# Efficient Mining of Spatiotemporal Patterns

Ilias Tsoukatos and Dimitrios Gunopulos

Computer Science Department, University of California Riverside
Riverside CA 92521 USA
{Etsouk,dg}@cs.ucr.edu

**Abstract.** The problem of mining spatiotemporal patterns is finding sequences of events that occur frequently in spatiotemporal datasets. Spatiotemporal datasets store the evolution of objects over time. Examples include sequences of sensor images of a geographical region, data that describes the location and movement of individual objects over time, or data that describes the evolution of natural phenomena, such as forest coverage. The discovered patterns are sequences of events that occur most frequently. In this paper, we present DFS_MINE, a new algorithm for fast mining of frequent spatiotemporal patterns in environmental data. DFS_MINE, as its name suggests, uses a Depth-First-Search-like approach to the problem which allows very fast discoveries of long sequential patterns. DFS_MINE performs database scans to discover frequent sequences rather than relying on information stored in main memory, which has the advantage that the amount of space required is minimal. Previous approaches utilize a Breadth-First-Search-like approach and are not efficient for discovering long frequent sequences. Moreover, they require storing in main memory all occurrences of each sequence in the database and, as a result, the amount of space needed is rather large. Experiments show that the I/O cost of the database scans is offset by the efficiency of the DFS-like approach that ensures fast discovery of long frequent patterns. DFS_MINE is also ideal for mining frequent spatiotemporal sequences with various spatial granularities. Spatial granularity refers to how fine or how general our view of the space we are examining is.

## 1 Introduction

In this paper, we consider the problem of finding frequent patterns of change in spatiotemporal datasets. Spatiotemporal datasets store the evolution of objects over time. Figure 1 presents an example dataset that contains the temperatures in the United States The discovered patterns are sequences of events that occur most frequently. The importance of the knowledge of such patterns is obvious.

The task of discovering such frequent patterns is extremely challenging, since the search space is extremely large. The problem becomes even more challenging when the sequences to be discovered are rather long. Despite the ubiquity of spatiotemporal

C.S. Jensen et al. (Eds.): SSTD 2001, LNCS 2121, pp. 425−442, 2001.

data, the problem of mining such data has not received a lot of attention. Previous approaches include finding frequent sequential patterns in sequence data ([3], [8], [14]), finding spatial association rules ([1], [2], [7]), clustering spatial datasets ([9], [10]) or answering statistical queries in spatial datasets ([12], [13]). Most current algorithms use a Breadth-First-Search approach. This has the disadvantage of exhausting all sequences of length k before moving on to examining the sequences of length k+1. Moreover, some of the existing solutions attempt to speed up the process of discovery by minimizing the number of database scans and by storing all necessary information in main memory. When the sequences to be mined are rather long, the amount of space required is enormous.

In this paper, we present DFS_MINE, a new algorithm for discovering frequent spatiotemporal sequences. The key features of our approach are: (1) DFS-MINE uses the lattice-theoretic approach to decompose the original search space. (2) It follows the concept of Depth-First-Search, that is, it tries to discover frequent sequences of length k without exhausting all the frequent sequences of length k-1. It uses information about frequent sequences already discovered to mine sequences of greater length. It backtracks, like DFS, to sequences of smaller length when all longer sequences of the chosen lattice path turn out to be non-frequent, thus, ensuring, fast discovery of long frequent patterns. (3) DFS_MINE does not enumerate all frequent sequences in the database. It discovers very fast only the maximal frequent sequences. (4) DFS_MINE determines the support of some sequences by using the theoretical background of the lattice (5) DFS-MINE does not aim at minimizing the database scans. It performs database scans to determine the frequency of a set of sequences. Despite that fact, it achieves fast discovery of the frequent sequences thanks to its DFS-like strategy. (6) It also does not require enormous amounts of memory. It only needs minimal space to store just two structures (the list of maximal frequent sequences and the list of minimal non frequent sequences) that allow an efficient representation of the search space. Experiments prove that DFS_MINE outperforms all existing solutions, as far as both time and space are concerned, especially when the sequences to be discovered are rather long.

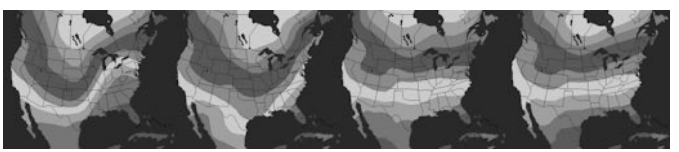


**Fig. 1.** Example spatiotemporal dataset (temperatures in the US)

Finally, we study the problem of mining spatiotemporal patterns in environmental data in various levels of spatial granularity. Spatial granularity refers to how fine or how general our view of the given space is. Whether we are mining in the very fine level of the cities or counties in the US or the very general level of the fifty states, DFS_MINE's strategy is ideal for mining in various spatial granularity levels, because, as we will show, it uses the results of the previous level in order to discover faster the frequent sequences of the next level.

The rest of this paper is organized as follows: in section 2, we define the problem of mining frequent spatiotemporal sequences in environmental data. Section 3 presents the necessary background on the concept of 'the lattice', which is used by DFS_MINE. In section 4, we present in detail the DFS_MINE algorithm. In section 5, we define the problem of mining in various spatial granularities and show how DFS_MINE is easily extended to address this problem as well. Section 6 briefly discusses related work on the subject and section 7 presents the results of the experiments we performed to evaluate DFS_MINE. Finally, we conclude in section 8.

## 2 Mining Spatiotemporal Patterns in Environmental Data

### 2.1 Definitions

The problem of mining sequential patterns in spatiotemporal data can be stated as follows: Let $A=\{A_1, A_2, ..., A_d\}$ be a set of distinct spatiotemporal attributes. For example, *atmospheric pressure P* and *temperature T* are two distinct attributes. A spatiotemporal item $I_j$ is a pair $(A_j , V_j)$ where $A_j$ is an attribute in $A$ and $V_j$ is a value assigned to it. For example $(T, 90)$ is an item, meaning that *temperature T=90F*.

A spatiotemporal itemset *IS* is a non-empty set of items of distinct attributes. An itemset is denoted $(I_1, I_2, ..., I_k)$ where $I_j$ is an item $(A_j, V_j)$. All attributes $A_j$ of the items $I_j$ must be distinct. For example, itemset $IS1=(H=60, P=1000, T=90)$ is a valid itemset, while itemset $IS2=(T=70, T=90)$ is not. An itemset with $k$ items is called *k-itemset*. A specific spatiotemporal event is a spatiotemporal itemset *IS* associated with some location $Lid_j$ and some point in time $t_k$.

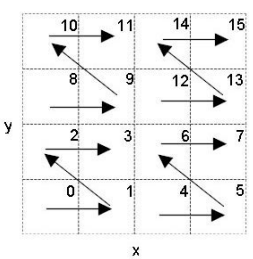



**Fig. 2.** 'Reverse-z' enumeration

A sequence $S$ is an ordered list of itemsets $IS_i$, i.e. an ordered list of spatiotemporal events, denoted as $S= (IS_1 \rightarrow IS_2 \rightarrow \ldots \rightarrow IS_n)$ where $IS_1, IS_2, ..., IS_n$ are itemsets. A sequence with $k$ items is called a *k-sequence*. For example $((T=70) \rightarrow (T=90, P=1.1))$ is a 3-sequence, which means that in a certain location, in one point in time the temperature T was 70 and some time later the temperature *T* was 90F and the pressure *P* was 1.1atm. An item can occur only once in an itemset, but it can occur multiple times in different itemsets of a sequence. In the rest, we will use the symbols: *T1, T2, …, P1, P2, …* to denote different values for temperature, pressure and humidity, etc.

A sequence $S1= (IS_1 \rightarrow IS_2 \rightarrow ... \rightarrow IS_n)$ is a *subsequence* of another sequence $S2= (IS_1 \rightarrow IS_2 \rightarrow ... \rightarrow IS_m)$, denoted as $S1 \leq S2$, if there exist integers $i1<i2<\ldots<i_n$ such that $ISi \subseteq ISi_j$ for all *ISi*. For example, sequence $(T1 \rightarrow P1T2)$ is a subsequence of $(H1T1 \rightarrow P2 \rightarrow H2P1T2)$, since the sequence elements $T1 \subseteq H1T1$ and $P1T2 \subseteq H2P1T2$. Sequence $(H1P1 \rightarrow T2)$ is not a subsequence of $(H1P1T2)$ and vice versa. A sequence *S1* is a *supersequence* of another sequence *S2*, denoted as $S1 \geq S2$, if *S2* is a

subsequence of *S1*. A specific event *E* has a unique identifier and contains a set of items. A location *L* also has a unique identifier *(location-id or Lid).* We use decimal representation for the *location-id*. The original space is partitioned in locations that are enumerated in a '*reverse-z*' manner. Figure 2 depicts the concept of the '*reverse-z*' enumeration. Each location has associated with it a list of events $\{E_1, E_2, ..., E_n\}$.

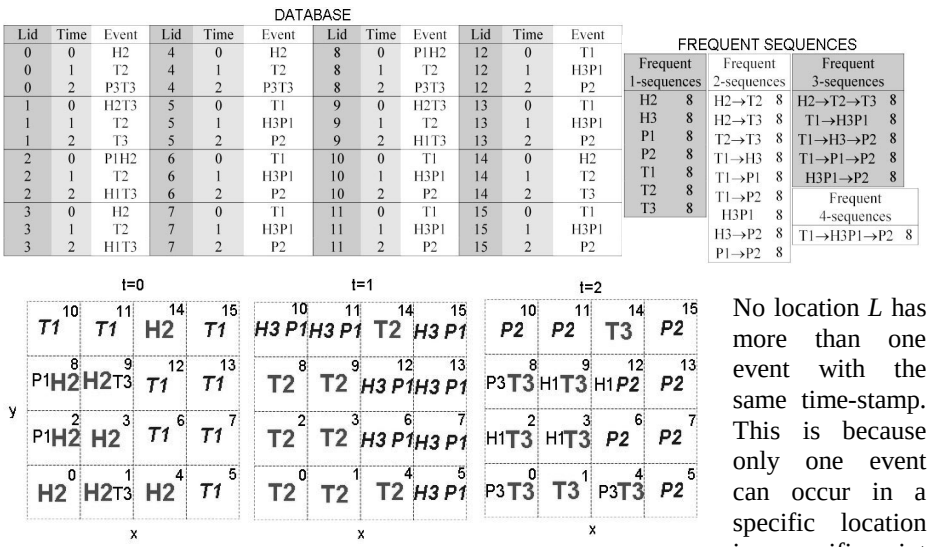



**Fig. 3.** Example Environmental Database

No location *L* has more than one event with the same time-stamp. This is because only one event can occur in a specific location in a specific point in time. The list of specific events associated with a location is sorted by time. Thus the list of events of a location is a sequence $E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_n$, called the location-sequence. A location-sequence is *L* is said to contain a sequence S, if $S \leq L$, i.e. if *S* is a subsequence of the location-sequence *L*. The *support* or *frequency* of a sequence *S*, denoted $\sigma(S)$, is the total number of times the sequence is encountered. Given a user specified threshold called *minimum support* (denoted *min_sup*), we say that a sequence is frequent if it occurs at least *min_sup* times. The set of frequent k-sequences is denoted $F_k$.

## 2.2 The Problem

Given a database *D* of location sequences and *min_sup*, the problem of mining spatiotemporal patterns is to find all frequent sequences in the database. For example consider figure 3, which presents the location sequences and the frequent sequences. The database has nine items *(H1, H2, H3, P1, P2, P3, T1, T2, T3)*, sixteen locations and three points in time. The figure also shows all the frequent sequences with *min_sup*=50% or 8 locations. The maximal frequent sequences are: $H2 \rightarrow T2 \rightarrow T3$ and $T1 \rightarrow H3P1 \rightarrow P2$.

## 3 The Lattice-Based Approach

Here we assume that the reader is familiar with basic concepts of lattice theory (see [4] for more). Zaki [14] formulated the frequent sequence mining problem as a search problem in a search space. In this section we follow his treatment. Let $P$ be a set. A partial order on $P$ is a binary relation $\leq$ on $P$ that is *reflexive: ($X \leq X$), anti-symmetric ($X \leq Y$* and $Y \leq X$ imply $X=Y$*)* and *transitive ($X \leq Y$* and $Y \leq Z$ imply $X \leq Z$, for all $X, Y, Z \in P$*)*. A partially ordered set $L$ is called a lattice if the two binary operations*:* *join* (denoted as $X \vee Y$) and *meet* (denoted as $X \wedge Y$) exist for all $X, Y \in L$. $L$ is a complete lattice if the *join* and *meet* exist for arbitrary subsets of $L$.

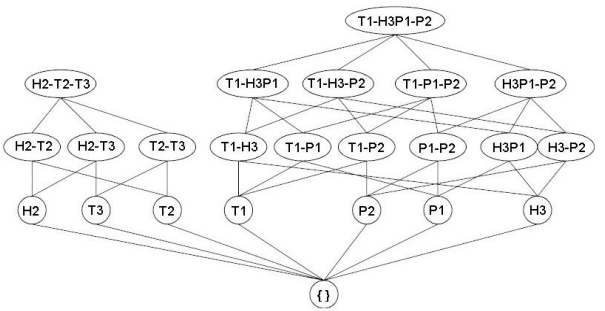



**Fig. 4.** Lattice induced by the maximal sequences T1→H3P1→P2 and H2→T2→T3

**Theorem 1 [14].** *Given a set E of events, the ordered set S of all possible sequences on the items is a complete lattice in which join and meet are given by union and intersection, respectively:* $\vee\{A_e | e \in E\} = \bigcup_{e \in E} A_e$, $\wedge\{A_e | e \in E\} = \bigcap_{e \in E} A_e$

The bottom element $\perp$ of the sequence lattice $S$ is $\perp=\{\}$, but the top element is undefined since in the abstract the sequence lattice is infinite. However, in all practical cases it is bounded and sparse. Figure 4 shows the sequence lattice induced by the maximal frequent sequences *T1→H3P1→P2* and *H2→T2→T3* for our example database. Efficient algorithms for finding all frequent sequences are based on the fact that all susequences of a frequent sequence have to be frequent.

## 4 DFS_MINE

In this section we present DFS_MINE in detail. The algorithm uses the lattice-theoretic approach to decompose the original search space. Its strategy follows the concept of Depth-First-Search: it tries to discover frequent sequences of length k+1 without exhausting all the frequent sequences of length k. The main idea is that if we discover fast a frequent k-sequence, then we do not need to waste time examining all of its subsequences, because they are certain to be frequent. A frequent k-sequence is

intersected with all frequent items to generate all candidate (k+1)-sequences, which are then scanned against the database. The only information DFS_MINE stores for each sequence is the *'useless set'* of each sequence. It also stores two structures: a list of maximal frequent sequences *(MaxFreqList)* and a list of minimal non frequent sequences *(MinNonFreqList).*

### 4.1 General

Given the minimum threshold of support *(min_sup),* we scan the database once looking for occurrences of items. The frequent items are kept in the list of frequent items *(FreqItems).* The items are inserted in *FreqItems* in a specific order, not necessarily alphabetical. Whenever we scan and use the frequent items in *FreqItems* in later steps, we always respect that order. To generate all candidate 2-sequences, we intersect all frequent items with each other in all possible combinations. This set is also scanned against the database. The frequent 2-sequences are inserted in the list of maximal frequent sequences *(MaxFreqList)* and the non-frequent 2-sequences are inserted in the list of minimal non-frequent sequences *(MinNonFreqList).* The algorithm uses the frequent 2-sequences to mine frequent patterns. The non-frequent 2-sequences are used for pruning of longer non-frequent sequences. The *Useless Set* of a k-sequence *S (S.Useless)* is a set of items that must not be intersected with k-sequence *S*.

### 4.2 Maximal Frequent Sequences List – Minimal Non-frequent Sequences List

DFS_MINE keeps a list of all maximal frequent sequences in memory. We define *'maximal frequent sequence'* as a sequence that is frequent and all of its supersequences are non-frequent sequences or have not yet been found to be frequent sequences. This list serves many purposes. First, it is the final result of the algorithm, since, at the end, it contains all maximal frequent sequences discovered in the database. Second, it is used for possibly determining whether a sequence is frequent or not. When a new candidate sequence *S* is generated, we check whether it is a subsequence of any of the sequences in *MaxFreqList*. If there exists such a sequence in the list, then, according to Lemma 1, sequence *S* is also frequent. A sequence *S* is inserted in *MaxFreqList* when all three of the following conditions hold: (1) *S* is not already in *MaxFreqList*, (2) *S* is not a subsequence of some maximal frequent sequence already in *MaxFreqList*, (3) *S* was scanned in the database and was found to be frequent. In this case, we need to insert the new sequence *S* in *MaxFreqList*. After the insertion, we scan all sequences of length less than *S.Length* and check whether they are subsequences of *S*. If so, they are removed, since they are frequent but no longer maximal. The structure is kept as a list of lists of sequences of equal length. The list of the lengths is sorted in decreasing order of length. As a result, the longest maximal sequences are kept in the beginning of the list. The reason for this is that the longer a frequent sequence is, the more likely it is to be a supersequence of some sequence *S'* which is currently under consideration. By keeping only the maximal frequent sequences, instead of all frequent sequences (maximal or not), we save a lot of

space while maintaining all the necessary information to represent the lattice of frequent sequences.

Dually, DFS_MINE also keeps a list of all minimal non-frequent sequences in memory. We define *'minimal non frequent sequence'* as a sequence that is non-frequent and all of its subsequences are frequent sequences. This list is used for pruning sequences. When a new candidate sequence *S* is generated, we check whether it is a supersequence of any of the sequences in *MinNonFreqList*, that is, if there exists any sequence in *MinNonFreqList* that is a subsequence of sequence *S*. If there exists such a sequence in the list, then, according to Lemma 1, sequence *S* is also non-frequent. A sequence *S* is inserted in *MinNonFreqList* when all three of the following conditions hold: (1) *S* is not already in *MinNonFreqList*, (2) *S* is not a supersequence of some minimal non-frequent sequence already in *MinNonFreqList* (3) *S* was scanned in the database and was found to be non-frequent. In this case, we need to insert the new sequence *S* in *MinNonFreqList*. After the insertion, we scan all sequences of length greater than *S.Length* and check whether they are supersequences of *S*. If so, they are removed, since they are non-frequent but no longer minimal. The structure is kept as a list of sequences sorted in increasing order of length.

### 4.3 Generating Sequences

DFS_MINE generates (k+1)-sequences by using a k-sequence and intersecting it with all frequent items $I_j$ that are in *FreqItems* but not in the useless set of k-sequence $S$ ($I_j \in FreqItems - S.Useless$). The intersection of an item $I_j$ with a k-sequence *S* involves inserting the item in all possible positions in the k-sequence. We use the symbol $SET(S+I_j)$ to express the set of candidate (k+1)-sequences that result from the intersection of k-sequence *S* with item $I_j$. There are two kinds of resulting sequences resulting: (a) the sequences in which the item is inserted as part of an existing itemset and (b) the sequences in which the item is inserted as a separate itemset. An itemset cannot contain more than one copy of the same item. As a result, some of the resulting candidate sequences are rejected. Figures 5a and 5b present an example. By performing intersection in this way though, we may generate duplicate sequences. Figure 5c presents a possible situation: from the same 3-sequence $A \rightarrow A \rightarrow A$, we can get the same 5-sequence $AD \rightarrow A \rightarrow AD$ through two different paths. To avoid generating duplicate sequences, we redefine the way intersection is performed.

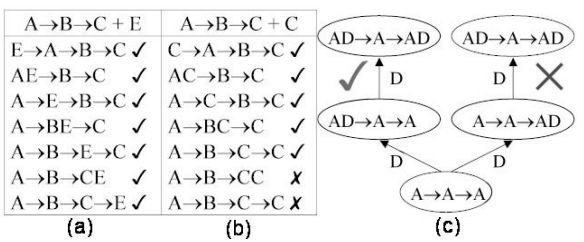



**Fig. 5.** Generating Sequences

**Definition 1.** *Intersection of k-sequence S with item I*

When we intersect sequence *S* with item *I*, we insert item *I* in all possible positions that follow its rightmost occurrence. If the item does not occur at all in the sequence, then it is inserted in all positions.

In figure 5c, inserting *D* only in positions following its rightmost occurrence prevents the generation of duplicates.

### 4.4 Examining the Candidate Sequences Generated

After intersecting sequence *S* with item $I_j$ and generating each new (k+1)-sequence *S'*, we check *MinNonFreqList*, looking for any minimal non-frequent sequence that may be subsequences of *S'*. If there exists such a sequence in *MinNonFreqList*, then apparently *S'* is non-frequent as well, it is pruned directly and removed from $SET(S+I_j)$. Otherwise, we check *MaxFreqList*, looking for any maximal frequent sequence that may be a supersequence of *S'*. If there exists such a sequence in *MaxFreqList*, then apparently *S'* is frequent as well, and does not need be scanned in the database because it is already determined to be frequent. It remains, though, in $SET(S+I_j)$, because it may generate longer frequent sequences. If no sequence in *MaxFreqList* was found to be a supersequence of sequence *S'* then it remains in $SET(S+I_j)$ and it has to be scanned against the database.

### 4.5 List of Candidate Sequences – *candList(S)*

We use the symbol $candList(S) = \bigcup_{Ij \in FreqList - S.Useless} SET(S + I_j)$ to express the set of all sets $SET(S+I_j)$.. In order to store *candList(S)* in memory we use a 'list of lists' structure. Each node of the list is associated with an item $I_j$ and $SET(S+I_j)$. This structure is used for determining more items to be inserted in the *useless set* of the sequence. The process will be explained in detail in a later subsection.

### 4.6 Adding Items to *S.Useless*

When intersecting k-sequence *S* with all items $I_j$ in *FreqItems-S.Useless*, each iteration *j* produces $SET(S+I_j)$. At the end of each iteration, after $SET(S+I_j)$ has been generated, item $I_j$ is inserted in the *Useless set* of the sequence *S*, because all possible candidate sequences that can result from the intersection have already been produced. As a result, during iteration *j*, when item $I_j$ is intersected with sequence *S*, the *Useless set* of sequence *S* contains all items $I_k$ with $j>k\geq 1$. This is very important for avoiding duplicates.

### 4.7 Inheriting the Useless Set

The *Useless set* of a k-sequence *S* contains important information that must not be ignored when we examine each one of the (k+1)-sequences that k-sequence *S* generated. The *Useless set* of k-sequence *S* is *'inherited'* by the (k+1)-sequences gen-

erated by *S* for two reasons: (1) to avoid generation of duplicates and (2) to avoid generation of sequences that are certain to be non-frequent.

### 4.7.1 Inheritance against Duplicate Sequences

Assume a k-sequence *S* that is intersected with item *A* which produces *SET(S+A)*. Then each (k+1)-sequence in *SET(S+A)* is intersected with item *B*. We use the symbol *SET(S+A,B)* to express the set of these candidate (k+2)-sequences. Assume now we intersect the k-sequence *S* first with *B* to get *SET(S+B)*, and then intersect each one of the resulting (k+1)-sequences with *A* to get *SET(S+B,A)*. Apparently, *SET(S+A,B)= SET(S+B,A)*. The reason for this is the way that (k+1)-sequences are generated from a k-sequence and an item, by inserting the item in all possible positions in the k-sequence. As a result, the two sets should not be examined twice. Figure 6 presents this argument visually. To avoid the above scenario, when intersecting a k-sequence *S* with an item $I_j$ in *FreqItems-S.Useless* to produce the $SET(S+I_j)$, then all items $I_k$ with $j>k\geq 1$ that were before $I_j$ in *FreqItems* must not be intersected with any of these (k+1)-sequences in $SET(S+I_j)$. For this reason, the useless set of each one of the (k+1)-sequences in $SET(S+I_j)$ must contain all items $I_k$ with $j>k\geq 1$, To achieve this, we force all (k+1)-sequences of $SET(S+I_j)$ to inherit *S.Useless* from k-sequence *S* when they are created, since, during each iteration *j*, all items $I_k$, with $j>k\geq 1$, are in the *S.Useless*

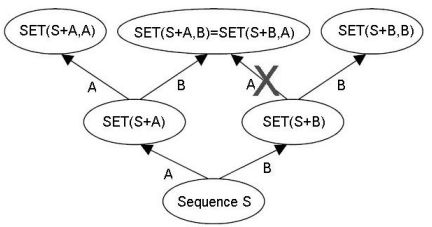



**Fig. 6.** Sequences generated more than once

### 4.7.2 Inheritance against Non-frequent Sequences

Assume we are intersecting 2-sequence $S=A\rightarrow B$ with items *D* and *E*. Figure 7 shows all the 3-sequences that are produced. Column (a) shows the 4-sequences produced from the intersection of *S* with *E*. Column (b) shows the 3-sequences produced from the intersection of *S* with *D*. Let's assume that all sequences of column (a) turn out to be non-frequent and that some of the sequences in column (b) are frequent. Assume we are now examining the frequent sequence $S'=D\rightarrow A\rightarrow B$ of column (b) and we intersect it with item *E*. We get the sequences in column (c). But as it can easily be seen from figure 7, all 4-sequences of column (c) have some 3-sequence of column (a) as a subsequence and cannot be frequent. As a result, *E* is a useless item for all the other

| A→B + E | A→B + D | D→A→B + E |
|---|---|---|
| E→A→B | *D→A→B* | **E→D→A→B** |
| AE→B | AD→B | **DE→A→B** |
| A→E→B | A→D→B | **D→E→A→B** |
| A→BE | A→BD | **D→AE→B** |
| A→B→E | A→B→D | **D→A→E→B** |
| Non-frequent | | **D→A→BE** |
| (a) | (b) | **D→A→B→E** (c) |

**Fig. 7.** Generated Sequences

(k+1)-sequences. To summarize, if $SET(S+I_k)= \emptyset$, for some item $I_k$, then item $I_k$ is inserted in the useless set of all (k+1)-sequences in all the other sets $SET(S+I_j)$.

### 4.8 Correctness of the DFS_MINE Algorithm

We present two theorems that ensure the correctness of the DFS_MINE algorithm. Their proofs are included in the full version of the paper.

**Theorem 2.** *DFS_MINE does not generate any sequence twice.*

**Theorem 3.** *DFS_MINE does not miss any frequent sequence.*

### 4.9 DFS_MINE: Implementation – Putting It All Together

```
DFS_MINE(min_sup, D)
  FreqItems={Frequent Items or 1-sequences};
  Freq2Seqs={Frequent 2-sequences};
  //insert all non frequent 2-sequences in MinNonFreqList
  MinNonFreqList={all non frequent 2-sequences};
  MaxFreqList={ };
  //calling DFS on the frequent 2-sequences
  For all sequences S in Freq2Seqs do
    DFS_VISIT(S, MaxFreqList, min_sup, D);
```

**Fig. 8.** Pseudocode for DFS_MINE algorithm

```
DFS_VISIT(S, MaxFreqList, min_sup, D)
  CandList={ };
  //generate sequences from the intersection of S with Ij
  For all frequent items Ij ∈ FreqItems-S.Useless
    GenerateSequences(S, Ij , SET(S+Ij) );
    CandList=CandList ∪ SET(S+Ij);
    S.Useless=S.Useless ∪ {Ij};
    If (there are sequences is CandList to be scanned)
        ScanSeqs(D, min_sup, CandList);
    MoreUseless={ };
    For all sets of sequences SET(S+Ij) in CandList
       //checking for more useless items
      if (SET(S+Ij))==∅,  MoreUseless=MoreUseless ∪ {Ij};
      else for all sequences Si in SET(S+Ij)
         //if frequent, insert it in MaxFreqList
         if (Si.freq>=min_sup)MaxFreqList=MaxFreqList∪{Si};
         //if not, insert it in MinNonFreqList
         else  MinNonFreqList=MinNonFreqList ∪{Si};
    //call DFS_VISIT recursively on all frequent sequences
    For all sequences Si in CandList
       Si.Useless=Si.Useless ∪ MoreUseless;
       DFS_VISIT(Si, MaxFreqList, min_sup, D);
```

**Fig. 9.** Pseudocode for function DFS_VISIT

Figure 8 shows the pseudocode for the DFS_MINE algorithm. Figure 9 shows the pseudocode for function *DFS_VISIT*. This is the function that performs the recursive DFS-like mining process. Sequence *S* is intersected with all items in *FreqItems-S.Useless*. We create *candList* to store the sets $SET(S+I_j)$. After each iteration *j*, item $I_j$ is inserted in *S.Useless*. If there are any $SET(S+I_j)=\emptyset$, item $I_j$ is an additional useless item that needs to be inherited by all the resulting (k+1)-sequences. We keep all these items in list *MoreUseless*. We scan against the database all sequences in *candList* that are not known to be frequent. We update *MaxFreqList* and *MinNonFreqList* accordingly, in case we have discovered some maximal frequent or minimal non-frequent sequence respectively. Each sequence S' in all $SET(S+I_j)$, for all $I_j$, inherits all the items in *MoreUseless*. Finally, we call function *DFS_VISIT(S')* on each one of them to continue the recursive DFS-like process.

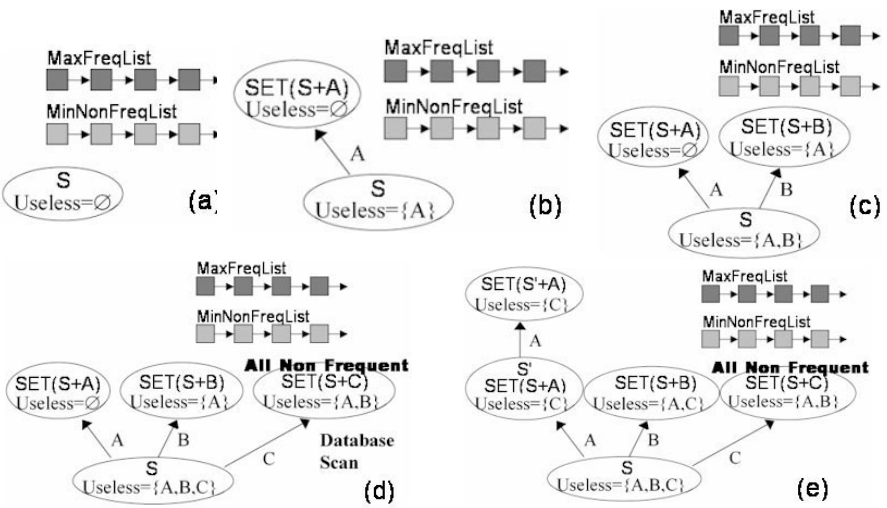



**Fig. 10.** DFS_MINE example

### 4.10 DFS_MINE Example

In this subection we present an example of how the algorithm works. Let's assume 3 frequent items *(A,B,C)* and a k-sequence *S* with $S.Useless=\emptyset$. *MaxFreqList* and *MinNonFreqList* are also available. We intersect with *A* to get *SET(S+A)* which inherits $S.Useless=\emptyset$. We check *MaxFreqList* and *MinNonFreqList* for each sequence in *SET(S+A)* to determine whether it is frequent or not. Item *A* is added to *S.Useless* to reflect the fact that the intersection with *A* has been completed. The same process is repeated for *B* and *C* and *S.Useless* is updated each time. When *S* has been intersected with all items, we scan the database for the sequences whose support is unknown. Let's assume that all sequences in *SET(S+C)* turn out to be non-frequent. Item *C* is then inserted in the *Useless set* of all sequences in *SET(S+A)* and *SET(S+B)* to reflect

the fact that the intersection with *C* will yield only non-frequent sequences. *MaxFreqList* and *MinNonFreqList* are updated accordingly with any new maximal frequent or minimal non-frequent sequences, respectively, that may have been discovered in this step. We then pick the first (k+1)-sequence *S'* in *SET(S+A)* and repeat the same process. All steps of the procedure described above can be seen in figures 10a through 10e.

# 5 Mining Frequent Spatiotemporal Sequences with Various Spatial Granularities

In this section, we are studying the problem of mining spatiotemporal sequences in various spatial granularities. So far, we have been studying the mining of frequent sequences in the given space. In the example database of figure 2, the given area had 16 distinct locations. It is very interesting, though, to generalize spatially the results we have found so far and mine frequent sequences in a higher level of spatial granularity. For example, we may have environmental data for all cities in the US. As a first step, we can discover frequent sequences, by mining the data considering each city independently from the others. But it is also interesting to examine the problem more generally by mining frequent sequences in the counties of the US, or even more generally, in the states of the US. We say that mining sequences by city is *mining in level 0 of spatial granularity*, mining sequences by county is *level 1* and *mining by state* is *level 2*. Therefore, the concept of *spatial granularity level* can be defined as *the number of times we have generalized the original space we are examining*. The way to change spatial granularity level is by joining certain subregions into one region and by considering the data that belonged to each subregion to belong to the new region we just created. We can apply the same algorithm (with certain modifications) in order to determine frequent sequences in these greater regions. The sequences that were found to be frequent in a lower level will still be frequent. In addition to these, we will show that other sequences, which were not frequent before, may also turn out to be frequent.

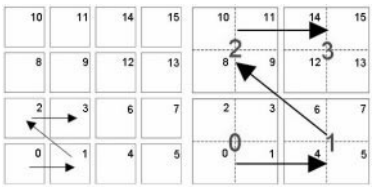
## 5.1 The Problem

Given a database *D* of environmental data, a minimum support threshold *min_sup* and the level of spatial granularity *gran_lev*, the problem of mining frequent spatiotemporal patterns in various spatial granularities is to find all frequent sequences in the database for each level of spatial granularity, from level 0 to level *gran_lev*.

The key observation in mining in various spatial granularities is that the sequences that are frequent in level *j-1* will also be frequent in level *j*. Also, sequences that were not frequent in level *j-1*, may now turn out to be frequent in level *j*. As a result, when moving in a higher level of spatial granularity, we can benefit a lot by using the frequent sequences that we just discovered in level *j-1* as starting points and build on them to discover even longer frequent sequences in level *j*. DFS_MINE's strategy is ideal for taking advantage of that observation, exactly because it forms candidate (k+1)-sequences by intersecting a k-sequence with all frequent items in all possible positions in the k-sequence.

### 5.2 Joining Subregions

To create a new region we join subregions together. We assume that we always join four subregions. We also assume that the original space contains a number of original regions that is a power of four. To join subregions into one, we use the following formula $Lid_k^j = \left\lfloor \frac{Lid_i^0}{4^j} \right\rfloor$ for *i=0,1,2,…N*, where *i* stands for the *i-th* distinct location-id of spatial granularity level 0 and *N* is the total number of locations we have initially. $Lid_k^j$ stands for the *k-th* distinct location-id of spatial granularity level *j*. As a result, *k=0,1,2,…* $N / 4^j$. Also, *j=0,1,2…gran_lev*. By using the above formula, we can calculate the *location-ids* of the new regions that we have in level *j*. For example, figure 11a shows the original space in the example database with the 16 distinct locations. Figure 11b shows how these 16 distinct locations are joined into groups of four to form the new four regions. Figure 11b also shows that the *'reverse-z'* enumeration is preserved.




(a) Level 0 (b) Level 1

**Fig. 11.** Two different spatial granularities

### 5.3 Implementation

As we said, the environmental data does not change; it is just its locality that changes. By joining four subregions of level *j-1* into a new greater region in level *j*, the events that occur in each one of these subregions are now considered to occur in the same location. As a result, all events of those four subregions need to be included in the location sequence of the new greater region. The location sequence of the new region

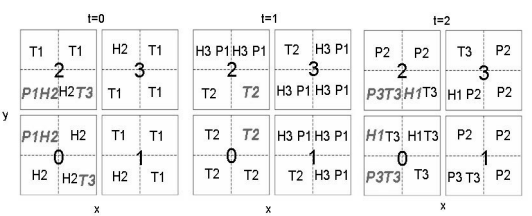



**Fig. 12.** Mining in level 1

| Lid | Location Sequence |
|---|---|
| 0 | H2P1T3→T2→H1P3T3 |
| 1 | H2T1→H3P1T2→P2P3T3 |
| 2 | H2P1T1T3→H3P1T2→H1P2P3T3 |
| 3 | H2T1→H3P1T2→H1P2T3 |

**Fig. 13.** Location sequences in level

| |
|---|
| H2T1→H3P1T2→H1P2T3 |
| H2T1→H3P1T2→H1P3T3 |
| H2P1T3→T2→H1P3T3 |

**Fig. 14.** Maximal Frequent Sequences in level 1

$Lid_k^j$ will contain all events that are occurring in any of the regions for which the formula $\left\lfloor \frac{Lid_i^0}{4^j} \right\rfloor$ yields the same value. Now that we are merging the events of the

four subregions together, we may have in the same itemset more than one items with the same attribute but with different values, which was against the valid definition of itemset (section 2). We need to relax this requirement and allow a location sequence to consist of itemsets containing items with the same attribute but with different values for the same attribute. Figure 12 shows the original space when we mine in spatial granularity level=1. Figure 13 shows all new four location-sequences.

As we mine in a higher level of granularity, the number of locations decreases by a factor of 4. *min_sup*, of course, does not change as a percentage. But the actual number of locations that an event must occur in to be considered frequent changes. For example, in the example database of figure 12, in level 0 with *min_sup=50%*, event *P3* occurred in only 3 out of the 16 locations and was not frequent. But in level 1 with *min_sup=50%*, the same event *P3* occurs in 3 out of the 4 locations, so it is frequent. This is an event that must from now on be intersected with frequent sequences to produce longer candidate sequences. For this reason, in level 0, when we scan the database for the first time, to discover the frequent items, we keep all items in list *FreqItems*, even the ones that are not frequent in level 0, because they may become frequent in higher levels. In each level we only use those items that have support greater than *min_sup*.

As we have said, the main 1 idea is to discover even longer frequent sequences in level *j* by using the frequent sequences of level *j-1* as starting points and by building on them. As a result, when mining in level *j*, we call function DFS_VISIT on all maximal frequent sequences of level *j-1* and intersect them with all frequent items to produce all candidate sequences of greater length. This way, it is much faster to discover even large frequent sequences in the current level. Moreover, at the end of level *j-1*, *MinNonFreqList* contains all the minimal non-frequent sequences of level *j-1*. When we move to level *j*, some of these minimal non-frequent sequences may now be frequent, for the same reasons that individual items may now be frequent. For this reason, we scan all sequences in *MinNonFreqList* against the database and determine the frequent ones. These along with the maximal frequent sequences of the previous level *j* will be the starting points for the mining process.

Finally, we must clear the *Useless set* of each sequence *S* of the sequences of level *j-1* that will be used as starting points in level *j*. This is because the whole process starts all over again and, as a result, even the items that were in *S.Useless* may now yield longer frequent sequences in level *j*.

### 5.4 Putting It All Together in DFS_MINE

We have shown what modifications are necessary to mine in multiple levels. Everything else remains the same. In the main function, we include a for-loop to implement the repeated procedure of mining in each level from level 0 to level *gran_lev*. *MaxFreqList* is the list of maximal sequences of the current level. *SpatialList* contains the maximal frequent sequences of each level. When mining in level *j-1*, we store the maximal frequent sequences in *MaxFreqList*. After finishing level *j-1*, we insert *MaxFreqList* in *SpatialList*, clear *MaxFreqList* and move to level *j*. In level *j*, we scan in the database the sequences of *MinNonFreqList* of the previous level *j-1* to determine which ones of those are now frequent. We will use as starting points of level *j* the maximal frequent sequences *(MaxFreqList)* of the previous level and the sequences of *MinNonFreqList* that were

now found to be frequent. Finally, we need to slightly modify function *DFS_VISIT* too, as we need to pass the current level of spatial granularity as an input parameter.

## 6 Related Work

In this section, we briefly discuss some related work that has been done in the field ([1], [2], [3], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]). One approach is given by Agrawal-Srikant in [1] and by Mannila et al. in [8]. Other mining techniques for spatial data are given in [12] and [13] but they do not consider evolution in time. Similarly, the spatial association rules introduced in [9] also mainly consider data that does not change in time. Techniques for mining spatial and temporal data were also given by [5], [6], [11].

Out of the very few approaches proposed so far, SPADE[14] seems to be the most efficient one. Its key features are: (1) it uses a vertical id-list database format, where each sequence is associated with a list of locations in which it occurs along with the time stamps. The id-lists for each sequence are kept in main memory. (2) It uses a lattice-theoretic approach to decompose the original search space (lattice) into smaller lattices (equivalence classes $[X]_{\theta k}$) which can be processed independently in main-memory. (3) To generate a (k+1)-sequence, SPADE intersects two k-sequences with the same (k-1)-length prefix (4) all frequent sequences can be enumerated via simple id-list intersections. (5) SPADE minimizes I/O costs by reducing database scans.

## 7 Experimental Results

In this section, we compare the performance of our DFS_MINE with SPADE. SPADE was implemented exactly as described in [11]. Experiments were performed on a Quad Xeon 550Mhz processor with 2GB RAM.

### 7.1 Synthetic Datasets

| Name | Size | C | S | I | D | Length 75% | Length 50% |
|---|---|---|---|---|---|---|---|
| A | 56K | 5 | 2 | 2 | 1000 | 3 | 4 |
| B | 3.99M | 5 | 2 | 2 | 50000 | 6 | 6 |
| C | 7.5M | 5 | 2 | 2 | 100000 | 5 | 5 |
| D | 30.6M | 5 | 2 | 2 | 500000 | 6 | 6 |
| E | 91K | 5 | 2 | 3 | 1000 | 7 | 10 |
| F | 101K | 5 | 2 | 4 | 1000 | 9 | 9 |
| G | 134K | 5 | 3 | 2 | 1000 | 10 | 13 |
| H | 137K | 5 | 3 | 3 | 1000 | 14 | 16 |
| I | 174K | 5 | 3 | 4 | 1000 | 15 | 17 |
| J | 166K | 5 | 4 | 2 | 1000 | 12 | 20 |
| K | 165K | 5 | 4 | 3 | 1000 | 12 | 23 |
| L | 150K | 10 | 4 | 4 | 1000 | 18 | 18 |
| M | 83K | 10 | 2 | 2 | 1000 | 3 | 8 |
| N | 179K | 10 | 3 | 3 | 1000 | 16 | 21 |

**Fig. 15.** Synthetic Databases

The synthetic datasets mimic real world scenarios, in which locations have various values for attributes such as temperature, atmospheric pressure, humidity etc. The datasets are generated using the following process. First *NI* maximal itemsets of average size *I* are generated by choosing from *N* items. Then *NS* maximal sequences of average size *S* are created by assigning itemsets from *NI* to each sequence. Next, a location, in which an average number of *C* events (*C* points in time) occur, is created. Sequences in *NS* are assigned to different locations. We make sure at least

one sequence has support 75% and one sequence has support 50%. The generation stops when *D* locations have been created. We set *NS*=5000, *NI*=25000, *N*=10000. Figure 15 shows the datasets with their parameter settings. The columns for 'Length 75%' and 'Length 50%' specify the length of the longest sequence that had support at least 75% and 50% respectively.

### 7.2 Comparison of DFS_MINE with SPADE

Figure 16 shows the results for DFS_MINE and SPADE. *'OOM'* stands for *'out of memory'*. Figures 17a and 17b compare DFS_MINE with SPADE on the synthetic datasets for two values of minimum support: 75% and 50%. Both of these graphs are in logarithmic scale. Figures 17a and 17b clearly indicate that the performance gap increases as the length of the sequence to be mined increases. The reason for this is the DFS-like strategy that DFS_MINE uses. As soon as a k-sequence is found, DFS_MINE intersects it with all frequent items and generates all candidate (k+1)-sequences, in an attempt to discover a frequent (k+1)-sequence. DFS_MINE, exactly because it works in DFS-like manner, does not enumerate all k-sequences before moving on to the (k+1)-sequences. This way, it skips examining all subsequences of some maximal frequent sequence, because they are certain to be frequent too, and achieves faster discovery of longer maximal sequences. DFS_MINE does not store the occurrences of sequences in id-list. Instead, it performs database scans each time a new sequence is discovered. That obviously costs time because of I/O, but its DFS-like approach that discovers long maximal frequent sequences much faster compensates for the time spent on database scans.

| DFS_MINE | | | | | SPADE | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | 75% | Space | 50% | Space | Name | 75% | Space | 50% | Space |
| A | 0.6 | 500K | 1.77 | 550K | A | 0.54 | 800K | 1.34 | 900K |
| B | 117.99 | 500K | 236.84 | 550K | B | 86.03 | 10M | 157.51 | 33M |
| C | 181.33 | 500K | 357.33 | 600K | C | 132.42 | 14M | 272.21 | 70M |
| D | 892.85 | 550K | 1061.52 | 675K | D | 693.18 | 180M | 911.28 | 245M |
| E | 4.59 | 1M | 15.89 | 1M | E | 3.06 | 4M | 13.04 | 8M |
| F | 8.16 | 1M | 13.71 | 1.5M | F | 6.75 | 8M | 10.98 | 9M |
| G | 12.72 | 1M | 147.68 | 1.5M | G | 11.65 | 5M | 177.45 | 50M |
| H | 103.36 | 1.4M | 1932.35 | 1.8M | H | 185.89 | 60M | 7173.6 | 300M |
| I | 194.2 | 2M | 5331.89 | 3M | I | 476.21 | 93M | 21016.88 | 580M |
| J | 32.82 | 1.6M | 23823.91 | 1.2M | J | 37.11 | 13M | OOM | 1.5G |
| K | 35.39 | 1.2M | 58739.65 | 2.5M | K | 39.05 | 14M | OOM | 1.5G |
| L | 2224.58 | 3M | 7866.46 | 3M | L | 17979.62 | 645M | 77166.65 | 995M |
| M | 0.74 | 1M | 6.56 | 1M | M | 0.62 | 3M | 5.81 | 4M |
| N | 479.5 | 1.4M | 31253.67 | 1.8M | N | 1582.72 | 172M | OOM | 1.5G |

**Fig. 16.** Experimental results for DFS_MINE and SPADE

On the other hand, SPADE uses a DFS-like approach only to a very short extent, since it only decomposes the original lattice to the $[X]_{\theta 1}$ equivalence classes. Apart from that, it uses a BFS approach and examines all sequences of length *k* before moving on to the sequences of length *k+1*. This is clearly inefficient, because SPADE takes too long before discovering long frequent sequences.

In general, the sequence problem is CPU bound, rather than I/O bound. This is due to the fact that the number of id-list intersections we need to perform (assuming we use SPADE) is exponential to the number of frequent items we have in the dataset. On the other hand, the cost of performing a database scan each time to determine the support of a sequence (assuming we use DFS_MINE) is only linear to the number of

frequent items in the database. As a result, as the frequent sequences grow longer, scanning the dataset, however large that may be, becomes less expensive than performing all possible id-list intersections between k-sequences.

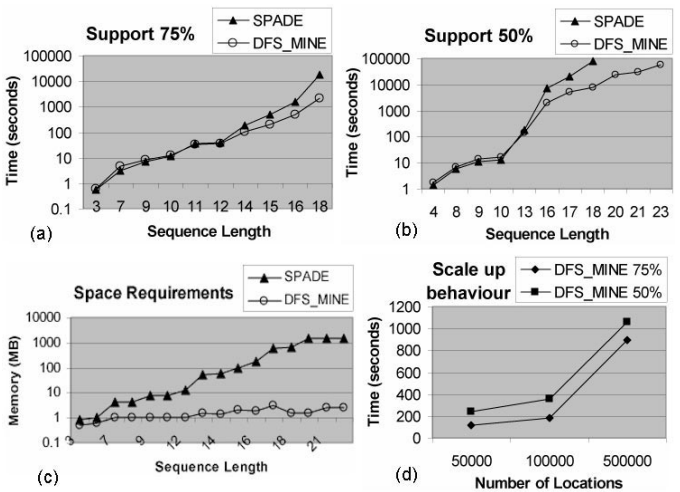



**Fig. 17.** Comparison between DFS_MINE and SPADE

As the figures show, for large datasets (more than 10 MB) and rather short maximal frequent sequences (of length approximately 3-6), SPADE performs better than DFS_ MINE. This is due to the fact that the sequences are rather short. As a result, the cost of the id-list intersections that SPADE performs is less than the cost of scanning a large dataset, which is what DFS_MINE does. We experimented with equally large datasets (more than 10MB) that contained much longer maximal frequent sequences (of length approximately 25-30) and SPADE ran out of memory.

This is another very important observation. As far as space is concerned, DFS_MINE is much more efficient than SPADE, as figure 17c shows. This is because DFS_MINE, inlike SPADE, does not need to store any significant amounts of information for each sequence. Our experiments showed that SPADE may require up to 1.5GB of memory. DFS_MINE, on the other hand, rarely did it use more than 5MB of memory. Figure 17c is also in logarithmic scale.

Finally, as far as scaling is concerned, we performed a set of experiments where we increased the number of locations from 50000 to 500000. Figure 17d presents the results. As it can be seen, DFS_MINE performs really well even in this aspect. When the number of locations increases from 50000 to 100000 (a factor of 2), the required time increases by a factor much less than 2. Moreover, when the number of locations increases from 100000 to 500000 (a factor of 5), the required time increases by much less than a factor of 5.

## 8 Conclusions

In this paper, we presented DFS_MINE, a new algorithm for fast mining of frequent spatiotemporal patterns in environmental data. First, we defined the problem of mining frequent spatiotemporal sequences. We then discussed in detail our solution: DFS_MINE, It uses a DFS-like approach to the problem, which allows very fast

discoveries of long sequential patterns. DFS_MINE does not enumerate all frequent sequences. Instead, it aims at discovering only the maximal frequent ones. When a maximal frequent sequence is discovered, we do not need to examine all of its subsequences, since they are certain to be frequent. DFS_MINE performs database scans discover frequent sequences rather than relying on information stored in main memory. This has the advantage that the amount of space required is minimal. We also defined the problem of mining spatiotemporal sequences in various spatial granularities. We defined the concept of spatial granularity and showed that DFS_MINE's strategy of using the results of the previous level to discover faster the frequent sequences of the next level is ideal for addressing this problem as well. The experiments that we performed showed that the I/O cost of the database scan that DFS_MINE performs is offset by the efficiency of the DFS-like approach that ensures fast discovery of long frequent patterns. DFS_MINE discovered long maximal frequent sequences faster than SPADE. The experiments also showed that DFS_MINE outperformed SPADE even as far as space requirements are concerned. DFS_MINE also had excellent scale-up properties with respect to the size of the database.

## References

1. R. Agrawal, R. Srikant: Fast Algorithms for Mining Association Rules, 20th VLDB, Santiago, Chile, Sept. 1994
2. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo: Fast discovery of association rules. Advances in Knowledge Discovery and Data Mining 1996
3. R. Agrawal, R. Srikant: Mining Sequential Patterns, Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT), Avignon, France, March 1996
4. B.A. Davey, H.A. Priestley: Introduction to lattices and Order. Cambridge University Press, 1990
5. Klosgen, W., 1995. Deviation and association patterns for subgroup mining in temporal, spatial, and textual data bases. Proc. First International Conference on Rough Sets and Current Trends in Computing, RSCTC`98, 1-18, Springer-Verlag, Berlin,.
6. Klosgen, W., 1998. Subgroup mining in temporal, spatial and textual databases. Proc. International Symposium on Digital Media Information Base, 246-261, World Scientific, Singapore.
7. K. Koperksi, J. Han: Discovery of Spatial Association Rules in Geographic Information Databases, 4th Int'l Symp. on Large Spatial Databases (SSD95), Maine, Aug. 1995.
8. H. Mannila, H. Toivonen, A. I. Verkamo: Discovery of frequent episodes in event sequences. Report C-1997-15, University of Helsinki, Department of Computer Science, February 1997.
9. R. T. Ng, J. Han: Efficient and Effective Clustering Methods for Spatial Data Mining. VLDB 1994
10. J. Sander, M. Ester, H-P Kriegel, X. Xu: Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications.
11. Stolorz, P., et al., 1995. Fast Spatio-Temporal Data Mining of Large Geophysical Sets. Proc. First International Conference on Knowledge Discovery and Data Mining, Montreal, Canada, 300-305, AAAI Press.
12. W. Wang, J. Yang, R. Muntz: STING: A Statistical Information Grid Approach to Spatial Data Mining, 23rd VLDB Conference, Athens, Greece, Aug 1997.
13. W. Wang, J. Yang, R. Muntz: STING+: An Approach to Active Spatial Data Mining, International Conference on Data Engineering (ICDE-99), Australia, March, 1999.
14. M. Zaki: Efficient Enumeration of Frequent Sequences, Machine Learning Journal 2001