Department of Electronic and Computer Engineering
Technical University of Crete

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science (MSc) at the Technical University of Crete

# Branch-and-bound nearest neighbor searching over unbalanced trie-structured overlays

Michail ARGYRIOU

The committee is consisted of:

Vasilis SAMOLADAS (supervisor)
Euripides G.M. PETRAKIS
Stavros CHRISTODOULAKIS

January 22, 2013

# Contents


**1 Introduction** **1**
- 1.1 Distributed nearest neighbor searching algorithms . . . 2
- 1.2 Building a data structure with emphasis on p2p environment and nearest neighbor queries 3
  - 1.2.1 Type of data . . . 3
  - 1.2.2 Type of queries . . . 3
  - 1.2.3 Dimensionality . . . 6
  - 1.2.4 Data Organization (aka Space Partition) . . . 6
  - 1.2.5 Is the database static or dynamic? . . . 8
  - 1.2.6 Storage Environment . . . 8
  - 1.2.7 Fault tolerance . . . 8
  - 1.2.8 Is the storage environment static or dynamic? . . . 9
  - 1.2.9 Distance function . . . 9
  - 1.2.10 Traversal Strategies over search hierarchies . . . 12
  - 1.2.11 Parallelism . . . 17
- 1.3 P2P Data Structure Frameworks . . . 17
  - 1.3.1 VBI . . . 18
  - 1.3.2 PGrid . . . 19
- 1.4 Outline . . . 20

**2 Related Work** **22**
- 2.1 GRaSP . . . 23
  - 2.1.1 Trie-structured networks . . . 23
  - 2.1.2 Analysis of trie-structured networks . . . 25
  - 2.1.3 Network maintenance . . . 25
  - 2.1.4 Applications of GRaSP . . . 27

**3 Nearest Neighbor Search** **29**
- 3.1 Hierarchical binary space partition . . . 29
- 3.2 K-nn Search Algorithm . . . 29
- 3.3 Data Updates . . . 31
- 3.4 Analysis of trie-structure networks . . . 33
  - 3.4.1 Latency Complexity Theorem . . . 33

**4 Performance Evaluation** **37**
- 4.1 Simulator . . . 37
- 4.2 Modeling P2P Network Performance . . . 37
  - 4.2.1 Maximum Throughput . . . 38

4.2.2 Fringe Size . . . . . 38
4.2.3 Fairness Index . . . . . 38
4.2.4 Latency . . . . . 38
4.3 Experimental Evaluation . . . . . 39
4.3.1 Test workloads . . . . . 39
4.3.2 Results for low dimensions . . . . . 40
4.3.3 Results for medium dimensions . . . . . 41
4.3.4 Results for high dimensions . . . . . 41

**5 Conclusion and Future Work 48**

# Chapter 1

# Introduction

Searching is one of the fundamental problems in Computer Science. We "search" for objects when we search for a web page related to a particular issue of interest (eg give me a page showing the GDP in Greece), when we make a hotel reservation (eg is there a free hotel room in Rome on 31 Dec?), when we buy an air ticket (eg is there an available seat on 31 Dec from Chania to Athens and at what price?), when we browse a digital map (show the map with center in Chania and radius 10km) or even when we withdraw money from an ATM (eg return my bank account). Searching is everywhere! Due to all these numerous application domains search has to deal with diverse data and query types.

Intuitively in order to retrieve objects we set a criterion and ask which data satisfy it. A naive approach would be asking all the objects sequentially if they satisfy the criterion, rank them by their relevance and return the most relative one (or more answers). Obviously this wouldn't be efficient. Therefore we pre-organize objects into a *data structure* (also called *index*) and ask the index in question. Notions such as relevant, ranking, index and efficiency arise in the search context.

Of particular interest for this thesis is the type of query that asks for the $k$ answers in a question given that the objects are stored in a pool of distributed peers. This is called a distributed *k-NN query* and such an example of such a query is presented on Figure 1.1. In this thesis we will extend an existing framework for constructing data structure called *GRaSP* which is presented in §2.1. GRaSP is a framework for constructing indices in P2P networks where data are stored in many computers (*peers*) and therefore a query has to traverse (*forwarded*) many peers (*hops*). At the moment GRaSP has been instantiated with rectangular and 3-sided queries (more on these types of queries in §2.1).

On the remaining introduction we will refer on issues regarding the construction of a distributed data structure for k-NN queries. Our motive will be exhibiting the inherent difficulties of designing a solution for such a problem and the design choices among which we will have to choose. Of special interest in the problem of space dimensionality and the organization of the space among peers, i.e. the space partition.

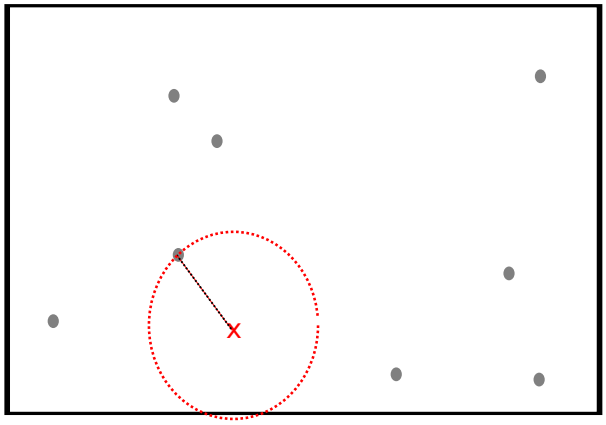


Figure 1.1: Example of a 1-nn query. Among all the candidate answer points the nearest point to the query is the one shown on the figure.

## 1.1 Distributed nearest neighbor searching algorithms

In this section we will present the literature for k-NN search on a p2p environment.

A naive approach would be a central peer (manager node) collecting all the data and locally performing k-NN classification. This has been performed to sensor networks[44]. But sensors are nodes of low-capabilities. Moreover they have energy constraints and therefore they cannot process large volume of data. On the contrary the common policy is transmitting measurements to a pc which does all the hard processing. Therefore this algorithm is not advised for sensor networks.

GHT*[12, 65] assumes a metric space for data and queries. It supports point, range and k-NN queries[11]. On the other hand it doesn't support deletions. Moreover, its performance is dependent on (a) the underlying tree structure which in turn is very sensitive to pivot selection[1], and more importantly (b) by consistent updating peer's state (locally saved trees called AST).

[47] proposes a k-NN algorithm over CAN[3]. Its search algorithm is similar to ours and is based on a typical branch-and-bound technique where the neighboring peers are asked and if they contain a candidate answer recursively.

[59] extends the distributed quadtree-based index[34, 58] by adding k-NN capabilities. It partitions the space using a quadtree. Each quadtree block is uniquely identified by its centroid, named as a control point. Then each control point is mapped to a DHT, such as Chord. The search algorithm similar to ours but parallelizes the search. The drawback is that each query starts from the root peer and this can cause a bottleneck in the system.

[1] For the definition of pivots and Distance-Based Indexing see §1.2.4

## 1.2 Building a data structure with emphasis on p2p environment and nearest neighbor queries

Designing a data structure to be efficient is not an easy task. Generally we cannot organize data in such a manner that all queries can be optimally answered. Therefore we construct custom data structures in relation to the problem in question. Accordingly, a Binary Search Tree is most suitable for searching 1-d data in memory, kd-tree for searching m-d data in memory, B-tree for search m-d data in disk, etc. From now on we present the design dimensions that should be taken into account for designing a data structure with emphasis on a distributed environment and nearest neighbor queries.

The most notable factors that should be taken into account when designing a data structure are the following:

### 1.2.1 Type of data

Data can be *discrete* or *continuous*. A type of data is discrete if there are only a finite number of values possible. These values can be *numerical*, e.g. the line of integer numbers, or *quantitative*, e.g. a boolean value (true or false) or a string. A type of data is continuous when it is defined within a range, e.g. temperature, length, the line of real numbers, etc. Moreover, the domain of a data type can be *bounded* (finite) or not. We are especially interested in data that have *extent* i.e., occupy space (the so called *spatial data*). Such are usually found in pattern recognition and image databases. Typical objects are lines (roads, rivers), intervals (time, space), regions of varying shape and dimensionality (e.g., lakes, countries, building, crop maps, polygons, polyhedra), and surfaces. Objects may be disjoint or could even overlap. Sometimes, objects are abstracted by *feature vectors* usually selected by experts in the application domain. Typical features are color, color moments, textures, shape descriptions, and so on, expressed using scalar values.

If feature vectors cannot be identified for objects then there are two alternatives assuming all we have is the *distance function* among objects (see [36]). The first one is *distance-based indexing*, where we compute distances of objects from a few selected objects (called *pivots* or *vantage points*) and use their distances to sort and index data. The second approach we can can follow if we cannot extract feature vectors from objects are the *embedding methods*. Given $N$ objects in a set $U$, the goal is to choose a value of $k$ and compute a set of $N$ corresponding points in a $k$-dimensional space, via an appropriate mapping $F$ that is applicable to all elements of $U$ and thus also to the query objects $q$, so that the distance between the $N$ corresponding points, using a suitably defined distance function $\delta$, is as close as possible to that given by the original distance function $d$ for the $N$ objects. For an alternative definition and use of the embedding methods in dimensionality reduction see Section 1.2.3.

### 1.2.2 Type of queries

Queries that have extent and arbitrary shape are called *similarity queries* in order to distinguish them from the usual queries found in databases. Most of the taxonomy here is

taken from [28]. On the following definitions we assume that the set of objects (*dataset*) is called $E^d$ and is $d$-dimensional and each object $o$ has extent $o.G$.

**Ranking Queries** Given a query object $q$, report all the objects $o \in E^d$ in order of distance from $q$, subject to some stopping condition.

**Exact Match Query EMQ, Object Query[28]** Given an object $o'$ with spatial extent $o'.G \subseteq E^d$, find all objects $o$ with the same spatial extent as $o'$.

$$EMQ(o') = \{o | o'.G = o.G\}$$

**Point Query PQ[28]** Given a point $p \in E^d$, find all objects $o$ overlapping $p$.

$$PQ(p) = \{o | p \cap o.G = p\}$$

The point query can be regarded as a special case of several of the following queries, such as the intersection query, the window query, or the enclosure query.

**Window Query WQ, Range Query[28]** Given a $d$-dimensional interval $I^d = [l_1, u_1]x[l_2, u_2]x...x[l_d, u_d]$ where $\{l_i \leq l_j | i < j\}$, $\{u_i \leq u_j | i < j\}$ and $\{l_i \leq u_i\}$, find all objects o having at least one point in common with $I^d$.

$$WQ(I^d) = \{o | I^d \cap o.G \neq \emptyset\}$$

The query implies that the window is *iso-oriented*, i.e. its faces are parallel to the coordinate axes. A more general variant is the region query that permits search regions to have arbitrary orientations and shapes.

**Three-sided Range Query** A *3-sided query* is a special case of 2-d range query, where the keys are planar points and a range is a rectangle bounded from 3 sides only; a range is determined by parameters $(a, b, c)$, and reports all keys $(x, y)$ with $a \leq x < b$ and $y < c$.

**Intersection Query IQ, Region Query, Overlap Query[28]** Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o having at least one point in common with o'.

$$IQ(o') = \{o | o'.G \cap o.G \neq \emptyset\}$$

**Enclosure Query EQ[28]** Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o enclosing o'.

$$EQ(o') = \{o | (o'.G \cap o.G) = o.G\}$$

**Containment Query CQ[28]** Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o enclosed by o'. $CQ(o') = \{o|(o'.G \cap o.G) = o.G\}$ The enclosure and the containment are symmetric to each other. They are both more restrictive formulations of the intersection query by specifying the result of the intersection to be one of the inputs.

**Adjacency Query AQ[28]** Given an object o' with spatial extent $o'.G \subseteq E^d$, find all object o adjacent to o'.

$$AQ(o') = \{o|o.G \cap o'.G \neq \emptyset \wedge o'.G^o \cap o.G^o = \emptyset\}$$

Here, $o'.G^o$ and $o.G^o$ denote the interiors of the spatial extents $o'.G$ and $o.G$, respectively.

**Nearest Neighbor Query NNQ[28],** ***post-office problem*****[40]** Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o having a minimum distance from o'.

$$NNQ(o') = \{o|\forall o'' : dist(o'.G, o.G) \leq dist(o'.G, o''.G)\}$$

Distance between extended spatial data objects is usually defined as the distance between their closest points. Common distance functions for points include the Euclidean and the Manhattan distance.

**Spatial Join[28, 33]** Given two collections R and S of spatial objects and a spatial predicate $\theta$, find all pairs of objects $(o, o') \in RxS$ where $\theta(o.G, o'.G)$ evaluates to true.

$$R \bowtie_\theta S = \{(o, o')|o \in R \wedge o' \in S \wedge \theta(o.G, o'.G)\}$$

As for the spatial predicate $\theta$, a brief survey of the literature [50, 51, 45, 27, 19, 8, 14] including $intersects()$, $contains()$, $isenclosedby()$, $distance()\Theta q$ (where $\Theta \in \{=, \leq, <, \geq, >\}$ and $q \in E^1$), $northwest()$, $adjacent()$ and $meets()$.

A closer inspection of these spatial predicates shows that the intersection join $R \bowtie_{intersects} S$ plays a crucial role for the computation in virtually all of those cases. For predicates such as contains, encloses, or adjacent, for example ,the intersection join is an efficient filter that yields a set of candidate solutions typically much smaller that the Cartesian product $RxS$.

**Generalized Range Search** Formally the *Generalized Range Search* problem is defined as following. Assume we denote the search space with the symbol $U$ (infinite set), the key space with $\mathcal{K} \subseteq 2^U$ (keys are subsets of $U$) and the range space with $\mathcal{R} \subseteq 2^U$ (ranges are subsets of $U$). Then assuming that the dataset $K \subset \mathcal{K}$ is stored on a network and given a general range query $r \subseteq R$ we want all the keys $A_K(r) = \{k \in K|k \cap r \neq \emptyset\}$ that answer it. Note that instances of the Generalized Search Problem are the problems of EMQ (§1.2.2), PQ (§1.2.2), WQ (§1.2.2), IQ (§1.2.2), CQ (§1.2.2)

**k-Nearest Neighbor Query, top-k Selection Query** Given a set $S$ of $n$ sites and a query point $q$, find a subset $S' \subseteq S$ of $k \leq n$ sites such that for any sites $p_1 \in S'$ and $p_2 \in S - S'$, $dist(q, p_1) \leq dist(q, p_2)$.

The k-nearest neighbor searching is used in a variety of applications, including knowledge discovery and data mining[60], CAD/CAM systems[42] and multimedia database [15].

Figures 1.2(a)-1.2(f) give some concrete examples.

### 1.2.3 Dimensionality

In respect to the *multi-dimensionality* [18] classifies the effects as follows: (1) Geometric Effects: when the dimension of a space increases, its volume is increased. In order to sample a high dimensional space, we need an exponentially growing number of points. This is called the *curse of dimensionality*[15]. (2) Effects of Partition: As the dimension increases the space partitioning becomes coarser. (3) Databases Environment: The query distribution is affected as the dimensionality of the data space increases. Let us add to all these the fact that there is no total order that preserves spatial proximity. Therefore we have to tackle with data that have no crisp hierarchical organization.

All this makes searching in high dimensions a hard problem we would like to reduce to a low dimensional representation. Of course heuristic solutions have been developed but none gives efficient and qualitative query results. These exploit the fact that fortunately the "inherent dimensionality" of a dataset is often much lower than the dimensionality of the underlying space. Therefore *dimensionality reduction techniques* (also called *embedding methods*; see §1.2.1) have been developed. Dimensionality reduction techniques work as following: given $N$ vectors in $n$ dimensions find the $k$ most important axes to project them, where $k \ll n$ and not necessarily belongs to $N$ ($k$ is user defined). Most prominent examples are the *space filling curves* (e.g. *z-ordering*[2] and *Hilbert curves*), *Eigenvalue analysis*[52] (using *Singular Value Decomposition* or *Karhunen-Loeve transform*), *FastMap*[26], *Principal Component Analysis* (*PCA*), *Independent Component Analysis* (*ICA*) *Multidimensional Scaling* (*MDS*), *Isomap*, etc. On Figure 1.2.3 we show exemplary how the FastMap reduces the number of dimensions and on Figure 1.2.3 we depict a few well-known space filling curves.

### 1.2.4 Data Organization (aka Space Partition)

We usually cluster the data into sets so that we exploit the shape of the query. For example if the query is an 1-d interval and the data are 1-d points and we look for all the data that the query intersects then it would be better if we divided the space into intervals. The choice for the partition of the data is called (*space partition*). Instead of partitioning the original space we can alternatively map the original space into other (usually of fewer dimensions) one and partition it. The motive may not be dimensions reduction like §1.2.3 but exploiting the data correlation. For example in a 2-d Cartesian space where all the data are lied on the x-axis we can map each point $(x, y)$ into $x$ (i.e. throw the y-coordinate)

(a) Point Query (b) Window Query (c) Intersection Query

(d) Enclosure Query (e) Containment Query (f) Adjacency Query

Figure 1.2: Queries[28]

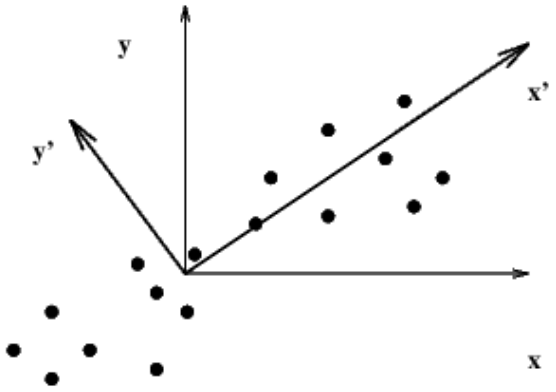



Figure 1.3: Dimensionality reduction with FastMap. FastMap embeds the 2-d points on xy in a lower-dimensional 1-d vector space onto x'-axis and then make use of a filter-and-refine algorithm in conjunction with multidimensional indexing methods to prune irrelevant candidates from consideration.

and get same results with less complexity.

Usual space partitioning in *distance-based indexing* are the *Ball-partitioning* (see Figure 1.5(a)) if we choose one object and the *Generalized-hyperplane partitioning* (see Figure 1.5(b)) if we choose two objects. Both create hierarchical partitionings.

### 1.2.5 Is the database static or dynamic?

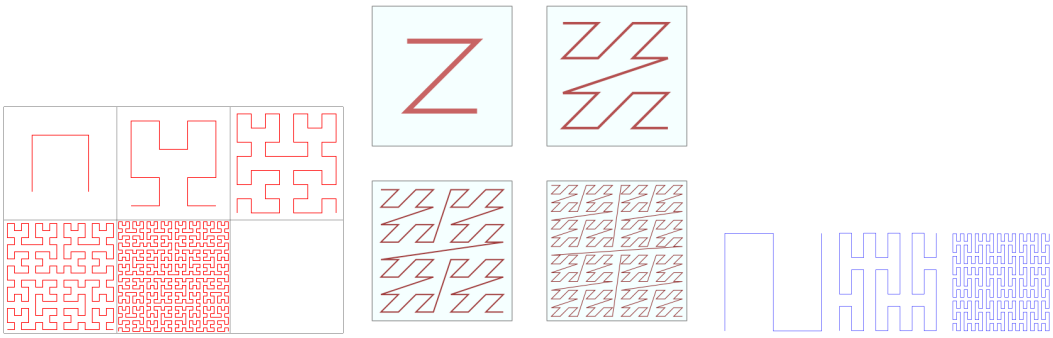
If we allow data insertions or/and deletions then the data structure may need constant reorganizing, e.g. if it the data structure is stored on disk and is a tree hierarchy then a leaf can exceed the max capacity ratio and need splitting into two leaves. Moreover if the tree is balanced it may need subtree rotations in order to remain balanced.

### 1.2.6 Storage Environment

Should the data structure be on the main memory, secondary memory of distributed? (deciding this we must make assumptions such as if the data fit in memory or if they are distributed or centralized).

### 1.2.7 Fault tolerance

Should the data structure be tolerant to failures, e.g. disk or peer failures. A common approach followed is inserting replication - the more replication the more tolerant the data



(a) Hilbert curve (b) Z order curve (c) Peano curve

Figure 1.4: Well-known space filling curves. The motive is that if two objects are located close together in original space, there should at least be a high probability that they are close together in the total order, i.e. in the one-dimensional image (embedded) space. For the organization of this total order one could then use a one-dimensional access method. Images taken from Wikipedia.

structure. Data structure reorganization may still be needed in case of failure.

### 1.2.8 Is the storage environment static or dynamic?

If the data structure is distributed and peer additions/removals are allowed then spreading updating the state of the rest of the peers may be necessary.

### 1.2.9 Distance function

The distance function is used to compute the distance among data, queries and data regions. Another use of a distance function is to derive feature vectors based on the inter-object distances (e.g. by using *FastMap*[26]).

#### Notation (Metric Space, Distance Function)

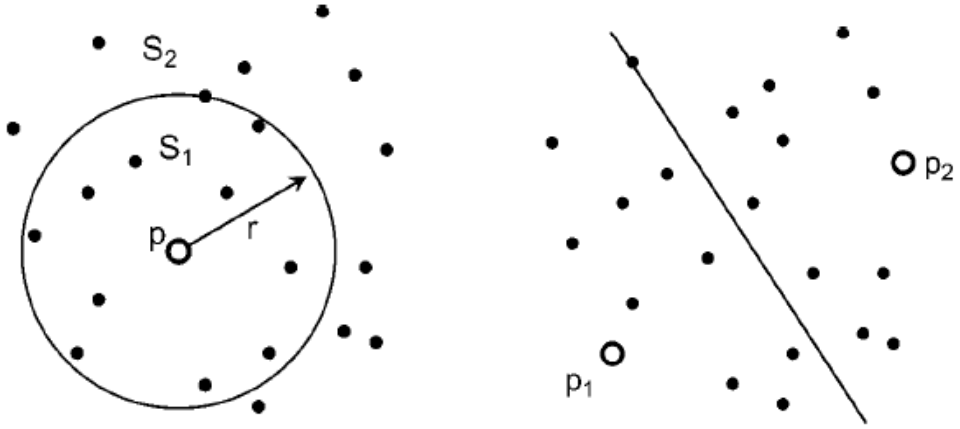
Our thesis is based on searching over metric spaces. A *metric space* $M = (D, d)$, is defined for a domain of object (or the objects' keys or indexed features) $D$ and a *distance function* $d$.

If d is a metric $(S, d)$ then the distance function $d$ must satisfy the following three properties, where $o_1, o_2, o_3 \in S$:

**symmetry** $d(o_1, o_2) = d(o_2, o_1)$

**nonnegativity** $d(o_1, o_2) \geq 0, d(o_1, o_2) = 0 iff 0_1 = o_2$

**triangle inequality** $d(o_1, o_3) \leq d(o_1, o_2) + d(o_2, o_3)$




(a) Ball Partitioning (b) Generalized Hyperplane Partitioning

Figure 1.5: Distance-based indexing. On (a) there is one pivot $p$ with radius $r$ and objects are distributed in two buckets, one containing all the objects in the circle and another one containing all the objects out of the circle. On (b) there are two pivots $p1$ and $p2$; objects belong to their closes pivot. Images taken from [36].

Of the distance metric properties, the triangle inequality is the key property for pruning the search space when processing queries.

### Common Distance Functions

Hereafter will present the most common distance functions found in the literature.

**Minkowski distances** The *Minkowski distance* of order p between two points $P = (x_1, x_2, ..., x_n)$ and $Q = (y_1, y_2, ..., y_n) \in R^n$ is defined as: $L_p = (\sum_{i=q}^{n} |x_i - y_i|)^{1/p}$. For $L_1$ is called *Manhattan* (also *City-Block distance*), for $L_2$ *Euclidean* and for $L_\infty = \lim_{p\to\infty}(\sum_{i=q}^{n} |x_i - y_i|)^{1/p} = max_{i=1}^{n}|x_i - y_i|$ is called *Chebyshev distance* (i.e. maximum distance or infinite distance or chessboard distance).

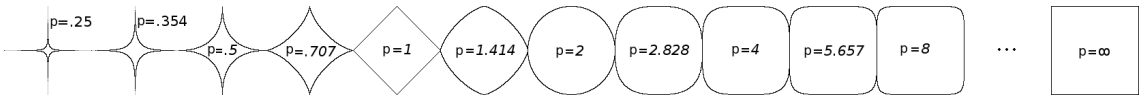



Figure 1.6: Unit circles with various values of p for the Minkowski distance.[wikipedia]

**Quadratic Form Distance** If $M$ is symmetric positive definite

$$d_M(\overrightarrow{x}, \overrightarrow{y}) = \sqrt{(\overrightarrow{x} - \overrightarrow{y})^T \cdot M \cdot (\overrightarrow{x} - \overrightarrow{y})}$$

**Edit Distance** The *edit distance*[62] between two strings of characters is the number of operations required to transform one of them into the other.

The distance between two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$ is defined as the minimum number of atomic edit operations (insert,delete, and replace) needed to transform string $x$ into string $y$. The atomic operations are defined formally as follows:

**insert** the character $c$ into the string $x$ at the position $i$: $ins(x, i, c) = x_1 x_2 \cdots x_i c x_{i+1} \cdots x_n$

**delete** the character at the position $i$ from the sting $x$: $del(x, i) = x_1 x_2 \cdots x_{i-1} x_{i+1} \cdots x_n$

**replace** the character at the position $i$ in $x$ with the new character $c$: $rep(x, i, c) = x_1 x_2 \cdots x_{i-1} c x_{i+1} \cdots x_n$

There are several different ways to define an edit distance, and there are algorithms to calculate its value under various definitions:

- Hamming (exact) distance
- Longest common subsequence problem
- Levenshtein distance[43]
- Damerau-Levenshtein distance
- Jaro-Winkler distance
- Wagner-Fischer edit distance
- Ukkonen's algorithm
- Hirschberg's algorithm

**Tree Edit Distance** *Tree Edit Distance* was introduced by [57] as a generalization of the well-known string edit distance problem [62] for the problem of comparing trees. This occurs in diverse areas such as structured text databases like XML[61, 22], computer vision, compiler optimization, natural language processing, computational biology and analysis of RNA molecules in computational biology[63]. For a survey on tree edit distance and related problems see [17].

Assuming a pair of ordered rooted trees $(A, B)$ the tree edit distance function equals minimal cost to transform source tree A into target tree B assuming the following edit operations:

**insert** inserting a node

**delete** deleting a node

**replace** replacing a node

**Jaccard's Coefficient** *Jaccard's Coefficient* is a similarity measure applicable to sets.

Given two sets A and B, *Jaccard's distance coefficient* is defined as the size of the intersection divided by the size of the union of the sample sets:

$$d_\delta(A, B) = 1 - \frac{A \cap B}{A \cup B}$$

Likewise *Jaccard's similarity coefficient* is defined as $d(A, B) = 1 - d_\delta(A, B)$.

As an example of an application that deals with sets, suppose we have access to a log file of web addresses (URLs) accessed by visitors to an Internet Cafe. Along with the addresses, visitor identifications are also stored in the log. The behavior of a user browsing the Internet can be expressed as the set of visited network sites and Jaccard's distance coefficient can be applied to assess the dissimilarity (or similarity) of individual users' search interests.

**Hausdorff Distance** Similar to Jaccard's coefficient measures set similarity but using element pair-wise distance function $d_e$ is defined. Specifically, the *Hausdorff distance*[37] is defined as follows.

$$d_H(X, Y) = max\{\sup_{x \in X} \inf_{y \in Y} d_e(x, y), \sup_{y \in Y} \inf_{x \in X} d_e(x, y)\}$$

Informally, two sets are close in the Hausdorff distance if every point of either set is close to some point of the other set. The Hausdorff distance is the longest distance you can be forced to travel by an adversary who chooses a point in one of the two sets, from where you then must travel to the other set.

A typical application is the comparison of shapes in image processing, where each shape is defined by a set of points in a 2-dimensional space.

### 1.2.10 Traversal Strategies over search hierarchies

#### Branch and Bound

Nearest neighbor searching algorithms (distributed or not) generally partition the space into non-overlapping regions and assign them to blocks or peers. The partition formed is usually searched by *branch and bound algorithms*. Branch and bound consists of a systematic enumeration of all candidate solutions (*branch step*), where large subsets of fruitless candidates are discarded en masse (*pruning step*), by using upper and/or lower estimated bounds of the quantity being optimized.

Roussopoulos et. al. [54] proposed a branch-and-bound algorithm for querying spatial points storing in an R-tree. Meanwhile, they introduced two useful metrics: *MINDIST*

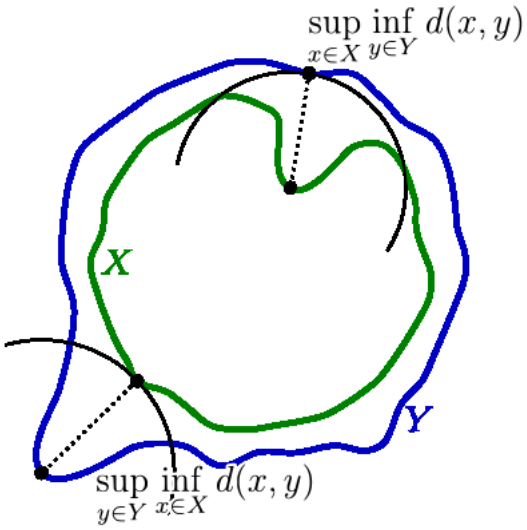



Figure 1.7: Components of the calculation of the Hausdorff distance between the green line X and the blue line Y (image taken from `http://en.wikipedia.org/wiki/Hausdorff_distance`)

and *MINMAXDIST* for ordering and pruning the search tree. The algorithm is briefly described as following.

1. Maintain a sorted buffer of at most k current nearest neighbors.
2. Initially set the search bound to be infinite.
3. Traverse the R-tree, always lower the search bound to the distance of the furthest nearest neighbor in the buffer, and prune the nodes with MINDIST over the search bound, until all nodes are checked.

This is a typical brand-and-bound algorithm for k-NN searching.

While traversing the search hierarchy (tree) we have many options which node to visit next. The choice plays a crucial role to the stopping time of our algorithm because finding fast the kth answer can rapidly prune candidate answers and save us from visiting a large part of the tree. On the following paragraph we describe two general traversal strategies over trees.

There are numerous ways of performing the search for knn queries, primarily depending on how the search hierarchy is traversed. [36] presents two algorithms that use two different traversal orders. The first algorithm makes use of *depth-first traversal*. The second algorithm, on the other hand, uses *best-first traversal*, which is based on the distances and, in a sense, breaks free of the shackles of the search hierarchy. We will study and compare these two strategies here in order we choose one for our search algorithm.

**Depth-First Algorithm**

The algorithm in listed on Figure 1.8(a). The key idea is assuming an initial maximum distance equal to $\infty$ until at least $k$ objects have been seen, and from then on is set to the

```
DFNEAREST(q, k, T)
 1 e ← root of the search hierarchy T
 2 NearestList ← NEWLIST(k) }/* a list for accumulating result */}
 3 NEARESTTRAVERSAL(q, NearestList, e)
 4 return NearestList

NEARESTTRAVERSAL(q, NearestList, e)
 1 ActiveBranchList ← child elements of e
 2 SORTBRANCHLIST(ActiveBranchList, q)
 3 for each element e' in ActiveBranchList do
 4   if d_t[e'](q, e') > MAXDIST(NearestList) then
 5     Exit loop
 6   if t[e'] = 0 then }/* e' is an object */}
 7     INSERT(NearestList, e', d_0(q, e'))
 8     /* element with largest distance is removed */
 9     /* if already k objects in list */
10   else
11     NEARESTTRAVERSAL(q, NearestList, e')
```

(a) Deapth First k-NN algorithm

```
BFNEAREST(q, k, T)
 1 Queue ← NEWPRIORITYQUEUE()
 2 NearestList ← NEWLIST(k)
 3 e ← root of the search hierarchy T
 4 ENQUEUE(Queue, e, 0)
 5 while not ISEMPTY(Queue) and
 6       MINDIST(Queue) < MAXDIST(NearestList) do
 7   e ← DEQUEUE(Queue)
 8   for each child element e' of e do
 9     if d_t[e'](q, e') ≤ MAXDIST(NearestList) then
10       if t[e'] = 0 then /* e' is an object */
11         INSERT(NearestList, e', d_t[e'](q, e'))
12         /* element with largest distance is removed */
13         /* if already k objects in list */
14       else
15         ENQUEUE(Queue, e', d_t[e'](q, e'))
16 return NearestList
```

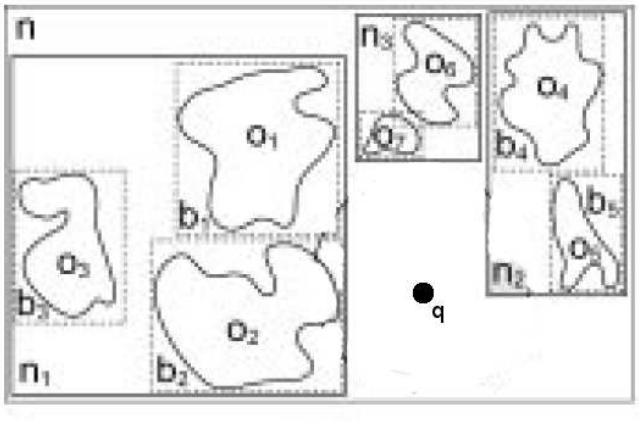
(b) Best First k-NN algorithm

Figure 1.8: (a) A depth-first k-nearest neighbor algorithm on a search hierarchy T given a query object q. (a) A best-first k-nearest neighbor algorithm on a search hierarchy T given a query object q.

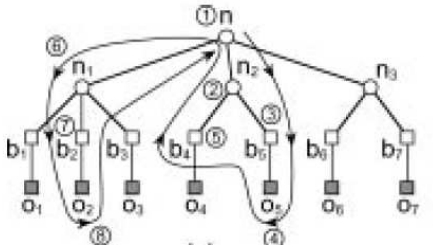
kth smallest distance seen so far. Clearly, the value of $\epsilon$ converges quicker to the distance of the $k$th nearest neighbor of $q$ if we see objects that are close to $q$ as early as possible. A heuristic that aims at this goal is such that at each non leaf element $e$ that is visited, we visit the children of $e$ in order of distance.

In the algorithm, the list $NearestList$ is used to store the $k$ candidate nearest neighbors (i.e., the $k$ objects seen so far that are closest to $q$), similar to the use of $RangeSet$ in the range search algorithm. The expression $MAXDIST(NearestList)$ has the value of the greatest distance among the objects in $NearestList$, or $\infty$ if there are still fewer than $k$ objects in the list. The call to $SORTBRANCHLIST$ in line 2 sorts the children of $e$ in order of distance, which determines the order in which the following for-loop iterates over the children. When the distance from $q$ of a child element is found to be greater than $MAXDIST(NearestList)$ (line 4), the ordering ensures that all subsequent child elements also have distances that are too large, so the for-loop can be terminated.

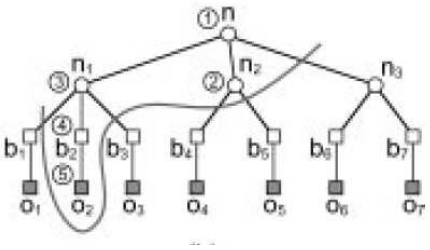
Figure 1.9(b) shows a trace of the algorithm with $k = 1$ on the set of objects (and associated bounding rectangles and nodes) and query object $q$ given in Figure 1.9(a). Initially, $NearestList$ is empty, which implies that $MAXDIST(NearestList)$ in line 4 evaluates to $\infty$. The root node $n$ is naturally the first to be visited during the traversal. The $ActiveBranchList$ computed for $n$ (in line 2) is $\{(n2, 44), (n1, 51), (n3, 80)\}$; the distance values come from Figure 1.9(a). The second node to be visited is then $n2$ , whose $ActiveBranchList$ is $(b5, 85), (b4, 88)$. The bounding rectangle $b5$ is visited next, followed by the associated object $o5$ at a distance of 91, which becomes the first candidate nearest neighbor to be inserted into $NearestList$ (line 7). Backtracking to $n2$, we must now visit the next element on its $ActiveBranchList$, namely $b4$ since its distance value is less than 91 (so the test in line 4 is false), but $o4$ is pruned from the search (via line 5) since $d(q, o4) = 112 > 91$. Thus, the algorithm backtracks to $n$, where the next element on the $ActiveBranchList$ is $n1$, which is promptly visited since its associated distance 51 is less than 91. The $ActiveBranchList$ for $n1$ is $\{(b2, 51), (b1, 63), (b3, 203)\}$, and




(a) Space partition in a dataset which contains objects $o_1$-$o_7$ bounded by Minimum Bounding Rectangles (MBR) and stored in leaf nodes $b1$-$b7$. The distance among the query $q$ and objects $o_1$-$o_7$ are the following: $Dist(q, n) = 0, Dist(q, n_2) = 44, Dist(q, n_1) = 51, Dist(q, b_2) = 51, Dist(q, o_2) = 57, Dist(q, b_1) = 63, Dist(q, n_3) = 80, Dist(q, b_7) = 80, Dist(q, o_7) = 82, Dist(q, b_5) = 85, Dist(q, b_4) = 88, Dist(q, o_5) = 91, Dist(q, o_1) = 98, Dist(q, b_6) = 103, Dist(q, o_6) = 104, Dist(q, o_4) = 112, Dist(q, b_3) = 203, Dist(q, o_3) = 205$.




(b) Depth First k-NN tree traversal




(c) Best First k-NN tree traversal

Figure 1.9: (b) A depiction of the traversal for the depth-first nearest neighbor algorithm for the data and query q of the dataset on (a) for $k = 1$, where the arrows indicate the direction of the traversal and the circled numbers denote order in which the elements are visited (i.e., the leaf and non-leaf elements that have not been pruned). (c) A depiction of the traversal for the best-first nearest neighbor algorithm of Figure 1.8(b) for the same argument values, with the curved line delimiting the portion of the search hierarchy that is visited.

since $d(q, b2) = 51 < 91$, we next visit $b2$, followed by the associated object $o2$, since its distance from $q$ of 57 is less than 91. At this point, $o2$ is inserted into $NearestList$ (line 7), causing $o4$ to be ejected, and $o2$ thus becomes the new candidate nearest neighbor. Finally, since the distance values of $b1$ (on the $ActiveBranchList$ of $n1$) and $n3$ (on the $ActiveBranchList$ of $n$) are both greater than 57, the traversal backtracks beyond the root and the algorithm terminates, having determined that $o2$ is the nearest neighbor of $q$.

**Best-First Algorithm**

It turns out that we can do better with different traversal strategies than the one presented in the previous section. To see why, observe that once we visit a child $e0$ of an element $e$, we are committed to traverse the entire subtree rooted at $e0$ (subject to the pruning condition) before another child of e can be visited. For example, in the traversal illustrated in Figure 1.9(b), we must visit $b5$ and $o5$ after the traversal reaches $n2$, even though $n1$ is closer to $q$ than $b5$. The above property is inherent in the fact that the algorithm of Figure 1.8(a) maintains a separate $ActiveBranchList$ for each element on the path from the root of the search hierarchy down to the current element. Thus, it may seem that we might improve on the depth-first strategy by somehow combining $ActiveBranchList$'s for different elements.

The best-first traversal strategy is indeed driven by what is in effect a global, combined $ActiveBranchList$ of all elements that have been visited. In the nearest neighbor algorithm of Figure 1.8(b), this global list is maintained with a priority queue Queue, with distances serving as keys. Initially, the root of the search hierarchy is inserted into the queue. Then, at each step of the algorithm, the element with the smallest distance is removed from the queue (i.e., it is "visited"), and its child elements either inserted into Queue or, for objects, into $NearestList$. The $NearestList$ variable and $MAXDIST(NearestList)$ have the same meaning as in the depth-first algorithm, while $MINDIST(Queue)$ denotes the smallest distance among the elements in $Queue$. Observe that $MINDIST(Queue)$ and $MAXDIST(NearestList)$ converge toward each other (from 0, being increased by line 7, and from $\infty$, being decreased by line 11, respectively), and that when the while-loop terminates, the value of $MAXDIST(NearestList)$ has reached the distance from q of its kth nearest neighbor. Also, note that, as a variation on this algorithm, we could remove elements from $Queue$ after the execution of the for-loop in case $MAXDIST(NearestList)$ has decreased (due to one or more insertions in line 11), which renders unnecessary the second part of the condition in line 6. However, in this case, the priority queue implementation must support efficient ejection of elements having keys greater than some value (i.e., $MAXDIST(NearestList)$).

One way to obtain an intuition about the best-first nearest neighbor algorithm is to consider the geometric case. If $q$ is a two-dimensional point, as in Figure 1.9(b), the search effectively proceeds by first drilling down the search hierarchy and then expanding a circular query region with $q$ as its center. Each time that the query region hits a non-leaf element $e$, we visit $e$, and the search terminates once the query region intersects at least $k$ objects. The order in which search hierarchy elements would be visited for the above example is depicted in Figure 8b. Initially, NearestList is empty and $Queue = \{(n, 0)\}$. Thus, we

"drill down" to $n$, causing its children to be inserted into the priority queue, which results in $Queue = \{(n2, 44), (n1, 51), (n3, 80)\}$ (all enqueued in line 15). At this point, the circular query is in effect expanded until its radius becomes equal to 44, the distance of $n2$ from $q$. Visiting $n2$ causes $b4$ and $b5$ to be inserted into the priority queue, resulting in $Queue = \{(n1, 51), (n3, 80), (b5, 85), (b4, 88)\}$. Next, $n1$ is visited and $b1$ through $b3$ enqueued, yielding $Queue = \{(b2, 51), (b1, 63), (n3, 80), (b5, 85), (b4, 88), (b3, 203)\}$. Visiting $b2$ leads to the insertion of $o2$ with distance value of 57 into $NearestList$ (in line 11), which has hitherto been empty. Since we now have $63 = MINDIST(Queue) \neq MAXDIST(NearestList) = 57$ (i.e., the second part of the condition in line 6 now evaluates to false), and the traversal is terminated with $o2$ as the nearest neighbor.

### 1.2.11 Parallelism

When a search request is sent from a peer to others there are two choices; forward the query to all peers in *parallel* at once and coalesce the results or forward the query *iteratively* to each on of them.

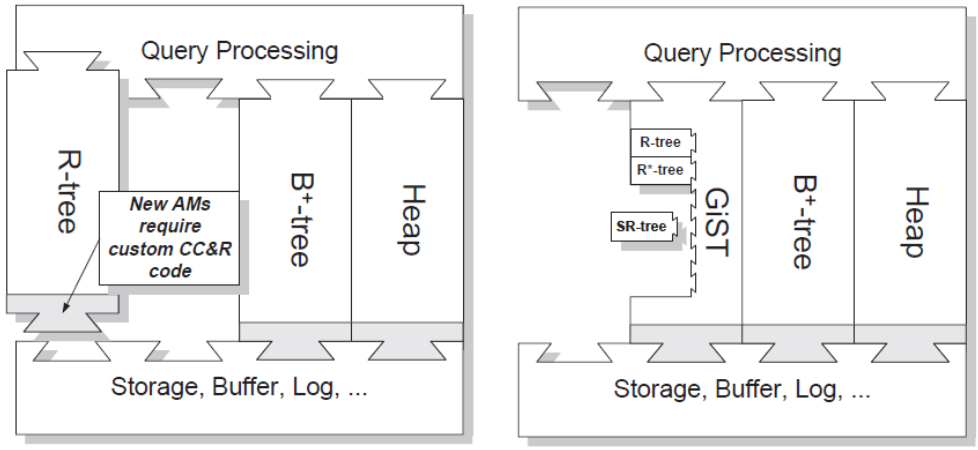
## 1.3 P2P Data Structure Frameworks

Due to the diversity of the data and the queries there is the necessity for a diversity of custom data structures. Along this line of thought we need systematic approaches to develop easily new data structures. In other words we need software packages called *frameworks*[2], that facilitate the construction of data structures for either memory, or disk or distributed environments.

The concept of frameworks is very old. For example in the field of Software Engineering several attempts had been made to construct frameworks that allowed the rapid composition and generation of new systems. A typical framework was GENESIS[1] in 1990. Soon the concept of frameworks was developed by the Databases field.
Maybe the most prominent example of such a framework is the *GiST* (*Generalized Search Trees*) framework which allows the development of data structures for the Secondary Memory with satisfactory performance. GiST is constantly being developed since 1995. It is a tree data structure stores data on leaves, supports search and update functions[35] and provides an API which further supports recovery and transactions[41]. GiST framework can be used to build a variety of search trees for Secondary Memory such as R-Tree[32], B-Tree[13], hB-tree[46] and RD-tree[64]. Not surprisingly GiST has been used to construct many indices for the well-known ORDBMS PostgreSQL[3]. As the PostgreSQL online manual says[4] "One advantage of GiST is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert".

[2] We interchangeably use the terms *framework*, *protocol* and *network*
[3] http://www.postgresql.org
[4] http://www.postgresql.org/docs/9.1/interactive/gist-intro.html

On Figure 1.3 we show how the GiST is integrated in a Relational Database and abstracts the notion of the data structure.




(a) Standard ORDBMS (b) ORDBMS with GiST

Figure 1.10: Access methods Interfaces. On (a) traditional Object Relational Database Management Systems (ORDBMS) if we want to create an extension to R-tree (e.g. R*-tree) we will have to create everything from scratch, e.g. searching, concurrency (CC) and replication (R). On (b) having GiST we can use we reuse its components and reduce our effort (complexity, number of code lines) to the minimum. Images taken from [41].

On the other hand for the case of P2P rudimentary work has been done. Typical examples of such frameworks are VBI[5], PGrid[6] and our own GRaSP[49] networks. This thesis comes to extend GRaSP to support nearest neighbor queries.

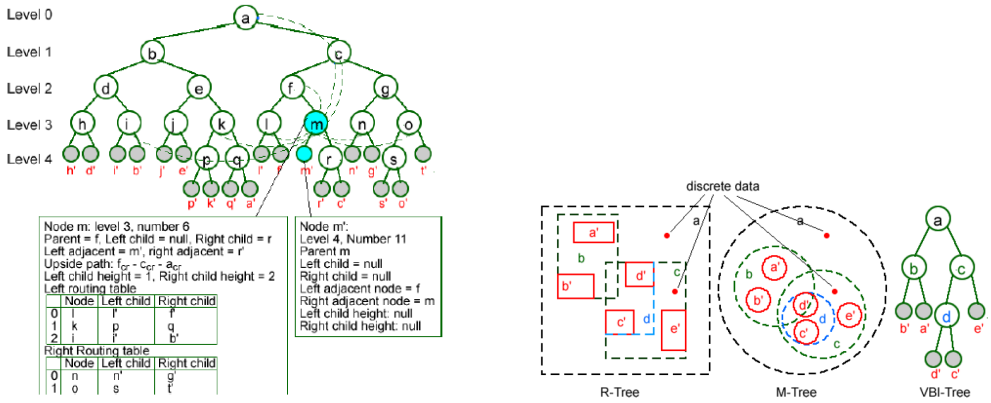
### 1.3.1 VBI

A distributed data structure oriented to generalized range queries (see §1.2.2) is the *Virtual Binary Index Tree*[5] (*VBI-tree*). On VBI peers are overlayed[5] over a balanced binary tree like they do on BATON[38]. The tree is only virtual, in the sense that peer nodes are not physically organized in a tree structure at all. The abstract methods defined can support any kind of hierarchical tree indexing structures in which the region managed by a node covers all regions managed by its children. Popular multidimensional hierarchical indexing structures that can be built on top of VBI include the R-tree[32], the X-tree[16], the SS-tree[25], the M-tree[21], and their variants. VBI guarantees that point queries (see §1.2.2) and range queries (see §1.2.2) can be answered within $O(logn)$ hops, where $n$ is the number of the peers. VBI specifies an effective load balancing strategy to allow nodes to balance their work load efficient. Validation has been made by applying the M-tree and

[5] An *overlay network* is a computer network which is built on top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. For example, many peer-to-peer networks are overlay networks because they run on top of the Internet. (definition taken from http://en.wikipedia.org/wiki/Overlay_network)

nearest neighbor queries over VBI.

The major drawback of VBI is that it suffers from congestion under large loads for low-dimensional rectangular range queries as has been recently proved by Blanas et al[56]. They compared VBI[5], PGrid[6], CAN[3] and MURK[4] and concluded that with regard to congestion PGrid is the most scalable while VBI is the least.




(a) VBI-Tree structure. The figure depicts the state of data nodes (leaf nodes) and routing nodes (internal nodes). Each routing node maintains links to its parent, its children, its adjacent nodes and its sideways routing tables. Each routing peer maintains an "upside table" stored in each routing peer, with information about regions covered by each of its ancestors.

(b) Two dimensional index structures. An R-Tree and an M-Tree in two dimensional space are mapped into VBI. It happens both trees to map into the same VBI-Tree.

Figure 1.11: VBI exemplary topologies. Images taken from [5].

### 1.3.2 PGrid

PGrid is a peer-to-peer lookup system based on a virtual distributed search tree, similarly structured as standard distributed hash tables. Figure 1.3.2 shows a simple P-Grid. Each peer holds part of the overall tree. Every participating peer's position is determined by its path, that is, the binary bit string representing the subset of the tree's overall information that the peer is responsible for. For example, the path of Peer 4 in Figure 1 is 10, so it stores all data items whose keys begin with 10. For fault tolerance multiple peers can be responsible for the same path, for example, Peer 1 and Peer 6. PGrid's query routing approach is as follows: For each bit in its path, a peer stores a reference to at least one other peer that is responsible for the other side of the binary tree at that level. Thus, if a peer receives a binary query string it cannot satisfy, it must forward the query to a peer that is "closer" to the result. In Figure 1.3.2, Peer 1 forwards queries starting with 1 to Peer 3, which is in Peer 1's routing table and whose path starts with 1. Peer 3 can either satisfy the query or forward it to another peer, depending on the next bits of the query. If Peer 1 gets a query starting with 0, and the next bit of the query is also 0, it is responsible for the query. If the next bit is 1, however, Peer 1 will check its routing table and forward

the query to Peer 2, whose path starts with 01.

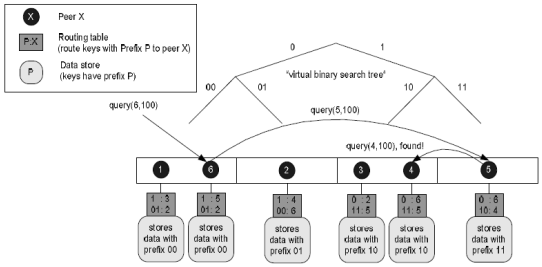



Figure 1.12: PGrid-Trie instance. Image taken from [6].

PGrid protocol doesn't give any guidelines on how to map the PeerIDs onto the key space. It's single requirement is the dimensionality of the key space to be 1.

PGrid was initially made to answer point queries (see §1.2.2) and handle 1-D data. The algorithm for point queries is pretty simple and described on [7]. Aberer[24] proposes the wrapping of PGrid with two equivalently algorithms for 1-D range queries (see §1.2.2): a sequential algorithm (*minmax algorithm*) and a superior parallel (*shower algorithm*). In the shower algorithm a peer forwards simultaneously a query it receives to neighbors that can answer it. The latency for the shower algorithm is $O(logn)$. The evaluation of the aforementioned searching algorithm has been made theoretically and empirically on PlanetLab.

An interesting application of the shower algorithm on PGrid has been realized by Blanas and Samoladas [56]. They mapped the peerIDs of the multi-dimensional PGrid on 1-dimension by applying to them zero-order space filling curve[2] thus making their modified PGrid version able to embody the shower protocol and facilitate multi-dimensional range queries.

## 1.4 Outline

On Chapter 2 we introduce the reader to GRaSP since our work is built on top of it. More specifically we familiarize the reader with the notion of the trie, how it is built when a peer joins the network, how the metric space is mapped to trie nodes and peers through hashing functions and how the data are inserted and the routing algorithm. The following Chapter 3 is the core of our thesis and contains our main contributions. More specifically building on GRaSP we provide user with a framework on top of unbalanced tries where nearest neighbor searching can be performed preserving all the good benefits of GRaSP, i.e. not requiring the trie to remain balanced, allowing the user to experiment with different space partition schemes adapted to the workload in question, etc. We prove the searching algorithm's good performance theoretically and on next Chapter 4 we evaluate it experimentally in terms of maximum throughput, latency, data fairness of index, etc with several workloads and for several dimensions. We summarize and conclude our the-

sis on Chapter 5.

# Chapter 2

# Related Work

In this chapter we delve into GRaSP. We formally present its underlying trie and how its nodes are mapped to space regions with a custom hashing function $S()$. After this the routing algorithm is presented used to route queries among peers. We demonstrate GRaSP's efficiency with some theoretical bounds. Data updates and peer joins are also presented. We conclude with two protocols developed with GRaSP, for tackling the multi-dimensional rectangular (MDRS) search problem and the 3-sided search problem.

## 2.1 GRaSP

In this section we explore GRaSP a P2P framework which tackles the case of generalized range search (see §1.2.2). GRaSP can be extended to particular range search problems with little effort, simply by providing a hierarchical space-partitioning function which maps binary strings to space ranges. In the following chapter, which delves into our main contribution, we extend GRaSP to support nearest neighbor searching.

### 2.1.1 Trie-structured networks

The goal in this section is to develop the necessary definitions needed in the description of GRaSP protocols, and in the analytical results of the next section.

#### Notation

We shall use the following notation: a binary string $x$ has length $|x|$, and $\epsilon$ denotes the empty string. We write $x \sqsubseteq y$ to denote that $x$ is a prefix of $y$. The longest common prefix of $x$ and $y$ is denoted by $x \uparrow y$ and their concatenation by $x \cdot y$. Finally, $x[i], 0 \leq i < |x|$ is the $i$-th symbol of $x$ (starting with 0), $x[: j]$ is the prefix of size $j$, $x[i :]$ is the suffix of size $|x| - i$ and $x[i : j] = x[i :][: j - i]$.

A *prefix code* is a finite set $\mathcal{P}$ of (finite) binary strings, with the following property: for every infinitely long binary string $x$, there is a *unique* $y \in \mathcal{P}$ such that $y \sqsubseteq x$.

The $i$-th complement of $x$ is $x[: i] \cdot \overline{x[i]}$ (for $i < |x|$).

#### Binary tries

A *binary trie* is a full binary tree, i.e., a binary tree where each non-leaf node has exactly two children. Each node $n$ of the trie can be associated with a binary string $I(n)$, called the *node ID*, by the following rule: the node ID of the root is the empty string, and for every other node $u$, with parent $v$, if it is a left child of $v$ then $I(u) = I(v) \cdot 0$, else $I(u) = I(v) \cdot 1$. A binary trie can be fully described by the set of node IDs of its leaves, which constitute a prefix code.

In the context of PGrid, any trie of $n$ leaves gives rise to a network of $n$ peers, where each peer is associated with a distinct leaf. Thus, we describe the shape of a trie-structured network by a prefix code $\mathcal{P}$. For simplicity, we often identify a peer with the node ID of the corresponding leaf (its peer ID), that is, the elements of $\mathcal{P}$ (which are binary strings) will be referred to as peers.

Let us fix a trie $\mathcal{P}$. For $p, q \in \mathcal{P}$, let

$$p \triangleright q = |p| - |p \uparrow q|$$

be called the gap from $p$ to $q$. Note that $0 \leq p \triangleright q \leq |p|$. Informally, $p \triangleright q$ can be described in terms of the trie: starting from $p$, one must ascend $p \triangleright q$ nodes in the trie, before it can descend towards $q$. Thus, $p \triangleright q$ is, in a sense, a distance function; note however that it is not symmetric.

The most useful law regarding gaps is:

**Theorem 1 (Routing rule)**

$$p \triangleright q < p \triangleright r \Leftrightarrow q \triangleright p < q \triangleright r \Leftrightarrow r \triangleright p = r \triangleright q$$

**Basic routing**

In order to route messages among the peers, each peer maintains a set of pointers to other peers. For peer $p \in \mathcal{P}$, define a $(|p|+1)$-partition of the set of peers $\mathcal{P}$ as follows:

$$N_i^p = \left\{ q \in \mathcal{P} \;\middle|\; p \triangleright q = i \right\} \text{ for } 0 \leq i \leq |p|.$$

Alternatively, each set $N_i$ consists of the leaves of the subtree (of the trie) rooted at the trie node with node ID equal to

$$p[:|p|-i] \cdot \overline{p[|p|-i]}$$

(the $(|p|-i)$-th complement of $p$).

The routing table of $p$ is constructed by selecting *uniformly at random* one peer from each $N_i^p$. Let $L_i^p \in N_i^p$ denote the selected peer.

In order to route a message from $p$ to $q$, the message is forwarded from $p$ to $r = L^p_{p \triangleright q}$. From $r$ it is recursively forwarded to $L^r_{r \triangleright q}$, and so on, until it reaches $q$. To see that this will happen, note that $p \triangleright r = p \triangleright q$, and by the routing rule, $q \triangleright p > q \triangleright r$: with each hop, the gap from the destination $q$ to the current node

**Hierarchical binary space partition**

Let $U$ be an arbitrary set, called the *search space*. The elements of $U$ are called points. A key space $\mathcal{K}$ is a family of non-empty subsets of $U$, whose elements are called keys. A range space $\mathcal{R}$ is also a family of non-empty subsets of $U$, whose elements are called ranges. A dataset $K \subseteq \mathcal{K}$ is a *finite* subset of the key space. Given a range $R \in \mathcal{R}$, and a dataset $K$, the answer $A_K(R)$ to $R$ over $K$ is the set of elements of $K$ which intersect $R$:

$$A_K(R) = \{x \in K | x \cap R \neq \emptyset\}.$$

A *hierarchical binary space partition* is a function $S$ from binary strings to subsets of $U$, such that $S(\epsilon) = U$, and for each string $u$, $\{S(u \cdot 0), S(u \cdot 1)\}$ is a partition of $S(u)$, i.e., $S(u \cdot 0) \cap S(u \cdot 1) = \emptyset$ and $S(u \cdot 0) \cup S(u \cdot 1) = S(u)$.

For a given space partition $S$, assign $S(p) \subseteq U$ to be the peer range of $p$. Since the peer IDs form a prefix code, it is easily seen that the peer ranges partition the search space.

Given a dataset $K$, each peer $p$ stores $A_K(S(p))$. Note that this storage scheme implies redundancy; keys may be stored in multiple peers.

**Range search**

To answer a range query for range $R$, starting from some initial peer $q$, all that is needed is to forward the search to those peers whose range intersects $R$. These peers will collectively report the full answer to the range query.

The following protocol can be used to locate the relevant peers:

**RangeSearch(peer** $p$**, range** $R$**, int** $l$**) {**
  **if(** $R \cap S(p) \neq \emptyset$ **) answerLocally(**$R$**);**
  **for(int** $i = l$**;** $i < |p|$**;** $i$**++)**
    **if(** $S(p[:i] \cdot \overline{p[i]}) \cap R \neq \emptyset$ **) RangeSearch(**$L^p_{|p|-i}$**,** $R$**,** $i+1$**);**
}

where the search is initiated by a call to **RangeSearch(**$q$**,**$R$**,**0). In terms of a P2P network, **RangeSearch(**$p$**,**$R$**,**$l$**)** is a message sent to $p$. Parameter $l$ is used to restrict the scope of search. It denotes that $p$ should only forward the search to the part of the network corresponding to a subtree of the trie, rooted at the trie node with nodeID $p[:\ l]$. This subtree includes every peer $q$ with $q \sqsubseteq p[:\ l]$. Peer $p$ fulfills this request by forwarding further to each network subset $N^p_{|p|-i}$, for $l \leq i < |p|$. However, the search is pruned for those $i$ where there will be no answer from the corresponding subtree. Routine **answerLocally(**$R$**)** poses range query $R$ to the set of keys stored at peer $p$, and reports $A_K(S(p)) \cap A_K(R)$ to the user.

### 2.1.2 Analysis of trie-structured networks

We now turn our attention to the cost of searching over tries. We are interested in two metrics: hop latency and congestion. *Hop latency* is the maximum distance (in terms of hops) from the initial peer to any peer reached during the search. On [49] we have showed that the average cost for generalized range queries is logarithmic. Actually we have showed that the routing diameter is $O(\log n)$, with high probability. The *congestion* at peer $p$ is the number of messages forwarded via $p$, when every peer in the network routes a search to some other, randomly chosen peer.n [49] we have showed that the average congestion is logarithmic.

### 2.1.3 Network maintenance

So far we have concentrated on describing and analyzing the performance of range search over a static P2P network. In most applications, P2P networks are in a continuous state of flux, as peers join and leave, often by failing. Also, new data arrives constantly and must be inserted into the network.

One of the most appealing features of GRaSP is that, as a straightforward extension of P-Grid, it inherits many of its protocols from it. P-Grid has been extensively studied in previous work, both by simulation and by implementation, and its performance is well documented. So, we shall only discuss those issues where GRaSP differentiates from

P-Grid. These issues are, insertion and deletion of keys, and peer joins. Other issues, including the handling of failing peers, peers leaving the network gracefully, updating of routing tables etc. are handled identically to P-Grid.

### Data updates

To insert a new key into the network, one can use the **RangeSearch** procedure of §2.1.1, where now in place of a search range we pass the actual key to be inserted. This key may be replicated to multiple peers. Deletions can be handled in a similar manner. For bulk updates, e.g., when a newly arriving peer wishes to index multiple keys, the same basic procedure applies, except that the set of indexed keys should be distributed at each forwarding step, in order to minimize the amount of data transferred over the network. Note that the hop latency of all these procedures is still $O(\log n)$ w.h.p.

### Peer joins

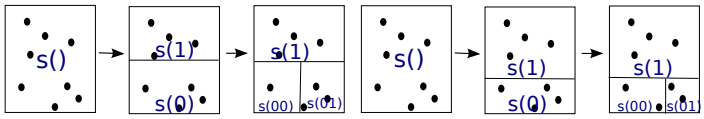
A new peer $p$ who wishes to join the network, must contact some existing peer whose network address is known to it, called the *bootstrap* peer. Then, it must select a *mate*, that is, an existing peer $q$ (which can be the bootstrap itself), whose region it will split, taking its place in the trie as a sibling of $q$. We now discuss the problem of *mate selection*.

It is desirable to select mates in a manner that tends to equalize load distribution among peers. However, what constitutes "load" may depend on the particular application; in fact, peers in the same P2P network may have differing concepts of load. Several works in the literature propose schemes that distribute the stored data evenly. Yet, many peers may consider storage a cheap resource to contribute, and may be more interested in reducing the network bandwidth contributed to the P2P network. By this reasoning, a general policy for mate selection may be hard to devise. Therefore, we discuss a few possible heuristics.

**Volume-balanced selection:** The goal here is to equalize the volumes of peer regions, by selecting mates with probability proportional the volume of their area. This can be done by selecting a point $x \in U$ from the search space, uniformly at random, The peer $q$ whose region contains $x$ is designated as mate. This protocol requires $O(\log n)$ routing messages, in order to route from the bootstrap to the mate. See Figure 2.1(a) for an exemplary volume-balanced partition.

**Data-balanced selection:** If an estimate of the distribution of indexed data is available, it may be used to equalize the number of keys stored by peers. A method similar to volume-balanced selection can be used: choose some point $x \in U$ according to the estimated data distribution, and locate the corresponding peer in $O(\log n)$ messages. Data-balanced tends to create balanced tries. See Figure 2.1(b) for an exemplary data-balanced partition.

**Uniform selection:** Random walks in a P2P network can be used to sample peers roughly uniformly [30]. In general, a walk of $O(\log n)$ hops is sufficient. In our network,




(a) Volume-balanced Partition (b) Data-balanced Partition

each peer can have a rough estimate of $\log n$ in the following way: each message routed through the network, carries a counter with the number of hops from the originating peer. Each peer observes the counter of messages routed through it, and maintains the maximum value of these counters, which is $O(\log n)$ w.h.p. When a peer is asked to bootstrap the join of a new peer, it performs a random walk of length equal to its estimate of $\log n$. One of the peers visited during the walk is selected as mate, according to some criterion (e.g., randomly, most loaded, etc.). This protocol needs $O(\log n)$ messages.

### 2.1.4 Applications of GRaSP

We now briefly turn our attention to specific range-search applications of GRaSP, which we have studied experimentally on [49].

#### Multidimensional Range Search

We introduced the problem of Range Search on §1.2.2. This type of search arises in numerous applications, and has recently received significant attention in the context of P2P networks [66, 20, 29, 53, 31, 9, 55].

For this type of problem, a natural choice for space partitioning is based on the idea of $k$-d trees. We can view a trie as a $k$-d tree, and split the space along one dimension each time, cycling through dimensions as we descend. If the expected data and query distributions are known, splits may not be even (similar to MURK [29]); typically, splits are even. Note that, in contrast to $k$-d trees in main memory, we are not concerned with keeping the trie balanced.

#### Three-sided Range Search

We introduced 3-sided queries on §1.2.2. Three-sided search arises in a large number of applications, and has been studied extensively. Optimal data structures are known both on main memory [48] and on disk [10].

To our knowledge, 3-sided search has not been studied before on P2P networks. In principle, it could be handled as a special case of 2-d range search, e.g., by the techniques of the previous section. In practice, such an approach would not be scalable, because of increased congestion. The problem lies in the shape of the queries: points with a low $y$-coordinate are much more likely to be returned than points with higher $y$-coordinates, even for uniformly distributed data and queries, inducing undue load on the peers that

store them. The imbalance in access frequencies is not detrimental for data structures—rather, it can be exploited, via caching, to *improve* performance. In a P2P setting though, accesses to data stored in the network should be relatively balanced, to avoid hotspots.

The solution presented in [49] ameliorates this problem by employing the versatility of GRaSP, to reduce contention by mapping the search problem within the setting of GRaSP such that accesses are more evenly distributed. The salient feature of our solution is that it attempts to introduce storage redundancy in such a way, that frequently accessed points (those with low $y$-coordinates) are stored in multiple peers, so that accesses to these keys can be distributed among the copies. For more information please refer to [49].

A possible space partition obtained by the above rules is seen in Fig. 2.1. It corresponds

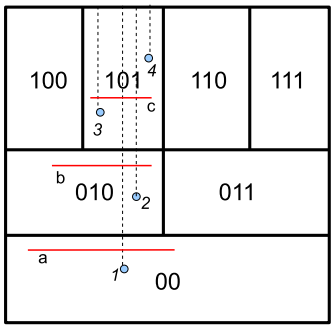



Figure 2.1: A 3-sided space partition for seven peers, marked by peer IDs.

to a hypothetical network of seven peers. This example also depicts four data points, marked 1 to 4, and three queries, marked a to c. Point 1 is relatively low, and is accessed by all three queries. However, point 1 is also replicated; it stored in peers 00, 010 and 101. Although all three queries shown will indeed return point 1, each query will access a different copy. For example, query a will access the copy in peer 00, whereas query c will access the copy in 101.
Our space partitioning scheme attempts to distribute the load rather than the data, by assigning peers into horizontal zones, so that each query will be answered by peers in the same zone. Peers in higher zones are assigned taller and narrower regions, whereas peers in lower zones are assigned shorter, wider regions. Thus, as long as query width decreases for taller queries, the number of peers accessed by each query will be relatively small. As this scheme redundancy, the total storage occupied in all peers may grow large, depending on the distribution of data and parameter $\lambda$. Fortunately, the redundancy is at most proportional to the number of zones, which should not grow too large, as the height of zones decreases exponentially. Also, if data is uniformly distributed, then the overall storage redundancy is constant, for any number of zones.

# Chapter 3

# Nearest Neighbor Search

In this chapter we extend GRaSP with k-NN capabilities. In order to do this we constrain the search domain to only metric spaces.

## 3.1 Hierarchical binary space partition

A *metric search space* $U = (\mathcal{K}, d)$, is defined as a domain of *points* $\mathcal{K}$ and a *distance function* $d$. A key space $\mathcal{K}$ is a family of non-empty subsets of $U$, whose elements are called *keys*. A nearest neighbor query space $\mathcal{Q}_k$ is also a family of non-empty subsets of $U$ of size $k$, whose elements are called *k nearest neighbor queries*. A *dataset* $K \subseteq \mathcal{K}$ is a *finite* subset of the key space.

If the indexed objects reside in a finite metric space $(\mathcal{K}, d)$ then the distance function $d$ must satisfy the following three properties, where $o_1, o_2, o_3 \in \mathcal{K}$:

**symmetry** $d(o_1, o_2) = d(o_2, o_1)$

**nonnegativity** $d(o_1, o_2) \geq 0, d(o_1, o_2) = 0$ iff $o_1 = o_2$

**triangle inequality** $d(o_1, o_3) \leq d(o_1, o_2) + d(o_2, o_3)$

Of the distance metric properties, the triangle inequality is the key property for pruning the search space when processing queries.

Given a query $Q_k \in \mathcal{Q}_k$, and a dataset $K$, the answer $A_K(Q_k)$ to $Q_k$ over $K$ is the set of $k$ elements of $K$ which satisfy the following property:

$$\begin{aligned} A_K(Q_k) = \{x_i \in K | i \in (1,k) : dist(x_i, Q_k) \leq dist(x_{i+1}, Q_k) \\ \wedge x' \in K \wedge x' \neq x_i : dist(x_k, Q_k) \leq dist(x', Q_k)\} \end{aligned}$$

## 3.2 K-nn Search Algorithm

Our nearest neighbor search algorithm is a typical sequential branch-and-bound algorithm like the one naively described on §2. We preferred a Best First traversal (see §1.2.10)

strategy because of the reasons presented in §1.2.10. Now we will delve into more detail in the algorithm which is presented on the following Figure 3.2.

```
   // initially calling: search(Q_k, p_0, ϵ, ∅, ∅)
searchKnn(Q_k, peer p, int l, priority queue F, priority queue A) {
   // p.answerLocally() returns the answers of peer p to the query set Q_k
   A = min_k{A ∪ p.answerLocally(A, Q_k)}
   for(int i = l; i < |p|; i++)
      insertIntoPriorityQueue(F, (i,p[: i] · p[i]), d(Q_k,S(p[: i] · p[i])));
   pruneFringe(F,pullHighestPriorityElement(A));
   if(F ≠ ∅) {
      (l_next,p_next) := removeNearestSubtrie(F);
      searchKnn(Q_k,p_next,l_next,F,A);
   }
}
```

Figure 3.1: The k-NN search algorithm. Recursively (a) it *branches* by adding to fringe $F$ all the recently found peers and sequentially (b) it *bounds* by pruning all the peers in fringe $F$ that are sure not to have a better answer and (c) lastly the updated query is forwarded to the most promising peer in fringe $F$.

Each k-NN query has a state which is forwarded at each hop. This state is consisted of five parameters: $Q_k$, $p$, $l$, $A$ and $F$.

$Q_k = (q, k)$ is the nearest neighbor query asked and contains the query point $q$ and the $k$ expected number of answers.

$p$ is the peer executing the nearest neighbor search algorithm.

Integer $l$ is used to restrict the scope of search. It denotes that the part of the network corresponding to a subtree of the trie, rooted at the trie node with nodeID $p[: l]$ may have candidate answers.

$A$ is a priority queue which contains all the answers (keys) found so far, Obviously $A$'s maximum size is $k$. The element $t$ of $A$ with the highest priority is the one which is nearest to the query $Q_k$, i.e. $t = \underset{a \in A}{\operatorname{argmin}} d(Q_k, a)$. Note that here the distance function is defined between two point.

The variable $F$ is a priority queue with elements parts of the trie that are promising to have better answers than the ones found so far (already contained in set $A$). By promising we mean and that the distance between the data region mapped by $S()$ to a subtree and the query is less than the $k$-th answer found so far (if we have already found $k$ answers). The head of the priority queue $F$ is the most promising subtrie. We could say that $F$ is a fringe which separates the trie into two disjoint sets. All the members of $F$ are the

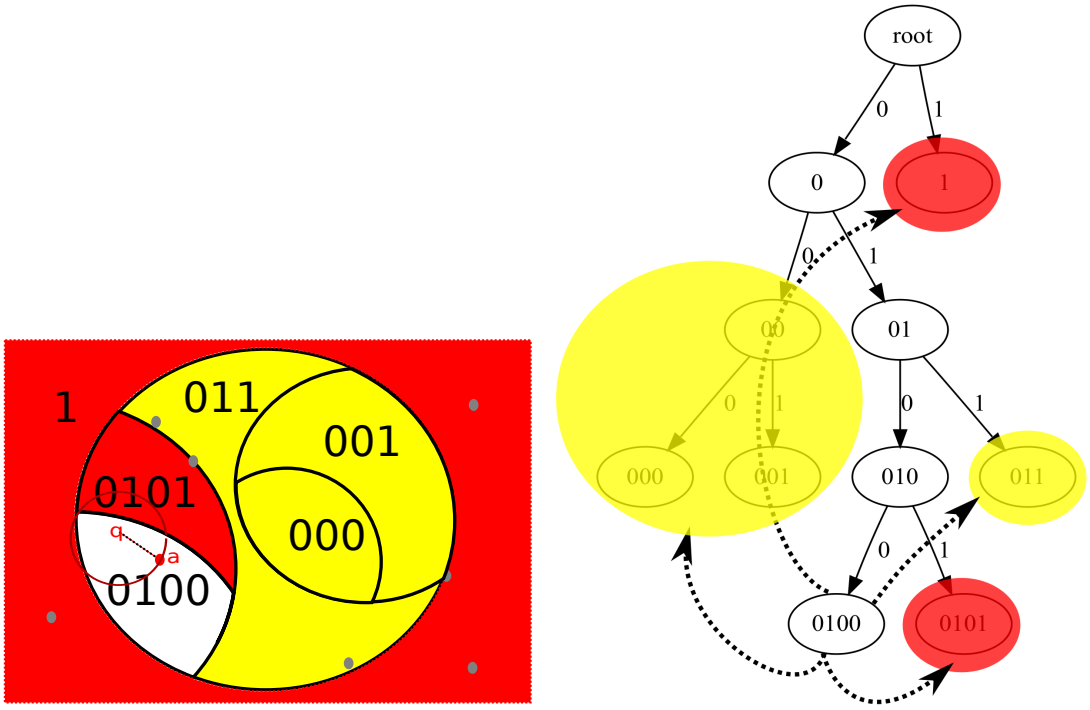
subtrees which the search algorithm hasn't explored yet and the the rest subtrees map to regions which the search algorithm has visited or discarded (pruned). Therefore we call $F$ simply *Fringe* of the query. $F$'s elements are of the form $(i, p)$. Integer $i$ is used to restrict the scope of search like $l$ does. $F$'s elements are sorted likewise to $A$, i.e. the element $(h, t)$ which has the highest priority is the one which is nearest to the query $Q_k$, i.e. $(h, t) = \underset{(i,p)\in F}{\operatorname{argmin}} d(Q_k, S(p[:i]))$. Note that here the distance function is defined between a point and a region.

To answer a nearest neighbor query for query $Q_k$, starting from some initial peer $p_0$, all that is needed is to forward the search at each hop to this peer that has the best candidate answer.
Initially, the peer ($p_0$) who asks the k-NN query executes the **knnSearch** algorithm for itself, i.e. **searchKnn(**$k$**,**$p_0$**,**$\epsilon$**,** $\emptyset$**,** $\emptyset$**)**. The peer answers the query locally by calling **answerLocally** which answers the query $Q_k$ on its own dataset $Ko$, i.e. reports the set $A_{Ko}(Q_k)$. The local results is merged with the answers found so far and after only the first (at most) k answers are kept. All the rest are discarded. Then the peer (the **for-loop**) examines it's routing table and adds to fringe $F$ every network subset that is not examined so far. by calling **insertIntoPriorityQueue** (the *branch step*). After, all the subtrees in the fringe that correspond to regions with minimum distance to the query further than the $k$th answer found so far are pruned because we know that they cannot contain better answers (the *prune step*). To fully illustrate this see Figure 3.2 for an instance of the search algorithm running and pruning candidate regions to visit.
**answerLocally** poses nearest neighbor query $Q_k$ to the set of keys stored at peer $p$, and reports $A_{K_p}(Q_k)$, where $K_p$ is the set of keys of peer $p$. **trimToK** keeps only the first $k$ elements of $A$.
**pullHighestPriorityElement** removes the answer $s$ from the $A$ which is furthest to the query. This point is used by **pruneFringe** to prune the fringe $F$ by discarding all elements $(i, q)$ of fringe $F$ which satisfy $d(Q_k, q[:i]) > d(Q_k, s)$ (*pruning distance*).
**removeNearestSubtrie** actually calls **pullHighestPriorityElement(F)** which returns the part of space with the most promising candidate answers. Then we recursively call the **searchKnn** algorithm for the peer $p_{next}$.

Last but not least a salient feature of the above algorithm is that it uses the distance function in two forms; to calculate the distance (a) between a point and a region and (b) between two points. Metrics such as the *MINDIST* and *MINMAXDIST* proposed in [54] can be used.

## 3.3 Data Updates

Let's explain here the process for data insertion. The hierarchical space partition mentioned on §2.1.1 implies the optional use of replication. In such cases we apply the **RangeSearch** algorithm described on §2.1.1 to locate the nodes responsible for the regions containing the new data. Afterwards we can insert the new datum to as many of them as we want. The proof for the correctness of the insertion algorithm under replication conditions follows.




(a) Ball-space Partition (b) The trie that maps to the ball-space partition

Figure 3.2: An exemplary ball-space partition (see §1.2.4) is presented and its trie of peers. Assume $a$ is the k-th most distant answer to query $q$. Should peer 0100 asks its neighbors 1,00,011,0101? The yellow regions are the ones rejected because their min distance from $q$ is further than a. The red regions are the ones that must be visited because its min distance from $q$ is less than $a$. The red regions are actually the Support Set of the query.

Consider for example the partition space depicted on Figure 3.3 where node $n1$ is responsible for the regions of the inner circle 1 and the middle circle 2 and node $n2$ is responsible for the middle circle 2 and the outer circle 3. Data of circle 2 can be replicated on either peer or both.
If $x$ is candidate answer and is located in $S(n1) \cap S(n2)$ then $n1$ and $n2$ can enter the fringe.
Assume that we insert $x$ only in the dataset of $n2$ and **searchKnn** examines only $n1$. Then a better answer is not found and $A$ remains unchanged. Therefore **pruneFringe** won't prune $n2$ and the search algorithm at last will visit $n2$. On the other hand, assume that we insert $x$ only in $n1$ and **searchKnn** examines only $n1$. Then $x$ enters $A$. But again **pruneFringe** doesn't prune $n2$ because $d(Q_k, S(n2)) <= d(Q_k, x)$.
All the rest cases i.e. $x \in S(n1) - S(n2)$ or $x \in S(n2) - S(n1)$ and $Q - K \in S(n1) - S(n2)$ or $Q - K \in S(n2) - S(n1)$ are handled similarly or even easier.

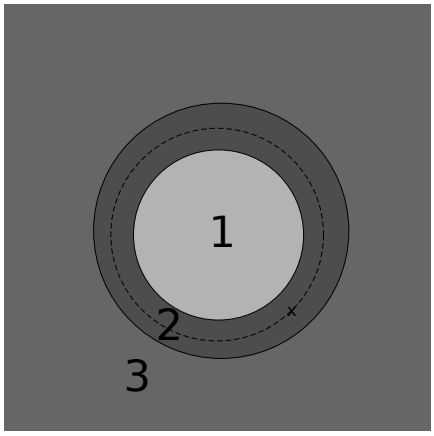



Figure 3.3: Instance of the correctness proof of the data insertion algorithm.

## 3.4 Analysis of trie-structure networks

We have shown that the search algorithm works correctly even if we insert selective replication to some of the peers that their regions overlap. The question now is if the selective replication is useful in terms of the number of messages exchanged among the peers. Because of the sequential nature of the search algorithm the selective replication would make the search algorithm to visit all the peers that contain replicas of a key we are looking for. Therefore in terms of message traffic replication is inefficient but it may be helpful in terms of fault tolerance.

### 3.4.1 Latency Complexity Theorem

First we define a useful term called Support Set. The *Support Set* of a query is the min. number of peers required to visit and ask to fully answer it. For the k-NN case these peers are the ones which own space regions that overlap the query range, i.e. the circle with center the query point and radius the distance of the query to the k-th answer (for 2-dimensional space) or the sphere in 3-dimensional space. An example is depicted on Figure 3.4. The peers which own the red regions should be at least be visited and asked.

Now we will give an upper bound for the messages exchanged by the searchKnn algorithm presented on §3.2.

**Theorem 2 (Latency Complexity Theorem)** *In a trie of $n$ leaves, the number of messages exchanged by searchKnn is bounded by $|T|O(log(n))$, where $T$ is the Support Set of the query.*

Let us consider a fixed trie $T_0$ with $n$ peers $P = \{p_1, \ldots, p_n\}$. A nearest neighbor query asks all the peers in a circle with center $Q_k$ and radius $\max_{x \in A} d(Q_k, x)\}$, i.e. all the peers satisfying the following property:

$$T = \{p \in P | d(Q_k, S(p)) \leq \max_{x \in A} d(Q_k, x)\} = \{t_1, \ldots, t_k\}$$

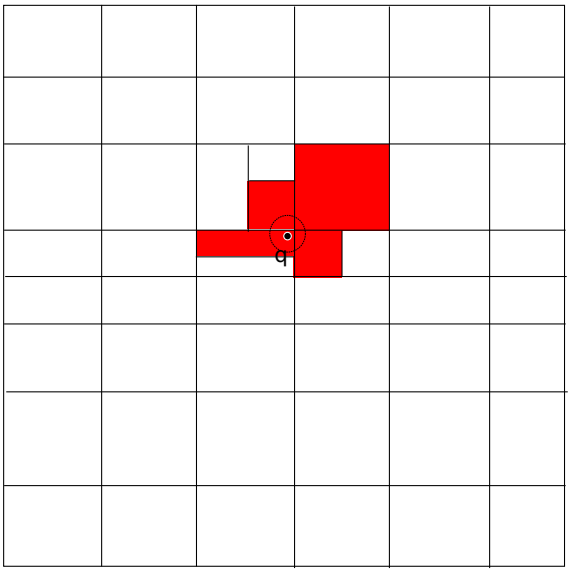


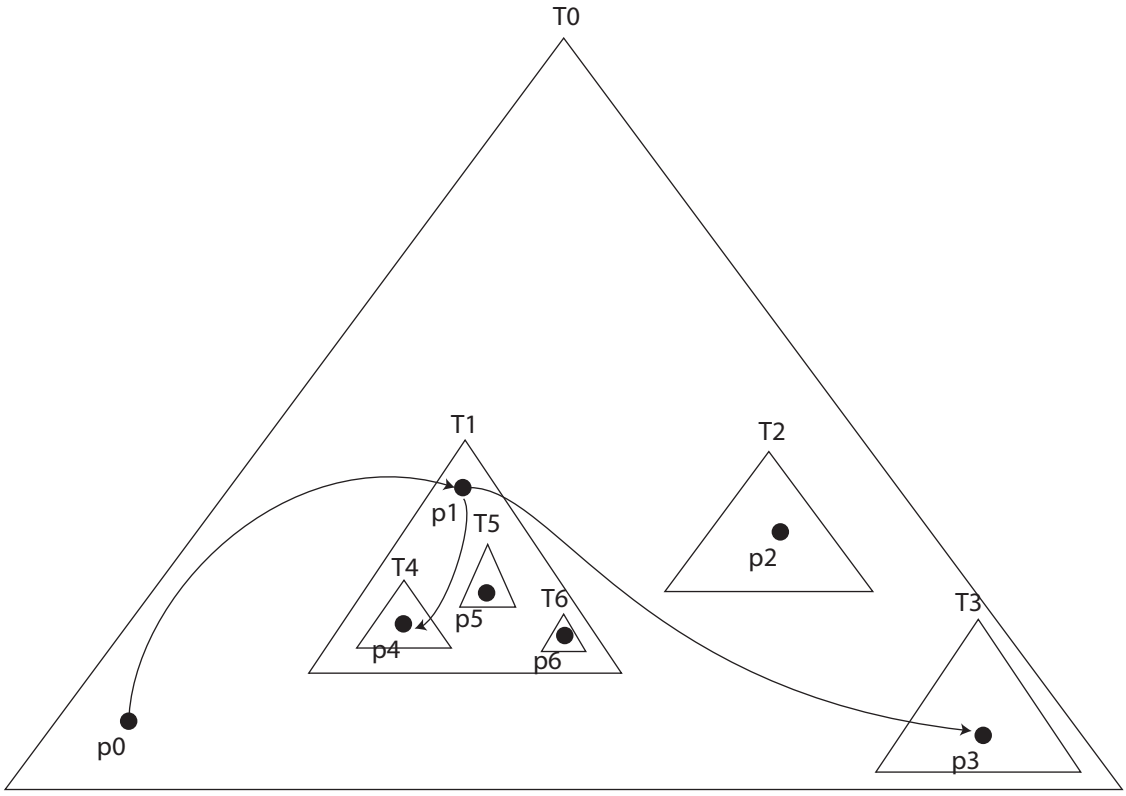

Figure 3.4: Figure shows the Support Set (colored in red) of the query $q$. The radius of the circle equals the distance between the query $q$ and the $k$th answer. All the regions in the circle colored red must be asked to fully answer the question. The peers that own these regions are called the *Support Set* of the query. Although the query covers a small region of the data space the number of peers and regions required to ask is proportionally high.

Therefore the minimum number of peers an optimum nearest neighbor search algorithm would have to ask would be $|T|$.

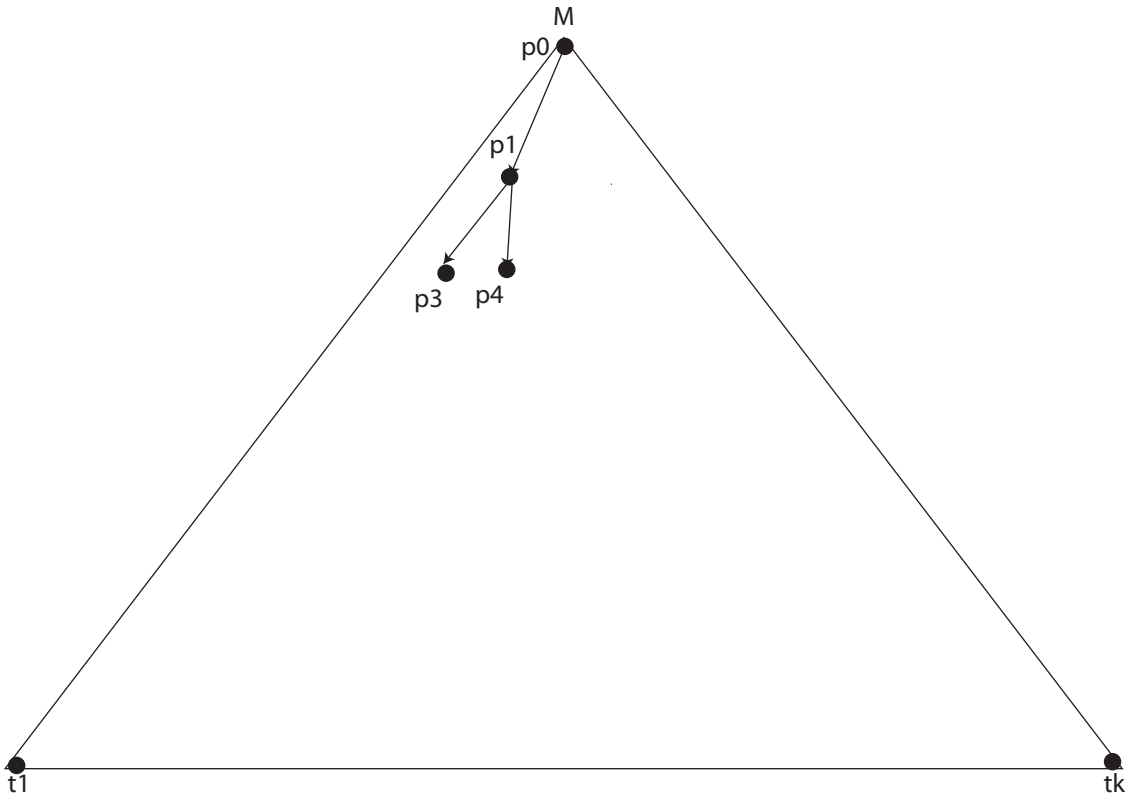
Assume searchKnn with query $Q_k$ starts from peer $p_0$. $p_0$ examines its routing table and adds to fringe $F$ the subtrees $T_1$,$T_2$,$T_3$ with representative neighbor peers $p_1$,$p_2$,$p_3$, because their distance from $Q_k$ is less than the pruning distance (see §3.2). Assume $p_0$ forwards query to $p_1$. $p_1$ adds to $F$ the subtrees $T4$,$T5$,$T6$ with representative neighbor peers $p_4$,$p_5$,$p_6$, because their distance from $Q_k$ is less than the pruning distance. Assume $p_1$ can forward query to $p_3$ and $p_4$ because $d(Q_k, S(T_3)) = d(Q_k, S(T_4))$, where $S(T_i)$ is the region a subtree is responsible for. Note that $p_1$ and $p_4$ belong to $T_1$. This is depicted on Figure 3.5(a) where two hops are possible from $p_1$, (a) hopping to $p3$ and subtree $T_3$ or (b) hopping to $p_4$ and remaining in the same subtree $T_1$. On both cases the peers will add to fringe their neighbor subtrees that satisfy the pruning distance criterion. But on the second case, i.e. if we stick to the policy of remaining in the same subtree that we are, the fringe's size will expand more slowly than the first case, i.e. hopping to other subtrees. The search will end when we find the last element of $T$.

If we place all the peers the search visits on a tree, with root node the peer $p_0$ initiating the query, leaf nodes the peers that contain answers i.e. elements of $T$ and inner nodes all the rest nodes which happen to be all the peers of the fringe $F$ then the *multicast tree* $M$ of Figure 3.5(b) is formed.
On the worst case the multicast tree $M$ is balanced. The number of the leafs of $T_0$ equals to the length of $T$, i.e. $|T|$. Therefore the number of nodes of the worst case multicast tree is at most $|T|D$, where $D$ is the diameter of the trie. But on [49] we have shown that the diameter of a trie with $n$ nodes is $O(logn)$ w.h.p.. Therefore the cost of algorithm

knnSearch in number of messages exchanged is $|T|O(logn)$.




(a) An instance of the algorithm knnSearch running. Peer $p_0$ forwards query $Q_k$ to $p_1$. After, peer $p_1$ can forward query to $p_4$ or $p_3$ because (assuming) they equally distant from $Q_k$.




(b) Instance of the Multicast Tree $M$ of trie $T$ of Figure 3.5(a): $p_0$ forwards query to $p_1$. $p_1$ can forward query to $p_4$ or $p_3$. On second case the query will remain in subtree $T_1$ forming smaller fringe.

Figure 3.5: Instance of the proof of the number of messages exchanged by algorithm knnSearch.

# Chapter 4

# Performance Evaluation

In order to evaluate GRaSP we have constructed a fast and scalable simulator. This is the mean of the evaluation and is presented on §4.1. On §4.2 we present the cost model on which we have based the evaluation, i.e. the metrics used to evaluate the quality of a network. On §4.3 we experiment with multi-dimensional synthetic and real workloads.

## 4.1 Simulator

In order to develop protocols over our framework we needed to develop a consistent API. This API gives the necessary mechanisms to represent the trie topology of our framework, the routing tables, the k-NN searching algorithm, the bootstrapping and to customize any Space Partitioning algorithm.

Our simulator is written in C++, is highly configurable, extendable, self-contained, fast and can provably support the simulation of large networks and the processing of many queries. It supports dynamic protocols, where node additions and removals are happening.

The simulator follows the Cycle Based Simulation pattern. The *Cycle Based Simulation* is a simplest simulation mode. Herein all the queries are executed sequentially one after the other. Each query is process in cycles. On each cycle either is processed locally by a node and/or forwarded to another node. When a query is fully answered is discarded from the simulated network and the next one is loaded.

## 4.2 Modeling P2P Network Performance

In order to evaluate our framework we borrow from the literature some performance metrics (see [56]) and introduce some new ones. Initially we formally describe each metric and later on we apply them to evaluate our experiments.

### 4.2.1 Maximum Throughput

*Maximum throughput* was proposed in [56], to quantify the resilience of a P2P network to contention by concurrent searches. Succinctly, assume that a workload of $Q$ queries is executed on a network. For each peer $p$, let $m_p$ be the number of messages received by $p$ due to the $Q$ queries. Assume further, that each peer can process at most one message per unit of time. Now, if queries from the workload arrive (stochastically) at a rate of $\Lambda$ queries per unit of time, each query with equal probability, then messages to peer $p$ shall arrive at a rate of $\frac{m_p}{Q}\Lambda$ messages per unit of time. Assuming that peer $p$ is not overloaded (messages do not arrive faster than it can process them), we have $\Lambda < \frac{Q}{m_p}$. Now, maximum throughput $\Lambda_{\max}$ is defined as the maximum value of $\Lambda$ such that no peer is overloaded:

$$\Lambda_{\max} = \frac{Q}{\max_p m_p}$$

Also, let $M$ be the message traffic, i.e. the expected number of messages per query. Then, if $n$ is the network size then $\Lambda_{\max} \leq n/M$, with equality holding in the ideal case where all traffic is distributed equally. Then, the ratio of traffic of the most loaded peer, over the average peer traffic, is $\frac{n/M}{\Lambda_{\max}}$.

### 4.2.2 Fringe Size

Of special interest for the scaling of the searching algorithm is the size of the *Fringe* we introduced in §3.2. Among all the messages exchanged for a gives query set and network size we examine the average and the maximum size of the fringe.

### 4.2.3 Fairness Index

Another metric pertinent to the data is the *Data Fairness Index*[39] which is defined as

$$FI = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} {x_i}^2}$$

where $x_i$ is the number of data points of peer $i$ and $n$ is the number of peers. In essence Data Fairness of Index shows the fairness of the distribution of the data on the peers; i.e. how are the data distributed among the peers. FI is continuous, scale independent, i.e. applies to any network size (even for a few peers only) and is bounded between 0 and 1 — 0 for maximal unfairness (when one peer holds all the data), 1 for maximum fairness (when all the peers have equal number of data).

### 4.2.4 Latency

Another interesting metric is the number of hops the query requires in order to fully be answered. We call this number the *Number of Peers Asked* and compare it with the theoretical lower bound, i.e. the size of the Support Set (see Figure 3.4) which we call it the *Min. Number of Peers Required To Ask*.

## 4.3 Experimental Evaluation

After presenting our extended framework and the cost model we are ready to evaluate its performance. Initially we present the workloads employed to evaluate our framework. We found or constructed such workloads in order to stress test our framework on various dimensions.

### 4.3.1 Test workloads

#### Datasets

In our experiments we used one synthetic and two real datasets of different dimensionality.

**Greece:** This real dataset was constructed from real geographic data; it contains 999578 random 2-d points along the road network of Greece [1].

**Corel:** This real dataset contains 68040 9-d points which represent the Color Moments of images from the Corel image collection [2].

**Uniform:** This synthetic dataset contains 1M 2-d points distributed uniformly. In this case our aim was experimenting with the number of dimensions.

#### Space Partition

For all our experiments we partitioned the space into peers by employing a *k-d tree space partition* (see [23], Chapter 10). We can view a trie as a k-d tree, and split the space along one dimension each time, cycling through dimensions as we descend. On splitting the space we have two options as we have already described in §2.1.3; splitting the volume in two equal-volume regions (Volume-balanced partition) or into two regions of equal number of data keys (Data-balanced partition). Note that, in contrast to k-d trees in main memory, we are not concerned with keeping the trie balanced.

#### Querysets

All the queries for all the datasets have been synthetically created with the following process; in order to construct a query we used the trie produced by a space partition of our choice and for each leaf we (uniformly) randomly picked the query. For distance function we sticked to the Euclidean metric[3] (see §1.2.9 on page 10). For a network size of $n$ peers we asked $n/3$ queries. We repeated the experiments 10 times and averaged the results. On a few metrics we used confidence intervals to indicate the reliability of an estimate.

[1] Source: http://www.rtreeportal.org/spatial.html

[2] Source: http://archive.ics.uci.edu/ml/datasets/Corel+Image+Features

[3] We used the proposed Euclidean distance are recommended from the download site

### 4.3.2 Results for low dimensions

In this section we evaluate our algorithm on a low 2-dimensional space by experimenting with the Greece dataset for various values of k, i.e. the number of answers asked. The questions raised is how well the frameworks performs with different space partitions, network sizes and number of answers k.

#### Space Partitioning: volume-balanced vs data-balanced

The Data-balanced partitioning shows better Data Fairness Index as shown on Fig. 4.1(b) where FI is about 0.6 in comparison to Fig. 4.4(b) where it fluctuates around 0.1. This is expected since data-balanced partition does exactly this; tries to equally partition data into peers. Note that on both cases FI remains almost constant irrelevant to the number of peers.

The max throughput of d.b.p. (Fig. 4.1(d)) is an order of magnitude more than the v.b.p. (Fig. 4.4(d)).

The mean and max fringe size of d.s.p (Fig. 4.1(a)). is the one fifth of the fringe size of v.b.p. (Fig 4.4(a)).

As we see on Fig. 4.4(c) the latency of v.b.p. is approximately 45 times more than the latency induced by d.b.p. (see Fig. 4.1(c)).

To sum up d.b.p. outweighs v.b.p. in terms of all the metrics.

#### Size of fringe

As was expected the fringe grows as the $k$ parameter increases (i.e. the requested number of answers) because a larger range of the trie should be searched. More specifically, for every ten fold of $k$ the max and mean fringe size are doubled (compare Fig 4.1(a) for $k = 1$ with Fig 4.2(a) for $k = 10$ and Fig 4.3(a) for $k = 100$).

The mean and max fringe size are increased logarithmically in relation to the number of peers (see for example Fig 4.3(a)).

#### Latency

Generally the number of peers asked by the search algorithm remains close to the optimum numbers of peers. To be more precise wee see from Fig. 4.1(c), Fig. 4.2(c) and Fig. 4.3(c) that it is at most 3.5 times more.

#### Maximum Throughput

As expected the maximum throughput is decreased when $k$ is decreased but only by a small factor (see figures 4.1(d), 4.2(d) and 4.3(d))

#### Data Fairness Index

Interestingly the Data Fairness Index seems to be irrelevant to the number of peers; see Fig. 4.1(b).

### 4.3.3 Results for medium dimensions

In this section we evaluate our algorithm for more than 2 dimensions. We use the real 9-dimensional ColorMoments dataset and the 9-dimensional synthetic uniform dataset.

#### Size of fringe

From Fig. 4.6(a) we see that the Fringe Size is bounded by a logarithmic distribution in relation to the network size. But on Figures 4.5(a) and 4.5(b) the fringe size especially the mean size is steeply becoming large for more than 7 dimensions (but still remains low; less than 30 for 9 dimensions).

#### Latency

The Latency Complexity Theorem Proof tells us that the number of messages is bounded by $|T|O(logn)$. The lower curve on Fig. 4.5(d) and Fig. 4.6(c) is the size of the Support Set $|T|$. See for example Fig. 4.6(c) for n=90000. The theoretical upper bound is $|T|O(logn) = 1150 * log(90000) = 1150 * 5 = 5750$ but the actual experimental result is 1500 messages only. This tells us that the theoretical bound is pessimistic, i.e. in action the algorithm is efficient.

#### Maximum Throughput

As we see on Fig. 4.6(d) the throughput is low but it isn't increased for large network sizes because of the lot of messages exchanged. This is not good but not bad. Actually it's still practical.

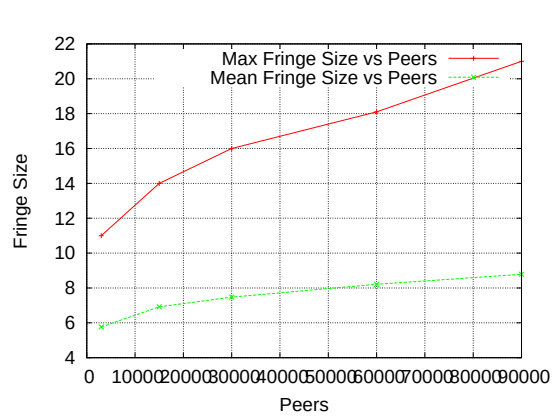
Referring now to Fig. 4.5(e) the throughput for more than 7 dimensions (i.e. for high dimensions) is very low which means there is no meaning experimenting with higher dimensions.

#### Data Fairness Index

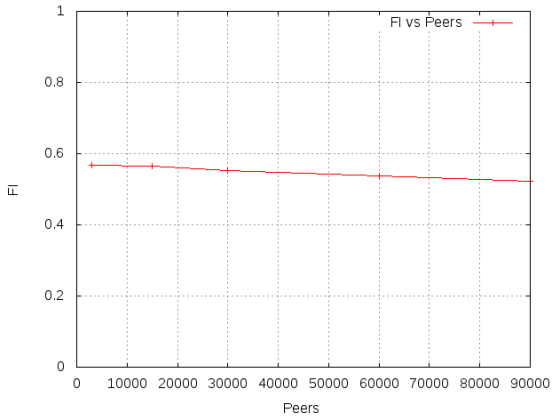
Data Fairness Index as on low dimensions remains constant and irrelevant to the number of peers as we see on Fig. 4.6(b) and Fig. 4.5(c). Of course the uniform dataset has better (higher) Data Fairness Index because the data are uniformly scattered and therefore its volume-balanced partition is the same if we did data-balanced partition.

### 4.3.4 Results for high dimensions

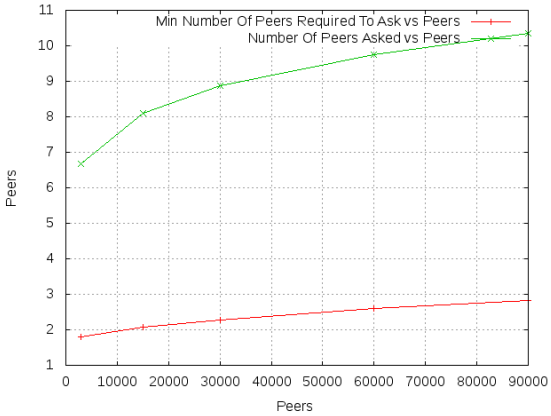
From the results shown on the previous section related to medium dimensions we see that the framework for more than 7-8 dimensions (see previous section for medium dimensions) is not efficient but there is nothing we can do because of the inherited curse



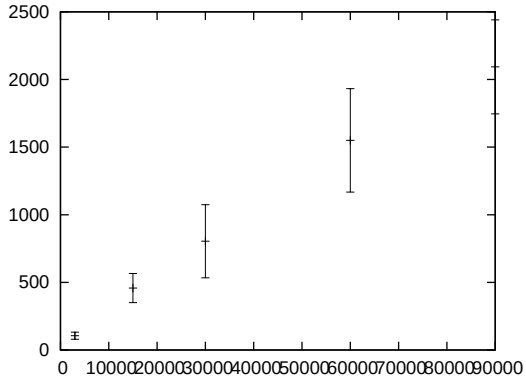

(a) Fringe Size




(b) Fairness Index for data distribution




(c) Latency



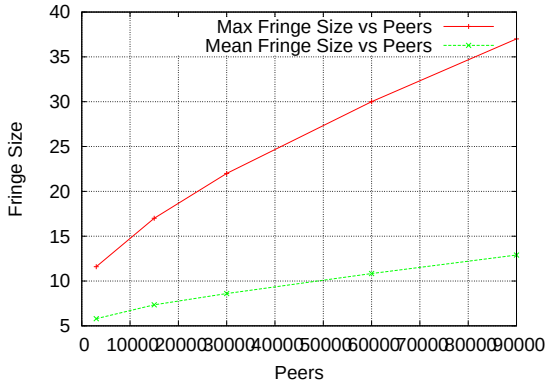
(d) Max. throughput

Figure 4.1: Results of k-NN search over network size for the Greece dataset, for data space partition and $k = 1$.

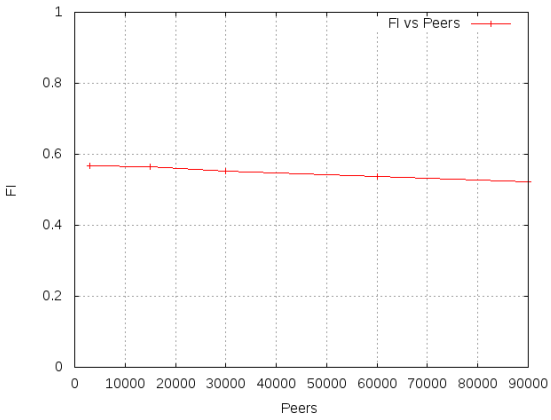
of dimensionality. Usually on memory and disk data structures the policy followed is just linear searching. In a p2p environment we can further experiment with *approximate nearest neighbor searching*.

## Figures

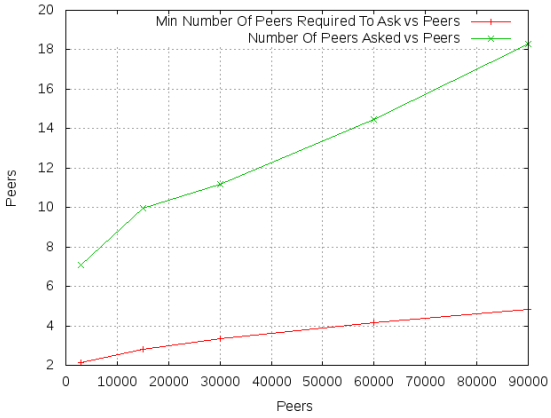
Herein we collectively list the figures of the results of all our experiments.



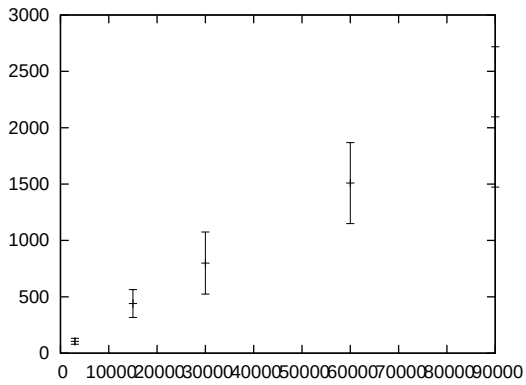

(a) Fringe Size




(b) Fairness Index for data distribution

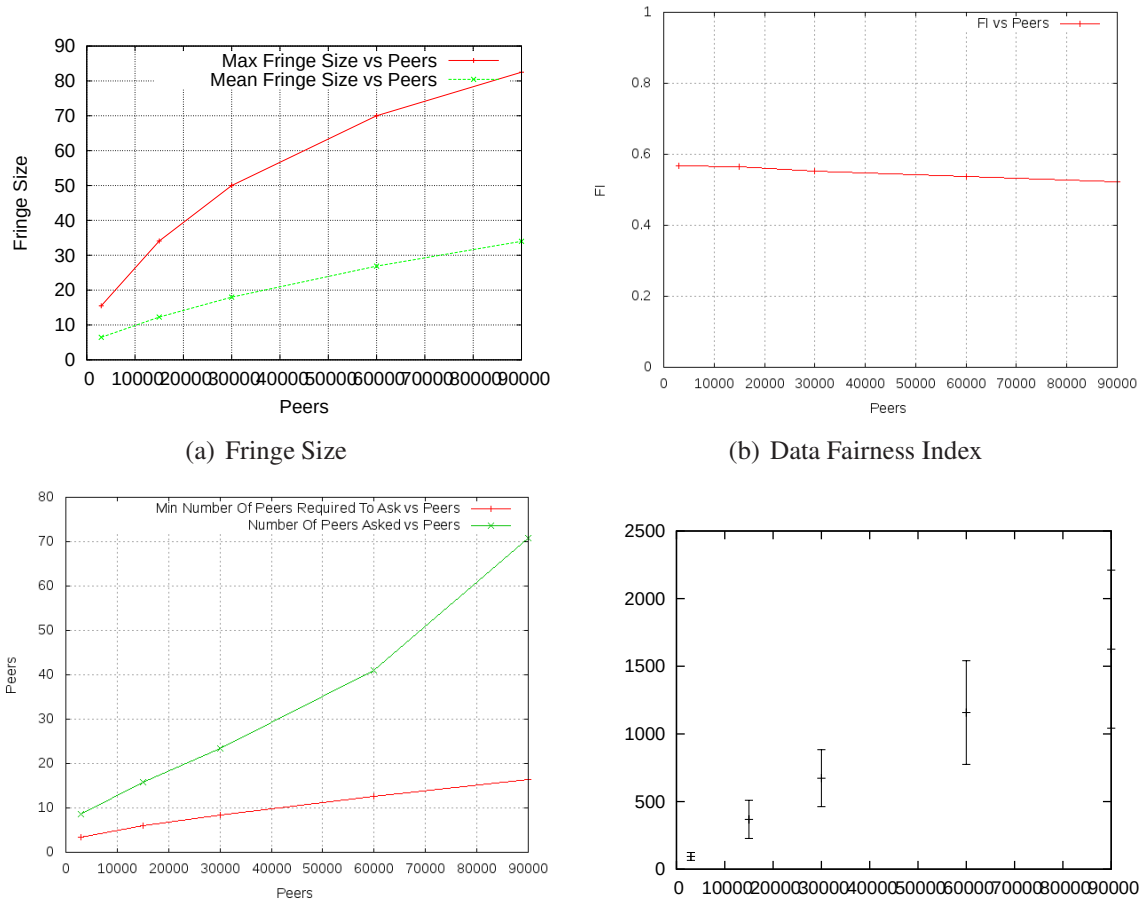



(c) Latency



(d) Max. throughput

Figure 4.2: Results of k-NN search over network size for the Greece dataset, for data space partition and $k = 10$.



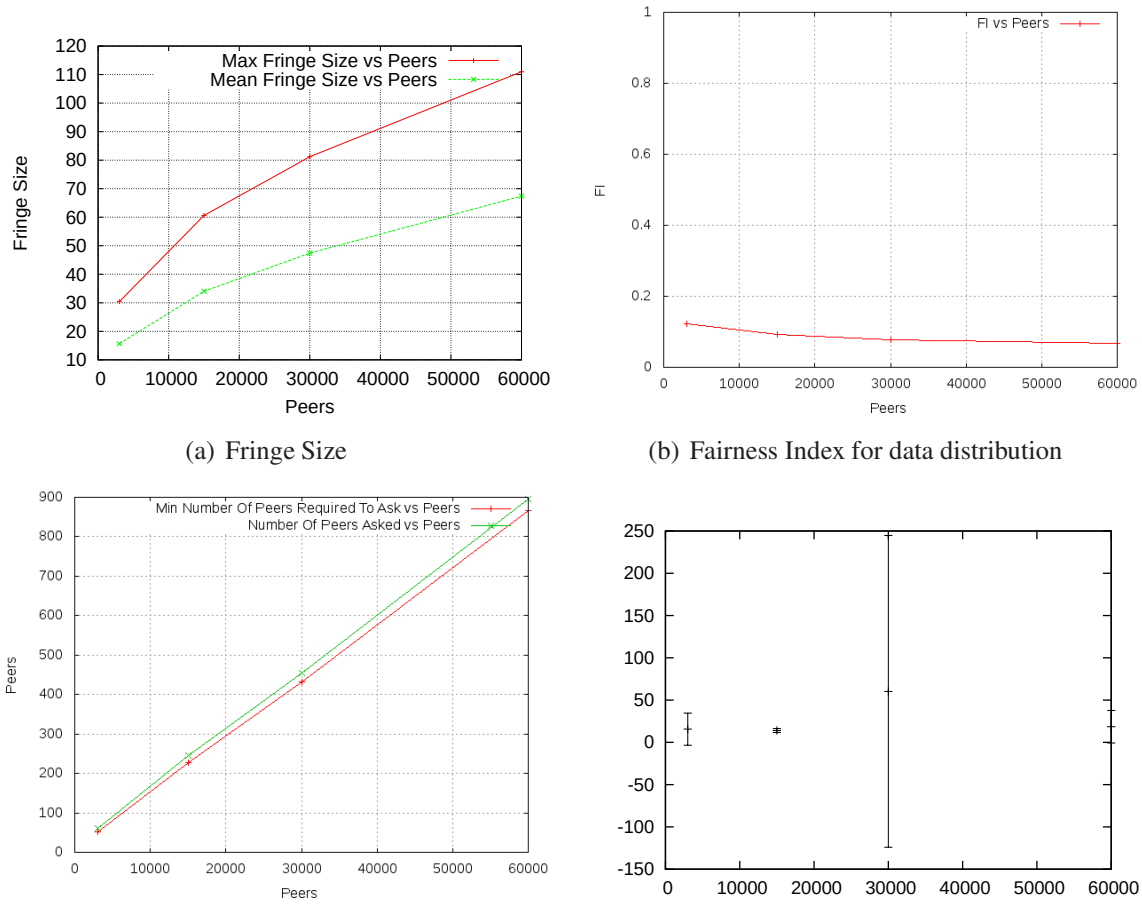

(a) Fringe Size

(b) Data Fairness Index

(c) Latency

(d) Max. throughput

Figure 4.3: Results of k-NN search over network size for the Greece dataset, for data space partition and $k = 100$.



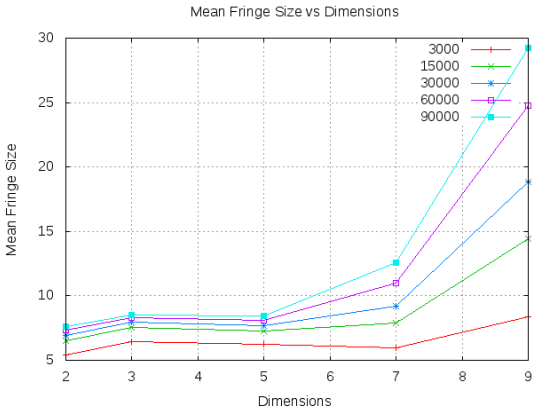

(a) Fringe Size

(b) Fairness Index for data distribution

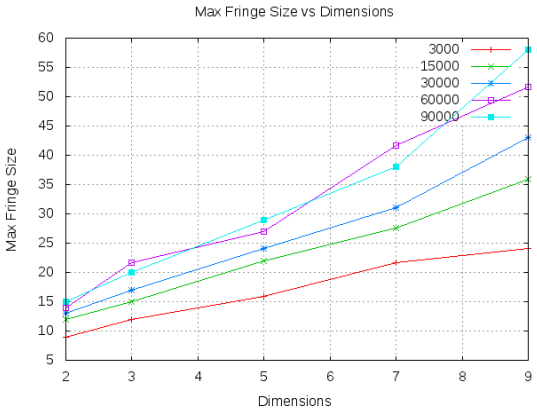
(c) Latency

(d) Max. throughput

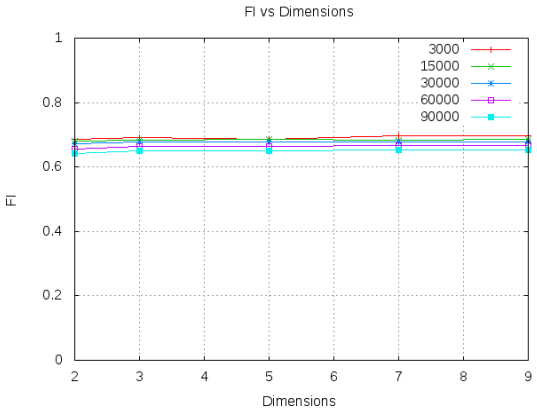
Figure 4.4: Results of k-NN search over network size for the Greece dataset, for volume space partition and $k = 1$.



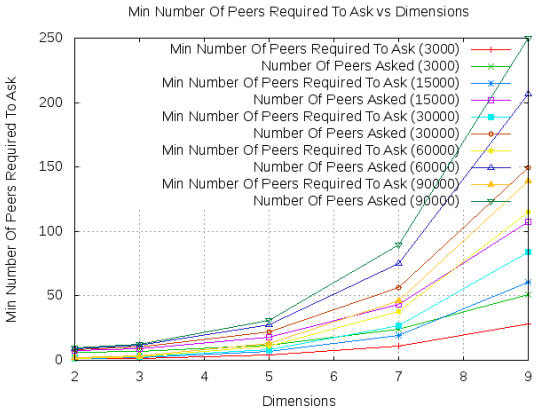

(a) Mean Fringe Size



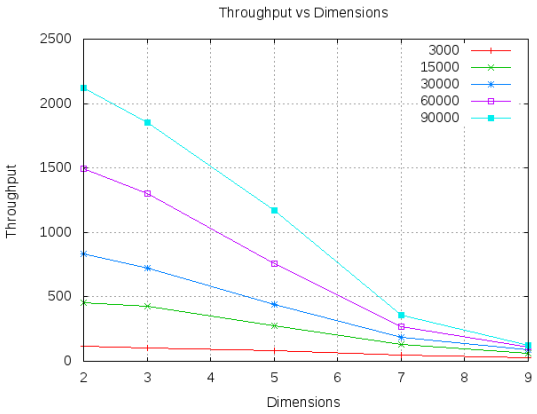

(b) Max Fringe Size




(c) Data Fairness Index

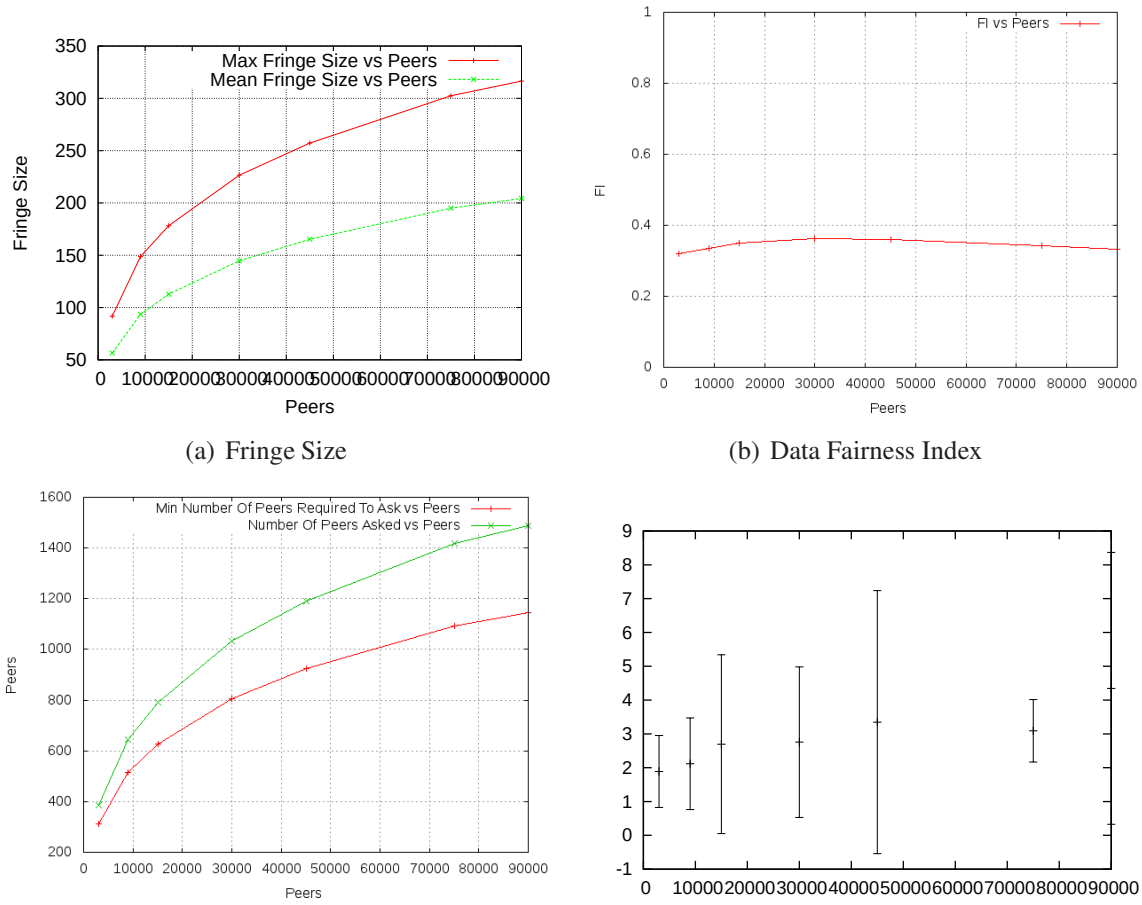



(d) Latency




(e) Max. Throughput

Figure 4.5: Results of k-NN search over number of dimensions, for the Uniform dataset, for volume space partition and $k = 1$.




(a) Fringe Size

(b) Data Fairness Index

(c) Latency

(d) Max. Throughput

Figure 4.6: Results of k-NN search over network size, for the ColorMoments dataset, for data space partition and $k = 1$.

# Chapter 5

# Conclusion and Future Work

This work comes to complement the one on [49] were we had crafted a solid distributed framework for generalized range queries where no assumptions (shape,dimensionality) are placed about the data and query types. Moreover our data structure is free of the height-balanced search tree limitation presented in adversary works. In this thesis we extend it to also support nearest neighbor queries with mathematically good guarantees. Experiments have proven the network size scalability for low dimensions. For higher dimensions the framework is as good as it can gets; approximate neighbor searching algorithms should be evaluated.

# Bibliography


[1] Don batory. on the difference between very large scale reuse and large scale reuse. in larry latour, steve philbrick, and chandu bhavsar, editors, proceedings of the fourth annual workshop on software reuse, november 1991.

[2] *A class of data structures for associative searching* (New York, NY, USA, 1984), ACM.

[3] *A scalable content-addressable network* (2001).

[4] *One Torus to Rule them All: Multidimensional Queries in P2P Systems* (2004).

[5] *VBI-Tree: A Peer-to-Peer framework for supporting multi-dimensional indexing schemes* (2006).

[6] ABERER, K. P-Grid: A self-organizing access structure for P2P information systems. *Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Lecture Notes in Computer Science 2172* (2001), 179–194.

[7] ABERER, K., PUNCEVA, M., HAUSWIRTH, M., AND SCHMIDT, R. Improving data access in p2p systems. *IEEE Internet Computing 6*, 1 (2002), 58–67.

[8] AREF, W. G., AND SAMET, H. The spatial filter revisited. In *Proc. 6th Internat. Sympos. Spatial Data Handling* (Sept. 1994), T. C. Waugh and R. G. Healey, Eds., International Geographical Union Comission on Geographic Information Systems, Association for Geographical Information, pp. 190–208.

[9] ARGE, L., EPPSTEIN, D., AND GOODRICH, M. T. Skip-webs: Efficient distributed data structures for multi-dimensional data sets. In *PODC* (2005), pp. 69–76.

[10] ARGE, L., SAMOLADAS, V., AND VITTER, J. On two-dimensional indexability and optimal range search indexing. In *PODS* (1999).

[11] BATKO, M., GENNARO, C., AND ZEZULA, P. Scalable and distributed similarity search in metric spaces.

[12] BATKO, M., GENNARO, C., AND ZEZULA, P. A scalable nearest neighbor search in p2p systems. In *DBISP2P* (2004), pp. 79–92.

[13] BAYER, R. Binary b-trees for virtual memory. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971* (1971), E. F. Codd and A. L. Dean, Eds., ACM, pp. 219–235.

[14] BECKER, L. *A New Algorithm and a Cost Model for Join Processing with the Grid File*. PhD thesis, Universiat-Gesamthochschule Siegen, Germany, 1992.

[15] BELLMAN, R. *Dynamic Programming*. Dover Publications, Mar. 2003.

[16] BERCHTOLD, S., KEIM, D. A., AND KRIEGEL, H.-P. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases* (San Francisco, U.S.A., 1996), T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds., Morgan Kaufmann Publishers, pp. 28–39.

[17] BILLE, P. A survey on tree edit distance and related problems. *Theor. Comput. Sci 337* (2005), 217–239.

[18] BÖHM, C., BERCHTOLD, S., AND KEIM, D. A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv. 33*, 3 (2001), 322–373.

[19] BRINKHOFF, T., KRIEGEL, H.-P., AND SEEGER, B. Efficient processing of spatial joins using R-trees. In *Proc. ACM SIGMOD* (1993), pp. 237–246.

[20] CHAWATHE, Y., RAMABHADRAN, S., RATNASAMY, S., LAMARCA, A., SHENKER, S., AND HELLERSTEIN, J. A case study in building layered DHT applications. In *SIGCOMM* (2005), pp. 97–108.

[21] CIACCIA, P., PATELLA, M., AND ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In *The VLDB Journal* (1997), pp. 426–435.

[22] COBENA, G., ABITEBOUL, S., AND MARIAN, A. Detecting changes in xml documents. In *In ICDE* (2001), pp. 41–52.

[23] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[24] DATTA, A., HAUSWIRTH, M., JOHN, R., SCHMIDT, R., AND ABERER, K. Range queries in trie-structured overlays. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 57–66.

[25] DAVID A. WHITE, R. J. Similarity indexing with the sstree. *In Proceedings of the 12th ICDE Conference* (1996), 516–523.

[26] FALOUTSOS, C., AND LIN, K.-I. Fastmap: a fast algorithm for indexing, datamining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1995), SIGMOD '95, ACM, pp. 163–174.

[27] GAEDE, V., AND FRITZ RIEKERT, W. Spatial access methods and query processing in the object-oriented gis godot. In *In Proc. of the AGDM'94 Workshop* (1994), pp. 40–52.

[28] GAEDE, V., AND GÃIJNTHER, O. Multidimensional access methods. *ACM Computing Surveys 30* (1997), 170–231.

[29] GANESAN, P., YANG, B., AND GARCIA-MOLINA, H. One torus to rule them all: Multidimensional queries in P2P systems. In *WebDB* (2004), pp. 19–24.

[30] GKANTSIDIS, C., MIHAIL, M., AND SABERI, A. Random walks in peer-to-peer networks: algorithms and evaluation. *Perform. Eval. 63*, 3 (2006), 241–263.

[31] GUPTA, A., AGRAWAL, D., AND EL ABBADI, A. Approximate range selection queries in peer-to-peer systems. In *CIDR* (2003).

[32] GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1984), pp. 47–57.

[33] GÃIJNTHER, O. Efficient computation of spatial joins. In *Data Engineering, 1993. Proceedings. Ninth International Conference on* (1993), pp. 50–59.

[34] HARWOOD, A., AND TANIN, E. Hashing spatial content over peer-to-peer networks. In *In Australian Telecommunications, Networks, and Applications Conference-ATNAC* (2003), pp. 1–5.

[35] HELLERSTEIN, J. M., NAUGHTON, J. F., AND PFEFFER, A. Generalized search trees for database systems. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland* (1995), U. Dayal, P. M. D. Gray, and S. Nishio, Eds., Morgan Kaufmann, pp. 562–573.

[36] HJALTASON, G. R., AND SAMET, H. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems 28* (2003), 2003.

[37] HUTTENLOCHER, D. P., KLANDERMAN, G. A., AND RUCKLIDGE, W. A. Comparing images using the hausdorff distance. *IEEE Trans. Pattern Anal. Mach. Intell. 15*, 9 (1993), 850–863.

[38] JAGADISH, H., OOI, B., AND VU, Q. Baton: A balanced tree structure for peer-to-peer networks, 2005.

[39] JAIN, R., CHIU, D., AND HAWE, W. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *ArXiv Computer Science e-prints* (Sept. 1998).

[40] KNUTH, D. E. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[41] KORNACKER, M. *Marcel Kornacker's thesis, Access Methods for Next-Generation Database Systems*. PhD thesis, University of California at Berkeley, 2000.

[42] KRIEGEL., H. Similarity search in cad database systems.

[43] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, 8 (1966), 707–710.

[44] LI, D., WONG, K. D., HU, Y. H., AND SAYEED, A. M. Detection, classification and tracking of targets in distributed sensor networks. In *IEEE Signal Processing Magazine* (2002), pp. 17–29.

[45] LO, M.-L., AND RAVISHANKAR, C. V. Spatial joins using seeded trees. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1994), ACM, pp. 209–220.

[46] LOMET, D. B., AND SALZBERG, B. The hb-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst. 15*, 4 (1990), 625–658.

[47] MASHAYEKHI, H., AND HABIBI, J. K-nearest neighbor search in peer-to-peer systems. 2010.

[48] MCCREIGHT, E. Priority search trees. 257–276.

[49] MICHAIL ARGYRIOU, VASILIS SAMOLADAS, S. B. Grasp: Generalized range search in peer-to-peer networks. In *InfoScale* (Napoli, Italy, June 4-6 2008).

[50] ORENSTEIN, J. A. Spatial query processing in an object-oriented database system. *SIGMOD Rec. 15*, 2 (1986), 326–336.

[51] PAPADIAS, D., SELLIS, T., THEODORIDIS, Y., AND EGENHOFER, M. J. Topological relations in the world of minimum bounding rectangles: a study with r-trees. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1995), ACM, pp. 92–103.

[52] PRESS, W., TEUKOLSKY, S., VETTERLING, W., AND FLANNERY, B. *Numerical Recipes in C*, 2nd ed. Cambridge University Press, Cambridge, UK, 1992.

[53] RATNASAMY, S., KARP, B., LI, Y., YU, F., ESTRIN, D., GOVINDAN, R., AND SHENKER, S. GHT: A geographic hash table for data-centric storage. In *WSNA* (2002), pp. 78–87.

[54] ROUSSOPOULOS, N., KELLEY, S., AND VINCENT, F. Nearest neighbor queries. In *SIGMOD* (1995), pp. 71–79.

[55] SHU, Y., OOI, B. C., TAN, K.-L., AND ZHOU, A. Supporting multi-dimensional range queries in peer-to-peer systems. In *P2P* (2005), pp. 173–180.

[56] SPYROS BLANAS, V. S. Contention-based performance evaluation of multidimensional range search in peer-to-peer networks. In *InfoScale* (Suzhou, China, June 6-8 2007), ACM.

[57] TAI, K.-C. The tree-to-tree correction problem. *J. ACM 26*, 3 (1979), 422–433.

[58] TANIN, E., HARWOOD, A., AND SAMET, H. Using a distributed quadtree index in peer-to-peer networks. *VLDB Journal 16* (2007), 165–178.

[59] TANIN, E., NAYAR, D., AND SAMET, H. An efficient nearest neighbor algorithm for p2p settings. In *Proceedings of the 2005 national conference on Digital government research* (2005), dg.o '05, Digital Government Society of North America, pp. 21–28.

[60] U. FAYYAD, G. PIATETSKY-SHAPIRO, P. S., AND UTHURUSAMY, R. Advances in knowledge discovery and data mining.

[61] UNIVERSITY, S. G., GUHA, S., AND SRIVASTAVA, D. Approximate xml joins, 2002.

[62] WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *J. ACM 21*, 1 (1974), 168–173.

[63] WATERMAN, M. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman and Hall, 1995. Chapters 13,14.

[64] YANG, W. S., CHUNG, Y. D., AND KIM, M. H. The rd-tree: a structure for processing partial-max/min queries in olap. *Inf. Sci. Appl. 146*, 1-4 (2002), 137–149.

[65] YU, X., AND YU, X. An adaptive algorithm for p2p k-nearest neighbor search in high dimensions. In *Control and Automation, 2007. ICCA 2007. IEEE International Conference on* (30 2007-june 1 2007), pp. 2140 –2145.

[66] ZHENG, C., SHEN, G., LI, S., AND SHENKER, S. Distributed segment tree: Support of range query and cover query over DHT. In *IPTPS* (2006).