Surprising results of trie-based FIM algorithms

Ferenc Bodon

bodon@cs.bme.hu

Department of Computer Science and Information Theory,

Budapest University of Technology and Economics

supervisor: Lajos Rónyai

Purpose of the work

The three central FIM algorithms:

- APRIORI,
- Eclat,
- FP-growth.

Two of them use **tries**.

Small details have considerable influence on efficiency.

5 details were theoretically and experimentally examined.

What kind of a trie

Three-level specification:

- Trie type: full trie , pruned trie, collapsed trie ≈ Patricia, O-trie,

What kind of a trie

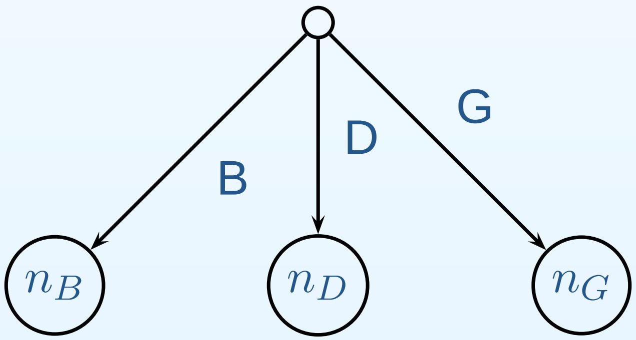
Three-level specification:

- trie type: full trie , pruned trie, collapsed trie ≈ Patricia, O-trie,
- edge representation:

What kind of a trie

Three-level specification:

- trie type: full trie, pruned trie, collapsed trie $\approx$ Patricia, O-trie,
- edge representation:



doubly-linked:

[(B,&n_B), (D,&n_D), (G,&n_G)]

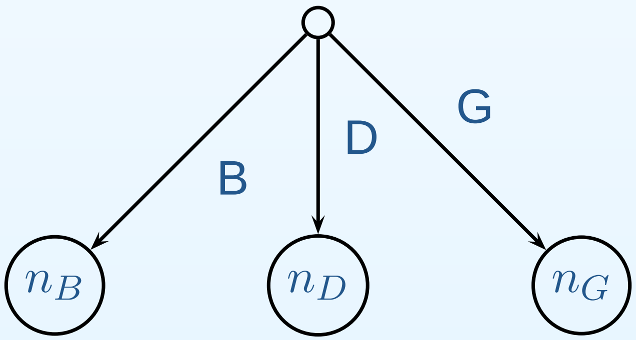
tabular:

[NIL, &n_B, NIL, &n_D, NIL, NIL, &n_G, NIL, $\cdots$]

What kind of a trie

Three-level specification:

- trie type: full trie , pruned trie, collapsed trie ≈ Patricia, O-trie,
- edge representation:



doubly-linked:

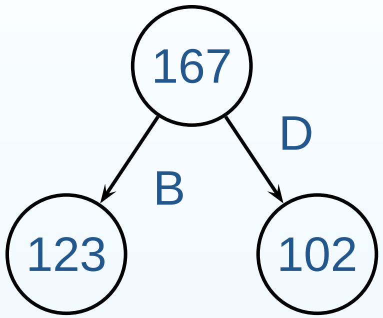
[(B,&n_B), (D,&n_D), (G,&n_G)]

tabular:

[NIL, &n_B, NIL, &n_D, NIL, NIL, &n_G, NIL, $\cdots$]

What kind of trie

- memory occupation

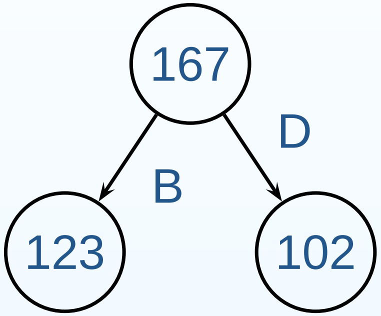


contiguous-memory based:

[2,167,B,6,D,8,0,123,0,102]

What kind of trie

- memory occupation



contiguous-memory based:

[2,167,B,6,D,8,0,123,0,102]

node-based:

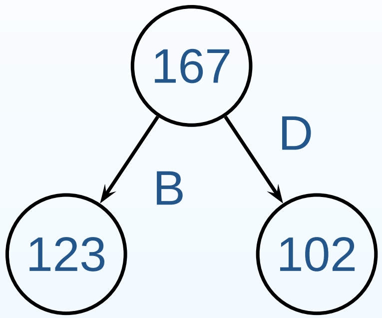
167,[B,·,D,·]

123,[]

102,[]

What kind of trie

- memory occupation



contiguous-memory based:

[2,167,B,6,D,8,0,123,0,102]

node-based:

167,[B,·,D,·]

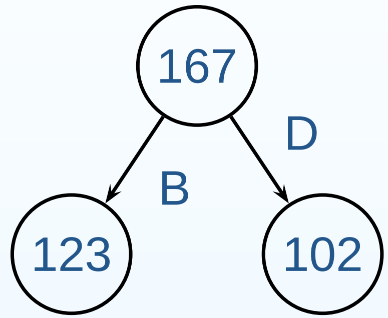
123,[]

102,[]

memory need:	16n (20n)	24n (28n)
modifi cation:	difficult	easy

What kind of trie

- memory occupation



contiguous-memory based:

[2,167,B,6,D,8,0,123,0,102]

node-based:

167,[B,·,D,·]

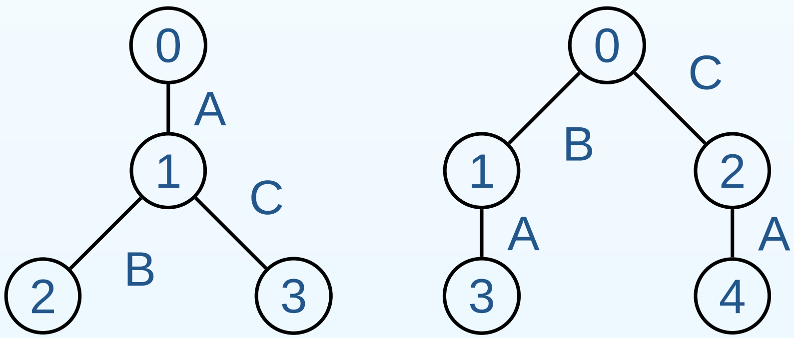
123,[]

102,[]

memory need:	16n (20n)	24n (28n)
modifi cation:	difficult	easy

1/5: effect of ordering

Tries store *sequences*.
sequence $\leftarrow$ itemset + total order on the items
The itemsets and the order together determines the trie



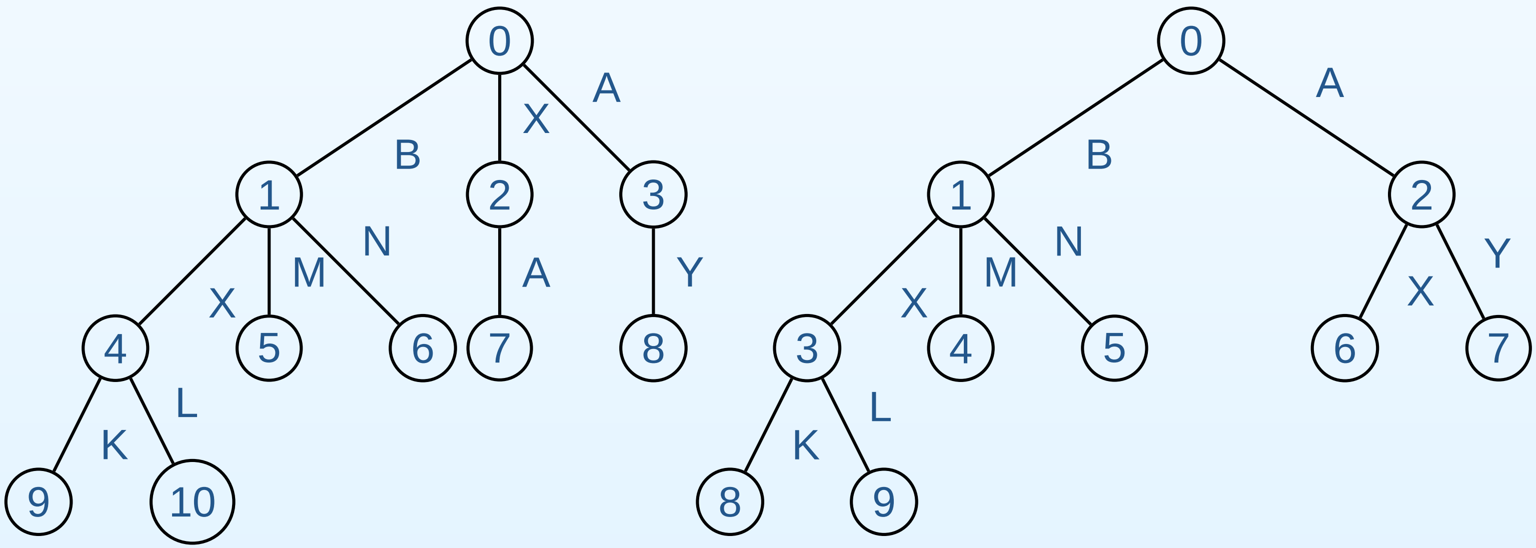
Question: Which order results in the minimum-size trie?

Theorem (Comer and Sethi). *Given I, $T \subseteq 2^I$ and integer k, it is NP-complete to decide if there exists a full trie that stores T, and the number of nodes is no more than k.*

1/5: effect of ordering

A simple heuristic: use the descending order according to the frequencies.
Reasoning: it has the most chance that two randomly chosen itemsets have the same prefix.

Example when heuristic does not result in the smallest trie:



1/5: effect of ordering

The heuristic works well on synthetic and real-life datasets. A kind of **homogeneity** exists.
Why is this important?

- In FP-growth: size of FP-tree is critical.
- In APRIORI: (1.) size of the trie that stores candidates is critical, (2.) order affects the support count method

Sensitivity of FP-tree:

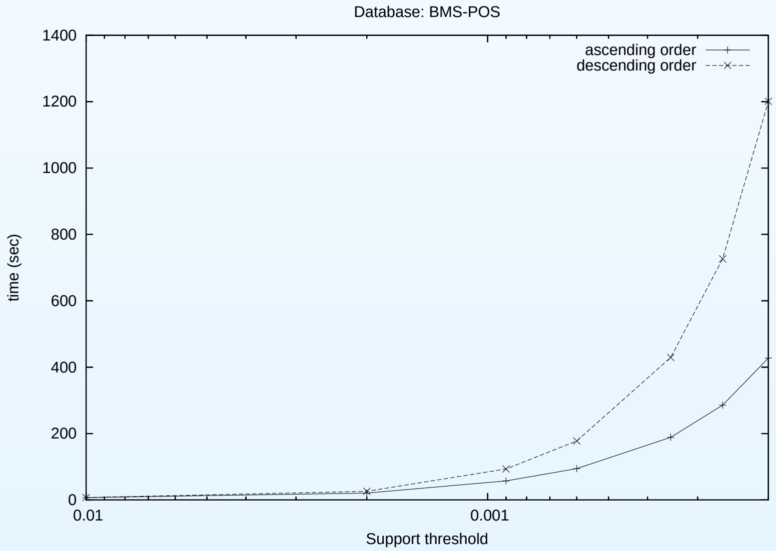
min_freq (%)	1	0.2	0.09	0.035	0.02
ascending	42.48	58.03	61.34	63.6	65.04
descending	27.58	39.74	41.69	43.66	44.10
random 1	29.84	42.30	44.49	46.60	46.41
random 2	36.98	48.97	55.02	56.85	56.72
random 3	34.87	52.18	55.68	58.01	55.50

Database: BMS-POS

1/5: effect of ordering

- In FIM the sensitivity does not matter.
- In FIM-related problems, where order can not be chosen freely this side-effect has to be taken into consideration

Support count of APRIORI and the order



1/5: effect of ordering

Results of the experiments:

- The memory need of APRIORI is not sensitive to the order.
- Ascending order according to frequencies results in the fastest APRIORI.

Argument: the most selective items are checked first.

2/5: storing the transactions

Let t be a transaction.
filtered t: infrequent items are removed from *t*.
Collect and store filtered transactions in memory

Advantages:

- IO cost is reduced,

2/5: storing the transactions

Let t be a transaction.
filtered t: infrequent items are removed from *t*.
Collect and store filtered transactions in memory

Advantages:

- IO cost is reduced,
- parsing costs are reduced,

2/5: storing the transactions

Let t be a transaction.
filtered t: infrequent items are removed from *t*.
Collect and store filtered transactions in memory

Advantages:

- IO cost is reduced,
- parsing costs are reduced,
- the number of support count method calls is reduced.

2/5: storing the transactions

Let t be a transaction.
filtered t: infrequent items are removed from *t*.
Collect and store filtered transactions in memory

Advantages:

- IO cost is reduced,
- parsing costs are reduced,
- the number of support count method calls is reduced.

Disadvantage:

- needs extra memory.

Question: What data-structure should be used?
Some possibilities: ordered list, trie, red-black tree

2/5: storing the transactions

Expectation: a trie needs the least memory.
Reasoning: it stores same prefixes only once.
Experiments:

min_ freq	sorted list	trie	RB-tree
0.05	12.4	52.5	13.8
0.02	16.2	76.0	17.1
0.0073	17.0	81.5	18.0
0.006	17.1	81.7	18.1

Database: T40I10D100K

In most cases trie needs the most memory (exception: `connect`, `accidents`)

Cause: a trie has much more nodes than a RB-tree has, and a node is expensive.

3/5: routing strategies at the nodes

How to find the edge to follow in APRIORI?
Given a node with a list of n edges and a part of the filtered transaction (t'), find matching labels.

- simultaneous traversal, $O(n + |t'|)$

3/5: routing strategies at the nodes

How to find the edge to follow in APRIORI?
Given a node with a list of n edges and a part of the filtered transaction (t'), find matching labels.

- simultaneous traversal, $O(n + |t'|)$
- binary search, $O(|t'| \log n)$ or $O(n \log |t'|)$

3/5: routing strategies at the nodes

How to find the edge to follow in APRIORI?
Given a node with a list of n edges and a part of the filtered transaction (t'), find matching labels.

- simultaneous traversal, $O(n + |t'|)$
- binary search, $O(|t'| \log n)$ or $O(n \log |t'|)$
- binary vector based, $O(n)$

3/5: routing strategies at the nodes

How to find the edge to follow in APRIORI?
Given a node with a list of n edges and a part of the filtered transaction (t'), find matching labels.

- simultaneous traversal, $O(n + |t'|)$
- binary search, $O(|t'| \log n)$ or $O(n \log |t'|)$
- binary vector based, $O(n)$
- indexvector based, $O(n)$

3/5: routing strategies at the nodes

How to find the edge to follow in APRIORI?
Given a node with a list of n edges and a part of the filtered transaction (t'), find matching labels.

- simultaneous traversal, $O(n + |t'|)$
- binary search, $O(|t'| \log n)$ or $O(n \log |t'|)$
- binary vector based, $O(n)$
- indexvector based, $O(n)$

Experiment: APRIORI is sensitive to the routing strategy
Winner: indexvector based

Runner up: simultaneous traversal

4/5: storing frequent itemsets

Only frequent itemsets of size ℓ are needed for generating candidates of size $\ell + 1$.
Nodes that are not on a path to any candidate slow down support count method.

- remove from the trie

4/5: storing frequent itemsets

Only frequent itemsets of size ℓ are needed for generating candidates of size $\ell + 1$.
Nodes that are not on a path to any candidate slow down support count method.

- remove from the trie
- store maximum length values for each node

4/5: storing frequent itemsets

Only frequent itemsets of size ℓ are needed for generating candidates of size $\ell + 1$.
Nodes that are not on a path to any candidate slow down support count method.

- remove from the trie
- store maximum length values for each node
- differentiate edges

4/5: storing frequent itemsets

Only frequent itemsets of size ℓ are needed for generating candidates of size $\ell + 1$.
Nodes that are not on a path to any candidate slow down support count method.

- remove from the trie
- store maximum length values for each node
- differentiate edges

Experiments:

- run-time is insensitive
- memory need can be greatly reduced.

5/5: deleting unimportant transactions

A filtered transaction is *unimportant* from the ℓ^{th} iteration, if it does not contain any $(\ell - 1)$-itemset candidate.

Heuristic: Unimportant transactions should be ignored.

Reasoning: They slow down support count (part of the trie is visited).

Experiments: Ignoring unimportant transactions slows down the algorithm.

Argument: It needs resources to determine if a transaction is unimportant or not. In most cases transactions are important (drawback of generate-and-test, breadth-first search method).

Conclusion

In a trie-based FIM algorithms trie-related issues have to be carefully examined.

Thank you for your attention!