On more efficient implementations of ILP systems

Siegfried Nijssen

snijssen@liacs.nl

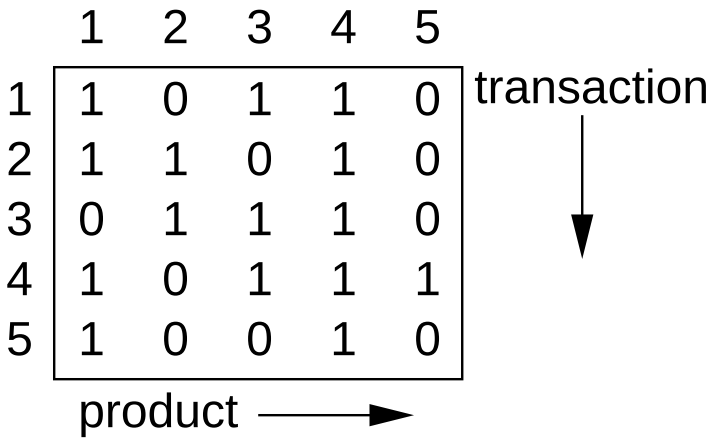
Universiteit Leiden

Leiden Institute of Advanced Computer Science

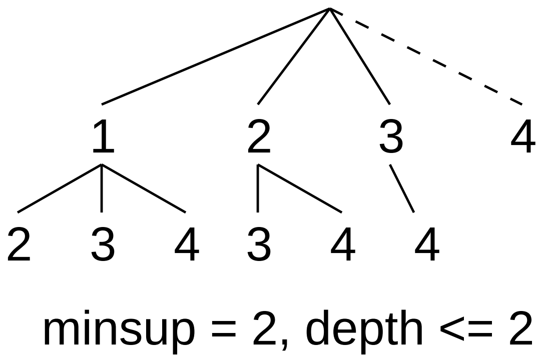
- Apriori (Pijls' implementation)
- Warmr (as I knew it)
- Farmer
  - description of the algorithm
  - results
  - some ideas
  - conclusions

Apriori (Pijls' implementation)

- representation of a database: a binary matrix in core memory



database

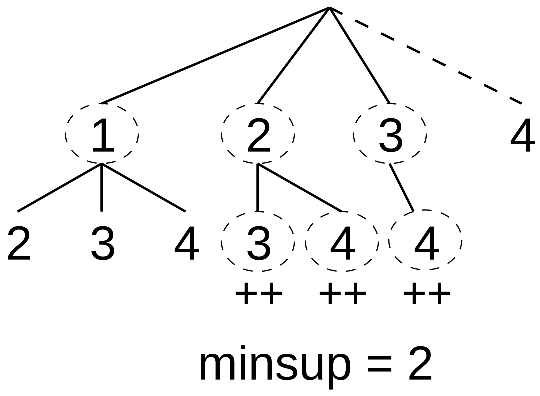


candidates

- representation of a set of item sets: a trie

Apriori (Pijls' implementation)

- counting: walk through tree for every transaction



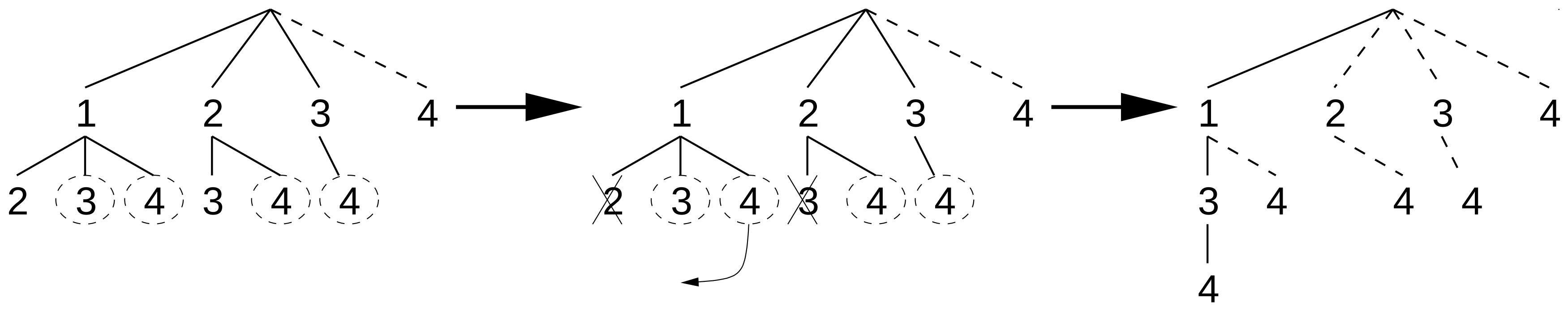
	1	2	3	4	5
1	1	0	1	1	0
2	1	1	0	1	0
3	0	1	1	1	0
4	1	0	1	1	1
5	1	0	0	1	0

transaction

product

Apriori (Pijls' implementation)

- expanding: remove and copy nodes



- by iterating counting and expanding, a tree with all frequent item sets is constructed.

Warmr (as I knew it)

- query $\approx$ item set

- representation of queries: a list of queries

- representation of a "database": a set of facts and background predicates

- counting:
  - ask every query separately to a Prolog engine

Warmr (as I knew it)

- expanding:
  - expand every query according to *mode declarations*
  - for every newly generated query, check using θ-subsumption whether the query:
    - "does not mean the same" as another query
    - is not implied by an infrequent query

Farmer - algorithm

Idea: make an implementation of Warmr based on Pijls' implementation of Apriori $\Rightarrow$ try to emulate Warmr as close as possible

- use a similar tree datastructure as Pijls
  - ▶ counting: speed-up by combining overlapping evaluations
  - ▶ expanding: speed-up by manipulating existing tree

Farmer - algorithm

- get rid of the Prolog implementation
  - no unpredictable behaviour (garbage collection, unknown hash structures, etc.)
  - allow execution on any machine (also my desktop PC)
- later: find something on θ-subsumption

- representation input:
  try to convert EVERYTHING into numbers: predicates, constants, ..., using a strict type system.

WARMR	FARMER
	`type(Client,1000).`
	`type(Product,[chips,cheese]).`
	`predicate(client(Client)).`
	`predicate(buys(Client,Product)).`
	`predicate(has(Client,Product)).`
	`predicate(large(Product)).`
	`predicate(fresh(Product)).`
`key(client(-C)).`	`key(client).`
`mode(1:buys(+C,#)).`	`mode(1,buys(+,chips)).`
	`mode(1,buys(+,cheese)).`
	`mode(1,large(+)).`
	`mode(1,fresh(+)).`
`mode(1:has(+C,-)).`	`mode(1,has(+,-)).`

Results in these tables:

type

number	const
0	1000
1	2

predicate

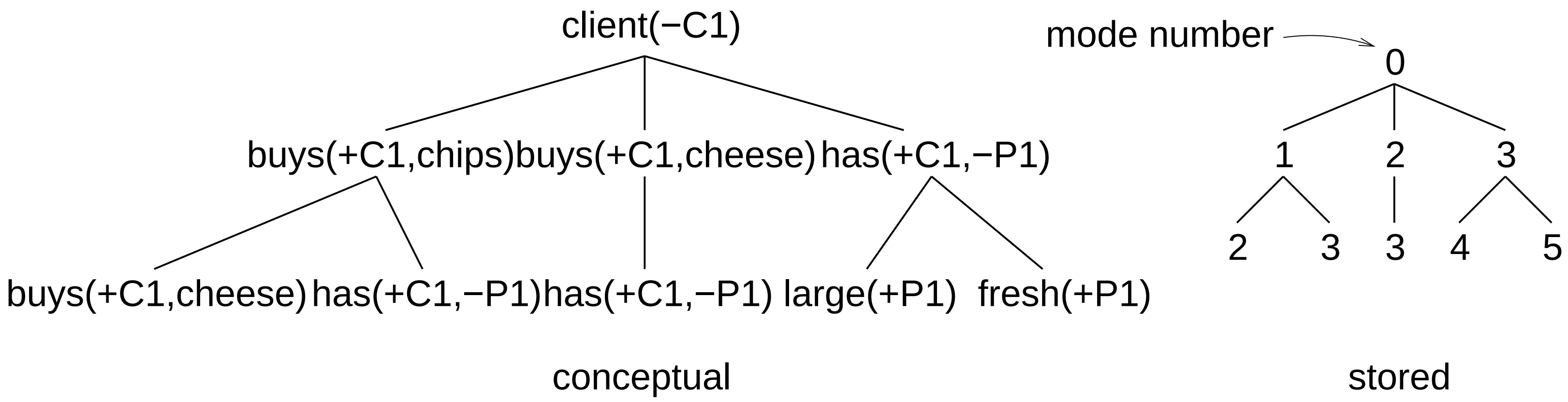
number	parameters
0	0
1	0, 1
2	0, 1
3	1
4	1

mode

number	n	predicate	parameters
0	1	0	$-$
1	1	1	$+$, 0
2	1	1	$+$, 1
3	1	2	$+$, $-$
4	1	3	$+$
5	1	4	$+$

Farmer - algorithm

- representation of queries: a tree (no conjunctions in nodes)



translation examples:

1 $\Rightarrow$ buys(+?,chips) $\Rightarrow$ buys(+C1,chips)

3 $\Rightarrow$ has(+?,-?) $\Rightarrow$ has(+C1,-?) $\Rightarrow$ has(+C1,-P1)

$m(v_1, \ldots, v_n) \Rightarrow v_i$: variable for ambiguous input

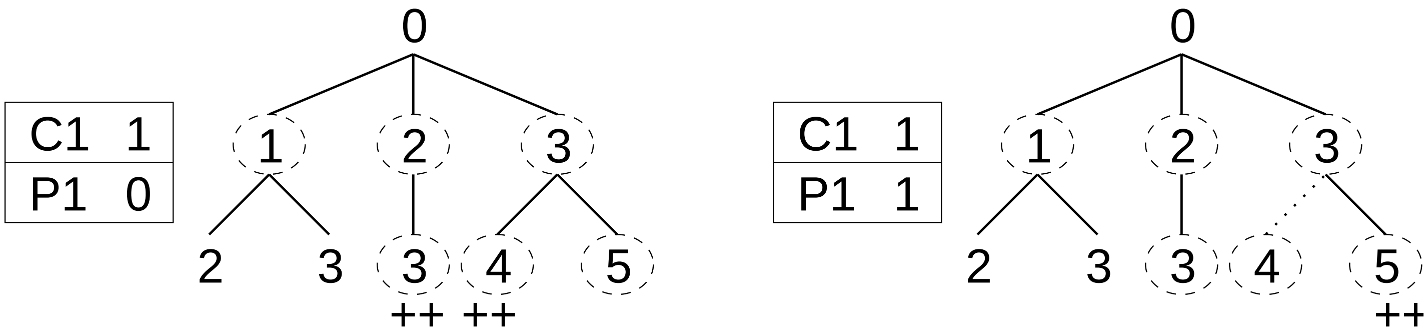
Farmer - algorithm

- representation of a database, current solutions:
  - ▶ no clauses allowed, only facts
  - ▶ allow modes to be evaluated by (C++) functions
  - ▶ build in core-memory tables of facts for other modes
    - ■ only inputs: binary matrix
    - ■ outputs: for every combination of inputs a list of outputs

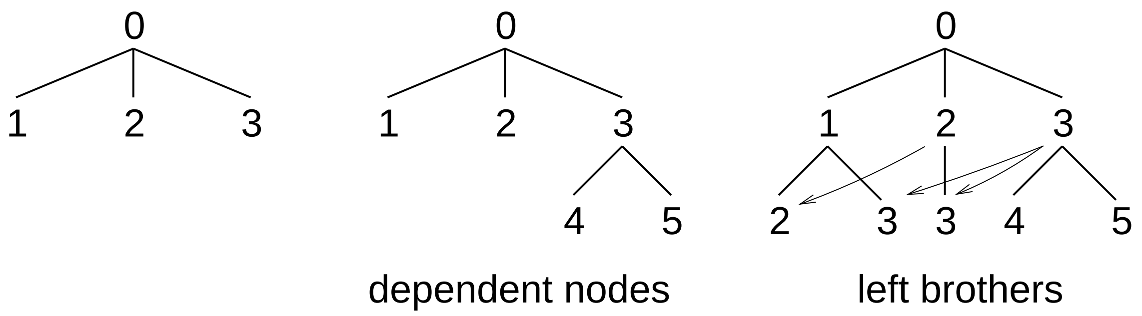
Farmer - algorithm

- counting: walk through tree for all key variable values
  - ▶ maintain table of variable values
  - ▶ similar evaluation pattern as Warmr with query packs

```
buys(1,cheese).  has(1,chips).  has(1,cheese).
large(chips).  fresh(cheese).
```



- expanding: copy & introduce new nodes
  - ▶ start with key as root (no other modes with only outputs)



  - ▶ dependent nodes: atoms which could not have been introduced earlier in a query
  - ▶ frequent left brothers: atoms which could already appear earlier, but are not currently included in a query

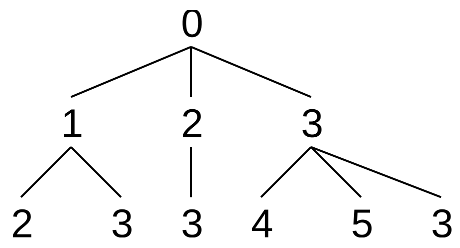
- duplicated nodes. To construct:

  `has(C1,P1),large(P1),has(C1,P2),fresh(P2)`

  should be applicated twice to the same input:

  `mode(2,has(+,-)).`

  Currently chosen solution: copy the parent as a child



  This allows the path `0-3-4-3-5(1)`

- this mechanism generates all queries defined by a bias; what about θ-subsumption?
  - Some equivalent queries could now be generated:
    - `has(C1,P1),large(P1),has(C1,P2),fresh(P2)`
      `has(C1,P1),fresh(P1),has(C1,P2),large(p2)`
    - `has(C1,P1),has(C1,chips)`
      `has(C1,chips)`

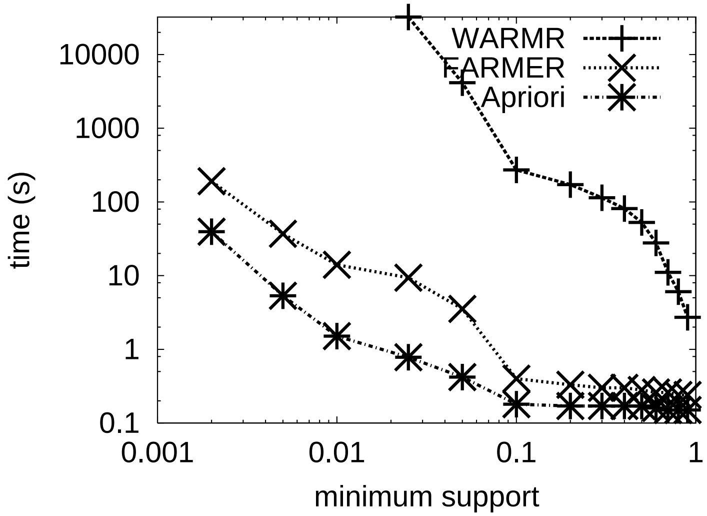
- **proposition**

  If the bias obeys to these two restrictions:

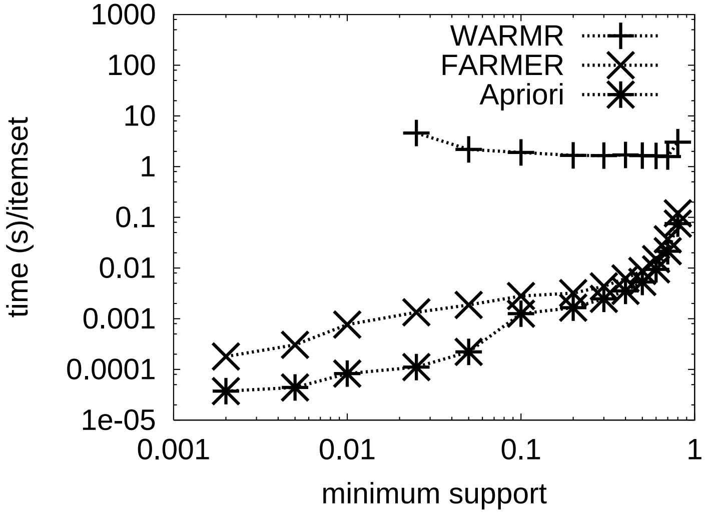
  - contains no two modes for the same predicate
  - does not allow duplication

  then Farmer does not generate two queries that θ-subsume eachother.

- experimental results
  - ▶ Apriori, Farmer, Warmr on frequent item sets

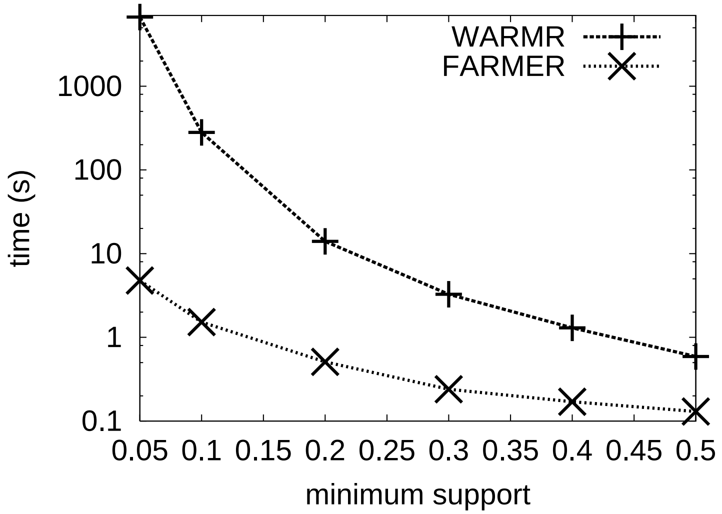


Itemsets Time

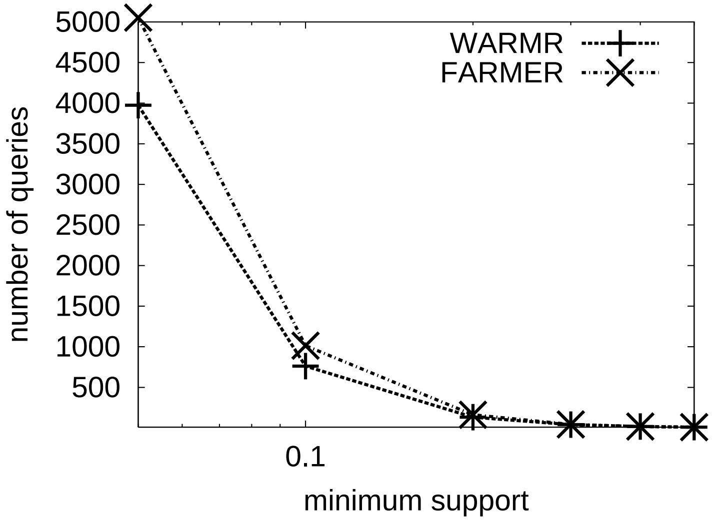


Time/Itemset

- Farmer and Warmr on the Bongard dataset



Queries Time



Queries nr

- Farmer on the multirelational bank dataset (PKDD'99 challenge):
  - several relations, eg. for clients, credit cards, transactions, loans.
  - each relation translated into a predicate

    `relation(+val1,-val2,-val2,...,-valn)`

  - for each type an `is` or a < predicate was added, bound to a C++ function

  rules were generated for 1.000.000 transactions, within a day on a SUN with 4GB core memory.

- attempts to lift the second restriction while still guaranteeing non-redundancy were unsuccessfull until now. One idea:

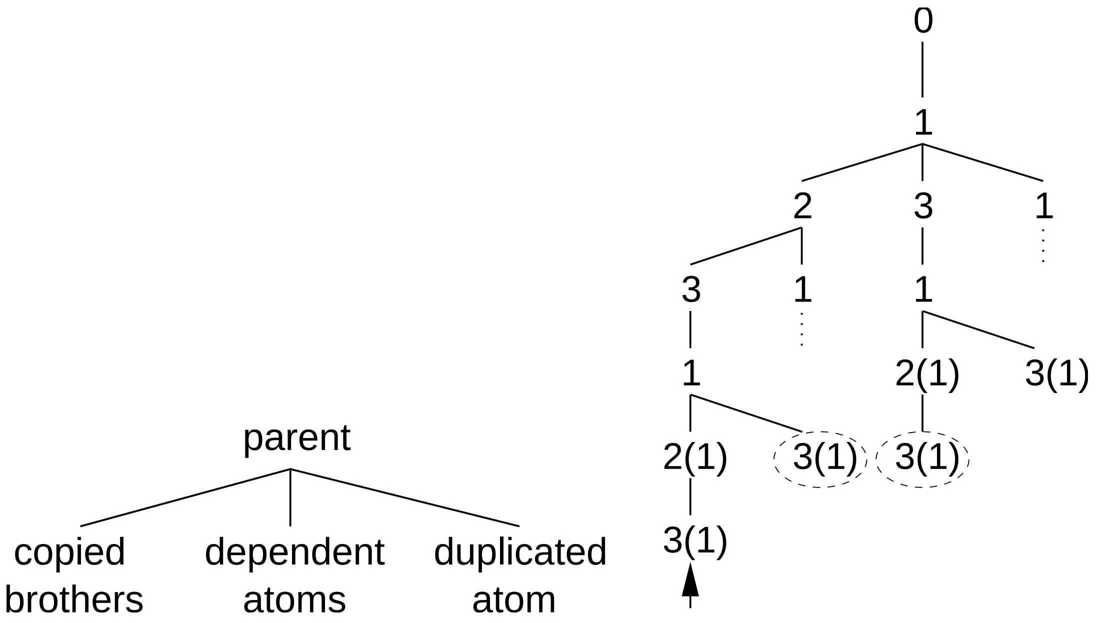
  1. from one query all permutations, as far as relevant for the last atom added, can be generated, eg.:

     `0-3-4-3-5(1)` $\Rightarrow$ `0-3-5-3-4(1)`

  2. if one maintains a fixed order of child generation, it can be shown that all but the leftmost equivalent query can safely be removed immediately

  3. consequently, it should be possible to remove redundant queries immediately after generation, with complexity at most equal to the number of leafs.

- on (2):
  - arrow will not be generated when leftmost equivalent (with circle) is removed



fixed order　　example tree

- conclusions
  + a speedup was obtained by reimplementing Warmr
  + a platform independent implementation was obtained
  + experiment succeeded on large dataset
  - database had to be loaded in core memory
  - too large number of resulting queries

Tilde (as I knew it)

- C4.5 decision tree-like algorithm, eg. using gain (ratio) heuristics and Fayyad and Irani discretization algorithm

- (conjunctions of) atoms in stead of attribute tests

- mode declaration mechanism to introduce new nodes

- lookahead declarations to allow the construction of conjunctions in a node
  - ▶ this is necessary as some atoms alone do not have an immediate positive effect on gain (ratio) heuristics

Gilder

- representation of “database” and tree similar to Farmer
- search strategy also uses modes and gain (ratio) heuristics
- currently: does not use lookahead
  - ▶ if an atom can be satisfied for (almost) all examples, then generate all dependent atoms
  - ▶ several subtrees are built, of which the best is chosen
  - ▶ use an adapted heuristic to chose the best subtree

Gilder

- adapted heuristic: punish large subtrees
  - eg. decrease gain (ratio) by a term dependent on the depth of the subtree
- do not use the Fayyad and Irani algorithm
  - for a continuous variable collect the minimum and maximum value for a continuous value
  - determine the best split using minimum values
  - determine the best split using maximum values
  - add lower-than or higher-than test