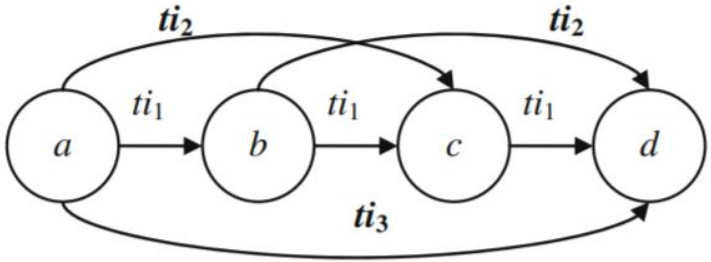
On Mining Multi-Time-Interval Sequential Patterns

Submitted By,
Rajath C Aralikatti - 181CO241
Sangeeth S V - 181CO246

Mining MTIS Patterns

Multi-Time-Interval Sequential (MTIS) Patterns are patterns that contain information about the time differences between sequentially occurring events. For example, a person who purchases the first volume of a book series, is likely to purchase the second one a week after and the third one another 2 weeks later and so on. By mining MTIS patterns, we seek to establish such timed relations between sequentially occurring events.

Multi-time-interval Sequential Patterns:



Time stamped Sequential Database

sid	Sequence
10	<(*a*, 1), (*b*, 3), (*c*, 3), (*a*, 5), (*e*, 5), (*c*, 10)>
20	<(*d*, 5), (*a*, 7), (*b*, 7), (*e*, 7), (*d*, 8), (*e*, 8), (*c*, 14), (*d*, 15)>
30	<(*a*, 8), (*b*, 8), (*e*, 11), (*d*, 12), (*b*, 13), (*c*, 13), (*c*, 16)>
40	<(*b*, 15), (*f*, 15), (*e*, 16), (*b*, 17), (*c*, 17)>

This is an example for a Sequential Database with time stamped data. Each item sequence is a list of tuples of the form (item, timestamp). For running our MTIS mining algorithms we must ensure that in each item sequence the items are ordered in the non-decreasing order of their time stamps.

MTIS Patterns

Let the MTIS pattern P = {a, (ti_1), b, (ti_2, ti_1), c, (ti_2, ti_1, ti_1), d}. This means that:

- The time between a&b is ti_1.
- The time between a&c is ti_2 and b&c is ti_1.
- The time between a&d is ti_2 , b&d is ti_1 and c&d is ti_1.

Let us suppose the item sequence A = {(a,1),(b,3),(c,5),(d,6),(e,8)} and the time intervals T = {ti_0: 0; ti_1: [0,3]; ti_2: [3,6]; ti_3: [6,∞)}.

Here, we can say that the pattern P is contained in A, because the time differences between events in the item sequence match the time intervals defined by the pattern.

Properties of MTIS Patterns

- Descending Property: Each successive time interval specified in any part of the MTIS pattern is always at most equal to the previous time interval. For example, (ti_3, ti_2, ti_2,ti_1) is valid, whereas, (ti_2, ti_3, ti_2, ti_1) is not valid.
- Support: Support of an MTIS pattern is defined as the ratio of number of item sequences in which it is contained to the total number of item sequences in the database.
- Anti-Monotonicity: If a MTIS pattern is frequent, so are all of its subsequences. Accordingly, if a MTIS pattern is not frequent, then its super sequence will not be either. A sequence is said to be frequent if it has more than the minimum support specified for the data.

MTIS Algorithms

There are **two algorithms** to mine MTIS patterns from time-stamped sequential data. The goal of these algorithms is to **reveal the time relationships** between all pairs of items. They are:

- ➔ **MI - Apriori Algorithm**
- ➔ **MI - PrefixSpan Algorithm**

Experimental results show that MI-PrefixSpan algorithm is generally faster on small length sequential data. MI-Apriori algorithm has better scalability in this scenario.

MI - Apriori Algorithm

The idea of MI - Apriori Algorithm is to generate higher order sequences (C_k) from the previous set of sequences (C_{k-1}). The fundamental principle behind the apriori algorithm is the anti-monotonicity property of MTIS patterns. So, we can say that every k-sequence must be formed from two (k-1)-subsequences that also satisfy the minimum support requirement.

$L_1 = \text{Find\_1} - \text{frequent} - \text{item}(S);$
For($k = 2; L_{k-1} \neq null; k++$){
$C_k = MI - Apriori\_gen(L_{k-1}, TI)$; //generate all possible candidates
For each sequence $c \in C_k$ {
For each sequence $s \in S$ {
If (c contained in s) then{
$c.count++;$
}
}
}
$L_k = \{c \in C_k | c.count \geq min\_sup\}$
}
return $\cup L_k$;

- The basic join operation seeks to take two k - MTIS patterns as input and combine them to form (k+1) - MTIS patterns. For example consider, $P_1 = \{a, (t_0), b, (t_1, t_1), e\}$ and $P_2 = \{b, (t_1), e, (t_3, t_2), c\}$.
- The idea is to check the equality between the latter (k-1) elements of one pattern and the first (k-1) elements of the other pattern. Here, we can see that $\{b, (t_1), e\}$ forms the first part of P_2 and the last part of P_1.
- Now, the new pattern $P' = \{a, (t_0), b, (t_1, t_1), e, (?, t_3, t_2), c\}$.
- The question mark (?) is computed using the time interval information matrix generated using the set of time intervals $\{t_0, t_1, t_2, t_3\}$.
- Here, $t_3 + t_0 = t_3 \Rightarrow P' = \{a, (t_0), b, (t_1, t_1), e, (t_3, t_3, t_2), c\}$.

$ti_0\,(t=0)$	ti_0	-	-	-
$ti_1\,(0<t\leq 3)$	ti_1	ti_1, ti_2	-	-
$ti_2\,(3<t\leq 6)$	ti_2	ti_2, ti_3	ti_3	-
$ti_3\,(6<t\leq \infty)$	ti_3	ti_3	ti_3	ti_3
	$ti_0\,(t=0)$	$ti_1\,(0<t\leq 3)$	$ti_2\,(3<t\leq 6)$	$ti_3\,(6<t\leq \infty)$

MI-Apriori Algorithm

Min Support = 0.75

Time Intervals = {ti_1:(0,5), ti_2:(5,inf)}

Seq. Id	Item Sequence Pattern
1	<(a,1), (b,3), (c,3), (a,5), (e,5), (c,10)>
2	<(d,5), (a,7), (b,7), (e,7), (d,8), (e,8), (c,14), (d,15)>
3	<(a,8), (b,8), (e,11), (d,12), (b,13), (c,13), (c,16)>
4	<(b,15), (f,15), (e,16), (b,17), (c,17)>

Generating 1-item Sequences

C_1

Candidates	Support
<a>	3
<b>	4
<c>	4
<e>	4
<d>	2
<f>	1

Pruning

L_1

Candidates	Support
<a>	3
<b>	4
<c>	4
<e>	4

Generating 2-item Sequences

C_2

Candidates	Support
<a, (ti_1), b>	3
<a, (ti_2), b>	0
<a, (ti_1), c>	2
<a, (ti_2), c>	3
<a, (ti_1), e>	3
<a, (ti_2), e>	0
<b, (ti_1), a>	1
<b, (ti_2), a>	0
<b, (ti_1), c>	3
<b, (ti_2), c>	3
<b, (ti_1), e>	4
<b, (ti_2), e>	0
<c, (ti_1), a>	1
<c, (ti_2), a>	0
<c, (ti_1), b>	0
<c, (ti_2), b>	0
<c, (ti_1), e>	1
<c, (ti_2), e>	0
<e, (ti_1), a>	0
<e, (ti_2), a>	0
<e, (ti_1), b>	2
<e, (ti_2), b>	0
<e, (ti_1), c>	3
<e, (ti_2), c>	1

Pruning

L_2

Candidates	Support
<a, (ti_1), b>	3
<a, (ti_2), c>	3
<a, (ti_1), e>	3
<b, (ti_1), c>	3
<b, (ti_2), c>	3
<b, (ti_1), e>	4
<e, (ti_1), c>	3

Generating 3-item Sequences

C_3

Candidates	Support
<a, (ti_1), b, (ti_1, ti_1), c>	2
<a, (ti_1), b, (ti_2, ti_2), c>	3
<a, (ti_1), b, (ti_1, ti_1), e>	3
<a, (ti_1), e, (ti_1, ti_1), c>	2
<b, (ti_1), e, (ti_1, ti_1), c>	2

Pruning

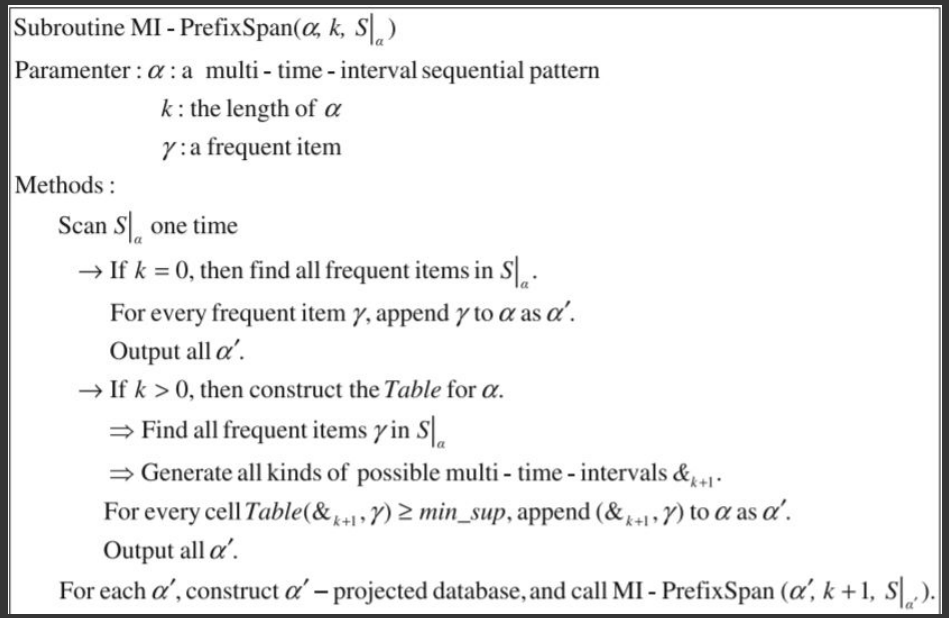
L_3

Candidates	Support
<a, (ti_1), b, (ti_2, ti_2), c>	3
<a, (ti_1), b, (ti_1, ti_1), e>	3

No further joins are possibe as per the join algorithm. Therefore, there exists a total of 13 (4 1-item, 7 2-item and 2 3-item) multi-time-interval sequential patterns.

MI - PrefixSpan Algorithm

The idea of MI - PrefixSpan Algorithm is to generate MTIS patterns recursively. In the prefixspan algorithm, we first select a prefix from the list of 1-frequent items. Then the frequent patterns starting with the selected prefix can be obtained from the projected database and recursively performing this algorithm.



Subroutine MI - PrefixSpan(α, k, $S|_\alpha$)

Paramenter : α : a multi - time - interval sequential pattern

k : the length of α

γ : a frequent item

Methods :

Scan $S|_\alpha$ one time

→ If $k = 0$, then find all frequent items in $S|_\alpha$.

For every frequent item γ, append γ to α as α'.

Output all α'.

→ If $k > 0$, then construct the *Table* for α.

⇒ Find all frequent items γ in $S|_\alpha$

⇒ Generate all kinds of possible multi - time - intervals $\&_{k+1}$.

For every cell $Table(\&_{k+1}, \gamma) \geq min\_sup$, append $(\&_{k+1}, \gamma)$ to α as α'.

Output all α'.

For each α', construct α' – projected database, and call MI - PrefixSpan (α', $k+1$, $S|_{\alpha'}$).

The projected database for a prefix is denoted by a 3-tuple notation for each row in the projected database.

- The first part denotes the sequence id in the database.
- The following list in the second part denotes the indices of the items in the prefix.
- The last part denotes the timestamps of each of the items in the prefix.

Consider the following:

- Item sequence: A = {(a, 1), (b, 3), (c, 3), (a, 5), (e, 5), (c, 10)} and id=10.
- Prefix: P = {a, (ti_1), c}, where ti_1 = (0, 3)
- The projection notation and the corresponding projected database would look like:
  [10 , [1, 3] , [1, 3]] = {(a, 5), (e, 5), (c, 10)}

MI-PrefixSpan Algorithm

Min Support = 0.75

Time Intervals = {ti_1:(0,5), ti_2:(5,inf)}

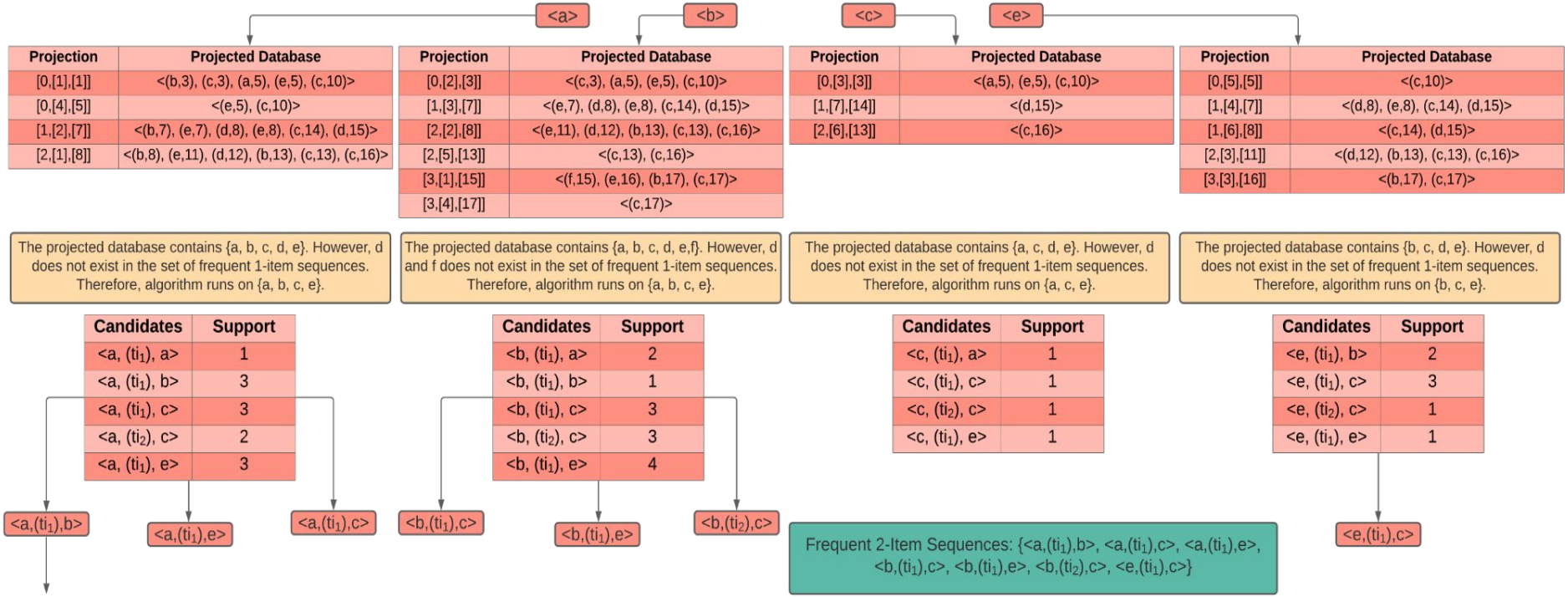
Seq. Id	Item Sequence Pattern
0	<(a,1), (b,3), (c,3), (a,5), (e,5), (c,10)>
1	<(d,5), (a,7), (b,7), (e,7), (d,8), (e,8), (c,14), (d,15)>
2	<(a,8), (b,8), (e,11), (d,12), (b,13), (c,13), (c,16)>
3	<(b,15), (f,15), (e,16), (b,17), (c,17)>

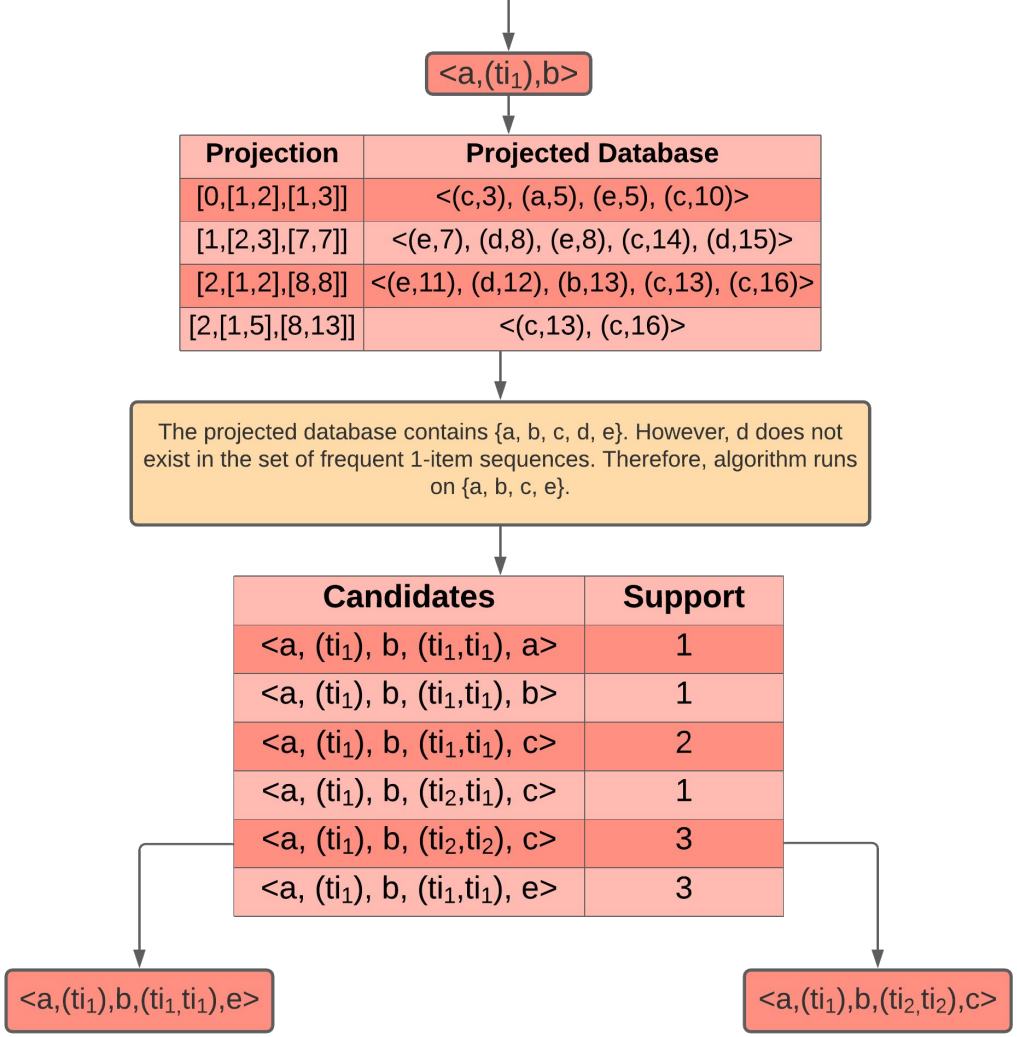
One Item Sets

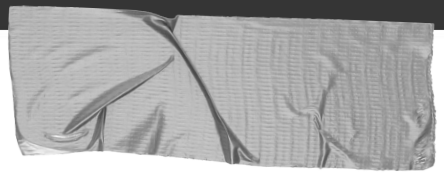
Candidates	Support
<a>	3
<b>	4
<c>	4
<e>	4
<d>	2
<f>	1

Frequent 1-Item Sequences: {<a>, <b>, <c>, <e>}

<a> <b> <c> <e>


<a>
<b>
<c>
<e>
Projection
Projected Database
[0,[1],[1]] <(b,3), (c,3), (a,5), (e,5), (c,10)>
[0,[4],[5]] <(e,5), (c,10)>
[1,[2],[7]] <(b,7), (e,7), (d,8), (e,8), (c,14), (d,15)>
[2,[1],[8]] <(b,8), (e,11), (d,12), (b,13), (c,13), (c,16)>
[0,[2],[3]] <(c,3), (a,5), (e,5), (c,10)>
[1,[3],[7]] <(e,7), (d,8), (e,8), (c,14), (d,15)>
[2,[2],[8]] <(e,11), (d,12), (b,13), (c,13), (c,16)>
[2,[5],[13]] <(c,13), (c,16)>
[3,[1],[15]] <(f,15), (e,16), (b,17), (c,17)>
[3,[4],[17]] <(c,17)>
[0,[3],[3]] <(a,5), (e,5), (c,10)>
[1,[7],[14]] <(d,15)>
[2,[6],[13]] <(c,16)>
[0,[5],[5]] <(c,10)>
[1,[4],[7]] <(d,8), (e,8), (c,14), (d,15)>
[1,[6],[8]] <(c,14), (d,15)>
[2,[3],[11]] <(d,12), (b,13), (c,13), (c,16)>
[3,[3],[16]] <(b,17), (c,17)>
The projected database contains {a, b, c, d, e}. However, d does not exist in the set of frequent 1-item sequences. Therefore, algorithm runs on {a, b, c, e}.
The projected database contains {a, b, c, d, e,f}. However, d and f does not exist in the set of frequent 1-item sequences. Therefore, algorithm runs on {a, b, c, e}.
The projected database contains {a, c, d, e}. However, d does not exist in the set of frequent 1-item sequences. Therefore, algorithm runs on {a, c, e}.
The projected database contains {b, c, d, e}. However, d does not exist in the set of frequent 1-item sequences. Therefore, algorithm runs on {b, c, e}.
Candidates
Support
<a, (ti1), a> 1
<a, (ti1), b> 3
<a, (ti1), c> 3
<a, (ti2), c> 2
<a, (ti1), e> 3
<b, (ti1), a> 2
<b, (ti1), b> 1
<b, (ti1), c> 3
<b, (ti2), c> 3
<b, (ti1), e> 4
<c, (ti1), a> 1
<c, (ti1), c> 1
<c, (ti2), c> 1
<c, (ti1), e> 1
<e, (ti1), b> 2
<e, (ti1), c> 3
<e, (ti2), c> 1
<e, (ti1), e> 1
<a,(ti1),b>
<a,(ti1),e>
<a,(ti1),c>
<b,(ti1),c>
<b,(ti1),e>
<b,(ti2),c>
<e,(ti1),c>
Frequent 2-Item Sequences: {<a,(ti1),b>, <a,(ti1),c>, <a,(ti1),e>, <b,(ti1),c>, <b,(ti1),e>, <b,(ti2),c>, <e,(ti1),c>}


<a,(ti1),b>
Projection
Projected Database
[0,[1,2],[1,3]]
<(c,3), (a,5), (e,5), (c,10)>
[1,[2,3],[7,7]]
<(e,7), (d,8), (e,8), (c,14), (d,15)>
[2,[1,2],[8,8]]
<(e,11), (d,12), (b,13), (c,13), (c,16)>
[2,[1,5],[8,13]]
<(c,13), (c,16)>
The projected database contains {a, b, c, d, e}. However, d does not exist in the set of frequent 1-item sequences. Therefore, algorithm runs on {a, b, c, e}.
Candidates
Support
<a, (ti1), b, (ti1,ti1), a>
1
<a, (ti1), b, (ti1,ti1), b>
1
<a, (ti1), b, (ti1,ti1), c>
2
<a, (ti1), b, (ti2,ti1), c>
1
<a, (ti1), b, (ti2,ti2), c>
3
<a, (ti1), b, (ti1,ti1), e>
3
<a,(ti1),b,(ti1,ti1),e>
<a,(ti1),b,(ti2,ti2),c>



Improvements

We have made **a few runtime optimizations** to the two algorithms. The goal of these optimizations is to produce slight changes in the implementation of the algorithms which can drastically affect the **runtime** and **memory utilization** of the algorithms as the problem is scaled.

- ➔ **MI - Apriori Algorithm**
  An optimization in the joinCk method by combining the pruning and candidate generation steps.
- ➔ **MI - PrefixSpan Algorithm**
  An optimization in the time-interval table generation step; reduces the memory utilization drastically and consequently reduces the runtime.

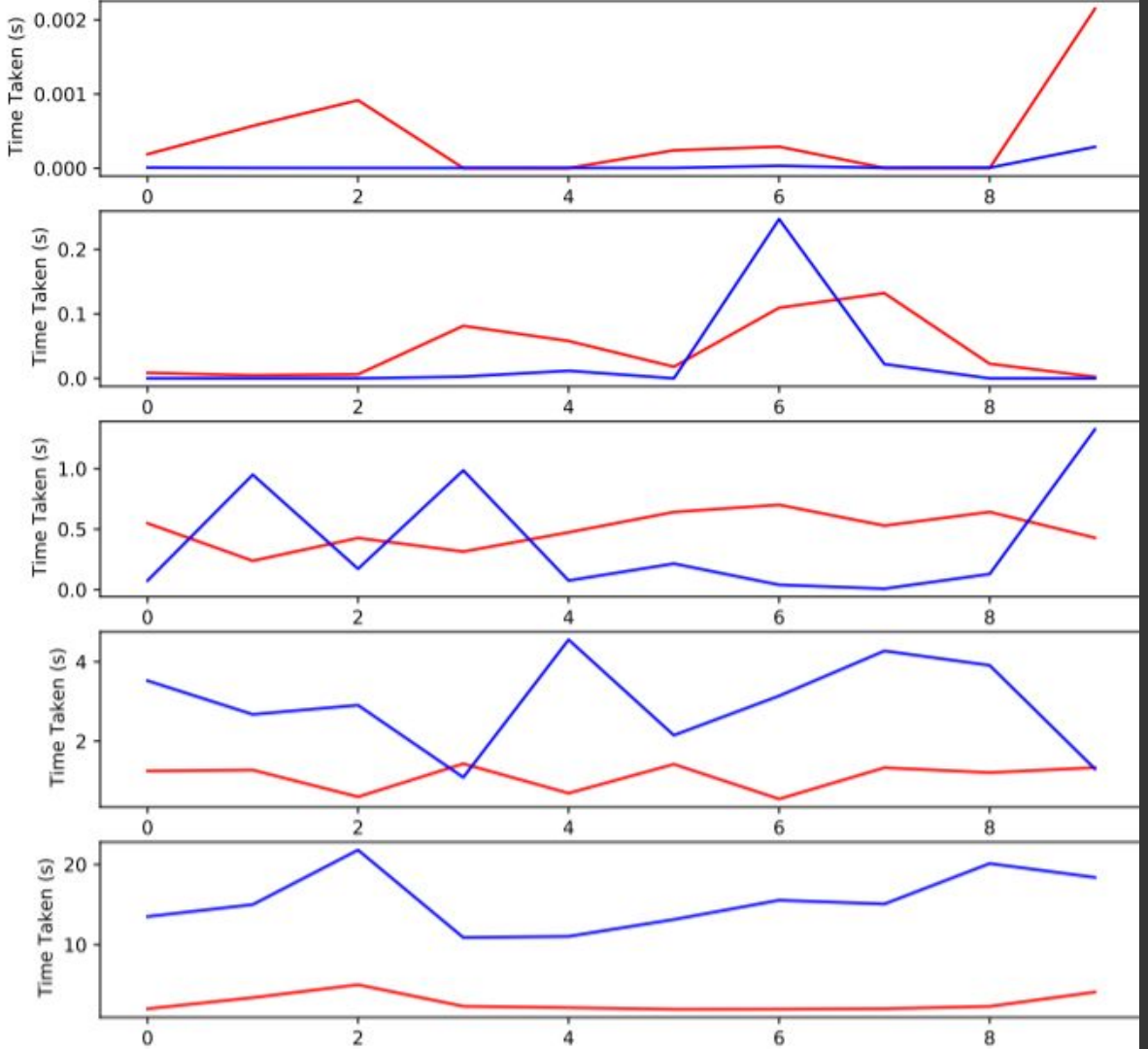
Optimization of Apriori Algorithm

The intended optimization to the MI - Apriori Algorithm is performed by modifying the joinCk method *(combines two sequences from C_{k-1} to obtain a candidate for C_k)*.

The approach is to combine the pruning and candidate generation steps, i.e, as each candidate c_i is generated, the minimum support requirement is checked immediately. Only if this requirement is satisfied, the candidate c_i is stored.

The potential candidate list (C_k) is usually pruned to obtain the list of candidates which satisfy the minimum support requirement. The memory used by this step is of the order $O(|C_k| + |L_k|)$. By combining the candidate generation with the pruning step, this space complexity will go down to $O(|L_k|)$.

The resulting reduction in loop iterations and the number of loads and stores consequently leads to an improvement in the runtime of the algorithm as well.



Graph Description

The graphs have each been plotted with varying sequence lengths. The lengths of the sequences vary as (1,10), (10,20), (20,30), (30,40) and (40,50). The red line denotes the MI-Apriori Algorithm that combines pruning and candidate generation and the blue line denotes the original implementation.

Inference

From the graph shown, we can clearly see that while there is not much change in the runtimes for executing the smaller sequences, there is a drastic difference in the runtime as the sequence length increases. The difference in runtime is tangible just from the small datasets it has been run on.

Optimization of PrefixSpan Algorithm

The intended optimization to the MI - PrefixSpan Algorithm is performed on the time interval table generation step *(generates all the time interval possibilities for expanding the prefix at the time)*. As all the time interval possibilities are being generated in the table, there is no possibility for error in generating MTIS patterns.

However, the memory utilization of the table generation step is of the order of $O(|T|^{(k-1)} * d)$, where |T| denotes the cardinality of the time interval list *(|T|=4 for the sample time list)*. Here, 'k' denotes the length of the MTIS pattern (new_prefix) we're trying to obtain and 'd' is the number of 1-frequent items.

In terms of scalability, the time complexity doesn't fare any better. In addition to the exponentially increasing table generation step, the support of each such generated pattern has to be computed.

Optimization of PrefixSpan Algorithm

The time complexity of the table generation step as well as computing the support is of the order of $O(d * |T|^{(k-1)} * s)$, where d is the number of distinct frequent items in the projected database and s is the time complexity for calculating the support.

The alternative proposed is, to simply iterate through the projected database and produce a frequency map for each item in the projected database. Using this frequency map, calculating the support is trivial and does not take any additional time.

The worst case time complexity and space complexity is $O(L)$ where L denotes the number of items in the projected database for a given prefix. As the length of the pattern increases, this could be a better alternative to the exponential time complexity of the table generation step.

Prefix: $\alpha' = \langle a, (ti_1), e \rangle$.

$[20,(2,6),(7,8)]$: $\langle (c,14),(d,15) \rangle$.
$[30,(1,3),(8,11)]$: $\langle (d,12),(b,13),(c,13),(c,16) \rangle$.

Table	c	d
(ti_0, ti_0)		
(ti_0, ti_1)		
(ti_0, ti_2)		
(ti_0, ti_3)		
(ti_1, ti_0)		
(ti_1, ti_1)		
(ti_1, ti_2)		
(ti_1, ti_3)		
(ti_2, ti_0)		
(ti_2, ti_1)		
(ti_2, ti_2)		
(ti_2, ti_3)		
(ti_3, ti_0)		
(ti_3, ti_1)		
(ti_3, ti_2)		
(ti_3, ti_3)		

Table	c	d
(ti_0, ti_0)		
(ti_0, ti_1)	—	—
(ti_0, ti_2)	—	—
(ti_0, ti_3)	—	—
(ti_1, ti_0)		
(ti_1, ti_1)		
(ti_1, ti_2)	—	—
(ti_1, ti_3)	—	—
(ti_2, ti_0)		
(ti_2, ti_1)		
(ti_2, ti_2)		
(ti_2, ti_3)		
(ti_3, ti_0)	—	—
(ti_3, ti_1)		
(ti_3, ti_2)		
(ti_3, ti_3)		

Table	c	d
(ti_0, ti_0)	—	—
(ti_0, ti_1)	—	—
(ti_0, ti_2)	—	—
(ti_0, ti_3)	—	—
(ti_1, ti_0)		
(ti_1, ti_1)		
(ti_1, ti_2)	—	—
(ti_1, ti_3)	—	—
(ti_2, ti_0)	—	—
(ti_2, ti_1)		
(ti_2, ti_2)		
(ti_2, ti_3)		
(ti_3, ti_0)	—	—
(ti_3, ti_1)	—	—
(ti_3, ti_2)		
(ti_3, ti_3)		

Problem Description

On the right is a prefix and it's projected database in the sample dataset. The minimum support is 0.5 ⇒2. Therefore only c and d are candidates for extension of the prefix. |T|=4, k=3, d=2.

Space Complexity=$O(|T|^{(k-1)}*d) = O(32)$

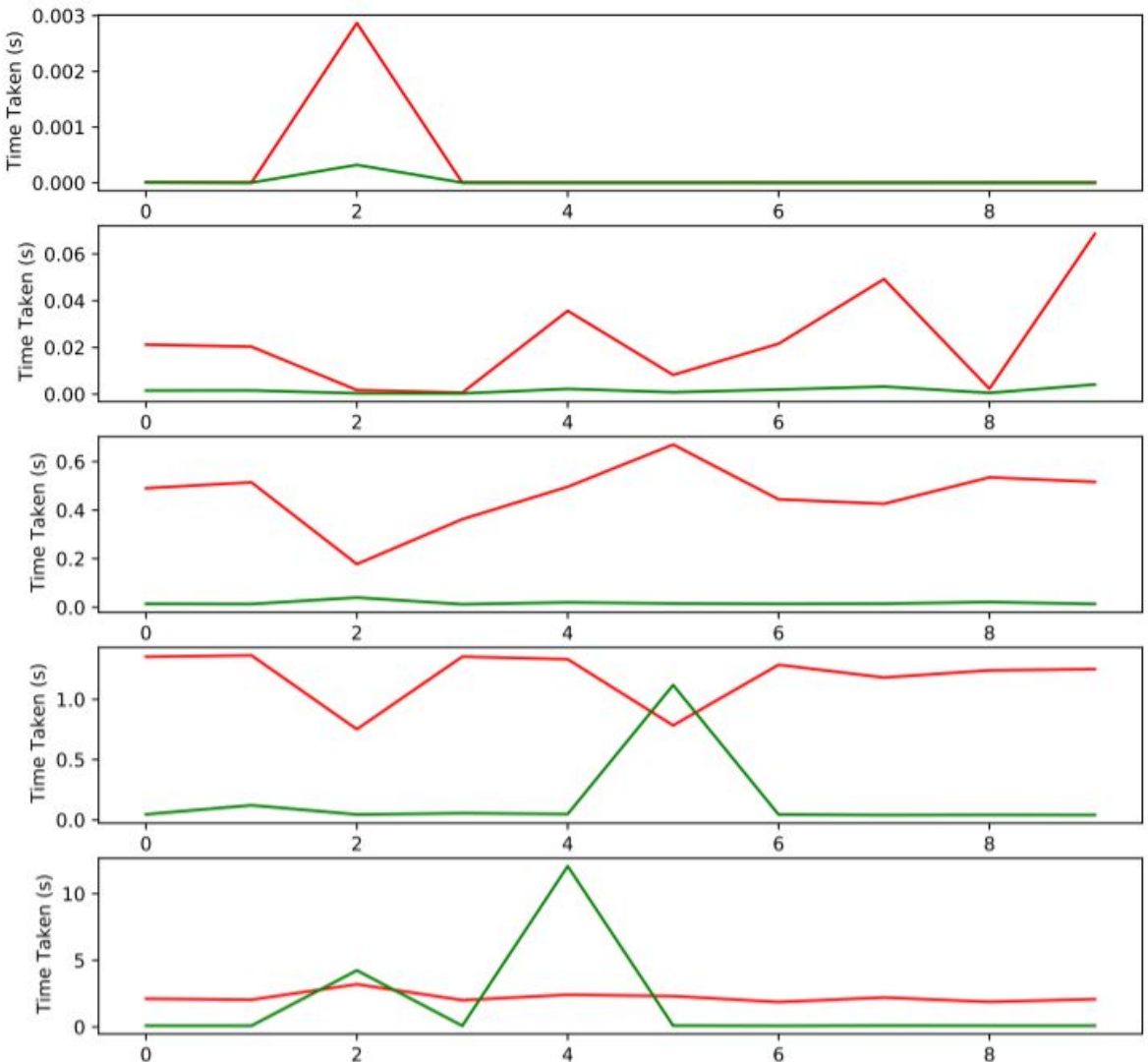
Time Complexity=$O(|T|^{k-1}*d*s) = O(32*s)$

Alternative Approach

Iterate through the projected database while maintaining the frequency map. Only c and d are candidates. L=5.

Space Complexity = O(L) = O(5).

Time Complexity = O(L) = O(5).



Graph Description

The graphs have each been plotted with varying sequence lengths. The lengths of the sequences vary as (1,10), (10,20), (20,30), (30,40) and (40,50). The red line denotes the faster MI-Apriori Algorithm implementation and the green line denotes the MI-PrefixSpan Algorithm implementation.

Inference

From the graph shown, we can clearly see that the prefixspan algorithm consistently outperforms even the optimized apriori algorithm. It even scales well with increasing sequence lengths even though experimental results to the contrary are mentioned in the paper.

Conclusion

We have discussed the two well-known algorithms for mining Multi-Time-Interval Sequential patterns from a database of sorted time-stamped item sequences. The two algorithms MI-Apriori and MI-PrefixSpan have also been optimized to run quicker while utilizing less space. Experimental results show that MI-PrefixSpan Algorithm works best for all kinds of datasets, followed by the optimized MI-Apriori Algorithm. Both algorithms are able to efficiently mine MTIS patterns from the randomly generated datasets.