VMSP: Efficient Vertical Mining of Maximal Sequential Patterns

Philippe Fournier-Viger[1]

Cheng-Wei Wu[2]

Antonio Gomariz[3]

Vincent Shin-Mu Tseng[2]

[1]University of Moncton, Canada

[2]National Cheng Kung University, Taiwan

[3]University of Murcia

May 8 2014 – 2:20 PM
Université de Montréal, André-Aisenstadt building, room 1140

AI 2014

Introduction

Sequential pattern mining:

- a data mining task with wide applications
- finding frequent subsequences in a **sequence database**.

Example:

minsup = 2

Sequence database

SID	Sequences
1	$\langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{e\} \rangle$
2	$\langle \{a, d\}, \{c\}, \{b\}, \{a, b, e, f\} \rangle$
3	$\langle \{a\}, \{b\}, \{f\}, \{e\} \rangle$
4	$\langle \{b\}, \{f, g\} \rangle$

Some sequential patterns

ID	Pattern	Supp.
p1	$\langle \{a\}, \{f\} \rangle$	3
p2	$\langle \{a\}, \{c\}\{f\} \rangle$	2
p3	$\langle \{b\}, \{f,g\} \rangle$	2
p4	$\langle \{g\}, \{e\} \rangle$	2
p5	$\langle \{c\}, \{f\} \rangle$	2
p6…	$\langle \{b\} \rangle$	4

Algorithms

Different approaches to solve this problem

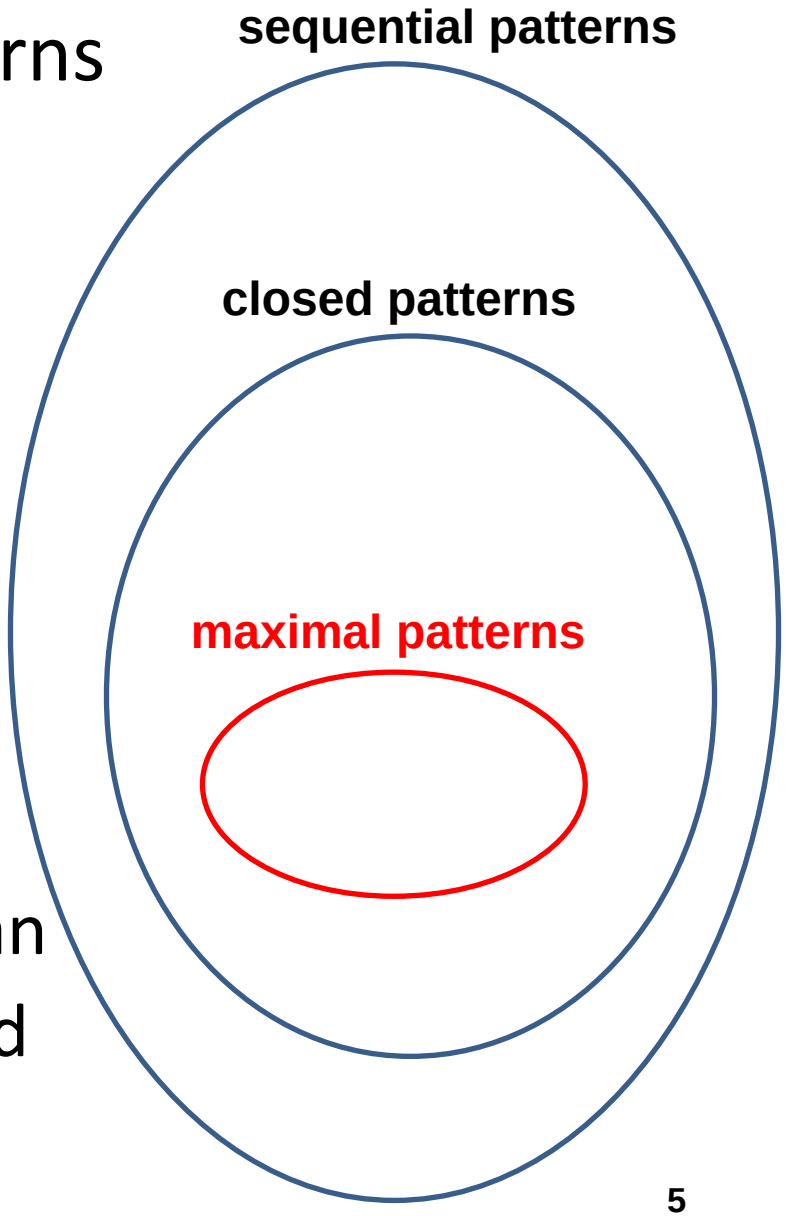
- Apriori-based
  (e.g. GSP)
- Pattern-growth
  (e.g. PrefixSpan)
- Discovery of sequential patterns using a vertical database representation
  (e.g. SPADE, SPAM, bitSPADE…)

The problem of redundancy

- **Observation**: if {a},{c},{f} is frequent, then the pattern {c},{f}, the pattern {a}, the pattern {c} … are frequent.
- Consider a frequent pattern of 20 distinct items.
- Its 2^{20}-1 subsequences are also frequent!
- Because of redundancy,
  - very time-consuming to analyze patterns,
  - require much more storage space.

A solution

- **Closed sequential patterns:** patterns that are not included in another pattern *having the same support*.
  - lossless
  - this set is still quite large for some applications
- **Maximal sequential patterns:** patterns that are not included in another pattern.
  - lossless with an extra database scan
  - generally much smaller than closed patterns



Multiple applications

- discovering frequent longest common subsequences in texts,
- analyzing DNA sequences,
- data compression,
- web log mining.

Example

A sequence database

SID	Sequences
1	⟨{a, b},{c},{f, g},{g},{e}⟩
2	⟨{a, d},{c},{b},{a, b, e, f}⟩
3	⟨{a},{b},{f},{e}⟩
4	⟨{b},{f, g}⟩

Patterns found for *minsup = 2*

Pattern	Sup.		Pattern	Sup.	
⟨{a}⟩	3	C	⟨{b},{g},{e}⟩	2	CM
⟨{a},{g}⟩	2		⟨{b},{f}⟩	4	C
⟨{a},{g},{e}⟩	2	CM	⟨{b},{f, g}⟩	2	CM
⟨{a},{f}⟩	3	C	⟨{b},{f},{e}⟩	2	CM
⟨{a},{f},{e}⟩	2	CM	⟨{b},{e}⟩	3	C
⟨{a},{c}⟩	2		⟨{c}⟩	2	
⟨{a},{c},{f}⟩	2	CM	⟨{c},{f}⟩	2	
⟨{a},{c},{e}⟩	2	CM	⟨{c},{e}⟩	2	
⟨{a},{b}⟩	2		⟨{e}⟩	3	
⟨{a},{b},{f}⟩	2	CM	⟨{f}⟩	4	
⟨{a},{b},{e}⟩	2	CM	⟨{f, g}⟩	2	
⟨{a},{e}⟩	3	C	⟨{f},{e}⟩	2	
⟨{a, b}⟩	2	CM	⟨{g}⟩	3	
⟨{b}⟩	4		⟨{g},{e}⟩	2	
⟨{b},{g}⟩	3	C			

C = Closed M = Maximal

Algorithms

- **MSPX**: approximate solution
- **DISMAPS**: for strings with no repeating items
- for the general problem:
  AprioriAdjust, MSPX, MFSPAN
  - AprioriAdjust is based on Apriori,
  - they all need to maintain a large set of intermediate candidates in memory during the mining process
- **MaxSP**:
  - most recent algorithm
  - does not maintain intermediate candidates in memory
  - only explore patterns occurring in the DB

Our proposal

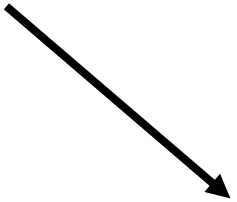
VMSP:

- discovers maximal sequential patterns,
- based on the SPAM search procedure
- integrates three novel strategies:
  - **EFN**: Efficient Filtering of Non-Maximal Patterns
  - **FME**: Forward Maximal Extension Checking
  - **CPC**: Candidate Pruning by Co-Occurrence Map

The SPAM search procedure

Step 1: creates a vertical representation of the database (SID lists):

SID	Sequences
1	$\langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{e\} \rangle$
2	$\langle \{a, d\}, \{c\}, \{b\}, \{a, b, e, f\} \rangle$
3	$\langle \{a\}, \{b\}, \{f\}, \{e\} \rangle$
4	$\langle \{b\}, \{f, g\} \rangle$



a	
SID	Itemsets
1	1
2	1,4
3	1
4	

b	
SID	Itemsets
1	1
2	3,4
3	2
4	1

c	
SID	Itemsets
1	2
2	2
3	
4	

d	
SID	Itemsets
1	
2	1
3	
4	

e	
SID	Itemsets
1	5
2	4
3	4
4	

f	
SID	Itemsets
1	3
2	4
3	3
4	2

g	
SID	Itemsets
1	3,4
2	
3	
4	2

The SPAM search procedure (2)

Step 2:

- identify frequent patterns containing a single item.
- recursively append items to each frequent pattern to generate larger patterns.
  - s-extension: <I1, I2, I3… In> with {a} is <I1, I2, I3… In, {a}>
  - i-extension: <I1, I2, I3… In> with {a} is <I1, I2, I3… In U{a}>
- The support of a larger pattern is calculated by intersecting SID lists:

a	
SID	Itemsets
1	1
2	1,4
3	1

support = 3

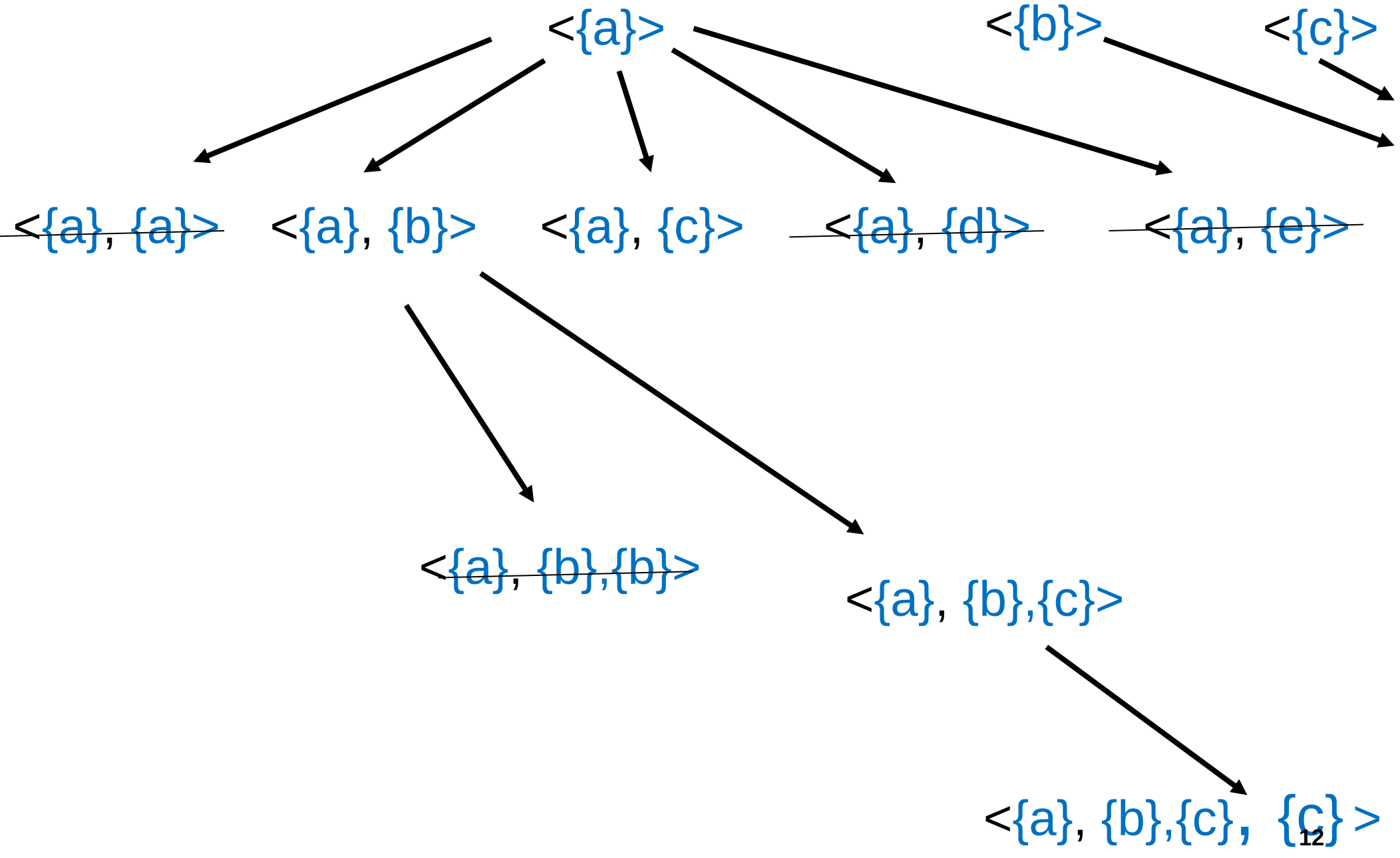
b	
SID	Itemsets
1	1
2	3,4
3	2
4	1

support = 4

<{a}, {b}>	
SID	Itemsets
1	1
2	1, 3
3	2

support = 3

The SPAM search procedure (3)

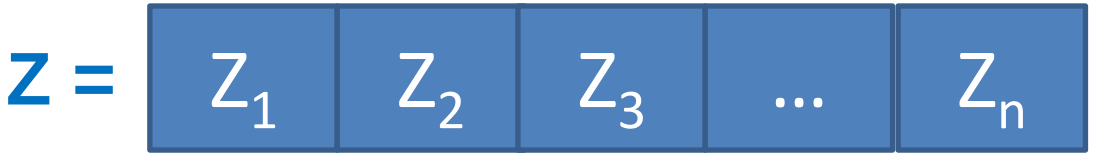


EFN: Efficient Filtering of Non-Maximal Patterns

- A structure **Z**
  - for storing maximal patterns
  - is initialized as empty.
- For each pattern $S = \{a_1, a_2, \ldots a_n\}$ found
  - **super-pattern checking:** if **S** is a subsequence of a pattern **X** in **Z**, then ***S*** is not maximal and is not added to **Z**.
    Otherwise, it is added to **Z.**
  - **sub-pattern checking:** if **S** is a super-sequence of a pattern **X** in **Z**, then ***X*** is not maximal and is removed from **Z**.

EFN: Efficient Filtering of Non-Maximal Patterns (cont'd)

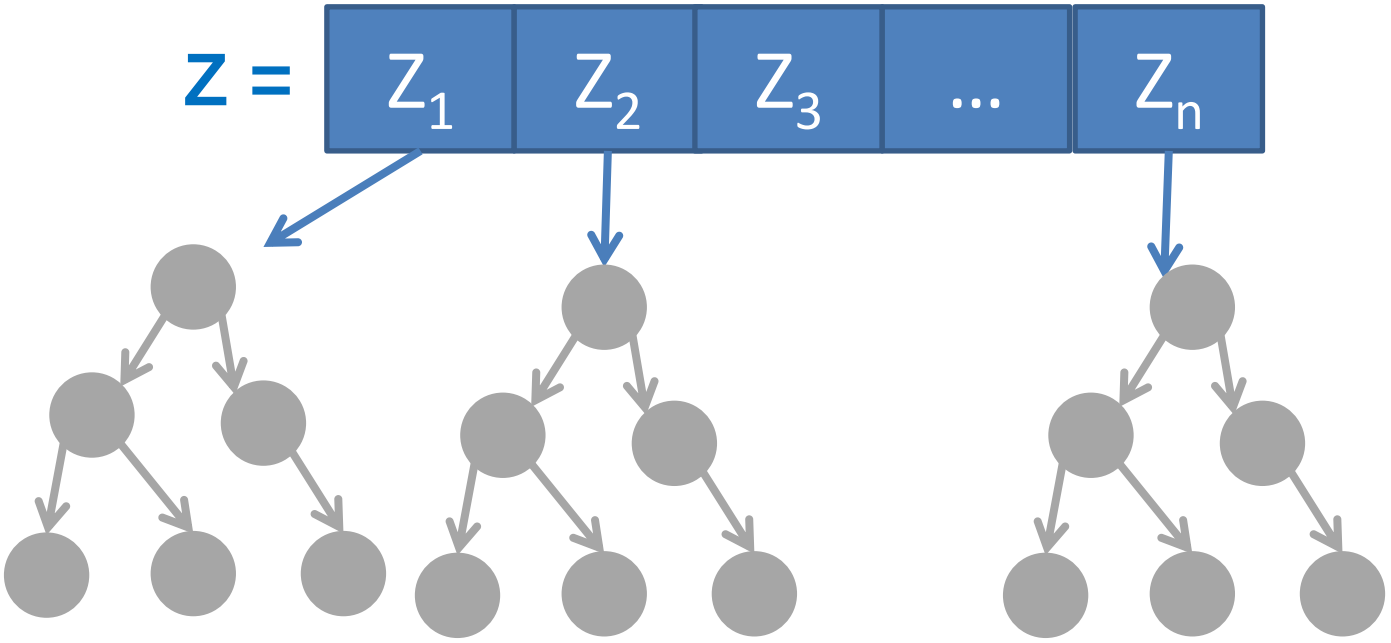
We implement **Z** as a List of heaps



The **k-th** list entry contains patterns of size **k**

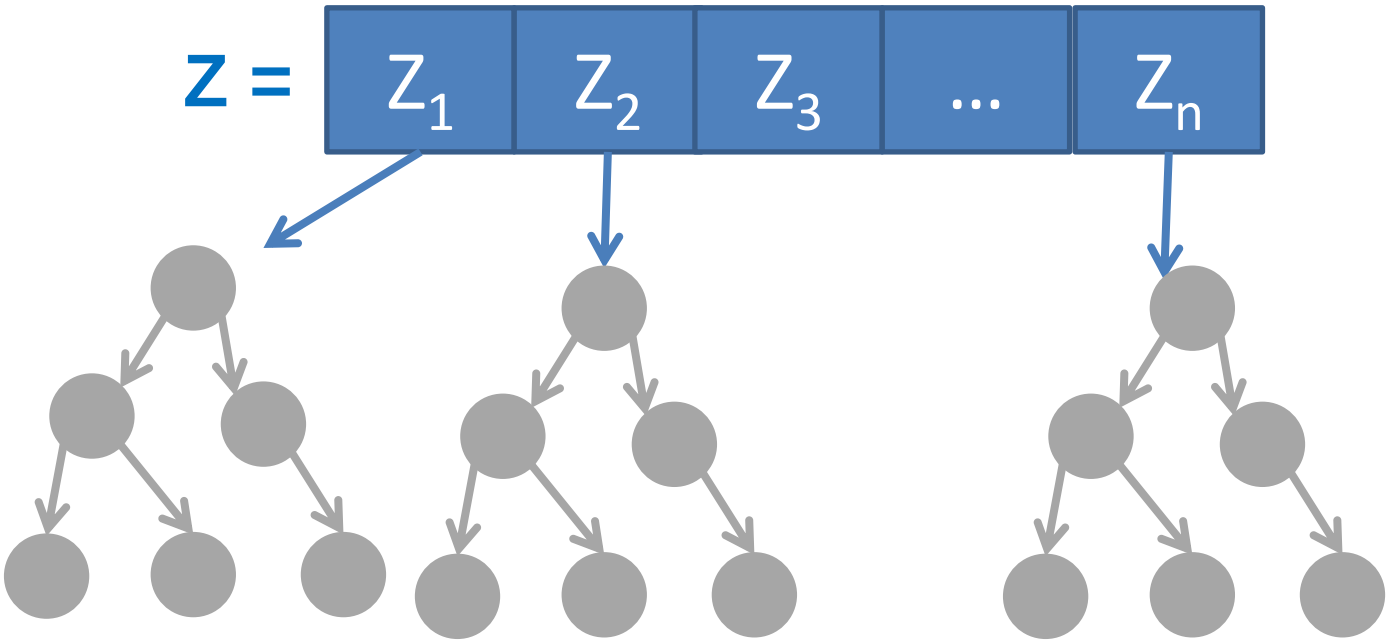
This allows to perform super-pattern checking and sub-pattern checking only with smaller and larger patterns

EFN: Efficient Filtering of Non-Maximal Patterns (cont'd)



- The **sum of items** in each pattern is calculated.
- Each **heap** orders patterns by decreasing sum of items.
- For each pattern S_a found and pattern S_b in Z_k, if $sum(S_a) < sum(S_a)$ we don't need to perform super-pattern checking with S_b and any following patterns in Z_k.
- Similar for sub-pattern-checking

EFN: Efficient Filtering of Non-Maximal Patterns (cont'd)



- **Support check optimization**:
  - A pattern cannot be contained in another pattern if its support is smaller.
  - A pattern cannot contain another pattern if its support is larger.

FME: Forward Maximal Extension Checking

- The algorithm performs a depth-first search (it grows patterns by appending items to smaller patterns one item at a time).
- We can avoid super-pattern checking for a pattern **S** if the recursive call to the search procedure with **S** produces a frequent pattern.

CPC: Candidate Pruning by Co-occurrence Map

- A structure **$CMAP_i$** stores every items that succeeds each item by **i-extension** at least *minsup* times.
- A similar structure **$CMAP_s$** stores every items that succeeds each item by **s-extension** at least *minsup* times.

$CMAP_i$	
item	is succeeded by (i-extension)
a	$\{b\}$
b	$\emptyset$
c	$\emptyset$
e	$\emptyset$
f	$\{g\}$
g	$\emptyset$

$CMAP_s$	
item	is succeeded by (s-extension)
a	$\{b, c, e, f\}$
b	$\{e, f, g\}$
c	$\{e, f\}$
e	$\emptyset$
f	$\{e, g\}$
g	$\emptyset$

This figure shows $CMAP_i$ and $CMAP_s$ when **minsup = 2**

CPC: Candidate Pruning by Co-occurrence Map

- **Pruning**: for a pattern **S**, an i-extension (s-extension) with an item **x** will result in an infrequent patterns if there exists a pair of items in the resulting pattern that is not in $CMAP_i$ ($CMAP_S$).
- This avoid performinig costly SID lists intersections.

$CMAP_i$	
item	is succeeded by (i-extension)
a	$\{b\}$
b	$\emptyset$
c	$\emptyset$
e	$\emptyset$
f	$\{g\}$
g	$\emptyset$

$CMAP_s$	
item	is succeeded by (s-extension)
a	$\{b, c, e, f\}$
b	$\{e, f, g\}$
c	$\{e, f\}$
e	$\emptyset$
f	$\{e, g\}$
g	$\emptyset$

This figure shows $CMAP_i$ and $CMAP_s$ when **minsup = 2**

Other optimizations

- SID lists are implemented as bitsets as in the **SPAM** and **BitSpade** algorithms.

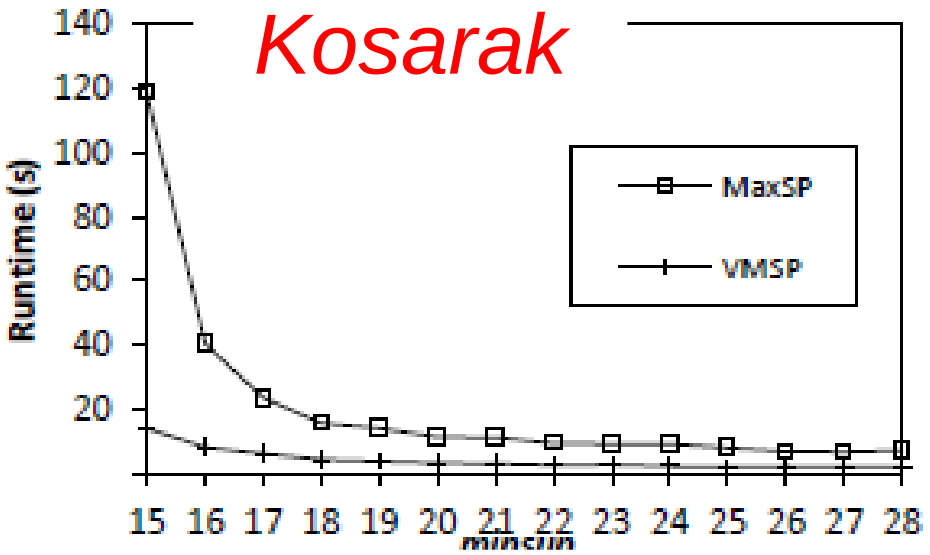
Experimental Evaluation

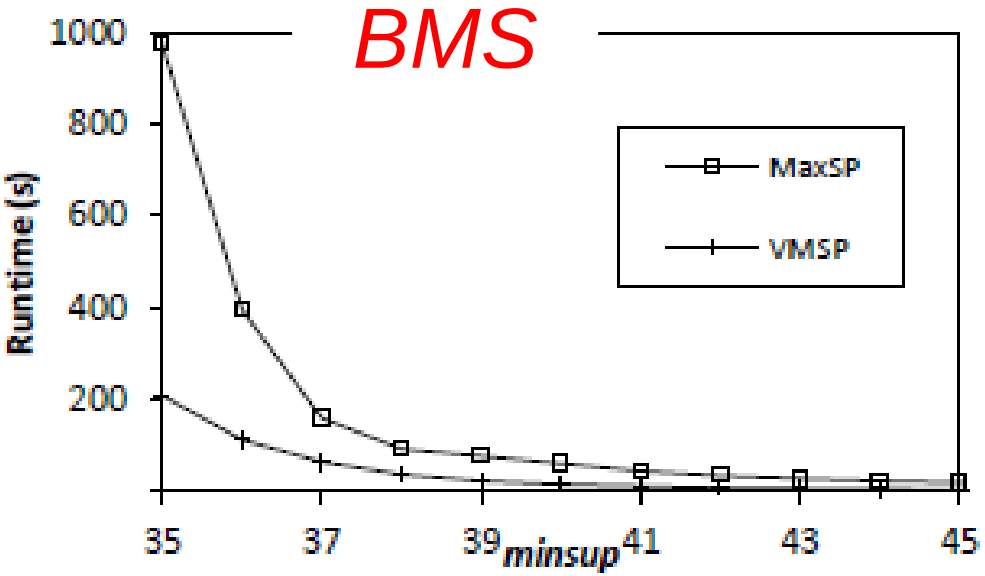
Datasets' characterictics

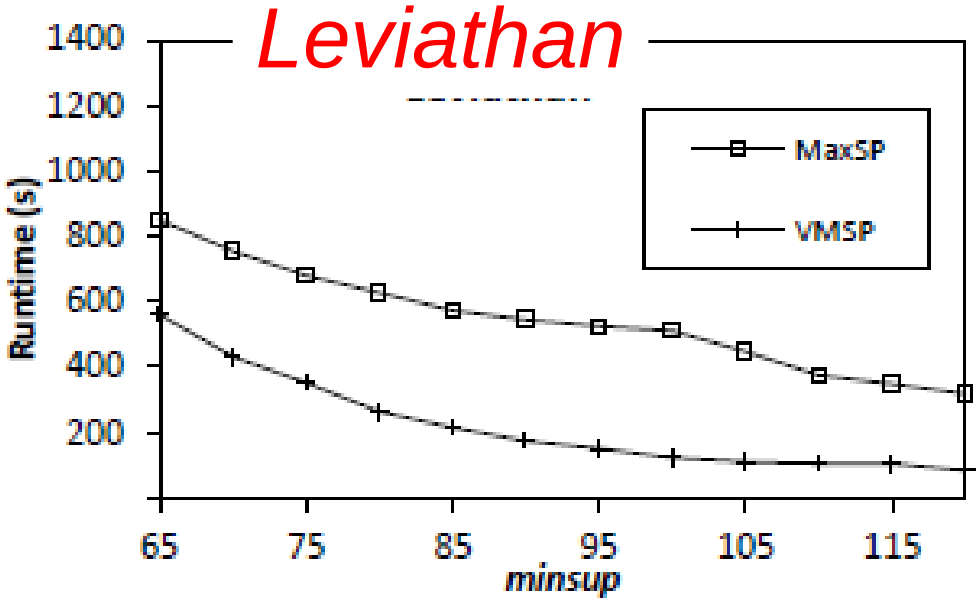
dataset	sequence count	item count	avg. seq. length (items)	type of data
Leviathan	5834	9025	33.81 (std= 18.6)	book
Snake	163	20	60 (std = 0.59)	protein sequences
FIFA	20450	2990	34.74 (std = 24.08)	web click stream
BMS	59601	497	2.51 (std = 4.85)	web click stream
Kosarak10k	10000	10094	8.14 (std = 22)	web click stream

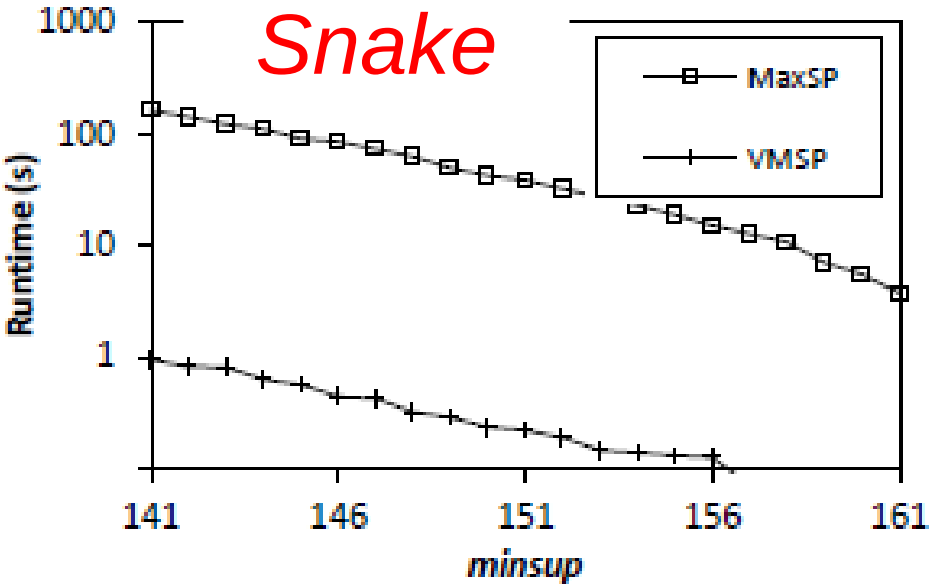
- VMSP vs MaxSP
- All algorithms implemented in Java
- Windows 7, 5 GB of RAM

Execution time



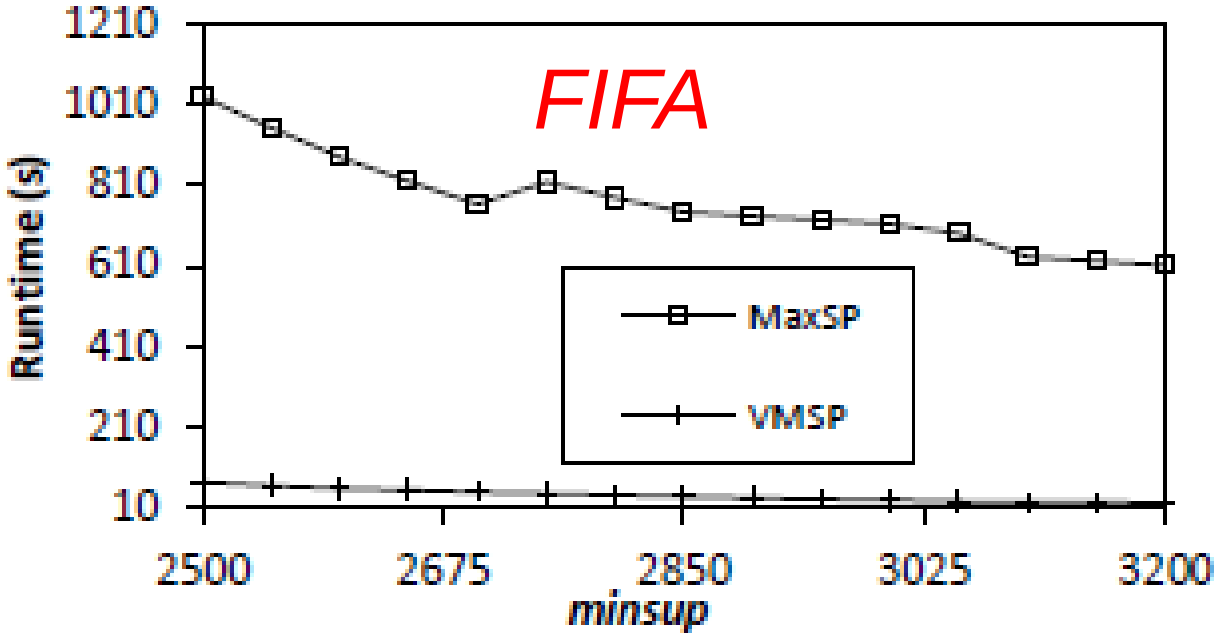






VMSP is up to 100 times faster than MaxSP

Execution time (cont'd)



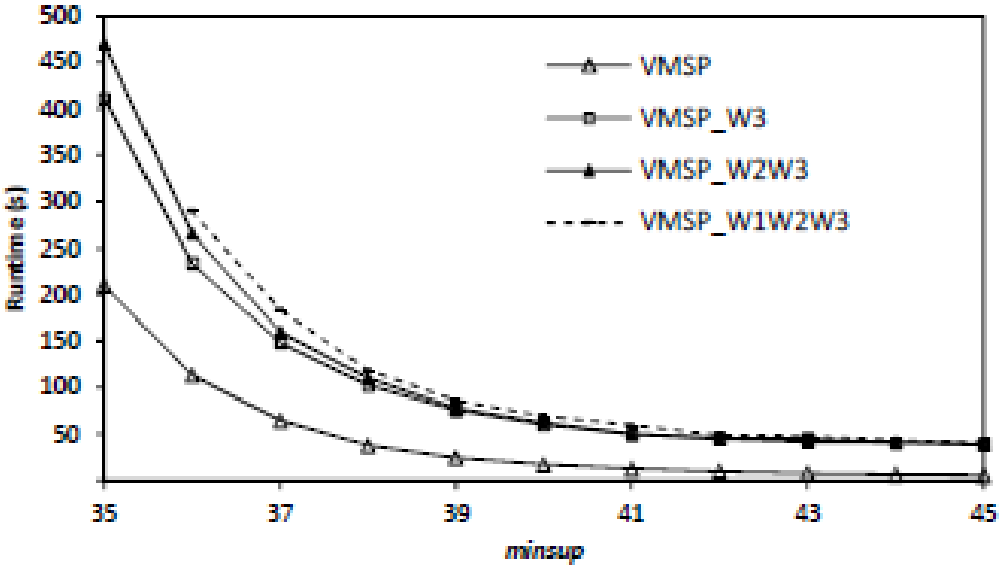
Maximum Memory Usage (MB)

Dataset	VMSP	MaxSP
BMS	840	**403**
Snake	**45**	380
Kosarak	1600	**393**
Leviathan	**911**	1150
FIFA	**611**	970

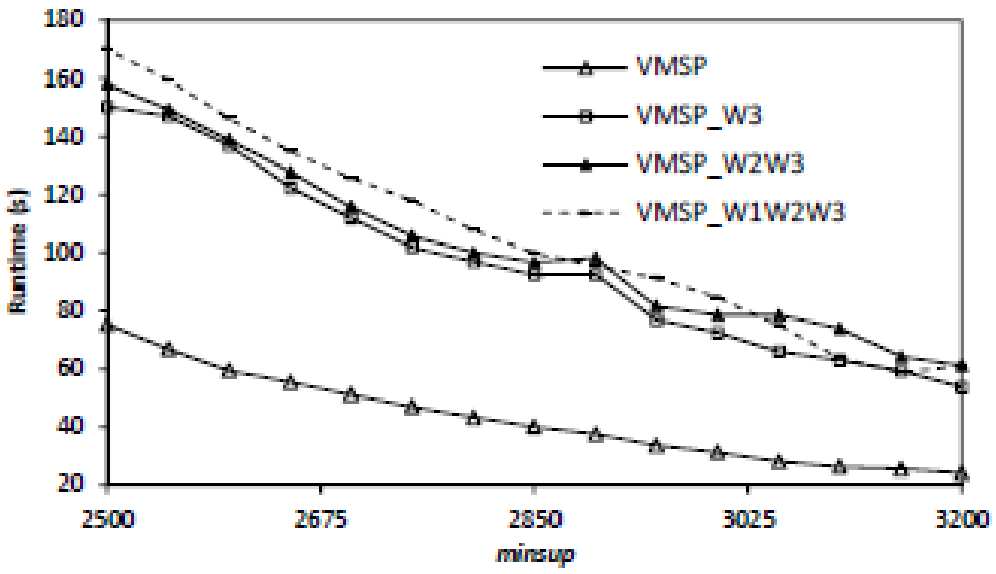
VMSP has the lowest memory consumption for 3 out of 5 datasets

Influence of the strategies

BMS



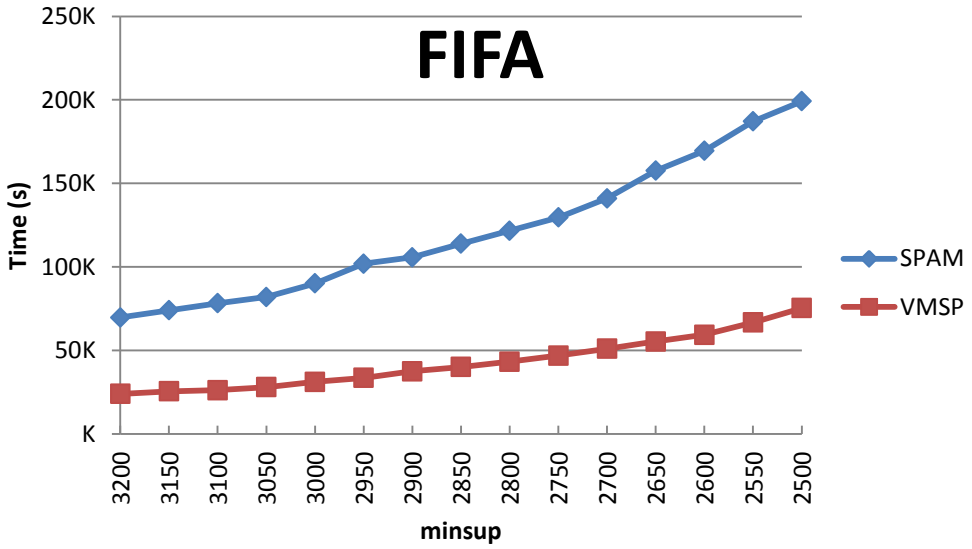
FIFA

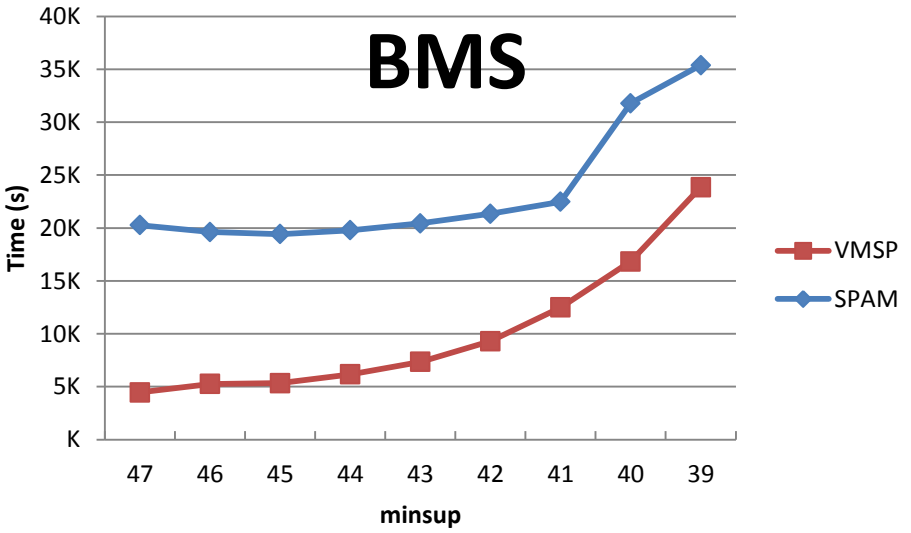


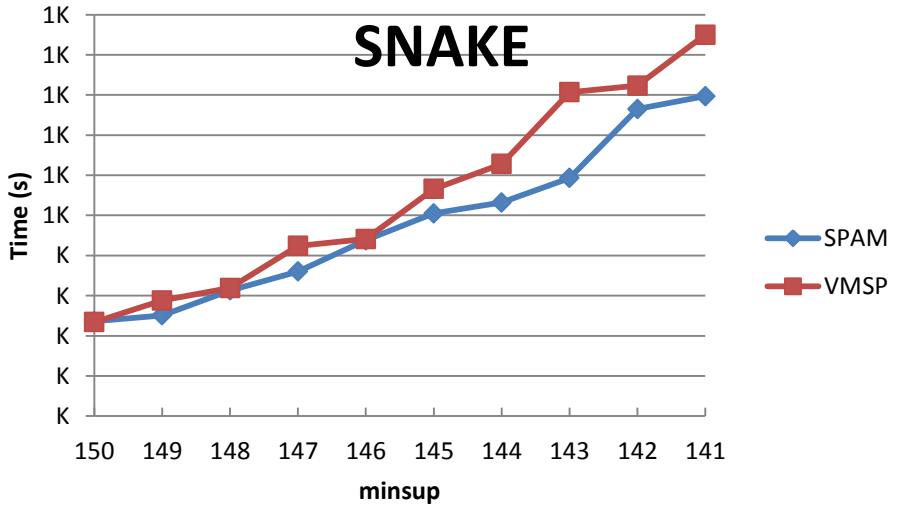
VMSP_W3 : without CPC strategy
VMSP_W2W3: without FME and CPC
VMSP W1W2W3: without FME, CPC and EFN

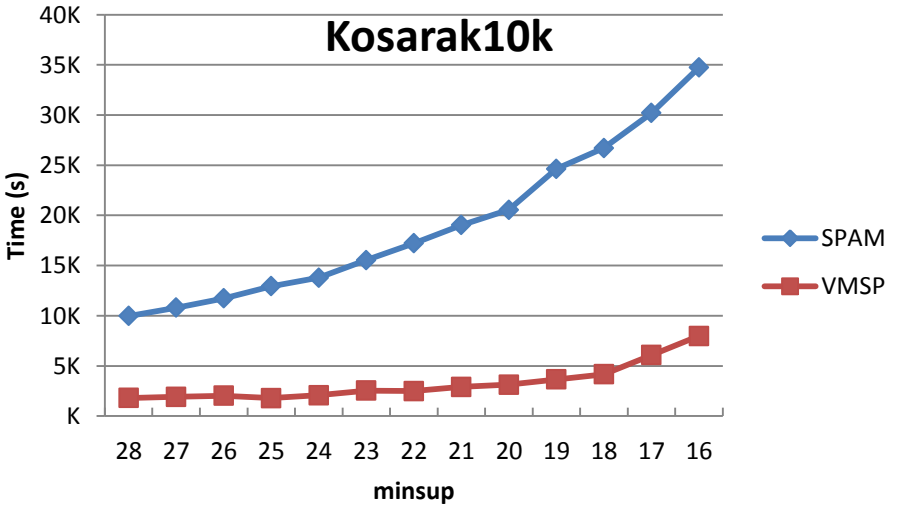
- Strategies improves the speed by up to 8 times.
- CPC is the most effective strategy

SPAM vs VMSP

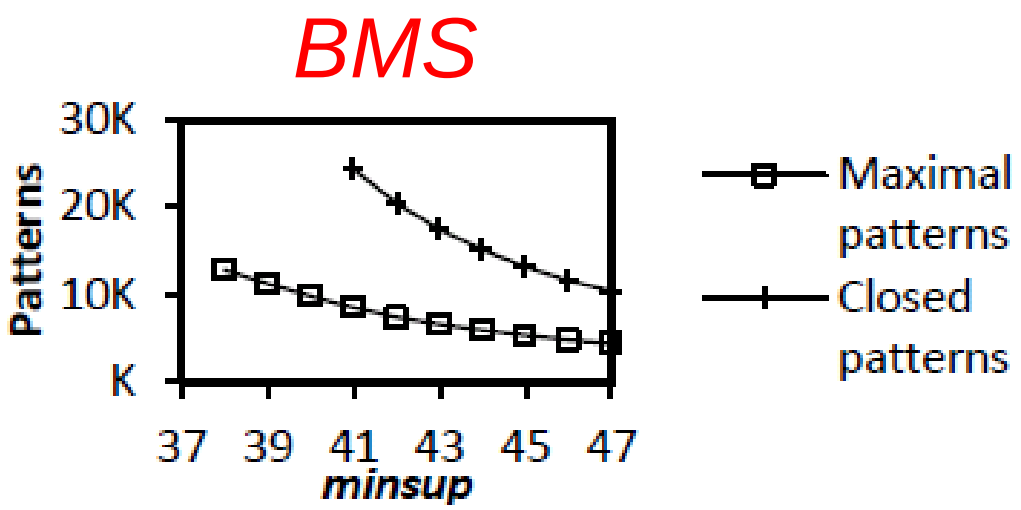

FIFA
Time (s)
250K
200K
150K
100K
50K
K
3200 3150 3100 3050 3000 2950 2900 2850 2800 2750 2700 2650 2600 2550 2500
minsup
SPAM
VMSP

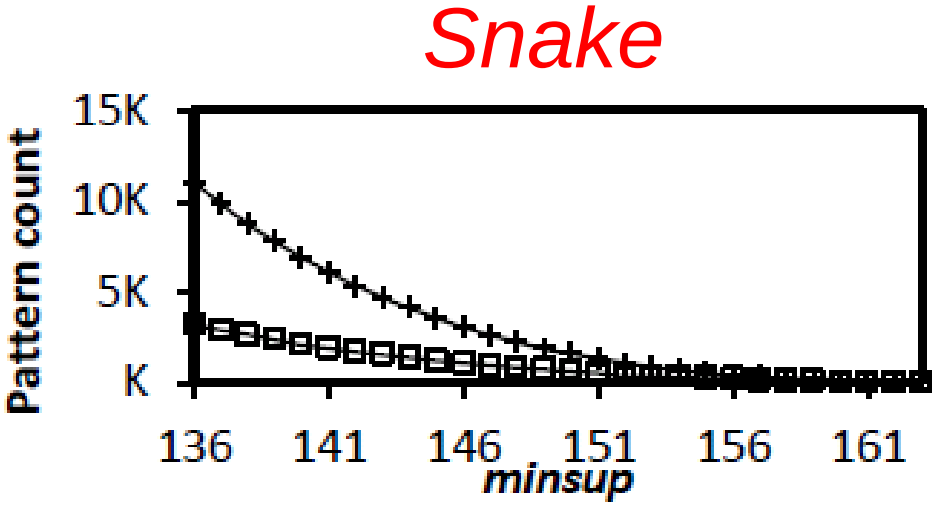

BMS
Time (s)
40K
35K
30K
25K
20K
15K
10K
5K
K
47 46 45 44 43 42 41 40 39
minsup
VMSP
SPAM

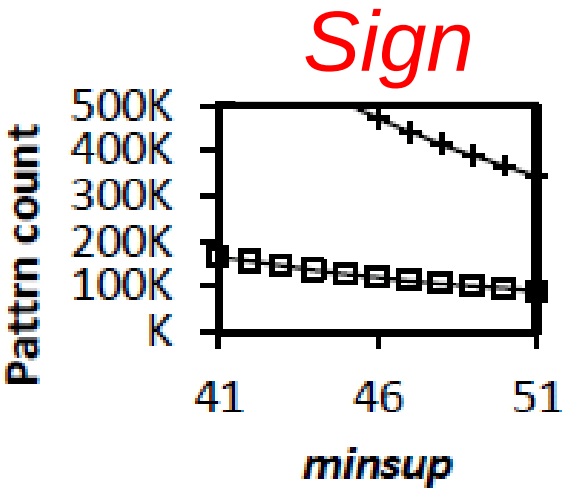

SNAKE
Time (s)
1K
1K
1K
1K
1K
1K
K
K
K
K
K
150 149 148 147 146 145 144 143 142 141
minsup
SPAM
VMSP

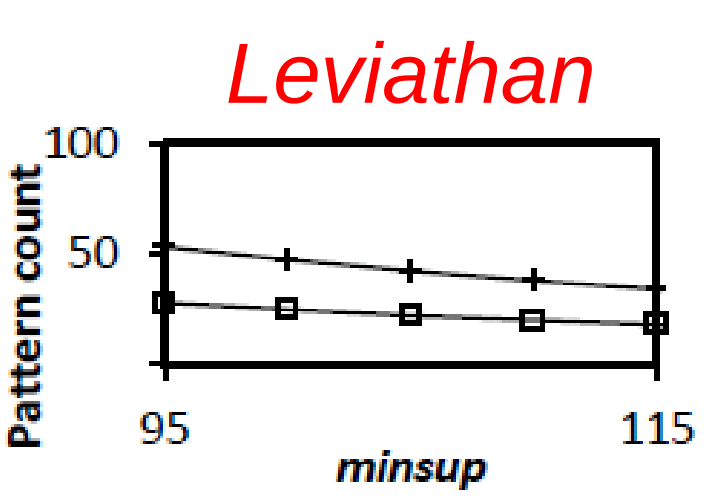

Kosarak10k
Time (s)
40K
35K
30K
25K
20K
15K
10K
5K
K
28 27 26 25 24 23 22 21 20 19 18 17 16
minsup
SPAM
VMSP

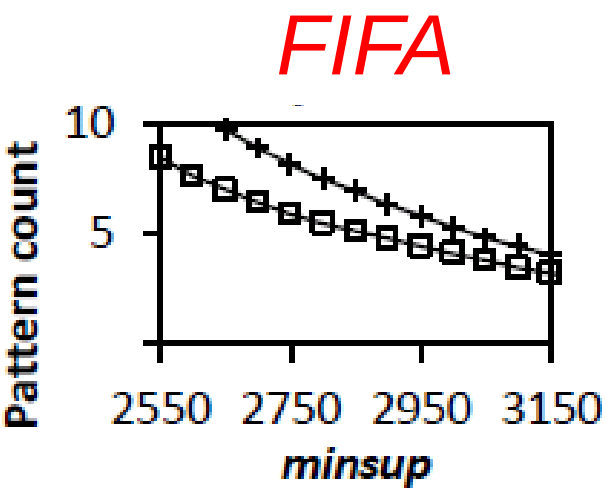
Pattern count











Much less maximal sequential patterns than closed patterns

eg.: Snake – 28 %, Sign = 25 %

Conclusion

- **VMSP**
  - a new vertical algorithm to discover **maximal sequential patterns**
  - includes three novel strategies:
    - **EFN**: Efficient Filtering of Non maximal patterns
    - **FME**: Forward-Maximal Extension checking
    - **CPC**: Candidate pruning with Co-occurrence map
  - up to 100 times faster than **MaxSP**
- Source code and datasets available as part of the **SPMF data mining library (**GPL 3).

Open source Java data mining software, 66 algorithms
http://www.phillippe-fournier-viger.com/spmf/

Thank you. Questions?

Open source Java data mining software, 55 algorithms
http://www.phillippe-fournier-viger.com/spmf/

This work has been funded by an NSERC grant.

Applications of SPMF

- Web usage mining
- Stream mining
- Optimizing join indexes in data warehouses
- E-learning
- Smartphone usage log mining
- Opinion mining on the web
- Insider thread detection on the cloud
- Classifying edits on Wikipedia
- Linguistics
- Library recommendation,
- restaurant recommendation,
- web page recommendation
- Analyzing DOS attack in network data
- Anomaly detection in medical treatment
- Text retrieval
- Predicting location in social networks
- Manufacturing simulations
- Retail sale forecasting
- Mining source code
- Forecasting crime incidents
- Analyzing medical pathways
- Intelligent and cognitive agents
- Chemistry