Noname manuscript No.
(will be inserted by the editor)

Evaluating Pattern Matching Queries for Spatial Databases

Yixiang Fang · Yun Li · Reynold Cheng · Nikos Mamoulis · Gao Cong

Received: Mar 10, 2018 / Accepted: date

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 whose 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. Moreover, we study efficient solutions to address two related problems of the SPM: (1) Find top-k matches that are close to a query location, and (2) Return partial matches for a query pattern. Experiments and case studies on real datasets show that our proposed solutions are highly effective and efficient.

Yixiang Fang
Department of Computer Science, The University of Hong Kong. E-mail: yxfang@cs.hku.hk
Yun Li
Department of Computer Science and Technology, Nanjing University. E-mail: liycser@gmail.com
Reynold Cheng
Department of Computer Science, The University of Hong Kong. E-mail: ckcheng@cs.hku.hk
Nikos Mamoulis
Department of Computer Science & Engineering, University of Ioannina. E-mail: nikos@cs.uoi.gr
Gao Cong
School of Computer Science and Engineering, Nanyang Technological University. E-mail: gaocong@ntu.edu.sg

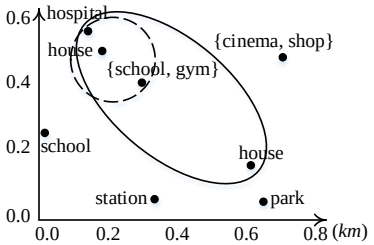


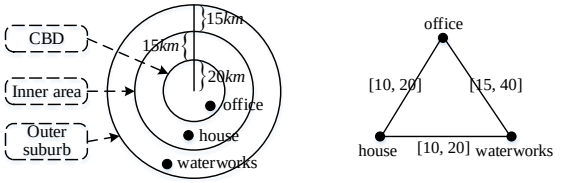
Fig. 1 An mCK query with Q={*house, school, hospital*}.

1 Introduction

Emerging location-based services (e.g., Google Maps) have raised plenty of research interest on the spatial-keyword query (SKQ) (e.g., [43,19,12,10]). 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 [43, 19], 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 Fig. 1, for example, D comprises spatial objects labeled with different keywords (e.g., *park* and *school*). Suppose that Q={*house, school, hospital*}, an answer to the mCK query is the set of objects circled by the dashed line in the figure.

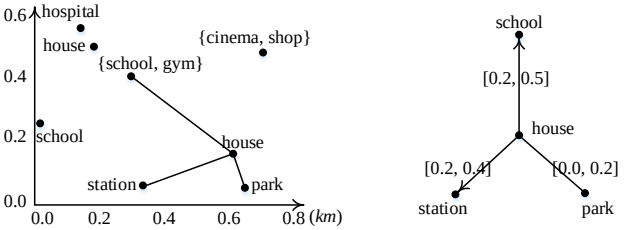
1.1 Motivation

Although SKQs are useful, they may not be able to precisely capture the user's intentions. Suppose that a user wishes to purchase a house, which is close to



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

Fig. 2 The human settlement and a spatial pattern [3].



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

Fig. 3 Illustrating the SPM query.

a school. Moreover, while the user does not want a hospital to be too close to her living space (e.g., for hygienic reasons), she wishes a hospital to be within a distance that makes it easily accessible. A hospital between $0.5km$ and $2km$ from the house would be desirable. This request may not be answered by an SKQ (e.g., [19]), which finds sets of objects which are all close to each other. In our example, the user would like to get as an answer the objects circled in the solid ellipse of Fig. 1.

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" [28,31]. This topic is interesting to urban planners and archaeologists. Fig. 2 illustrates a human settlement [3]. 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 site is built in the outer suburbs. Hence, the expert might want to retrieve objects for (*office*, *house*, *waterworks*), which are located in the CBD, the inner city, and the outer suburbs, respectively. 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 by existing SKQ formulations.

1.2 Proposal

To allow spatial relationships among keywords to be conveniently specified, we propose the *spatial pattern matching* (SPM) query. As shown in Fig. 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 distance (e.g., $0.5km$ or $1km$) of a school p, then he/she has a high chance to be admitted to p [26]. The user 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 Fig. 2(b), the spatial pattern for the human settlement example is shown.

As discussed before, an SKQ (e.g., [43,19,12,10]) can only return objects that are spatially close to each other. SPM queries are reminiscent to multi-way spatial joins studied in previous work [30,25]. 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) [45,7], 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-relationship are different from 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., Fig. 3(a)) into a graph, and then run a GPM algorithm on it. Moreover, as shown by our experiments in Section 7, the adapted GPM solutions (i.e., [45,7]) are very inefficient, calling for faster solutions.

1.3 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 dis-

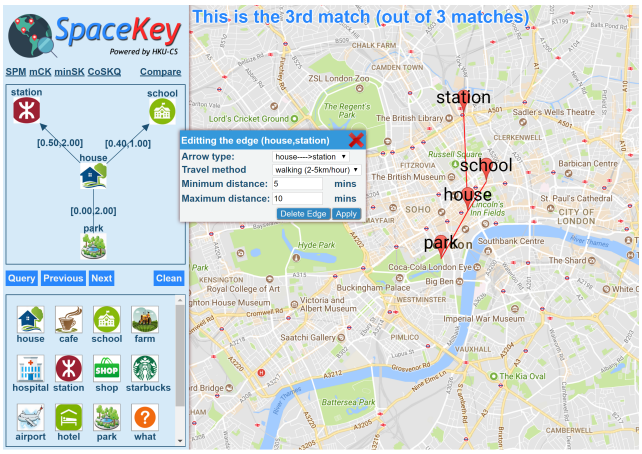


Fig. 4 The user interface of SpaceKey [16].

tances between two object types; and (2) *exclusion* and *inclusion*. Fig. 3(b) illustrates the exclusion relationship ($\rightarrow$), which expresses that (1) a school should be at least $0.2km$ from a house but not more than $0.5km$; and (2) no school should be in the vicinity of $0.2km$ 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 multi-way join approaches [38, 45] 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 the multi-way join paradigm directly, it is easy to implement. In addition, we 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.

We have implemented our solutions in a system, called *SpaceKey*. Its user interface is illustrated in Fig. 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 pop-up panel, which overlaps with the map in Fig. 4. [1] After clicking the "Query" button, the user can view the matches on the map one by one. The SpaceKey system also allows users to visually compare the results of different SKQs. For more details of SpaceKey, please refer to [16].

Our experience with SpaceKey is that sometimes the number of results returned by an SPM query is enormous. We say that an SPM query is *over-matched*, if it has a huge number (e.g., thousands) of results. A user may have difficulty to rank the numerous results and choose the best ones. To address this issue, we propose the top-k SPM problem. Given a pattern and an integer k, the top-k SPM aims at returning the top-k matches, whose values for a scoring function are the lowest. An example scoring function is one that measures the the average distance of a matched object from the location that a query is issued. To answer top-k SPM queries, we design fast algorithms, which are more efficient than brute-force approaches.

In SpaceKey, an SPM query may not return any result. This can be because the query pattern is rare, and consequently there are no instances that precisely match it. A user may either have to accept that no result is returned, or modify her/his query, with the hope that a result is yielded. We call these queries *under-matched*, and address the problem by proposing the partial PSM (or PSPM) query. Given a pattern P, the PSPM query returns objects that have a "close" match with P. More specifically, the PSPM query finds object sets that match the largest sub-graph of P. This increases the chance that a result is returned for an SPM query. A user can first run an SPM query. If no result is returned, SpaceKey will ask the user whether a less-precise result is acceptable, and if this is the case, an PSPM query will be executed. In this paper, we study how PSPM queries can be efficiently evaluated.

We experimentally evaluate our proposed SPM algorithms on real datasets. The results show that our best approach is over an order of magnitude faster than baseline techniques adapted from GPM. We conducted a case study that evaluates the practicality of SPM queries, showing that SPM queries often return better results for target applications than SKQ queries. We also test the proposed algorithms for answering the top-k SPM and PSPM queries, and show that they are faster than baseline approaches.

An earlier version of this paper is [13]; see also our SpaceKey demonstration [16]. Compared to [13, 16], the additional contributions of this paper are the introduction of the top-k SPM and PSPM queries and the proposal of efficient solutions for them. We also extensively evaluate the performance of these solutions by experiments on real datasets.

Organization. We formulate the SPM problem in Section 2. Sections 3 and 4 present our SPM solutions `MPJ` and `MSJ` respectively. We introduce the problem definitions and solutions for top-k SPM and PSPM in Sections 5 and 6 respectively. The experimental results are reported in Section 7. We review related work in Section 8 and conclude in Section 9.

[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.

2 The SPM Problem

2.1 Problem Definition

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 Fig. 3(a), for example, the object at (0.6, 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 1 summarizes the notations used in the paper.

Let us now define spatial patterns.

Definition 1 (spatial pattern[2]) A spatial pattern P is a graph $P(V, E)$ of n vertices $\{v_1, v_2, \cdots, v_n\}$ and m edges, such that 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.

For example, consider the edge *house*→*school* with distance interval $[0.2, 0.5]$ (km) in the pattern of Fig. 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 $0.2km$ and $0.5km$; and (3) there does not exist any object with keyword *school*, which is less than $0.2km$ from o_s. The arrow in *house*→*school* is expressed as *house excludes school*, and captures the user's intention of not getting any match for which the *house* has a *school* object less than $0.2km$ from it. This condition is useful to a user who wants to find a house not too close to any 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 Fig. 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 define 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 $doc(o_k)$ and $doc(o_l)$ include w_i and w_j, respectively, and the objects satisfy the distance constraints of (v_i, v_j).

Definition 3 (match) Given a spatial pattern $P(V, E)$ and a set S of objects, S is a match of P if there exists an injection $\phi : V \rightarrow S$, such that for all $v, v' \in V$, if $(v, v') \in E$, then the object pair $(\phi(v), \phi(v'))$ forms an e-match of (v, v').

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

In Fig. 3(a), for instance, the four objects connected in solid lines are a match of the pattern in Fig. 3(b) and they form an answer to 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 Fig. 3(a), any two or three linked objects are a partial match of the pattern in Fig. 3(b).

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

Proof. Please refer to our technical report [39]. □

A naive solution to solve the SPM problem takes up to $O(|D|^n)$ time, which is exponential to the number of vertices n. However, in practice n is not large, motivating us to develop efficient exact algorithms despite the intractability.

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

Table 1 Frequently used notations and their 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
$P(V, E)$	spatial pattern with vertex and edge sets V and E
n, m	number 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
ξ	maximum number of partial matches generated

2.2 Baseline Solutions: S-MDJ and S-VF3

We first propose basic SPM evaluation techniques by adapting existing GPM solutions [45,7]. Given an SPM query pattern P, we can follow three steps: first create a graph G using P, then convert pattern P to another pattern P' by removing its distance intervals and signs, and finally find all the matches of P' in G using a GPM solver. Specifically:

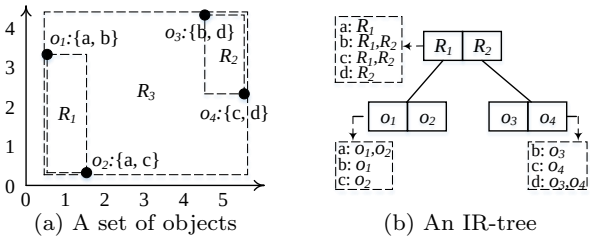
Step-1: For each edge (v_i, v_j) of P, we find a set O_i of objects that match with w_i. For each object $o \in O_i$ we perform two range queries in $O(o, l_{i,j})$ and $O(o, u_{i,j})$, to get their answers $L_{i,j}$ and $U_{i,j}$ which contain objects matched with w_j, respectively. Note that, if v_i *excludes* v_j (i.e., $v_i \rightarrow v_j$) and $L_{i,j} \neq \emptyset$, then we skip o (the cases where the sign of the edge is $v_i \leftarrow v_j$ or $v_i \leftrightarrow v_j$ are handled similarly). 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 e-matches of this edge.

Step-2: For the two objects in each e-match, we create two vertices with w_i and w_j in G and link them with an edge.

Step-3: We generate pattern P' by removing distance intervals and signs from P. Afterwards, any GPM algorithm can be applied to extract all matches of P' from G. Note that, for each edge of P, all its e-matches have been included into G, so all the matches of P could be extracted from G.

In this paper, we consider two GPM approaches MD-Join [45] and VF3 [7] and denote them by `S-MDJ` and `S-VF3`, respectively. Their time complexities could be up to $O(m|D|^2 + |D|^n)$, since there are at most $|D|^2$ e-matches for each edge, and the maximum number of matches is $|D^n|$.

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 [9] to index the objects in D. To build the IR-tree, we first build an R-tree and



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

Fig. 5 An example of IR-tree.

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. Fig. 5(a) gives an example of four objects $\{o_1, \cdots, o_4\}$, and the IR-tree built for these objects is depicted in Fig. 5(b). The inverted files of nodes are described in the dashed rectangle boxes.

3 The MPJ Algorithm

The main problem of the GPM-based solutions is that, to answer an SPM query, they need to generate a graph G and a pattern P', before running a GPM algorithm. 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 classic multi-way join [45], which is easy to implement.

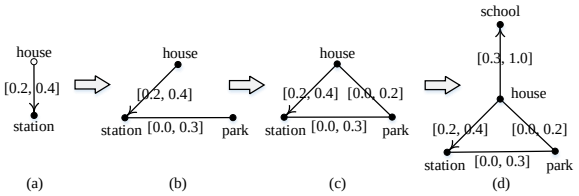


Fig. 6 Illustrating the process of `MPJ`.

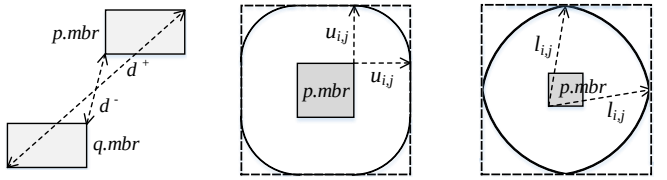
We first propose a join algorithm called pair-join (`PJ`) to find all the e-matches for each edge of P. Based on `PJ`, we develop the `MPJ` algorithm, which joins these e-matches of single edges, according to a particular order, to obtain all the matches of P. In Fig. 6, we show the query process of `MPJ` for the pattern in Fig. 6(d) with a particular join order. We first present `PJ` in Section 3.1, then discuss the join order and the `MPJ` algorithm in Sections 3.2 and 3.3 respectively.

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

3.1 The PJ Algorithm

We first consider edges with signs v_i–v_j and v_i→v_j. We will consider the other two signs later. To compute the e-matches of an edge, we assume that there is an IR-tree built for D. The rationale of adopting the IR-tree index is two-fold: (1) The IR-tree, as an R-tree extension, can easily handle edges with both inclusion-relationship and exclusion-relationship in the join process; (2) The IR-tree has been shown to be very efficient for joint spatial keyword queries [37].

Given an IR-tree and an edge with keywords w_i and w_j, PJ finds all the matched pairs of IR-tree nodes level by level in a top-down manner. Specifically, for the root level, the root node and itself form a matched pair (assume that the IR-tree has both keywords w_i and w_j). Then, we find the child node pairs that match and follow them, repeating the same process until all the matched objects at the leaf level are found.



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

Fig. 7 Illustrating the candidate pairs in PJ.

We now explain when a pair of nodes match. Let p and q be two non-leaf nodes, whose inverted files contain w_i and w_j respectively, at 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. Fig. 7(a) illustrates d^+ and d^-. We call (p, q) a *matched* pair of nodes, if $[d^-, d^+] \cap [l_{i,j}, u_{i,j}] \neq \emptyset$.

Intuitively, if p and q's MBRs are far from or too close to each other, then we cannot find any pair of objects under them that form an e-match. We illustrate this using Fig. 7(b) and 7(c). If q's MBR does not intersect the outer area bounded by the solid line in Fig. 7(b), then q does not match with p, since the minimum distance of their MBRs must be larger than $u_{i,j}$; Similarly, if q's MBR is fully covered by the outer area bounded by the solid line in Fig. 7(c), then q does not match with p, since the maximum distance of their MBRs must be less than $l_{i,j}$.

To prune unmatched node pairs, we exploit a key property of the IR-tree. That is, for each node in the IR-tree, its MBR must be contained by the MBR of its parent node. As a result, after finding all matched node pairs at a specific tree level, for the next (lower) level, we can directly find the matched node pairs from their child node pairs, and ignore all the other node pairs. By repeating this process level by level, we can safely prune a large number of unmatched pairs of nodes and obtain all the e-matches. Algorithm 1 presents PJ.

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' ← ∅;//a map of matched pairs in next level;
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;
```

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. The output of PJ is Φ, a map of all the e-matches. It maintains a map Φ for keeping track of all the matched pairs at a specific level. Φ contains key-value pairs, where the key is a node/object and the value is the set of its candidates (line 1). We assume that Φ is associated with two methods "keySet" and "getKey", where "keySet" returns the key set of Φ, and "getKey" returns the value set for a key in Φ. PJ initializes a matched pair for the *root* node (line 2). Next, it finds candidate pairs level by level (lines 3-19).

At each iteration, PJ enumerates all the candidate pairs in Φ (lines 5,8). 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 is updated for keeping matched pairs at the next level (line 19). Finally, PJ returns Φ (line 20).

We now consider edges with other signs. $v_i \leftarrow v_j$ is handled as $v_j \rightarrow v_i$ and PJ is directly applied. For $v_i \leftrightarrow v_j$, we 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.

3.2 The Join Order for MPJ

The order of performing joins for the edges has a significant effect on efficiency [38,45]. We illustrate this by Example 1.

Example 1 Consider a pattern of 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 only have to scan the tuples and check for each of them 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, which gives us up to 50,000 tuples as candidates for v_2–v_3–v_1. Next, for v_1–v_2, we check whether each of these tuples satisfies the distance constraint.

Clearly, *Order1* has lower computational cost than *Order2*. The reason is that for edges with mutual inclusion (e.g., v_3–v_1 in *Order1*), we may avoid applying PJ, because we can scan the linked tuples and check their distance constraints. However, for edges with other signs, we cannot avoid PJ. For example, if we replace v_3–v_1 by $v_1 \rightarrow v_3$ in Example 1 and use *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 can we quickly estimate the number of e-matches for such edges without running PJ? Some existing cost models are based on R-trees [29] 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 Fig. 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. Lemma 2 states that, by sampling a certain number of matched pairs, we can accurately estimate the number r of e-matches.

Lemma 2 (Estimation) *Let p ($p>0$) be 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 $0<\epsilon<1$,*

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

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

Proof. To prove the lemma, we need to prove that (1) $\Pr\left(Y \leq (1-\epsilon)E[Y]\right) \leq \delta$; and (2) $\Pr\left(Y \geq (1+\epsilon)\,E[Y]\right) \leq \delta$. The proof of (1) is exactly the same with the proof of Lemma 5.1 of [8]. We can prove (2) in a similar manner. We skip the details due to space limitations. □

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 $\frac{1}{p}$ [1]. 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 $|O_i| \cdot |O_j|$ pairs and r matched pairs, we have $p=\frac{r}{|O_i| \cdot |O_j|}$. Thus, we conclude $E[Y]=\frac{s}{r} \cdot |O_i| \cdot |O_j|$. 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 $s=O(\frac{8}{\epsilon^2} \ln \frac{1}{\delta})$, to estimate $E[Y]$ well, which further implies $r \approx \frac{s}{E[Y]} \cdot |O_i| \cdot |O_j|$.

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 $\zeta \in [0,1]$, and stop sampling if we cannot see s e-matches after sampling $|O_i| \cdot |O_j| \cdot \zeta$ 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 (DP) [38]. However, this method is

inefficient because it has to perform PJ for all edges before finding the best order and the search space can also be very large [38,45]. 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, ζ;
   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 Continuing Example 1, 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. □

3.3 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 Fig. 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 [39].

During the join process, assume that the m sub-patterns are $P_1, \cdots, P_m$, where P_1 contains a single edge and P_m is P, and the numbers of partial matches generated for these sub-patterns are $\xi_1, \cdots, \xi_m$, respectively. Then, let us denote the maximum number of partial matches by ξ=max$\{\xi_1, \cdots, \xi_m\}$. Clearly, we have $|\Psi| \leq \xi \leq |D|^n$. We show the time complexity of MPJ in Lemma 3.

Lemma 3 MPJ *completes in* $O(m\zeta|D|^2+\xi)$ *time.*

Proof. The PJ algorithm takes $O(|D|^2)$ time, as there are at most $|D|^2$ e-matches. Since we sample at most $\zeta \cdot |D|^2$ pairs for each edge, MPJOrder costs $O(m\zeta|D|^2)$. Thus, MPJ completes in $O(m\zeta|D|^2+\xi)$ time. □

4 The MSJ algorithm

In this section, we propose a new algorithm called the multi-star-join (or MSJ). MSJ uses the concept of *bounded pattern*, which is a refined form of the query pattern that can be computed by dynamic programming. The bounded pattern can help in pruning partial matches during the join process. In addition, compared to MPJ, MSJ determines the join order in a more efficient way, which does not rely on sampling. Finally, during the join process MSJ considers the edges in a collective manner with two pruning criteria. With the help of these features, MSJ is much more efficient than MPJ, as we will demonstrate experimentally.

4.1 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 show this by Example 3.

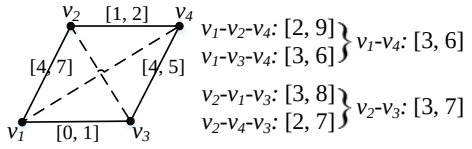


Fig. 8 Illustrating the bounded pattern.

Example 3 Consider a pattern P in Fig. 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 clique graph 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 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*

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

Proof. Since the upper bound distance from $\widehat{v_i}$ to $\widehat{v_j}$ must not conflict with the upper bound distance of each path from $\widehat{v_i}$ to $\widehat{v_j}$, $\widehat{u_{i,j}}$ is the minimum one among all the paths. Also, it is easy to see that $\widehat{u_{i,j}}$ can be computed recursively. Therefore, Eq (2) holds. □

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 [2].

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

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

Proof. Let o_i, o_j, and o_k be three objects, where o_i and o_k constitute an e-match of $\widehat{v_i}$–$\widehat{v_k}$, and o_k and o_j constitute an e-match of $\widehat{v_k}$–$\widehat{v_j}$. This implies that we have $|o_i, o_k| \in [\widehat{l_{i,k}}, \widehat{u_{i,k}}]$ and $|o_k, o_j| \in [\widehat{l_{k,j}}, \widehat{u_{k,j}}]$. We prove the lemma by considering the three items in the rightmost part one by one.

If $[\widehat{l_{i,k}}, \widehat{u_{i,k}}] \cap [\widehat{l_{k,j}}, \widehat{u_{k,j}}] \neq \emptyset$, then it is possible that $|o_i, o_k| = |o_k, o_j|$. By triangle inequality, we have $|o_i, o_j| \ge |o_i, o_k| - |o_k, o_j| = 0$, which implies that $\widehat{l_{i,j}} = 0$. Similarly, if $\widehat{u_{i,k}} < \widehat{l_{k,j}}$, by triangle inequality, we have $|o_i, o_j| \ge |o_k, o_j| - |o_i, o_k| \ge \widehat{l_{k,j}} - \widehat{u_{i,k}}$, which implies that $\widehat{l_{i,j}} \ge \widehat{l_{k,j}} - \widehat{u_{i,k}}$. The case of $\widehat{l_{i,k}} > \widehat{u_{k,j}}$ can be illustrated in a similar way. By enumerating all the other vertices $k \in [1, n]$, we get a tighter lower bound $\widehat{l_{i,j}}$. □

Continuing Example 3, let i=1 and 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 that we can use them to refine P, which may reduce the query evaluation 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 $[\widehat{l_{i,j}}, \widehat{u_{i,j}}] \subset [l_{i,j}, u_{i,j}]$, we delete (v_i, v_j), as any set of objects matched with $P \backslash e$ 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.

Example 4 Consider the pattern P in Fig. 9, and three different cases for edge $e = (v_2, v_3)$. 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 P 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 the edge to [2, 4] by criterion ❸. □

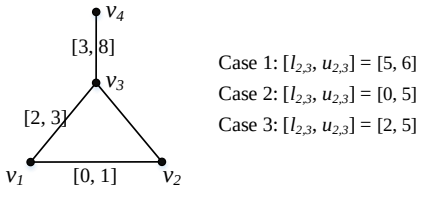


Fig. 9 Illustrating pattern refining.

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 invalidates this pair, since v_i excludes v_j; so, we cannot increase $l_{i,j}$.

4.2 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 topological 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 [37]. This implies that, the number of non-leaf nodes is much smaller than that of leaf nodes. Meanwhile, the non-leaf nodes, especially those at 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. Thus, 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 apply the `PJ` algorithm until the handling of leaf nodes, to find all the matched pairs of non-leaf nodes at 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 the corresponding 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.

4.3 Two Pruning Criteria

We now introduce two pruning criteria: *star-pruning* and *anchor-pruning*.

Star-pruning relies on the key observation that, if an object o_i 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 matching with v_i's neighbors. In other words, if there do not exist $|nb(v_i)|$ neighbors of o_i that match with v_i's neighbors, then we can safely 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. Then, we scan all the e-matches to potentially prune objects, as follows. Let o_i be an object matched with w_i; we initialize a counter c_i for o_i to 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. Then, we use the following lemma to potentially prune o_i.

Lemma 6 (Star-pruning) *If an object o_i that matches with vertex $v_i \in P$ satisfies $c_i<|nb(v_i)|$, then o_i can be pruned.*

Proof. The lemma directly follows the observation. □

Hence, by scanning all e-matches once, we can complete the star-pruning.

Anchor-pruning is motivated it by Example 5.

Example 5 Consider a pattern P with four edges in solid lines and two orders in Fig. 10. From $\widehat{P}$, we know that the distance interval on $(\widehat{v_1}, \widehat{v_3})$ is [0, 3]. Assume that 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, since by triangle inequality, $|o_1, o_3| \in [6, 8]$ is not in [0, 3]. □

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'-th 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

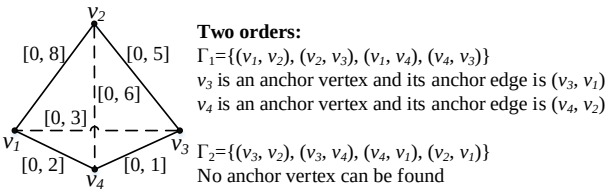
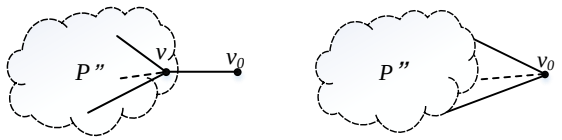


Fig. 10 Illustrating anchor vertices.

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*. 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 [4] of the graph of P.*



(a) v_0's degree is one (b) v_0's degree is at least two

Fig. 11 The degree of anchor vertex.

Proof. Let G be the graph of P by removing its keywords, distance intervals, and arrows on the edges. Suppose v_0 is an anchor vertex. To find the anchor vertices, recall that we form a pattern P' incrementally by inserting edges of Γ and anchor edges. Given an order Γ, we denote the pattern formed by edges appearing before edges containing v_0 by P''. Meanwhile, we have computed its bounded pattern $\widehat{P''}$. Notice that in $\widehat{P''}$, the upper and lower bounds of the distance between each pair of vertices are *globally* tight.

Let us now consider a new edge e=(v_0, v) of Γ, where v is a neighbor of v_0 and is in P''. After inserting e, the distance from v_0 to any vertex v_k in P'' can be bounded by triangle inequality using Lemmas 4 and 5. If v_0 only has one edge linked with P'' as shown in Fig. 11(a), we have $[\widehat{l_{k,0}}, \widehat{u_{k,0}}]$=$[\widehat{l_{k,0}}', \widehat{u_{k,0}}']$, and then v_0 cannot be an anchor vertex. Therefore, v_0 must have a degree of two or more. Since $[\widehat{l_{k,0}}, \widehat{u_{k,0}}] \subset [\widehat{l_{k,0}}', \widehat{u_{k,0}}']$, there should be at least two paths of edges from v_k to v_0 in a pattern formed by P'' and edges linked with v_0. In other words, v_0 is in a circle, in which each vertex has a degree of two or more. Hence, the anchor vertex v_0 is in the 2-core of the graph G. □

Note that pruning highly relies on the join order. For example, order Γ_2 in Fig. 10 does not result in 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 that 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.

4.4 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 partial matches in Ψ;
17         else  prune some partial matches in Ψ by Φ_k;
18 return Ψ;
```

Based on the bounded pattern computation and the two pruning criteria, we develop the `MSJ` algorithm. We first compute the bounded pattern $\widehat{P}$ of P using 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 $\widehat{P}$ 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 $\Phi_1, \Phi_2, \cdots, \Phi_m$ denote the sets of e-matches for all the edges in Γ respectively. Note that Φ_i $(1 \leq i \leq m)$ 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).

Since the patterns are typically 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.

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

Proof. Let us analyze the cost of each step in `MSJ`. First, computing the bounded pattern using Lemmas 4 and 5 (line 1) takes $O(n^3)$ time, since all the triples of vertices are enumerated. Second, `MSJOrder` finds all the non-leaf matched pairs for each edge in P (line 2), so its time cost is $O\left(m(|D|/B)^2 + m\right)$, where $B \geq 2$ is the fanout of the IR-tree. Third, invoking `PJ` for each edge (lines 5-7) takes $O(m|D|^2)$ time, and the star-pruning (line 8) also takes $O(m|D|^2)$ since we just scan all the e-matches. Fourth, computing the anchors incrementally costs $O(n^4)$, and the maximum number of partial matches generated during the join process (lines 9-17) is $O(\xi)$, where $\xi \geq |\Psi|$. Thus, `MSJ` takes $O(n^4 + m|D|^2 + \xi)$ time. Although the complexity is high, `MSJ` runs very fast practically, since P is typically small ($m, n \ll |D|$), and the pruning criteria are very effective. □

5 The Top-k SPM Problem and Algorithms

In this section, we introduce the top-k SPM problem and propose an efficient algorithm that solves it. Recall that the objective is to handle *over-matched* SPM queries, which return a very large number (e.g., thousands) of results, by ranking their matches using a scoring function and presenting to the user only the top-k matches.

5.1 Problem Definition

Top-k SPM requires a scoring function to rank the matches of the query pattern. To define the scoring function, various factors, such as POI rating and quality, as well as the query user's location, could be considered. In this paper, we use the user location and its distances to the matches to rank these matches. We will study other functions in the future.

Now we formally introduce the scoring function.

Definition 4 (scoring function) Given a set S of spatial objects and a location loc, the score of S w.r.t. loc is defined as

$$f(S, loc) = \max_{o \in S} |o, loc|. \tag{4}$$

Intuitively, if a user is at location loc, she would find S attractive if $f(S, loc)$ is small, which means that the user is geographically close to all objects in S. Based on the definition of $f(S, loc)$, we now define top-k SPM queries as follows.

Problem 2 (top-k SPM) Given a database D and a spatial pattern P, a query location loc, and a positive integer k, the top-k SPM returns a list of k matches, whose scores w.r.t. loc are the smallest.

Remark. The function max in $f(S, loc)$ can be replaced by other aggregate functions (e.g., *min*, *sum* and *avg*) and our proposed algorithm can be easily adapted to handle these cases.

5.2 An Efficient Algorithm

A basic method to answer the top-k SPM query is to find all the matches using one of previous SPM algorithms (e.g., `MSJ`), then rank these matches based on their values of the scoring function, and finally return the top-k matches. The major drawback of this method is that, when the number of matches is much larger than k, many matches which are not in the top-k list are also computed, and this may incur many unnecessary computations. To improve the efficiency, we propose an *incremental matching* algorithm (`IncMatch`).

The fact that a top-k SPM query aims at finding matches that are close to the query location loc inspires us to approach the problem by applying searches incrementally in increasing distance from loc. In specific, we first choose a vertex v_s from P as the starting vertex. Then, we consider objects, which match with v_s's keyword w_s in increasing distance from loc, during which we check whether they are part of P's matches. Note that the score of the k-th best match found so far gives us a bound which can be used for termination. This process is executed iteratively, until we have checked all the objects matched with w_s or the next object is

guaranteed not to produce a better match than the current k-th best. Note that an alternative of examining objects matched with v_s is to use incremental nearest neighbor (INN) search; however, an efficient INN algorithm would need a spatial index for the objects of v_s, which is typically not available (for any arbitrary keyword w_s).

Algorithm 4: `IncMatch`

```
   Input: k, P, loc
   Output: top-k matches;
1  compute the bounded pattern P̂;
2  Let the starting vertex v_s be the vertex of P, whose
     keyword frequency in D is the smallest;
3  μ ← 0, θ ← 10km;
4  M ← ∅, Q ← ∅, inside ← 0, outside ← γ,
     flag ← true;
5  while flag do
6      while Q=∅ do
7          if (inside>μ and M.size() ≥ k) or the
             annular area does not overlap the area that
             D covers then
8              flag ← false, break;
9          Q ← {o ∈ D|w_s ∈ doc(o) ∧ |o, loc| ∈
           [inside, outside]};
10         inside ← outside, outside ← outside + γ;
11     if flag=false then break;
12     while Q ≠ ∅ do
13         o ← Q.pop();
14         if f(o, loc)>μ and M.size() ≥ k then
15             flag ← false, break;
16         R ← FindMatches(loc, P, P̂, o);
17         M.add(R) and update μ;
18 return the top-k matches in M;
```

Algorithm 4 presents the steps of `IncMatch`. First, we compute the bounded pattern $\widehat{P}$ (line 1). Then, we set as the starting vertex v_s the one whose word frequency is the smallest in P (line 2). Next, we initialize variables μ, M, Q, *inside*, *outside*, *flag*, and θ, where μ is the upper bound of the score of the top-k matches, M is a priority queue of matches ranked by their score values in ascending order, Q is another priority queue of objects ranked by their distances to *loc* in an ascending order, *inside* and *outside* are the radiuses that bound the annular area centered at *loc*, *flag* indicates whether the program continues, and θ is a constant value.

In the outer while loop (lines 5-17), we first find objects labeled with w_s in the annular area and keep them into Q (lines 6-10). Note that each time we increase the radius of annular area by θ. Then, we pop the next object of Q and run `FindMatches` (Algorithm 5) to find matches of P around the object, adding its matches to M, and updating μ (lines 12-17). The while loop stops (lines 7-8, 12-15) once *inside* or the distances of all objects in Q are larger than μ.

Algorithm 5: `FindMatches`

```
   Input: loc, P, P̂, o
   Output: matches around object o;
1  map ← ∅;
2  for vertex v_j ∈ P other than v_s do
3      if v_s → v_j then
4          L ← {o ∈ D|w_j ∈ doc(o) ∧ |o, loc| ∈ [0, l̂_{s,j}]};
5          if L ≠ ∅ then return ∅;
6      L ← {o ∈ D|w_j ∈ doc(o) ∧ |o, loc| ∈ [l̂_{s,j}, û_{s,j}]};
7      if v_s ← v_j then
8          L ← find objects using line 4;
9          prune objects in L that do not satisfy the
             exclusion-relationship;
10     if L = ∅ then return ∅;
11     else map.put(j, L);
12 for edge (v_i, v_j) in P that not linked to v_s do
13     get e-matches of (v_i, v_j) using map.get(i),
         map.get(j);
14 link e-matches and objects of map to form matches;
15 return matches of P;
```

To find matches of P around a particular object o matched with v_s, a simple method is to run `MSJ` for the sub-pattern $P \backslash v_s$ (subgraph of P without v_s and edges linked to it), and then link the returned matches with o. However, since there may exist many objects matched with v_s, many matches of $P \backslash v_s$ will be computed repeatedly by `MSJ`. To tackle this issue, we develop the method `FindMatches`, as shown in Algorithm 5.

`FindMatches` first initializes a map for keeping vertices and their matched objects (line 1). Then, it finds objects that are matched with vertices (except v_s) in P, whose distances to o satisfy the distance constraints according to $\widehat{P}$ (lines 2-11). For each vertex v_j in P, if the edge is $v_s \rightarrow v_j$, when there are objects with keyword w_j that have distances less than $l_{s,j}$ from $o.loc$, `FindMatches` stops and $\emptyset$ is returned (lines 3-5). If the edge is $v_s \leftarrow v_j$, objects that do not satisfy the exclusion-ship restrictions are pruned (lines 7-9). As soon as there are no matched objects, `FindMatches` stops and $\emptyset$ is returned (line 10). Finally, for edges that are not linked to v_s, we check all possible combinations of the corresponding objects to get e-matches, and then link these e-matches and objects in *map* to obtain matches. Example 6 illustrates the functionality of `IncMatch`.

Example 6 Consider the pattern P and the set D of spatial objects shown in Fig. 12. Let k=2 and *loc* be the query location. First, v_1 is selected as the starting vertex and θ=0.5km. Then, `IncMatch` finds the inside

circle region to find objects labeled with a. Next, it considers o_1 and invokes `FindMatches`, which finds o_1, o_2, and o_3; these three objects satisfy all the constraints of P and thus they constitute a match. Afterwards, `IncMatch` considers o_4 as the next match of v_1. Since no objects matching v_2 and v_3 can be found, `IncMatch` increases the radius of the annular area to find more matches. Then, it considers o_{12} and finds that o_{12}, o_{13}, and o_{14} form another match of P. μ is updated as the distance between loc and o_{13}. o_8 is ignored since o_{10} is too close to o_9. Finally, `IncMatch` stops as the newly expanded *inside* is larger than μ.

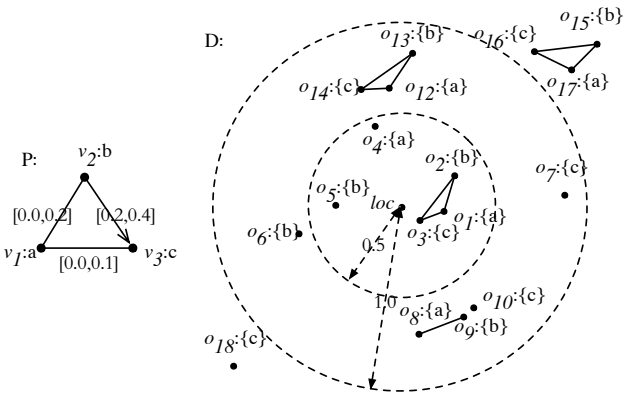


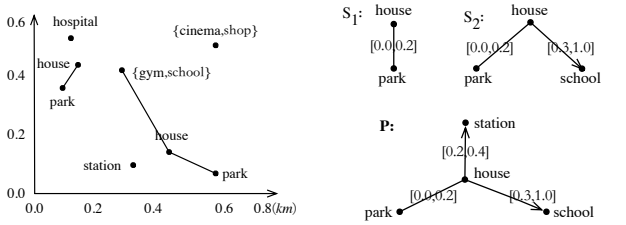
Fig. 12 Illustrating how `IncMatch` works.

6 The PSPM Problem and Algorithms

In this section, we introduce the *partial SPM* (PSPM) problem and suggest methods for solving it. As discussed before, PSPM can be used to handle *under-matched* SPM queries, which return no results for the query patterns. In practice, for an SPM query with pattern P, if the number of matches is zero, then the system may automatically run a PSPM query for P.

6.1 Problem Definition

Inspired by existing work on approximate algorithms of GPM [42,44], which find subgraphs that match with sub-patterns of the graph pattern, in this paper we propose to find sets of objects that match with as many edges in the spatial pattern P as possible. That is, we aim at finding all the object sets that match with the *maximal* sub-patterns of P, which have at least one exact match in the database. Note that it does not need the query user to specify additional constraints. Also, it can provide users helpful hints for changing the pattern.



(a) Spatial object set D (b) Spatial patterns

Fig. 13 Illustrating the PSPM query.

Before introducing PSPM in detail, we first give some notations. We call any connected subgraph, S, of P a *sub-pattern* of P, and denote their relationship by $S \subseteq P$. We call a pattern *feasible* if it has at least one match in the spatial database; otherwise, *infeasible*.

Definition 5 (maximal feasible sub-pattern) A sub-pattern S is a maximal feasible sub-pattern of P, if S is feasible and there does not exist another feasible sub-pattern S' of P such that $S \subset S'$.

For example, let P in Fig. 13(b) be the query pattern. P is infeasible w.r.t. database D of Fig. 13(a). In Fig. 13(b), we also show two sub-patterns S_1 and S_2 of P. Both of them are feasible sub-patterns since they match with the two sets of objects linked by solid lines in Fig. 13(a) respectively. In addition, S_2 is a maximal feasible sub-pattern. Based on Definition 5, we formally define the PSPM problem as follows.

Problem 3 (Partial Spatial Pattern Matching) Given a spatial pattern P, if P is infeasible, PSPM returns all the maximal feasible sub-patterns of P and their matches from the database D.

In Fig. 13(a), the objects with keywords *house*, {*gym, school*} and *park* linked in solid lines form a match of S_2 in Fig. 13(b), and they form an answer of this PSPM query. Note that if the maximal feasible sub-patterns are too many and/or they have too many matches, the top-k solution presented in Section 5 can be adapted for ranking these sub-patterns and matches in order to present to the user a concise answer set.

6.2 The Basic Method

A straightforward method to answer the PSPM query is to enumerate all sub-patterns of P, then compute all the matches of each sub-pattern, and finally return all the maximal feasible sub-patterns and their matches. The major limitation of this method is that the number of sub-patterns is exponentially large, and the huge

computational overhead may render the method impractical. To alleviate this issue, we take advantage of an *anti-monotonicity* property.

Lemma 9 (Anti-monotonicity) *Given a database D and a pattern P, if P is feasible, then any sub-pattern of P is also feasible.*

Proof. The lemma directly follows the observation. □

Lemma 9 allows us to stop checking all the super patterns of a sub-pattern P', once we have verified that P' is infeasible, which efficiently reduce the search space. Based on the anti-monotonicity property, we develop a baseline method, called `Basic`. It begins by examining the sub-patterns consisting of a single edge. Then, it repeatedly executes the following two steps to retrieve feasible sub-patterns with more edges until no feasible sub-patterns are found.

- **Candidate Generation.** For any two feasible sub-patterns which only differ in one edge, unify them to a candidate sub-pattern, if the resulting graph is connected.
- **Verification.** For each candidate sub-pattern P', mark P' as a feasible pattern if P' has matches.

Algorithm 6: `Basic`

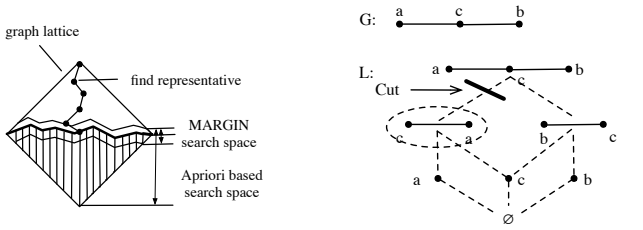
```
   Input: root, P;
   Output: all the maximal feasible sub-patterns
1  initialize l = 0, T using P;
2  initialize Φ_i (i = 1 to m);
3  for i ← 1 to m do Φ_i ← run PJ for the edge e_i;
4  while T ≠ ∅ do
5      l ← l + 1, Λ_l ← ∅;
6      for each P' ∈ T do
7          get e-matches of edges in P';
8          perform star-pruning for the e-matches;
9          join the edges-matches, and get Ψ;
10         if Ψ ≠ ∅ then Λ_l.put(P', Ψ);
11     if Λ_l ≠ ∅ then
12         generate candidate patterns using Λ_l and
             update T;
13     else
14         break;
15 return the sub-patterns and their matches in Λ_{l-1};
```

We present `Basic` in Algorithm 6. The input is an IR-tree and a query pattern P. We first initialize some variables (lines 1-2), where Φ_i denotes the set of e-matches for the i-th edge of P, l=0 indicating the number of edges, and a set T of sub-patterns with each being an edge of P. Then, for each edge of P, we find all the e-matches (line 3). In the while loop (lines 4-15), for each sub-pattern P', we first get the e-matches of edges in P' from Φ_1, Φ_2, $\cdots$, Φ_m (line 7). After that, we perform star-pruning and verify whether P' is feasible (lines 8-9). To do this, we adapt `MSJ` such that it stops when the first match of P' is found. If Ψ contains a match, i.e., P' is feasible, we put it into a map Λ_l (line 10). Next, if Λ_l is nonempty, we generate the next-level candidate patterns and update T (lines 11-12), as discussed in the above **Candidate Generation** step. Otherwise, we stop (lines 13-14). Finally, we output the results of this PSPM query (line 15).

6.3 The Advanced Method

Although `Basic` can answer a PSPM query, it needs to examine all the feasible sub-patterns. In this section, we show that it is possible to find the solution by examining only a small number of feasible sub-patterns. We denote this method by `Advanced`; it differs from `Basic` not only in the generation of candidate sub-patterns, but also in their verification. In the following, we first review MARGIN [34], an algorithm proposed for mining maximal frequent patterns from graph databases, which is different from our problem. Then, we discuss the candidate sub-pattern generation by adapting MARGIN and verification in our context. Finally, we present the overall `Advanced` method.

6.3.1 Preliminaries: MARGIN



(a) Search space [34]. (b) Illustrating lattice and cut.

Fig. 14 Key concepts in MARGIN.

In [34], Thomas et al. proposed MARGIN, the first non-Apriori algorithm for maximal frequent subgraph mining. Apriori explores all frequent subgraphs by performing a bottom-up traversal of the search space; however, the maximal frequent subgraphs often lie in the middle of the search space, which implies that most of the exploration could have been avoided. MARGIN [34] restricts the search space by visiting only subgraphs that lie on the border of frequent and infrequent subgraphs, as shown in Fig. 14(a).

To shrink the search space, MARGIN relies on a key concept, called *lattice*, which is defined as follows.

Definition 6 (lattice [34]) A *lattice* L of graph G is a hierarchical structure, in which each node represents a subgraph of G. At each level of L, all the nodes correspond to sub-graphs of the same size, i.e., those having the same numbers of edges. The bottom node, corresponding to the empty subgraph, forms level 0, nodes of singleton vertices form level 1, and nodes corresponding to size-i subgraphs form level $i+1$ for i>0. A node CR is a *child* of node R, if the graph in R is the subgraph of the graph in CR, and the graphs differ by exactly one edge; conversely, the node R is a *parent* of CR. Each node is linked to its child nodes in the lattice.

Example 7 Fig. 14(b) gives an example of a lattice L for graph G. It consists of seven nodes; the bottom-most node is the empty subgraph and the top-most node represents G. Each parent-child pair is connected by a dashed line. □

Definition 7 (cut [34]) A *cut* in a lattice is a pair of nodes (CR, R), where CR is the child of R and R is frequent while CR is not.

In Fig. 14(b), suppose that the graph circled in dashed line is frequent and the graph G is infrequent, then this pair of nodes forms a cut, which is marked by a bold solid line.

To find all the maximal frequent subgraphs, MARGIN works as follows. First, it finds an initial cut by deleting edges of G one by one until a frequent subgraph R is found. Then, it invokes a method `ExpandCut` [34] on the cut (CR, R), which finds nearby cuts and recursively calls itself on each newly found cut until no more cuts can be found.

We illustrate this by an example. Assume that the cut in Fig. 15(a) is the initial cut (CR, R). `ExpandCut` finds nearby cuts as follows: (1) Report all pairs of nodes consisting of CR and CR's frequent parents as new cuts. Thus (C, P_1) in Fig. 15(b) is reported as a new cut. (2) For each frequent parent R_f of CR that has an infrequent child CR_i, report (CR_i, R_f) as a new cut, e.g., (C_2,P) in Fig. 15(c). (3) For each frequent parent R_f of CR_i, consider each of its frequent child R_i. Report (M, CR_i) as a new cut, where M is the common child of CR_i and CR, such as (M, C_1) in Fig. 15(c). (4) For each infrequent parent R_i of CR, consider any frequent parent S_f of R_i. Report (R_i, S_f) as a new cut, e.g., (P_2, S_1) and (P_2, S_3) in Fig. 15(d). The detailed algorithm `ExpandCut` can be founded in [34].

As proved in [34], by using `ExpandCut`, MARGIN can find all the maximal frequent subgraphs by considering only a small number of frequent subgraphs.

6.3.2 Algorithm Details

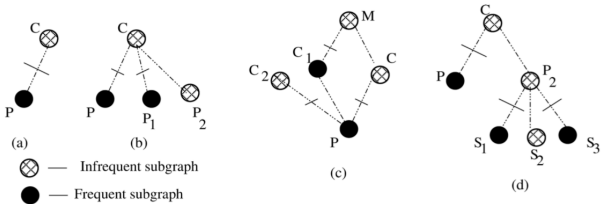


Fig. 15 Illustrating how `ExpandCut` works [34].

In `Advanced`, to avoid verifying all the feasible sub-patterns, we adapt MARGIN so that we only need to verify a small number of sub-patterns. Moreover, we develop a fast technique to check whether a sub-pattern is feasible. Algorithm 7 presents `Advanced`. In specific, we first run `PJ` for each edge of P, then find an initial s-cut (c, p) and invoke method `ExpandCutSPM` on it, and finally return all the matches of patterns. Next, we introduce `ExpandCutSPM`.

Algorithm 7: `Advanced`

Input: P, *root*;
Output: all the maximal feasible sub-patterns and their matches;

```
1 LF ← ∅ for each edge e_i of P do Φ_i ← run PJ for e_i;
2 Find an initial s-cut (c, p);
3 Run ExpandCutSPM(LF, c, p);
4 return sub-patterns in LF with their matches;
```

1. Method `ExpandCutSPM` is adapted from method `ExpandCut` [34] with three modifications:

- We use all the sub-patterns of P to build the lattice;
- We define a spatial-cut, or s-cut, as a pair nodes (CR, R), where R is the parent of CR, and CR is not feasible while R is feasible;
- We develop a method `VerifyPattern` to verify whether a sub-pattern is feasible or not in `ExpandCutSPM`.

To prove the correctness of `ExpandCutSPM`, we first present an interesting property, called *upper*-◇-property.

Lemma 10 (The *upper*-◇-property) *Given the pattern lattice L of query pattern P, any two child nodes of a node $p \in L$ have a common child node.*

Proof. We can transform the pattern lattice to a graph lattice by removing the distance intervals and edge signs from the sub-patterns. Since the *upper*-◇-property holds for the graph lattice, which has been proved to be correct by [34], and the removal operations do not affect the child-parent relationship, the *upper*-◇-property holds for the pattern lattice. □

Lemma 11 `ExpandCutSPM` *finds all maximal feasible sub-patterns of P.*

Proof. `ExpandCut` is proved to be correct when the purpose is to find all maximal frequent subgraphs from a graph [34]. Its correctness holds if the problem satisfies the following constraints [34]:
(1) The search space is a subset of elements in a lattice.
(2) The *upper*-$\diamond$-property holds.
(3) The elements of the lattice satisfy either the monotonicity or the anti-monotonicity property.
(4) There exists a candidate set $\mathcal{C}$, also called "boundary" set: if the anti-monotonicity property holds for the lattice, $\mathcal{C}$ is a set such that each element of $\mathcal{C}$ satisfies the given user-constraints while one of its immediate child nodes does not; if monotonicity holds for the lattice, each element of $\mathcal{C}$ satisfies the constraints while one of their parents does not satisfy the constraints.
(5) The solution set can be generated from $\mathcal{C}$.

For our PSPM problem, we have already proved the *upper*-$\diamond$-property and anti-monotonicity, so constraints (2) and (3) hold. Also, it is easy to observe that other constraints hold as well. Hence, `ExpandCutSPM` finds all maximal feasible sub-patterns of P. □

Similar to `ExpandCut`, `ExpandCutSPM` allows us to find all the feasible sub-patterns through verifying a small fraction of feasible sub-patterns, which results in high efficiency.

2. Method `VerifyPattern`. A naive method to verify whether a sub-pattern is feasible is to invoke one of the SPM algorithms (e.g., `MSJ`) and count the number of matches returned. This, however, may be very inefficient because there may be an exponential number of matches. To enable more efficient verification, we now propose a `VerifyPattern` method, which dramatically reduces the computational cost, as it verifies the candidate sub-patterns without computing any matches. We motivate its design by Example 8.

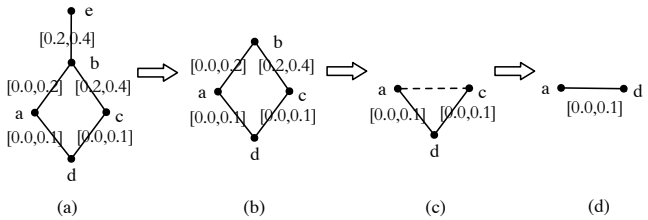


Fig. 16 Illustrating how `VerifyPattern` works.

Example 8 To verify whether the pattern in Fig. 16(a) is feasible or not, we can first run `PJ` for edge (e,b). Then, for vertex b, we keep a list of objects which match with it and are from the returned e-matches, and simplify the pattern to the one in Fig. 16(b). Next, we run `PJ` for (a,b) and (a,c), link their e-matches, filter these partial matches by the objects kept on vertex b, and get pairs of objects, each of which match with a and c respectively. Similarly, we can perform the same operations for remaining edges. Clearly, the method above keeps at most $|D|^3$ partial matches. In contrast, using `MSJ` may result in $|D|^5$ matches in the worst case. □

Why is the method discussed above faster for verification? The reason is that, for each sub-pattern of P, after linking the e-matches, the results are simplified. For example, after linking the e-matches of (b,a) and (b,c), we get a list of *triples* and then simplify them as a list of *tuples*, each of which match with a and c respectively. Thus, we do not need to list all the matches of P, which significantly improves the efficiency.

We now formally introduce `VerifyPattern`, which works in an iterative manner. Each time, we first pick the vertex with the minimum degree from P. Then, we get the e-matches of its neighboring edges, link them together as in `MPJ`, and simplify the join results. Let us denote the picked vertex by v. We have the following simplification criteria.

❶ **v's degree is 1:** we let its neighbor be u, and simplify all the e-matches of (v,u) as a set S of objects, which match with the keyword of u. We put (u,S) as a key-value pair in a map X.

❷ **v's degree is 2:** suppose its two neighbors are u and w. We first link the e-matches of (v,u) and (v,w) and get a list of triples. If v is a key in X, we delete triples which do not contain an object in the set X.get(v). Then, for each triple, we remove the object matched with v and get a tuple. Note that tuples with the same pair of objects are simplified as one tuple. Finally, we get a set T of tuples matched with (u,w) and put((u,w), T) as a key-value pair in a map Y.

❸ **v's degree is at least k ($k \geq 3$):** let v's k neighbors be v_1, v_2, $\cdots$, v_k. Similar to criterion ❷, we first link the e-matches of all its edges and get a list of k triples. Next, for each vertex or edge, if it is a key in X or Y, then we filter these k-tuples using its values in X and Y. Finally, for each pair of v's neighbors, we simplify these k-triples as a list of tuples, and put them into Y.

After the simplification for one vertex, we remove it and its incident edges from P. The process above will be performed iteratively until P does not contain any edge. Notice that, during this process, if we cannot find any match for a particular vertex, then P should be infeasible and we can stop immediately.

Algorithm 8 presents `VerifyPattern`. The input is P and the e-match sets of its edges, which must be nonempty; otherwise P must be infeasible (line 1). We first initialize two maps X and Y, which maintain the simplified join results (line 2). Then, in the while loop

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. 17 Datasets used in our experiments.

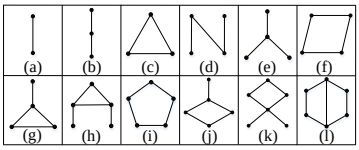


Fig. 18 Structures of patterns.

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. 19 Parameter settings.

Algorithm 8: `VerifyPattern`

Input: P, Φ_1, Φ_2, $\cdots$, Φ_m;
Output: whether P is feasible or not;
1 **for** $i \leftarrow 1$ to m **do if** $\Phi_i=\emptyset$ **then return** false;
2 $X \leftarrow \emptyset$, $Y \leftarrow \emptyset$; //maintain the simplified results;
3 **while** P *has at least one edge* **do**
4 $v \leftarrow$ the vertex with the minimum degree in P;
5 **if** v*'s degree is 1* **then**
6 let v's neighbor be u, and $e_i=(v, u)$;
7 $S \leftarrow$ objects that match with u and are in Φ_i;
8 $S \leftarrow$ filter objects in S using X and Y;
9 **if** $S \neq \emptyset$ **then** X.put(u, S);
10 **else return** false;
11 **else if** v*'s degree is 2* **then**
12 let v's neighbors be u, w, and $e_j=(v, u)$, $e_k=(v, w)$;
13 $T \leftarrow$ link the e-matches in Φ_j and Φ_k;
14 $T \leftarrow$ simply the triples in T to tuples using X and Y;
15 **if** $T \neq \emptyset$ **then** Y.put((u, w), T);
16 **else return** false;
17 **else**
18 Similar to lines 10-13;
19 remove v and its incident edges from P;
20 **return** *true*;

(lines 3-19), each time we pick the vertex v with the minimum degree and process it using the simplification criteria. After that, we remove v and its incident edges from P. If P is feasible, the loop will stop when P does not have any edge (line 20); otherwise it stops earlier.

Discussion. Each time we process the vertex with minimum degree in the residual sub-pattern of P. Let $k_{\max}$ be the maximum value of degrees of all the vertices processed. Note that $k_{\max}$ is also called the *maximum core number* of the k-cores in a graph and we usually have $k_{\max} \ll n$ [32]. Since for each vertex there are at most $k_{\max}$ neighboring edges, at most $|D|^{k_{\max}+1}$ partial matches will be considered. Besides, we need to keep at most $\frac{k_{\max}(k_{\max}-1)}{2}|D|^2$ e-matches for each vertex or edge in the maps X and Y. Note that the while loop is executed at most n–1 times. Hence, the overall worst-case time cost is $O(n|D|^{k_{\max}+1})$, which is much lower than that of answering an SPM query.

7 Experiments

7.1 Setup

Datasets. We use four real datasets. Fig. 17 reports statistics about them (number of objects, number of unique/total keywords). Dataset UK contains points of interest (e.g., banks) 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 Fig. 18). 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 our technical report [39]. For each graph G in Fig. 18, 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).

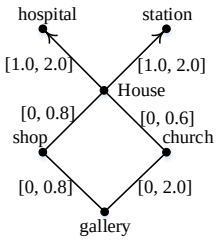
Step-2: For each vertex v_i with one of its neighbor v_j, we introduce a parameter η=90% such that the probabilities for the four different signs, i.e., $v_i \rightarrow v_j$, $v_j \leftarrow v_i$, $v_i \leftrightarrow v_j$, and v_i–v_j, are $\eta\times(1–\eta)$, $\eta\times(1–\eta)$, $(1–\eta)\times(1–\eta)$, 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 $5km$.

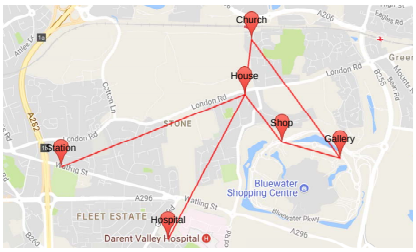
By following the steps above, for each structure, we generate 20 patterns with each having at least one match. Thus, there are 240 patterns for each dataset.

Queries. We use an IR-tree index [37], having a fanout B=100, i.e., the maximum number of children

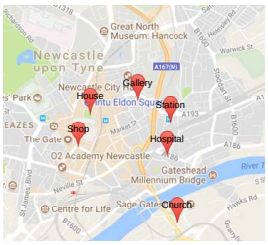
Fig. 20 mCK result for pattern in Figure 4 (measure: km).



(a) Pattern



(b) A match of pattern in (a)



(c) mCK query result

Fig. 21 Case study results for the pattern in (a) (measure: km).

of each node; the whole IR-tree is kept in memory. The inverted object list for each keyword, used by `MPJOrder`, is stored in a single file on disk. We consider five parameters: ϵ (used in `MPJOrder`), δ (used in `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 19. 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.

7.2 Effectiveness and Efficiency of SPM Queries

7.2.1 A Case Study

We consider the UK dataset, and two patterns. The first pattern is shown in the top-left panel of Fig. 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 Fig. 21(a). It can be applied to find 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 space limitations, we only show one match for each pattern. For comparison, we use the mCK query [43,19], whose input is the set of keywords in a pattern.

The results of the SPM and mCK queries for the first pattern are depicted in Fig. 4 and 20, respectively. From Fig. 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-relationship of edges (e.g., *house*→*school*) into consideration. Similarly, in Fig. 21, the SPM query can find a set of objects exactly matching the pattern in Fig. 21(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 in finding spatial objects with various distance conditions.

7.2.2 Effectiveness of Estimation Method in MPJ

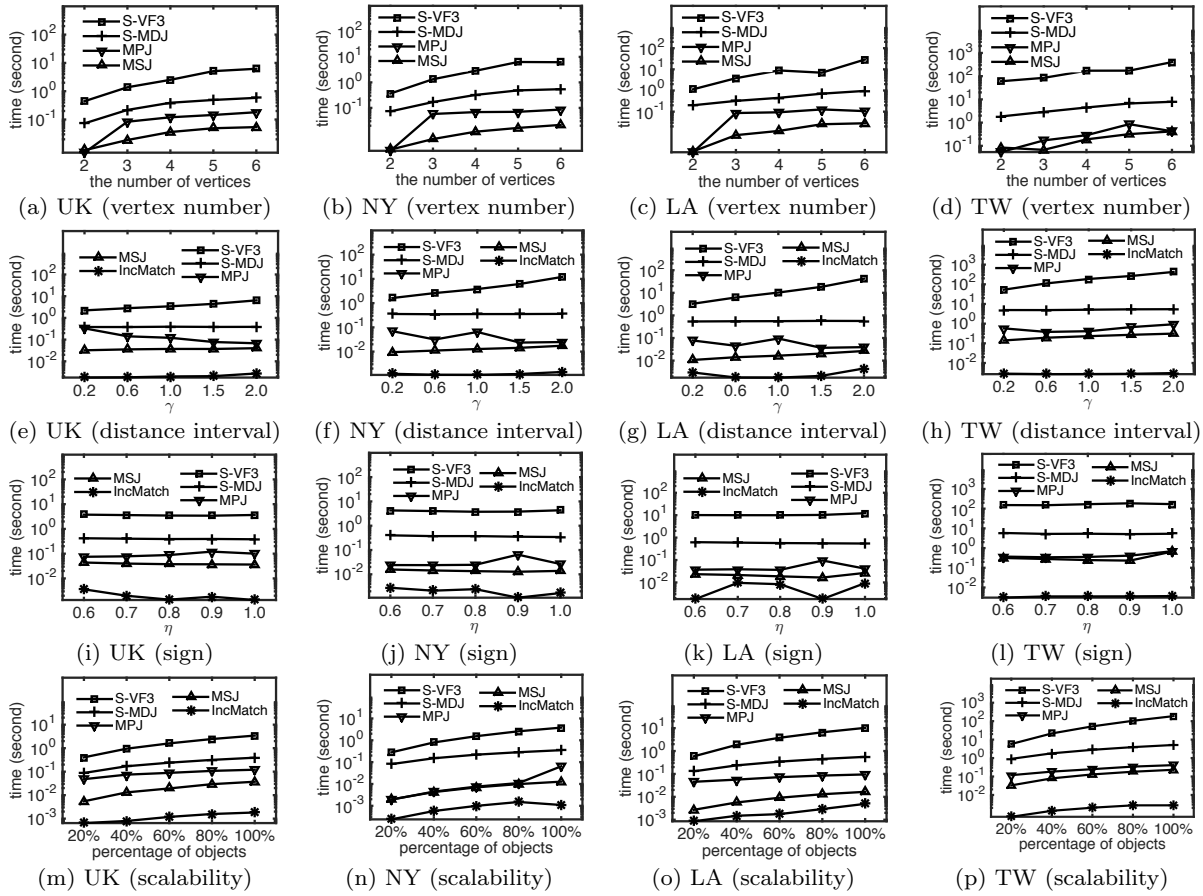
In `MPJ`, we estimate the number of e-matches for each edge with mutual inclusion using a sampling method. By Lemma 2, the estimation method theoretically guarantees that the failure probability is at most δ if the multiplicative error is set as ϵ. In this experiment, we evaluate the effect of ϵ and δ on the actual error. Consider an edge in a pattern. Let r and $\widehat{r}$ be its actual and estimated numbers of e-matches, respectively. The error rate is then 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 the 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 (ζ=0.5). Finally, we show the average error. Note that the true number of e-matches is computed by `PJ`.

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.

7.2.3 Efficiency Results

Effect of pattern size. For each dataset, we divide its patterns into five groups according to their vertex numbers. Fig. 22(a)-22(d) report the average runtime of a query for each group. Generally, as the number of vertices in the patterns increases, the performance gaps among the algorithms become larger. The time cost of `S-VF3` and `S-MDJ` does not always increase with



(a) UK (vertex number) (b) NY (vertex number) (c) LA (vertex number) (d) TW (vertex number)

(e) UK (distance interval) (f) NY (distance interval) (g) LA (distance interval) (h) TW (distance interval)

(i) UK (sign) (j) NY (sign) (k) LA (sign) (l) TW (sign)

(m) UK (scalability) (n) NY (scalability) (o) LA (scalability) (p) TW (scalability)

Fig. 22 Efficiency results of SPM queries.

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 the baseline algorithms and `MSJ` is over an order of magnitude faster than them. 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 queries 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 have similar efficiency. Thus, for small datasets, `MPJ` is an alternative to `MSJ`, as it is easier to implement.

Effect of 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 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 query in Fig. 22(e)-22(h).

Clearly, with the increase of γ, all algorithms except `MPJ` takes more time. 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. However, this is not the case for `MPJ` on dataset TW. Recall that `MPJ` consists of three parts, namely running MPJOrder, running PJ, and joining e-matches. Although the time of running `PJ` and joining edge matches increases when γ grows, the time of running `MPJOrder` decreases since the sampling time decreases because there are more e-matches for each edge. For datasets UK, LA and NY, running `MPJOrder` takes more time than running `PJ` and the join process, so the overall running time decreases as γ grows. On the other hand, for dataset TW, after increasing γ, the number of objects involved is much larger, so `MPJOrder` takes less time and the overall running time increases as γ grows.

Note that, in the worst case, the number of matches could be exponentially large. For example, if we set γ=100, then all our algorithms fail to process almost all of the patterns with 4 or more vertices within one minute, because either the number of matches is too large or they run out of memory.

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 runtime of a query for each group in Fig. 22(i)-22(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 query evaluation becomes 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 Fig. 19, select a percentage of χ from its objects randomly, and obtain four sub-datasets. Fig. 22(m)-22(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 `S-VF3` and `S-MDJ`, and `MSJ` scales the best.

Optimization techniques in `MSJ`. Recall that there are three key optimization techniques in `MSJ`, namely bounded patterns, star-pruning, and anchor-pruning. We modify `MSJ` such that it has 4 different variants, denoted by V_1, V_2, V_3, and V_4. V_1 is a version of `MSJ` that applies none of the above techniques is used, while V_1, V_2, and V_3 denote simplified versions of `MSJ` which do not implement one of these optimization techniques, respectively. We report the average running time of each pattern on two datasets in Table 2.

Observe that all the variants are slower than `MSJ`, which implies that all of them indeed help improving the efficiency. Among these variants, V_1 is the slowest one, since it does not use any optimization technique. Besides, we can see that star-pruning is more effective for pruning than anchor-pruning. We remark that the reported results are average running times over all the 240 patterns. Recall that these optimization techniques are mainly designed for large and complicated patterns. Specifically, bounded patterns and star-pruning are designed for patterns with at least three vertices, while anchor-pruning can improve efficiency only when the pattern has at least four vertices.

Table 2 Benefit of optimization techniques in `MSJ` (ms).

Datasets	V_1	V_2	V_3	V_4	`MSJ`
NY	14.06	13.53	13.12	12.92	12.79
LA	20.67	18.18	18.06	17.68	17.57

Join ordering methods. In this experiment, we compare the efficiency of the three methods that can be used to determine the join order, namely dynamic programming (`DP`), `MPJOrder`, and `MSJOrder`. The average runtimes of these methods on NY and LA are reported in Fig. 23. For small patterns, the difference between these methods is insignificant, so we use patterns with 5 and 6 vertices in this experiment. As can be seen, `DP` takes the longest time to obtain the join order. The reasons are two-fold. First, `DP` needs to compute all the e-matches for each edge before computing the optimal join order while the other two methods may skip some edges. Second, `DP` needs to search a large search space, while the other methods work in a greedy manner. Besides, `MSJOrder` is faster than `MPJOrder`, because the number of matched non-leaf nodes enumerated in `MSJOrder` is much smaller than that of the e-matches.

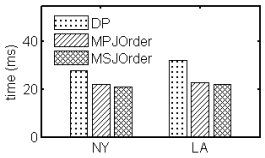


Fig. 23 Join orders.

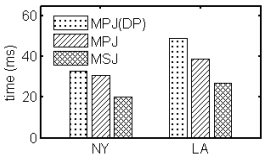
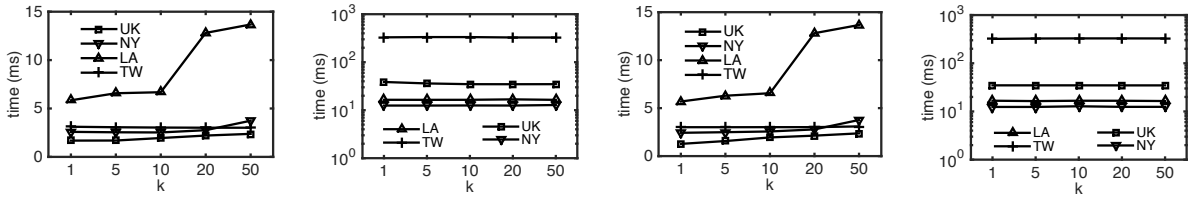


Fig. 24 SPM queries.

To assess the effect of these methods, we create a variant of `MPJ`, denoted by `MPJ(DP)`, which uses `DP` to compute the join order. Then, we use the patterns of the experiment above and run `MPJ(DP)`, `MPJ`, and `MSJ` on NY and LA datasets, and report the overall runtime in Fig. 24. Clearly, `MSJ` is the fastest approach. Besides, we observe that although `DP` can compute the optimal join order which may reduce the join time cost, it is still slower than `MPJ` since the time dedicated for join order optimization is more.

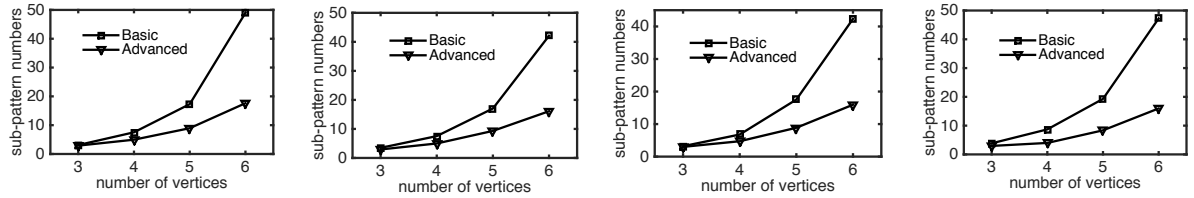
7.3 Efficiency of Top-k SPM

To evaluate top-k SPM, we consider all datasets and the 240 spatial patterns introduced in Section 7.1. For each pattern, we generate two query locations following different distributions. That is, one follows a *random distribution*, and the other one follows the *object distribution*, i.e., locations with more objects have higher



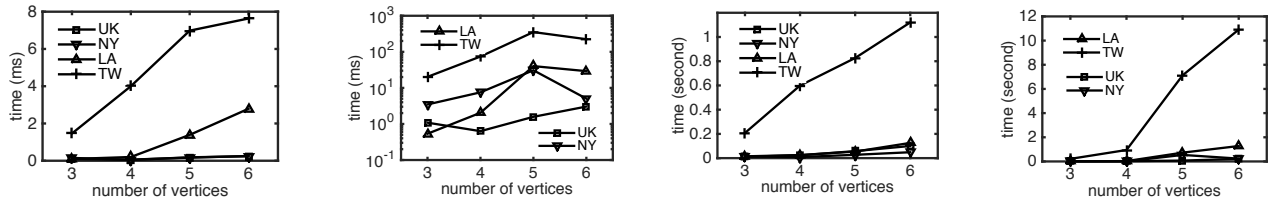
(a) `IncMatch` (random) (b) `MSJ` (random) (c) `IncMatch` (grid) (d) `MSJ` (grid)

Fig. 25 Efficiency results of top-k SPM queries.



(a) UK (search space) (b) NY (search space) (c) LA (search space) (d) TW (search space)

Fig. 26 Comparing the numbers of verified sub-patterns in PSPM queries.



(a) `VerifyPattern` (verification) (b) `MSJ` (verification) (c) `Advanced` (vertex number) (d) `Basic` (vertex number)

Fig. 27 Efficiency results of PSPM queries.

probabilities to be selected as query locations. As a result, for each dataset, we get two groups of top-k SPM queries, each containing 240 queries whose query locations follow a specific distribution.

In the experiments, we set the default value of k to 10. We vary the value of k in $\{1, 5, 10, 20, 50\}$, and run `MSJ` and `IncMatch` for these two groups of queries on each dataset. Since the lengths of the distance intervals of these patterns are a few kilometers, for simplicity we set θ=20km in `IncMatch`. Fig. 25 presents the average runtime for each query.

Effect of k. We observe that `IncMatch` consistently performs faster than `MSJ`, and the runtime cost of `MSJ` is almost stable when k varies from 1 to 50. This is because, to find the top-k matches, `MSJ` first computes all the matches, and this dominates the overall cost. On the other hand, `IncMatch` finds the top-k in an incremental manner and does not need to compute all the matches, so it performs faster. In addition, by varying k from 1 to 50, the running time of `IncMatch` increases.

Nevertheless, the running time of `IncMatch` is not proportional to the sizes of datasets. For example, for the LA dataset which is not the largest one, it takes the longest time. The main reason is that `IncMatch` adopts a local search strategy. That is, it finds the top-k matches from a spatial circle centered at the query location loc and incrementally increases the radius of the circle by θ. In other words, the search space is in a small region surrounding the query location loc, irrespectively to the size of the dataset.

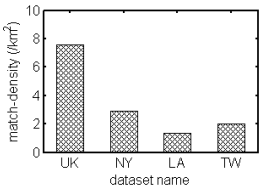


Fig. 28 Match-density.

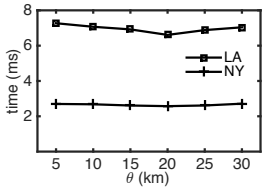


Fig. 29 Effect of θ.

To further investigate the performance fluctuation on different datasets, we introduce a concept, called *match-density*. Let the circle containing the top-k matches be $O(loc, r)$, where r is the radius. Then, the match-density is defined as $\frac{k}{\pi \times r^2}$. Intuitively, if the match-density is small, `IncMatch` takes more time to find the top-k matches, since each time it increases r by θ and finally stops when k matches are found. For each dataset, we consider the 240 queries and compute their average match-density (k=10). The results are depicted in Fig.

28. As can be seen, LA has the lowest match-density while UK has the highest match-density, so IncMatch takes more time on the LA compared to UK. Thus, the performance mainly depends on the local match-density.

Effect of the location generation methods. From the results, we observe that, again IncMatch performs faster than MSJ, and the two groups of queries with different distributions of query locations achieve almost the same efficiency on each dataset. Therefore, the proposed algorithm IncMatch is very efficient and robust.

Effect of the distance interval length. We vary the value of γ which controls the length $|u_{i,j} - l_{i,j}|$ and report the average running time of IncMatch for a query in Fig. 22(e)-22(h). Clearly, as the size of the interval grows, the running time of IncMatch increases.

Effect of signs. We vary the value of η and report the average running time of IncMatch for a query in Fig. 22(i)-22(l). We observe that, as the value of η increases, the running time of IncMatch changes slightly, not obeying any rule. This is because, FindMatches does not have to compute e-matches of edges with exclusion-relationship at first as MSJ does. Instead, IncMatch processes edges with exclusion-relationship or inclusion-relationship in a similar way.

Scalability. We report the scalability test results of IncMatch in Fig. 22(m)-22(p). As can be seen, IncMatch scales near linearly with the size of dataset.

Effect of θ. As shown in Fig. 29, when θ varies from $5km$ to $30km$, the cost of IncMatch does not change much. Thus, the efficiency of IncMatch is not very sensitive to the parameter θ, so we set it to $20km$ in our experiments. In practice, users can simply set it as the mean of distance values in the pattern.

7.4 Effectiveness and Efficiency of PSPM

Recall that PSPM is applied when the pattern P has no exact matches in the database. To evaluate PSPM, we need to create patterns that do not have matches. Specifically, we focus on the 11 structures of Fig. 18(b) to 18(l). For each of them, we follow the steps of pattern generation in Section 7.1 and generate 20 patterns, such that each one of them has no exact match in the database and its maximal feasible sub-patterns have more than half of the edges of the pattern. Thus, we get 220 patterns for each dataset.

As discussed in Sections 6.2 and 6.3, to generate candidate sub-patterns, Basic works in an Apriori fashion, whereas Advanced adapts MARGIN [34]. We first compare the search space (i.e., the numbers of verified sub-patterns) of these candidate sub-pattern generation methods. The results are reported in Fig. 26. The search space of Advanced is consistently smaller than that of Basic. Besides, with the number of vertices in the patterns increased, the gaps become larger.

Next, in Fig. 27, we report the average runtime for each PSPM query on each dataset.

Pattern verification. In this experiment, we compare the efficiency of VerifyPattern with a naive method, i.e., MSJ, for verifying whether a sub-pattern is feasible or not. The average cost of verifying a sub-pattern is shown in Fig. 27(a)-27(b). We observe that, when pattern sizes get larger, VerifyPattern is almost two orders of magnitude faster than MSJ. The reason is that MSJ is designed for computing all matches of the query pattern, while VerifyPattern only checks whether the pattern has matches or not. Meanwhile, for large datasets, the cost tends to be larger, because larger datasets have more partial matches.

Effect of pattern size. For each dataset, we divide its query patterns into four groups according to their vertex numbers. Fig. 27(c)-27(d) report the average query cost for each group. We observe that Advanced is consistently faster than Basic, and as the pattern size increases, the performance gap becomes larger. Moreover, Advanced is around an order of magnitude faster than Basic on the three large datasets. First, Advanced uses ExpandCutSPM to restrict the search space. Second, Advanced uses VerifyPattern, which is faster for verifying whether a sub-pattern is feasible. Besides, the overall running time increases with the growth of dataset sizes. The main reason is that the verification time increases as the sizes of datasets grow.

8 Related work

Spatial keyword queries (SKQs). Recently, spatial queries [9,14,15,17] have been extensively studied and SKQ is one of the most well-studied queries. In the literature, there are two specific types of SKQs. The first type (e.g., [11,37,40,9]) 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* [11]) were proposed to enable fast query evaluation. In [37,40], the top-k SKQ is studied. The authors in [20] proposed an SKQ, which continuously returns k objects when the query location moves. In [41], this solution was extended to apply to road networks. In [23], Mahmood et al. proposed a query language, called Atlas, an SQL extension to express spatial keyword group queries.

The second type of queries take as arguments a set of keywords and return a group of objects [43,19,6,12,

10] that are close to each other, and collectively 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 [43,19], 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 7, our SPM query captures users' requirements better than the mCK query. Its variants include [10], which minimizes a different distance cost function, and [12], which considers ratings of objects. The authors of [6] consider the distance between the query location and the returned group for the SKQs. A recent work [22] 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. [30] express query constraints as graphs and retrieve objects satisfying the query graphs, by using the R-tree index [5]. 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 [24]. The optimization of these join queries was studied in [25]. However, these studies cannot be applied to solving our SPM query. The reasons are two-fold: First, existing multi-way spatial join studies mainly focus on spatial databases that are not associated with keywords. Second, existing multi-way spatial join studies do not consider the exclusion-ship among objects, which is a novel feature in our SPM pattern. In addition, we adapted a recent graph pattern matching algorithm (called MDJ), which works in a similar manner to multi-way spatial join, and performed an experimental study. The results show that the adapted MDJ solution, called `S-MDJ`, is very inefficient, calling for faster solutions.

Graph pattern matching (GPM). Given a graph G and a pattern graph P, the GPM query [18,45,36, 7] extracts a set R of subgraphs of G, where for each $r \in R$, r matches with P. Zou et al. [45] study the GPM problem on undirected graphs. Tong et al. [36] propose a solution for common shapes of P, such as a line, a loop, and a star. A recent work [7] proposes a fast GPM algorithm VF3 based on subgraph isomorphism. Another recent work [33] finds groups of objects matching with a clique pattern, but it does not consider any keywords. 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 typically do not have any distance requirement [7] or just have an upper bound distance [45] on each edge, while in our patterns, each edge not only has the minimum/maximum distance requirements, but also the inclusion/exclusion-relationship. Thus, all these methods cannot be used to process SPM queries directly. We adapted two GPM solutions [45, 7] as baselines, but they were found inefficient.

There are also previous studies that perform approximate GPM; some of them, find subgraphs that match with a sub-pattern. Zhang et al. [42] propose to find subgraphs, whose edit distances with the pattern are less than a predefined threshold. Zhu et al. [44] retrieve matches of the query graph with the number of possible missing edges bounded by a given threshold. Similar to [44], Mongiovi et al. [27] focus on inexact matches of the query graph with at most number r of deletions. Tian et al. [35] find approximate matches not only allowing missing edges and nodes but also supporting node mismatches (nodes with different labels are matched). To find approximate matches for our SPM queries, we borrow ideas from these studies and find partial matches of the maximal feasible sub-patterns of the query pattern.

9 Conclusions and Future Work

In this paper, we study the spatial pattern matching (or SPM) problem. We first show that this problem is computationally intractable. Then we propose two efficient algorithms for SPM queries, namely `MPJ` and `MSJ`. Moreover, we study two useful variants of the SPM problem, which are called the top-k SPM and partial SPM (PSPM) problems. The top-k SPM finds the k nearest matches to a query location, and the PSPM query finds partial matches that maximally match with the query pattern, when there are no exact matches for it. Based on the SPM query, we developed a demo system, called SpaceKey, which allows users to explore patterns in spatial database. Furthermore, we performed experiments on real datasets, and the results show that our SPM queries are more effective than the state-of-the-art SKQs. In addition, the `MSJ` algorithm is up to an order of magnitude faster than the baseline solutions. Finally, the algorithms of top-k and PSPM queries are very efficient compared to baseline alternatives.

In the future, we plan to increase the expressiveness power of SPM queries. For example, we will include logical operations (e.g., AND and OR) into SPM, supporting for instance the case where we want to find a house that has nearby a hospital or a doctor. In addition, it would be interesting to investigate other possible solutions for handling the under-matched case. For example, a possible solution is to automatically relax

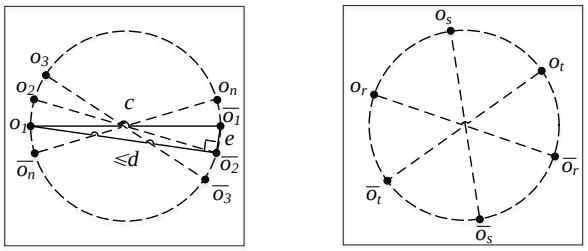
the distance constraints of the pattern slightly so that it has matches in the database.

References

1. `https://en.wikipedia.org/wiki/Geometric_distribution`.
2. `https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`.
3. Settlement patterns. http://geography.parkfieldprimary.com/the-united-kingdom/settlement-patterns, 2017.
4. V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.
5. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. *SIGMOD*, pages 237–246, 1993.
6. X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384. ACM, 2011.
7. V. Carletti et al. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. *TPAMI*, 2017.
8. F. Chen and X. Wu. Perfect pipelining for streaming large file in peer-to-peer networks. In *Theoretical Computer Science*, pages 27–38, 2014.
9. L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. *PVLDB*, pages 217–228, 2013.
10. D. Choi, J. Pei, and X. Lin. Finding the minimum spatial keyword cover. In *ICDE*, pages 685–696. IEEE, 2016.
11. G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *VLDB*, 2(1):337–348, 2009.
12. K. Deng, X. Li, J. Lu, and X. Zhou. Best keyword cover search. *TKDE*, 27(1):61–73, 2015.
13. Y. Fang, R. Cheng, G. Cong, N. Mamoulis, and Y. Li. On spatial pattern matching. In *ICDE*, pages 293–304. IEEE, 2018.
14. Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu. Effective community search over large spatial graphs. *PVLDB*, 10(6):709–720, 2017.
15. Y. Fang, R. Cheng, W. Tang, S. Maniu, and X. Yang. Scalable algorithms for nearest-neighbor joins on big trajectory data. *TKDE*, 28(3):785–800, 2016.
16. Y. Fang, R. Cheng, J. Wang, Budiman, G. Cong, and N. Mamoulis. SpaceKey: exploring patterns in spatial databases. In *ICDE*, pages 1577–1580. IEEE, 2018.
17. Y. Fang, Z. Wang, R. Cheng, X. Li, S. Luo, J. Hu, and X. Chen. On spatial-aware community search. *TKDE*, pages 1–1, 2019.
18. B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 6:45–53, 2006.
19. T. Guo, X. Cao, and G. Cong. Efficient algorithms for answering the m-closest keywords query. In *SIGMOD*, pages 405–418. ACM, 2015.
20. W. Huang, G. Li, K.-L. Tan, and J. Feng. Efficient safe-region construction for moving top-k spatial keyword queries. In *CIKM*, pages 932–941. ACM, 2012.
21. J. Jin, N. An, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *ICDE*, pages 525–534, 2000.
22. J. Liu, K. Deng, H. Sun, Y. Ge, X. Zhou, and C. S. Jensen. Clue-based spatio-textual query. *PVLDB*, 10(5):529–540, 2017.
23. A. R. Mahmood, W. G. Aref, A. M. Aly, and M. Tang. Atlas: on the expression of spatial-keyword group queries using extended relational constructs. In *SIGSPATIAL*, page 45. ACM, 2016.
24. N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. *SIGMOD*, 28(2):1–12, 1999.
25. N. Mamoulis and D. Papadias. Multiway spatial joins. *TODS*, 26(4):424–475, 2001.
26. Ministry of Education of Singapore. https://www.moe.gov.sg/admissions/primary-one-registration/allocation, 2017.
27. M. Mongiovi et al. Sigma: a set-cover-based inexact graph matching algorithm. *Journal of bioinformatics and computational biology*, 8(02):199–218, 2010.
28. J. Niemelä. Ecology and urban planning. *Biodiversity and conservation*, 8(1):119–131, 1999.
29. D. Papadias, N. Mamoulis, and Y. Theodoridis. Processing and optimization of multiway spatial joins using r-trees. In *PODS*, 1999.
30. D. Papadias et al. Algorithms for querying by spatial structure. In *VLDB*, pages 546–557, 1998.
31. 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*, 30(1):24–34, 2002.
32. S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.
33. M. Tang et al. Similarity group-by operators for multi-dimensional relational data. *TKDE*, 28(2):510–523, 2016.
34. L. T. Thomas, S. R. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. *TKDD*, 4(3):10, 2010.
35. Y. Tian et al. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972. IEEE, 2008.
36. H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, pages 737–746, 2007.
37. D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *TKDE*, 2012.
38. Y. Wu, J. M. Patel, and H. Jagadish. Structural join order selection for xml query optimization. In *ICDE*, pages 443–454. IEEE, 2003.
39. Y. Fang, Y. Li, R. Cheng, N. Mamoulis, G. Cong. On spatial pattern matching. `http://www.cse.unsw.edu.au/~z3525370/spm2019.pdf`.
40. C. Zhang, Y. Zhang, W. Zhang, and X. Lin. Inverted linear quadtree: Efficient top-k spatial keyword search. *TKDE*, 28(7):1706–1721, 2016.
41. C. Zhang, Y. Zhang, W. Zhang, X. Lin, M. A. Cheema, and X. Wang. Diversified spatial keyword search on road networks. In *EDBT*, pages 367–378, 2014.
42. S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1-2):1185–1194, 2010.
43. D. Zhang et al. Keyword search in spatial databases: towards searching by document. In *ICDE*, pages 688–699. IEEE, 2009.
44. G. Zhu et al. Treespan: efficiently computing similarity all-matching. In *SIGMOD*, pages 529–540. ACM, 2012.
45. L. Zou, L. Chen, and M. T. Özsu. Distance-join: pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.

A Proof of Lemma 1

We prove the lemma by a reduction from the 3-SAT problem. An instance of the 3-SAT problem consists of ϕ=$C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause C_j=$x_j \vee y_j \vee z_j$(j=1,2, $\cdots, m$) and $\{x_j, y_j, z_j\} \subset \{u_1, \bar{u_1}, \cdots, u_n, \bar{u_n}\}$. The decision problem is to determine whether we can assign a value (true or false) to each variable $u_i, i = (1, 2, \cdots, n)$, such that ϕ is true. To transform an instance of the 3-SAT problem to an instance of the SPM problem, we first describe how to map the variables into spatial objects, and then discuss how to associate the keywords.



(a) Variables transformation. (b) Objects of a clause.

Fig. 30 Illustrating the NP-hard proof.

We assume all the spatial objects are in a unit square $[0, 1]^2$ data space. Let d be a value in $(0, 1)$. We consider a circle, whose center is at the center of the data space and has a diameter c=$d + \sigma$, where σ is a small positive value. We will discuss how to set the value of σ later. For variable u_1, we randomly place a corresponding spatial object o_1 on the circle. For its negation $\bar{u_1}$, we place an object $\bar{o_1}$ diametrically opposite on the circle, which implies $|o_1, \bar{o_1}|$=c. For the rest variables $u_2, u_3, \cdots, u_n$ and their negations, we place objects on the circle in the same way, such that $|o_1, o_2|$=$|o_2, o_3|$ =$\cdots$= $|o_{n-1}, o_n|$= $|o_n, \bar{o_1}|$ =$\cdots$=$|\bar{o_n}, o_1|$=e, where e can be computed using the cosine theorem:

$$e = \sqrt{\frac{c^2}{4} + \frac{c^2}{4} - 2 \cdot \frac{c^2}{4} \cdot cos\frac{180^\circ}{n}} = \sqrt{\frac{c^2}{2}(1 - cos\frac{180^\circ}{n})}. \quad (5)$$

We denote the set of all the placed $2n$ objects by Λ. Figure 30(a) illustrates the placement of objects in Λ. We now prove that it is possible to set a positive value of σ, such that for any object $o_i(\bar{o_i})$, the distance from it to any object in Λ, except $\bar{o_i}(o_i)$, is at most d. Let us consider o_1. Since the object furthest away from it in $\Lambda - \{\bar{o_1}\}$ is $\bar{o_2}$ (or o_n), we need to have $|o_1, \bar{o_2}| \leq d$. Notice that $o_1, \bar{o_1}$ and $\bar{o_2}$ form a right triangle. By pythagorean theorem, we have

$$|o_1, \bar{o_2}| = \sqrt{|o_1, \bar{o_1}|^2 - |\bar{o_1}, \bar{o_2}|^2} = \sqrt{c^2 - e^2} \leq d. \quad (6)$$

Considering Eq (5), we have $\sqrt{\frac{1}{2}(d + \sigma)^2(1 + cos\frac{180^\circ}{n})} \leq d$. Hence, to ensure the distance from o_1 to any object in $\Lambda - \{\bar{o_1}\}$ being at most d, we set σ as

$$0 < \sigma \leq \frac{d}{\sqrt{\frac{1}{2}(1 + cos\frac{180^\circ}{n})}} - d. \quad (7)$$

Then, we discuss how to associate keywords. For each pair of objects o_i and $\bar{o_i}$, we create one keyword w_i for them (i=1,2,$\cdots, n$). In other words, o_i and $\bar{o_i}$ share keyword w_i, and the only holders of w_i are o_i and $\bar{o_i}$. In addition, for each clause C_j in the instance ϕ of 3-SAT problem, we create one keyword v_j (j=1,2,$\cdots, m$) and associate it to the three objects corresponding to the three variables in C_j. Thus, given a 3-SAT instance ϕ, we have a spatial pattern P, in which (1) there are $(n + m)$ vertices (each corresponding to a distinct keyword); (2) each pair of vertices has an edge with a distance interval $[0, d]$; and (3) each pair of vertices is with the mutual inclusion.

Next, to complete the proof, we need to prove that: (1) a satisfying assignment of the 3-SAT instance ϕ determines a set of spatial objects matched with the spatial pattern P; (2) if there exists a feasible solution to the SPM problem, i.e., a set of objects match with the pattern P, then there also exists a satisfying assignment of ϕ. We first show that (1) holds. A satisfying assignment of ϕ means that, for any pair of variables u_i and $\bar{u_i}$ one of them must be true, and any clause is also true. All the objects that correspond to variables with true values form a match of P. This is because for each edge of P, we can find a pair of objects that match with the edge. In the following, we focus on proving (2).

Assume that we have a set Ψ of spatial objects matched with P, where objects in Ψ contains all the keywords $w_1, \cdots, w_n$ and $v_1, \cdots, v_m$, and the distance of each pair of objects is at most d. Consider any specific clause C_k=$u_r \vee u_s \vee u_t$ in ϕ, where $1 \leq r, s, t \leq n$ and $1 \leq k \leq m$. Since Ψ contains an object with keyword w_r, one of u_r and $\bar{u_r}$ must be assigned to be true. Similarly, we have this for u_s and $\bar{u_s}$, u_t and $\bar{u_t}$, respectively. It is easy to observe that, if any one of u_r, u_s, and u_t is true, then the value of C_k is true. The only assignment which makes the value of C_k false is when all the values of u_r, u_s, and u_t are assigned to be false. Next, we prove this case, however, cannot happen by contradiction. We show six objects corresponding to variables of C_k in Figure 30(b), where objects o_h and $\bar{o_h}$ ($h \in \{r, s, t\}$) correspond to variables u_h and $\bar{u_h}$ respectively. Suppose above case happens, which implies Ψ contains objects $\bar{o_r}$, $\bar{o_s}$, and $\bar{o_t}$. Since Ψ contains an object with keyword v_k, whose only holders are objects o_r, o_s, and o_t, Ψ must contain at least one of them. As a result, Ψ contains at least one pair of the three pairs of objects: o_r and $\bar{o_r}$, o_s and $\bar{o_s}$, and o_t and $\bar{o_t}$. However, we have $|o_r, \bar{o_r}|$=$|o_s, \bar{o_s}|$ =$|o_t, \bar{o_t}|$=$c \geq d$, which implies that Ψ is not a valid match since the distance requirement is not satisfied. Thus, the case that all of u_r, u_s, and u_t are assigned to be false cannot happen.

Therefore, we conclude that if there exists a feasible solution to the SPM problem, then there also exists an assignment of ϕ, which makes it true, and (2) holds. Hence, the proof is complete. □

B The Detailed Steps of MPJ

Algorithm 9 presents the pseudocodes of `MPJ`. The input is the *root* of the IR-tree and a spatial pattern P, and the output is the set, Ψ, of all the matches of P. We first initialize two variables Ψ and Φ (line 1), where Ψ maintains the partial matches and Φ is a temporary variable. Then, we run `MPJOrder` to get Γ and Υ (line 2). Next, we consider edges in Γ sequentially (lines 3-12). In case that the edge is a forward edge (lines 4-7), if it is with mutual inclusion, we run `PJ` to obtain all the e-matches of this edge; otherwise, we get the join results from Υ directly, and expand Ψ using Φ such that each partial match matches with a larger subgraph of P. In case that the edge is backward (lines 9-13), we prune the partial matches: if the edge is with mutual inclusion, we prune partial matches if the distances of the corresponding objects

Algorithm 9: `MPJ`

```
   Input: root, P;
   Output: Ψ, all the matches;
1  initialize Ψ ← ∅, Φ ← ∅;
2  run MPJOrder, and get Γ and ϒ;
3  for each edge (v_i, v_j) of Γ do
4      if it is a forward edge then
5          if v_i–v_j then Φ ←PJ(w_i, w_j, l_{i,j}, u_{i,j}, –);
6          else            Φ ← ϒ.get((v_i, v_j));
7          Ψ ← Ψ.link(Φ);
8      else
9          if v_i–v_j then
10             prune some partial matches in Ψ;
11         else
12             Φ ← ϒ.get((v_i, v_j));
13             prune some partial matches in Ψ by Φ;
14 return Ψ;
```

are not in $[l_{i,j}, u_{i,j}]$; otherwise, we get the join result Φ from Υ and prune partial matches if the corresponding objects are not in Φ. Finally, we return the set Ψ (line 14).