To justify this approach formally, we consider the set of all closed frequent item sets for $s_{\min} = 1$, that is, the set

$$\mathcal{C}_T(1) = \left\{ I \subseteq B \;\middle|\; \exists S \subseteq T : S \neq \emptyset \land I = \textstyle\bigcap_{t \in S} t \right\}.$$

- The set $\mathcal{C}_T(1)$ satisfies the following simple recursive relation:

$$\begin{aligned} \mathcal{C}_\emptyset(1) &= \emptyset, \\ \mathcal{C}_{T \cup \{t\}}(1) &= \mathcal{C}_T(1) \cup \{t\} \cup \{I \mid \exists s \in \mathcal{C}_T(1) : I = s \cap t\}. \end{aligned}$$

- Therefore we can start the procedure with an empty set of closed item sets and then process the transactions one by one.

- In each step update the set of closed item sets by adding the new transaction t and the additional closed item sets that result from intersecting it with $\mathcal{C}_T(1)$.

- Also, the support of already known closed item sets may have to be updated.

Ista: Cumulative Transaction Intersections

- The core implementation problem is to find a **data structure** for storing the closed item sets that allows to quickly compute the intersections with a new transaction and to merge the result with the already stored closed item sets.

- For this we rely on a **prefix tree**, each node of which represents an item set.

- The algorithm works on the prefix tree as follows:

  - At the beginning an empty tree is created (dummy root node); then the transactions are processed one by one.

  - Each new transaction is first simply added to the prefix tree. Any new nodes created in this step are initialized with a support of zero.

  - In the next step we compute the intersections of the new transaction with all item sets represented by the current prefix tree.

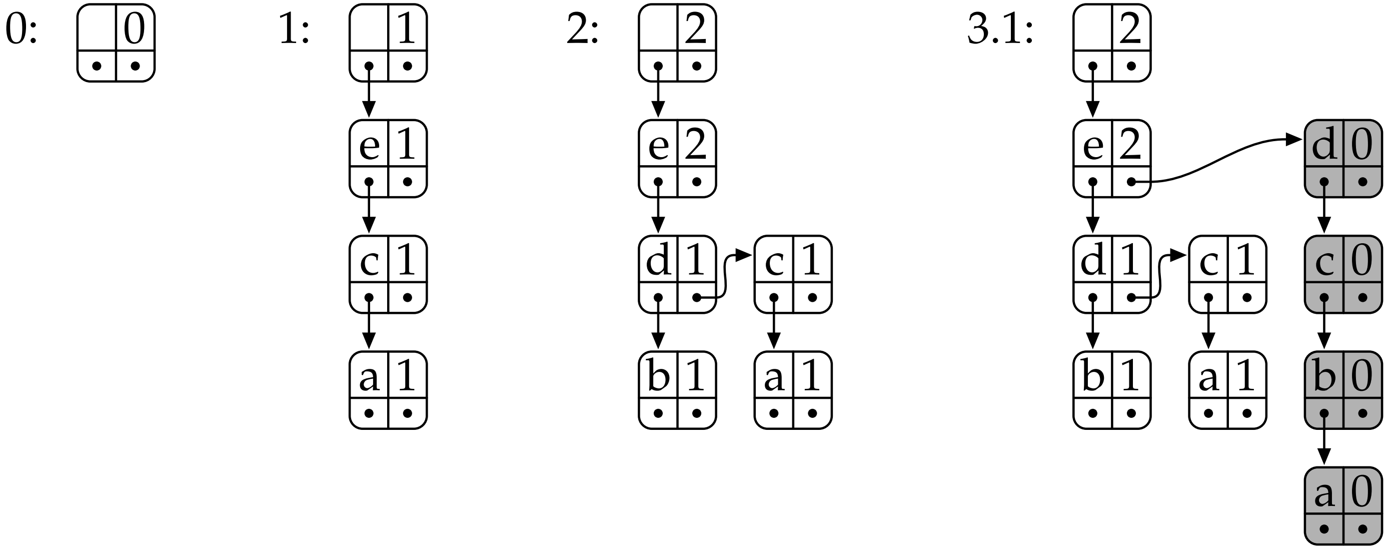
  - A recursive procedure traverses the prefix tree selectively (depth-first) and matches the items in the tree nodes with the items of the transaction.

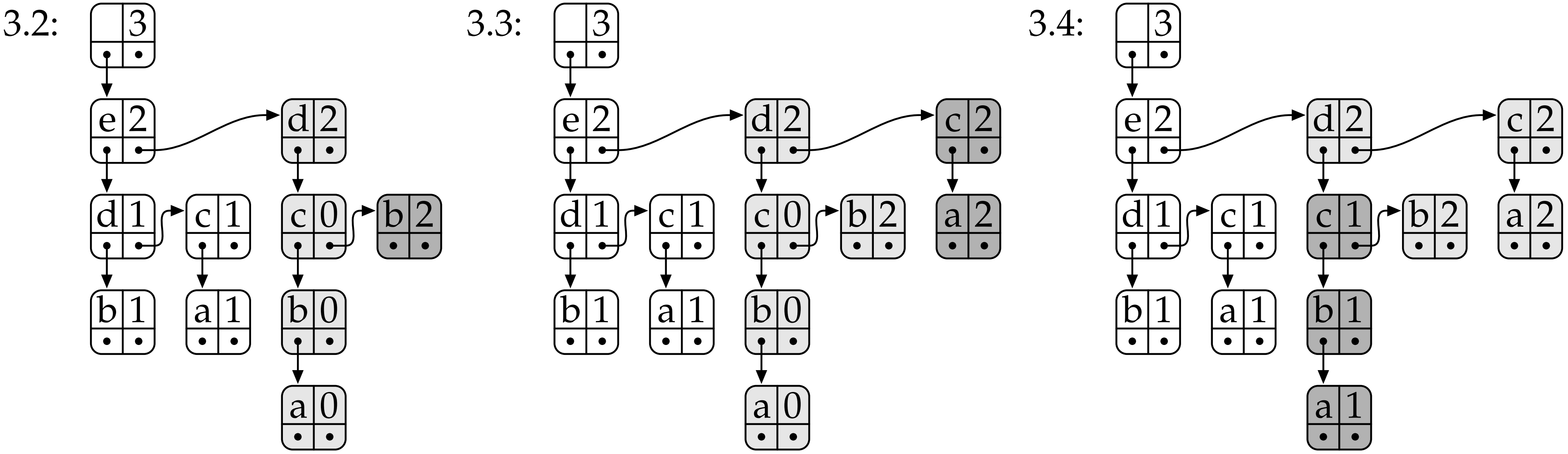
- **Intersecting with and inserting into the tree can be combined.**

Ista: Cumulative Transaction Intersections

transaction database

t_1	$e\ c\ a$
t_2	$e\ d\ b$
t_3	$d\ c\ b\ a$





Ista: Data Structure

```
typedef struct _node {          /* a prefix tree node */
  int step;                     /* most recent update step */
  int item;                     /* assoc. item (last in set) */
  int supp;                     /* support of item set */
  struct _node *sibling;        /* successor in sibling list */
  struct _node *children;       /* list of child nodes */
} NODE;
```

- Standard first child / right sibling node structure.
  - Fixed size of each node allows for optimized allocation.
  - Flexible structure that can easily be extended
- The "step" field indicates whether the support field was already updated.
- The step field is an "incremental marker", so that it need not be cleared in a separate traversal of the prefix tree.

Ista: Pseudo-Code

```
void isect (NODE∗ node, NODE **ins)
{                                  /* intersect with transaction */
  int  i;                          /* buffer for current item */
  NODE *d;                         /* to allocate new nodes */
  while (node) {                   /* traverse the sibling list */
    i = node->item;                /* get the current item */
    if (trans[i]) {                /* if item is in intersection */
      while ((d = ∗ins) && (d→item > i))
        ins = &d->sibling;         /* find the insertion position */
      if (d                        /* if an intersection node with */
      && (d->item == i)) {         /* the item already exists */
        if (d->step >= step) d->supp--;
        if (d->supp < node->supp)
          d->supp = node->supp;
        d->supp++;                 /* update intersection support */
        d->step = step; }          /* and set current update step */
```

Ista: Pseudo-Code

```
      else {                        /* if there is no corresp. node */
        d = malloc(sizeof(NODE));
        d->step = step;             /* create a new node and */
        d->item = i;                /* set item and support */
        d->supp = node->supp+1;
        d->sibling = *ins; *ins = d;
        d->children = NULL;
      }                             /* insert node into the tree */
      if (i <= imin) return;        /* if beyond last item, abort */
      isect(node->children, &d->children); }
    else {                          /* if item is not in intersection */
      if (i <= imin) return;        /* if beyond last item, abort */
      isect(node->children, ins);
    }                               /* intersect with subtree */
    node = node->sibling;           /* go to the next sibling */
  } /* end of while (node) */
} /* isect() */
```

Ista: Keeping the Repository Small

- In practice we will not work with a minimum support $s_{\text{min}} = 1$.
- Removing intersections early, because they do not reach the minimum support is difficult: in principle, enough of the transactions to be processed in the future could contain the item set under consideration.
- Improved processing with item occurrence counters:
  - In an initial pass the frequency of the individual items is determined.
  - The obtained counters are updated with each processed transaction. They always represent the item occurrences in the unprocessed transactions.
- Based on these counters, we can apply the following pruning scheme:
  - Suppose that after having processed k of a total of n transactions the support of a closed item set I is $s_{T_k}(I) = x$.
  - Let y be the minimum of the counter values for the items contained in I.
  - If $x + y < s_{\text{min}}$, then I can be discarded, because it cannot reach s_{min}.

Ista: Keeping the Repository Small

- One has to be careful, though, because I may be needed in order to form subsets, namely those that result from intersections of it with new transactions.

  These subsets may still be frequent, even though I is not.

- As a consequence, an item set I is not simply removed, but those **items are selectively removed** from it that do not occur frequently enough in the remaining transactions.

- Although in this way non-closed item sets may be constructed, no problems for the final output are created:

  - either the reduced item set also occurs as the intersection of enough transactions and thus is closed,

  - or it will not reach the minimum support threshold and then it will not be reported.

Summary Ista

Basic Processing Scheme

- Cumulative intersection of transactions (incremental/on-line/stream mining).
- Combined intersection and repository extensions (one traversal).
- Additional pruning is possible for batch processing.

Advantages

- Effectively linear in the number of items.
- Very fast for transaction databases with many more items than transactions.

Disadvantages

- Exponential in the number of transactions.
- Very slow for transaction databases with many more transactions than items

Software

- `http://www.borgelt.net/ista.html`

Experimental Comparison: Data Sets

- **Yeast**
  Gene expression data for baker's yeast *(saccharomyces cerevisiae)*.
  300 transactions (experimental conditions), about 10,000 items (genes)

- **NCI 60**
  Gene expression data from the Stanford NCI60 Cancer Microarray Project.
  64 transactions (experimental conditions), about 10,000 items (genes)

- **Thrombin**
  Chemical fingerprints of compounds (not) binding to Thrombin
  (a.k.a. fibrinogenase, (activated) blood-coagulation factor II etc.).
  1909 transactions (compounds), 139,351 items (binary features)

- **BMS-Webview-1 transposed**
  A web click stream from a leg-care company that no longer exists.
  497 transactions (originally items), 59602 items (originally transactions).

Experimental Comparison: Programs and Test System

- The Carpenter and IsTa programs are my own implementations.

  Both use the same code for reading the transaction database
  and for writing the found frequent item sets.

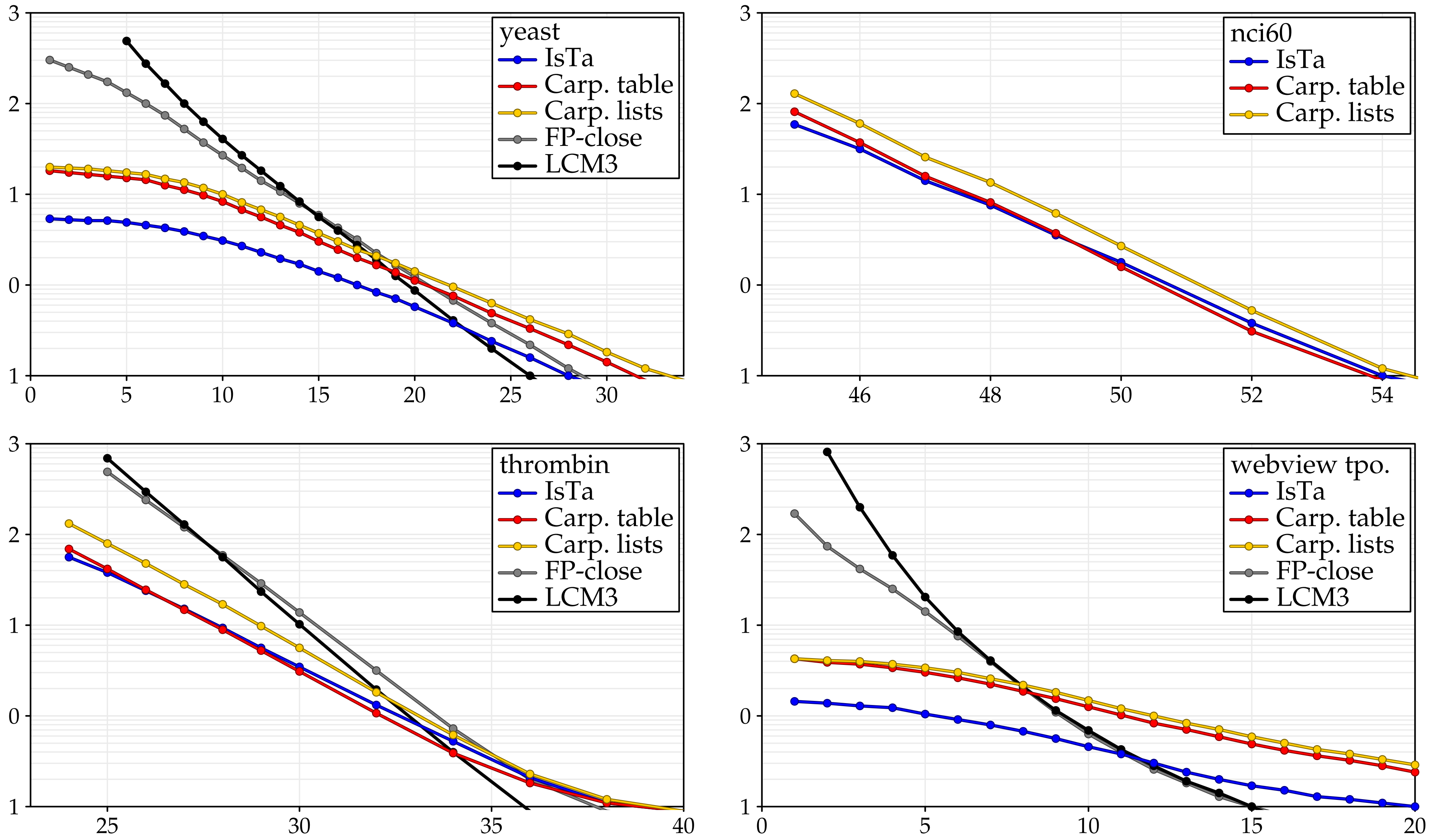
- These programs and their source code can be found on my web site:
  `http://www.borgelt.net/fpm.html`

  ○ Carpenter `http://www.borgelt.net/carpenter.html`
  ○ IsTa `http://www.borgelt.net/ista.html`

- The versions of FP-close (FP-growth with filtering for closed frequent item sets) and LCM3 have been taken from the Frequent Itemset Mining Implementations (FIMI) Repository (see `http://fimi.ua.ac.be/`).
  FP-close won the FIMI Workshop competition in 2003, LCM2 in 2004.

- All tests were run on an Intel Core2 Quad Q9650@3GHz with 8GB memory running Ubuntu Linux 14.04 LTS (64 bit);
  programs were compiled with GCC 4.8.2.

Experimental Comparison: Execution Times



Decimal logarithm of execution time in seconds over absolute minimum support.

Searching for Closed and Maximal Item Sets with Item Set Enumeration

Filtering Frequent Item Sets

- If only closed item sets or only maximal item sets are to be found with item set enumeration approaches, the found frequent item sets have to be filtered.

- Some useful notions for filtering and pruning:

  - The **head** $H \subseteq B$ of a search tree node is the set of items on the path leading to it. It is the prefix of the conditional database for this node.

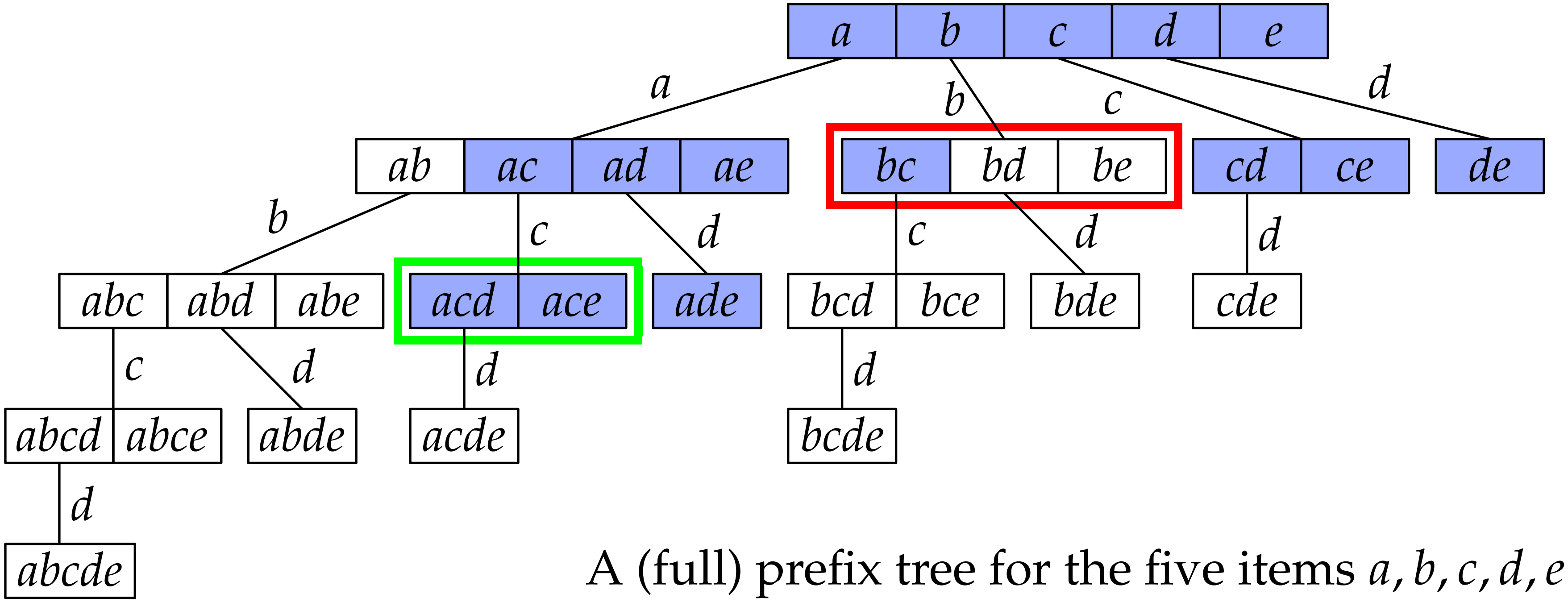
  - The **tail** $L \subseteq B$ of a search tree node is the set of items that are frequent in its conditional database. They are the possible extensions of H.

  - Note that $\forall h \in H : \forall l \in L : \quad h < l$
    (provided the split items are chosen according to a fixed order).

  - $E = \{i \in B - H \mid \exists h \in H : h > i\}$ is the set of **excluded items**.
    These items are not considered anymore in the corresponding subtree.

- Note that the items in the tail and their support in the conditional database are known, at least after the search returns from the recursive processing.

Head, Tail and Excluded Items



A (full) prefix tree for the five items a, b, c, d, e.

- The blue boxes are the frequent item sets.
- For the encircled search tree nodes we have:

red: head $H = \{b\}$, tail $L = \{c\}$, excluded items $E = \{a\}$

green: head $H = \{a, c\}$, tail $L = \{d, e\}$, excluded items $E = \{b\}$

Closed and Maximal Item Sets

- When filtering frequent item sets for closed and maximal item sets the following conditions are easy and efficient to check:
  - If the tail of a search tree node is *not* empty, its head is *not* a maximal item set.
  - If an item in the tail of a search tree node has the same support as the head, the head is *not* a closed item set.
- However, the inverse implications need not hold:
  - If the tail of a search tree node is empty, its head is not necessarily a maximal item set.
  - If no item in the tail of a search tree node has the same support as the head, the head is not necessarily a closed item set.
- The problem are the **excluded items**, which can still render the head non-closed or non-maximal.

Closed and Maximal Item Sets

Check the Defining Condition Directly:

- **Closed Item Sets**:

  Check whether $\exists i \in E: \quad K_T(H) \subseteq K_T(i)$

  or check whether $\displaystyle\bigcap_{k \in K_T(H)} (t_k - H) \neq \emptyset.$

  If either is the case, H is not closed, otherwise it is.

  Note that the intersection can be computed transaction by transaction.
  It can be concluded that H is closed as soon as the intersection becomes empty.

- **Maximal Item Sets:**

  Check whether $\exists i \in E: \quad s_T(H \cup \{i\}) \geq s_{\min}.$

  If this is the case, H is not maximal, otherwise it is.

Closed and Maximal Item Sets

- Checking the defining condition directly is trivial for the tail items, as their support values are available from the conditional transaction databases.

- As a consequence, all item set enumeration approaches for closed and maximal item sets check the defining condition for the tail items.

- However, checking the defining condition can be difficult for excluded items, since additional data (beyond the conditional transaction database) is needed to determine their occurrences in the transactions or their support values.

- It can depend on the database structure used whether a check of the defining condition is efficient for the excluded items or not.

- As a consequence, some item set enumeration algorithms do not check the defining condition for the excluded items, but rely on a repository of already found closed or maximal item sets.

- With such a repository it can be checked in an indirect way whether an item set is closed or maximal.

Checking the Excluded Items: Repository

- Each found maximal or closed item set is stored in a repository.

  (Preferred data structure for the repository: prefix tree)

- It is checked whether a superset of the head H with the same support has already been found. If yes, the head H is neither closed nor maximal.

- Even more: the head H need not be processed recursively, because the recursion cannot yield any closed or maximal item sets. Therefore the current subtree of the search tree can be pruned.

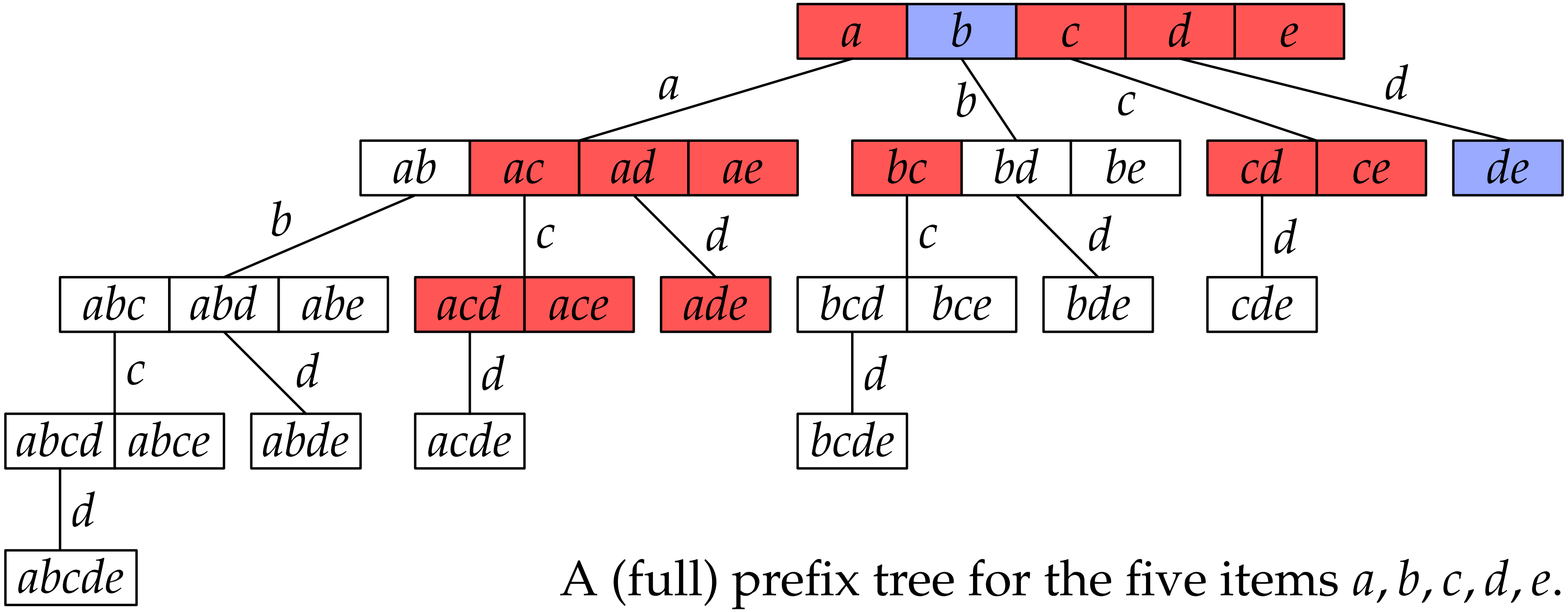
- Note that with a repository depth-first search has to proceed from left to right.

  - We need the repository to check for possibly existing closed or maximal supersets that contain one or more excluded item(s).

  - Item sets containing excluded items are considered only in search tree branches to the left of the considered node.

  - Therefore these branches must already have been processed in order to ensure that possible supersets have already been recorded.

Checking the Excluded Items: Repository



A (full) prefix tree for the five items a, b, c, d, e.

- Suppose the prefix tree would be traversed from right to left.
- For none of the frequent item sets $\{d, e\}$, $\{c, d\}$ and $\{c, e\}$ it could be determined with the help of a repository that they are not maximal, because the maximal item sets $\{a, c, d\}$, $\{a, c, e\}$, $\{a, d, e\}$ have not been processed then.

Checking the Excluded Items: Repository

- If a superset of the current head H with the same support has already been found, the head H need not be processed, because it cannot yield any maximal or closed item sets.

- The reason is that a found proper superset $I \supset H$ with $s_T(I) = s_T(H)$ contains at least one item $i \in I - H$ that is a perfect extension of H.

- The item i is an excluded item, that is, $i \notin L$ (item i is not in the tail). (If i were in L, the set I would not be in the repository already.)

- If the item i is a perfect extension of the head H, it is a perfect extension of all supersets $J \supseteq H$ with $i \notin J$.

- All item sets explored from the search tree node with head H and tail L are subsets of $H \cup L$ (because only the items in L are conditionally frequent).

- Consequently, the item i is a perfect extension of all item sets explored from the search tree node with head H and tail L, and hence none of them can be closed.

Checking the Excluded Items: Repository

- It is usually advantageous to use not just a single, global repository, but to create conditional repositories for each recursive call, which contain only the found closed item sets that contain H.

- With conditional repositories the check for a known superset reduces to the check whether the conditional repository contains an item set with the next split item and the same support as the current head.

  (Note that the check is executed before going into recursion, that is, before constructing the extended head of a child node. If the check finds a superset, the child node is pruned.)

- The conditional repositories are obtained by basically the same operation as the conditional transaction databases (projecting/conditioning on the split item).

- A popular structure for the repository is an FP-tree, because it allows for simple and efficient projection/conditioning.

  However, a simple prefix tree that is projected top-down may also be used.

Closed and Maximal Item Sets: Pruning

- If only closed item sets or only maximal item sets are to be found, additional pruning of the search tree becomes possible.

- **Perfect Extension Pruning / Parent Equivalence Pruning (PEP)**

  - Given an item set I, an item $i \notin I$ is called a **perfect extension** of I, iff the item sets I and $I \cup \{i\}$ have the same support: $s_T(I) = s_T(I \cup \{i\})$ (that is, if all transactions containing I also contain the item i).

    Then we know: $\quad \forall J \supseteq I: \quad s_T(J \cup \{i\}) = s_T(J).$

  - As a consequence, no superset $J \supseteq I$ with $i \notin J$ can be closed. Hence i can be added directly to the prefix of the conditional database.

- Let $X_T(I) = \{i \mid i \notin I \land s_T(I \cup \{i\}) = s_T(I)\}$ be the set of all perfect extension items. Then the whole set $X_T(I)$ can be added to the prefix.

- Perfect extension / parent equivalence pruning can be applied for both closed and maximal item sets, since all maximal item sets are closed.

Head Union Tail Pruning

- If only maximal item sets are to be found, even more additional pruning of the search tree becomes possible.

- **General Idea:** All frequent item sets in the subtree rooted at a node with head H and tail L are subsets of $H \cup L$.

- **Maximal Item Set Contains Head $\cup$ Tail Pruning (MFIHUT)**
  - ○ If we find out that $H \cup L$ is a subset of an already found maximal item set, the whole subtree can be pruned.
  - ○ This pruning method requires a left to right traversal of the prefix tree.

- **Frequent Head $\cup$ Tail Pruning (FHUT)**
  - ○ If $H \cup L$ is not a subset of an already found maximal item set and by some clever means we discover that $H \cup L$ is frequent, $H \cup L$ can immediately be recorded as a maximal item set.

Alternative Description of Closed Item Set Mining

- In order to avoid redundant search in the partially ordered set $(2^B, \subseteq)$, we assigned a unique parent item set to each item set (except the empty set).

- Analogously, we may structure the set of closed item sets by assigning **unique closed parent item sets**. [Uno *et al.* 2003]

- Let $\leq$ be an item order and let I be a closed item set with $I \neq \bigcap_{1 \leq k \leq n} t_k$. Let $i_* \in I$ be the (uniquely determined) item satisfying

$$s_T(\{i \in I \mid i < i_*\}) > s_T(I) \qquad \text{and} \qquad s_T(\{i \in I \mid i \leq i_*\}) = s_T(I).$$

  Intuitively, the item i_* is the greatest item in I that is not a perfect extension. (All items greater than i_* can be removed without affecting the support.)
  Let $I_* = \{i \in I \mid i < i_*\}$ and $X_T(I) = \{i \in B - I \mid s_T(I \cup \{i\}) = s_T(I)\}$.
  Then the canonical parent $\pi_C(I)$ of I is the item set

$$\pi_C(I) = I_* \cup \{i \in X_T(I_*) \mid i > i_*\}.$$

  Intuitively, to find the canonical parent of the item set I, the reduced item set I_* is enhanced by all perfect extension items following the item i_*.

Alternative Description of Closed Item Set Mining

- Note that $\bigcap_{1 \leq k \leq n} t_k$ is the smallest closed item set for a given database T.

- Note also that the set $\{i \in X_T(I_*) \mid i > i_*\}$ need not contain all items $i > i_*$, because a perfect extension of $I_* \cup \{i_*\}$ need not be a perfect extension of I_*, since $K_T(I_*) \supset K_T(I_* \cup \{i_*\})$.

- For the recursive search, the following formulation is useful:

  Let $I \subseteq B$ be a closed item set. The **canonical children** of I (that is, the closed item sets that have I as their canonical parent) are the item sets

  $$J = I \cup \{i\} \cup \{j \in X_T(I \cup \{i\}) \mid j > i\}$$

  with $\forall j \in I : i > j$ and $\{j \in X_T(I \cup \{i\}) \mid j < i\} = X_T(J) = \emptyset$.

- The union with $\{j \in X_T(I \cup \{i\}) \mid j > i\}$ represents perfect extension or parent equivalence pruning: all perfect extensions in the tail of $I \cup \{i\}$ are immediately added.

- The condition $\{j \in X_T(I \cup \{i\}) \mid j < i\} = \emptyset$ expresses that there must not be any perfect extensions among the excluded items.

Experiments: Reminder

- **Chess**
  A data set listing chess end game positions for king vs. king and rook.
  This data set is part of the UCI machine learning repository.

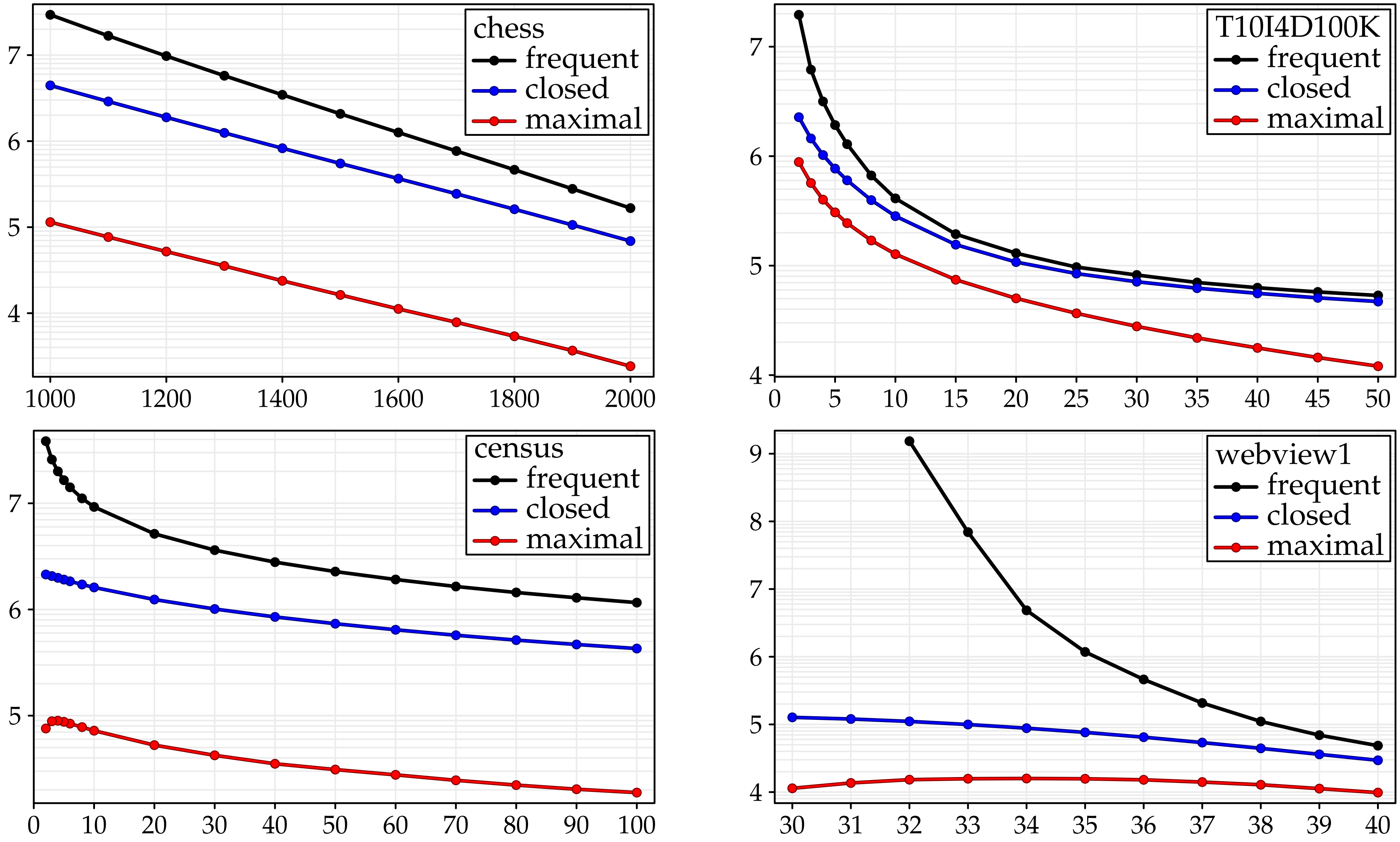
- **Census**
  A data set derived from an extract of the US census bureau data of 1994, which was preprocessed by discretizing numeric attributes.
  This data set is part of the UCI machine learning repository.

- **T10I4D100K**
  An artificial data set generated with IBM's data generator.
  The name is formed from the parameters given to the generator (for example: 100K = 100000 transactions).

- **BMS-Webview-1**
  A web click stream from a leg-care company that no longer exists.
  It has been used in the KDD cup 2000 and is a popular benchmark.

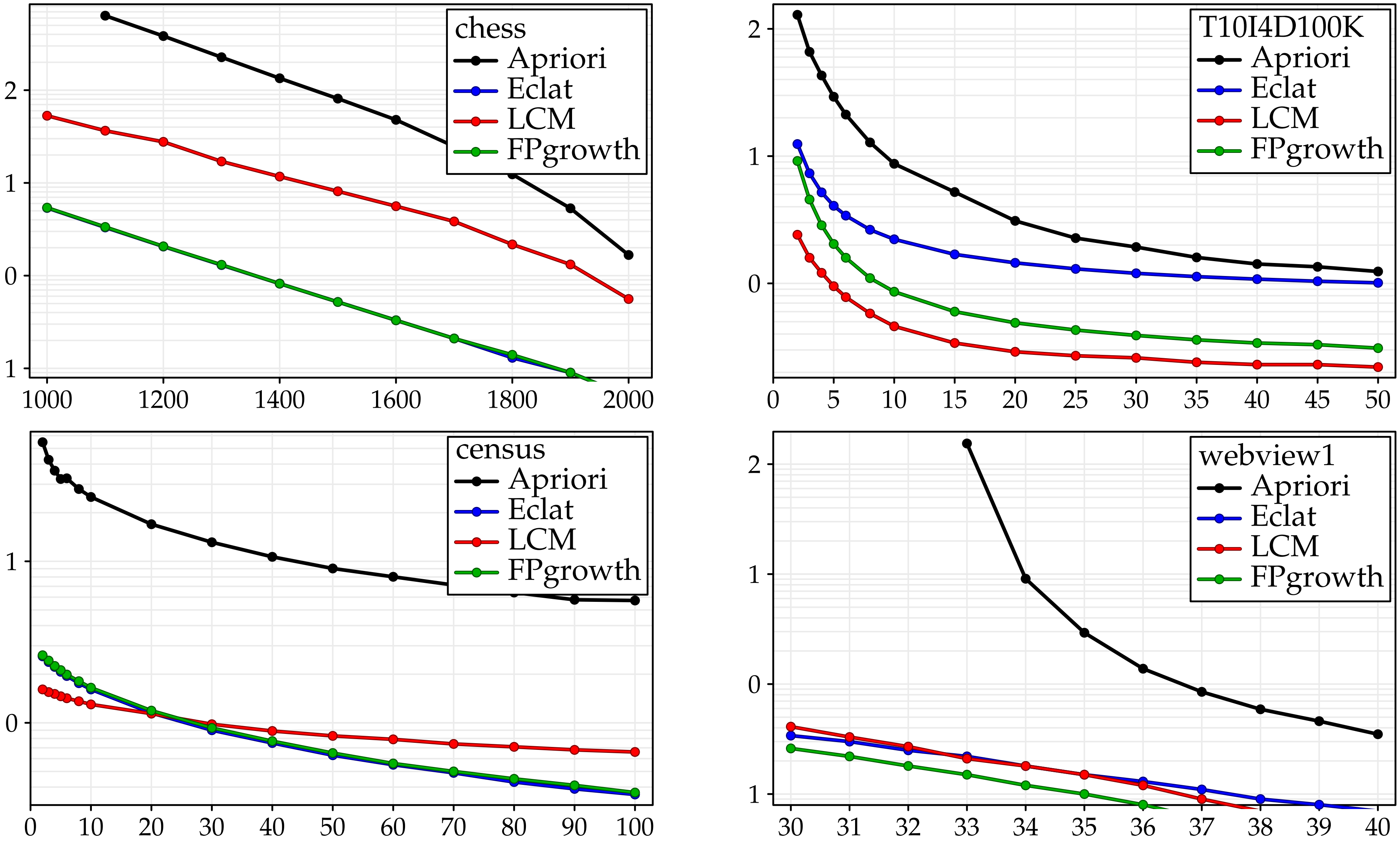
- All tests were run on an Intel Core2 Quad Q9650@3GHz with 8GB memory running Ubuntu Linux 14.04 LTS (64 bit); programs compiled with GCC 4.8.2.

Types of Frequent Item Sets



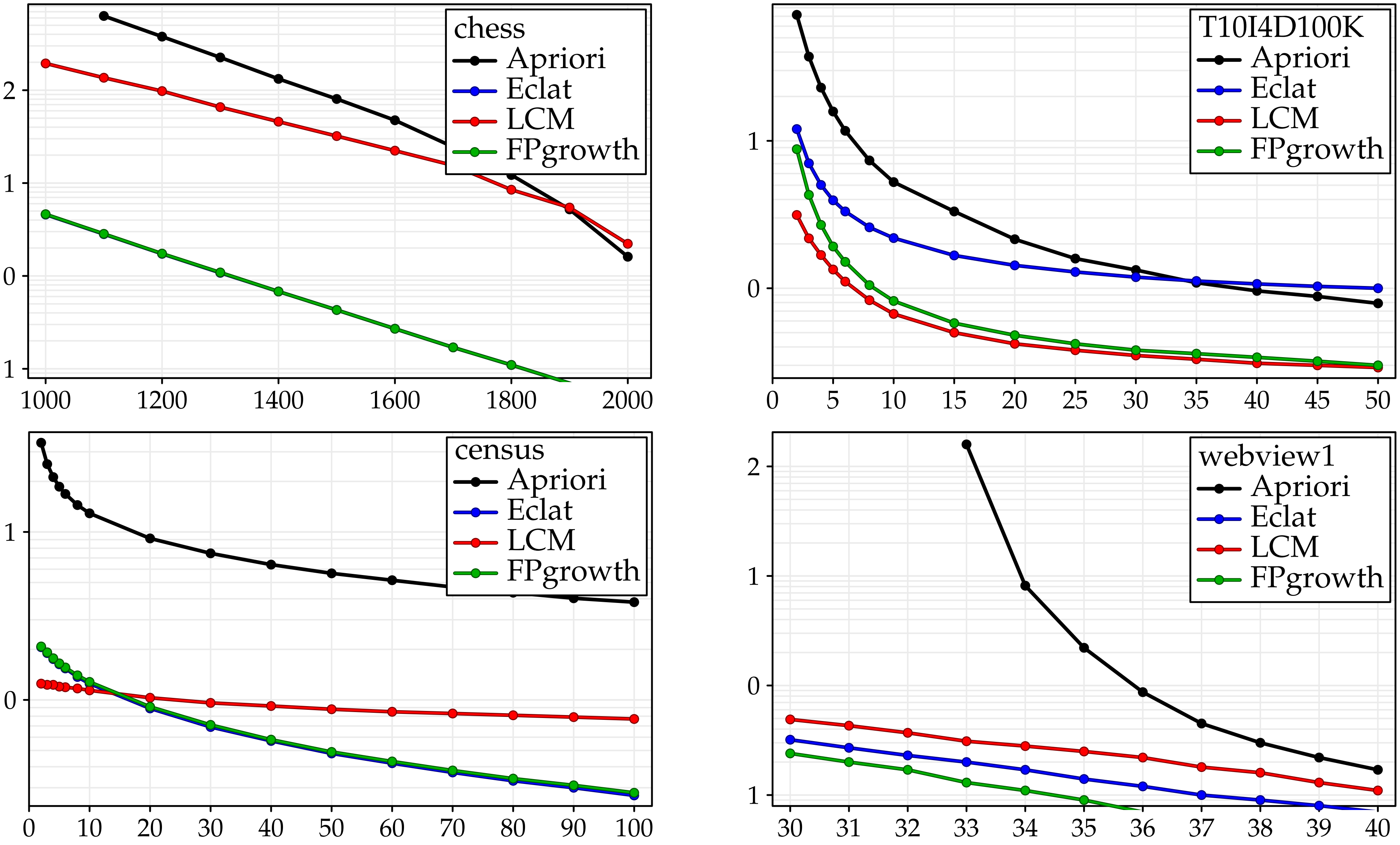
Decimal logarithm of the number of item sets over absolute minimum support.

Experiments: Mining Closed Item Sets



Decimal logarithm of execution time in seconds over absolute minimum support.

Experiments: Mining Maximal Item Sets



Decimal logarithm of execution time in seconds over absolute minimum support.

Additional Frequent Item Set Filtering

Additional Frequent Item Set Filtering

- **General problem of frequent item set mining:**

  The number of frequent item sets, even the number of closed or maximal item sets, can exceed the number of transactions in the database by far.

- Therefore: Additional filtering is necessary to find
  the ′′relevant″ or "interesting″ frequent item sets.

- General idea: **Compare support to expectation.**
  - Item sets consisting of items that appear frequently
    are likely to have a high support.
  - However, this is not surprising:
    we expect this even if the occurrence of the items is independent.
  - Additional filtering should remove item sets with a support
    close to the support expected from an independent occurrence.

Additional Frequent Item Set Filtering

Full Independence

- Evaluate item sets with

  $$\varrho_{\text{fi}}(I) = \frac{s_T(I) \cdot n^{|I|-1}}{\prod_{i \in I} s_T(\{i\})} = \frac{\hat{p}_T(I)}{\prod_{i \in I} \hat{p}_T(\{i\})}.$$

  and require a minimum value for this measure.
  ($\hat{p}_T$ is the probability estimate based on T.)

- Assumes full independence of the items in order to form an expectation about the support of an item set.

- Advantage: Can be computed from only the support of the item set and the support values of the individual items.

- Disadvantage: If some item set I scores high on this measure, then all $J \supset I$ are also likely to score high, even if the items in $J - I$ are independent of I.

Additional Frequent Item Set Filtering

Incremental Independence

- Evaluate item sets with

$$\varrho_{\text{ii}}(I) \;=\; \min_{i \in I} \frac{n\, s_T(I)}{s_T(I - \{i\}) \cdot s_T(\{i\})} \;=\; \min_{i \in I} \frac{\hat{p}_T(I)}{\hat{p}_T(I - \{i\}) \cdot \hat{p}_T(\{i\})}.$$

  and require a minimum value for this measure.
  ($\hat{p}_T$ is the probability estimate based on T.)

- Advantage: If I contains independent items,
  the minimum ensures a low value.

- Disadvantages: We need to know the support values of all subsets $I - \{i\}$.

  If there exist high scoring independent subsets I_1 and I_2
  with $|I_1| > 1$, $|I_2| > 1$, $I_1 \cap I_2 = \emptyset$ and $I_1 \cup I_2 = I$,
  the item set I still receives a high evaluation.

Additional Frequent Item Set Filtering

Subset Independence

- Evaluate item sets with

$$\varrho_{\text{si}}(I) \;=\; \min_{J \subset I, J \neq \emptyset} \frac{n\, s_T(I)}{s_T(I-J) \cdot s_T(J)} \;=\; \min_{J \subset I, J \neq \emptyset} \frac{\hat{p}_T(I)}{\hat{p}_T(I-J) \cdot \hat{p}_T(J)}.$$

  and require a minimum value for this measure.
  ($\hat{p}_T$ is the probability estimate based on T.)

- Advantage: Detects all cases where a decomposition is possible and evaluates them with a low value.

- Disadvantages: We need to know the support values of all proper subsets J.

- Improvement: Use incremental independence and in the minimum consider only items $\{i\}$ for which $I - \{i\}$ has been evaluated high.

  This captures subset independence “incrementally”.

Summary Frequent Item Set Mining

- With a **canonical form** of an item set the Hasse diagram can be turned into a much simpler **prefix tree** (⇒ divide-and-conquer scheme using conditional databases).

- **Item set enumeration** algorithms differ in:

  ◦ the **traversal order** of the prefix tree: (breadth-first/levelwise versus depth-first traversal)

  ◦ the **transaction representation**: *horizontal* (item arrays) versus *vertical* (transaction lists) versus *specialized data structures* like FP-trees

  ◦ the **types of frequent item sets** found: *frequent* versus *closed* versus *maximal item sets* (additional pruning methods for closed and maximal item sets)

- An alternative are **transaction set enumeration** or **intersection** algorithms.

- **Additional filtering** is necessary to reduce the size of the output.

Example Application:

Finding Neuron Assemblies in Neural Spike Data

Biological Background

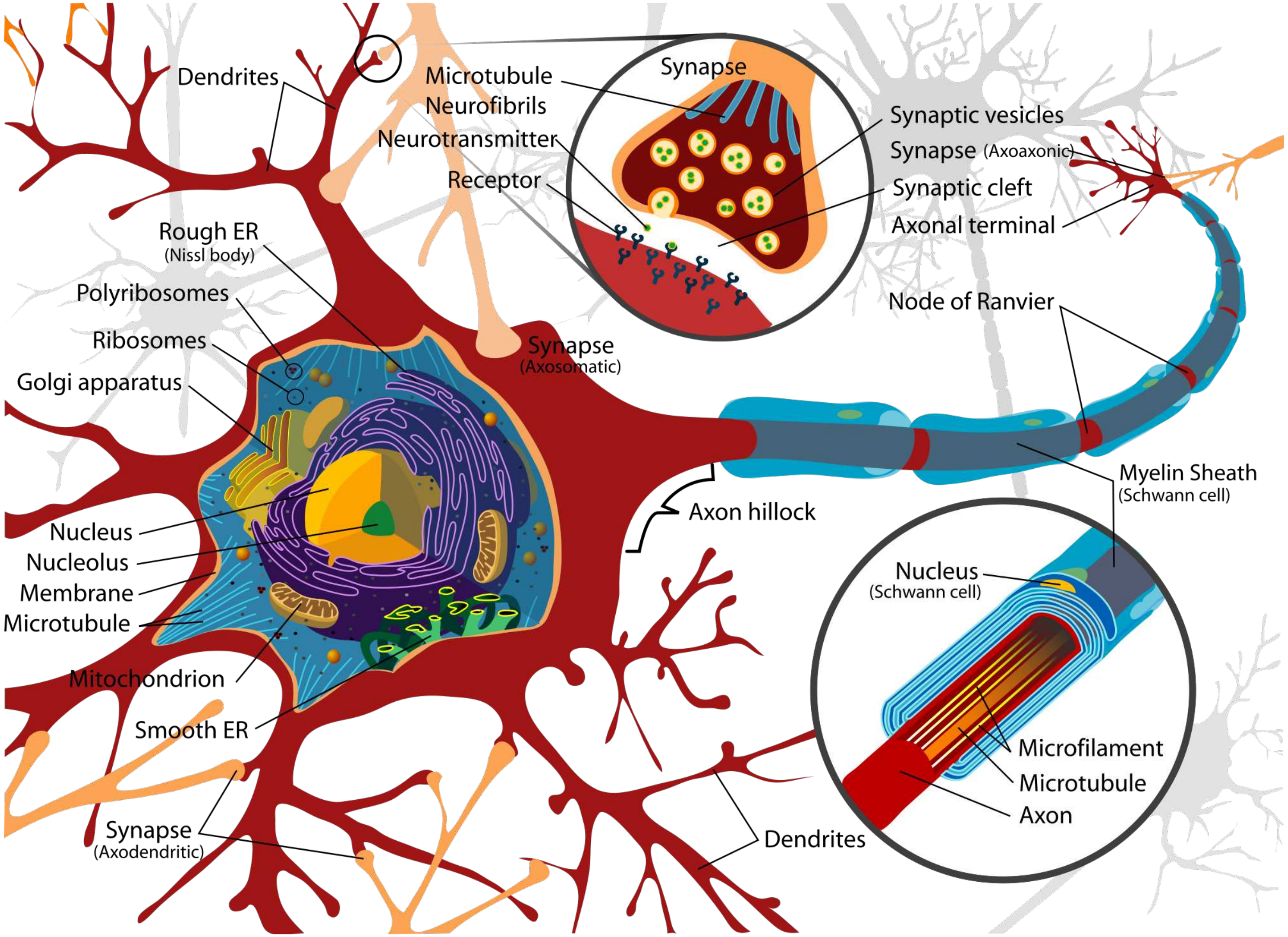
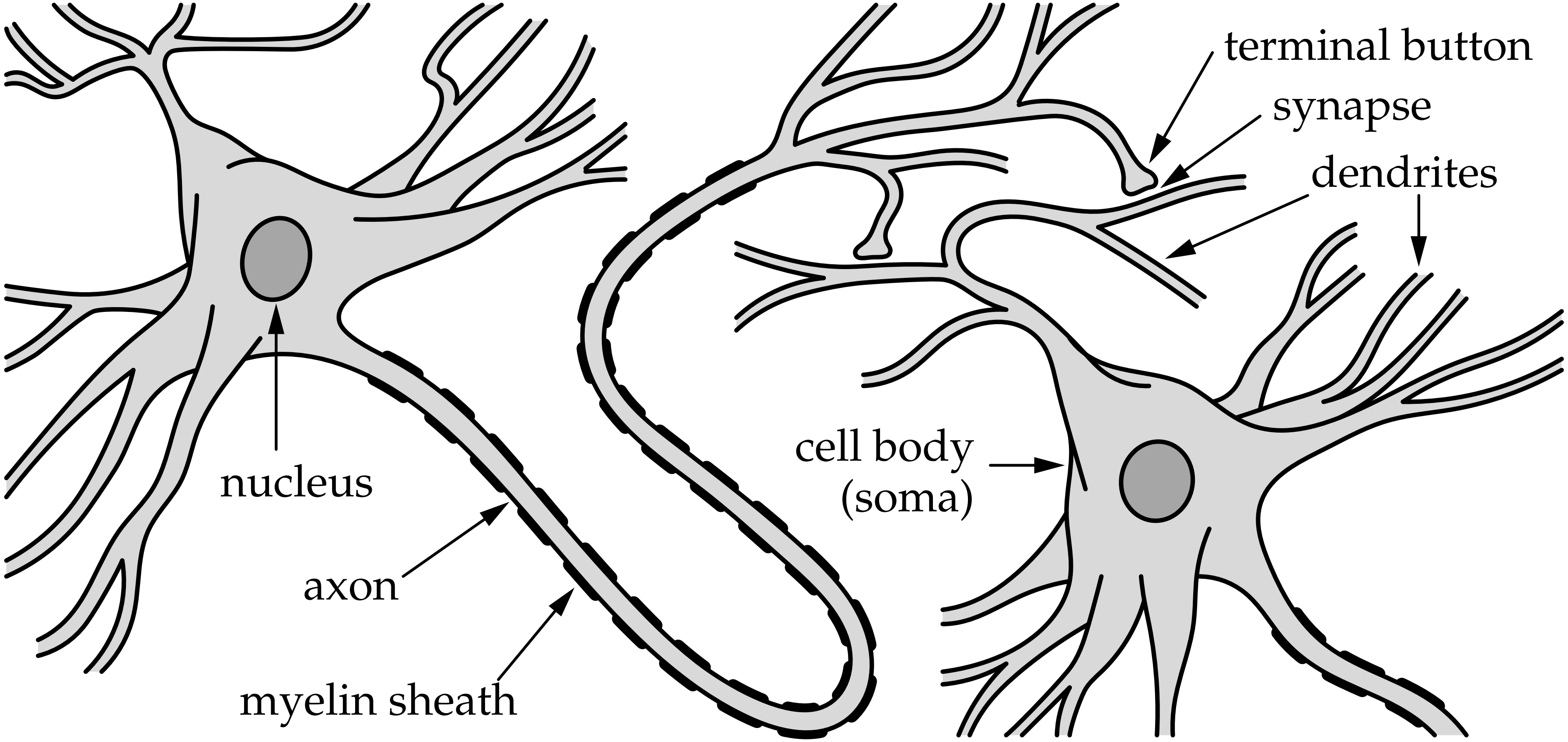


Diagram of a typical myelinated vertebrate motoneuron (source: Wikipedia, Ruiz-Villarreal 2007), showing the main parts involved in its signaling activity like the *dendrites*, the *axon*, and the *synapses*.

Biological Background

Structure of a prototypical neuron (simplified)



Biological Background

(Very) simplified description of neural information processing

- Axon terminal releases chemicals, called **neurotransmitters**.
- These act on the membrane of the receptor dendrite to change its polarization.
  (The inside is usually 70mV more negative than the outside.)
- Decrease in potential difference: **excitatory** synapse
  Increase in potential difference: **inhibitory** synapse
- If there is enough net excitatory input, the axon is depolarized.
- The resulting **action potential** travels along the axon.
  (Speed depends on the degree to which the axon is covered with myelin.)
- When the action potential reaches the terminal buttons,
  it triggers the release of neurotransmitters.

Recording the Electrical Impulses (Spikes)

pictures not available in online version

Signal Filtering and Spike Sorting

picture not available in online version

An actual recording of the electrical potential also contains the so-called **local field potential (LFP)**, which is dominated by the electrical current flowing from all nearby dendritic synaptic activity within a volume of tissue. The LFP is removed in a preprocessing step (high-pass filtering, ∼300Hz).

picture not available in online version

Spikes are detected in the filtered signal with a simple threshold approach. Aligning all detected spikes allows us to distinguishing multiple neurons based on the shape of their spikes. This process is called **spike sorting**.

Multi-Electrode Recording Devices

picture not available
in online version

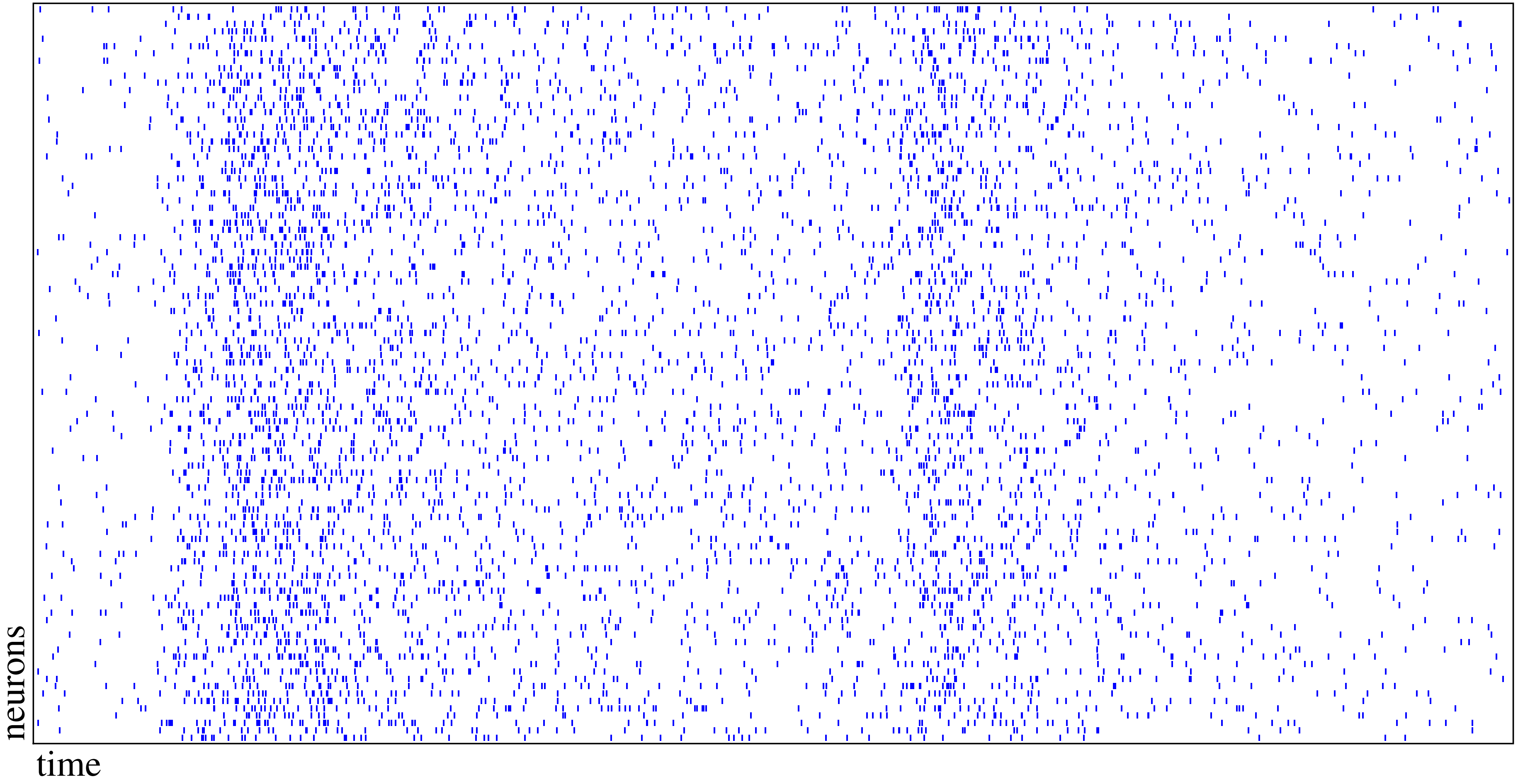
Several types of multi-electrode recording devices have been developed in recent years and are in frequent use nowadays.

Disadvantage of these devices:
need to be surgically implanted.

Advantages: High resolution in time, space and electrical potential.

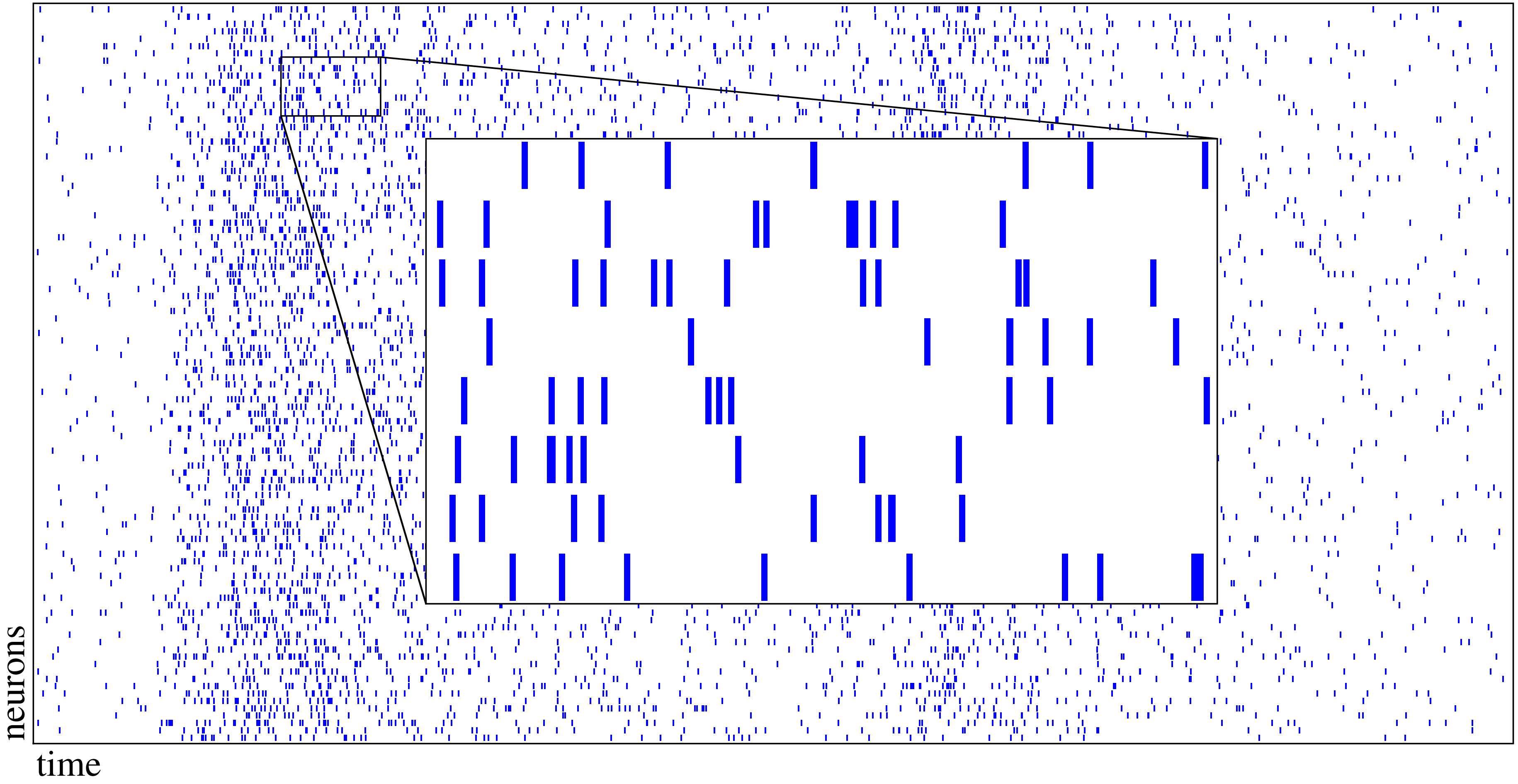
pictures not available in online version

Dot Displays of Parallel Spike Trains



- Simulated data, 100 neurons, 3 seconds recording time.
- Each blue dot/vertical bar represents one spike.

Dot Displays of Parallel Spike Trains



- Simulated data, 100 neurons, 3 seconds recording time.
- Each blue dot/vertical bar represents one spike.

Higher Level Neural Processing

- The **low-level mechanisms** of neural information processing are fairly well understood (neurotransmitters, excitation and inhibition, action potential).
- The **high-level mechanisms**, however, are a topic of current research. There are several competing theories (see the following slides) how neurons code and transmit the information they process.
- Up to fairly recently it was not possible to record the spikes of enough neurons in parallel to decide between the different models.

  However, new measurement techniques open up the possibility to record dozens or even up to a hundred neurons in parallel.
- Currently methods are investigated by which it would be possible to check the validity of the different coding models.
- Frequent item set mining, properly adapted, could provide a method to test the **temporal coincidence coding hypothesis** (see below).

Models of Neural Coding

picture not available in online version

Frequency Code Hypothesis
[Sherrington 1906, Eccles 1957, Barlow 1972]

Neurons generate different frequency of spike trains as a response to different stimulus intensities.

Models of Neural Coding

picture not available in online version

Temporal Coincidence Hypothesis
[Gray et al. 1992, Singer 1993, 1994]

Spike occurrences are modulated by local field oscillation (gamma).
Tighter coincidence of spikes recorded from different neurons
represent higher stimulus intensity.

Models of Neural Coding

picture not available in online version

Delay Coding Hypothesis
[Hopfield 1995, Buzsáki and Chrobak 1995]

The input current is converted to the spike delay.
Neuron 1 which was stimulated stronger reached the threshold earlier
and initiated a spike sooner than neurons stimulated less.
Different delays of the spikes (d2-d4) represent
relative intensities of the different stimuli.

Models of Neural Coding

picture not available in online version

Spatio-Temporal Code Hypothesis

Neurons display a causal sequence of spikes in relationship to a stimulus configuration. The stronger stimulus induces spikes earlier and will initiate spikes in the other, connected cells in the order of relative threshold and actual depolarization. The sequence of spike propagation is determined by the spatio-temporal configuration of the stimulus as well as the intrinsic connectivity of the network. Spike sequences coincide with the local field activity. Note that this model integrates both the temporal coincidence and the delay coding principles.

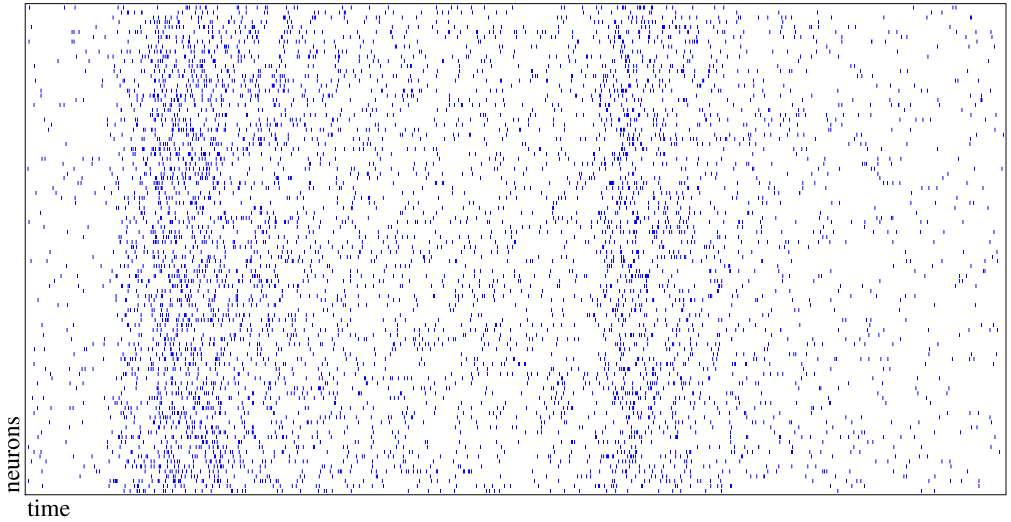
Models of Neural Coding

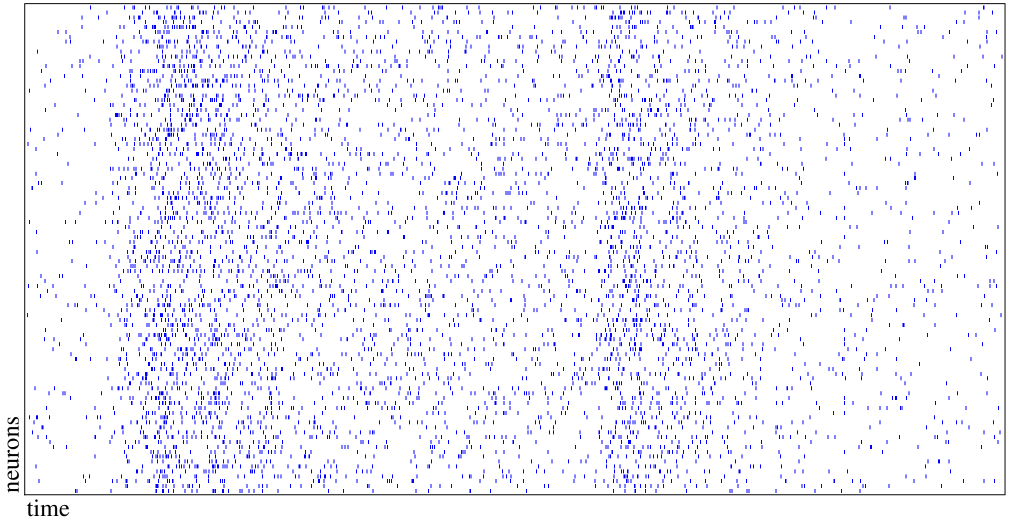
picture not available in online version

Markovian Process of Frequency Modulation
[Seidermann et al. 1996]

Stimulus intensities are converted to a sequence of frequency enhancements and decrements in the different neurons. Different stimulus configurations are represented by different Markovian sequences across several seconds.

Finding Neuron Assemblies in Neural Spike Data





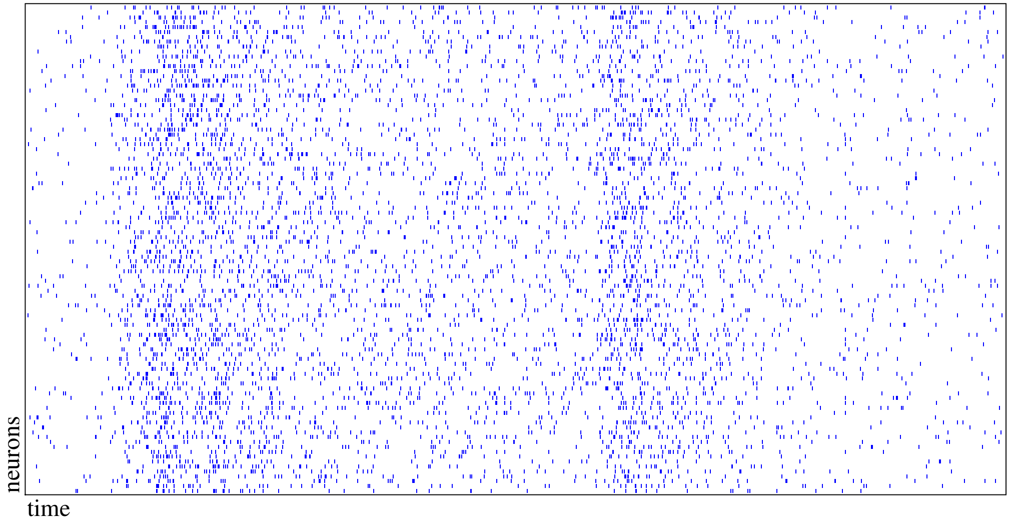
- Dot displays of (simulated) parallel spike trains.

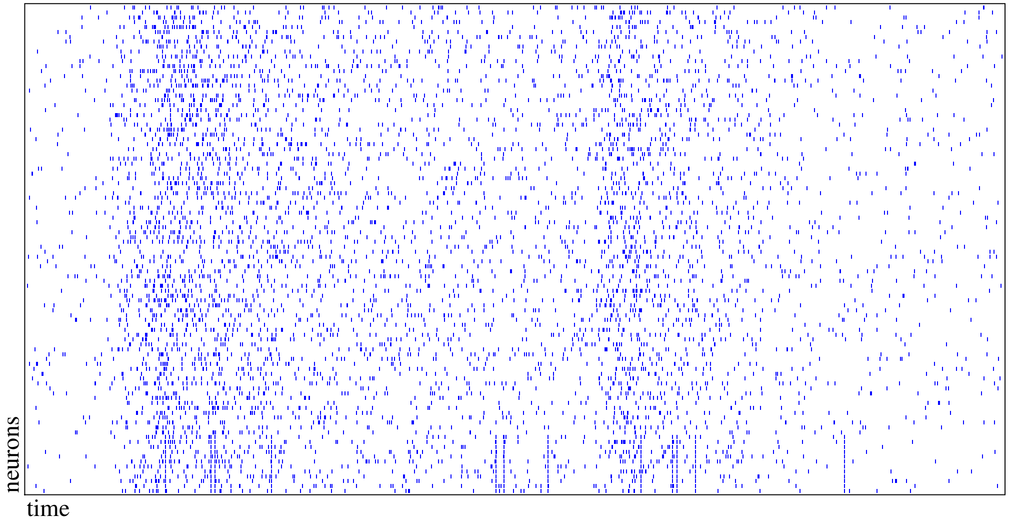
vertical:	neurons	(100)
horizontal:	time	(3 seconds)

- In one of these dot displays, 12 neurons are firing synchronously.

- Without proper frequent pattern mining methods,
  it is virtually impossible to detect such synchronous firing.

Finding Neuron Assemblies in Neural Spike Data





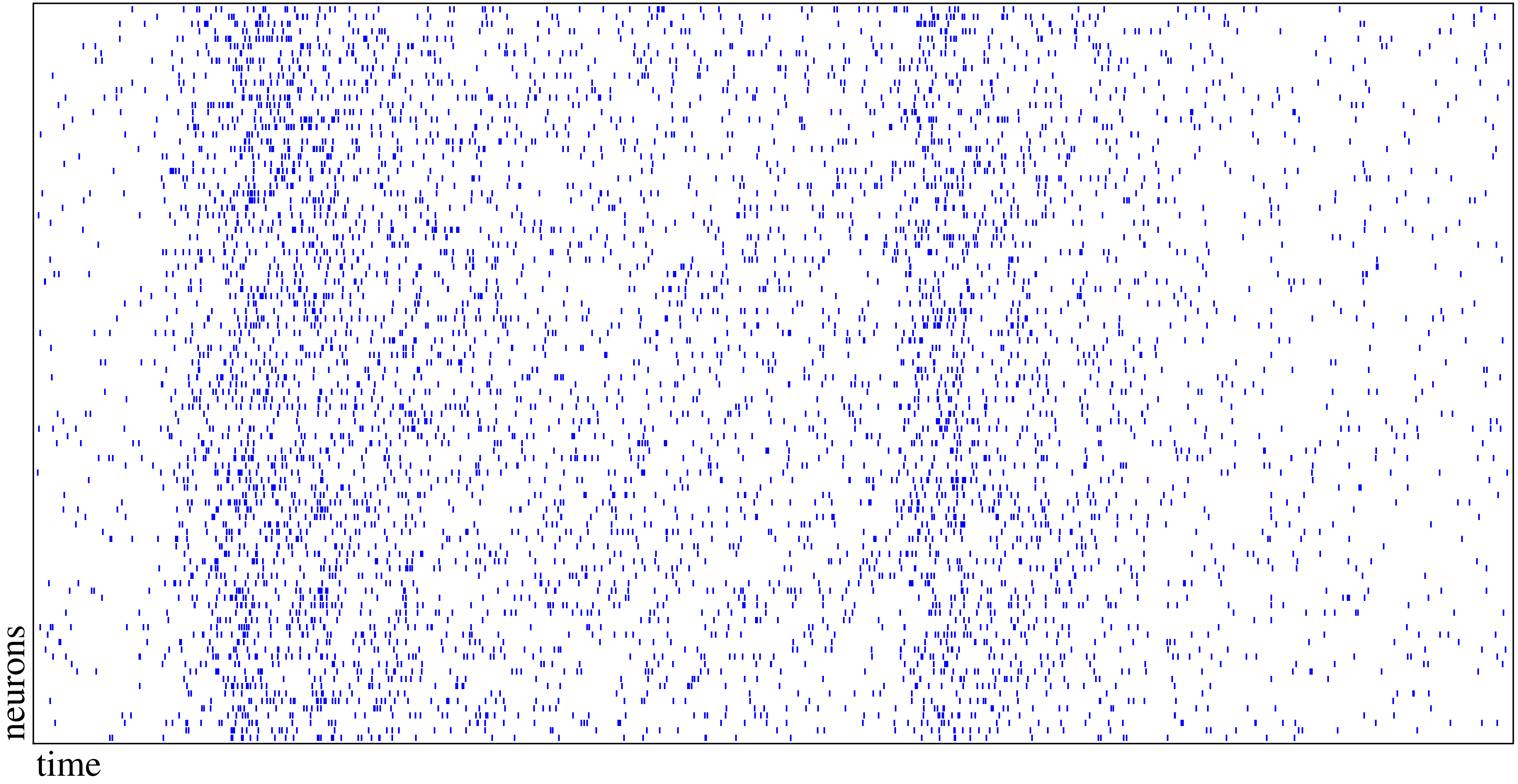
- If the neurons that fire together are grouped together, the synchronous firing becomes easily visible.

  left: copy of the diagram on the right of the preceding slide
  right: same data, but with relevant neurons collected at the bottom.

- A synchronously firing set of neurons is called a **neuron assembly**.

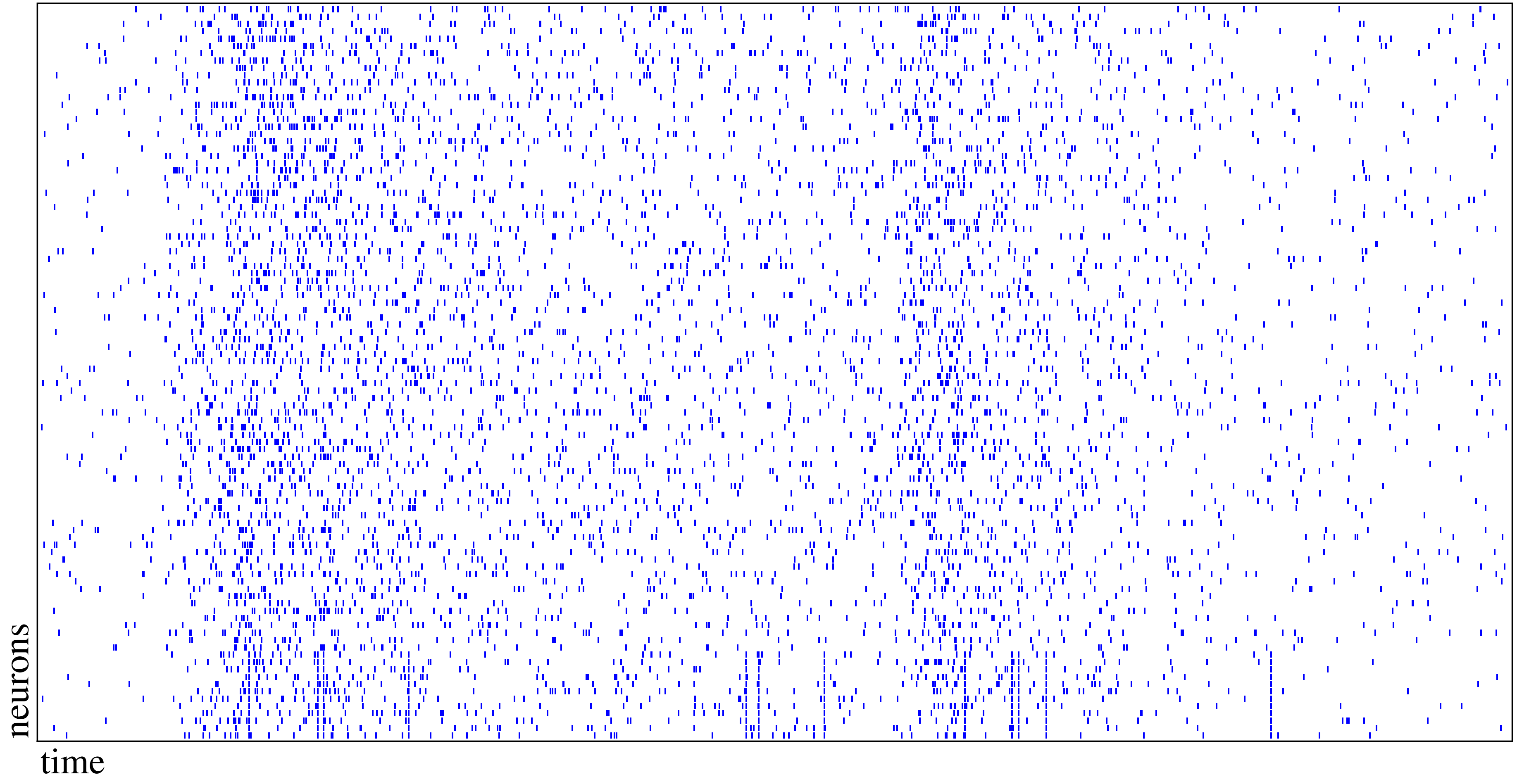
- Question: How can we find out which neurons to group together?

Finding Neuron Assemblies in Neural Spike Data



- Simulated data, 100 neurons, 3 seconds recording time.
- There are 12 neurons that fire synchronously 12 times.

Finding Neuron Assemblies in Neural Spike Data



- Simulated data, 100 neurons, 3 seconds recording time.
- Moving the neurons of the assembly to the bottom makes the synchrony visible.

Finding Neuron Assemblies in Neural Spike Data

A Frequent Item Set Mining Approach

- The neuronal spike trains are usually coded as pairs of a neuron id and a spike time, sorted by the spike time.
- In order to make frequent item set mining applicable, time bins are formed.
- Each **time bin** gives rise to one **transaction**.
  It contains the set of **neurons** that fire in this time bin (**items**).
- Frequent item set mining, possibly restricted to maximal item sets, is then applied with additional filtering of the frequent item sets.
- For the (simulated) example data set such an approach detects the neuron assembly perfectly:

  ```
  73 66 20 53 59 72 19 31 34 9 57 17
  ```

Finding Neuron Assemblies in Neural Spike Data

Translation of Basic Notions

mathematical problem	market basket analysis	spike train analysis
item	product	neuron
item base	set of all products	set of all neurons
— (transaction id)	customer	time bin
transaction	set of products bought by a customer	set of neurons firing in a time bin
frequent item set	set of products frequently bought together	set of neurons frequently firing together

- In both cases the input can be represented as a binary matrix (the so-called *dot display* in spike train analysis).

- Note, however, that a dot display is usually rotated by 90^o:
usually customers refer to rows, products to columns,
but in a dot display, rows are neurons, columns are time bins.

Finding Neuron Assemblies in Neural Spike Data

Core Problems of Detecting Synchronous Patterns:

- **Multiple Testing**
  If many statistical tests are conducted, one loses control of the significance level.
  For fairly small numbers of tests, effective correction procedures exist.
  However, the number of potential patterns and the number of tests is huge.

- **Induced Patterns**
  If synchronous spiking activity is present in the data, not only the actual assembly, but also subsets, supersets and overlapping sets of neurons are detected.

- **Temporal Imprecision**
  The spikes of neurons that participate in synchronous spiking cannot be expected to be perfectly synchronous.

- **Selective Participation**
  Varying subsets of the neurons in an assembly may participate in different synchronous spiking events.

Neural Spike Data: Multiple Testing

- If 1000 tests are carried out, each with a significance level $\alpha = 0.01 = 1\%$, around 10 tests will turn out positive, signifying nothing.
  The positive test results can be explained as mere chance events.

- Example: 100 recorded neurons allow for $\binom{100}{3} = 161,700$ triplets and $\binom{100}{4} = 3,921,225$ quadruplets.

- As a consequence, even though it is very unlikely that, say, **four specific neurons** fire together three times if they are independent, it is fairly likely that we observe **some set of four neurons** firing together three times.

- Example: 100 neurons, 20Hz firing rate, 3 seconds recording time, binned with 3ms time bins to obtain 1000 transactions.

  The event of 4 neurons firing together 3 times has a p-value of $\leq 10^{-6}$ (χ^2-test).

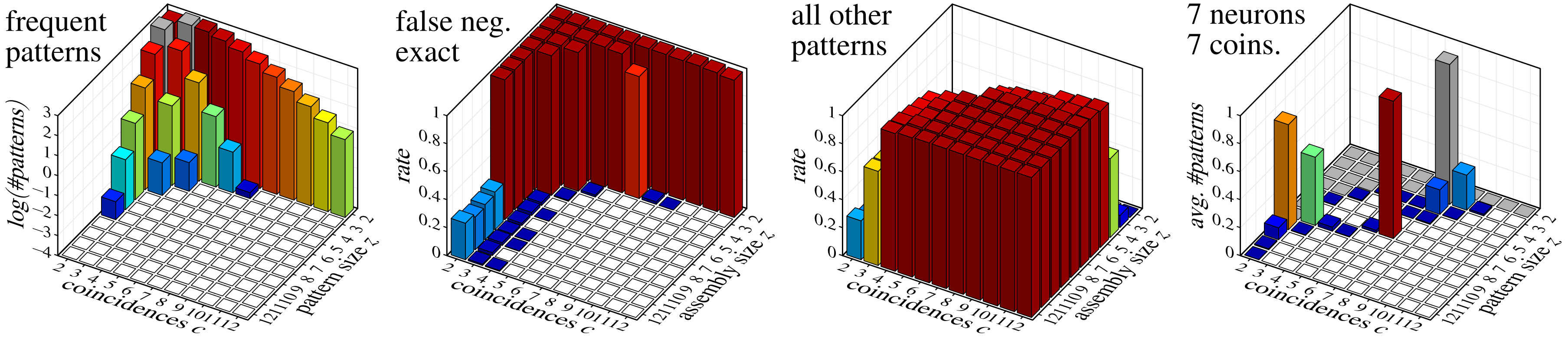
  The average number of such patterns in independent data is greater than 1 (data generated as independent Poisson processes).

Neural Spike Data: Multiple Testing

- Solution: shift statistical testing to **pattern signatures** $\langle z, c \rangle$,
  where z is the number of neurons (pattern size)
  and c the number of coincidences (pattern support). [Picado-Muiño *et al.* 2013]

- Represent null hypothesis by generating sufficiently many **surrogate data sets**
  (e.g. by spike time randomization for constant firing rate).
  (Surrogate data generation must take data properties into account.)

- Remove all patterns found in the original data set for which a counterpart
  (same signature) was found in some surrogate data set (closed item sets).
  (Idea: a counterpart indicates that the pattern could be a chance event.)

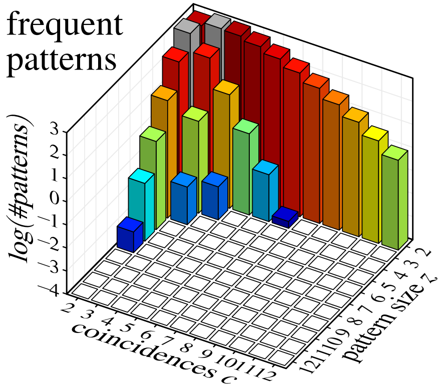


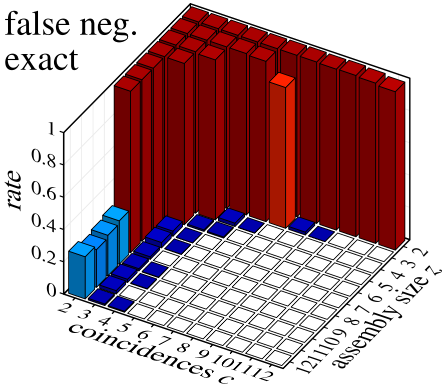
Neural Spike Data: Induced Patterns

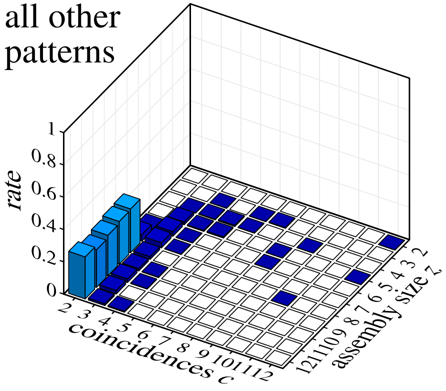
- Let A and B with $B \subset A$ be two sets left over after primary pattern filtering, that is, after removing all sets I with signatures $\langle z_I, c_I \rangle = \langle |I|, s(I) \rangle$ that occur in the surrogate data sets.

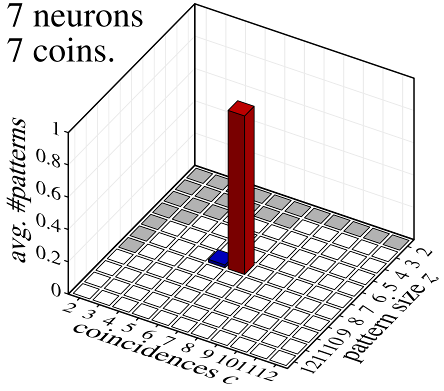
- The set A is *preferred* to the set B iff $(z_A - 1)c_A \geq (z_B - 1)c_B$, that is, if the pattern A covers at least as many spikes as the pattern B if one neuron is neglected. Otherwise B is preferred to A. (This method is simple and effective, but there are several alternatives.)

- Pattern set reduction keeps only sets that are preferred to all of their subsets and to all of their supersets. [Torre *et al.* 2013]









Neural Spike Data: Temporal Imprecision

The most common approach to cope with temporal imprecision, namely **time binning**, has several drawbacks:

- **Boundary Problem:**
  Spikes almost as far apart as the bin width are synchronous if they fall into the same bin, but spikes close together are not seen as synchronous if a bin boundary separates them.

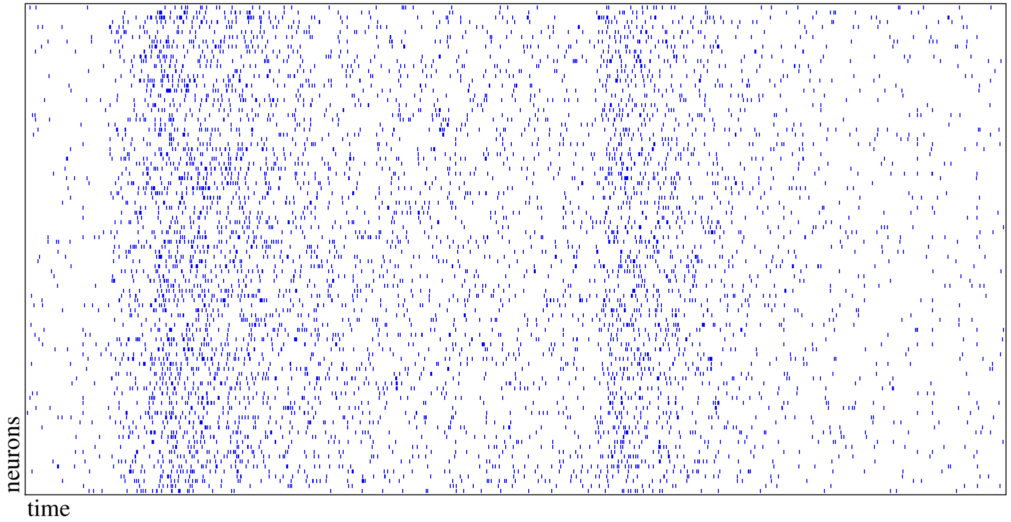
- **Bivalence Problem:**
  Spikes are either synchronous (same time bin) or not,
  no graded notion of synchrony (precision of coincidence).

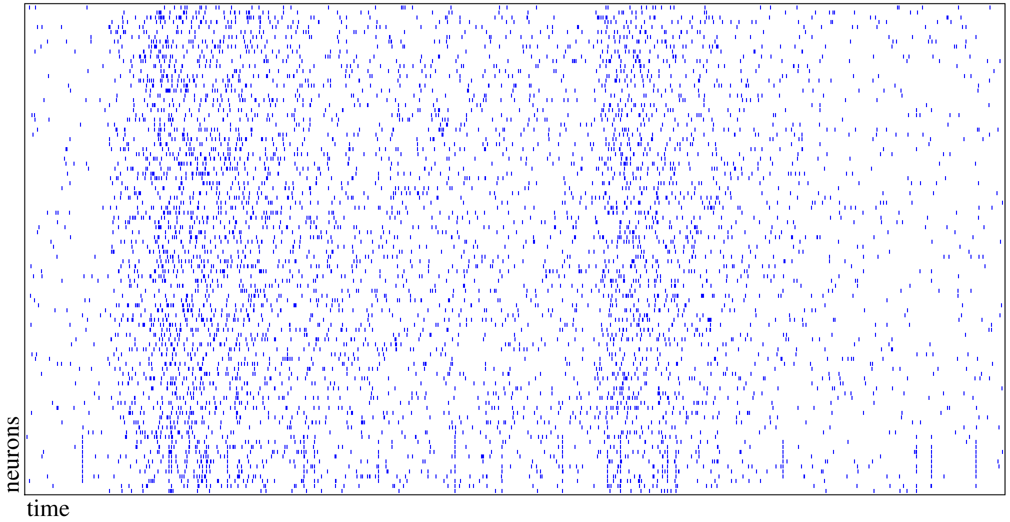
It is desirable to have **continuous time approaches**
that allow for a **graded notion of synchrony**.

Solution: **CoCoNAD** (Continuous time ClOsed Neuron Assembly Detection)

- Extends frequent item set mining to point processes.
- Based on sliding window and MIS computation.
  [Borgelt and Picado-Muiño 2013, Picado-Muiño and Borgelt 2014]

Neural Spike Data: Selective Participation





- Both diagrams show the same (simulated) data, but on the right the 20 neurons of the assembly are collected at the bottom.

- Only about 75% of the neurons (randomly chosen) participate in each synchronous firing. Hence there is no frequent item set comprising all of them.

- Rather a frequent item set mining approach finds a large number of frequent item sets with 12 to 16 neurons.

- Possible approach: **fault-tolerant frequent item set mining**.

Association Rules

Association Rules: Basic Notions

- Often found patterns are expressed as **association rules**, for example:

  If a customer buys **bread** and **wine**,
  then she/he will probably also buy **cheese**.

- Formally, we consider rules of the form $X \to Y$, with $X, Y \subseteq B$ and $X \cap Y = \emptyset$.

- **Support of a Rule** $X \to Y$:

  Either: $\varsigma_T(X \to Y) = \sigma_T(X \cup Y)$ (more common: rule is correct)

  Or: $\varsigma_T(X \to Y) = \sigma_T(X)$ (more plausible: rule is applicable)

- **Confidence of a Rule** $X \to Y$:

$$c_T(X \to Y) = \frac{\sigma_T(X \cup Y)}{\sigma_T(X)} = \frac{s_T(X \cup Y)}{s_T(X)} = \frac{s_T(I)}{s_T(X)}$$

  The confidence can be seen as an estimate of $P(Y \mid X)$.

Association Rules: Formal Definition

Given:

- a set $B = \{i_1, \ldots, i_m\}$ of items,
- a tuple $T = (t_1, \ldots, t_n)$ of transactions over B,
- a real number $\varsigma_{\min}$, $0 < \varsigma_{\min} \leq 1$, the **minimum support**,
- a real number $c_{\min}$, $0 < c_{\min} \leq 1$, the **minimum confidence**.

Desired:

- the set of all **association rules**, that is, the set

$$\mathcal{R} = \{R : X \rightarrow Y \mid \varsigma_T(R) \geq \varsigma_{\min} \wedge c_T(R) \geq c_{\min}\}.$$

General Procedure:

- Find the frequent item sets.
- Construct rules and filter them w.r.t. $\varsigma_{\min}$ and $c_{\min}$.

Generating Association Rules

- Which minimum support has to be used for finding the frequent item sets depends on the definition of the support of a rule:
  - If $\varsigma_T(X \to Y) = \sigma_T(X \cup Y)$,
    then $\sigma_{\min} = \varsigma_{\min}$ or equivalently $s_{\min} = \lceil n\varsigma_{\min} \rceil$.
  - If $\varsigma_T(X \to Y) = \sigma_T(X)$,
    then $\sigma_{\min} = \varsigma_{\min} c_{\min}$ or equivalently $s_{\min} = \lceil n\varsigma_{\min} c_{\min} \rceil$.

- After the frequent item sets have been found, the rule construction then traverses all frequent item sets I and splits them into disjoint subsets X and Y ($X \cap Y = \emptyset$ and $X \cup Y = I$), thus forming rules $X \to Y$.
  - Filtering rules w.r.t. confidence is always necessary.
  - Filtering rules w.r.t. support is only necessary if $\varsigma_T(X \to Y) = \sigma_T(X)$.

Properties of the Confidence

- From $\forall I : \forall J \subseteq I : s_T(I) \leq s_T(J)$ it obviously follows

$$\forall X, Y : \forall a \in X : \quad \frac{s_T(X \cup Y)}{s_T(X)} \geq \frac{s_T(X \cup Y)}{s_T(X - \{a\})}$$

  and therefore

$$\forall X, Y : \forall a \in X : \quad c_T(X \rightarrow Y) \geq c_T(X - \{a\} \rightarrow Y \cup \{a\}).$$

  That is: **Moving an item from the antecedent to the consequent cannot increase the confidence of a rule.**

- As an immediate consequence we have

$$\forall X, Y : \forall a \in X : \quad c_T(X \rightarrow Y) < c_{\min} \;\rightarrow\; c_T(X - \{a\} \rightarrow Y \cup \{a\}) < c_{\min}.$$

  That is: **If a rule fails to meet the minimum confidence, no rules over the same item set and with items moved from antecedent to consequent need to be considered.**

Generating Association Rules

```
function rules ($F$);                                   (∗ — generate association rules ∗)
    $R := \emptyset$;                                   (∗ initialize the set of rules ∗)
    forall $f \in F$ do begin                           (∗ traverse the frequent item sets ∗)
        $m := 1$;                                       (∗ start with rule heads (consequents) ∗)
        $H_m := \bigcup_{i \in f} \{\{i\}\}$;           (∗ that contain only one item ∗)
        repeat                                          (∗ traverse rule heads of increasing size ∗)
            forall $h \in H_m$ do                       (∗ traverse the possible rule heads ∗)
                if $\frac{s_T(f)}{s_T(f-h)} \geq c_{\min}$       (∗ if the confidence is high enough, ∗)
                then $R := R \cup \{[(f - h) \rightarrow h]\}$;   (∗ add rule to the result ∗)
                else $H_m := H_m - \{h\}$;              (∗ otherwise discard the head ∗)
            $H_{m+1} :=$ candidates$(H_m)$;             (∗ create heads with one item more ∗)
            $m := m + 1$;                               (∗ increment the head item counter ∗)
        until $H_m = \emptyset$ or $m \geq |f|$;        (∗ until there are no more rule heads ∗)
    end;                                                (∗ or antecedent would become empty ∗)
    return $R$;                                         (∗ return the rules found ∗)
end; (∗ rules ∗)
```

Generating Association Rules

```
function candidates ($F_k$)                         (* generate candidates with $k+1$ items *)
begin
    $E := \emptyset$;                                   (* initialize the set of candidates *)
    forall $f_1, f_2 \in F_k$                              (* traverse all pairs of frequent item sets *)
    with  $f_1 = \{a_1, \ldots, a_{k-1}, a_k\}$               (* that differ only in one item and *)
    and   $f_2 = \{a_1, \ldots, a_{k-1}, a'_k\}$              (* are in a lexicographic order *)
    and   $a_k < a'_k$ do begin                            (* (the order is arbitrary, but fixed) *)
        $f := f_1 \cup f_2 = \{a_1, \ldots, a_{k-1}, a_k, a'_k\}$;     (* union has $k+1$ items *)
        if $\forall a \in f : \; f - \{a\} \in F_k$                (* only if all subsets are frequent, *)
        then $E := E \cup \{f\}$;                          (* add the new item set to the candidates *)
    end;                                             (* (otherwise it cannot be frequent) *)
    return $E$;                                        (* return the generated candidates *)
end (* candidates *)
```

Frequent Item Sets: Example

transaction database

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{c, b, e\}$
10: $\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7 $\{b\}$: 3 $\{c\}$: 7 $\{d\}$: 6 $\{e\}$: 7	$\{a, c\}$: 4 $\{a, d\}$: 5 $\{a, e\}$: 6 $\{b, c\}$: 3 $\{c, d\}$: 4 $\{c, e\}$: 4 $\{d, e\}$: 4	$\{a, c, d\}$: 3 $\{a, c, e\}$: 3 $\{a, d, e\}$: 4

- The minimum support is $s_{\min} = 3$ or $\sigma_{\min} = 0.3 = 30\%$ in this example.

- There are $2^5 = 32$ possible item sets over $B = \{a, b, c, d, e\}$.

- There are 16 frequent item sets (but only 10 transactions).

Generating Association Rules

Example: $I = \{a, c, e\}$, $X = \{c, e\}$, $Y = \{a\}$.

$$c_T(c, e \to a) = \frac{s_T(\{a, c, e\})}{s_T(\{c, e\})} = \frac{3}{4} = 75\%$$

Minimum confidence: 80%

association rule	support of all items	support of antecedent	confidence
$b \to c$	3 (30%)	3 (30%)	100%
$d \to a$	5 (50%)	6 (60%)	83.3%
$e \to a$	6 (60%)	7 (70%)	85.7%
$a \to e$	6 (60%)	7 (70%)	85.7%
$d, e \to a$	4 (40%)	4 (40%)	100%
$a, d \to e$	4 (40%)	5 (50%)	80%

Support of an Association Rule

The two rule support definitions are not equivalent:

transaction database

1: $\{a, c, e\}$
2: $\{b, d\}$
3: $\{b, c, d\}$
4: $\{a, e\}$
5: $\{a, b, c, d\}$
6: $\{c, e\}$
7: $\{a, b, d\}$
8: $\{a, c, d\}$

two association rules

association rule	support of all items	support of antecedent	confidence
$a \rightarrow c$	3 (37.5%)	5 (62.5%)	60.0%
$b \rightarrow d$	4 (50.0%)	4 (50.0%)	100.0%

Let the minimum confidence be $c_{\min} = 60\%$.

- For $\varsigma_T(R) = \sigma(X \cup Y)$ and $3/8 < \varsigma_{\min} \leq 4/8$ only the rule $b \rightarrow d$ is generated, but not the rule $a \rightarrow c$.

- For $\varsigma_T(R) = \sigma(X)$ there is no value $\varsigma_{\min}$ that generates only the rule $b \rightarrow d$, but not at the same time also the rule $a \rightarrow c$.

Rules with Multiple Items in the Consequent?

- The general definition of association rules $X \to Y$ allows for multiple items in the consequent (i.e. $|Y| \geq 1$).
- However: If $a \to b, c$ is an association rule,
  then $a \to b$ and $a \to c$ are also association rules.

  Because: (regardless of the rule support definition)

  $$\begin{aligned} \varsigma_T(a \to b) &\geq \varsigma_T(a \to b, c), & c_T(a \to b) &\geq c_T(a \to b, c), \\ \varsigma_T(a \to c) &\geq \varsigma_T(a \to b, c), & c_T(a \to c) &\geq c_T(a \to b, c). \end{aligned}$$

- The two simpler rules are often sufficient (e.g. for product suggestions), even though they contain less information.
  - $a \to b, c$ provides information about the *joint* conditional occurence of b and c (condition a).
  - $a \to b$ and $a \to c$ only provide information about the *individual* conditional occurrences of b and c (condition a).

  In most applications this additional information does not yield any additional benefit.

Rules with Multiple Items in the Consequent?

- If the rule support is defined as $\varsigma_T(X \to Y) = \sigma_T(X \cup Y)$, we can go one step further in ruling out multi-item consequents.

- If $a \to b, c$ is an association rule, then $a, b \to c$ and $a, c \to b$ are also association rules.

  Because: (confidence relationships always hold)

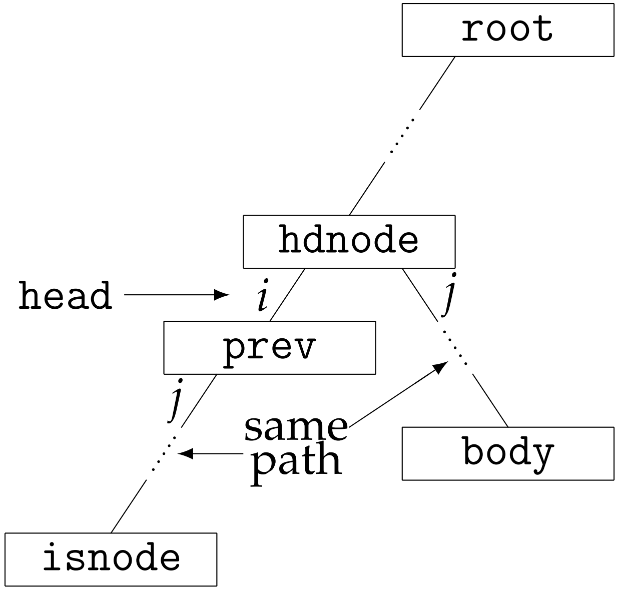
$$\begin{aligned} \varsigma_T(a, b \to c) &\geq \varsigma_T(a \to b, c), \qquad & c_T(a, b \to c) &\geq c_T(a \to b, c), \\ \varsigma_T(a, c \to b) &\geq \varsigma_T(a \to b, c), \qquad & c_T(a, c \to b) &\geq c_T(a \to b, c). \end{aligned}$$

- Together with $a \to b$ and $a \to c$, the rules $a, b \to c$ and $a, c \to b$ contain effectively the same information as the rule $a \to b, c$, although in a different form.

- For example, product suggestions can be made by first applying $a \to b$, hypothetically assuming that b is actually added to the shopping cart, and then applying $a, b \to c$ to suggest both b and c.

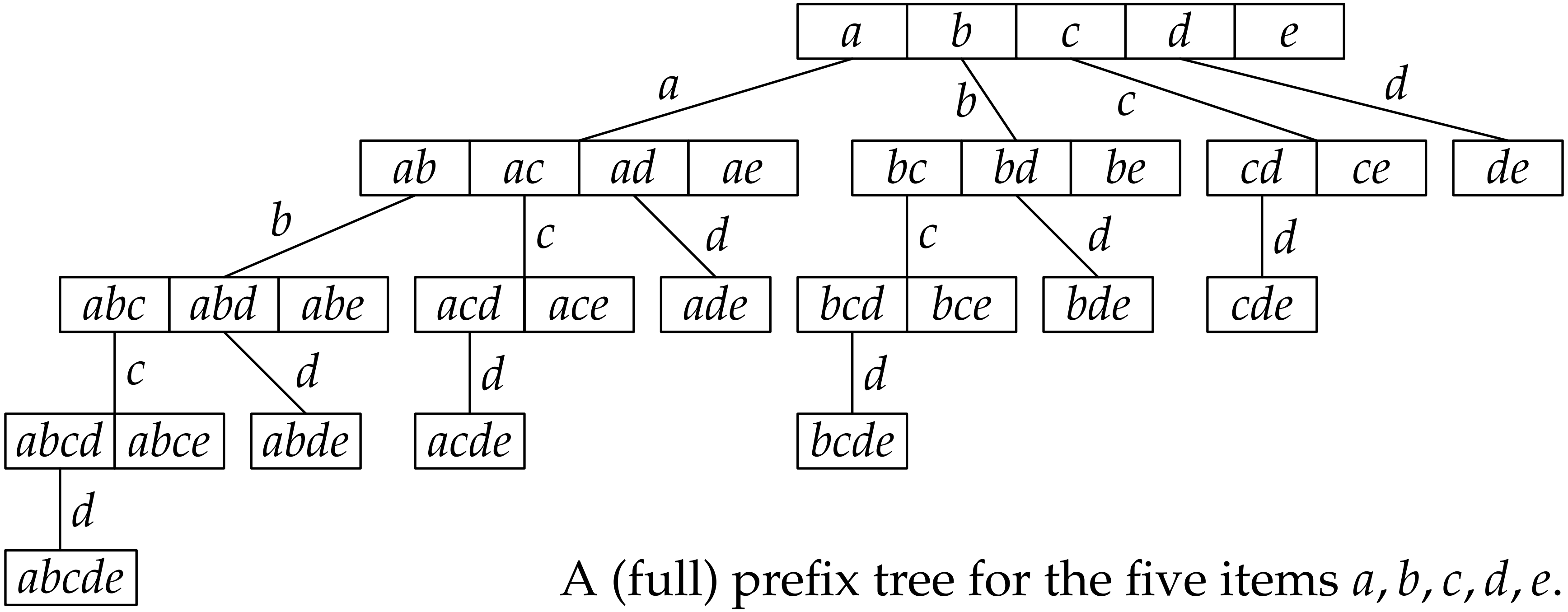
Rule Extraction from Prefix Tree

- Restriction to rules with one item in the head/consequent.
- Exploit the prefix tree to find the support of the body/antecedent.
- Traverse the item set tree breadth-first or depth-first.
- For each node traverse the path to the root and generate and test one rule per node.



- First rule: Get the support of the body/antecedent from the parent node.
- Next rules: Discard the head/consequent item from the downward path and follow the remaining path from the current node.

Reminder: Prefix Tree



A (full) prefix tree for the five items a, b, c, d, e.

- Based on a global order of the items (which can be arbitrary).
- The item sets counted in a node consist of
  - ○ all items labeling the edges to the node (common prefix) and
  - ○ one item following the last edge label in the item order.

Additional Rule Filtering: Simple Measures

- General idea: Compare $\hat{P}_T(Y \mid X) = c_T(X \to Y)$
  and $\hat{P}_T(Y) = c_T(\emptyset \to Y) = \sigma_T(Y)$.

- (Absolute) confidence difference to prior:

$$d_T(R) = |c_T(X \to Y) - \sigma_T(Y)|$$

- Lift value:

$$l_T(R) = \frac{c_T(X \to Y)}{\sigma_T(Y)}$$

- (Absolute) difference of lift value to 1:

$$q_T(R) = \left| \frac{c_T(X \to Y)}{\sigma_T(Y)} - 1 \right|$$

- (Absolute) difference of lift quotient to 1:

$$r_T(R) = \left| 1 - \min\left\{ \frac{c_T(X \to Y)}{\sigma_T(Y)}, \frac{\sigma_T(Y)}{c_T(X \to Y)} \right\} \right|$$

Additional Rule Filtering: More Sophisticated Measures

- Consider the 2×2 contingency table or the estimated probability table:

	$X \not\subseteq t$	$X \subseteq t$	
$Y \not\subseteq t$	n_{00}	n_{01}	$n_{0.}$
$Y \subseteq t$	n_{10}	n_{11}	$n_{1.}$
	$n_{.0}$	$n_{.1}$	$n_{..}$

	$X \not\subseteq t$	$X \subseteq t$	
$Y \not\subseteq t$	p_{00}	p_{01}	$p_{0.}$
$Y \subseteq t$	p_{10}	p_{11}	$p_{1.}$
	$p_{.0}$	$p_{.1}$	1

- $n_{..}$ is the total number of transactions.
  $n_{.1}$ is the number of transactions to which the rule is applicable.
  n_{11} is the number of transactions for which the rule is correct.

  It is $\quad p_{ij} = \frac{n_{ij}}{n_{..}}, \quad p_{i.} = \frac{n_{i.}}{n_{..}}, \quad p_{.j} = \frac{n_{.j}}{n_{..}} \quad$ for $i, j = 1, 2$.

- General idea: Use measures for the strength of dependence of X and Y.

- There is a large number of such measures of dependence originating from statistics, decision tree induction etc.

An Information-theoretic Evaluation Measure

Information Gain (Kullback and Leibler 1951, Quinlan 1986)

Based on Shannon Entropy $H = -\sum_{i=1}^{n} p_i \log_2 p_i$ (Shannon 1948)

$$\begin{aligned} I_{\text{gain}}(X,Y) &= \overbrace{H(Y)} - \overbrace{H(Y|X)} \\ &= \underbrace{-\sum_{i=1}^{k_Y} p_{i.} \log_2 p_{i.}} - \underbrace{\sum_{j=1}^{k_X} p_{.j} \left(-\sum_{i=1}^{k_Y} p_{i|j} \log_2 p_{i|j} \right)} \end{aligned}$$

$H(Y)$ Entropy of the distribution of Y

$H(Y|X)$ *Expected entropy* of the distribution of Y if the value of the X becomes known

$H(Y) - H(Y|X)$ Expected entropy reduction or *information gain*

Interpretation of Shannon Entropy

- Let $S = \{s_1, \ldots, s_n\}$ be a finite set of alternatives having positive probabilities $P(s_i)$, $i = 1, \ldots, n$, satisfying $\sum_{i=1}^{n} P(s_i) = 1$.

- **Shannon Entropy:**

$$H(S) = -\sum_{i=1}^{n} P(s_i) \log_2 P(s_i)$$

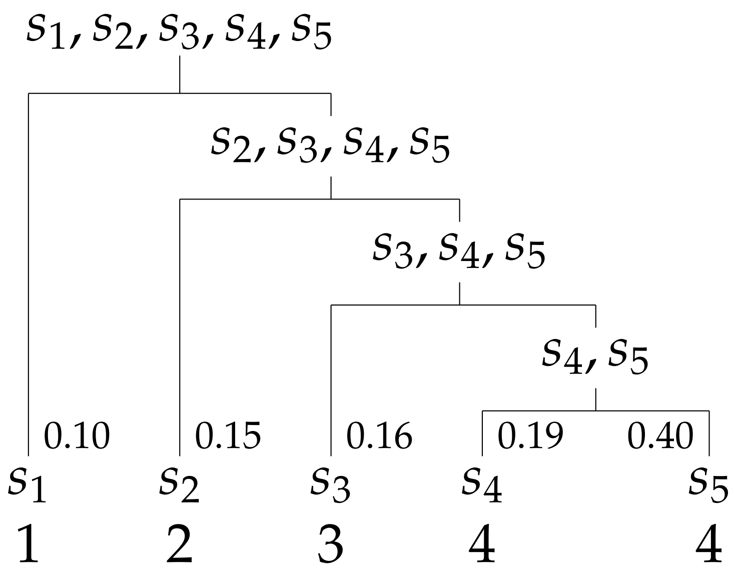
- Intuitively: **Expected number of yes/no questions that have to be asked in order to determine the obtaining alternative.**
  - ○ Suppose there is an oracle, which knows the obtaining alternative, but responds only if the question can be answered with "yes" or "no".
  - ○ A better question scheme than asking for one alternative after the other can easily be found: Divide the set into two subsets of about equal size.
  - ○ Ask for containment in an arbitrarily chosen subset.
  - ○ Apply this scheme recursively → number of questions bounded by $\lceil \log_2 n \rceil$.

Question/Coding Schemes

$$P(s_1) = 0.10, \quad P(s_2) = 0.15, \quad P(s_3) = 0.16, \quad P(s_4) = 0.19, \quad P(s_5) = 0.40$$

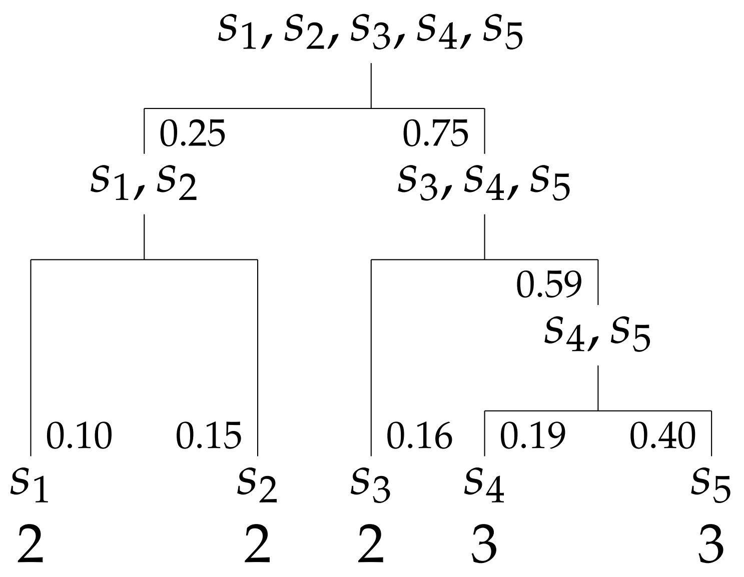
Shannon entropy: $-\sum_i P(s_i) \log_2 P(s_i) = 2.15$ bit/symbol

Linear Traversal



Code length: 3.24 bit/symbol
Code efficiency: 0.664

Equal Size Subsets



Code length: 2.59 bit/symbol
Code efficiency: 0.830

Question/Coding Schemes

- Splitting into subsets of about equal size can lead to a bad arrangement of the alternatives into subsets → high expected number of questions.

- Good question schemes take the probability of the alternatives into account.

- **Shannon-Fano Coding** (1948)
  - Build the question/coding scheme top-down.
  - Sort the alternatives w.r.t. their probabilities.
  - Split the set so that the subsets have about equal *probability* (splits must respect the probability order of the alternatives).

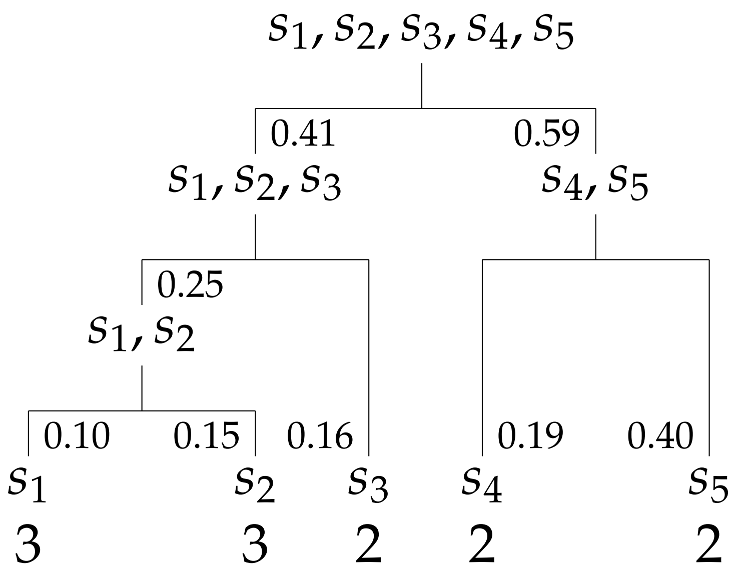
- **Huffman Coding** (1952)
  - Build the question/coding scheme bottom-up.
  - Start with one element sets.
  - Always combine those two sets that have the smallest probabilities.

Question/Coding Schemes

$$P(s_1) = 0.10, \quad P(s_2) = 0.15, \quad P(s_3) = 0.16, \quad P(s_4) = 0.19, \quad P(s_5) = 0.40$$

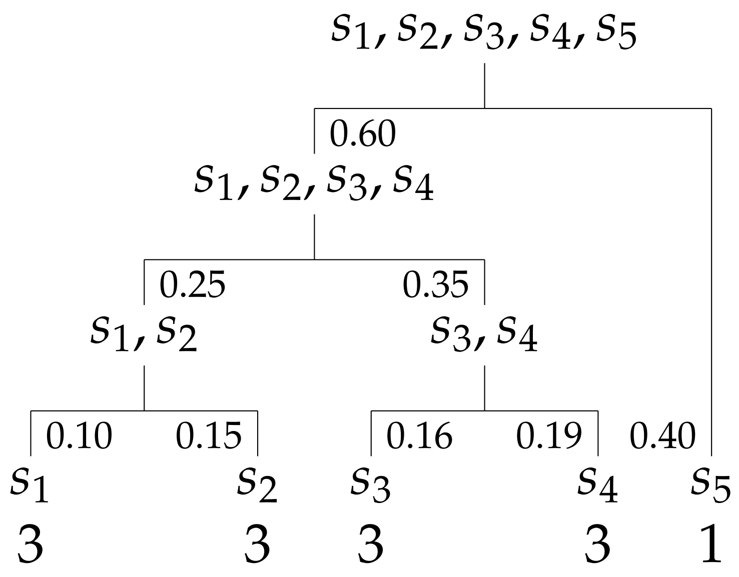
Shannon entropy: $\quad -\sum_i P(s_i) \log_2 P(s_i) = 2.15$ bit/symbol

Shannon–Fano Coding (1948)



Code length: 2.25 bit/symbol
Code efficiency: 0.955

Huffman Coding (1952)



Code length: 2.20 bit/symbol
Code efficiency: 0.977

Question/Coding Schemes

- It can be shown that Huffman coding is optimal
  if we have to determine the obtaining alternative in a single instance.
  (No question/coding scheme has a smaller expected number of questions.)

- Only if the obtaining alternative has to be determined in a sequence
  of (independent) situations, this scheme can be improved upon.

- Idea: Process the sequence not instance by instance,
  but combine two, three or more consecutive instances and
  ask directly for the obtaining combination of alternatives.

- Although this enlarges the question/coding scheme, the expected number
  of questions per identification is reduced (because each interrogation
  identifies the obtaining alternative for several situations).

- However, the expected number of questions per identification
  of an obtaining alternative cannot be made arbitrarily small.
  Shannon showed that there is a lower bound, namely the Shannon entropy.

Interpretation of Shannon Entropy

$$P(s_1) = \tfrac{1}{2}, \quad P(s_2) = \tfrac{1}{4}, \quad P(s_3) = \tfrac{1}{8}, \quad P(s_4) = \tfrac{1}{16}, \quad P(s_5) = \tfrac{1}{16}$$

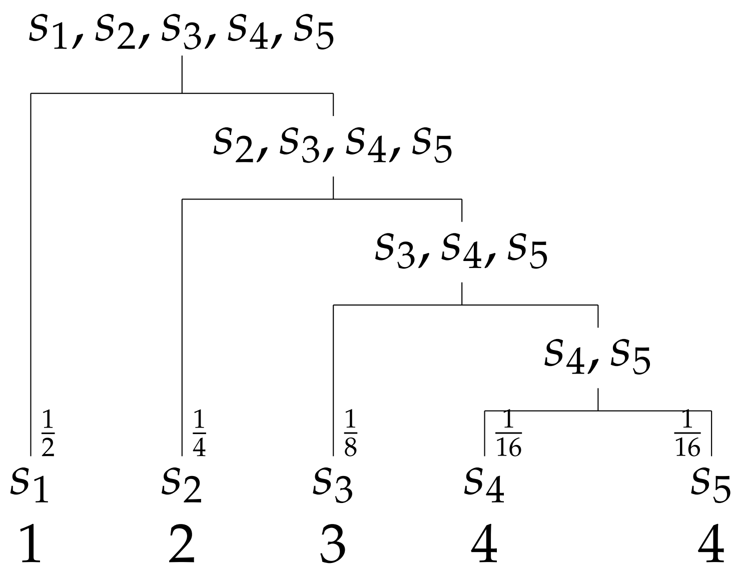
Shannon entropy: $\quad -\sum_i P(s_i) \log_2 P(s_i) = 1.875$ bit/symbol

If the probability distribution allows for a perfect Huffman code (code efficiency 1), the Shannon entropy can easily be interpreted as follows:

$$-\sum_i P(s_i) \log_2 P(s_i) = \sum_i \underbrace{P(s_i)}_{\substack{\text{occurrence}\\ \text{probability}}} \cdot \underbrace{\log_2 \frac{1}{P(s_i)}}_{\substack{\text{path length}\\ \text{in tree}}}.$$

In other words, it is the expected number of needed yes/no questions.

Perfect Question Scheme



Code length: 1.875 bit/symbol
Code efficiency: 1

A Statistical Evaluation Measure

χ^2 Measure

- Compares the actual joint distribution with a **hypothetical independent distribution**.

- Uses absolute comparison.

- Can be interpreted as a difference measure.

$$\chi^2(X,Y) = \sum_{i=1}^{k_X} \sum_{j=1}^{k_Y} n_{..} \frac{(p_{i.}p_{.j} - p_{ij})^2}{p_{i.}p_{.j}}$$

- Side remark: Information gain can also be interpreted as a difference measure.

$$I_{\text{gain}}(X,Y) = \sum_{j=1}^{k_X} \sum_{i=1}^{k_Y} p_{ij} \log_2 \frac{p_{ij}}{p_{i.}p_{.j}}$$

A Statistical Evaluation Measure

χ^2 Measure

- Compares the actual joint distribution with a **hypothetical independent distribution**.

- Uses absolute comparison.

- Can be interpreted as a difference measure.

$$\chi^2(X,Y) = \sum_{i=1}^{k_X} \sum_{j=1}^{k_Y} n_{..} \frac{(p_{i.}p_{.j} - p_{ij})^2}{p_{i.}p_{.j}}$$

- For $k_X = k_Y = 2$ (as for rule evaluation) the χ^2 measure simplifies to

$$\chi^2(X,Y) \;=\; n_{..} \frac{(p_{1.}\;p_{.1} - p_{11})^2}{p_{1.}(1 - p_{1.})p_{.1}(1 - p_{.1})} \;=\; n_{..} \frac{(n_{1.}n_{.1} - n_{..}n_{11})^2}{n_{1.}(n_{..} - n_{1.})n_{.1}(n_{..} - n_{.1})}.$$

Examples from the Census Data

All rules are stated as

```
consequent <- antecedent (support%, confidence%, lift)
```

where the support of a rule is the support of the antecedent.

Trivial/Obvious Rules

```
edu_num=13 <- education=Bachelors  (16.4, 100.0, 6.09)
sex=Male <- relationship=Husband  (40.4, 99.99, 1.50)
sex=Female <- relationship=Wife  (4.8, 99.9, 3.01)
```

Interesting Comparisons

```
marital=Never-married <- age=young sex=Female  (12.3, 80.8, 2.45)
marital=Never-married <- age=young sex=Male  (17.4, 69.9, 2.12)

salary>50K <- occupation=Exec-managerial sex=Male (8.9, 57.3, 2.40)
salary>50K <- occupation=Exec-managerial  (12.5, 47.8, 2.00)

salary>50K <- education=Masters  (5.4, 54.9, 2.29)
hours=overtime <- education=Masters  (5.4, 41.0, 1.58)
```

Examples from the Census Data

```
salary>50K <- education=Masters  (5.4, 54.9, 2.29)
salary>50K <- occupation=Exec-managerial  (12.5, 47.8, 2.00)
salary>50K <- relationship=Wife  (4.8, 46.9, 1.96)
salary>50K <- occupation=Prof-specialty  (12.6, 45.1, 1.89)
salary>50K <- relationship=Husband  (40.4, 44.9, 1.88)
salary>50K <- marital=Married-civ-spouse  (45.8, 44.6, 1.86)
salary>50K <- education=Bachelors  (16.4, 41.3, 1.73)
salary>50K <- hours=overtime  (26.0, 40.6, 1.70)

salary>50K <- occupation=Exec-managerial hours=overtime
              (5.5, 60.1, 2.51)
salary>50K <- occupation=Prof-specialty hours=overtime
              (4.4, 57.3, 2.39)
salary>50K <- education=Bachelors hours=overtime
              (6.0, 54.8, 2.29)
```

Examples from the Census Data

```
salary>50K <- occupation=Prof-specialty
              marital=Married-civ-spouse  (6.5, 70.8, 2.96)

salary>50K <- occupation=Exec-managerial
              marital=Married-civ-spouse  (7.4, 68.1, 2.85)

salary>50K <- education=Bachelors
              marital=Married-civ-spouse  (8.5, 67.2, 2.81)

salary>50K <- hours=overtime
              marital=Married-civ-spouse  (15.6, 56.4, 2.36)

marital=Married-civ-spouse <- salary>50K  (23.9, 85.4, 1.86)
```

Examples from the Census Data

```
hours=half-time <- occupation=Other-service age=young
                   (4.4, 37.2, 3.08)

hours=overtime <- salary>50K  (23.9, 44.0, 1.70)
hours=overtime <- occupation=Exec-managerial  (12.5, 43.8, 1.69)
hours=overtime <- occupation=Exec-managerial salary>50K
                  (6.0, 55.1, 2.12)
hours=overtime <- education=Masters  (5.4, 40.9, 1.58)

education=Bachelors <- occupation=Prof-specialty
                       (12.6, 36.2, 2.20)
education=Bachelors <- occupation=Exec-managerial
                       (12.5, 33.3, 2.03)
education=HS-grad   <- occupation=Transport-moving
                       (4.8, 51.9, 1.61)
education=HS-grad   <- occupation=Machine-op-inspct
                       (6.2, 50.7, 1.6)
```

Examples from the Census Data

```
occupation=Prof-specialty <- education=Masters
                             (5.4, 49.0, 3.88)
occupation=Prof-specialty <- education=Bachelors sex=Female
                             (5.1, 34.7, 2.74)
occupation=Adm-clerical   <- education=Some-college sex=Female
                             (8.6, 31.1, 2.71)

sex=Female <- occupation=Adm-clerical  (11.5, 67.2, 2.03)
sex=Female <- occupation=Other-service  (10.1, 54.8, 1.65)
sex=Female <- hours=half-time  (12.1, 53.7, 1.62)

age=young <- hours=half-time  (12.1, 53.3, 1.79)
age=young <- occupation=Handlers-cleaners  (4.2, 50.6, 1.70)
age=senior <- workclass=Self-emp-not-inc  (7.9, 31.1, 1.57)
```

Summary Association Rules

- **Association Rule Induction is a Two Step Process**
  - Find the frequent item sets (minimum support).
  - Form the relevant association rules (minimum confidence).

- **Generating the Association Rules**
  - Form all possible association rules from the frequent item sets.
  - Filter "interesting" association rules
    based on minimum support and minimum confidence.

- **Filtering the Association Rules**
  - Compare rule confidence and consequent support.
  - Information gain, χ^2 measure
  - In principle: other measures used for decision tree induction.

Mining More Complex Patterns

Mining More Complex Patterns

- The search scheme in Frequent Graph/Tree/Sequence mining is the same, namely the general scheme of searching with a canonical form.

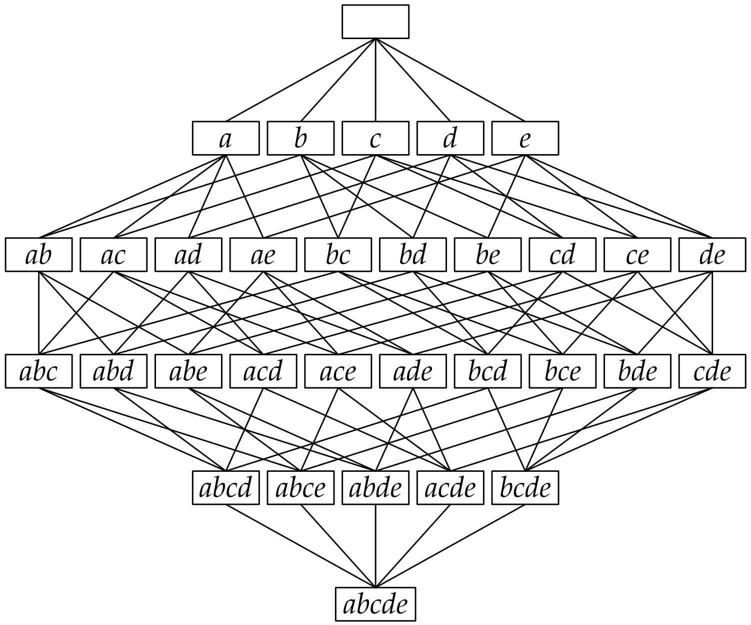
- **Frequent (Sub)Graph Mining** comprises the other areas:
  - Trees are special graphs, namely graphs that are singly connected.
  - Sequences can be seen as special trees, namely chains (only one or two branches — depending on the choice of the root).

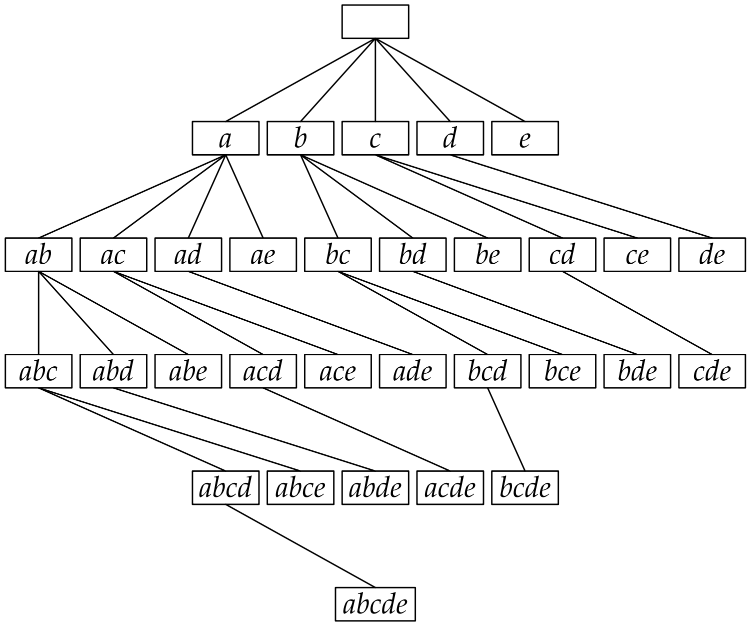
- **Frequent Sequence Mining** and **Frequent Tree Mining** can exploit:
  - Specialized canonical forms that allow for more efficient checks.
  - Special data structures to represent the database to mine, so that support counting becomes more efficient.

- We will treat *Frequent (Sub)Graph Mining* first and will discuss optimizations for the other areas later.

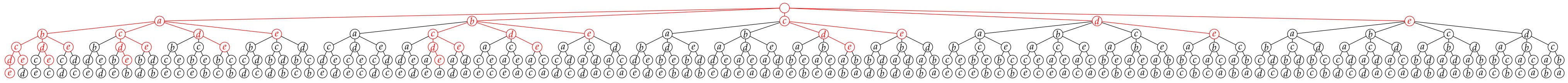
Search Space Comparison

Search space for sets: (5 items)





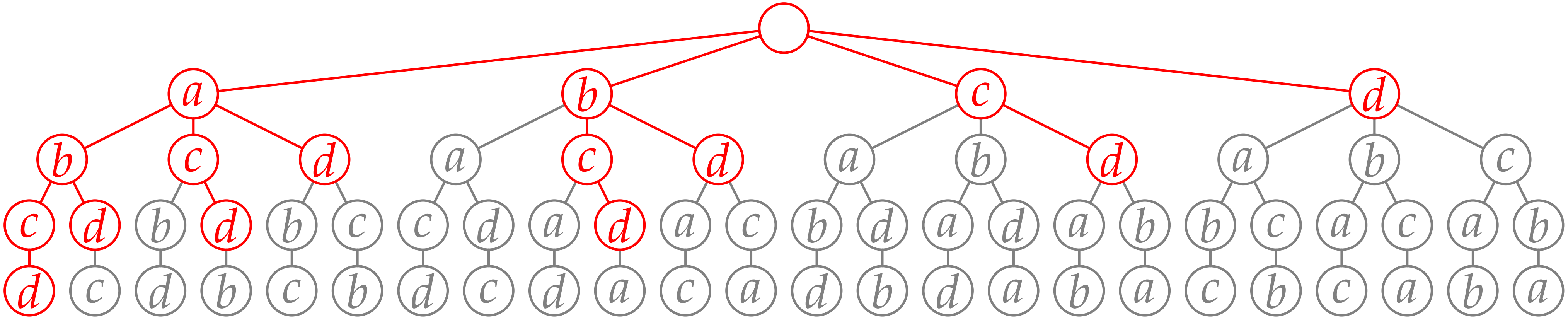
Search space for sequences: (5 items, no repetitions)



- Red part corresponds to search space for sets (top right).

Search Space Comparison

Search space for sequences: (4 items, no repetitions)


a
b
c
d
b c d a c d a b d a b c
c d b d b c c d a d a c b d a d a b b c a c a b
d c d b c b d c d a c a d b d a b a c b c a b a

- Red part corresponds to search space for sets.
- The search space for (sub)sequences is considerably larger than the one for sets.
- However: support of (sub)sequences reduces faster with increasing length.
  - Out of *k* items only one set can be formed,
    but *k*! sequences (every order yields a different sequences).
  - All *k*! sequences cover the set (tendency towards higher support).
  - To cover a specific sequence, a specific order is required.
    (tendency towards lower support).

Motivation:

Molecular Fragment Mining

Molecular Fragment Mining

- **Motivation: Accelerating Drug Development**
  - Phases of drug development: pre-clinical and clinical
  - Data gathering by high-throughput screening:
    building molecular databases with activity information
  - Acceleration potential by intelligent data analysis:
    (quantitative) structure-activity relationship discovery
- **Mining Molecular Databases**
  - Example data: NCI DTP HIV Antiviral Screen data set
  - Description languages for molecules:
    SMILES, SLN, SDfile/Ctab etc.
  - Finding common molecular substructures
  - Finding discriminative molecular substructures

Accelerating Drug Development

- Developing a new drug can take **10 to 12 years**
  (from the choice of the target to the introduction into the market).

- In recent years the **duration** of the drug development processes **increased** continuously; at the same the **number** of substances under development **has gone down** drastically.

- Due to high investments pharmaceutical companies must secure their market position and competitiveness by only a **few, highly successful drugs**.

- As a consequence the chances for the development of drugs for target groups

  - with **rare diseases** or
  - with **special diseases in developing countries**

  are considerably reduced.

- A significant **reduction of the development time** could mitigate this trend or even reverse it.

(Source: Bundesministerium für Bildung und Forschung, Germany)

Phases of Drug Development

- **Discovery and Optimization of Candidate Substances**
  - High-Throughput Screening
  - Lead Discovery and Lead Optimization
- **Pre-clinical Test Series** (tests with animals, ca. 3 years)
  - Fundamental test w.r.t. effectiveness and side effects
- **Clinical Test Series** (tests with humans, ca. 4–6 years)
  - Phase 1: ca. 30–80 healthy humans
    Check for side effects
  - Phase 2: ca. 100–300 humans exhibiting the symptoms of the target disease
    Check for effectiveness
  - Phase 3: up to 3000 healthy and ill humans at least 3 years
    Detailed check of effectiveness and side effects
- **Official Acceptance as a Drug**

Drug Development: Acceleration Potential

- The length of the pre-clinical and clinical tests series can hardly be reduced, since they serve the purpose to ensure the safety of the patients.
- Therefore approaches to speed up the development process usually target the **pre-clinical phase** before the animal tests.
- In particular, it is tried to improve the search for new drug candidates (*lead discovery*) and their optimization (*lead optimization*).

Here Frequent Pattern Mining can help.

One possible approach:

- With high-throughput screening a very large number of substances is tested automatically and their activity is determined.
- The resulting molecular databases are analyzed by trying to find **common substructures** of active substances.

High-Throughput Screening

On so-called **micro-plates** proteins/cells are automatically combined with a large variety of chemical compounds.

pictures not available in online version

High-Throughput Screening

The filled micro-plates are then evaluated in **spectrometers**
(w.r.t. absorption, fluorescence, luminescence, polarization etc).

pictures not available in online version

High-Throughput Screening

After the measurement the substances are classified as **active** or **inactive**.

picture not available in online version

By analyzing the results one tries to understand the dependencies between molecular structure and activity (w.r.t. the chosen target).

QSAR —
Quantitative Structure-Activity Relationship Modeling

In this area a large number of data mining algorithms are used:

- frequent pattern mining
- feature selection methods
- decision trees
- neural networks etc.

Example: NCI DTP HIV Antiviral Screen

- Among other data sets, the National Cancer Institute (NCI) has made the **DTP HIV Antiviral Screen Data Set** publicly available.
- A large number of chemical compounds where tested whether they protect human CEM cells against an HIV-1 infection.
- Substances that provided 50% protection were retested.
- Substances that reproducibly provided 100% protection are listed as **“confirmed active” (CA)**.
- Substances that reproducibly provided at least 50% protection are listed as **“moderately active” (CM)**.
- All other substances are listed as **“confirmed inactive” (CI)**.
- 325 **CA**, 877 **CM**, 35 969 **CI** (total: 37 171 substances)

Form of the Input Data

Excerpt from the NCI DTP HIV Antiviral Screen data set (SMILES format):

```
737,  0,CN(C)C1=[S+][Zn]2(S1)SC(=[S+]2)N(C)C
2018, 0,N#CC(=CC1=CC=CC=C1)C2=CC=CC=C2
19110,0,OC1=C2N=C(NC3=CC=CC=C3)SC2=NC=N1
20625,2,NC(=N)NC1=C(SSC2=C(NC(N)=N)C=CC=C2)C=CC=C1.OS(O)(=O)=O
22318,0,CCCCN(CCCC)C1=[S+][Cu]2(S1)SC(=[S+]2)N(CCCC)CCCC
24479,0,C[N+](C)(C)C1=CC2=C(NC3=CC=CC=C3S2)N=N1
50848,2,CC1=C2C=CC=CC2=N[C-](CSC3=CC=CC=C3)[N+]1=O
51342,0,OC1=C2C=NC(=NC2=C(O)N=N1)NC3=CC=C(Cl)C=C3
55721,0,NC1=NC(=C(N=O)C(=N1)O)NC2=CC(=C(Cl)C=C2)Cl
55917,0,O=C(N1CCCC[CH]1C2=CC=CN=C2)C3=CC=CC=C3
64054,2,CC1=C(SC[C-]2N=C3C=CC=CC3=C(C)[N+]2=O)C=CC=C1
64055,1,CC1=CC=CC(=C1)SC[C-]2N=C3C=CC=CC3=C(C)[N+]2=O
64057,2,CC1=C2C=CC=CC2=N[C-](CSC3=NC4=CC=CC=C4S3)[N+]1=O
66151,0,[O-][N+](=O)C1=CC2=C(C=NN=C2C=C1)N3CC3
...
```

identification number, activity (2: CA, 1: CM, 0: CI), molecule description in SMILES notation

Input Format: SMILES Notation and SLN

SMILES Notation: (e.g. Daylight, Inc.)

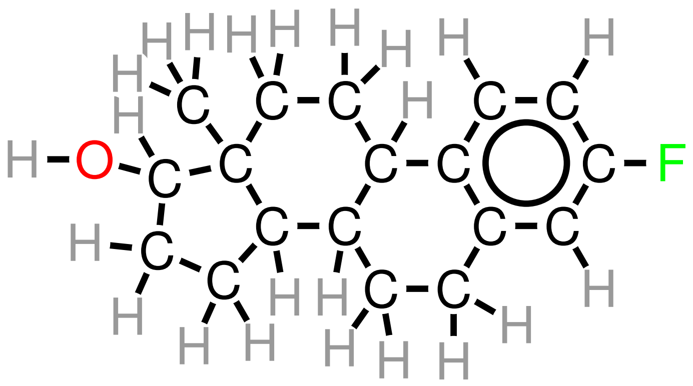
```
c1:c:c(-F):c:c2:c:1-C1-C(-C-C-2)-C2-C(-C)(-C-C-1)-C(-O)-C-C-2
```

SLN (SYBYL Line Notation): (Tripos, Inc.)

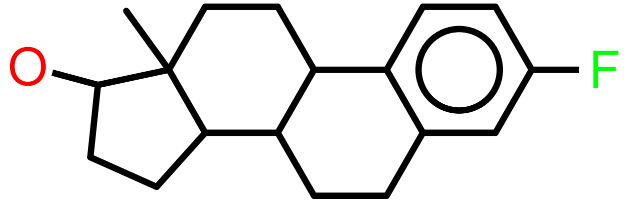
```
C[1]H:CH:C(F):CH:C[8]:C:@1-C[10]H-CH(-CH2-CH2-@8)-C[20]H-C(-CH3)
(-CH2-CH2-@10)-CH(-CH2-CH2-@20)-OH
```

Represented Molecule:

Full Representation



Simplified Representation



Input Format: Grammar for SMILES and SLN

General grammar for (linear) molecule descriptions (SMILES and SLN):

```
Molecule  ::=  Atom Branch
Branch    ::=  ε
            |  Bond Atom Branch
            |  Bond Label Branch
            |  ( Branch ) Branch
Atom      ::=  Element LabelDef
LabelDef  ::=  ε
            |  Label LabelDef
```

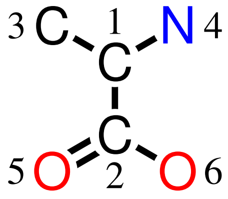
black: non-terminal symbols
blue : terminal symbols

The definitions of the non-terminals "Element", "Bond", and "Label" depend on the chosen description language. For SMILES it is:

```
Element  ::=  B | C | N | O | F | [H] | [He] | [Li] | [Be] | ...
Bond     ::=  ε | - | = | # | : | .
Label    ::=  Digit | % Digit Digit
Digit    ::=  0 | 1 | ... | 9
```

Input Format: SDfile/Ctab

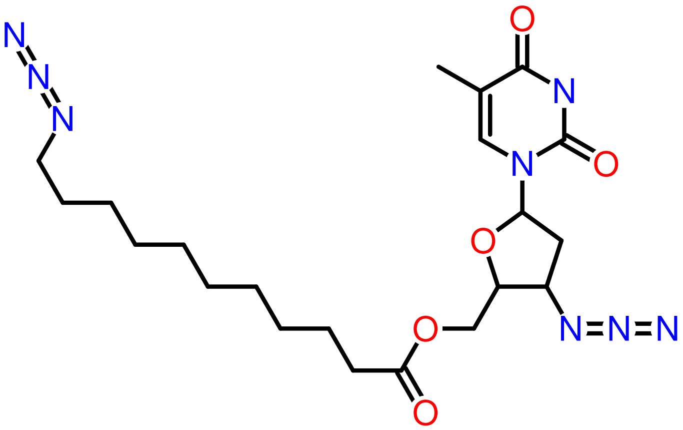
```
L-Alanine (13C)
user initials, program, date/time etc.
comment
  6  5  0  0  1  0              3 V2000
   -0.6622    0.5342    0.0000 C   0  0  2  0  0  0
    0.6622   -0.3000    0.0000 C   0  0  0  0  0  0
   -0.7207    2.0817    0.0000 C   1  0  0  0  0  0
   -1.8622   -0.3695    0.0000 N   0  3  0  0  0  0
    0.6220   -1.8037    0.0000 O   0  0  0  0  0  0
    1.9464    0.4244    0.0000 O   0  5  0  0  0  0
  1  2  1  0  0  0
  1  3  1  1  0  0
  1  4  1  0  0  0
  2  5  2  0  0  0
  2  6  1  0  0  0
M  END
> <value>
0.2

$$$$
```

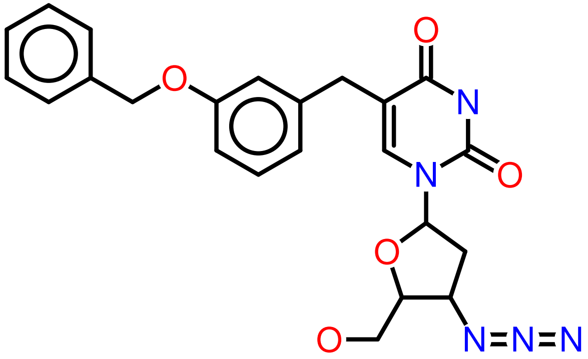


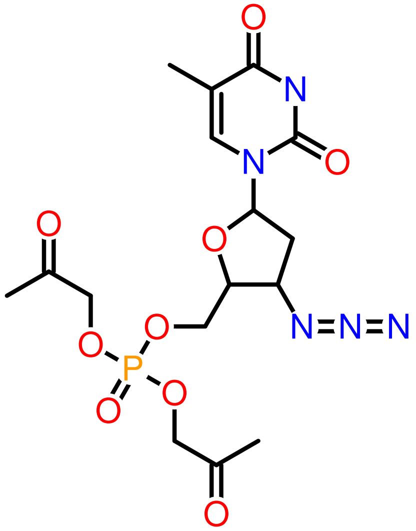
SDfile: Structure-data file
Ctab: Connection table (lines 4–16)

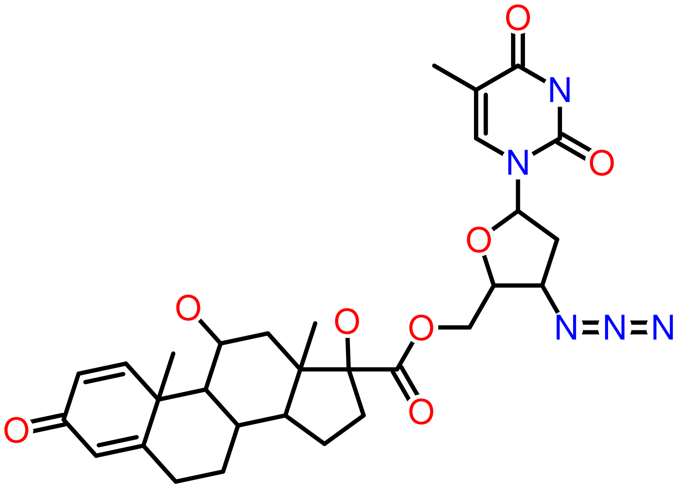
Finding Common Molecular Substructures

Some Molecules from the NCI HIV Database

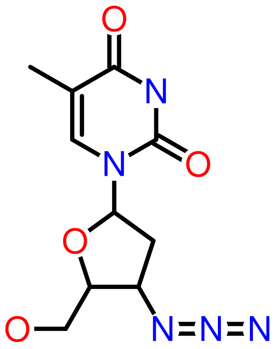








Common Fragment



Finding Molecular Substructures

- **Common Molecular Substructures**
  - Analyze only the active molecules.
  - Find molecular fragments that appear frequently in the molecules.

- **Discriminative Molecular Substructures**
  - Analyze the active and the inactive molecules.
  - Find molecular fragments that appear frequently in the active molecules and only rarely in the inactive molecules.

- **Rationale in both cases**:
  - The found fragments can give hints which structural properties are responsible for the activity of a molecule.
  - This can help to identify drug candidates (so-called *pharmacophores*) and to guide future screening efforts.

Frequent (Sub)Graph Mining

Frequent (Sub)Graph Mining: General Approach

- Finding frequent item sets means to find **sets of items that are contained in many transactions**.

- Finding frequent substructures means to find **graph fragments that are contained in many graphs** in a given database of attributed graphs (user specifies minimum support).

- Graph structure of vertices and edges has to be taken into account.

  ⇒ Search partially ordered set of graph structures instead of subsets.

  Main problem: **How can we avoid redundant search?**

- Usually the search is restricted to **connected substructures**.
  - Connected substructures suffice for most applications.
  - This restriction considerably narrows the search space.

Frequent (Sub)Graph Mining: Basic Notions

- Let $A = \{a_1, \ldots, a_m\}$ be a set of **attributes** or **labels**.

- A **labeled** or **attributed graph** is a triplet $G = (V, E, \ell)$, where
  - V is the set of vertices,
  - $E \subseteq V \times V - \{(v, v) \mid v \in V\}$ is the set of edges, and
  - $\ell : V \cup E \to A$ assigns labels from the set A to vertices and edges.

  Note that G is *undirected* and *simple* and contains *no loops*.
  However, graphs without these restrictions could be handled as well.

  Note also that several vertices and edges may have the same attribute/label.

Example: **molecule representation**

- Atom attributes: atom type (chemical element), charge, aromatic ring flag

- Bond attributes: bond type (single, double, triple, aromatic)

Frequent (Sub)Graph Mining: Basic Notions

Note that for labeled graphs the same notions can be used as for normal graphs.
Without formal definition, we will use, for example:

- A vertex v is **incident** to an edge e, and the edge is **incident** to the vertex v, iff $e = (v, v')$ or $e = (v', v)$.
- Two different vertices are **adjacent** or **connected** if they are incident to the same edge.
- A **path** is a sequence of edges connecting two vertices. It is usually understood that no edge (and no vertex) occurs twice.
- A graph is called **connected** if there exists a path between any two vertices.
- A **subgraph** consists of a subset of the vertices and a subset of the edges. If S is a (proper) subgraph of G we write $S \subseteq G$ or $S \subset G$, respectively.
- A **connected component** of a graph is a subgraph that is connected and maximal in the sense that any larger subgraph containing it is not connected.

Frequent (Sub)Graph Mining: Basic Notions

Note that for labeled graphs the same notions can be used as for normal graphs. Without formal definition, we will use, for example:

- A vertex of a graph is called **isolated** if it is not incident to any edge.
- A vertex of a graph is called a **leaf** if it is incident to exactly one edge.
- An edge of a graph is called a **bridge** if removing it increases the number of connected components of the graph.

  More intuitively: a bridge is the only connection between two vertices, that is, there is no other path on which one can reach the one from the other.

- An edge of a graph is called a **leaf bridge** if it is a bridge and incident to at least one leaf.

  In other words: an edge is a leaf bridge if removing it creates an isolated vertex.

- All other bridges are called **proper bridges**.

Frequent (Sub)Graph Mining: Basic Notions

- Let $G = (V_G, E_G, \ell_G)$ and $S = (V_S, E_S, \ell_S)$ be two labeled graphs.

  A **subgraph isomorphism** of S to G or an **occurrence** of S in G is an injective function $f : V_S \to V_G$ with

  - $\forall v \in V_S : \quad \ell_S(v) = \ell_G(f(v)) \quad$ and
  - $\forall (u, v) \in E_S : \quad (f(u), f(v)) \in E_G \quad \wedge \quad \ell_S((u, v)) = \ell_G((f(u), f(v)))$.

  That is, the mapping f preserves the connection structure and the labels.

  If such a mapping f exists, we write $S \sqsubseteq G$ (note the difference to $S \subseteq G$).

- There may be several ways to map a labeled graph S to a labeled graph G so that the connection structure and the vertex and edge labels are preserved.

  It may even be that the graph S can be mapped in several different ways to the same subgraph of G. This is the case if there exists a subgraph isomorphism of S to itself (a so-called *graph automorphism*) that is not the identity.

Frequent (Sub)Graph Mining: Basic Notions

Let S and G be two labeled graphs.

- S and G are called **isomorphic**, written $S \equiv G$, iff $S \sqsubseteq G$ and $G \sqsubseteq S$.

  In this case a function f mapping S to G is called a **graph isomorphism**.

  A function f mapping S to itself is called a **graph automorphism**.

- S is **properly contained in** G, written $S \sqsubset G$, iff $S \sqsubseteq G$ and $S \not\equiv G$.

- If $S \sqsubseteq G$ or $S \sqsubset G$, then there exists a (proper) subgraph G' of G, (that is, $G' \subseteq G$ or $G' \subset G$, respectively), such that S and G' are isomorphic.

  This explains the term "subgraph isomorphism".

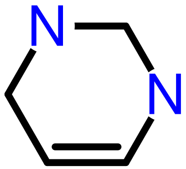
- The **set of all connected subgraphs** of G is denoted by $\mathcal{C}(G)$.

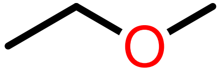
  It is obvious that for all $S \in \mathcal{C}(G) : S \sqsubseteq G$.

  However, there are (unconnected) graphs S with $S \sqsubseteq G$ that are not in $\mathcal{C}(G)$.
  The set of all (connected) subgraphs is analogous to the power set of a set.

Subgraph Isomorphism: Examples

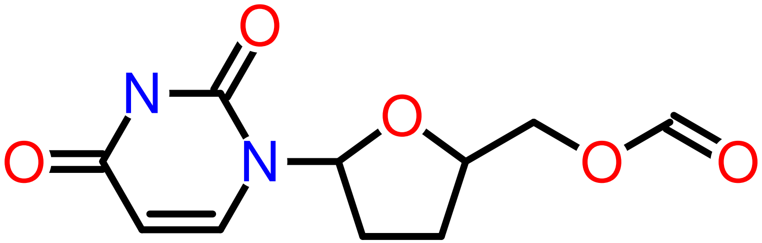
S_1





S_2

G



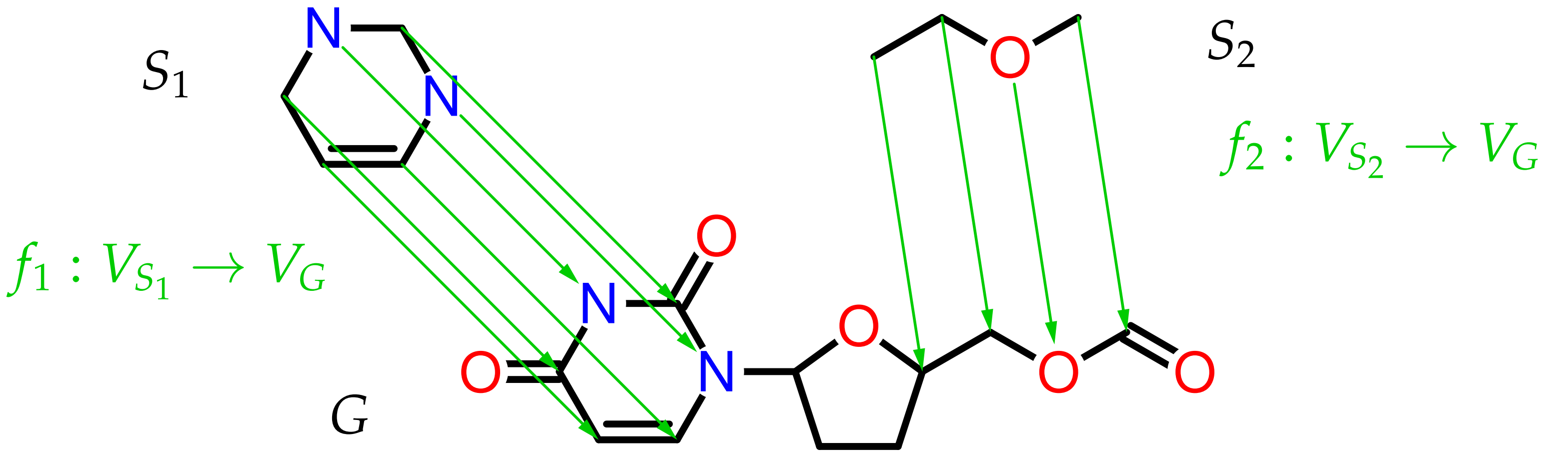
- A molecule G that represents a graph in a database and two graphs S_1 and S_2 that are contained in G.

- The subgraph relationship is formally described by a mapping f of the vertices of one graph to the vertices of another:

$$G = (V_G, E_G), \qquad S = (V_S, E_S), \qquad f : V_S \to V_G.$$

- This mapping must preserve the connection structure and the labels.

Subgraph Isomorphism: Examples



- The mapping must preserve the connection structure:

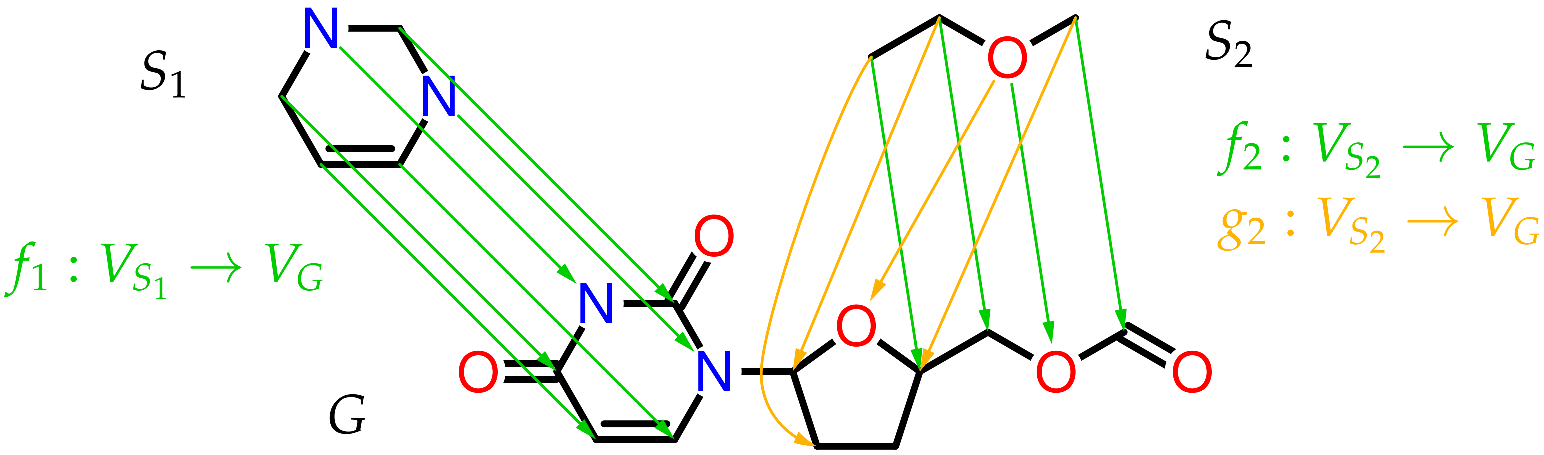
$$\forall (u,v) \in E_S: \quad (f(u), f(v)) \in E_G.$$

- The mapping must preserve vertex and edge labels:

$$\forall v \in V_S: \ \ell_S(v) = \ell_G(f(v)), \qquad \forall (u,v) \in E_S: \ \ell_S((u,v)) = \ell_G((f(u), f(v))).$$

Here: oxygen must be mapped to oxygen, single bonds to single bonds etc.

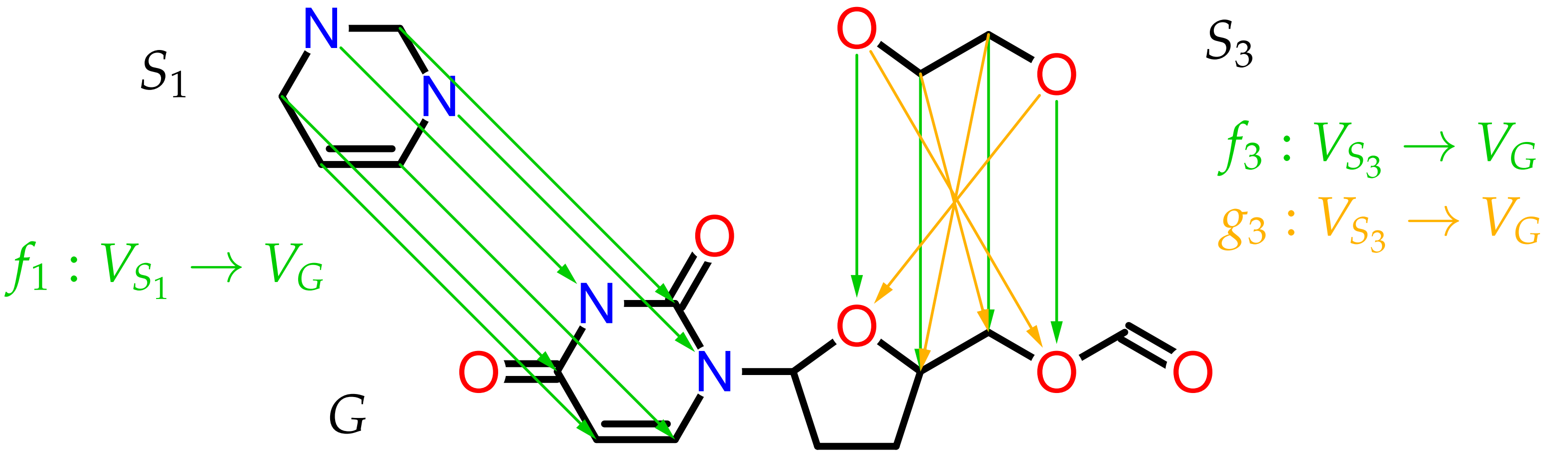
Subgraph Isomorphism: Examples



- There may be more than one possible mapping / occurrence. (There are even three more occurrences of S_2.)
- However, we are currently only interested in whether there exists a mapping.

  (The number of occurrences will become important when we consider mining frequent (sub)graphs in a single graph.)
- Testing whether a subgraph isomorphism exists between given graphs S and G is **NP-complete** (that is, requires exponential time unless $\text{P} = \text{NP}$).

Subgraph Isomorphism: Examples



- A graph may be mapped to itself (**automorphism**).
- Trivially, every graph possesses the identity as an automorphism. (Every graph can be mapped to itself by mapping each vertex to itself.)
- If a graph (fragment) possesses an automorphism that is not the identity there is more than one occurrence *at the same location* in another graph.
- The number of occurrences of a graph (fragment) in a graph can be huge.

Frequent (Sub)Graph Mining: Basic Notions

Let S be a labeled graph and $\mathcal{G}$ a tuple of labeled graphs.

- A labeled graph $G \in \mathcal{G}$ **covers** the labeled graph S or the labeled graph S is **contained in** a labeled graph $G \in \mathcal{G}$ iff $S \sqsubseteq G$.

- The set $K_{\mathcal{G}}(S) = \{k \in \{1, \ldots, n\} \mid S \sqsubseteq G_k\}$ is called the **cover** of S w.r.t. $\mathcal{G}$.

  The cover of a graph is the index set of the database graphs that cover it.

  It may also be defined as a tuple of all labeled graphs that cover it (which, however, is complicated to write in formally correct way).

- The value $s_{\mathcal{G}}(S) = |K_{\mathcal{G}}(S)|$ is called the **(absolute) support** of S w.r.t. $\mathcal{G}$.

  The value $\sigma_{\mathcal{G}}(S) = \frac{1}{n}\,|K_{\mathcal{G}}(S)|$ is called the **relative support** of S w.r.t. $\mathcal{G}$.

  The support of S is the number or fraction of labeled graphs that contain it.

  Sometimes $\sigma_{\mathcal{G}}(S)$ is also called the *(relative) frequency* of S w.r.t. $\mathcal{G}$.

Frequent (Sub)Graph Mining: Formal Definition

Given:

- a set $A = \{a_1, \ldots, a_m\}$ of attributes or labels,
- a tuple $\mathcal{G} = (G_1, \ldots, G_n)$ of graphs with labels in A,
- a number $s_{\min} \in \mathbb{N}, 1 \leq s_{\min} \leq n,$ or (equivalently)
  a number $\sigma_{\min} \in \mathbb{R}, 0 < \sigma_{\min} \leq 1,$ the **minimum support**.

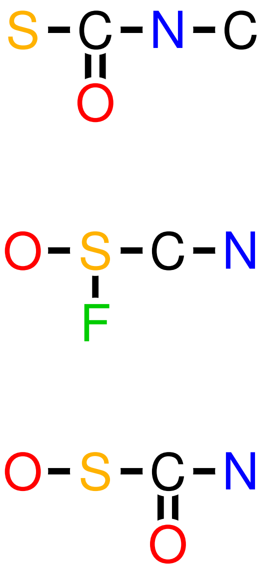
Desired:

- the set of **frequent (sub)graphs** or **frequent fragments**, that is,
  the set $F_{\mathcal{G}}(s_{\min}) = \{S \mid s_{\mathcal{G}}(S) \geq s_{\min}\}$ or (equivalently)
  the set $\Phi_{\mathcal{G}}(\sigma_{\min}) = \{S \mid \sigma_{\mathcal{G}}(S) \geq \sigma_{\min}\}$.

Note that with the relations $s_{\min} = \lceil n\sigma_{\min} \rceil$ and $\sigma_{\min} = \frac{1}{n} s_{\min}$ the two versions can easily be transformed into each other.

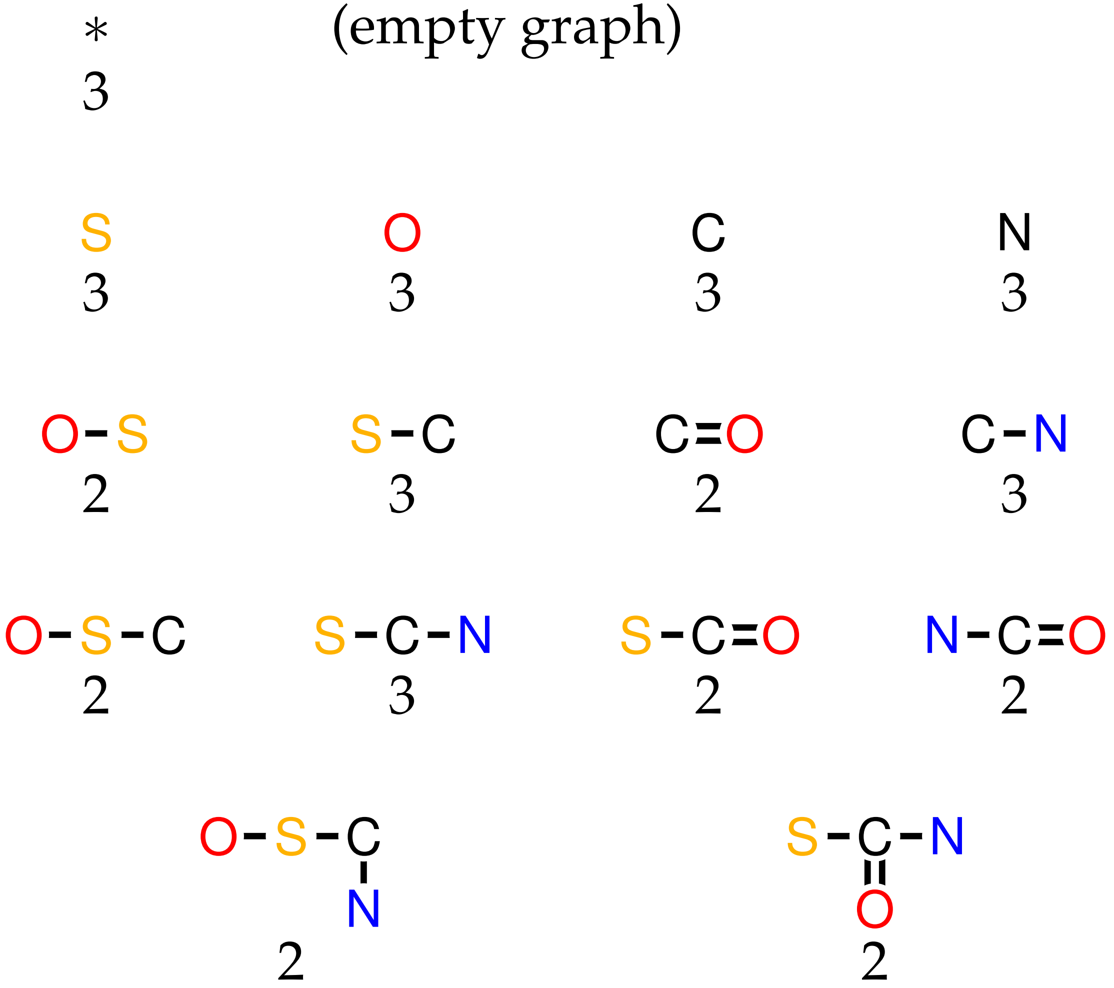
Frequent (Sub)Graphs: Example

example molecules
(graph database)



The numbers
below the subgraphs
state their support.

frequent molecular fragments ($s_{\min} = 2$)



Properties of the Support of (Sub)Graphs

- A **brute force approach** that enumerates all possible (sub)graphs, determines their support, and discards infrequent (sub)graphs is usually **infeasible**:

  The number of possible (connected) (sub)graphs,
  grows very quickly with the number of vertices and edges.

- **Idea:** Consider the properties of a (sub)graph's cover and support, in particular:

  $$\forall S : \forall R \supseteq S : \quad K_{\mathcal{G}}(R) \subseteq K_{\mathcal{G}}(S).$$

  This property holds, because $\forall G : \forall S : \forall R \supseteq S : \quad R \sqsubseteq G \to S \sqsubseteq G$.

  Each additional edge is another condition a database graph has to satisfy.
  Graphs that do not satisfy this condition are removed from the cover.

- It follows: $$\forall S : \forall R \supseteq S : \quad s_{\mathcal{G}}(R) \leq s_{\mathcal{G}}(S).$$

  That is: **If a (sub)graph is extended, its support cannot increase.**

  One also says that support is **anti-monotone** or **downward closed**.

Properties of the Support of (Sub)Graphs

- From $\forall S: \forall R \supseteq S: s_{\mathcal{G}}(R) \leq s_{\mathcal{G}}(S)$ it follows

  $$\forall s_{\min}: \forall S: \forall R \supseteq S: \quad s_{\mathcal{G}}(S) < s_{\min} \;\rightarrow\; s_{\mathcal{G}}(R) < s_{\min}.$$

  That is: **No supergraph of an infrequent (sub)graph can be frequent.**

- This property is often referred to as the **Apriori Property**.

  Rationale: Sometimes we can know *a priori*, that is, before checking its support by accessing the given graph database, that a (sub)graph cannot be frequent.

- Of course, the contraposition of this implication also holds:

  $$\forall s_{\min}: \forall R: \forall S \subseteq R: \quad s_{\mathcal{G}}(R) \geq s_{\min} \;\rightarrow\; s_{\mathcal{G}}(S) \geq s_{\min}.$$

  That is: **All subgraphs of a frequent (sub)graph are frequent.**

- This suggests a compressed representation of the set of frequent (sub)graphs.

Reminder: Partially Ordered Sets

- A **partial order** is a binary relation $\leq$ over a set S which satisfies $\forall a, b, c \in S$:
  - $a \leq a$ (reflexivity)
  - $a \leq b \wedge b \leq a \;\Rightarrow\; a = b$ (anti-symmetry)
  - $a \leq b \wedge b \leq c \;\Rightarrow\; a \leq c$ (transitivity)
- A set with a partial order is called a **partially ordered set** (or **poset** for short).
- Let a and b be two distinct elements of a partially ordered set $(S, \leq)$.
  - if $a \leq b$ or $b \leq a$, then a and b are called **comparable**.
  - if neither $a \leq b$ nor $b \leq a$, then a and b are called **incomparable**.
- If all pairs of elements of the underlying set S are comparable, the order $\leq$ is called a **total order** or a **linear order**.
- In a total order the reflexivity axiom is replaced by the stronger axiom:
  - $a \leq b \vee b \leq a$ (totality)

Properties of the Support of (Sub)Graphs

Monotonicity in Calculus and Analysis

- A function $f : \mathbb{R} \to \mathbb{R}$ is called **monotonically non-decreasing** if $\quad \forall x, y : \quad x \leq y \;\Rightarrow\; f(x) \leq f(y)$.
- A function $f : \mathbb{R} \to \mathbb{R}$ is called **monotonically non-increasing** if $\quad \forall x, y : \quad x \leq y \;\Rightarrow\; f(x) \geq f(y)$.

Monotonicity in Order Theory

- Order theory is concerned with arbitrary partially ordered sets. The terms *increasing* and *decreasing* are avoided, because they lose their pictorial motivation as soon as sets are considered that are not totally ordered.
- A function $f : S_1 \to S_2$, where S_1 and S_2 are two partially ordered sets, is called **monotone** or **order-preserving** if $\forall x, y \in S_1 : \; x \leq y \;\Rightarrow\; f(x) \leq f(y)$.
- A function $f : S_1 \to S_2$, is called **anti-monotone** or **order-reversing** if $\forall x, y \in S_1 : \; x \leq y \;\Rightarrow\; f(x) \geq f(y)$.
- In this sense the support of a (sub)graph is anti-monotone.

Properties of Frequent (Sub)Graphs

- A subset R of a partially ordered set $(S, \leq)$ is called **downward closed** if for any element of the set all smaller elements are also in it:

  $$\forall x \in R : \forall y \in S : \quad y \leq x \quad \Rightarrow \quad y \in R$$

  In this case the subset R is also called a **lower set**.

- The notions of **upward closed** and **upper set** are defined analogously.

- For every $s_{\min}$ the set of frequent (sub)graphs $F_{\mathcal{G}}(s_{\min})$ is downward closed w.r.t. the partial order $\sqsubseteq$:

  $$\forall S \in F_{\mathcal{G}}(s_{\min}) : \quad R \sqsubseteq S \quad \Rightarrow \quad R \in F_{\mathcal{G}}(s_{\min}).$$

- Since the set of frequent (sub)graphs is induced by the support function, the notions of **up-** or **downward closed** are transferred to the support function:

  Any set of graphs w.r.t. a support threshold $s_{\min}$ is up- or downward closed.

  $F_{\mathcal{G}}(s_{\min}) = \{S \mid s_{\mathcal{G}}(S) \geq s_{\min}\}$ (frequent (sub)graphs) is downward closed,
  $I_{\mathcal{G}}(s_{\min}) = \{S \mid s_{\mathcal{G}}(S) < s_{\min}\}$ (infrequent (sub)graphs) is upward closed.

Types of Frequent (Sub)Graphs

Maximal (Sub)Graphs

- Consider the set of **maximal (frequent) (sub)graphs / fragments**:

$$M_{\mathcal{G}}(s_{\min}) = \{S \mid s_{\mathcal{G}}(S) \geq s_{\min} \land \forall R \supset S : s_{\mathcal{G}}(R) < s_{\min}\}.$$

  That is: A (sub)graph is maximal if it is frequent,
  but none of its proper supergraphs is frequent.

- Since with this definition we know that

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) : \quad S \in M_{\mathcal{G}}(s_{\min}) \ \lor \ \exists R \supset S : s_{\mathcal{G}}(R) \geq s_{\min}$$

  it follows (can easily be proven by successively extending the graph S)

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) : \exists R \in M_{\mathcal{G}}(s_{\min}) : \quad S \subseteq R.$$

  That is: **Every frequent (sub)graph has a maximal supergraph.**

- Therefore: $$\forall s_{\min} : \quad F_{\mathcal{G}}(s_{\min}) = \bigcup_{S \in M_{\mathcal{G}}(s_{\min})} \mathcal{C}(S).$$

Reminder: Maximal Elements

- Let R be a subset of a partially ordered set $(S, \leq)$.

  An element $x \in R$ is called **maximal** or a **maximal element** of R if

  $$\forall y \in R: \quad x \leq y \quad \Rightarrow \quad x = y.$$

- The notions **minimal** and **minimal element** are defined analogously.

- Maximal elements need not be unique, because there may be elements $y \in R$ with neither $x \leq y$ nor $y \leq x$.

- Infinite partially ordered sets need not possess a maximal element.

- Here we consider the set $F_{\mathcal{G}}(s_{\min})$ together with the partial order $\sqsubseteq$:

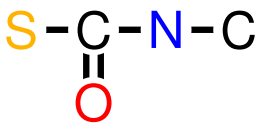
  The **maximal** (frequent) **(sub)graphs** are the maximal elements of $F_{\mathcal{G}}(s_{\min})$:

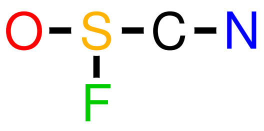
  $$M_{\mathcal{G}}(s_{\min}) = \{S \in F_{\mathcal{G}}(s_{\min}) \mid \forall R \in F_{\mathcal{G}}(s_{\min}): \; S \sqsubseteq R \;\Rightarrow\; S \equiv R\}.$$

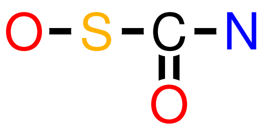
  That is, no supergraph of a maximal (frequent) (sub)graph is frequent.

Maximal (Sub)Graphs: Example

example molecules (graph database)



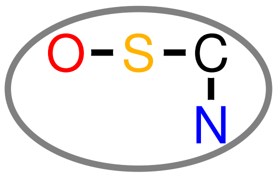




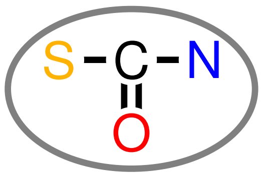
frequent molecular fragments ($s_{\min} = 2$)

$*$ (empty graph)
3

S	C	O	N
3	3	3	3
O-S	S-C	C=O	C-N
2	3	2	3
O-S-C	S-C-N	S-C=O	N-C=O
2	3	2	2



2



2

The numbers below the subgraphs state their support.

Limits of Maximal (Sub)Graphs

- The set of maximal (sub)graphs captures the set of all frequent (sub)graphs, but then we know only the support of the maximal (sub)graphs.

- About the support of a non-maximal frequent (sub)graphs we only know:

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) - M_{\mathcal{G}}(s_{\min}) : \quad s_{\mathcal{G}}(S) \geq \max_{R \in M_{\mathcal{G}}(s_{\min}), R \supset S} s_{\mathcal{G}}(R).$$

  This relation follows immediately from $\forall S : \forall R \supseteq S : s_{\mathcal{G}}(S) \geq s_{\mathcal{G}}(R)$, that is, a (sub)graph cannot have a lower support than any of its supergraphs.

- Note that we have generally

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) : \quad s_{\mathcal{G}}(S) \geq \max_{R \in M_{\mathcal{G}}(s_{\min}), R \supseteq S} s_{\mathcal{G}}(R).$$

- **Question:** Can we find a subset of the set of all frequent (sub)graphs, which also preserves knowledge of all support values?

Closed (Sub)Graphs

- Consider the set of **closed (frequent) (sub)graphs / fragments**:

$$C_{\mathcal{G}}(s_{\min}) = \{S \mid s_{\mathcal{G}}(S) \geq s_{\min} \wedge \forall R \supset S : s_{\mathcal{G}}(R) < s_{\mathcal{G}}(S)\}.$$

  That is: A (sub)graph is closed if it is frequent,
  but none of its proper supergraphs has the same support.

- Since with this definition we know that

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) : \quad S \in C_{\mathcal{G}}(s_{\min}) \ \vee \ \exists R \supset S : s_{\mathcal{G}}(R) = s_{\mathcal{G}}(S)$$

  it follows (can easily be proven by successively extending the graph S)

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) : \exists R \in C_{\mathcal{G}}(s_{\min}) : \quad S \subseteq R.$$

  That is: **Every frequent (sub)graph has a closed supergraph.**

- Therefore: $$\forall s_{\min} : \quad F_{\mathcal{G}}(s_{\min}) = \bigcup_{S \in C_{\mathcal{G}}(s_{\min})} \mathcal{C}(S).$$

Closed (Sub)Graphs

- However, not only has every frequent (sub)graph a closed supergraph, but it has a **closed supergraph with the same support**:

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) : \exists R \supseteq S : \quad R \in C_{\mathcal{G}}(s_{\min}) \ \wedge \ s_{\mathcal{G}}(R) = s_{\mathcal{G}}(S).$$

  (Proof: consider the closure operator that is defined on the following slides.) Note, however, that the supergraph need not be unique — see below.

- The set of all closed (sub)graphs preserves knowledge of all support values:

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) : \quad s_{\mathcal{G}}(S) = \max_{R \in C_{\mathcal{G}}(s_{\min}), R \supseteq S} s_{\mathcal{G}}(R).$$

- Note that the weaker statement

$$\forall s_{\min} : \forall S \in F_{\mathcal{G}}(s_{\min}) : \quad s_{\mathcal{G}}(S) \geq \max_{R \in C_{\mathcal{G}}(s_{\min}), R \supseteq S} s_{\mathcal{G}}(R)$$

  follows immediately from $\forall S : \forall R \supseteq S : s_{\mathcal{G}}(S) \geq s_{\mathcal{G}}(R)$, that is, a (sub)graph cannot have a lower support than any of its supergraphs.

Reminder: Closure Operators

- A **closure operator** on a set S is a function $cl : 2^S \to 2^S$, which satisfies the following conditions $\forall X, Y \subseteq S$:
  - $X \subseteq cl(X)$ (cl is extensive)
  - $X \subseteq Y \;\Rightarrow\; cl(X) \subseteq cl(Y)$ (cl is increasing or monotone)
  - $cl(cl(X)) = cl(X)$ (cl is idempotent)

- A set $R \subseteq S$ is called **closed** if it is equal to its closure:

  R is closed $\;\Leftrightarrow\;$ $R = cl(R)$.

- The **closed (frequent) item sets** are induced by the closure operator

$$cl(I) = \bigcap_{k \in K_T(I)} t_k.$$

  restricted to the set of frequent item sets:

$$C_T(s_{\min}) = \{I \in F_T(s_{\min}) \mid I = cl(I)\}$$

Closed (Sub)Graphs

- **Question:** Is there a closure operator that induces the closed (sub)graphs?
- At first glance, it appears natural to transfer the operation

  $$cl(I) = \bigcap_{k \in K_T(I)} t_k$$

  by replacing the intersection with the **greatest common subgraph**.
- Unfortunately, this is not possible, because the greatest common subgraph of two (or more) graphs need not be uniquely defined.
  - Consider the two graphs (which are actually chains):

    $$A - B - C \qquad \text{and} \qquad A - B - B - C.$$

  - There are two greatest (connected) common subgraphs:

    $$A - B \qquad \text{and} \qquad B - C.$$

- As a consequence, the intersection of a set of database graphs can yield a *set of graphs* instead of a single common graph.

Reminder: Galois Connections

- Let $(X, \preceq_X)$ and $(Y, \preceq_Y)$ be two partially ordered sets.

- A function pair (f_1, f_2) with $f_1 : X \to Y$ and $f_2 : Y \to X$ is called a **(monotone) Galois connection** iff
  - $\forall A_1, A_2 \in X : \quad A_1 \preceq_X A_2 \quad \Rightarrow \quad f_1(A_1) \preceq_Y f_1(A_2)$,
  - $\forall B_1, B_2 \in Y : \quad B_1 \preceq_Y B_2 \quad \Rightarrow \quad f_2(B_1) \preceq_Y f_2(B_2)$,
  - $\forall A \in X : \forall B \in Y : \quad A \preceq_X f_2(B) \quad \Leftrightarrow \quad B \preceq_Y f_1(A)$.

- A function pair (f_1, f_2) with $f_1 : X \to Y$ and $f_2 : Y \to X$ is called an **anti-monotone Galois connection** iff
  - $\forall A_1, A_2 \in X : \quad A_1 \preceq_X A_2 \quad \Rightarrow \quad f_1(A_1) \succeq_Y f_1(A_2)$,
  - $\forall B_1, B_2 \in Y : \quad B_1 \preceq_Y B_2 \quad \Rightarrow \quad f_2(B_1) \succeq_X f_2(B_2)$,
  - $\forall A \in X : \forall B \in Y : \quad A \preceq_X f_2(B) \quad \Leftrightarrow \quad B \preceq_Y f_1(A)$.

- In a monotone Galois connection, both f_1 and f_2 are monotone, in an anti-monotone Galois connection, both f_1 and f_2 are anti-monotone.

Reminder: Galois Connections

Galois Connections and Closure Operators

- Let the two sets X and Y be power sets of some sets U and V, respectively, and let the partial orders be the subset relations on these power sets, that is, let

$$(X, \preceq_X) = (2^U, \subseteq) \qquad \text{and} \qquad (Y, \preceq_Y) = (2^V, \subseteq).$$

- Then the combination $f_1 \circ f_2 : X \to X$ of the functions of a Galois connection is a **closure operator** (as well as the combination $f_2 \circ f_1 : Y \to Y$).

Galois Connections in Frequent Item Set Mining

- Consider the partially order sets $(2^B, \subseteq)$ and $(2^{\{1,\ldots,n\}}, \subseteq)$.

  Let $\quad f_1 : \quad 2^B \to 2^{\{1,\ldots,n\}}, \quad I \mapsto K_T(I) = \{k \in \{1, \ldots, n\} \mid I \subseteq t_k\}$

  and $\quad f_2 : \quad 2^{\{1,\ldots,n\}} \to 2^B, \quad J \mapsto \bigcap_{j \in J} t_j = \{i \in B \mid \forall j \in J : i \in t_j\}.$

- The function pair (f_1, f_2) is an **anti-monotone Galois connection**. Therefore the combination $f_1 \circ f_2 : 2^B \to 2^B$ is a **closure operator**.

Galois Connections in Frequent (Sub)Graph Mining

- Let $\mathcal{G} = (G_1, \ldots, G_n)$ be a tuple of database graphs.

- Let U be the set of all subgraphs of the database graphs in $\mathcal{G}$, that is, $U = \bigcup_{k \in \{1,\ldots,n\}} \mathcal{C}(G_k)$ (set of connected (sub)graphs)

- Let V be the index set of the database graphs in $\mathcal{G}$, that is $V = \{1, \ldots, n\}$ (set of graph identifiers).

- $(2^U, \subseteq)$ and $(2^V, \subseteq)$ are partially ordered sets. Consider the function pair

$$f_1 : 2^U \to 2^V, \qquad I \mapsto \{k \in V \mid \forall S \in I : S \sqsubseteq G_k\}. \qquad \text{and}$$
$$f_2 : 2^V \to 2^U \qquad J \mapsto \{S \in U \mid \forall k \in J : S \sqsubseteq G_k\},$$

- The pair (f_1, f_2) is a Galois connection of $X = (2^U, \subseteq)$ and $Y = (2^V, \subseteq)$:
  - ○ $\forall A_1, A_2 \in 2^U : \qquad A_1 \subseteq A_2 \quad \Rightarrow \quad f_1(A_1) \supseteq f_1(A_2),$
  - ○ $\forall B_1, B_2 \in 2^V : \qquad B_1 \subseteq B_2 \quad \Rightarrow \quad f_2(B_1) \supseteq f_2(B_2),$
  - ○ $\forall A \in 2^U : \forall B \in 2^V : \qquad A \subseteq f_2(B) \quad \Leftrightarrow \quad B \subseteq f_1(A).$

Galois Connections in Frequent (Sub)Graph Mining

- Since the function pair (f_1, f_2) is an (anti-monotone) Galois connection, $f_1 \circ f_2 : 2^{\mathcal{U}} \rightarrow 2^{\mathcal{U}}$ is a **closure operator**.
- This closure operator can be used to define the closed (sub)graphs:

  A subgraph S is **closed** w.r.t. a graph database $\mathcal{G}$ iff

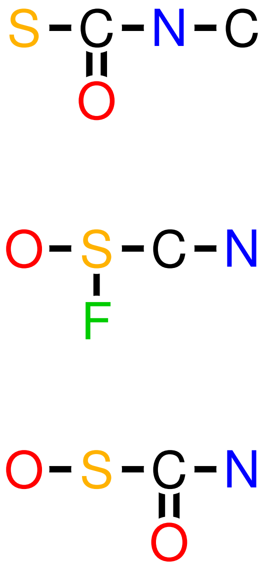
$$S \in (f_1 \circ f_2)(\{S\}) \qquad \wedge \qquad \nexists\, G \in (f_1 \circ f_2)(\{S\}) : \; S \sqsubset G.$$

- The generalization to a Galois connection takes formally care of the problem that the greatest common subgraph may not be uniquely determined.
- Intuitively, the above definition simply says that a subgraph S is closed iff
  - ○ it is a (connected) common subgraph of all database graphs containing it &
  - ○ no supergraph is also a (connected) common subgraph of all of these graphs.

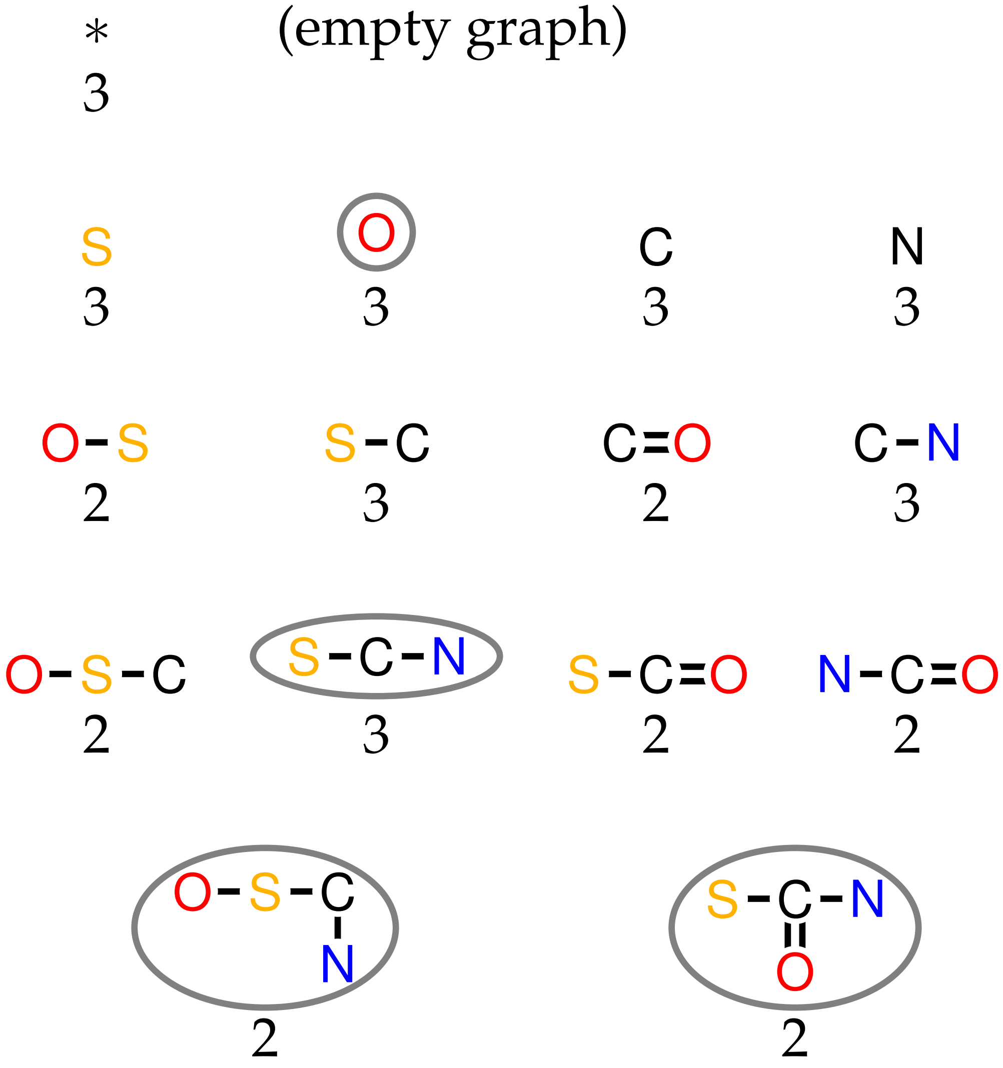
  That is, a subgraph S is closed if it is **one** of the greatest common (connected) subgraphs of all database graphs containing it.
- The Galois connection is only needed to prove the closure operator property.

Closed (Sub)Graphs: Example

example molecules
(graph database)



frequent molecular fragments ($s_{\min} = 2$)



The numbers below the subgraphs state their support.

Types of Frequent (Sub)Graphs

- **Frequent (Sub)Graph**

  Any frequent (sub)graph (support is higher than the minimum support):

  I frequent $\quad \Leftrightarrow \quad s_{\mathcal{G}}(S) \geq s_{\min}$

- **Closed (Sub)Graph**

  A frequent (sub)graph is called *closed* if no supergraph has the same support:

  I closed $\quad \Leftrightarrow \quad s_{\mathcal{G}}(S) \geq s_{\min} \quad \wedge \quad \forall R \supset S : s_{\mathcal{G}}(R) < s_{\mathcal{G}}(S)$

- **Maximal (Sub)Graph**

  A frequent (sub)graph is called *maximal* if no supergraph is frequent:

  I maximal $\quad \Leftrightarrow \quad s_{\mathcal{G}}(S) \geq s_{\min} \quad \wedge \quad \forall R \supset S : s_{\mathcal{G}}(R) < s_{\min}$

- Obvious relations between these types of (sub)graphs:
  - All maximal and all closed (sub)graphs are frequent.
  - All maximal (sub)graphs are closed.

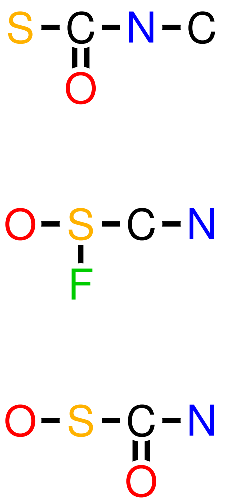
Searching for Frequent (Sub)Graphs

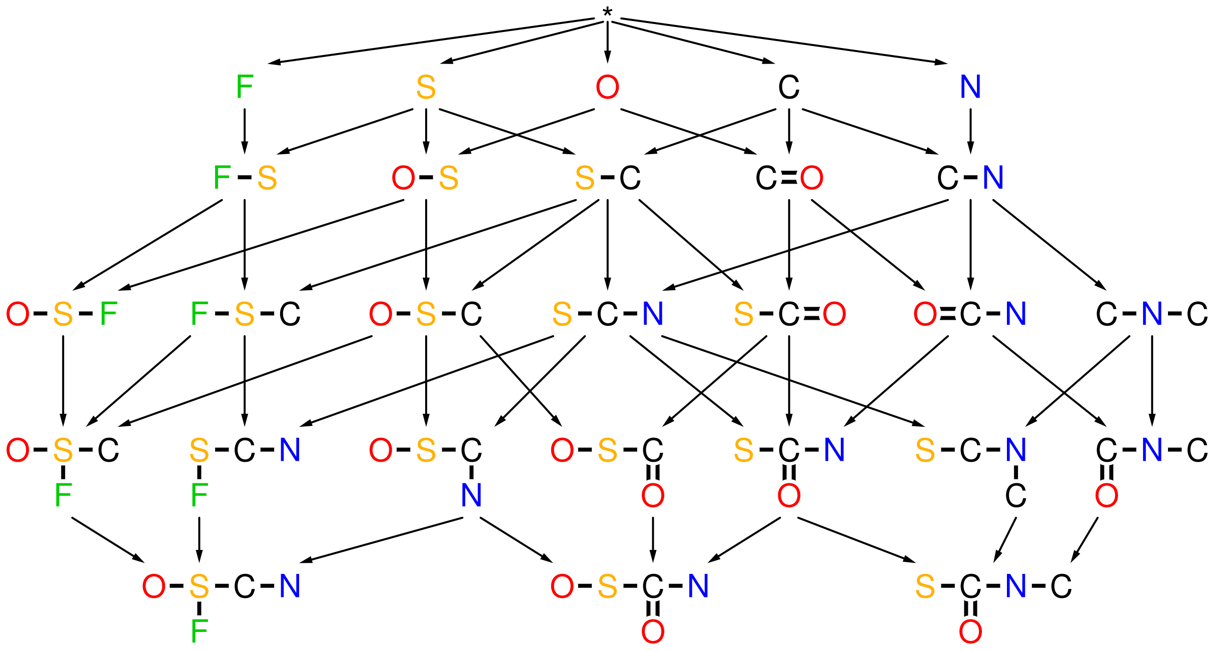
Partially Ordered Set of Subgraphs

Hasse diagram ranging from the empty graph to the database graphs.

- The subgraph (isomorphism) relation defines a partial order on (sub)graphs.
- The empty graph is (formally) contained in all database graphs.
- There is usually no (natural) unique largest graph.

example molecules:



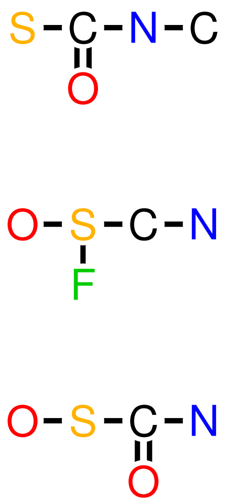


Frequent (Sub)Graphs

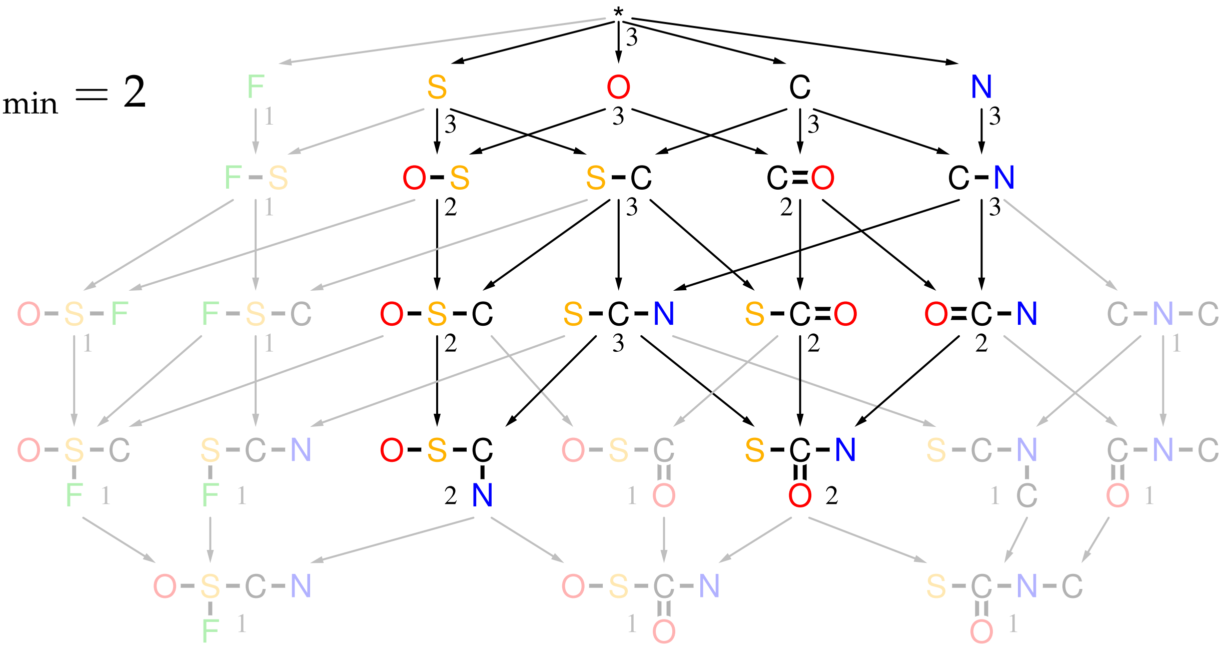
The frequent (sub)graphs form a partially ordered subset at the top.

- Therefore: the partially ordered set should be searched top-down.
- Standard search strategies: breadth-first and depth-first.
- Depth-first search is usually preferable, since the search tree can be very wide.

example molecules:



$s_{\min} = 2$

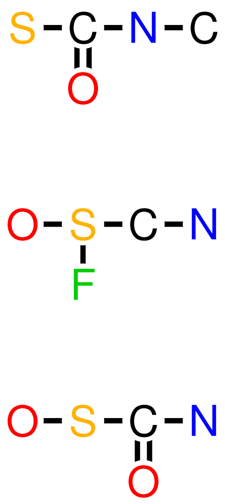


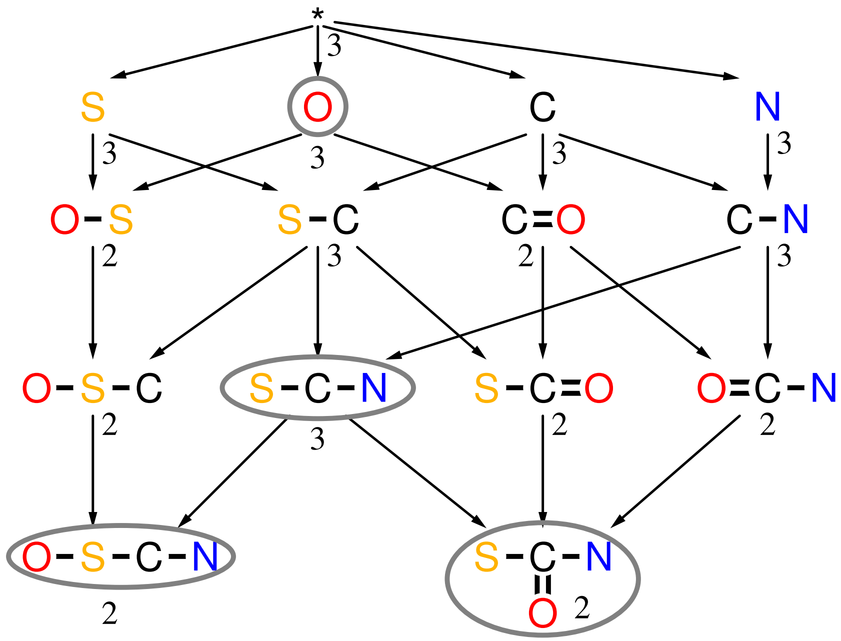
Closed and Maximal Frequent (Sub)Graphs

Partially ordered subset of frequent (sub)graphs.

- Closed frequent (sub)graphs are encircled.
- There are 14 frequent (sub)graphs, but only 4 closed (sub)graphs.
- The two closed (sub)graphs at the bottom are also maximal.

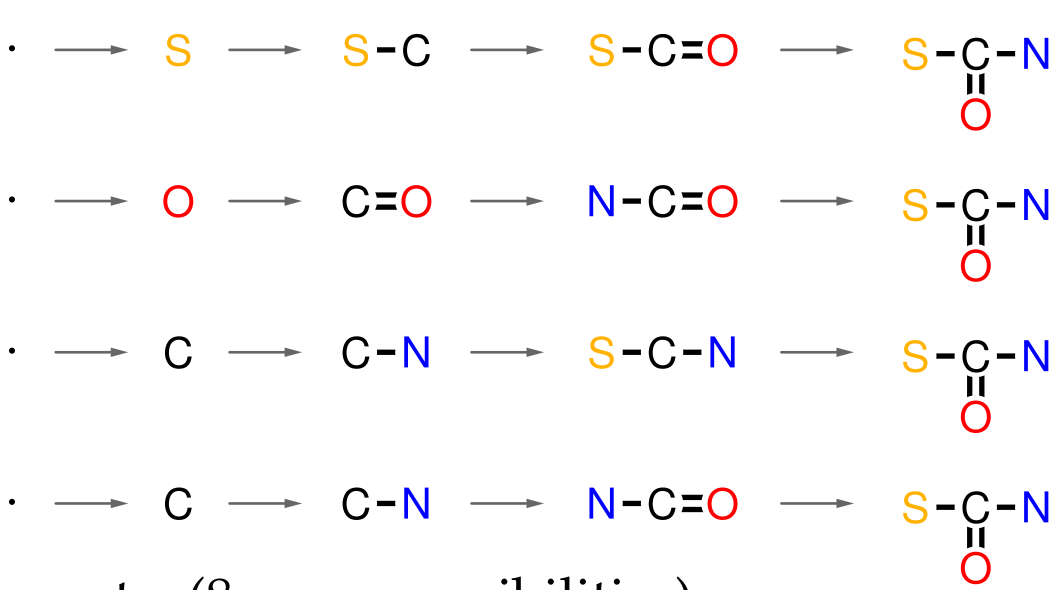
example molecules:





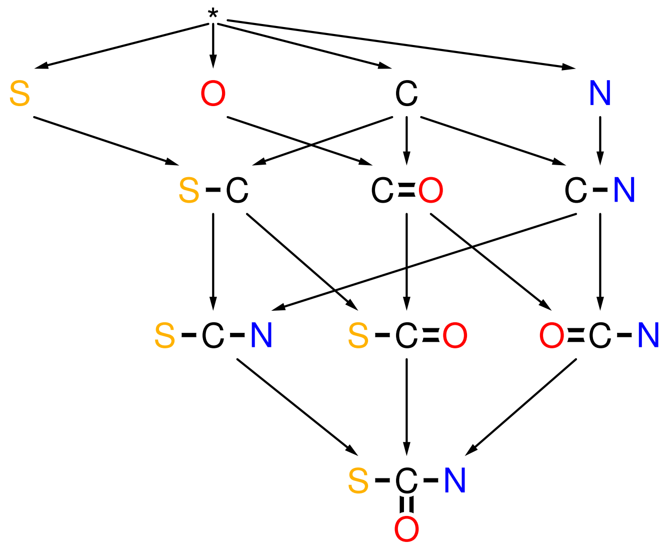
Basic Search Principle

- **Grow (sub)graphs into the graphs of the given database.**
  - ○ Start with a single vertex (seed vertex).
  - ○ Add an edge (and maybe a vertex) in each step.
  - ○ Determine the support and prune infrequent (sub)graphs.
- Main problem: **A (sub)graph can be grown in several different ways.**



· → S → S-C → S-C=O → S-C(=O)-N
· → O → C=O → N-C=O → S-C(=O)-N
· → C → C-N → S-C-N → S-C(=O)-N
· → C → C-N → N-C=O → S-C(=O)-N

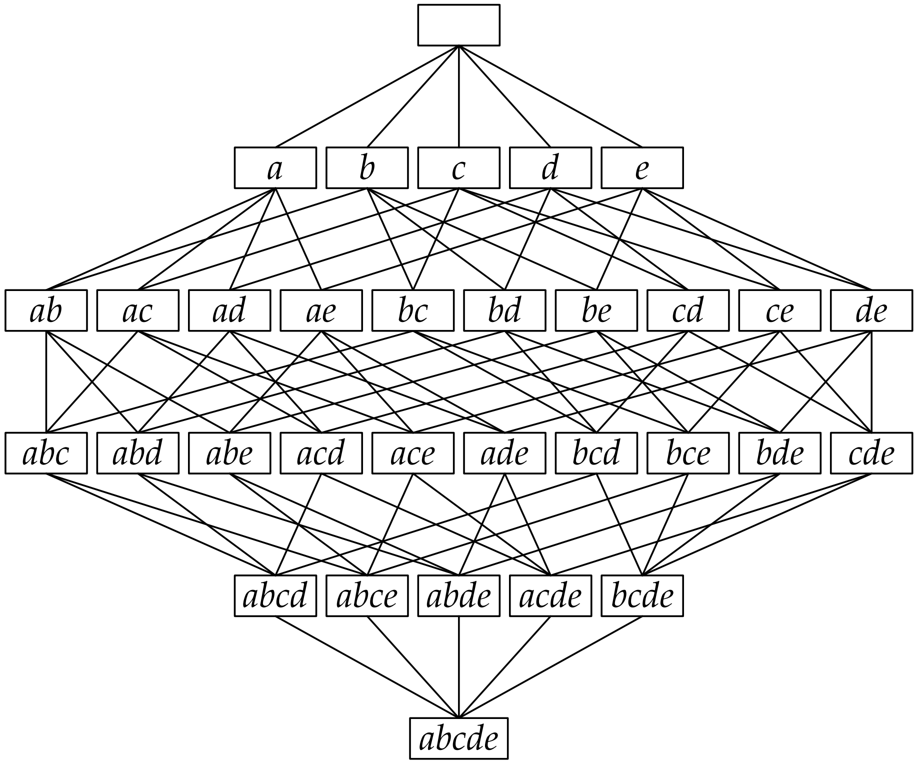
etc. (8 more possibilities)

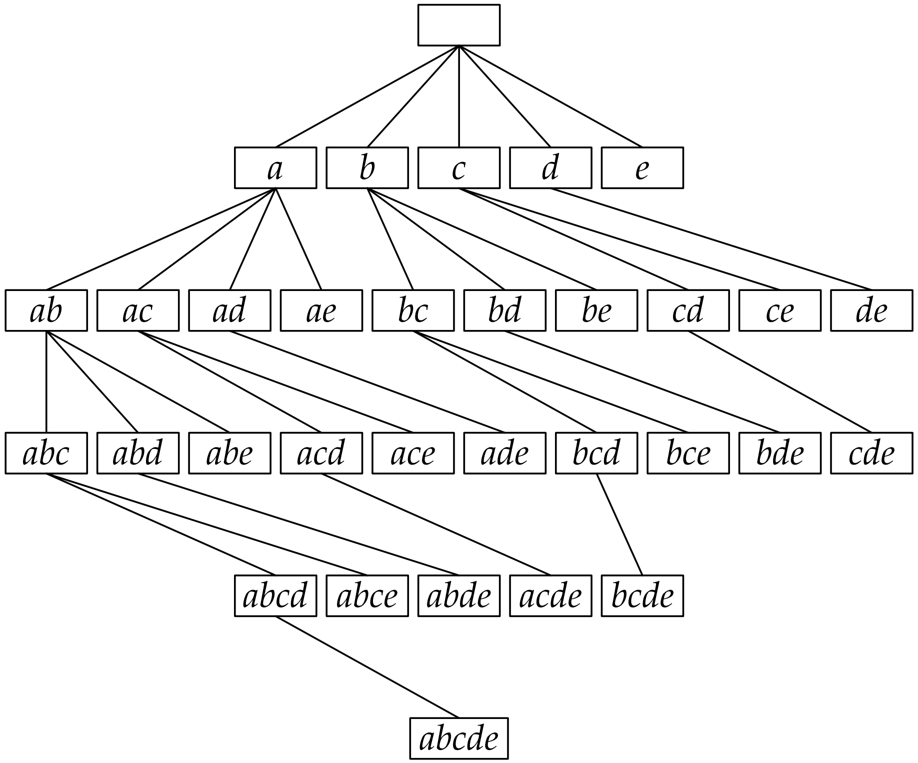


Reminder: Searching for Frequent Item Sets

- We have to search the partially ordered set $(2^B, \subseteq)$ / its Hasse diagram.
- Assigning unique parents turns the Hasse diagram into a tree.
- Traversing the resulting tree explores each item set exactly once.

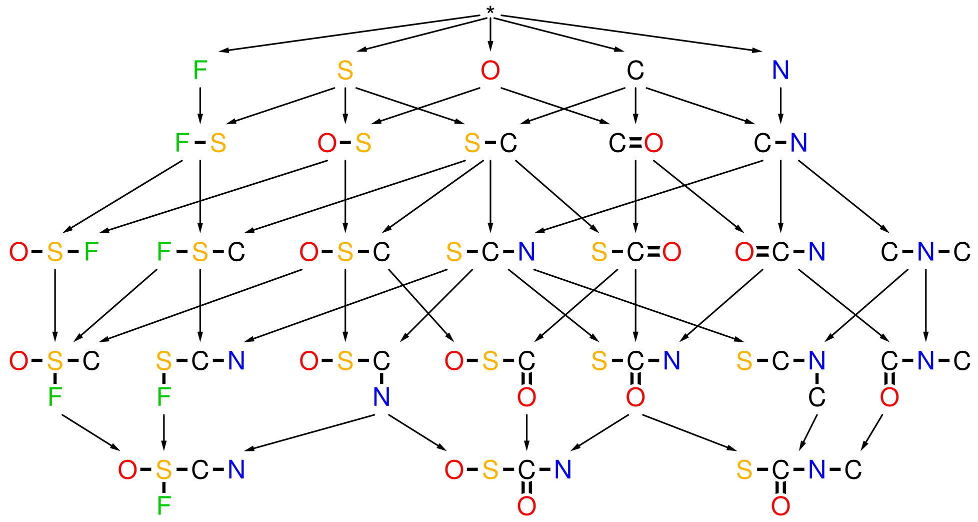
Hasse diagram and a possible tree for five items:

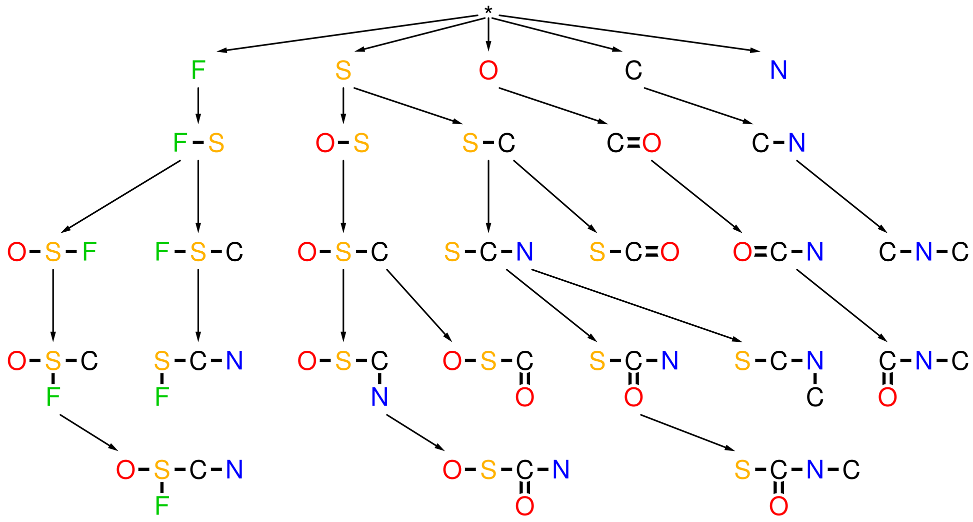




- We have to search the partially ordered set of (connected) (sub)graphs ranging from the empty graph to the database graphs.
- Assigning unique parents turns the corresponding Hasse diagram into a tree.
- Traversing the resulting tree explores each (sub)graph exactly once.

Subgraph Hasse diagram and a possible tree:





Searching with Unique Parents

Principle of a Search Algorithm based on Unique Parents:

- **Base Loop:**
  - Traverse all vertex attributes (their unique parent is the empty graph).
  - Recursively process all vertex attributes that are frequent.
- **Recursive Processing:**

  For a given frequent (sub)graph S:
  - Generate all extensions R of S by an edge or by an edge and a vertex (if the vertex is not yet in S) for which S is the chosen unique parent.
  - For all R: if R is frequent, process R recursively, otherwise discard R.
- **Questions:**
  - How can we formally assign unique parents?
  - (How) Can we make sure that we generate only those extensions for which the (sub)graph that is extended is the chosen unique parent?

Assigning Unique Parents

- Formally, the set of all **possible parents** of a (connected) (sub)graph S is

  $$\Pi(S) = \{R \in \mathcal{C}(S) \mid \nexists\, U \in \mathcal{C}(S) : R \subset U \subset S\}.$$

  In other words, the possible parents of S are its *maximal proper subgraphs*.

- Each possible parent contains exactly **one edge less** than the (sub)graph S.

- If we can define a (uniquely determined) **order on the edges** of the graph S, we can easily single out a unique parent, the **canonical parent** $\pi_c(S)$:

  - Let e^* be the **last edge** in the order that is **not a proper bridge**. (that is, e^* is either a leaf bridge or no bridge).
  - The **canonical parent** $\pi_c(S)$ is the graph S **without the edge** e^*.
  - If e^* is a leaf bridge, we also have to remove the created isolated vertex.
  - If e^* is the only edge of S, we also need an order of the vertices, so that we can decide which isolated vertex to remove.
  - Note: if S is connected, then $\pi_c(S)$ is connected, as e^* is not a proper bridge.

Assigning Unique Parents

- In order to define an **order of the edges** of a given (sub)graph, we will rely on a **canonical form** of (sub)graphs.

- Canonical forms for graphs are more complex than those for item sets (reminder on next slide), because we have to capture the connection structure.

- A canonical form of a (sub)graph is a special representation of this (sub)graph.
  - ○ Each (sub)graph is described by a **code word**.
  - ○ It describes the graph structure and the vertex and edge labels (and thus implicitly orders the edges and vertices).
  - ○ The (sub)graph can be reconstructed from the code word.
  - ○ There may be multiple code words that describe the same (sub)graph.
  - ○ One of the code words is singled out as the **canonical code word**.

- There are two main principles for canonical forms of graphs:

  ○ **spanning trees** and ○ **adjacency matrices**.

Support Counting

Subgraph Isomorphism Tests

- Generate extensions based on global information about edges:
  - Collect triplets of source vertex label, edge label, destination vertex label.
  - Traverse the (extendable) vertices of a given fragment
    and attach edges based on the collected triplets.
- Traverse database graphs and test whether generated extension occurs.
  (The database graphs may be restricted to those containing the parent.)

Maintain List of Occurrences

- Find and record all occurrences of single vertex graphs.
- Check database graphs for extensions of known occurrences.
  This immediately yields the occurrences of the extended fragments.
- Disadvantage: considerable memory is needed for storing the occurrences.
- Advantage: fewer extended fragments and (possibly) faster support counting.

Canonical Forms of Graphs

Reminder: Canonical Form for Item Sets

- An item set is represented by a **code word**; each letter represents an item.

  The code word is a word over the alphabet B, the item base.

- There are $k!$ possible code words for an item set of size k, because the items may be listed in any order.

- By introducing an (arbitrary, but fixed) **order of the items**, and by comparing code words lexicographically, we can define an order on these code words.

  Example: $abc < bac < bca < cab$ for the item set $\{a, b, c\}$ and $a < b < c$.

- The lexicographically smallest code word for an item set is the **canonical code word**.

  Obviously the canonical code word lists the items in the chosen, fixed order.

In principle, the same general idea can be used for graphs.
However, a global order on the vertex and edge attributes is not enough.

Canonical Forms of Graphs: General Idea

- Construct a **code word** that uniquely identifies an (attributed or labeled) graph up to automorphisms (that is, symmetries).
- **Basic idea:** The characters of the code word describe the edges of the graph.
- **Core problem:** Vertex and edge attributes can easily be incorporated into a code word, but how to describe the connection structure is not so obvious.
- The vertices of the graph must be numbered (endowed with unique labels), because we need to specify the vertices that are incident to an edge.
  (Vertex labels need not be unique; several vertices may have the same label.)
- Each possible numbering of the vertices of the graph yields a code word, which is the concatenation of the (sorted) edge descriptions ("characters").
  (Note that the graph can be reconstructed from such a code word.)
- The resulting list of code words is sorted lexicographically.
- The lexicographically smallest code word is the **canonical code word**.
  (Alternatively, one may choose the lexicographically greatest code word.)

Searching with Canonical Forms

- Let S be a (sub)graph and $w_c(S)$ its canonical code word.

  Let $e^*(S)$ be the last edge in the edge order induced by $w_c(S)$ (i.e. the order in which the edges are described) that is *not* a proper bridge.

- **General Recursive Processing with Canonical Forms:**

  For a given frequent (sub)graph S:

  - Generate all extensions R of S by a single edge or an edge and a vertex (if one vertex incident to the edge is not yet part of S).
  - Form the canonical code word $w_c(R)$ of each extended (sub)graph R.
  - If the edge $e^*(R)$ as induced by $w_c(R)$ is the edge added to S to form R and R is frequent, process R recursively, otherwise discard R.

- **Questions:**

  - How can we formally define canonical code words?
  - Do we have to generate all possible extensions of a frequent (sub)graph?

Canonical Forms: Prefix Property

- Suppose the canonical form possesses the **prefix property**:

  Every prefix of a canonical code word is a canonical code word itself.

  ⇒ The edge e^* is always the last described edge.

  ⇒ The longest proper prefix of the canonical code word of a (sub)graph S not only describes the canonical parent of S, but is its canonical code word.

- The general recursive processing scheme with canonical forms requires to construct the **canonical code word** of each created (sub)graph in order to decide whether it has to be processed recursively or not.

  ⇒ We know the canonical code word of any (sub)graph that is processed.

- With this code word we know, due to the **prefix property**, the canonical code words of all child (sub)graphs that have to be explored in the recursion *with the exception of the last letter* (that is, the description of the added edge).

  ⇒ We only have to check whether the code word that results from appending the description of the added edge to the given code word is canonical.

Searching with the Prefix Property

Principle of a Search Algorithm based on the Prefix Property:

- **Base Loop:**
  - Traverse all possible vertex attributes, that is, the canonical code words of single vertex (sub)graphs.
  - Recursively process each code word that describes a frequent (sub)graph.

- **Recursive Processing:**

  For a given (canonical) code word of a frequent (sub)graph:

  - Generate all possible extensions by an edge (and maybe a vertex). This is done by appending the edge description to the code word.
  - Check whether the extended code word is the **canonical code word** of the (sub)graph described by the extended code word (and, of course, whether the described (sub)graph is frequent).

    If it is, process the extended code word recursively, otherwise discard it.

The Prefix Property

- **Advantages of the Prefix Property:**
  - Testing whether a given code word is canonical can be simpler/faster than constructing a canonical code word from scratch.
  - The prefix property usually allows us to easily find simple rules to *restrict the extensions* that need to be generated.
- **Disadvantages of the Prefix Property:**
  - One has reduced freedom in the definition of a canonical form.

    This can make it impossible to exploit certain properties of a graph that can help to construct a canonical form quickly.
- In the following we consider mainly canonical forms having the prefix property.
- However, it will be discussed later how additional graph properties can be exploited to improve the construction of a canonical form if the prefix property is not made a requirement.

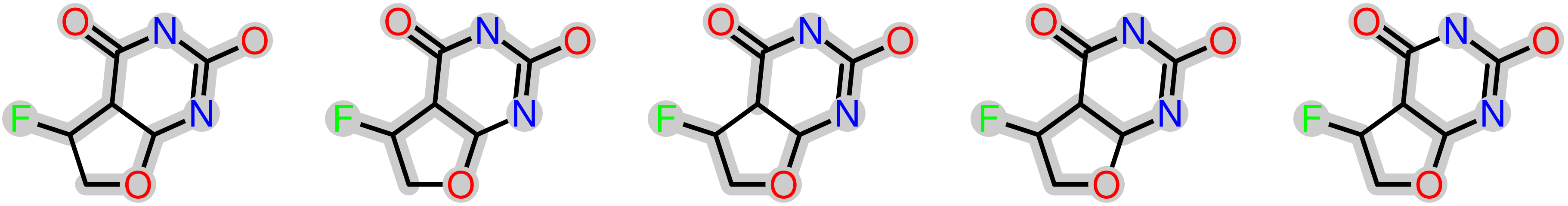
Canonical Forms based on Spanning Trees

Spanning Trees

- A (labeled) graph G is called a **tree** iff for any pair of vertices in G there exists *exactly one path* connecting them in G.

- A **spanning tree** of a (labeled) connected graph G is a subgraph S of G that
  - is a tree and
  - comprises all vertices of G (that is, $V_S = V_G$).

Examples of spanning trees:



- There are $1 \cdot 9 + 5 \cdot 4 = 6 \cdot 5 - 1 = 29$ possible spanning trees for this example, because both rings have to be cut open.

Canonical Forms based on Spanning Trees

- A **code word** describing a graph can be formed by
  - ○ systematically constructing a **spanning tree** of the graph,
  - ○ **numbering the vertices** in the order in which they are visited,
  - ○ describing each edge by the numbers of the vertices it connects, the edge label, and the labels of the incident vertices, and
  - ○ listing the edge descriptions in the order in which the edges are visited. (Edges closing cycles may need special treatment.)

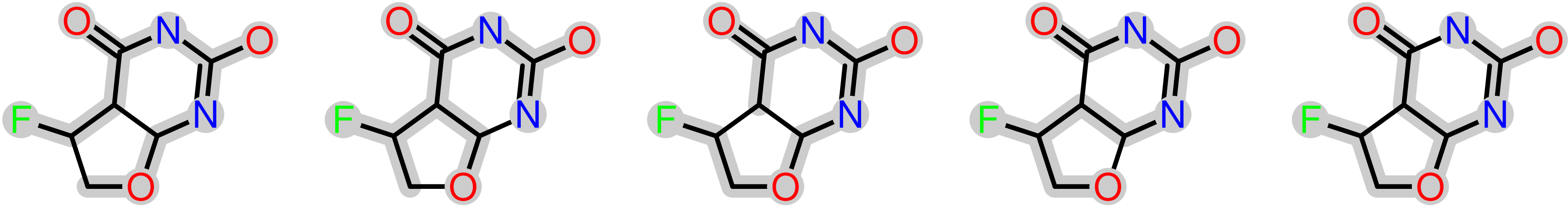
- The most common ways of constructing a spanning tree are:

○ **depth-first search**	⇒	gSpan	[Yan and Han 2002]
○ **breadth-first search**	⇒	MoSS/MoFa	[Borgelt and Berthold 2002]

An alternative way is to visit all children of a vertex before proceeding in a depth-first manner (can be seen as a variant of depth-first search). Other systematic search schemes are, in principle, also applicable.

Canonical Forms based on Spanning Trees

- Each starting point (choice of a root) and each way to build a spanning tree systematically from a given starting point yields a different code word.



There are 12 possible starting points and several branching points.

As a consequence, there are several hundred possible code words.

- The lexicographically smallest code word is the **canonical code word**.

- Since the edges are listed in the order in which they are visited during the spanning tree construction, this canonical form has the **prefix property**:

  If a prefix of a canonical code word were not canonical, there would be a starting point and a spanning tree that yield a smaller code word.
  (Use the canonical code word of the prefix graph and append the missing edge.)

Canonical Forms based on Spanning Trees

- An **edge description** consists of
  - the indices of the source and the destination vertex (definition: the source of an edge is the vertex with the smaller index),
  - the attributes of the source and the destination vertex,
  - the edge attribute.

- Listing the edges in the order in which they are visited can often be characterized by a **precedence order** on the describing elements of an edge.

- Order of individual elements (conjectures, but supported by experiments):
  - Vertex and edge attributes should be sorted according to their frequency.
  - Ascending order seems to be recommendable for the vertex attributes.

- **Simplification:** The source attribute is needed only for the first edge and thus can be split off from the list of edge descriptions.

Canonical Forms: Edge Sorting Criteria

- **Precedence Order for Depth-first Search:**
  - destination vertex index (ascending)
  - source vertex index (descending) ⇐
  - edge attribute (ascending)
  - destination vertex attribute (ascending)

- **Precedence Order for Breadth-first Search:**
  - source vertex index (ascending)
  - edge attribute (ascending)
  - destination vertex attribute (ascending)
  - destination vertex index (ascending)

- **Edges Closing Cycles:**

  Edges closing cycles may be distinguished from spanning tree edges, giving spanning tree edges absolute precedence over edges closing cycles.
  Alternative: Sort them between the other edges based on the precedence rules.

Canonical Forms: Code Words

From the described procedure the following code words result (regular expressions with non-terminal symbols):

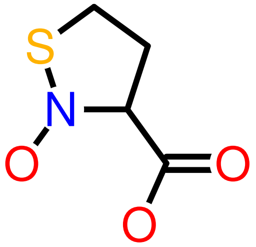
- **Depth-First Search:** $a\,(i_d\,\underline{i_s}\,b\,a)^m$
- **Breadth-First Search:** $a\,(i_s\,b\,a\,i_d)^m$ (or $a\,(i_s\,i_d\,b\,a)^m$)

where n the number of vertices of the graph,
m the number of edges of the graph,
i_s index of the source vertex of an edge, $i_s \in \{0, \ldots, n-2\}$,
i_d index of the destination vertex of an edge, $i_d \in \{1, \ldots, n-1\}$,
a the attribute of a vertex,
b the attribute of an edge.

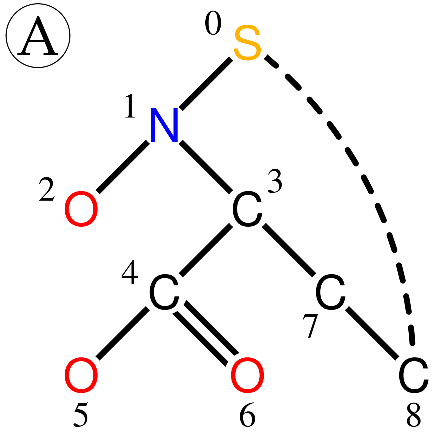
The order of the elements describing an edge reflects the precedence order.

That i_s in the depth-first search expression is underlined is meant as a reminder that the edge descriptions have to be sorted descendingly w.r.t. this value.

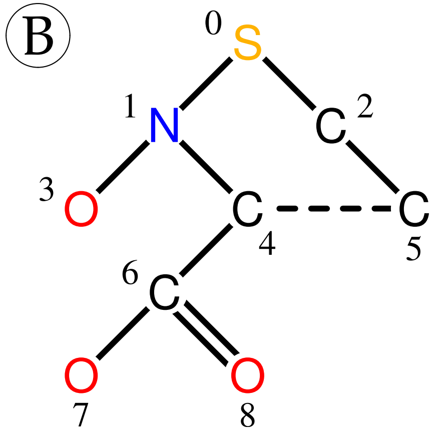
Canonical Forms: A Simple Example



example molecule



depth-first



breadth-first

Order of Elements: S $\prec$ N $\prec$ O $\prec$ C **Order of Bonds:** $- \prec =$

Code Words:

```
A:   S 10-N 21-O 31-C 43-C 54-O 64=O 73-C 87-C 80-C

B:   S 0-N1 0-C2 1-O3 1-C4 2-C5 4-C5 4-C6 6-O7 6=O8
```

(Reminder: in A the edges are sorted *descendingly* w.r.t. the second entry.)

Checking for Canonical Form: Compare Prefixes

- **Base Loop:**
  - Traverse all vertices with a label no greater than the current root vertex (first character of the code word; possible roots of spanning trees).

- **Recursive Processing:**
  - The recursive processing constructs alternative spanning trees and compares the code words resulting from it with the code word to check.
  - In each recursion step one edge is added and its description is compared to the corresponding one in the code word to check.
  - If the new edge description is **larger**, the edge can be skipped (new code word is lexicographically larger).
  - If the new edge description is **smaller**, the code word is not canonical (new code word is lexicographically smaller).
  - If the new edge description is **equal**, the suffix of the code word is processed recursively (code word prefixes are equal).

Checking for Canonical Form

```
function isCanonical ($w$: array of int, $G$: graph) : boolean;
var $v$ : vertex;                              (* to traverse the vertices of the graph *)
    $e$ : edge;                                (* to traverse the edges of the graph *)
    $x$ : array of vertex;                     (* to collect the numbered vertices *)
begin
  forall $v \in G.V$ do $v.i := -1$;           (* clear the vertex indices *)
  forall $e \in G.E$ do $e.i := -1$;           (* clear the edge markers *)
  forall $v \in G.V$ do begin                  (* traverse the potential root vertices *)
    if $v.a < w[0]$ then return false;           (* if $v$ has a smaller label, abort *)
    if $v.a = w[0]$ then begin                 (* if $v$ has the same label, check suffix *)
      $v.i := 0$; $x[0] := v$;                 (* number and record the root vertex *)
      if not rec($w$, 1, $x$, 1, 0)            (* check the code word recursively and *)
      then return false;                       (* abort if a smaller code word is found *)
      $v.i := -1$;                             (* clear the vertex index again *)
    end;
  end;
  return true;                                 (* the code word is canonical *)
end   (* isCanonical *)                        (* for a breadth-first search spanning tree *)
```

Checking for Canonical Form

```
function rec ($w$: array of int, $k$ : int, $x$: array of vertex, $n$: int, $i$: int) : boolean;
        (* $w$: code word to be tested *)
        (* $k$:  current position in code word *)
        (* $x$:  array of already labeled/numbered vertices *)
        (* $n$:  number of labeled/numbered vertices *)
        (* $i$:  index of next extendable vertex to check; $i < n$ *)
var $d$ : vertex;                              (* vertex at the other end of an edge *)
    $j$ : int;                                 (* index of destination vertex *)
    $u$ : boolean;                             (* flag for unnumbered destination vertex *)
    $r$ : boolean;                             (* buffer for a recursion result *)
begin
    if $k \geq \text{length}(w)$ return true;     (* full code word has been generated *)
    while $i < w[k]$ do begin                  (* check whether there is an edge with *)
        forall $e$ incident to $x[i]$ do       (* a source vertex having a smaller index *)
            if $e.i < 0$ then return false;
        $i := i + 1$;                          (* if there is an unmarked edge, abort, *)
    end;                                       (* otherwise go to the next vertex *)
    …
```

Checking for Canonical Form

```
  ...
  forall $e$ incident to $x[i]$ (in sorted order) do begin
    if $e.i < 0$ then begin                    (* traverse the unvisited incident edges *)
      if $e.a < w[k+1]$ then return false;  (* check the *)
      if $e.a > w[k+1]$ then return true;   (* edge attribute *)
      $d$ := vertex incident to $e$ other than $x[i]$;
      if $d.a < w[k+2]$ then return false;  (* check destination *)
      if $d.a > w[k+2]$ then return true;   (* vertex attribute *)
      if $d.i < 0$ then $j := n$ else $j := d.i$;
      if $j < w[k+3]$ then return false;    (* check destination vertex index *)

      [...]                                    (* check suffix of code word recursively, *)
                                               (* because prefixes are equal *)

    end;
  end;
  return true;                                 (* return that no smaller code word *)
end   (* rec *)                                (* than $w$ could be found *)
```

Checking for Canonical Form

```
    ...
    forall $e$ incident to $x[i]$ (in sorted order) do begin
        if $e.i < 0$ then begin            (∗ traverse the unvisited incident edges ∗)

            [...]                          (∗ check the current edge ∗)

            if $j = w[k+3]$ then begin     (∗ if edge descriptions are equal ∗)
                $e.i := 1$; $u := d.i < 0$;   (∗ mark edge and number vertex ∗)
                if $u$ then begin $d.i := j$; $x[n] := d$; $n := n+1$; end
                $r :=$ rec($w, k+4, x, n, i$);      (∗ check recursively ∗)
                if $u$ then begin $d.i := -1$; $n := n-1$; end
                $e.i := -1$;               (∗ unmark edge (and vertex) again ∗)
                if not $r$ then return false;
            end;                           (∗ evaluate the recursion result: ∗)
        end;                               (∗ abort if a smaller code word was found ∗)
    end;
    return true;                           (∗ return that no smaller code word ∗)
end    (∗ rec ∗)                           (∗ than $w$ could be found ∗)
```

Restricted Extensions

Canonical Forms: Restricted Extensions

Principle of the Search Algorithm up to now:

- Generate *all possible extensions* of a given canonical code word by the description of an edge that extends the described (sub)graph.
- Check whether an extended code word is canonical (and its (sub)graph frequent). If it is, process the extended code word recursively, otherwise discard it.

Straightforward Improvement:

- For some extensions of a given canonical code word it is easy to see that they will not be canonical themselves.
- The trick is to check whether a spanning tree **rooted at the same vertex and built in the same way** up to the extension edge yields a code word that is smaller than the created extended code word.
- This immediately rules out edges attached to certain vertices in the (sub)graph (only certain vertices are *extendable*, that is, can be incident to a new edge) as well as certain edges closing cycles.

Canonical Forms: Restricted Extensions

Depth-First Search: Rightmost Path Extension

- **Extendable Vertices:**
  - Only vertices on the **rightmost path** of the spanning tree may be extended.
  - If the source vertex of the new edge is not a leaf, the edge description must not precede the description of the downward edge on the path.

    (That is, the edge attribute must be no less than the edge attribute of the downward edge, and if it is equal, the attribute of its destination vertex must be no less than the attribute of the downward edge's destination vertex.)

- **Edges Closing Cycles:**
  - Edges closing cycles must start at an extendable vertex.
  - They must lead to the rightmost leaf (vertex at end of rightmost path).
  - The index of the source vertex must precede the index of the source vertex of any edge already incident to the rightmost leaf.

Canonical Forms: Restricted Extensions

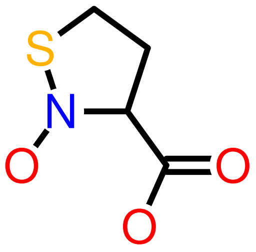
Breadth-First Search: Maximum Source Extension

- **Extendable Vertices:**
  - Only vertices having an index no less than the **maximum source index** of edges that are already in the (sub)graph may be extended.
  - If the source of the new edge is the one having the maximum source index, it may be extended only by edges whose descriptions do not precede the description of any downward edge already incident to this vertex.

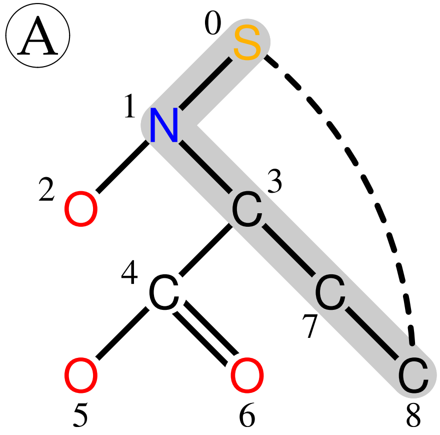
    (That is, the edge attribute must be no less, and if it is equal, the attribute of the destination vertex must be no less.)

- **Edges Closing Cycles:**
  - Edges closing cycles must start at an extendable vertex.
  - They must lead "forward", that is, to a vertex having a larger index than the extended vertex.

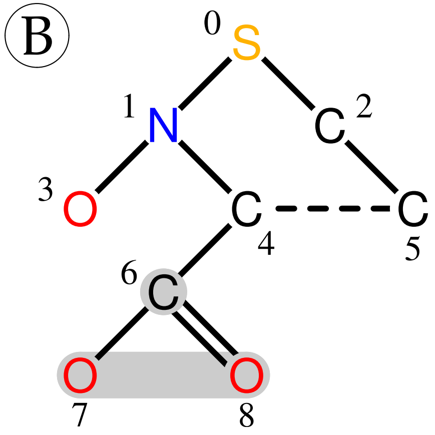
Restricted Extensions: A Simple Example



example molecule



depth-first



breadth-first

Extendable Vertices:

A: vertices on the rightmost path, that is, 0, 1, 3, 7, 8.

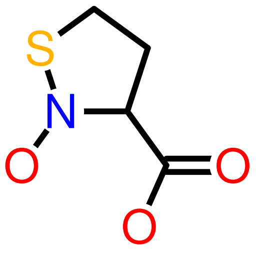
B: vertices with an index no smaller than the maximum source, that is, 6, 7, 8.

Edges Closing Cycles:

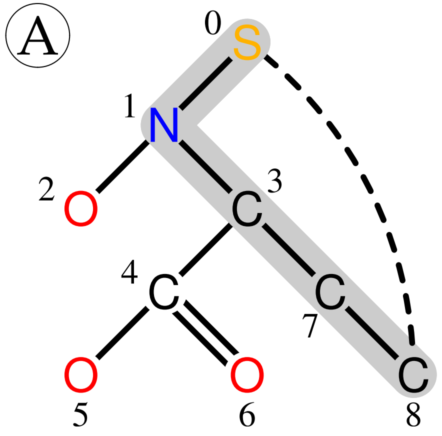
A: none, because the existing cycle edge has the smallest possible source.

B: an edge between the vertices 7 and 8.

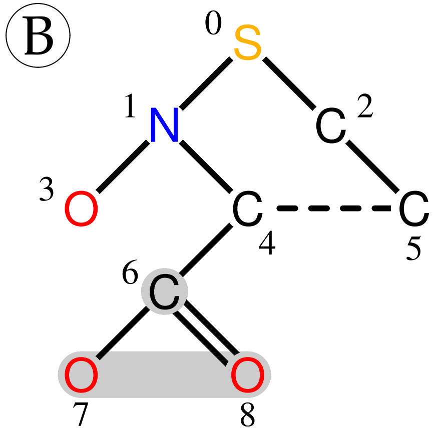
Restricted Extensions: A Simple Example



example molecule



depth-first



breadth-first

If other vertices are extended, a tree *with the same root* yields a smaller code word.

Example: attach a single bond to a carbon atom at the leftmost oxygen atom

```
A:  S 10-N 21-O 31-C 43-C 54-O 64=O 73-C 87-C 80-C 92-C
    S 10-N 21-O 32-C ···

B:  S 0-N1 0-C2 1-O3 1-C4 2-C5 4-C5 4-C6 6-O7 6=O8 3-C9
    S 0-N1 0-C2 1-O3 1-C4 2-C5 3-C6 ···
```

Canonical Forms: Restricted Extensions

- The rules underlying restricted extensions provide only a one-sided answer to the question whether an extension yields a canonical code word.

- **Depth-first search canonical form**

  - If the extension edge *is not* a rightmost path extension, then the resulting code word is *certainly* not canonical.

  - If the extension edge *is* a rightmost path extension, then the resulting code word *may or may not be* canonical.

- **Breadth-first search canonical form**

  - If the extension edge *is not* a maximum source extension, then the resulting code word is *certainly* not canonical.

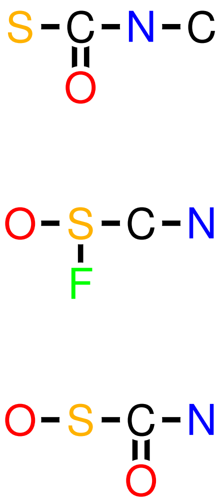
  - If the extension edge *is* a maximum source extension, then the resulting code word *may or may not be* canonical.

- As a consequence, a **canonical form test** is still necessary.

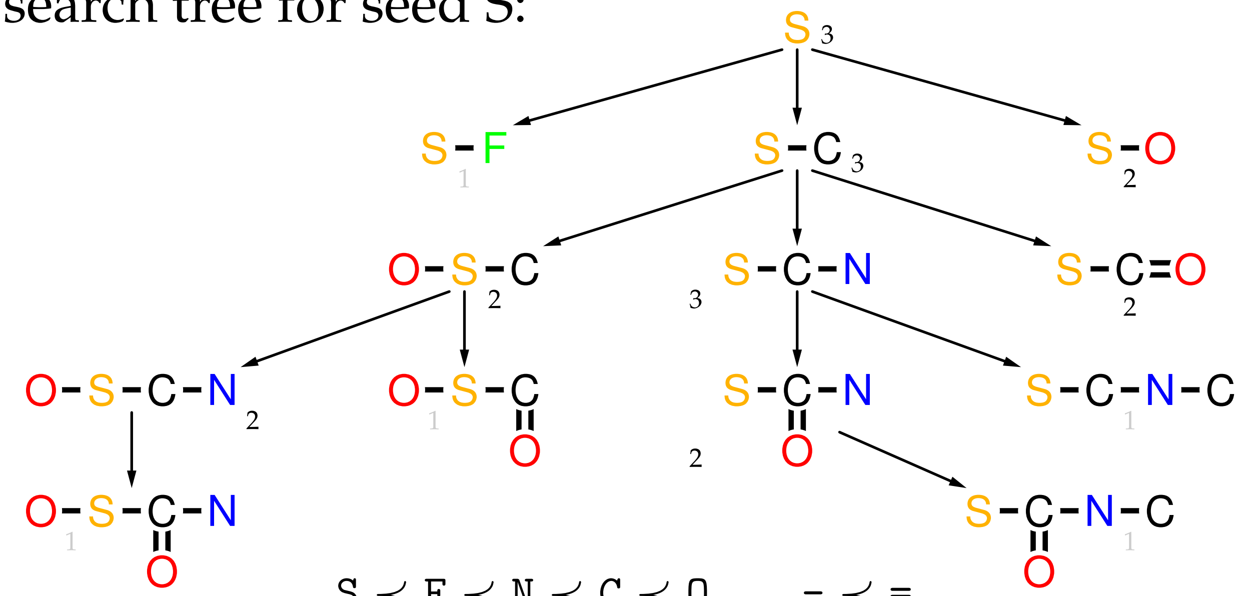
Example Search Tree

- Start with a single vertex (seed vertex).
- Add an edge (and maybe a vertex) in each step (*restricted extensions*).
- Determine the support and prune infrequent (sub)graphs.
- Check for canonical form and prune (sub)graphs with non-canonical code words.

example molecules:



search tree for seed S:



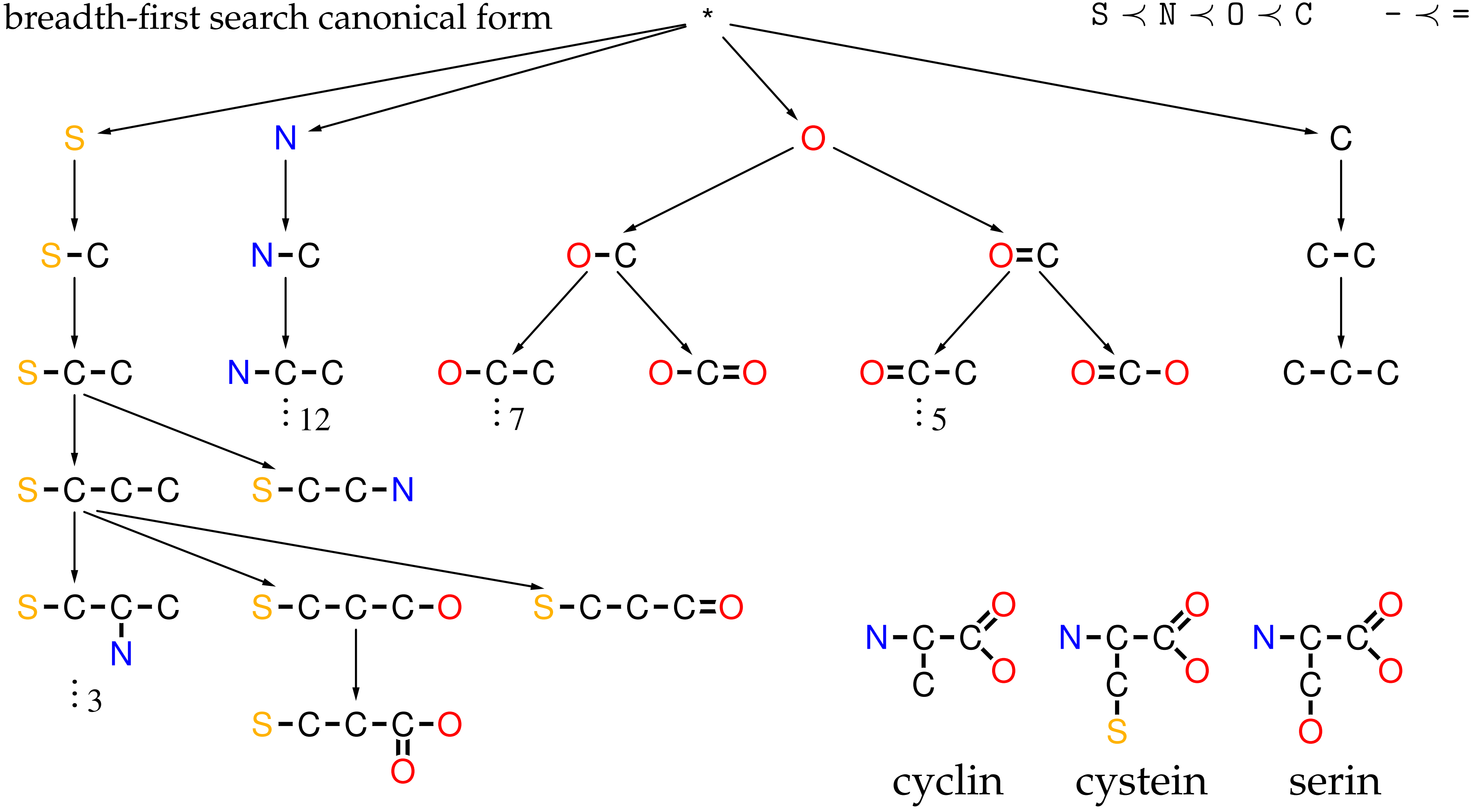
S ≺ F ≺ N ≺ C ≺ O   - ≺ =

breadth-first search canonical form

Searching without a Seed Atom

breadth-first search canonical form

$S \prec N \prec O \prec C$ $\quad - \prec =$



- Chemical elements processed on the left are excluded on the right.

Comparison of Canonical Forms

(depth-first versus breadth-first spanning tree construction)

Canonical Forms: Comparison

Depth-First vs. Breadth-First Search Canonical Form

- With breadth-first search canonical form the extendable vertices are much easier to traverse, as they always have consecutive indices:

  One only has to store and update one number, namely the index of the maximum edge source, to describe the vertex range.

- Also the check for canonical form is slightly more complex (to program; not to execute!) for depth-first search canonical form.

- The two canonical forms obviously lead to different branching factors, widths and depths of the search tree.

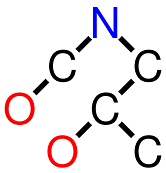
  However, it is not immediately clear, which form leads to the "better" (more efficient) structure of the search tree.

- The experimental results reported in the following indicate that it may depend on the data set which canonical form performs better.

Advantage for Maximum Source Extensions

Generate all substructures (that contain nitrogen) of the example molecule:

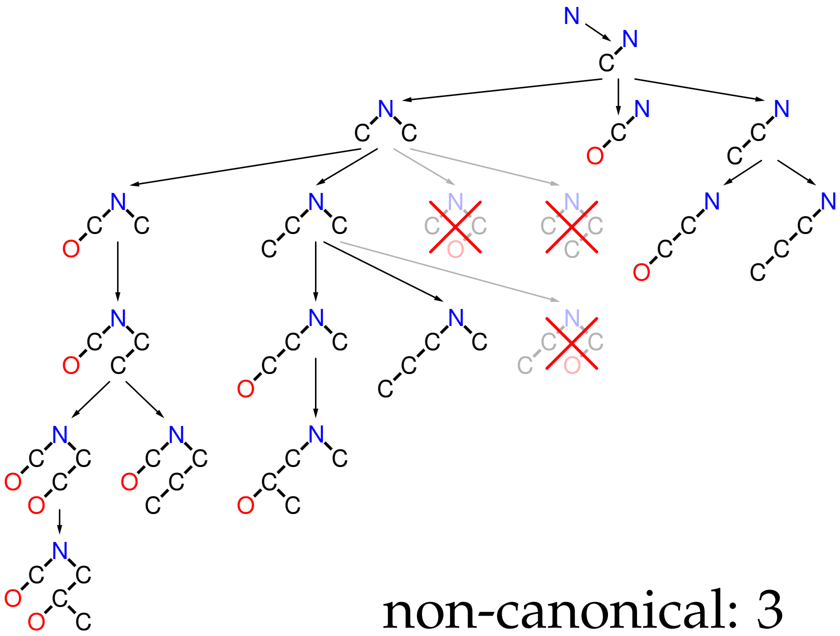
(N ≺ O ≺ C)



Problem: The two branches emanating from the nitrogen atom start identically. Thus rightmost path extensions try the right branch over and over again.

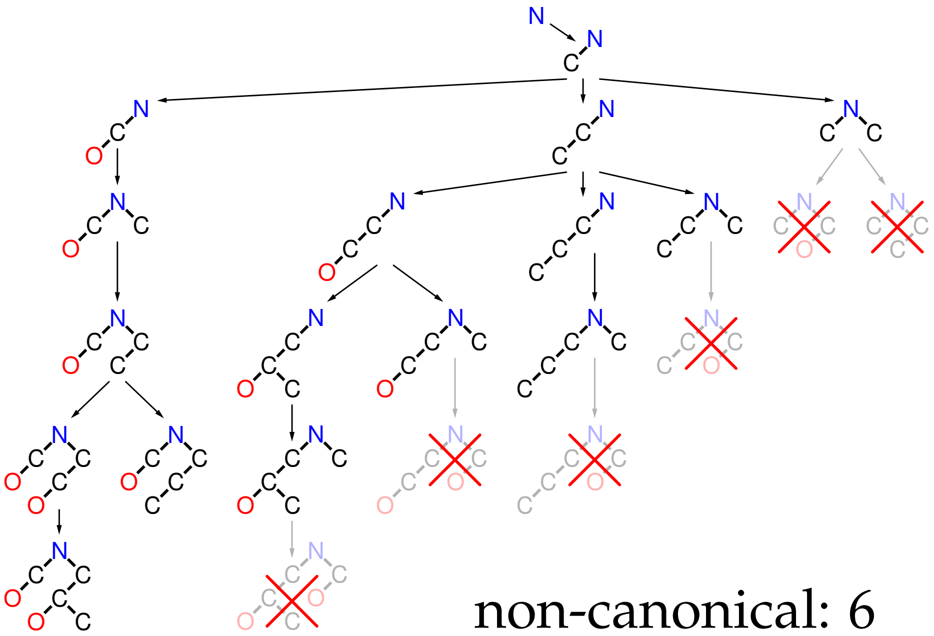
Search Trees with

Maximum Source Extension:



non-canonical: 3

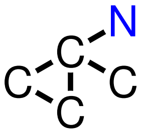
Rightmost Path Extension:



non-canonical: 6

Advantage for Rightmost Path Extensions

Generate all substructures (that contain nitrogen) of the example molecule:

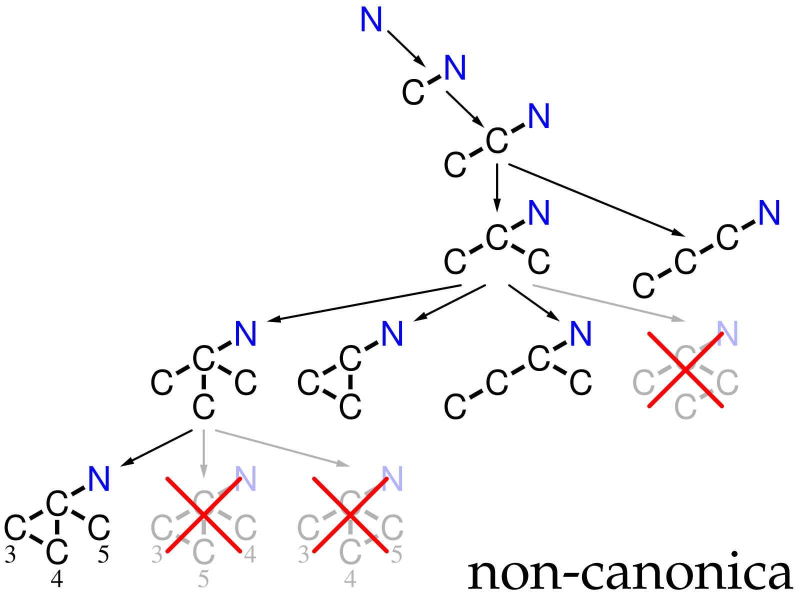


(N ≺ C)

Problem: The ring of carbon atoms can be closed between any two branches (three ways of building the fragment, only one of which is canonical).

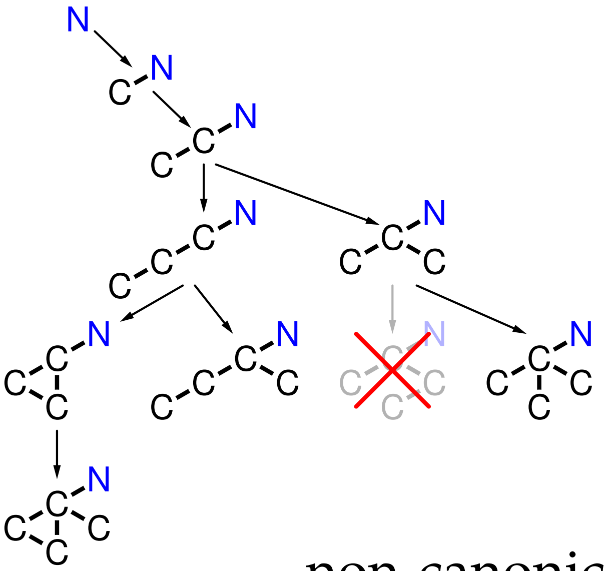
Search Trees with

Maximum Source Extension:



non-canonical: 3

Rightmost Path Extension:

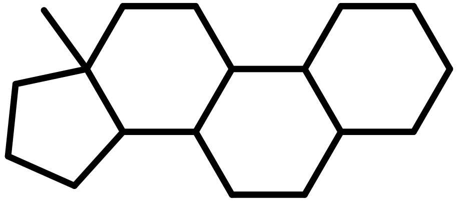


non-canonical: 1

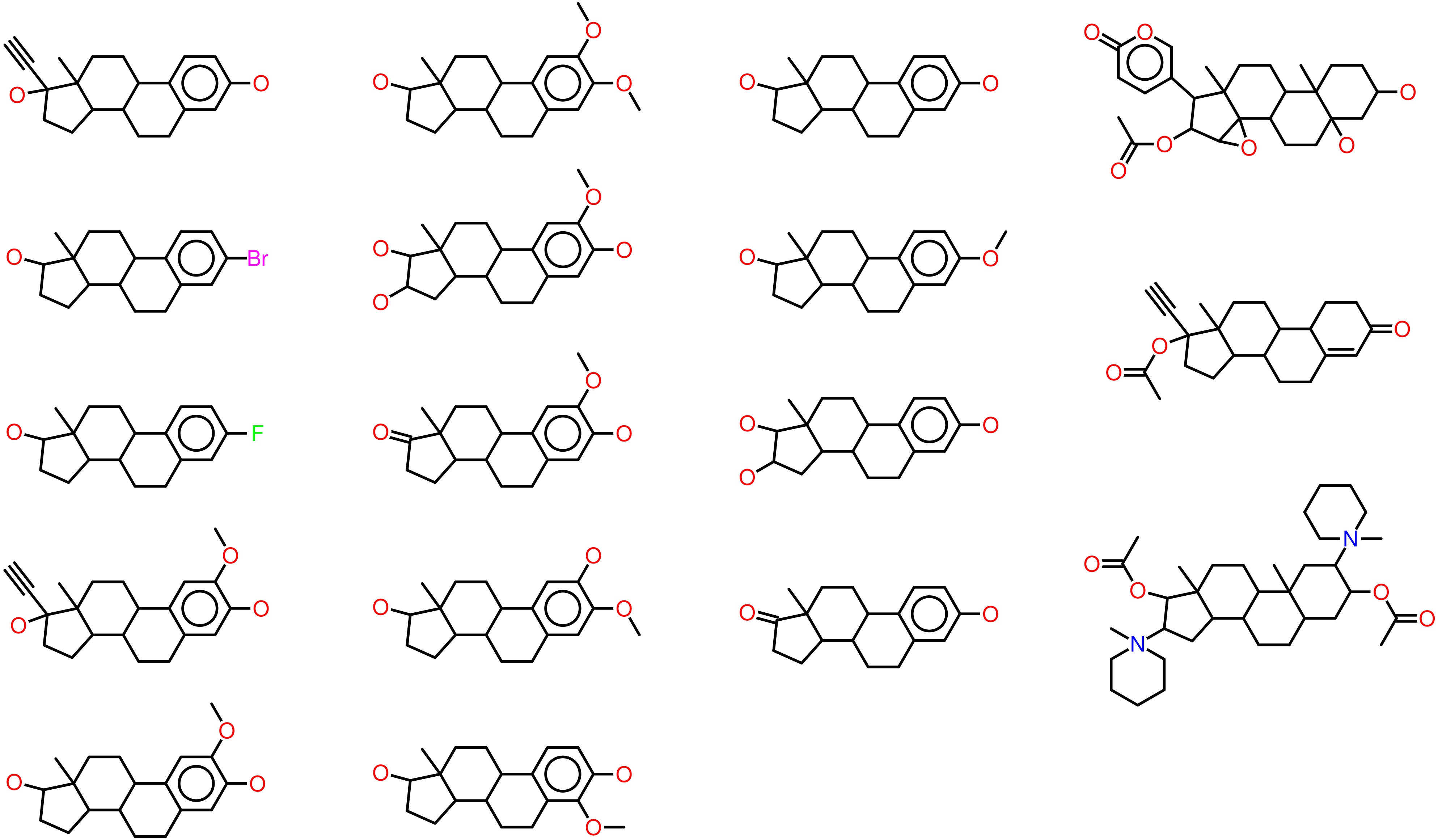
Experiments: Data Sets

- **Index Chemicus — Subset of 1993**
  - 1293 molecules / 34431 atoms / 36594 bonds
  - Frequent fragments down to fairly low support values are trees (no/few rings).
  - Medium number of fragments and closed fragments.

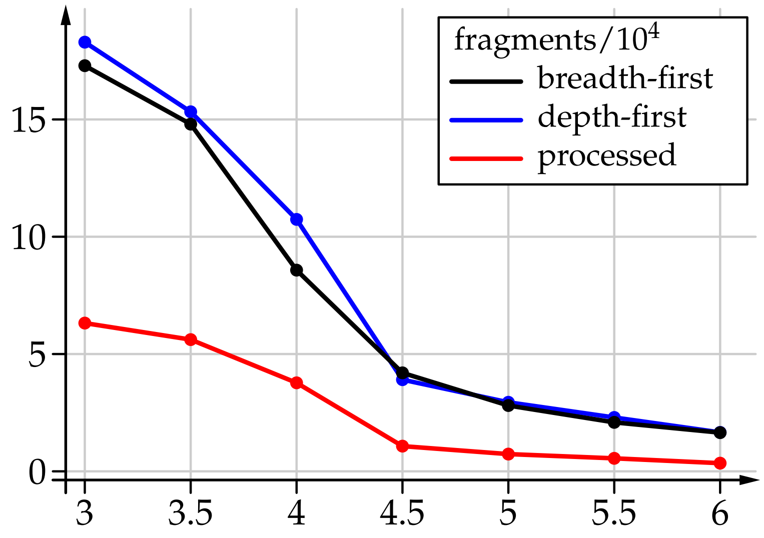
- **Steroids**
  - 17 molecules / 401 atoms / 456 bonds
  - A large part of the frequent fragments contain one or more rings.
  - Huge number of fragments, still large number of closed fragments.

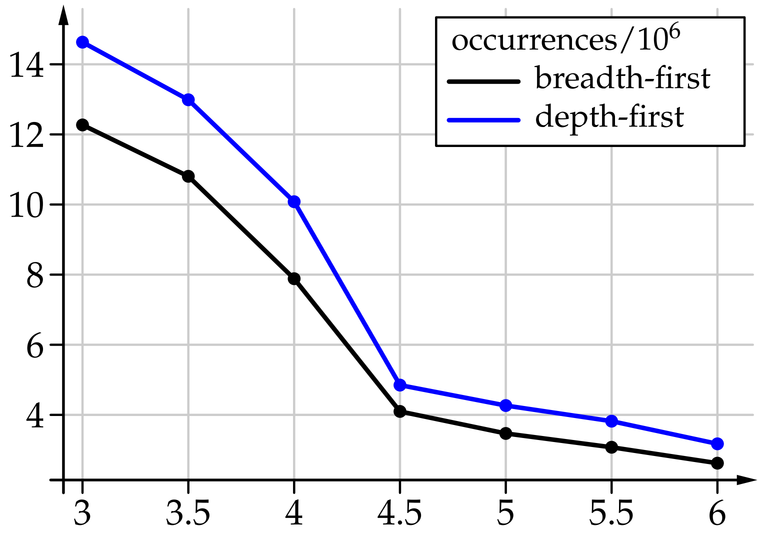


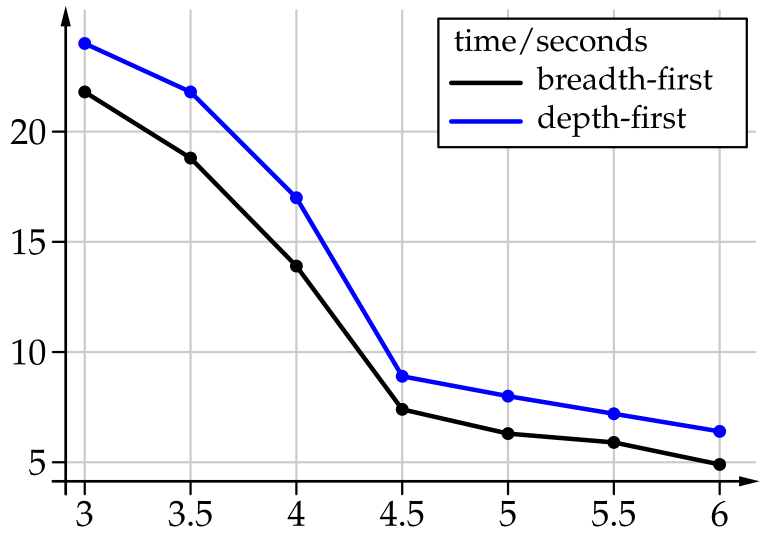
Steroids Data Set



Experiments: IC93 Data Set

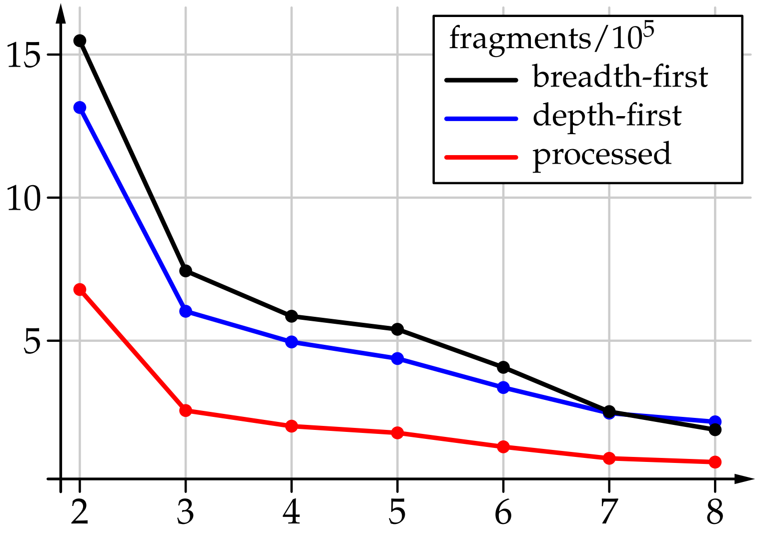


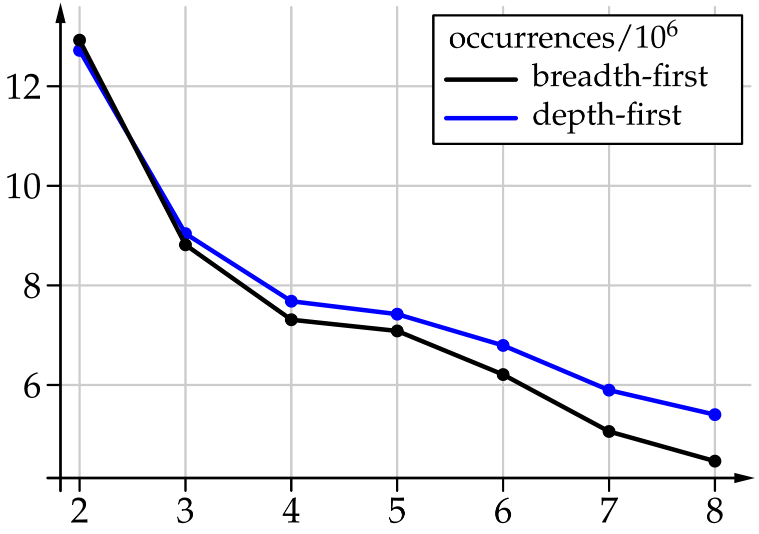


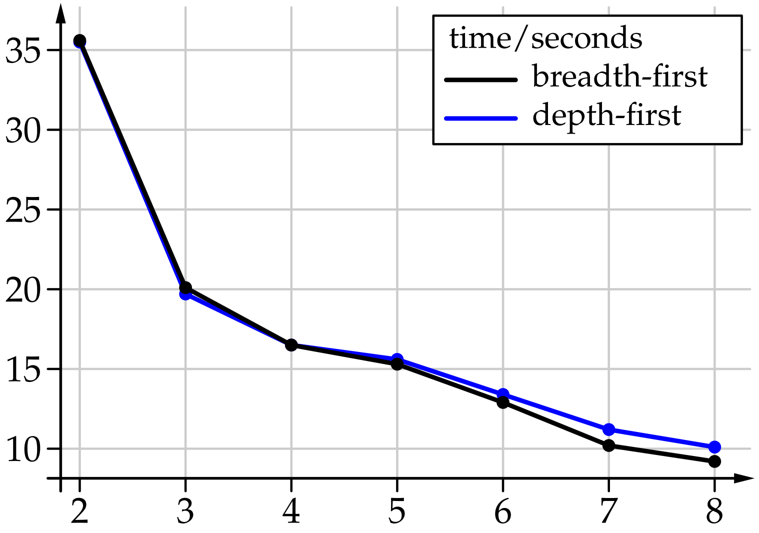


Experimental results on the IC93 data. The horizontal axis shows the minimum support in percent. The curves show the number of generated and processed fragments (top left), number of processed occurrences (top right), and the execution time in seconds (bottom left) for the two canonical forms/extension strategies.

Experiments: Steroids Data Set







Experimental results on the steroids data. The horizontal axis shows the absolute minimum support. The curves show the number of generated and processed fragments (top left), number of processed occurrences (top right), and the execution time in seconds (bottom left) for the two canonical forms/extension strategies.

Equivalent Sibling Pruning

Alternative Test: Equivalent Siblings

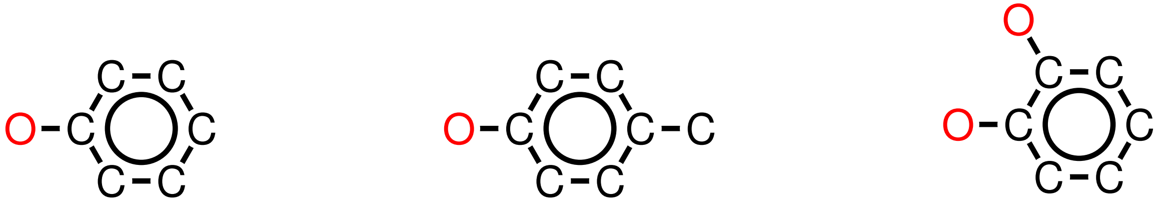
- **Basic Idea:**
  - ○ If the (sub)graph to extend exhibits a certain symmetry, several extensions may be equivalent (in the sense that they describe the same (sub)graph).
  - ○ At most one of these sibling extensions can be in canonical form, namely the one *least restricting future extensions* (lex. smallest code word).
  - ○ Identify equivalent siblings and keep only the maximally extendable one.

- **Test Procedure for Equivalence:**
  - ○ Get any graph in which both of two sibling (sub)graphs to compare occur. (If there is no such graph, the siblings are not equivalent.)
  - ○ Mark any occurrence of the first (sub)graph in the graph.
  - ○ Traverse all occurrences of the second (sub)graph in the graph
    and check whether all edges of an occurrence are marked.
    If there is such an occurrence, the two (sub)graphs are equivalent.

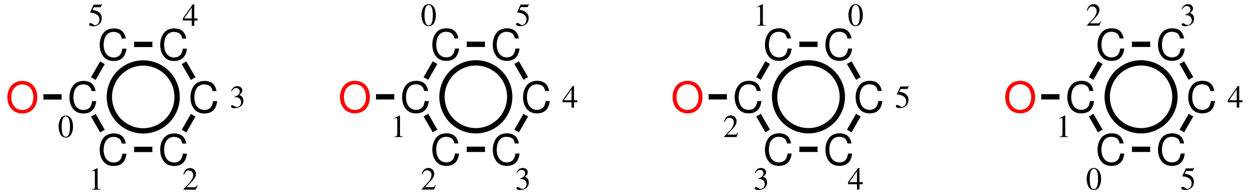
Alternative Test: Equivalent Siblings

If siblings in the search tree are equivalent,
only the one with the least restrictions needs to be processed.

Example: Mining phenol, p-cresol, and catechol.



Consider extensions of a 6-bond carbon ring (twelve possible occurrences):



Only the (sub)graph that **least restricts future extensions**
(i.e., that has the lexicographically smallest code word) can be in canonical form.

Use depth-first canonical form (rightmost path extensions) and $\mathtt{C} \prec \mathtt{O}$.

Alternative Test: Equivalent Siblings

- **Test for Equivalent Siblings before Test for Canonical Form**
  - Traverse the sibling extensions and compare each pair.
  - Of two equivalent siblings remove the one that restricts future extensions more.

- **Advantages:**
  - Identifies some code words that are non-canonical in a simple way.
  - Test of two siblings is at most linear in the number of edges and at most linear in the number of occurrences.

- **Disadvantages:**
  - Does not identify all non-canonical code words, therefore a subsequent canonical form test is still needed.
  - Compares all pairs of sibling (sub)graphs, therefore it is quadratic in the number of siblings.

Alternative Test: Equivalent Siblings

The effectiveness of equivalent sibling pruning depends on the canonical form:

Mining the **IC93 data** with 4% minimum support

	depth-first	breadth-first
equivalent sibling pruning	156 (1.9%)	4195 (83.7%)
canonical form pruning	7988 (98.1%)	815 (16.3%)
total pruning	8144	5010
(closed) (sub)graphs found	2002	2002

Mining the **steroids data** with minimum support 6

	depth-first	breadth-first
equivalent sibling pruning	15327 (7.2%)	152562 (54.6%)
canonical form pruning	197449 (92.8%)	127026 (45.4%)
total pruning	212776	279588
(closed) (sub)graphs found	1420	1420

Alternative Test: Equivalent Siblings

Observations:

- Depth-first form generates more duplicate (sub)graphs on the IC93 data and fewer duplicate (sub)graphs on the steroids data (as seen before).
- There are only very few equivalent siblings with depth-first form on both the IC93 data and the steroids data.

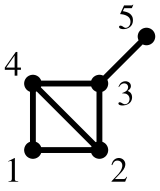
  (Conjecture: equivalent siblings result from “rotated” tree branches, which are less likely to be siblings with depth-first form.)
- With breadth-first search canonical form a large part of the (sub)graphs that are not generated in canonical form (with a canonical code word) can be filtered out with equivalent sibling pruning.
- On the IC93 data no difference in speed could be observed, presumably because pruning takes only a small part of the total time.
- On the steroids data, however, equivalent sibling pruning yields a slight speed-up for breadth-first form ($\sim 5\%$).

Canonical Forms based on Adjacency Matrices

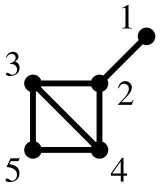
Adjacency Matrices

- A (normal, that is, unlabeled) graph can be described by an **adjacency matrix**:
  - A graph G with n vertices is described by an $n \times n$ matrix $\mathbf{A} = (a_{ij})$.
  - Given a numbering of the vertices (from 1 to n), each vertex is associated with the row and column corresponding to its number.
  - A matrix element a_{ij} is 1 if there exists an edge between the vertices with numbers i and j and it is 0 otherwise.

- Adjacency matrices are *not* unique:
  Different numberings of the vertices lead to different adjacency matrices.



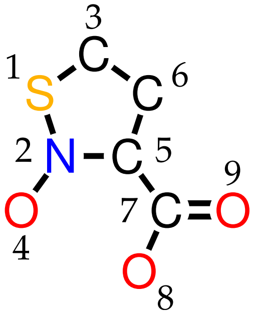
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	0
5	0	0	1	0	0



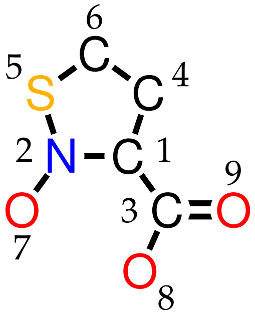
	1	2	3	4	5
1	0	1	0	0	0
2	1	0	1	1	0
3	0	1	0	1	1
4	0	1	1	0	1
5	0	0	1	1	0

Extended Adjacency Matrices

- A labeled graph can be described by an **extended adjacency matrix**:
  - If there is an edge between the vertices with numbers i and j the matrix element a_{ij} contains the label of this edge and the special label $\times$ (the empty label) otherwise.
  - There is an additional column containing the vertex labels.
- Of course, extended adjacency matrices are also *not* unique:



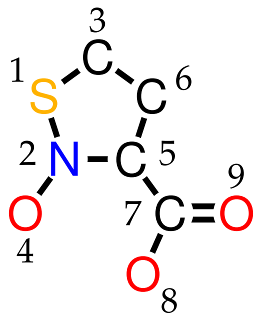
		1	2	3	4	5	6	7	8	9
1	S	×	−	−	×	×	×	×	×	×
2	N	−	×	×	−	−	×	×	×	×
3	C	−	×	×	×	×	−	×	×	×
4	O	×	−	×	×	×	×	×	×	×
5	C	×	−	×	×	×	−	−	×	×
6	C	×	×	−	×	−	×	×	×	×
7	C	×	×	×	×	−	×	×	−	=
8	O	×	×	×	×	×	×	−	×	×
9	O	×	×	×	×	×	×	=	×	×



		1	2	3	4	5	6	7	8	9
1	C	×	−	−	−	×	×	×	×	×
2	N	−	×	×	×	−	×	−	×	×
3	C	−	×	×	×	×	×	×	−	=
4	C	−	×	×	×	×	−	×	×	×
5	S	×	−	×	×	×	−	×	×	×
6	C	×	×	×	−	−	×	×	×	×
7	O	×	−	×	×	×	×	×	×	×
8	O	×	×	−	×	×	×	×	×	×
9	O	×	×	=	×	×	×	×	×	×

From Adjacency Matrices to Code Words

- An (extended) adjacency matrix can be turned into a **code word** by simply listing its elements row by row.
- Since for undirected graphs the adjacency matrix is necessarily symmetric, it suffices to list the elements of the upper (or lower) triangle.
- For sparse graphs (few edges) listing only column/label pairs can be advantageous, because this reduces the code word length.



Regular expression (non-terminals):
$(a\ (\ i_c\ b\ )^*\ )^n$

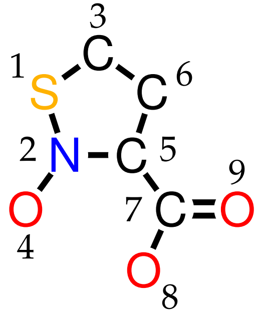
		1	2	3	4	5	6	7	8	9
1	S	×	–	–	×	×	×	×	×	×
2	N	–	×	×	–	–	×	×	×	×
3	C	–	×	×	×	×	–	×	×	×
4	O	×	–	×	×	×	×	×	×	×
5	C	×	–	×	×	×	–	–	×	×
6	C	×	×	–	×	–	×	×	×	×
7	C	×	×	×	×	–	×	×	–	=
8	O	×	×	×	×	×	×	–	×	×
9	O	×	×	×	×	×	×	=	×	×

code word:

```
S 2- 3-
N 4- 5-
C 6-
O
C 6- 7-
C
C 8- 9=
O
O
```

From Adjacency Matrices to Code Words

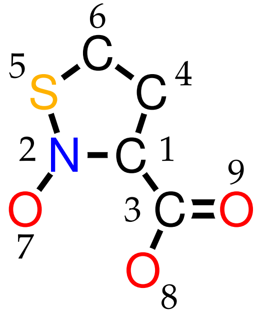
- With an (arbitrary, but fixed) order on the label set A (and defining that integer numbers, which are ordered in the usual way, precede all labels), code words can be compared lexicographically: (`S` $\prec$ `N` $\prec$ `O` $\prec$ `C` ; `-` $\prec$ `=`)



`S 2-3-N 4-5-C 6-O C 6-7-C C 8-9=O O`

$<$

`C 2-3-4-N 5-7-C 8-9=C 6-S 6-C O O O`



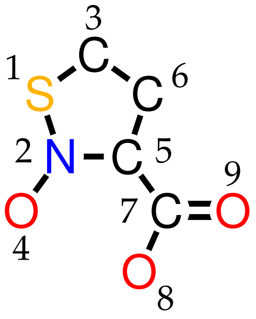
- As for canonical forms based on spanning trees, we then define the lexicographically smallest (or largest) code word as the **canonical code word**.

- Note that adjacency matrices allow for a *much larger number of code words*, because any numbering of the vertices is admissible.

  For canonical forms based on spanning trees, the vertex numbering must be compatible with a (specific) construction of a spanning tree.

From Adjacency Matrices to Code Words

- There are many ways of turning an adjacency matrix into a code word:



		1	2	3	4	5	6	7	8	9
1	S	×	–	–	×	×	×	×	×	×
2	N	–	×	×	–	–	×	×	×	×
3	C	–	×	×	×	×	–	×	×	×
4	O	×	–	×	×	×	×	×	×	×
5	C	×	–	×	×	×	–	–	×	×
6	C	×	×	–	×	–	×	×	×	×
7	C	×	×	×	×	–	×	×	–	=
8	O	×	×	×	×	×	×	–	×	×
9	O	×	×	×	×	×	×	=	×	×

```
lower triangle:    columnwise:
S                  S N C O C C C O O
N 1 -              | 1 -
C 1 -              | 1 -
O 2 -              | 2 -
C 2 -              | 2 -
C 3 -  5 -         | 3 -  5 -
C 5 -              | 5 -
O 6 -  7 -         | 7 -
O 7 =              | 7 =
```

(Note that the columnwise listing needs a separator character "|".)

- However, the rowwise listing restricted to the upper triangle (as used before) has the advantage that it has a property analogous to the **prefix property**.

  If the destination vertex label is added to the matrix elements, it is even equivalent to breadth-first search spanning tree canonical form.

  In contrast to this, the two forms shown above do not have this property.

Exploiting Vertex Signatures / Fingerprints

Canonical Form and Vertex and Edge Labels

- Vertex and edge labels help considerably to construct a canonical code word or to check whether a given code word is canonical:

  Canonical form check or construction are usually (much) slower/more difficult for unlabeled graphs or graphs with few different vertex and edge labels.

- The reason is that with vertex and edge labels constructed code word prefixes may already allow us to make a decision between (sets of) code words.

- Intuitive explanation with an extreme example:

  Suppose that all vertices of a given (sub)graph have different labels. Then:

  - The root/first row vertex is uniquely determined: it is the vertex with the smallest label (w.r.t. the chosen order).
  - The order of each vertex's neighbors in the canonical form is determined at least by the vertex labels (but maybe also by the edge labels).
  - As a consequence, constructing the canonical code word is straightforward.

Canonical Form and Vertex and Edge Labels

- The complexity of constructing a canonical code word is caused by equal edge and vertex labels, which make it necessary to apply a **backtracking** algorithm.

- **Question:** Can we exploit graph properties (that is, the connection structure) to distinguish vertices/edges with the same label?

- **Idea:** Describe how the (sub)graph under consideration "looks from a vertex".

  This can be achieved by constructing a "local code word" (**vertex signature**):

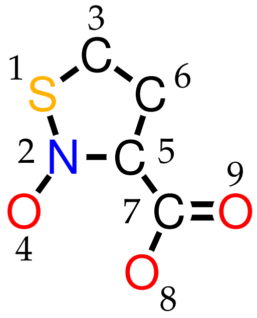
  - Start with the label of the vertex.
  - If there is more than one vertex with a certain label, add a (sorted) list of the labels of the incident edges.
  - If there is more than one vertex with the same list, add a (sorted) list of the lists of the adjacent vertices.
  - Continue with the vertices that are two edges away and so on.

Constructing Vertex Signatures / Fingerprints

The process of constructing vertex signatures is best described as an **iterative subdivision of equivalence classes**:

- The initial signature of each vertex is simply its label.
- The vertex set is split into equivalence classes based on the initial vertex signature (that is, the vertex labels).
- Equivalence classes with more than one vertex are then processed by appending the (sorted) labels of the incident edges to the vertex signature. The vertex set is then repartitioned based on the extended vertex signature.
- In a second step the (sorted) signatures of the adjacent vertices are appended.
- In subsequent steps these signatures of adjacent vertices are replaced by the updated vertex signatures.
- The process stops when no replacement splits an equivalence class.

Constructing Vertex Signatures / Fingerprints

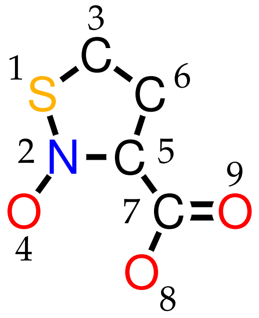


vertex	signature
1	S
2	N
4	O
8	O
9	O
3	C
6	C
5	C
7	C

Vertex Signatures / Fingerprints, Step 1

- The initial vertex signatures are simply the vertex labels.
- There are four equivalence classes: S, N, O, and C.
- The equivalence classes S and N need not further processing, because they already contain only a single vertex.
- However, the vertex signatures O and C need to be extended in order to split the corresponding equivalence classes.

Constructing Vertex Signatures / Fingerprints

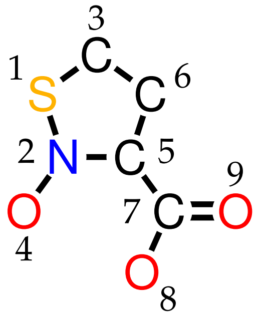


vertex	signature
1	S
2	N
4	O -
8	O -
9	O =
3	C --
6	C --
5	C ---
7	C --=

Vertex Signatures / Fingerprints, Step 2

- The vertex signatures of the classes that contain more than one vertex are extended by the sorted list of labels of the incident edges.
- This distinguishes the three oxygen atoms, because two is incident to a single bond, the other to a double bond.
- It also distinguishes most carbon atoms, because they have different sets of incident edges.
- Only the signatures of carbons 3 and 6 and the signatures of oxygens 4 and 9 need to be extended further.

Constructing Vertex Signatures / Fingerprints



vertex	signature
1	S
2	N
4	O - N
8	O - C --=
9	O =
3	C -- S C --
6	C -- C -- C ---
5	C ---
7	C --=

Vertex Signatures / Fingerprints, Step 3

- The vertex signatures of carbons 3 and 6 and of oxygens 4 and 9 are extended by the sorted list of vertex signatures of the adjacent vertices.

- This distinguishes the two pairs (carbon 3 is adjacent to a sulfur atom, oxygen 4 is adjacent to a nitrogen atom).

- As a result, all equivalence classes contain only a single vertex and thus we obtained a unique vertex labeling.

- With this unique vertex labeling, constructing a canonical code word becomes very simple and efficient.

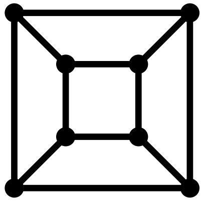
Weisfeiler–Lehman Isomorphism Test

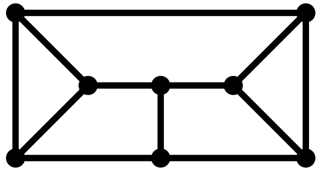
- Constructing vertex signatures / fingerprints can help testing whether two given (labeled) graphs are isomorphic:
  - Construct the vertex signatures / fingerprints / equivalence classes of the two given graphs that are suspected to be isomorphic.
  - If the vertex signatures / fingerprints / equivalence classes differ (even if at an intermediate step already, so early stopping is possible) the two graphs cannot be isomorphic.
  - A simple sorted list of its vertex signatures / fingerprints can be used as a code word of each graph:
    Two graphs can only be isomorphic if their code words coincide.
- This method is called the **Weisfeiler–Lehman Isomorphism Test**.
- It is an imperfect or one-sided test:
  - If the code words differ, the graphs cannot possibly be isomorphic.
  - If the code words coincide, the graphs may or may not be isomorphic.

Elements of Vertex Signatures / Fingerprints

- Using only (sorted) lists of labels of incident edges and adjacent vertices cannot always distinguish all vertices.

  Example: For the following two (unlabeled) graphs such vertex signatures cannot split the sole equivalence class:





- The equivalence class can be split for the right graph, though, if the number of adjacent vertices that are adjacent is incorporated into the vertex signature. There is also a large variety of other graph properties that may be used.

- For neither graph the equivalence classes can be reduced to single vertices. For the left graph it is not even possible at all to split the equivalence class.

- The reason is that both graphs possess **automorphisms** other then the identity.

Automorphism Groups

- Let $F_{\text{auto}}(G)$ be the set of all **automorphisms** of a (labeled) graph G.

  The **orbit** of a vertex $v \in V_G$ w.r.t. $F_{\text{auto}}(G)$ is the set

  $$o(v) = \{u \in V_G \mid \exists f \in F_{\text{auto}}(G) : u = f(v)\}.$$

  Note that it is always $v \in o(v)$, because the identity is always in $F_{\text{auto}}(G)$.

- The vertices in an orbit cannot possibly be distinguished by vertex signatures, because the graph "looks the same" from all of them.

- In order to deal with orbits, one can exploit that the automorphisms $F_{\text{auto}}(G)$ of a graph G form a **group** (the **automorphism group** of G):

  - During the construction of a canonical code word,
    detect automorphisms (vertex numberings leading to the same code word).
  - From found automorphisms, **generators** of the group of automorphisms can be derived. These generators can then be used to avoid exploring implied automorphisms, thus speeding up the search. [McKay 1981]

Canonical Form and Vertex Signatures

- **Advantages of Vertex Signatures:**
  - Vertices with the same label can be distinguished in a preprocessing step.
  - Constructing canonical code words can thus become much easier/faster, because the necessary backtracking can often be reduced considerably.

    (The gains are usually particularly large for graphs with few/no labels.)

- **Disadvantages of Vertex Signatures:**
  - Vertex signatures can refer to the graph as a whole and thus may be different for subgraphs.

    (Vertices with different signatures in a subgraph may have the same signature in a supergraph and vice versa.)
  - As a consequence it can be difficult to ensure that the resulting canonical form has the **prefix property**.

    In such a case one may not be able to restrict (sub)graph extensions or to use the simplified search scheme (only code word checks).

Repository of Processed Fragments

Repository of Processed Fragments

- **Canonical form pruning** is the predominant method to avoid redundant search in frequent (sub)graph mining.

- The obvious alternative, a **repository of processed (sub)graphs**, has received fairly little attention. [Borgelt and Fiedler 2007]
  - Whenever a new (sub)graph is created, the repository is accessed.
  - If it contains the (sub)graph, we know that it has already been processed and therefore it can be discarded.
  - Only (sub)graphs that are not contained in the repository are extended and, of course, inserted into the repository.

- If the repository is laid out as a hash table with a carefully designed hash function, it is competitive with canonical form pruning.

  (In some experiments, the repository-based approach could outperform canonical form pruning by 15%.)

Repository of Processed Fragments

- Each (sub)graph should be stored using a minimal amount of memory (since the number of processed (sub)graphs is usually huge).
  - ○ Store a (sub)graph by listing the edges of one occurrence. (Note that for connected (sub)graphs the edges also identify all vertices.)
- The containment test has to be made as fast as possible (since it will be carried out frequently).
  - ○ Try to avoid a full isomorphism test with a hash table: Employ a hash function that is computed from local graph properties. (Basic idea: combine the vertex and edge attributes and the vertex degrees.)
  - ○ If an isomorphism test is necessary, do quick checks first: number of vertices, number of edges, first containing database graph etc.
  - ○ Actual isomorphism test: mark stored occurrence and check for fully marked new occurrence (cf. the procedure of equivalent sibling pruning).

Canonical Form Pruning versus Repository

- **Advantage of Canonical Form Pruning**

  Only one test (for canonical form) is needed in order to determine whether a (sub)graph needs to be processed or not.

- **Disadvantage of Canonical Form Pruning**

  It is most costly for the (sub)graphs that are created in canonical form.
  (→ slowest for fragments that have to be processed)

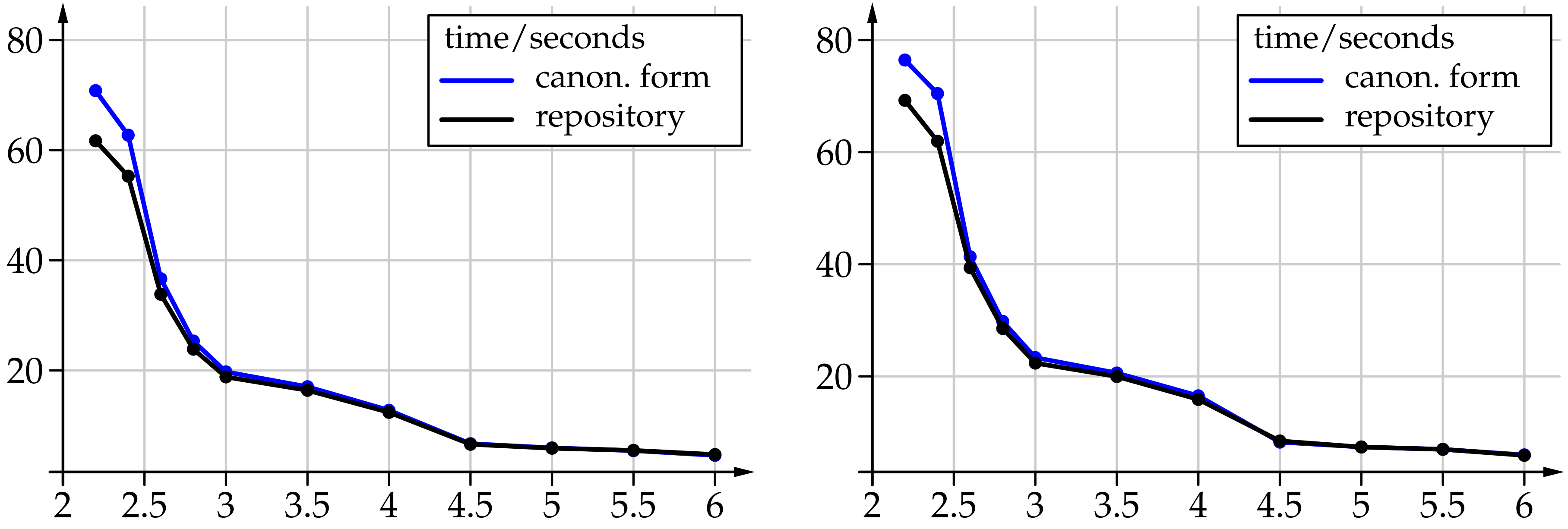
- **Advantage of Repository-based Pruning**

  Often allows to decide very quickly that a (sub)graph has not been processed.
  (→ often fastest for fragments that have to be processed)

- **Disadvantages of Repository-based Pruning**

  Multiple isomorphism tests may be necessary for a processed fragment.
  Needs far more memory than canonical form pruning.
  A repository is very difficult to use in a parallelized algorithm.

Canonical Form vs. Repository: Execution Times

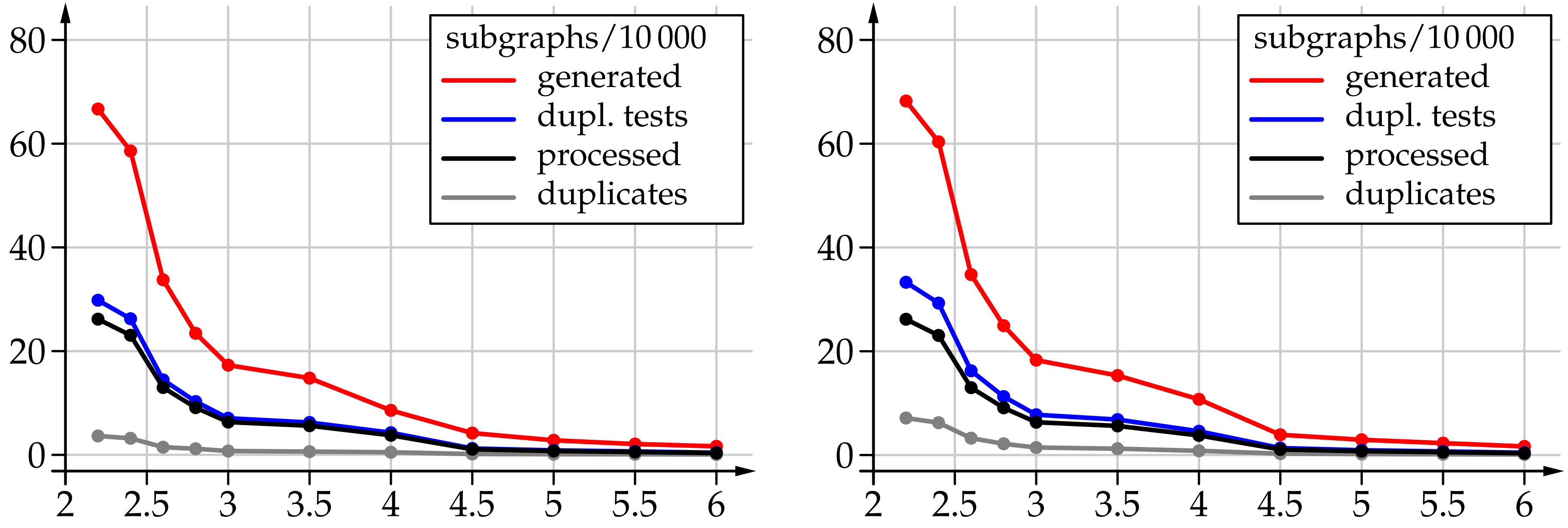


- Experimental results on the IC93 data set, search time in seconds (vertical axis) versus minimum support in percent (horizontal axis).

- Left: maximum source extensions

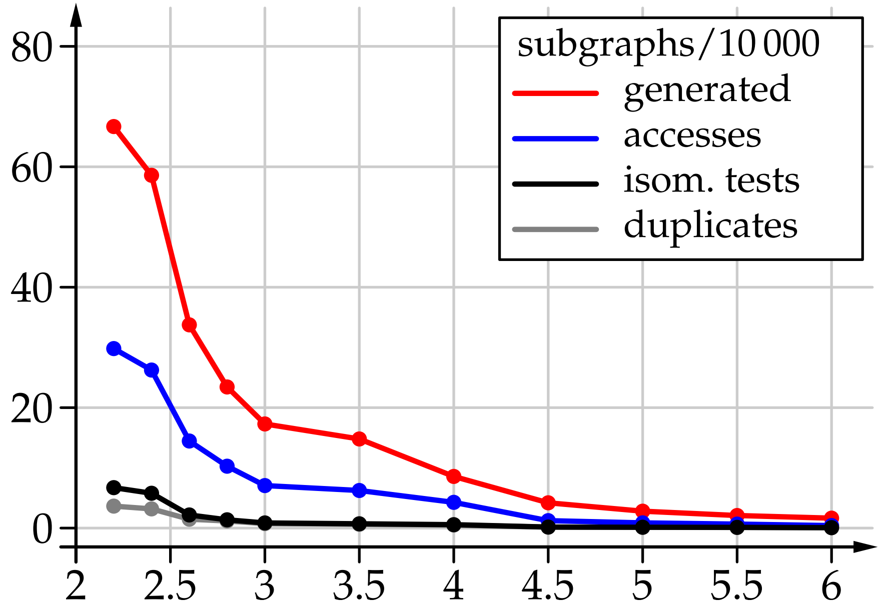
- Right: rightmost path extensions

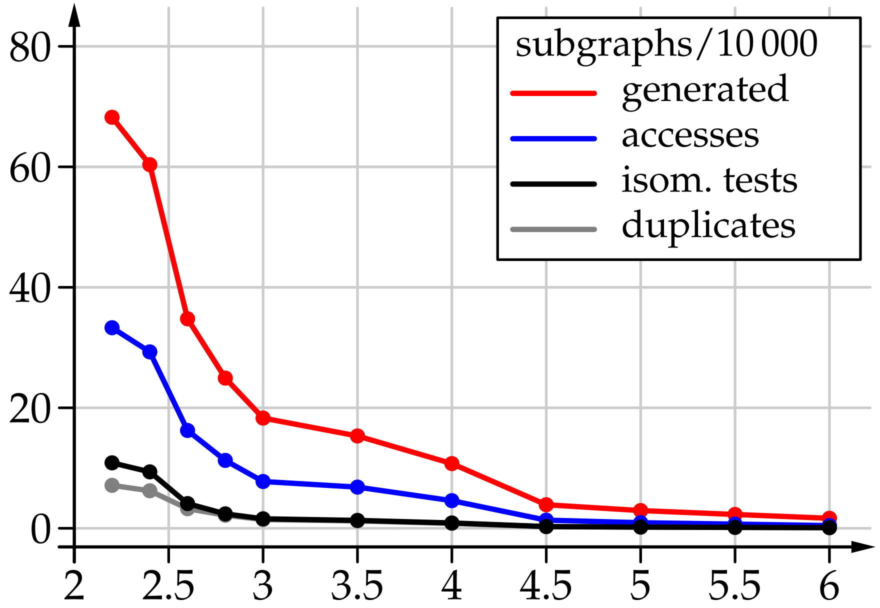
Canonical Form vs. Repository: Numbers of (Sub)Graphs



- Experimental results on the IC93 data set, numbers of subgraphs used in the search.
- Left: maximum source extensions
- Right: rightmost path extensions

Repository Performance





- Experimental results on the IC93 data set, performance of repository-based pruning.

- Left: maximum source extensions

- Right: rightmost path extensions

Perfect Extension Pruning

Reminder: Perfect Extension Pruning for Item Sets

- If only closed item sets or only maximal item sets are to be found, additional pruning of the search tree becomes possible.

- Suppose that during the search we discover that

  $$s_T(I \cup \{i\}) = s_T(I)$$

  for some item set I and some item $i \notin I$. (That is, I is not closed.) We call the item i a **perfect extension** of I. Then we know that

  $$\forall J \supseteq I: \quad s_T(J \cup \{i\}) = s_T(J).$$

  This can most easily be seen by considering that $K_T(I) \subseteq K_T(\{i\})$ and hence $K_T(J) \subseteq K_T(\{i\})$, since $K_T(J) \subseteq K_T(I)$.

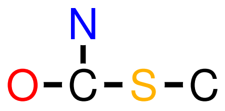
- As a consequence, no superset $J \supseteq I$ with $i \notin J$ can be closed. Hence i can be added directly to the prefix of the conditional database.

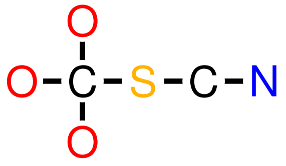
The same basic idea can also be used for graphs, but needs modifications.

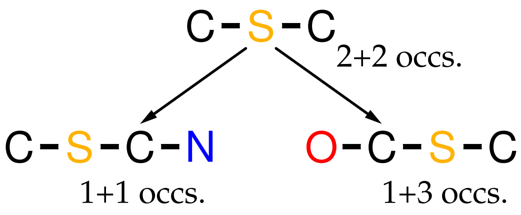
Perfect Extensions

- An extension of a graph (fragment) is called **perfect**,
  if it can be applied to all of its occurrences in exactly the same way.

- **Attention:** It may not be enough to compare the support and
  the number of occurrences of the graph fragment (necessary, but not sufficient).
  (Even though perfect extensions must have the same support and
  an integer multiple of the number of occurrences of the base fragment.)







Neither is a single bond to nitrogen a perfect extension of `O-C-S-C`
nor is a single bond to oxygen a perfect extension of `N-C-S-C`.

However, we need that a perfect extension of a graph fragment
is also a perfect extension of any supergraph of this fragment.

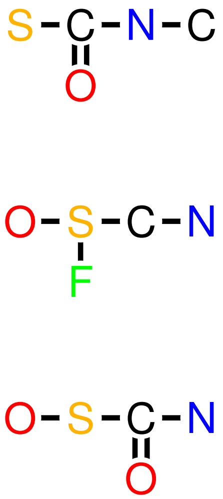
- **Consequence:** It may be necessary to check whether all occurrences
  of the base fragment lead to the same number of extended occurrences.

Partial Perfect Extension Pruning

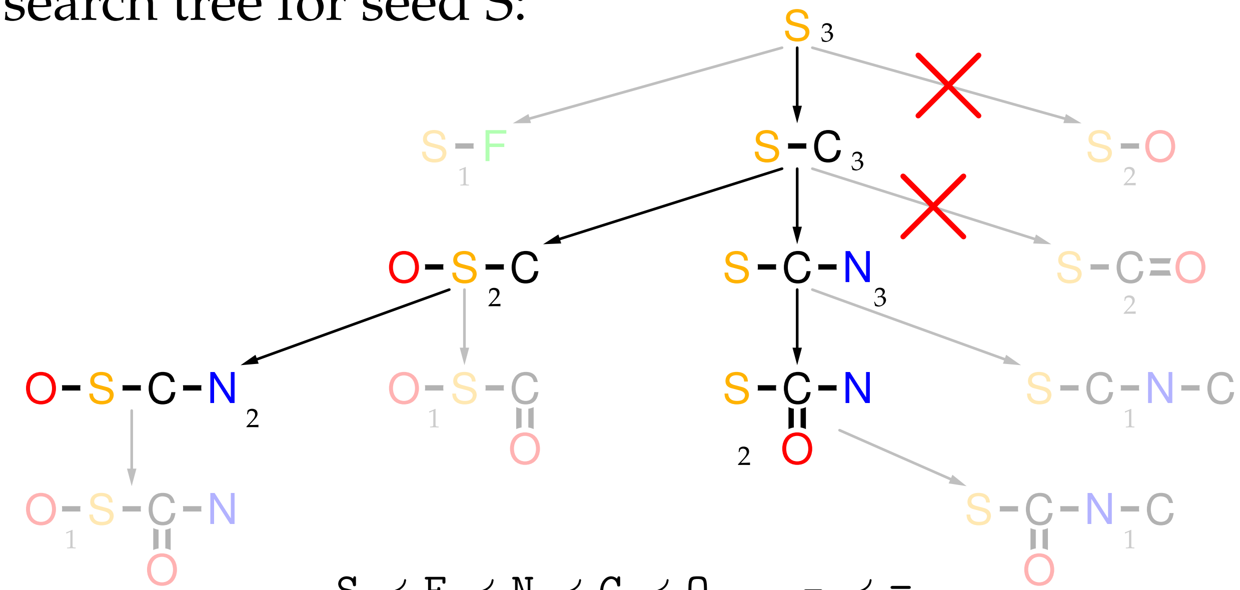
- **Basic idea of perfect extension pruning:**
  First grow a fragment to the biggest common substructure.

- **Partial perfect extension pruning:** If the children of a search tree vertex are ordered lexicographically (w.r.t. their code word), no fragment in a subtree to the right of a perfect extension branch can be closed. [Yan and Han 2003]

example molecules:



search tree for seed S:



S ≺ F ≺ N ≺ C ≺ O   - ≺ =

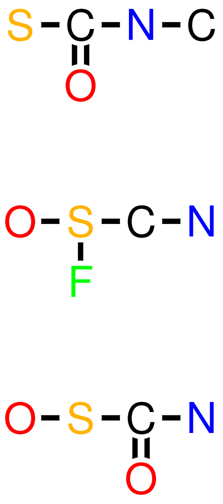
breadth-first search canonical form

Full Perfect Extension Pruning

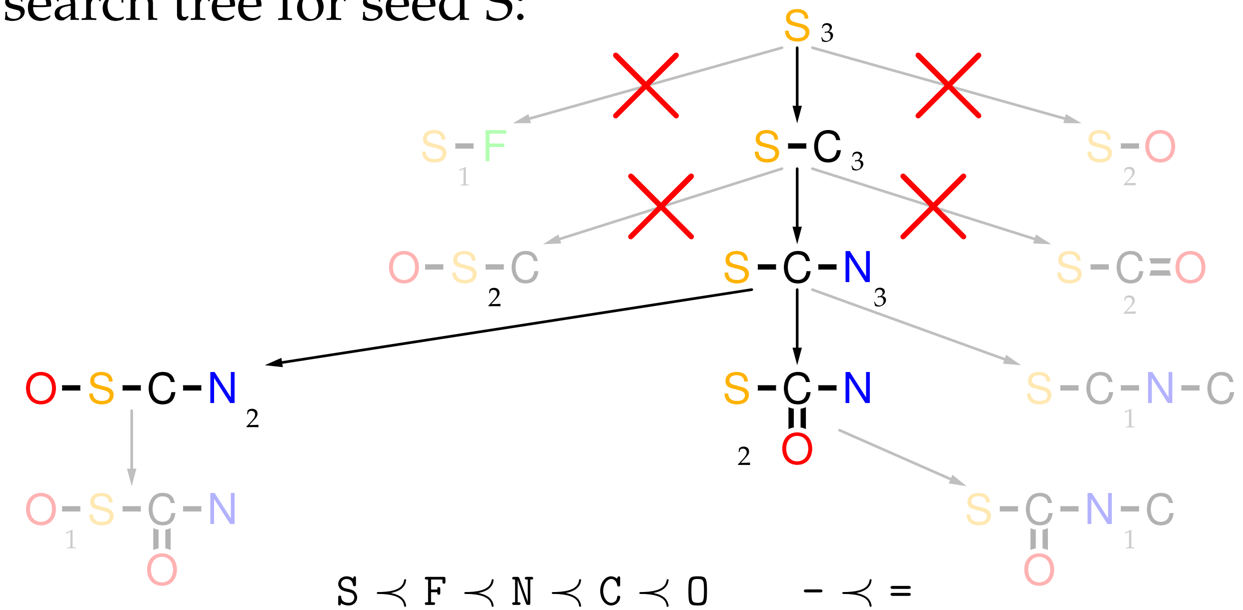
- **Full perfect extension pruning:** [Borgelt and Meinl 2006]
  Also prune the branches to the left of the perfect extension branch.

- **Problem:** This pruning method interferes with canonical form pruning, because the extensions in the left siblings cannot be repeated in the perfect extension branch (restricted extensions, “simple rules” for canonical form).

example molecules:



search tree for seed S:



S ≺ F ≺ N ≺ C ≺ O   - ≺ =

breadth-first search canonical form

Code Word Reorganization

- **Restricted extensions:**
  Not all extensions of a fragment are allowed by the canonical form.
  Some can be checked by simple rules (rightmost path/max. source extension).

- **Consequence:** In order to make canonical form pruning and full perfect extension pruning compatible, the restrictions on extensions must be mitigated.

- **Example:**
  The core problem of obtaining the search tree on the previous slide is how we can avoid that the fragment `O-S-C-N` is pruned as non-canonical:
  - The breadth-first search canonical code word for this fragment is
    `S 0-C1 0-O2 1-N3`.
  - However, with the search tree on the previous slide it is encoded as
    `S 0-C1 1-N2 0-O3`.

- **Solution:** Deviate from appending the description of a new edge.
  Allow for a (strictly limited) code word reorganization.

Code Word Reorganization

- In order to obtain a proper code word, it must be possible to shift descriptions of new edges past descriptions of perfect extension edges in the code word.

- The code word of a fragment consists of two parts:
  - a **prefix** ending with the last non-perfect extension edge and
  - a (possibly empty) **suffix** of perfect extension edges.

- A new edge description is usually appended at the end of the code word. This is still the standard procedure if the suffix is empty.

  However, if the suffix is not empty, the description of the new edge may be inserted into the suffix or even moved directly before the suffix. (Whichever possibility yields the lexicographically smallest code word.)

- Rather than to actually shift and modify edge description, it is technically easier to rebuild the code word from the prefix. (In particular, renumbering the vertices is easier in this way.)

Code Word Reorganization: Example

- **Shift** an extension to the proper place and renumber the vertices:

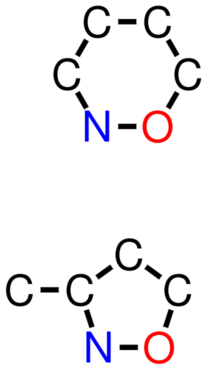
1. Base fragment: S-C-N	canonical code:	S	0-C1	1-N2		
2. Extension to O-S-C-N	(non-canonical!) code:	S	0-C1	1-N2	0-O3
3. Shift extension	(invalid) code:	S	0-C1	0-O3	1-N2
4. Renumber vertices	canonical code:	S	0-C1	0-O2	1-N3

- **Rebuild** the code word from the prefix:
  - The root vertex (here the sulfur atom) is always in the fixed part. It receives the initial vertex index, that is, 0 (zero).
  - Compare two possible code word prefixes: S 0-O1 and S 0-C1. Fix the latter, since it is lexicographically smaller.
  - Compare the code word prefixes S 0-C1 0-O2 and S 0-C1 1-N2 Fix the former, since it is lexicographically smaller.
  - Append the remaining perfect extension edge: S 0-C1 0-O2 1-N3

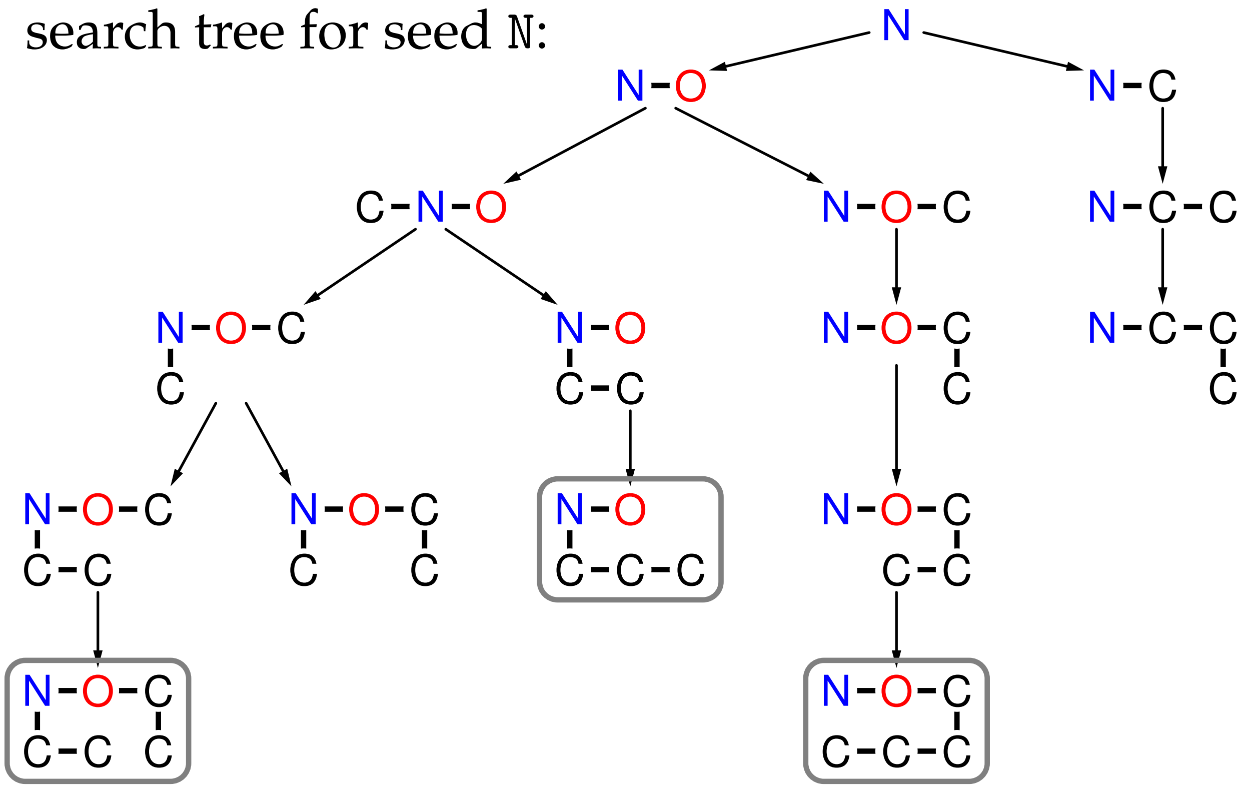
breadth-first search canonical form; S $\prec$ N $\prec$ C $\prec$ O; - $\prec$ =

Perfect Extensions: Problems with Cycles/Rings

example molecules:



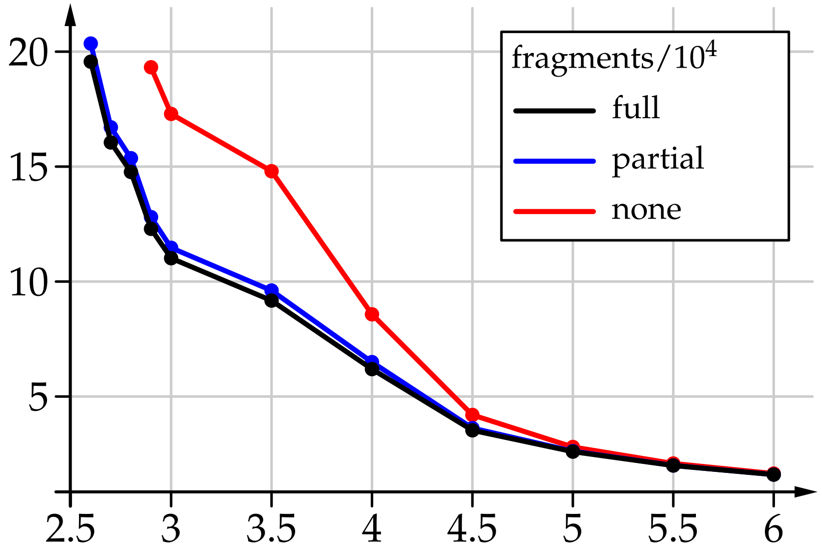
search tree for seed N:

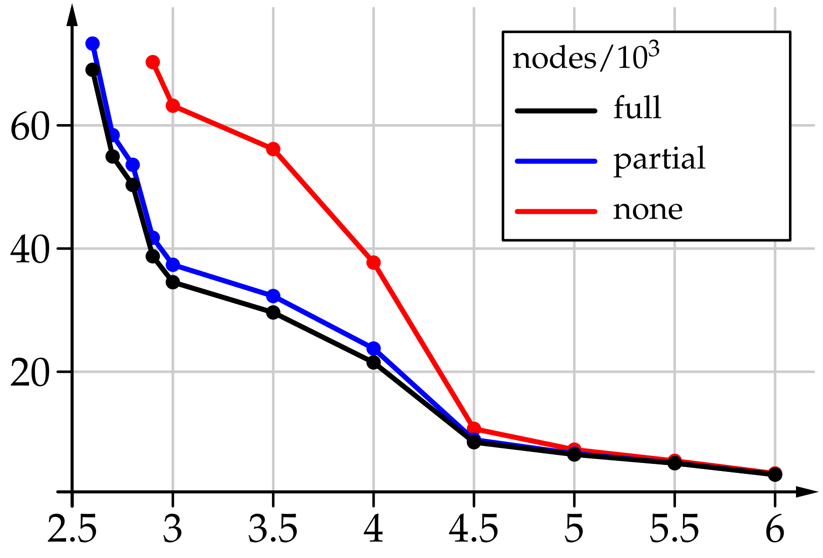


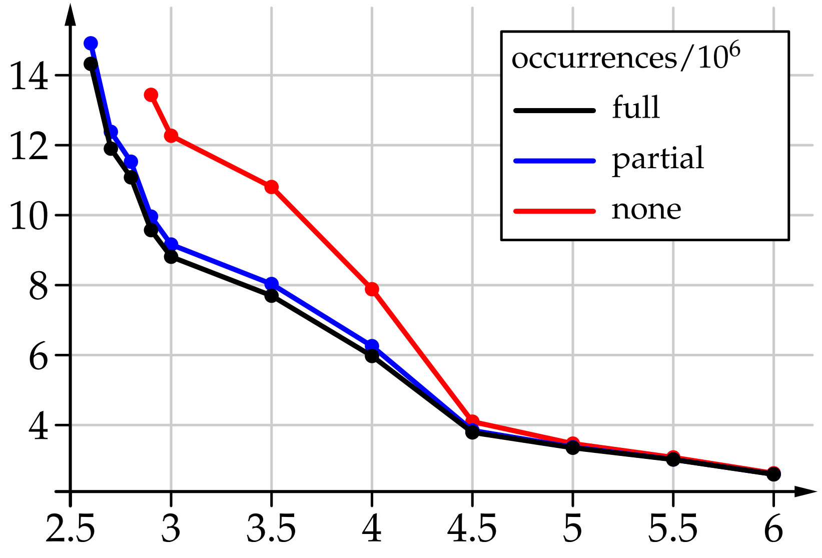
- **Problem:** Perfect extensions in cycles may not allow for pruning.
- **Consequence: Additional constraint** [Borgelt and Meinl 2006]

  Perfect extensions must be bridges or edges closing a cycle/ring.

Experiments: IC93 without Ring Mining

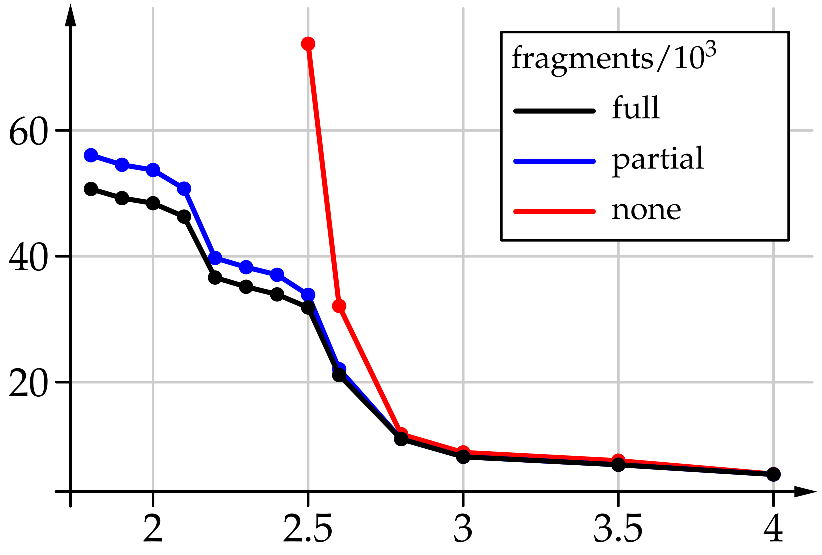


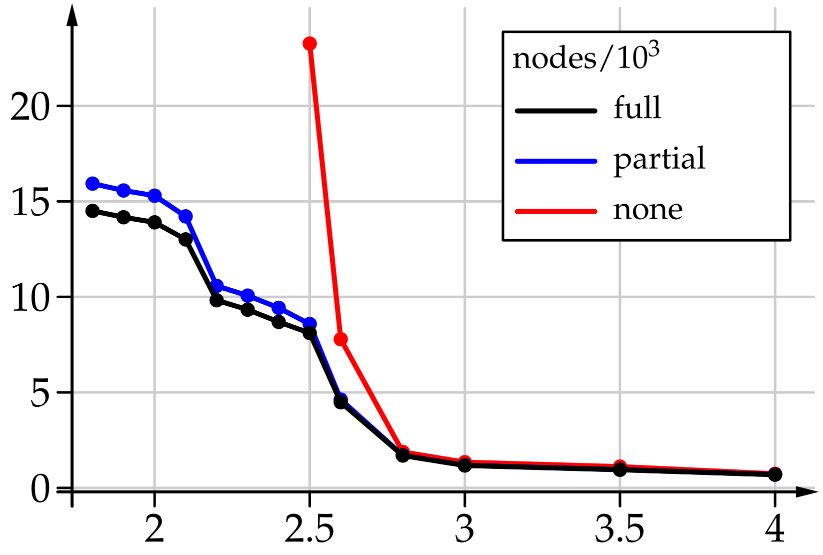


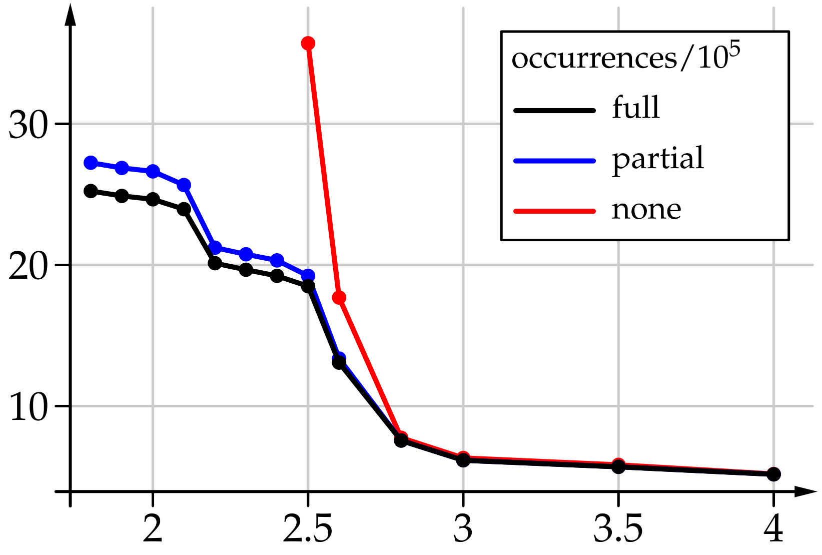


Experimental results on the IC93 data, obtained without ring mining (i.e., with single bond extensions). The horizontal axis shows the minimum support in percent. The curves show the number of generated fragments (top left), the number of processed occurrences (bottom left), and the number of search tree nodes (top right) for the three different methods.

Experiments: IC93 with Ring Mining







Experimental results on the IC93 data, obtained with ring mining. The horizontal axis shows the minimum support in percent. The curves show the number of generated fragments (top left), the number of processed occurrences (bottom left), and the number of search tree nodes (top right) for the three different methods.

Extensions for Molecular Fragment Mining

Extensions of the Search Algorithm

- **Rings** [Hofer, Borgelt, and Berthold 2004; Borgelt 2006]
  - Preprocessing: Find rings in the molecules and mark them.
  - In the search process: Add all atoms and bonds of a ring in one step.
  - Considerably improves efficiency and interpretability.
- **Carbon Chains** [Meinl, Borgelt, and Berthold 2004]
  - Add a carbon chain in one step, ignoring its length.
  - Extensions by a carbon chain match regardless of the chain length.
- **Wildcard Atoms** [Hofer, Borgelt, and Berthold 2004]
  - Define classes of atoms that can be seen as equivalent.
  - Combine fragment extensions with equivalent atoms.
  - Infrequent fragments that differ only in a few atoms from frequent fragments can be found.

Ring Mining: Treat Rings as Units

- **General Idea of Ring Mining**:
  A ring (cycle) is either contained in a fragment as a whole or not at all.

- **Filter Approaches**:
  - (Sub)graphs/fragments are grown edge by edge (as before).
  - Found frequent graph fragments are filtered:
    Graph fragments with incomplete rings are discarded.
  - Additional search tree pruning:
    Prune subtrees that yield only fragments with incomplete rings.

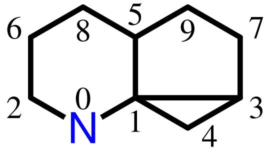
- **Reordering Approach**
  - If an edge is added that is part of one or more rings,
    (one of) the containing ring(s) is added as a whole (all of its edges are added).
  - Incompatibilities with canonical form pruning are handled
    by reordering code words (similar to full perfect extension pruning).

Ring Mining: Preprocessing

Ring mining is simpler after preprocessing the rings in the graphs to analyze:

Basic Preprocessing: (for filter approaches)

- Mark all edges of rings in a user-specified size range.
  (molecular fragment mining: usually rings with 5 – 6 vertices/atoms)
- Technically, there are two ring identification parts per edge:
  - A marker in the edge attribute,
    which fundamentally distinguishes ring edges from non-ring edges.
  - A set of flags identifying the different rings an edge is contained in.
    (Note that an edge can be part of several rings.)



Extended Preprocessing: (for reordering approach)

- Mark *pseudo-rings*, i.e. rings of smaller size than the user specified, but which consist only of edges that are part of rings within the user-specified size range.

Filter Approaches: Open Rings

Idea of Open Ring Filtering:

If we require the output to have only complete rings, we have to identify and remove fragments with ring edges that do not belong to any complete ring.

- Ring edges have been marked in the preprocessing.

  ⇒ It is known which edges of a grown (sub)graph are ring edges (in the underlying graphs of the database).

- Apply the preprocessing procedure to a grown (sub)graph, but
  - ○ keep the marker in the edge attribute;
  - ○ only set the flags that identify the rings an edge is contained in.

- Check for edges that have a ring marker in the edge attribute, but did not receive any ring flag when the (sub)graph was reprocessed.

- If such edges exist, the (sub)graph contains unclosed/open rings, so the (sub)graph must *not* be reported.

Filter Approaches: Unclosable Rings

Idea of Unclosable Ring Filtering:

Grown (sub)graphs with open rings that cannot be closed by future extensions can be pruned from the search.

- Canonical form pruning allows to restrict the possible extensions of a fragment.

  ⇒ Due to previous extensions certain vertices become unextendable.

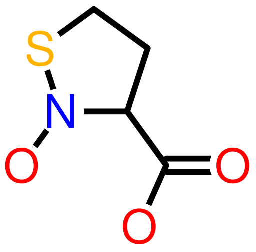
  ⇒ Some rings cannot be closed by extending a (sub)graph.

- Obviously, a necessary (though not sufficient) condition for all rings being closed is that every vertex has either zero or at least two incident ring edges.

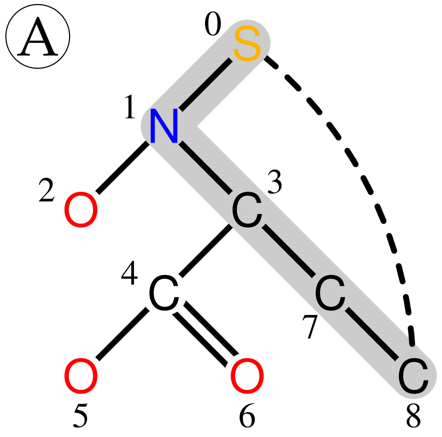
  If there is a vertex with only one incident ring edge,
  this edge must be part of an incomplete ring.

- If an unextendable vertex of a grown (sub)graph has only one incident ring edge, this (sub)graph can be pruned from the search (because there is an open ring that can never be closed).

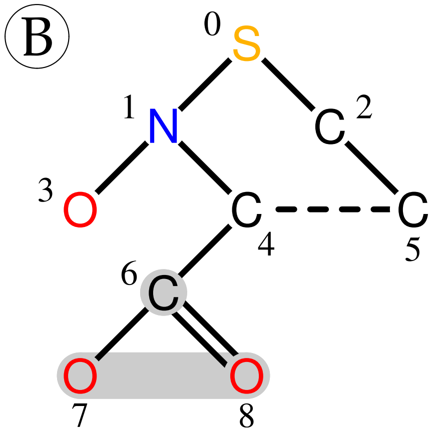
Reminder: Restricted Extensions



example molecule



depth-first



breadth-first

Extendable Vertices:

A: vertices on the rightmost path, that is, 0, 1, 3, 7, 8.

B: vertices with an index no smaller than the maximum source, that is, 6, 7, 8.

Edges Closing Cycles:

A: none, because the existing cycle edge has the smallest possible source.

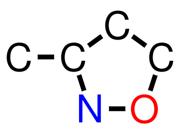
B: an edge between the vertices 7 and 8.

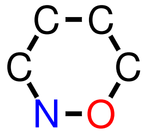
Filter Approaches: Merging Ring Extensions

Idea of Merging Ring Extensions:

The previous methods work on individual edges and hence cannot always detect if an extension only leads to fragments with complete rings that are infrequent.

- Add all edges of a ring, thus distinguishing extensions that
  - start with the same individual edge, but
  - lead into rings of different size or different composition.





- Determine the support of the grown (sub)graphs and prune infrequent ones.
- Trim and merge ring extensions that share the same initial edge.

Advantage of Merging Ring Extensions:

- All extensions are removed that become infrequent when completed into rings.
- All occurrences are removed that lead to infrequent (sub)graphs once rings are completed.

A Reordering Approach

- **Drawback of Filtering:**

  (Sub)graphs are still extended edge by edge. ⇒ Fragments grow fairly slowly.

- **Better Approach:**

  - Add all edges of a ring in one step. (When a ring edge is added, create one extended (sub)graph for each ring it is contained in.)
  - Reorder certain edges in order to comply with canonical form pruning.

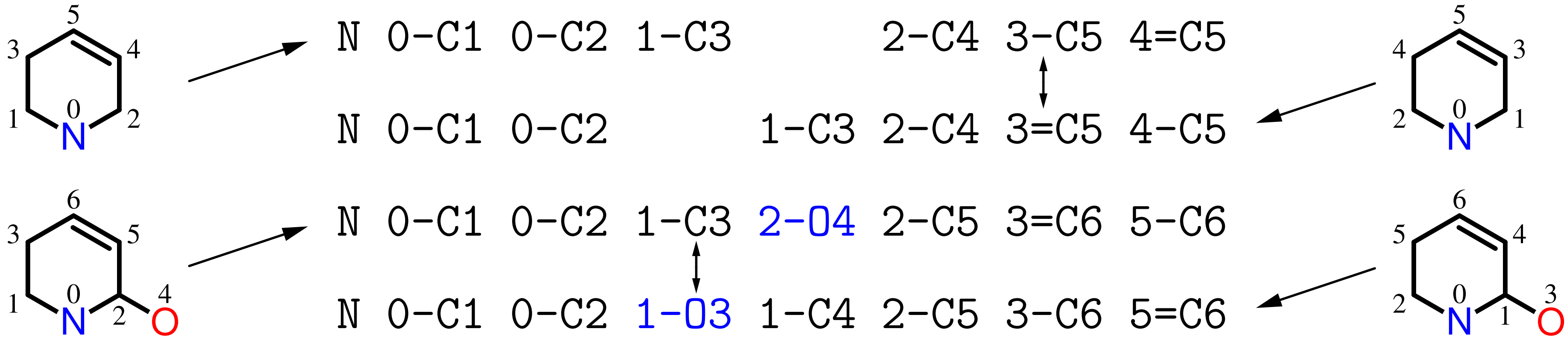
- **Problems of a Reordering Approach:**

  - One must allow for insertions between already added ring edges (because branches may precede ring edges in the canonical form).
  - One must not commit too early to an order of the edges (because branches may influence the order of the ring edges).
  - All possible orders of (locally) equivalent edges must be tried, because any of them may produce valid output.

Problems of Reordering Approaches

One must not commit too early to an order of the edges.

Illustration: effects of attaching a branch to an asymmetric ring. $\texttt{N} \prec \texttt{O} \prec \texttt{C},\ \texttt{-} \prec \texttt{=}$



```
N 0-C1 0-C2 1-C3      2-C4 3-C5 4=C5
N 0-C1 0-C2      1-C3 2-C4 3=C5 4-C5
N 0-C1 0-C2 1-C3 2-O4 2-C5 3=C6 5-C6
N 0-C1 0-C2 1-O3 1-C4 2-C5 3-C6 5=C6
```

- W.r.t. a breadth-first search canonical form, the edges of the ring can be ordered in two different ways (upper two rows).

  The upper/left is the canonical form of the pure ring.

- With an attached branch (close to the root vertex), the other ordering of the ring edges (lower/right) is the canonical form.

Keeping Non-Canonical Fragments

Solution of the early commitment problem:

Maintain (and extend) both orderings of the ring edges and
allow for deviations from the canonical form beyond "fixed" edges.

- **Principle:** keep (and, consequently, also extend) fragments that are not in canonical form, but that could become canonical once branches are added.

- Needed: a rule which non-canonical fragments to keep and which to discard.

- Idea: adding a ring can be seen as adding its initial edge as in an edge-by-edge procedure, and some additional edges, the positions of which are not yet fixed.

- As a consequence we can split the code word into two parts:
  - a **fixed prefix**, which is also built by an edge-by-edge procedure, and
  - a **volatile suffix**, which consists of the additional (ring) edges.

Keeping Non-Canonical Fragments

- **Fixed prefix of a code word:**
  The prefix of the code word up to (and including) the last edge added in an edge-by-edge manner.

- **Volatile suffix of a code word:**
  The suffix of the code word after the last edge added in an edge-by-edge manner (with this last edge excluded).

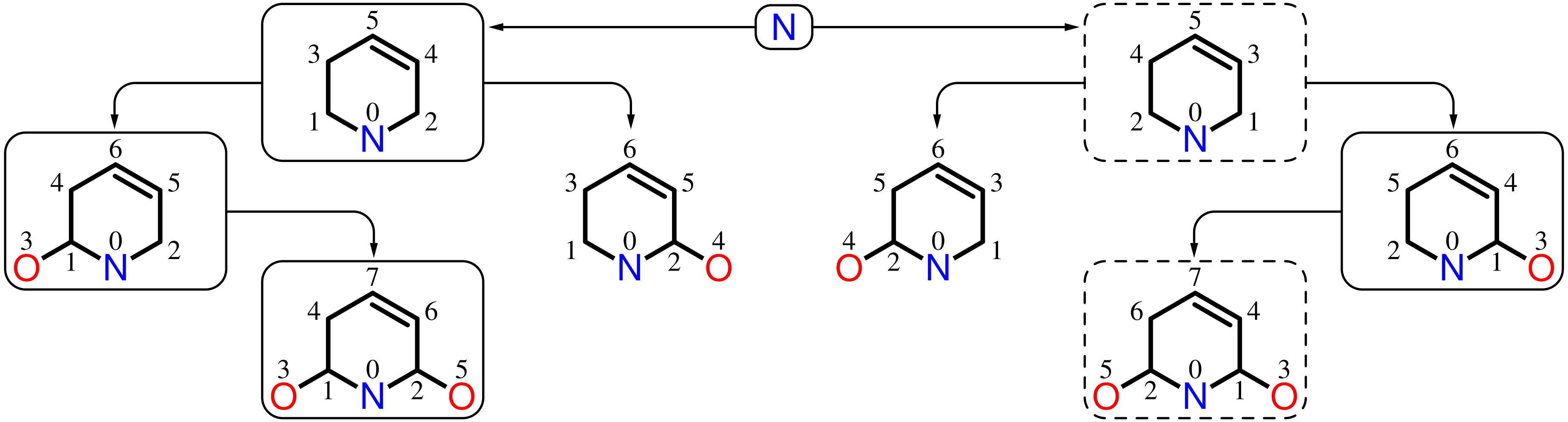
- **Rule for keeping non-canonical fragments:**

  If the current code word deviates from the canonical code word in the fixed part, the fragment is pruned, otherwise it is kept.

- **Justification of this rule:**
  - If the deviation is in the fixed part, no later addition of edges can have any effect on it, since the fixed part will never be changed.
  - If, however, the deviation is in the volatile part, a later extension edge may be inserted in such a way that the code word becomes canonical.

Search Tree for an Asymmetric Ring with Branches

Maintain (and extend) both orderings of the ring edges and allow for deviations from the canonical form beyond *fixed* edges.



The edges of a grown subgraph are split into

- *fixed edges* (edges that could have been added in an edge-by-edge manner),
- *volatile edges* (edges that have been added with ring extensions and before/between which edges may be inserted).

Search Tree for an Asymmetric Ring with Branches

- The search constructs the ring with both possible numberings of the vertices.
  - The form on the left is canonic, so it is kept.
  - In the fragment on the right only the first ring bond is fixed,
    all other bonds are volatile.

    Since the code word for this fragment deviates from the canonical one only at the 5th bond, we may not discard it.

- On the next level, there are two canonical and two non-canonical fragments.

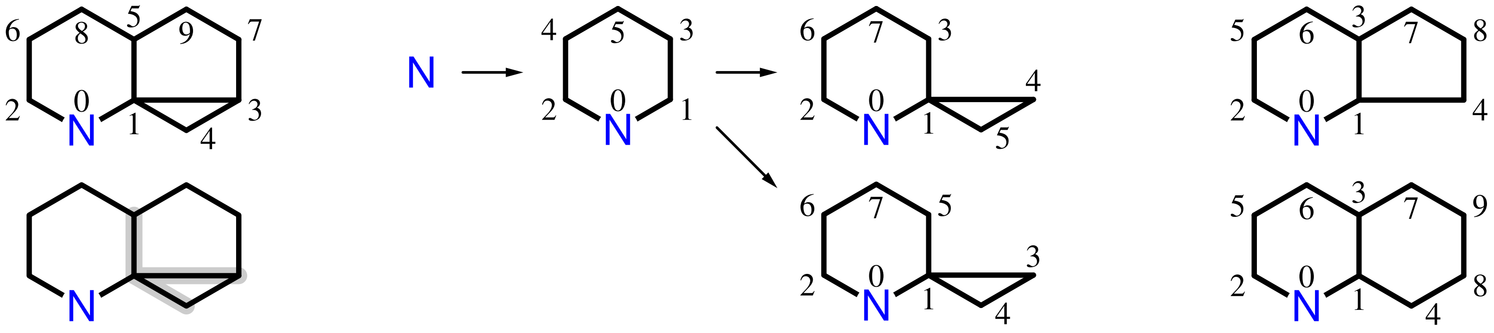
  The non-canonical fragments both differ in the fixed part,
  which now consists of the first three bonds, and thus are pruned.

- On the third level, there is one canonical and one non-canonical fragment.

  The non-canonical fragment differs in the volatile part (the first four bonds are fixed, but it deviates from the canonical code word only in the 7th bond) and thus may not be pruned from the search.

Connected and Nested Rings

Connected and nested rings can pose problems, because in the presence of **equivalent edges** the order of these edges cannot be determined locally.



- Edges are (locally) **equivalent** if they start from the same vertex, have the same edge attribute, and lead to vertices with the same vertex attribute.
- Equivalent edges must be **spliced** in all ways, in which the order of the edges already in the (sub)graph and the order of the newly added edges is preserved.
- It is necessary to consider **pseudo-rings** for extensions, because otherwise not all orders of equivalent edges are generated.

Splicing Equivalent Edges

- In principle, **all possible orders of equivalent edges** have to be considered, because any of them may in the end yield the canonical form.

  We cannot (always) decide locally which is the right order, because this may depend on edges added later.

- Nevertheless, we may not reorder equivalent edges freely, as this would interfere with keeping certain non-canonical fragments:

  By keeping some non-canonical fragments we already consider some variants of orders of equivalent edges. These must not be generated again.

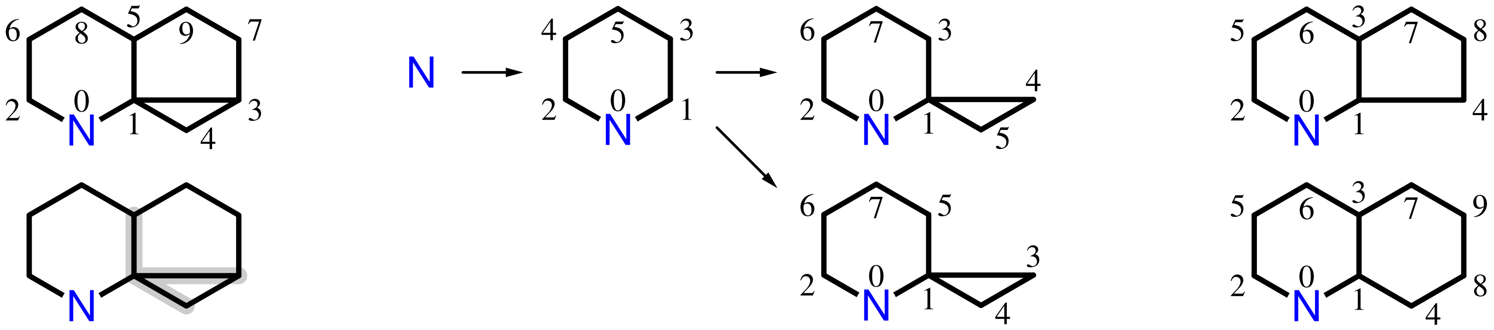
- **Splicing rule for equivalent edges:** (breadth-first search canonical form)

  The order of the equivalent edges already in the fragment must be maintained, and the order of the equivalent new edges must be maintained.

  The two sequences of equivalent edges may be merged in a “zip-like” manner, selecting the next edge from either list, but preserving the order in each list.

The Necessity of Pseudo-Rings

The **splicing rule** explains the necessity of **pseudo-rings**:
Without pseudo-rings it is impossible to achieve canonical form in some cases.



- If we could only add the 5-ring and the 6-ring, but not the 3-ring, the upward bond from the atom numbered 1 would always precede at least one of the other two bonds that are equivalent to it (since the order of existing bonds must be preserved).

- However, in the canonical form the upward bond succeeds both other bonds, and this we can achieve only by adding the 3-bond ring first.

Splicing Equivalent Edges

- The considered **splicing rule** is for a breadth-first search canonical form.

  In this form equivalent edges are adjacent in the canonical code word.

- In a depth-first search canonical form equivalent edges can be far apart from each other in the code word.

  Nevertheless some "splicing" is necessary to properly treat equivalent edges in this canonical form, even though the rule is slightly simpler.

- **Splicing rule for equivalent edges:** (depth-first search canonical form)

  The first new ring edge has to be tried in all locations in the volatile part of the code word, where equivalent edges can be found.

- Since we cannot decide locally which of these edges should be followed first when building the spanning tree, we have to try all of these possibilities in order not to miss the canonical one.

Avoiding Duplicate Fragments

- The splicing rules still allow that the same fragment can be reached in the same form in different ways, namely by adding (nested) rings in different orders.

  Reason: we cannot always distinguish between two different orders
  in which two rings sharing a vertex are added.

- Needed: an **augmented canonical form test**.

- **Ideas** underlying such an augmented test:

  ○ The requirement of complete rings introduces dependences between edges:

    The presence of certain edges *enforces* the presence of certain other edges.

  ○ The same code word of a fragment is created several times, but each time with a **different fixed part**:

    The position of the first edge of a ring extension (after reordering) is the end of the fixed part of the (extended) code word.

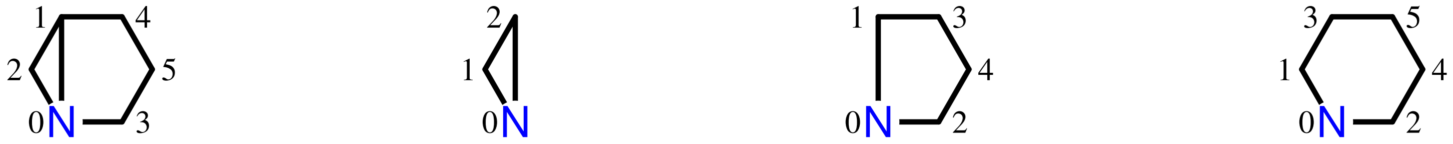
Ring Key Pruning

Dependences between Edges

- The requirement of complete rings introduces dependences between edges. (Idea: consider forming sub-fragments with only complete rings.)
- A ring edge e_1 of a fragment **enforces the presence** of another ring edge e_2 iff the set of rings containing e_1 is a subset of the set of rings containing e_2.
  - In order for a ring edge to be present in a sub-fragment, at least one of the rings containing it must be present.
  - If a ring edge e_1 enforces a ring edge e_2, it is not possible to form a sub-fragment with only complete rings that contains e_1, but not e_2.
  - Obviously, every ring edge enforces at least its own presence.
  - In order to capture also non-ring edges by such a definition, we define that a non-ring edge enforces only its own presence.

Ring Key Pruning

Example of Dependences between Edges



(All edge descriptions refer to the vertex numbering in the fragment on the left.)

- In the fragment on the left, any edge in the set $\{(0,3),(1,4),(3,5),(4,5)\}$ enforces the presence of any other edge in this set, because all of these edges are contained exactly in the 5-ring and the 6-ring.

- In the same way, the edges $(0,2)$ and $(1,2)$ enforce each other, because both are contained exactly in the 3-ring and the 6-ring.

- The edge $(0,1)$, however, only enforces itself and is enforced only by itself.

- There are no other enforcement relations between edges.

Ring Key Pruning

(Shortest) Ring Keys

- We consider prefixes of code words that contain $4k + 1$ characters, $k \in \{0, 1, \ldots, m\}$, where m is the number of edges of the fragment.

- A prefix v of a code word vw (whether canonical or not) is called a **ring key** iff each edge described in w is enforced by at least one edge described in v.

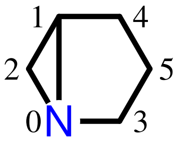
- The prefix v is called a **shortest ring key** of vw iff it is a ring key and there is no shorter prefix that is a ring key for vw.

  Note: The shortest ring key of a code word is uniquely defined,
  but depends, of course, on the considered code word.

- Idea of (Shortest) Ring Key Pruning:

  Discard fragments that are formed with a code word, the fixed part of which is not a shortest ring key.

Ring Key Pruning



- **Example** of (shortest) ring key(s):

  Breadth-first search (canonical) code word:

	`N 0-C1`	`0-C2`	`0-C3`	`1-C2`	`1-C4`	`3-C5`	`4-C5`
Edges:	e_1	e_2	e_3	e_4	e_5	e_6	e_7

- `N` is obviously not a ring key, because it enforces no edges.

- `N 0-C1` is not a ring key, because it does not enforce, for example, e_2 or e_3.

- `N 0-C1 0-C2` is not a ring key, because it does not enforce, for example, e_3.

- `N 0-C1 0-C2 0-C3` is the shortest ring key, because
  $e_4 = (1, 2)$ is enforced by $e_2 = (0, 2)$ and
  $e_5 = (1, 4)$, $e_6 = (3, 5)$ and $e_7 = (4, 5)$ are enforced by $e_3 = (0, 3)$.

- Any longer prefix is a ring key, but not a shortest ring key.

Ring Key Pruning

- If only code words with fixed parts that are shortest ring keys are extended, it suffices to check whether the fixed part is a ring key.

- Induction anchor: If a fragment contains only one ring, the first ring edge enforces the other ring edges and thus the fixed part is a shortest ring key.

- Induction step:

  - Let vw be a code word with fixed part v and volatile part w, for which the prefix v is a shortest ring key.

  - Extending this code word generally transforms it into a code word $vuxw'$.
    u describes edges originally described by parts of w (u may be empty),
    x is the description of the first new edge and
    w' describes the remaining old and new edges.

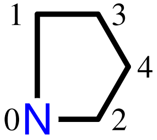
  - The code word $vuxw'$ cannot have a shorter ring key than vux, because the edges described in vu do not enforce the edge described by x.

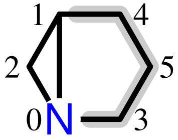
Ring Key Pruning

Test Procedure of Ring Key Pruning

- Check for each volatile edge whether it is enforced by at least one fixed edge:
  - Mark all rings in the considered fragment (set ring flags).
  - Remove all rings containing a given volatile edge *e* (clear ring flags).
  - If by this procedure a fixed ring edge becomes flagless,
    the edge *e* is enforced by it, otherwise the edge *e* is not enforced.

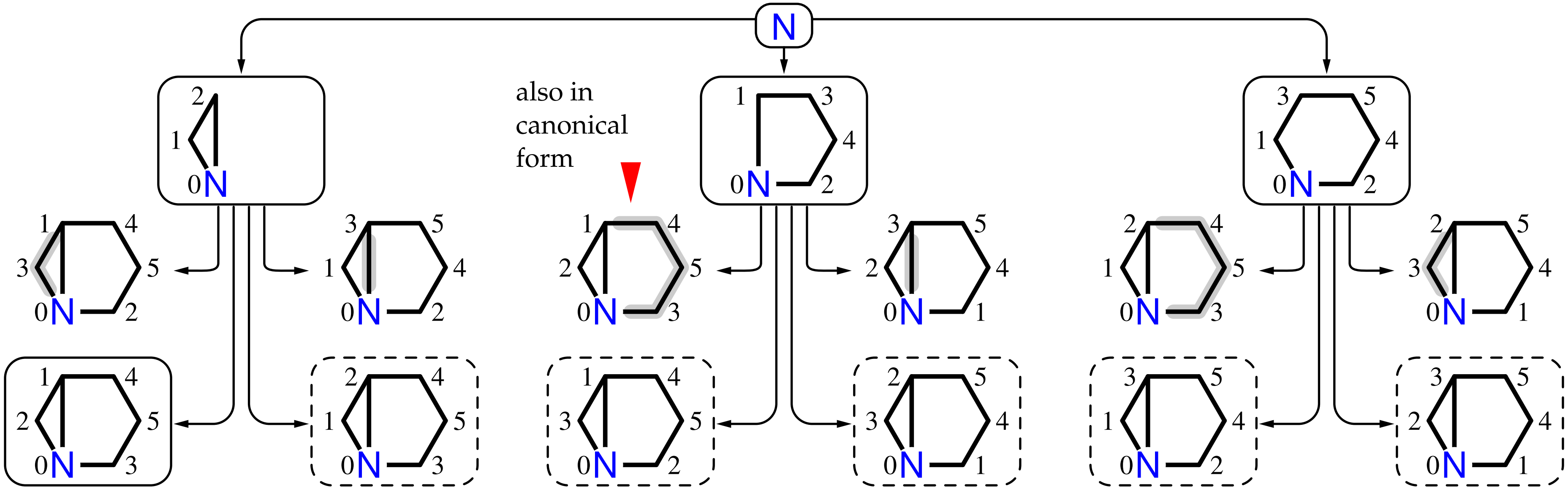
- **Example:**





  - Extending the 5-ring yields the fragment on the right in canonical form with the first two edges (that is, $e_1 = (0, 1)$ and $e_2 = (0, 2)$) fixed.
  - The prefix `N 0-C1 0-C2` is not a ring key (the gray edges are not enforced) and hence the fragment is discarded, even though it is in canonical form.

Search Tree for Nested Rings



(solid frame: extended and reported; dashed frame: extended, but not reported; no frame: pruned)

- The full fragment is generated twice in each form (even the canonical).
- **Augmented Canonical Form Test:**
  - The created code words have different fixed parts.
  - Check whether the fixed part is a shortest ring key.

Search Tree for Nested Rings

- In all fragments in the bottom row of the search tree (fragments with frames) the first three edges are fixed, the suffix is volatile.

  The prefix `N 0-C1 0-C2 0-C3` describing these edges is a shortest ring key. Hence these fragments are kept and processed.

- In the row above it (fragments without frames), only the first two edges are fixed, the suffix is volatile.

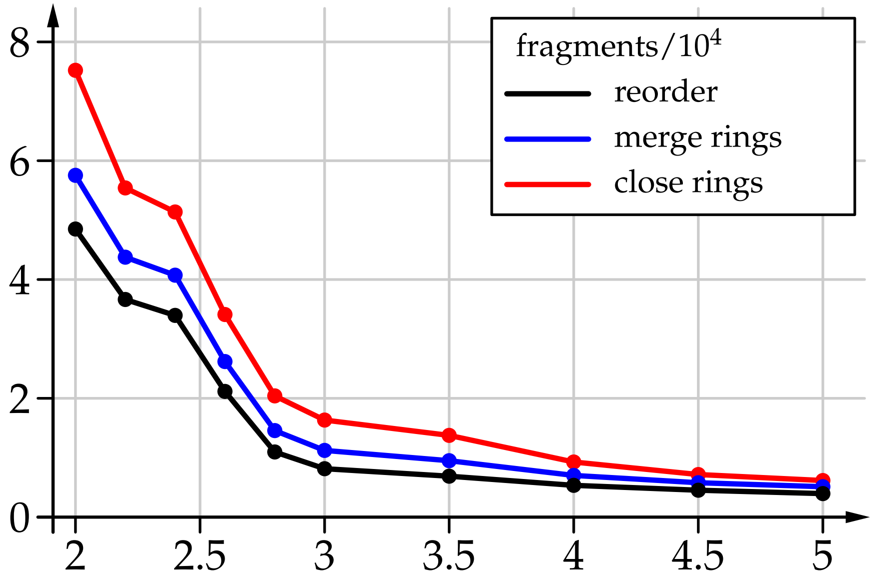
  The prefix `N 0-C1 0-C2` describing these edges is not a ring key. (The gray edges are not enforced.) Hence these fragments are discarded.

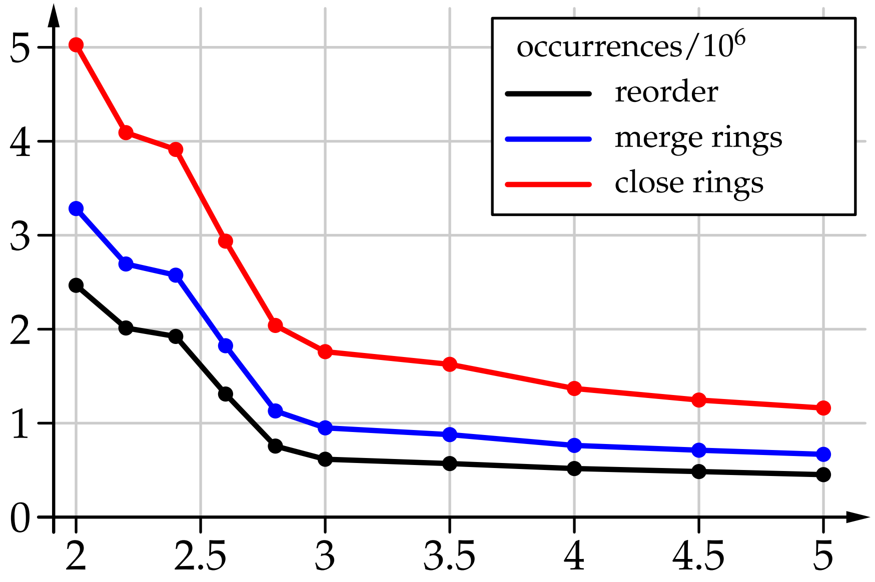
- Note that for all single ring fragments two of their four children are kept, even though only the one at the left bottom is in canonical form.

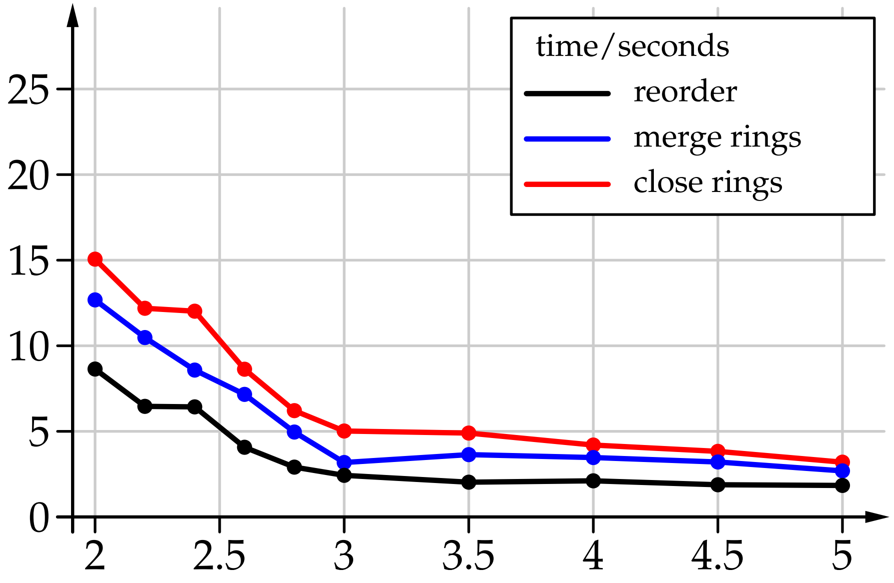
  The reason is that the deviation from the canonical form resides in the volatile part of the fragment.

  By attaching additional rings any of these fragments may become canonical.

Experiments: IC93

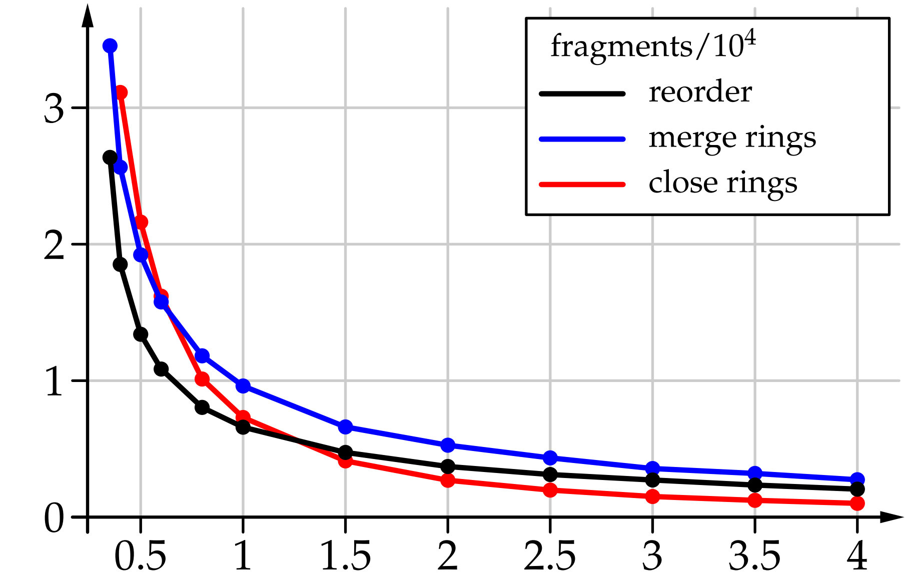


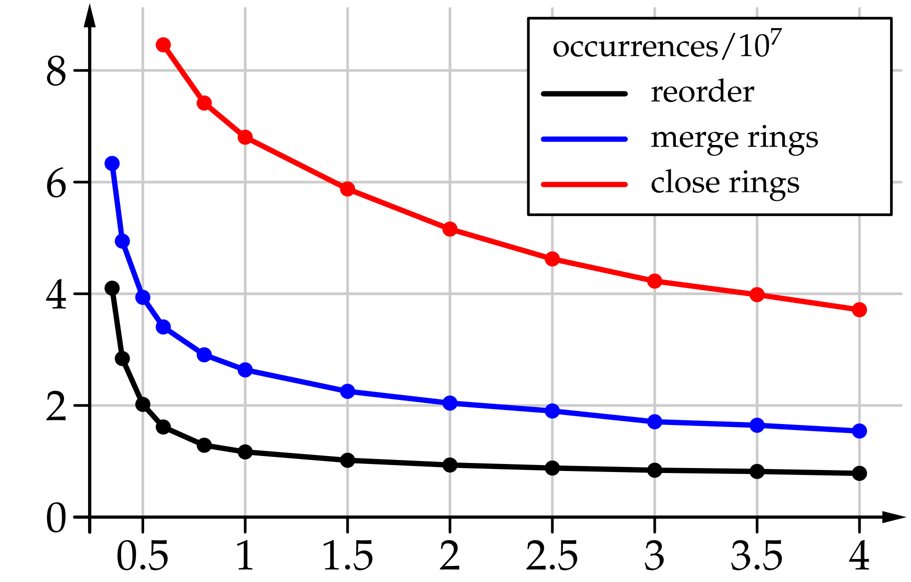


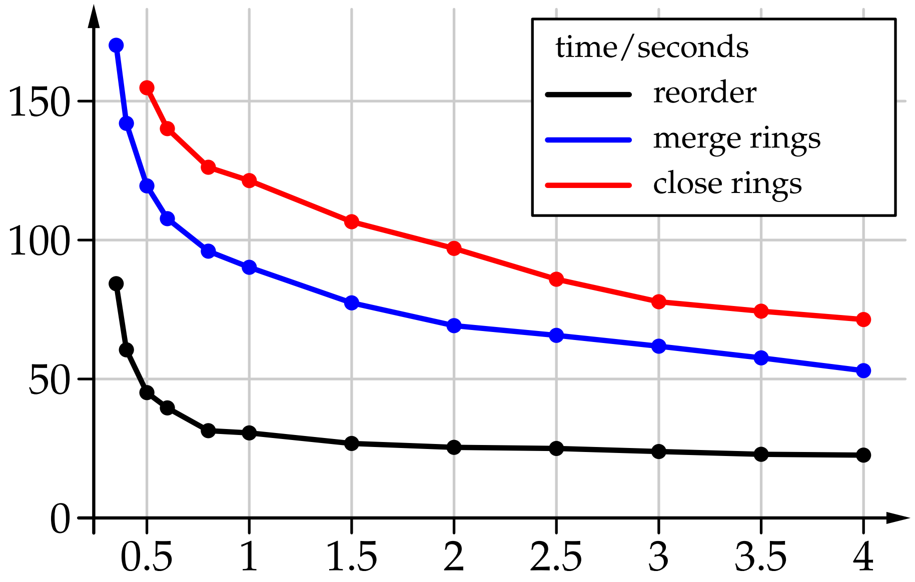


Experimental results on IC93 data. The horizontal axis shows the minimum support in percent. The curves show the number of generated fragments (top left), the number of processed occurrences (top right), and the execution time in seconds (bottom left) for the three different strategies.

Experiments: NCI HIV Screening Database





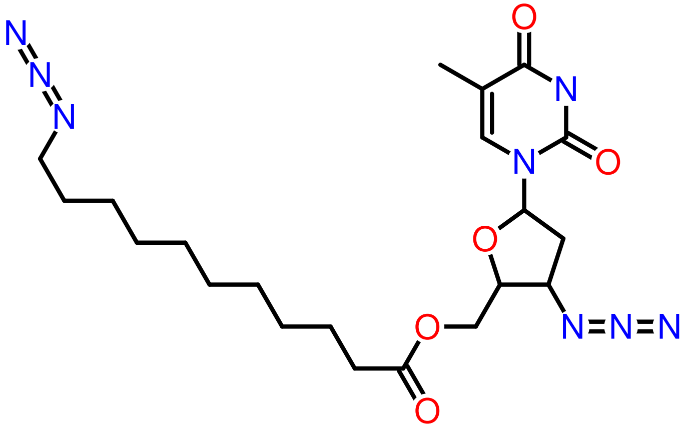


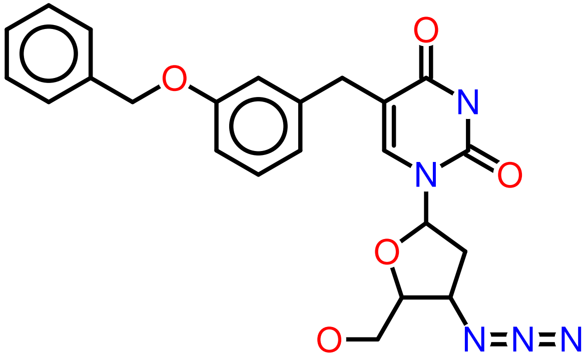
Experimental results on the HIV data. The horizontal axis shows the minimum support in percent. The curves show the number of generated fragments (top left), the number of processed occurrences (top right), and the execution time in seconds (bottom left) for the three different strategies.

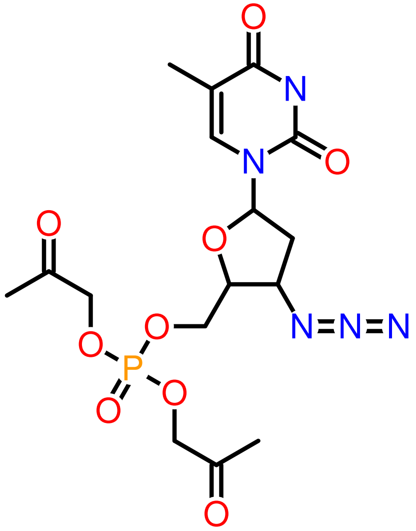
Found Molecular Fragments

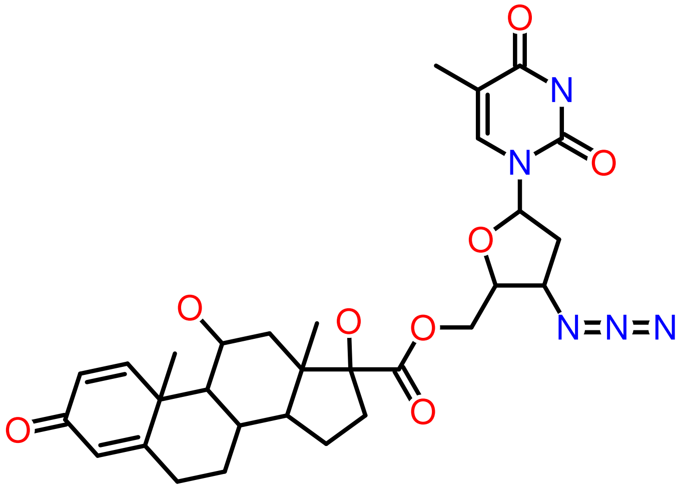
NCI DTP HIV Antiviral Screen: AZT

Some Molecules from the NCI HIV Database

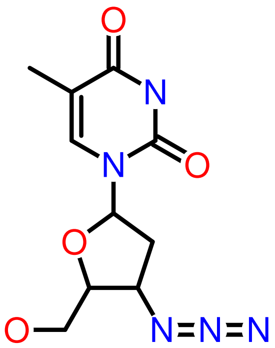






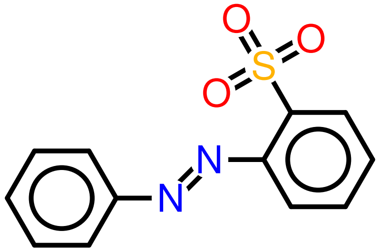


Common Fragment

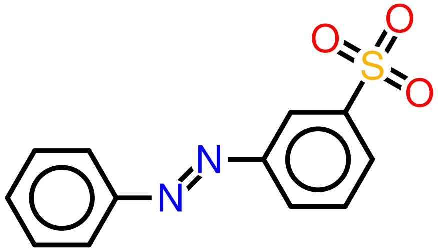


NCI DTP HIV Antiviral Screen: Other Fragments

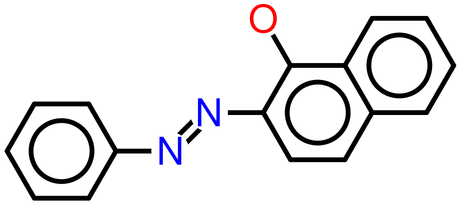
Fragment 1:
CA: 5.23%
CI/CM: 0.05%



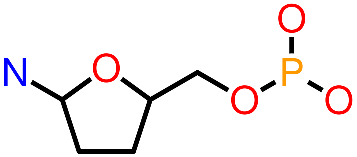
Fragment 2:
CA: 4.92%
CI/CM: 0.07%



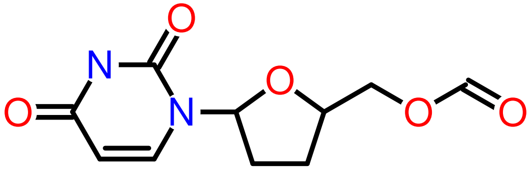
Fragment 3:
CA: 5.23%
CI/CM: 0.08%



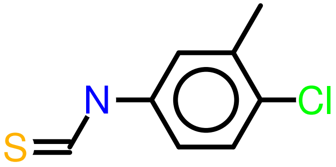
Fragment 4:
CA: 9.85%
CI/CM: 0.07%



Fragment 5:
CA: 10.15%
CI/CM: 0.04%

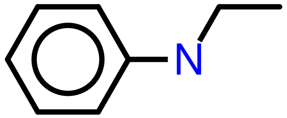


Fragment 6:
CA: 9.85%
CI/CM: 0.00%

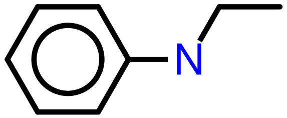


Experiments: Ring Extensions

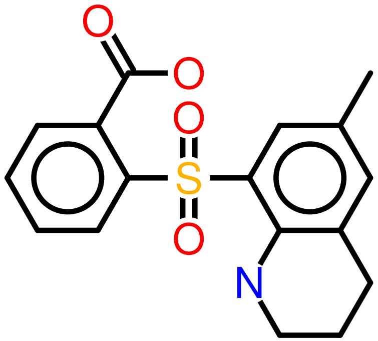
Improved Interpretability



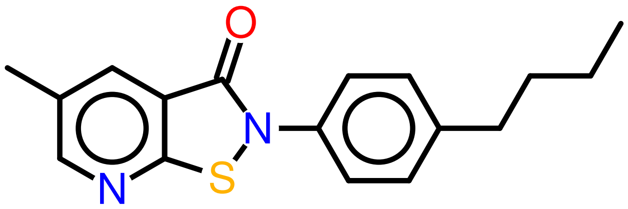
Fragment 1
basic algorithm
freq. in CA: 22.77%



Fragment 2
with ring extensions
freq. in CA: 20.00%



NSC #667948



NSC #698601

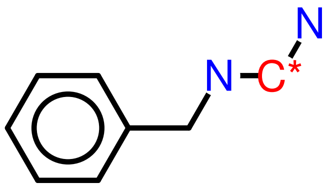
Compounds from the NCI cancer data set that contain Fragment 1 but not 2.

Experiments: Carbon Chains

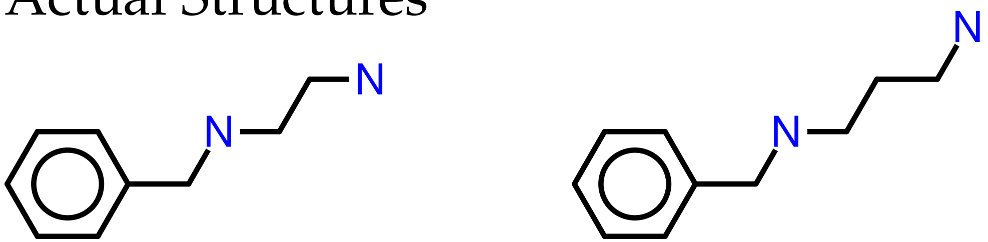
- Technically: Add a carbon chain in one step, ignoring its length.
- Extension by a carbon chain match regardless of the chain length.
- Advantage: Fragments can represent carbon chains of varying length.

Example from the NCI Cancer Dataset:

Fragment with Chain



Actual Structures

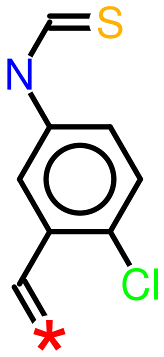


freq. CA:	1.48%
freq. CI:	0.13%

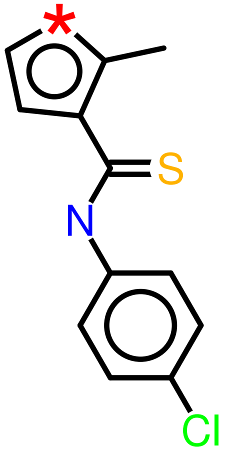
Experiments: Wildcard Atoms

- Define classes of atoms that can be considered as equivalent.
- Combine fragment extensions with equivalent atoms.
- Advantage: Infrequent fragments that differ only in a few atoms from frequent fragments can be found.

Examples from the NCI HIV Dataset:



	∗=O	∗=N
CA:	5.5%	3.7%
CI/CM:	0.0%	0.0%



	∗=O	∗=S
CA:	5.5%	0.01%
CI/CM:	0.0%	0.0%

Summary Frequent (Sub)Graph Mining

- Frequent (sub)graph mining is closely related to frequent item set mining: **Find frequent (sub)graphs** instead of frequent subsets.

- A core problem of frequent (sub)graph mining is how to avoid redundant search. This problem is solved with the help of **canonical forms of graphs**. Different canonical forms lead to different behavior of the search algorithm.

- The restriction to **closed fragments** is a lossless reduction of the output. All frequent fragments can be reconstructed from the closed ones.

- A restriction to closed fragments allows for additional pruning strategies: partial and full **perfect extension pruning**.

- Extensions of the basic algorithm (particularly useful for molecules) include: **Ring Mining**, **(Carbon) Chain Mining**, and **Wildcard Vertices**.

- A **Java implementation** for molecular fragment mining is available at: `http://www.borgelt.net/moss.html`

Mining a Single Graph

Reminder: Basic Notions

- A **labeled** or **attributed graph** is a triplet $G = (V, E, \ell)$, where
  - V is the set of vertices,
  - $E \subseteq V \times V - \{(v, v) \mid v \in V\}$ is the set of edges, and
  - $\ell : V \cup E \to A$ assigns labels from the set A to vertices and edges.

- Let $G = (V_G, E_G, \ell_G)$ and $S = (V_S, E_S, \ell_S)$ be two labeled graphs.

  A **subgraph isomorphism** of S to G or an **occurrence** of S in G is an injective function $f : V_S \to V_G$ with

  - $\forall v \in V_S : \quad \ell_S(v) = \ell_G(f(v)) \quad$ and
  - $\forall (u, v) \in E_S : \quad (f(u), f(v)) \in E_G \quad \wedge \quad \ell_S((u, v)) = \ell_G((f(u), f(v)))$.

  That is, the mapping f preserves the connection structure and the labels.

Anti-Monotonicity of Subgraph Support

Most natural definition of subgraph support in a single graph setting:
number of occurrences (subgraph isomorphisms).

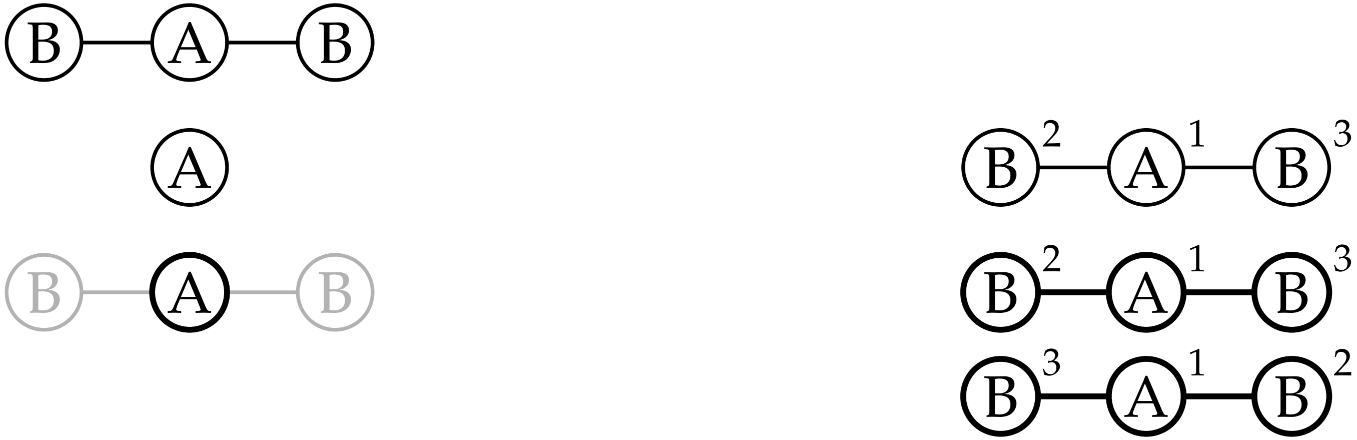
Problem: The number of occurrences of a subgraph is **not anti-monotone**.

Example: $s_G(A) = 1$ $s_G(B{-}A{-}B) = 2$

input graph:

subgraphs:

occurrences:



But: Anti-monotonicity is vital for the efficiency of frequent subgraph mining.

Question: How should we define subgraph support in a single graph?

Anti-Monotonicity of Subgraph Support

Most natural definition of subgraph support in a single graph setting: **number of occurrences** (subgraph isomorphisms).

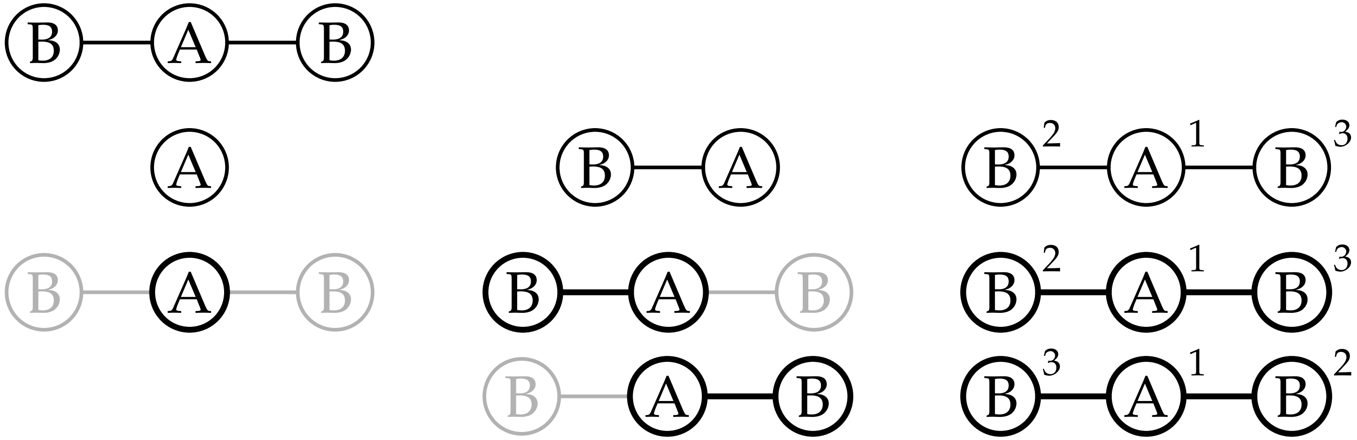
Problem: The number of occurrences of a subgraph is **not anti-monotone**.

Example: $s_G(A) = 1$ $s_G(A{-}B) = 2$ $s_G(B{-}A{-}B) = 2$

input graph:

subgraphs:

occurrences:



But: Anti-monotonicity is vital for the efficiency of frequent subgraph mining.

Question: How should we define subgraph support in a single graph?

Relations between Occurrences

- Let f_1 and f_2 be two subgraph isomorphisms of S to G and
  $V_1 = \{v \in V_G \mid \exists u \in V_S : v = f_1(u)\}$ and
  $V_2 = \{v \in V_G \mid \exists u \in V_S : v = f_2(u)\}$.

  The two subgraph isomorphisms f_1 and f_2 are called

  - **overlapping**, written $f_1 \circledcirc f_2$, iff $V_1 \cap V_2 \neq \emptyset$,
  - **equivalent**, written $f_1 \circ f_2$, iff $V_1 = V_2$,
  - **identical**, written $f_1 \equiv f_2$, iff $\forall v \in V_S : f_1(v) = f_2(v)$.

- Note that identical subgraph isomorphisms are equivalent and that equivalent subgraph isomorphisms are overlapping.

- There can be non-identical, but equivalent subgraph isomorphisms, namely if S possesses an automorphism that is not the identity.

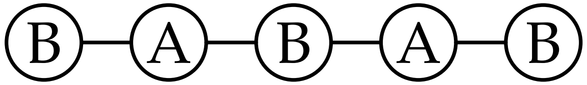
Overlap Graphs of Occurrences

Let $G = (V_G, E_G, \ell_G)$ and $S = (V_S, E_S, \ell_S)$ be two labeled graphs and let V_O be the set of all occurrences (subgraph isomorphisms) of S in G.

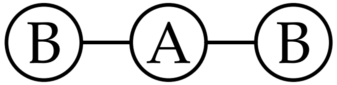
The **overlap graph** of S w.r.t. G is the graph $O = (V_O, E_O)$, which has the set V_O of occurrences of S in G as its vertex set and the edge set $E_O = \{(f_1, f_2) \mid f_1, f_2 \in V_O \wedge f_1 \not\equiv f_2 \wedge f_1 \between f_2\}$.

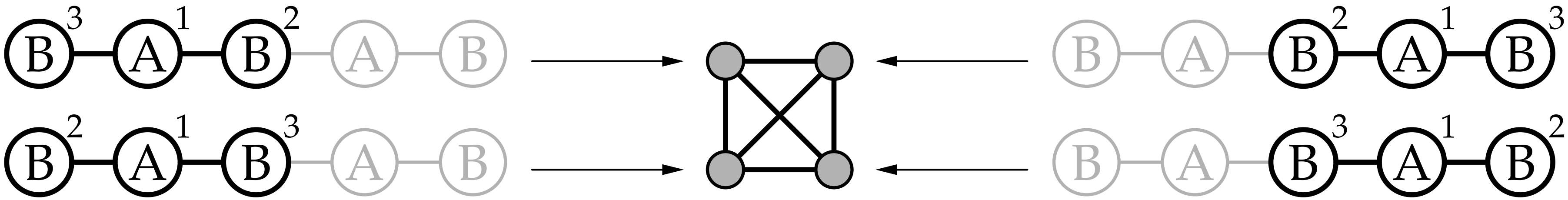
Example:

input graph:



subgraph:





Maximum Independent Set Support

Let $G = (V, E)$ be an (undirected) graph with vertex set V
and edge set $E \subseteq V \times V - \{(v, v) \mid v \in V\}$.

An **independent vertex set** of G is a set $I \subseteq V$ with $\forall u, v \in I : (u, v) \notin E$.

I is a **maximum independent vertex set** (or **MIS** for short) iff

- it is an independent vertex set and
- for all independent vertex sets J of G it is $|I| \geq |J|$.

Notes: Finding a maximum independent vertex set is an NP-complete problem.
However, a greedy algorithm usually gives very good approximations.

Let $O = (V_O, E_O)$ be the overlap graph of the occurrences
of a labeled graph $S = (V_S, E_S, \ell_S)$ in a labeled graph $G = (V_G, E_G, \ell_G)$.

The **maximum independent set support** (or **MIS-support** for short)
of S w.r.t. G is the size of a maximum independent vertex set of O.

Finding a Maximum Independent Set

- Unmark all vertices of the overlap graph.
- **Exact Backtracking Algorithm**
  - Find an unmarked vertex with maximum degree and try two possibilities:
  - Select it for the MIS, that is, mark it as selected and mark all of its neighbors as excluded.
  - Exclude it from the MIS, that is, mark it as excluded.
  - Process remaining vertices recursively and record the best solution found.
- **Heuristic Greedy Algorithm**
  - Select a vertex with the minimum number of unmarked neighbors and mark all of its neighbors as excluded.
  - Process the remaining vertices of the graph recursively.
- In both algorithms vertices with less than two unmarked neighbors can be selected and all of their neighbors marked as excluded.

Anti-Monotonicity of MIS-Support: Preliminaries

Let $G = (V_G, E_G, \ell_G)$ and $S = (V_S, E_S, \ell_S)$ be two labeled graphs.

Let $T = (V_T, E_T, \ell_T)$ be a (non-empty) proper subgraph of S
(that is, $V_T \subset V_S$, $V_T \neq \emptyset$, $E_T = (V_T \times V_T) \cap E_S$, and $\ell_T \equiv \ell_S|_{V_T \cup E_T}$).

Let f be an occurrence of S in G.

An occurrence f' of the subgraph T is called a **T-ancestor** of the occurrence f iff $f' \equiv f|_{V_T}$, that is, if f' coincides with f on the vertex set V_T of T.

Observations:

For given G, S, T and f the T-ancestor f' of the occurrence f is uniquely defined.

Let f_1 and f_2 be two (non-identical, but maybe equivalent) occurrence of S in G.

f_1 and f_2 overlap if there exist overlapping T-ancestors f_1' and f_2'
of the occurrences f_1 and f_2, respectively.

(Note: The inverse implication does not hold generally.)

Anti-Monotonicity of MIS-Support: Proof

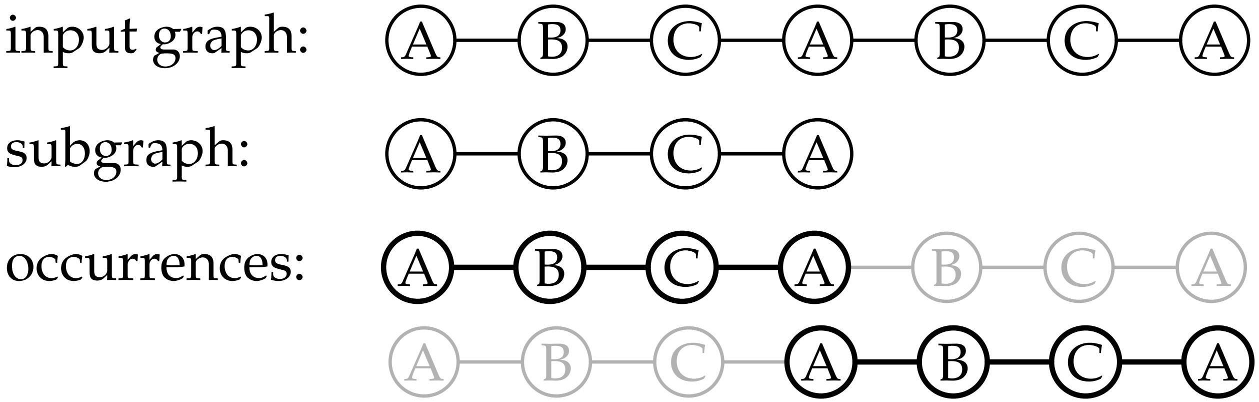
Theorem: MIS-support is anti-monotone.

Proof: We have to show that the MIS-support of a subgraph S w.r.t. a graph G cannot exceed the MIS-support of any (non-empty) proper subgraph T of S.

- Let I be an arbitrary independent vertex set of the overlap graph O of S w.r.t. G.
- The set I induces a subset I' of the vertices of the overlap graph O' of an (arbitrary, but fixed) subgraph T of the considered subgraph S, which consists of the (uniquely defined) T-ancestors of the vertices in I.
- It is $|I| = |I'|$, because no two elements of I can have the same T-ancestor.
- With similar argument: I' is an independent vertex set of the overlap graph O'.
- As a consequence, since I is arbitrary, every independent vertex set of O induces an independent vertex set of O' of the same size.
- Hence the maximum independent vertex set of O' must be at least as large as the maximum independent vertex set of O.

Harmful and Harmless Overlaps of Occurrences

Not all overlaps of occurrences are harmful:



input graph:

subgraph:

occurrences:

Let $G = (V_G, E_G, \ell_G)$ and $S = (V_S, E_S, \ell_S)$ be two labeled graphs and let f_1 and f_2 be two occurrences (subgraph isomorphisms) of S to G.

f_1 and f_2 are called **harmfully overlapping**, written $f_1 \bullet\!\!\bullet f_2$, iff

- they are equivalent or [Fiedler and Borgelt 2007]
- there exists a (non-empty) proper subgraph T of S, so that the T-ancestors f_1' and f_2' of f_1 and f_2, respectively, are equivalent.

Harmful Overlap Graphs and Subgraph Support

Let $G = (V_G, E_G, \ell_G)$ and $S = (V_S, E_S, \ell_S)$ be two labeled graphs and let V_H be the set of all occurrences (subgraph isomorphisms) of S in G.

The **harmful overlap graph** of S w.r.t. G is the graph $H = (V_H, E_H)$, which has the set V_H of occurrences of S in G as its vertex set and the edge set $E_H = \{(f_1, f_2) \mid f_1, f_2 \in V_H \land f_1 \not\equiv f_2 \land f_1 \bullet\!\!\bullet f_2\}$.

Let $H = (V_H, E_H)$ be the harmful overlap graph of the occurrences of a labeled graph $S = (V_S, E_S, \ell_S)$ in a labeled graph $G = (V_G, E_G, \ell_G)$.

The **harmful overlap support** (or **HO-support** for short) of the graph S w.r.t. G is the size of a maximum independent vertex set of H.

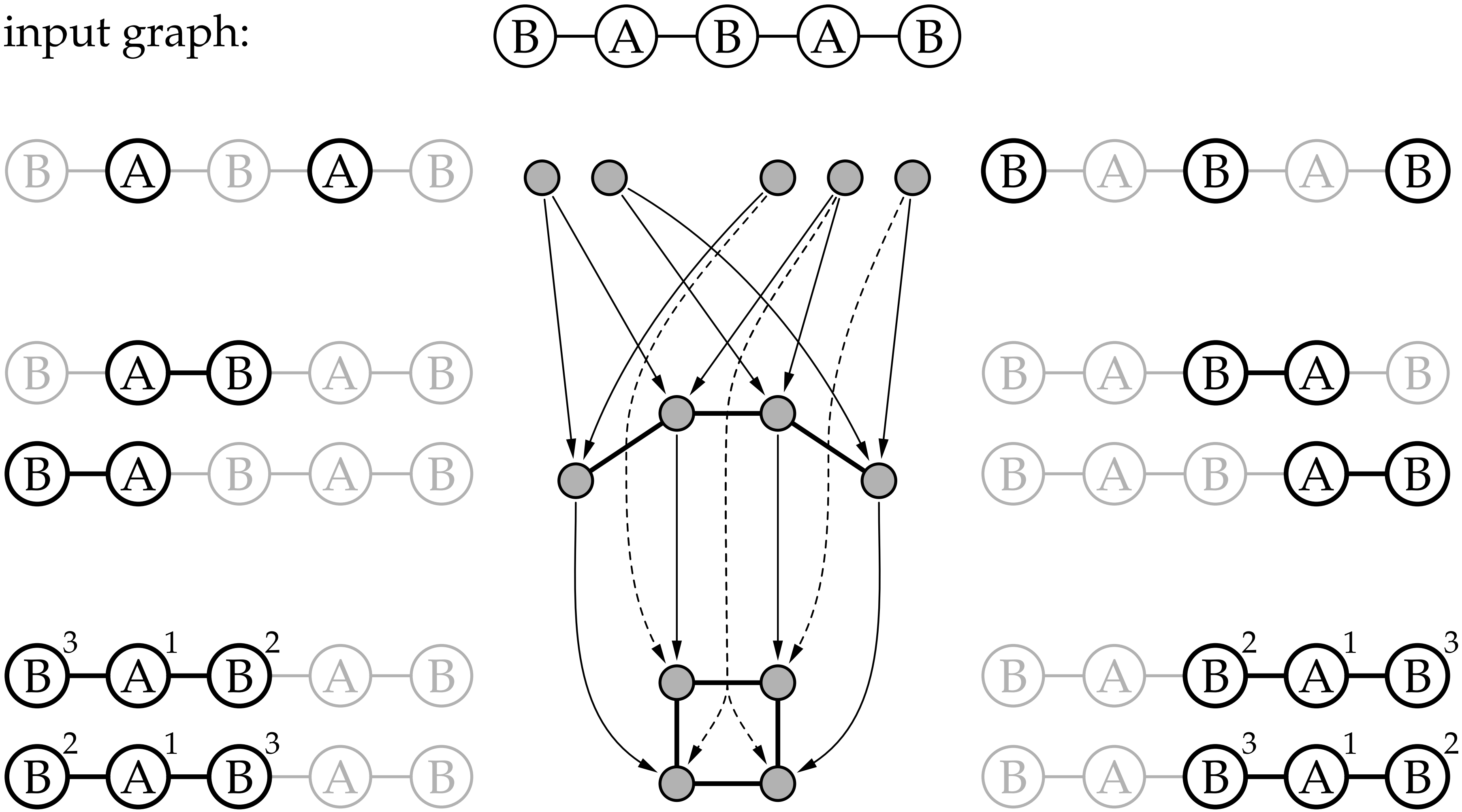
Theorem: HO-support is anti-monotone.

Proof: Identical to proof for MIS-support.

(The same two observations hold, which were all that was needed.)

Harmful Overlap Graphs and Ancestor Relations

input graph:



Subgraph Support Computation

Checking whether two occurrences overlap is easy, but:

How do we check whether two occurrences overlap harmfully?

Core ideas of the harmful overlap test:

- Try to construct a subgraph $S_E = (V_E, E_E, \ell_E)$ that yields equivalent ancestors of two given occurrences f_1 and f_2 of a graph $S = (V_S, E_S, \ell_S)$.
- For such a subgraph S_E the mapping $g : V_E \to V_E$ with $v \mapsto f_2^{-1}(f_1(v))$, where f_2^{-1} is the inverse of f_2, must be a bijective mapping.
- More generally, g must be an **automorphism** of S_E, that is, a subgraph isomorphism of S_E to itself.
- Exploit the properties of automorphisms to exclude vertices from the graph S that cannot be in V_E.

Subgraph Support Computation

Input: Two (different) occurrences f_1 and f_2
of a labeled graph $S = (V_S, E_S, \ell_S)$
in a labeled graph $G = (V_G, E_G, \ell_G)$.

Output: Whether f_1 and f_2 overlap harmfully.

1) Form the sets $V_1 = \{v \in V_G \mid \exists u \in V_S : v = f_1(u)\}$
and $V_2 = \{v \in V_G \mid \exists u \in V_S : v = f_2(u)\}$.

2) Form the sets $W_1 = \{v \in V_S \mid f_1(v) \in V_1 \cap V_2\}$
and $W_2 = \{v \in V_S \mid f_2(v) \in V_1 \cap V_2\}$.

3) If $V_E = W_1 \cap W_2 = \emptyset$, return *false*, otherwise return *true*.

- V_E is the vertex set of a subgraph S_E that induces equivalent ancestors.
- Any vertex $v \in V_S - V_E$ cannot contribute to such equivalent ancestors.
- Hence V_E is a maximal set of vertices for which g is a bijection.

Restriction to Connected Subgraphs

The search for frequent subgraphs is usually restricted to **connected graphs**.

We cannot conclude that no edge is needed if the subgraph S_E is not connected: there may be a connected subgraph of S_E that induces equivalent ancestors of the occurrences f_1 and f_2.

Hence we have to consider subgraphs of S_E in this case.
However, checking all possible subgraphs is prohibitively costly.

Computing the edge set E_E of the subgraph S_E:

1) Let $E_1 = \{(v_1, v_2) \in E_G \mid \exists (u_1, u_2) \in E_S : (v_1, v_2) = (f_1(u_1), f_1(u_2))\}$
   and $E_2 = \{(v_1, v_2) \in E_G \mid \exists (u_1, u_2) \in E_S : (v_1, v_2) = (f_2(u_1), f_2(u_2))\}$.

2) Let $F_1 = \{(v_1, v_2) \in E_S \mid (f_1(v_1), f_1(v_2)) \in E_1 \cap E_2\}$
   and $F_2 = \{(v_1, v_2) \in E_S \mid (f_2(v_1), f_2(v_2)) \in E_1 \cap E_2\}$.

3) Let $E_E = F_1 \cap F_2$.

Restriction to Connected Subgraphs

Lemma: Let $S_C = (V_C, E_C, \ell_C)$ be an (arbitrary, but fixed) connected component of the subgraph S_E and let $W = \{v \in V_C \mid g(v) \in V_C\}$

(reminder: $\forall v \in V_E : g(v) = f_2^{-1}(f_1(v))$, g is an automorphism of S_E)

Then it is either $W = \emptyset$ or $W = V_C$.

Proof: (by contradiction)

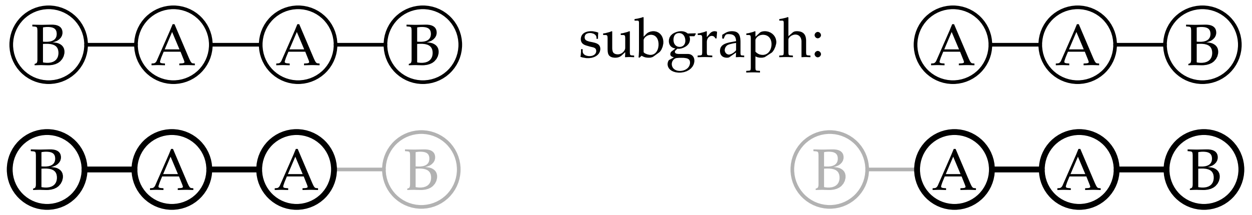
- Suppose that there is a connected component S_C with $W \neq \emptyset$ and $W \neq V_C$.
- Choose two vertices $v_1 \in W$ and $v_2 \in V_C - W$.
- v_1 and v_2 are connected by a path in S_C, since S_C is a connected component. On this path there must be an edge (v_a, v_b) with $v_a \in W$ and $v_b \in V_C - W$.
- It is $(v_a, v_b) \in E_E$ and therefore $(g(v_a), g(v_b)) \in E_E$ (g is an automorphism).
- Since $g(v_a) \in V_C$, it follows $g(v_b) \in V_C$.
- However, this implies $v_b \in W$, contradicting $v_b \in V_C - W$.

Further Optimization

The test can be further optimized by the following simple insight:

- Two occurrences f_1 and f_2 overlap harmfully if $\exists v \in V_S : f_1(v) = f_2(v)$, because then such a vertex v alone gives rise to equivalent ancestors.
- This test can be performed very quickly, so it should be the first step.
- Additional advantage: connected components consisting of isolated vertices can be neglected afterward.

A simple example of harmful overlap *without identical images*:



input graph:

subgraph:

occurrences:

Note that the subgraph inducing equivalent ancestors can be arbitrarily complex even if $\forall v \in V_S : f_1(v) \neq f_2(v)$.

Final Procedure for Harmful Overlap Test

Input: Two (different) occurrences f_1 and f_2
of a labeled graph $S = (V_S, E_S, \ell_S)$
in a labeled graph $G = (V_G, E_G, \ell_G)$.

Output: Whether f_1 and f_2 overlap harmfully.

1) If $\exists v \in S : f_1(v) = f_2(v)$, return *true*.

2) Form the edge set E_E of the subgraph S_E (as described above) and
form the (reduced) vertex set $V'_E = \{v \in V_S \mid \exists u \in V_S : (v, u) \in E_E\}$.
(Note that V'_E does not contain isolated vertices.)

3) Let $S^i_C = (V^i_C, E^i_C), 1 \leq i \leq n,$
be the connected components of $S'_E = (V'_E, E_E)$.

   If $\exists i; 1 \leq i \leq n : \exists v \in V^i_C : g(v) = f_2^{-1}(f_1(v)) \in V^i_C,$
   return *true*, otherwise return *false*.

Alternative: Minimum Number of Vertex Images

Let $G = (V_G, E_G, \ell_G)$ and $S = (V_S, E_S, \ell_S)$ be two labeled graphs and let F be the set of all subgraph isomorphisms of S to G. Then the **minimum number of vertex images support** (or **MNI-support** for short) of S w.r.t. G is defined as

$$\min_{v \in V_S} \left|\{u \in V_G \mid \exists f \in F : f(v) = u\}\right|.$$

[Bringmann and Nijssen 2007]

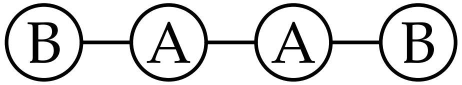
Advantage:

- Can be computed much more efficiently than MIS- or HO-support. (No need to determine a maximum independent vertex set.)

Disadvantage:

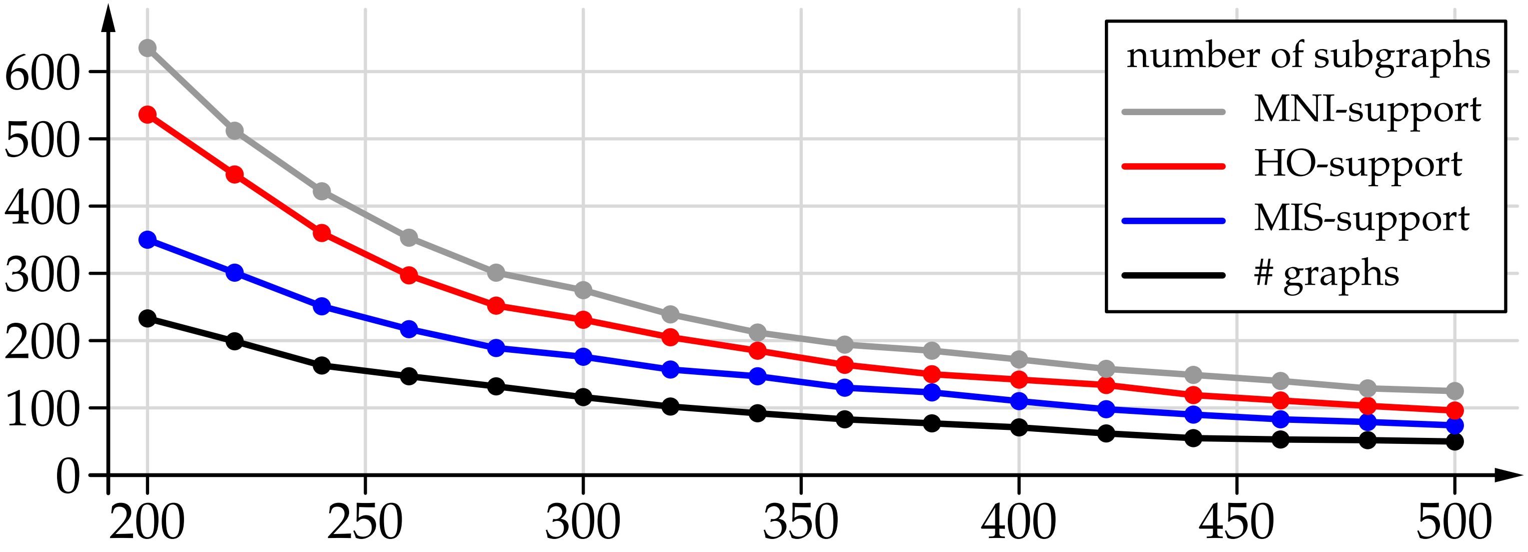
- Often counts both of two equivalent occurrences. (Fairly unintuitive behavior.)

Example:

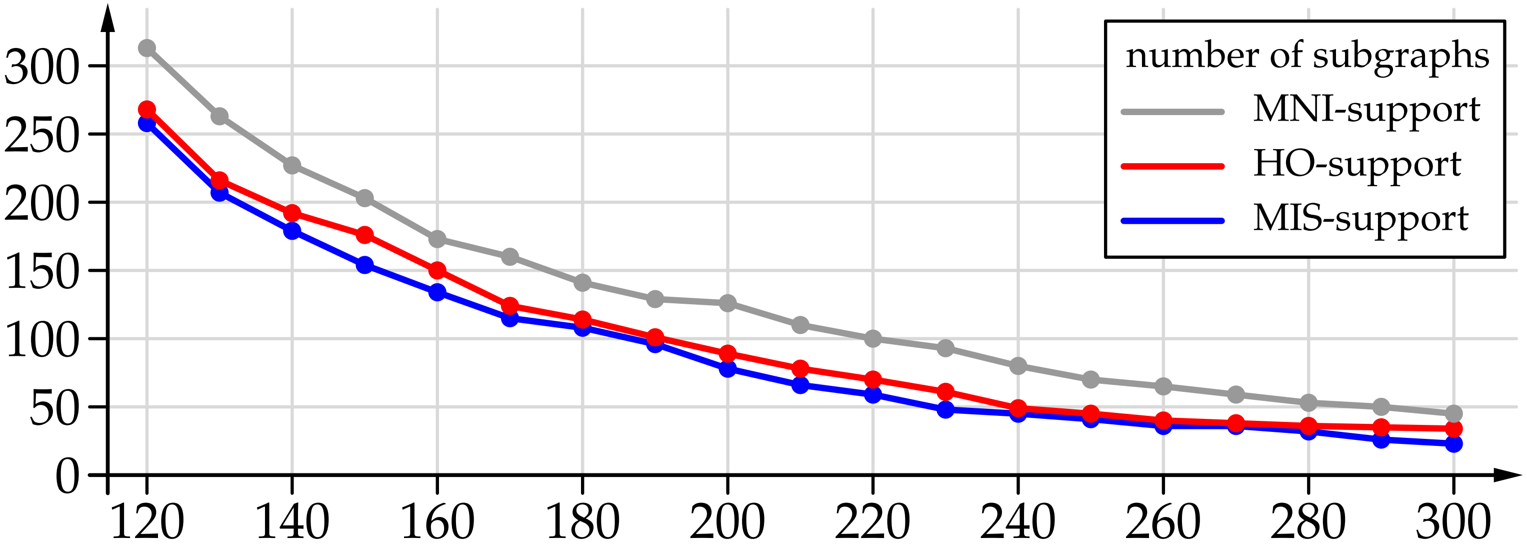


Experimental Results

Index
Chemicus
1993



Tic-
Tac-
Toe
not win



Summary

- Defining subgraph support in the single graph setting:
  maximum independent vertex set of an overlap graph of the occurrences.

- **MIS-support is anti-monotone**.
  Proof: look at induced independent vertex sets for substructures.

- Definition of **harmful overlap support** of a subgraph:
  existence of equivalent ancestor occurrences.

- Simple procedure for testing whether two occurrences overlap harmfully.

- **Harmful overlap support is anti-monotone**.

- Restriction to connected substructures and optimizations.

- Alternative: **minimum number of vertex images**.

- **Software:** `http://www.borgelt.net/moss.html`

Frequent Sequence Mining

Frequent Sequence Mining

- **Directed versus undirected sequences**
  - Temporal sequences, for example, are always directed.
  - DNA sequences can be undirected (both directions can be relevant).

- **Multiple sequences versus a single sequence**
  - Multiple sequences: purchases with rebate cards, web server accesses.
  - Single sequence: alarms in telecommunication networks.

- **(Time) points versus time intervals**
  - Points: DNA sequences, alarms in telecommunication networks.
  - Intervals: weather data, movement analysis (sports medicine).
  - Further distinction: one object per (time) point versus multiple objects.

Frequent Sequence Mining

- **Consecutive subsequences versus subsequences with gaps**
  - $a\ c\ b\ a\ b\ c\ b\ a$ always counts as a subsequence *abc*.
  - $a\ c\ b\ a\ b\ c\ b\ c$ may not always count as a subsequence *abc*.

- **Existence of an occurrence versus counting occurrences**
  - Combinatorial counting (all occurrences)
  - Maximal number of disjoint occurrences
  - Temporal support (number of time window positions)
  - Minimum occurrence (smallest interval)

- **Relation between the objects in a sequence**

○ items:	only precede and succeed
○ labeled time points:	$t_1 < t_2$, $t_1 = t_2$, and $t_1 > t_2$
○ labeled time intervals:	relations like *before*, *starts*, *overlaps*, *contains* etc.

Frequent Sequence Mining

- **Directed sequences** are easier to handle:
  - The (sub)sequence itself can be used as a code word.
  - As there is only one possible code word per sequence (only one direction), this code word is necessarily canonical.

- **Consecutive subsequences** are easier to handle:
  - There are fewer occurrences of a given subsequence.
  - For each occurrence there is exactly one possible extension.
  - This allows for specialized data structures (similar to an FP-tree).

- **Item sequences** are easiest to handle:
  - There are only two possible relations and thus patterns are simple.
  - Other sequences are handled with state machines for occurrence tests.

A Canonical Form for Undirected Sequences

- If the sequences to mine are not directed, a subsequence can *not* be used as its own code word, because it does *not* have the **prefix property**.

- The reason is that an undirected sequence can be read forward or backward, which gives rise to two possible code words, the smaller (or the larger) of which may then be defined as the **canonical code word**.

- Examples (that the prefix property is violated):
  - Assume that the item order is $a < b < c \ldots$ and that the lexicographically smaller code word is the canonical one.
  - The sequence *bab*, which is canonical, has the prefix *ba*, but the canonical form of the sequence *ba* is rather *ab*.
  - The sequence *cabd*, which is canonical, has the prefix *cab*, but the canonical form of the sequence *cab* is rather *bac*.

- As a consequence, we have to look for a different way of forming code words (at least if we want the coding scheme to have the prefix property).

A Canonical Form for Undirected Sequences

- A (simple) possibility to form canonical code words having the prefix property is to handle (sub)sequences of **even and odd length separately**.

  In addition, **forming the code word is started in the middle**.

- **Even length:** The sequence $a_m\ a_{m-1}\ \ldots\ a_2\ a_1\ b_1\ b_2\ \ldots\ b_{m-1}\ b_m$
  is described by the code word $a_1\ b_1\ a_2\ b_2\ \ldots\ a_{m-1}\ b_{m-1}\ a_m\ b_m$
  or by the code word $b_1\ a_1\ b_2\ a_2\ \ldots\ b_{m-1}\ a_{m-1}\ b_m\ a_m$.

- **Odd length:** The sequence $a_m\ a_{m-1}\ \ldots\ a_2\ a_1\ a_0\ b_1\ b_2\ \ldots\ b_{m-1}\ b_m$
  is described by the code word $a_0\ a_1\ b_1\ a_2\ b_2\ \ldots\ a_{m-1}\ b_{m-1}\ a_m\ b_m$
  or by the code word $a_0\ b_1\ a_1\ b_2\ a_2\ \ldots\ b_{m-1}\ a_{m-1}\ b_m\ a_m$.

- The lexicographically smaller of the two code words is the **canonical code word**.

- Such sequences are **extended** by adding a pair $a_{m+1}\ b_{m+1}$ or $b_{m+1}\ a_{m+1}$, that is, by adding one item at the front and one item at the end.

A Canonical Form for Undirected Sequences

The code words defined in this way clearly have the **prefix property**:

- Suppose the prefix property would *not* hold.

  Then there exists, without loss of generality, a canonical code word

  $$w_m \quad = a_1\ b_1\ a_2\ b_2\ \ldots\ a_{m-1}\ b_{m-1}\ a_m\ b_m,$$

  the prefix w_{m-1} of which is not canonical, where

  $$w_{m-1} = a_1\ b_1\ a_2\ b_2\ \ldots\ a_{m-1}\ b_{m-1},$$

- As a consequence, we have $v_m > w_m$, where

  $$v_m \quad = b_1\ a_1\ b_2\ a_2\ \ldots\ b_{m-1}\ a_{m-1}\ b_m\ a_m,$$

  and $v_{m-1} < w_{m-1}$, where

  $$v_{m-1} = b_1\ a_1\ b_2\ a_2\ \ldots\ b_{m-1}\ a_{m-1}.$$

- However, $v_{m-1} < w_{m-1}$ implies $v_m < w_m$,
  because v_{m-1} is a prefix of v_m and w_{m-1} is a prefix of w_m,
  but $v_m < w_m$ contradicts $v_m > w_m$.

A Canonical Form for Undirected Sequences

- Generating and comparing the two possible code words takes *linear time*. However, it can be improved by maintaining an additional piece of information.

- For each sequence a **symmetry flag** is computed:

$$s_m = \bigwedge_{i=1}^{m} (a_i = b_i)$$

- The symmetry flag can be maintained in constant time with

$$s_{m+1} = s_m \ \wedge \ (a_{m+1} = b_{m+1}).$$

- The **permissible extensions** depend on the symmetry flag:
  - ○ if $s_m =$ true, it must be $a_{m+1} \leq b_{m+1}$ for the result to be canonical;
  - ○ if $s_m =$ false, any relation between a_{m+1} and b_{m+1} is acceptable.

- This rule guarantees that exactly the canonical extensions are created. Applying this rule to check a candidate extension takes **constant time**.

Sequences of Time Intervals

- A (labeled or attributed) **time interval** is a triplet $I = (s, e, l)$, where s is the start time, e is the end time and l is the associated label.

- A **time interval sequence** is a set of (labeled) time intervals, of which we assume that they are maximal in the sense that for two intervals $I_1 = (s_1, e_1, l_1)$ and $I_2 = (s_2, e_2, l_2)$ with $l_1 = l_2$ we have either $e_1 < s_2$ or $e_2 < s_1$. Otherwise they are merged into one interval $I = (\min\{s_1, s_2\}, \max\{e_1, e_2\}, l_1)$.

- A **time interval sequence database** is a tuple of time interval sequences.

- Time intervals can easily be ordered as follows:

  Let $I_1 = (s_1, e_1, l_1)$ and $I_2 = (s_2, e_2, l_2)$ be two time intervals. It is $I_1 \prec I_2$ iff

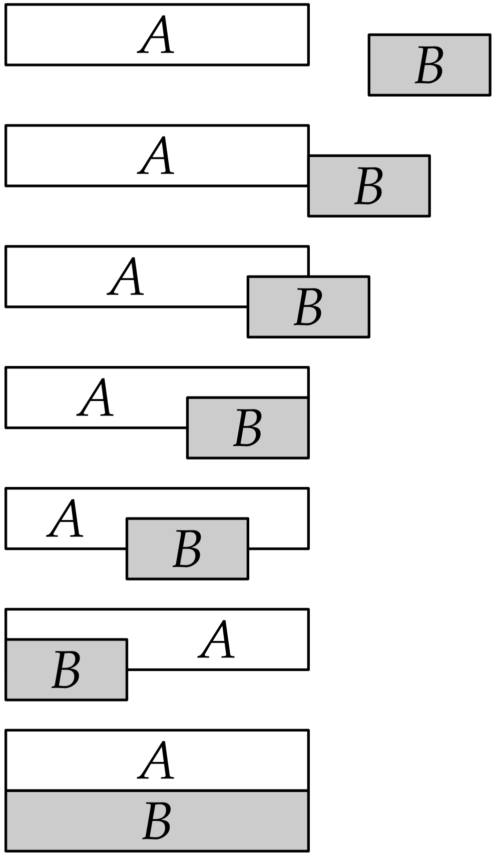
  - $s_1 < s_2$ or
  - $s_1 = s_2$ and $e_1 < e_2$ or
  - $s_1 = s_2$ and $e_1 = e_2$ and $l_1 < l_2$.

  Due to the assumption made above, at least the third option must hold.

Allen’s Interval Relations

Due to their temporal extension, time intervals allow for several different relations.

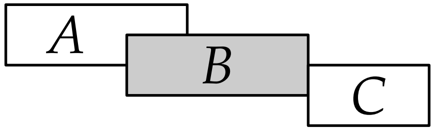
A commonly used set of relations between time intervals are **Allen’s interval relations**. [Allen 1983]



A before B	B after A
A meets B	B is met by A
A overlaps B	B is overlapped by A
A is finished by B	B finishes A
A contains B	B during A
A is started by B	B starts A
A equals B	B equals A

Temporal Interval Patterns

- A temporal pattern must specify the relations between all referenced intervals. This can conveniently be done with a matrix:



$$\begin{array}{cccc} & A & B & C \\ A & \mathrm{e} & \mathrm{o} & \mathrm{b} \\ B & \mathrm{io} & \mathrm{e} & \mathrm{m} \\ C & \mathrm{a} & \mathrm{im} & \mathrm{e} \end{array}$$

- Such a temporal pattern matrix can also be interpreted as an adjacency matrix of a graph, which has the interval relationships as edge labels.

- Generally, the input interval sequences may be represented as such graphs, thus mapping the problem to frequent (sub)graph mining.

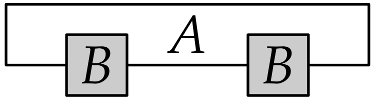
- However, the relationships between time intervals are constrained (for example, "*B* after *A*" and "*C* after *B*" imply "*C* after *A*"). These constraints can be exploited to obtain a simpler canonical form.

- In the **canonical form**, the intervals are assigned in increasing time order to the rows and columns of the temporal pattern matrix. [Kempe 2008]

Support of Temporal Patterns

- The support of a temporal pattern w.r.t. a single sequence can be defined by:
  - ○ Combinatorial counting (all occurrences)
  - ○ Maximal number of disjoint occurrences
  - ○ Temporal support (number of time window positions)
  - ○ Minimum occurrence (smallest interval)

- However, several of these definitions suffer from the fact that such support is not *anti-monotone* or *downward closed*:



The support of "A contains B" is 2, but the support of "A" is only 1.

- Nevertheless an exhaustive pattern search can be ensured, without having to abandon pruning with the **Apriori property**.

  The reason is that with minimum occurrence counting the relation "contains" is the only one that can lead to support anomalies like the one shown above.

Weakly Anti-Monotone / Downward Closed

- Let $\mathcal{P}$ be a pattern space with a (proper) subpattern relationship $\sqsubset$ and let s be a function from $\mathcal{P}$ to the real numbers, $s : \mathcal{P} \to \mathbb{R}$.

  For a pattern $S \in \mathcal{P}$, let $P(S) = \{R \mid R \sqsubset S \ \wedge \ \nexists Q : R \sqsubset Q \sqsubset S\}$ be the set of all *parent patterns* of S.

  The function s on the pattern space $\mathcal{P}$ is called

  ○ **strongly anti-monotone** or **strongly downward closed** iff

  $$\forall S \in \mathcal{P} : \forall R \in P(S) : \quad s(R) \geq s(S),$$

  ○ **weakly anti-monotone** or **weakly downward closed** iff

  $$\forall S \in \mathcal{P} : \exists R \in P(S) : \quad s(R) \geq s(S).$$

- The support of temporal interval patterns is weakly anti-monotone (at least) if it is computed from minimal occurrences.

- If temporal interval patterns are extended *backward in time*, the **Apriori property** can safely be used for pruning. [Kempe 2008]

Summary Frequent Sequence Mining

- Several different **types of frequent sequence mining** can be distinguished:
  - single and multiple sequences, directed and undirected sequences
  - items versus (labeled) intervals, single and multiple objects per position
  - relations between the objects, definition of pattern support

- All common types of frequent sequence mining possess canonical forms for which **canonical extension rules** can be found.

  With these rules it is possible to check in constant time whether a possible extension leads to a result in canonical form.

- A **weakly anti-monotone** support function can be enough to allow pruning with the **Apriori property**.

  However, in this case it must be made sure that the canonical form assigns an appropriate parent pattern in order to ensure an exhaustive search.

Frequent Tree Mining

Frequent Tree Mining: Basic Notions

- Reminder: A **path** is a sequence of edges connecting two vertices in a graph.
- Reminder: A (labeled) graph G is called a **tree** iff for any pair of vertices in G there exists *exactly one path* connecting them in G.
- A tree is called **rooted** if it has a distinguished vertex, called the **root**.

  Rooted trees are often seen as directed: edges are directed *away* from the root.
- If a tree is not rooted (that is, if there is no distinguished vertex), it is called **free**.
- A tree is called **ordered** if for each vertex there exists an order on its incident edges.

  If the tree is *rooted*, the order may be defined on the outgoing edges only.
- Trees of whichever type are much easier to handle than frequent (sub)graphs, because it is mainly the cycles (which may be present in a general graph) that make it difficult to construct the canonical code word.

Frequent Tree Mining: Basic Notions

- Reminder: A **path** is a sequence of edges connecting two vertices in a graph.
- The **length of a path** is the number of its edges.
- The **distance** between two vertices of a graph G is the length of a shortest path connecting them.

  Note that in a tree there is exactly one path connecting two vertices, which is then necessarily also the shortest path.
- In a rooted tree the **depth** of a vertex is its distance from the root vertex.

  The root vertex itself has depth 0.

  The **depth** of a tree is the depth of its deepest vertex.
- The **diameter** of a graph is the largest distance between any two vertices.
- A **diameter path** of a graph is a path having a length that is the diameter of the graph.

Rooted Ordered Trees

- For **rooted ordered trees** code words derived from spanning trees can directly be used: the spanning tree is simply the tree itself.

- However, the **root** of the spanning tree **is fixed**: it is simply the root of the rooted ordered tree.

- In addition, the **order of the children** of each vertex is **fixed**: it is simply the given order of the outgoing edges.

- As a consequence, once a traversal order for the spanning tree is fixed (for example, depth-first or a breadth-first traversal), there is only **one possible code word**, which is necessarily the canonical code word.

- Therefore **rightmost path extension** (for a depth-first traversal) and **maximum source extension** (for a breadth-first traversal) obviously provide a canonical extension rule for rooted ordered trees.

  There is no need for an explicit test for canonical form.

Rooted Unordered Trees

- **Rooted unordered trees** can most conveniently be described by so-called **preorder code words**.

- Preorder code words are closely related to spanning trees that are constructed with a depth-first search, because a preorder traversal is a depth-first traversal. However, their special form makes it easier to compare code words for subtrees.

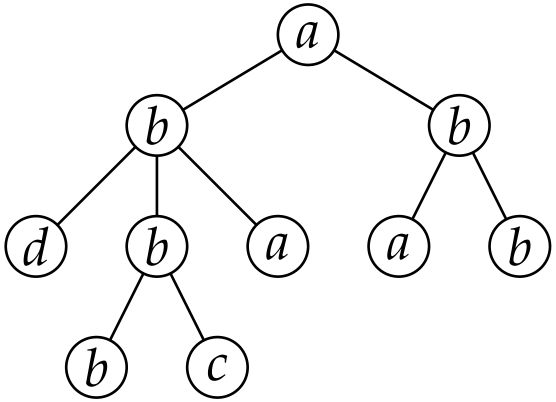
- The preorder code words we consider here have the general form

$$a\ (\ d\ b\ a\ )^m,$$

where m is the number of edges of the tree, $m = n - 1$,
n is the number of vertices of the tree,
a is a vertex attribute / label,
b is an edge attribute / label, and
d is the depth of the source vertex of an edge.

The source vertex of an edge is the one that is closer to the root (smaller depth). The edges are listed in the order in which they are visited in a preorder traversal.

Rooted Unordered Trees



For simplicity we omit edge labels. In rooted trees edge labels can always be combined with the destination vertex label (that is, the label of the vertex that is farther away from the root).

- The above rooted unordered tree can be described by the code word

```
a 0b 1d 1b 2b 2c 1a 0b 1a 1b
```

- Note that the code word consists of substrings that describe the subtrees:

$$\overbrace{\texttt{a}\ \texttt{0}\ \overbrace{\texttt{b}\ \texttt{1}\ \overbrace{\texttt{d}}\ \texttt{1}\ \overbrace{\texttt{b}\ \texttt{2}\ \overbrace{\texttt{b}}\ \texttt{2}\ \overbrace{\texttt{c}}}\ \texttt{1}\ \overbrace{\texttt{a}}}\ \texttt{0}\ \overbrace{\texttt{b}\ \texttt{1}\ \overbrace{\texttt{a}}\ \texttt{1}\ \overbrace{\texttt{b}}}}$$

The subtree strings are separated by a number stating the depth of the parent.

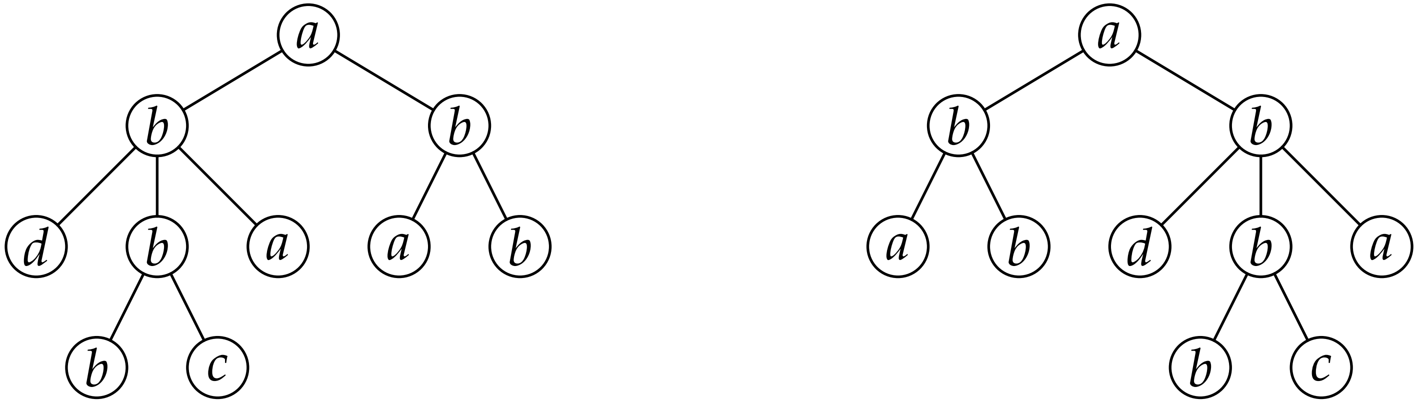
Rooted Unordered Trees

Exchanging code words on the same level exchanges branches/subtrees.

$$\overbrace{\mathtt{a}\ \mathtt{0}\ \overbrace{\mathtt{b}\ \mathtt{1}\ \overbrace{\mathtt{d}}\ \mathtt{1}\ \overbrace{\mathtt{b}\ \mathtt{2}\ \overbrace{\mathtt{b}}\ \mathtt{2}\ \overbrace{\mathtt{c}}}\ \mathtt{1}\ \overbrace{\mathtt{a}}}\ \mathtt{0}\ \overbrace{\mathtt{b}\ \mathtt{1}\ \overbrace{\mathtt{a}}\ \mathtt{1}\ \overbrace{\mathtt{b}}}}$$

For example, in this code word the children of the root are exchanged:

$$\overbrace{\mathtt{a}\ \mathtt{0}\ \overbrace{\mathtt{b}\ \mathtt{1}\ \overbrace{\mathtt{a}}\ \mathtt{1}\ \overbrace{\mathtt{b}}}\ \mathtt{0}\ \overbrace{\mathtt{b}\ \mathtt{1}\ \overbrace{\mathtt{d}}\ \mathtt{1}\ \overbrace{\mathtt{b}\ \mathtt{2}\ \overbrace{\mathtt{b}}\ \mathtt{2}\ \overbrace{\mathtt{c}}}\ \mathtt{1}\ \overbrace{\mathtt{a}}}}$$



Rooted Unordered Trees

- All possible preorder code words can be obtained from one preorder code word by exchanging substrings of the code word that describe sibling subtrees.

  (This shows the advantage of the vertex depth compared to the vertex index: no renumbering of the vertices is necessary in such a exchange.)

- By defining an (arbitrary, but fixed) order on the vertex labels and using the standard order of the integer numbers, the code words can be compared lexicographically.

  (Note that vertex labels are always compared to vertex labels and integers to integers, because these two elements alternate.)

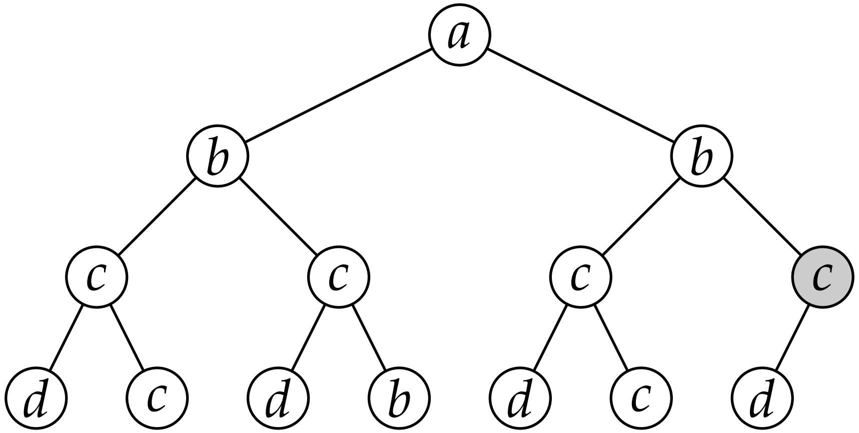
- Contrary to the common definition used in all earlier cases, we define the lexicographically *greatest* code word as the **canonical code word**.

- The canonical code word for the tree on the previous slides is

```
a 0b 1d 1b 2c 2b 1a 0b 1b 1a
```

Rooted Unordered Trees

- In order to understand the core problem of obtaining an extension rule for rooted unordered trees, consider the following tree:



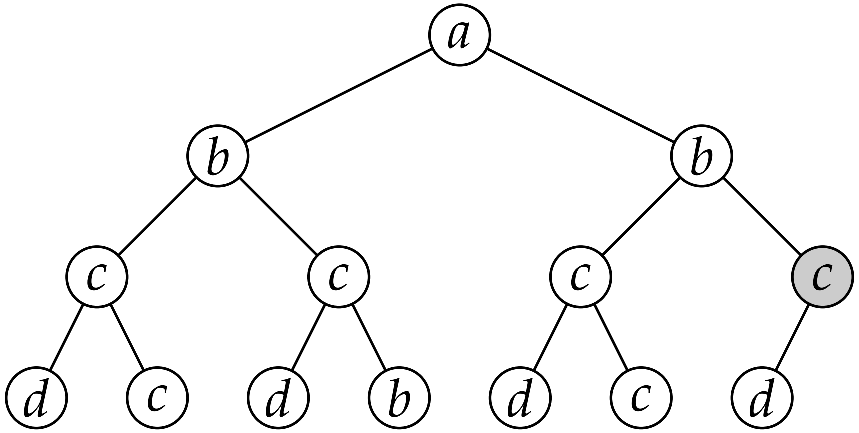
- The canonical code word results from the shown order of the subtrees:

```
a 0b 1c 2d 2c 1c 2d 2b 0b 1c 2d 2c 1c 2d
```

  Any exchange of subtrees leads to a lexicographically *smaller* code word.

- How can this tree be extended by adding a child to the gray vertex?
  That is, what label may the child vertex have if the result is to be canonical?

Rooted Unordered Trees



- To begin with, we observe that the child must not have a label succeeding “d”, because otherwise exchanging the new vertex with the other child of the gray vertex would yield a lexicographically *larger* code word:

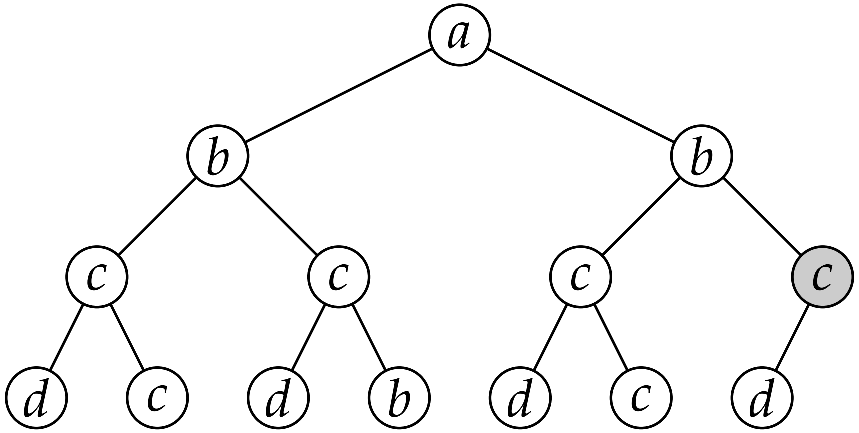
```
a 0b 1c 2d 2c 1c 2d 2b 0b 1c 2d 2c 1c 2d 2e
```

$<$

```
a 0b 1c 2d 2c 1c 2d 2b 0b 1c 2d 2c 1c 2e 2d
```

- The children of a vertex must be sorted descendingly w.r.t. their labels.

Rooted Unordered Trees



- Secondly, we observe that the child must not have a label succeeding “c”, because otherwise exchanging the subtrees of the parent of the gray vertex would yield a lexicographically *larger* code word:

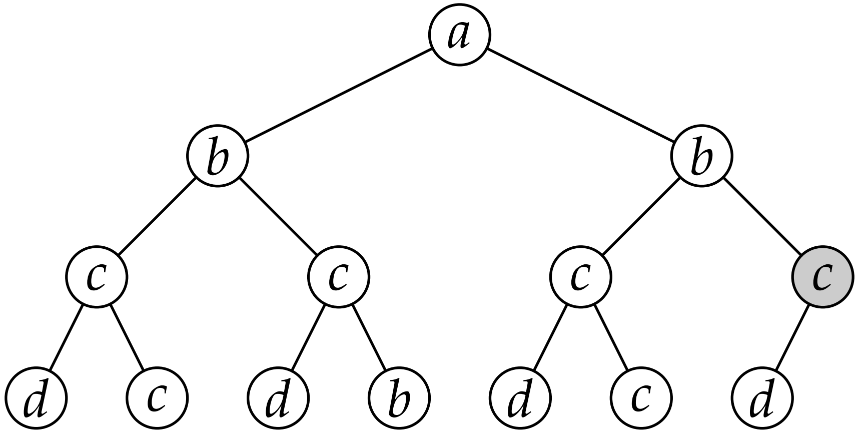
```
a 0b 1c 2d 2c 1c 2d 2b 0b 1c 2d 2c 1c 2d 2d
```

$$<$$

```
a 0b 1c 2d 2c 1c 2d 2b 0b 1c 2d 2d 1c 2d 2c
```

- The subtrees of any vertex must be sorted descendingly w.r.t. their code words.

Rooted Unordered Trees



- Thirdly, we observe that the child must not have a label succeeding “b”, because otherwise exchanging the subtrees of the root vertex of the tree would yield a lexicographically *larger* code word:

```
a  0b 1c 2d 2c 1c 2d 2b 0b 1c 2d 2c 1c 2d 2c
                       <
a  0b 1c 2d 2c 1c 2d 2c 0b 1c 2d 2c 1c 2d 2b
```

- The subtrees of any vertex must be sorted descendingly w.r.t. their code words.

Rooted Unordered Trees

- That a possible exchange of subtrees at vertices closer to the root never yields looser restrictions is no accident.

- Suppose a rooted tree is described by a canonical code word

  $$\texttt{a 0 b 1 } w_1 \texttt{ 1 } w_2 \texttt{ 0 b 1 } w_3 \texttt{ 1 } w_4.$$

  Then we know the following relationships between subtree code words:

  - $w_1 \geq w_2$ and $w_3 \geq w_4$, because otherwise an exchange of subtrees at the vertices labeled with “b” would lead to a lexicographically larger code word.
  - $w_1 \geq w_3$, because otherwise an exchange of subtrees at the vertex labeled “a” would lead to a lexicographically larger code word.

- Only if $w_1 = w_3$, the code words w_1 and w_3 do not already determine the order of the subtrees of the vertex labeled with “a”. In this case we have $w_2 \geq w_4$.

  However, then we also have $w_3 = w_1 \geq w_2$,
  showing that w_2 provides no looser restriction of w_4 than w_3.

Rooted Unordered Trees

As a consequence, we obtain the following simple **extension rule**:

- Let w be the canonical code word of the rooted tree to extend and let d be the depth of the rooted tree (that is, the depth of the deepest vertex). In addition, let the considered extension be xa with $x \in \mathbb{N}_0$ and a a vertex label.

- Let y be the smallest integer for which w has a suffix of the form $y\, w_1 w_2\, y\, w_1$ with $y \in \mathbb{N}_0$ and w_1 and w_2 strings not containing any $y' \leq y$ (w_2 may be empty). If w does not possess such a suffix, let $y = d$ (depth of the tree).

- If $x > y$, the extension is canonical if and only if $xa \leq w_2$.

- If $x \leq y$, check whether w has a suffix xw_3, where w_3 is a string not containing any integer $x' \leq x$.

  If w has such a suffix, the extended code word is canonical if and only if $a \leq w_3$.

  If w does not have such a suffix, the extended code word is always canonical.

- With this extension rule no subsequent canonical form test is needed.

Rooted Unordered Trees

The discussed extension rule is very efficient:

- Comparing the elements of the extension takes **constant time**
  (at most one integer and one label need to be compared).

- Knowledge of the strings w_3 for all possible values of x $(0 \leq x < d)$
  can maintained in **constant time**:

  It suffices to record the starting points of the substrings
  that describe the rightmost subtree on each tree level.
  At most one of these starting points can change with an extension.

- Knowledge of the value of y and the two starting points of the string w_1 in w
  can be maintained in **constant time**:

  As long as no two sibling vertices carry the same label, it is $y = d$.
  If a sibling with the same label is added, y is set to the depth of the parent.
  $w_1 = a$ occurs at the position of the w_3 for y and at the extension vertex label.
  If a future extension differs from w_2, it is $y = d$, otherwise w_1 is extended.

Free Trees

- **Free trees** can be handled by combining the ideas of how to handle *sequences* and *rooted unordered trees*.

- Similar to sequences, free trees of even and odd diameter are treated separately.

- General ideas for a canonical form for free trees:

  - **Even Diameter:**
    The vertex in the middle of a diameter path is uniquely determined.
    This vertex can be used as the root of a rooted tree.

  - **Odd Diameter:**
    The edge in the middle of a diameter path is uniquely determined.
    Removing this edge splits the free tree into two rooted trees.

- Procedure for growing free trees:

  - First grow a diameter path using the canonical form for sequences.

  - Extend the diameter path into a tree by adding branches.

Free Trees

- Main problem of the procedure for growing free trees:

  The initially grown diameter path must remain identifiable.

  (Otherwise the *prefix property* cannot be guaranteed.)

- In order to solve this problem it is exploited that in the canonical code word for a rooted unordered tree code words describing paths from the root to a leaf vertex are lexicographically increasing if the paths are listed from left to right.

- **Even Diameter:**
  The original diameter path represents two paths from the root to two leaves.
  To keep them identifiable, these paths must be the lexicographically smallest and the lexicographically largest path leading to this depth.

- **Odd Diameter:**
  The original diameter path represents one path from the root to a leaf in each of the two rooted trees the free tree is split into.
  These paths must be the lexicographically smallest paths leading to this depth.

Summary Frequent Tree Mining

- **Rooted ordered trees**
  - The root is fixed and the order of the children of each vertex is fixed.
  - Both *rightmost path extension* and *maximum source extension* obviously provide a canonical extension rule for rooted ordered trees.
- **Rooted unordered trees**
  - The root is fixed, but there is no order of the children.
  - There exists a canonical extension rule based on sorted preorder strings (constant time for finding allowed extensions). [Luccio et al. 2001, 2004]
- **Free trees**
  - No vertex is fixed as the root, there is no order on adjacent vertices.
  - There exists a canonical extension rule based on depth sequences (constant time for finding allowed extensions) [Nijssen and Kok 2004]

Summary Frequent Pattern Mining

Summary Frequent Pattern Mining

- Possible types of patterns: **item sets**, **sequences**, **trees**, and **graphs**.
- A core ingredient of the search is a **canonical form** of the type of pattern.
  - Purpose: ensure that each possible pattern is processed at most once. (Discard non-canonical code words, process only canonical ones.)
  - It is desirable that the canonical form possesses the **prefix property**.
  - Except for general graphs there exist **canonical extension rules**.
  - For general graphs, **restricted extensions** allow to reduce the number of actual canonical form tests considerably.
- Frequent pattern mining algorithms prune with the **Apriori property**:

$$\forall P : \forall S \supset P : \quad s_{\mathcal{D}}(P) < s_{\min} \rightarrow s_{\mathcal{D}}(S) < s_{\min}.$$

  That is: **No super-pattern of an infrequent pattern is frequent.**

- **Additional filtering** is important to single out the relevant patterns.

Software

Software for frequent pattern mining can be found at

- my web site: `http://www.borgelt.net/fpm.html`

○ Apriori	`http://www.borgelt.net/apriori.html`
○ Eclat	`http://www.borgelt.net/eclat.html`
○ FP-Growth	`http://www.borgelt.net/fpgrowth.html`
○ RElim	`http://www.borgelt.net/relim.html`
○ SaM	`http://www.borgelt.net/sam.html`
○ MoSS	`http://www.borgelt.net/moss.html`

- the Frequent Item Set Mining Implementations (FIMI) Repository
  `http://fimi.ua.ac.be/`

  This repository was set up with the contributions to the FIMI workshops in 2003 and 2004, where each submission had to be accompanied by the source code of an implementation. The web site offers all source code, several data sets, and the results of the competition.