Paper 528
Tree-Miner: Mining Sequential Patterns from SP-Tree

PAKDD 2020

Authors

Redwan Ahmed Rizvee
Department of Computer Science and Engineering, University of Dhaka, Bangladesh.

Mohammad Fahim Arefin
Department of Computer Science and Engineering, University of Dhaka, Bangladesh.

Dr.Chowdhury Farhan Ahmed
Professor, Department of Computer Science and Engineering, University of Dhaka, Bangladesh.

Our Contributions

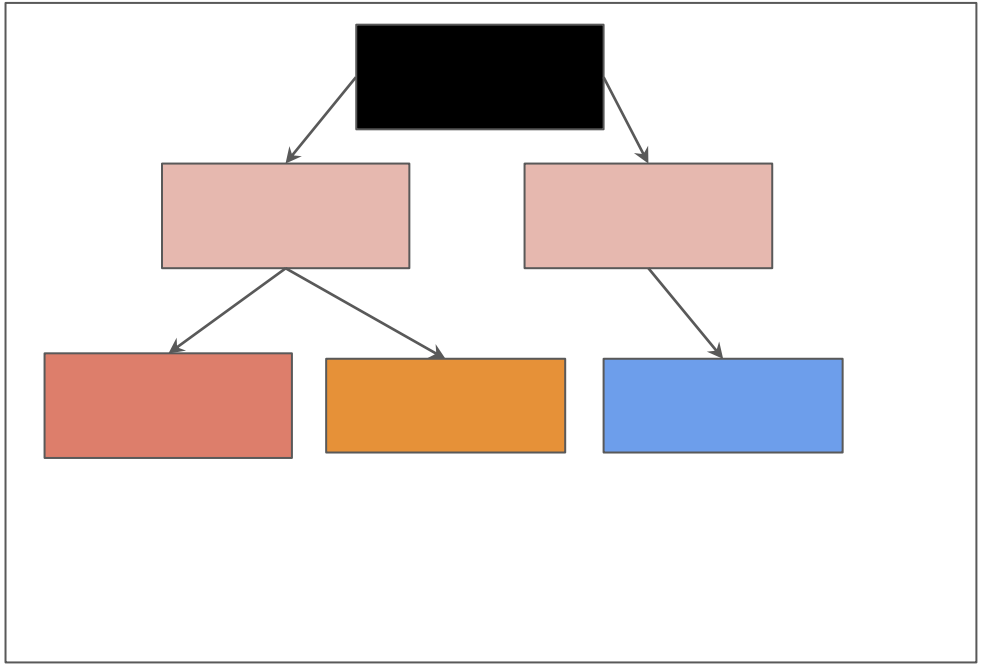
We have three major contributions in this work -

- **SP-Tree:** A Tree alike structure to store the sequential pattern database
- **Tree-Miner:** A mining algorithm to mine the sequential patterns from the SP-Tree.
- **Pruning Mechanisms:** A set of pruning mechanisms to prune the search space during pattern generation.

SP-Tree: Motivation

Why do we want to make a tree alike storing structure?

- A tree alike structure is helpful to track down items/events efficiently
- Specially during **incremental/dynamic database problem** which branch or subtree is modified can give a significant amount of advantage during pattern generation to control the search space.
- Additional advantage in **interactive mining problem** also.



Sample Sequential Database

Each transaction consists of items within events.

a) 4 Transactions
b) Itemset = {a, b, c, d, e, f, g}

Sample Database

SID	Transactions
1	**{a}{abc}{ac}{d}{cf}**
2	**{ad}{c}{bc}{ae}**
3	**{ef}{ab}{df}{c}{b}**
4	**{e}{g}{af}{c}{b}{c}**

Sample Sequential Database: A different interpretation

a) 4 Transactions
b) Itemset = {a, b, c, d, e, f, g}

Item **a** lying in 2nd event.

Item **a** lying in 1st event.

Item **a** lying in 3rd event.

Sample Database

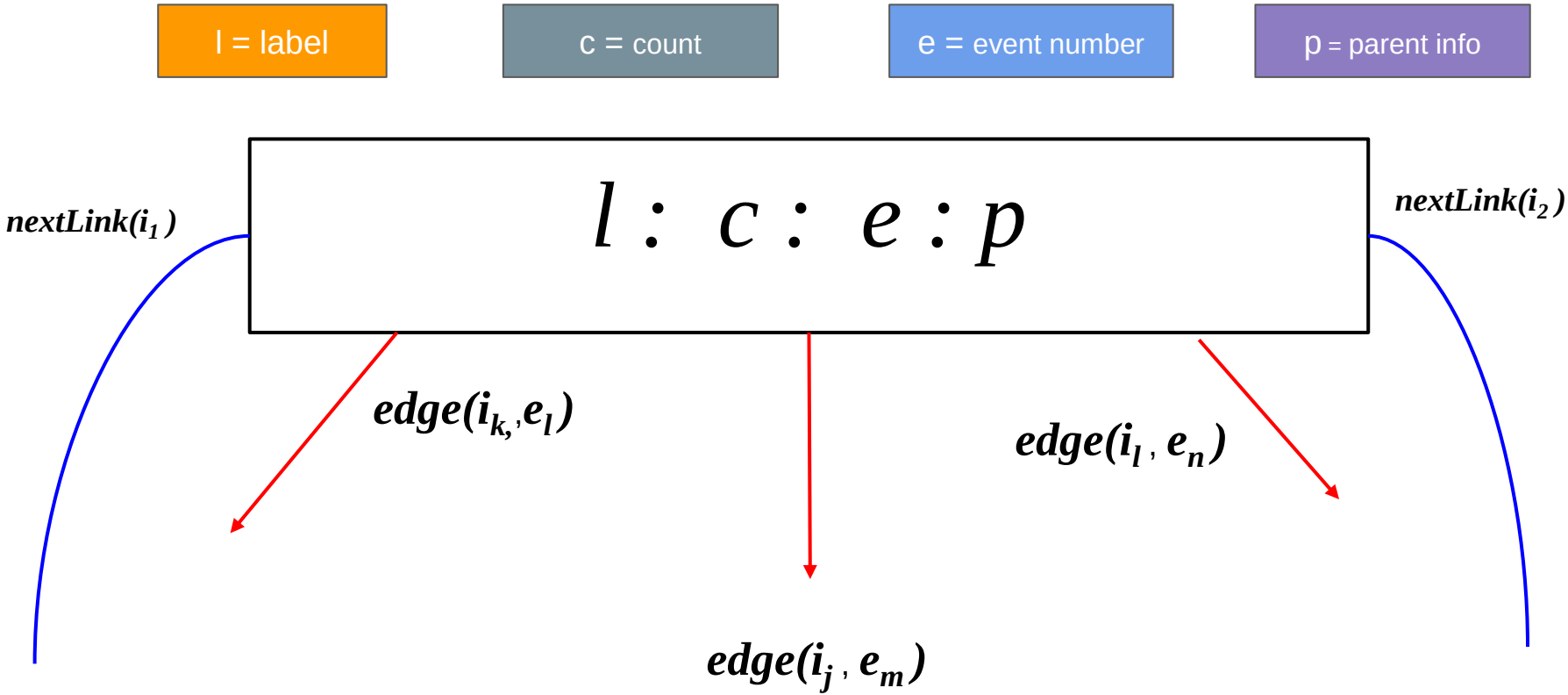
SID	Transactions
1	**{a}{abc}{ac}{d}{cf}**
2	**{ad}{c}{bc}{ae}**
3	**{ef}{ab}{df}{c}{b}**
4	**{e}{g}{af}{c}{b}{c}**

So, each item in each sequence also have a corresponding event number

SP-Tree: Node Structure

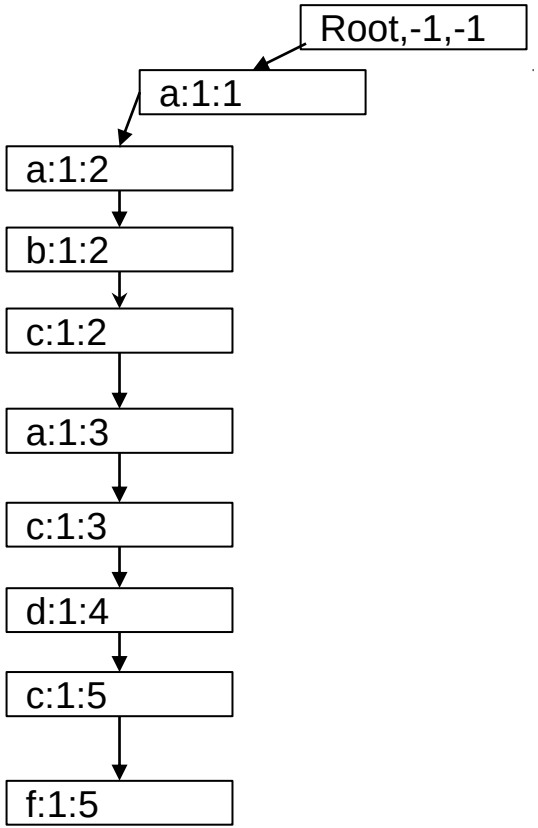
1. **Label**, l : Each node will represent an item and that item is the node's label.
2. **Event Number**, e: Each node will also have an event number which represents the node's label/item's event number.
3. **Count,** c: During prefix sharing the number of times, this node is visited.
4. **Next Link,** $<n_1, n_2, n_3, ..., n_k>$: Next links for an item i, from a node N denotes the first occurrence for that item in the subtrees of N. This is helpful to access the nodes faster to generate the patterns.
5. **Parent Info,** p: Parent Info for an item i, denotes those items which are in the same event as i, in its predecessor nodes. For example, in transaction **{a}{abc}{ac}{d}{cf}**, second event contains 3 items **a**, **b** and **c**. item **c**'s parent info will contain **<a,b>**. This information will be saved using bitset. So, if **a** is indexed as 0 and **b** as 1, then for **c**, p will be 11 [setting 1 in 0^{th} and 1^{st} bit]. This information is helpful during pattern generation and bitset representation provides significant advantage by giving scope to perform bitset operations.
6. **Child edges**$<c_1, c_2, c_3, ..., c_k>$**:**As it is a tree structure, it may have child nodes (if it is not a leaf node) for an item and event number combination. .

SP-Tree: Node Structure Visualization



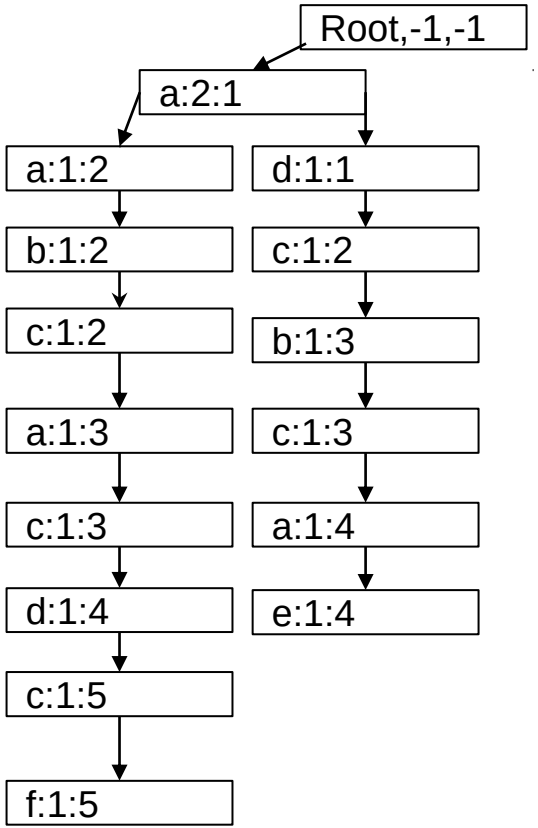
Simulated Tree From The Database(SP-Tree)

Inserted: {a}{abc}{ac}{d}{cf}
[Transaction 1 inserted]



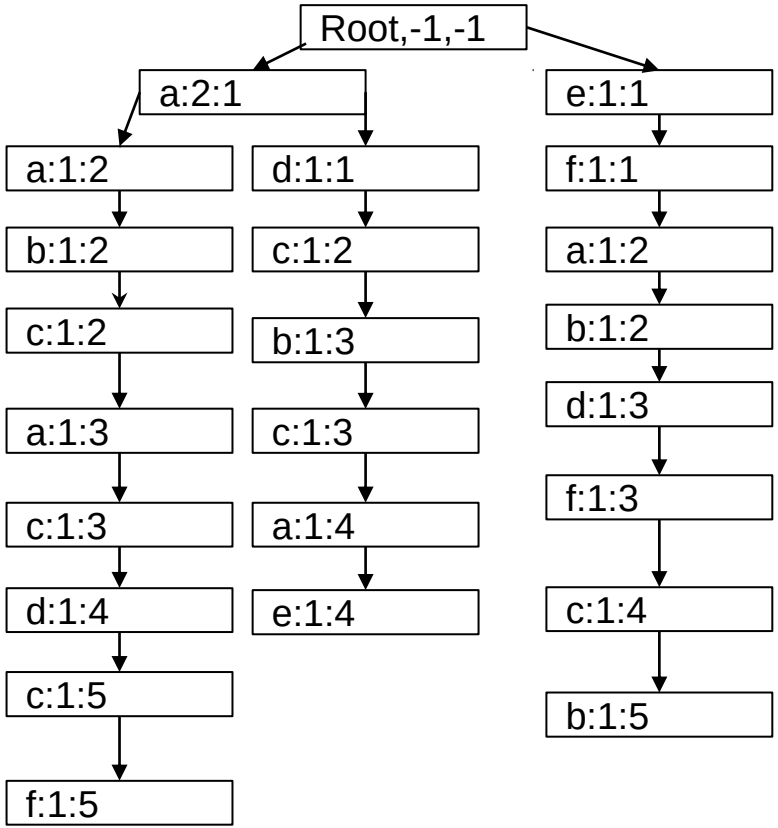
Simulated Tree From The Database(SP-Tree)

Inserted: {ad}{c}{bc}{ae}
[Transaction 2 inserted]



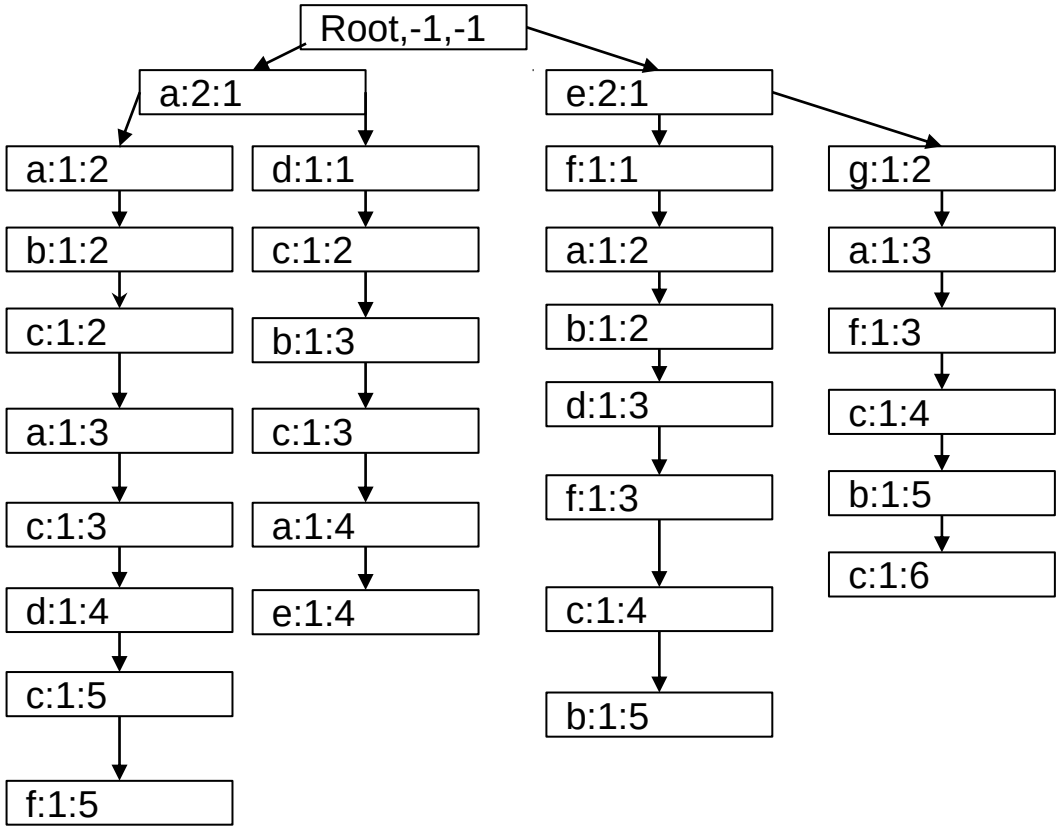
Simulated Tree From The Database(SP-Tree)

Inserted:{ef}{ab}{df}{c}{b}
[Transaction 3 inserted]



Simulated Tree From The Database(SP-Tree)

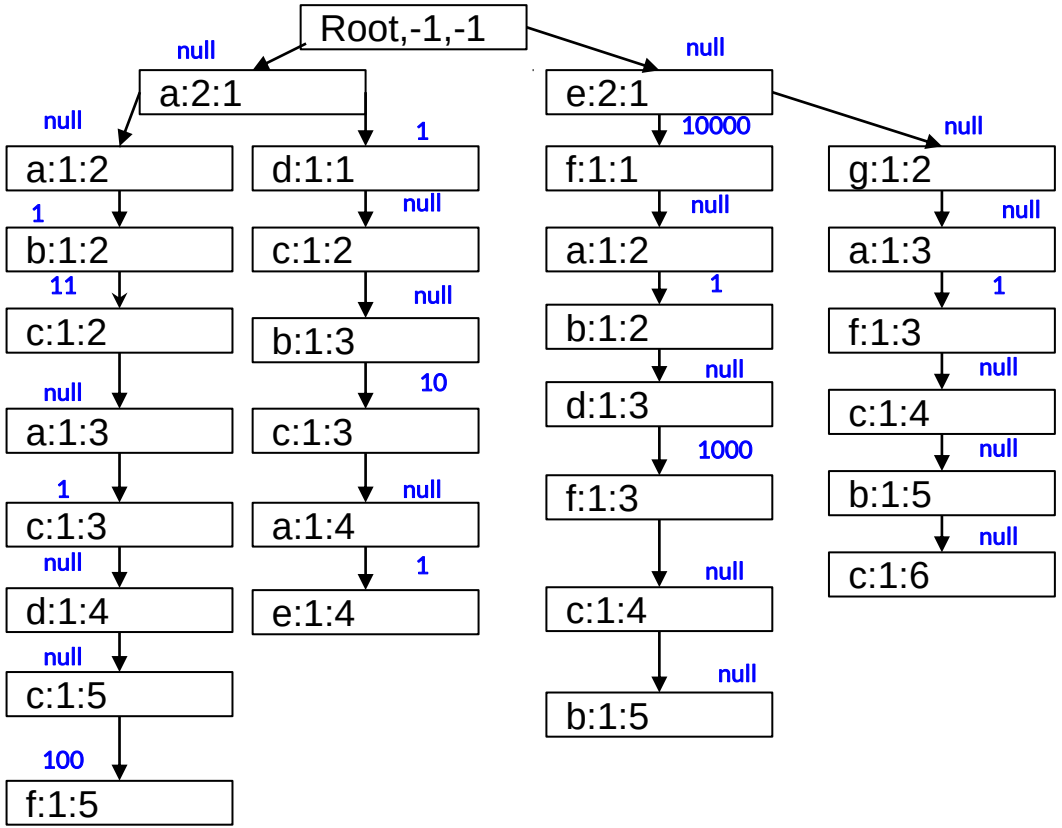
Inserted:{e}{g}{af}{c}{b}{c}
[Transaction 4 inserted]



Simulated Tree From The Database(SP-Tree)

Coding

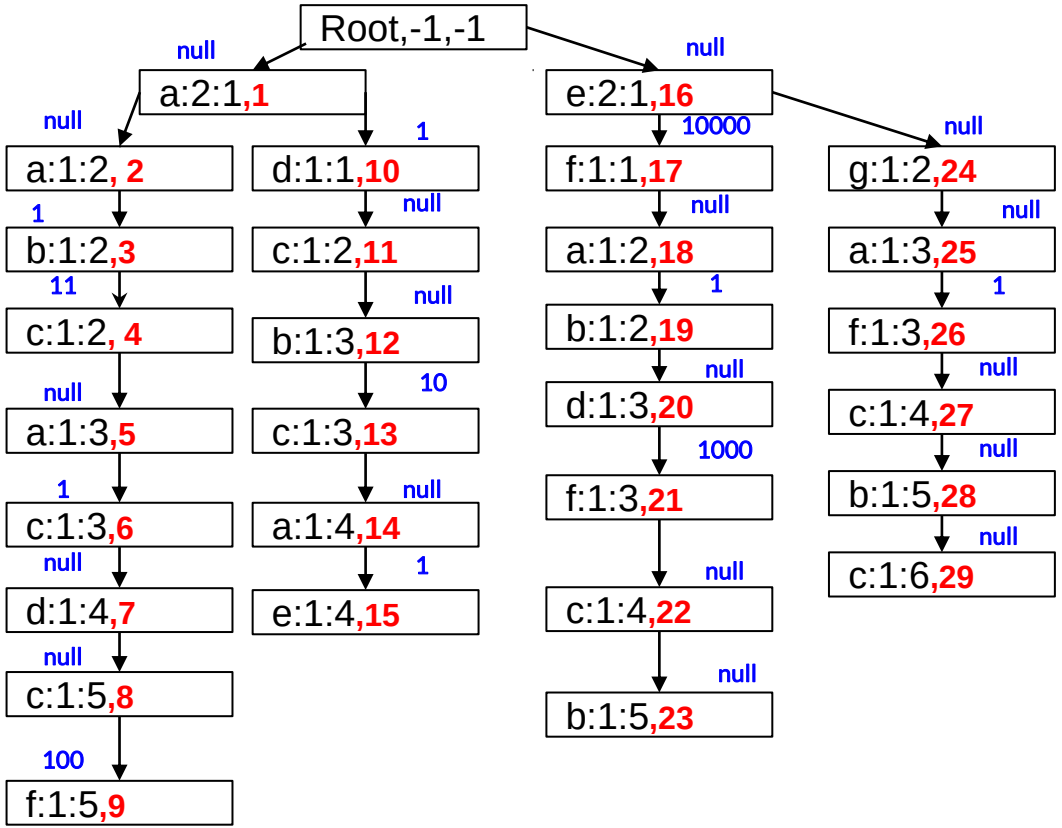
a	0
b	1
c	2
d	3
e	4
f	5
g	6



Simulated Tree From The Database(SP-Tree)

Coding

a	0
b	1
c	2
d	3
e	4
f	5
g	6



For discussion simplicity we have given nodes an id with red color

Definition of *nextLink*:

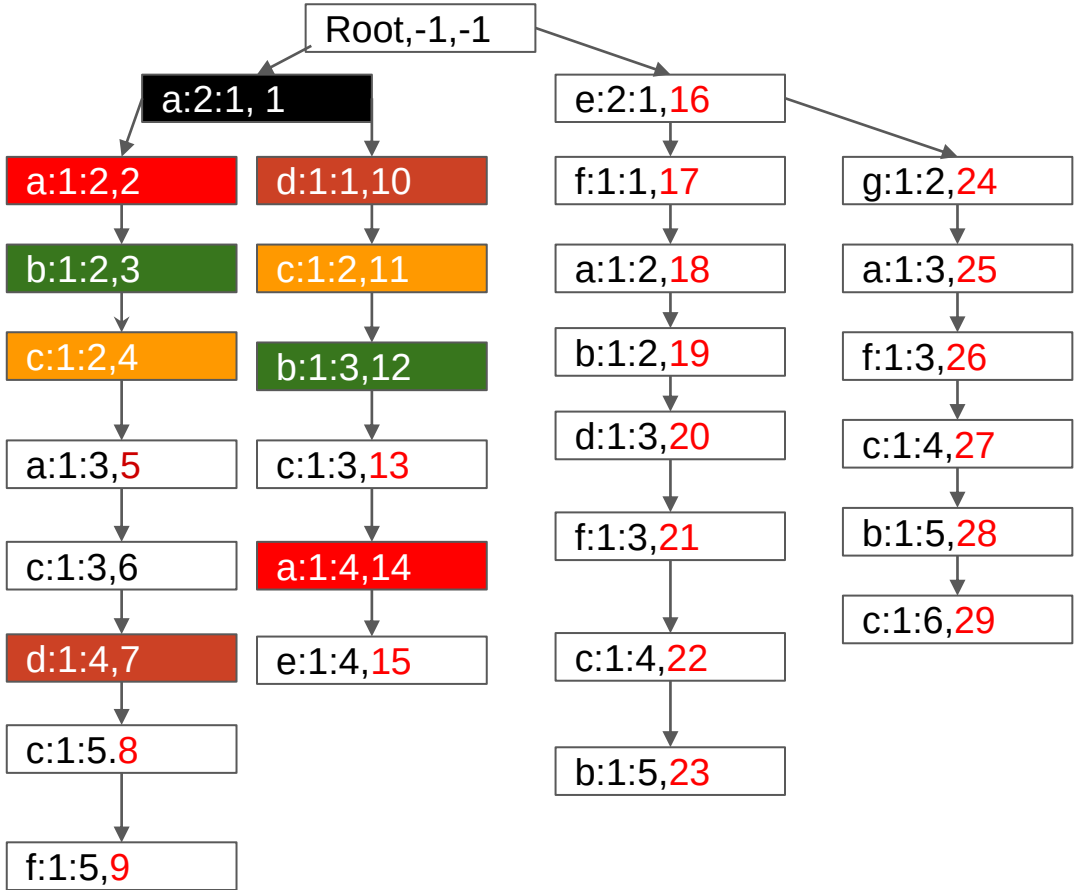
> *"NextLinks for a symbol **α** from current node **A** help us to reach nodes **B** which will expand the pattern for the symbol **α** from **A**."*

Properties of the *nextLink*

1. These links help to expand pattern for a symbol **(α)** from a node**(A)** by reaching node**(B)** [**B**'s label is **α** and thus why link was created from **A** for **α**].
2. These links for a node(**A**) attach only those nodes(**B**) which are in the subtree of (**A**) and first occurrences in different branches.

NextLinks basically help to access nodes faster for a symbol to generate patterns. The nodes directly contribute to the pattern frequency along with pattern generation. Due to having ***NextLinks***, pattern generation gets significantly faster.

nextLinks of node 1



Sym	nextLinks for sym	Color codes
a	2, 14	
b	3, 12	
c	4,11	
d	7,10	

Calculation of NextLink

```
1: function NextLinkGeneration(node)
2:     new_nextLink = {}
3:     for (each child c_i of node) do
4:         child_nextLink = NextLinkGeneration(c_i)
5:         new_nextLink = new_nextLink ∪ child_nextLink
6:     new_nextLink[node.l] = node /*update nextLink for node's label*/
7:     return new_nextLink
```

We can efficiently calculate nextLinks through recursive function calling.

Mining Patterns from SP-Tree with Tree Miner Algorithm

Mining Patterns from SP-Tree with Tree Miner Algorithm

The following concepts will be discussed:

1) **Pattern Formation Rules**: How patterns are generated and their frequency is calculated from the tree.
2) **Discussion of two types of pattern extensions**: Sequence Extension and Itemset Extension.
3) **Pruning Mechanisms**: Discussion related to pruning mechanisms to reduce search space.

Tree-Miner: Pattern Formation Rules

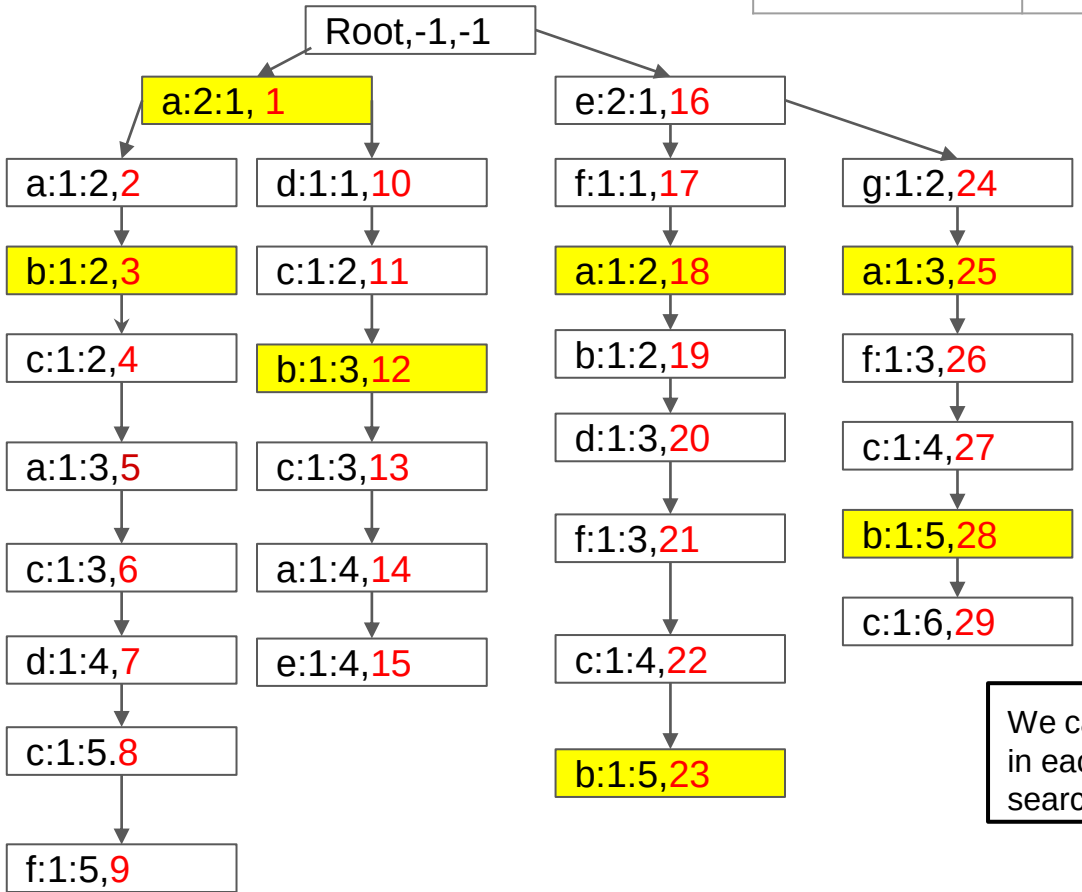
1) Node combinations from SP-Tree makes a pattern and from different subtrees the first node combinations are always chosen.
2) Node’s count attribute’s value denotes the pattern’s frequency.

Pattern: {a}{b}

nodes	frequency
{1,3},{1,12},{18,23} {25,28}	1+1+1+1=4

nodes	frequency
{3},{12},{23}{28}	1+1+1+1=4

Pattern {a}{b} ends at these nodes



Always consider the first occurrence in each disjoint subtree

Last nodes in each occurrence contribute to the pattern's total frequency. So, using these nodes we can denote the pattern's positions in the tree.

Using next links we can reach these nodes faster

We can say that upto these nodes (last nodes in each contribution) we have covered the search space.

Discussion of two types of pattern extensions

There are two kinds of extensions for a pattern.

1) Pattern Extension as Sequence (Sequence Extension). For example -

<(a)(ab)>  <(a)(ab)(c)>

Extending a pattern by adding new event

2) Pattern Extension as Itemset (Itemset Extension).

<(a)(ab)>  <(a)(abc)>

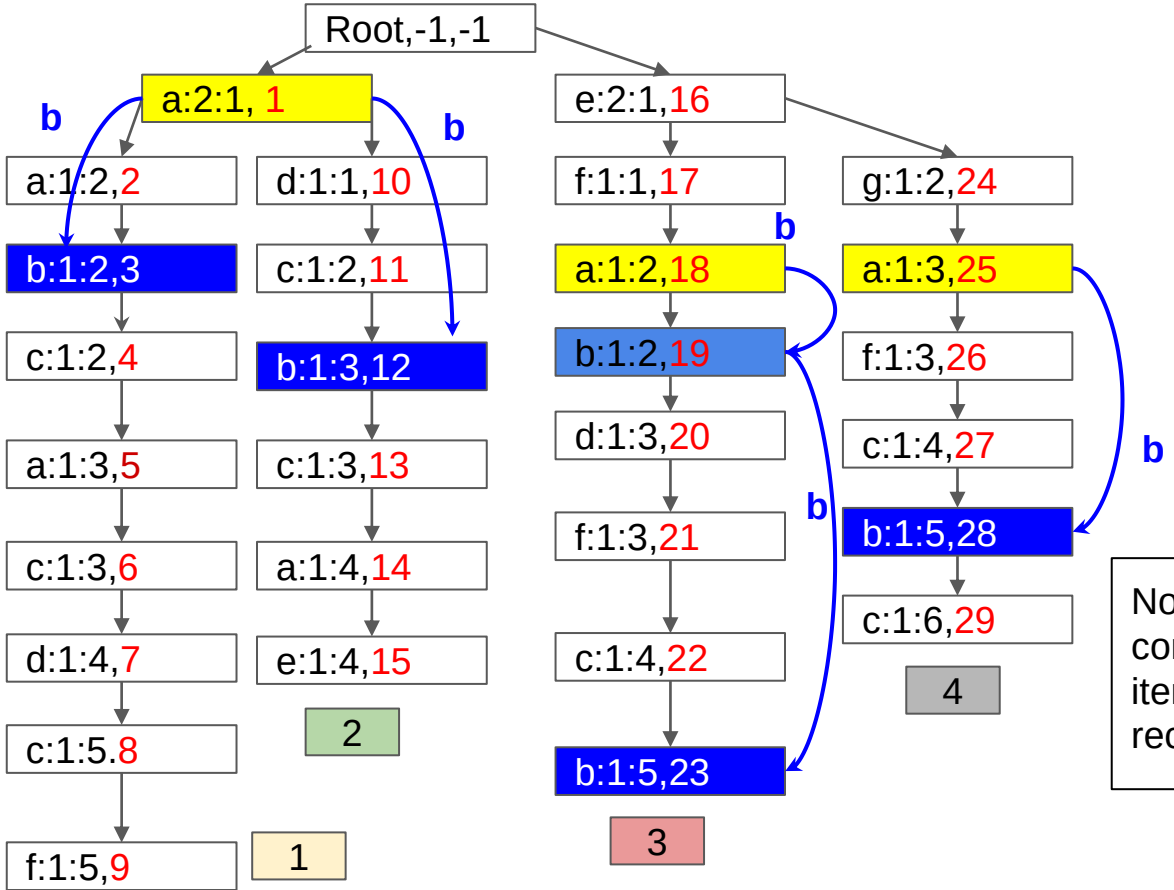
Extending a pattern by adding new item in the last event.

Sequence Extension

To extend a pattern **P** from a node **A** for a symbol **α** as sequence extension(<P(α)>), we need to do the following -

1. Get the first occurrence of the nodes for the symbol **α** from the current node **A** in its underlying disjoint subtrees. Those nodes can be found through traversing by nextLink for symbol **α** from node **A**.
2. If the nodes reached through nextLink provides nodes in the same event/itemset, we need to go deeper in the subtree(through recursive next link moves) to find nodes not belonging to the same itemset. [*may need to go maximum 2 depth*]

Sequence Extension Example: {a} -> {a}{b}



pattern	nodes	frequency
{a}	{1},{18},{25}	2+1+1=4
{a}{b}	3,12,23,28	4

1) 1 -> **3** [{a}{b} found in path 1]
2) 1 -> **12** [{a}{b} found in path 2]
3) 18 -> 19 -> **23** [{a}{b} found in path 3]
4) 25 -> **28** [{a}{b} found in path 4]

Node 18 & 19 could not be directly concatenated because they were in same itemset, so we went deeper through recursive next links.

Sequence Extension: Pseudocode

```
1: function SequenceExtension(node_list, sym)
2:     new_nodes = {}, act_freq = 0, over_freq = 0
3:     for (each node n_i ∈ node_list) do
4:         for (each node n_j ∈ n_i.nextLink[sym]) do
5:             over_freq = over_freq + n_j.count
6:             if (n_j and n_i are not in same event) then
7:                 new_nodes = new_nodes ∪ n_j
8:                 act_freq = act_freq + n_j.count
9:             else
10:                for (each node n_k ∈ n_j.nextLink[sym]) do
11:                    new_nodes = new_nodes ∪ n_k
12:                    act_freq = act_freq + n_k.count
13:    Return new_nodes, act_freq, over_freq
```

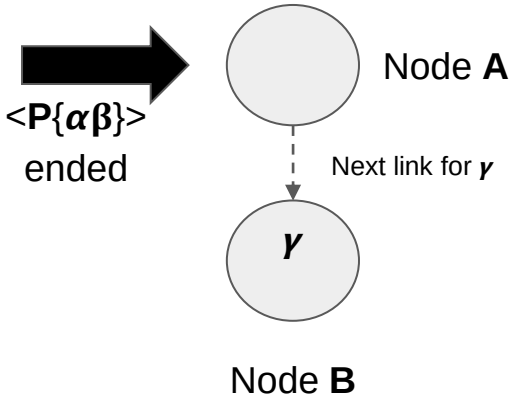
Itemset Extension

To extend a pattern<**P**{$\boldsymbol{\alpha\beta}$}> from a node **A** for a symbol $\boldsymbol{\gamma}$(<**P**{$\boldsymbol{\alpha\beta}$}> ⇨ <**P**{$\boldsymbol{\alpha\beta\gamma}$}>) as itemset extension, we need to do the following -

1. Get the first occurrence of the nodes for the symbol $\boldsymbol{\gamma}$ from the current node **A** in its underlying disjoint subtrees. Those nodes can be found through traversing by nextLink for symbol $\boldsymbol{\gamma}$ from node **A**.
2. Nodes must belong to the same itemset. If they are not then we need to go deeper in the subtree to find such nodes (first occurrence which satisfies all the constraints in disjoint subtrees). Simply to put, we need to find such node **B** in the subtree of **A** with label $\boldsymbol{\gamma}$ which have nodes with label $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ in its predecessor nodes and which belongs to the same event as **B**. To, search such nodes efficiently we will use **Parent Info**'s bitset representation and execute bitwise AND operations.

Itemset Extension: Node connections

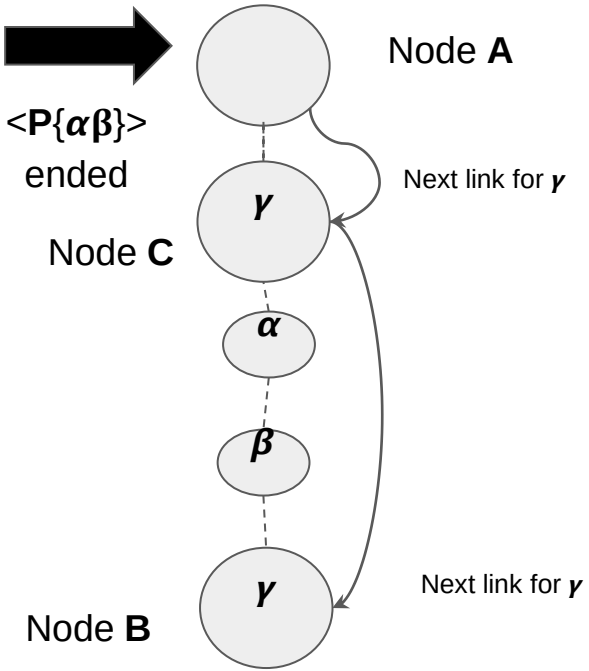
Pattern Expansion: <**P**{***α*β**}> ⇨ <**P**{***α*β*γ***}>



Direct Node Connection:
From node **A**, we reach node **B** with label ***γ*** through next link of **A**. **B** and **A** have same event number, so **A** and **B** can be concatenated({A,B}) to generate pattern <**P**{***α*β*γ***}>

Itemset Extension: Node connections

Pattern Expansion: <**P{$\alpha\beta$}**> ⇨ <**P{$\alpha\beta\gamma$}**>

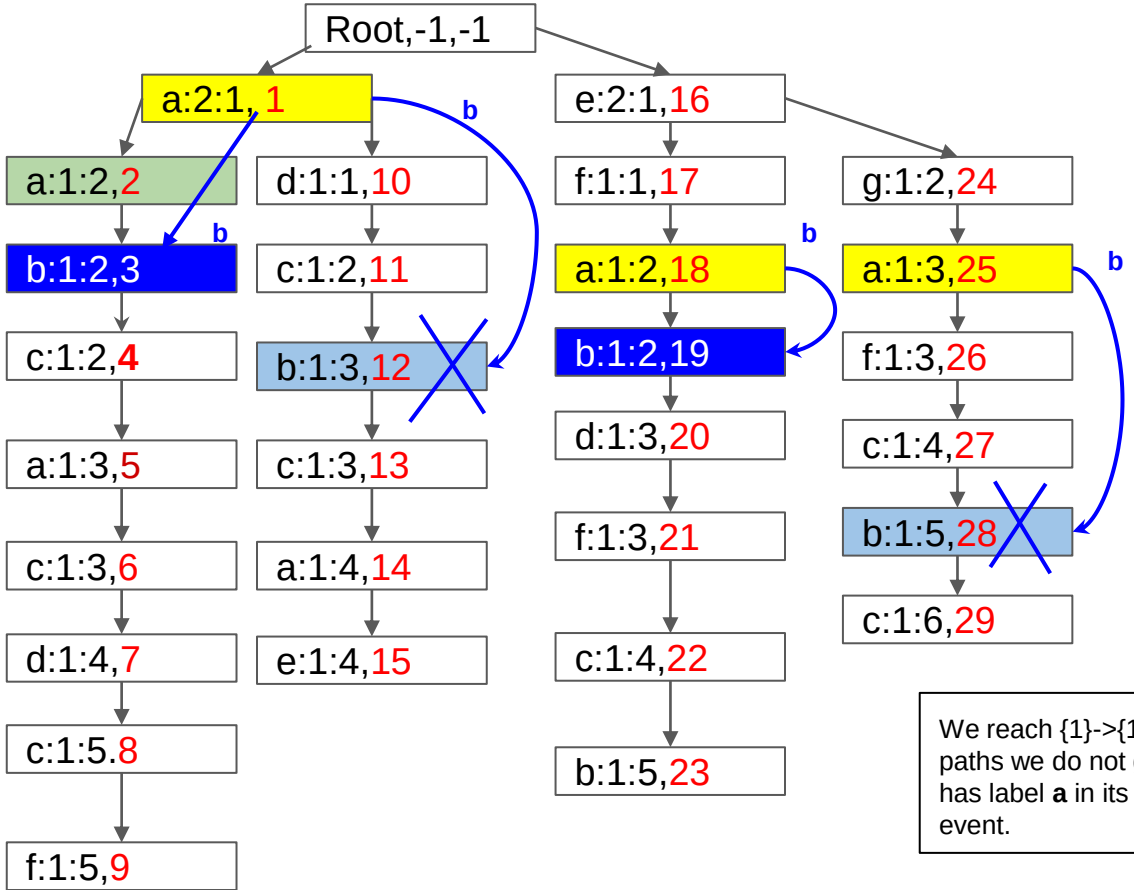


Indirect Node Connection:
From node **A**, we reach node **C** through next link for γ. But they are not in same event(**A**, **C** can not be concatenated). So, in A's subtree through recursive next links (one or more) we look for a node **B** which has α and β in its predecessor nodes and are in the same event as **B**. Then we can say that <**P{$\alpha\beta\gamma$}**> Is formed by reaching node **B** from **A**.

Parent Info comes into play here. Suppose we index as, α=0 and β=1. So, we can create a bitset of SET=11. Now, we can perform bitset AND operation between SET and parent Info of **B**. After AND operation if we see that we have 1 in both 0th and 1st position then, we can say that **B** has satisfied the constraint and can make the pattern.

We have used dynamic bitset based representation to make the performance better through bit operations. But, other array based or hashmap based representations will also work. For a node we basically have to check some concerned items only to find if they exist in the same event or not.

Itemset Extension Example: <{a}> -> <{ab}>



pattern	nodes	frequency
{a}	{1},{18},{25}	2+1+1=4
{ab}	{2,3}->{3} {18.19} -> {19}	1+1=2

Direct Node Connection: {18,19} forms pattern {ab} [both have same event number]

Indirect Node Connection: We reach node 3 from node 1 through next link. But their event number is not same. So {1,3} can not be concatenated.
But node 3(parent info bitset 1) has event a in its predecessor node (node 2) which is in the same event as it. So, {2,3} can be concatenated and we will get {2,3} -> {3}

We reach {1}->{12} and {25}->{28}, but in these paths we do not get such node with label **b** which has label **a** in its predecessor nodes with same event.

Itemset Extension: Pseudocode

```
1: function ITEMSETEXTENSION(node_list, sym, last_event_bitset)
2:     Q = {}, act_freq = 0, new_nodes = {}
3:     for (each node n_i ∈ node_list) do
4:         Q = Q ∪ {n_i, 0}, act_freq = act_freq + n_i.count
5:     while (Q is not empty) do
6:         if (act_freq <= min_sup) then Return null, −1
7:         {u, level} = Q.front()
8:         Q = Q − {u, level}
9:         bitset = 0
10:        if level > 0 then
11:            bitset = u.parentInfo AND last_event_bitset
12:        if (bitset == last_event_bitset) then new_nodes = new_nodes ∪ u
13:        else
14:            act_freq = act_freq − u.count
15:            for (each node n_i ∈ u.nextLink[sym]) do
16:                Q = Q ∪ {n_i, level + 1}
17:                act_freq = act_freq + n_i.count
18:    Return new_nodes, act_freq
```

Recursive Pattern Generation

With recursive extension ***sList*** and ***iList*** will prune.

$$P,\ S=\{.., I_j, I_k, I_l, ...\},\ I=\{.., I_a, I_b, I_c, .. \}$$

$$P`,\ S`=\{.., I_k, I_l, ...\},\ I`=\{.., I_b, I_c, .. \} \mid S` \subseteq S, I` \subseteq I$$

$$P``,\ S``=\{.., I_l, ...\},\ I``=\{.., I_c, .. \} \mid S`` \subseteq S`, I`` \subseteq I`$$

S = symbols which will perform **Sequence Extension** on a pattern

I = symbols which will perform **Itemset Extension** on a pattern

sList ***iList***

Pruning Mechanisms

Main goal is to reduce and compact ***sList*** and ***iList*** as much as possible.

1. **Co-Existing Item Table Based Pruning**: A data structure, Co-Existing Item Table is used to get an idea which symbols may extend a pattern as sequence extension and itemset extension during ***sList*** and ***iList*** generation.
2. **sList and iList pruning**: During ***sList*** and ***iList*** formation considering their mutual dependencies a pruning can be performed to reduce ***sList*** and ***iList*** size.
3. **Heuristic iList pruning**: During calculation of ***sList*** intersecting symbol's (the symbol which lies in both ***sList*** and ***iList***) can be pruned from ***iList*** based on some heuristic support/frequency value.
4. **Level wise frequency based iList Pruning**: During support calculation for an item ***i*** as itemset extension based on level wise approximate frequency, pruning decision can be taken.

Co-Existing Item Table Based Pruning

SID	Transactions
1	**{a}{abc}{ac}{d}{cf}**
2	**{ad}{c}{bc}{ae}**
3	**{ef}{ab}{df}{c}{b}**
4	**{e}{g}{af}{c}{b}{c}**

Item	Sequence Extending Symbols[S-column]	Itemset extending symbols[I-column]
a	{a:2}, {b:4}, {c:4}, {d:2}, {e:1}, {f:2}	{b:2}, {c:1}, {d:1},{ e:1},{f:1}
b	{a:2}, {b:1}, {c:3}, {d:2}, {e:1},{ f:2}	{c:2}
c	{a:2}, {b:3}, {c:3}, {d:1}, {e:1}, {f:1}	
d	{a:1}, {b:2}, {c:3}, {e:1},{ f:1}	{f:1}
e	{a:2}, {b:2}, {c:2}, {d:1}, {f:2}	{f:1}
g	{a:1}, {b:1}, {c:1}, {f:1}	

P{ab}**X** => {X must belong to the S-column for both a and b and satisfy threshold constraints}

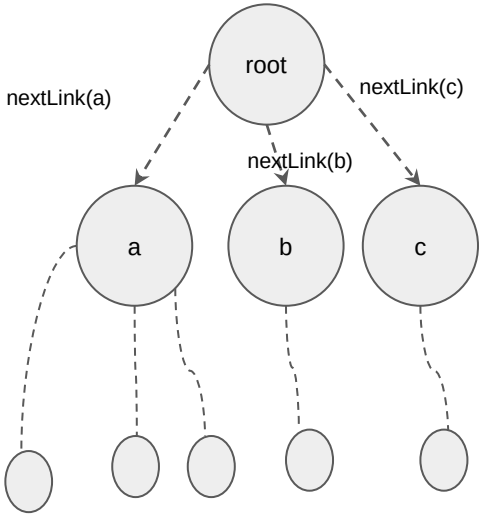
P{ab**X**} => {X must belong to the I-column for both a and b and satisfy threshold constraints}

We have adopted this pruning mechanism to reduce our search space

If X fails to satisfy then X can not perform extension.

Fournier-Viger, P., Gomariz, A., Campos, M., Thomas, R. (2014). Fast Vertical Sequential Pattern Mining Using Co-occurrence Information. Proc. 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2014) Part 1, Springer, LNAI, 8443. pp. 40-52.

S-Column Calculation: CMAPs



Using next links we can calculate CMAPs efficiently

We may need to go maximum 2 depth through links

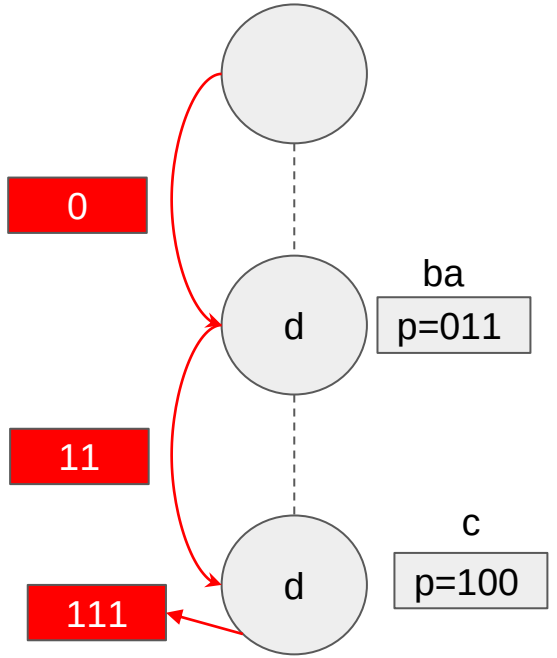
We use event numbers of the nodes to take decisions

```
1: function CalculateCMAPs(void)
2:     CMAP_s = {}
3:     for (each symbol s_i ∈ root.nextLink) do
4:         for (each node u_j ∈ root.nextLink[s_i]) do
5:             for (each symbol s_k ∈ u_j.nextLink) do
6:                 for (each node v_l ∈ u_j.nextLink[s_k]) do
7:                     if (v_l and u_j are not in same event) then
8:                         CMAP_s[s_i][s_k] = CMAP_s[s_i][s_k] + v_l.count
9:                     else
10:                        for (each node w_m ∈ v_l.nextLink[s_k]) do
11:                            CMAP_s[s_i][s_k] = CMAP_s[s_i][s_k] + w_m.count
```

I-Column Calculation: CMAPi

```
1: function RecursiveUpdateCMAPi(node, sym, found_parent_items)
2:     bitset = found_parent_items AND node.parentInfo
3:     if (bitset is not zero) then
4:         for (each set bit b_i ∈ bitset) do
5:             s =GetSymbol(b_i)
6:             CMAP_i[sym][s] = CMAP_i[sym][s] + node.count
7:     new_bitset = found_parent_items OR bitset
8:     if (AllLexSmallFnd(found_parent_items, sym)) then Return
9:     for (each node c_k ∈ node.nextLink[sym]) do
10:        RecursiveUpdateCMAPi(c_k, sym, new_bitset)
11: function CalculateCMAPi(void)
12:    CMAP_i = {}
13:    for (each symbol s_i ∈ root.nextLink) do
14:        for (each node n_j ∈ root.nextLink[s_i]) do
15:            found_parent_items = 0
16:            RecursiveUpdateCMAPi(n_j, s_i, found_parent_items)
```

We have used ***parentInfo's*** bitset representation and ***nextLink*** to calculate the table efficiently.



To calculate ***CMAPi*** for a symbol ***S,*** we use recursive next links to reach nodes and update the items which can be appended to it to perform itemset extension and fill ***CMAPi*** table.

sList and *iList* Pruning

sList*, *iList

sList = {A, B, C, D}
iList = { A, B, D}
P

After frequency Measure through SP-Tree

s`List*, *i`List

sList = {A, B, D}
iList = { A, B}

These satisfy support threshold and can extend **P**

For each item $j \in$ ***s`List*** can extend **P** as sequence extension. Ex. **P{*j*}**

For each item $j \in$ ***i`List*** can extend **P** as itemset extension. Ex. **{p*j*}.**

$\mathbf{k_j}$ = lexicographically smaller than j in the concerned item list.

For Recursive extensions:

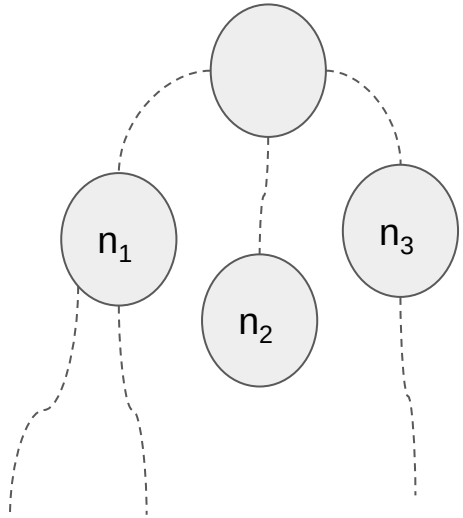
Iteration	New Pattern	New *sList*	New *iList*
For each item $j \in$ ***s`List***	P{*j*}	***s`List***	***s`List*** - $\{j \cup k_j\}$
For each item $j \in$ ***i`List***	{p*j*}	***s`List***	***i`List*** - $\{j \cup k_j\}$

This is a very popular pruning technique and have been mentioned in various sequential pattern mining literature. We also have adopted this pruning in our algorithm to reduce search space.

Heuristic iList Pruning

a) If a symbol ***i*** lies in both ***sList*** and ***iList***, then this pruning is applicable
b) During frequency calculation for such ***i*** as sequence extension we may get a over-estimated pattern frequency, ***F*** by taking the first nodes' count attribute's value reached through the first level of next links in different branches.
c) If ***F*** fails to satisfy support-threshold then ***i*** can directly be pruned without any support calculation from ***iList***.

This value ***F*** is an overestimated frequency for pattern **{p*i*}** and always actual support of **{p*i*}** <= ***F***.

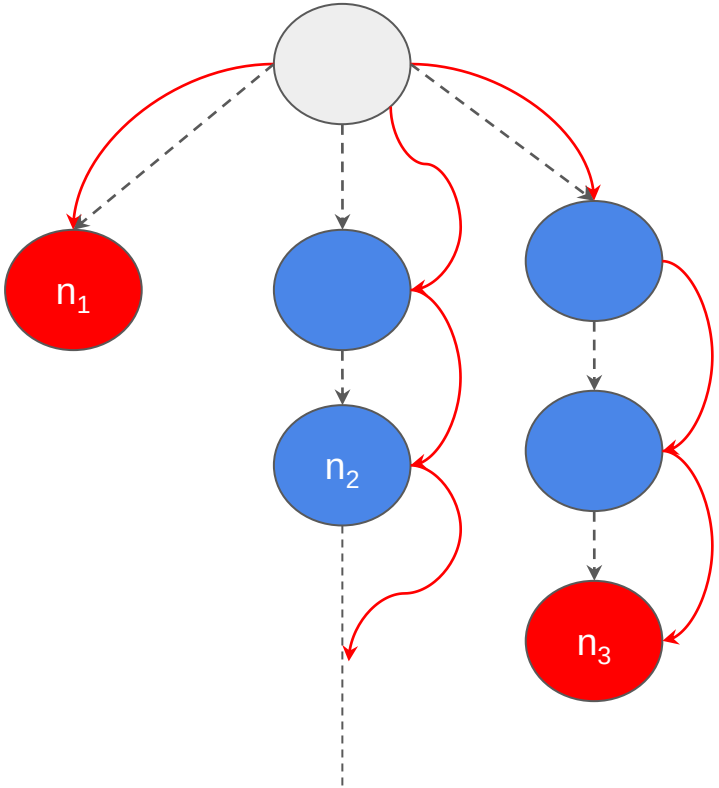


$F = n_1.count + n_2.count + n_3.count$

Level wise frequency based iList Pruning

During support calculation for an item ***i*** as itemset extension with a pattern ***P***, we can reduce the search space based on current level wise frequency. If that frequency fails to satisfy support threshold, then remaining search is completely unnecessary. We have embedded this logic in **ItemsetExtension** function.

	No need to go deeper in subtree, node is already found
	Need to go deeper in subtree to get the actual node.



We may need to go more deeper to get the actual nodes which will extend ***P*** as ***{Pi}***.

But based on current frequency(not actual, approximate max) we can stop searching if it fails to satisfy support threshold.

If (***n_1.count*** +***n_2.count*** + ***n_3.count***)<=support threshold) We do not need to search more. Because this is the maximum support can be gained, if we go deeper the support may stay equal or gets reduced.

Tree-Miner: Merging all the pruning and mining pseudocode

```
1: function PatternExtension(P, node_list, sList, iList, last_event_bitset)
2:     /*Co Existing Item Table Based Pruning*/
3:     for (each item i ∈ sList) do
4:         for (each set bit b_j ∈ last_event_bitset) do
5:             sym =GetSymbol(b_j)
6:             if (CMAP_s[sym][i] < min_sup) then sList = sList − {i}
7:     for (each item k ∈ iList) do
8:         for (each set bit b_j ∈ last_event_bitset) do
9:             sym =GetSymbol(b_j)
10:            if (CMAP_i[sym][k] < min_sup) then iList = iList − {k}
11:    S_nodes = {}, I_nodes = {}
12:    /*Support count and Heuristic iList Pruning*/
13:    for (each item i ∈ sList) do
14:        new_nodes, act_freq, over_freq =SequenceExtension(node_list, i)
15:        if (act_freq >= min_sup) then S_nodes[i] = new_nodes
16:        else sList = sList − {i}
17:        if (over_freq < min_sup and i ∈ iList) then iList = iList − {i}
18:    for (each item i ∈ iList) do
19:        new_nodes, act_freq =ItemsetExtension(node_list, i, last_event_bitset)
20:        if (act_freq >= min_sup) then I_nodes[i] = new_nodes
21:        else iList = iList − {i}
22:    used = {}
23:    for (each item i ∈ sList) do /*Lexicographic reverse sorted iteration*/
24:        iList' = used, idx =GetIndex(i)
25:        bitset =Create bitset setting 1 in idx^th position
26:        PatternExtesion(P{i}, S_nodes[i], sList, iList', bitset)
27:        used = used ∪ {i}
28:    used = {}, bitset = last_event_bitset
29:    for (each item i ∈ iList) do /*Lexicographic reverse sorted iteration*/
30:        idx =GetIndex(i)
31:        bitset =Set bit in idx^th position
32:        PatternExtension({Pi}, I_nodes[i], sList, used, bitset)
33:        used = used ∪ {i}
```

Recursive calling phase

Pruning Phase

Experimental Analysis

We conducted several experiments on a 64 bit machine having intel Core i7-3770 CPU @ 3.40GHz x 8, 8 GB RAM and Linux 16.04 Operating System. All the implementations were in C++ Language.

We measure the performance based on the following criteria.

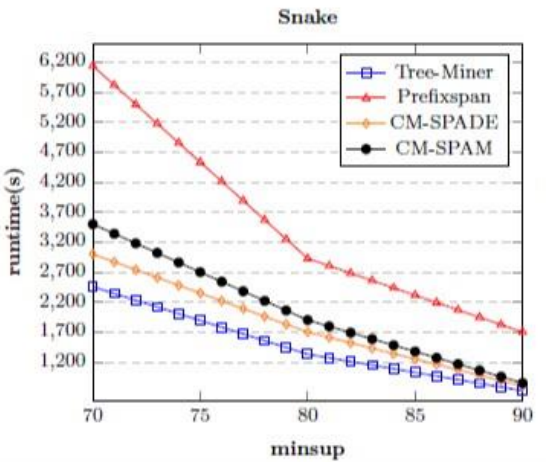
a) Runtime. (Tree-Miner using SP-Tree **Vs** CM-SPAM, CM-SPADE, Prefixspan)
b) Memory Consumption (Tree-Miner using SP-Tree **Vs** CM-SPAM, CM-SPADE, Prefixspan)
c) Structure Construction time.

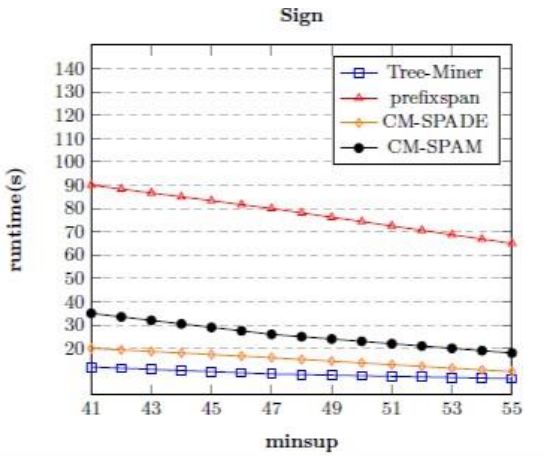
Experimental Analysis

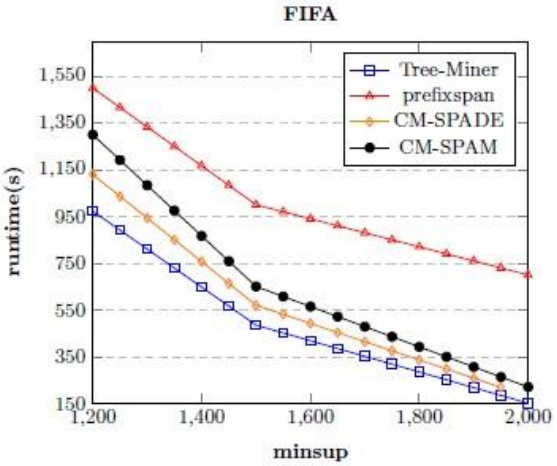
We have conducted our performance on various real life and synthetic datasets and among them we will show the results in the datasets of following table. In other datasets our performance were quite similar.

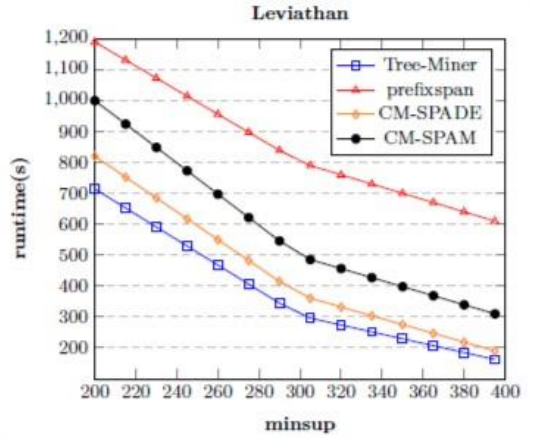
Dataset Name	Sequence	Distinct Item	Avg. Seq Lengths	Type
Snake	163	20	60	protein sequences
FIFA	20450	2990	34.74	web click stream
Leviathan	5834	9025	33.81	book
BMS	59601	497	2.51	web click stream
Sign	730	267	51.99	language utterances
Bible	36369	13905	17.84	book

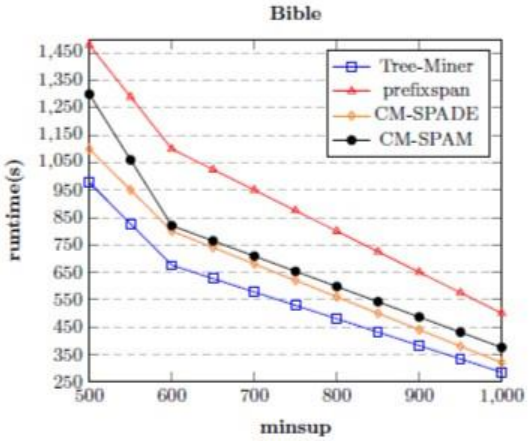
Runtime Comparison

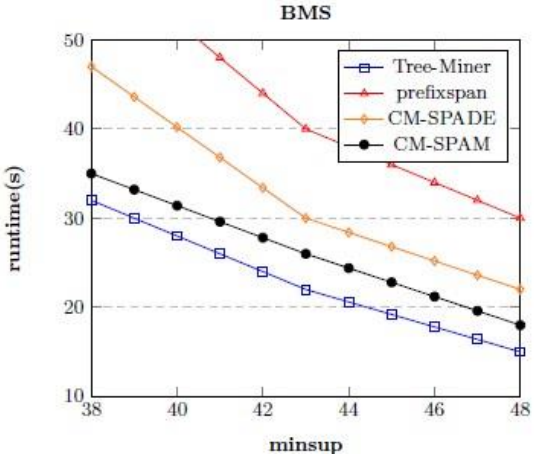








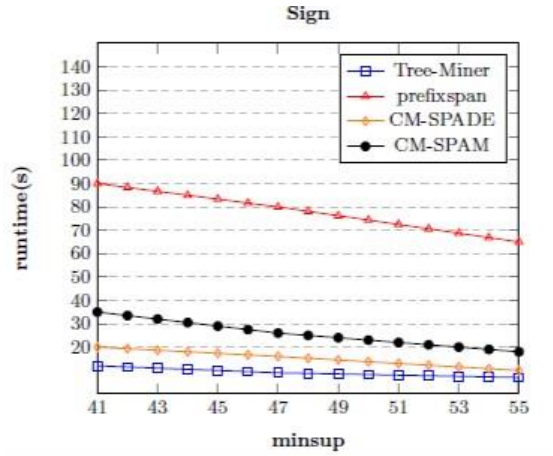




Runtime Comparison Discussion

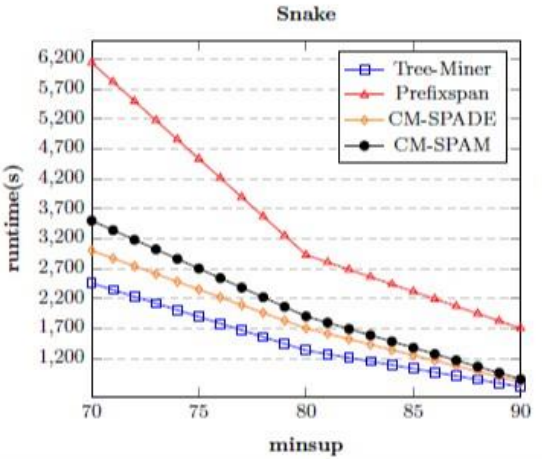
our approach performs comparatively better than other state-of-the-art algorithms while it outperforms Prefixspan to a huge extent while CM-SPADE and CM-SPAM with comparatively closer but significant amount.

Main superiority of our approach is, through next link it reduces the search space faster and efficiently and it does not need to generate any projected database and it also does not need any other structure to calculate the support of a pattern rather than only SP-Tree nodes.



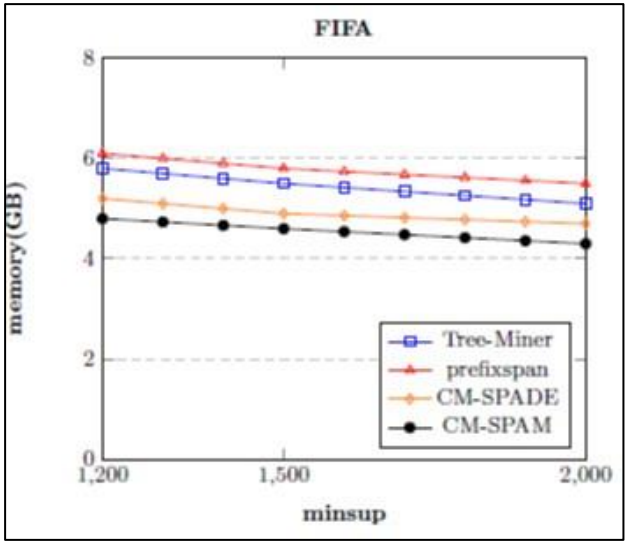
In dense datasets performance improves significantly through prefix sharing

In Sparse datasets through next link moves we can reduce the search space faster to a huge extent



Memory Comparison And Structure Construction Time

Our framework takes slightly more memory compared to CM-SPADE and CM-SPAM but less memory compared to prefixspan, because we do not need to make any projected database.



Memory Comparison

SP-Tree Structure and CMAP table provides a significant improvement during mining through Tree-Miner algorithm and their construction time is quite insignificant to mining time, so it does not create any bottleneck.

Dataset	Threshold(%)	Mining(Second)	Construction Time(Second)
Snake	42.94	2456	0.29
Sign	5.62	15.12	0.25
Fifa	5.87	973.44	1.19
Leviathan	3.43	714.4	0.89
Bible	1.37	977	2.71
BMS	0.064	27.1	0.1

Structure Construction Time Vs Mining Time

Future Works

- Main intention behind designing a tree like structure was to get efficiency during mining using the tree properties and by better controlling and accessing the items and events lying in the database and due to having build-once-mine-many property this structure also becomes handy for interactive mining problem.
- Beside another main goal was to design for an incremental and dynamic database version based on the established framework. Because if we have a tree alike complete structure then it would be easier to capture the modifications and changes. Our future works will include this problem.

In this literature, we wanted to provide a new viewing angle to solve this very classical problem.

THANK YOU