Discussion Section Week 9

CS 145

Reminder

- HW 4
  - Due on 12/01 (Fri) 11:59 PM (Today!)

- Final Project Report
  - Due on 12/10 (Sun) 11:59 PM
  - Brief submission guidelines are already on CCLE
  - More detailed guidelines will be announced by this week
    - E.g., script format and essential contents

Overview

- Frequent Pattern Mining
  - Apriori Algorithm
  - FP Growth Algorithm
  - Pattern Evaluation Methods
- Sequential Pattern Mining
  - GSP
  - SPADE
  - PrefixSpan
- Dynamic Time Warping

Apriori Algorithm

Apriori Algorithm

- The Apriori Algorithm is an influential algorithm for mining frequent itemsets for boolean association rules.
- Given a set of transactions containing items, find frequent itemsets which occur together in a transaction.
- Key Concepts:
  - Frequent Itemsets: The sets of item which has minimum support (denoted by L_i for i^{th}-Itemset).
  - Apriori Property: Any subset of frequent itemset must be frequent.
  - Join Operation: To find L_k , a set of candidate k-itemsets is generated by joining L_{k-1} with itself.

Apriori Algorithm

- Find the frequent itemsets: the sets of items that have minimum support
  - A subset of a frequent itemset must also be a frequent itemset, i.e. if {AB} is a frequent itemset, both {A} and {B} should be frequent itemsets
  - Iteratively find frequent itemsets with cardinality from 1 to k (k-itemset)

- Use the frequent itemsets to generate association rules

Apriori Algorithm: Pseudo code

- Join Step: C_k is generated by joining L_{k-1} with itself
- Prune Step: Any (k-1)-itemset that is not frequent cannot be a subset of a frequent k-itemset
- Pseudo-code

C_k: Candidate itemsets of size k

L_k : frequent itemsets of size k

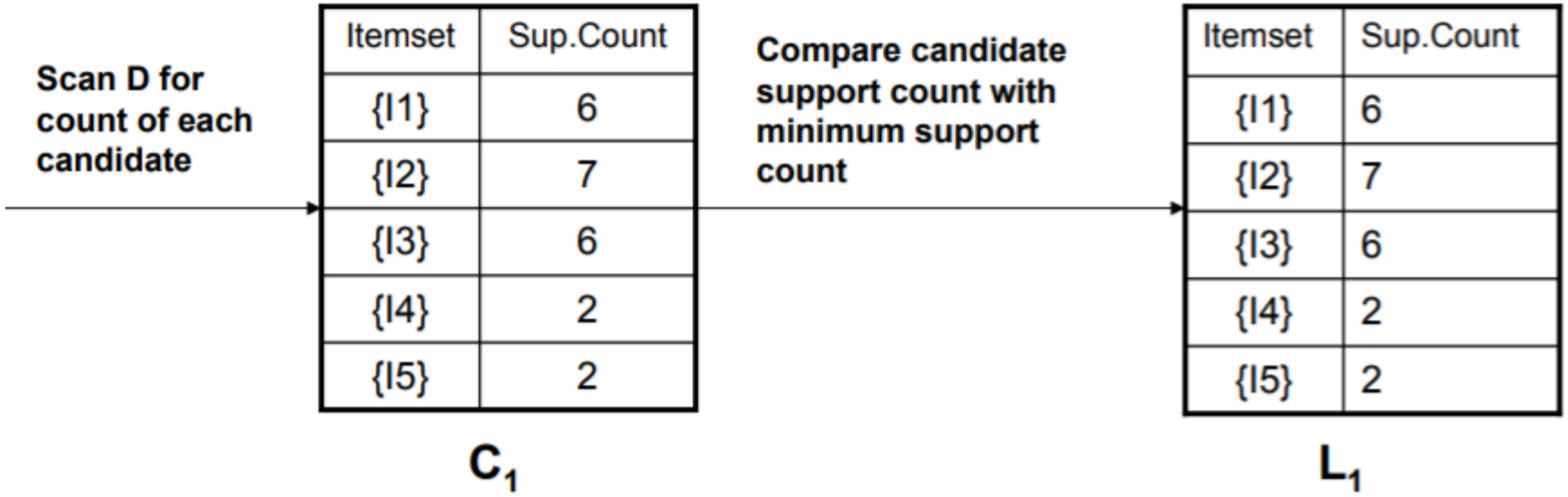
L_1 = {frequent items};
for (k = 2; L_{k-1} !=∅; k++) **do begin**
  C_k = candidates generated from L_{k-1};
  for each transaction t in database do
    increment the count of all candidates in C_k that are contained in t
  L_k = candidates in C_k with min_support
end
return $\cup_k L_k$;

Apriori Algorithm: Example

- Consider a database D, consisting of 9 transactions
- Suppose min. support count required is 2
- We have to find out the frequent itemset using Apriori algorithm.

TID	List of Items
T100	I1, I2, I5
T100	I2, I4
T100	I2, I3
T100	I1, I2, I4
T100	I1, I3
T100	I2, I3
T100	I1, I3
T100	I1, I2 ,I3, I5
T100	I1, I2, I3

Generating 1-itemset Frequent Pattern

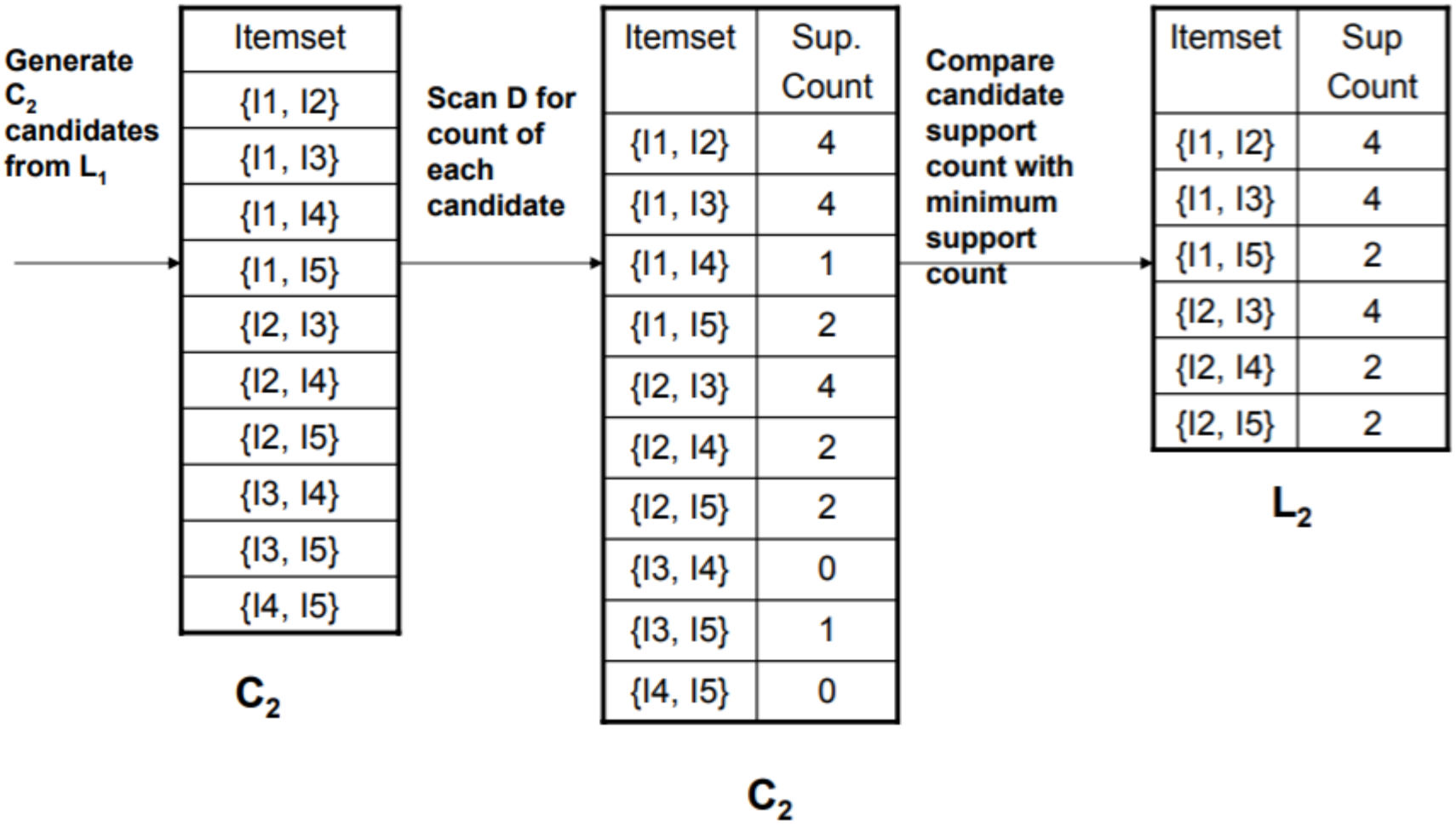


- The set of frequent 1-itemsets, L_1 , consists of the candidate 1-itemsets satisfying minimum support.
- In the first iteration of the algorithm, each item is a member of the set of candidate

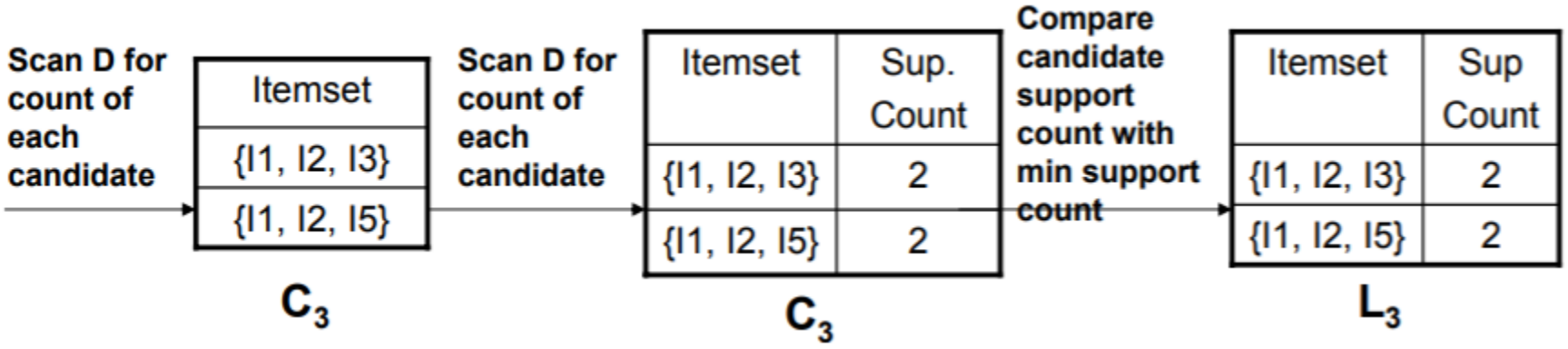
Generating 2-itemset Frequent Pattern

- To discover the set of frequent 2-itemsets, L_2 , the algorithm uses L_1 Join L_1 to generate a candidate set of 2-itemsets, C_2
- Next, the transactions in D are scanned and the support count for each candidate itemset in C_2 is accumulated
- The set of frequent 2-itemsets, L_2 , is then determined, consisting of those candidate 2-itemsets in C_2 having minimum support.
- Note: We haven’t used Apriori Property yet.

Generating 2-itemset Frequent Pattern



Generating 3-itemset Frequent Pattern



- The generation of the set of candidate 3-itemsets, C_3 , involves use of the Apriori Property
- In order to find C_3, we compute L_2 Join L_2
- $C_3 = L_2$ Join L_2 = {{I1, I2, I3}, {I1, I2, I5}, {I1, I3, I5}, {I2, I3, I4}, {I2, I3, I5}, {I2, I4, I5}}
- Now, Join step is complete and Prune step will be used to reduce the size of C_3. Prune step helps to avoid heavy computation due to large C_k

Generating 3-itemset Frequent Pattern

- Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we can determine that last four candidates cannot possibly be frequent. How?
- For example, lets take {I1, I2, I3}. The 2-item subsets of it are {I1, I2}, {I1, I3} & {I2, I3}. Since all 2-item subsets of {I1, I2, I3} are members of L_2, We will keep {I1, I2, I3} in C_3.
- Lets take another example of {I2, I3, I5} which shows how the pruning is performed. The 2-item subsets are {I2, I3}, {I2, I5} & {I3,I5}.
- BUT, {I3, I5} is not a member of L_2 and hence it is not frequent violating Apriori Property. Thus We will have to remove {I2, I3, I5} from C_3.
- Therefore, C_3 = {{I1, I2, I3}, {I1, I2, I5}} after checking for all members of result of Join operation for Pruning.
- Now, the transactions in D are scanned in order to determine L_3, consisting of those candidates 3-itemsets in C_3 having minimum support.

Generating 4-itemset Frequent Pattern

- The algorithm uses L3 Join L3 to generate a candidate set of 4-itemsets, C4. Although the join results in {{I1, I2, I3, I5}}, this itemset is pruned since its subset {{I2, I3, I5}} is not frequent.
- Thus, C4 = ϕ , and algorithm terminates, having found all of the frequent items.
- This completes our Apriori Algorithm.
- Final set of frequent patterns:
  - L = {{I1}, {I2}, {I3}, {I4}, {I5}, {I1,I2}, {I1,I3}, {I1,I5}, {I2,I3}, {I2,I4}, {I2,I5}, {I1,I2,I3}, {I1,I2,I5}}

FP Growth Algorithm

FP Growth Algorithm

- Allows frequent itemset discovery without candidate itemset generation.
- Two step approach:
- Step 1: Build a compact data structure called the FP-tree
  - Built using 2 passes over the data-set.
- Step 2: Extracts frequent itemsets directly from the FP-tree
  - Traversal through FP-Tree

Core Data Structure: FP-Tree

- Nodes correspond to items and have a counter
- FP-Growth reads 1 transaction at a time and maps it to a path
- Fixed order is used, so paths can overlap when transactions share items (when they have the same prefix)
  - In this case, counters are incremented
- Pointers are maintained between nodes containing the same item, creating singly linked lists (dotted lines)
- The more paths that overlap, the higher the compression.
- Frequent itemsets are extracted from the FP-Tree.

Example Dataset

TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}

F-list = a-b-c-d-e

Header table

Item	Pointer
a	
b	
c	
d	
e	

FP-Tree Construction

- FP-Tree is constructed using 2 passes over the data-set

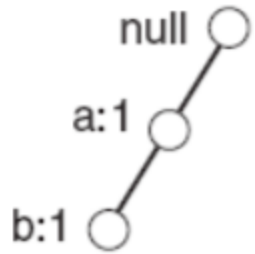
- Pass 1:
  - Scan data and find support for each item.
  - Discard infrequent items.
  - Sort frequent items in decreasing order based on their support.
  - For our example: a, b, c, d, e
  - Use this order when building the FP-Tree, so common prefixes can be shared.

FP-Tree Construction

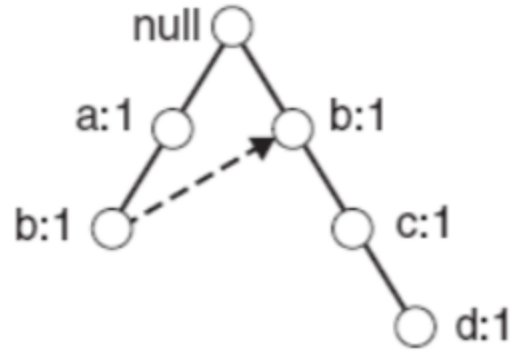
- Pass 2: construct the FP-Tree
- Read transaction 1: {a, b}
  - Create 2 nodes a and b and the path null → a → b. Set counts of a and b to 1.
- Read transaction 2: {b, c, d}
  - Create 3 nodes for b, c and d and the path null → b → c → d. Set counts to 1.
  - Note that although transaction 1 and 2 share b, the paths are disjoint as they don't share a common prefix. Add the link between the b's.
- Read transaction 3: {a, c, d, e}
  - It shares common prefix item a with transaction 1 so the path for transaction 1 and 3 will overlap and the frequency count for node a will be incremented by 1. Add links between the c's and d's.
- Continue until all transactions are mapped to a path in the FP-tree

Transaction Data Set

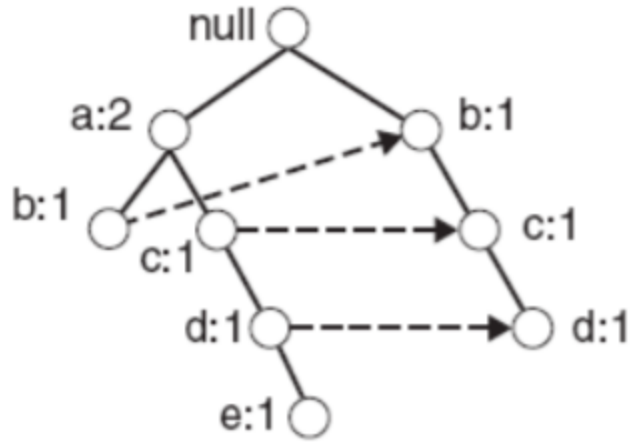
TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}



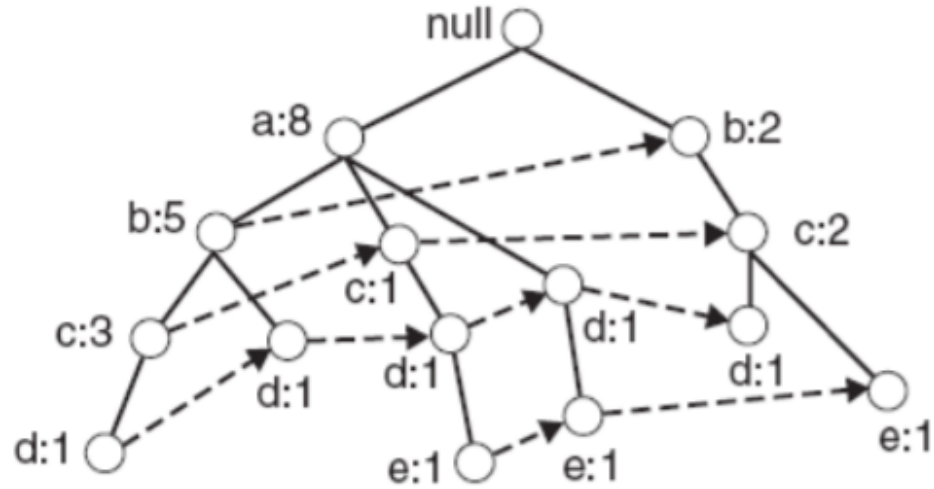
(i) After reading TID=1



(ii) After reading TID=2



(iii) After reading TID=3



(iv) After reading TID=10

TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}

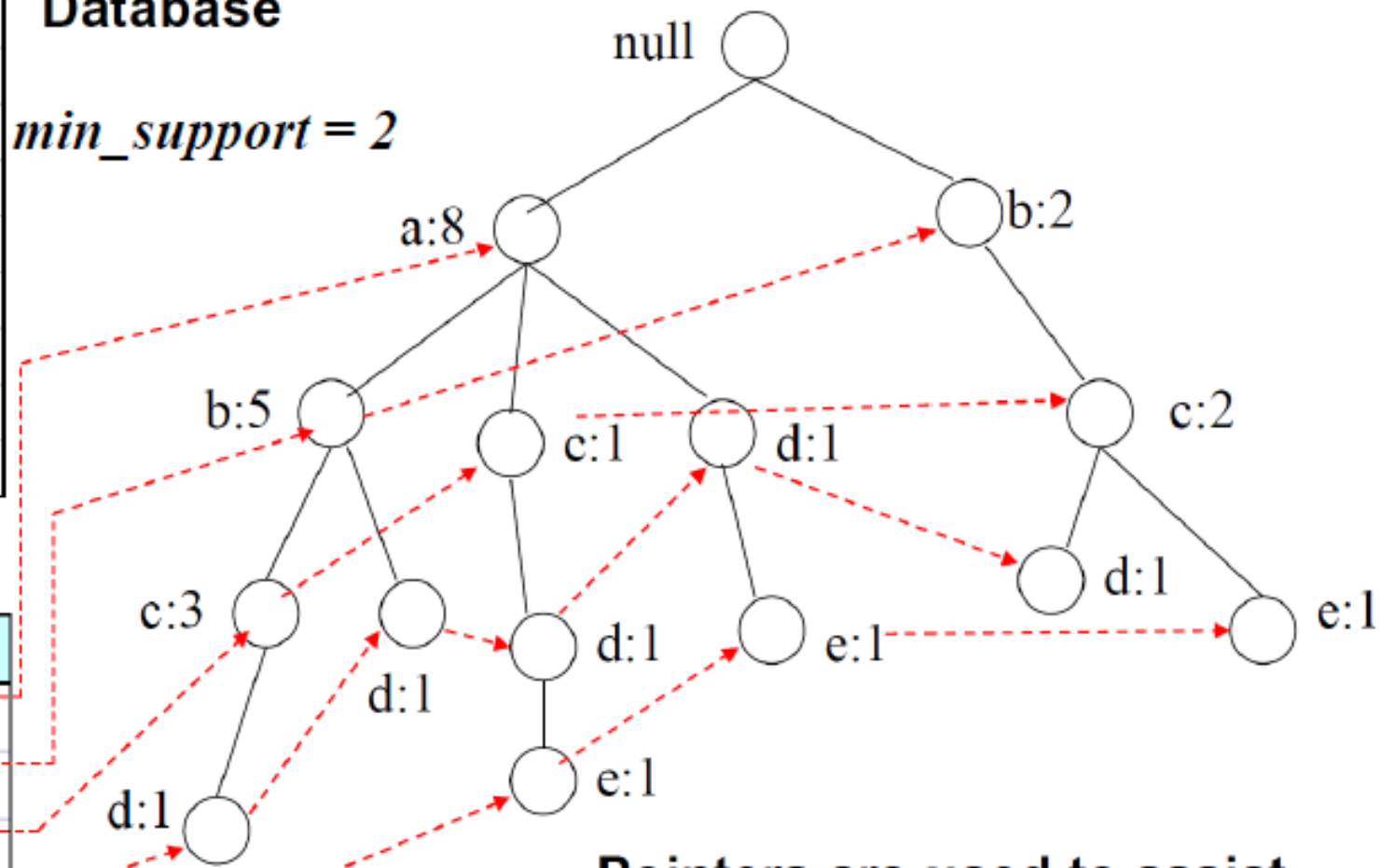
Transaction Database

min_support = 2

F-list = a-b-c-d-e

Header table

Item	Pointer
a	
b	
c	
d	
e	



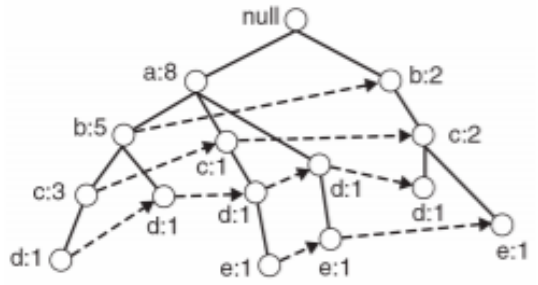
Pointers are used to assist frequent itemset generation

FP-Tree size

- The FP-Tree usually has a smaller size than the uncompressed data – typically many transactions share items (and hence prefixes).
- Best case scenario: all transactions contain the same set of items.
  - 1 path in the FP-tree
- Worst case scenario: every transaction has a unique set of items (no items in common)
  - Size of the FP-tree is at least as large as the original data.
  - Storage requirements for the FP-tree are higher – need to store the pointers between the nodes and the counters.

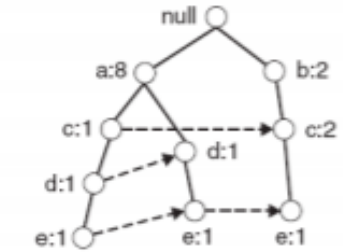
Frequent Itemset Generation

- FP-Growth extracts frequent itemsets from the FP-tree.
- Bottom-up algorithm – from the leaves towards the root
  - Divide and conquer: first look for frequent itemsets ending in e, then de, etc… then d, then cd, etc…
- First, extract prefix path sub-trees ending in an item(set)

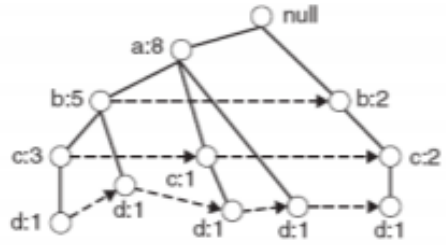


↑ Complete FP-tree

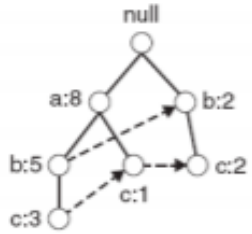
→ **Example:** prefix path sub-trees



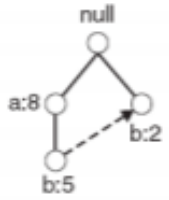
(a) Paths containing node e



(b) Paths containing node d



(c) Paths containing node c

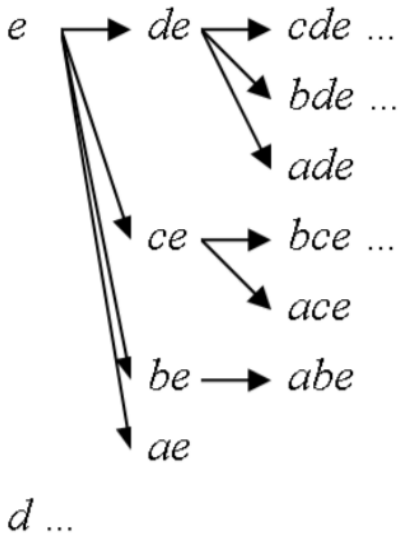


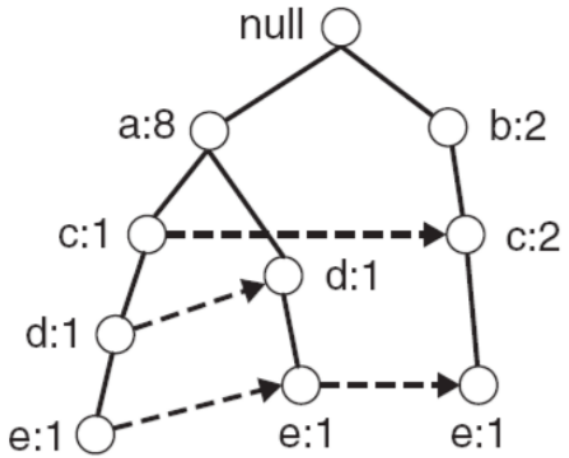
(d) Paths containing node b

(e) Paths containing node a

Frequent Itemset Generation

- Each prefix path sub-tree is processed recursively to extract the frequent itemsets. Solutions are then merged.
  - E.g. the prefix path sub-tree for e will be used to extract frequent itemsets ending in e, then in de, ce, be and ae, then in cde, bde, cde, etc.
  - Divide and conquer approach

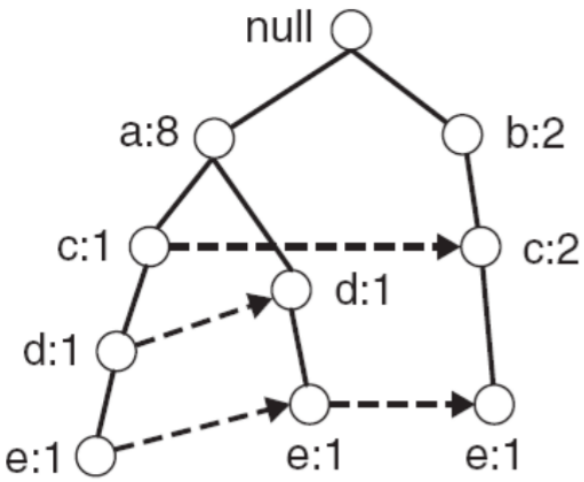




Prefix path sub-tree ending in *e*.

Frequent Itemset Generation Example

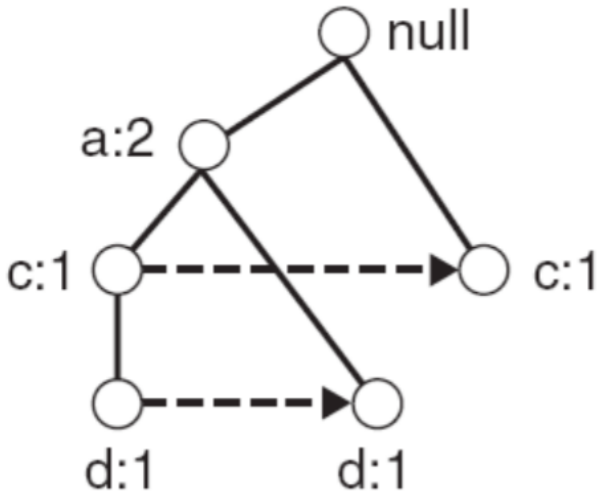
- Let minSup= 2 and extract all frequent itemsets containing e.
  - Obtain the prefix path sub-tree for e:
- Check if e is a frequent item by adding the counts along the linked list (dotted line). If so, extract it.
  - Yes, count is 3 so {e} is extracted as a frequent itemset.
- As e is frequent, find frequent itemsets ending in e, i.e. de, ce, be and ae.
  - decompose the problem recursively.
  - To do this, we must first obtain the conditional FP-tree for e.



Conditional FP-Tree

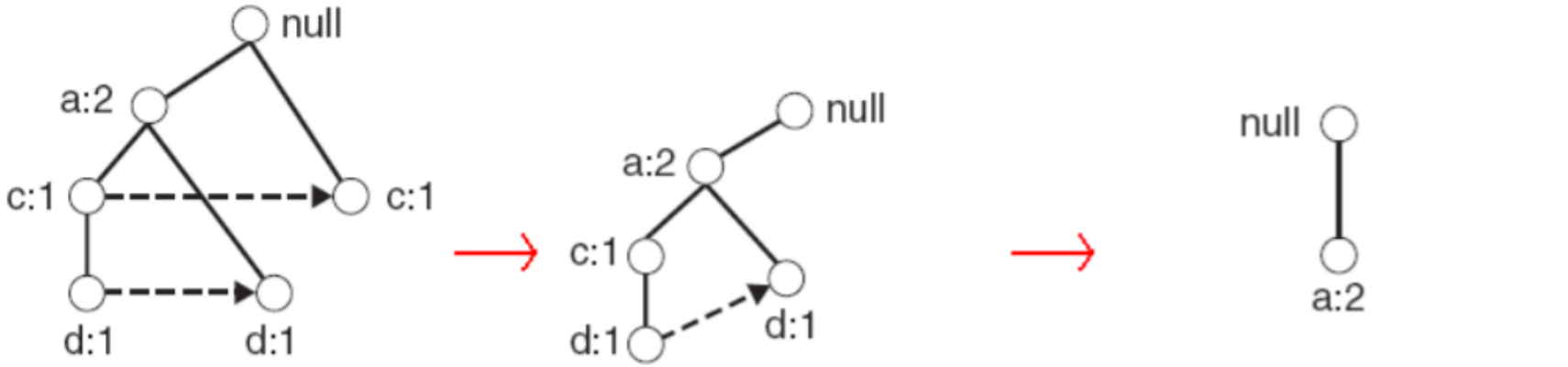
- The FP-Tree that would be built if we only consider transactions containing a particular itemset (and then removing that itemset from all transactions).
- Example: FP-Tree conditional on e. (find F-list and header table again)

TID	Items
~~1~~	~~{a,b}~~
~~2~~	~~{b,c,d}~~
3	{a,c,d,~~e~~}
4	{a,d,~~e~~}
~~5~~	~~{a,b,c}~~
~~6~~	~~{a,b,c,d}~~
~~7~~	~~{a}~~
~~8~~	~~{a,b,c}~~
~~9~~	~~{a,b,d}~~
10	{b,c,~~e~~}



Conditional FP-Tree

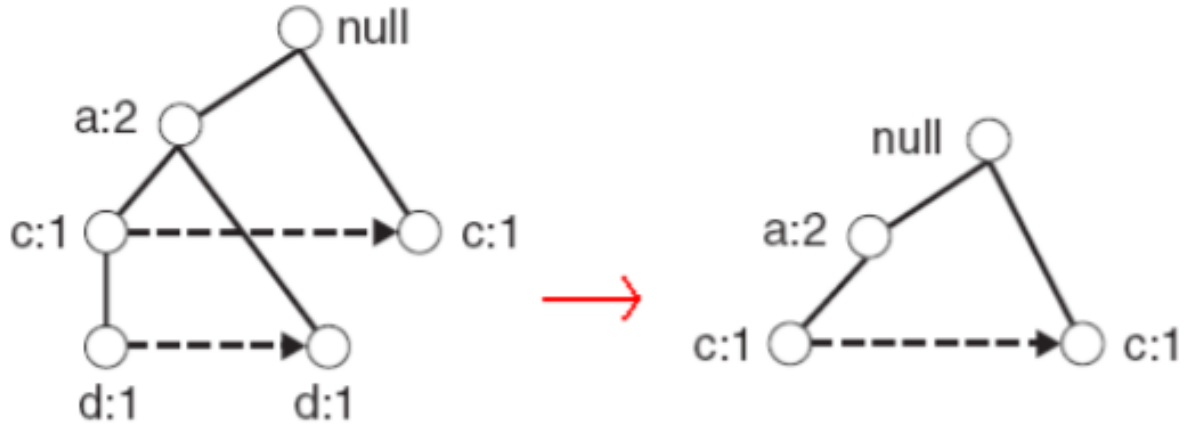
- Use the the conditional FP-tree for e to find frequent itemsets ending in de, ce and ae
  - Note that be is not considered as b is not in the conditional FP-tree for e.
  - For each of them (e.g. de), find the prefix paths from the conditional tree for e, extract frequent itemsets, generate conditional FP-tree, etc... (recursive)
  - Example: e → de → ade ({d, e}, {a, d, e} are found to be frequent)



Conditional FP-tree for e Prefix paths ending in de Conditional FP-tree for de

Conditional FP-Tree

- Use the the conditional FP-tree for e to find frequent itemsets ending in de, ce and ae
- Example: e → ce ({c, e} is found to be frequent)



Conditional FP-tree for e Prefix paths ending in ce

- etc… (ae too, then do the whole thing for b,... etc)

Result

- Frequent itemsets found (ordered by suffix and order in which they are found):

Suffix	Frequent Itemsets
e	{e}, {d,e}, {a,d,e}, {c,e},{a,e}
d	{d}, {c,d}, {b,c,d}, {a,c,d}, {b,d}, {a,b,d}, {a,d}
c	{c}, {b,c}, {a,b,c}, {a,c}
b	{b}, {a,b}
a	{a}

Discussion

- Advantages of FP-Growth
  - only 2 passes over data-set
  - Compresses data-set
  - no candidate generation
  - much faster than Apriori
- Disadvantages of FP-Growth
  - FP-Tree may not fit in memory
  - FP-Tree is expensive to build
    - Trade-off: takes time to build, but once it is built, frequent itemsets are read off easily.
    - Time is wasted (especially if support threshold is high), as the only pruning that can be done is on single items.
    - support can only be calculated once the entire data-set is added to the FP-Tree.

Pattern Evaluation Methods

Misleading Strong Association Rules

- Not all strong association rules are interesting

	Basketball	Not basketball	Sum (row)
Cereal	2000	1750	3750
Not cereal	1000	250	1250
Sum(col.)	3000	2000	5000

- Shall we target people who play basketball for cereal ads? *play basketball* $\Rightarrow$ *eat cereal* [40%, 66.7%]
- Hint: What is the overall probability of people who eat cereal?
  - 3750/5000 = 75% > 66.7%!
- Confidence measure of a rule could be misleading

Other Measures

- From association to correlation
  - Lift
  - χ^2
  - All_confidence
  - Max_confidence
  - Kulczynski
  - Cosine

Interestingness Measure: Correlations (Lift)

- *play basketball* ⇒ *eat cereal* [40%, 66.7%] is misleading
  - The overall % of people eating cereal is 75% > 66.7%.
- *play basketball* ⇒ *not eat cereal* [20%, 33.3%] is more accurate, although with lower support and confidence
- Measure of dependent/correlated events: lift

$$lift = \frac{P(A \cup B)}{P(A)P(B)} \quad P(A \cap B)$$

	Basketball	Not basketball	Sum (row)
Cereal	2000	1750	3750
Not cereal	1000	250	1250
Sum(col.)	3000	2000	5000

$$lift(B, C) = \frac{2000/5000}{3000/5000 * 3750/5000} = 0.89$$

$$lift(B, \neg C) = \frac{1000/5000}{3000/5000 * 1250/5000} = 1.33$$

1: independent
>1: positively correlated
<1: negatively correlated

Correlation Analysis (Nominal Data)

- χ^2 **(chi-square) test**

$$\chi^2 = \sum \frac{(Observed - Expected)^2}{Expected}$$

- Independency test between two attributes
  - The larger the χ^2 value, the more likely the variables are related
- The cells that contribute the most to the χ^2 value are those whose actual count is very different from the expected count under independence assumption
- Correlation does not imply causality
  - # of hospitals and # of car-theft in a city are correlated
  - Both are causally linked to the third variable: population

When Do We Need Chi-Square Test?

- Considering two attributes A and B
  - A: a nominal attribute with c distinct values, $a_1, \dots, a_c$
    - E.g., Grades of Math
  - B: a nominal attribute with r distinct values, $b_1, \dots, b_r$
    - E.g., Grades of Science
- Question: Are A and B related?

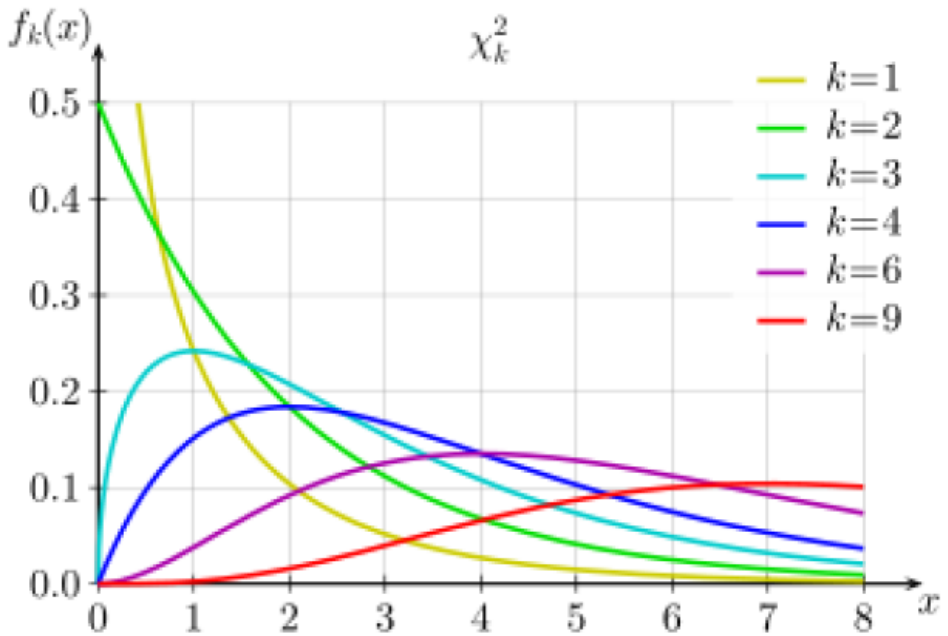
How Can We Run Chi-Square Test?

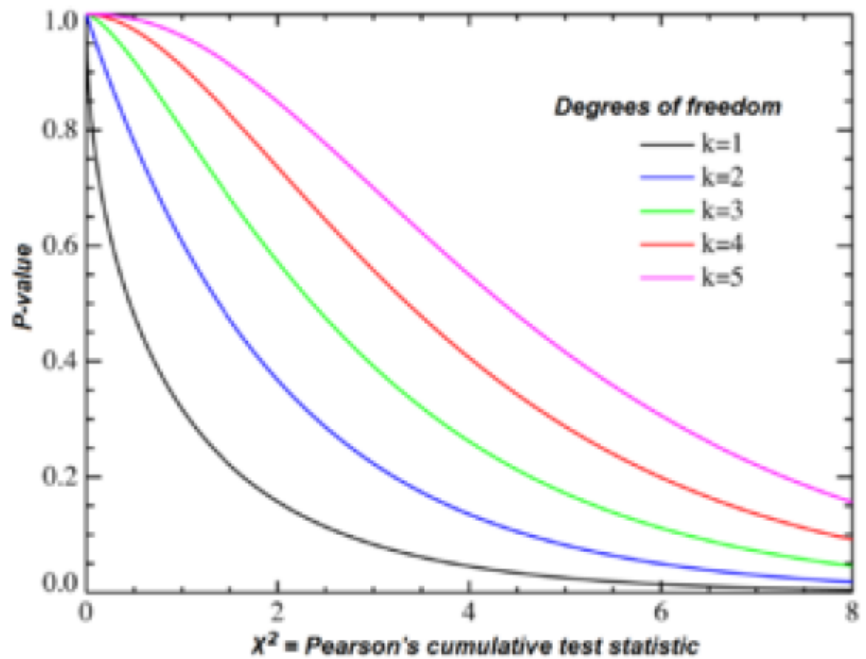
- Constructing contingency table
  - Observed frequency o_{ij}: number of data objects taking value b_i for attribute B and taking value a_j for attribute A

	a_1	a_2	…	a_c
b_1	o_{11}	o_{12}	…	o_{1c}
b_2	o_{21}	o_{22}	…	o_{2c}
…	…	…	…	…
b_r	o_{r1}	o_{r2}	…	o_{rc}

- Calculate expected frequency $e_{ij} = \frac{count(B=b_i) \times count(A=a_j)}{n}$
  - Null hypothesis: A and B are independent

- The Pearson χ^2 statistic is computed as:
  - $X^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(o_{ij} - e_{ij})^2}{e_{ij}}$
  - Follows Chi-squared distribution with degree of freedom as $(r-1) \times (c-1)$





Chi-Square Calculation: An Example

	Play chess	Not play chess	Sum (row)
Like science fiction	250(90)	200(360)	450
Not like science fiction	50(210)	1000(840)	1050
Sum(col.)	300	1200	1500

- χ^2 (chi-square) calculation (numbers in parenthesis are expected counts calculated based on the data distribution in the two categories)

$$\chi^2 = \frac{(250-90)^2}{90} + \frac{(50-210)^2}{210} + \frac{(200-360)^2}{360} + \frac{(1000-840)^2}{840} = 507.93$$

- It shows that like_science_fiction and play_chess are correlated in the group
  - Degree of freedom = (2-1)(2-1) = 1
  - P-value = $P(X^2>507.93) = 0.0$
    - Reject the null hypothesis => A and B are dependent

Are *lift* and χ^2 Good Measures of Correlation?

- Lift and χ^2 are affected by null-transaction
  - E.g., number of transactions that do not contain milk nor coffee
- All_confidence
  - $\text{all\_conf}(A,B)=\min\{P(A|B),P(B|A)\}$
- Max_confidence
  - $\max\_conf(A,B)=\max\{P(A|B),P(B|A)\}$
- Kulczynski
  - $Kulc(A,B) = \frac{1}{2}(P(A|B) + P(B|A))$
- Cosine
  - $cosine(A,B) = \sqrt{P(A|B) \times P(B|A)}$

Sequential Pattern Mining

Sequence Data Base

- A sequence database consists of ordered elements or events

A *transaction database*

TID	itemsets
10	a, b, d
20	a, c, d
30	a, d, e
40	b, e, f

A *sequence database*

SID	sequences
10	<a(abc)(ac)d(cf)>
20	<(ad)c(bc)(ae)>
30	<(ef)(ab)(df)cb>
40	<eg(af)cbc>

Sequence

- Event / element
  - A non-empty set of items, e.g., e=(ab)
- Sequence
  - An ordered list of events, e.g., $s = < e_1\ e_2\ \dots\ e_l >$
- Length of a sequence
  - The number of instances of items in a sequence
  - The length of < (ef) (ab) (df) c b > is 8 (Not 5!)

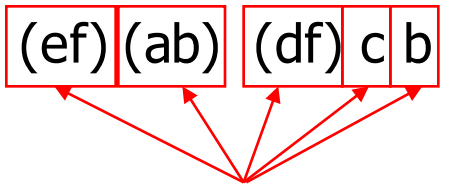
Subsequence vs. Super sequence

- Given two sequences $\alpha = < a_1\ a_2\ \dots\ a_n >$ and $\beta = < b_1\ b_2\ \dots\ b_m >$
- Subsequence
  - α is called a subsequence of β, denoted as $\alpha \subseteq \beta$
  - If there exist integers $1 \leq j_1 < j_2 < \dots < j_n \leq m$ such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}$,…, $a_n \subseteq b_{jn}$
- Super sequence
  - β is a super sequence of α

- Example:
  - <a(bc)dc> is a subsequence of <a(abc)(ac)d(cf)>

Sequential Pattern Mining

- Given a set of sequences and support threshold, find the complete set of *frequent* subsequences

A *sequence* : 

A *sequence database*

SID	sequence
10	<a(abc)(ac)d(cf)>
20	<(ad)c(bc)(ae)>
30	<(ef)(ab)(df)cb>
40	<eg(af)cbc>

An element may contain a set of items.
Items within an element are unordered and we list them alphabetically.

<a(bc)dc> is a *subsequence* of <a(abc)(ac)d(cf)>

Given *support threshold* *min_sup* =2, <(ab)c> is a *sequential pattern*

Methods for Sequential Pattern Mining

- Apriori-based Approaches
  - GSP – Generalized Sequential Pattern Mining
  - SPADE – Sequential PAttern Discovery using Equivalent class

- Pattern-Growth-based Approach
  - PrefixSpan

Generalized Sequential Pattern Mining (GSP)

GSP – Generalized Sequential Pattern Mining

- GSP (Generalized Sequential Patterns)
  - Multi-pass algorithm
  - Candidate generate and test approach

- Strength
  - Pruning candidates by Apriori
- Weakness
  - Generate lots of candidates

The Apriori Property of Sequential Patterns

- A basic property: Apriori (Agrawal & Sirkant'94)
  - If a sequence *S* is not frequent,
    then none of the super-sequences of *S* is frequent
  - E.g, <hb> is infrequent so do <hab> and <(ah)b>

Seq. ID	Sequence
10	<(bd)cb(ac)>
20	<(bf)(ce)b(fg)>
30	<(ah)(bf)abf>
40	<(be)(ce)d>
50	<a(bd)bcb(ade)>

Given *support threshold*
min_sup =2

GSP Algorithm

- Initially, every item in DB is a candidate of length-1
- For each level (i.e., sequences of length-k) do
  - Scan database to collect support count for each candidate sequence
  - Generate candidate length-(k+1) sequences from length-k frequent sequences using Apriori
- Repeat until no frequent sequence or no candidate can be found

Finding Length-1 Sequential Patterns

- Initial candidates:
  - <a>, <b>, <c>, <d>, <e>, <f>, <g>, <h>
- Scan database once, count support for candidates

min_sup =2

Seq. ID	Sequence
10	<(bd)cb(ac)>
20	<(bf)(ce)b(fg)>
30	<(ah)(bf)abf>
40	<(be)(ce)d>
50	<a(bd)bcb(ade)>

Cand	Sup
<a>	**3**
<b>	**5**
<c>	**4**
<d>	**3**
<e>	**3**
<f>	**2**
~~<g>~~	1
~~<h>~~	1

Generating Length-2 Candidates

51 length-2 Candidates

	<a>	<b>	<c>	<d>	<e>	<f>
<a>	<aa>	<ab>	<ac>	<ad>	<ae>	<af>
<b>	<ba>	<bb>	<bc>	<bd>	<be>	<bf>
<c>	<ca>	<cb>	<cc>	<cd>	<ce>	<cf>
<d>	<da>	<db>	<dc>	<dd>	<de>	<df>
<e>	<ea>	<eb>	<ec>	<ed>	<ee>	<ef>
<f>	<fa>	<fb>	<fc>	<fd>	<fe>	<ff>

	<a>	<b>	<c>	<d>	<e>	<f>
<a>		<(ab)>	<(ac)>	<(ad)>	<(ae)>	<(af)>
<b>			<(bc)>	<(bd)>	<(be)>	<(bf)>
<c>				<(cd)>	<(ce)>	<(cf)>
<d>					<(de)>	<(df)>
<e>						<(ef)>
<f>						

Without Apriori property, 8*8+8*7/2=92 candidates

Apriori prunes 44.57% candidates

How to Generate Candidate in General

- From L_{k-1} to C_k
- Step 1: Join
  - $s_1 \; and \; s_2$ can join
    if dropping first item in s_1 is the same as dropping the last item in s_2
  - Example:
    - <(12)3> join <(2)34> = <(12)34>
    - <(12)3> join <(2)(34)> = <(12)(34)>
- Step 2: Pruning
  - Check all length *k-1* subsequence of a candidate is contained in L_{k-1}

GSP Example

- Initial candidates:
  - <a>, <b>, <c>, <d>, <e>, <f>, <g>, <h>
- Scan database once, count support for candidates

min_sup =2

Seq. ID	Sequence
1	<(cd)(abc)(abf)(acdf)>
2	<(abf)e>
3	<(abf)>
4	<(dgh)(bf)(agh)>

Cand	Sup
<a>	**4**
<b>	**4**
<f>	**4**
<d>	**2**
~~<c>~~	1
~~<e>~~	1
~~<g>~~	1
~~<h>~~	1

GSP Example (Cont'd)

min_sup =2

Seq. ID	Sequence
1	<(cd)(abc)(abf)(acdf)>
2	<(abf)e>
3	<(abf)>
4	<(dgh)(bf)(agh)>

- C_2: Length-2 candidates
  - 4 supports: <(bf)>
  - 3 supports: <(ab)> <(af)>
  - 2 supports: <ba> <da> <db> <df> <fa>
  - ~~1 support: <aa> <ab> <ad> <af> <ba> <bb> <bd> <bf> <da> <db><dd> <df> <fa> <fb> <fd> <ff> <(ad)> <(bd)> <(bf)> <(df)>~~

Cand	Sup
<a>	**4**
<b>	**4**
<f>	**4**
<d>	**2**

- L_2: Length-2 frequent sequences
  - <ba> <da> <db> <df> <fa> <(ab)> <(af)> <(bf)>

GSP Example (Cont'd)

min_sup =2

Seq. ID	Sequence
1	<(cd)(abc)(abf)(acdf)>
2	<(abf)e>
3	<(abf)>
4	<(dgh)(bf)(agh)>

- L_2: Length-2 frequent sequences
  - <ba> <da> <db> <df> <fa> <(ab)> <(af)> <(bf)>
- C_3: Length-3 candidates generated by join
  - <ba> and <(ab)> = <b(ab)> {1}
  - <ba> and <(af)> = <b(af)> {1}
  - <da> and <(ab)> = <d(ab)> {1}
  - <da> and <(af)> = <d(af)> {1}
  - <db> and <(bf)> = <d(bf)> {1, 4}
  - <db> and <ba> = <dba> {1, 4}
  - <df> and <fa> = <dfa> {1, 4}
  - <fa> and <(ab)> = <f(ab)> {}
  - <fa> and <(af)> = <f(af)> {1}
  - <(ab)> and <(bf)> = <(abf)> {1,2,3}
  - <(ab)> and <ba> = <(ab)a> {1}
  - <(af)> and <fa> = <(af)a) {1}
  - <(bf)> and <fa> = <(bf)a> {1, 4}
- L_3: Length-3 frequent sequences
  - <dba> <dfa> <(abf)> <(bf)a> <d(bf)>
- C_4: Length-4 candidates generated by join
  - <d(bf)> and <(bf)a> = <d(bf)a> {1, 4}
  - <(abf)> and <(bf)a> = <(abf)a> {1}
- L_4: Length-4 frequent sequences
  - <d(bf)a>

Short Summary of GSP

- Benefits from the Apriori pruning
  - Reduces search space
- Bottlenecks
  - Scans the database multiple times
  - Generates a huge set of candidate sequences

The SPADE Algorithm

The SPADE Algorithm

- SPADE – Sequential PAttern Discovery using Equivalent class
- A vertical format sequential pattern mining method
- A sequence database is mapped to a large set of
  - Item: <sequence_ID (SID), event_ID (EID)>
- Mapping from horizontal to vertical format requires only one scan
- Support of k-sequences can be determined by joining the ID lists of (k-1) sequences

The SPADE Algorithm (Cont'd)

SID	EID	Items
1	1	a
1	2	abc
1	3	ac
1	4	d
1	5	cf
2	1	ad
2	2	c
2	3	bc
2	4	ae
3	1	ef
3	2	ab
3	3	df
3	4	c
3	5	b
4	1	e
4	2	g
4	3	af
4	4	c
4	5	b
4	6	c

a		b		...
SID	EID	SID	EID	...
1	1	1	2	
1	2	2	3	
1	3	3	2	
2	1	3	5	
2	4	4	5	
3	2			
4	3			

ab			ba			...
SID	EID (a)	EID(b)	SID	EID (b)	EID(a)	...
1	1	2	1	2	3	
2	1	3	2	3	4	
3	2	5				
4	3	5				

aba				...
SID	EID (a)	EID(b)	EID(a)	...
1	1	2	3	
2	1	3	4	

Short Summary of SPADE

- Benefits:
  - Reduces scans of the sequence database
- Bottlenecks:
  - But large set of candidates are still generated

Bottlenecks of Candidate Generate-and-test

- A huge set of candidates generated.
  - Especially 2-item candidate sequence.
- Multiple Scans of database in mining.
  - The length of each candidate grows by one at each database scan.
- Inefficient for mining long sequential patterns.
  - A long pattern grow up from short patterns
  - An exponential number of short candidates

PrefixSpan

PrefixSpan

- PrefixSpan – Prefix-Projected Sequential Pattern Growth

- Pattern Growth – does not require candidate generation
  - Constructs FP-tree
  - Projected databases associated with each frequent item are generated from FP-tree
  - Builds prefix patterns which it concatenates with suffix patterns to find frequent patterns

Prefix and Suffix

- <a>, <aa>, <a(ab)> and <a(abc)> are *prefixes* of sequence <a(abc)(ac)d(cf)>
  - Note <a(ac)> is not a prefix of <a(abc)(ac)d(cf)>
- Given the sequence <a(abc)(ac)d(cf)>

Prefix	*Suffix*
<a>	<(abc)(ac)d(cf)>
<aa>	<(_bc)(ac)d(cf)>
<ab>	<(_c)(ac)d(cf)>

(_bc) means: the last element in the prefix together with (bc) form one element

Prefix-based Projection

- Given a sequence, α, let β be a subsequence of α, and α' is be subsequence of α
  - α' is called a projection of α w.r.t. prefix β, if only and only if
    - α' has prefix β, and
    - α' is the maximum subsequence of α with prefix β

- Example
  - <ad(cf)> is a projection of w.r.t. prefix of <a(abc)(ac)d(cf)> w.r.t. the prefix <ad>

SID	sequence
10	<a(abc)(ac)d(cf)>
20	<(ad)c(bc)(ae)>
30	<(ef)(ab)(df)cb>
40	<eg(af)cbc>

Projected (Suffix) Database

- The collection of suffixes of projections of sequences in the database w.r.t. the prefix sequential pattern α
- Example
  - <a>-projected database
    - <(abc)(ac)d(cf)>
    - <(_d)c(bc)(ae)>
    - <(_b)(df)cb>
    - <(_f)cbc>
  - <ab>-projected database
    - <(_c)(ac)d(cf)>
    - <(_c)(ae)>
    - <c>

SID	sequence
10	<a(abc)(ac)d(cf)>
20	<(ad)c(bc)(ae)>
30	<(ef)(ab)(df)cb>
40	<eg(af)cbc>

Mining Sequential Patterns by Prefix Projections

- Step 1: find length-1 sequential patterns
  - <a>, <b>, <c>, <d>, <e>, <f>
- Step 2: divide search space. The complete set of sequence patterns can be partitioned into 6 subsets:
  - The ones having prefix <a>;
  - The ones having prefix <b>;
  - …
  - The ones having prefix <f>
- Step 3: mine each subset recursively via corresponding projected databases

SID	sequence
10	<a(abc)(ac)d(cf)>
20	<(ad)c(bc)(ae)>
30	<(ef)(ab)(df)cb>
40	<eg(af)cbc>

Finding Sequence Patterns with Prefix <a>

SID	sequence
10	<a(abc)(ac)d(cf)>
20	<(ad)c(bc)(ae)>
30	<(ef)(ab)(df)cb>
40	<eg(af)cbc>

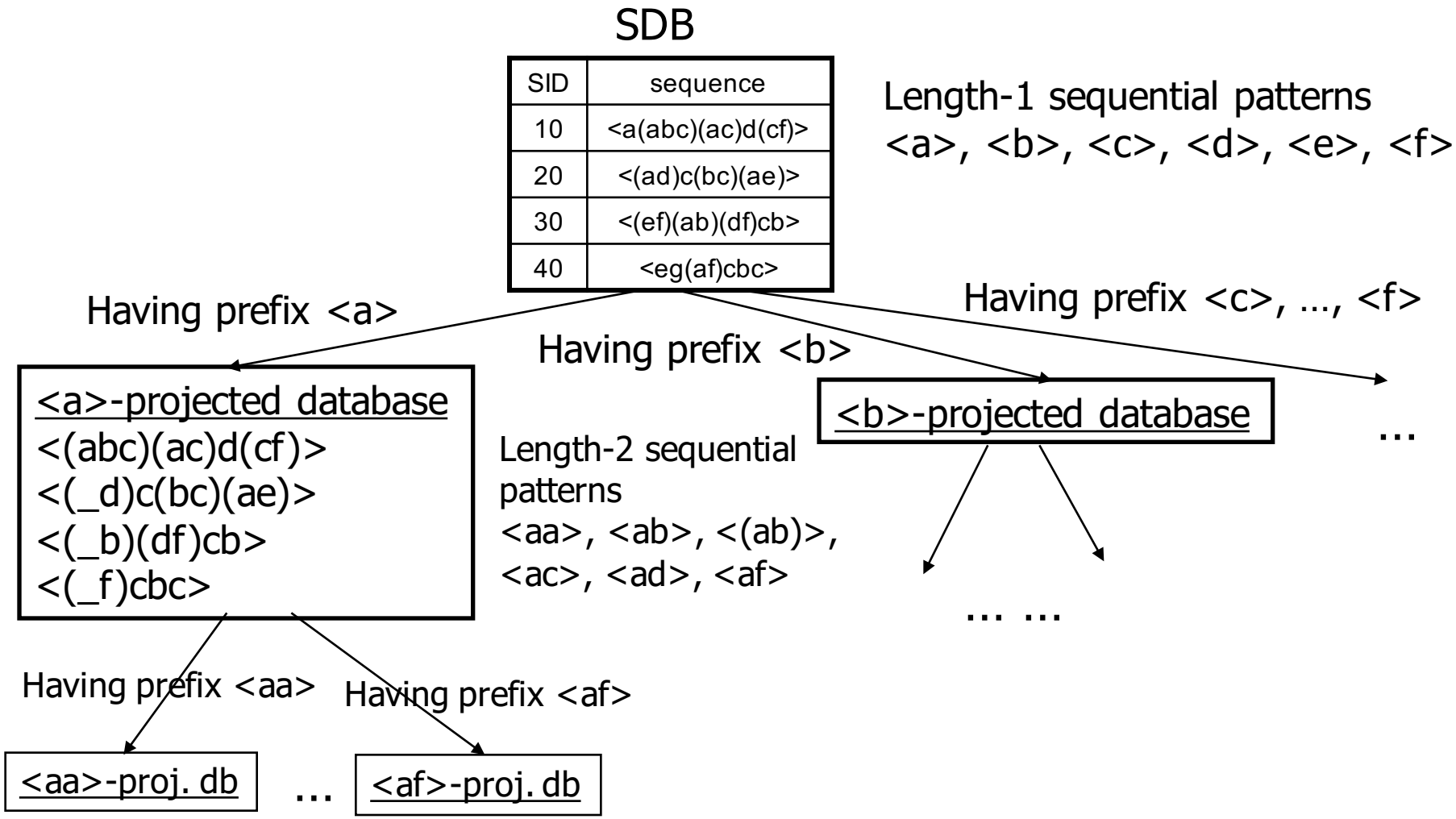
- Only need to consider projections w.r.t. <a>
  - <a>-projected database:
    <(abc)(ac)d(cf)> <(_d)c(bc)(ae)> <(_b)(df)cb> <(_f)cbc>
- Find all the length-2 sequence patterns.
  Having prefix <a>: <aa> <ab> <(ab)> <ac> <ad> <af>
  - Further partition into 6 subsets
    - Having prefix <aa>;
    - …
    - Having prefix <af>

Why are those 6 subsets?

- By scanning the <a>-projected database once, its locally frequent items are identified as
  - a:2, b:4, _b: 2, c: 4, d: 2, and f: 2

- Thus, all the length-2 sequential patterns prefixed with <a> are found, and they are:
  - <aa>: 2, <ab>: 4, <(ab)>: 2, <ac>: 4, <ad>: 2, and <af>:2

Completeness of PrefixSpan



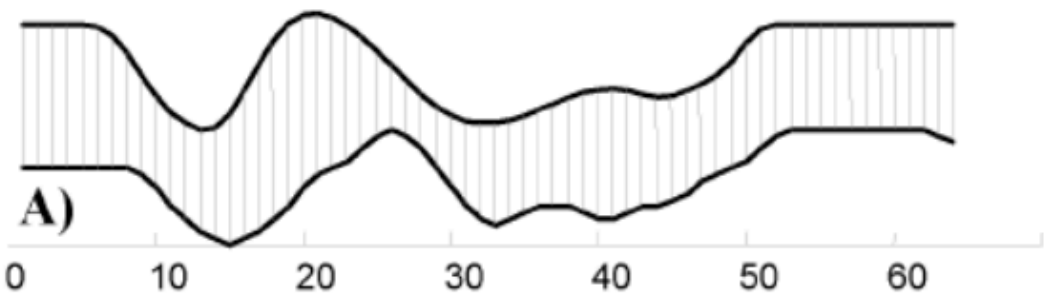
Short Summary of PrefixSpan

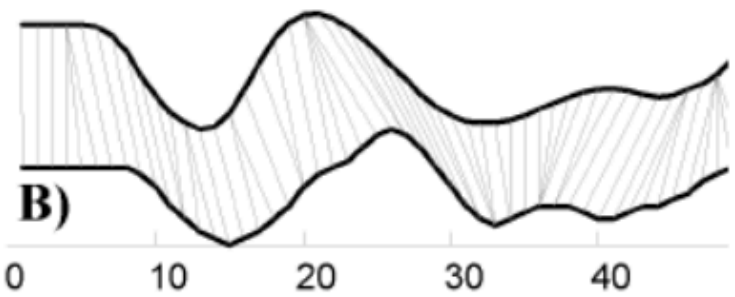
- No candidate sequence needs to be generated
- Projected databases keep shrinking
- Major cost of PrefixSpan: constructing projected databases
  - Can be improved by pseudo-projections

Dynamic Time Warping (DTW)

Dynamic Time Warping (DTW)

- For two sequences that do not line up well in X-axis, but share roughly similar shape
  - We need to warp the time axis to make better alignment





Goal of DTW

- Given
  - Two sequences (with possible different lengths):
    - $X = \{x_1, x_2, \ldots, x_N\}$
    - $Y = \{y_1, y_2, \ldots, y_M\}$
  - A local distance (cost) measure between x_n and y_m: $c(x_n, y_m)$
- Goal:
  - Find an alignment between X and Y, such that, the overall cost is minimized

Represent an Alignment by Warping Path

- An (N,M)-warping path is a sequence $p = (p_1, p_2, \ldots, p_L)$ with $p_l = (n_l, m_l)$, satisfying the three conditions:
  - Boundary condition: $p_1 = (1,1), p_L = (N, M)$
    - Starting from the first point and ending at last point
  - Monotonicity condition: n_l and m_l are non-decreasing with l
  - Step size condition:
    - $p_{l+1} - p_l \in \{(0,1), (1,0), (1,1)\}$
    - Move one step right, up, or up-right

Optimal Warping Path

- The total cost given a warping path p
  - $c_p(X, Y) = \sum_l c(x_{n_l}, y_{m_l})$
- The optimal warping path p*
  - $c_{p^*}(X, Y) = \min\{c_p(X, Y) | p \text{ is an } (N, M) - \text{warping path}\}$
- DTW distance between X and Y is defined as:
  - the optimal cost $c_{p^*}(X, Y)$

Dynamic Programming for DTW

- Dynamic programming:
  - Let D(n,m) denote the DTW distance between X(1,...,n) and Y(1,...,m)
  - D is called accumulative cost matrix
    - Note D(N,M) = DTW(X,Y)
  - Recursively calculate D(n,m)
    - $D(n,m) = \min\{D(n-1,m), D(n,m-1), D(n-1,m-1)\} + c(x_n, y_m)$
    - When m or n = 1
      - $D(n,1) = \sum_{k=1:n} c(x_k, y_1)$;
      - $D(1,m) = \sum_{k=1:m} c(x_1, y_k)$;

Time complexity: O(MN)

DTW Example

- Sequence 1: 1 1 2 3 2 0
- Sequence 2: 0 1 1 2 3 2 1
- $c(x, y) = (x - y)^2$

1	7	7	3	6	2	2
2	7	7	2	2	1	5
3	6	6	2	1	2	11
2	2	2	1	2	2	6
1	1	1	2	6	7	8
1	1	1	2	6	7	8
0	1	2	6	15	19	19
	1	1	2	3	2	0