Superseding Nearest Neighbor Search on Uncertain Spatial Databases

Sze Man Yuen[1] Yufei Tao[1] Xiaokui Xiao[2] Jian Pei[3] Donghui Zhang[4]

Chinese University of Hong Kong[1]
{*smyuen*, *taoyf*}@*cse.cuhk.edu.hk*

Nanyang Technological University[2]
xiaokui@ntu.edu.sg

Simon Fraser University[3]
jpei@cs.sfu.ca

Northeastern University[4]
donghui@ccs.neu.edu

Abstract

This paper proposes a new problem, called *superseding nearest neighbor search*, on uncertain spatial databases, where each object is described by a multidimensional probability density function. Given a query point q, an object is a *nearest neighbor* (NN) *candidate* if it has a non-zero probability to be the NN of q. Given two NN candidates o_1 and o_2, o_1 *supersedes* o_2 if o_1 is more likely to be closer to q. An object is a *superseding nearest neighbor* (SNN) of q, if it supersedes all the other NN-candidates. Sometimes no object is able to supersede every other NN candidate. In this case, we return the *SNN-core* — the *minimum* set of NN-candidates *each of which* supersedes *all* the NN-candidates outside the SNN-core. Intuitively, the SNN-core contains the best objects, because any object outside the SNN-core is worse than *all* the objects in the SNN-core. We show that the SNN-core can be efficiently computed by utilizing a conventional multidimensional index, as confirmed by extensive experiments.

Keywords: Nearest neighbor, uncertain, spatial database.
To appear in *IEEE TKDE*.

1 Introduction

Uncertain databases have received a large amount of attention from the database community in recent years [5, 7, 9, 12, 14, 17]. In such a database, an object is described by a probability density function (pdf). For example, Figure 1a shows the possible locations of four objects A, B, C, D. Specifically, object A has probabilities 0.4 and 0.6 of being at points $A[1]$ and $A[2]$, respectively. We refer to $A[1]$ and $A[2]$ as the *instances* of A. Similarly, object B also has two instances $B[1]$ and $B[2]$, at which B is located with likelihood 0.6 and 0.4, respectively. Object C (D) has only one instance $C[1]$ ($D[1]$), i.e., its location has no uncertainty. It is worth mentioning that modeling of an uncertain object as a set of instances is a common approach in the literature [9, 12, 14, 17].

We consider *nearest neighbor* (NN) queries on uncertain objects. In general, there may not exist any object that is guaranteed to be the NN. For instance, assume that the query point q is at the cross in Figure 1a. Object A must be the NN of q if it is at $A[1]$. However, A cannot be the NN of q if it is at $A[2]$, in which case C is definitely closer to q. Combining both facts, it is clear that no object can be claimed as the NN with absolute certainty.

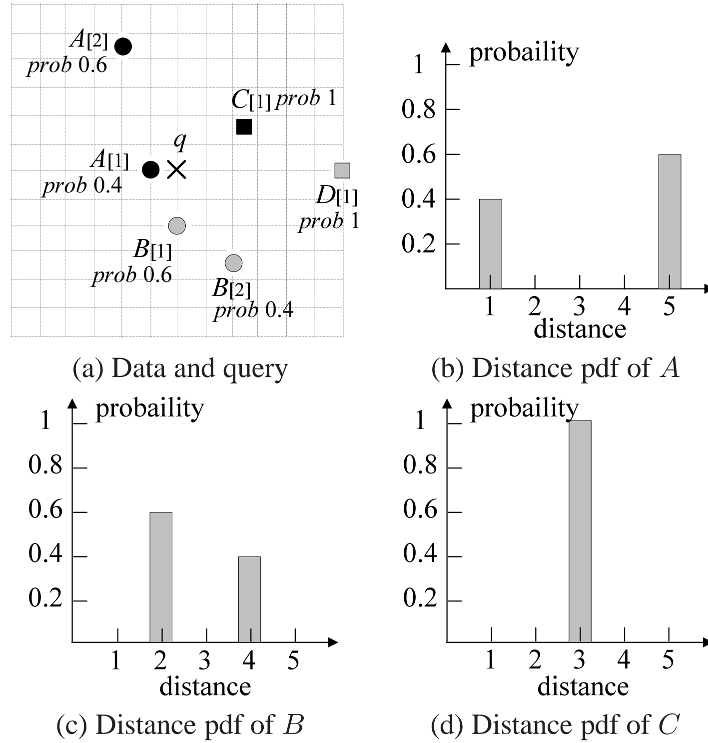
We say that an object is an *NN-candidate* if it *may be* the NN. The explanation earlier shows that A is an NN-candidate. Similarly, B is also an NN-candidate since it is the NN provided that it is at $B[1]$ and A is at $A[2]$. C is another NN-candidate, because it is the NN as long as A and B are at $A[2]$ and $B[2]$, respectively. However, D is not an NN-candidate, as its distance to q is larger than that of C. Apparently, when the number of NN-candidates is large, returning all of them to the user is a poor choice. Hence, it is important to select the *best few* NN-candidates. Many existing methods fulfill this purpose by analyzing objects' *NN-probabilities* [5], namely, the probability that an object is the NN. In this paper, we provide a new perspective to look at the issue: by analyzing objects' mutual superiority.

Before going into the details, let us first consider a relevant question: given two objects o and o', which is better? This *pairwise competition* has a clear answer when o and o' are precise points — the one closer to the query point q wins the competition, i.e., it *supersedes* the loser. How about o and o' being uncertain? The answer is still clear: the one *more likely* to be closer to q is better. Formally, o *supersedes* o' if the probability that q is nearer to o than to o' exceeds 0.5.

For example, consider objects A and B in Figure 1a, whose distances to q follow the pdfs in Figures 1b and 1c, respectively. For example, the distance pdf of A is 0.4 (0.6) at distance 1 (5), because A has probability 0.4 (0.6) to be located at point $A[1]$ ($A[2]$). As q is closer to A than to B only if A has distance 1, the probability that q is closer to A (than to B) equals 0.4. This implies that q has probability $1 - 0.4 = 0.6$ to be closer to B, namely, B supersedes A. By the same reasoning, it is easy to verify that C supersedes A, and B supersedes C.

Figure 2 shows the resulting *superseding graph*. In this graph, there is a vertex for every NN-candidate (hence, D is absent in the graph). The edge from C to A indicates that C supersedes A. The other edges follow the same semantics. Clearly, B is the best object, as it supersedes both A and C. We say that B is a *superseding nearest neighbor* (SNN) of q, and return it to the user.

In Figure 2, an object (i.e., B) supersedes all other NN-candidates. Such an "all-game winner", however, does not always exist, namely, every object may lose in at least one pairwise competition. Figure 3a presents another example with six uncertain objects A, B, ..., F. Every object has three instances, each with probability $1/3$ (e.g., A may be located at $A[1]$, $A[2]$, or $A[3]$ with equal chance). Figure 3b presents the resulting superseding graph. Clearly, no object supersedes the other NN-candidates. Furthermore, unlike its counterpart on precise points, the superseding relationship does not obey transitivity on uncertain objects, as is



(a) Data and query
(b) Distance pdf of A
(c) Distance pdf of B
(d) Distance pdf of C

Figure 1: A nearest neighbor query on uncertain data

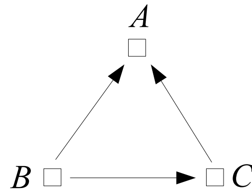
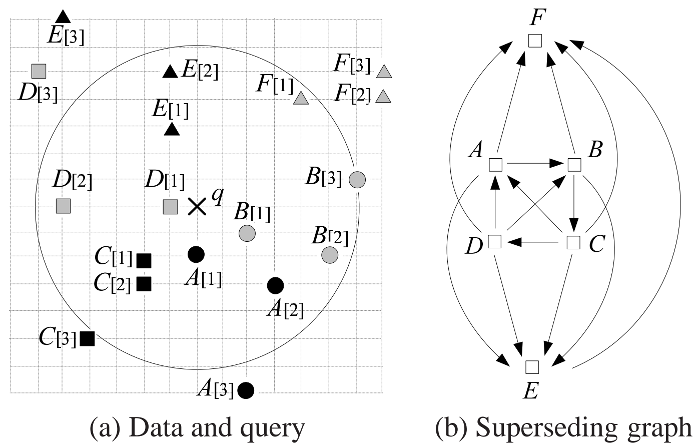


Figure 2: The superseding graph for Figure 1



(a) Data and query
(b) Superseding graph

Figure 3: Illustration of an SNN-core

obvious in the cycle $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$.

To remedy this problem, we propose to return the *SNN-core*, i.e., the smallest set of NN-candidates *each of which* supersedes *all* the NN-candidates outside the core. In Figure 3b, the SNN-core has four objects: A, B, C, D (it is not worth considering E and F, as they are worse than *all* the objects in the SNN-core). We present a systematic study on the problem of SNN-core computation. First, we formalize this new concept, and identify its interesting properties. In particular, we show that the SNN-core is *always unique*, thus eliminating the question of "which SNN-core would be better if there were several". Our second contribution is a set of algorithms for finding SNN-cores. These algorithms utilize a conventional R-tree (commonly available in commercial DBMS) to drastically prune the search space to achieve I/O efficiency.

SNN search can be applied in any application where it makes sense to issue NN queries on uncertain objects. NN retrieval on uncertain objects does not have a unique, clear, answer, which has motivated the development of several definitions of "nearest neighbor" in this context (see Section 2.2). All of these definitions are complement to each other because (i) each of them is reasonable under a certain interpretation of what is a "good" NN, and (ii) no definition subsumes the others, i.e., the "NN" found by one definition can be a poor one by another. SNN can be regarded as another way to define good NNs, but with several nice features of its own. In particular, this is the first definition that is based on *mutual superiority*. Note that disregarding mutual superiority may cause disputes on fairness in practice. For example, consider each object to be a cab, and the goal of NN search is to recommend a cab to a customer. Say cab A is returned; then, the driver of cab B may complain about loss of business if B actually has higher probability (than A) to be closer to the customer.

The rest of the paper is organized as follows. Section 2 reviews the previous work that is directly related to ours. Section 3 formally defines the problem of SNN retrieval and illustrates its characteristics. Section 4 develops an algorithm that computes the SNN-core based on a complete superseding graph. Section 5 proposes a faster algorithm that is able to produce the SNN-core without deriving the whole superseding graph. Section 6 settles some extensional issues. Section 7 experimentally evaluates our solutions. Finally, Section 8 concludes the paper with directions for future work.

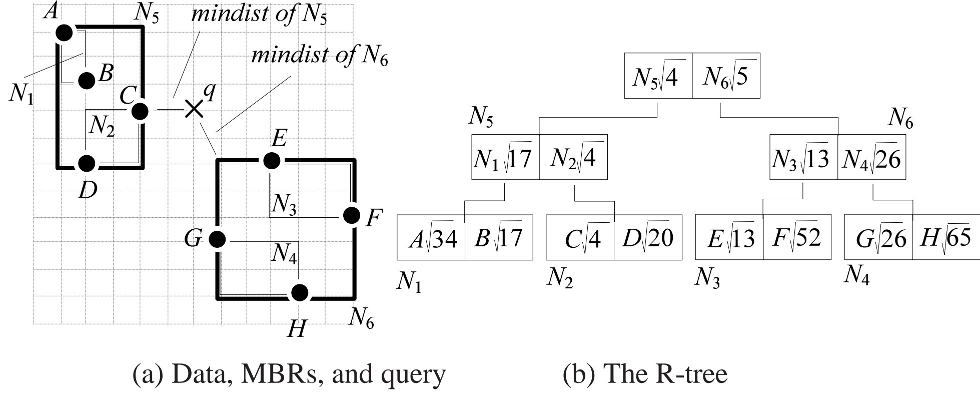
2 Related Work

In Section 2.1, we discuss the existing research about NN retrieval on precise data (i.e., no uncertainty). Then, Section 2.2 surveys the NN solutions on uncertain objects.

2.1 Nearest Neighbor Search on Precise Data

NN retrieval has been extensively studied in databases, computational geometry, machine learning, etc. In the sequel, we focus on the most important results in the database literature, paying particular attention to the *best-first* (BF) algorithm, since it is employed in our technique.

Best-first. BF, developed by Hjaltason and Samet [8], assumes an R-tree [1] on the underlying dataset. We will explain the algorithm using the dataset of 8 points A, B, ..., H in Figure 4a, and the R-tree in Figure 4b. The rectangles in Figure 4a demonstrate the minimum bounding rectangles (MBR) of the nodes in the R-tree (e.g., rectangle N_1 denotes the MBR of the leaf node enclosing A and B). A concept crucial in BF is the *minimum distance* (mindist) from an MBR to the query point q. For example, in Figure 4a (where the query is the cross), the mindist of N_5 is the length 2 of the segment between q and C, while the mindist of N_6 is the length $\sqrt{5}$ of the segment between q and the upper-left corner of N_6. For convenience, in Figure 4b, we associate each entry in the tree with its mindist to q. Specially, for a leaf entry, the mindist is simply the



(a) Data, MBRs, and query (b) The R-tree

Figure 4: NN search with an R-tree

distance from the corresponding data point to q.

BF uses a min-heap $\mathcal{H}$ to manage the (intermediate/leaf) entries that have been seen so far but not yet processed. The sorting keys of the entries are their mindists. Initially, $\mathcal{H}$ includes only the root entries: $\mathcal{H} = \{N_5, N_6\}$. As N_5 tops the heap, it is de-heaped; accordingly, node N_5 is accessed, and its entries N_1 and N_2 are added to the heap: $\mathcal{H} = \{N_2, N_6, N_1\}$. Similarly, next BF visits node N_2, and $\mathcal{H}$ becomes $\{C, N_6, N_1, D\}$. Now, the top of $\mathcal{H}$ is a data point C, which is guaranteed to be the NN.

BF is *incremental*, meaning that, if allowed to run continuously, it will output the data points in ascending order of their distances to q. For example, after reporting C, BF can be used to find the 2nd NN by continuing on the current $\mathcal{H} = \{N_6, N_1, D\}$ in the same way. Specifically, the next node accessed is N_6, changing $\mathcal{H}$ to $\{N_3, N_1, D, N_4\}$, and still the next is N_3, leading to $\mathcal{H} = \{E, N_1, D, N_4, F\}$. Now a data point E tops $\mathcal{H}$, and it is the 2nd NN. It can be shown that BF is *optimal* in the sense that it requires the fewest I/O accesses to find any number of NNs, among all the algorithms using the same R-tree.

Other Works. Roussopoulos et al. [15] propose another NN algorithm that performs depth-first search on an R-tree. This algorithm requires less memory than BF, but may need to access more nodes. Solutions based on R-trees, however, have poor performance in high-dimensional spaces [19], because the structure of the R-tree deteriorates significantly as the dimensionality increases. This observation leads to several algorithms specifically designed for high-dimensional NN search (see [10] and the references therein). The above solutions assume that the distance between two objects can be calculated quickly, whereas Seidl and Kriegel [16] consider the case where distance evaluation is expensive. Finally, it is worth mentioning that NN retrieval has numerous variations such as *reverse NN search* [11], *aggregate NN search* [13], *continuous NN search* [18], etc.

2.2 NN Search on Uncertain Data

Let us represent the pdf of an uncertain object o as $o.pdf(.)$, such that $o.pdf(x)$ gives the possibility that o is located at location x. Specially, $o.pdf(x) = 0$, if o cannot appear at x.

Expected-distance Principle. Given an NN query point q, a naive approach is to return the object with the smallest *expected distance* to q. However, the expected distance is not a reliable indicator of the quality of an object. To understand this, consider Figure 5, which shows the possible instances of objects A, B, and C. Intuitively, A is the best object, because it is almost sure (i.e., with 99% probability) to be the NN of q. However, A has a large expected distance, because its instance $A[2]$ is faraway from q. In fact, without

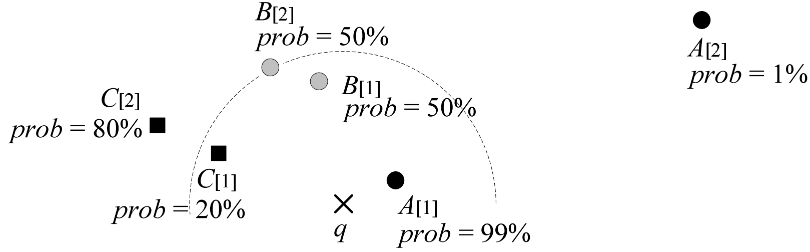


Figure 5: NN probability calculation

affecting the NN-probability of A, we can *arbitrarily* increase the expected distance of A, by pushing $A[2]$ sufficiently away from q.

PR-principle. A better approach is to report the object with the largest NN-probability [2]. Formally, the NN-probability $P_{NN}(o)$ of an object o is given by [5]:

$$P_{NN}(o) = \int_x o.pdf(x) \cdot P_{NN}(o|x) \, \mathrm{d}x \tag{1}$$

where $P_{NN}(o|x)$ is the probability of o being the NN on condition that it is located at x. Alternatively, $P_{NN}(o|x)$ is the likelihood that all other objects (i.e., except o) fall outside the circle centering at q with radius $dist(x, q)$ (i.e., the distance between x and q). For example, let o be object B in Figure 5 and x be its instance $B[2]$. Then, $P_{NN}(B|B[2])$ is the probability 0.8% that both A and C lie outside the dotted circle, i.e., A and C at $A[2]$ and $C[2]$, respectively. In general, NN-probabilities can be costly to calculate; this problem is recently relieved by Kriegel et al. [12] with a clustering approach. The work of [12] also addresses the case where the query location is uncertain. Reynold et al. [5] consider the related problem of retrieving all points whose NN-probabilities are at least a certain threshold (as opposed to reporting only the few objects with the greatest NN-probabilities). An approximate version of the problem, called *probabilistic verifier*, has recently been studied in [4].

The PR-principle is a reasonable way to define the results of NN queries on uncertain data. A common criticism is that sometimes even the highest NN probability can be quite low, and multiple objects may have almost the same NN probabilities. In any case, the PR-principle is orthogonal to our SNN approach. As will be shown in the experiments, for most queries, the SNN-core contains only a single object that is *not* the object with the greatest NN-probability. In practice, the PR-principle and our SNN method are nice complements to each other. First, they provide two interesting options to a user, each with its unique features. Second, they can even be combined to provide more reliable results. For example, a user may want to find objects that (i) are in the SNN-core, and (ii) their NN-probabilities are among the top-t in the dataset, where t is a user parameter.

Other Works. Dai et al. [7] address a different version of uncertain NN search. Specifically, they assume *existentially* uncertain objects. Namely, an object may not belong to the database, but in case it does, its location is precise. In our context, an object definitely exists, but its location is uncertain. The solution of [7] is specific to its settings, and cannot be adapted to our problem.

An NN query can be regarded as an instance of top-1 search. If we define the *score* of an object o as its distance to the query point q, then the goal is to find the top-1 object with the lowest score. This creates the opportunity of applying top-k methods to NN queries. Several top-k algorithms [9, 17, 20] have been proposed for uncertain data. At $k = 1$, they extract the object that has the smallest score with the largest probability. In other words, they advocate the same result as the PR-principle.

3 Problem Definitions and Basic Characteristics

Let $\mathcal{D}$ be a set of uncertain objects. Following the previous work [9, 12, 17, 20], we consider the *discrete pdf model*. To simplify analysis, we first consider that (i) each object o is associated with s points $o[1]$, $o[2]$, ..., $o[s]$, called the *instances*, and (ii) o may appear at any of its instances with an identical probability $1/s$. Equivalently, the pdf of o is given by:

$$o.pdf(x) = \begin{cases} 1/s & \text{if } x = \text{any of } o[1], ..., o[s] \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

In Section 6, we will discuss the general discrete pdf model, where each object can have different numbers of instances, and each instance can be associated with a different probability. Furthermore, the discussion will also be extended to continuous pdfs.

Use q to denote a precise query point. Let $maxdist(o, q)$ be the largest distance between q and all the instances of o, or formally:

$$maxdist(o, q) = \max_{i=1}^{s}\{dist(o[i], q)\}. \tag{3}$$

Let $minmax$ be the smallest $maxdist(o, q)$ of all objects $o \in \mathcal{D}$:

$$minmax = \min_{\forall o \in \mathcal{D}}\{maxdist(o, q)\}. \tag{4}$$

The value of $minmax$ allows us to easily identify those objects with NN-probabilities (calculated by Equation 1) that are larger than 0:

Lemma 1 (NN-candidate)**.** *An object o is an* NN-candidate *of q if at least one of its instances is within distance $minmax$ to q, that is,* $\min_{i=1}^{s}\{dist(o[i], q)\} < minmax$.

Proof. Obvious and omitted. □

We give the name *minmax-circle* to the circle centering at q with radius $minmax$. By Lemma 1, an object is an NN-candidate if and only if it has at least one instance in the minmax-circle. We use $\mathcal{N}$ to represent the set of all NN-candidates, and impose an ordering on $\mathcal{N}$:

Definition 1 (RI-list)**.** *The* ranked instance list *(RI-list) sorts the instances of all objects in $\mathcal{N}$ in ascending order of their distances to q (breaking ties randomly).* □

For example, the dataset $\mathcal{D}$ in Figure 3a has six objects A, ..., F, each of which has $s = 3$ instances. Consider, for instance, object B; its $maxdist(B, q)$ equals the distance between q and instance $B[3]$. Figure 3a also shows the circle centering at q with radius $maxdist(B, q)$. Clearly, except B, no object has all the instances inside the circle, indicating $minmax = maxdist(B, q)$. Hence, the circle is the minmax-circle. Furthermore, every object has at least one instance in the circle, and hence, is an NN-candidate. In other words, $\mathcal{N} = \{A, B, C, D, E, F\}$. The RI-list ranks the instances of all NN-candidates in ascending order of their distances to q, namely:

$$\{D[1], A[1], B[1], C[1], E[1], C[2], A[2], D[2], E[2],$$
$$B[2], F[1], B[3], C[3].D[3], A[3], F[2], F[3], E[3]\}.$$

Given two objects o and o', we use $o \prec o'$ to denote the event that o is closer to q than o'. The probability $P\{o \prec o'\}$ that this event occurs depends on the distribution of the instances of o and o'. Formally,

$$P\{o \prec o'\} = \sum_{i=1}^{s} o.pdf(o[i]) \cdot P\{o \prec o' | o = o[i]\}. \tag{5}$$

where $P\{o \prec o' | o = o[i]\}$ is the probability of o being closer to q than o' provided that o is located at instance $o[i]$. In fact, $P\{o \prec o' | o = o[i]\}$ is essentially the percentage of the instances of o' that rank after $o[i]$ in the RI-list. To illustrate, assume that o and o' are objects B and D in Figure 3a respectively, and $o[i]$ is $B[3]$. In the RI-list, D has only one instance $D[3]$ after $B[3]$. Hence, $P\{B \prec D | B = B[3]\}$ equals 1/3 (recall that D has 3 instances in total).

Obviously, $P\{o \prec o'\} + P\{o' \prec o\} = 1$. Therefore, whether o or o' is preferred by q is determined by the relationship of $P\{o \prec o'\}$ and 0.5:

Definition 2 (Superseding Relationship)**.** *Given two objects o and o', o is said to* supersede o' *if $P\{o \prec o'\} > 0.5$. In case $P\{o \prec o'\} = 0.5$, the superseding relationship between o and o' is randomly decided.* □

This leads to the superseding graph:

Definition 3 (Superseding Graph)**.** *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the* superseding graph*, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ the set of edges. $\mathcal{G}$ is a directed graph that has a vertex for each NN-candidate, i.e., $\mathcal{V} = \mathcal{N}$. For any two objects o and o' in $\mathcal{N}$, if o supersedes o', $\mathcal{E}$ has an edge from o to o'; otherwise, $\mathcal{E}$ has an edge from o' to o.* □

Figure 3b gives the superseding graph for the example of Figure 3a, summarizing the superseding relationships between all pairs of NN-candidates. Since each vertex in a superseding graph represents an object, in the sequel the terms "vertex" and "object" will be used interchangeably. Now we are ready to introduce the SNN-core.

Definition 4 (SNN-core)**.** *Given a superseding graph $\mathcal{G}$, the* SNN-core *is a set $\mathcal{S}$ of vertices in $\mathcal{G}$ satisfying two conditions:*

- *(*superseding requirement*) every object in $\mathcal{S}$ supersedes all objects in $\mathcal{V} - \mathcal{S}$;*
- *(*minimality requirement*) no proper subset of $\mathcal{S}$ fulfills the previous condition.*

Each object in $\mathcal{S}$ is called a superseding nearest neighbor *(SNN) of q.* □

Consider again Figure 3b. The SNN-core is $\mathcal{S} = \{A, B, C, D\}$. Indeed, each object in $\mathcal{S}$ supersedes all the NN-candidates (E and F) outside $\mathcal{S}$. Moreover, $\mathcal{S}$ is minimal, because if any object is deleted from $\mathcal{S}$, the remaining $\mathcal{S}$ no longer satisfies the superseding requirement.

Properties. The following lemma shows the uniqueness of SNN-core.

Lemma 2. *A superseding graph has exactly one SNN-core.*

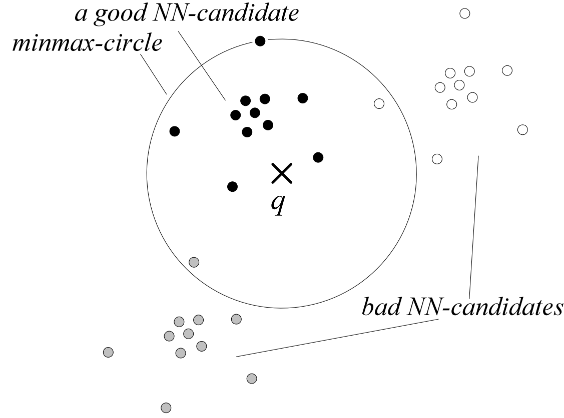


Figure 6: Intuition behind the effectiveness of SNN-core

Proof. Let $\mathcal{V}$ be the vertex set of the superseding graph $\mathcal{G}$. Assume, on the contrary, that there are two SNN-cores $\mathcal{S}_1$ and $\mathcal{S}_2$. Obviously, neither can be a subset of the other, due to the minimality requirement of SNN-core. Hence, there exists an object o such that $o \in \mathcal{S}_1$ yet $o \notin \mathcal{S}_2$, and an object o' such that $o' \notin \mathcal{S}_1$ yet $o' \in \mathcal{S}_2$. Since $\mathcal{S}_1$ is an SNN-core, o should supersede any object in $\mathcal{V} - \mathcal{S}_1$. Thus, o supersedes o'. Similarly, as $\mathcal{S}_2$ is an SNN-core, o' needs to supersede o. This is impossible because two objects cannot supersede each other. $\square$

In practice, the size of the SNN-core is much smaller than the number of NN-candidates. To understand why, first note that intuitively an NN-candidate is preferred by the customer if most of its instances are close to the query point q. *However, even a good NN-candidate may have an instance relatively far from* q. *Conversely, even a bad NN-candidate may have an instance close to* q. To illustrate, Figure 6 shows three uncertain objects whose instances are colored in black, grey, and white respectively. Note that the instance distribution of each object simulates a popular Gaussian modeling [3]. The black object is a good NN-candidate, and the only one in the SNN-core. Notice that the other two objects become NN-candidates by having very few instances (here, only one) inside the minmax-circle. For a query on a real dataset, there are often many such bad NN-candidates, explaining why usually a majority of the NN-candidates have extremely small NN-probabilities. It is rather unlikely for these NN-candidates to enter the SNN-core.

Problem. Our objective is to compute the SNN-core with the minimum computational cost. In the subsequent sections, we present two algorithms to solve this problem.

4 The Full-Graph Approach

Our first algorithm, *full-graph*, finds the SNN-core in two steps: (i) first compute the entire superseding graph $\mathcal{G}$, and then (ii) derive the SNN-core from $\mathcal{G}$. Next, we elaborate the details of these steps.

First Step. The goal is to obtain all the s instances of every NN-candidate. $\mathcal{G}$ is easily determined once these instances are ready (the superseding relationship between any NN-candidates o, o' can be resolved by evaluating $P\{o \prec o'\}$ according to Equation 5). To facilitate instance retrieval, we deploy an R-tree to index the instances of all objects. Given a query point q, we invoke the best-first (BF) algorithm [8] reviewed in Section 2.1 to extract instances in ascending order of their distances to q. We terminate BF as soon as s instances of each NN-candidate have been fetched.

The only question is how to determine which objects are NN-candidates. For this purpose, it suffices to count the number of instances of each object seen so far. The set $\mathcal{N}$ of NN-candidates is determined when the count of any object reaches s (i.e., the number of instances per object) for the first time — $\mathcal{N}$ includes all the objects, of which at least one instance has been encountered before this moment. We refer to this moment as the *minmax-moment*, because this is the time when the value of $minmax$ (Equation 4) is finalized.

Note that after the minmax-moment, BF still needs to continue (until it has acquired all the instances of the objects in $\mathcal{N}$), but the instances of any object outside $\mathcal{N}$ can be ignored. It is worth mentioning that this phase essentially extracts the entire RI-list (Definition 1) in its sorted order.

Second Step. We proceed to explain how to discover the SNN-core from a superseding graph $\mathcal{G}$. The starting point is to identify a vertex of $\mathcal{G}$ that definitely belongs to SNN-core. Referring to the number of incoming edges at a vertex as its *in-degree*, we have:

Lemma 3. *The vertex with the lowest in-degree must be in the SNN-core.*

Proof. Assume that o is a vertex with the lowest in-degree x in $\mathcal{G}$ but o is not in the SNN-core. Consider any vertex o' in the SNN-core, which by definition must supersede o. As the in-degree of o' is at least x, o' is superseded by at least x vertices, all of which must be in the SNN-core (see Lemma 4), and hence, supersede o. Thus, we have found $x + 1$ objects superseding o, contradicting the definition of x. □

In general, it is possible to have multiple vertices in $\mathcal{G}$ having the lowest in-degree. In this case, all of them are in the SNN-core. Once we have identified a vertex that belongs to the SNN-core, we immediately know some others that must also be in the SNN-core, as indicated in the next lemma:

Lemma 4. *If a vertex o is in the SNN-core, then all the vertices superseding o must also belong to the SNN-core.*

Proof. This is obvious, because otherwise it contradicts the fact that every object in the SNN-core supersedes any object outside the SNN-core. □

The two lemmas motivate the following strategy to retrieve the SNN-core from a superseding graph $\mathcal{G}$. First, we add to the core the vertices that have the minimum in-degree in $\mathcal{G}$. Then, given a new vertex o in the SNN-core, we also include all the objects o' superseding o, if o' is not already there. This is repeated until no more vertex can be inserted in the SNN-core.

Example 1. We illustrate the strategy using the superseding graph in Figure 3b. Objects C and D have the lowest in-degree 1, and hence, are added to the SNN-core $\mathcal{S}$. Since C is in the core and B supersedes C, by Lemma 4, B also belongs to the core, and is thus added to $\mathcal{S}$ (which is now $\{B, C, D\}$). Similarly, the incorporation of B in the core further leads to the inclusion of A (because A supersedes B). This results in the final SNN-core $\mathcal{S} = \{A, B, C, D\}$. □

Full-graph does not retrieve any object with no instance in the minmax-circle. This is a property of the BF algorithm [8].

5 The Pipeline Approach

The *full-graph* algorithm can be inefficient. Consider the extreme case where every NN candidate has a tiny probability (e.g. 0.001%) of being very faraway from the query q. Since *full-graph* needs to acquire all instances of every NN candidate, it may end up examining the whole database. Intuitively, if most instances of an object are close to q, we should be able to determine that the object is in the SNN-core without retrieving its farway instances. Similarly, it may also be possible to decide the SNN-core without generating the whole superseding graph $\mathcal{G}$.

Motivated by this, next we present the *pipeline* algorithm that entails lower I/O cost. *Pipeline* incrementally retrieves object instances in ascending order of their distances to q, calculates the superseding edges and prunes objects *during* the retrieval, and terminates without fetching all the instances of NN candidates or the complete $\mathcal{G}$. Implementation of this idea has several challenges. First, when to stop retrieving instances of objects? Second, how to determine the SNN-core from a partial $\mathcal{G}$? We will answer these questions in this section.

5.1 The Algorithm

As with *full-graph*, *pipeline* also utilizes the best-first (BF) algorithm to retrieve the RI-list (Definition 1) in its sorted order. While unfolding the RI-list gradually, *pipeline* maintains a *conservative core* $\mathcal{G}^*$. Intuitively, $\mathcal{G}^*$ captures the portion of the final superseding graph $\mathcal{G}$ that has been revealed so far, and is relevant to the SNN-core.

At the beginning of *pipeline*, $\mathcal{G}^*$ has only a single vertex, called *unseen*. This is a special vertex, whose presence means that the SNN-core may contain an object that has not been encountered yet. As soon as we can assert that no such object can exist, *unseen* is removed. Except *unseen*, every other vertex in $\mathcal{G}^*$ represents a seen object that *may* be in the SNN-core. Once we can disqualify an object, we mark it as *pruned*, and delete its vertex from $\mathcal{G}^*$.

Specifically, *pipeline* executes in iterations, where each iteration includes four phases:

1. (*Vertex phase*) Get the next instance in the RI-list (using the BF algorithm). Let o be the object that owns the instance. Add a vertex o to $\mathcal{G}^*$ if (i) o is not already in $\mathcal{G}^*$, (ii) o is not marked as *pruned*, and (iii) and vertex *unseen* is still in $\mathcal{G}^*$.

   If this is the minmax-moment (having seen all the s instances of an object for the first time, as defined in Section 4), remove vertex *unseen* from $\mathcal{G}^*$ (at least one instance of every NN-candidate has been encountered).

   It is worth mentioning that *pipeline* may finish before reaching the minmax-moment. Vertex *unseen* may also be deleted from $\mathcal{G}^*$ before this moment, as explained shortly.

2. (*Edge phase*) Decide as many edges in $\mathcal{G}^*$ as possible from the instances examined so far.

3. (*Pruning phase*) If an object is disqualified from the SNN-core, discard its vertex in $\mathcal{G}^*$ and mark the object as *pruned*. If we can assert that no unseen object can be in the SNN-core, discard vertex *unseen*.

4. (*Validating phase*) If we can conclude that the vertices in the current $\mathcal{G}^*$ constitute the SNN-core, terminate the algorithm.

Figure 7 presents the pesudocode of *pipeline*. In the sequel, we demonstrate *pipeline* by using it to compute the SNN-core for the example of Figure 3a, where each object has $s = 3$ instances.

Algorithm *pipeline*

/* the code assumes that the elements of the RI-list can be retrieved in ascending order of their distances to the query point */

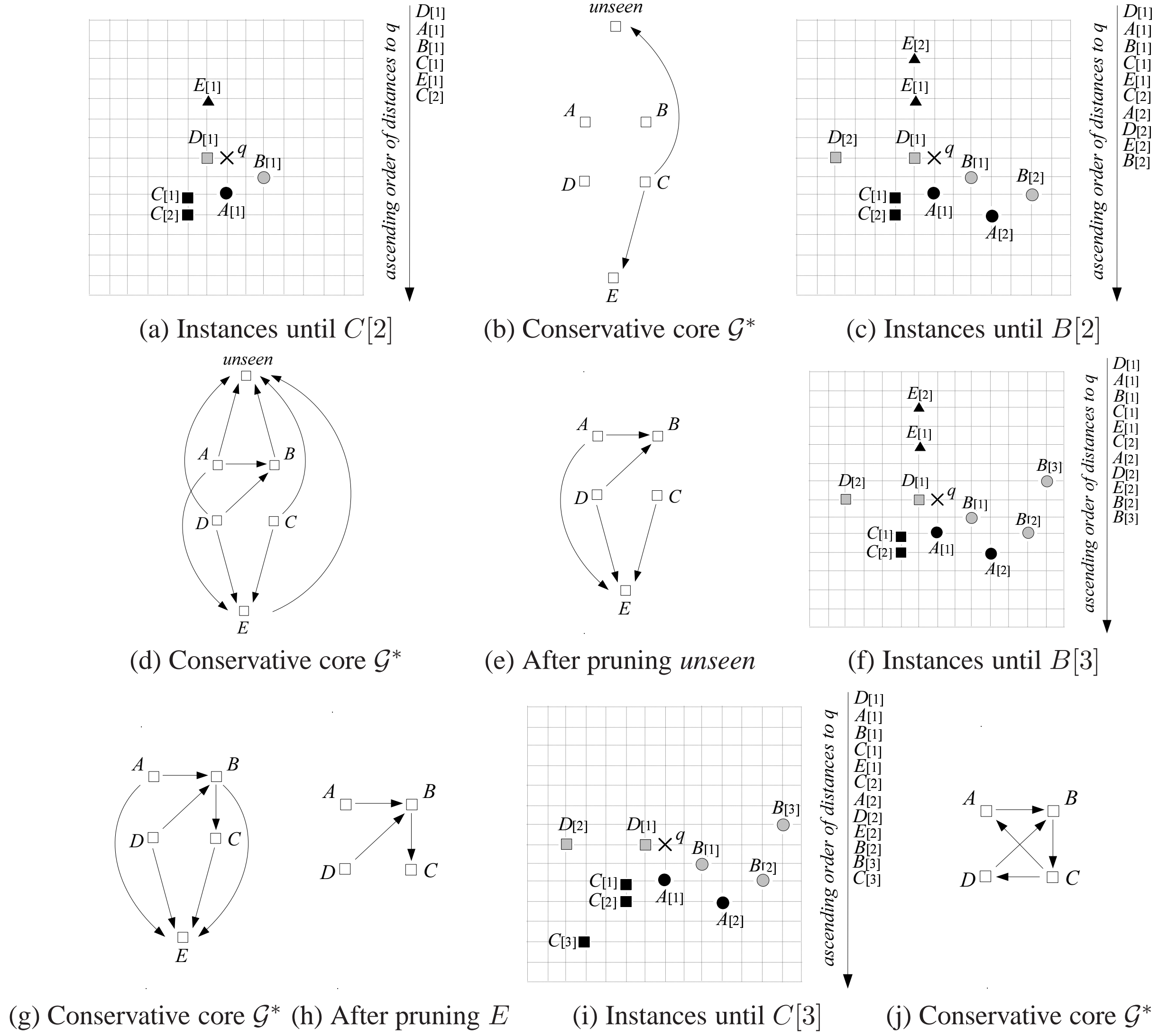
```
1.  G* is a graph with only a single vertex, named unseen, with no edge
2.  while the RI-list has not been exhausted
        /* vertex phase */
3.      get the next instance of the RI-list, which, say, belongs to object o
4.      if (G* does not have a vertex for o) and (o is not marked as pruned) and
        (vertex unseen is still in G*)
5.          add a vertex o to G*
6.      if all the instances of o have been seen /* namely, the minmax-moment */
7.          remove vertex unseen from G*
        /* edge phase */
8.      add as many edges to G* as possible
        /* pruning phase */
9.      for each vertex o in G*
10.         if o can be pruned
11.             remove o from G* and all of its incident edges
12.             if o corresponds to an object then mark o as pruned
                /* otherwise, o is vertex unseen */
        /* validating phase */
13.     if all the vertices of G* must be in the SNN-core
14.         return the set of vertices
```

Figure 7: The *pipeline* algorithm

Example 2. As mentioned earlier, *pipeline* invokes the BF algorithm to retrieve object instances in ascending order of their distances to the query point q. Consider the moment after seeing $C[2]$. Figure 8a shows all the instances already accessed, and Figure 8b presents the current conservative core $\mathcal{G}^*$. $\mathcal{G}^*$ has five "regular" vertices A, B, ..., E, created by the vertex phase when the first instances of those objects were fetched. $\mathcal{G}^*$ also has the vertex *unseen*, implying that (based on the instances obtained so far) the SNN-core may involve other objects that have not been encountered.

The edge phase of this iteration adds two edges to $\mathcal{G}^*$ as shown in Figure 8b, corresponding to two superseding relationships that can be determined. For example, C must supersede E, even though we have only obtained their instances $C[1]$, $E[1]$, $C[2]$ (in this order). This is because the event $C \prec E$ (i.e., C is closer to the query point q than E) definitely occurs when (i) C is at $C[1]$ (happening with probability 1/3), or (ii) C is at $C[2]$ (probability 1/3) and E is not at $E[1]$ (probability 2/3). Hence, the probability $P\{C \prec E\}$ of event $C \prec E$ is *at least* $\frac{1}{3} + \frac{1}{3} \cdot \frac{2}{3} = \frac{5}{9}$. As this *lower bound* is already greater than 50%, it is safe to conclude that C supersedes E. Similarly, C also supersedes any unseen NN-candidate o. Specifically, as all instances of o are farther to q than both $C[1]$ and $C[2]$, the event $C \prec o$ occurs with probability at least $\frac{2}{3} > 50\%$.

Pipeline continues to examine object instances, and meanwhile, generates more edges of $\mathcal{G}^*$ in the way explained earlier. Its pruning phase has no effect until after obtaining $B[2]$. Figure 8c presents the instances fetched up to $B[2]$, and Figure 8d gives the current $\mathcal{G}^*$. At this point, *pipeline* claims that no unseen object can possibly appear in the SNN-core. To understand this, notice that in Figure 8d, A, B, ..., E all supersede the vertex *unseen* (representing any object not encountered so far). Thus, the SNN-core must necessarily be



(a) Instances until $C[2]$ (b) Conservative core $\mathcal{G}^*$ (c) Instances until $B[2]$

(d) Conservative core $\mathcal{G}^*$ (e) After pruning *unseen* (f) Instances until $B[3]$

(g) Conservative core $\mathcal{G}^*$ (h) After pruning E (i) Instances until $C[3]$ (j) Conservative core $\mathcal{G}^*$

Figure 8: Rationale of *pipeline*

a subset of $\{A, B, C, D, E\}$ due to the minimality requirement of SNN-core (see Definition 4). Accordingly, we discard vertex *unseen* from $\mathcal{G}^*$, resulting in the new $\mathcal{G}^*$ in Figure 8e. From now on, we can ignore the instances of any object other than A, B, ..., E.

The next pruning happens after retrieving $B[3]$. Figure 8f shows the instances fetched so far and Figure 8g gives the current $\mathcal{G}^*$. Clearly, A, B, C, D all supersede E, which, therefore, cannot be in the SNN-core (again, due to the minimality requirement). Hence, the pruning phase further simplifies $\mathcal{G}^*$ into Figure 8h, and marks E as *pruned*.

Pipeline finishes after retrieving one more instance $C[3]$. Figure 8i presents the instances (of the objects still in $\mathcal{G}^*$) visited so far, and Figure 8j illustrates the corresponding conservative core $\mathcal{G}^*$. It is easy to verify that the SNN-core cannot be any smaller than $\{A, B, C, D\}$ due to the loops $A \rightarrow B \rightarrow C \rightarrow A$ and $B \rightarrow C \rightarrow D \rightarrow B$ in $\mathcal{G}^*$. Therefore, the validation phase terminates the algorithm, and the remaining instances of the RI-list are not visited. Notice that, the validation does not require the edge between A and D.

$\square$

The above discussion explains the high-level framework of *pipeline*. In the sequel, we will elaborate the details of the edge, pruning, and validation phases (the vertex phase is simple and has been clarified earlier).

5.2 Edge Phase

The goal of this phase is to determine as many edges in the conservative core $\mathcal{G}^*$ as possible, based on the instances already examined. Let $\mathcal{V}^*$ be the set of vertices in $\mathcal{G}^*$. Denote $m = |\mathcal{V}^*|$, and the vertices in $\mathcal{V}^*$ as o_1, o_2, ..., o_m.

Recall that each vertex in $\mathcal{G}^*$ corresponds to an object. The only exception is the special vertex *unseen*, which may not exist in $\mathcal{G}^*$, but in case it does, it represents any object that has not been encountered. Due to the special nature of *unseen*, we give double semantics to o_1. Specifically, in case *unseen* appears in $\mathcal{G}^*$, it is denoted as o_1. Otherwise, o_1 is a regular vertex representing an object. The other vertices o_2, ..., o_m, on the other hand, are always regular.

The edge phase maintains a two-dimensional *lower bound matrix* low with m rows and m columns. The entry $low[i, j]$ ($1 \leq i, j \leq m$) at the i-th row and j-th column stores a lower bound of the probability $P\{o_i \prec o_j\}$ that object o_i is closer to the query point q than object o_j. There are some special cases:

- For $i = j$, $low[i, j]$ is always 0.
- In case o_1 is vertex *unseen*, (i) $low[1, i] = 0$ for $2 \leq i \leq m$, and (ii) $low[i, 1]$ ($2 \leq i \leq m$) is a lower bound of the probability that o_i is closer to q than any object that has not been encountered.

At the beginning of *pipeline*, $\mathcal{V}^*$ has a single vertex *unseen*. Hence, $m = 1$ and low has only one cell $low[1, 1] = 0$. As *pipeline* executes, the size of low varies with $\mathcal{V}^*$. Whenever $low[i, j]$ ($1 \leq i, j \leq m$) exceeds 0.5, we add to $\mathcal{G}^*$ an edge from o_i to o_j.

The edge phase also maintains a one-dimensional array cnt with size m. The i-th element $cnt[i]$ ($1 \leq i \leq m$) records how many instances of object o_i have been retrieved so far. In case o_1 is the vertex *unseen*, $cnt[1] = 0$.

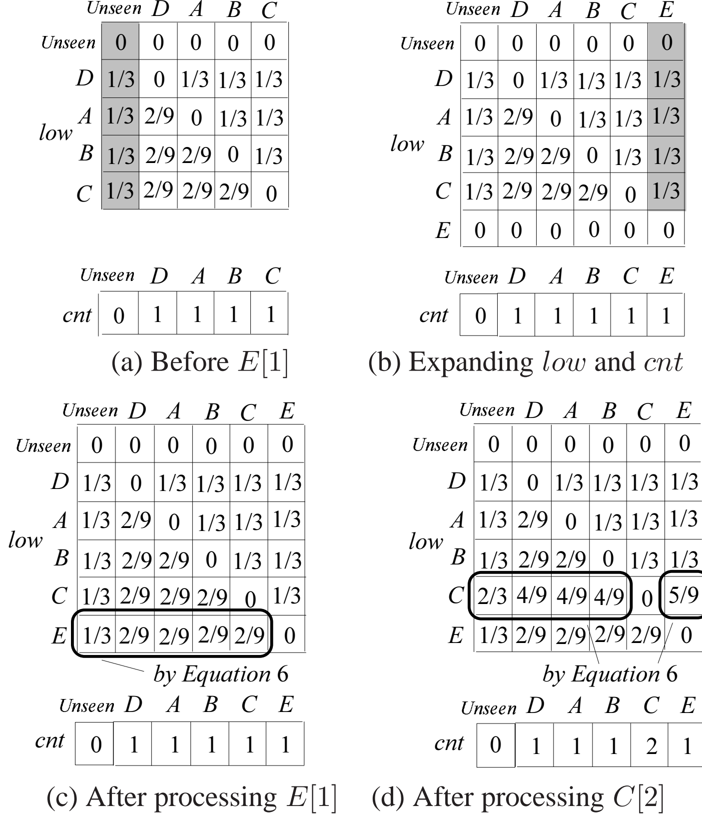
In each iteration, the edge phase updates array low based on cnt and the instance acquired from the vertex phase. Let o_i (for some $i \in [1, m]$) be the object that owns the instance. The edge phase increases $cnt[i]$ by 1, and updates $m - 1$ entries in low, i.e.:

$$low[i, 1], ..., low[i, i - 1], low[i, i + 1], ..., low[i, m].$$

Specifically, $low[i, j]$ ($j = 1, ..., i - 1, i + 1, ..., m$) is updated as:

$$low[i, j] = low[i, j] + \frac{1}{s}\left(1 - \frac{cnt[j]}{s}\right). \tag{6}$$

It remains to explain what happens when a vertex is inserted or deleted in $\mathcal{G}^*$. Insertion may happen in the vertex phase, if an instance of a new object is obtained and vertex *unseen* is still in $\mathcal{G}^*$. Let o_{m+1} be the new object. We expand cnt with a new element $cnt[m + 1] = 0$. As for low, we first add an $(m + 1)$-st column to low, and copy the value of $low[i, 1]$ to $low[i, m + 1]$ for each $i \in [1, m]$. Then, we add an $(m + 1)$-st row to low with all zeros. Deletion, on the other hand, may occur in the vertex phase (i.e., removing the *unseen* vertex at the minmax-moment) or pruning phase (after asserting that a vertex cannot be in the SNN-core). Deleting a vertex involves only discarding its entry in cnt, and the corresponding row and column in low.



(a) Before $E[1]$ (b) Expanding low and cnt

(c) After processing $E[1]$ (d) After processing $C[2]$

Figure 9: Illustration of the edge phase

Example 3. We demonstrate the maintenance of matrix low using the instances in Figure 8a. Figure 9a shows the contents of arrays low and cnt right before the iteration of *pipeline* that retrieves instance $E[1]$. Since $E[1]$ is the first instance of E, the vertex phase adds a new vertex of E to $\mathcal{G}^*$. Accordingly, low receives a new column and a new row, as shown in Figure 9b. Note that everything already in the old low of Figure 9a is directly retained in Figure 9b. Furthermore, the grey cells in Figure 9b are copied from the first column of the old low, and the last row of the new low contains only zeros.

Then, the edge phase processes $E[1]$ by modifying the row in low corresponding to E, according to Equation 6. For example, since the old $low[E, \textit{unseen}] = 0$ in Figure 9b, $s = 3$, and $cnt[\textit{unseen}] = 0$, the new $low[E, \textit{unseen}]$ is computed by $0 + \frac{1}{3}(1 - 0/3) = \frac{1}{3}$. Similarly, given the old $low[E, D] = 0$ and $cnt[D] = 1$, $low[E, D]$ is derived as $0 + \frac{1}{3}(1 - 1/3) = \frac{2}{9}$.

In the next iteration, *pipeline* fetches $C[2]$. No new vertex is spawned in $\mathcal{G}^*$ as this is the second instance of C seen. The edge phase updates low to the one in Figure 9d. Now that both $low[C, \textit{unseen}]$ and $low[C, E]$ are over 0.5, the edge phase creates the two edges in Figure 8b. □

5.3 Pruning Phase

Again, let $\mathcal{V}^*$ be the set of vertices in the conservative core $\mathcal{G}^*$. The pruning phase eliminates the vertices in $\mathcal{G}^*$ that cannot appear in the SNN-core. This is to solve the following graph problem: identify the smallest set S_{ret} of vertices such that, for any vertex $o \in S_{ret}$ and $o' \in \mathcal{V}^* - S_{ret}$, $\mathcal{G}^*$ has an edge from o to o'. We call S_{ret} the *retention-set*; all the vertices in $\mathcal{V}^* - S_{ret}$ can be discarded.

For example, given the $\mathcal{G}^*$ in Figure 9d, $S_{ret} = \{A, B, C, D, E\}$, and thus, vertex *unseen* is discarded. Similarly, if G^* is the graph in Figure 9g, $S_{ret} = \{A, B, C, D\}$, leading to the elimination of E.

The retention set S_{ret} can be computed in a way analogous to finding the SNN-core from a full superseding graph (see Section 4). Initially, S_{ret} is empty. First, we place in S_{ret} the vertices in $\mathcal{G}^*$ with the smallest in-degrees (recall that the in-degree of a vertex is the number of edges pointing at it). Whenever a vertex o appears in $\mathcal{G}^*$, we also insert all the vertices that either supersede o, or are not connected to o in $\mathcal{G}^*$. This process is repeated until no other vertex can be added.

To illustrate, let $\mathcal{G}^*$ be the graph in Figure 8g. As A has the minimum in-degree 0, it is the first vertex in S_{ret}. Since C and D do not have edges with A, they also enter S_{ret}, which becomes $\{A, C, D\}$. As B supersedes C, the inclusion of C requires adding B to S_{ret} as well. At this point, the S_{ret} is finalized as $\{A, B, C, D\}$ because no other vertex can be inserted.

5.4 Validating Phase

The validating phase determines whether the current conservative core $\mathcal{G}^*$ is *shrinkable*, namely, if there is any chance that some vertices can be pruned from $\mathcal{G}^*$ in the future, after more edges become available in $\mathcal{G}^*$. In case the answer is no, we can immediately claim that the current vertices of $\mathcal{G}^*$ constitute the SNN-core; otherwise, another iteration of *pipeline* is necessary.

Whether $\mathcal{G}^*$ is shrinkable depends on the following question: is there any non-empty subset S_{wit} of $\mathcal{V}^*$ such that all the *existing* edges of $\mathcal{G}^*$ between S_{wit} and $\mathcal{V}^* - S_{wit}$ follow the same direction? ($\mathcal{V}^*$ is the set of vertices in $\mathcal{G}^*$.) Refer to the S_{wit} as the *witness-set*. $\mathcal{G}^*$ is shrinkable if and only if a witness-set exists.

Consider the $\mathcal{G}^*$ in Figure 8e. It is shrinkable due to the witness set $\{A\}$, noticing that all the *known* edges between $S_{wit} = \{A\}$ and $\mathcal{V}^* - S_{wit} = \{B, C, D, E\}$ are from S_{wit} to $\mathcal{V}^* - S_{wit}$. In other words, it is possible for the SNN-core to be $\{A\}$ (which would be true if all the missing edges at A turned out to be pointing away from A; as shown in Figure 8j, this does not actually happen, which, however, cannot be predicted at the stage of Figure 8e). In general, there can be multiple witness sets, e.g., $\{D\}$ is another in Figure 8e. On the other hand, the $\mathcal{G}^*$ in Figure 8j has no witness set, causing the termination of *pipeline*.

To find a witness set, we borrow the concept of *strongly connected*. Specifically, a directed graph is strongly connected if, for any two vertices o and o' in the graph, there is a path from o to o', and another from o' to o. Then:

Lemma 5. *A conservative core $\mathcal{G}^*$ is not shrinkable if and only if it is strongly connected.*

Proof. The if direction: Assume, to the contrary, that a strongly connected $\mathcal{G}^*$ is shrinkable. Let V^* be the vertex set of $\mathcal{G}^*$. Since $\mathcal{G}^*$ is shrinkable, $\mathcal{V}^*$ can be devided into two groups V_1 and $V_2 = \mathcal{V}^* - V_1$ such that all the edges between the groups are from V_1 to V_2. Thus, it is not possible to travel from a vertex in V_2 to a vertex in V_1, contradicting the fact that $\mathcal{G}^*$ is strongly connected.

The only-if direction: Assume, on the contrary, that $\mathcal{G}^*$ is un-shrinkable but it is not strongly connected. $\mathcal{G}^*$ must have a pair of vertices o and o' such that there is a path from o to o' but not from o' to o. Put all the vertices which can be reached from o' in V_2 and the other vertices in $V_1 = \mathcal{V}^* - V_2$. Obviously, V_1 is not empty because it includes at least o. All the edges between V_1 and V_2 must be from V_1 to V_2 (otherwise, V_2 has not incorporated all the vertices reachable from o'). This implies that G is shrinkable (V_1 is the retention set), causing a contradiction. □

For instance, the $\mathcal{G}^*$ in Figure 8j is strongly connected. For example, let us inspect vertices A and B. Indeed, there is a path from A to B (just the edge $A \to B$), and another from B to A (i.e., $B \to C \to A$). This is true for each pair of vertices in $\mathcal{G}^*$. Checking whether $\mathcal{G}^*$ is strongly connected can be done using a standard depth-first algorithm [6].

5.5 Discussion

The I/O performance of *pipeline* is never worse than the *full-graph* algorithm in Section 4. To understand this, note that (i) the conservative core $\mathcal{G}^*$ is necessarily complete (i.e., no missing edges) after the entire RI-list has been fetched, and (ii) *pipeline* definitely terminates given a complete $\mathcal{G}^*$. As *full-graph* always extracts the whole RI-list, it incurs at least the I/O cost of *pipeline*.

On the other hand, *pipeline* may finish by examining much fewer instances than *full-graph*. As explained in Figure 6, for a practical query, many bad NN-candidates have few instances in the minmax-circle. In this case, before exploring the instances of bad candidates. *pipeline* has read most instances of the good NN-candidates, and thus, already collected enough information to determine the superseding relationships between good and bad candidates. Hence, the algorithm may prune the bad NN-candidates without reading their instances at all.

Finally, in terms of asymptotical performance, *full-graph* has smaller time complexity. In the worst case, it scans everything of the database in $O(n)$ time (treating s as a constant), produces the full superseding graph in $O(n^2)$ time, and finds the SNN-core also in $O(n^2)$ time as well, leading to an overall complexity of $O(n^2)$. *Pipeline*, on the other hand, may need to invoke a pruning phase, which is the most expensive step of an iteration, once for every single instance. This phase may require $O(n^2)$ time in the worst case, rendering an overall complexity of $O(n^3)$. Nevertheless, it is worth noting that the above complexity analysis holds only under the very pessimistic assumption that nothing can be pruned.

6 Extension

So far our analysis assumes that each object is represented by the same number s of instances, and each instance has an identical pdf-value $1/s$. This section removes this assumption. Specifically, we will modify the proposed algorithms to the scenario where each object can have any number of instances, each of which may have a different pdf-value.

Modeling and Concepts. Let $o.s$ be the number of instances needed to represent an uncertain object o, and denote those instances as $o[1]$, $o[2]$, ..., $o[o.s]$. The pdf of o may take any positive value $o.pdf(o[i])$ at each instance $o[i]$ $(1 \le i \le o.s)$, as long as $\sum_{i=1}^{o.s} o.pdf(o[i]) = 1$. Given a query point q, without loss of generality, we retain our notation convention that an instance with a smaller index is closer to q, namely, $dist(o[i], q) \le dist(o[j], q)$ for any $1 \le i < j \le o.s$.

All the definitions in Section 3 remain the same under the above modeling. The only difference lies in computing the probability $P\{o \prec o'\}$ that an object o is closer to q than another o'. Equation 5 still correctly quantifies $P\{o \prec o'\}$, but the upper limit s of the summation must be replaced by $o.s$. The calculation of $P\{o \prec o' | o = o[i]\}$ deserves detailed clarification. Consider the RI-list as formulated in Definition 1, and let x be the smallest integer such that $o'[x]$ ranks after $o[i]$ in the RI-list (in other words, $o'[x]$ is the first instance of o' behind $o[i]$). Then,

$$P\{o \prec o' | o = o[i]\} = o.pdf(o[i]) \cdot \sum_{j=x}^{o'.s} o'.pdf(o'[j]). \tag{7}$$

Indexing. We still use an R-tree to index the instances of all objects. The only difference is that, each leaf entry, in addition to keeping the coordinates of an instance, also stores the pdf-value of the instance.

Algorithms. Our first solution *full-graph* does not require any change. In *pipeline* the only modification is in the edge phase. The first change is the interpretation of the array cnt. Recall that each element $cnt[i]$ ($1 \leq i \leq m$, with m being the array size) is maintained for a vertex o_i in the current conservative core $\mathcal{G}^*$. If o_i is a regular vertex denoting an object, $cnt[i]$ equals the sum of the pdf-values of all the instances of $o[i]$ already retrieved (instead of simply the number of those instances as in Section 5.2). Specially, in case $i = 1$ and o_1 is the vertex *unseen*, $cnt[1]$ is still fixed to 0 (same as in Section 5.2).

The second change is the way cnt and low are updated based on an instance acquired from the vertex step. To facilitate explanation, assume that the instance belongs to object o_i (for some $i \in [1, m]$), and specifically, is the object's i'-th instance (for some $i' \in [1, o_i.s]$). Then, $cnt[i]$ should be increased by $o_i.pdf(o_i[i'])$ (as opposed to only 1 in Section 5.2). Furthermore, array low is not updated based on Equation 6; instead, the update is according to:

$$low[i, j] = low[i, j] + o_i.pdf(o_i[i']) \cdot (1 - cnt[j]). \tag{8}$$

Supporting continuous pdfs. In this paper we focus on objects with discrete pdfs. The problem of SNN-core retrieval can also be defined on continuous pdfs (such as Gaussian) in the same manner. An obvious solution in that context is to first convert a continuous pdf to a discrete one by sampling, and then apply our techniques directly. Alternatively, we may also index the uncertainty regions of the continuous pdfs[1] with a spatial access method such as an R-tree, and then, use the method of [5] to retrieve all the NN-candidates. After deciding the superseding relationships of all pairs of objects, we can use the *full-graph* algorithm to compute the SNN-core. The *pipeline* algorithm, on the other hand, does not appear to be easily adaptable to the continuous pdfs. Finding the SNN-core without retrieving all NN-candidates remains an interesting open problem.

7 Experiments

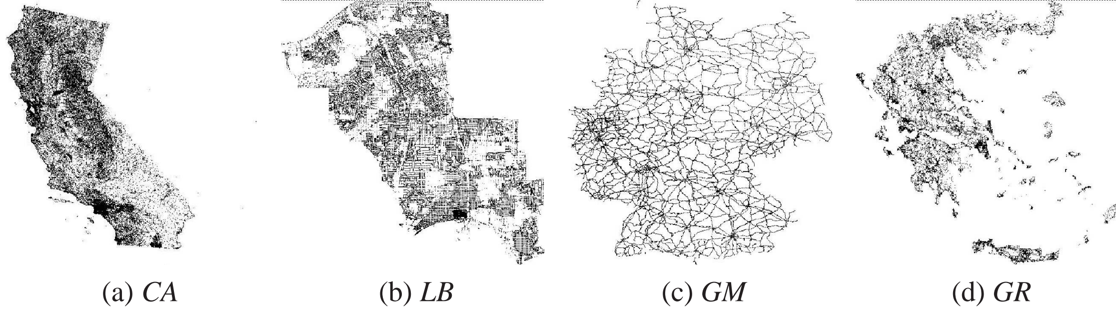
This section experimentally evaluates the proposed techniques. There are two primary objectives. First, we aim at demonstrating that SNN-cores have a small size under a large number of settings with different data distributions, numbers of instances per object, sizes of uncertainty regions, etc. Second, we will examine the efficiency of the two algorithms *full-graph* and *pipeline*. All the experiments are performed on a machine running a 2.13Ghz CPU and 1 giga byte memory.

7.1 Properties of the SNN-core

We choose four real spatial datasets: *CA*, *LB*, *GM*, *GR*, with cardinalities 10113, 10131, 10017, and 10063, respectively (downloadable at *www.census.gov/geo/www/tiger*). Figure 10 shows their data distributions, which correspond to locations in California, Long Beach County, Germany, and Greece, respectively. The data space is normalized to have range [0, 10000] on every dimension.

Given a value s, we transform *CA* into an uncertain database *CA*-s. Specifically, for every point p from *CA*, we generate an uncertain object o with s instances in *CA*-s as follows. First, we create a square $U(p)$ centering at p with side length 600. $U(p)$ is the *uncertainty region* of o, i.e., the area where the instances of o can appear. The instances are created based on the data distribution of *CA* inside $U(p)$. To implement this idea, we impose a grid with resolution 100×100 over the data space. Each cell stores a counter, equal

[1]The uncertainty region of a pdf is the area where the pdf has a positive value.



(a) *CA* (b) *LB* (c) *GM* (d) *GR*

Figure 10: Dataset visualization

dataset	(I) avg. SNN-core size	(II) avg num. of NN candidates	(III) % of NN-cand. in SNN-core	(IV) num. of queries with > 1 SNNs	(V) max. SNN-core size	(VI) num. of PR-correct	(VII) num. of ED-correct
CA-15	1.23	162	0.6%	1	24	38 (of 100)	74 (of 100)
CA-25	1.13	194	0.7%	1	14	35	78
CA-35	1.06	208	0.5%	2	5	42	81
CA-45	1.03	220	0.5%	1	4	39	89
CA-55	1.02	228	0.4%	1	3	50	86
LB-15	1.54	199	0.7%	5	13	20	79
LB-25	1.14	229	0.5%	3	13	28	77
LB-35	1.00	249	0.4%	0	1	20	78
LB-45	1.00	261	0.4%	0	1	31	82
LB-55	1.00	271	0.4%	0	1	23	80
GM-15	1.49	314	0.4%	6	15	17	60
GM-25	1.16	400	0.3%	3	8	13	69
GM-35	1.07	433	0.2%	2	5	13	76
GM-45	1.05	455	0.2%	2	4	17	81
GM-55	1.13	471	0.2%	3	8	17	81
GR-15	1.15	332	0.3%	4	4	20	71
GR-25	1.07	390	0.3%	3	3	35	78
GR-35	1.20	444	0.3%	3	3	26	72
GR-45	1.14	464	0.2%	4	4	23	67
GR-55	1.10	476	0.2%	2	4	41	80

Table 1: Properties of SNN-core (identical uncertainty regions)

to the number of points of *CA* in the cell's extent. Let *cell*(p) be the set of cells covered by $U(p)$. Then, the probability that an instance of o appears in a cell c of *cell*(p), equals the ratio between the counter of c over the sum of the counters of all cells in *cell*(p). After identifying the cell to appear in, the position of the instance is randomly decided in the cell. In this way, the pdf of o depends on the location of p.

In *CA*-s, the uncertainty regions of all objects have the same size. We also synthesize another uncertain dataset *CA*-s-*var*, where the uncertainty regions are not equally large. Specifically, *CA*-s-*var* is generated in the same way as *CA*-s except that the side length of $U(p)$ can be 600 and 1000 with an equal probability.

In the same fashion, given every other real dataset X (= *LB*, *GM*, *GR*) and a value of s, we create two uncertain datasets named X-s and X-s-*var*, respectively. We will experiment with 5 values of s: 15, 25, 35, 45, and 55, resulting in totally 40 different datasets. We index each dataset using an R*-tree with page size

dataset	(I) avg. SNN-core size	(II) avg num. of NN candidates	(III) % of NN-cand. in SNN-core	(IV) num. of queries with > 1 SNNs	(V) max. SNN-core size	(VI) num. of PR-correct	(VII) num. of ED-correct
CA-15-*var*	1.12	243	0.4%	3	5	42 (of 100)	77 (of 100)
CA-25-*var*	1.15	286	0.4%	2	10	49	82
CA-35-*var*	1.02	311	0.3%	1	3	35	90
CA-45-*var*	1.02	325	0.3%	1	3	44	91
CA-55-*var*	1.00	346	0.3%	0	1	50	91
LB-15-*var*	1.13	256	0.4%	3	6	23	81
LB-25-*var*	1.04	299	0.3%	3	3	26	90
LB-35-*var*	1.09	318	0.3%	3	4	31	80
LB-45-*var*	1.04	344	0.3%	3	3	28	92
LB-55-*var*	1.02	347	0.3%	1	3	23	85
GM-15-*var*	1.14	427	0.3%	3	7	17	60
GM-25-*var*	1.18	499	0.2%	4	7	13	69
GM-35-*var*	1.06	556	0.2%	3	3	13	76
GM-45-*var*	1.14	593	0.2%	2	8	17	81
GM-55-*var*	1.00	602	0.2%	0	1	17	81
GR-15-*var*	1.10	516	0.2%	3	5	35	63
GR-25-*var*	1.08	428	0.3%	2	6	42	56
GR-35-*var*	1.00	531	0.2%	0	1	39	75
GR-45-*var*	1.06	577	0.2%	1	7	30	66
GR-55-*var*	1.04	645	0.2%	1	5	29	77

Table 2: Properties of SNN-core (variable uncertainty regions)

4096 bytes. A *query workload* contains 100 query points randomly distributed in the data space.

Results on Objects with Identical Uncertainty Regions. Table 1 demonstrates various statistics about the SNN-cores retrieved by a query workload, on different datasets. We will explain the columns in turn. Column I shows the average number of points in an SNN-core, Column II gives the average number of NN-candidates per query, and Column III presents the percentage of NN-candidates that belong to the SNN-core. Clearly, the average size of an SNN-core is close to 1, and is significantly smaller than the average number of NN-candidates.

Column IV further demonstrates how many queries in a workload have more than one object in their SNN-cores. Column V indicates the size of the largest SNN-core in a workload. We notice that A majority of queries have only one object in their SNN-cores. When the number s of instances is small, occasionally an SNN-core has a large size. For $s \geq 35$, the SNN-core size is consistently low. We point out that, in many applications, an object is usually represented with a large number of instances (e.g., over 100 in [14]) to provide a good approximation of its pdf. The above observation suggests that the SNN-core is expected to be very small in those applications. It is worth mentioning that we do not observe any obvious correlation between the SNN-core size and the number of NN-candidates.

Recall that, as discussed in Section 2.2, there exist two other principles for selecting the best objects for an NN query on uncertain data. Specifically, the *PR-principle* advocates the object with the largest NN probability. The *expected-distance* (ED) *principle*, on the other hand, recommends the object with the lowest expected distance to the query point. Next, we study how often these principles happen to return the same result as our SNN approach. We say that a query is *PR-* (*ED-*) *correct*, if its SNN-core has exactly

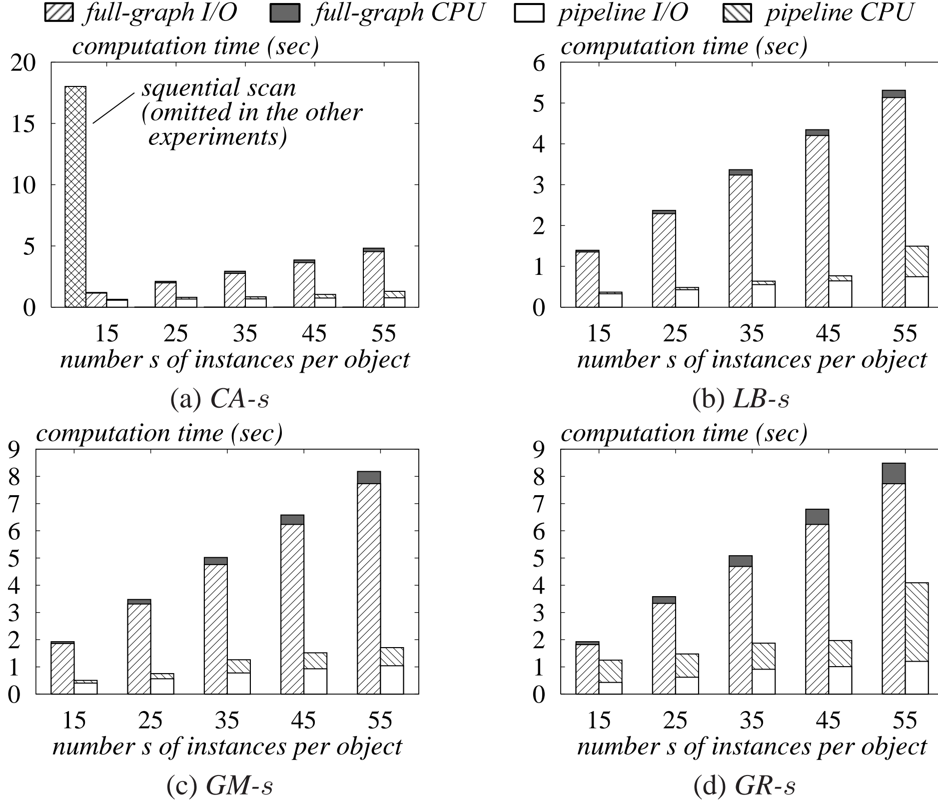


Figure 11: Efficiency comparison (identical uncertainty regions)

one object, which turns out to be the same as the result under the PR- (ED-) principle. Column VI (VII) demonstrates the number of PR- (ED-) correct queries in a workload which, as mentioned earlier, has 100 queries in total. Apparently, neither the PR- nor the ED-principle guarantees the same result as our SNN approach.

Results on Objects with Variable Uncertainty Regions. Next, we repeat the above experiments using the datasets where the uncertainty regions of various objects can have different sizes. Table 2 presents the results, confirming the earlier observations.

Summary. It is clear that our SNN approach is able to recommend a very small number of best objects under a large variety of circumstances. This number tends to be lower when each object is represented by a larger number of instances. Furthermore, for a majority of queries (over 95%), their SNN-cores actually include only a single object.

7.2 Efficiency of Our Algorithms

Now we proceed to evaluate the cost of SNN-core retrieval, by comparing the efficiency of the proposed *full-graph* and *pipeline* algorithms.

Results on Objects with Identical Uncertainty Regions. The first set of experiments adopts datasets *CA*-15, *CA*-25, ..., *CA*-55 where each object is represented with s = 15, ..., 55 instances, respectively. On every dataset, we deploy each algorithm to answer a workload of queries, and measure the average cost per query.

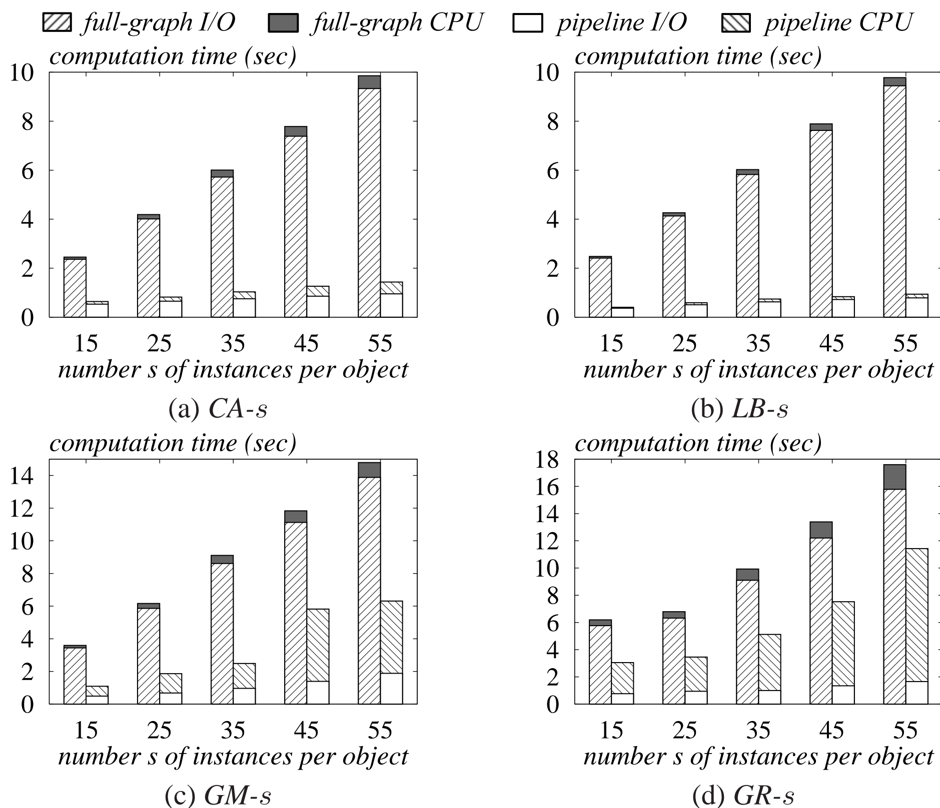


Figure 12: Efficiency comparison (variable uncertainty regions)

Figure 11a plots the cost as a function of s. For comparison, at $s = 15$, we also include the time of a naive sequential-scan solution that simply scans the entire database. The performance of *full-graph* and *pipeline* is further broken down into two parts, corresponding to the CPU and I/O overhead, respectively.

Both proposed algorithms are faster than sequential scan by an order of magnitude. Therefore, we omit sequential scan in the other experiments. As expected, *pipeline* significantly outperforms *full-graph*. Observe that *pipeline* entails slightly higher CPU time, because it involves a sophisticated incremental pruning mechanism. The extra CPU overhead pays off, because the mechanism leads to substantial reduction in the number of R-tree nodes that must be accessed. The benefits are even more obvious when the number of per-object instances grows. Figures 11b-11d present the results of the same experiment on the other datasets. Similar phenomena can be observed.

Results on Objects with Variable Uncertainty Regions. Figure 12 shows the results of the same experiments on the datasets where various objects have uncertainty regions with different sizes. Again, *pipeline* is consistently faster than its competitor. SNN-retrieval incurs higher cost compared to Figure 11. This is due to the increase in the number of NN-candidates, as is clear by comparing the Column III of Tables 1 and 2.

Summary. Both *full-graph* and *pipeline* need to access only a fraction of the database, as indicated by their much smaller cost than sequential scan. *Pipeline* entails considerably lower overall cost than *full-graph*. In particular, the former requires higher CPU time but incurs considerably lower I/O overhead. The advantage of *full-graph*, however, is its simple implementation. Furthermore, this algorithm is expected to be faster than *pipeline* in scenarios where nodes of R-trees can be accessed efficiently (i.e., main-memory databases).

8 Conclusions

Nearest neighbor search on uncertain objects does not have an obvious answer, because typically no object is the NN with absolutely certainty. In this paper, we propose to return the *SNN-core* as the query result. The SNN-core is the minimum set of NN-candidates each of which supersedes all the NN-candidates outside the SNN-core. Our experiments reveal that, for a majority of queries, the SNN-core contains a only single object. This makes SNN-core a highly useful type of results for NN search, where it is important to minimize the number of reported objects. We develop two algorithms for fast computation of the SNN-core. Utilizing a multidimensional index, both algorithms are able to find the SNN-core by accessing only a fraction of the database.

Our work also indicates several directions for future work. First, it would be interesting to analyze the overhead of SNN retrieval, in order to derive a cost model that accurately predicts the query time. Such a model is important for query optimization in practice. Second, the concept of SNN-core can be integrated with any variation of NN search. It remains unclear how to adapt the proposed solutions to those variations. Third, our discussion focuses on spatial data with low dimensionality. It is a challenging topic to study the retrieval of SNN-cores in high-dimensional spaces.

Acknowledgements

This work was supported by GRF grants 4169/09, 4173/08, 4161/07, and 1020/06 from HKRGC.

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM Management of Data (SIGMOD)*, pages 322–331, 1990.

[2] G. Beskales, M. A. Soliman, and I. F. Ilyas. Efficient search for the top-k probable nearest neighbors in uncertain databases. *PVLDB*, 1(1):326–339, 2008.

[3] C. Bohm, A. Pryakhin, and M. Schubert. The gauss-tree: Efficient object identification in databases of probabilistic feature vectors. In *Proc. of International Conference on Data Engineering (ICDE)*, page 9, 2006.

[4] R. Cheng, J. Chen, M. F. Mokbel, and C.-Y. Chow. Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 973–982, 2008.

[5] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Querying imprecise data in moving object environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(9):1112–1127, 2004.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2001.

[7] X. Dai, M. L. Yiu, N. Mamoulis, Y. Tao, and M. Vaitis. Probabilistic spatial queries on existentially uncertain data. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 400–417, 2005.

[8] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.

[9] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: a probabilistic threshold approach. In *Proc. of ACM Management of Data (SIGMOD)*, pages 673–686, 2008.

[10] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[11] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of ACM Management of Data (SIGMOD)*, pages 201–212, 2000.

[12] H.-P. Kriegel, P. Kunath, and M. Renz. Probabilistic nearest-neighbor query on uncertain objects. In *Database Systems for Advanced Applications (DASFAA)*, pages 337–348, 2007.

[13] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems (TODS)*, 30(2):529–576, 2005.

[14] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *Proc. of Very Large Data Bases (VLDB)*, pages 15–26, 2007.

[15] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of ACM Management of Data (SIGMOD)*, pages 71–79, 1995.

[16] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. *Proc. of ACM Management of Data (SIGMOD)*, 27(2):154–165, 1998.

[17] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 896–905, 2007.

[18] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002.

[19] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of Very Large Data Bases (VLDB)*, pages 194–205, 1998.

[20] K. Yi, F. Li, G. Kollios, and D. Srivastava. Efficient processing of top-k queries in uncertain databases. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 1406–1408, 2008.