Deep of Enumeration Algorithms

1. Implementation and Data-Driven Speeding Up

- Motivation and situation
- Frequent itemset mining
- Maximal clique enumeration

Enumeration is Already Efficient

- Enumeration algorithms we have seen are output polynomial time, linear in output size in particular

- On the other hand, enumeration algorithms output exponentially many solutions, so we might think that the problem sizes are usually small (up to 100)

- …so, "input size is constant" would be valid
  and thus enumeration algorithms are optimal.
  ("less than 100" is constant)
  □ this would be true in "theoretical sense"

- …however, there exist other kinds of applications

Big Data Applications

- In practice, enumeration is widely used in data mining / data engineering area

  - frequent pattern mining, candidate enumeration, community mining, feasible solution enumeration…

- In such areas, input data is often big data

- Indeed, #solutions is small, often **O(n)** to **O(n2)**
  thus, actually **"tractable large-scale problems"**

Why #Solutions is Small?

- …#solutions seem to easily increase to exponential, however…

  + if exponentially many solutions, many solutions are similar, thus quite redundant

  + We don't want to have such many solutions!
    □ they are intractable (too long time for post process)

  + Even though #solutions is huge, the modeling was bad, from the beginning

Ex) #Maximal cliques in the large sparse graphs are not huge, but #independent sets (no vertex pair is connected) are extremely huge

Constant Time Enumeration

- …so, enumeration should take short time per solution
  - in particular, constant time for each

- However, handling big data in constant in an iteration is hard
  - we need techniques to compute without looking the whole data

+ data structure for dynamic computation, data compression to unify the operation, ancestor-processing for reducing descendants…

- Further, engineering techniques help the improvements
  + memory saving
  + make the computation fitting to the architecture

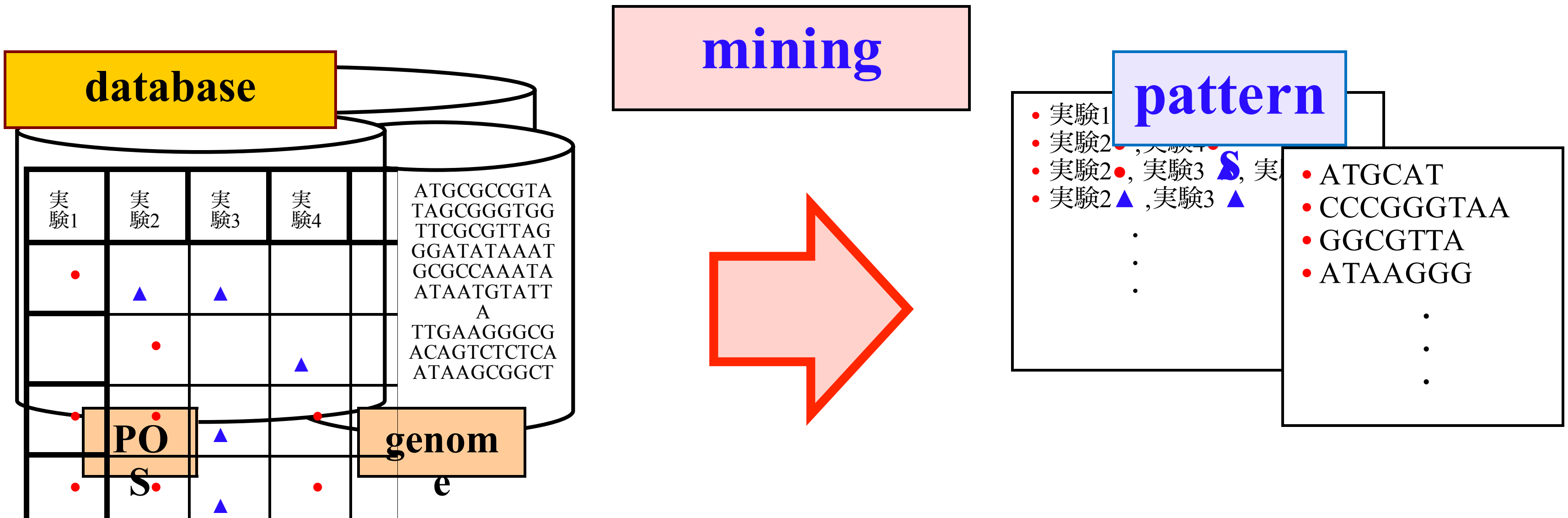
See the techniques in itemset mining and clique enumeration

3-1 Frequent Itemset (LCM)

Frequent Pattern Mining

- Problem of enumerating all frequently appearing patterns in big data
  (pattern = itemsets, item sequence, short string, subgraphs,…)
- Nowadays, one of the fundamental problems in data mining
- Many applications, many algorithms, many researches

High Speed Algorithms are important



Applications of Pattern Mining

Market Data

- **Books** & **coffee** are frequently sold together
- **Male** coming at **Night** tends to purchase foods with bit higher prices…

•••

automatic classification

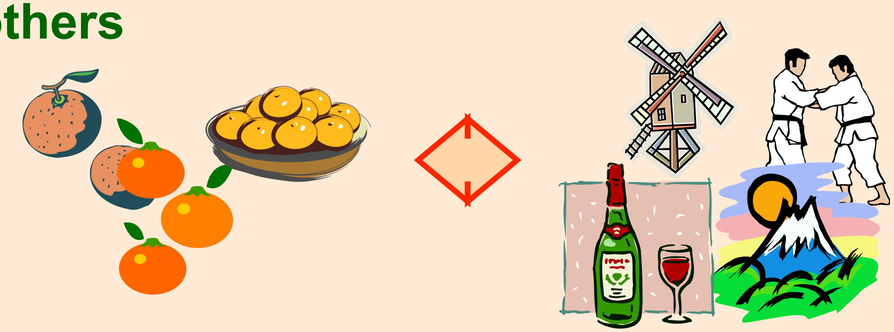
geneZ : ●○★
geneZ : ●○★
•

gene A : ●△▲
geneB : ●△▲
geneD : ●△▲
•••

geneF1 : ■□
geneF2 : ■□
•••

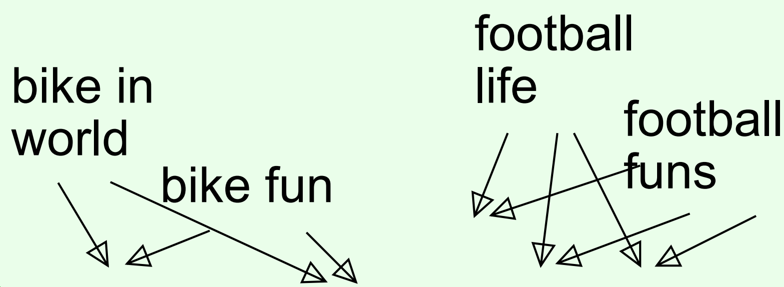
Image Recognition

- find features distinguishing **orange** vs **others**



clustering topics in Web pages

- by links, keywords, dates, and their combinations



Fundamental, thus applicable to vatious areas

Transaction Database

- A database ***D*** such that each transaction (record) **T** is a subset of itemset **E**, i.e., $\forall \mathbf{T} \in D, \mathbf{T} \subseteq \mathbf{E}$

For itemset **P**,

occurrence of P: a transaction of ***D*** including **P**

occurrence set of P (Occ(P)): set of occurrences of **P**

frequency of P (frq(P)): cardinality of **Occ(P)**

$$D = \begin{array}{l} \boxed{1,2,5,6,7,9} \\ 2,3,4,5 \\ \boxed{1,2,7,8,9} \\ 1,7,9 \\ 2,7,9 \\ 2 \end{array}$$

Occ({1,2})
= { {1,2,5,6,7,9},
{1,2,7,8,9} }

Occ({2,7,9})
= { {1,2,5,6,7,9},
{1,2,7,8,9},
{2,7,9} }

Frequent Itemset

frequent itemset: an itemset included in at least **σ** transactions of ***D***
(a set whose frequency is at least **σ**) (**σ** is given, and called **minimum support**)

Ex) itemsets included in at least 3 transactions in ***D***

D =
1,2,5,6,7,9
2,3,4,5
1,2,7,8,9
1,7,9
2,7,9
2

included in at least 3
{1} {2} {7} {9}
{1,7} {1,9}
{2,7} {2,9} {7,9}
{1,7,9} {2,7,9}

Frequent itemset mining is to enumerate all frequent itemsets of the given database and minimum support σ

Backtracking Algorithm

- Set of frequent itemsets is monotone

  (any subset of a frequent itemset is also frequent)

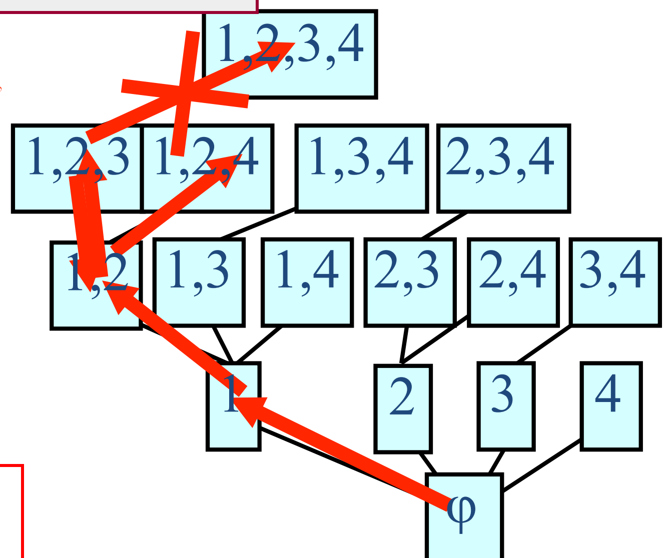
  □ backtrack algorithm is applicable

Backtrack (**P**)

1. **output P**
2. **for** each **e** > tail of **P** (maximum item in **P**)

   if **P∪e** is frequent **then call Backtrack** (**P∪e**)

O(|*D*|)



+ #recursive calls = #frequent itemsets

+ a recursive call (iteration) takes time

(**n-** (tail of **P**))×(time for frequency counting)

O(n|*D*|) per solution is too long

Shorten "Time for One Solution"

- Time per solution is polynomial, but too long

- Each **P∪e** needs to compute its frequency

  + Simply, check each transaction includes **P∪e** or not

    worst case: linear time in the database size

    average: **max{** #transactions, **frq(P)×|P| }**

  + Constructing efficient index, such as binary tree, is very difficult, for inclusion relation

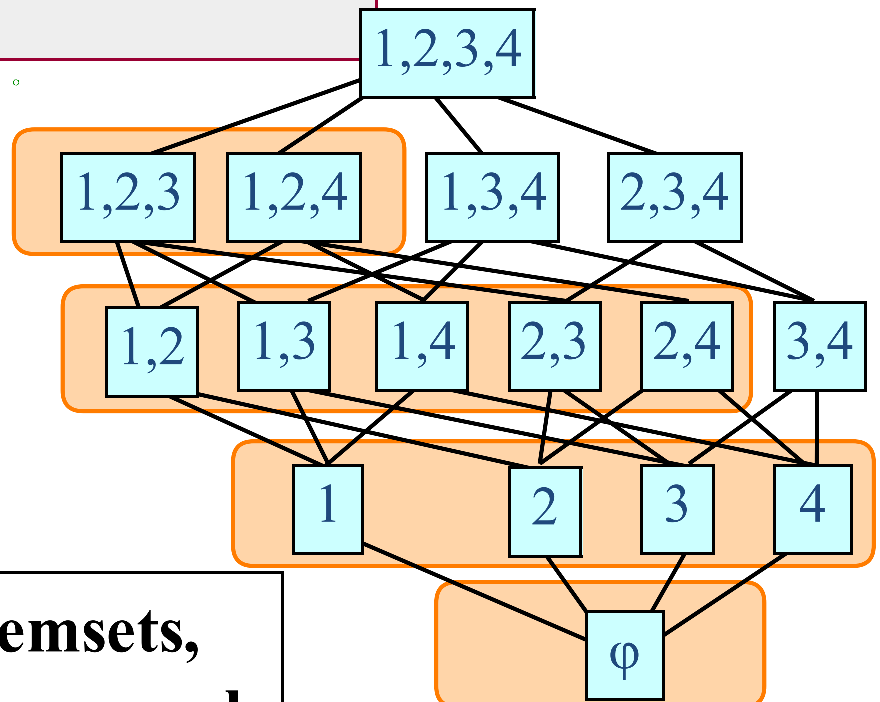
1,2,5,6,7,9
2,3,4,5
1,2,7,8,9
1,7,9
2,7,9
2

Algorithm for fast computation is needed

(a) Breadth-first Search

Apriori (P)

1. ***D*0={φ}, k := 1**
2. **while *D*k-1 ≠φ**
3. **for** each **P ∈ *D*k-1**
4. **for** each **e** not in **P**
5. **if** **P∪e** is frequent **then** insert **P∪e** to ***D*k**



- Before computing frequency of **P∪e**, check whether **P∪e -f** is in ***D*k** or not for all **f ∈P**
- If some are not, **P∪e** is not frequent

We can prune some infrequent itemsets, but takes much memory and time for search

(b) Using Bit Operations

- Represent each transaction/itemset by a bit sequence

{1,3,7,8,9} [101000111]

{1,2,4,7,9} [110100101]

[100000101]

- Intersection can be computed by AND operation

  (64 bits can be computed at once!)

  Also, memory efficient, if the database is dense

- On the other hand, very bad for sparse database

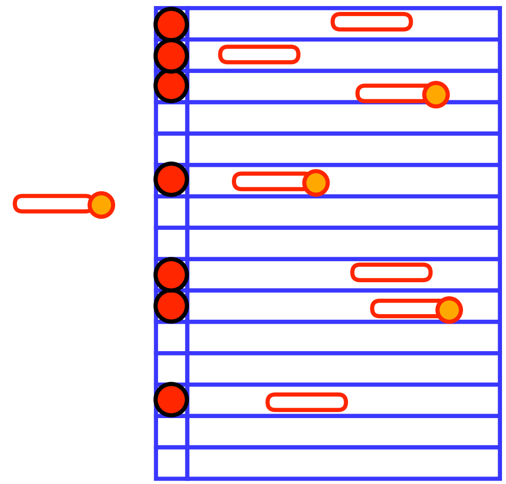
But, incompatible with the database reduction, explained later

(c) Down Project

- Any occurrence of **P∪e** includes **P** (□ included in **Occ(P)**)

  □ to find transactions including **P∪e** ,

  we have to see only transactions in **Occ(P)**



- **T∈ Occ(P)** is included in **Occ(P∪e)**

  if and only if **T** includes **e**

- By computing **Occ(P∪e)** from **Occ(P)**,

  we do not have to scan the whole database

□ Computation time is reduced much

Example: Down Project

- See the update of **Occ(P)**

+ **Occ(ϕ) = {A,B,C,D,E,F}**

Occ({2})

+ **Occ({2}) = {A,B,C,D,E,F} ∩ {A,B,C,E,F}**
  = {A,B,C,E,F}

Occ({7})

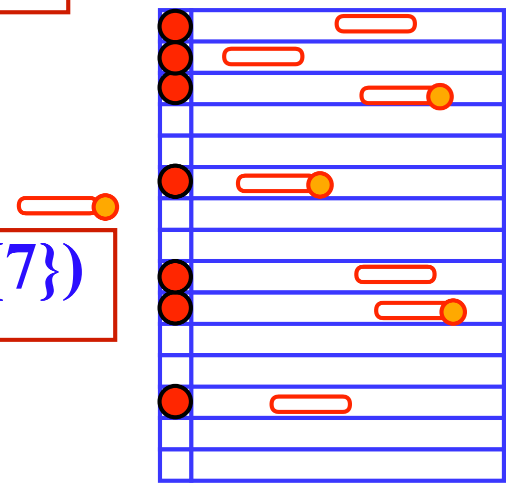
+ **Occ({2,7}) = {A,B,C,E,F} ∩ {A,C,D,E}**
  = {A,C,E}

Occ({9})

+ **Occ({2,7,9}) = {A,C,E} ∩ {A,C,D,E}**
  = {A,C,E}

Occ({4})

+ **Occ({2,7,9,4}) = {A,C,E} ∩ {B}**
  = ϕ



A: 1,2,5,6,7,9
B: 2,3,4,5
C: 1,2,7,8,9
D: 1,7,9
E: 2,7,9
F: 2

Intersection Efficiently

- **$T \in$ Occ(P)** is included in **Occ(P∪e)** if and only if **T** includes **e**

  □ **Occ(P∪e)** is the intersection of **Occ(P)** and **Occ({e})**

- Taking the intersection of two itemsets can be done by scanning the itemsets simultaneously in the increasing order of items (itemsets have to be sorted)

$$
\begin{array}{rllll}
 & \{1, & 3, & 7,8,9\} & \\
\cap & \{1,2, & 4,7, & 9\} & \\
= & \{1, & & 7, & 9\}
\end{array}
$$

Linear time in #scanned items □ sum of their sizes

Using Delivery

- Taking intersection for all **e** at once, fast computation is available

1. Set empty bucket for each item
2. For each transaction **T** in **Occ(P)**,
   + Insert **T** to the buckets of all item **e** included in **T**

□ After the execution, the bucket of **e** becomes **Occ(P∪e)**

Delivery (**P**)
1. **bucket[e] := ϕ** for all **e**
2. **for** each **T∈P**
3. **for** each **e∈T, e > tail(P)** insert **T** to **bucket[e]**

A: 1,2,5,6,7,9
B: 2,3,4,5
C: 1,2,7,8,9
D: 1,7,9
E: 2,7,9
F: 2

⇨

1: A,C,D
2: A,B,C,E,F
3: B
4: B
5: A,B
6: A
7: A,C,D,E
8: C
9: A,C,D,E

Time for Delivery

Delivery (**P**)
1. **jump** := ϕ, **bucket[e]** := ϕ for all **e**
2. **for** each **T∈P**
3. **for** each **e∈T, e > tail(P)**
4. **if bucket[e]** = ϕ **then** insert **e** to **jump**
5. insert **T** to **bucket[e]**
6. **end for**
7. **end for**

- Comp. time is **ΣT∈Occ(P) |{e | e∈T, e > tail(P)}|**

- Computation time is reduced by sorting the items in each transaction, in the initialization

A: 1,2,5,6,7,9
B: 2,3,4,5
C: 1,2,7,8,9
D: 1,7,9
E: 2,7,9
F: 2

Delivery

- Compute the denotations of $P \cup \{i\}$ for **all** i's at once,

D =
1,2,5,6,7,9
2,3,4,5
1,2,7,8,9
1,7,9
2,7,9
2

A	**1**	A **2**		A	A **5**	**6**	**7**	A	**9**
B		2	3	4	5				
C	**1**	C **2**					C **7**	C **8**	**9**
D	**1**						**7**	D	**9**
E		2					7		9
F		2							

P = {1,7}

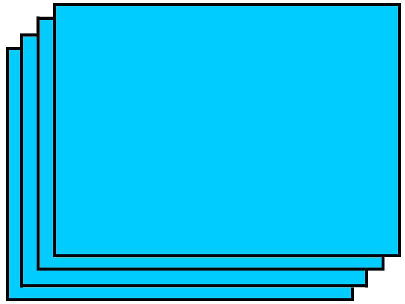
Check the frequency for all
items to be added in linear
time of the database size

Generating the recursive calls in reverse
direction, we can re-use the memory

Intuitive Image of Iteration Cost

- Simple frequency computation scan

The whole data, for each **PUe**



(n-t)

- Set inte

and **Oc**

□ **n-t**

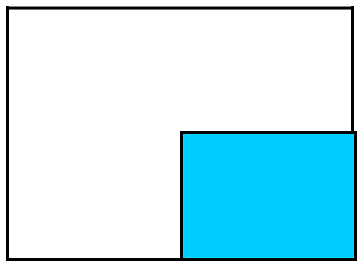
items lar

Advantage is more

(n-t)

- Delivery scans items larger than **t**

of transactions included in **Occ(P)**



t

Bottom-wideness

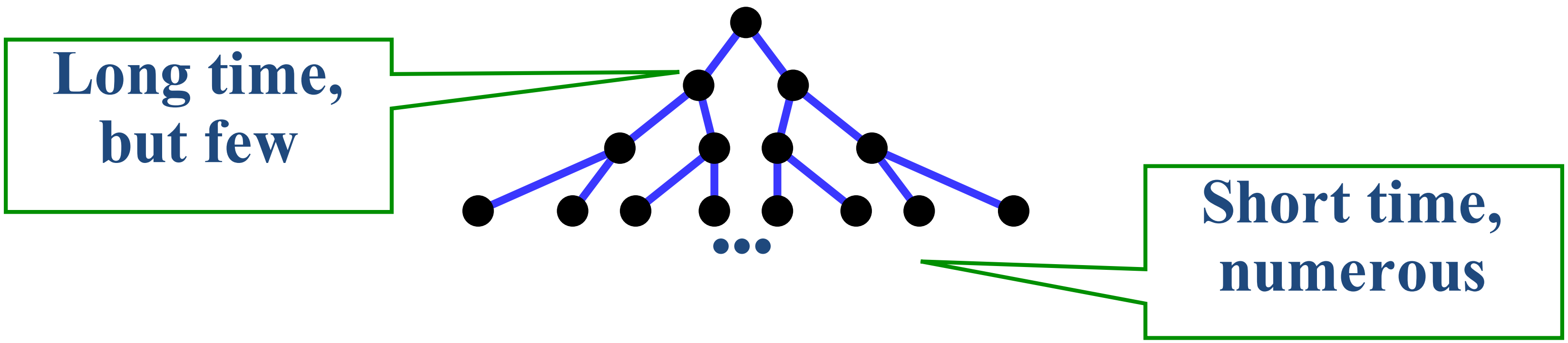
- In the deep levels of the recursion, frequency of **P** is small

□ Time for delivery is also short

- Backtrack generates several recursive calls in each iteration

  □ Recursion tree spreads exponentially, as going down

  □ Computation time is dominated by the bottom-level exponentially many iterations



Almost all iterations takes short time

□ In total, average time per iteration is also short

Even for Large Support

- When **σ** is large, **|Occ(P)|** is large in bottom levels
  - Bottom-wideness doesn't work

- Speed up bottom levels by **database reduction**

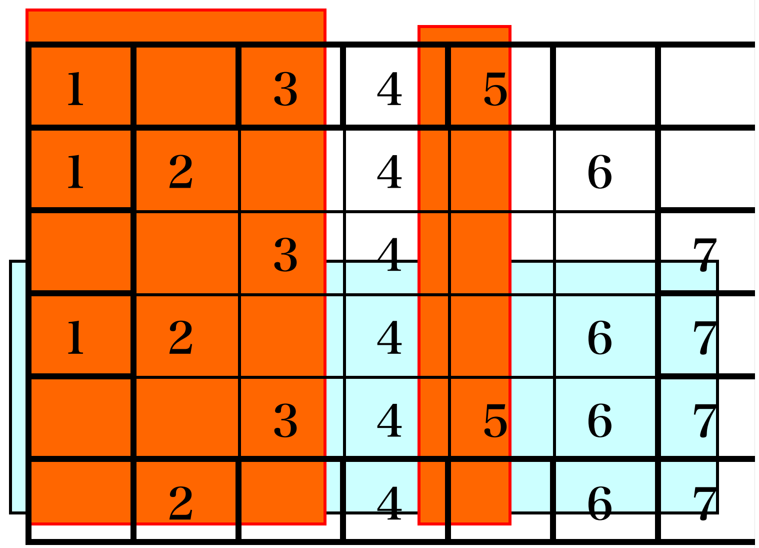
(1) delete items smaller than added item most recently

(2) delete items infrequent in the database induced by **Occ(P)**

(they never be added to the solution, in the recursive calls)

(3) unify the identical transactions

- In real data, usually the size of reduced database is constant, in bottom levels



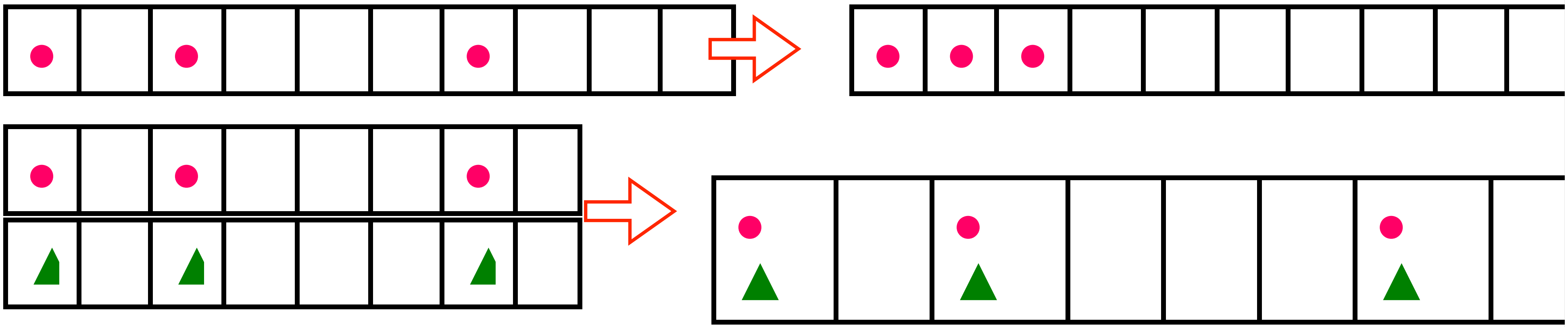
fast as much as small σ

Synergy with Cache

- Efficient implementation needs “hit/miss ratio” of cache
  - open the loops
  - change memory allocation

for i=1 to n { x[i]=0; }

□ for i=1 to n step 3 { x[i]=0; x[i+1]=0; x[i+2]=0; }



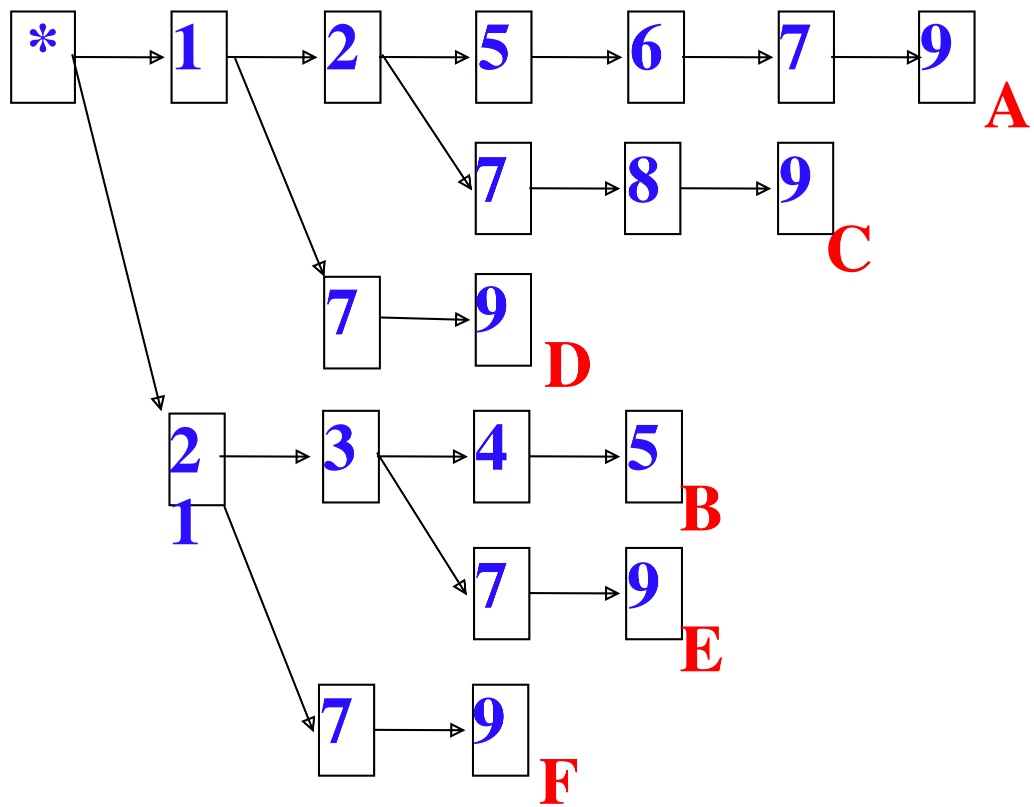
By database reduction, memory for deeper levels fits cache
□ Bottom-wideness implies “cache hits almost all accesses”

Compression by Trie/Prefix Tree

- Regarding each transaction as a string, we can use trie / prefix tree to store the transactions, to save memory usage
  - Orthogonal to delivery, shorten the time to scan

(disadvantage is overhead, for its representations)

A: 1,2,5,6,7,9
B: 2,3,4,5
C: 1,2,7,8,9
D: 1,7,9
E: 2,3,7,9
F: 2,7,9



3-2 Result of Competition

Competition: **FIMI04**

- **FIMI:** Frequent Itemset Mining Implementations

  + A satellite workshop of **ICDM** (International Conference on Data Mining). Competition on implementations for frequnet/closed/maximal frequent itemsets enumeration

  FIMI 04 is the second, and the last

- The first has 15, the second has 8 submissions

Rule and Regulation:

+ **Read data file, and output all solutions to a file**
+ Time/memory are evaluated by time/memuse command
+ direct CPU operations (such as pipeline control) are forbidden

Environment： **FIMI04**

- **CPU, memory:** Pentium4 3.2GHz、1GB RAM
  OS, Language, compiler: Linux, C, gcc

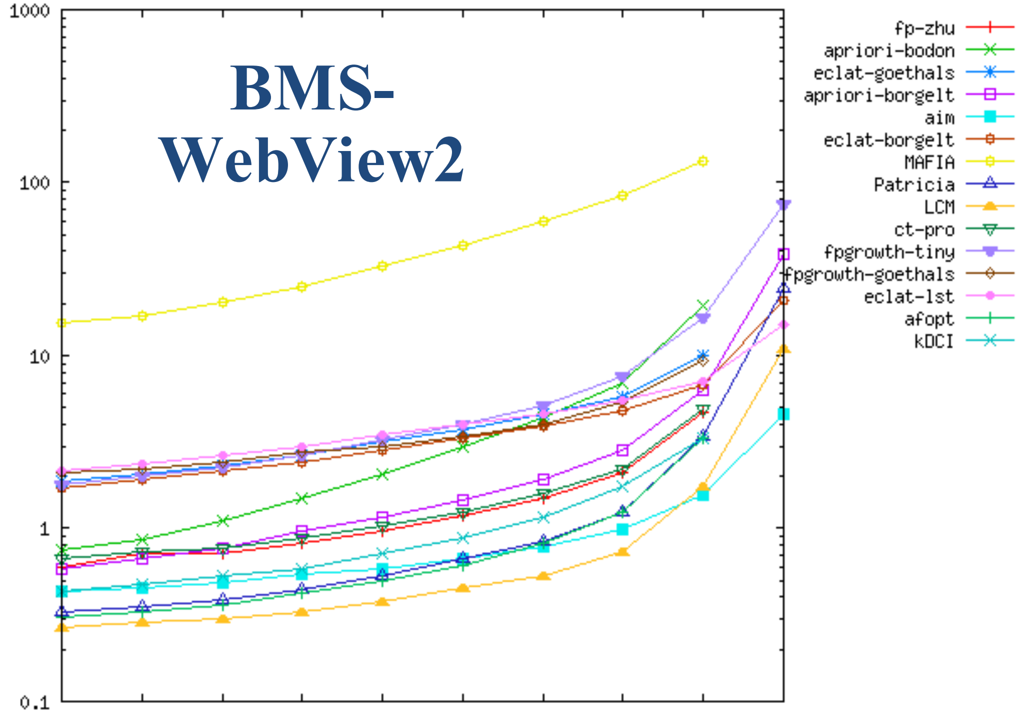
- **dataset:**
  + **real-world data:** sparse, many items
  + **machine learning repository:** dense, few items, structured
  + **synthetic data:** sparse, many items, random
  + **dense real-world data:** very dense, few items

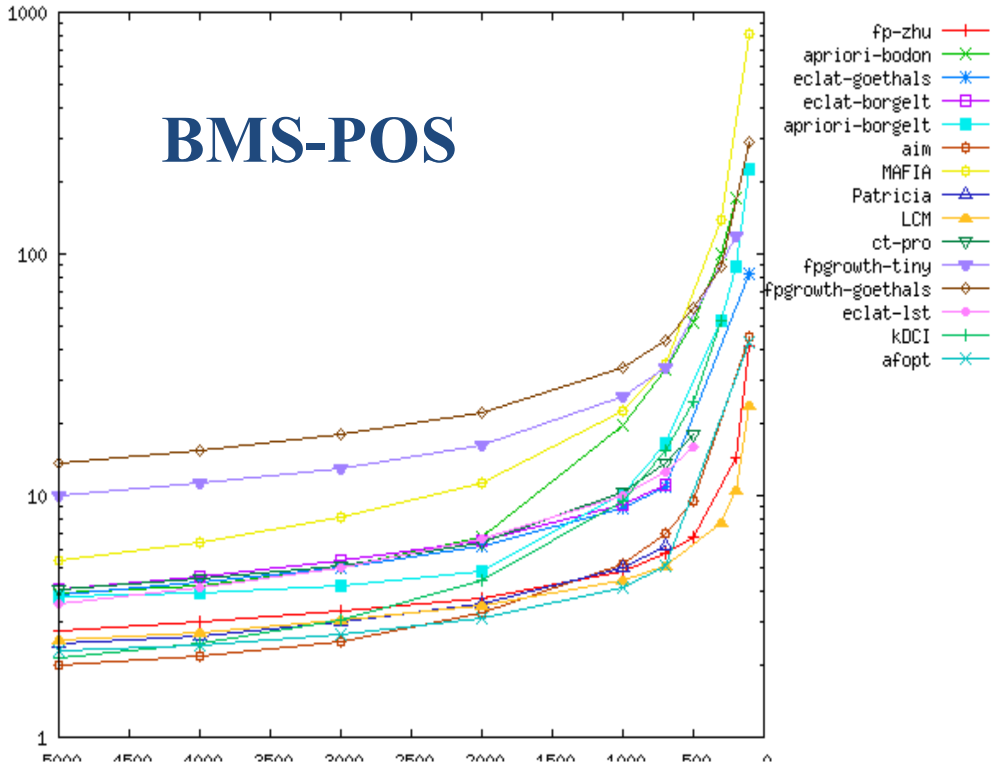
LCM ver.2 (Uno, Arimura, Kiyomi) won the Award

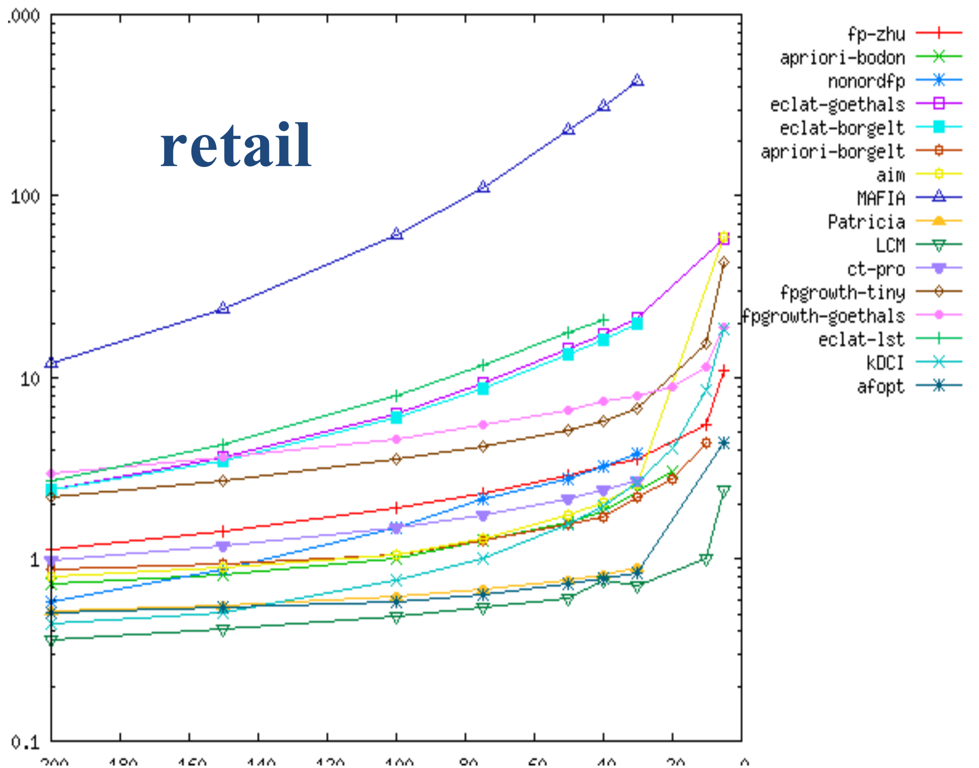
Award and Prize

Prize is {beer, nappy}
the "Most Frequent Itemset"

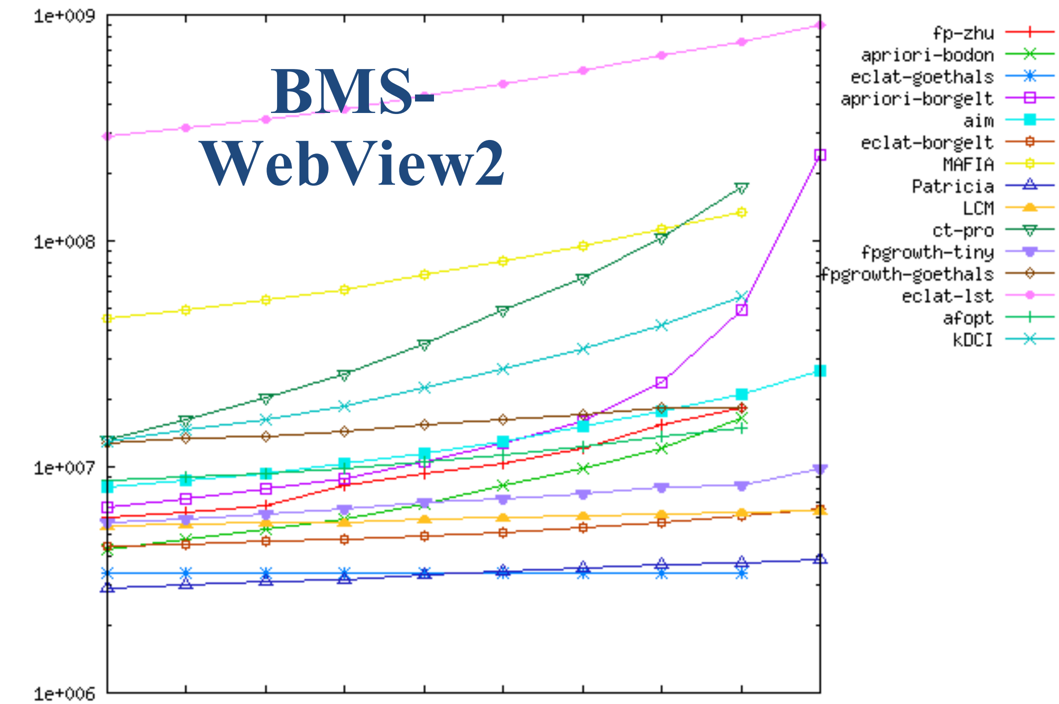
Real-world data (sparse) average size 5-10

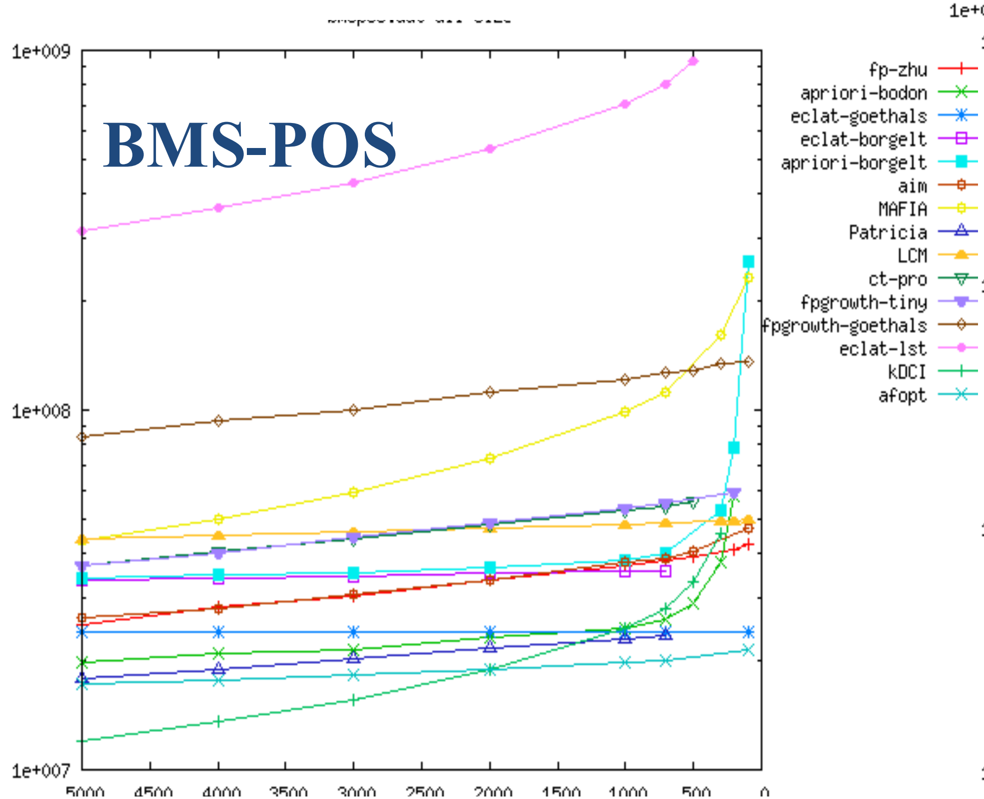


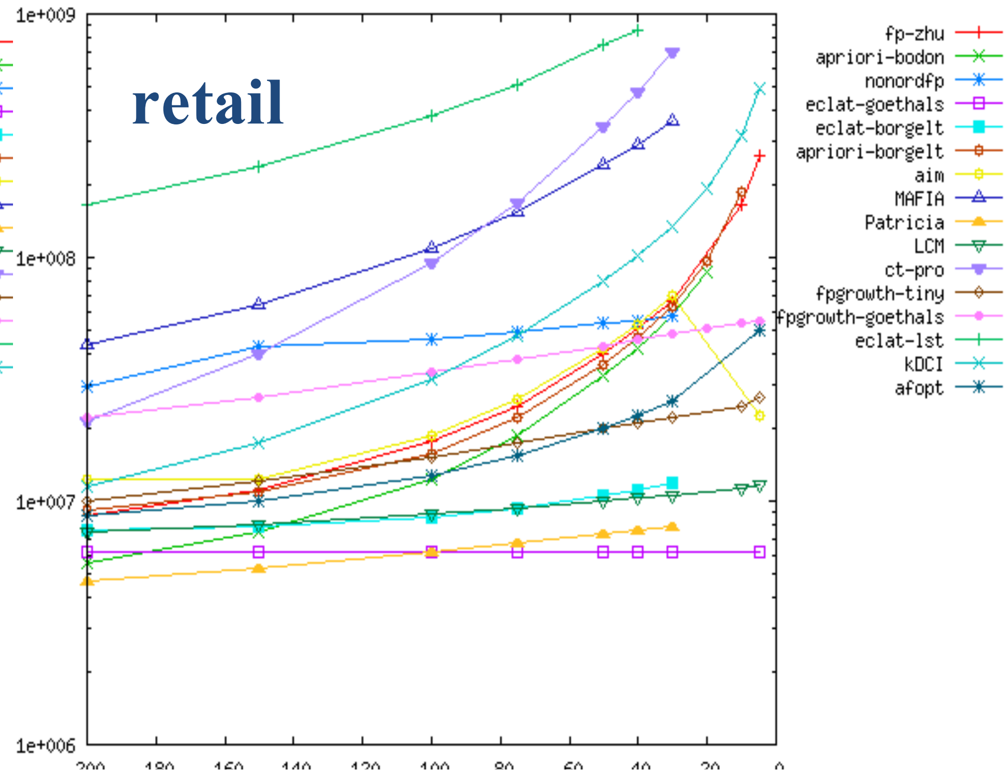




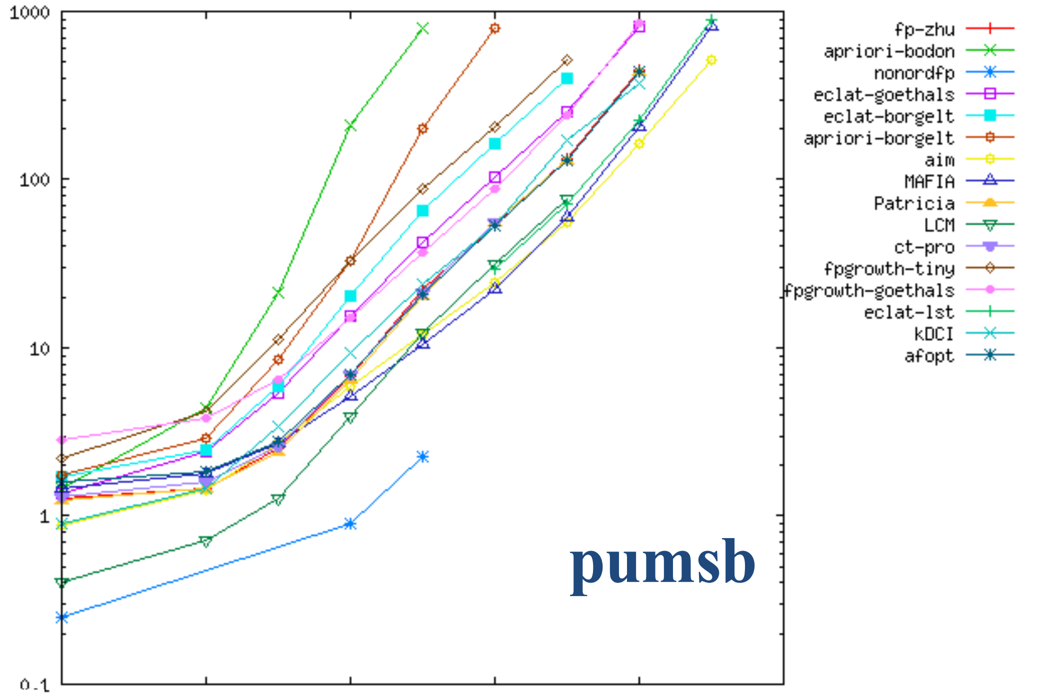
Real-world data(sparse) memory usage

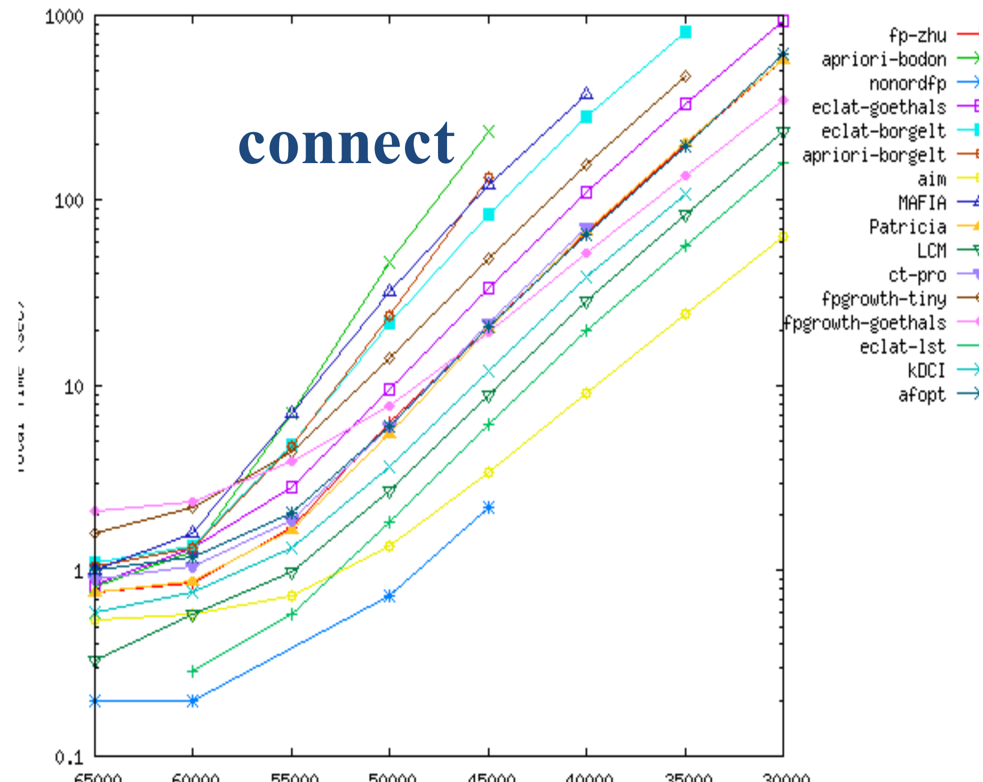


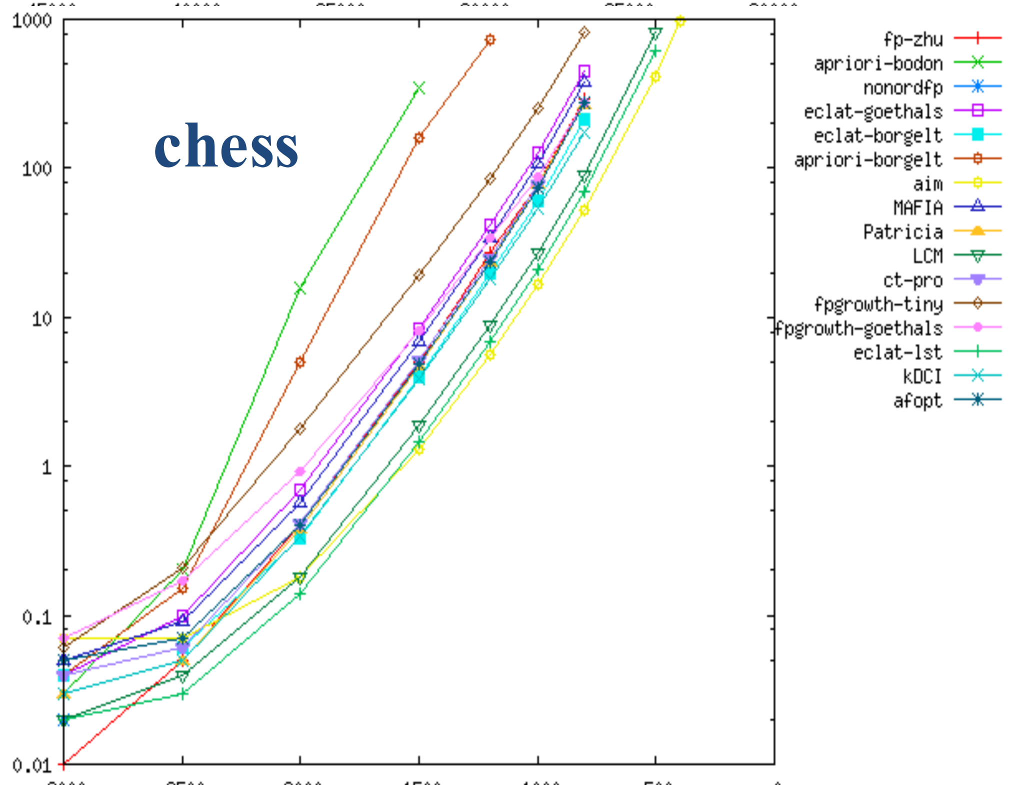




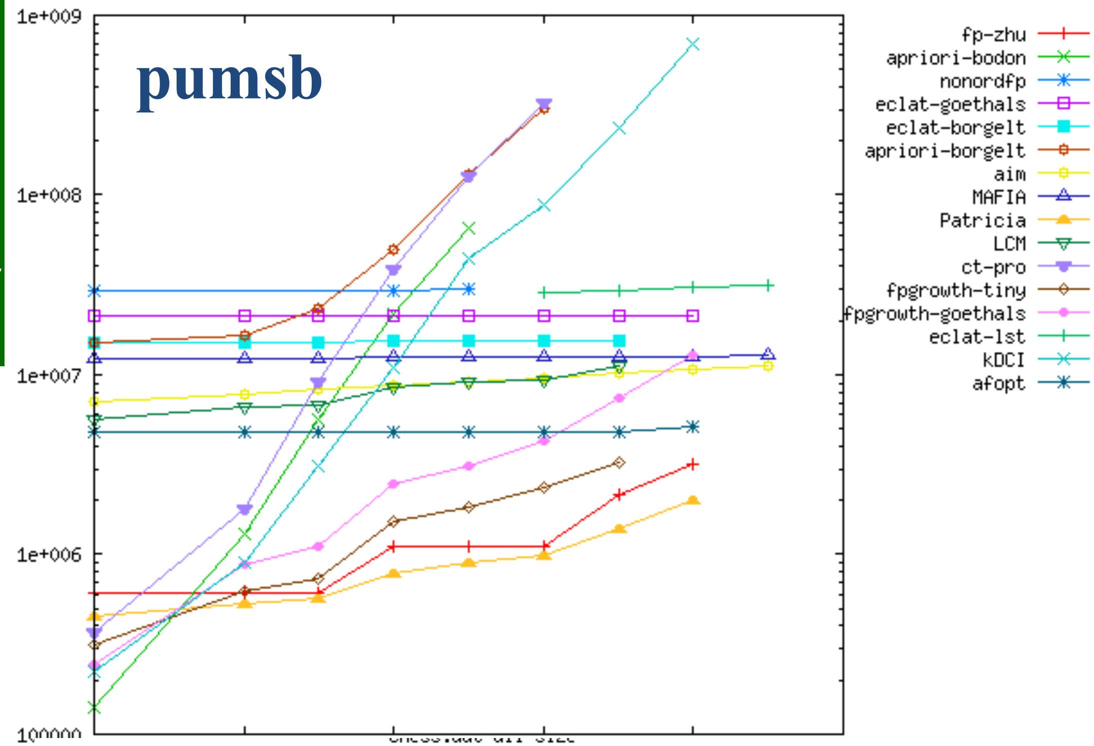
Dense (50%) structured data

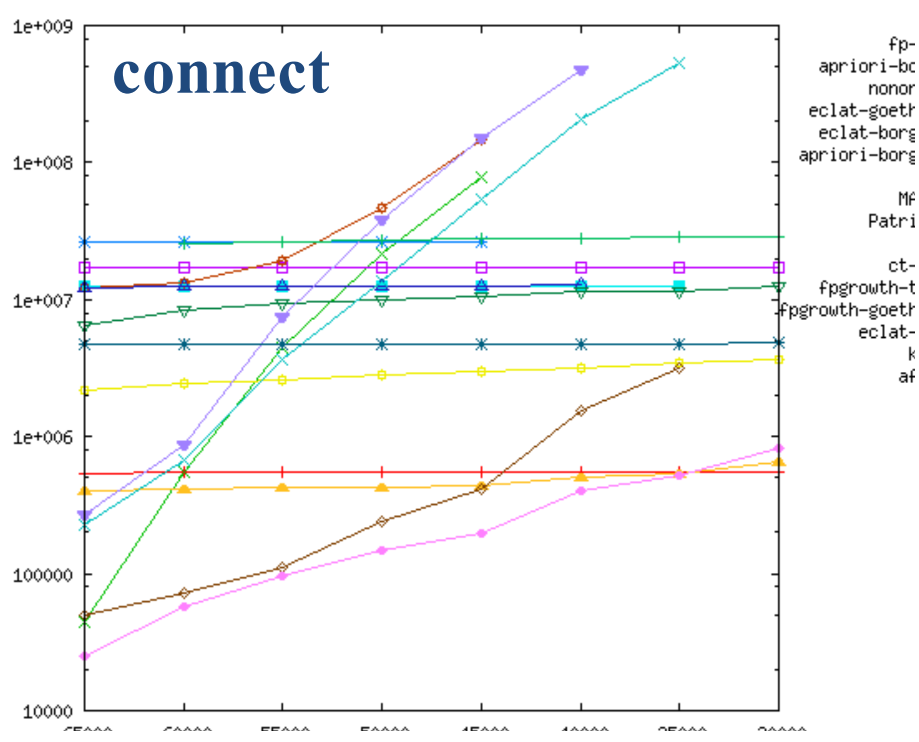


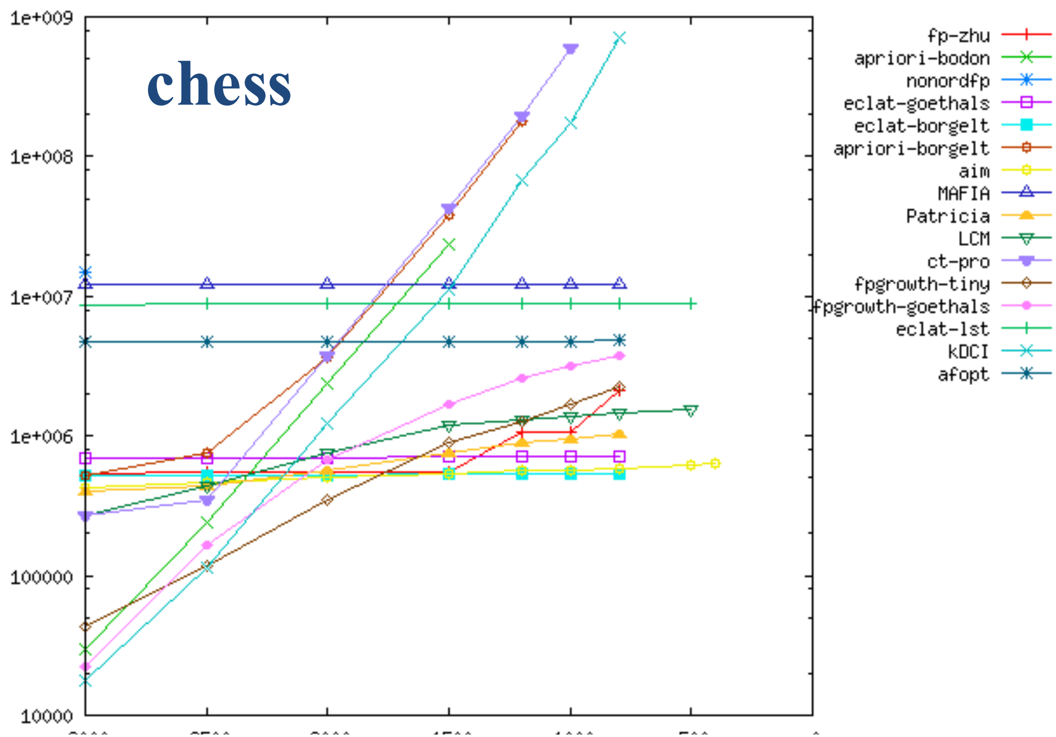




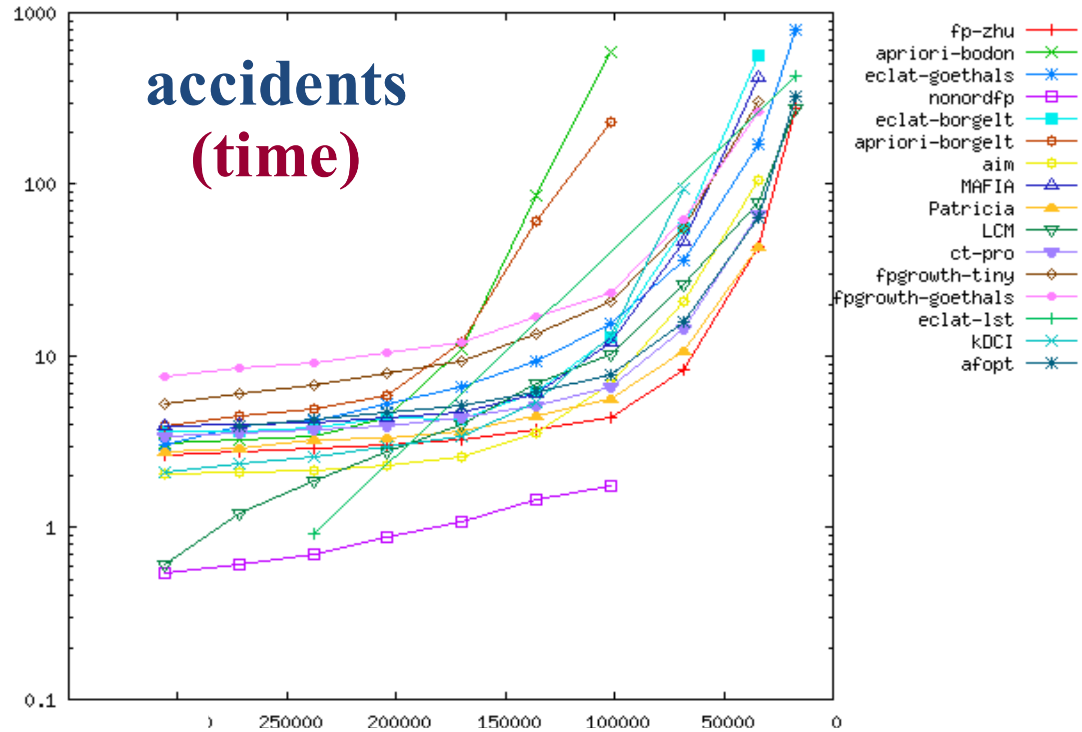
Memory usage: dense structured data



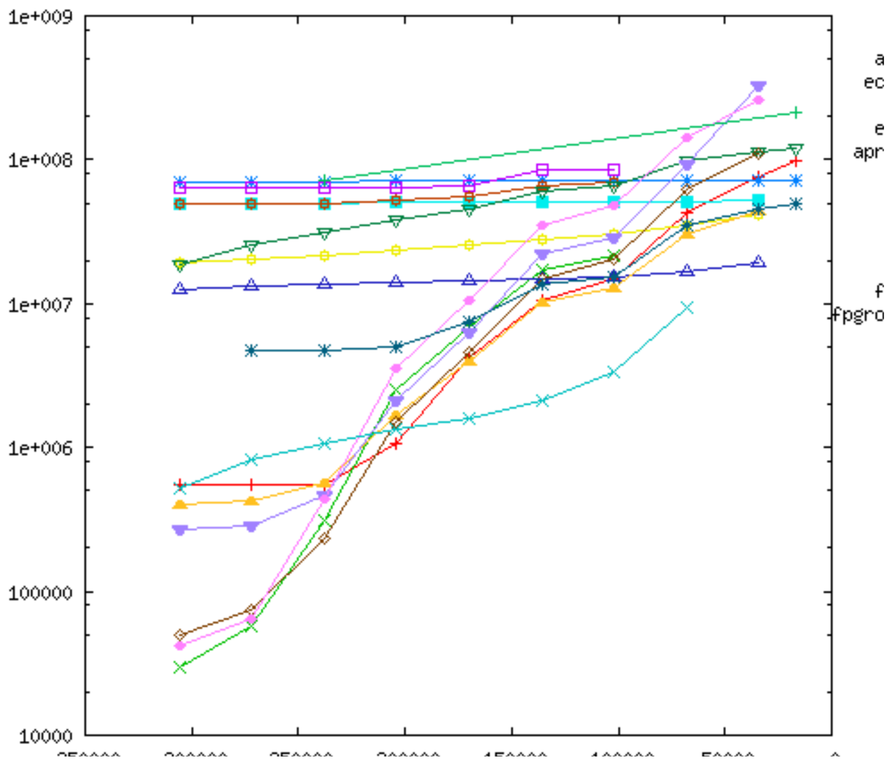


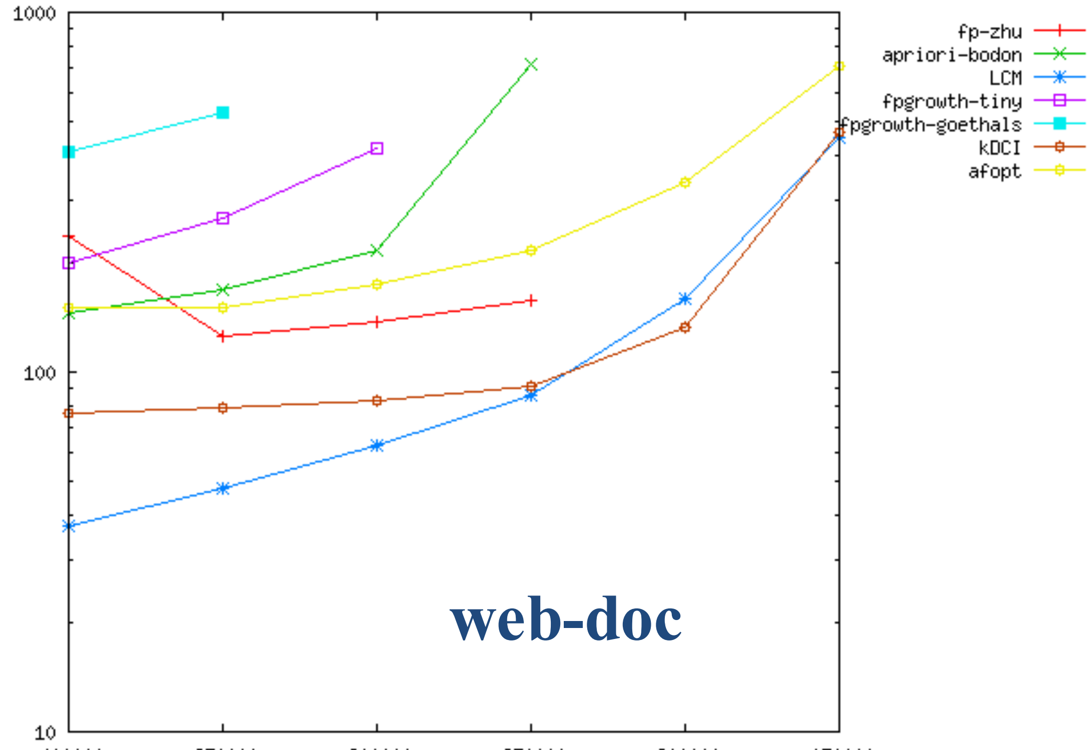


Dense real-world/ Large scale/ data



accidents

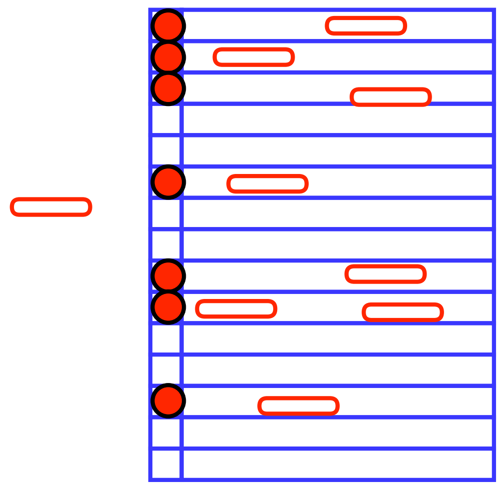




Other Frequent Patterns

- Bottom-wideness, delivery and database reduction are available for many kinds of other frequent pattern mining

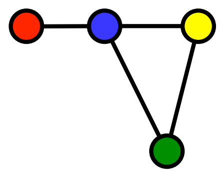
  + string, sequence, time series data
  + matrix
  + geometric data, figure, vector
  + graph, tree, path, cycles…



pattern

{A,C,D}

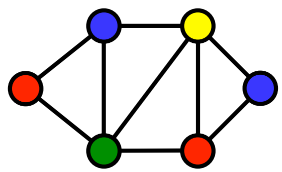
XYZ



record

{A,B,C,D,E}

AXccYddZf



3-3 Closed Itemset Enumeration

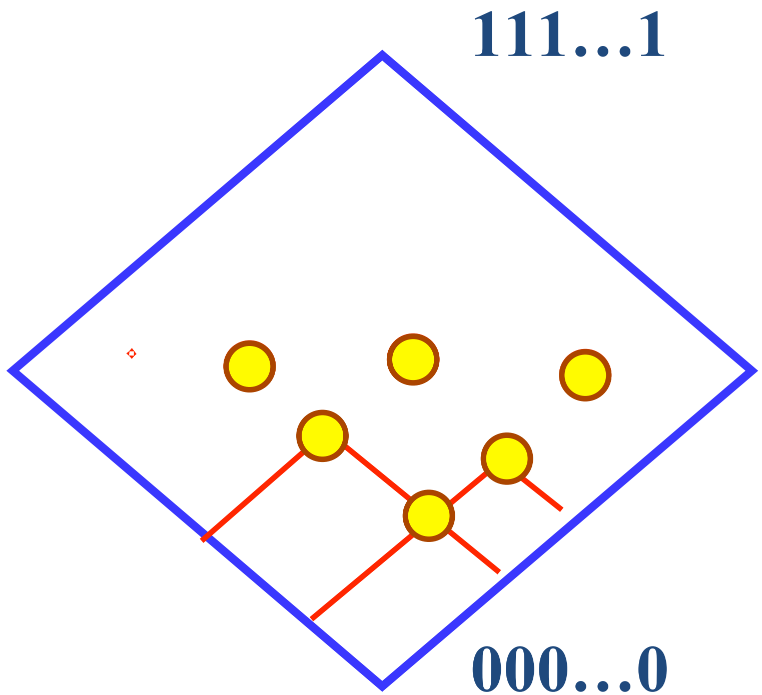
Disadvantage of Frequent Itemset

- To find interesting(deep) frequent itemsets, we need to set **σ** small
   numerous solutions will appear

- Without loss of information, we want to shift the problem (model)

1. maximal frequent itemsets
included in no other frequent itemsets

2. closed itemsets
maximal among those
having the same occurrence set



Ex) Maximal Frequent / Closed Itemsets

- Classify frequent itemsets by their occurrence sets

D =
1,2,5,6,7,9
2,3,4,5
1,2,7,8,9
1,7,9
2,7,9
2

included in at least 3

{1} **{2}** **{7}** **{9}**
{1,7} **{1,9}**
{2,7} **{2,9}** **{7,9}**
{1,7,9} **{2,7,9}**

frequent closed

maximal frequent

A closed itemset is the intersection of its occurrences

Advantage and Disadvantage

maximal

- existence of output polynomial time algorithm is open
- fast computation is available by pruning like maximal cliques
- few solutions but sensitive against the change of **σ**

closed

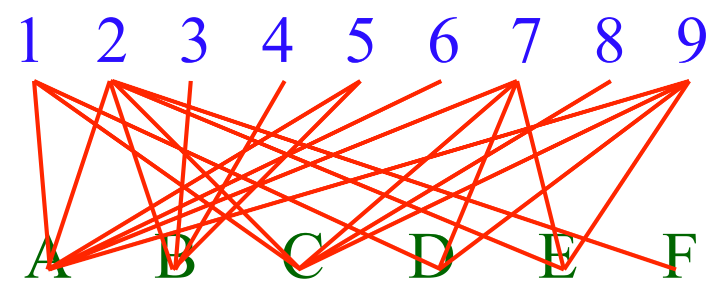
- polynomial time enumeratable by reverse search
- discrete algorithms and bottom-wideness fasten computation
- no loss w.r.t occurrence sets
- no advantage for noisy data (no decrease of solution)

Both can be enumerated O(1) time on average, 10k-100k / sec.

Bipartite Graph Representation

- Items and transactions are vertices, and the inclusion relations are the edges

D =
A: 1,2,5,6,7,9
B: 2,3,4,5
C: 1,2,7,8,9
D: 1,7,9
E: 2,7,9
F: 2



- itemset and transactions including it
  □ bipartite clique of the graph
- itemset and its occurrence set
  □ bipartite clique maximal on the transaction side
- closed itemset and its occurrence set □ maximal bipartite clique

From Adjacency Matrix

- See the adjacency matrix of the bipartite graph

D=
A: 1,2,5,6,7,9
B: 2,3,4,5
C: 1,2,7,8,9
D: 1,7,9
E: 2,7,9
F: 2

	1	2	3	4	5	6	7	8	9
A	1	1			1	1	1		1
B		1	1	1	1				
C	1	1					1	1	1
D	1						1		1
E		1					1		1
F		1							

- itemset and transactions including it
  - a submatrix all whose cells are 1

Methods for Closed Itemset Enumeration

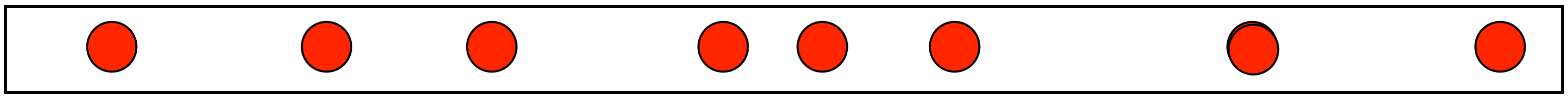
- **Based on frequent itemset enumeration**
  - enumerate frequent itemsets, and output only closed ones
  - can not get advantage of fewness of closed itemsets

- **Storage ＋ pruning**
  - store all solutions found, and use them for pruning
  - pretty fast
  - memory usage is a bottle neck

- **Reverse search ＋ database reduction （LCM）**
  - computation is efficient
  - no memory for previously found solutions

Neighbor Relation of Closed Itemsets

- Remove items from a closed itemset, in decreasing ordering



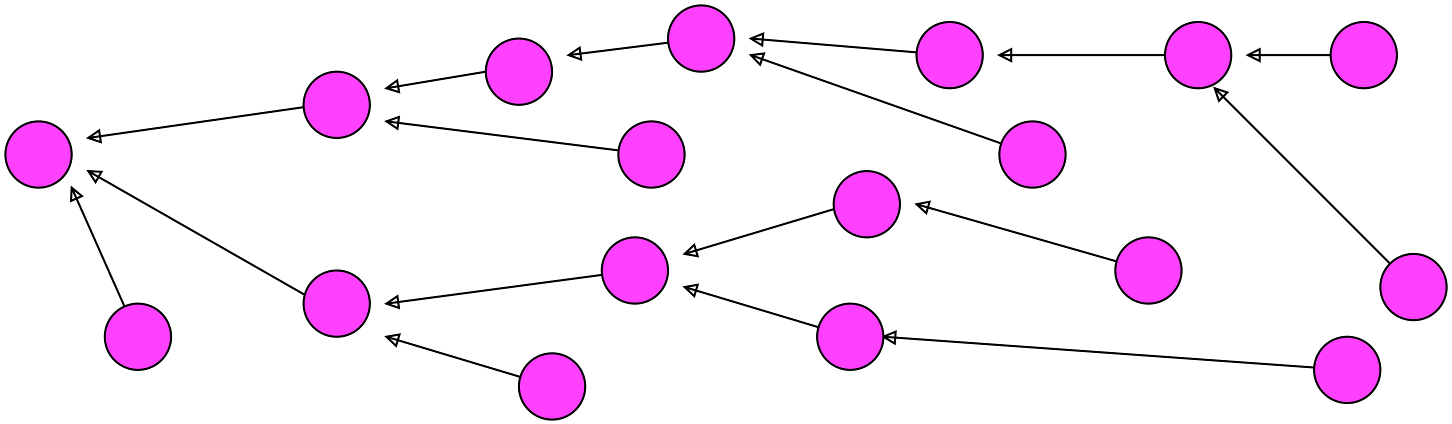
- At some point, occurrence set expands
- Compute the closed itemset of the expanded occurrence set
- The obtained closed itemset is the **parent** (uniquely defined)

- Frequency of the parent is always larger than the child, thus the parent-child relation is surely acyclic

  - The parent-child relation induces a directed spanning tree

Reverse Search

Parent-child relation induces a directed spanning tree



DFS search on the tree can find all solutions

- Enumeration method for all children of a parent is enough to search
- If children are found polynomial time on average, output polynomial

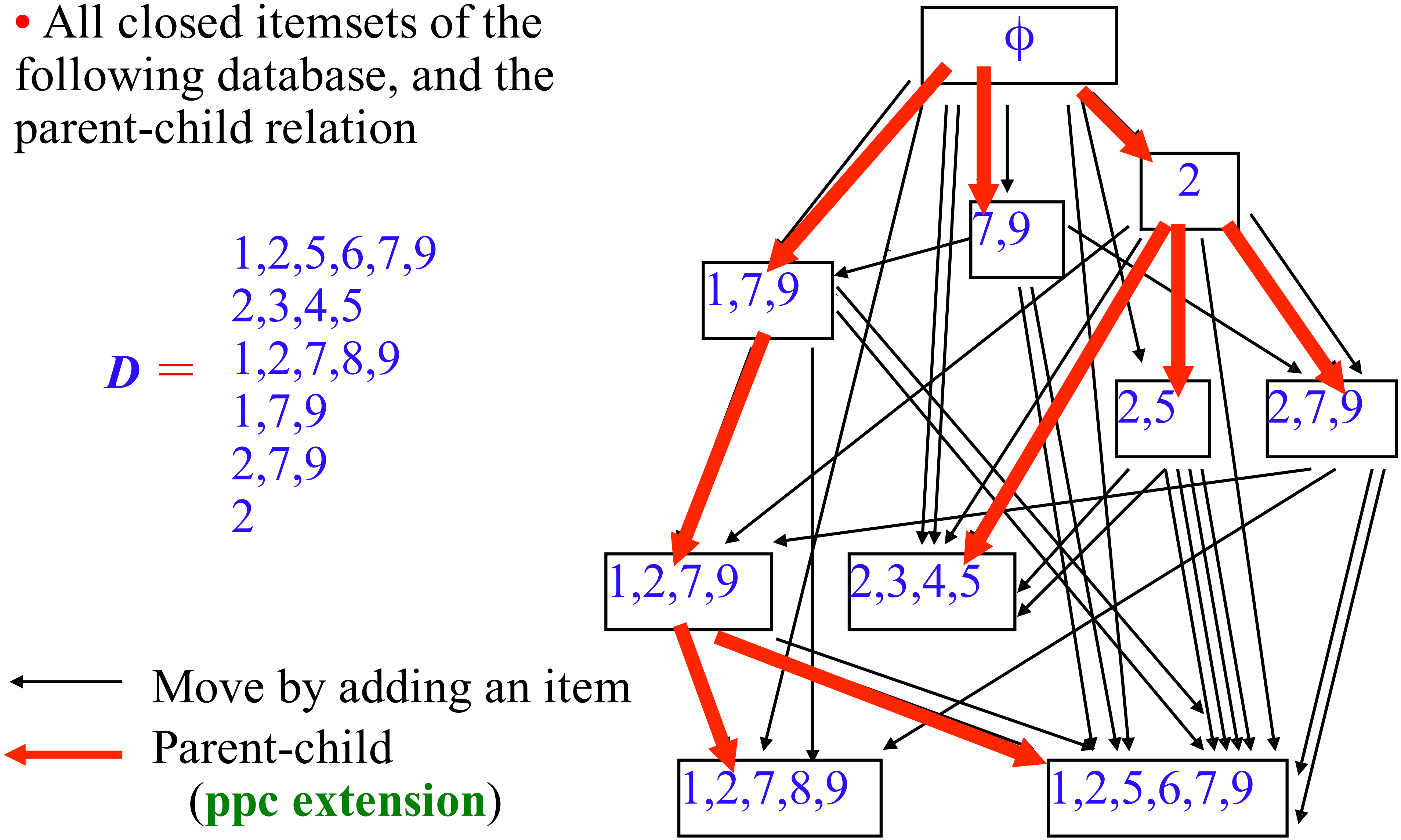
(child is obtained by adding an item to the parent)

Acyclic relation and polytime children enumeration are sufficient to polytime enumeration of any kind of objects

Ex) Parent-child Relation

- All closed itemsets of the following database, and the parent-child relation

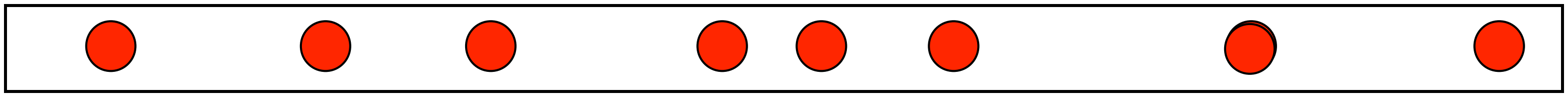
$D =$ 1,2,5,6,7,9
2,3,4,5
1,2,7,8,9
1,7,9
2,7,9
2



Computing the Children

- Let **e** be the item removed most recently to obtain the parent

By adding **e** to the parent, its occurrence set will be the occurrence set of the child



- A child is obtained by adding an item and computing the closed itemset

- However, itemsets obtained in this way are not always children

- Necessary and sufficient condition to be a child is "no item appears preceding to **e**" by closure operation (**prefix preserving closure extension (ppc-extension)**)

Database Reduction

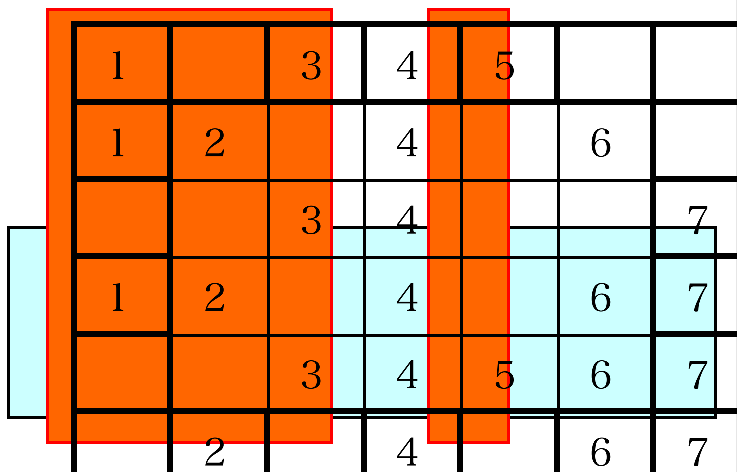
- We want to reduce the database as frequent itemset enumeration
- However, can not remove smaller items (than last added item **e**)
  - □ computation of ppc extension needs them

- However,
  - + if larger items are identical, included in **Occ** at the same time
  - + so, only the intersection of the smaller parts is needed

store the intersection of transactions having the same large items

- The intersection of many transactions
  has a small size

no much loss compared with large σ



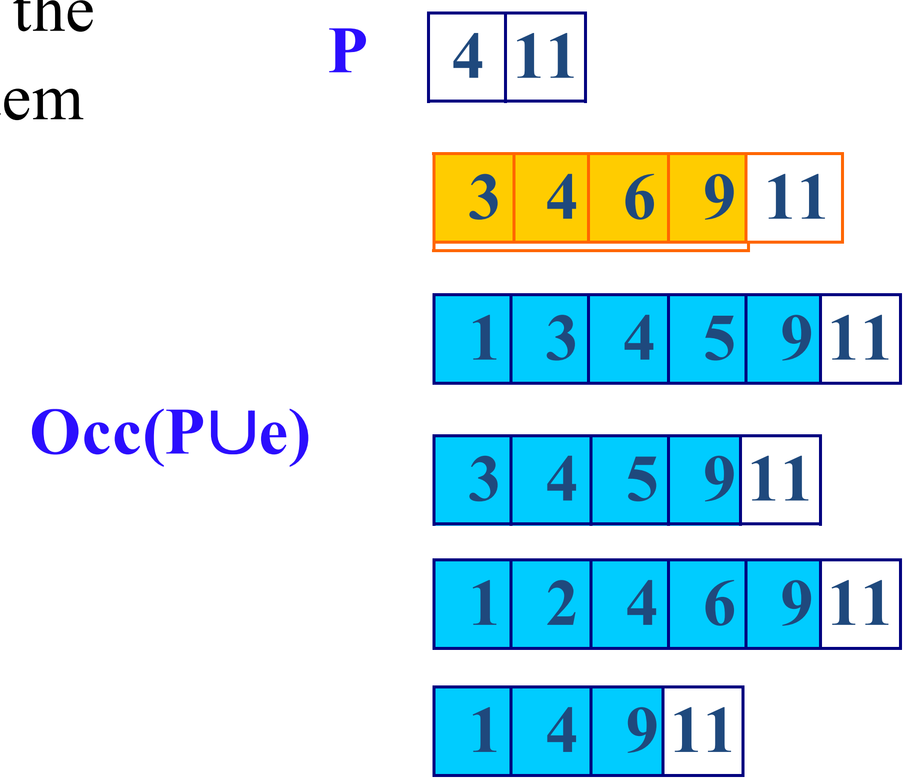
Cost for Comparison

- We can simply compute the intersection of **Occ(P∪e)**, but would be redundant. We do just "checking the equality of the intersection **Occ(P∪e)** and **P**", thus no need to scan all
  - We can stop when we confirmed that they are different

- Trace each occurrence of **P∪e** in the increasing order, and check each item appears all occurrences or not

- If an item appears in all, check whether it is included in **P** or not

- Proceed the operations from the last operated item



Using Bit Matrix

- Sweep pointer is a technique for sparse style data.
  We can do better if we have adjacency matrix

- But, adjacency matrix is so huge (even for construction)

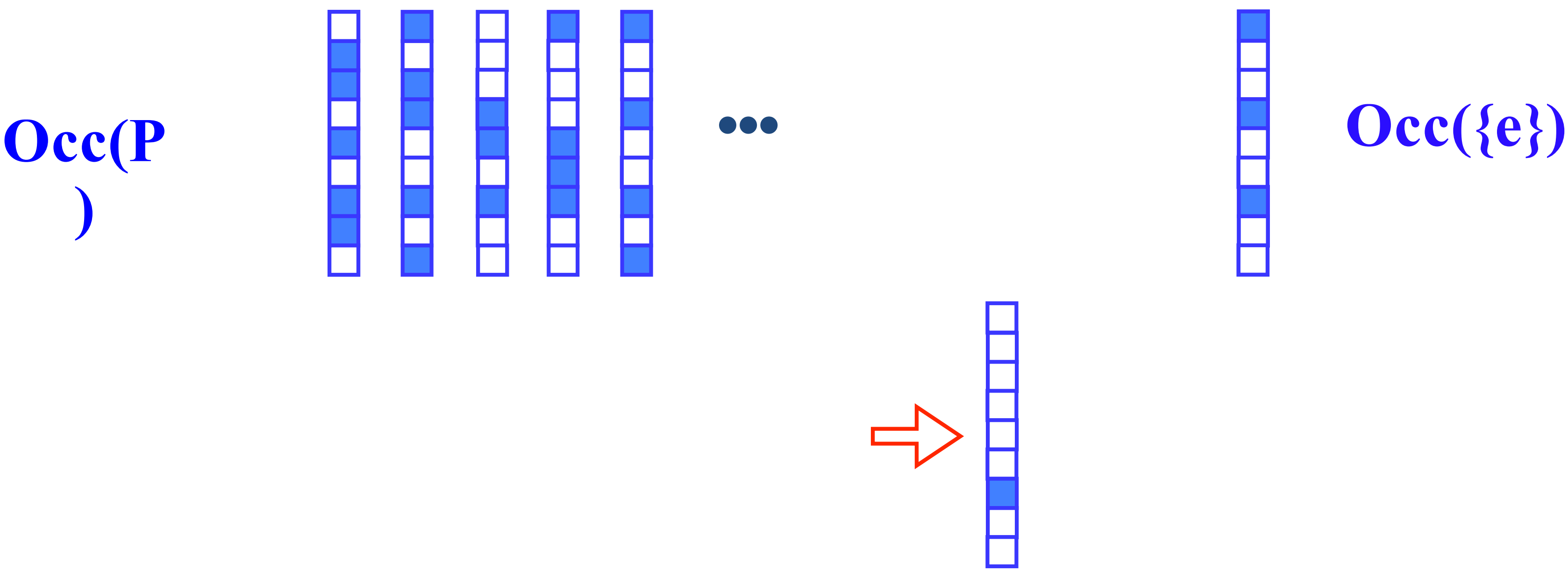
- Use adjacency matrix when the occurrence set becomes sufficiently small

□ By representing the matrix by bitmap, each column (corresponding to an item) fits one variable!

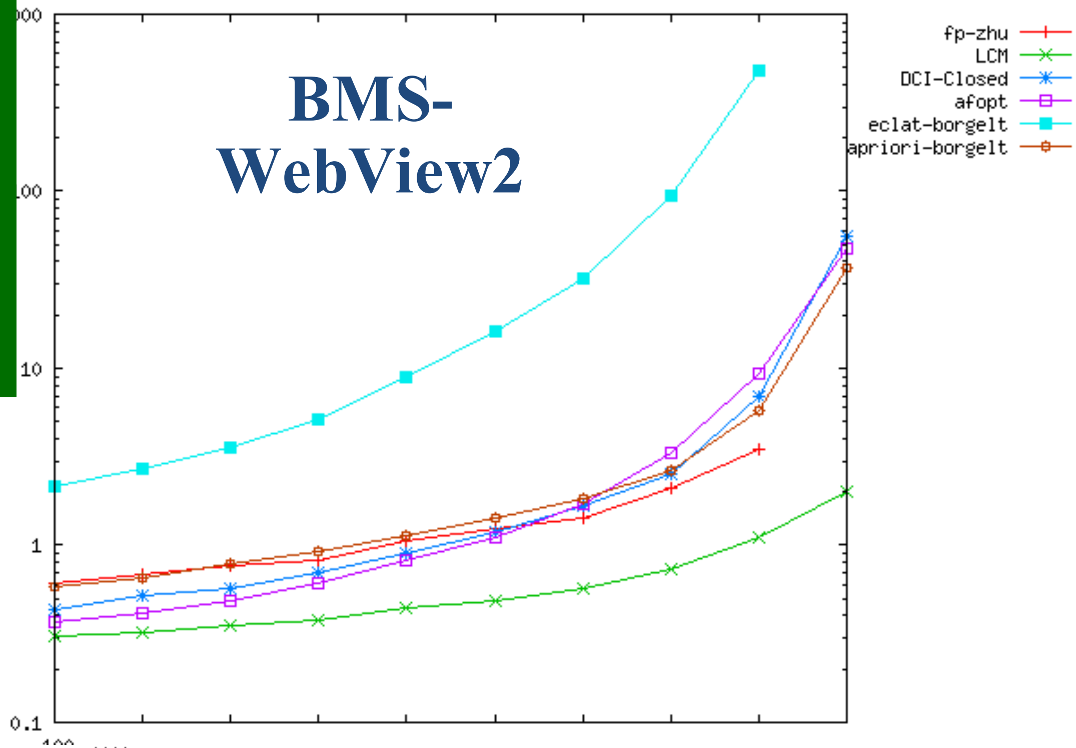
O(1) Time Computation of Bit Matrix

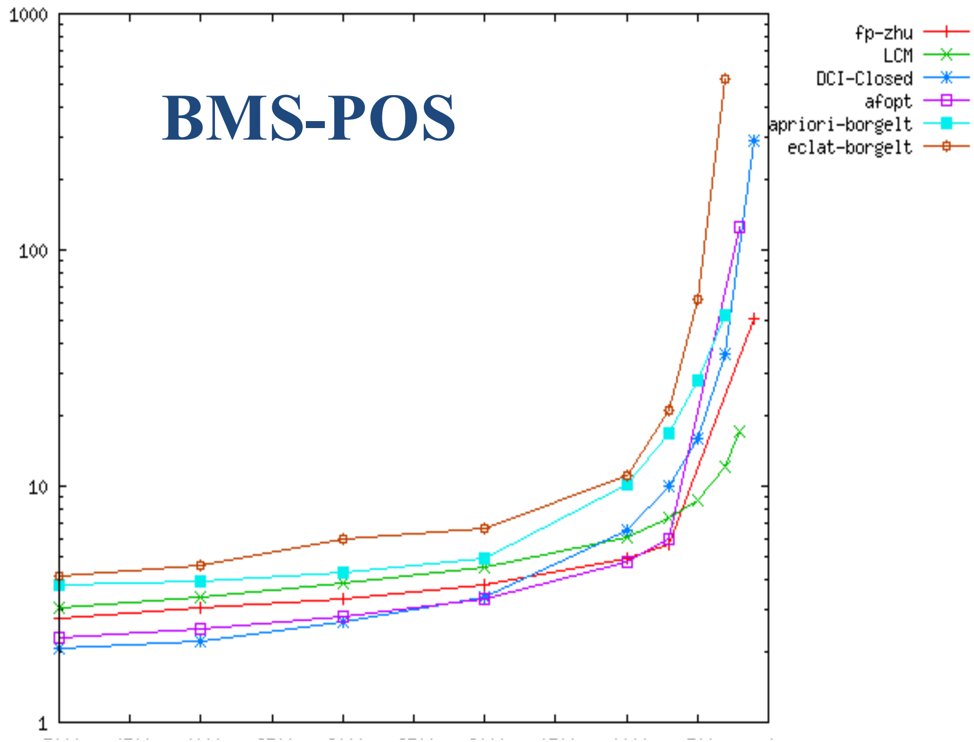
- By storing the occurrences including each item by a variable, we can check whether all occurrence of **$P \cup e$** includes an item or not in **O(1)** time

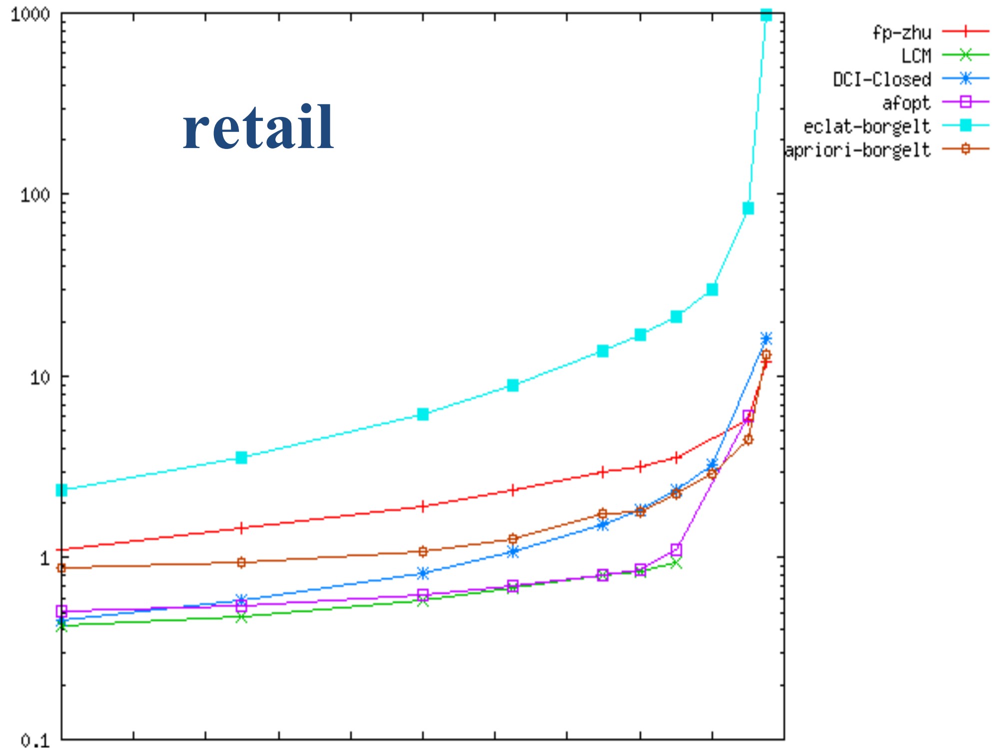
- Take the intersection of bit patterns of **Occ({i})** and **Occ($P \cup e$)**
- If **i** is included in all occurrences of **$P \cup e$**, their intersection is equal to **Occ({i})**



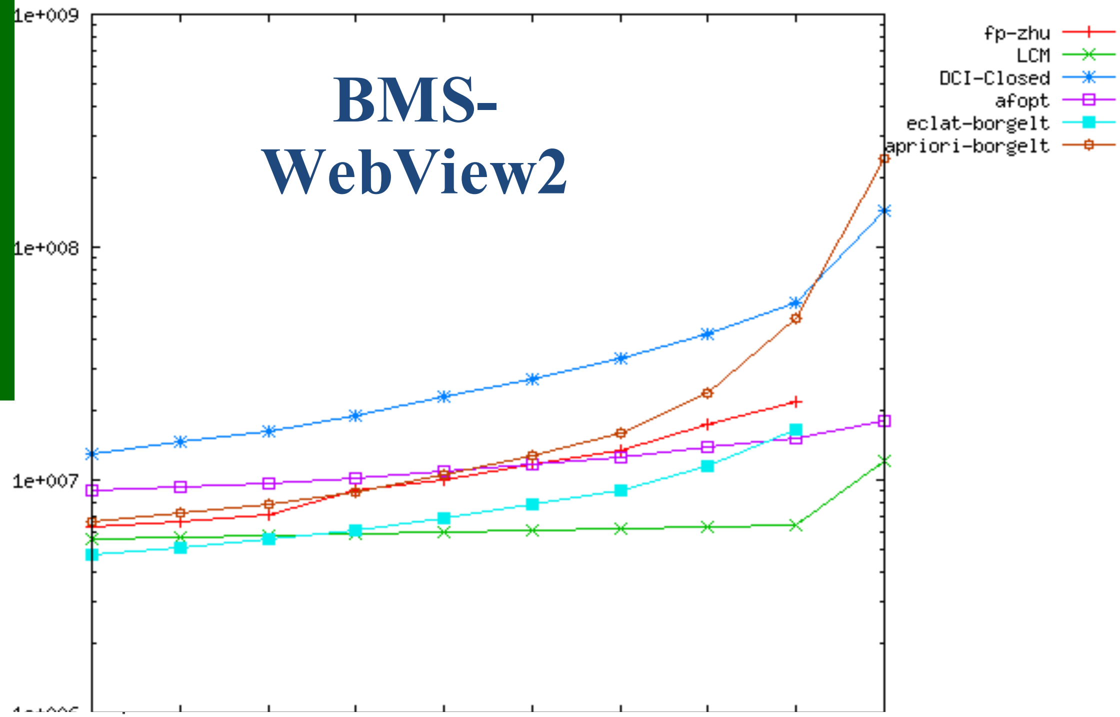
real-world data (sparse) average size 5-10

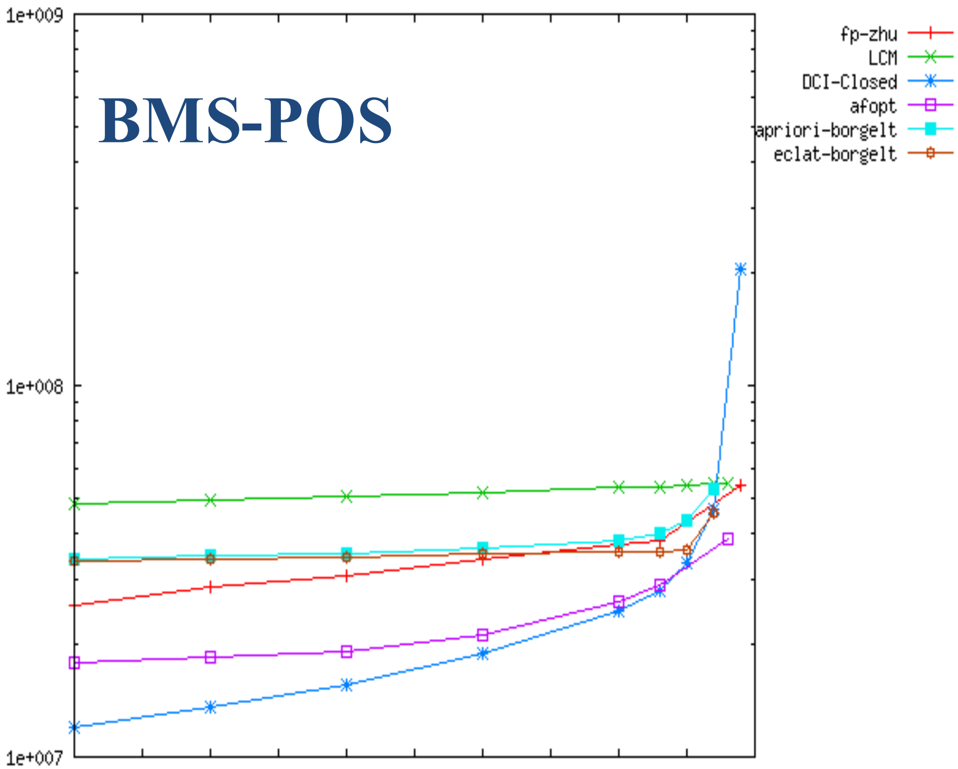


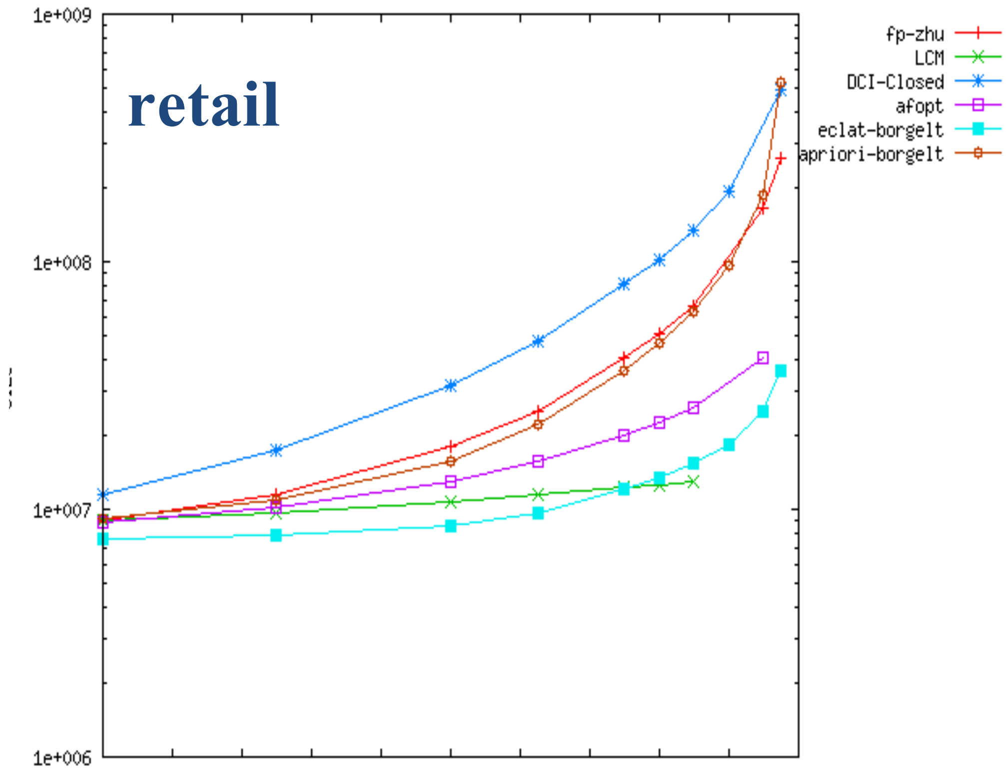




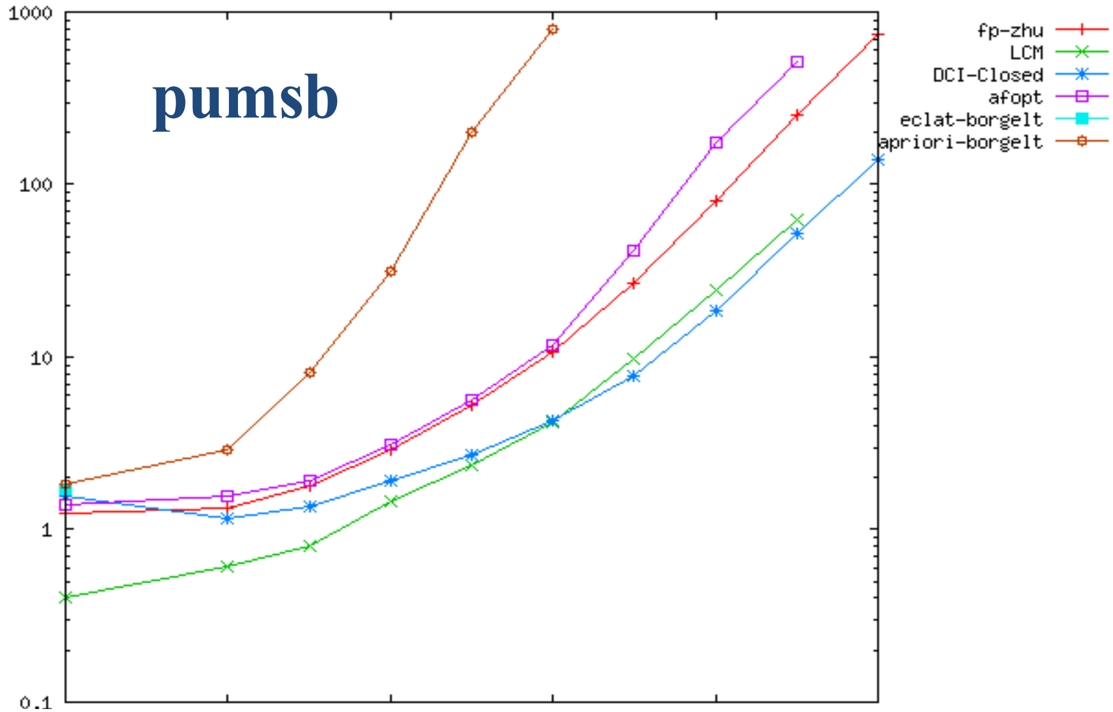
real-world data (sparse) memory usage

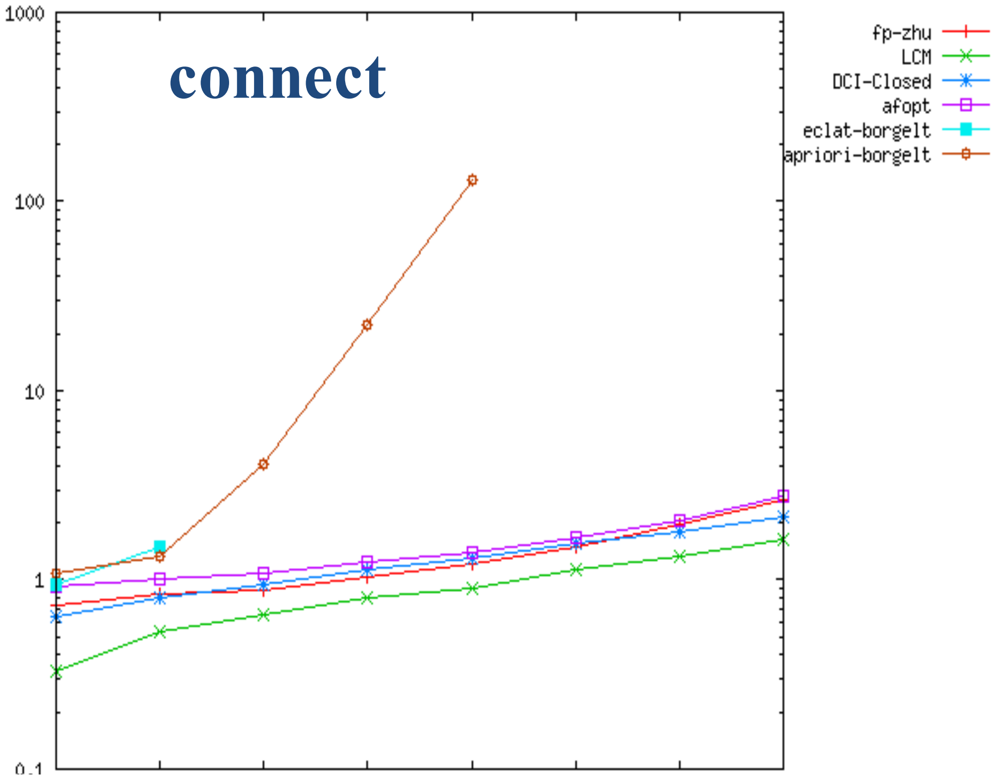


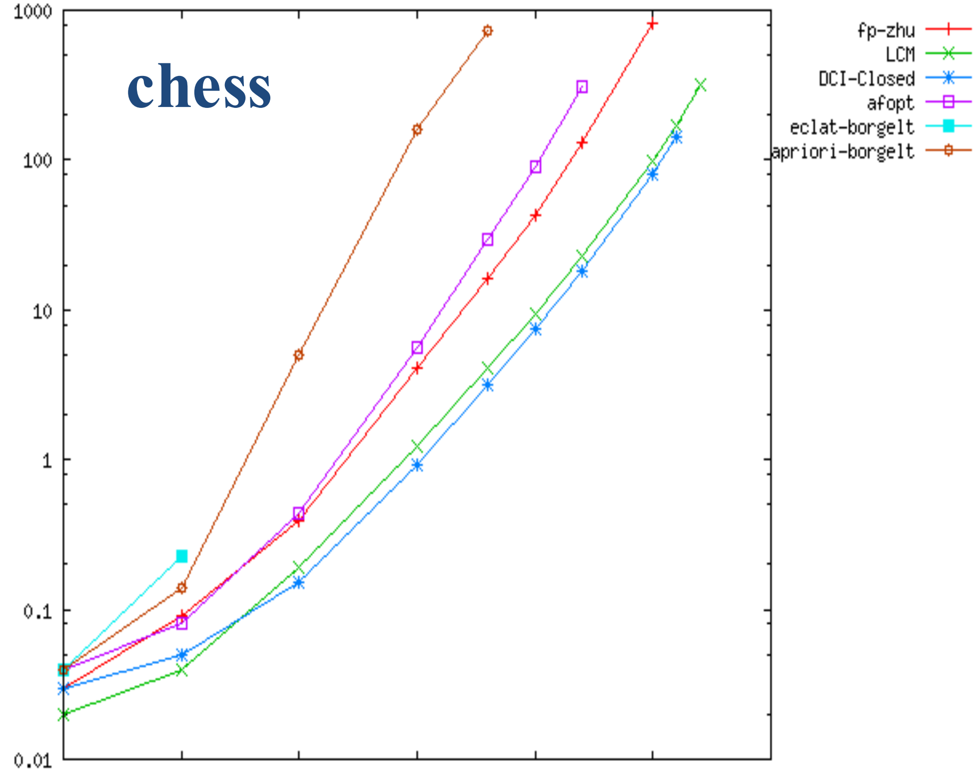




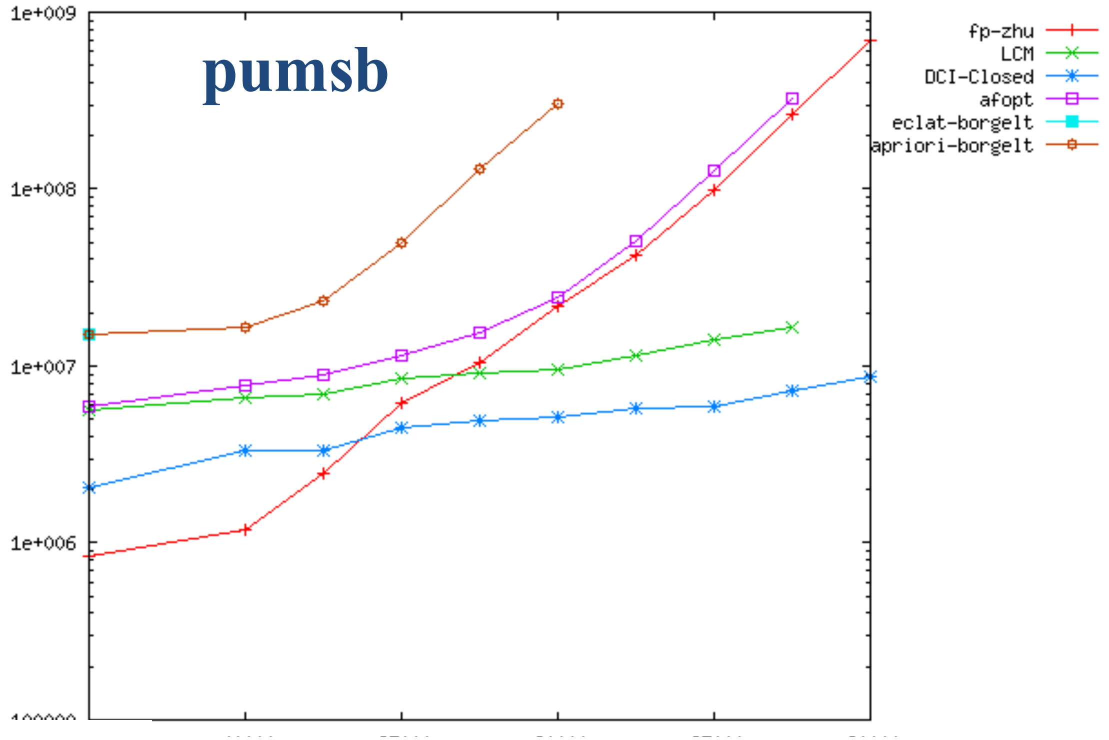
dense (50%) structured data

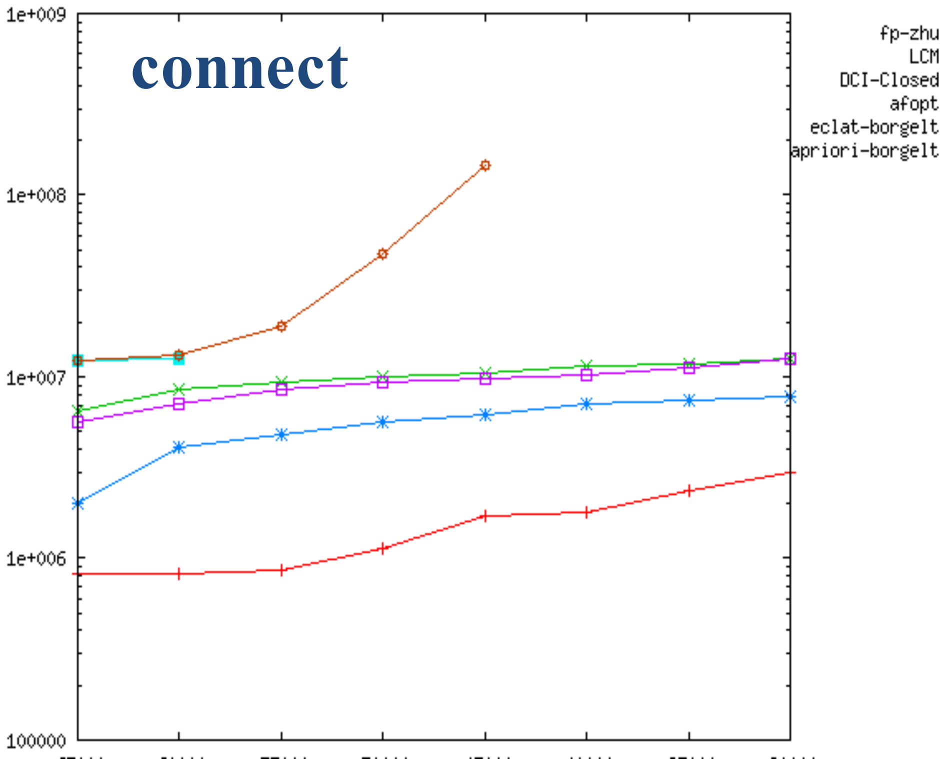


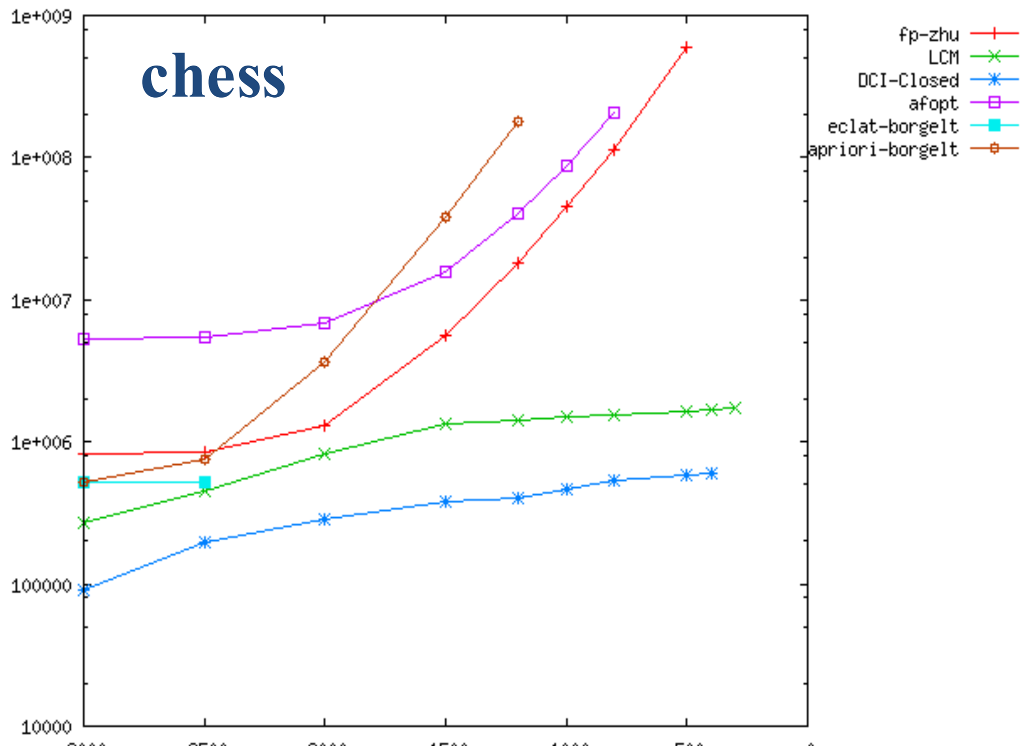




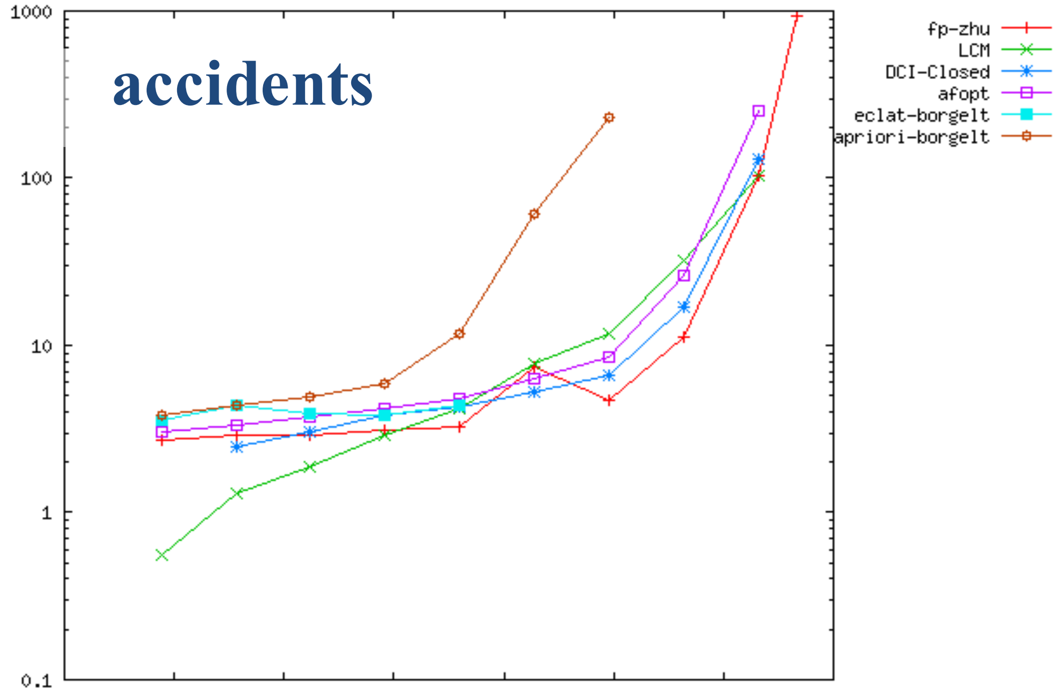
dense structured data, memory usage



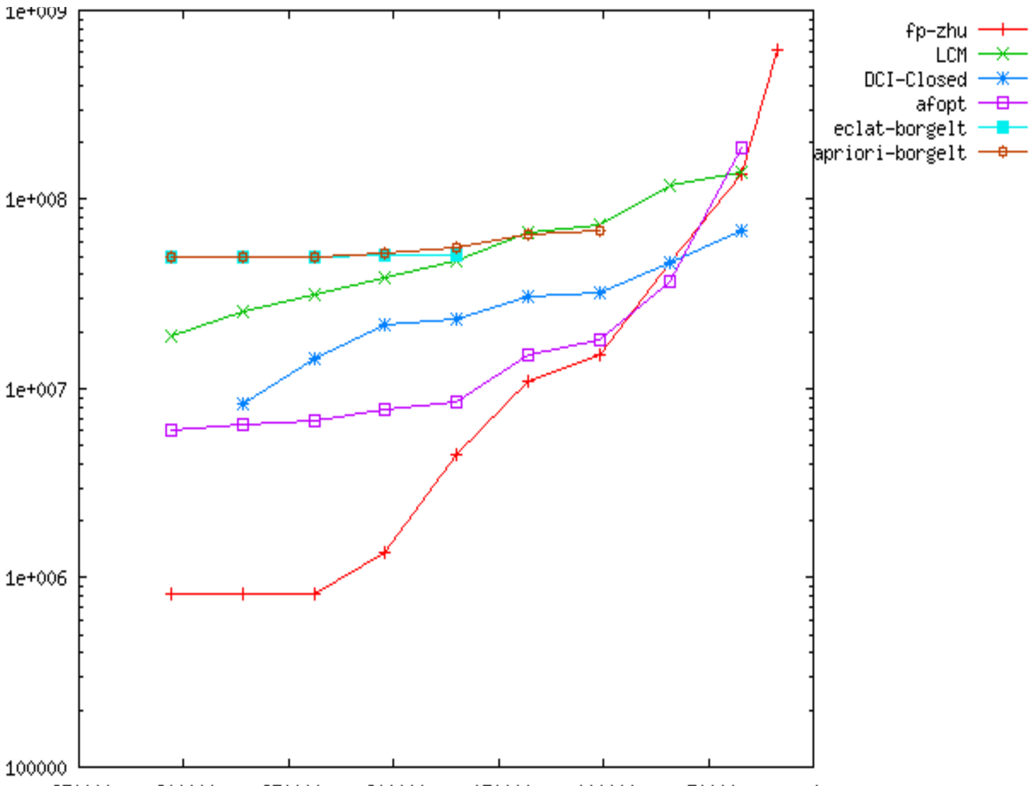


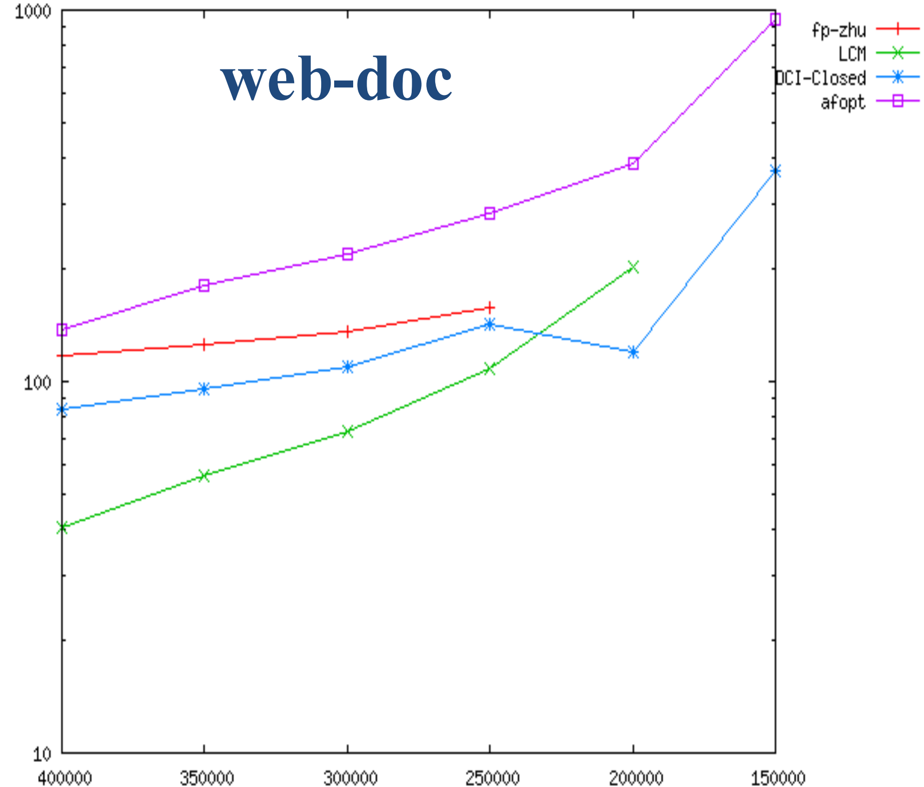


dense real data
large scale data



accidents memory

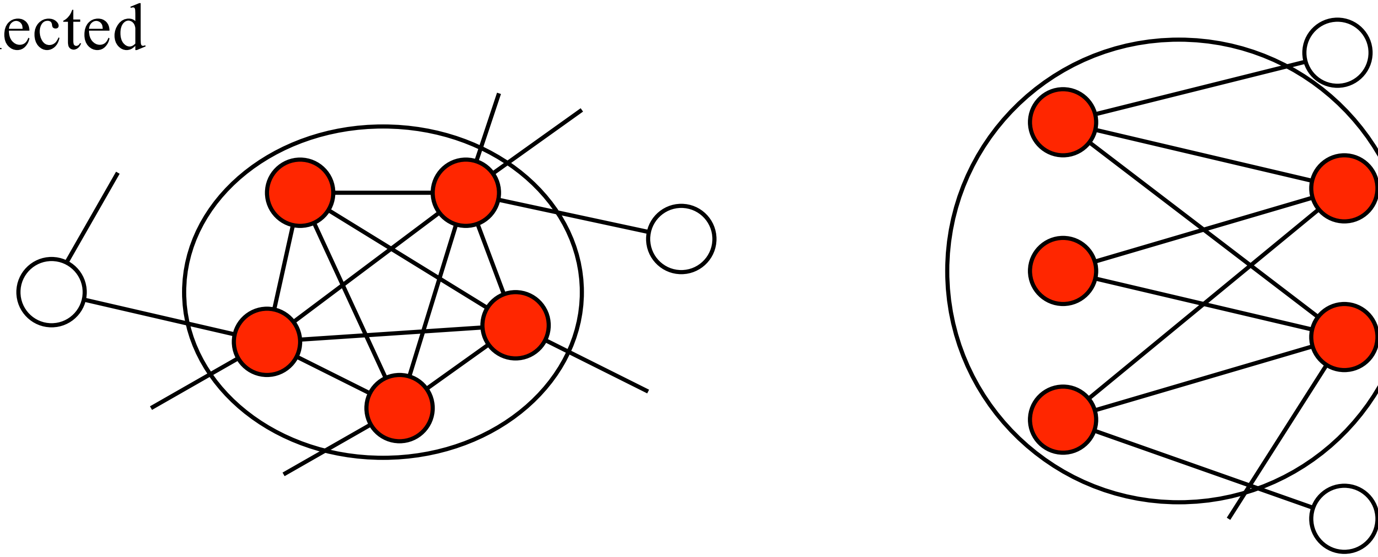




3-4 Maximal Clique Enumeration

Clique Enumeration

Clique: a subgraph that is a complete graph (any two vertices are connected



- Finding a maximum size is NP-complete
- Bipartite clique enumeration **is** converted to clique enumeration
- Finding a maximal clique is easy (**O(|E|)** time)
- Many researches and many applications, with many models

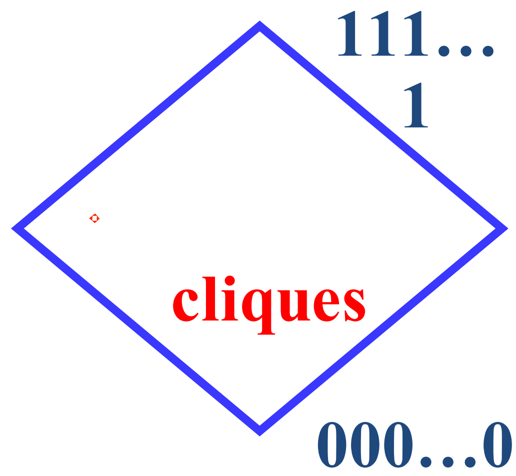
Monotone

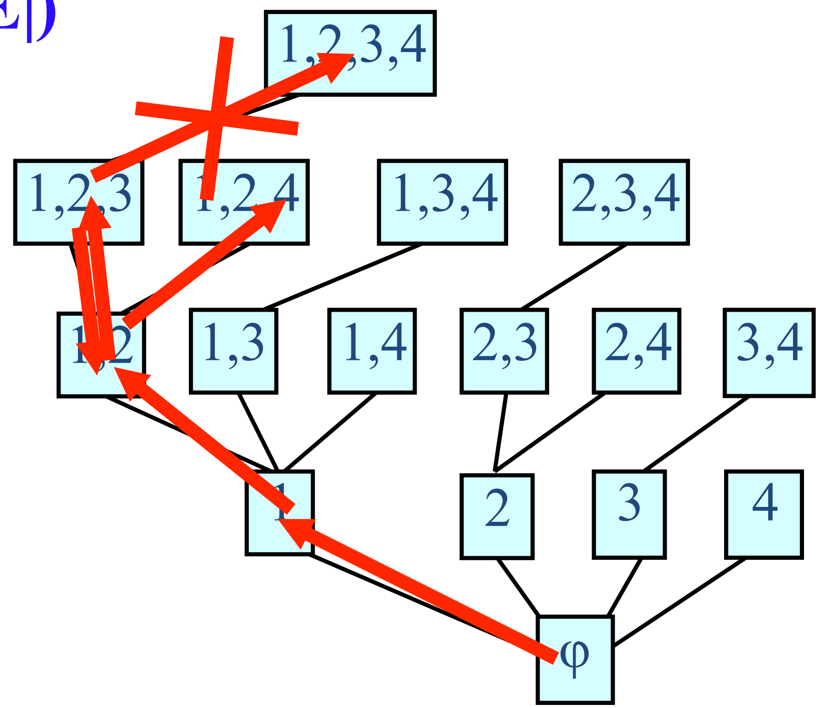
- Set of cliques is monotone, since any subset of a clique is also a clique

- Backtracking works

- The check being a clique takes **O(|E|)** time, and at most **|V|** recursive calls

- **O(|V| |E|)** per clique





Like Refine Search

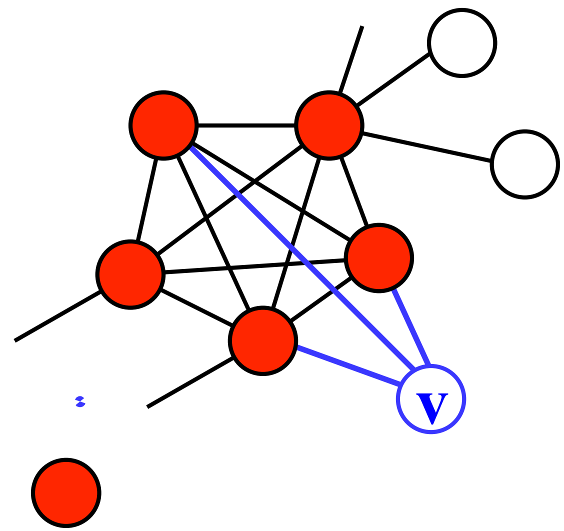
- … We want to find vertices can be added to a clique

Addible adjacent to all vertices of clique

keep the set of addible vertices (CAND) in advance

- When add a vertex **v** to clique,

addible vertex is still addible adjacent to **v**



The update involved by adding v

intersection of **CAND** and **N(v)**

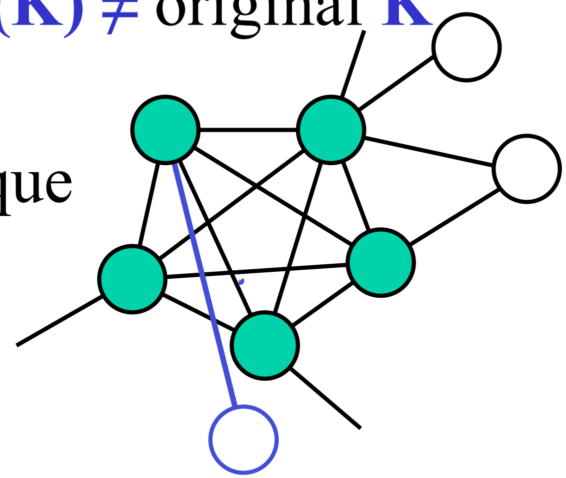
(**N(v)** is the neighbors of **v**)

O(δ(v)) time per iteration, where δ(v) is the degree of v

Adjacency on Maximal Cliques

- **C(K) :=** lexicographically smallest maximal clique including **K** (greedily add vertices from smallest index)

- For maximal clique **K**, remove vertices iteratively, from largest index
- At the beginning **C(K) = K**, but at some point **C(K)** ≠ original **K**



- Define the parent **P(K)** of **K** by the maximal clique (uniquely defined) .
- The lexicographically smallest maximal clique (= root) has no parent

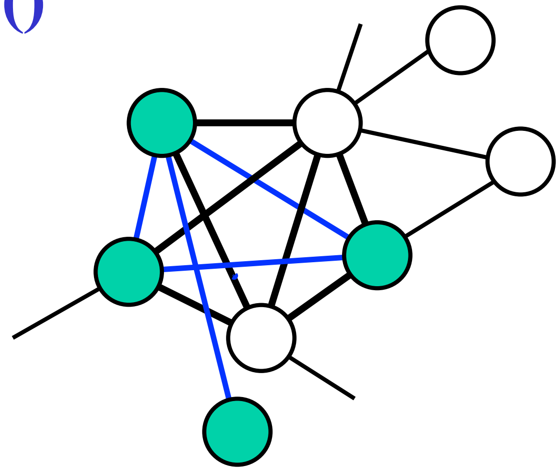
- **P(K)** is always lexicographically smaller than **K**

□ the parent-child relation is acyclic, thereby induces tree

Finding Children

- **K[v]** : The maximal clique obtained by adding vertex **v** to **K**, remove vertices not adjacent to **v**, and take **C()**

  □ **K[v] := C(K ∩ N(v)∪{v})**



- **K'** is a child of **K** □ **K' = K[v]** for some **v**

  K[v] for all v are sufficient to check

- For each **K[v]**, we compute **P(K[v])**

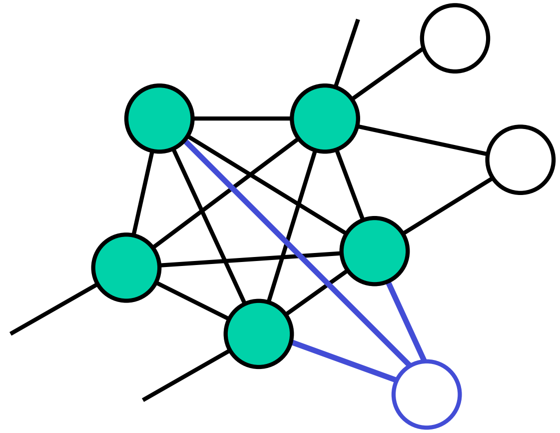
  If it is equal to **K** to, **K[v]** is a child of **K**

All children of K can be found by at most |V| checks, thus an iteration takes O(|V| |E|) time □ O(|V| |E|) per maximal clique

- Note that **C(K)** and **P(K)** can be computed in **O(|E|)** time

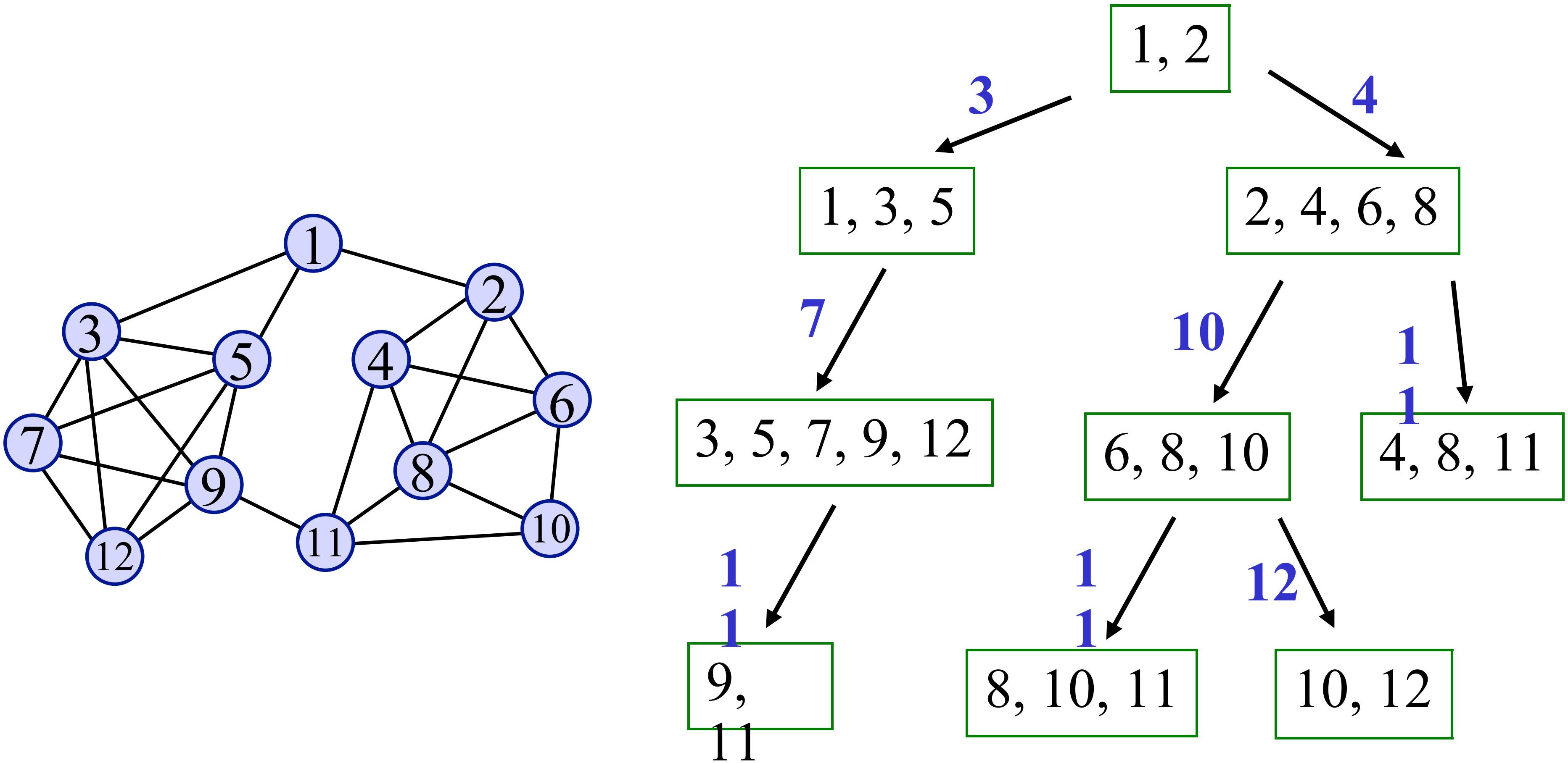
Pseudo Code for Maximal Clique

EnumMaxcliq (**K**)
1. **output K**
2. **for** each vertex **v** not in **K**
3. **K' := K[v]** (**= C(K∩N(v)∪v)**)
4. **if P(K') = K then call EnumMaxcliq** (**K**)
5. **end for**



Example

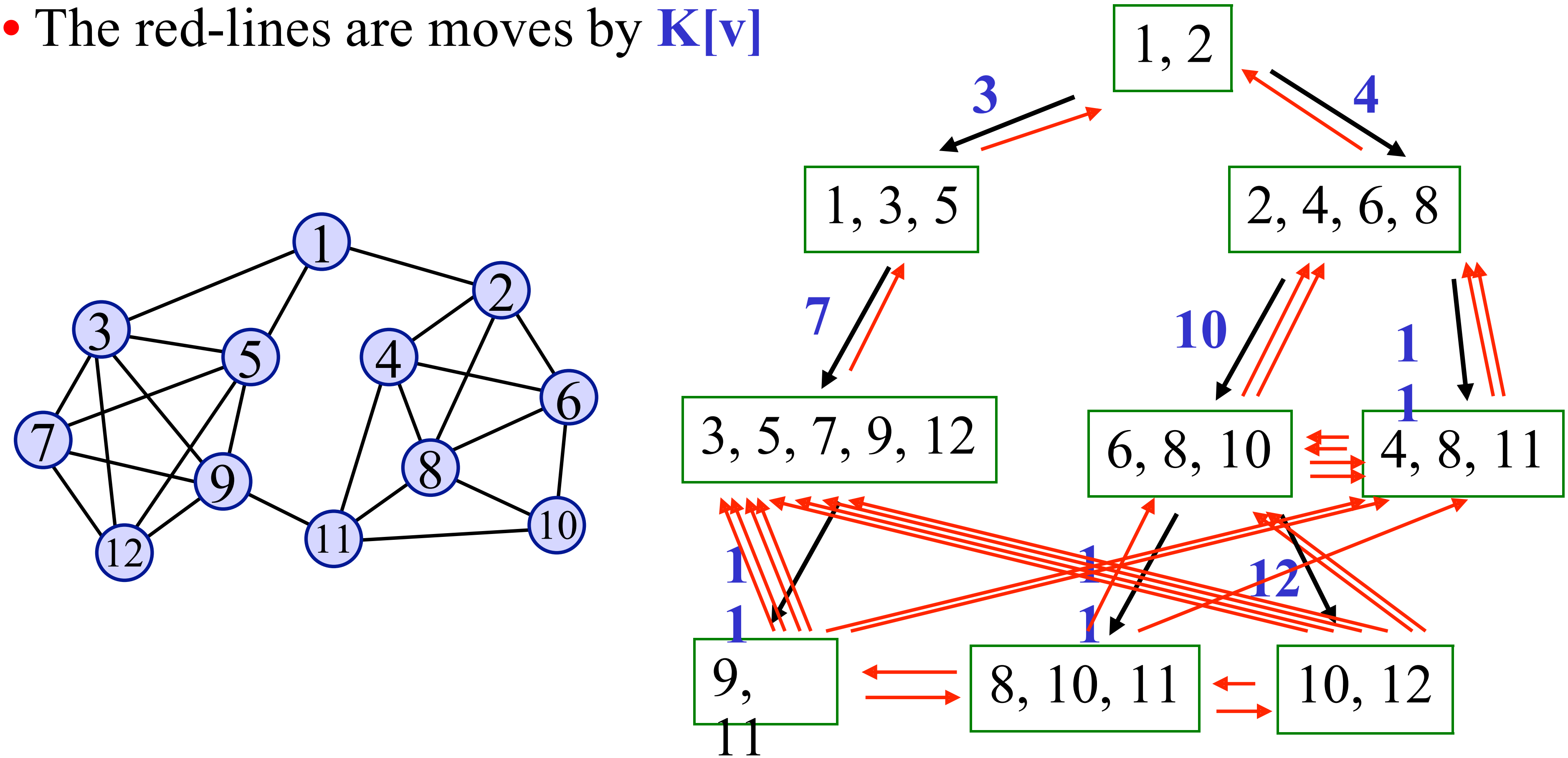
- The parent-child relation on the left graph



Example

- The parent-child relation on the left graph

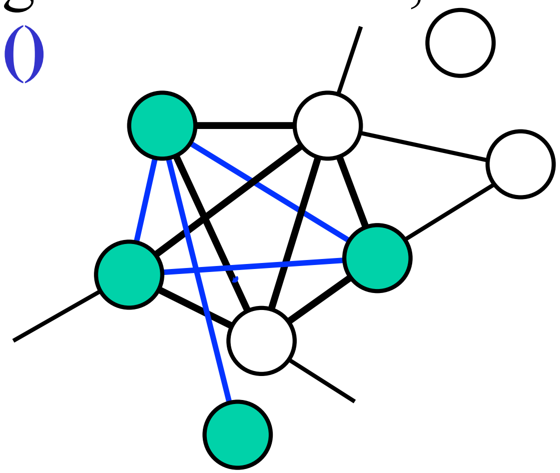
- The red-lines are moves by **K[v]**



Finding Children Quickly

- **K[v]** ： The maximal clique obtained by adding vertex **v** to **K**, remove vertices not adjacent to **v**, and take **C()**

  □ **K[v] := C(K ∩ N(v)∪{v})**



- **K'** is a child of **K** □ **K' = K[v]** for some **v**

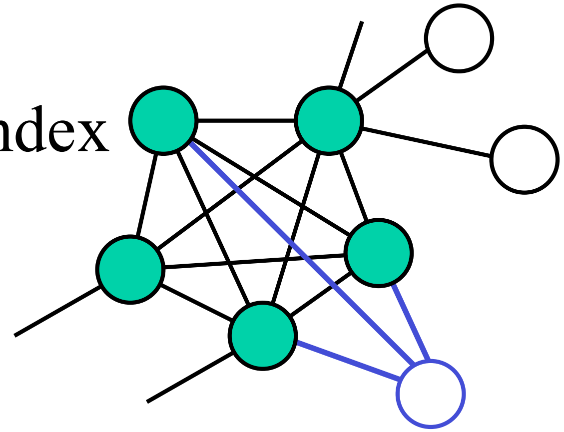
- **v** is adjacent no vertex in **K** □ **K[v]= C({v})** □ **P(K[v])** is root

□ if **K** ≠ root, **v** is adjacent none of **K** □ **K[v]** is not a child

We have to check only the vertices adjacent to some of **K**, that are at most **(Δ+1)2**

Computing C(K)

- **CAND(K)**: the set of vertices adjacent all vertices in **K**

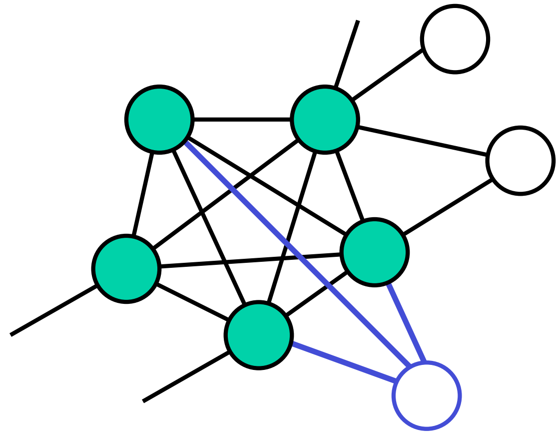


- To compute **C(K)**, we add to **K** the minimum index among **CAND(K)**, until **CAND(K) =φ**
- **CAND(K∪v)** = **CAND(K) ∩ N(v)** thus computable in **O(Δ)** time (**Δ:** maximum degree)
- Repetitions (=maximum clique size) is at most **Δ**, the total time is **O(Δ2)**

C(K) can be computed in **O(Δ2)** time

Computing P(K)

- Let **r(v)** be #vertices in **K** adjacent to **v**
  - □ **r(v) = |K|** ⇒ addible to **K**



- Delete vertices in **K** from maximum index, and update **r(v)** for all necessary **v** (deletion of **u** needs **O(δ(u))** time for update)

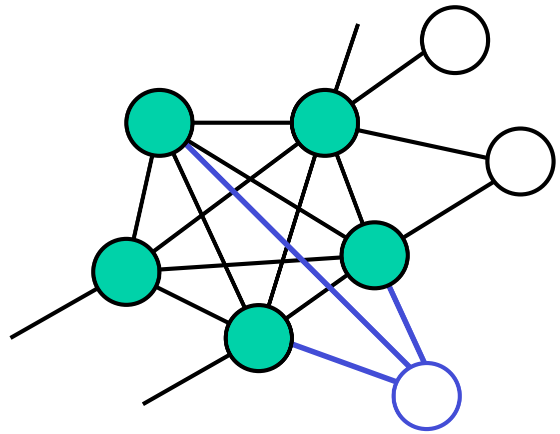
- If a vertex **v** satisfies **r(v) = |K|** after deleting **u**, compare **v** and **u**
- If **v < u**, **C(K-{,…,v})** never include **u**, thus it is the parent

P(K) can be computed in **O(Δ2)** time

Two Routines

Comp_CK (K={v1,…,vk})

1. K' := K, CAND := N(v1) ∩…∩ N(vk)
2. **if** CAND = ϕ **return** K'
3. v := minimum vertex in CAND
4. K' := K'∪v, CAND := CAND ∩ N(v)
5. **go to** 2.



Comp_PK (K)

compute r(v)

1. **for** each vertex v, r(v) := 0
2. **for** each vertex v in K,
   for each vertex u in N(v)-K, r(u) := r(u)+1
3. remove v from K that has maximum index
4. **for** each vertex u in N(v), r(u) := r(u)-1
5. find minimum u among vertices r(u) = |K|
6. **if** u < v or K = ϕ **then return** C(K)
7. **go to** 3.

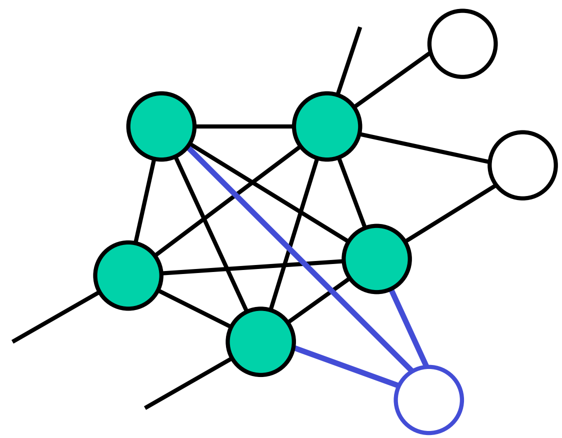
use buckets and update them for this

Complexity Analysis

EnumMaxcliq (K)
1. **output K** — $O(\Delta)$ time
2. **for** each vertex **v** adjacent to some vertices of **K** — $O(\Delta 2)$ repetitions
3.   **K' := K[v]**   (= C(K∩N(v)∪v)) — $O(\Delta 2)$ time
4.   **if P(K') = K then call EnumMaxcliq (K)**
5. **end for**

$O(\Delta 2)$ time

- Taken together, each iteration takes **$O(\Delta 4)$** time



References

Frequent Itemsets

T. Uno, M. Kiyomi, H. Arimura, LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets, ICDM'04 Workshop FIMI'04 (2004)

T. Uno, T. Asai, Y. Uchida, H. Arimura, An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases, LNAI 3245, 16-31 (2004)

A. Pietracaprina, D. Zandolin, Mining Frequent Itemsets using Patricia Tries, ICDM'03 Workshop FIMI'03 (2003)

C. Lucchese, S. Orlando, R. Perego, DCI Closed: A Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets, ICDM workshop FIMI'04 (2004)

G. Liu, H. Lu, J. Xu Yu, W. Wei, X. Xiao, AFOPT: An Efficient Implementation of Pattern Growth Approach, ICDM'03 Workshop FIMI'03 (2003)

J. Han, J. Pei, Y. Yin, Mining Frequent Patterns without Candidate Generation, SIGMOD 2000, 1-12 (2000)

G. Grahne, J. Zhu, Efficiently Using Prefix-trees in Mining Frequent Itemsets, ICDM'03 Workshop FIMI'03 (2003)

B. Racz, nonordfp: An FP-growth variation without rebuilding the FP-tree, ICDM'04 Workshop FIMI'04 (2004)

References

Clique

K. Makino, T. Uno, New Algorithms for Enumerating All Maximal Cliques, SWAT2004, LNCS 3111, 260-272 (2004)

E. Tomita, A. Tanaka, H. Takahashi, The Worst-case Time Complexity for Generating all Maximal Cliques and computational experiments", Theoretical Computer Science 363, 28-42 (2006)

D. Eppstein, D. Strash: Listing All Maximal Cliques in Large Sparse Real-World Graphs, SEA2011, LNCS 6630, 364-375 (2011)

D. Eppstein, M. Löffler, D. Strash, Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time, ISAAC2010, LNCS 6506, 403-414 (2010)

Exercise 3

Speed up

3-1. We want to design an algorithm for enumerating four cycles (cycles of length four) in a huge sparse graph. When the algorithm recursively adds an edge, how can we speed up iterations by removing unnecessary parts from input graphs recursively.

3-2. For given **m** permutations of **1,…,n**, we want to enumerate all subsequences appearing at least **k** of them. How can we reduce the database to reduce the computation time?

(subsequence is a sequence of numbers such that the numbers appear in the sequence without changing the order. For example, **(1,2,3)** is a subsequence of **(1,4,2,5,6,3)**.

3-3. We want to enumerate independent sets (no two vertices are connected). What data structure can we use to speed up iterations?

Speed up

3-4. We can construct an algorithm for enumerating all paths connecting given vertices **s** and **t**, by adding an edges one by one recursively. For large scale graphs, what should we do for modeling, and speeding up?

3-5. What kind of techniques should we use to speed up the algorithm for enumerating pseudo cliques in a large scale graph?

3-6. A leaf-elimination ordering of a tree **T** is a vertex ordering obtained by removing leaves of **T** iteratively. Design an algorithm for enumerating all leaf-elimination ordering, and way to speed up. Discuss about the complexity.

Speed up

3-7. A decreasing sequence of numbers **a1,…,an** is a subsequence **b1,…,bm** s.t. **bi > bi+1** holds for any **i** (subsequence is a sequence of numbers that appears in **a1,…,an** without changing the order). Design an algorithm to enumerate all "maximal" decreasing ordering (we assume that no two numbers are the same).

3-8. For a Markov chain defined on state set **V**, design an algorithm to enumerate all state sequences starting from **S∈V**, with moving 10 times. Discuss about speeding up.

Bottom-wideness

3-9. We first find a triangle **X** from a graph and iteratively add vertices to **X** which is adjacent to at least 3 vertices of **X**, to make a cluster (we do this to enumerate clusters). We want to enumerate all such structures, so how can we make the algorithm efficient?

3-10. For given a set of axis-parallel rectangles in a plane, we want to enumerate all rectangles obtained by intersecting of some rectangles in the set. Discuss available enumeration techniques, and #solutions.

3-11. For given a set of data strings, we want enumerate all strings s.t. there are at least **σ** substrings of some data strings have Hamming distance at most k to the string. Consider how to construct efficient algorithm with bottom-wideness.

3-12. Design an algorithm for enumerating all vertex sets **U** of a graph **G=(V,E)** s.t. the maximum degree in **G[U]** is at most **k**. Discuss about speeding up, and existence of polynomial time algorithm for enumerate only maximal ones.

3-13. For given a database whose records are graphs having a common vertex set, design an algorithm for enumerating pairs of graphs s.t., the symmetric difference between them is composed of at most **k** edges.