OPTIMISING ASSOCIATION RULE ALGORITHMS USING ITEMSET ORDERING

ES2001

Peterhouse College, Cambridge

Frans Coenen, Paul Leng and Graham Goulbourne

The Department of Computer Science

The University of Liverpool

Introduction: The archetypal problem --- shopping basket analysis

- Which items tend to occur together in shopping baskets?
  - – Examine database of purchase transactions
  - – look for *associations*
- Find **Association Rules**:

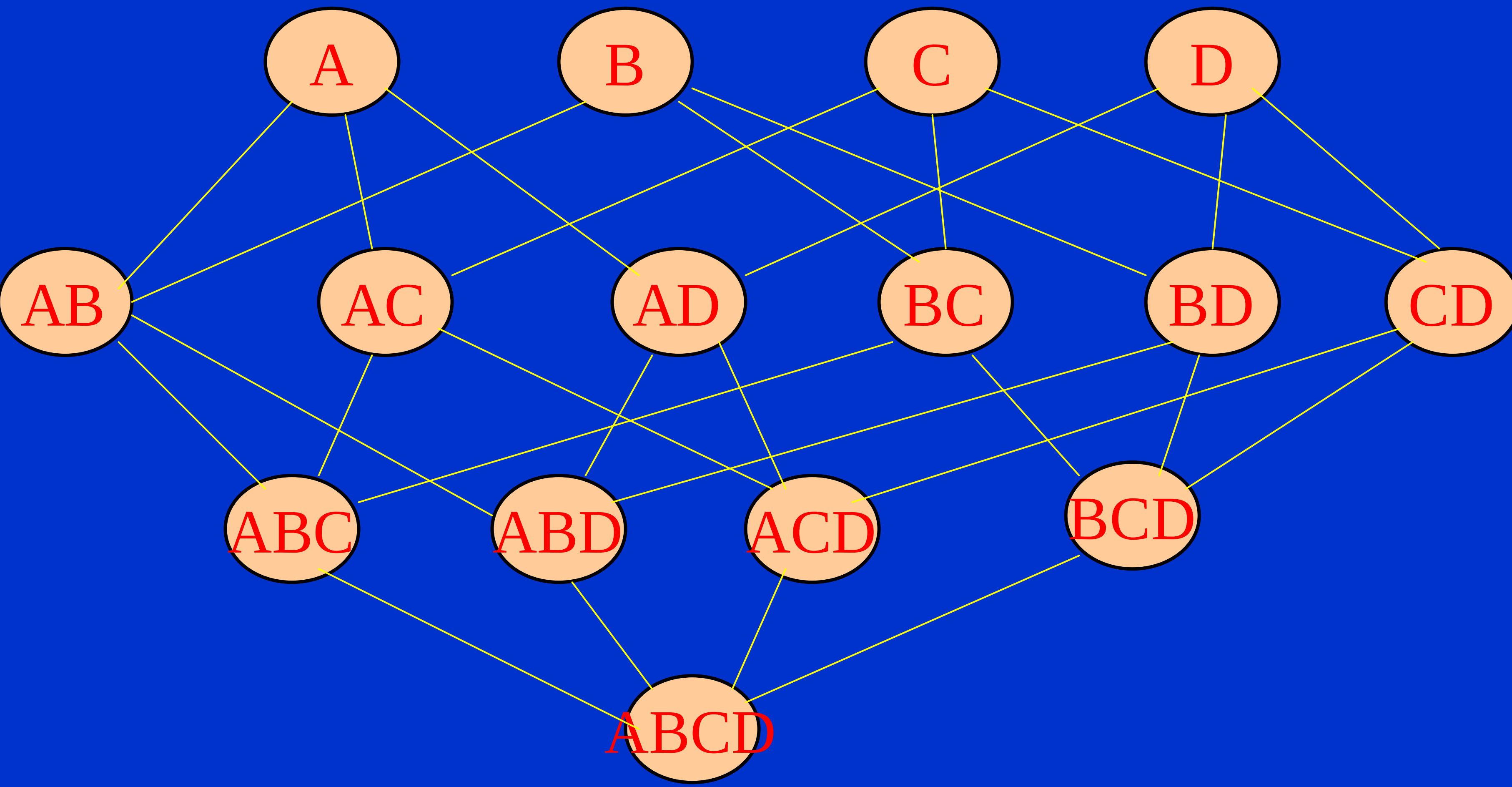
PQ -> X

When P and Q occur together, X is likely to occur also

Support and Confidence

- The *support* for a rule A->B is the number (proportion) of cases in which AB occur together
- The *confidence* for a rule is the ratio of support for rule to support for its antecedent
- **The problem:** Find all rules for which support and confidence exceed some threshold (the *frequent* sets)
- **Support** is the difficult part (confidence follows)

Lattice of attribute-subsets



Apriori Algorithm

- Breadth-first lattice traversal:
  - on each iteration *k*, examine a *Candidate Set* C_k of sets of k attributes:
  - Count the support for all members of C_k (one pass of the database, requiring all k-subsets of each record to be examined)
  - Find the set L_k of sets with required support
  - Use this to determine C_{k+1}, the set of sets of size k+1 all of whose subsets are in L_k

Performance

- Requires x+1 database passes (where x is the size of the largest frequent set)
- Candidate sets can become very large (especially if database is dense)
- Examining k-subsets of a record to identify all members of C_k present is time-consuming
- So: unsatisfactory for databases with densely-packed records

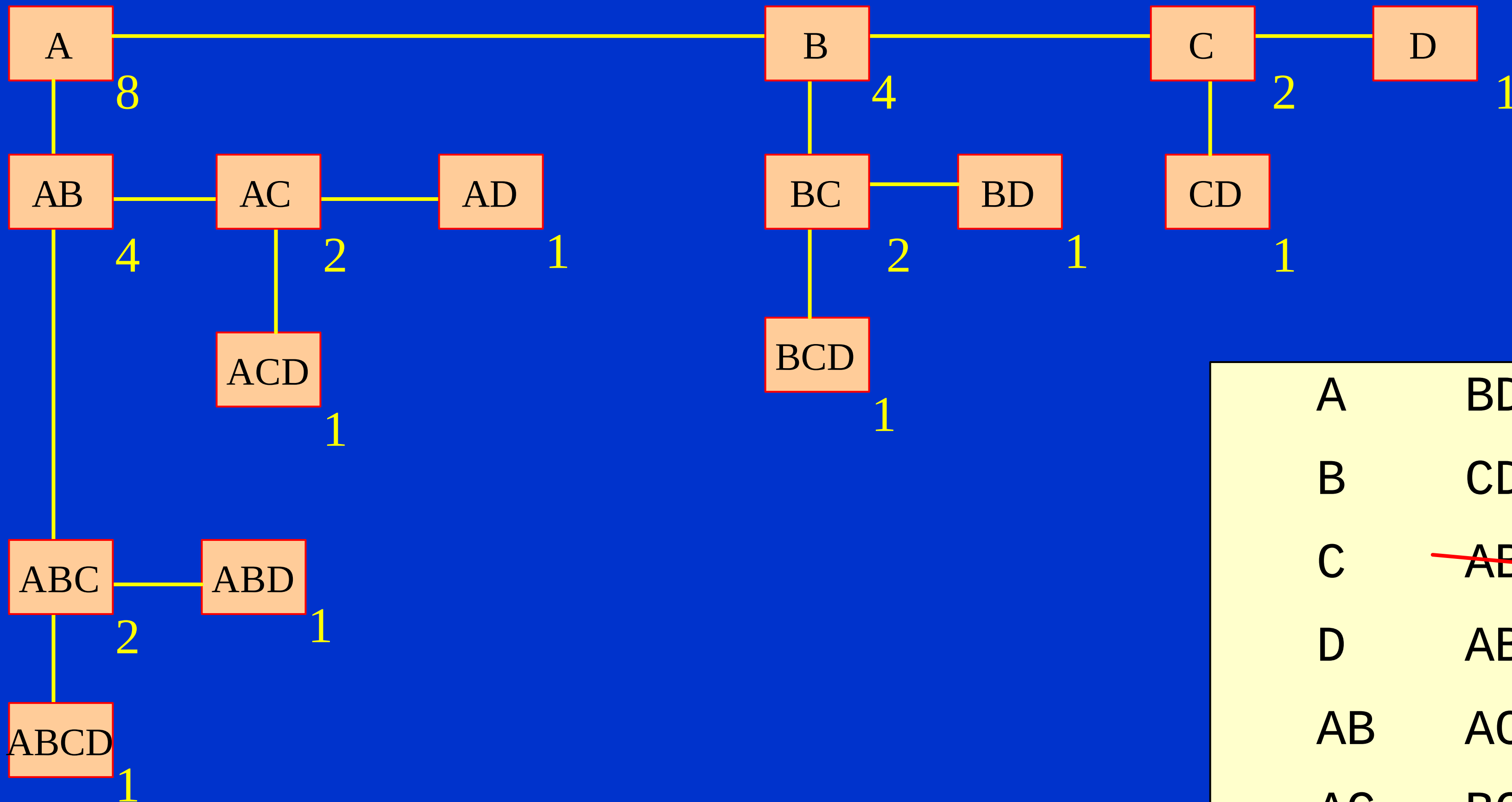
Computing support via Partial support totals

- Use a single database pass to count the sets present (not subsets): this gives us m' *partial* support-counts ($m' \leq m$, the database size)
- Use this set of counts to compute the *total* support for subsets
- Gains when records duplicated ($m' << m$)
- More important: allows us to reorganise data for efficient computation

Building the tree

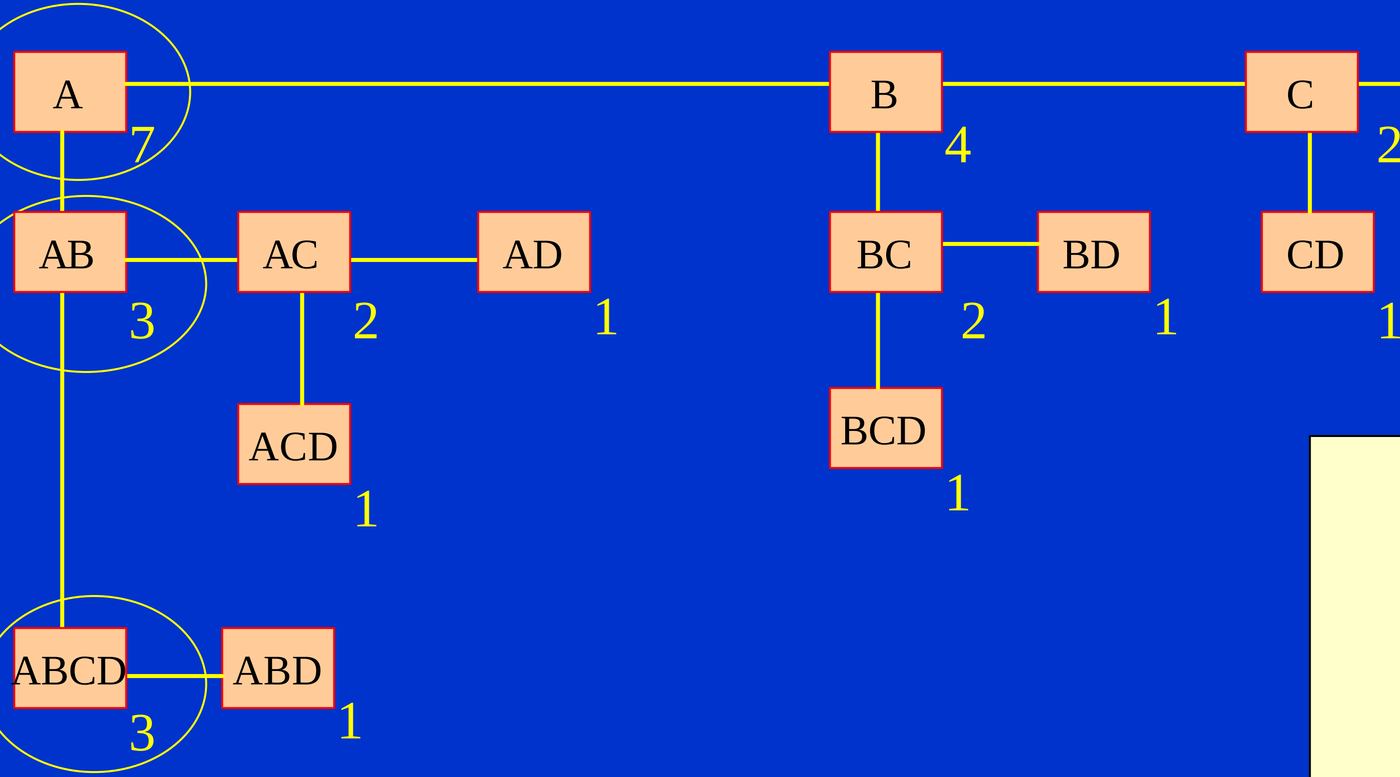
- For each record *i* in database:
  - find the set *i* on the tree;
  - increment support-count for all sets on path to *i*
  - if set not present on tree, create a node for it
- Tree is built dynamically (size ~m rather than 2^n)
- Building tree has already counted support deriving from successor-supersets (leading to *interim* support-count Q_i)

Set enumeration tree: The P-tree



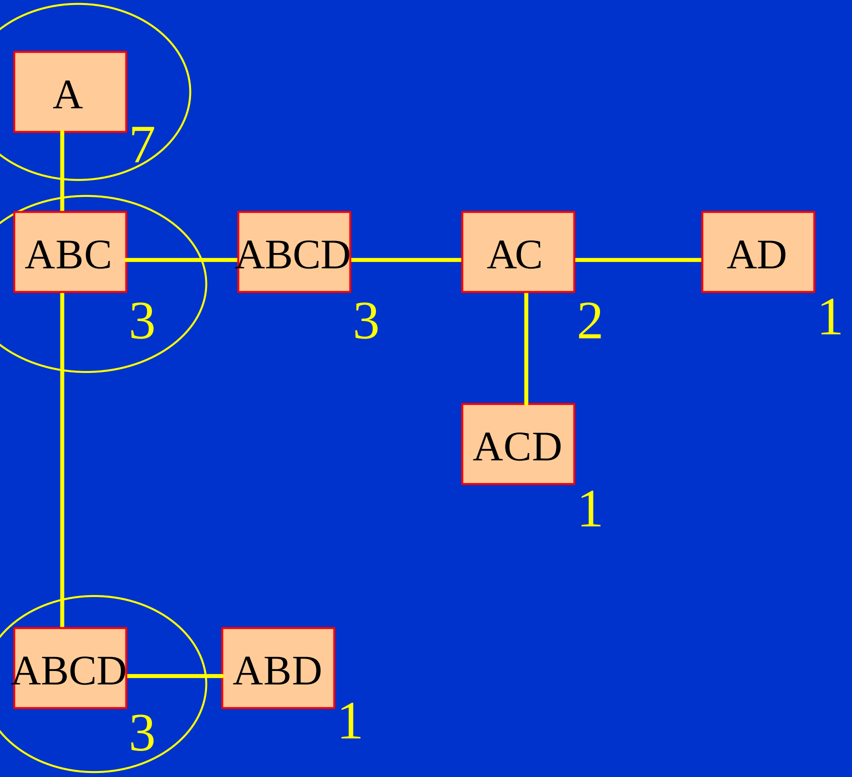
A	BD
B	CD
C	~~ABC~~
D	ABD
AB	ACD
AC	BCD
AD	ABCD
BC	

Set enumeration tree: The P-tree



A	BD
B	CD
C	~~ABC~~
D	ABD
AB	ACD
AC	BCD
AD	ABCD
BC	

Dummy Nodes



A

AC

~~AD~~

ABC

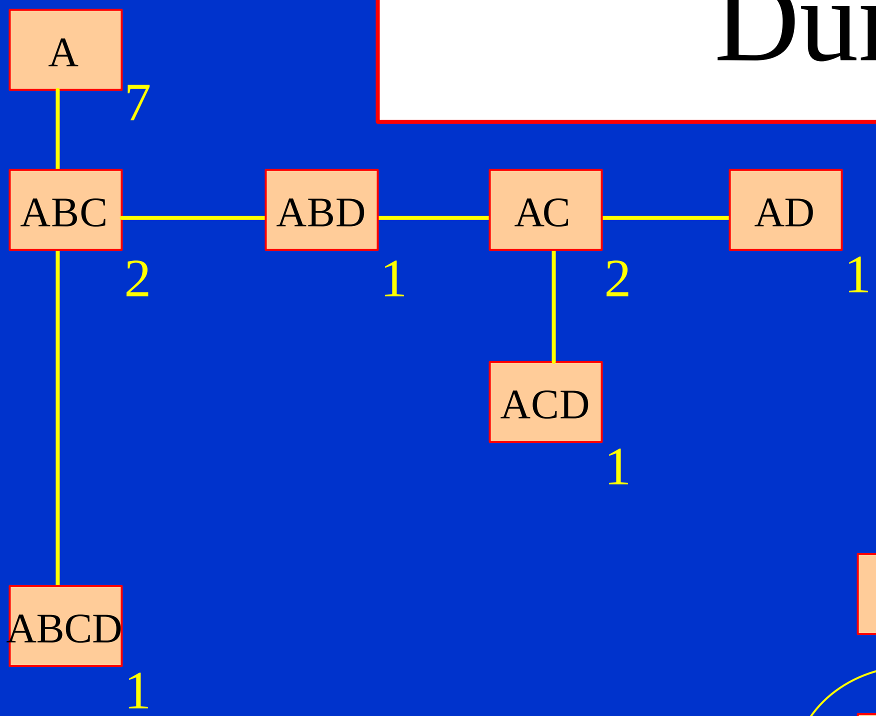
ABD

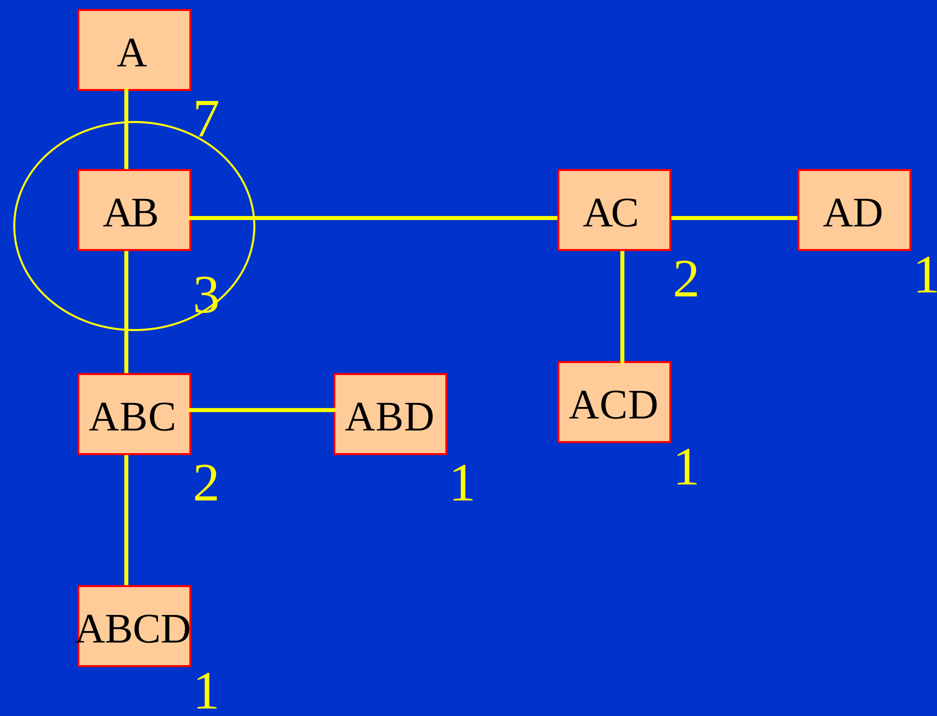
ACD

ABCD

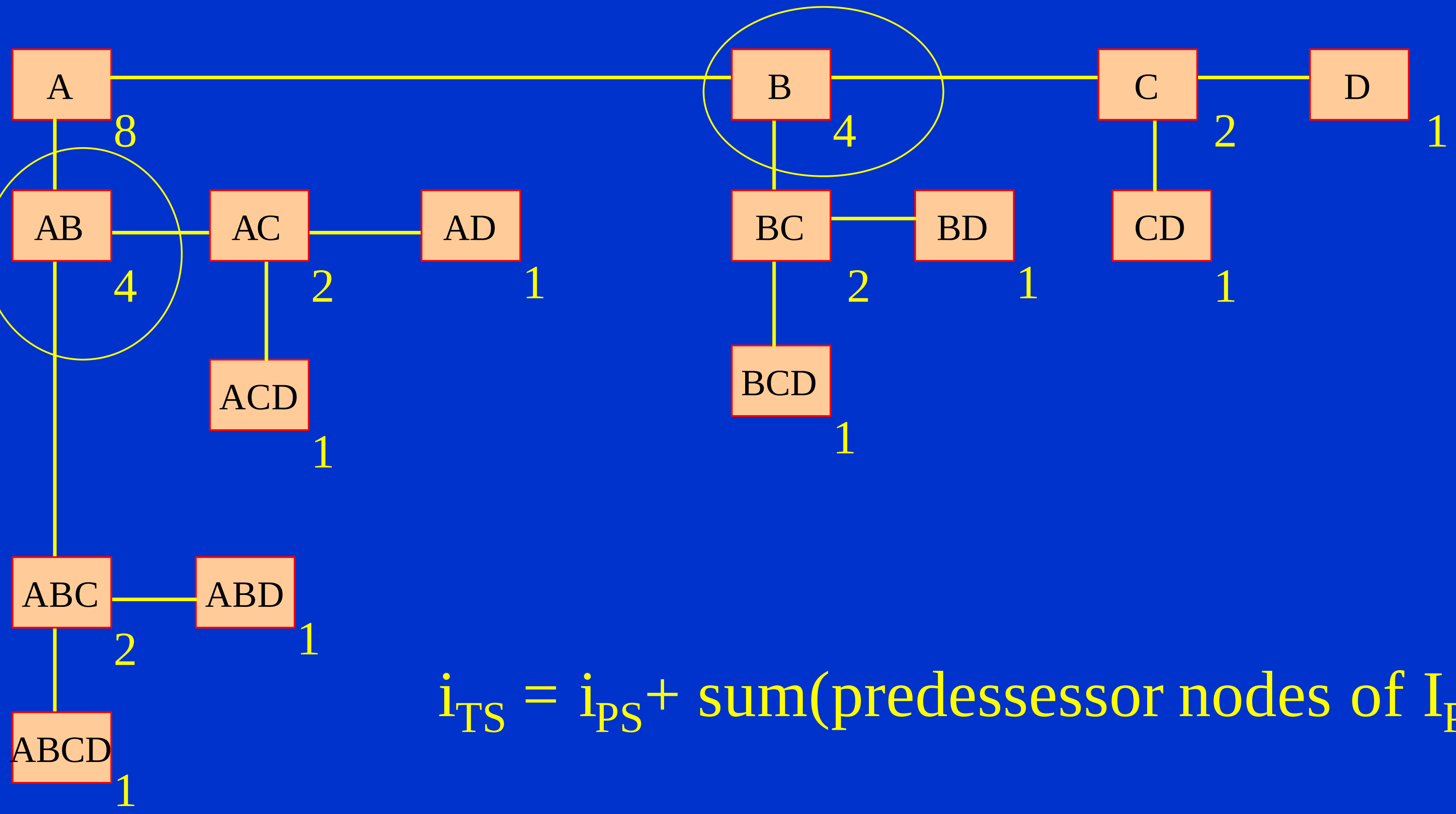
Dummy Node

A
AC
AD
ABC
ABD
ACD
ABCD





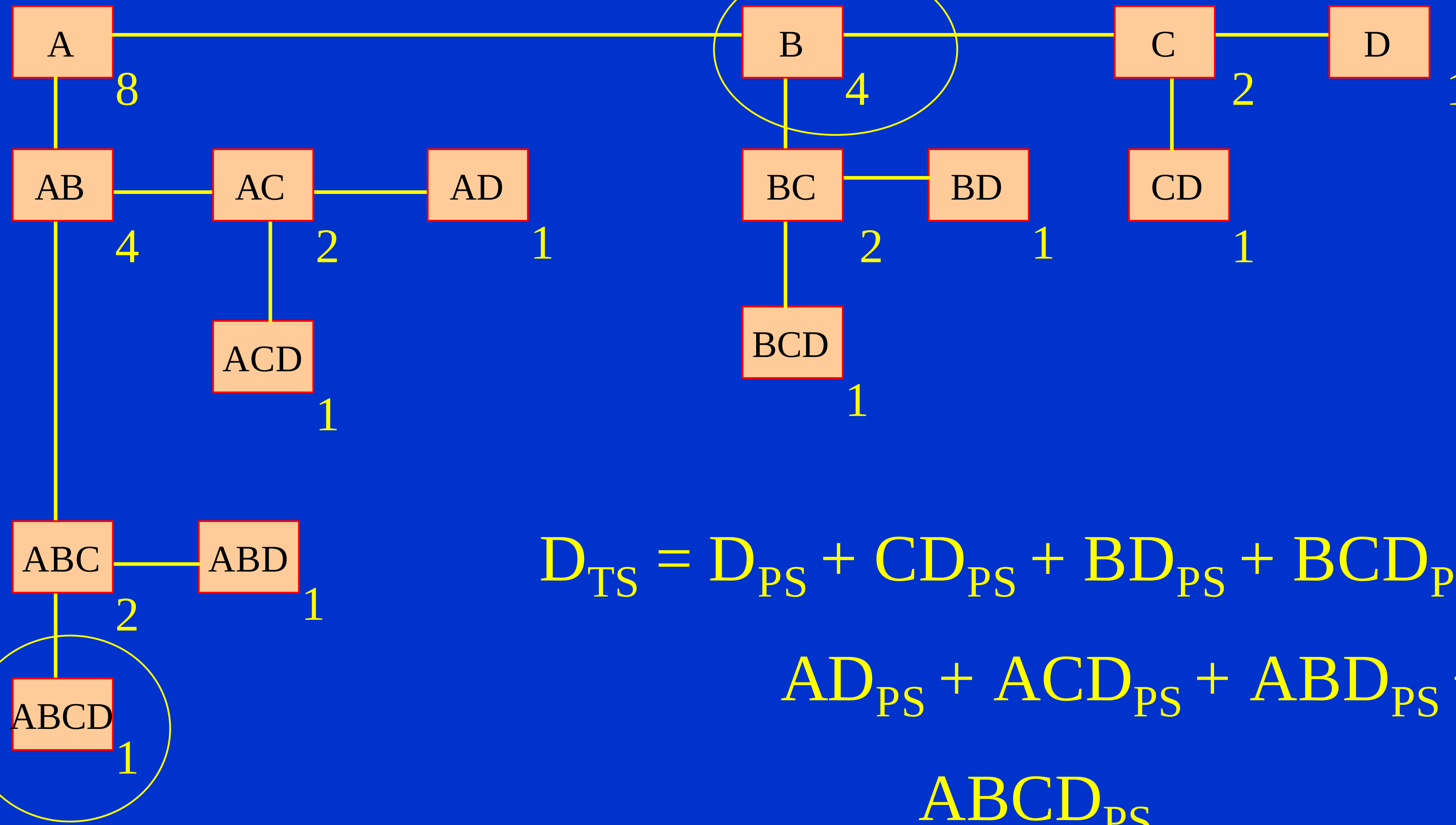
Calculating total support



$i_{TS} = i_{PS} +$ sum(predessessor nodes of I_{PS})

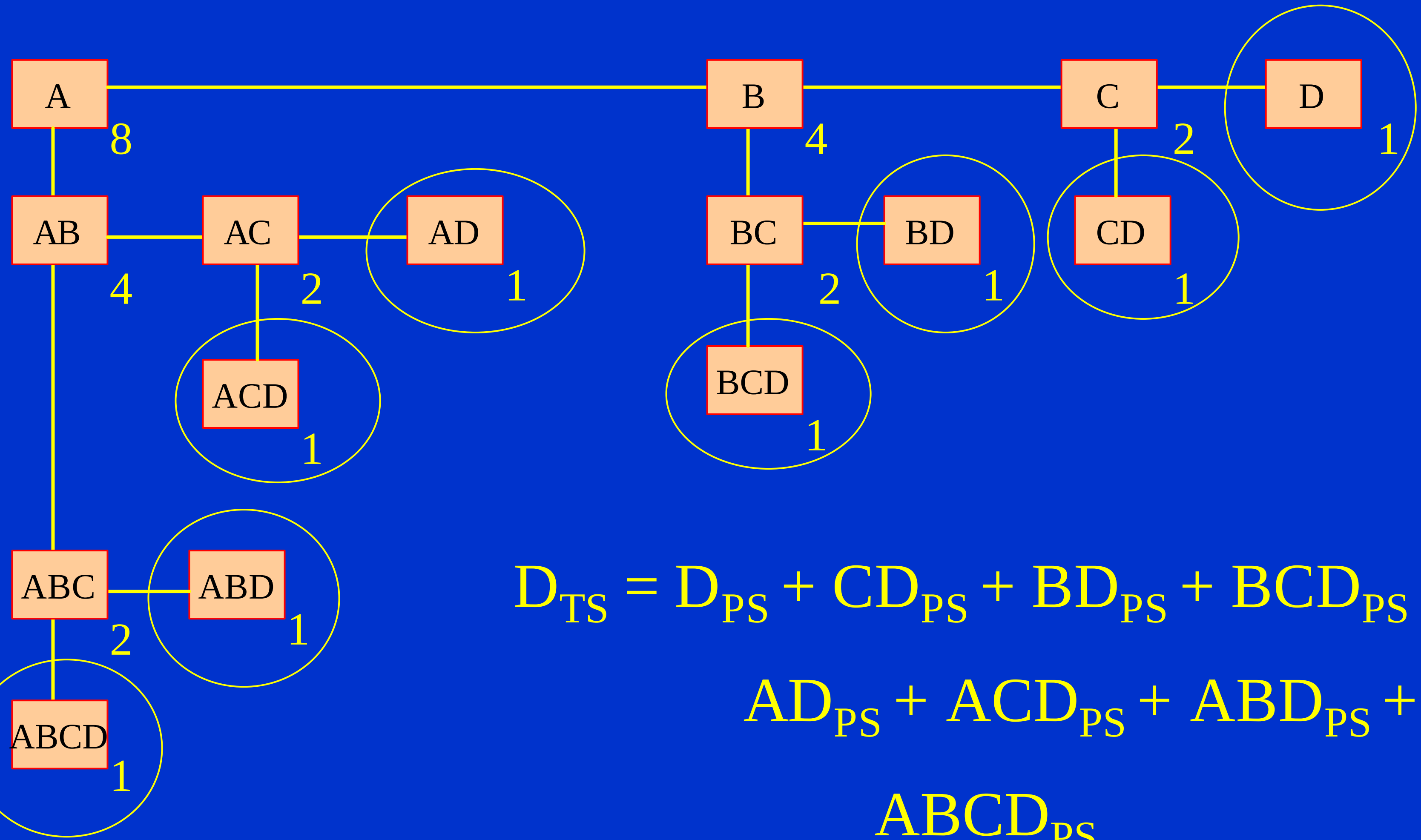
$B_{TS} = B_{PS} + AB_{PS}$

Calculating total support



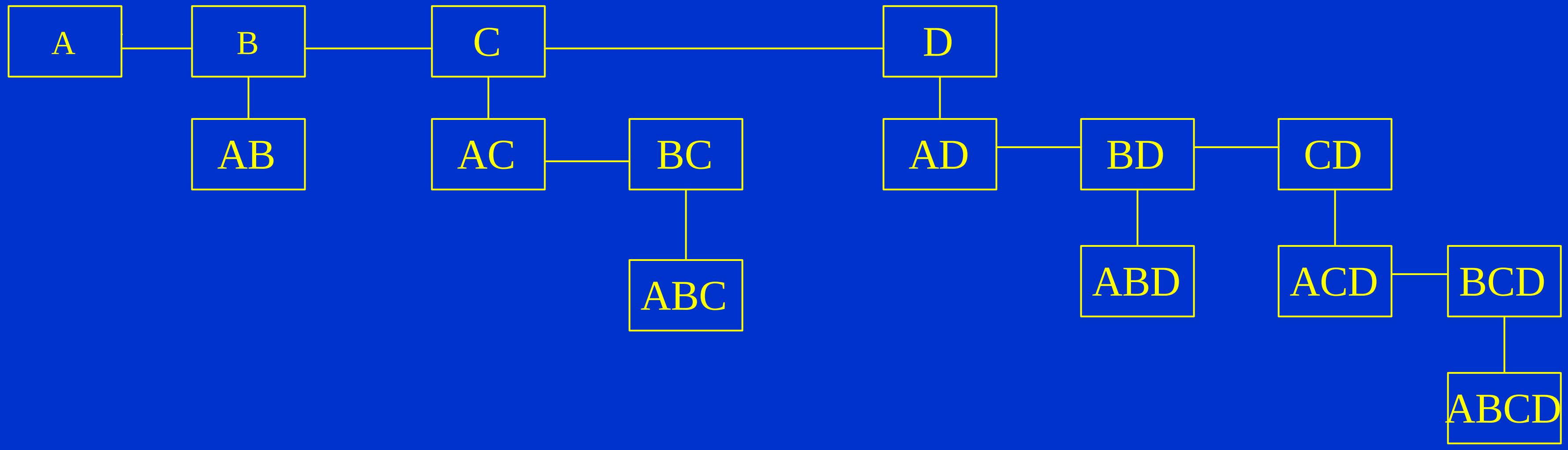
$$D_{TS} = D_{PS} + CD_{PS} + BD_{PS} + BCD_{PS} + AD_{PS} + ACD_{PS} + ABD_{PS} + ABCD_{PS}$$

Calculating total support



$$D_{TS} = D_{PS} + CD_{PS} + BD_{PS} + BCD_{PS} + AD_{PS} + ACD_{PS} + ABD_{PS} + ABCD_{PS}$$

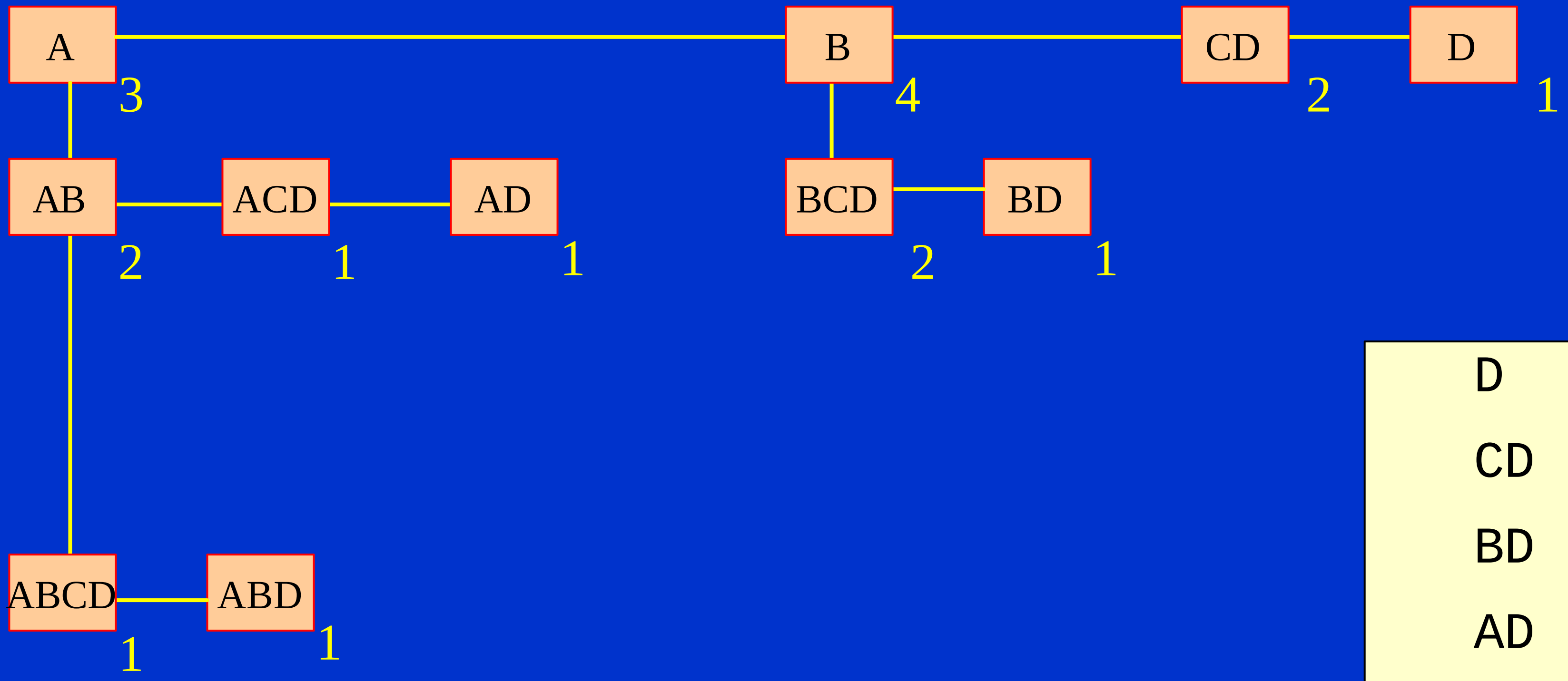
Computing total supports: The T-tree



Itemset Ordering

- Advantages gained from partial computation is not equally distributed throughout the set of candidates.
- For candidate early in the lexicographic order most of the support calculation is complete
- If we know the frequency of single items sets we can order the tree so that the most common item sets appear first and thus reduced the effort required for total support counting.

Set enumeration tree: The P-tree



D

CD

BD

AD

BCD

ACD

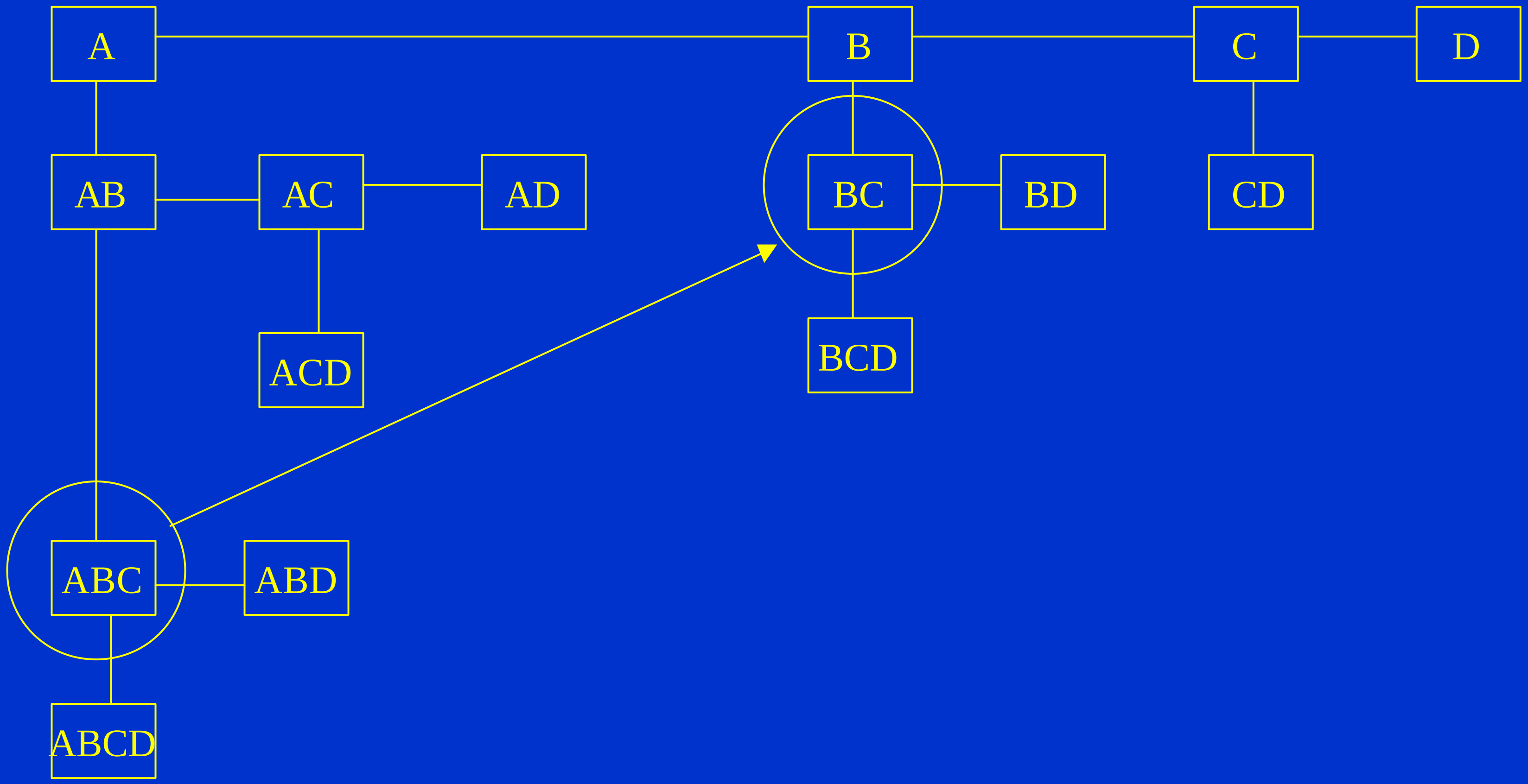
ABD

ABCD

Computing Total Supports

- Have already computed *interim* support Q_i for set *i*
- *Total* support = $Q_i + \sum P_j$ (adding support for predecessor-supersets)

Example

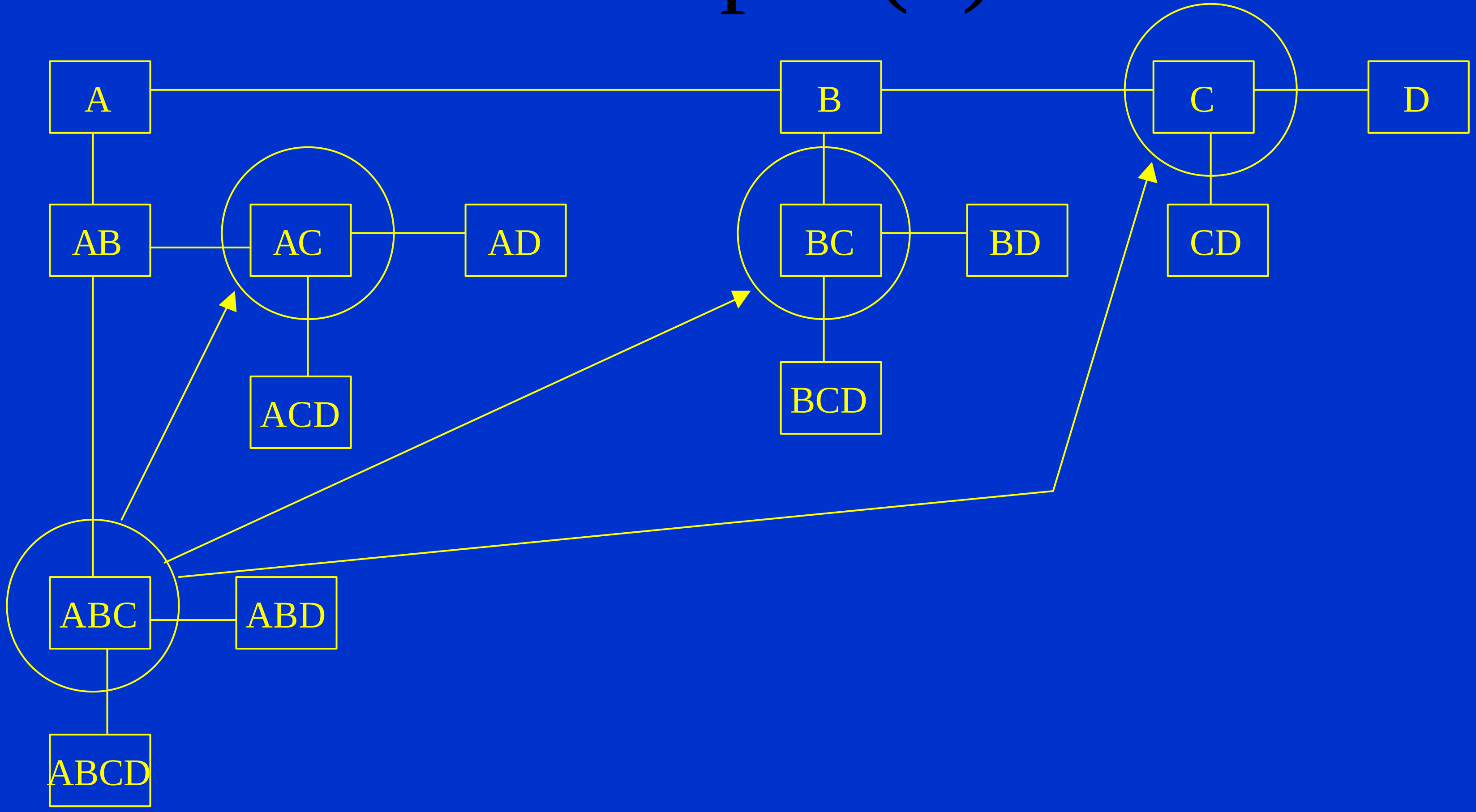


- To complete total for BC, need to add support stored at ABC

General summation algorithm

- For each node j in tree:
  - for all sets i in *Target set* T:
    - if i is a subset of j and i is not a subset of the parent of j, add Q_j to total for i

Example (2)



- Add support stored at ABC to support for AC, BC and C
- No need to add to A, AB (already counted) or to B (will have AB added, including ABC)

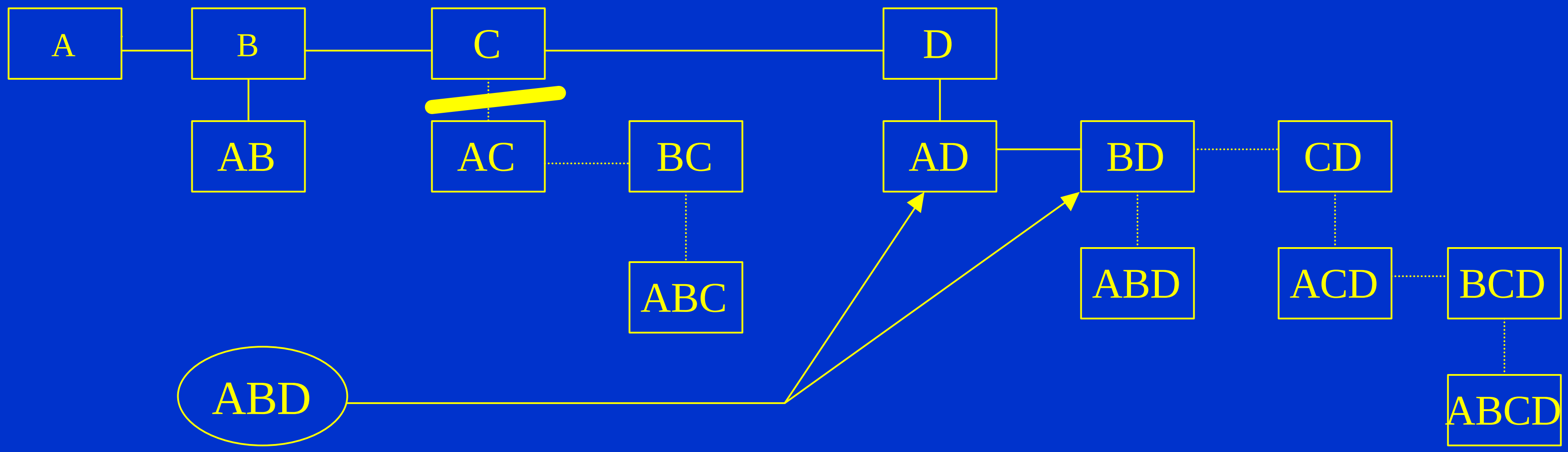
Modified algorithm

- Problem: still have 2^n Totals to count
  - So use Apriori type algorithm
- Count C_1, C_2 etc in repeated passes of tree

Algorithm *Apriori-TFP* (Total-from -Partial)

- For each node j in P-tree:
  - i is attribute not in parent node
  - starting at node i of T-tree:
    - walk the tree until (parent of) node j is reached, adding support to all subsets of j at the required level
- On completion, *prune* the tree to remove unsupported sets
- Generate the next level and repeat

Illustration



Pass 1: C not supported, so do not add AC,BC,CD to tree
Pass2: (eg) Item ABD from P-tree added to AD and BD (tree is walked from D to BD)

Advantages

- 1. Duplication in records reduces size of tree
- 2. Fewer subsets to be counted: eg, for a record of r attributes, Apriori counts r(r-1)/2 subset-pairs; our method only r-1
- 3. T-tree provides an efficient localisation of candidates to be updated in Apriori-TFP

Related Work

- The FP-tree (Han et al.), developed contemporaneously, has similar properties, but:
  - FP-tree stores a single item only at each node (so more nodes)
  - FP-tree builds in more links to implement *FP-growth* algorithm
  - Conversely, P-tree is generic: *Apriori-TFP* is only one possible algorithm

Experimental results (1)

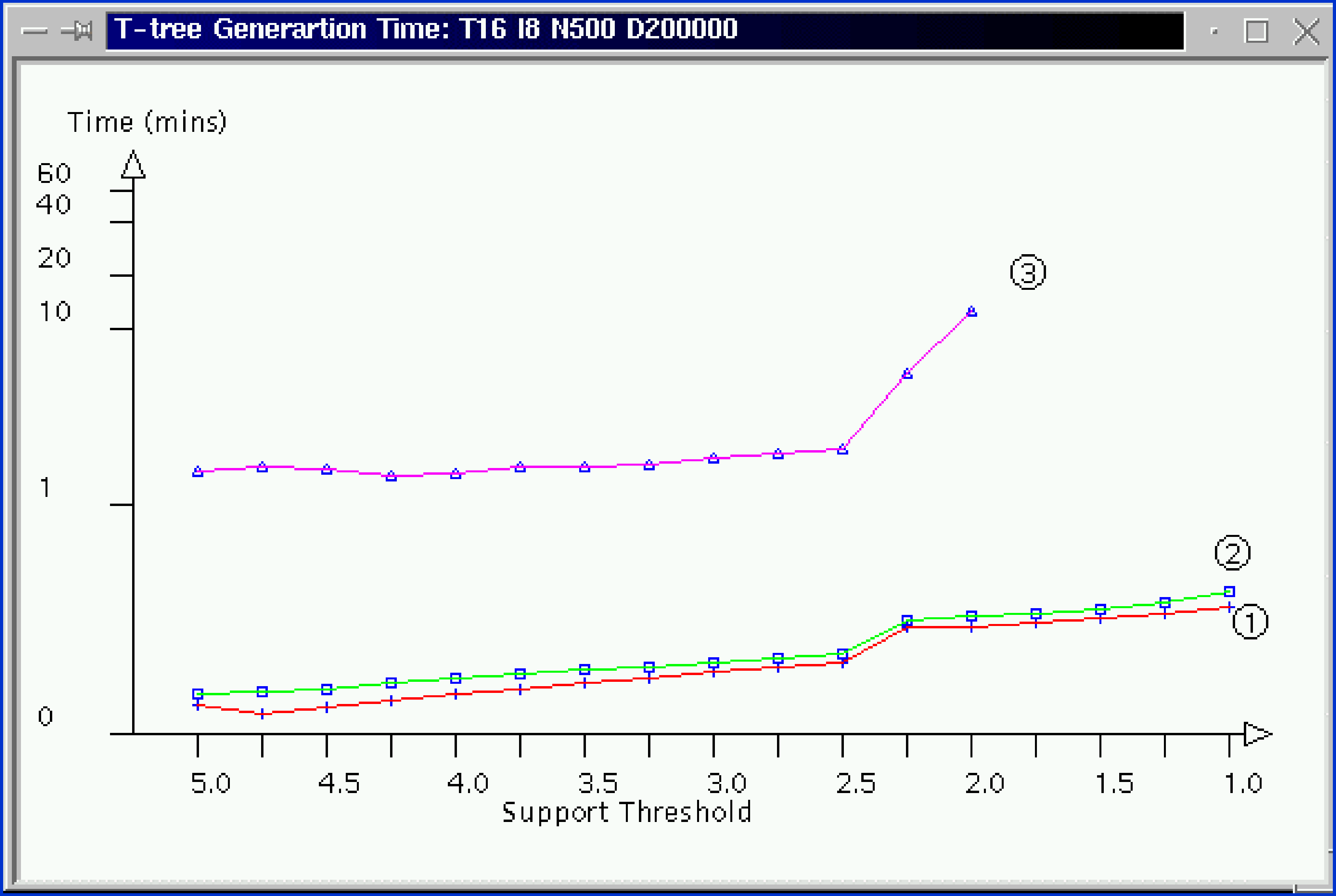
- Size and construction time for P-tree:
  - almost independent of N (number of attributes)
  - scale linearly with M (number of records)
  - seems to scale linearly as database density increases
  - less than for FP-tree (because of more nodes and links in latter)

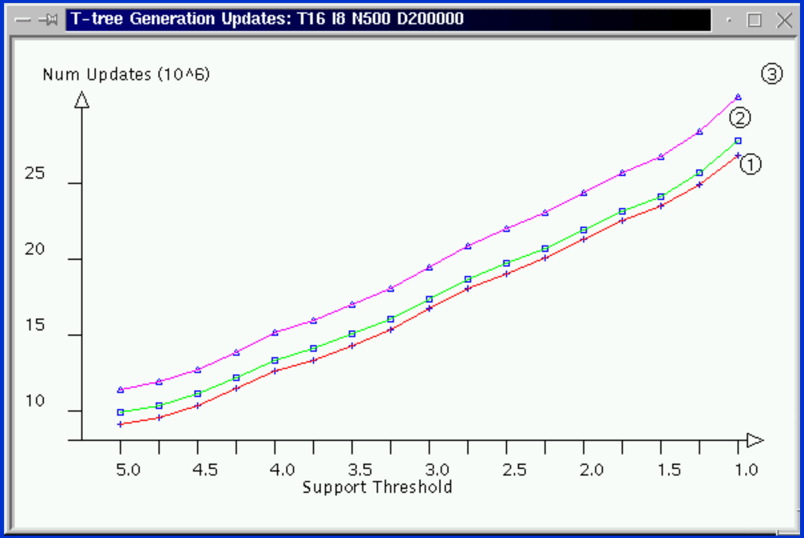
Experimental results (2): time to produce all frequent sets

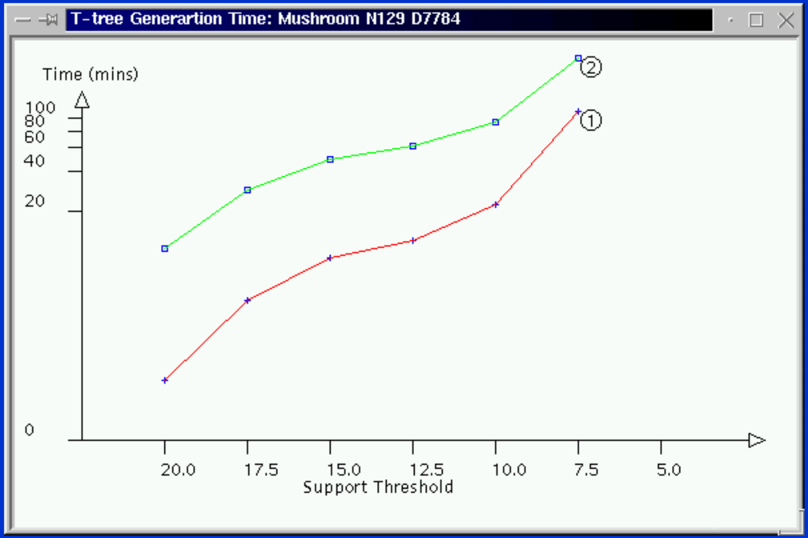
T25.I10.N1K.D10K

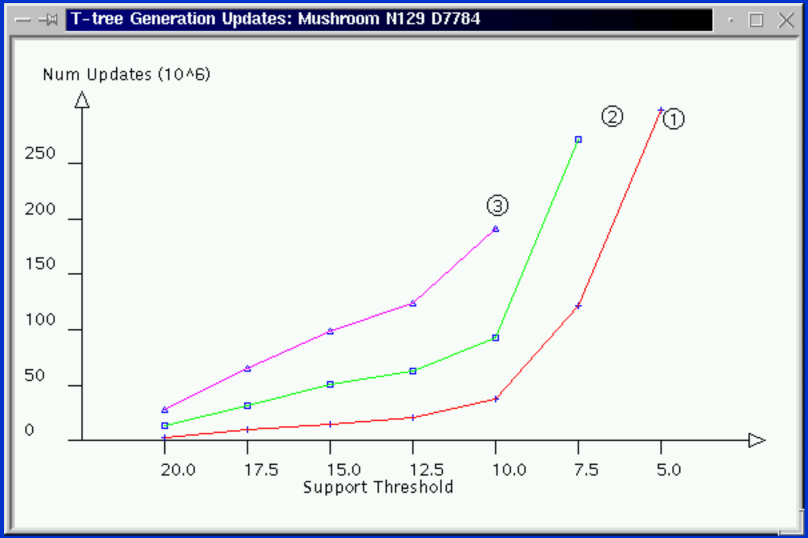
Continuing work

- Optimise using item ordering heuristic: (as used in FP-growth)
- Explore other algorithms (eg *Partition)* applied to P-tree
- Hybrid methods, using different algorithms for subtrees
  - (exhaustive methods may be effective for small very densely-populated subtrees)

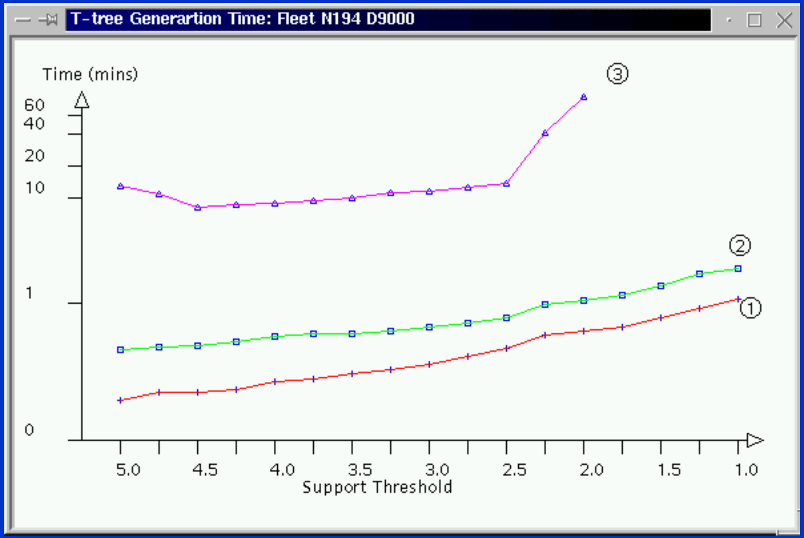

T-tree Generartion Time: T16 I8 N500 D200000
Time (mins)
60
40
20
10
1
0
③
②
①
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Support Threshold


T-tree Generation Updates: T16 I8 N500 D200000
Num Updates (10^6)
25
20
15
10
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Support Threshold
3
2
1

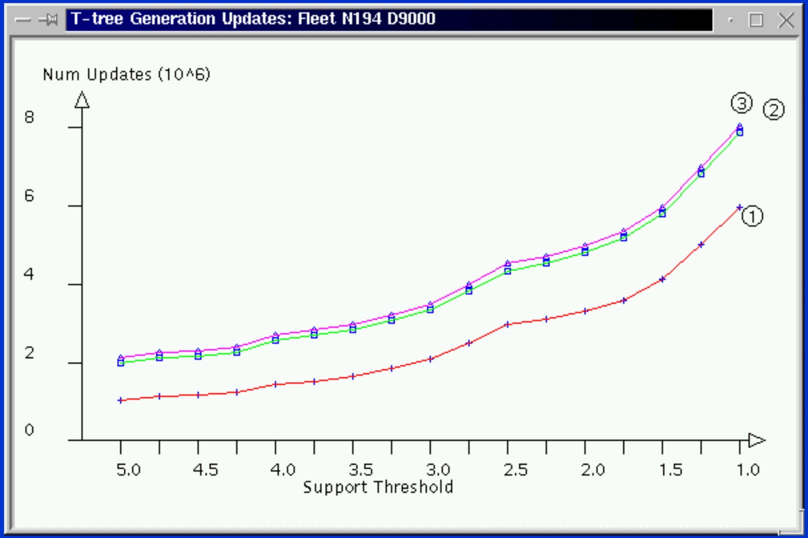

T-tree Generartion Time: Mushroom N129 D7784
Time (mins)
100
80
60
40
20
0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
Support Threshold
1
2



T-tree Generation Updates: Mushroom N129 D7784
Num Updates (10^6)
250
200
150
100
50
0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
Support Threshold
1
2
3



T-tree Generartion Time: Fleet N194 D9000
Time (mins)
60
40
20
10
1
0
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Support Threshold
3
2
1


T-tree Generation Updates: Fleet N194 D9000
Num Updates (10^6)
8
6
4
2
0
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Support Threshold
3
2
1