Module 3
Sequential Patterns

VPAC Education Grant Round 2
EDPNMO006/2001
David Taniar and Kate Smith
Monash University
Discovering sequential Pattern

Problem statements

- Given a set of transaction D that each consists of the following fields: customer-id(tid), transaction-time(t), and the items purchased in the transaction.
- Find the inter-transaction patterns/sequences that satisfy minimum support (minsup), minimum gap (mingap), maximum gap (maxgap) and window size (wsizex) given by user.
- Example: Sequence for customer ID 10.

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Time</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2/Jan/2001</td>
<td>Coke, potatoes, twisty</td>
</tr>
<tr>
<td>10</td>
<td>10/Jan/2001</td>
<td>canola oil, chicken, fish</td>
</tr>
<tr>
<td>10</td>
<td>1/Feb/2001</td>
<td>fish fillet, white bread</td>
</tr>
</tbody>
</table>
Sequential Pattern- general definitions (1)

- A sequence $s$ is an ordered list of itemsets $i$. We denote itemset $i$ as $(i_1, i_2, ..., i_m)$ and a sequence $s$ by $\langle s_1, s_2, ..., s_n \rangle$ where $s_j \subseteq i$.

- Customer sequence - a set of transaction of a customer ordered by increasing transaction time ($t$). e.g. Given a set of itemset $i$ for a customer ordered by transaction time $t_1, t_2, ..., t_n$. The customer sequence is $\langle i(t_1), i(t_2), ..., i(t_n) \rangle$.

- A sequence $\langle s_1, s_2, ..., s_n \rangle$ is contained in another sequence $\langle s'_1, s'_2, ..., s'_m \rangle$ if there exist integers $j_1 < j_2 < ... < j_n$ such that $s_1 \subseteq s'_{j_1}$, $s_2 \subseteq s'_{j_2}$, ..., $s_n \subseteq s'_{j_n}$, for $j_n \leq m$. 

**Sequential Pattern- general definitions (2)**

Example:

\( \langle (5\ 6)\ (7) \rangle \) is contained in \( \langle (4\ 5)\ (4\ 5\ 6\ 7)\ (7\ 9\ 10) \rangle \) because \( (5\ 6) \subseteq (4\ 5\ 6\ 7) \) and \( (7) \subseteq (7\ 9\ 10) \).

Whereas \( \langle (3\ 5) \rangle \) is not contained in \( \langle (3)\ (5) \rangle \).
Sequential Pattern- general definitions (3)

- s’ is subsequence of s if s contained s’.
- The length of a sequence is the number of items in the sequence. A sequence of length k is called a k-sequence.
- Given a set of customer sequence D, the support of a sequence s is the fraction of total D that contained s.
- large sequence (lseq) - the sequence that has minimum support (minsup).
Sequential Pattern

Example:

\[ t - \text{number of days from 1/Jan/2001} \]

<table>
<thead>
<tr>
<th>Customer ID</th>
<th>Transaction time(t) days</th>
<th>Items Bought</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>10, 20</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>200</td>
<td>2</td>
<td>10, 20, 30, 40</td>
</tr>
<tr>
<td>200</td>
<td>3</td>
<td>30, 50</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
<td>10, 40, 50</td>
</tr>
<tr>
<td>300</td>
<td>7</td>
<td>40, 70</td>
</tr>
<tr>
<td>300</td>
<td>8</td>
<td>90</td>
</tr>
<tr>
<td>400</td>
<td>5</td>
<td>40</td>
</tr>
</tbody>
</table>

Database sorted by customer ID.

<table>
<thead>
<tr>
<th>Customer ID</th>
<th>Customer Sequence</th>
<th>Sequential Pattern</th>
<th>Support (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(10) (20) (30)</td>
<td>(10) (20)</td>
<td>50</td>
</tr>
<tr>
<td>200</td>
<td>(10 20 30 40) (30 50) (10 40 50)</td>
<td>(10) (30)</td>
<td>50</td>
</tr>
<tr>
<td>300</td>
<td>(40 70) (90)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>(40)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Customer sequence

(2) (3)

Sequence with support 50%
Sequential Pattern- time and sliding window constraints

(1)

- **window size** - maximum time span between first and last itemset in an element (an element consisted of one or more itemset).
- **minimum gap** - minimum time gap between consecutive itemset in a sequence.
Sequential Pattern- time and sliding window constraints (2)

- **maximum gap** - maximum time gap between first itemset of the previous.
- Given a sequence \( s = (s_1, s_2, s_3) \) and \( s_1 = (i_{t1}, i_{t2}, i_{t3}) \), \( s_2 = (i_{t4}) \), \( s_3 = (i_{t5}, i_{t6}) \), with itemset \( i_{tj} \); \( 1 < j < 6 \), and transaction time \( t_i \) that satisfy the time constraints above, the \( startime(s_i) = t_1 \) and \( endtime(s_i) = t_3 \).
Sequential Pattern- time and sliding window constraints

- Given a customer sequence \( c = (c_1, c_2, \ldots, c_m) \) and another sequence \( s = (s_1, s_2, \ldots, s_n) \), \( c \) contained \( s \) if
  - \( s_i \) is contained in \( w_i = \bigcup c_j \) for \( k_i \leq j \leq p_i \), \( p_i < k_{i+1} \), \( k_1 > 0 \), \( p_n \leq m \).
  - \( \text{endtime}(w_i) - \text{starttime}(w_i) \leq \text{wsize} \)
  - \( \text{endtime}(w_i) - \text{starttime}(w_{i-1}) \leq \text{maxgap} \)
  - \( \text{starttime}(w_i) - \text{endtime}(w_{i-1}) > \text{mingap} \)
Sequential Pattern- time and sliding window constraints

Example: minsup = 2

<table>
<thead>
<tr>
<th>Customer-ID</th>
<th>Transaction time (days)</th>
<th>Items bought</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>(A C)</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>(B C D)</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
<td>(C D)</td>
</tr>
<tr>
<td>350</td>
<td>2</td>
<td>(A D)</td>
</tr>
<tr>
<td>350</td>
<td>4</td>
<td>(B D)</td>
</tr>
<tr>
<td>500</td>
<td>4</td>
<td>(A B)</td>
</tr>
<tr>
<td>500</td>
<td>8</td>
<td>(B C)</td>
</tr>
</tbody>
</table>

Customers transactions

Window size = 0

<table>
<thead>
<tr>
<th>large 2-sequence</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) (B)</td>
<td>3</td>
</tr>
<tr>
<td>(A) (C)</td>
<td>2</td>
</tr>
<tr>
<td>(A) (D)</td>
<td>2</td>
</tr>
<tr>
<td>(B) (C)</td>
<td>2</td>
</tr>
</tbody>
</table>

Window size = 3

<table>
<thead>
<tr>
<th>large 2-sequence</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) (C)</td>
<td>2</td>
</tr>
<tr>
<td>(B) (C)</td>
<td>2</td>
</tr>
</tbody>
</table>

large 2-sequence for wsize=0.

large 2-sequence for wsize=3.
Mining sequential pattern - GSP algorithm (1)

- Multiple passes algorithm as in Apriori.
- GSP is a generalization of the AprioriAll algorithm.

**General algorithm**

1. **pass k = 1**
   - Find all large 1-sequence.

2. **pass k > 1**
   - Generate candidate sequences $C_k$ from pass $(k-1)$ large sequences.
   - Count supports for candidates sequence $C_k$
   - Gets large k-sequences.
Mining sequential pattern - GSP algorithm (2)

- Prune large k-sequences where some (k-1) contiguous subsequences is not in large (k-1)-sequences.
Mining sequential pattern - GSP algorithm

Some definitions

- Let \( L_k \) denote the set of all frequent k-sequences, and
  \( C_k \) the set of candidate k-sequences.
- Given a sequence \( s = \langle s_1, s_2, \ldots, s_n \rangle \) and subsequence
  \( c \), \( c \) is a contiguous subsequence of \( s \) if,
  1. \( c \) is derived from \( s \) by dropping an item(s) from
     either \( s_1 \) or \( s_n \), or
  2. \( c \) is derived from \( s \) by dropping an item from an
     element \( s_i \) which has at least 2 items, or
  3. \( c \) is a contiguous subsequence of \( c' \), and \( c' \) is a
     contiguous subsequence of \( s \).
Mining sequential pattern - GSP algorithm

Example:
Given \( s = \langle (1 \ 3) \ (4) \ (5 \ 6) \ (7 \ 8) \rangle \)
\( \langle (1) \ (4) \ (5 \ 6) \ (7) \rangle, \langle (4) \ (5) \rangle, \langle (3) \ (4) \ (6) \rangle \) are contiguous subsequence of \( s \), \( \langle (1 \ 3) \ (5) \ (8) \rangle, \langle (1) \ (4) \ (8) \rangle \) are not.
GSP algorithm - Candidates generation (1)

Join Phase

- for all $s_1 \in L_{k-1}$ and $s_2 \in L_{k-1}$, join $s_1$ and $s_2$ if the subsequence obtained by dropping the first item of $s_1$ is the same as the subsequence obtained by dropping the last item of $s_2$.
- Given $x$ the last element of $s_1$, $y$ the last item of $s_2$, the new candidate is $(s_1, (y))$ if $y$ is single item element, or add $y$ to $x$ otherwise.
- For $C_2$, for all $x \in L_1$ and $y \in L_1$, join $x$ and $y$ to become $\langle (x) (y) \rangle$ and $\langle (x, y) \rangle$. 
GSP algorithm - Candidates generation (2)

- **Prune Phase**
  - Delete candidates sequences $C_k$ that have a contiguous $(k-1)$- subsequence whose is not in $L_{k-1}$.
  - if there is no max-gap constraint, we also delete candidate sequences $C_k$ that have any subsequence not in $L_{k-1}$.
GSP algorithm - Candidates generation

- Example:

<table>
<thead>
<tr>
<th>No</th>
<th>L3</th>
<th>after join</th>
<th>after pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((1 2) (3))</td>
<td>((1 2) (3 4))</td>
<td>((1 2) (3 4))</td>
</tr>
<tr>
<td>2</td>
<td>((1 2) (4))</td>
<td>((1 2) (3 4))</td>
<td>((1 2) (3 5))</td>
</tr>
<tr>
<td>3</td>
<td>((1) (3 4))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>((1 3) (5))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>((2) (3 4))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>((2) (3) (5))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Candidates 4-sequence
GSP algorithm - Counting support

- For fast counting, the candidate sequences are stored in the hash tree as in Apriori.
- Adding candidate sequences to hash-tree
  - All nodes are initially created as leaf node.
  - Start from root node, we hash on item number 0.
  - At interior node of depth \(d\), we hash on \(d\)-th item to go to depth \(d+1\).
  - When we reach a leaf node we add the candidate sequences to the candidate list.
  - When the number of candidate sequences in a leaf node exceeds a threshold, we convert it to an interior node by hash on the \(d\)-th item of the candidates sequence.
Adding Candidate sequences

Hash Function

Part of hash tree for k=3

Source Joshi M. et.al.
**GSP algorithm - Counting support**

Given a customer sequence q.

**Update support count**

- *Interior node* - if it is the root node, we apply hash function to each item in q, and recursively apply this to the node in the corresponding bucket.

- *Interior node* (not root node). If we reached this node by hashing on an item x whose transaction-time is t. We apply the hash function to each item in q whose transaction-time is in \([t - wsize, t + max(wsize, maxgap)]\).

- *Leaf node* - for each candidate sequence s, check if q contained s, then update the support count.
Example

Let $s = \langle (1, 4; t=0) (2, 5, 3; t=5) (1, t=12) (1, 9, 3; t=15) (3, 2; t=20) \rangle$, wsize=6, maxgap=10, mingap=0, $[t-6, t+10]$
Contains Test (1)

Find all candidate sequences $s$ in *leaf node* that contained in $q$.

- **Forward Phase**
  - Finds successive element of $s_i \in s$ in $q$ as long as the *maxgap* constraint between the $s_i$ and $s_{i-1}$ is satisfied.
  - If one of $s_i$ element is not found in $q$ then $q$ does not contained $s$.
  - If maxgap or mingap constraint is violated, we move to backward phase.
Contains Test (2)

- Backward Phase
  - If $s_i$ is the current element in $q$ and the $\text{endtime}(s_i)$ is $t$, finds the first set of transactions containing $s_{i-1}$ whose transaction times are after $t - \text{maxgap}$.
  - find $s_{i-j}$ in $q$ until $\text{maxgap}$ between $s_{i-j}$ and $s_{i-j-1}$ ($1 \leq j < (i-1)$) is satisfy.
  - if any of the $s_i$ element is not found in $q$, then $s$ is not contained in $q$. 
### GSP algorithm - Counting support

**Window size** = 0  
**Max gap** = 6  
**Min gap** = 3  

**Candidate sequence**  
\( \langle (A B) (D) (E) \rangle \)

---

**Customer sequence**

<table>
<thead>
<tr>
<th>Transaction time(t)</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(A, B)</td>
</tr>
<tr>
<td>4</td>
<td>(B, C, D)</td>
</tr>
<tr>
<td>8</td>
<td>(C, E)</td>
</tr>
<tr>
<td>15</td>
<td>(A, D)</td>
</tr>
<tr>
<td>20</td>
<td>(A, B, D)</td>
</tr>
<tr>
<td>25</td>
<td>(B, E)</td>
</tr>
<tr>
<td>30</td>
<td>(C, D)</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>forward</th>
<th></th>
<th>backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>t</td>
<td>mingap cons.</td>
<td>maxgap cons.</td>
</tr>
<tr>
<td>(A, B)</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>(D)</td>
<td>15'</td>
<td>*</td>
<td>/</td>
</tr>
<tr>
<td>(D)</td>
<td>20</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>(E)</td>
<td>25</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* constraint satisfied  
/ constraint not satisfied
GSP algorithm - Finding a single element (1)

- Use during the forward and the backward phase to find a candidate element in customer sequence after a time \( t \).

Fast finding

- Create an array of list with size \( \text{maxitem} \), where \( \text{maxitem} \) is the maximum items number in the database.

- create a list for each item in customer sequence \( d \) to hold the time \( T \) where this item occur in a transaction of \( d \).
GSP algorithm - Finding a single element (1)

- to find an element e in d after time t, for each item i ∈ e, traverse the list corresponding to index i in the array and find the time T > t.
- If the endtime(e) - starttime(e) ≤ wsize, then it is done. Otherwise t is set to endtime(e) - wsize, and search again in the array.
- If the an item after time t is not found, then d is not contains the candidate sequence.
GSP algorithm - Finding a single element

Example:

Customer sequence

<table>
<thead>
<tr>
<th>Transaction time(t)</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(A B)</td>
</tr>
<tr>
<td>4</td>
<td>(B C D)</td>
</tr>
<tr>
<td>8</td>
<td>(C E)</td>
</tr>
<tr>
<td>15</td>
<td>(A D)</td>
</tr>
<tr>
<td>20</td>
<td>(A B D)</td>
</tr>
<tr>
<td>25</td>
<td>(B E)</td>
</tr>
<tr>
<td>30</td>
<td>(C D)</td>
</tr>
</tbody>
</table>

Transform representation

<table>
<thead>
<tr>
<th>Item Array</th>
<th>Times (T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>20</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>25</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Parallel Count Distribution (1)

- Similar to count distribution of apriori algorithm.
- Steps
  - Partition database into $p$ partition ($p$ is the total number of processor).
  - pass $k=1$
    - Each processor $P^i$ asynchronously find 1-sequences support count from its local partition.
    - Synchronous to exchange local sequence support count.
    - Each processor $P^i$ find the same set large 1-sequences.
Parallel Count Distribution (2)

pass $k > 1$

- Generate candidate $k$-sequences using large $(k-1)$-sequences.
Parallel Count Distribution (3)

- Add all candidate k-sequences $C_k$ into hash tree.
- $P_i$ scan its local partition to update support count for $C_k$.
- Synchronous all processor and perform allReduce to exchange $C_k$ local support count.
- Each processor gets large k-sequences from $C_k$. 
Parallel Data Distribution (1)

- Similar to Data Distribution in Apriori algorithm.
- Involves partitioning both the data and the candidate sequences.

**Steps**

- Pass K=1
  - similar to Count Distribution
- Pass K > 1
  - Generate candidate sequences using previous iteration large sequences. Divide 1/P of them to each processor.
  - Use local and received data sequences to update candidate sequences support counts.
Parallel Data Distribution (2)

- Gets large sequences
- Gather all large sequences from all processors
- Generate Sequential Rules
GSP algorithm - Parallel implementation

- **Input format**
  1. Configuration file (config.txt)
     - number of customer sequences
     - maximum value of items.
  2. Database file (input.txt)
     - CustID TransID T item\(_1\) item\(_2\) … item\(_m\)

**Assumptions:**
- CustID’s are groups together in ascending order according to TransID.
- T is days from a specified date (i.e. 2 days from 01/02/2002)
- items are sorted in lexicographical order.
Output Format

- Sequential rules file
  - srulexx.txt where xx is the rank number.
- Rules format
  - \[ (6)(20) \] Support:5 (12.50%)
  - 5 is the count support, and 12.50% is the support in percentages.

User will prompt for max gap, min gap, window size, and minimum support.
Source Files

- Sequence.cpp, Sequence.h - contains main functions for sequential rules generation
- Seqnode.cpp, Seqnode.h - contains list of itemset
- Itemset.cpp, Itemset.h - list of items
- Dbtrans.cpp, Dbtrans.h - customer sequences and configurations file input functions
- HashTree.cpp, HashTree.h - hash tree data structures implementations
- Hashtab.cpp, Hashtab.h - hash table implementations
Global variables

- unsigned int MIN_SUPPORT /* minimum support */
- unsigned int MAX_GAP /* maximum time gap */
- unsigned int MIN_GAP /* minimum time gap */
- unsigned int WINDOW_SIZE /* window size */
- int DBASE_MAX_ITEM /* items max value */
- long DBASE_NUM_SEQ /* number of customer transactions db in local partition */
- long GLOBAL_NUM_SEQ /* total sequences in all partitions */
- list<Sequence *> * freq_seqs /* large sequence */
Object classes (1)

- class Itemset {...
  - stores itemset in a sequence
  - items in itemset are store in a list
  - transaction time of this itemset (event)
  - others members support methods

1. class Itemset {
   private:
     list<int> * lst_item;     //item in the itemset.
     int event;     //time when this transaction is done.
   5. public:
      Itemset(int=0);
      ~Itemset();
      Itemset(const Itemset & iset); //copy constructor
      int subset(list<int> & iv);     //subset method.
   10. int getevent()const { return event; }
      void setevent(int e) { event = e; }
      void insert(int new_i) { lst_item->push_back(new_i); }
      int size() { return lst_item->size(); }
      list<int>& get_item()const { return *lst_item; }
   15. ....//overloaded operator
};
Object classes (2)

- class Sequence { ... }
  - store a list of itemset
  - will becomes the storage node in a hash tree
  - performs sub-sequence and contiguous sequence functions
  - private members access and support methods

1.    class Sequence {
        private:
            list<Itemset *> * element; //list of Itemset.
            US_INT support; //support for this sequence.
        5.    US_INT SeqID; //customer ID or sequence ID.
            Itemset ** elementarr; //store the Itemset in list<Itemset
                             *>
                //for fast element access.
    public:
        Sequence(int);
        Sequence(const Sequence&); //copy constructor
    10.   ~Sequence();
        void addElement(const Itemset & elem) const;
        void addElement(const list<Itemset *>**)&const;
        void addElement(Itemset *);

        /*Private data access*/
        list<Itemset *> * & getElement() { return element; }
    15.   US_INT getID () const { return SeqID; }
        void setID(US_INT sid) { SeqID = sid; }
        US_INT & getsupport() { return support; }
        int size() const { return element->size(); }
int * toArray(int &size) const;

20. Itemset ** toArrayElem(int &elemsize);

Itemset & elem(US_INT index) const; // operator to access
     // the index-th element.
Itemset & elem(US_INT index);
int item(int index); // operator to access index(th) item not
     // element.

/* Contiguous Sequence and Subsequence checking function */
bool IsContain(Sequence &ns); // check if this sequence is
     // contain in of ns. Use for candidate
     // count.

25. bool IsSubsequence(const Sequence &);
    /* Operator definition */
    bool operator==(const Sequence & ) const;
    friend ostream & operator<<(ostream &os, const Sequence &);
    friend list<US_INT>** createItemsarr(const Sequence &);
    friend bool findfirstelement(list<US_INT>**, 
        ITER(Itemset *) & , US_INT,
        US_INT &, US_INT & /*in out */);

• The methods in lines 19, lines 20 and lines 25 are the core method for this class. The
  IsContain() method use use to check whether a customer sequence contains the sequence
  in this class or not.
• The IsSubsequence() method is use to check whether this sequence is contains in a
  customer sequences (as argument) or not when there is no maximum gap constraint.
• The findfirstelement() method find the existence of an element a time t.
Object classes (3)

■ class HashTree {...}
  • stores the candidate sequences at each iteration
  • if it is a interior node, the hash table is use to store
    the pointer to the child in the next level, or if it is a
    leaf node, the list structure is use to store the
    candidate sequences
  • performs private members support functions

1. class HashTree {
   private:
   HashTree **theHTable; //Hash table for the internal node.
   list<Sequence *> * theSList; //list of candidate sequence in
   leaf node.
   5. int theFlg; //use for support count.
   int theDepth; //depth of this node.
   static int theCount; //number of node (internal/leaf) in the
   tree.
   static unsigned int theThreshold; //maximum number of itemset in
   //a leaf node before rehash/split.
   static int theFanOut; //branching factor for the hash table
   10. static int theMaxDepth; //maximum allowable depth for the hash
       //tree.
   void rehash(); // procedure for converting leaf node to a
   interior node
   public:
   HashTree(int=0);
   ~HashTree();
   15. int hash(int); //hash function
   void addSequence(Sequence *); //add a new candidate to the hash
   tree.
   int isLeaf(){ return (theHTable == NULL); }
int& flag() { return theFlg; }  // store transaction ID.
int& depth() { return theDepth; }  // this hashtree depth.
20. HashTree **&htable() { return theHTable; }

list<Sequence *> * & seqList() { return theSList; }
static int& count() { return theCount; }
static int& fanout() { return theFanOut; }
static unsigned int& threshold() { return theThreshold; }
25. static int& maxdepth() { return theMaxDepth; }

static void calc_fanout(int num, int dep)
{
    theMaxDepth = dep;
    theFanOut = (int)ceil(pow(num/theThreshold, 1.0/dep));
    if (theFanOut < 1) theFanOut = 1;
30. }
friend ostream& operator << (ostream&, HashTree&);
};
GSP Implementations (1)

- Main functions (Sequence.cpp)
  - new_cand() - create a new candidate sequence from two candidate sequence
  - can_join() - check if two candidate sequence can be joined during candidates generation step
  - IncrementCount() - update support counts in leaf node
  - countSupport() - traverse the hash tree to update support counts
  - GSP() - main GSP algorithm loop
function GSP() pseudo-code

call find_F1_seq()
while(large sequence count > 0)
call joinandprune()
if (no candidate) stop
for all sequence s in the database
call countSupport(s)
end for
call getHTSupport() /*get candidate seq’s local support*/
call MPI_Barrier() /*synchronize all processors*/
call MPI_Allreduce() /*get local support counts*/
call updateHtSupport() /*update global support counts*/
call get_large_sequence() /*get large sequences*/
call Output_rule() /*output sequential rules*/
end while

1. void GSP(Dbtrans * & db, int my_rank)
   { char rulefile[50];
     ofstream * fout;
     long totalrules =0;
   5. Hash_tbl * htab = NULL;
       ITER(Sequence *) s1,e1;
       HashTree *ht;
       US_INT * cand_sup;
       US_INT * global_sup;
10. US_INT index;
       int sid;
       Sequence * buffer=NULL;
       US_INT cand_count;
       int iter;
15. freq_seqs = new list<Sequence *>;

       find_F1_seq(db,freq_seqs);
       iter =1;
       sprintf(rulefile,"srule%d.txt",my_rank);
       fout = new ofstream(rulefile);
while(freq_seq->size() > 0)
{
    iter++;
    cand_count = joinandprune(iter,freq_seq,ht,htab);
    /*If no more candidate, just stop */
    if (cand_count <= 0)
    {
        delete ht;
    }
    if (htab != NULL)
    {
        delete htab;
        break;
    }
    ITER(Sequence *) itr_sstart = freq_seq->begin();
    ITER(Sequence *) itr_send = freq_seq->end();
    //free up Sequence memory for previous loop.
    while(itr_sstart != itr_send)
    {
        delete (*itr_sstart);
        itr_sstart++;
    }
    //delete all list node container.
    freq_seq->erase(freq_seq->begin(), freq_seq->end());
    buffer = new Sequence(0);
    while(db->get_next_sequence(sid,buffer))
    {
        assert(buffer != NULL);
        countSupport(ht,buffer,-1);
        delete buffer;
        buffer = new Sequence(0);
    }
    cand_sup = new US_INT[cand_count]; /*allocate mem to store candidate sup*/
    global_sup = new US_INT[cand_count]; /*allocate mem for allreduce */
    getHtSupport(ht,cand_sup,index=0); /*get local support count*/
    MPI_Barrier(io_comm); /* all processor synchronize here */
    /*sum up global support count */
    MPI_Allreduce(cand_sup,global_sup,cand_count,MPI_UNSIGNED,
                  MPI_SUM,io_comm);
    /*update local support count, to global support*/
    updateHtSupport(ht,global_sup,index=0);
    //destroy hash table from the previous loop
    if (htab != NULL)
    {
        delete htab;
    }
//hash table to store large sequence candidates
//in the current loop.
50.     htab = new Hash_tbl(DBASE_MAX_ITEM);
    get_large_sequence(freq_seqs,ht,htab);
    totalrules+=  Output_rule(fout,freq_seqs,my_rank);
    delete ht;
    delete [] cand_sup;
55. }

    ...}

• Above is main looping for the GSP algorithm.
• Lines 2- 15 are the local variables definitions and initializations
• Lines 16, the find_F1_seq() function is calls to find the large 1-sequence
• Lines 18, each processor will open file to store the association rules.
• To main loop start at line 20 and end at lines 55
• In lines 22, the joinandprune() function is calls to generate the k-th iteration candidates
  sequences.
• Lines 36-41 perform support counts update by calling  countSupport() function.
• Lines 44, the getHtSupport() function is calls to get the local support counts.
• Lines 45 and 46, all processors are synchronized to exchange the local support counts to
  obtain the global support counts.
• Lines 47, the updateHtSupport() function is calls to update the support counts in the hash
  tree.
• Lines 51, the get_large_sequence() function is calls to find the large sequences.
• Finally, on lines 52, the Output_rule() function is calls to output the association rules.
GSP Implementations (3)

- void Find_F1_seq()
  - scan local partition customer sequences to find local support counts.
  - exchange local support counts with other processor to obtain global support counts.
  - each processor independently find all large 1-sequences from its local array structure.
  - store the large sequences in a list template.

1. void find_F1_seq(Dbtrans * & db, list<Sequence *> * & F1)
   {
      int sid;
      long i;
      Sequence * buffer = new Sequence(0);
   
5.   ITER(Itemset *) itrs,itre;
   ITER(int) itr_ints,itr_inte;

   US_INT * itemsid = new US_INT[DBASE_MAX_ITEM+1];

   US_INT * global_sup = new US_INT[DBASE_MAX_ITEM +1];
   char * bitmap = new char[DBASE_MAX_ITEM +1];

10.  for (i=0;i<= DBASE_MAX_ITEM;i++)
      itemsid[i] = 0;
      memset(bitmap,’0’,DBASE_MAX_ITEM+1); //set the flag to 0.
while(db->get_next_sequence(sid,buffer))
{
    itrs = buffer->getElement()->begin();
    itre = buffer->getElement()->end();
    //for all elements in the data sequence
    while(itrs != itre)
    {
        itr_ints = (*itrs)->get_item().begin();
        itr_inte = (*itrs)->get_item().end();
        //for all items in a itemset/element
        while(itr_ints != itr_inte)
        {
            //not yet update for this d-sequence
            if(bitmap[*itr_ints] == '0')
            {
                itemsid[*itr_ints]++;//update support count
                //set the flag updated for this
                bitmap[*itr_ints] = '1';
            }
            itr_ints++;//next item
        }
        itrs++;//next element
    }//while
    itrs = buffer->getElement()->begin();
    while(itrs != itre)
    {
        delete (*itrs);
        itrs++;
    }//while
    //delete all list node.
    buffer->getElement()->erase(buffer->getElement()->begin(),
                               buffer->getElement()->end());
    memset(bitmap,'0',DBASE_MAX_ITEM+1);
});//while
delete [] bitmap;
MPI_Barrier(io_comm);
MPI_Allreduce(itemsid,global_sup,DBASE_MAX_ITEM+1,
              MPI_UNSIGNED,MPI_SUM,io_comm);
delete [] itemsid;
40. Itemset * iset;  Sequence * seq;
   //get all large 1-sequence.
for(i=1;i<= DBASE_MAX_ITEM;i++)
{
    if (global_sup[i] >= MIN_SUPPORT)
    {
        iset = new Itemset(0);
        assert(iset != NULL);
        iset->insert(i);
        seq = new Sequence(0);
        assert(seq!= NULL);
        seq->getsupport() = global_sup[i];
        seq->addElement(iset);
        F1->push_back(seq);
    }
}
55. delete [] global_sup;

• Lines 2-12: local variables are declared or initialization
• The bitmap variable in lines 9 is use to check whether an item in a customer sequence support count already updated or not. One sequence only allowed to update the support count of one item once.
• The main loop start at lines 13 and end at lines 36.
• Lines 16-27 : update the support counts of 1-sequence
• Lines 38 and 39: synchronized all processors and exchange local support counts
• Lines 41-53: find large 1-sequences and put it into a list (lines 53).

Notes:
Item/product ID cannot be 0 and it must be an unsigned integer value.
GSP Implementations (4)

- void joinandprune()
  - performs self-joined on large candidate sequences and produce a new set of candidates sequence.
  - if current iteration \( k > 2 \), also perform pruning using the definition defined earlier
  - calls can_join() and new_cand() functions to help it operations
  - add new candidate sequences into hash tree

```c
long joinandprune(int iter, list<Sequence*>* & Fk, 
                   HashTree * & ht, Hash_tbl * & htab)
{
    //... 
    ht = new HashTree(); /*create new candidate Hash Tree*/ 
    //...
    if (iter == 2)
    {
        join 1-sequence x and y to produce 2-sequences <(x y)> 
        and <(x)(y)>.
    } else
    {
        itrs_L1 = Fk->begin();
        itre_L1 = Fk->end(); /*generate new candidate using sort-merge method*/
        while(itrs_L1 != itre_L1)
        {
            itrs_L2 = Fk->begin();
            while(itrs_L2 != itre_L1)
            {  //don’t join the same element
                if (itrs_L2 == itrs_L1)
                    { itrs_L2++;
                      continue;
                    }
            }
        }
    }
```
if (can_join(*itrs_L1, *itrs_L2))
20.
    {
        tempsq =
        new_cand(*itrs_L1, *itrs_L2, htab);
        if (tempsq != NULL)
            {
                ht->addSequence(tempsq);
25.                cand_count++;
            }
        }/*else
        itrs_L2++;/*
30.        itrs_L1++;/*
    }
 }//else
    return cand_count;

• Lines 4: a new hash tree object is created.
• Lines 5-8: generate candidates 2-sequences from large 1-sequences.
• Lines 9-31: generate candidate k-sequences from large (k-1)-sequences. Function
can_join() is calls at lines 19, function new_cand() is calls at lines 21 and method
dAddSequence() at lines 24 is calls to add new candidate sequence to hash tree.
GSP Implementations (5)

- **new_cand()**
  - create a new candidate sequence from two sequences. Also check whether all contiguous-subsequence of the new candidate sequence is large or not.
  - The hash table store all the previous iteration large sequences.
  - calls contiguous_sequence() function to generate contiguous subsequence

```c
1. Sequence * new_cand(Sequence * &c1, Sequence * &c2, Hash_tbl * &htab)
{
    Sequence * new_c_seq = new Sequence(*c1);
    ITER(Itemset *) c2_elem = c2->getElement()->end();
    5. c2_elem--;
    if ((c2_elem)->size() == 1)
    {   Itemset * tempiset = new Itemset(0);
        tempiset->insert((*c2_elem)->get_item()->begin());
        new_c_seq->addElement(tempiset);
        tempiset=NULL;
    }
    else
    {   /**************************************************************************
        * concate last item of c1 into
        * last element of new candidate
        **************************************************************************/
        ITER(Itemset *) lelem = new_c_seq->getElement()->end();
        lelem--;
        15. ITER(int) litem = (*c2_elem)->get_item()->end();
        litem--;
        (*lelem)->insert(*litem);
    }
    Sequence *cont_seq;
}
ITER(Itemset *) itr_scand, itr_ecand, temp_itr;
itr_scand= new_c_seq->getElement()->begin();
itr_ecand = new_c_seq->getElement()->end();
int arr_size, i;

int * items_arr = new_c_seq->toArray(arr_size);
int * arr elemsize = new int[new_c_seq->size()];
i=0;
while(itr_scand != itr_ecand)
    { arr elemsize[i] = (*itr_scand)->size();
        i++;
    itr_scand++;
}
int j, numelem, sumsize;
umelem = new_c_seq->size();
j=0; sumsize = 0;

while(j < numelem)
    { if ((j != 0) && (j != (numelem -1)) && (MAX_GAP > 0))
        { if (arr elemsize[j] == 1)
            sumsize++;
            j++;
        continue;
    }
i =0;
while(i < arr elemsize[j])
    { sumsize++;
        cont_seq = contiguous_sequence(items_arr, arr elemsize, 
umelem, sumsize);
        if(!htab->lookup(cont_seq))
            { delete cont_seq;
                delete new_c_seq;
                return NULL;
            }
delete cont_seq;
        cont_seq = NULL;
        i++;
    }
j++;
} //while
return new_c_seq;
• Lines 1 - 4: local variables and initializations.
• Lines 3: the c1 sequence is copied to the new candidate sequence.
• Lines 5-18: the new candidate is created by extend the sequence of c1 with last item of c2.
• Lines 24-25: the new candidate sequence is converted into an array of items for fast access. The arr_elemsize array store the size of each element for the converted array.
• Lines 35-52: the new candidate sequence is check whether all its contiguous sub-sequence is in previous iteration large sequence. Since all the large sequences from the previous iteration are stored in the hash table. The contiguous sub-sequence checking will be fast.
• Lines 45: the contiguous_sequence() is use to check for contiguous subsequence.
• Lines 37-42: we check if there is no max_gap constraint, it is alright to drop an element in the new candidate sequence with single item.
GSP Implementations (6)

- count_support()
  - increment the candidate sequences support counts
  - traverse the hash tree recursively until reached the leaf node and then perform contained checks.
  - calls IncrementCount() function to update the support counts.
  - the forward and backward phase is perform to check whether a customer sequence contains a candidate sequence in a leaf node.

```c
1. void countSupport(HashTree *ht, Sequence *ns, int ld)
   { int hashval;
     static int maxval = (WINDOW_SIZE > MAX_GAP) ? WINDOW_SIZE : MAX_GAP;
     if (ht->isLeaf()) // leaf node
5.   { if ((ht->flag() != ns->getID()) && (ht->seqList()) &&
          (ht->seqList()->size() > 0))
       { ht->flag() = ns->getID();
         IncrementCount(ht,*ns);
       }
     }
10.   else if (ld == -1) // interior node, root node
15.   { ITER(Itemset *) itr_s,itr_e;
       ITER(int) itr_ints,itr_inte;
       itr_s = ns->getElement()->begin();
       itr_e = ns->getElement()->end();
       // hash all items in the data sequence
       while(itr_s != itr_e)
       { itr_ints = (*itr_s)->get_item().begin();
         itr_inte = (*itr_s)->get_item().end();
         while(itr_ints != itr_inte)
         {
```
20. hashval = ht->hash(*itr_ints);
    assert(hashval >= 0);
    if ((ht->htable())[hashval] != NULL)
        countSupport((ht->htable())[hashval],
             ns,(*itr_s)->getevent());
    itr_ints++;
}
25. itr_s++;
} //while
} else //INTERIOR NODE, NOT ROOT NODE.
{ int mintime = ld - WINDOW_SIZE;
30. int maxtime = ld + maxval;
    ITER(Itemset *) itr_start, itr_end;
    int i,j;
    if (mintime < 0)
        mintime = 0;
    itr_start = ns->getElement()->begin();
    itr_end = ns->getElement()->end();
    while((itr_start != itr_end) &&
          ((*itr_start)->getevent() < mintime))
        itr_start++;
    while((itr_start != itr_end) &&
          ((*itr_start)->getevent() <= maxtime))
40. {  i = (*itr_start)->size();
        j =0;
        while(j < i)
            { hashval = ht->hash(**itr_start)[j]);
                assert(hashval >=0);
            if ((ht->htable())[hashval] !=NULL)
                countSupport((ht->htable())[hashval],ns,
                     (*itr_start)->getevent());
                j++;
            }
        itr_start++;
    }
50. } //while
} //else

• Lines 4-9: if it is a leaf node and current leaf node haven’t updated then the
  IncrementCount() function is calls to update the support counts on the candidate
  sequences in this leaf node.
• Lines 10-27: we are in the root node. Here we hash on each items in the customer sequence to go to the child node.

• Lines 28-50: we are in the interior node of the hash tree. Here we hashed on items in the customer sequence that satisfy the maximum gap, window size constraints and then perform recursive calls.

• Lines 29-30: the time range of qualified items to hash was calculated.
GSP Implementations (7)

- **IncrementCount() function**
  - increment the support counts of candidate sequences in the leaf node of hash tree
  - calls the IsContain() method of sequence class to check whether a customer sequence contains a candidate sequence or not

1. void IncrementCount(HashTree *&ht, Sequence &ns)
   
   ```
   list<Sequence>::iterator sitr, seitr;
   sitr = ht->seqList()->begin();
   seitr = ht->seqList()->end();
   ```

5. while(sitr != seitr)
   
   ```
   if ((*sitr)->IsContain(ns))
       (*sitr)->getSupport()++;
       sitr++;
   ```

10. }

- On lines 7 the IsContain() method is calls to check whether current candidate sequence (sitr) is contained in customer sequence (ns). If it is the support count is updated.
GSP Implementations (8)

- IsContain() method (seqnode.cpp)
  - return true if this candidate sequence is contained in the customer sequence.
  - performs backward and forward phases to find elements in customer sequence that contains elements in this candidate sequence.

```cpp
1. bool Sequence::IsContain(Sequence & dataseq) {
    list<US_INT> ** itemstime;
    int t_nextelem;
5. struct TimeInfo{
        US_INT starttime;
        US_INT endtime;
    } * arr_elemtime;
    arr_elemtime = new TimeInfo[this->element->size()];
10. US_INT starttime, endtime;
    bool ISCONTAIN;
    itemstime = createItemsarr(dataseq);
    assert(itemstime != NULL);
    ITER(Itemset *) Cand_sitr, Cand_eitr;
15. Cand_sitr = this->element->begin();
    Cand_eitr = this->element->end();
    t_nextelem=0;
    ISCONTAIN = true;
    int i =-1;
20. US_INT maxtimegap;
```
while(ISCONTAIN && (Cand_sitr != Cand_eitr))
{
    //forward phase
    //find first element of current candidate element
    //after time prev_start_time.
    ISCONTAIN = findfirstelement(itemstime,Cand_sitr, t_nextelem, starttime,endtime);
    if (!ISCONTAIN) //candidate element not found
        break;
    else
    {
        i++;
        arr_elmetime[i].starttime = starttime;
        arr_elmetime[i].endtime = endtime;
    }

    if (t_nextelem == 0)
    {
        Cand_sitr++;
        t_nextelem = endtime + MIN_GAP;
        //added min_gap to cater min time constraint.
        continue;
    }

    maxtimegap = endtime - arr_elmetime[i-1].starttime;
    //Check MAX_GAP and MIN_GAP constraints
    if ((maxtimegap > MAX_GAP) && (MAX_GAP != 0))
    {//Backward Phase
        t_nextelem = endtime - MAX_GAP;
        do
        {
            Cand_sitr--; //"pulls up" previous element
            i--;
            ISCONTAIN = findfirstelement(itemstime,Cand_sitr, t_nextelem, starttime,endtime);
            if (!ISCONTAIN)
                break;
        }
        while (Cand_sitr != Cand_eitr);
    }

    arr_elmetime[i].starttime = starttime;
    arr_elmetime[i].endtime = endtime;
    maxtimegap = endtime - arr_elmetime[i-1].starttime;
    if (maxtimegap <= MAX_GAP)
    {
        t_nextelem = endtime + MIN_GAP;
        break;
    }
}
lines 5-8: this structure is used to store the start time and end time of elements in the candidate sequence found in the customer sequence so that we can use it in the backward phase later on.

- The forward phase starts at lines 21 to find the element of the candidate sequence in the customer sequence after time $t_{\text{nextelem}}$. The start time and end time of this element if returned in the pass in parameters.

- Lines 24: if an element of the candidate sequence cannot be found in the customer sequence then stop the iteration and return false.

- Lines 30-34: if it is the first element of the candidate sequence, then no need to check for time constraints, continue to the next element.

- Lines 37-53 are the backward phase. In lines 41, the previous element is pulled up to check again in the customer sequence with a different start time calculated in lines 39.

- Lines 48-51: if the max_gap and min_gap constraints are met again, then it will switch to the forward phase by breaking the loop.
Data Dis. - send/recv data sequences format

- Send format for
  - \([\text{sid}][\text{num}]([\text{tid}][\text{T}][\text{n}][\text{item1}][\text{items}])_1\ldots ([\text{tid}][\text{T}][\text{n}][\text{item1}][\text{items}])_\text{num}\)
  - sid - sequence ID
  - num - number of element in the sequence
  - T - transaction time
  - tid - transaction ID
  - n - number of items in a transaction/itemset
  - \([\ ]\) means one entry in an array
- Buffer ended with -111 for end of buffer and -999 for end of transfer.

- Example:
  A sequence with two itemsets,
  22 105 3 45 56
  22 106 10 60

  \([22][2][105][3][2][45][56][106][10][1][60][-111]\)

- Done by CopyBuffer() and DecodeBuffer() functions.
Data Dis. - send/recv large sequences format

- Format of sending/receiving large sequences
  
  
  
  [num][sup][\([i_1], [i_2], ..., [i_n]\)]_\text{num} \quad (\text{num}_\text{num}, [i_1], ..., [i_n])_\text{num}

  - num - number of elements in this sequence
  - sup - support count
  - n - number of items in an element/itemset

• Example:

- A large sequence \([(3, 5), (45, 55)]\) with support count 10 will be encoded as follow.

  \[2][10][2][3][5][2][45][55]\]
1. void GSP(Dbtrans * & db)
   
   freq_seqs = new list<Sequence *>
   find_F1_seq(db,freq_seqs); //gets large 1-sequence
5. iter = 1;
   buftot = (int*)malloc(sizeof(int) * processor_size);
   recvdisp = (int*)malloc(sizeof(int) * processor_size);
   while(freq_seqs->size() > 0)
   {
10.   cand_count = joinandprune(iter,freq_seqs,ht,htab);
15.   if( myseq > 0)
110.   
115.      sendsize = 0;
120.      PackLargeSeq(sendbuf,freq_seqs,sendsize);
125.   }
20.   
25.   MPI_Allgather(&sendsize,1, MPI_INT, buftot,1, MPI_INT,io_comm);
30.   recvsize = 0;
35.   for(i=0; i < processor_size; i++)
40.      recvsize += buftot[i];
45.   if(recvsize <= 0)
50.      break;
55.      recvdisp[0] = 0;
60.      for(i=1; i < processor_size; i++)
65.         recvdisp[i] = recvdisp[i-1] + buftot[i-1];
70.         recvbuf = (int*)malloc(sizeof(int) * recvsize);
75.         MPI_Allgatherv(sendbuf,sendsize, MPI_INT, recvbuf, 
80.            buftot, recvdisp, MPI_INT, io_comm);
85.         UnpackLargeSeq(recvbuf,recvsize,freq_seqs,htab);
90.      totalrules += Output_rule(fout,freq_seqs);
95.   delete ht;
100. 
105. }
• Lines 3: freq_seqs is the list that will always store the large sequences result of current iteration.
• Lines 4: find the large 1-sequences.
• Lines 9: start of the main loop of the algorithm.
• Lines 10: generate candidate sequences using last iteration large sequences. The candidate sequences are stored in the hash tree structure. The i-th candidate sequence is assigned to the (i\%P) processor. Thus each processor will have nearly equal number of candidate sequences.
• Lines 11: Update candidate support counts using data from local and other processor.
• Lines 13: Gets large sequences.
• Lines 17: Pack the large sequences in this processor into the send buffer to send to other processor.
• Lines 22: Processors gather all the number of send buffer size.
• Lines 24-25: Sum up the total size of data that going to receive from all processor.
• Lines 26: If there is no more large sequences, then stop the main loop.
• Lines 32: Receive all large sequences from all other processor.
• Lines 34: Unpack the received raw data and convert them into the proper format of large sequences along with their support count.
• Lines 35: Output large sequences in parallel.
End of Module 3
Sequential Patterns