On Spatial Pattern Matching

Yixiang Fang[1], Reynold Cheng[1], Gao Cong[2], Nikos Mamoulis[3], Yun Li[4]

[1]*The University of Hong Kong,* [2]*Nanyang Technological University,* [3]*University of Ioannina,* [4]*Nanjing University*

[1]{yxfang,ckcheng}@cs.hku.hk, [2]gaocong@ntu.edu.sg, [3]nikos@cs.uoi.gr, [4]liycser@gmail.com

***Abstract*— In this paper, we study the *spatial pattern matching* (SPM) query. Given a set D of spatial objects (e.g., houses and shops), each with a textual description, we aim at finding all combinations of objects from D that match a user-defined *spatial pattern* P. A pattern P is a graph where vertices represent spatial objects, and edges denote distance relationships between them. The SPM query returns the instances that satisfy P. An example of P can be "a *house* within 10-minute walk from a *school*, which is at least $2km$ away from a *hospital*". The SPM query can benefit users such as house buyers, urban planners, and archaeologists. We prove that answering such queries is computationally intractable, and propose two efficient algorithms for their evaluation. Extensive experimental evaluation and cases studies on four real datasets show that our proposed solutions are highly effective and efficient.**

I. Introduction

Emerging location-based services (e.g., Google Maps) have raised plenty of research interest [1], [2], [3], [4], [5], [6]. Particularly, the spatial-keyword query (SKQ) (e.g., [7], [3], [8], [4]) has been extensively studied. In general, an SKQ returns sets of spatial objects whose locations are close to each other, and whose descriptions are relevant to a set of user-given text strings (called *keyword set*). The keyword set reflects the kinds of objects that a user is interested. A typical SKQ is the mCK query [7], [3], which finds, given a spatial database D and a keyword set Q, the set of spatial objects from D, such that they cover all the keywords of Q, and the maximum distance between any pair of objects is minimized. In Figure 1, for example, D comprises spatial objects labeled with different keywords (e.g., *park* and *station*). Suppose that Q={*house, school, hospital*}, the answer to the mCK query is the set of objects circled in dashed line in the figure.

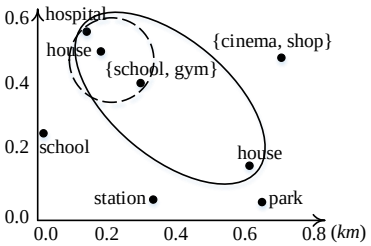
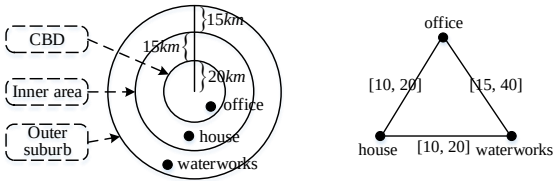


Fig. 1. An mCK query with Q={*house, school, hospital*}.

Although SKQs are useful, they may not be able to precisely capture the user's intention. Suppose that a user wishes to purchase a house, which is close to a school and a hospital. Moreover, while the user does not want the hospital to be too close to her living space (e.g., for hygienic reasons), she wishes the hospital to be accessible in a reasonable distance. A hospital between $0.5km$ and $2km$ from the house would be desirable. This request may not be answered by an SKQ (e.g., [3], [4]) – a user may always get instances where the three objects are close to each other. The objects circled in the solid ellipse of Figure 1 are in fact those that interest her most.

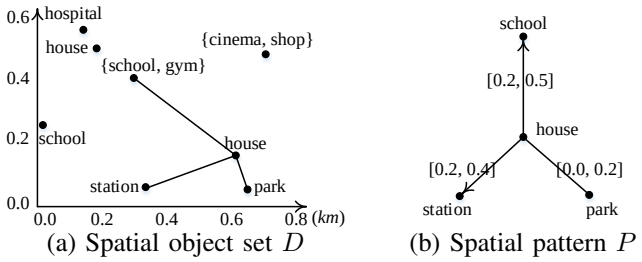


(a) human settlement area (b) a spatial pattern

Fig. 2. The human settlement and a spatial pattern [9].

Let us consider another example where specifying spatial relationships for query keywords is important. In geography domain, *human settlement* is the study of the human land-use patterns, or the "evidence within a given region of the physical remnants of communities and networks" [10]. This topic is interesting to urban planners and archaeologists. Figure 2 illustrates a human settlement [9]. An urban planning expert may conjecture that in a certain city, an office is located in the CBD (Central Business District); a house is in the inner city; a waterworks is built in the outer suburbs. He/she would like to retrieve objects for (*office*, *house*, *waterworks*), which are respectively located in the CBD, the inner city, and outer suburbs. The objects retrieved can be the subject of further analysis and case studies. In this example, the three kinds of objects interesting to the user, located in different areas, are separated by some distance constraints (e.g., each pair of object has a distance in a certain range). However, these distance relationships between keywords cannot be expressed in an existing SKQ.

To allow spatial relationships among keywords to be conveniently specified, we propose the *spatial pattern matching* (SPM) query. As shown in Figure 3, given a spatial database D (in (a)) and a *spatial pattern* P (in (b)), SPM finds all the instances of P in D. Notice that P is a graph, where each vertex corresponds to an object with a keyword attached, and each edge is augmented with a spatial distance relationship. For example, the user can specify that the house found should be within the vicinity of $[0.2, 0.5]$ (km) from a school. In a number of countries (e.g., Singapore), if a student lives within a particular vicinity (e.g., $0.5km$ or $1km$) of a school p, then he/she has a high chance to be admitted to p [11]. The user



(a) Spatial object set D (b) Spatial pattern P

Fig. 3. Illustrating the SPM query.

may also want the house to be at least $0.2km$ from the school to avoid noise. In this example, the four objects connected in solid lines, which satisfy all the constraints of the spatial pattern P, is an instance (or a *match*) of P. In Figure 2(b), the spatial pattern for the human settlement example is shown.

As discussed before, an SKQ (e.g., [7], [3], [8], [4]) can only return objects that are spatially close to each other. SPM queries are reminiscent to multi-way spatial joins studied in previous work [12], [13]. However, those solutions are not designed to use keywords and exclusion relationship (to be discussed later) to find spatial pattern instances. As a result, pure spatial indexes, such as the R-tree, cannot be used unless they are built on-the-fly for each vertex, which is typically expensive. Another related topic is graph pattern matching (GPM) [14], [15], which aims at finding subgraphs matching a query pattern from a large graph. However, using GPM techniques to solve SPM problems is not straightforward because (1) the spatial patterns associated with distance intervals and inclusion/exclusion-ship are different with graph patterns, and (2) the solutions to the GPM problem are mainly designed for graphs, rather than spatial objects which are often indexed by R-tree like structures. To adapt the GPM solutions for solving the SPM queries, we first have to transform the set of spatial objects involved (e.g., Figure 3(a)) into a graph, and then run a GPM algorithm on it. As shown by experiments in Section V, the adapted GPM solutions (i.e., [14], [15]) are very inefficient, calling for faster solutions.

Our contributions. We present a formal definition of a spatial pattern. We propose several distance constraints for a spatial pattern, which specify (1) minimum and maximum distances between two object types; and (2) *exclusion* and *inclusion*. Figure 3(b) illustrates the exclusion relationship ($\rightarrow$), which expresses that (1) a school should be at least $0.3km$ from a house but not more than $1.0km$; and (2) no school should be in the vicinity of $0.3km$ of a house. We then define the SPM problem and show that it is NP-hard. To answer the query, we propose two efficient algorithms. The first one, called multi-pair-join (or `MPJ`), is adapted from the multi-way join [16], [14] considering edges of the spatial pattern. We also develop a sampling-based estimation method to guide the execution order of the joins. Since this solution follows existing multi-way join, it is easy to implement. We further develop a faster solution customized for SPM queries. This solution, called the multi-star-join (or `MSJ`), derives the lower and upper bounds of distances between object instances based on dynamic programming. We also introduce two pruning criteria to improve query performance.

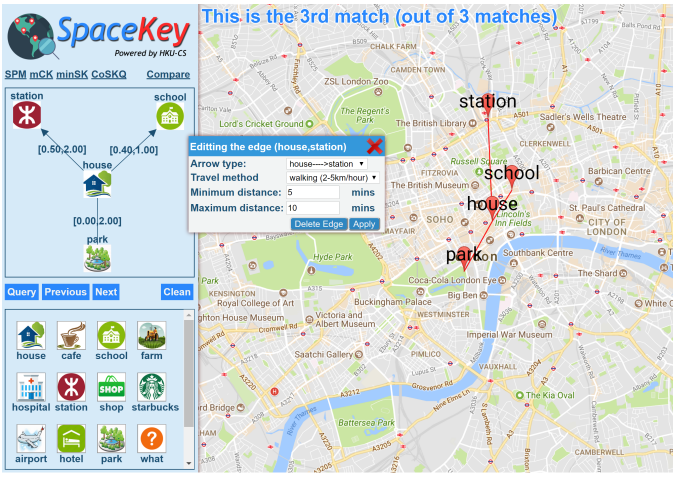


Fig. 4. The user interface of SpaceKey [17].

We have experimentally evaluated our proposed methods on real datasets, which shows that our best solution is over an order of magnitude faster than the baselines adapted from GPM solutions. We have also conducted a case study. The results show that SPM queries typically return more relevant results for target applications than state-of-the-art SKQ solutions.

In addition, we have developed a system called *SpaceKey*, which supports the SPM query. Its user interface is shown in Figure 4. To draw a pattern, a user can drag icons (representing keywords) from the panel (bottom-left) to create vertices (top-left), and then create edges by linking pairs of icons. Their distance intervals and relationship can be edited using the panel (bottom-right) [1]. After clicking the "Query" button, the user can view the matches on the map one by one. It also allows users to visually compare the results of different SKQs. More details of introducing *SpaceKey* could be found in [17].

Organization. We formulate the SPM problem in Section II. Sections III and IV present our solutions `MPJ` and `MSJ` respectively. We report experimental results in Section V. We review related work in Section VI and conclude in Section VII.

II. Problem Definition

A. The SPM Problem

Let D be a database of spatial objects (or *objects* for brevity). Each object $o_i \in D$ $(1 \leq i \leq |D|)$ has 2D coordinates (x_i, y_i), and is associated with a set of keywords, denoted by $doc(o_i)$. In Figure 3(a), for example, the object at (0.55, 0.1) has a keyword "house". We say that o_i *matches* with a keyword w, if $w \in doc(o_i)$. Given two objects o_i and o_j, we use $|o_i, o_j|$ to denote their Euclidean distance. We denote a spatial circle with center o and radius r by $O(o, r)$. Table I summarizes the notations used in the paper.

[1]The user can input the lower/upper bounds of the intervals based on his experience and expertise. Alternatively, the system can be designed to give suggestions, based on, for instance, the previous users' inputs or query results.

Let us now study the definition of spatial pattern.

Definition 1 (spatial pattern [2]). *A spatial pattern P is a simple graph (V, E) of n vertices $\{v_1, v_2, \cdots, v_n\}$ and m edges, and the following constraints hold:*

- *Each vertex $v_i \in V$ has a keyword w_i;*
- *Each edge $(v_i, v_j) \in E$ has a distance interval $[l_{i,j}, u_{i,j}]$, where $l_{i,j}$ ($u_{i,j}$) is the lower (respectively upper) bound of distances between two matching objects in D;*
- *Each edge $(v_i, v_j) \in E$ is associated with one of the signs: (1) $v_i \rightarrow v_j$; (2) $v_i \leftarrow v_j$; (3) $v_i \leftrightarrow v_j$; and (4) v_i–v_j.*

To illustrate Definition 1, consider the edge *house*→*school* with distance interval [0.3, 1.0] (*km*) in the pattern of Figure 3(b). Intuitively, the user wishes to retrieve two objects (say, o_s and o_t) such that: (1) o_s and o_t have keywords *house* and *school* respectively; (2) the distance of o_s from o_t is between $0.3km$ and $1.0km$; and (3) there does not exist any object with keyword *school*, which is less than $0.3km$ from o_s. We say that *house excludes school*, denoted by *house*→*school*, to express the user's wish of not getting any match where a *school* object is found to be less than $0.3km$ from a *house* object. This can be useful to a user who wants to find a house that is not too close to a school (e.g., to avoid the noise and crowd caused by school). Let (v_i, v_j) be an edge in E, with distance interval $[l_{i,j}, u_{i,j}]$. Also, let o_k and o_l be the two objects returned in a match of E, where $w_i \in doc(o_k)$ and $w_j \in doc(o_l)$. We now discuss the four possible *signs* of an edge in Definition 1:

- $v_i \rightarrow v_j$ [v_i excludes v_j]: No object with keyword w_j in D should have a distance less than $l_{i,j}$ from o_k.
- $v_i \leftarrow v_j$ [v_j excludes v_i]: No object with keyword w_i in D should have a distance less than $l_{i,j}$ from o_l.
- $v_i \leftrightarrow v_j$ [mutual exclusion]: No object with keyword w_j in D should have a distance less than $l_{i,j}$ from o_k, and the distance of any object with keyword w_i in D should be at least $l_{i,j}$ away from o_l.
- v_i–v_j [mutual inclusion]: The occurrence of any object (other than o_k and o_l) with keywords w_i and w_j in D with distance shorter than $l_{i,j}$ is allowed.

For example, in the pattern of Figure 3(b), *house excludes school*, and *house* has a *mutual inclusion* with *park*.

Remarks. The notion of spatial pattern can be extended to support other query requirements. For example, each vertex of P may carry multiple keywords. Also, the distance constraint can be changed, in order to express that the distance between two objects is within multiple distance intervals. Although we assume the distance metric is Euclidean, other measures, such as the road network distance, can also be considered.

For convenience, we use $nb(v_i)$ to denote the set of neighbors of vertex $v_i \in P$. We say that two spatial objects o_k and o_l form an *e-match* of an edge (v_i, v_j), as follows:

Definition 2 (e-match). *Two objects o_k and o_l constitute an e-match of (v_i, v_j), if o_k and o_l match with w_i and w_j respectively, and satisfy the distance constraints of (v_i, v_j).*

[2]In context without ambiguity, we simply call it a *pattern*.

TABLE I
NOTATIONS AND MEANINGS.

Notation	Meaning
D	set of spatial objects
$o_i(x_i, y_i)$	spatial object in D, with 2D coordinates (x_i, y_i)
$doc(o_i)$	set of keywords of o_i
P	spatial pattern with vertex and edge sets V and E
n, m	numbers of vertices and edges in V and E
v_i, w_i	vertex v_i with keyword w_i in P
$[l_{i,j}, u_{i,j}]$	distance interval on edge (v_i, v_j)
$nb(v_i)$	set of neighbor vertices of $v_i \in P$
$\widehat{P}$	bounded pattern of P
$O(o, r)$	circle with center o and radius r
$\|o_i, o_j\|$	the Euclidean distance between o_i and o_j
Γ	join order (in the form of a list of edges)
Ψ	SPM query result set

Definition 3 (match). *Given a spatial pattern P and a set S of objects, S is a match of P if: (1) for each edge of P, there is an e-match in S; and (2) there does not exist any proper subset S' of S such that for each edge of P, there is an e-match in S'.*

Problem 1 (Spatial Pattern Matching). *Given a spatial pattern P, SPM returns all the matches of P in D.*

In Figure 3(a), for instance, the four objects connected in solid lines is a match of the pattern in Figure 3(b), and it is the answer of this SPM query. We call a set of objects a *partial* match of P, if it is a match of a subgraph of P. For example, in Figure 3(a), any two or three linked objects are a partial match of the pattern in Figure 3(b).

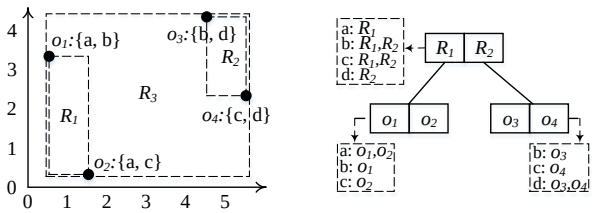
Lemma 1 (Hardness). *The SPM problem is NP-hard.*

The SPM problem can be reduced from the classical 3-SAT problem. The proofs of all the lemmas can be found in the full version of this paper [18]. A naive solution to solve the SPM problem takes $O(|D|^n)$ time, which is exponential to the number of vertices n. However, in practice the size of the pattern is often small, which motivates us to develop efficient exact algorithms despite the intractability.

B. Baseline Solutions: S-MDJ and S-VF3

To solve the SPM problem, we propose basic solutions by adapting the existing solutions to GPM [14], [15], which aim to find subgraphs that match a graph pattern in a graph. Given an SPM query, the main idea is that we first create an graph G using P, then simply pattern P to another pattern P' by removing its distance intervals and signs, and find all the matches of P' from G using a GPM solver. Specifically, we have the following three steps.

Step-1: For each edge (v_i, v_j) of P, we find a set O_i of objects that match with w_i. For each object $o \in O_i$ we perform two range queries in $O(o, l_{i,j})$ and $O(o, u_{i,j})$, to get their answers $L_{i,j}$ and $U_{i,j}$ which contain objects matched with w_j, respectively. Note that, if v_i excludes v_j and $L_{i,j} \neq \emptyset$, then we skip o. Next, for each object o' in $U_{i,j} \backslash L_{i,j}$, (o, o') forms an e-match of (v_i, v_j). As a result, we can get all the



(a) A set of objects (b) An IR-tree

Fig. 5. An example of IR-tree.

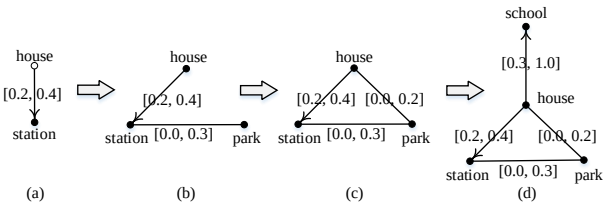


Fig. 6. Illustrating the process of MPJ.

e-matches of this edge. In case that the sign of the edge is $v_i \leftarrow v_j$ or $v_i \leftrightarrow v_j$, we can perform a similar computation.
Step-2: For the two objects in each e-match, we create two vertices with w_i and w_j resp., and link them with an edge.
Step-3: We generate pattern P' by removing distance intervals and signs from P. Afterwards, any GPM solution can be applied to extracting all the matches of P' from G.

In this paper, we use two GPM solutions MD-Join [14] and VF3 [15] and denote the adapted algorithms by S-MDJ and S-VF3 respectively. Their time complexities are $O(m|D|^2 + |D|^n)$, since there are at most $|D|^2$ e-matches for each edge and the total number of matches could be $|D|^n$.

In Step-1, we need to perform keyword search and range queries over the dataset D. To facilitate this step, we use the IR-tree structure [1] to index the objects in D. To build the IR-tree, we first build an R-tree and then associate an inverted file to each node [3] as follows. In each leaf node, each keyword is associated with a postings list, i.e., the list of objects containing the keyword. In the inverted file of each non-leaf node, each keyword is associated the list of child nodes containing it. Figure 5(a) gives an example of four objects $\{o_1, \cdots, o_4\}$, and the IR-tree built for these objects is depicted in Figure 5(b). The inverted files of nodes are described in the dashed rectangle boxes.

III. The MPJ Algorithm

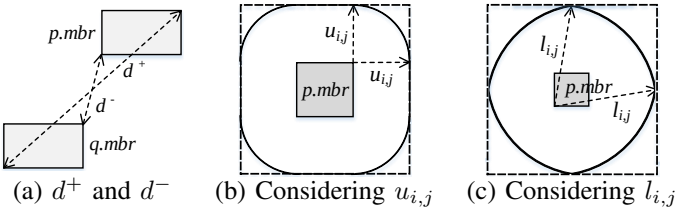
The major problem of baseline solutions is that, to answer an SPM query, it needs to generate a graph G and a pattern P', before running a GPM solution. This may not be efficient, when D is large. To improve the performance, in this section we propose a multi-pair-join (MPJ) algorithm by adapting the classical multi-way join [14], which is easy to implement.

We first propose a join algorithm called pair-join (PJ) to find all the e-matches for each edge of P. Based on PJ, we develop the MPJ algorithm, which joins these e-matches of single edges, according to a particular order, to obtain all the matches of P. In Figure 6, we show the query process of MPJ for the pattern in Figure 6(d) with a particular join order. We first present PJ in Section III-A, then discuss the join order and the MPJ algorithm in Sections III-B and III-C respectively.

A. The PJ Algorithm

We first consider edges with signs v_i–v_j and $v_i \rightarrow v_j$. We will consider the other two kinds of signs later. To compute the e-matches of an edge, we assume that there is an IR-tree built for D. The rationales of adopting the IR-tree index are two-fold: (1) IR-tree is a kind of R-tree, and it can easily handle edges with both inclusion-ship and exclusion-ship in the join process; (2) The IR-tree has been demonstrated to be very efficient for joint spatial keyword queries [19]. Next, we will discuss these two advantages in more detail.

Given an edge, PJ exploits the IR-tree and finds *matched* pairs of non-leaf nodes level by level in a top-down manner, and finally finds all the e-matches in the leaf level. We now illustrate the concept of *matched* pairs of nodes. Let p and q be two non-leaf nodes, whose inverted files contain w_i and w_j respectively, in the same level of the IR-tree. We define their MBRs' maximum distance d^+ as the maximum distance between any two points in their MBRs. Their minimum distance d^- can be defined similarly. Figure 7(a) illustrates d^+ and d^-. We call (p, q) a *matched* pair of nodes, if $[d^-, d^+] \cap [l_{i,j}, u_{i,j}] \neq \emptyset$.



(a) d^+ and d^- (b) Considering $u_{i,j}$ (c) Considering $l_{i,j}$

Fig. 7. Illustrating the candidate pairs in PJ.

Intuitively, if p and q's MBRs are far from, or too close to each other, then we cannot find any e-match from their MBRs. We illustrate this using Figures 7(b) and 7(c). If q's MBR does not intersect with the outer area bounded with solid line in Figure 7(b), then it cannot be a candidate of p, since their distance must be larger than $u_{i,j}$; Similarly, if q's MBR is fully covered by the outer area bounded with solid line in Figure 7(c), then q cannot be a candidate of p, since their distance must be less than $l_{i,j}$. By finding the candidate pairs level by level, we can safely prune a large number of unmatched pairs of nodes. Algorithm 1 presents PJ.

The input of PJ is the $root$ of an IR-tree, and an edge (v_i, v_j)$\in P$, where λ denotes the sign from v_i to v_j, and the output is Φ, a map of all the e-matches. It first initializes h, the height of the IR-tree, and Φ, a map where the key is a node/object and the value is the list of its candidates (line 1). Then, it initializes a matched pair for the $root$ node (line 2). Next, it finds candidate pairs level by level (lines 3-19). Specifically, we enumerate all the candidate pairs in Φ (lines 5,8). Note

[3]To avoid ambiguity, we use "node" to mean "IR-tree node", and "vertex" to mean "vertex" of spatial pattern in this paper.

that $\Phi.keySet$ gets all the keys of Φ and $\Phi.getKey(p)$ returns the value of the key p. For each pair (p, q), we get its child pairs which contain w_i and w_j respectively by checking their inverted files using function *invFile(w)* (lines 7,9). For each child pair (p', q'), we compute its MBRs' maximum and minimum distances (lines 10-11). Note if p is a leaf node, d^+ and d^- equal to $|p', q'|$. If v_i excludes v_j and d^+ is less than $l_{i,j}$, we mark the boolean variable $flag$ as true and skip p' (lines 12-14,17). Otherwise, if it is a matched pair, we put q' into Λ, a list for collecting p''s candidates (lines 15-16). After that, p' and its candidates are collected into a new map Φ' (line 18). The map Φ of candidate pairs is updated level by level (line 19). Finally, we return Φ (line 20).

Algorithm 1: PJ

```
   Input: root, w_i, w_j, [l_{i,j}, u_{i,j}], λ;
   Output: Φ, all the e-matches;
1  h ← height(root), Φ ← ∅;
2  Λ.add(root), Φ.add(root, Λ);
3  for i ← 1 to h do
4      Φ' ← ∅;
5      for p ∈ Φ.keySet() do
6          Λ ← ∅, flag ← false;
7          for p' ∈ p.invFile(w_i) do
8              for q ∈ Φ.getKey(p) do
9                  for q' ∈ q.invFile(w_j) do
10                     d⁻ ← MinDist(p'.mbr, q'.mbr);
11                     d⁺ ← MaxDist(p'.mbr, q'.mbr);
12                     if d⁺<l_{i,j} then
13                         if λ is "→" then
14                             flag ← true; break;
15                     else if d⁻ ≤ u_{i,j} then
16                         Λ.add(q');
17                 if flag=true then break;
18             if flag=false then Φ'.add(p', Λ);
19     Φ ← Φ'; //update Φ
20 return Φ;
```

We now consider edges with other two kinds of edges. For $v_i \leftarrow v_j$, we can reverse it as $v_j \rightarrow v_i$ and run PJ directly. For $v_i \leftrightarrow v_j$, we can run PJ for edges $v_i \rightarrow v_j$ and $v_j \rightarrow v_i$ separately, and then return the e-matches satisfying both of them, i.e., the intersection of these two sets of e-matches.

B. The Join Order for MPJ

The order of performing joins for the edges has a significant effect on the overall performance [16], [14]. We illustrate this by two orders of performing PJ in Example 1.

Example 1. *Consider a pattern P with vertices $\{v_1, v_2, v_3\}$, and edges $\{v_1$–v_2, v_2–v_3, v_3–$v_1\}$. Suppose there are 2, 50, and 1000 e-matches for these edges respectively.* □

Order1: We run PJ for edges v_1–v_2 and v_2–v_3 first, and then get at most 100 tuples for v_1–v_2–v_3 by linking their results. Then for v_3–v_1, we do not need to run PJ, since we can just need to scan each tuple and check whether the distance between the third and first objects is in $[l_{1,3}, u_{1,3}]$.

Order2: We consider edges v_2–v_3 and v_3–v_1 first, and then may get 50,000 tuples for v_2–v_3–v_1. Next, for v_1–v_2, we check whether each of these tuples satisfies its distance constraint.

Clearly, *Order1* tends to need less computational cost than *Order2*. The reason is that for edges with mutual inclusion (e.g., v_3–v_1 in *Order1*), we may skip performing PJ for them, because we can scan the linked tuples and check their distance constraints. However, for edges with other signs, we cannot skip them. For example, let us modify Example 1 by replacing v_3–v_1 with $v_1 \rightarrow v_3$ and using *Order1*. For any tuple $<o_1,o_2,o_3>$ matched with v_1–v_2 and v_2–v_3, we cannot claim it is a match of P, even if $l_{1,3} \leq |o_1, o_3| \leq u_{1,3}$. This is because, there may exist other objects matched with w_3 in the circle $O(o_1, l_{1,3})$, which invalidates this tuple.

Intuitively, a good join order should avoid performing PJ for edges having large numbers of e-matches with mutual inclusion. How to quickly estimate the numbers of e-matches for such edges without running PJ? Some existing cost models are based on R-trees [20] and density histograms [21]. However, these models assume that the entire dataset(s) are possible instances of each node, whereas in our case the pattern instances include only objects that satisfy the keyword constraints at each vertex. In addition, as shown in Figure 7, the regions to be queried in SPM are irregular, i.e., they are neither circles nor rectangles, which renders approaches based on rectilinear space division inaccurate. To address this issue, we propose an effective and efficient estimation method.

Estimation. Consider vertices v_i and v_j with mutual inclusion, i.e., v_i–v_j. Let O_i and O_j be the sets of objects matched with w_i and w_j respectively. We consider a random pair (o_i, o_j) of objects, where $o_i \in O_i$ and $o_j \in O_j$, as a random variable. We propose Lemma 2, which states that, by sampling certain number of matched pairs, we can accurately estimate the number r of e-matches.

Lemma 2 (Estimation)**.** *Suppose p (p>0) is the probability that a random pair is a matched pair. Let X_i be the number of sampled pairs to see the i-th matched pair after seeing the (i–1)-th matched pair. Let the total number of sampled pairs to see s matched pairs be $Y=\sum_{i=1}^{s} X_i$. Then, for any 0<ϵ<1,*

$$\Pr\left(|Y - E[Y]| \geq \epsilon E[Y]\right) \leq \delta, \tag{1}$$

where $\delta=\exp\left(-\frac{s\epsilon^2}{8}\right)$.

It is easy to observe that, the random variables X_i's follow the geometric distribution with success probability p, and so the expectation is $\frac{1}{p}$ [22]. Since $Y=\sum_{i=1}^{s} X_i$, we get $E[Y]=\frac{s}{p}$ and also $p=\frac{s}{E[Y]}$. On the other hand, since there are $|O_i| \cdot |O_i|$ pairs and r matched pairs, we have $p=\frac{r}{|O_i| \cdot |O_i|}$. Thus, we conclude $E[Y]=\frac{s}{r} \cdot |O_i| \cdot |O_i|$. By Lemma 2, $E[Y]$ can be well approximated by Y. Hence, given ϵ and δ, we can sample pairs until seeing s matched pairs, where $s=O(\frac{8}{\epsilon^2} \ln \frac{1}{\delta})$, and well estimate $E[Y]$, which further implies $r \approx \frac{s}{Y} \cdot |O_i| \cdot |O_j|$.

To make this guarantee more concrete, consider the following example. Let ϵ and δ be 0.25. Then we have s=177, which means we can stop the sampling after seeing 177 e-matches. This is very efficient in practice if there are over thousands of e-matches. Note that, to avoid infinite sampling for the case

p=0, we introduce a threshold $\theta \in [0, 1]$, and stop sampling if we cannot see s e-matches after sampling $|O_i| \cdot |O_j| \cdot \theta$ pairs.

In addition, since the goal of the estimation is to determine a good join order for a query pattern, we only need to derive the topological orders of the cost of these edges. This means that, it may not be necessary to accurately estimate the cost, and thus we do not need to set very small values for ϵ and δ.

Order. The optimal order can be computed by dynamic programming [16]. However, this method is inefficient due to the large solution space [16], [14], beating the purpose of finding a good order faster than the time needed for query evaluation. To alleviate this issue, we propose an efficient greedy solution (i.e., heuristic query optimization), called `MPJOrder`. Specifically, we perform two steps: First, we perform `PJ` for edges that are not with mutual inclusion. Second, we randomly select a starting vertex, and perform graph search incrementally starting from this vertex. During the search process, we always greedily visit edges, whose estimated numbers of e-matches are the smallest, and put the visited edges into Γ, a list keeping the order. We call an edge a *forward* edge, if at least one of its vertices is not in edges of the current Γ, or a *backward* edge if all of its vertices are in edges of the current Γ.

Algorithm 2: `MPJOrder`

```
   Input: root, P, δ, ϵ, θ;
   Output: Γ, the join order of MPJ;
1  Γ ← ∅, Q ← ∅, U ← ∅, ϒ ← ∅;
2  for each edge (v_i, v_j) of P do
3      if v_i→v_j or v_i←v_j or v_i↔v_j then
4          Φ ← perform PJ for this edge;
5          ϒ.add((v_i, v_j), Φ);
6  randomly select a vertex v ∈ P, and add it to U;
7  for u ∈ nb(v) do
8      if v–u then Q.add((v, u), estimate(v–u));
9      else        Q.add((v, u), ϒ.get((v, u).size);
10 while Q.size > 0 do
11     (v_i, v_j) ← Q.pop();
12     Γ.add((v_i, v_j));
13     if v_i ∈ U and v_j ∈ U then continue;
14     v ← a newly considered vertex in (v_i, v_j) and U;
15     for u ∈ nb(v) ∧ U Γ.add((v, u));
16     for u ∈ nb(v) \ U do
17         if v–u then Q.add(v, u), estimate(v–u));
18         else        Q.add(v, u), ϒ.get(v, u).size);
19     U.add(v);
20 return Γ;
```

Algorithm 2 presents `MPJOrder`. Given an IR-tree, a pattern P, some parameters of estimation (δ, ϵ, and θ), it outputs the join order Γ. We first initialize some variables, where Γ is a list, Q is a priority queue in which edges are ranked by their estimated numbers of e-matches in ascending order, U keeps the visited vertices, and Υ maintains the join results for edges that are not with mutual inclusion. Then, we run `PJ` for edges that are not with mutual inclusion (lines 2-5). Next, we randomly select a vertex v and put its edges into Q (lines 6-9). Note that the function `estimate`(v-u) performs sampling to estimate the number of matched pairs. In the loop (lines 10-19), we first add the edge with the minimum number of e-matches to Γ (lines 11-12). If the edge is backward (line 13), we continue to dequeue an edge from Q; otherwise, we enqueue v's neighbors (lines 14-18). The new vertex v is marked as visited (line 19). Finally, Γ is returned (line 20).

We illustrate the steps of `MPJOrder` by Example 2.

Example 2. *Continue Example 1, and let v_1 be the starting vertex in* `MPJOrder`. *Q is initialized by two forward edges v_1–v_2 and v_1–v_3. First, we dequeue v_1–v_2, add it to Γ, and add v_2–v_3 to Q. Then, we dequeue v_2–v_3 and add it to Γ. Also, v_1–v_3 is added to Γ because it is a backward edge.* □

C. The MPJ Algorithm

After computing the join order by `MPJOrder`, we handle the edges one by one following the order, and link the results incrementally as illustrated by Figure 6. Specifically, for forward edges, we expand the partial matches, such that they match with a larger subgraph of P; while for backward edges, we prune some partial matches using the distance constraints. The detailed steps of `MPJ` are described in [18].

Lemma 3. *The time complexity of* `MPJ` *is* $O(m\theta|D|^2+|D|^n)$.

IV. The MSJ Algorithm

In this section, we propose a new algorithm called the multi-star-join (or `MSJ`). Compared with `MPJ`, the main advantages of `MSJ` are three-fold: First, we introduce a novel concept called *bounded pattern*, which can be computed by dynamic programming. We show that, it is not only useful for refining the pattern but also pruning partial matches during the join process. Second, `MSJ` determines the join order in a more efficient way, which does not rely on sampling. Third, in the join process `MSJ` considers the edges in a collective manner with two pruning criteria. With such optimizations, `MSJ` is able to achieve higher efficiency than `MPJ`, as shown in Section V.

A. The Bounded Pattern

The design of the bounded pattern is based on the key observation that, the distance between any two vertices in P can be bounded. We illustrate this by Example 3.

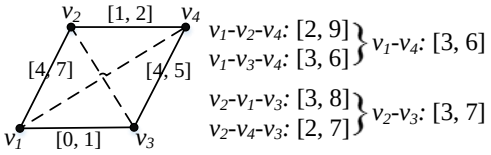


Fig. 8. Illustrating the bounded pattern.

Example 3. *Consider a pattern P in Figure 8 where the four edges are in solid lines. Since the distance intervals on v_1–v_2 and v_2–v_4 are [4, 7] and [1, 2] respectively, the lower and upper bounds of the distance from v_1 to v_4 are 2 and 9 by triangle inequality. Similarly, we can derive the bounds using v_1–v_3 and v_3–v_4. Thus, the distance between two objects matched with v_1 and v_4 in a match of P must be in [3, 6].* □

Given a spatial pattern P, we define its ***bounded pattern*** $\widehat{P}$ as a graph, which is a clique satisfying properties:

- There are n vertices $\{\widehat{v_1}, \widehat{v_2}, \cdots, \widehat{v_n}\}$;
- Each vertex is linked with each other vertex;
- $\forall(\widehat{v_i}, \widehat{v_j})$ of $\widehat{P}$, its distance interval $[\widehat{l_{i,j}}, \widehat{u_{i,j}}]$ is initialized as $[l_{i,j}, u_{i,j}]$ if $(v_i, v_j) \in P$, or $[0, +\infty]$ if $(v_i, v_j) \notin P$.
- The distance intervals on all the edges are further computed by dynamic programming using Lemmas 4 and 5.

Lemma 4 (Upper bound)**.** *The upper bound distance between any two vertices $\widehat{v_i}$ and $\widehat{v_j}$ in $\widehat{P}$ is*

$$\widehat{u_{i,j}} = \min_{1 \le k \le n} \{\widehat{u_{i,j}}, \widehat{u_{i,k}} + \widehat{u_{k,j}}\}. \tag{2}$$

Apparently, $\widehat{u_{i,j}}$ equals to the shortest path distance from $\widehat{v_i}$ to $\widehat{v_j}$, if we replace the distance interval on each edge $(\widehat{v_i}, \widehat{v_j})$ by a value $\widehat{u_{i,j}}$, so we can use Floyd-Warshall algorithm [23].

Lemma 5 (Lower bound)**.** *The lower bound distance between any two vertices $\widehat{v_i}$ and $\widehat{v_j}$ in $\widehat{P}$ is*

$$\widehat{l_{i,j}} = \max_{1 \le k \le n} \begin{cases} 0 & [\widehat{l_{i,k}}, \widehat{u_{i,k}}] \cap [\widehat{l_{k,j}}, \widehat{u_{k,j}}] \neq \emptyset \\ \widehat{l_{k,j}} - \widehat{u_{i,k}} & \widehat{u_{i,k}} < \widehat{l_{k,j}} \\ \widehat{l_{i,k}} - \widehat{u_{k,j}} & \widehat{l_{i,k}} > \widehat{u_{k,j}} \end{cases}. \tag{3}$$

Continue Example 3, and let i=1, j=4. When k=2, since $\widehat{l_{1,2}}$=4>$\widehat{u_{2,4}}$=2, we have $\widehat{l_{1,4}}$=2; when k=3, since $\widehat{u_{1,3}}$=1<$\widehat{l_{3,4}}$=4, we have $\widehat{l_{1,4}}$=3. Thus, we have $\widehat{l_{1,4}}$=3 by Lemma 5.

Refining patterns. We can observe that, when computing the lower and upper bound distances between any two vertices using Eqs (2) and (3), we have considered all the paths between them, and so they are *globally* tight. This implies, we can use them to refine P, which may reduce the query computational cost.

Let e=v_i–v_j be an edge with mutual inclusion. We have the following refining criteria:

❶ If $[l_{i,j}, u_{i,j}] \cap [\widehat{l_{i,j}}, \widehat{u_{i,j}}] = \emptyset$, then P is a wrong pattern, since no pair of objects can satisfy the distance constraint.

❷ If $[\widehat{l_{i,j}}, \widehat{u_{i,j}}] \subset [l_{i,j}, u_{i,j}]$, then we delete (v_i, v_j), since any set of objects matching with P–$\{e\}$ is also a match of P.

❸ If neither criterion ❶ nor criterion ❷ can be applied, then we refine $[l_{i,j}, u_{i,j}]$ as $[l_{i,j}, u_{i,j}] \cap [\widehat{l_{i,j}}, \widehat{u_{i,j}}]$, since any set of objects matched with P is also a match of $\widehat{P}$.

We illustrate above refining criteria by Example 4.

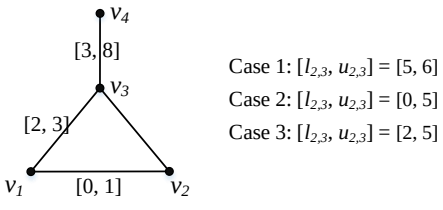


Fig. 9. Illustrating pattern refining.

Example 4. *Consider a pattern in Figure 9, and the edge e= (v_2, v_3) has three different cases. Note that $[\widehat{l_{2,3}}, \widehat{u_{2,3}}]$ is always a subinterval of [1, 4]. If $[l_{2,3}, u_{2,3}]$=[5, 6], then it is a wrong pattern by criterion ❶; if $[l_{2,3}, u_{2,3}]$=[0, 5], then we delete e by criterion ❷; and if $[l_{2,3}, u_{2,3}]$ =[2, 5], we update it as [2, 4] by criterion ❸.* □

If the relationship between v_i and v_j is not mutual inclusion, we simply replace criteria ❷ and ❸ by criterion ❹ as below.

❹ If $\widehat{u_{i,j}} < u_{i,j}$, we simply refine $[l_{i,j}, u_{i,j}]$ as $[l_{i,j}, \widehat{u_{i,j}}]$.

Notice that in criterion ❹, $l_{i,j}$ is not updated. The reason is that, if v_i excludes v_j, then for any objects o_s and o_t matched with w_i and w_j respectively, although $|o_s, o_t|$ may be in $[\widehat{l_{i,j}}, \widehat{u_{i,j}}]$ where $\widehat{l_{i,j}} > l_{i,j}$ and $\widehat{u_{i,j}} < u_{i,j}$, there may exist other objects matched with w_j in $O(o_s, l_{i,j})$, which invalids this pair, since v_i excludes v_j, and so we cannot increase $l_{i,j}$.

B. The Join Order for MSJ

With a careful study, we find that `MPJOrder` has two limitations: (1) among all the possible object pairs for two vertices, if only a very small proportion (e.g., 0.01%) of them could constitute e-matches, then we have to sample many pairs according to Lemma 2. (2) it may not be necessary to accurately estimate the number of e-matches for each edge, since the goal is to determine a topology order. Let us reconsider Example 1. Since the numbers of e-matches for the edges vary greatly, we may determine the order without estimating them accurately. To avoid these issues, we propose another simple yet effective and efficient method to determine the join order, denoted by `MSJOrder`.

`MSJOrder` relies on a key observation that, in an IR-tree (or other tree-based indexes), with a typical node capacity in the hundreds and a fill-factor of approximately 0.7, the leaf level makes up well beyond 99% of the index [19]. This implies that, the number of non-leaf nodes is much smaller than that of leaf nodes. Meanwhile, the non-leaf nodes, especially those in the lowest level, generally well summarize the objects' locations, which inspires the design of `PJ`. For example, given an edge (v_i, v_j), if the maximum and minimum distances between two nodes' MBRs are larger (smaller) than $u_{i,j}$ ($l_{i,j}$), then all the object pairs from them cannot be matched. Therefore, we propose to use the number of matched non-leaf node pairs to approximate the join order.

Specifically, we perform three steps in `MSJOrder`. First, for each edge, we follow `PJ` algorithm except the last step of handling leaf nodes, and find all the matched pairs of non-leaf nodes in the lowest level. Second, we count the number of matched pairs of non-leaf nodes for each edge. Third, we perform the same greedy algorithm as that of `MPJOrder`, where the estimated numbers of e-matches of edges are replaced by their numbers of matched non-leaf node pairs, and obtain a join order Γ. Note that, all the sets of matched pairs of non-leaf nodes are kept after running `MSJOrder`, as they will be reused later in the join process.

C. Two Pruning Criteria

We now introduce two interesting pruning criteria: *star-pruning* and *anchor-pruning*, which greatly speedup the query.

Star-pruning relies on a key observation that, if an object is in a match of P and matches with w_i (i.e., the keyword of

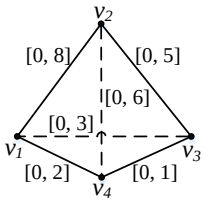
vertex v_i), then there are at least $|nb(v_i)|$ objects matched with v_i's neighbors respectively. In other words, if there does not exist $|nb(v_i)|$ objects respectively matched with v_i's neighbors, then we can safely prune this object. We formally state the star-pruning by Lemma 6.

Lemma 6 (Star-pruning)**.** *Let o_i be an object matched with w_i, and c_i be an integer variable initialized to be 0. For each neighbor v_j of v_i, if there is at least one e-match containing o_i for (v_i, v_j), then we increase c_i by 1. Finally, if c_i<$|nb(v_i)|$, we prune o_i.*

After obtaining the order Γ by `MSJOrder`, we compute the e-matches of all the edges, except those which are backward with mutual inclusion, as their distance intervals will be considered in the join process. By scanning all these e-matches only once, we can check whether each object can be pruned or not by Lemma 6.

Anchor-pruning is motivated it by Example 5.

Example 5. *Consider a pattern P with four edges in solid lines and two orders in Figure 10. From $\widehat{P}$, we know that the distance interval on ($\widehat{v_1}$, $\widehat{v_3}$) is [0, 3]. Suppose we follow order Γ_1, and let the sub-pattern formed only by the first two edges in Γ_1 be P'. By computing $\widehat{P'}$, we know that the distance between any two objects that match with w_1 and w_3 in a match of P' is in [0, 13]. After performing the join for the first two edges in Γ_1, if we get a partial match S=$\{o_1,o_2,o_3\}$, which matches with P', o_i matches w_i, $|o_1, o_2|$=7, and $|o_2, o_3|$=1, we can prune S directly and do not need to consider it when processing the last two edges in Γ_1, because by triangle inequality, we have $|o_1, o_3| \in [6, 8]$ is not in [0, 3].* □



Two orders:
Γ_1={(v_1, v_2), (v_2, v_3), (v_1, v_4), (v_4, v_3)}
v_3 is an anchor vertex and its anchor edge is (v_3, v_1)
v_4 is an anchor vertex and its anchor edge is (v_4, v_2)

Γ_2={(v_3, v_2), (v_3, v_4), (v_4, v_1), (v_2, v_1)}
No anchor vertex can be found

Fig. 10. Illustrating anchor vertices.

We call this pruning *anchor-pruning*. More formally, consider the subgraph formed by the first k edges of Γ be P'. Let v_i and v_j be two vertices in the k'- and k-th edges (k'<k). We call v_j an *anchor* vertex, if $[\widehat{l_{i,j}}, \widehat{u_{i,j}}] \subset [\widehat{l_{i,j}}', \widehat{u_{i,j}}']$, where $\widehat{l_{i,j}}'$ and $\widehat{u_{i,j}}'$ are the lower and upper bound distances between $\widehat{v_i}$ and $\widehat{v_j}$ in the bounded pattern of P'. Moreover, the edge (v_j, v_i), which may not be in P, is called v_j's *anchor edge*. Note that, a pattern may have multiple anchor vertices. Lemma 7 states that, the anchor vertices are in a small subgraph of P.

Lemma 7. *The anchor vertices are in the largest sub-pattern of P in which each vertex has at least two neighbors. The graph of the sub-pattern is also known as the 2-core [24] of the graph of P.*

Notice that this pruning highly relies on the join order. For example, if we use order Γ_2 in Figure 10, then we do not have such pruning. Given an order Γ, to find the anchor vertices, we first find the vertex set T in the 2-core of the graph of P. Then, we form a new pattern P', which is initialized as an empty pattern, incrementally by inserting edges of Γ and anchor edges. Once an edge is inserted, we compute the bounded pattern of P' and check whether the newly added vertex is an anchor vertex by verifying whether it is in T and comparing the distance intervals. In addition, if we find the newly added vertex is an anchor vertex, we insert its anchor edges and their distance intervals into P'. After inserting all the edges of Γ into P', we can find all the anchor vertices as well as their anchor edges.

D. The MSJ Algorithm

Algorithm 3: `MSJ`

```
   Input: root, P;
   Output: Ψ, all the matches;
1  compute the bounded pattern P̂ and refine P using P̂;
2  run MSJOrder and get Γ;
3  find a set Π of anchors vertices from the 2-core of P;
4  Ψ ← ∅, Φ1 ← ∅, Φ2 ← ∅, ···, Φm ← ∅;
5  for i ← 1 to m do
6      if e_i is forward or backward without mutual inclusion then
7          Φ_i ← run PJ for the edge e_i;
8  perform star-pruning for Φ1, Φ2, ···, Φm;
9  for k ← 1 to m do
10     let e_k=(v_i, v_j) be the k-th edge in Γ;
11     if e_k is a forward edge then
12         Ψ ← Ψ.link(Φ_k);
13         let v be latest considered vertex in e_k;
14         if v ∈ Π then perform anchor-pruning;
15     else
16         if v_i–v_j then prune some partial matches in Ψ;
17         else          prune some partial matches in Ψ by Φ_k;
18 return Ψ;
```

Based on the bounded pattern and two pruning criteria, we develop the `MSJ` algorithm. We first compute the bounded pattern $\widehat{P}$ of P using the dynamic programming and refine P. Then in the query process, we find the matched non-leaf node pairs for all the edges of P in a collective manner, through which the join order is computed. Finally, we follow the order and compute all the matches by linking these e-matches.

Algorithm 3 presents `MSJ`. The input of `MSJ` is an IR-tree and a pattern P, and the output is all the matches of P. We first compute the bounded pattern $\widehat{P}$ (see the pseudocodes in [18]) and refine P (line 1). Then, we perform `MSJOrder` to obtain the order Γ (line 2). Next, we find the anchors using the bounded pattern $\widehat{P}$ and the order Γ (line 3). For each edge of Γ, we find all the e-matches (lines 5-7), where Φ_1, Φ_2, $\cdots$, Φ_m denote the sets of e-matches for all the edges in Γ respectively. Note that Φ_i ($1 \le i \le m$) is an empty set if the i-th edge is backward with mutual inclusion, since its distance constraint will be considered during the join process. After that, we perform star-pruning (line 8). The join process (lines 9-17) is similar to that of `MPJ`, except that when the newly considered vertex is an anchor vertex, we perform the anchor-pruning (line 14). Finally, we return all the matches (line 18).

Name	Objects	Unique words	Total words
UK	182,317	45,371	550,663
NY	485,059	116,546	1,143,013
LA	724,952	161,489	1,833,486
TW	2,000,000	715,565	9,926,629

Fig. 11. Datasets used in our experiments.

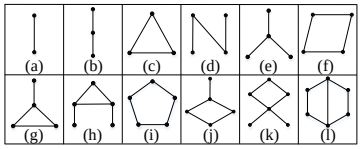


Fig. 12. structures of patterns.

Parameter	Range	Default
ϵ (MPJ)	0.15, 0.2, 0.25, 0.3, 0.35	0.25
δ (MPJ)	0.15, 0.2, 0.25, 0.3, 0.35	0.25
γ	0.2, 0.6, 1.0, 1.5, 2.0	1.0
η	60%, 70%, 80%, 90%, 100%	90%
χ	20%, 40%, 60%, 80%, 100%	100%

Fig. 13. Parameter settings.

Lemma 8. MSJ *completes in* $O(n^4+m|D|^2+|D|^n)$ *time.*

Since the patterns are often small, i.e., $n, m \ll |D|$, the time complexities of MPJ and MSJ are comparable. However, as shown later, although MPJ is intuitive and easy to implement, MSJ runs faster than MPJ experimentally, as it refines the pattern by the bounded pattern and uses two pruning criteria.

V. Experiments

A. Setup

Datasets. We use four real datasets. Figure 11 reports their numbers of objects, as well as the unique and total numbers of keywords. Dataset UK contains points of interest (e.g., banks and cinemas) in UK (www.pocketgpsworld.com). Datasets NY and LA are collected using Google Place API in New York and Los Angeles, respectively. In these datasets, each object has a set of keywords (e.g., "food"), and a pair of latitude and longitude values representing its location. Dataset TW is crawled from Twitter in US. Each geo-tweet is treated as a spatial object, its keywords are extracted from the tweet, and its location is a pair of latitude and longitude values.

Patterns. To create spatial patterns for the experiments, we first make 12 different undirected graphs (see Figure 12). These graphs vary in terms of number of nodes and edges. Four of them have been used in example patterns before, and the remaining eight graphs are illustrated by examples in the full version [18]. For each graph G in Figure 12, a spatial pattern for each dataset is generated by three steps:

Step-1: For each vertex $v \in G$, we add a keyword randomly selected following the distribution of keywords' frequencies (i.e., a keyword contained by more objects has a higher probability to be selected).

Step-2: For each vertex v_i with one of its neighbor v_j, we introduce a parameter η=90% such that the probabilities for the four different signs, i.e., $v_i \rightarrow v_j$, $v_j \leftarrow v_i$, $v_i \leftrightarrow v_j$, and v_i–v_j, are $\eta\times(1-\eta)$, $\eta\times(1-\eta)$, $(1-\eta)\times(1-\eta)$, and $\eta\times\eta$ respectively.

Step-3: For each edge (v_i, v_j), we attach a distance interval $[l_{i,j}, u_{i,j}]$ to it. If the edge is of sign v_i–v_j, $l_{i,j}$ is a random value in [0, $1km$] and the interval length, i.e., $u_{i,j}-l_{i,j}$, follows a Gaussian distribution with mean $1km$ and standard deviation $1km$; otherwise, $l_{i,j}$ is a random value in [0, $10km$] and the interval length follows a Gaussian distribution with mean $5km$ and standard deviation $5km$.

By following steps above, for each structure, we generate 20 patterns with each having at least one match in the dataset. Thus, there are 240 patterns for each dataset.

Queries. We use the IR-tree index [19], where the fanout B = 100, i.e., the maximum number of children of each node, the non-leaf nodes are kept in memory, and the leaf nodes are stored in disk. The inverted object list of each keyword, used by MPJOrder, is stored in a single file on disk. We consider five parameters: ϵ (MPJOrder), δ (MPJOrder), γ (length of distance intervals), η (percentages of signs), and χ (percentage of objects). The ranges of these parameters and their default values are shown in Figure 13. When varying a certain parameter, the values for all the other parameters are set to their default values. We implement our algorithms in Java, and run experiments on a machine having a quad-core Intel i7-3770 3.40GHz processor, 16GB of memory, and a 1TB of disk, with Ubuntu-12.04.1 installed.

B. Experimental Results

1) A Case Study: We consider the UK dataset, and two patterns. The first pattern is shown in the top-left panel of Figure 4 and it can be used to find houses that are close to stations, schools, and parks, but not too close to schools and stations (i.e., avoiding noise and crowd). The second pattern is depicted in Figure 15(a). It can be applied to finding houses which are close to churches, galleries, shops, hospitals, and stations, but not too close to hospitals and stations (i.e., avoiding infection and crowd). We run algorithm MSJ for SPM queries. Due to the space limitation, we only show one match for each pattern. For comparison, we use the mCK query [7], [3], whose input is the set of keywords in a pattern.

The results of SPM query and mCK query of the first pattern are depicted in Figure 4 and Figures 14 respectively. From Figure 4, we can observe that the four places in red balloons well match with the pattern, while the result of the mCK query is different, i.e., the distance from the house to the school is less than $0.4km$, which is not expected by the user. The reasons are that: (1) the mCK query does not consider the explicit distance requirements among the objects; and (2) it also does not take the exclusion-ship of edges (e.g., *house*→*school*) into consideration. Similarly, in Figure 15, the SPM query can find a set of objects exactly matched with the pattern in Figure 15(a). In contrast, the mCK query may find a house which is too close to the hospital and station (i.e., their distances are less than $1km$). Therefore, we conclude that the SPM query is more effective for finding spatial objects with various distance conditions.

2) Effectiveness of Estimation Method in MPJ: Recall that in MPJ, we estimate the number of e-matches for each edge with mutual inclusion using a sampling method. By Lemma 2, the estimation method theoretically guarantees that, the failure probability is at most δ if the multiplicative error is set as ϵ. In this experiment, we evaluate the effect of ϵ and δ on the actual error. Consider an edge in a pattern. Let r and $\widehat{r}$ be its

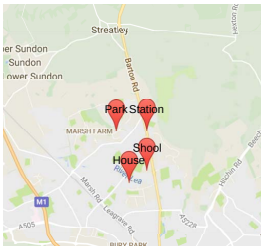
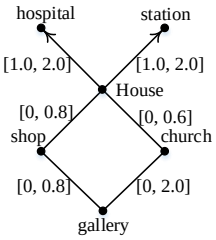
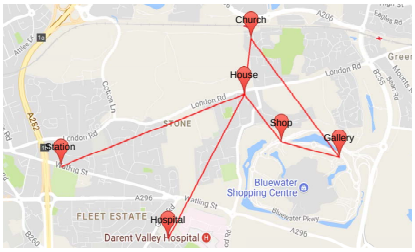


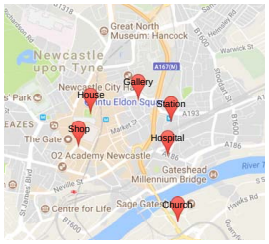
Fig. 14. mCK query result for pattern in Figure 4 (measure: km).



(a) Pattern



(b) A match of pattern in (a)

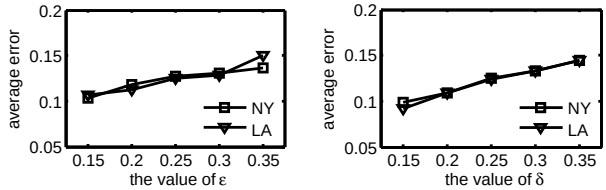


(c) mCK query result

Fig. 15. Case study results for the pattern in (a) (measure: km).

actual and estimated numbers of e-matches, respectively. The estimation error can be defined as: $error$=$\frac{|r-\widehat{r}|}{r}$.

For each dataset, we first collect all the edges, which are with mutual inclusion (i.e., the signs of edges are "–"), from all the patterns. Then, we vary the values of ϵ and δ from 0.15 to 0.35, and run the estimation method (θ=0.5). Finally, we compute the average error. Note that the actual number of e-matches is computed by the PJ algorithm.



(a) Effect of ϵ (b) Effect of δ

Fig. 16. Estimation method in MPJ.

We report the average estimation error on NY and LA datasets in Figure 16. As expected, the error increases when the values of ϵ and δ grow. However, the actual error is much lower than its corresponding theoretical error. For example, when the values of ϵ and δ are 0.25, the actual error is around 0.12. In our experiments, we set the values of ϵ and δ to 0.25.

3) Comparing Join Orders: As mentioned before, the main difference between MPJOrder and MSJOrder is that, when determining the orders, MPJOrder uses the numbers of e-matches for the edges, while MSJOrder considers the numbers of matched non-leaf nodes for the edges.

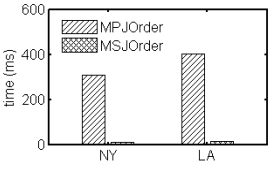


Fig. 17. Efficiency.

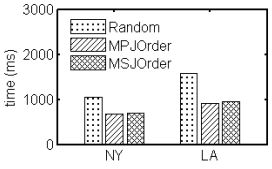


Fig. 18. Order quality.

We first compare the efficiency of running MPJOrder and MSJOrder. The results on NY and LA datasets are reported in Figure 17. We observe that MSJOrder is over an order of magnitude faster than MPJOrder. The main reason is that the number of matched non-leaf nodes is much smaller than that of e-matches, and thus MSJOrder performs very fast.

We further compare the quality of generated orders. Here, the "quality" of a particular order means its effect on the efficiency of the join process. For comparison, we also include a method determining the join order randomly, denoted by Random. To make a fair comparison, we adapt the MSJ algorithm such that, for each query, it can find the matches with any predefined order. We run the adapted MSJ with orders generated by the three methods on NY and LA datasets respectively, and report the running time of the join process in Figure 18 (the time of running these order methods is not included). Clearly, Random achieves the lowest order quality, which makes the join cost nearly twice larger than that of other methods. For MPJOrder and MSJOrder, the join process takes similar time cost on each dataset. This indicates that they generate join orders of similar quality. However, MSJOrder is much faster than MPJOrder, making it a better option.

4) Efficiency Results: We evaluate the efficiency of S-VF3, S-MDJ, MPJ, and MSJ. The results are reported in Figure 19. **Effect of pattern size.** For each dataset, we divide its patterns into five groups according to their vertex numbers. Figures 19(a)-19(d) report the efficiency for each group. Generally, with the increase of number of vertices in the patterns, the performance gaps among these algorithms become larger. The time cost of S-VF3 and S-MDJ does not always increase with the number of vertices on the last two datasets. This is because, when building the graph before running the GPM solvers, they need to enumerate more pairs and their numbers fluctuate greatly on different datasets.

MPJ and MSJ are consistently faster than baseline algorithms and MSJ is over an order of magnitude faster than baseline algorithms. This is because, when computing e-matches of edges of the pattern, MPJ and MSJ work in a joint manner, while S-VF3 and S-MDJ perform keyword search and range query separately. Meanwhile, MSJ is 2 to 5 times faster than MPJ. The reasons are three-fold. First, MSJ refines the patterns using their bounded patterns. Second, the pruning criteria of MSJ are very effective for pruning partial matches. Third, MSJOrder is more efficient than MPJOrder.

In addition, we ran MPJ and MSJ on a small sub-dataset ($|D|$=5,000) of the UK dataset, and found that they achieved similar efficiency. Thus, for small datasets, MPJ could be a practical alternative to MSJ, as it is easier to implement.

Effect of the distance interval length. For each edge (v_i, v_j) in the patterns, we vary the length (i.e., $|u_{i,j}$–$l_{i,j}|$) of the

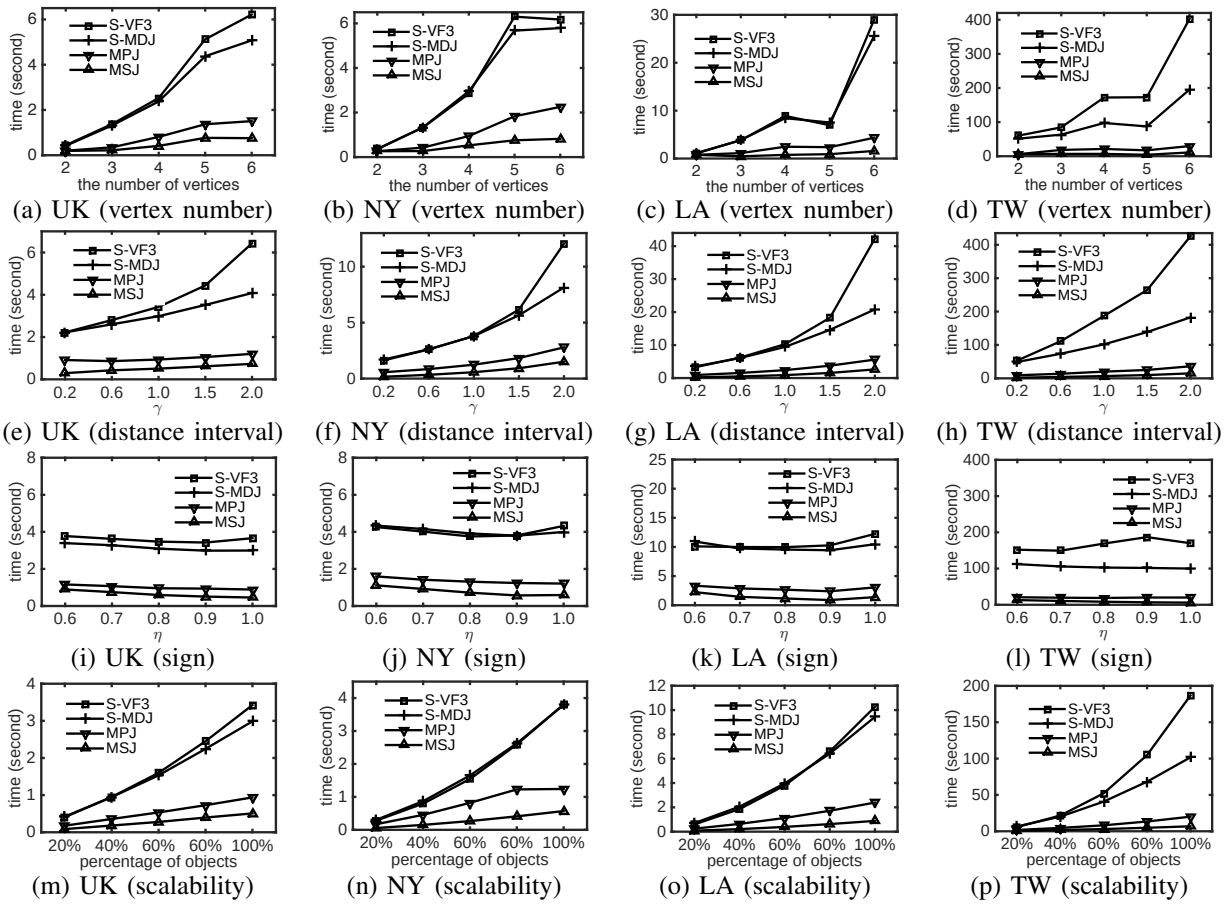


Fig. 19. Efficiency results of SPM queries.

distance interval using a parameter γ, such that the length of the distance interval increases γ times, where $\gamma \in \{0.2, 0.6, 1.0, 1.5, 2.0\}$. Specifically, we reset the upper bound distance $u_{i,j}$ as $l_{i,j}+(u_{i,j}-l_{i,j})\times\gamma$, and get five patterns, each of which corresponds to a value of γ. We report the average running time for each group in Figures 19(e)-19(h). Clearly, as the value of γ grows, the running time of each algorithm increases. This is because, a larger value of γ means a larger distance interval, which implies that more object sets are matched with the patterns and thus additional time is needed.

Effect of signs. Recall that in pattern generation, for each edge (v_i, v_j) we use a parameter η to control the percentages of edges with different signs. Now for the patterns of each dataset, we reset the signs of edges by varying η in $\{0.6, 0.7, 0.8, 0.9, 1.0\}$, and obtain five groups of patterns correspondingly. Note that the keywords and distance intervals remain unchanged. We report the average query time for each group in Figures 19(i)-19(l). We observe that, as the value of η increases, the running time of all the algorithms decreases slightly. This is because, when η becomes larger, more edges are with mutual inclusion. According to `MPJOrder` and `MSJOrder`, we can skip the join for more edges with mutual inclusion, and thus the query could be faster. However, edges with exclusion can be processed faster than edges with mutual inclusion, because fewer e-matches can be found. As a result, the overall running time does not change much.

Scalability. For each dataset, we vary the value of χ as shown in Figure 13, select a percentage of χ from its objects randomly, and obtain four sub-datasets. Figures 19(m)-19(p) report the scalability over these sub-datasets. As can be seen, both `MPJ` and `MSJ` scale near linearly with the size of dataset. Moreover, `MPJ` scales better than the baseline algorithms `S-VF3` and `S-MDJ`, and `MSJ` scales the best.

VI. Related work

Spatial keyword queries (SKQs). There are two kinds of SKQs in the literature. The first type (e.g., [25], [19], [26], [1]) takes as input the location where the query is issued, and a set of keywords. A list of k objects is returned, each of which is near to the query location, and is relevant to the keywords. Efficient indexes (e.g., *IR-tree* [25]) were proposed to enable fast query evaluation. In [19], [26], the top-k SKQ is studied. The authors in [27] proposed an SKQ, which continuously returns k objects when the query location moves. In [28], the solution is extended to for road networks.

The second type of queries takes as arguments a set of keywords and returns a group of objects [7], [3], [2], [8], [4] that are close to each other, and which together match the set of query keywords. Compared to the first type, this type of queries is more related to our SPM query. A representative query is the m-closest keyword (mCK) query [7], [3], which finds a group of objects that collectively contain all the m query keywords, and the maximum distance between any

two objects returned is minimized. However, as discussed in Section V, our SPM query captures users' requirements better than the mCK query. Its variants include [4] that minimizes a different distance cost function, and [8] that considers ratings of objects. The authors of [2] consider the distance between the query location and the returned group, in addition to the requirements that the returned group of objects cover query keywords and they are near to each other. A recent work [29] queries the POIs similar to a given keyword-based clue.

The SPM query is also related to the multi-way spatial join. Papadias et al. [12] express query constraints as graphs and retrieve n-tuples of objects satisfying the query graphs, by extending join with the R-tree index [30]. However, the objects that instantiate the vertices are not determined by keyword filters, but they are taken from the entire dataset(s). The join between two inputs, one of which is indexed by an R-tree, as well as multi-way joins that use this as a module, are studied in [31]. The optimization of these join queries was studied in [13]. However, none of these studies considers keywords and the exclusion-ship among objects, and their applicability and efficiency for solving SPM queries is questionable.

Graph pattern matching (GPM). Given a graph G and a pattern graph P, the GPM query [14], [15] extracts a set R of subgraphs of G, where for each $r \in R$, r matches with P. Zou et al. [14] study the GPM problem on undirected graphs. A recent work [15] proposes a fast GPM algorithm VF3 based on subgraph isomorphism. However, these solutions are mainly designed for graph databases, rather than spatial databases where objects are indexed by R-tree like structures. Moreover, the graph patterns often do not have distance requirement [15] or just have an upper bound distance [14] on each edge, while in spatial patterns, each edge has not only the minimum/maximum distance requirements, but also the inclusion/exclusion-ship. Thus, all these methods cannot be used to answer the SPM query directly. We adapt two GPM solutions [14], [15] to answer SPM queries as baselines, but they are not efficient as shown by our experiments.

VII. Conclusions

In this paper, we examine the spatial pattern matching (or SPM) problem. We first show that this problem is computationally intractable. Then we propose two efficient algorithms, namely `MPJ` and `MSJ`, for the SPM query. Our experimental results on real datasets show that our SPM queries are more effective than the state-of-the-art SKQs. Moreover, the `MSJ` algorithm is up to an order of magnitude faster than the baseline solutions which are adapted from GPM solutions.

In the future, we plan to increase to expansiveness power of the SPM query. For example, we will make the pattern to support more logical operations (e.g., "AND" and "OR"), supporting for instance the case where we want to find a house that has nearby a hospital or a doctor. Another change is to allow users to express directions, saying that a school has to be on the north of a house. It is also interesting to find sets of objects that are partially matched with the query pattern, if there is no match for the pattern in the database.

Acknowledgments

Reynold Cheng and Yixiang Fang were supported by the Research Grants Council of Hong Kong (RGC Projects HKU 17229116 and 17205115) and HKU (Projects 104004572, 102009508, 104004129). Gao Cong was supported by MOE Tier-2 grant MOE2016-T2-1-137 and MOE Tier-1 grant RG31/17. Nikos Mamoulis has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 657347.

References

[1] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: an experimental evaluation," *PVLDB*, pp. 217–228, 2013.
[2] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi, "Collective spatial keyword querying," in *SIGMOD*. ACM, 2011, pp. 373–384.
[3] T. Guo, X. Cao, and G. Cong, "Efficient algorithms for answering the m-closest keywords query," in *SIGMOD*. ACM, 2015, pp. 405–418.
[4] D. Choi, J. Pei, and X. Lin, "Finding the minimum spatial keyword cover," in *ICDE*. IEEE, 2016, pp. 685–696.
[5] Y. Fang et al, "Scalable algorithms for nearest-neighbor joins on big trajectory data," *TKDE*, vol. 28, no. 3, pp. 785–800, 2016.
[6] Y. Fang et al, "Effective community search over large spatial graphs," *PVLDB*, vol. 10, no. 6, pp. 709–720, 2017.
[7] D. Zhang et al., "Keyword search in spatial databases: towards searching by document," in *ICDE*. IEEE, 2009, pp. 688–699.
[8] K. Deng, X. Li, J. Lu, and X. Zhou, "Best keyword cover search." *TKDE*, vol. 27, no. 1, pp. 61–73, 2015.
[9] "Settlement patterns," http://geography.parkfieldprimary.com/the-united-kingdom/settlement-patterns, 2017.
[10] J. Schnaiberg, J. Riera, M. G. Turner, and P. R. Voss, "Explaining human settlement patterns in a recreational lake district: Vilas county, wisconsin, usa," *Environmental Management*, vol. 30, no. 1, pp. 24–34, 2002.
[11] S. Ministry of Education, https://www.moe.gov.sg/admissions/primary-one-registration/allocation, 2017.
[12] D. Papadias, N. Mamoulis, and B. Delis, "Algorithms for querying by spatial structure," in *VLDB*, 1998, pp. 546–557.
[13] N. Mamoulis and D. Papadias, "Multiway spatial joins," *TODS*, vol. 26, no. 4, pp. 424–475, 2001.
[14] L. Zou, L. Chen, and M. T. Özsu, "Distance-join: pattern match query in a large graph database," *PVLDB*, vol. 2, no. 1, pp. 886–897, 2009.
[15] V. Carletti et al., "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3," *TPAMI*, 2017.
[16] Y. Wu, J. M. Patel, and H. Jagadish, "Structural join order selection for xml query optimization," in *ICDE*. IEEE, 2003, pp. 443–454.
[17] Y. Fang et al, "SpaceKey: exploring patterns in spatial databases," in *ICDE*. IEEE, 2018.
[18] Y. Fang, R. Cheng, G. Cong, N. Mamoulis, Y. Li, "On spatial pattern matching," http://i.cs.hku.hk/~yxfang/spm2017.pdf.
[19] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen, "Joint top-k spatial keyword query processing," *TKDE*, 2012.
[20] D. Papadias, N. Mamoulis, and Y. Theodoridis, "Processing and optimization of multiway spatial joins using r-trees," in *PODS*, 1999.
[21] J. Jin, N. An, and A. Sivasubramaniam, "Analyzing range queries on spatial data," in *ICDE*. IEEE, 2000, pp. 525–534.
[22] https://en.wikipedia.org/wiki/Geometric_distribution.
[23] https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm.
[24] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.
[25] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *VLDB*, vol. 2, no. 1, pp. 337–348, 2009.
[26] C. Zhang, Y. Zhang, W. Zhang, and X. Lin, "Inverted linear quadtree: Efficient top-k spatial keyword search," *TKDE*, 2016.
[27] W. Huang, G. Li, K.-L. Tan, and J. Feng, "Efficient safe-region construction for moving top-k spatial keyword queries," in *CIKM*, 2012.
[28] C. Zhang, Y. Zhang, W. Zhang, X. Lin, M. A. Cheema, and X. Wang, "Diversified spatial keyword search on road networks." in *EDBT*, 2014.
[29] J. Liu, K. Deng, H. Sun, Y. Ge, X. Zhou, and C. S. Jensen, "Clue-based spatio-textual query," *PVLDB*, vol. 10, no. 5, pp. 529–540, 2017.
[30] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," *SIGMOD*, pp. 237–246, 1993.
[31] N. Mamoulis and D. Papadias, "Integration of spatial join algorithms for processing multiple inputs," *SIGMOD*, vol. 28, no. 2, pp. 1–12, 1999.