DISCOVERING PREVALENT CO-LOCATION PATTERN IN DIFFERENT DENSITY SPATIAL DATA WITHOUT DISTANCE THRESHOLDS

Final Year Project Final Report

Tran Duy Hai

Le Anh Thang

Ha Trong Nguyen

A thesis submitted in part fulfilment of the degree of BSc. (Hons.) in Computer Science with the supervision of M.S.E Le Dinh Huynh

Bachelor of Computer Science

Hoa Lac campus - FPT University

09 September 2023

Acknowledgement

The last four months are a time that every student has to go through. This is also the last time for students to work together in groups with the dedicated guidance of teachers under this beloved school. Firstly, we would like to thank our instructor, Mr. Le Dinh Huynh, Mr. Tran Van Ha for their enthusiastic guidance over the past time, helping us to complete this thesis. Secondly, we also thank FPT University for giving us a good learning environment for us to study and develop well. Finally, we always remember the encouragement to us to improve ourselves every day.

Abstract

A prevalent spatial co-location pattern (PSCP) refers to a group of different features that their instances occur frequently within a spatial neighborhood. The neighbor of instances is typically evaluated based on the spatial separation between them. If the spatial separation is not greater than a threshold value set by users, they are considered to be neighboring each other. However, determining an appropriate distance threshold for each specific spatial dataset is challenging for users, as it requires careful analysis of the dataset. To address the issue, we propose an algorithm called Delaunay triangulation k-order clique (DTkC) to discover PSCPs without distance thresholds. This algorithm integrates three phases: the spatial neighbor hierarchy structure of instances is created by Delaunay triangulation, employing k-order neighbors allows users to select an appropriate level from the neighbor structure, a clique-based approach is designed to store compactly neighboring instances and quickly collect co-location instances of each candidate pattern to filter PSCPs. We conducted experimental analysis on both synthetic and real-world datasets, to demonstrate the effectiveness of the DTkC algorithm in terms of generating the number of PSCPs, execution time, and memory consumption.

Keywords: Prevalent spatial co-location pattern, Delaunay triangulation, k-order neighbos, Cliques.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 The Problem

Data analysis in the spatial domain is a crucial area in data mining where prevalent spatial co-location patterns (PSCPs) play a significant role as they exhibit clear associations of features among the studied objects in geographical space. Particularly, we not only investigate individual features of objects but also explore the relationships among multiple objects in space, their combinations and organization, which contain valuable hidden information that needs to be extracted to gain a deeper understanding of the overall spatial structure. Examining these associations can help us identify the general rules of these features, aiding in forecasting and making informed decisions while optimizing resource management and allocation. For example, when investigating the underlying causes of a specific issue, such as in the healthcare domain where the aim is to establish specialized healthcare facilities targeting specific diseases, an analysis of PSCPs becomes essential [18]. By studying PSCPs within residential areas, we can explore the associations between different diseases and the daily life habits of the population. These patterns provide valuable insights for decision-making regarding the development or enhancement of healthcare facilities in specific geographical locations while minimizing resource wastage. PSCPs are highly effective in various other domains such as criminology [3], public safety [4], business [10], disease control [5], transportation [14], and so on.

Most of the proposed PSCP mining algorithms use a distance threshold to identify neighbor relationships between spatial instances. Using a minimum distance threshold requires users to determine an appropriate threshold value. If this value is not carefully chosen, it can lead to overlooking important PSCPs or generating meaningless PSCPs. For example, important PSCPs are missed as a result of setting a small distance threshold, while a large value of that creates excessive computations, memory consumption, and too many redundant PSCPs.

Additionally, the minimum distance threshold can be influenced by data density. In high-density areas, a small value of the distance threshold can yield a large number of neighboring instances, while in low-density areas, it may lead to the omission of neighboring instances (even none neighbor relationship is formed). If the distance threshold is set to a large value, the neighbor relationship between instances is suitably constructed in the low-density areas. However, most instances in the high-density areas form neighbor relationships, many inappropriate neighbor relationships are constructed. This issue makes the process of complex spatial datasets very challenging.

Moreover, the traditional PSCP mining algorithms, which follow a generation test candidate framework [18,19,11,12,15] that is similar to the Apriori algorithm [6], are difficult to deal with when there are a large number of neighboring instances formed in the dataset. Since these algorithms first need to generate a set of candidates, then they collect all co-location instances of each candidate by generating groups of instances and verifying the neighbor relationship of these instances in these groups. This step is quite time-consuming [18,10].

To overcome the above problems, the following work has been carried out:

(1) The neighbor relationship between instances is determined automatically by using Delaunay triangulation (DT).

(2) The concept of k-order neighbors in DT is used to allow users to adjust neighboring instances for their specific needs in exploring SPCPs.

(3) A modified clique-based PSCP mining algorithm is developed. First, all neighboring instances are enumerated by cliques. Next, these cliques are arranged into a special hash table structure. Then all co-location instances of any candidates can be conveniently queried from the structure. Finally, PSCPs can be filtered efficiently.

1.2 Related Work

PSCP is an important branch of data mining, so there are many algorithms have been proposed. Partial join [19] and joinless [18] search for PSCPs by identifying co-location instances of candidates using separate cliques and star neighborhoods, respectively. However, both the two approaches face challenges in terms of computational complexity. To improve mining performance, co-location pattern instance-tree [11], improved co-location pattern instance-tree [12], CP tree-based [17], clique-based instances driven schema (IDS) [2], and so on were developed to enhance the ability to search for SPCPs. This helps reduce the number of required joins and optimize the search process. Beside The proposed framework [17] utilizes CP-trees to construct trees for each spatial feature and generate co-location candidates from the trees. This allows for more efficient searching of co-location candidates compared to the join-less method.

The above algorithms offer improving efficiency, but they may still face scalability challenges when dealing with large and dense datasets. The computational complexity can increase significantly as the dataset size grows, leading to longer processing times or even infeasibility. Thus, parallel PSCP mining algorithms have been proposed such as MapReduce [15], Hadoop [16], and GPU [1]. It divides the data into smaller blocks and processes them in parallel on computing nodes, thus improving performance and speeding up the mining process.

In the mentioned algorithms, a distance threshold is used as a parameter for the process of searching PSCPs. However, the inconvenience of using this parameter has been discussed above. Some studies have proposed to solve that problem, e.g., k nearest neighbor graph [7] and Delaunay triangulation (DT)-based [13,8]. A newly improved algorithm based on DT, that employs three filters (i.e., feature constraint, global edge constraint, and local constraint) to eliminate redundant edges on the original DT, has also been proposed [9]. Nevertheless, these algorithms suffer from challenges when dealing with large datasets or long patterns, as the number of neighboring instances expands exponentially, demanding significant storage space and posing difficulties in computational efficiency.

2 Background

Feature (F): A feature refers to a characteristic or attribute of an object or entity in a dataset.
$F = \{f_1,\ldots f_n\}$
Instance (S): An instance, also known as an observation or data point, represents a single occurrence or example in a dataset. A set of instances $S = \{S1 \cup S2 \cup \ldots \cup Sn\}$, where Si $(1 \leq i \leq n)$ is a set of instances of feature fi.
Neighbor relationship (R): Neighbor relationship refers to the proximity or spatial closeness between objects or instances in a dataset. It describes the spatial relationship or adjacency between data points based on their locations.
Clique (clq): refers to a set of objects or instances that are mutually connected or related (each pair of instances are neighbors).
Maximal Clique (max-clq): A maximal clique is a clique that cannot be extended further by adding additional objects or instances while preserving the property of being a clique.
Colocation pattern (c): A colocation pattern refers to a spatial pattern where a set of objects or instances frequently co-occur or co-locate together within a specified spatial area.
c ($c \subseteq F$)
Row instance (I): refers to a single record or entry in a dataset.
I ($I \subseteq S$)
Table instance of c (T(c)): A collection of row instances
Participation ratio: It represents the relative level of participation of an instance in a colocation pattern compared to the total number of patterns.

$$PR(c, f_i) = \frac{number\ of\ distinct\ instance\ \text{fi in T(c)}}{Number\ of\ instace\ \text{fi in S}} \qquad (1)$$

Participation index (PI): The participation index is a measure that quantifies the level of participation or involvement of an instance in a colocation pattern.
$PI(c) = min_{f_i \in c}\{PR(c, f_i)\}$
Prevalence threshold (min_prev): The prevalence threshold is a predetermined value or threshold that is used to determine the minimum level of occurrence or frequency required for a colocation pattern.
Prevalent colocation: A prevalent colocation refers to a colocation pattern that occurs frequently or exceeds a predefined prevalence threshold in a dataset (if $PI(c) \geq min\_prev$).
For example, in Figure.1 show table instance of the graph from Figure 5. In Figure 5, a spatial dataset is shown consisting of four features: A, B, C, and D, along with 12 instances. The neighbors in Figure 5 are connected by edges, indicating their proximity relationship. Examples of cliques include some cliques like {A.2, B.3, C.2}, {A.3, C.1}, and {A.2, C.2}. Among them, {A.2, B.3, C.2} represents a maximal clique of the colocation pattern {A, B, C}. Figure 1 displays the instance table of colocation patterns. The colocation pattern {A, D} includes row-instances such as {A.3, D.1}, {A.1, D.2}, and {A.1, D.3}.

By calculating the PR({A, D}, A) = = $\frac{number\ of\ instance\ of\ feature\ A\ in\ T(\{A,D\})}{Number\ of\ instace\ of\ feature\ A\ \text{in}\ S}$=$\frac{|A1,A3|}{|A1,A2,A3|}$= $\frac{2}{3}$ and PR({A, D}, D) = $\frac{number\ of\ instance\ of\ feature\ D\ in\ T(\{A,D\})}{Number\ of\ instace\ of\ feature\ D\ \text{in}\ S}$=$\frac{|D1,D2,D3|}{|D1,D2,D3|}$=1, we obtain the PI({A, D}) = $\frac{2}{3}$ (the minimum of the two PR values of features A and D in the colocation {A, D}).
Assuming a prevalent threshold of 0.6, {A, D} qualifies as a prevalent colocation pattern because PI({A, D}) > 0.6.

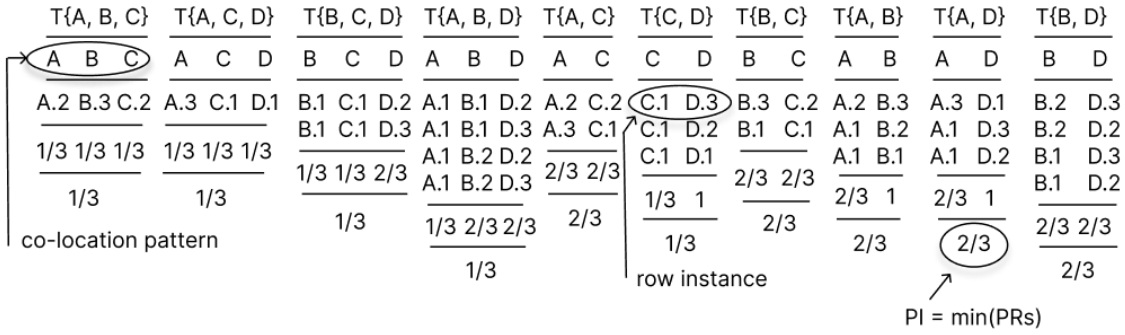


Figure 1: Table instance

3 Methodology

We propose a combined algorithm called Delaunay Triangulation K-order Clique (DTKC), which integrates the DT-based approach (presented in section 3.1), K-order neighbors (presented in section 3.2), and Clique approach (presented in section 3.3). After that, we generate candidate (3.4) and use prevalent co-location filtering (3.5).

3.1 The DT-based approach

Below are the figures illustrating the results of the steps in the DT-based approach. Figure 2 presents the input as spatial instances. Figure 3 create triangle structures by Delaunay triangulation method. Figures 4, 5, and 6 depict the construction of the neighborhoods of instances for mining colocation patterns using the constrained Delaunay triangulation.

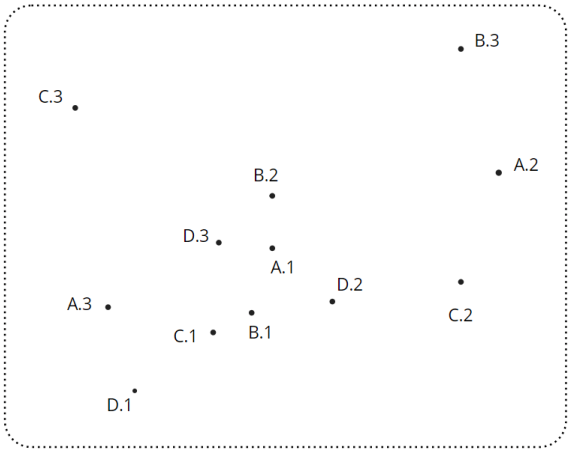


Figure 2: Spatial instance

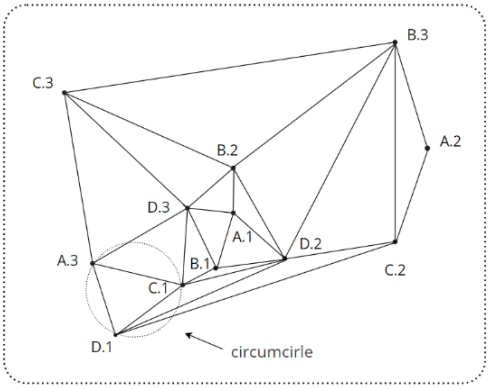


Figure 3: Original Delaunay triangulation

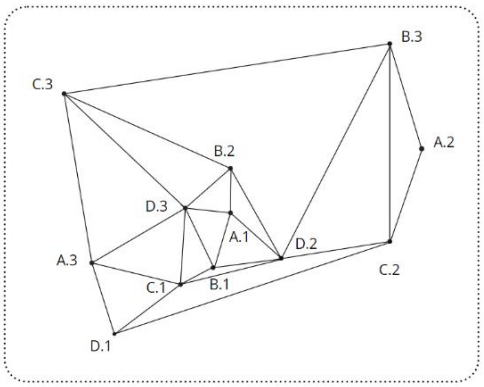


Figure 4: DT after constraint 1

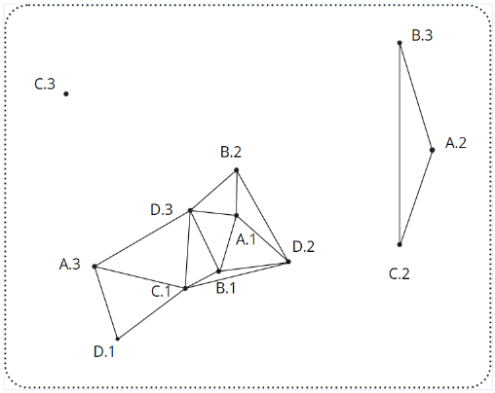


Figure 5: DT after constraint 2

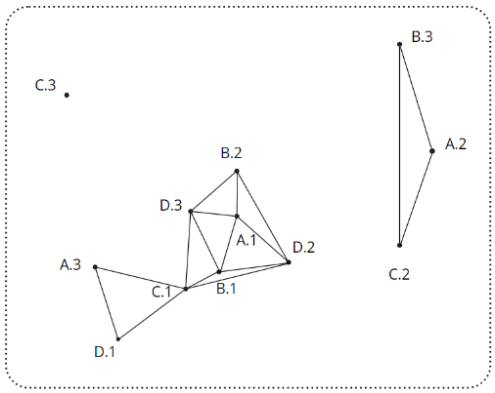


Figure 6: DT after constraint 3

3.1.1 DT-based neighbors

Definition 1: Delaunay Triangle of S (DT(S)) [9]: Let S be a collection of points or instances. The Delaunay Triangulation (DT) of S is a geometric structure that consists of a set of non-overlapping triangles, forming a connected graph. The DT is constructed in such a way that no point in S lies within the circumcircle of any triangle in the triangulation. This results in a unique and explicit representation of the spatial data's topology. The Delaunay Triangulation provides a comprehensive understanding of the relationships between the points, revealing the underlying connectivity and proximity within the dataset.

In this context, the term "circumcircle" can be understood as follows:
Consider a triangle $\Delta = \{v_i\}$ $(1 \leq i \leq 3)$, where each v_i represents a vertex. The circumcircle of Δ refers to the one and only circle that intersects all three vertices of the triangle and doesn't contain any other point inside. This circle is specifically constructed to pass through the three vertices of the triangle.

For example, Figure 3 show the result of spatial data in Figure 2: all triangles constructed
Definition 2: Neighborhood [9]: Neighborhood is classified based on the minimum number of edges required to form a path connecting a point under consideration with other points in DT(S).
For example, consider the following:
Let S be a set of spatial instances. DT(S) refers to the Delaunay Triangulation of S. When considering two points f_{a_b} and f_{x_y} in DT(S), f_{a_b} is considered a first-order neighbor of f_{x_y} if and only if there exists a direct edge connecting both f_{a_b} and f_{x_y}. Additionally, f_{x_y} is considered a k-order neighbor of f_{a_b} if the shortest path based on the edges formed by DT(S) from f_{a_b} to f_{x_y} includes k edges.

As depicted in Figure 3, the neighborhood relationship among the instances in Figure 2 are naturally and clearly represented by a cohesive and interconnected graph. Nevertheless, when applying the Delaunay Triangulation (DT) method to mine spatial colocation patterns directly, certain issues arise that require resolution: instance with same feature still can be connected like B.2B.3 and D.1D.2 in Figure 3, some edges are too long compared to the whole locality B.3D.3 in Figure 3, so the neighbor relationship may not be correct. Therefore, we will use three constraints [9] in DT-based constraints in the Section 3.2.2.

3.1.2 DT-based constraints

The DTC (DT-based colocation pattern mining) algorithm [9] have proposed three constraints:
First, the ***feature constraint*** confirmed that when examining the neighborhoods of instances for a pattern, a condition is that two neighboring instances must belong to distinct feature types.
Second, the ***global edge constraint*** insisted that there are some edges in the global triangulation whose lengths are excessively long. Since the original Delaunay Triangulation (DT) does not take these conditions into account, it cannot be directly utilized for mining colocation patterns:

Definition 3: Redundant Feature Edge: A redundant feature edge is an edge that links two vertices having identical feature types within the Delaunay Triangulation (DT). All redundant feature edges are eliminated from the DT to ensure the validity of the resulting structure.

For example, in Figure 4, we have removed all redundant feature edge B.2B.3 and D.1D.2 in Figure 3.

Definition 4: Degree [9]: The Degree of a vertex f_{i_j} in the DT is the number of Delaunay edges connected to f_{i_j},, denoted as deg(f_{i_j}).
For example, in Figure 3, the degree of B.2 is 5 because there are 5 edges connect to B.2: B.2C.3, B.2B.3, B.2D.3, B.2A.1, B.2D.2.

Definition 5: Local mean ($\mu_{local(f_{i_j})}$): refers to the average length of all the edges that are directly connected to f_{i_j} in the Delaunay Triangulation (DT). It represents the average distance between f_{i_j} and its neighboring instances in the graph, calculated:

$$\mu_{local(f_{i_j})} = \frac{1}{\deg(f_{i_j})} \sum_{k=1}^{\deg(f_{i_j})} |d_t| \quad (2)$$

where $|d_{(t)}|$ is the Euclidean length of the Delaunay edge $d_{(t)}$

Definition 6: Local Standard Deviation ($\sigma_{local(f_{i_j})}$): refers to the measure of variability or dispersion in the lengths of all the edges that are directly connected to the instance f_{i_j} in the Delaunay Triangulation (DT), calculated:

$$\sigma_{local(f_{i_j})} = \sqrt{\frac{1}{\deg(f_{i_j})-1} \sum_{t=1}^{\deg(f_{i_j})} (|d_t| - \mu_{local})^2} \quad (3)$$

Definition 7: Global mean ($\mu_{global(DT)}$) : It represents the overall average distance between connected vertices in the triangulation. By calculating the sum of all edge lengths and dividing it by the total number of edges, we can obtain the global mean value:

$$\mu_{global(DT)} = \frac{1}{N(d_{(t)})} \sum_{t=1}^{N(d_{(t)})} | d_t | \quad (4)$$

Where $N(d_{(t)})$ is the numbers of all edges in the DT

Definition 8: Global Standard Deviation ($\sigma_{global(DT)}$): represents the variability or spread of edge lengths across all edges in the Delaunay Triangulation (DT). It measures how much the lengths of the edges deviate from the average length, providing insights into the overall distribution of edge lengths, calculated:

. $$\mu_{global(DT)} = \frac{1}{N(d_{(t)})} \sum_{t=1}^{N(d_{(t)})} | d_t | \quad (5)$$

Definition 9: Global positive edge: an edge is classified as a global positive edge if its length does not exceed the global distance constraint associated with the corresponding instance f_{i_j}. In other words, if the length of an edge connecting to $f_{i_j}$is less than or equal to the specified global distance constraint, it is considered a global positive edge, global distance constraint f_{i_j} ($const_{global(f_{i_j})}$) can be calculated:

$$const_{global(f_{i_j})} = \mu_{global(DT)} + \sigma_{global(DT)} \frac{\mu_{global(DT)}}{\mu_{local(f_{i_j})}} \quad (6)$$

The global distance constraint of a vertex is a measure that considers statistical information about the average length and standard deviation of all edges in the Delaunay Triangulation (DT), as well as the local mean of the vertex itself. This constraint provides statistical insights into the spatial positioning of the vertex within the dataset and its neighboring instances.

Each instance in the dataset has its own defined proximity region. If the length of an edge directly connected to an instance f_{i_j} is greater than the global distance constraint of f_{i_j}, that edge is considered invalid and is subsequently removed from the DT. This ensures that only edges within the defined proximity range are retained, preserving the validity and accuracy of the DT.

For example, in Figure 5, we have removed all redundant edges on a global level: A.3C.3, C.2D.2, C.3D.3, B.2C.3, B.3C.3, C.2D.1, B.3D.2.

Third, after satisfying the second constraint, we obtain distinct clusters and the ***local edge constraint*** will remove some edges that are excessively long in each subgraph:

Definition 10: Mean Local Standard Deviation ($\mu_{\sigma_{local}}$): The local standard deviation value of a vertex represents the variability or dispersion of edges incident to that vertex. By averaging the local standard deviation values of all vertices, we can obtain a measure of the overall variability within the graph G (G is subgraph):

$$\mu_{\sigma_{local}} = \frac{\Sigma\sigma_{local(f_{i_j})}}{N_{vertex(G)}} \text{ for } f_{i_j} \in G \quad (7)$$

Where $N_{vertex(G)}$ is the number of vertices of G and f_{i_j} is a vertex of G

Definition 11: Local positive edge: refers to an edge that is connected to an instance in f_{i_j} a graph. It is considered a local positive edge if its length does not exceed the local distance constraint of f_{i_j}. The local distance constraint of f_{i_j} represents a limit or threshold on the maximum length of edges that are considered valid or acceptable for f_{i_j}.

$$const_{local(f_{i_j})} = \mu^2_{local(f_{i_j})} + \beta\mu_{\sigma_{local(f_{i_j})}} \quad (8)$$

Where $\mu^2_{local(f_{i_j})}$ represents the average length of edges connecting the first-order and second-order neighbors of f_{i_j}. This metric captures the typical distance between f_{i_j} and its nearby points, providing a local understanding of the spatial relationships.

The parameter β controls the sensitivity of the local positive edges. A local positive edge refers to an edge whose length does not exceed the local distance constraint of f_{i_j}. When β is set to a smaller value, the local distance constraint becomes more strictly, resulting in a reduced number of retained local positive edges in the DT. In this study, a default value of 1 is used for β [20].

And in the Figure 6, the local edge constraint helps us redundant edge on a local level: A.3D.3

Considering the instances within the second-order neighbors of f_{i_j} can be likened to observing a microscope's field of view. By examining subgraphs with varying densities, finer details can be taken into account. The $const_{local(f_{i_j})}$ is designed to identify long edges within each density subgraph specifically. From a statistical perspective, these edges, formed by the points within the second-order neighbors of f_{i_j} in a subgraph, can be viewed as a small sample.

3.1.3 DT-based construct algorithm

Algorithm 1:

Input: S: a set of spatial instances.
Output: DT_{local}: a set of all edges after implementing 3 filtering strategies

Steps
1) DT= **Delaunay_triangulates(S)**
2) $DT_{feature}$ = **filter_feature(DT)**
3) DT_{global} = **filter_global_edge(DTfeature)**
4) DT_{local} = **filter_local_edge(DTglobal)**

Algorithm 1 describes the implementation of the DT-based approach. With a set of spatial instances as input as shown in Figure 1. Firstly, the algorithm will create triangular structures using the Delaunay Triangulation (DT) method like Figure 2 (step 1). Next, the algorithm continues to use three filters: filter the edges connecting instances with the same feature like Figure 3 (step 2), filter global positive edges like Figure 4 (step 3), filter local positive edges like Figure 5 (step 4). Finally, the algorithm will return a set of all edges representing the neighbor structure of the spatial instances.

3.2 K-order neighbors approach

Definition 12: ***Big neighborhood*** [2]: Big neighborhood of instance s is the instances s' that satisfy both the relationship R(s', s) with s and belong to sets with higher indices of feature than the set containing instance s.

According to *Definition 2* about Neighborhood in DT-based, K-order neighbors algorithm will obtain all big neighborhoods from the first – order neighbors to the k-order neighbors of the instance that you are considering.

Algorithm 2: K-order neighbors algorithm

Input:

```
S: a set of spatial instances
DE: a set of all Delaunay edges after DT-based construct algorithm
K: order neighbors that user specifies
```

Output:

```
BNs(p): All big neighborhood of each instance in S
```

Steps:

```
1) For instance p in S:
2)  k =0
3)  count=0
4)  QV.add(p)
5)  DU.add(p)
6)  SetPointK.add(p)
7)  K_order.add(SetPointK)
8)  While(NotEmpty(QV) and k<K)
9)   Count++
10)  Curr_instance=QV.Out
11)  For Instance first_order in First_order(Curr_instance):
12)   Curr_Edge= Edge(Curr_instance, first_order)
13)   If DE.contains(Curr_Edge) && SE.not_contains(Curr_Edge)
14)    SE.add(Curr_Edge)
15)    If !DU.contains(first_order) && first_order € BNs(Curr_instance)
16)     DU.add(first_order)
17)     QV.add(first_order)
18)     SetPointK.add(first_order)
19)    End if
20)   End if
21)  End for
22) End while
23) DU.remove(p)
24)BNs(p).addAll(DU)
25)End for
```

The Algorithm 2 describes the way we find K-order neighbors. Steps 11 to 21 involve searching for the first-order neighbors of each point in the queue QV (which includes the instance itself and its neighbors for retrieval). By iterating the process of finding the first-order neighbors of instances in QV (from step 8 to step 22), we generate the first-order neighbors, second-order neighbors, and so on until we create the K-order neighbors. However, if the instance itself does not have K-order neighbors, the search process stops when the queue QV is empty, ensuring that we find all possible neighbors for each instance (as stated in the condition in step 8). And in the step 23 we remove the instance itself to get all neighbors from first-order neighbors to k-order neighbors. We add all instances to a list BNs of p (step 24) and then consider other instances in S.

For example, the result of Algorithm 2 (All Big neighborhoods of each instance in spatial data S) for Figure 6 is 3 Figure as follow with different K:

Table 1: Big neighbourhood lists with K=1

Instance	BNs	Instance	BNs	Instance	BNs	Instance	BNs
A.1	B.1, B.2, D.2, D.3	B.1	C.1, D.2, D.3	C.1	D.1, D.2, D.3	D.1	-
A.2	B.3, C.2	B.2	D.2, D.3	C.2	-	D.2	-
A.3	C.1, D.1	B.3	C.2	C.3	-	D.3	-

Table 2: Big neighbourhood lists with K=2

Instance	BNs	Instance	BNs	Instance	BNs	Instance	BNs
A.1	B.1, B.2, C.1, D.2, D.3	B.1	C.1, D.1, D.2, D.3	C.1	D.1, D.2, D.3	D.1	-
A.2	B.3, C.2	B.2	C.1, D.2, D.3	C.2	-	D.2	-
A.3	B.1, B.2, C.1, D.2, D.3	B.3	C.2	C.3	-	D.3	-

Table 3: Big neighbourhood lists with K=3

Instance	BNs	Instance	BNs	Instance	BNs	Instance	BNs
A.1	B.1, B.2, C.1, D.2, D.3	B.1	C.1, D.1, D.2, D.3	C.1	D.1, D.2, D.3	D.1	-
A.2	B.3, C.2	B.2	C.1, D.1, D.2, D.3	C.2	-	D.2	-
A.3	C.1, D.1	B.3	C.2	C.3	-	D.3	-

3.3 Clique approach: Depth-First Clique Instance drive schema (DFCIS)

Considering a spatial dataset's neighborhood list, we construct a tree structure for identifying cliques. Then, we will introduce a new efficient schema based on Instance driven schema (IDS) [2] to find clique.

Definition 13: ***H_s-Clique*** [2]: A clique cl is specifically termed an H_s-Clique if its head corresponds to the instance s. All clique starting with instance s is referred to as an H_s-Cliques.

Definition 14: An Instances-Driven-Schema-Based clique tree [2] (referred to as I-tree) is a hierarchical structure defined as follows:

1) It comprises a single root node labeled as "root".
2) Each node, except for the root, contains two properties: instance-name and node-link. The instance-name stores the identifier of the represented instance, while the node-link points to the next sibling node that represents a larger instance. If there is no larger instance, the node-link is set to null.
3) The nodes that have the root as their parent are referred to as head-nodes. A head-node, representing an instance "s", is symbolized as hn_s. Any descendant node of a head-node that represents an instance "s" is denoted as n_s.
4) An I-clique encompasses a collection of instances that is denoted by a leaf node and all its ancestor nodes. Within an I-clique, every pair of instances satisfies the condition of being neighbors.

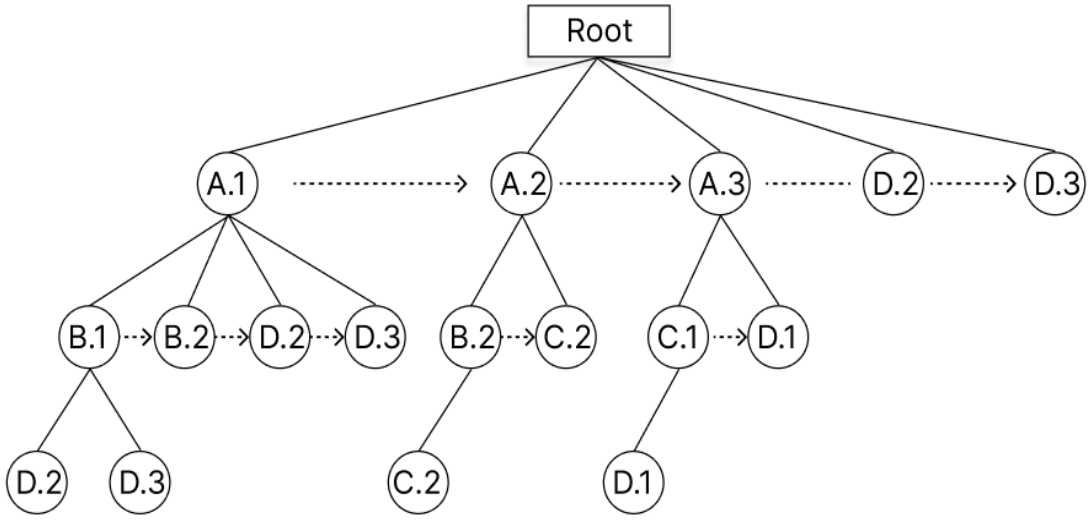


Figure 7: An example of I-tree

To establish an I-tree, several lemmas and definitions are provided as follows.

Lemma 1 [2]: In the case of an instance s, all H_s-Cliques can be generated from BNs(s).

Proof: Since BNs(s) encompasses instances that satisfy the conditions for Hs-Cliques: satisfies the relationship R(s', s) and contains instances with higher-level features.

Lemma 2 [2]: Given an n-size candidate clique clq = $\{s_1, s_2, \ldots, s_n\}$, for every $s_i \in clq$ $(1 \leq i < n)$ and $\forall s_j (j > i)$, if $R(s_i, s_j)$ = true, then clq is a clique.

Proof: For every $s_i \in clq$ $(1 \leq i < n)$ always exist at least one instance s_j have higher-level feature and satisfies the relationship $R(s_i, s_j)$.

For example, in Figure 7, with a candidate clique: A.1B.1D.2, A.1 have relationship with B.1 and D.2, B.1 have relationship with A.1 and D.2, D.2 have relationship with B.1 and A.1. So the candidate clique A.1B.1D.2 is a clique.

Definition 15: ***Right Sibling Instances*** [2]: Regarding a node n_s within an I-tree, its set of right sibling instances comprises instances represented by its right siblings possessing distinct features, denoted as $RS(n_s)$.

Lemma 3 [2]: When considering a head-node hn_s within an I-tree, the children of hn_s correspond to $BNs(hn_s)$. In the case of a non-root node n_s, the children of n_s are obtained by intersecting $BNs(n_s)$ with $RS(n_s)$.

Proof: The children of n_s need to satisfy both the relationship R with n_s ($BNs(n_s)$) and parent of n_s (it is BNs(parent of n_s) that doesn't include n_s, so it is $RS(n_s)$). So the children of n_s are obtained by intersecting $BNs(n_s)$ with $RS(n_s)$.

For example, in the example for *Lemma 2*, A.1B.1D.2 is a clique, D.2 is the child of B.1 because it is the intersectant instance of BNs(B.1) with RS(B.1).

Definition 16: ***Clear Node***: A node is called a Clear Node if all its children are determined to be big neighborhood of that node - the set of k-neighboring instances of ns.

Lemma 4: For a head node hn_s, for one of the other non-root nodes n_s, if $Childrens(n_s) = BNs(n_s)$ then mark n_s as Clear Node. For a Head-node hn_s, check if hn_s is Clear Node or not, if hn_s is Clear Node then we skip finding cliques of this node.

Proof: Since Childrens(ns) = BNs(ns), when n_s becomes a new head node, we have already traversed these cliques within n_s because it does not miss any of the instances in BNs(ns). In other words, there is already an existing clique that contains any clique created when ns is not a new head node, as it has a length longer than at least one instance (the parent of n_s in the old clique).

Definition 17: ***Can Clique***: represents the status or condition that determines whether a clique can be generated or not during the execution of the algorithm.

Definition 18: ***Old Instances list***: is the list contains all old instance that have been traversed through.

Lemma 5: For a Head-node hn_s, with each non-root nodes ns not in Old Instances list or the rank of that node is not greater than K (order neighbors that user specifies), mark Can Clique = true and push n_s in oi. At leaf node check if Can Clique = true then generate clique and mark Can Clique = false, else if Can Clique = false then don't generate this clique from this leaf node.

Proof: The process marks Can Clique as True when encountering a new node (Old Instances list doesn't contain it) that have the rank more than K (the node with rank K will link to new clique avoid missing important cliques). We check Can Clique at the leaf node to ensure that the final clique has been created. If Can Clique is True, it indicates that there is a new node within that clique, allowing us to generate that clique. Conversely, if no new node is encountered, it implies that we have already generated the clique containing this particular clique.

Lemma 6: For a Head-node hn_s when we meet a new node while traversing, The Old Instance list will remove all instance which is the children of that node to make sure collect all cliques correctly.

Proof: When we meet a new node, the node can have more than one child. According to *Lemma 5*, at the leaf node of first clique of first children, we will mark Can Clique = false, and when we traverse the second, third clique, … we will miss the clique if we don't meet a new node, but the clique can get because we have already met the new node in that clique (their parent node).

Algorithm 3: Depth-First Clique Instance Schema(DFCIS)

Input:
nbs: neighborhood list(including BNs of each instance)
S: set of instances
Output:
Clqs: list of I-cliques

Steps:

```
1) iTree = Initialize_Itree(); //To create a root node on the tree
2) For Each instance p in S Do // To get Hs-Cliques for each p
3)  stack = Initialize_stack(); //depth-first manner
4)  headNode = iTree.AddHeadNode(p); //Get Hs-Cliques
5)  stack.push(headNode); //Add headNode to the top of stack
6)  OldInstance = Initialize_Set();
7)  While NotEmpty(stack) Do
8)   CurNode = stack.pop(); //Get a node from top of stack
9)   If OldInstance.NotContain(CurNode) or RankOf(currNode) <=K Then
10)   CanClique = True //Mark can generate new clique
11)   OldInstance.Add(CurNode); //Add CurNode to OldInstance
12)  End If
13)  ChildNode = GetChildren(CurNode); // Lemma 3, Lemma 4 and Lemma 6
14)  If IsEmpty(ChildNode) Then
15)   If CanClique is True Then
16)    CanClique = False; //Mark can't generate new clique
17)    Clqs.Add(GetClique(CurNode)); //Add clique to the result set
18)   End If
19)   iTree.RemoveAncestors(CurNode);
20)  Else
21)   iTree.AddNode(CurNode, ChildNode); //Add ChildNode to CurNode
22)   stack.push(ChildNode.Reverse())
23)  End If
24) End While
25)End For
```

The algorithm 3 follows a depth-first traversal approach to generate Hs-Cliques for each instance in S (step 2 – 25). It maintains a stack for node traversal and keeps track of visited instances (step 8 and step 22). The algorithm traverses and utilizes *Lemma 5* to ensure the collection of all cliques when encountering a new node in the clique branch and avoids collecting sub-cliques within the already collected cliques corresponding to each head node hn_s (step 9 -18). The algorithm get the children node according to *Lemma 3*, *Lemma 4* and *Lemma 6* (we will present in Algorithm 4) (step13). After generating an I-clique, a pruning operation is performed to optimize memory usage. When a node ns is identified as a leaf node, it can be safely pruned. Moreover, if the parent node of ns becomes a leaf node after ns is pruned, the parent node can also be safely pruned. This pruning process continues recursively until one of ns's ancestor nodes retains one or more children after one of its children has been pruned (step 19). The algorithm continues until all instances have been processed.

Algorithm 4: Get Children follow a depth-first traversal

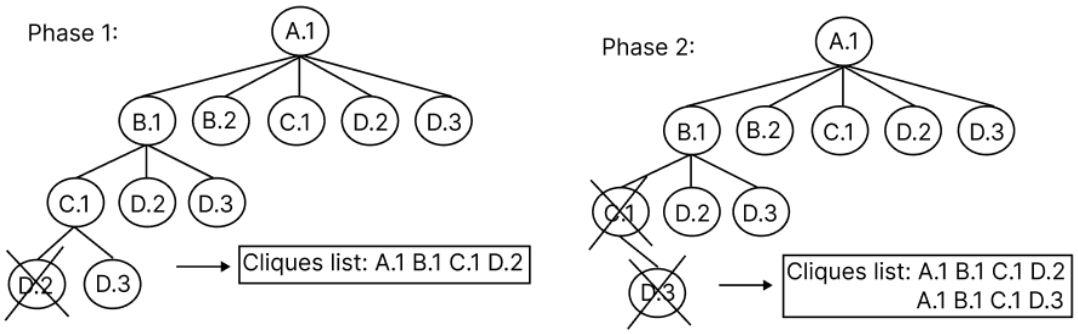
Input: currNode - the current node
Output: childrenNodes - a list of child nodes

Steps:

```
1) If the currNode.parent.IsRoot:
2)  If the currNode.IsClearNode:
3)   return the childrenNodes list.
4)  End if
5)  For each point p in the currNode.BNs:
6)   p.parent=currNode
7)   childrenNodes.add(p)
8)  End for
9)   For each node in childrenNodes do:
10)   node.R_sibling=getR_Sibling(node)
11)  End for
12)Else:
13) For each point p in currNode.BNs:
14)  If CheckSiblingContain(currNode.R_sibling, p):
15)   p.parent=currNode
16)   childrenNodes.add(p)
17)   If(CanClique)
18)    OldInstance.remove(p)
19)   End if
20)  End if
21) End for
22) For each node in childrenNodes do:
23)  node.R_sibling=getR_Sibling(node)
24) End for
25) If childrenNodes.size()=currNode.BNs.size()
26)  currNode.IsClearNode=true
27) End if
28)End if
29)Return childrenNodes
```

The algorithm 4 first checks if the current node is a head node or not (step 1). If it is a head node, first it checks whether head node is clear Node (step 2) according to *Lemma 4* (*Lemma 4* is used to determine whether each head node ns is a clear node or not. This helps us avoid traversing clear nodes (when it is head node) and reduces computational time), if true return the children Nodes list (step 3), if it is not Clear Node then it processes each point in its big neighborhood (step 5), sets the head node is the parent of each point (step 6), and adds them to the children Nodes list (step 7). Then, it sets the Right sibling instances for each node in the list (step 9-11). Else If the current node is not a head node (step 12), it processes each point p in its big neighborhood (step 13), check if p belongs to the set of right sibling instances of the current node or not (step 14). If True, then sets current node is the parent for each eligible point (step 15), and adds them to the children Nodes list (step 16). Then, if Can Clique is true (meet a new node while traversing), remove all children of current node in Old Instance according to *Lemma 6*(step 17-19). Again, it sets the Right Sibling Instances for each node in the list (step 22-24). Finally, if the size of children Nodes list is equal to the size of the big neighborhood of the current node, it sets current Node is the Clear Node (step 25 -27). The algorithm returns the children Nodes list (step 29).

For example, we will describe how the Algorithm 3 execute in data at Table 2 follow all Figure 8 A, B, C, D, E. At step 1 (we assign the step of collecting the first clique as Step 1), we get all children of each node in the stack and when meet the leaf node D.2, we collect the first clique A.1B.1C.1D.2 and each instance in that clique will be added to Old Instances List then remove the leaf node D.2. At step 2, we collect the clique A.1B.1C.1D.3 because D.3 is the new node and it will be added to Old Instances List then remove the leaf node D.3 and C.1 because C.1 have no children else. At step 3, we remove the leaf node D.2 and it is in Old Instances List so we don't add clique A.1B.1D.2 to Cliques list (Can Clique is still false). Then step 4 we remove D.3 and B.1 and don't add the clique A.1B.1D.3 (Can Clique is still false). Finally, after traversing all node in I-tree with head node A.1, we collect Cliques list as Figure 8E, the reason why A.1C.1D.2, A.1C.1D.3, A.1D.2, A.1D.3 still in the Cliques list because C.1, D.2, D.2 have the rank 2 (not greater than k) so it is also added to Cliques list.



A B

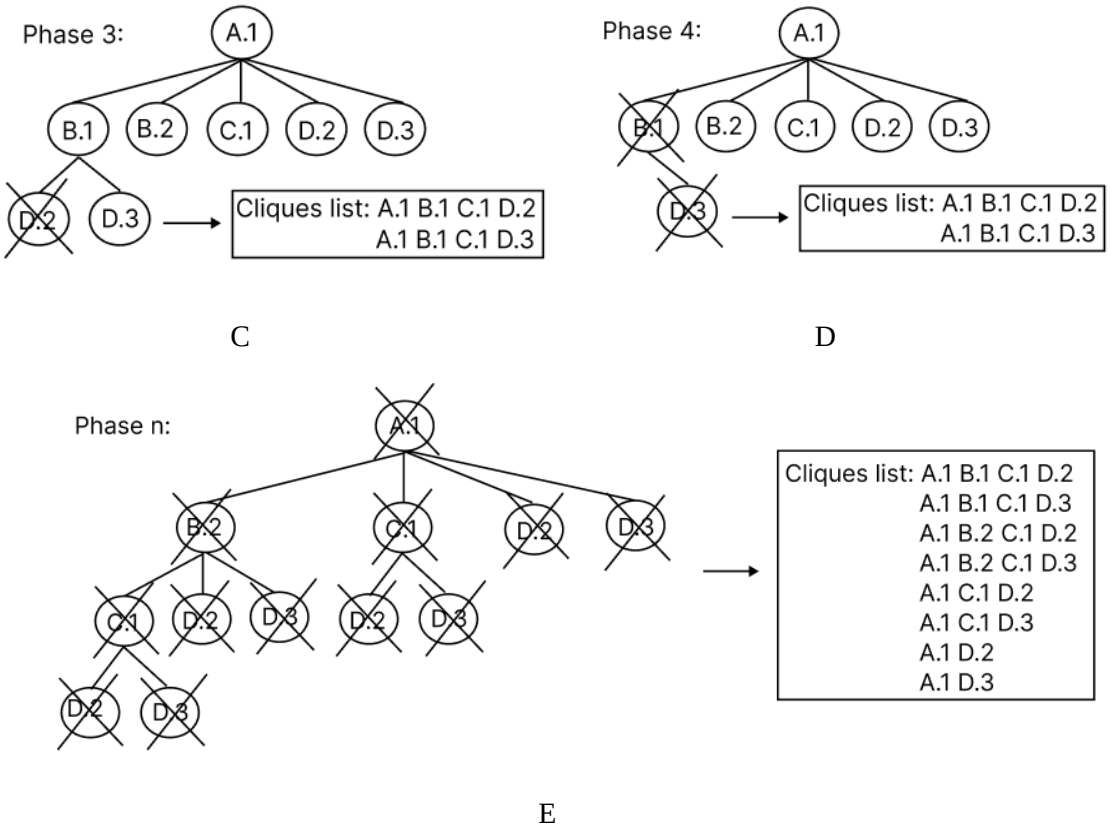


Figure 8: The DFCIS approach for head node A.1. A, the first step when collect the first clique. B, The second step. C, The third step. D, The fourth step. E, the n-th step when finish collect cliques from head node A.1.

3.4 Candidate generation

Definition 17: A ***Compressed clique hash*** (C-hash) [2]: is a data structure consisting of key-value pairs that efficiently stores and organizes information by grouping features and associating them with their corresponding instances, where:

(1) The key represents a set of features, denoted as Fc.

(2) The value is a collection of hash structures, each containing a key-value pair. Here, the key represents a specific feature f that belongs to Fc, and the value' represents a set of instances associated with feature f. The union of all key' values forms the complete key.

Figure 9 illustrates an example of a C-hash according to *Definition 17*. For a clique clq = {A.3, B.1, C.1, D.1}, the key set is {A, B, C, D}, and the values associated with this key set are [<A, {A.1}>, <B, {B.1}>, <C, {C.1}>, <D, {D.1}>]. Other cliques clqs are {A.3, B.1, C.1, D.2}, {A.3, B.1, C.1, D.3}, {A.1, B.1, C.1, D.2}, {A.1, B.1, C.1, D.3}, {A.1, B.1, C.1, D.2}, {A.1, B.2, C.1, D.2}, {A.1, B.2, C.1, D.3} shares the same key set {A, B, C, D}. Consequently, the value of the key set {A, B, C, D} is updated to [<A, {A.1, A.3}>, <B, {B.1, B.2}>, <C, {C.1}>, <D, {D.1, D.2, D.3}>]. The candidates can be derived from the keys of the C-hash.

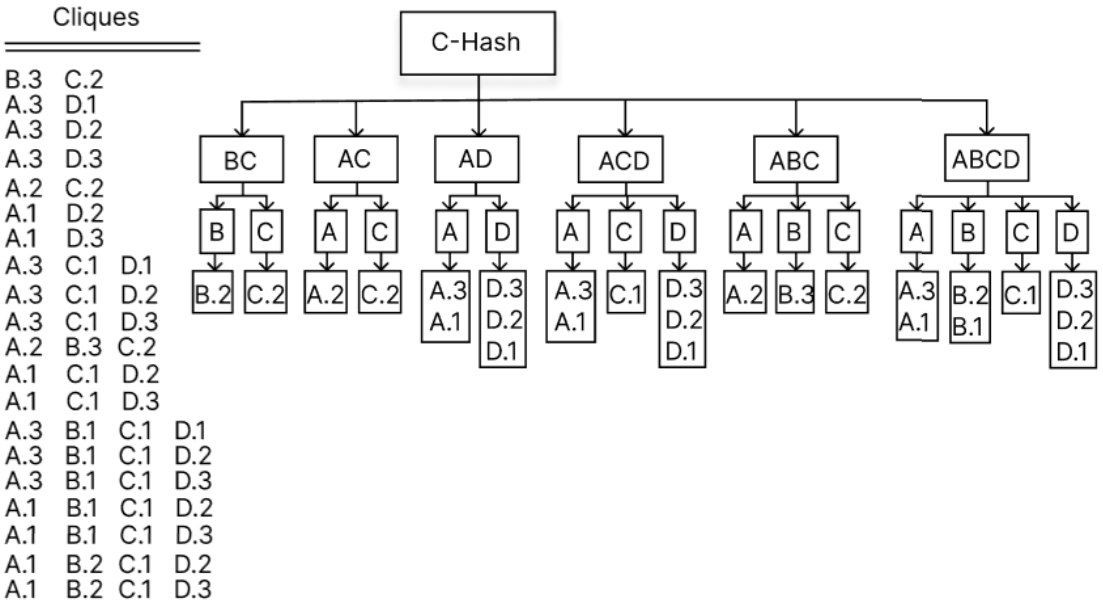


Figure 9: C-hash example

Algorithm 5: Candidate generation [2]

Input:
clqs: Cliques generated by DFCIS
Output:
C-Hash: c-hash structure

Steps:

```
1)For Clique clq in clqs do
2) newKey=GetFeatures(clq) //Get the features of instances in clq
3) If Not chash.ContainsKey(newKey) Then
4)  chash.AddKeyAndInitialize(newKey): //Do initialize operations
5) End if
6) For Feature f in newKey do
7)  chash[newKey][f].AddInstances(clq);
8) EndFor
9)End For
```

The algorithm 5 create the structure of C-hash. For each clique in the input list of cliques, the algorithm generates a new key called "newkey," which is a set of features extracted from that particular clique (step 2). It only adds the newkey to the C-hash if it does not already exist in the hash structure (steps 3-5). This ensures that the C-hash contains a unique set of features, reducing memory storage and improving data retrieval efficiency. Subsequently, the value is a key-value pair, where the value is the instance itself and the key corresponds to the feature associated with that instance (steps 6-8). The process continues until all cliques have been processed.

3.5 Prevalent co-location filtering approach

Definition 18: ***Direct subset***: Direct subset of a co-location size k is the set of all co-location size k-1 created by removed 1 element from co-location size k.

Definition 19: ***Subsets***: is set of all colocation patterns generated from a candidate colocation have size from 2 to candidate.length – 1.

Lemma 7: If candidate is a prevalent colocation pattern, all subsets also are prevalent colocation pattern, and conversely.

Proof: Since candidate colocations have a length longer than all their subsets, we can apply the PI formula. As a result, the PI of each colocation pattern in the subsets will always be higher or equal to the PI of the candidate colocation.

Algorithm 6: Prevalent co-location filtering [2]

Input:
A C-Hash
min_prev
Output:
cs: list of all prevalent colocation with PIs

Steps:

```
1) for each key in C-Hash:
2)  candidates.add(key)
3)  C-hash_key.add(key)
4) End for
5) candidate.sort()
6) while candidates is not empty:
7)  currCandidate = candidates.First
8)  pi = CalculatePI(currCandidate, C-Hash, C-hash_key)
9)  if pi >= min_prev:
10)  candidates.remove(currCandidate)
11)  cs.put(currCandidate, pi)
12)  subsets = GetAllSubsets(currCandidate)
13)  PIs = CalculatePIs(subsets)
14)  cs.putAll(subsets,PIs)
15)  candidates.remove(subset)
16) Else:
17)  candidates.remove(currCandidate)
18)  GetDirectSub(currCandidate, cs, candidates)
19)  candidates.Sort
20) End if
21)End while
22)Return cs
```

The algorithm 6 iterates over the keys in C Hash and adds them to the candidates list and C-hash_key list (step1-4). It then sorts the candidates list in descending order (step 5). The algorithm enters a loop where it retrieves the first candidate from the list and calculates its PI value (step 7-8). Based on *Lemma 7*, if the PI value is greater than or equal to min_prev (step 9), it means the candidate is a prevalent colocation pattern, it adds the candidate and its PI to the cs map then remove it from candidates list and find the subset then calculates the PI values for all its subsets (step 10-13). Then the subsets and their PI values are added to the cs map (step 14) and remove subset from candidates list (step 15). If the PI value is below min_prev (step 16), the algorithm removes the candidate from the candidates list and based on *Lemma 7*, algorithm adds the direct subset of the candidate to the candidates list (for checking prevalent colocation pattern) and sorts the candidates list in descending order again (step 17-19). The loop continues until the candidates list is empty (step 6-21), and finally, the cs map is returned (step 21). The way we calculate the PI will be present in algorithm 7.

Algorithm 7: Calculate PI

Input:
currCandidate: the current candidate
CHash
C-hash_key: a list of keys from CHash
Output:
PI: Participation index of current candidate

Steps:
```
1) minPRs = 1
2) Inst =newMap()
3) For i = 0 to currCandidate.length:
4)  F = currCandidate[i]
5)  Inst.putkey(F).newSet()
6) End for
7) supersets = GetSuperSets(CHash, currCandidate, Chash_key)
8) For each CDs in supersets:
9)  CD = CHash.getkey(CDs)
10) For i = 0 to currCandidate.length:
11)  F = currCandidate.get(i)
12)  Inst.getkey(F).addvalue(CD.getkey(F))
13) End for
14)End for
15)For i = 0 to currCandidate.length:
16) F = currCandidate[i]
17) count_f = size of Inst[F]
18) prs = count_f / Features[F]
19) If minPRs >= prs:
20)  minPRs = prs
21) End if
22)End for
23)Return PI=minPRs
```

The algorithm "Calculate PI" calculates the Participation Index (PI) for a given current candidate. The algorithm iterates over the features in the current candidate, retrieves the instances corresponding to each feature from the supersets, calculates the participation ratio for each feature, and updates the minimum PI value accordingly. The result is the minimum PI value among all features in the current candidate, representing the overall participation level of instances in those features.

4 Experimental results and analysis

4.1 Experiment Setting

In this section, a set of experiments are conducted to examine the performance of the DTkC algorithm. We choose joinless [18], CP-tree-based (named Condense) [17], and Delaunay triangulation-based co-location mining (DTC) [9] to com pare. All programs in our experiments were coded using the Java programming language, available on GitHub1, and were performed on a Laptop with Intel(R) Core (TM) i7-8550u CPU@1.8- 4.0 GHz and 16 GB main memory.

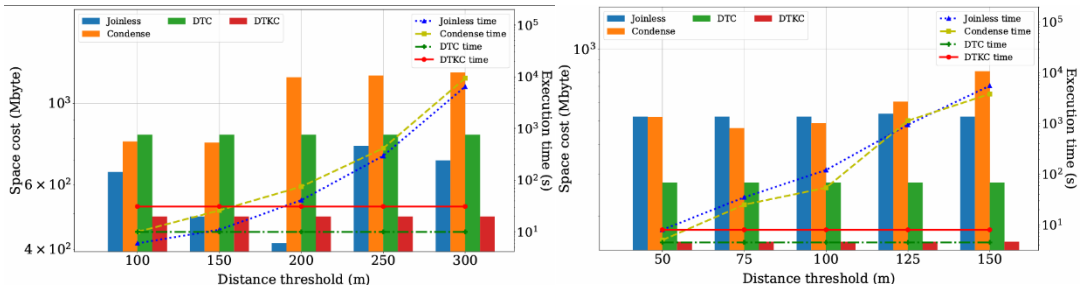
Datasets: Four real datasets that are collected from points of interest in Beijing, China [2], Las Vegas, Toronto, USA2, and United Kingdom (UK)3, were used in our experiments. Moreover, two synthetic datasets, that were produced by a generator [18], were also used in our experiments. Table 4 lists some basic characteristics of the datasets used in this experiment.

Table 4: The datasets used in our experiments

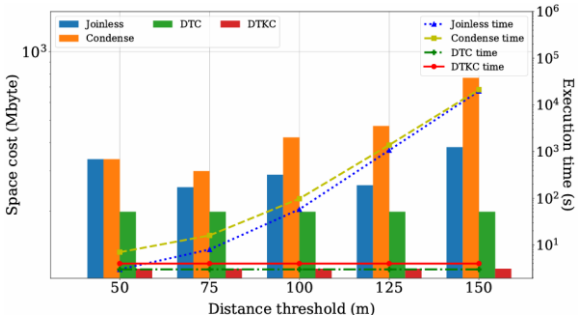
Name	Area	#feature	#instances	Distribution
Beijing	135km x 224km	17	90,257	Centralized + dense
Las Vegas	38km x 63km	19	31,592	Sparse + dense
Toronto	23km x 56km	19	20,309	Sparse + dense
UK	12,84km x 13,867km	26	143,621	Sparse + dense
Synthetic	5000km x 5000km	15	*	Dense

4.2 Compare the mining performance

The first experiment compares the mining performance of the five algorithms including running time and memory consumption based on the variations of two parameters: the minimum distance threshold (only for Joinless and Condense) and the minimum prevalence threshold.



(a) Beijing (b) Las Vegas



(c) Toronto

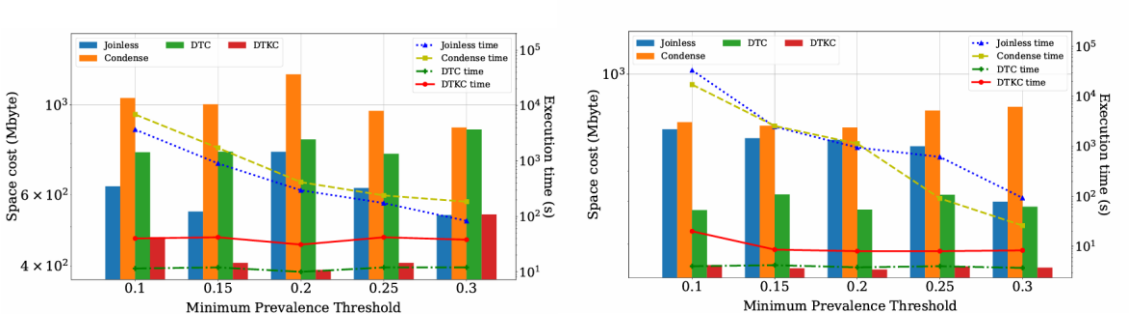
Figure 10: The performance of compared algorithms on different distance thresholds (min_{prev} = 0.2 for all)

On different distance thresholds: Figure 10 shows the result on both running time and memory consumption of the compared algorithms when modifying the minimum distance at values of 100m, 150m, 200m, 250m, and 300m, and in our algorithm, DTkC sets k=2, i.e., 2-order neighbors. We can observe that DTC and DTkC do not change their running time because they are not dependent on the minimum distance threshold parameter. Since the DTC algorithm only simply uses DT to obtain the neighbor relationship between instances, then it uses a merging strategy, that is, merging from triangles to quadrilaterals, merging from quadrilaterals to pentagons, etc., to find row instances of high size co-location patterns. However, the strategy generates only few high size patterns, so its execution time will be less than our algorithm.

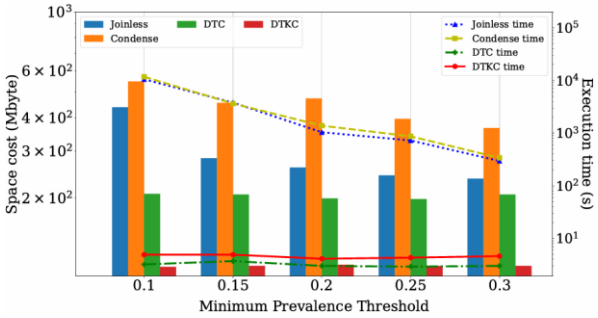
For the joinless and condensed algorithms, their running time will increase multiplicatively as distance thresholds increase. At lower values of distance thresh olds, the running time of all algorithms is approximately equivalent. However, when increasing this parameter by a small amount, it leads to a significant increase in running time for the joinless and condensed algorithms, especially for dense data types like the ones we are using.

For Joinless and Condensed, when the minimum distance threshold is increased, the number of neighboring instances also increases, leading to a higher number of candidate pattern evaluations and a larger volume of row instances. Consequently, a significant increase in memory consumption.

For DTC and DTkC algorithms, since they are not dependent on the minimum distance threshold, the memory consumption is always stable and lower compared to the aforementioned algorithms. However, DTkC exhibits significantly lower memory consumption than the DTC algorithm. This is attributed to the utilization of the depth-first clique search strategy instead of the merging operation used in DTC to generate row instances.



(a) Beijing d = 250m (b) Las Vegas d = 125m



(c) Toronto d = 125m

Figure 11: The performance of compared algorithms on different prevalence thresh olds.

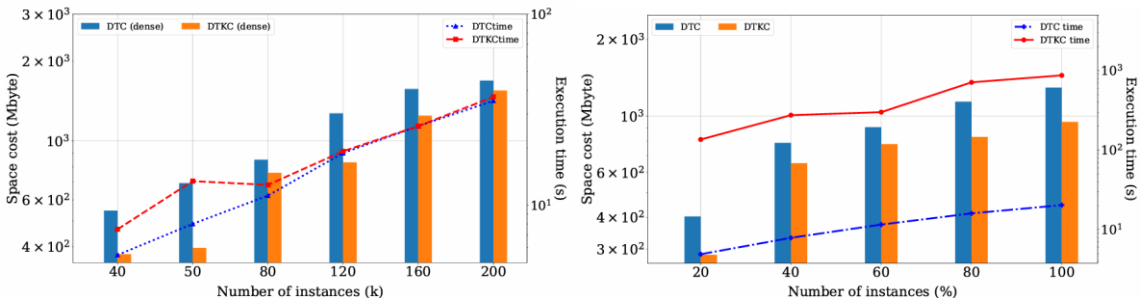
On different prevalence thresholds: Figure 11 describes the performance of the five algorithms on different prevalence thresholds. We observe that with a smaller prevalence threshold (e.g., 0.1), the computation times of Joinless and Condense are significantly large. Although the computation times decrease when the prevalence threshold is increased, they still remain considerably high compared to DTC and DTkC.

The memory consumption of Joinless and Condense decreases to some extent when the prevalence threshold increases. This is because these algorithms start the computation from small candidate co-locations and gradually expand to larger ones. As a result, when the prevalence threshold increases, the number of high size patterns decreases, leading to a decrease in the number of candidate co-locations that need to be computed.

DTC and DTkC utilize obtaining table instances first and C-Hash structures for candidate co-locations, respectively. Therefore, changing the prevalence threshold does not significantly impact memory consumption during the computation of PIs for candidate co-locations and the filtering of prevalent patterns. However, DTkC still exhibits significantly lower memory consumption.

4.3 Evaluate the scalability of DTkC

On different numbers of instances: Figure 12 shows the computation time of the DTC and DTkC algorithms with an increasing number of instances (min_{prev} = 0.2, k = 2). We can see that in Figure 12(a), the dataset is dense and the number of instances is large, the computation time for both the merging step and the depth-first clique search increases significantly. However, in Figure 12(b), only the computation time for DTkC increases rapidly, while the computation time for DTC increases at a slower pace. This occurs because in the less dense dataset, DTC generates fewer new polygons through the merging method, whereas DTkC has an increasing number of new cliques along with the some of sub-cliques that were not eliminated all by the depth-first clique search. As a result, it takes more time to traverse the larger I-tree structure.



(a) Synthetic dataset (b) UK dataset

Figure 12: Space cost and execution time on different numbers of instances.

However, in both datasets in Figure 12, the space cost of DTkC is always smaller than that of DTC. This is because the depth-first clique search generates fewer candidate co-locations compared to the merging method in DTC. In addition, another reason is that dataset in Figure 12(b) has more features compared to Figure 12(a), resulting in a larger average number of neighbors generated in DTkC compared to DTC. This also leads to a significantly larger number of cliques to traverse in the I-tree due to the higher average number of neighbors in DTkC, while the average number of neighbors in DTC is around 4-5.

Based on the above analysis, it can be seen that the DTkC algorithm is more suitable for dense datasets compared to DTC and datasets have fewer features. This is because it effectively reduces space cost while keeping the increase in execution time relatively low compared to the DTC algorithm.

On different values of k: Finally, we compare the scalability of DTkC based on the parameter k and the number of instances. It can be observed that both the execution time and space cost of DTkC show an increasing trend. However, the increase in space cost is not significant compared to the rate of increase in execution time. It is evident that the rapid increase in execution time of DTkC is an inherent characteristic of this dense dataset. However, it is uncommon to choose a large value for k, as the average number of neighbors can reach tens or more. The advantage lies in the fact that the space cost does not increase significantly, which is beneficial when dealing with dense data.

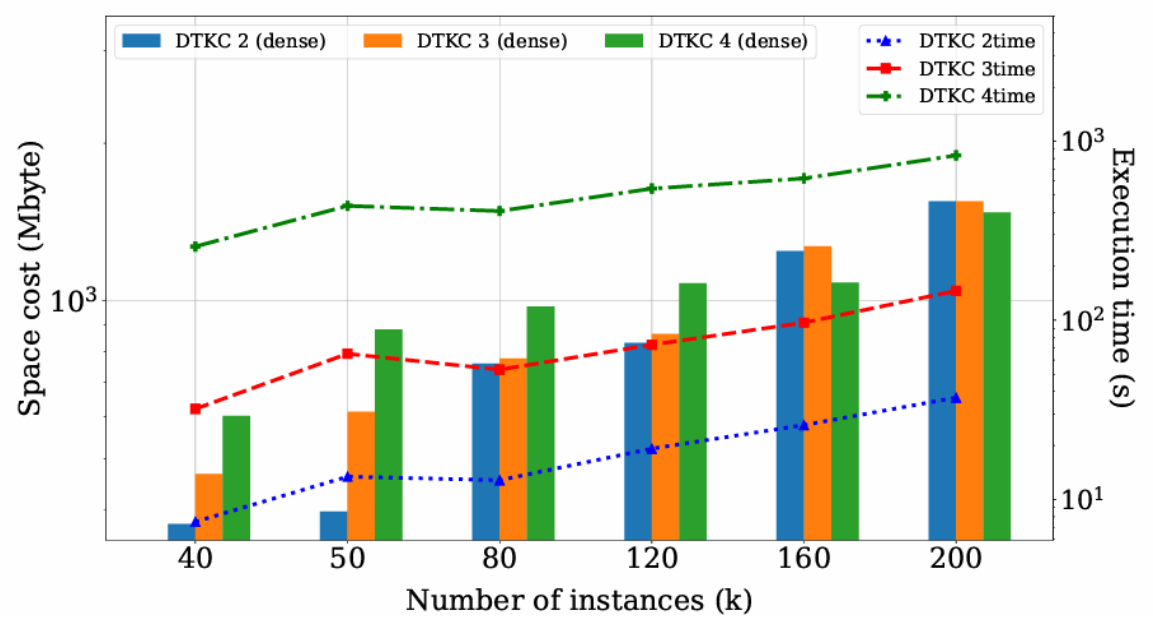


Figure 13: Space cost and execution time on different k values (min_{prev} = 0.2).

5 Conclusion

Mining PSCPs based on a distance threshold is challenging for users as it often leads to either missing or excessive patterns that may not align with their research objectives because it is difficult to find a suitable value of the thresh old. This work proposes a combined algorithm called DTkC, which leverages a Delaunay triangulation-based approach to address the issue of determining neighbor relationships. The algorithm also incorporates the concept of k-order neighbors, allowing users to choose neighbor hierarchies based on their specific research needs rather than being limited to a fixed number of neighbors. Furthermore, DTkC uses a depth-first clique search strategy, resulting in enhanced computational efficiency and reduced memory consumption. The efficiency of DTkC is proved on both real and synthetic datasets and in multiple respects.

In the future, we aim to improve the performance of the DTkC algorithm for datasets with moderate distribution density or datasets with a large number of features. As the parameter k increases by only one unit, the number of collected co-location patterns significantly increases. To address this issue and obtain a more accurate count of co-location patterns while avoiding data redundancy, we intend to explore and integrate suitable methods or techniques into the algorithm.

References

[1] Andrzejewski, W., Boinski, P.: Efficient spatial co-location pattern mining on multiple gpus. Expert Systems with Applications 93, 465–483 (2018).

[2] Bao, X., Wang, L.: A clique-based approach for co-location pattern mining. Information Sciences 490, 244–264 (2019).

[3] Lee, I., Phillips, P.: Urban crime analysis through areal categorized multivariate associations mining. Applied Artificial Intelligence 22(5), 483–499 (2008).

[4] Leibovici, D.G., Claramunt, C., Le Guyader, D., Brosset, D.: Local and global spatio-temporal entropy indices based on distance-ratios and co-occurrences dis tributions. International Journal of Geographical Information Science 28(5), 1061 1084 (2014).

[5] Li, J., Adilmagambetov, A., Mohomed Jabbar, M.S., Zaïane, O.R., Osornio-Vargas, A., Wine, O.: On discovering co-location patterns in datasets: a case study of pollutants and child cancers. GeoInformatica 20(4), 651–692 (2016).

[6] Luna, J.M., Fournier-Viger, P., Ventura, S.: Frequent itemset mining: A 25 years review. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 9(6), e1329 (2019).

[7] Qian, F., Chiew, K., He, Q., Huang, H.: Mining regional co-location patterns with k nng. Journal of Intelligent Information Systems 42, 485–505 (2014).

[8] Sundaram, V.M., Paneer, P., et al.: Discovering co-location patterns from spatial domain using a delaunay approach. Procedia engineering 38, 2832–2845 (2012).

[9] Tran, V., Wang, L.: Delaunay triangulation-based spatial colocation pattern mining without distance thresholds. Statistical Analysis and Data Mining: The ASA Data Science Journal 13(3), 282–304 (2020).

[10] Tran, V., Wang, L., Chen, H., Xiao, Q.: Mcht: A maximal clique and hash table based maximal prevalent co-location pattern mining algorithm. Expert Systems with Applications 175, 114830 (2021).

[11] Wang, L., Bao, Y., Lu, J., Yip, J.: A new join-less approach for co-location pattern mining. In: 2008 8th IEEE International Conference on Computer and Information Technology. pp. 197–202. IEEE (2008).

[12] Wang, L., Bao, Y., Lu, Z.: Efficient discovery of spatial co-location patterns using the icpi-tree. The Open Information Systems Journal 3(1) (2009).

[13] Yang, X., Cui, W.: A novel spatial clustering algorithm based on delaunay triangulation. In: International Conference on Earth Observation Data Processing and Analysis (ICEODPA). vol. 7285, pp. 916–924. SPIE (2008).

[14] Yao, X., Jiang, X., Wang, D., Yang, L., Peng, L., Chi, T.: Efficiently mining max imal co-locations in a spatial continuous field under directed road networks. Infor mation Sciences 542, 357–379 (2021).

[15] Yoo, J.S., Boulware, D., Kimmey, D.: A parallel spatial co-location mining algo rithm based on mapreduce. In: 2014 IEEE international congress on big data. pp. 25–31. IEEE (2014).

[16] Yoo, J.S., Boulware, D., Kimmey, D.: Parallel co-location mining with mapreduce and nosql systems. Knowledge and Information Systems 62, 1433–1463 (2020).

[17] Yoo, J.S., Bow, M.: A framework for generating condensed co-location sets from spatial databases. Intelligent Data Analysis 23(2), 333–355 (2019).

[18] Yoo, J.S., Shekhar, S.: A joinless approach for mining spatial colocation patterns. IEEE Transactions on Knowledge and Data Engineering 18(10), 1323–1337 (2006).

[19] Yoo, J.S., Shekhar, S., Smith, J., Kumquat, J.P.: A partial join approach for mining co-location patterns. In: Proceedings of the 12th annual ACM international workshop on Geographic information systems. pp. 241–249 (2004).

[20] Deng, Min, et al. "An adaptive spatial clustering algorithm based on Delaunay triangulation." *Computers, Environment and Urban Systems* 35.4, 320-332 (2011)