Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Future Generation Computer Systems 25 (2009) 64–76
FGCS
FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Distance browsing in distributed multimedia databases

Fabrizio Falchi[a,*], Claudio Gennaro[a], Fausto Rabitti[a], Pavel Zezula[b]

[a] *ISTI-CNR, Pisa, Italy*
[b] *Masaryk University, Brno, Czech Republic*

Received 31 August 2007; received in revised form 7 February 2008; accepted 20 February 2008
Available online 26 February 2008

Abstract

The state of the art of searching for non-text data (e.g., images) is to use extracted metadata annotations or text, which might be available as a related information. However, supporting real content-based audiovisual search, based on similarity search on features, is significantly more expensive than searching for text. Moreover, such search exhibits linear scalability with respect to the dataset size, so parallel query execution is needed.

In this paper, we present a Distributed Incremental Nearest Neighbor algorithm (*DINN*) for finding closest objects in an incremental fashion over data distributed among computer nodes, each able to perform its local Incremental Nearest Neighbor (*local-INN*) algorithm. We prove that our algorithm is optimum with respect to both the number of involved nodes and the number of *local-INN* invocations. An implementation of our *DINN* algorithm, on a real P2P system called *MCAN*, was used for conducting an extensive experimental evaluation on a real-life dataset.

The proposed algorithm is being used in two running projects: SAPIR and NeP4B.
© 2008 Elsevier B.V. All rights reserved.

Keywords: Distributed; Incremental; Nearest neighbor; Similarity search; Peer-to-peer; MCAN; Content addressable networks; Metric spaces

1. Introduction

A large component of the web content nowadays consists of non-text data, such as images, music, animations, and videos. Current search engines index web documents by their textual content. For instance, web tools for performing image searching (such as the ones provided by Google, Yahoo! or MSN Live Search) simply index the text associated with the image and the ALT attribute of the IMG tag used to provide a description of an image.

Image indexing methods based on content-based analysis or pattern matching (which for instance analyzes the characteristics of images, i.e., features, such as colors and shapes) are usually not exploited at all. The problem is that these processes are significantly more expensive than text analysis. Nevertheless, what is more important is that the search on the level of features exhibits linear scalability with respect to the data search size, which is not acceptable for the expected dimension of the problem. The reason is that for this kind of data the appropriate search methods are based on similarity paradigms that typically exploit range queries and nearest neighbor queries. These queries are computationally more intensive than the exact match, because conventional inverted indexes used for text are not applicable for such data.

Besides multimedia information retrieval, there are other applications, such as bioinformatics, data mining, pattern recognition, machine learning, computer vision, that can take advantage of the similarity search paradigm. However, different applications have in general different similarity functions. A convenient way to address this problem and achieve one solution for several purposes is to formalize the similarity by the mathematical notion of the *metric space*. Here data elements are assumed to be objects from a metric space where pairwise distances between the objects can be determined and where any distance satisfies the properties of *symmetry*, *non-negativity*, *identity*, and *triangle inequality* [16]. In this respect, the metric space approach to similarity searching is highly extensible. However, our Distributed Incremental Nearest Neighbor (*DINN*) algorithm does even not require the

* Corresponding author. Tel.: +39 0503153139; fax: +39 0503153464.
E-mail addresses: fabrizio.falchi@isti.cnr.it (F. Falchi), claudio.gennaro@isti.cnr.it (C. Gennaro), fausto.rabitti@isti.cnr.it (F. Rabitti), zezula@fi.muni.cz (P. Zezula).

0167-739X/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2008.02.007

objects to be metric — we only suppose that the distance is non-negative.

To address the problems of scalability, P2P communication paradigm seems to be a convenient approach, and several scalable and distributed search structures have been proposed even for the most generic case of metric space searching (see [3] and [4] for a survey). A common characteristic of all these existing approaches is the autonomy of the peers with no need of central coordination or flooding strategies. Since there are no bottlenecks, the structures are scalable and high performance is achieved through parallel query execution on individual peers.

Since the number of closest objects is typically easier to specify than establishing a search range, users prefer the *nearest neighbors* to the *range* queries. For example, given an image, it is easier to ask for 10 most similar ones according to an image similarity criterion than to define the similarity threshold quantified as a real number. However, nearest neighbors algorithms are typically more difficult to implement, and in P2P environments the situation is even worse. The main reason is that traditional (optimum) approaches [10] are based on a *priority queue* with a ranking criterion, which sequentially decides the order of accessed data buckets. In fact, the existence of centralized entities and sequential processing are completely in contradiction with decentralization and parallelism objectives of any P2P search network. Things are further complicated by the natural necessity of some applications to retrieve the nearest neighbor in an incremental fashion, because the number of desired neighbors is unknown in advance. By incremental, we mean that such an algorithm computes the neighbors one by one, without the need to re-compute the query from scratch.

An important example of the application of Incremental Nearest Neighbor is processing of complex queries, that is queries involving more than one feature overlay, such as: find all images most similar to the query image with respect to the color and the shape at once. In this situation, we do not know how many neighbors must be retrieved in individual layers before the best object is found that satisfies the complex condition. In fact, the widely used A_0 (also called Fagin's Algorithm) [5] as well as the *threshold* algorithm [6] suppose that each single source for a specific feature is able to perform an *INN* algorithm.

In this paper, we present a first attempt to approach the Incremental Nearest Neighbor problem for P2P-based systems. Our proposed solution, based on a generalization of the algorithm proposed in [10] for hierarchical centralized structures, is optimal and independent of any specific P2P architecture — it can be applied to any *Scalable and Distributed Data Structure* (SDDS), P2P system, and Grid-based similarity search infrastructure. We implemented our algorithm on a real P2P system called *MCAN* [8,9] and we conducted an extensive experimental evaluation on a real-life dataset of 1,000,000 objects. *MCAN* is a scalable distributed similarity search structure for metric data (for a survey see [3]) which extends the Content-Addressable Network (CAN) (a well-known Distributed Hash Table).

The *DINN* algorithm is being used in two running projects: SAPIR[1] and NeP4B.[2] The European project SAPIR (Search on Audiovisual content using peer-to-peer Information Retrieval) aims at finding new ways to analyze, index, and retrieve the tremendous amounts of speech, image, video, and music that are filling our digital universe, going beyond what the most popular engines are still doing, that is, searching using text tags that have been associated to multimedia files. SAPIR is a three-year research project that aims at breaking this technological barrier by developing a large-scale, distributed peer-to-peer infrastructure that will make it possible to search for audiovisual content by querying the specific characteristics (i.e. features) of the content. SAPIR's goal is to establish a giant peer-to-peer network, where users are peers that produce audiovisual content using multiple devices (e.g., cell phones) and service providers will use more powerful peers that maintain indexes and provide search capabilities

NeP4B (Networked Peers for Business), is an Italian project aiming at innovative ICTs solutions for Small and Medium size Enterprises (SMEs), by developing an advanced technological infrastructure to enable companies of any nature, size and geographic location to search for partners, negotiate and collaborate without limitations and constraints. The infrastructure will base on independent and interoperable semantic peers which behave as nodes of a virtual network. The project vision is towards an Internet-based structured marketplace where companies can access the huge amount of information already present in vertical portals and corporate databases and use it for dynamic, value-adding collaboration purposes. In the NeP4B P2P infrastructure the semantic peers represent aggregations of SMEs with similar activities and the multimedia objects are descriptions/presentations of their products/services extracted from the companies' web sites.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 provides an overview of our proposed solution while the formal definition is given in Section 4. In Section 5 we report the results of an extensive experimental evaluation of the *DINN* over the *MCAN*. Conclusions and future work are discussed in Section 6.

An earlier version of this paper has been presented at the Second International Conference on Scalable Information Systems (INFOSCALE 2007) [7].

2. Related work

Our proposed solution is based on a generalization of the algorithm proposed in [10]. The incremental nearest neighbor algorithm defined in [10] is applicable whenever the search space is structured in a hierarchical manner. The algorithm starts off by initializing the queue of pending requests with the root of the search structure — since the order of entries in this queue is crucial, they refer to it as the *priority queue*. In the main loop, the element closest to the query is taken off

[1] http://www.sapir.eu/.
[2] http://dbgroup.unimo.it/nep4b/.

the queue. If it is an object, it reports it as the next nearest object. Otherwise, the child elements of the element in the search hierarchy are inserted into the *priority queue*.

In [15] an efficient algorithm to perform *k-NN* in a P2P system (specifically the Chord [14]) is proposed. The algorithm uses the same priority queue-based approach of [10]. As far as we know, it is the first attempt to extend [10] to the distributed environment making use of the parallelism of the P2P network.

They define their algorithm for a hierarchical index (as in [10]). To provide distributed hashing of spatial data they use a distributed quadtree index they developed, although they say that other indices can be utilized as well (e.g., P2P R-trees [11]). The query is first initiated on a single peer in the P2P network. This peer maintains the *priority queue* of quadtree blocks (mapping to a control point each) that are being processed for the query. To process a block, they have to contact from this query initiating peer, the peer that owns that block, i.e., the control point. Hence, in their parallel algorithm, they contact, rather than just the top entry of the *priority queue*, a multiple number of these peers.

3. DINN outline

The *INN* algorithm [10] was defined for a large class of centralized hierarchical spatial data structures. Instead our *DINN* algorithm is distributed and not limited to hierarchical structures. Thus it can be used over SDDSs, P2P systems and Grid infrastructures. Our algorithm is built over nodes which are able to perform locally an *INN* between the objects they store (this will be formalized in Assumption 1).

In particular, we reformulate the definition of *priority queue* (*Queue*) given in [10] by considering as elements of *Queue*, objects and nodes (or peers). We prove that our algorithm is optimal, in terms of both number of involved nodes and *local-INN* invocations. The elements of *Queue* are ordered according to a *key* which is always associated with both objects and nodes. The *key* associated with each object is the distance between the query and the object itself. Instead the *key* associated with each node is a lower bound for the distance between the query and the next result coming from the node. While for an already involved node this lower bound can be simply the distance from the query of the last object retrieved by its *local-INN*, for the not yet involved nodes a naive solution could be to always use 0 as lower bound. However, this would imply all nodes to be involved for every similarity query. To avoid this, we suppose that each node is able to evaluate this lower bound for every node it knows (in P2P systems they are called neighbors).

Furthermore, in P2P systems there is no global knowledge of the network. Thus, we make an assumption (see Assumption 2) regarding the ability to find the next most promising node (by considering the lower bound mentioned before). This assumption replaces the *consistency condition* used in [10] for hierarchical data structures. We prove that our assumption can be satisfied under one of two simpler conditions (see Section 4.3.3) which are common for data structures able to perform similarity search.

During the *DINN* algorithm execution, *Queue* contains a certain number of entries sorted in order of decreasing *key*. Entries can be both nodes and objects. Because of the values used as *key*, when a node is after an object we are sure that no better results than the object itself can be found in the node. The algorithm proceeds by processing *Queue* from the top. Basically if the first entry of the queue is an object, this object is the result of the *DINN*. In case the first entry is a node, we invoke its *local-INN*. The resulting object of this invocation is placed in *Queue* and its distance from the query allows us to update the entry with a more accurate (greater) lower bound which moves the node backward in *Queue*.

This outlined implementation is intrinsically sequential, since a single step of the algorithm involves only the first element of *Queue* at a time. In the second part of the paper, we straightforwardly generalize the algorithm introducing parallelism by invoking the *local-INN* algorithm of more than one node simultaneously. The precise definition of the algorithms is provided in the next section. Examples are given to help understanding the algorithm.

4. The DINN algorithm

4.1. Definitions and notation

In this subsection we provide a number of definitions and notations required to define the *DINN* algorithm.

Notation:

- $\mathcal{N}$ is the set of the nodes participating in the distributed system
- $\mathcal{D}$ is the objects domain
- $\mathcal{X}_i \subset \mathcal{D}$ is the set of the objects stored in a given node $N_i \in \mathcal{N}$
- $\mathcal{X} = \bigcup_i \mathcal{X}_i$ is the set of the objects stored in the whole network.

As in [10], our *DINN* is based on a *priority queue*:

Definition 1. A *priority queue* (*Queue*) is a set of pairs $\langle element, \vartheta \rangle$ ordered according to *key* $\vartheta \in \mathbb{R}^+$. An element can be either an object or a node.

In order to avoid involving all the nodes in the *DINN* execution, we suppose there is the possibility to evaluate a lower bound (δ) for the distances between the objects stored in a certain node and any given object in $\mathcal{D}$.

Definition 2. Given a node $N_i \in \mathcal{N}$ and an object $x \in \mathcal{D}$ we define $\delta : \mathcal{N} \times \mathcal{D} \to \mathbb{R}^+$ as a lower bound for the distances between x and all the objects stored in N_i (i.e., $\mathcal{X}_i$):

$$\delta(N_i, x) \leq \min\{d(y, x),\ y \in \mathcal{X}_i\}.$$

Note that this lower bound could even be 0 for every node. Thus we do not strictly require this lower bound to be evaluable, but we use it for efficiency in case it can be given. In case each node $N_i \in \mathcal{N}$ of a given distributed data structure is responsible for a portion $\mathcal{D}_i$ of the domain $\mathcal{D}$ we will say that δ is *strong* iff:

$$\forall N_i \in \mathcal{N}, \quad \delta(N_i, x) = 0 \Leftrightarrow x \in \mathcal{D}_i.$$

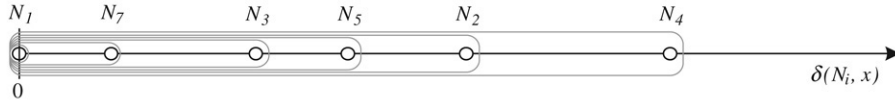


Fig. 1. *Downward closed* set of nodes with respect to *object*.

In defining our *DINN* algorithm we will use the general notion of *downward closed set*. We will limit this notion to a set of nodes with respect to a given object (making use of the lower bound δ defined above).

Definition 3. A set of nodes $\mathcal{N}_x$ is *downward closed* with respect to an object $x \in \mathcal{D}$ iff $\forall N_j, N_i \in \mathcal{N}$:

$$N_i \in \mathcal{N}_x \wedge \delta(N_j, x) < \delta(N_i, x) \Rightarrow N_j \in \mathcal{N}_x.$$

In other words, if a set of nodes is *downward closed* with respect to an object $x \in \mathcal{D}$, there are no nodes, out of the set, with a lower bound lesser to those of the nodes in the set. In Fig. 1 we give an example of *downward closed* sets of nodes. The position of each node N_i on the axis is determined by $\delta(N_i, x)$. The nodes are grouped in all the possible *downward closed* sets. Note that, by Definition 3, each set contains N_1 which has the minimum δ from x. Moreover, each *downward closed* set of nodes contains any node between N_1 and the furthest away node in the set itself. Obviously, the position on the axis of the nodes depends on the particular object x.

Another special set of nodes we will refer in the algorithm definition is the set of nodes whose lower bound δ is less than a given $r \in \mathbb{R}^+$:

Definition 4. Let x be an object in $\mathcal{D}$ and $r \in \mathbb{R}^+$. $\mathcal{N}_{x,r}$ is the set of nodes in $\mathcal{N}$ that could have objects closer to x than r, i.e.,

$$\mathcal{N}_{x,r} = \{N_i : N_i \in \mathcal{N} \wedge \delta(N_i, x) \le r\}.$$

In Table 1, we report the list of symbols used in this paper with their corresponding meaning. Please note that in this table also symbols that will be defined and used in the next section can be found.

Table 1
Notation

Symbol	Meaning
$\mathcal{N}$	The set of the nodes participating in the distributed system
N_i	A node participating in the distributed system ($N_i \in \mathcal{N}$)
$\mathcal{D}$	The objects domain
x	An object in the domain ($x \in \mathcal{D}$)
$d(x, y)$	The distance between x and y ($x, y \in \mathcal{D}$)
r	A value in $\mathbb{R}^+$
$\mathcal{X}_i$	$\mathcal{X}_i \subset \mathcal{D}$ is the set of the objects stored in a given node $N_i \in \mathcal{N}$
$\mathcal{X}$	The set of the objects stored in the whole network ($\mathcal{X} = \bigcup_i \mathcal{X}_i$)
$\delta(N_i, x)$	The lower bound for the distances between $x \in \mathcal{D}$ and all the objects stored in N_i (i.e., $\mathcal{X}_i$)
$\mathcal{N}_{x,r}$	The set of nodes in $\mathcal{N}$ that could have objects closer to x than r
$\mathcal{N}_x$	A subset of nodes $\mathcal{N}$ which is either *downward closed* (with respect to x) or empty
N_n	The closest node to x in ($\mathcal{N} \setminus \mathcal{N}_x$)
$\mathcal{N}^*$	The set of nodes that already performed a *local-INN*
e	An element, either an object or a node, in *Queue*
ϑ	$\vartheta \in \mathbb{R}^+$ is the key used for ordering the elements of *Queue*
Queue	The set of pairs $\langle element, \vartheta \rangle$ ordered according to *key* ϑ
$\bar{k}$	The number of objects already retrieved by the previous invocations of the *DINN*
k^+	The number of next neighbors we want to retrieve
k_{ans}	The number of results already found by the *DINN* during the current invocation
$\hat{k}$	$\hat{k} = k^+ - k_{ans}$
$x_{\hat{k}}$	$x_{\hat{k}} \in \mathcal{X}$ is the $\hat{k}$-th object in *Queue*
p	$p \in [0, 1]$ is the parameter used to set the degree of parallelization of the *DINN*
l_i	$l_i \in \mathcal{X}_i$ is the last object returned by N_i

4.2. Assumptions

Our *DINN* algorithm is based on two assumptions.

Assumption 1. Each node $N_i \in \mathcal{N}$ is able to perform a *local-INN* algorithm over the objects $\mathcal{X}_i \subseteq \mathcal{D}_i$ it stores.

Assumption 2. Let $x \in \mathcal{D}$ be an object in the domain. Let $\mathcal{N}_x \subseteq \mathcal{N}$ be a subset of nodes which is either *downward closed* (with respect to x) or empty. Let $N_n \in (\mathcal{N} \setminus \mathcal{N}_x)$ be the closest node to x in $(\mathcal{N} \setminus \mathcal{N}_x)$, i.e.,

$$N_n = \arg\min_{N_i}\{\delta(N_i, x), N_i \in (\mathcal{N} \setminus \mathcal{N}_x)\}.$$

Whenever an arbitrary node $N_c \in \mathcal{N}$ *knows* $\mathcal{N}_x$ (i.e., would be able to contact all the nodes in $\mathcal{N}_x$), N_c must be able to check if N_n exists (i.e., $(\mathcal{N} \setminus \mathcal{N}_x) \neq \emptyset$) and, eventually, to contact it.

Assumption 1 is needed because our *DINN* algorithm is built over nodes which are able to perform a *local-INN*.

Assumption 2 is necessary for engaging the nodes in the *DINN* algorithm execution as it progresses. Basically, given the lower bound δ defined in Definition 2, we require a mechanism for adding the nodes to *Queue* in order of increasing δ from a query q. In case there is some replication in the distributed system, there could be two or more nodes $N_j \in \mathcal{N}$ for which $\delta(N_j, x) = 0$. However, we only need to find one of them.

When $\mathcal{N}_x \neq \emptyset$, Assumption 2 means that the distributed system must be able to search for the next most promising node (N_n) given that we already know a set of nodes ($\mathcal{N}_x$) which are more, or equally, promising (by considering δ) than the next one (i.e., $\mathcal{N}_x$ is *downward closed*).

The role of the *downward closed* subset $\mathcal{N}_x$ will be clarified in the next section which will extensively discuss the algorithm. However, we can anticipate that, because of the algorithm definition, it is a subset of the nodes that, at any given time during the algorithm execution, has already been asked for a *local-INN* execution. In particular, if $\mathcal{N}_x = \emptyset$, Assumption 2 means that any node $N_c \in \mathcal{N}$ must be able to find (using some

routing mechanism provided by the distributed system), a node $N_n \in \mathcal{N}$ for which the distance $\delta(N_n, x)$ is minimum.

If, for a specific data structure, it is not possible to evaluate the lower bound δ, we can consider $\delta(N_i, q) = 0$ for every node $N_i \in \mathcal{N}$. In this case the order in which the nodes are added to *Queue* is undefined. However in this case, we will involve all the nodes in (almost) every execution of the *DINN* algorithm. In fact, given that there is no lower bound for the distance between the objects stored in a given node and the query, we cannot exclude any node a priori.

Please note that we do not suppose that in the distributed system there is a global knowledge of the network. We only assume that there is a way (usually a routing mechanism) to find the most promising node for the algorithm progress. It can also be noted that if δ is *strong*, the first node added to *Queue* is the node N_n that would contain x (i.e., $\delta(N_n, x) = 0$). Therefore, in this case, the problem of finding the most promising node becomes similar to the *lookup* problem in Distributed Hash Tables.

While Assumption 2 is the most generic one, there are simpler assumptions that can substitute it. In fact, in Section 4.3.3, we illustrate two sufficient conditions for Assumption 2. Condition 1 guarantees that the next most promising node is always in *Queue* by just collecting information about neighbors of yet involved nodes. On the other hand, Condition 2 is easily satisfied by data structures able to perform similarity search because it basically makes use of the capability of a system to perform range queries.

4.3. The algorithm

In this section we present the definition of our *DINN* algorithm for retrieving objects in order of decreasing similarity with respect to a given query q. In particular, we will define the process of retrieving the next closest object to q at each *DINN* invocation. In Section 4.4 we will present a message reduction optimization in case we want to retrieve more than one object at each *DINN* invocation. Finally in Section 4.5 the proposed algorithm will be extended to parallelize the operations made by distinct nodes.

To perform the *DINN* we need to define a node that takes the role of coordinating node (N_c). A good candidate for this role is the initiating node (i.e., the node requesting the search). Another good candidate, in case δ is *strong* (see Definition 2) is the node that would store the query (i.e., $\delta(N_c, x) = 0$). However, the definition of our *DINN* algorithm is independent of the particular choice of the coordinating node. This choice only affects the number of messages exchanged during the query execution.

As in [10] we need a *Queue* (see Definition 1) in which elements are ordered according to their *key* (see Definition 5). Moreover, when an object and an element have the same *key*, the object comes before the node in *Queue*. In *Queue* nodes will be assigned a different *key* (ϑ) depending on whether they have already returned objects or not. Thus, we will use the following notation:

Notation 1. $\mathcal{N}^* \subset \mathcal{N}$ is the set of nodes that already performed a *local-INN*.

An important part of the *DINN* algorithm definition is the definition of the *keys* used to order elements in *Queue*.

Definition 5. Given a query object $q \in \mathcal{D}$ we define the *key* ϑ as:

- $\vartheta_x = d(x, q)$, for any object $x \in \mathcal{D}$;
- $\vartheta_{N_i} = \delta(N_i, q)$, for any node N_i that has not yet been asked for a *local-INN* (i.e., $N_i \notin \mathcal{N}^*$);
- $\vartheta_{N_i} = d(l_i, q)$, for any $N_i \in \mathcal{N}^*$, where $l_i \in \mathcal{X}_i$ is the last object that N_i returned when performing its *local-INN*.

Note that both *keys* used for nodes are lower bounds for the distance between the query q and the next result coming from the *local-INN* invocation on node N_i.

The *DINN* algorithm consists of a loop in which:

1. If *Queue* is empty, the closest node (N_n) to q that has not yet performed a *local-INN* is added to *Queue*. In case N_n does not exist, the *DINN* terminates (there are no more objects in the distributed data structures);
2. Else, if the first element in *Queue* is a node (N_i), this node is asked to perform a *local-INN*. Then the returned result $l_i \in \mathcal{X}_i$ is added to *Queue* and the *key* of N_i is updated with $\vartheta_{N_i} = d(l_i, q)$. In case N_i did not return any object (i.e., it has already returned all its objects), the N_i is removed from *Queue*;
3. Else, if the first element in *Queue* is an object x: let N_n be the closest node to q that has not yet performed a *local-INN* and has $\delta(N_n, q) < d(x, q)$; if N_n exists, add it to *Queue*, otherwise the loop is exited returning x as the next result. Note that if $\mathcal{N}^*$ is *downward closed* N_n can be found because of Assumption 2. We prove $\mathcal{N}^*$ to be *downward closed* in Corollary 1 (Section 4.3.1).

Queue must be kept alive for future request of more results. Obviously, the requester can close the session asserting that no more results will be asked. In this case *Queue* can be discarded.

In Algorithm 1 we give a definition of the *DINN* algorithm using a pseudo language. The functions and procedures used in Algorithm 1 are defined as follows:

- FIRST(*Queue*): returns the first element in *Queue*.
- LOCALINN(q, N_i): asks node N_i to return the next result according to its *local-INN* with respect to the query q.
- ENQUEUE(*Queue*, $\langle e, \vartheta \rangle$): adds element e, either an object or a node, to *Queue* with key ϑ.
- UPDATEKEY(*Queue*, $\langle N_i, r \rangle$): updates the *key* of node N_i in *Queue* with the value $r \in \mathbb{R}^+$.
- EXQUEUE(*Queue*, e): removes element e and its *key* from *Queue*.
- GETNEXTNODEINR(q, $\mathcal{N}^*$, r): returns $\arg\min_{N_i}\{\delta(N_i, q), N_i \in (\mathcal{N}_{q,r} \setminus \mathcal{N}^*)\}$.
- GETNEXTNODE(q, $\mathcal{N}^*$): returns $\arg\min_{N_i}\{\delta(N_i, q), N_i \in (\mathcal{N} \setminus \mathcal{N}^*)\}$.

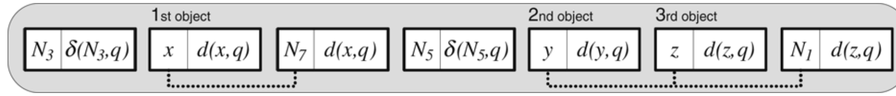


Fig. 2. Snapshot of the priority queue at a given time during the execution of the *DINN* algorithm.

Algorithm 1 Distributed Incremental Nearest Neighbor Algorithm

loop
  if *Queue* is empty **then**
    $N_i \Leftarrow$ GETNEXTNODE$(q, \mathcal{N}^*)$
    if $N_i = NULL$ **then**
      Return NULL
    end if
    ENQUEUE(*Queue*, $\langle N_i, \delta(N_i, q)\rangle$)
  else if FIRST(*Queue*) is an *object* **then**
    $x \Leftarrow$ FIRST(*Queue*)
    $N_i \Leftarrow$ GETNEXTNODEINR$(q, \langle \mathcal{N}^*, d(x, q)\rangle)$
    if $N_i = NULL$ **then**
      Return x
    end if
    ENQUEUE(*Queue*, $\langle N_i, \delta(N_i, q)\rangle$)
  else if FIRST(*Queue*) is a *node* **then**
    $N_i \Leftarrow$ FIRST(*Queue*)
    $x \Leftarrow$ LOCALINN(q, N_i)
    $\mathcal{N}^* \Leftarrow \mathcal{N}^* \cup N_i$
    if $x \neq NULL$ **then**
      ENQUEUE(*Queue*, $\langle x, d(x, q)\rangle$)
      UPDATEKEY($\langle N_i, d(x, q)\rangle$)
    else {node N_i has no more objects}
      EXQUEUE(*Queue*, N_i)
    end if
  end if
end loop

Note that if $\mathcal{N}^*$ is always *downward closed* with respect to q, because of Assumption 2 it is possible to implement the function GETNEXTNODE$(q, \mathcal{N}^*)$. We prove this in Corollary 1 (Section 4.3.1). Please note also that GETNEXTNODEINR$(q, \mathcal{N}^*, r)$ can be implemented using GETNEXTNODE$(q, \mathcal{N}^*)$. On the other side, using GETNEXTNODEINR, we can realize GETNEXTNODE increasing r until a node is found. However, GETNEXTNODEINR$(q, \mathcal{N}^*, r)$ can be more efficiently implemented considering that it does not need to return a node if it is farther away than r from q.

In Fig. 2 we give an example of *Queue* at a given time during the *DINN* execution. The dotted lines show from which node every object comes from. Let us suppose that we are searching for the next nearest object to the query q and we have already found some results which are no more in *Queue*. In fact, whenever a result is found it is moved out of *Queue*. The next element in *Queue* is N_3. Thus, we have to to invoke the N_3 *local-INN* to retrieve its next result. Let w be the next result retrieved by N_3. Once w is retrieved, both w and N_3 are put in the *Queue* with the same *key* $d(w, q)$. However, because w is an object, it will be before N_3 in *Queue* (see the algorithm definition). If $d(w, q) < d(z, q)$ then w is the first element in *Queue* and thus it is also the next result of the *DINN*. Otherwise z is the first element in *Queue* and also the next result.

4.3.1. Correctness

In this section we prove that our *DINN* algorithm is correct, i.e., it returns objects in order of increasing distance (decreasing similarity) from the query q (Theorem 1).

First of all, to guarantee that it is possible to define GETNEXTNODEINR and GETNEXTNODE for a given distributed system under Assumption 2, we must prove that $\mathcal{N}^*$ is always *downward closed* with respect to q:

Corollary 1. *At any time during the DINN algorithm execution, the set of nodes $\mathcal{N}^*$ (i.e., the set of nodes that already performed a* local-INN *) is* downwardclosed *with respect to the query q.*

Proof. We prove the corollary using induction. When the algorithm starts *Queue* is empty and a node N_i is added to *Queue* using GETNEXTNODE$(q, \emptyset)$ (usually $\delta(N_i, q) = 0$). After N_i has been asked for a result, $\mathcal{N}^*$ contains only N_i and is *downward closed* by definition of GETNEXTNODE. At any given time during the algorithm execution, let N_n be the node, if it exists, returned either by the function GETNEXTNODEINR$(q, r, \mathcal{N}^*)$ or by the function GETNEXTNODE$(q, \mathcal{N}^*)$. Because of the functions definitions, if N_n exists, there is no other node $N_j \in \mathcal{N}^*$ for which $\delta(N_j, q) < \delta(N_n, q)$. Then $(\mathcal{N}^* \cup N_n)$ is still *downward closed* with respect to q. □

Theorem 1 (*Correctness*). *Let $\mathcal{R}$ be the set of objects already returned by the DINN algorithm. Whenever DINN returns an object x there are no objects nearer to the query:*

$$\forall y \in \mathcal{X}, \quad d(y, q) < d(x, q) \Rightarrow y \in \mathcal{R}.$$

Proof. By definition of $\mathcal{X}$ there must be a node $N_j \in \mathcal{N}$ for which $y \in \mathcal{X}_j$. Using Definition 4, $d(y, q) < d(x, q) \Rightarrow N_j \in \mathcal{N}_{q,d(x,q)}$. Because of the algorithm definition, GETNEXTNODEINR$(x, d(x, q), \mathcal{N}^*)$ did not return any node. Then, by GETNEXTNODEINR definition, $(\mathcal{N}_{q,d(x,q)} \setminus \mathcal{N}^*) = \emptyset$ and then $N_j \in \mathcal{N}^*$ (i.e., y belongs to a node which has already been asked for a *local-INN*). If $N_y \in \mathcal{N}^*$ has some not returned objects by algorithm definition N_j is in *Queue* with *key* $d(l_i, q)$ (where $l_i \in \mathcal{X}_i$ is the last object it returned). Because x is first, $d(l_i, q) \geq d(x, q) > d(y, q)$. Then y must be between the objects N_i already returned, which are either in $\mathcal{R}$ or in *Queue*. But y cannot be in the priority because x is first and objects are ordered according to their distance from the query, then $y \in \mathcal{R}$. □

4.3.2. Optimality

In this section we prove that our *DINN* algorithm is optimal in terms of number of involved nodes (Theorem 2) and number of *local-INN* invocations (Theorem 3).

Theorem 2. *The DINN is **optimal with respect to the number of involved nodes** given the lower bound δ.*

Proof. The theorem can be rewritten as follows. Let $\mathcal{N}^*$ be the set of involved nodes, $x \in \mathcal{X}$ the last object returned by the *DINN* and $q \in \mathcal{D}$ the query object. Whenever the *local-INN* of N_i is invoked, the lower bound δ of the distance between q and the objects in N_i is less than the distance between q and x, i.e.,

$$N_i \in \mathcal{N}^* \Rightarrow \delta(N_i, q) \leq d(x, q).$$

Because of the algorithm definition (see Algorithm 1), the last returned object x was at the head of *Queue* and each node is requested to perform a *local-INN* result only when they are at the head of *Queue*. Because $\delta(N_i, q)$ and $d(x, q)$ are used as *key* for not yet involved nodes and objects respectively (see Definition 5), the last equation always holds. In fact, both objects and nodes are ordered in *Queue* according to their *keys*. □

Theorem 3. *The DINN is **optimal with respect to the number of local-INN invocations** given the lower bounds δ for the not yet involved nodes, and $d(l_i, q)$ for the yet involved nodes.*

Proof. In Theorem 2 we proved that the *DINN* is optimal in terms of number of involved nodes. Thus, *DINN* is optimal in terms of *local-INN* first invocations. Moreover, being $\vartheta_{N_i} = d(l_i, q)$ the *key* (used to order the elements in *Queue*) for a node N_i that already performed a *local-INN* (see Definition 5), whenever N_i is asked to retrieve its next result (using its *local-INN*) we are sure that the *DINN* next result will be further away than $d(l_i, q)$. In fact, we are using as *key* in *Queue* $d(x, q)$ for every object x and a lower bound for $d(y_i, q)$ for every node N_i (see Definition 5). □

4.3.3. Sufficient conditions for Assumption 2

In this section we give two conditions which are sufficient for Assumption 2. Condition 1 guarantees that the next most promising node is always in *Queue* just collecting information about neighbors of yet involved nodes (i.e., without generating more messages) and is satisfied by *MCAN* which we used in our experiments. On the other hand, Condition 2 makes use of the capability to perform range queries and is thus easily satisfied by data structures able to perform similarity search (as the ones presented in [3,4]).

Condition 1. Let $\mathcal{N}_q$ be a *downward closed* set of nodes with respect to an object $q \in \mathcal{D}$. For any given $N_i \in \mathcal{N}$, let $\mathcal{N}_i \subseteq \mathcal{N}$ be the set of nodes which N_i is able to contact directly independently from the execution of the current *DINN* algorithm. Let $N_n \in \mathcal{N}$ be the closest node to the query (according to δ) which is not in $\mathcal{N}_x$ (as defined in Assumption 2). If N_n exists, it is in the union of the set of nodes known by the nodes in $\mathcal{N}_x$:

$$N_n = \arg\min_{N_i}\{\delta(N_i, q), N_i \in (\mathcal{N}_{q,r} \setminus \mathcal{N}_x)\} \in \bigcup\{\mathcal{N}_i, N_i \in \mathcal{N}_x\}.$$

Theorem 4. *Condition 1 is sufficient for Assumption 2.*

Proof. By Condition 1, N_c can ask each node $N_i \in \mathcal{N}^*$ which are the nodes it has knowledge about ($\mathcal{N}_i$). Sorting the union of them ($\bigcup\{\mathcal{N}_i, N_i \in \mathcal{N}_x\}$) N_c is able to find N_n. Thus, Assumption 2 is satisfied. □

Condition 1 basically says that it is always possible to pass from one node N_{n-1} to the next one (N_n) just using the information we found in the previous nodes. The information we need is the knowledge they have about other nodes (typically neighbors). This condition is very useful to efficiently implement GETNEXTNODE and it is satisfied by *MCAN* which is used in our experiments.

Condition 2. For any given object $q \in \mathcal{D}$ and $r \in \mathbb{R}^+$, every node $N_i \in \mathcal{N}$ is able to know all the nodes (their addresses) in $\mathcal{N}_{x,r}$.

Theorem 5. *Condition 2 is sufficient for Assumption 2.*

Proof. By Condition 2, N_c can ask for all the nodes in $\mathcal{N}_{q,r}$. If $(\mathcal{N}_{q,r} \setminus \mathcal{N}_x) \neq \emptyset$, the next node N_n is the nearest to the query in $(\mathcal{N}_{q,r} \setminus \mathcal{N}_x)$. Otherwise, if $(\mathcal{N}_{q,r} \setminus \mathcal{N}_x) = \emptyset$, N_c can try again increasing r until $r \leq d_{\max}$. In this last case N_n does not exist. □

Please note that all distributed data structures able to perform a range query, should be able to satisfy Condition 2 (and then Assumption 2). Under Condition 2 GETNEXTNODEINR is efficiently implemented while GETNEXTNODE can be realized increasing r until either a node is found, using GETNEXTNODEINR, or r exceeds the maximum possible value of d (i.e., $d_{\max} = \max(d(y, x), y, x \in \mathcal{D})$).

4.3.4. Considerations

The major differences between our *DINN* algorithm and the *INN* defined in [10] are:

- Once a node comes at the head of the queue we do not ask it to return all its objects ordered according to their distances from the query. This would be the natural extension for the *INN* algorithm, but, in a distributed environment, such an algorithm could not be scalable. Therefore, we ask it to return its next object using its *local-INN*;
- Whenever a node returns an object, we move it back in the queue using $d(l_i, q)$ as new *key* (l_i is the last object the N_i returned as a result). Please note that $d(l_i, q)$ is a lower bound for the distance between q and the next result coming from the *local-INN* of N_i;
- The original *INN* algorithm requests a *consistency condition* (Definition 1 of [10]) to ensure that once a node reaches the head of the queue no other nodes can return objects with a distance smaller than the head node *key*. This condition has been defined for hierarchical data structure thus limiting the use of their *INN* algorithm. In our *DINN* we replaced the *consistency condition* with Assumption 2.

4.4. Message reduction

In this section we give an extension of our *DINN* to reduce the number of messages when we want to retrieve the next $k^+ \geq 1$ objects. The price to be paid for the messages reduction is the possibility to ask a node to retrieve more objects than what is strictly necessary. At any given time during the execution of the *DINN*:

Notation 2. Let $\overline{k}$ be the number of objects already retrieved by the previous invocations of the *DINN*,

Notation 3. Let k^+ be the number of objects more we want to retrieve, and

Notation 4. Let $k_{ans} \leq k^+$ be the number of results already found by the *DINN* during the current invocation.

If a node N_i is first in *Queue* we ask this node to retrieve the next $\hat{k}$ results where:

$\hat{k} = k^+ - k_{ans}$.

Because $\hat{k}$ represents the number of objects we need to end the given task (i.e., retrieving the next k^+ objects) we are sure that we will never involve N_i again before the current task will be completed. Note that, by definition, $\hat{k} \geq 1$ always holds until the current task is completed.

Furthermore, we can reduce the number of unnecessary objects retrieved, by considering the distance of the $\hat{k}$-th object, if it exists, in *Queue*.

Definition 6. At any given time during the *DINN* algorithm execution, let $x_{\hat{k}} \in \mathcal{X}$ be the $\hat{k}$-th object, if it exists, in *Queue*. To guarantee that node N_i will be involved only once during the current task, we ask node N_i to perform a sequence of *local-INN* invocations until at least one of the following conditions is true:

- $\hat{k}$ more objects have been retrieved ($\hat{k} = k^+ - k_{ans}$);
- $d(l_i, q) \geq d(x_{\hat{k}}, q)$, where l_i is the last object retrieved;
- all the objects stored in N_i have been retrieved.

The results coming from N_i are added to *Queue*. If all the objects stored in N_i have been retrieved N_i is removed from *Queue*, otherwise its *key* is updated with $\vartheta_{N_i} = d(l_i, q)$ and then $\vartheta_{N_i} \geq d(x_{\hat{k}}, q)$. At this stage there are two possibilities: either the $\hat{k}$ enqueued objects are before N_i or N_i is after $x_{\hat{k}}$. In both cases at least $\hat{k}$ objects are before N_i in *Queue*. Thus, we will not involve N_i again in retrieving the next $\hat{k}$ results.

In Fig. 2 we give an example of *Queue* at a given time during the *DINN* execution. The dotted lines show from which node every object comes from. Let us suppose that we are searching for the next $k^+ = 5$ objects and we have already found the next $k_{ans} = 2$ results (they are no more in *Queue*). We still have to search for the next $\hat{k} = k^+ - k_{ans} = 5 - 2 = 3$ results. The $\hat{k}$-th object $x_{\hat{k}}$ in *Queue* is z. Using the proposed extension, the *DINN* will ask node N_3 to retrieve objects (using its *local-INN*) until either 3 objects have been found or the last object l_3 retrieved by N_3 has distance $d(l_3, q) \geq d(z, q)$.

4.5. Parallelization

The *DINN* algorithm presented in Section 4.3 always involves only the most promising node — the first in *Queue*. In this section we give a parallelized version of our *DINN*.

Generally speaking, the *k-NN* operation, is not an easy operation to parallelize as the *RangeQuery* is. To execute a *RangeQuery*, every single node can perform the search among its objects without considering the results coming from other nodes. Given the query and the range, each node can search among its objects regardless the results found in other peers. To parallelize the *DINN* algorithm we must accept the possibility to ask a node to give its next result even if it could be not necessary. Furthermore, in a parallelized *DINN* it is possible to involve nodes which would not be involved by the serial execution.

Let us assume that at a given time during the algorithm execution x_1 is the first object in *Queue*. In principle it is possible that we will ask all the nodes before x_1 in *Queue* to invoke their *local-INN*(e.g., if all these nodes return results further away from q than x_1). To parallelize the *DINN* execution, we can decide to ask all the nodes before x_1 to retrieve the next object.

We now give a definition of *DINN* parallelization which can be also used in combination with the message optimization given in Definition 6.

Definition 7. Let $x_{\hat{k}} \in \mathcal{X}$ be the $\hat{k}$-th object in *Queue* and $d(x_{\hat{k}}, q)$ its distance from the query. Let $p \in [0, 1]$ be the parallelization parameter. We parallelize the *DINN* asking all the nodes $N_i \in$ *Queue* whose $\vartheta_{N_i} \leq p\, d(x_{\hat{k}}, q)$. In other words, using Definition 5, a node $N_i \in$ *Queue* is involved iff:

- $\vartheta_{N_i} = \delta(N_i, q) \leq p\, d(x_{\hat{k}}, q)$, in case $N_i \in \mathcal{N} \setminus \mathcal{N}^*$ (i.e., N_i has not yet been asked for a *local-INN*);
- $\vartheta_{N_i} = d(l_i, q) \leq p\, d(x_{\hat{k}}, q)$, otherwise (i.e., $N_i \in \mathcal{N}^*$) where $l_i \in \mathcal{X}_i$ is the last object that N_i returned invoking its *local-INN*.

Any involved node is asked to retrieve its next object invoking its *local-INN*. However, using the *DINN* optimization for *k-INN* search (see Definition 6), any node can be asked to perform more than one *local-INN* with a single message. However, in this case, there are nodes that are not at the top of *Queue*, asked to retrieve objects. We can then consider the case in which there are objects before them in *Queue*. Let $\tilde{k}_{N_i}$ be the objects in *Queue* before node N_i. The maximum number of objects we are interested in retrieving from N_i is no more $\hat{k}$ but $\hat{k} - \tilde{k}_{N_i}$.

In Fig. 2 we give a snapshot of *Queue* at a given time during the *DINN* execution. As said before, the dotted lines show from which node each object comes from. As before, let us suppose that we are searching for the next $k^+ = 5$ objects and we have already found the next $k_{ans} = 2$ results. We still have to search for the next $\hat{k} = 3$ results. Using the proposed extension, the *DINN* will ask node N_3, N_5 and N_7 to invoke their *local-INN* and they all will work in parallel. If we also use the message reduction optimization, N_3 will be asked to retrieve at most 3

objects, while N_5 and N_7 will be asked to retrieve at most 2 objects. All of them will stop the iteration of their *local-INN* if $d(l, q) \geq d(z, q)$, where l is the last object they retrieved.

Unfortunately, there could be some nodes (N_i) not yet in *Queue* for which $\vartheta_{N_i} \leq p\, d(x_{\hat{k}}, q)$. In fact, the *DINN* algorithm does guarantee only that the next most promising node is present in *Queue* before asking the first node in *Queue* to perform a *local-INN*. In this case the *DINN* algorithm will continue to be correct, but the parallelization would be reduced. To better parallelize the *DINN* algorithm it is useful to put more nodes in *Queue* than necessary. As said before, parallelizing the *DINN* can increase the total cost. For this reason a parametrized parallelization is useful to find the desired trade-off between total and parallel cost.

Definition 8. Let $\hat{k} \in \mathbb{N}^+$, and $x_{\hat{k}} \in \mathcal{X}$ the $\hat{k}$-th object, if it exists, in *Queue* which is, by definition, ordered. Let $p \in [0, 1]$ be the parallelization parameter. We ask all the nodes in *Queue* whose $\vartheta \leq p\ d(x_{\hat{k}}, q)$ until at least one of the following conditions is true (as in Definition 6):

- $\hat{k}$ more objects have been retrieved ($\hat{k} = k^+ - k_{ans}$);
- $d(l_i, q) \geq d(x_{\hat{k}}, q)$, where l_i is the last object retrieved;
- all the objects stored in N_i have been retrieved.

Note that, since $\hat{k} \leq k^+$, the degree of parallelization does depend on k^+. In other words, the more objects we request at each invocation of the *DINN* algorithm, the greater the degree of parallelization we obtain with the same p.

In case $x_{\hat{k}}$ does not exist (i.e., there are less than $\hat{k}$ objects in *Queue*), we involve just the first node (which is at the top of *Queue*). Once $x_{\hat{k}}$ appears in *Queue*, the parallelization is used again.

Another choice, in case $x_{\hat{k}}$ does not exist, is to use, in place of $d(x_{\hat{k}}, q)$, the distance from the query of the last object in *Queue*. In this case the operation would became more parallel but also more expensive considering its total cost. The degree of parallelization of the *DINN* is also related to the number of nodes present in *Queue*. Thus, it is important to have more than only the next most promising node N_n (see Assumption 2) in *Queue*. Different strategies can be used to efficiently put nodes in *Queue* depending on the specific data structure that is used. In our implementation of the *DINN* over the *MCAN*, we decided to put in *Queue* the neighbors of every involved node.

5. DINN over MCAN

The *MCAN* [8,9] is a scalable distributed similarity search structure for metric data. Extending the Content-Addressable Network (CAN), which is a well-known Distributed Hash Table, *MCAN* is able to perform distributed similarity searches between objects assuming that the objects, together with the used distance, are metric. For a complete description of *MCAN* see [8]. A comparison of *MCAN* with similar distributed similarity search structure for metric data can be found in [3].

MCAN satisfies Condition 1 which guarantees Assumption 2 as demonstrated in Theorem 4 (see Section 4.3.3). In fact, it can be proved that in *MCAN* if a node N_i is neighbor of a node N_j that is closer to the query than N_i and $\delta(N_j, q) > 0$, then N_j is also neighbor of at least one other node which is closer to the query than N_j. In other words *MCAN* satisfies Condition 1 and thus also Assumption 2. In fact, given a set of nodes $\mathcal{N}^*$ *downward closed* with respect to q, the node N_n is always between the neighbors of at least a node $N_j \in \mathcal{N}^*$ (Theorem 4).

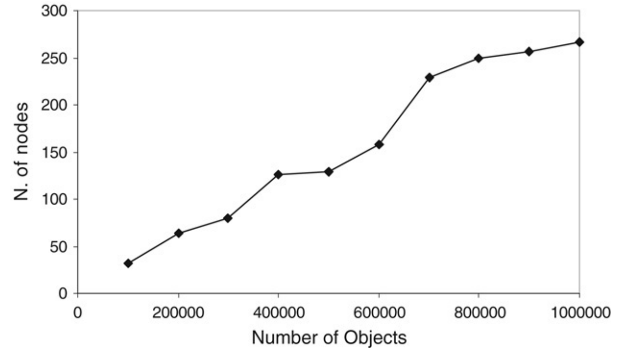


Fig. 3. N. of nodes as dataset grows.

5.1. Experimental results

Experiments have been conducted using a real-life dataset of 1,000,000 objects using real nodes in a LAN network. Each object is a 45-dimensional *vector* of extracted color image features. The similarity of the vectors was measured by a *quadratic-form distance* [13]. The same dataset has been used for [9,3,12,1,4]. The dimensionality used for the *MCAN* is 3 as in [9]. All the presented performance characteristics of query processing have been taken as an average over 100 queries with randomly chosen query objects.

To study scalability with respect to the number of objects, we limited the number of objects each node can maintain (the same has been done in [2,8,9,3,12,4]). When a node exceeds its space limit it splits by sending a subset of its objects to a free node that takes the responsibility for a part of the original region. Note that, limiting the number of objects each node can maintain, we simulate the simultaneous growing of dataset and number of nodes. In Fig. 3 we show the number of nodes as the dataset grows.

The parallelization and the number of messages reduction are tuned varying respectively parameter p, defined in Definition 8, and k^+ (i.e., the objects requested at each invocation of the *DINN* algorithm). As described in Section 4.4, the more the objects (k^+) we request at each invocation, the greater degree of parallelization we obtain with the same p.

Usually evaluation methodologies of metric space access methods are based on the number of distance computations. However, to give a fair performance evaluation, we base our evaluation on the number of *local-INN* invocations. This evaluation approach has the advantage to be independent of the particular *local-INN* implementation. Furthermore, different nodes could even have different *local-INN* implementations. We use the following two characteristics to measure the computational costs of a query:

- *total number of* local-INNs — the sum of the number of *local-INN* invocations on all involved nodes,

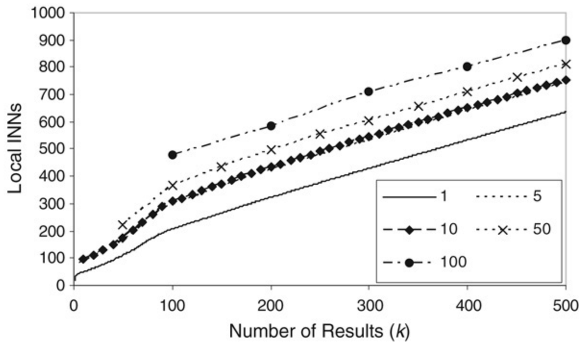


Fig. 4. N. of *local-INN* invocations for different k^+ (parallelization parameter $p = 0$).

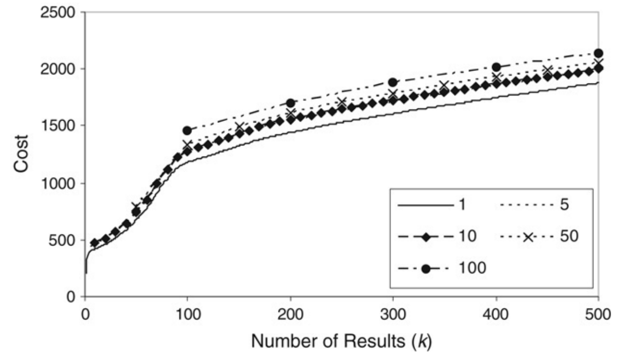


Fig. 6. Estimated cost for different k^+ ($p = 0$).

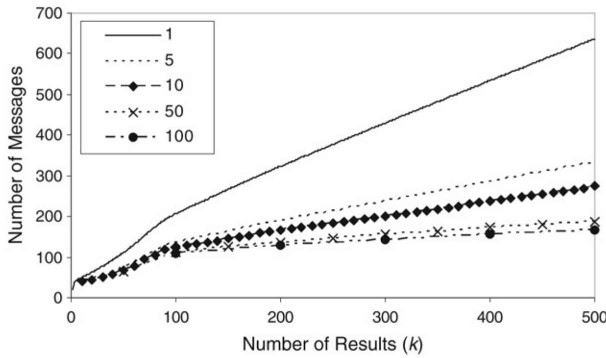


Fig. 5. N. of messages for different k^+ ($p = 0$).

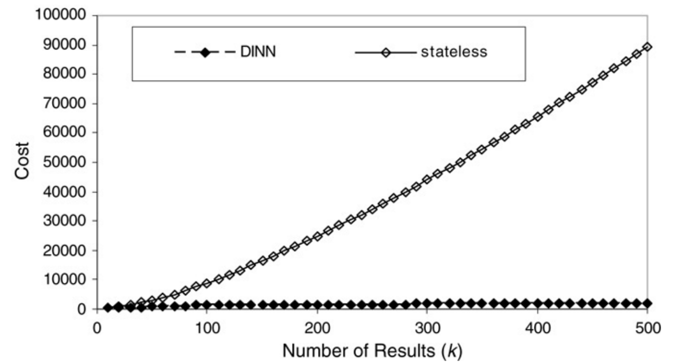


Fig. 7. Total estimated costs ($p = 0, k^+ = 10$).

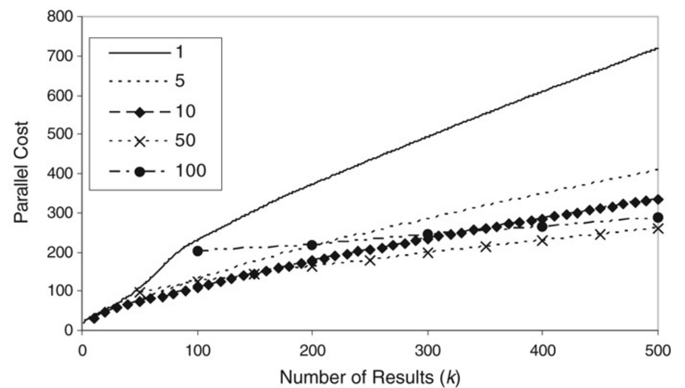
- *parallel computations* — the maximum number of *local-INN* invocations performed in a sequential manner during the parallel query processing.

Note that the total number of *local-INNs* corresponds to the cost on a centralized version of the specific structure while the parallel computations, together with the number of messages, directly effects the response time.

In Fig. 4 we show the total number of *local-INNs* for $p = 0$ (i.e., no parallelization) for different k^+ as function of the number of results k. Note that, to obtain the same number of results k varying k^+, we need $\lceil k/k^+ \rceil$ *DINN* invocations. While increasing k^+ does not seem worthwhile since the total number of *local-INNs* increases, the advantage of greater k^+ is evident observing the number of messages exchanged during the *DINN* execution in Fig. 5. In fact, as said in Section 4.4, increasing k^+, we can reduce the number of messages.

Since obtaining the first result from a *local-INN* in an arbitrary node is significantly more expensive than obtaining the next ones, a more realistic approach is to consider the cost of the first result of a *local-INN* as several times the cost of subsequent *local-INN* invocations. In Fig. 6 we report the same result of Fig. 4, but assuming that the first invocation cost of a *local-INN* is 10 times the cost of subsequent invocations. In this case the gap between the graphs for different k^+ remains but it decreases. Note that, since in this case there is no parallelization, there is no difference between the parallel and total cost.

In Fig. 7 we show the estimated cost for retrieving up to 500 objects, 10 by 10 (i.e., $k^+ = 10$) comparing the defined *DINN*



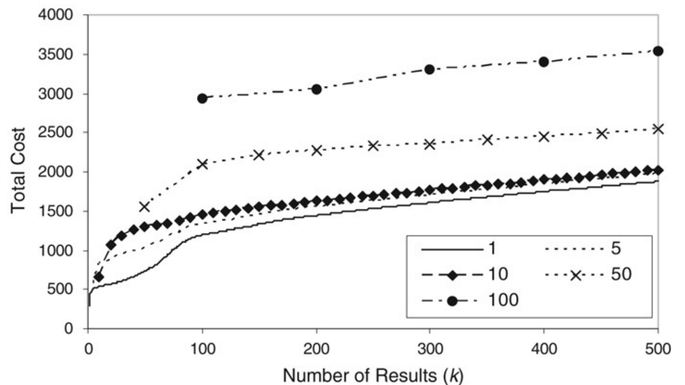
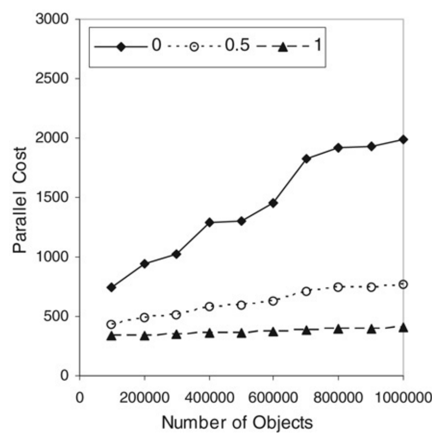
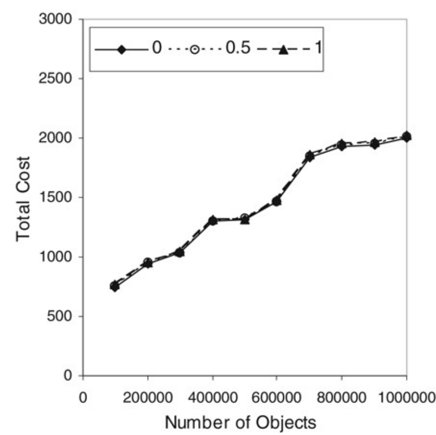


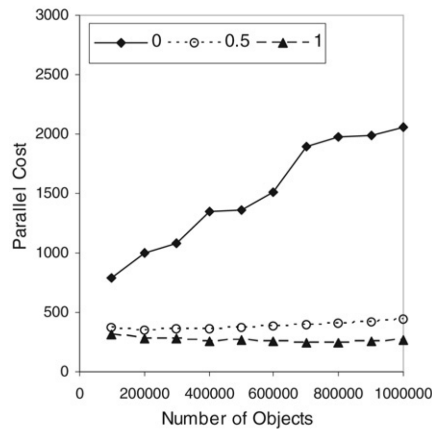
Fig. 8. Parallel and total estimated costs for different k^+ ($p = 1$).

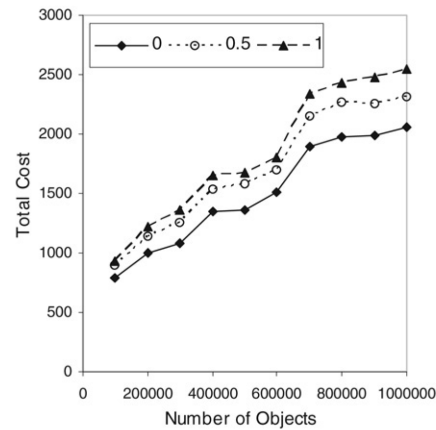
(a) $k^+ = 1$.





(b) $k^+ = 10$.





(c) $k^+ = 50$.

Fig. 9. Parallel and total Estimated Costs for obtaining 500 results for various values of the parameter p. Each subfigure reports the result presented obtained using different k^+.

with a *stateless* execution of the *DINN* in which after searching first 10 objects we destroy *Queue* and then we ask for the next 10 objects (thus requesting a 20-NearestNeighbor search from scratch) and so on. Here we want to underline that the use of an Incremental Nearest Neighbor algorithm when the number of desired neighbors is unknown in advance is mandatory to preserve efficiency. In fact the cost of retrieving the next k^+ once a given number of results has already been retrieved using a stateless approach is prohibitive.

Let us now consider the parallelized version of the *DINN* defined in Section 4.5. In Fig. 8 we compare the total and parallel cost when $p = 1$ (i.e., maximizing the parallelization). The graph of the parallel cost demonstrates the advantage of the parallel execution. Observing for instance $k = 100$ for the