Universidade Estadual de Campinas
Instituto de Computação

Javier Alvaro Vargas Muñoz

Large-Scale Indexing of High Dimensional Data via Nearest Neighbor Graphs

Indexação de Grandes Coleções de Dados Altamente Dimensionais usando Grafos de Vizinhos mais Próximos

CAMPINAS
2020

Javier Alvaro Vargas Muñoz

Large-Scale Indexing of High Dimensional Data via Nearest Neighbor Graphs

Indexação de Grandes Coleções de Dados Altamente Dimensionais usando Grafos de Vizinhos mais Próximos

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Ricardo da Silva Torres
Co-supervisor/Coorientador: Prof. Dr. Zanoni Dias

Este exemplar corresponde à versão final da Tese defendida por Javier Alvaro Vargas Muñoz e orientada pelo Prof. Dr. Ricardo da Silva Torres.

CAMPINAS
2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

V426L

Vargas Muñoz, Javier Alvaro, 1993-
Large-scale indexing of high dimensional data via nearest neighbor graphs / Javier Alvaro Vargas Muñoz. – Campinas, SP : [s.n.], 2020.

Orientador: Ricardo da Silva Torres.
Coorientador: Zanoni Dias.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Método K-vizinho mais próximo. 2. Teoria dos grafos. 3. Análise por agrupamento - Processamento de dados. 4. Programação genética (Computação). I. Torres, Ricardo da Silva, 1977-. II. Dias, Zanoni, 1975-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Indexação de grandes coleções de dados altamente dimensionais usando grafos de vizinhos mais próximos
Palavras-chave em inglês:
K-nearest neighbor
Graph theory
Cluster analysis - Data processing
Genetic programming (Computer science)
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Ricardo da Silva Torres [Orientador]
José Fernando Rodrigues Júnior
Altigran Soares da Silva
Esther Luna Colombini
Hélio Pedrini
Data de defesa: 12-03-2020
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-5809-4228
- Currículo Lattes do autor: http://lattes.cnpq.br/5385182122254592

Universidade Estadual de Campinas
Instituto de Computação

Javier Alvaro Vargas Muñoz

Large-Scale Indexing of High Dimensional Data via Nearest Neighbor Graphs

Indexação de Grandes Coleções de Dados Altamente Dimensionais usando Grafos de Vizinhos mais Próximos

Campinas, 12 de março de 2020

Acknowledgements

I am sincerely grateful to Prof. Ricardo Torres and Prof. Zanoni Dias for their orientation and advices during this research work. Also, I would like to thank to Prof. Marcos André Gonçalves for his collaboration and contributions to this work.

I thank to my family, especially to my father, mother and brother John, that always were supporting me.

I thank to my friends, especially to those from the "Rep-Vep" for helping me to reduce the stress a bit during this doctorate period.

I thank the Prof. Helio Pedrini, Prof. Edleno Silva de Moura, and Dra. Fernanda Andaló for their valuable comments and critique during the qualification exam process, which were very important to improve the quality of this work.

I thank to Reasoning for Complex Data (RECOD) Laboratory and Institute of Computing at Unicamp, for providing the infrastructure that make possible this research work. Also, I would like to thank to the graduate program staff for their excellent work at supporting students.

This work was partially funded by the National Council for Scientific and Technological Development – CNPq (grant #164726/2018-7), and by the São Paulo Research Foundation (FAPESP) with grants #2014/12236-1, #2015/24494-8, #2016/50250-1, and #2017/20945-0. I would also like to thank to the FAPESP – Microsoft Virtual Institute (grants #2013/50155-0 and #2014/50715-9). This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) – Finance Code 001. I also thank the Norwegian University of Science and Technology (NTNU) for its support.

Resumo

Uma tarefa comum em muitas tarefas relacionadas às áreas de Recuperação de Informação, Visão Computacional e Aprendizado de Máquina, que envolvem objetos multimídia (p.ex., textos, imagens, vídeos), é a Busca dos Vizinhos Mais Próximos. Essa tarefa consiste em retornar o subconjunto de objetos de uma coleção que são mais similares a um objeto de consulta. Objetos multimídia são comummente mapeados para representações vetoriais altamente dimensionais utilizando técnicas guiadas pelos dados ou soluções artesanais, com o objetivo de facilitar o processamento desses objetos. Na última década, muitas dessas coleções de dados altamente dimensionais têm crescido massivamente com o uso extensivo das mídias sociais, rapidamente alcançando a escala de milhões de itens de dados, e, em alguns casos, bilhões. Isso tem motivado muitos esforços dedicados ao desenvolvimento de estruturas de dados para dar suporte a buscas eficientes dos vizinhos mais próximos em grandes coleções de dados. Recentemente a criação de Grafos de Vizinhos mais Próximos tem ganhado muita atenção já que essa abordagem tem demonstrado um desempenho consistentemente melhor que as abordagens clássicas: Árvores de Particionamento do Espaço (p.ex., KD-Trees) e *Hashing*. A ideia principal é criar um grafo onde cada vértice corresponde a um único objeto da coleção, e cada um deles está conectado com os outros mais similares na coleção. Motivados pelo sucesso desse grupo de técnicas em recentes *benchmarks*, quando avaliados em coleções de milhões de itens de dados, nós focamos esta tese em pesquisar novas abordagens para a criação de grafos de vizinhos mais próximos e algoritmos para realizar buscas nessas estruturas, com o objetivo de dar suporte a buscas mais eficientes e precisas. Além disso, como a maioria das técnicas existentes para busca de vizinhos mais próximos, incluindo as baseadas em grafos, não são capazes de escalar até coleções com bilhões de itens de dados, principalmente devido a problemas relacionados à memória, nós também investigamos técnicas para compressão de representações vetoriais e estratégias de poda para criação de grafos muito esparsos. As principais contribuições apresentadas nesta tese são: (i) o desenvolvimento de uma nova abordagem para criar eficientemente grafos esparsos de vizinhos mais próximos, baseado no resultado de múltiplos agrupamentos hierárquicos; (ii) a introdução de novas heurísticas que usam KD-Trees para melhorar os resultados das buscas por meio de uma melhor seleção do vértice inicial e guiando a navegação do grafo; (iii) um novo arcabouço supervisionado para conduzir a navegação em grafos de vizinhos mais próximos baseada na informação topológica dos vértices, reduzindo o número de vértices explorados para encontrar os vizinhos mais próximos verdadeiros; (iv) a primeira técnica baseada em grafos de vizinhos mais próximos que suporta buscas eficientes em coleções com bilhões de itens de dados, usando uma quantidade razoável de recursos computacionais. Os experimentos realizados para validar nossas propostas evidenciaram ganhos consistentes em relação às abordagens clássicas e resultados competitivos com o estado da arte em cenários com milhões e bilhões de itens de dados.

Abstract

A principal routine in many Information Retrieval, Computer Vision, and Machine Learning tasks involving multimedia objects (e.g., text, image, video) is the Nearest Neighbor (NN) search. This problem consists in returning a subset of objects from a collection that is more similar to a query object. Multimedia objects are commonly mapped to a high dimensional vector representations employing hand-crafted or data-driven descriptors aiming to facilitate their processing. In the last decade, many of these collections of high dimensional data have grown massively with the extensive use of social media, quickly reaching the scale of millions and, in some cases, billions. This has motivated a lot of efforts dedicated in the development of data structures to support efficient NN searches on large collections. Recently, the creation of Nearest Neighbor Graphs has gained a lot of attention since they have demonstrated to perform consistently better than classical approaches, e.g., Space Partitioning Trees (e.g., KD-Trees) and Hashing. The idea is to create a graph where each vertex corresponds to a unique collection's object, and each of them is connected with the other similar ones. Motivated by the success of these groups of techniques in recent benchmarks, when evaluated on million-size datasets, we focus on this thesis to investigate novel approaches to creation of nearest neighbor graphs and algorithms to perform search over them, aiming to support more efficient and accurate NN searches. Furthermore, since most of existing techniques for NN search, including state-of-the-art graph-based approaches, are not able to scale up-to billion-size datasets, this principally caused by memory related issues, we also investigate compression techniques for vector representations and pruning strategies for creation of very sparse graphs. The principal contributions presented in this thesis are: (i) the development of a novel approach to create efficiently sparse nearest neighbors graphs, based on the results of multiple hierarchical clustering executions; (ii) the introduction of novel heuristics that employ classical KD-Trees to improve graph search results by better selecting the initial vertex and *guiding* the graph traversal; (iii) a novel learning framework that supports the navigation on nearest neighbor graphs based on the topological information of vertices, reducing the number of vertices explored to reach the true nearest neighbors; (iv) the first nearest neighbor graph-based technique that supports nearest neighbor searches on billion-size datasets, using a reasonable resource consumption. The experiments conduced to validate our proposals evidenced consistent gains over classic approaches and competitive results with state of the art in both million and billion scale scenarios.

List of Figures

List of Tables

List of Abbreviations and Acronyms

ADC	Asymmetric Distance Computation
ANNS	Approximate Nearest Neighbor Search
AQ	Additive Quantization
FLANN	Fast Library for Approximate Nearest Neighbors
GIST	Global-Invariant Scale Transform
GNO-IMI	Generalized Non-Orthogonal Inverted Multi-Index
GP	Genetic Programming
HCNNG	Hierarchical Clustering-based Nearest Neighbor Graph
HNSW	Hierarchical Navigable Small World
IMI	Inverted Multi-Index
IVFADC	InVerted File with Asymmetric Distance Computation
LSH	Locality Sensitive Hashing
NN	Nearest Neighbor
OPQ	Optimized Product Quantization
PCA	Principal Component Analysis
PQ	Product Quantization
SD	Search Dependent
SDC	Symmetric Distance Computation
SI	Search Independent
SIFT	Scale-Invariant Feature Transform
SW	Small World
VLAD	Vector of Locally Aggregated Descriptors

Contents

Chapter 1

Introduction

A large amount of multimedia content is being generated every second due to the large use of low-cost portable devices and the growth of internet access in the last decades. This led to the creation of large collections of multimedia content, and, at the same time, to the need for mechanisms that, efficiently, support the task of retrieving relevant information from these collections. Since it is too difficult to process raw multimedia objects (e.g., image, text, audio, and video), advances in multimedia representation have allowed the design of techniques that map multimedia objects into vector spaces (commonly in the scale of hundreds or thousand of dimensions) fostering the creation of effective search systems. There are two main aspects to be considered for this kind of data: dimensionality and size. In this work, we refer as *dimensionality* to the number of dimensions of vector representations, and as *size* to the number of objects in the collection.

In this context, the Nearest Neighbor (NN) search is a problem broadly studied in the literature. This problem consists in searching for the nearest vectors to a query vector from a set of vectors with arbitrary dimensionality (that represents multimedia objects), considering a given distance function. This problem is involved in several applications such as large-scale image search [43,57], semantic document retrieval [20], global image feature matching for scene recognition [29], human pose estimation [50,59], 3D reconstruction [15], and classification with a large number of classes [30]. There are two well-known variants of this problem: the K-NN and r-NN (or range query). The first consists in searching for the set of K-vectors that are closer to the query than any other vector in the set. The second aims to find all vectors that are inside of the hypersphere with the query as center and radius r. This thesis refers only to the first problem (K-NN).

The naïve solution to this problem is the linear search algorithm, that is, to scan the entire collection of feature vectors and return those with the lowest distance to the query vector. However, this solution is not scalable in scenarios with millions or billions of vectors and high concurrency of queries. Alternative data structures were proposed for efficient exact searches yielding a logarithmic time complexity (e.g., KD-Tree [11] and R-Tree [26]) for low dimensional data. However, their efficiency drops dramatically, with performance similar to linear search, when the data dimensionality increases. Since more accurate methods for multimedia representation usually employ high dimensional vector spaces, the use of these data structures for exact search is unpractical. Many Approximate Nearest Neighbor Search (ANNS) approaches were proposed in the literature, that performs searches efficiently on high dimensional data at the cost of losing precision of results. Different from other research fields, usually these “approximate” techniques are not able to guarantee a certain error rate on search results, the accuracy of results will depend principally on the quantity of objects explored from the collection.

Two classic approaches for the ANNS problem were employed in the literature: tree indexing schemes and hashing-based solutions. The first presents a widely-used way to organize data. At each tree level, data are split into subsets, based on some criteria, which are then recursively applied to each subset until some stop condition is reached. Search is performed traversing from the root to the leaves. Usually, tree indexing techniques present cheap costs for index construction. Within this category, popular methods include: Randomized KD-Trees [61], Hierarchical K-Means [47], FLANN [49], and VP-Tree [70]. On the other hand, hashing-based techniques aim to use a hash function to create binary signatures of original feature vectors. Those functions map nearest vectors to the same bucket or the nearest buckets in a hash table. Then, search can be performed around the query bucket in the hash table. Differently from traditional hash, collisions are intended to be maximized. The use of multiple hash functions increases the probability of finding the nearest neighbor, but also increases the storage cost to save the hash tables. Locality Sensitive Hashing (LSH) [2, 10, 44] (and their variants) is one of the best-known methods of this group. Furthermore, the generated binary signatures help to reduce the storage cost of vectors in memory, which is of paramount importance when the database size scales up-to billions of vectors. Also, this allows a fast hamming distance computation between binary signatures.

Recently, the use of nearest-neighbors graphs has attracted a lot of attention since they have demonstrated in recent works [4, 22, 27, 46] to outperform consistently classic approaches mentioned above. In NN-graph-based techniques, firstly, all feature vectors of the collection are indexed by means of a graph, where each feature vector is associated with a unique vertex and connected to the other closer ones (according to a given distance function). Then, search of NN is performed by traversing the graph in a greedy way, with similar strategy to the *hill climbing* optimization procedure [54]. Search starts in some vertex, and, in each step, selects the neighbor of current vertex that is closer to the query as the next vertex to be explored. The traversal stops commonly when a maximum number of vertices explored is reached.

The only known approach to creation of exact NN graphs relies on employing linear search to find the k-NN over the entire collection, which is unpractical for large collections due to its quadratic complexity on the collection size. Thus, all approaches found in the literature proposed approximate efficient approaches for the creation of NN graphs. In this thesis, we investigate efficient approaches for creation of NN graphs based on the results of multiple clustering executions. Furthermore, we expand this algorihtm to be able to index datasets with billions of vectors, considering the limitations of time and memory. We also propose some heuristics and a learning framework to improve the navigation process in greedy search algorithm for NN graphs. The research topics covered by our work are shown in Figure 1.1.

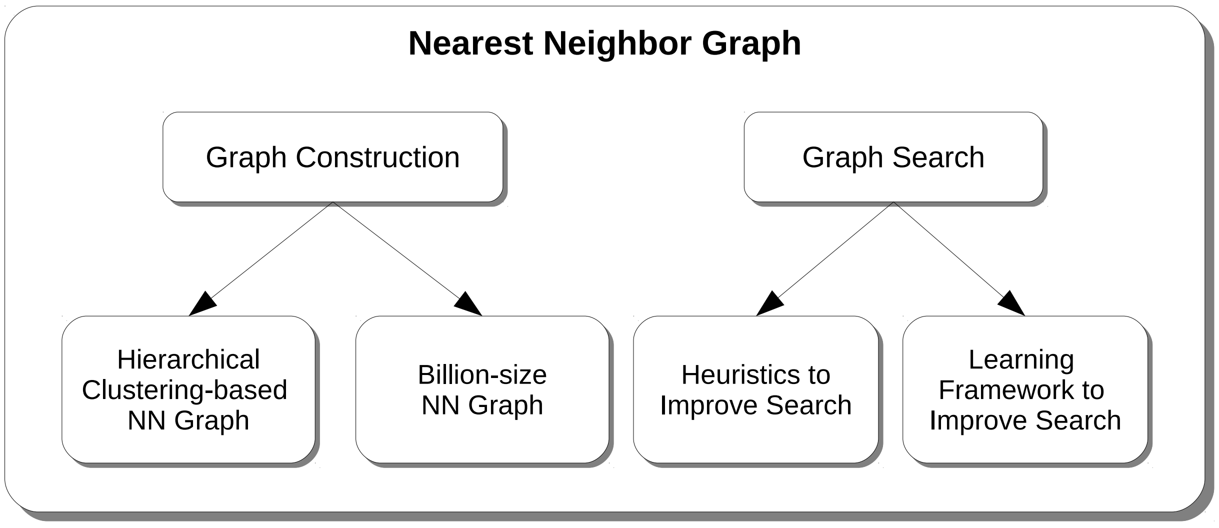

Nearest Neighbor Graph
Graph Construction
Graph Search
Hierarchical Clustering-based NN Graph
Billion-size NN Graph
Heuristics to Improve Search
Learning Framework to Improve Search

Figure 1.1: Research topics covered in this thesis.

1.1 Motivation

In this section, we present the motivational aspects of our work.

1.1.1 Hierarchical Clustering-Based Nearest Neighbor Graphs

Clustering is among the most important unsupervised learning tasks, which has been applied successfully in a wide range of domains, such as exploratory data analysis, machine learning, artificial intelligence, pattern recognition, computer vision, information retrieval, etc. In the context of the Euclidean space, a clustering procedure aims to group vectors into clusters with respect to their proximity in the feature space. Since the idea of nearest neighbor graphs is also similar, to connect vectors with others near to them, the clusters' information can be exploited to determine the neighbors of vectors. Clustering algorithms found in the literature have usually a lower complexity than brute force approach described above to create exact NN graphs, addressing the time-related scalability issues when working with large collections. Specially, the hierarchical clustering has a $O(N \log N)$ bounded time complexity (being N the dataset size), which makes it very suitable in scenarios with large datasets. We investigate in this thesis approaches to exploit the information of hierarchical clustering results to construct efficiently an NN graph to support more efficient and accurate NN searches. Moreover, along with the NN graph construction, we introduce two heuristics: one for selection of a non-fully randomized starting vertex for search, and the other for pruning edges at search time, avoiding explore unnecessary vertices.

1.1.2 Learning to Navigate on Nearest Neighbor Graphs

Methods for construction of NN graphs found in the literature differ considerably in their strategy, but the search algorithms employed do not change significantly from the greedy approach mentioned before. In any step of the graph traversal, the next vertex is selected

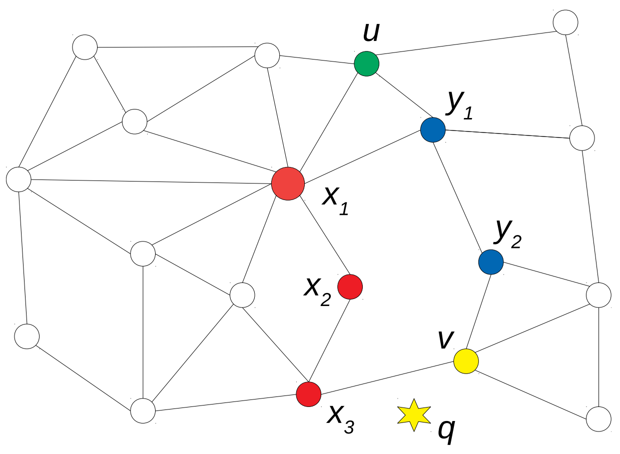


Figure 1.2: Example of two different paths from u to the nearest neighbor to q (v).

based only on its distance to the query. The selection of the next vertex to be explored plays an important role at converging efficiently to the true nearest neighbors. Other kinds of properties of vertices could help to improve the selection of the next vertex, such as, the local topological properties of vertices. For example, if one vertex has a very high degree, then we could avoid exploring it, since it would be very costly to visit all its neighbors. Figure 1.2 shows a synthetic example of a search on an NN graph. In this example, if the navigation was only based on the distance to the query, then starting in vertex u, the next vertex to be visited would be x_1, since it is its closest neighbor to the query q. Following this criterion, the path chosen to reach the nearest neighbor to the query q would be $u \to x_1 \to x_2 \to x_3 \to v$. As all neighbors of vertices on this path need to be visited, all neighbors of x_1 (the vertex with higher degree) will be visited, increasing significantly the cost of this path (totalizing 12 vertices explored). However, if we consider the degree of vertices in the selection of next vertex, we could follow an alternate path with a lower cost; in the example, $u \to y_1 \to y_2 \to v$ (totalizing 8 vertices explored).

There are many measures that capture different aspects of the local topology of vertices' neighborhood [16], and all this information can be combined to help in the selection of next vertex at graph navigation, as seen in the case of the degree of vertices. One focus of this thesis is on the exploitation of different local topological properties via a Genetic Programming-based learning framework to improve search result accuracy on NN graphs. Genetic Programming has been shown to perform well in scenarios like this, combining successfully different evidences [1, 17, 19, 39, 65].

1.1.3 Scaling Nearest Neighbor Graphs to Billion-Size Datasets

Given the massive growing of multimedia data in the past years, the design of efficient retrieval mechanisms that support searches over billions of objects has gained a lot of attention from the information retrieval and computer vision communities. The scale of data presents some limitations to common approaches for NN search, such as those based on space partitioning trees [11, 47, 49, 53]. Their main limitation relies on the main memory needed to load all feature vectors in order to support efficient searches, leading to the acquisition of very costly architectures. For example, a dataset containing one billion of vectors with 100 dimensions, storing each dimension value as a single-precision float (4 bytes), would require 372.52 GB of RAM just for load the feature vectors, without taking into account the memory required for storing the data structure. Furthermore, this number of dimensions is low compared to some real applications, where vectors with thousands of dimensions are generated. Hashing approaches can deal with this memory issues since they produce a binary signature of original vectors, commonly employing just a couple of dozen of bits per vector. In recent years, a new research line has emerged, focusing on developing compressing vectors approaches based on quantization [6, 8, 23, 34, 71]. These approaches, as in the case of hashing-based techniques, produce compact representations of original vectors employing just a couple of dozen of bits. However, differently from hashing, quantization-based techniques are able to reconstruct original vectors from their compact representations, with a small error. This is very convenient at the moment of comparing two vectors, since with hashing-based compact representations the comparison can be only done in the hamming space, but with the quantization-based it can be done in the original feature space by reconstructing the vectors. Also, a better memory-accuracy tradeoff of quantization-based approaches has been shown [23,52] when compared with hashing-based in the task of ANNS.

On the other hand, as mentioned before, NN graph techniques [22,27,46] have demonstrated to outperform consistently the classic approaches for ANNS. However, results reported on those works only employed datasets with at most a dozen of millions of vectors. In the scenario discussed above, with billions of vectors, state-of-the-art NN graph techniques suffer from the same scalability issues as classical indexing schemes, related to the memory required to load the whole dataset in memory. Additionaly, NN graphs carry the cost of storing the graph itself (with billions of vertices and edges). As current techniques for construction of NN graphs could generate graphs with up to hundred of neighbors per vertex, it will lead to a similar memory cost as to load the whole dataset on memory. In this thesis, we investigate the application of quantization-based compression techniques in the creation of NN graphs to deal with the memory issues related to the storage of vectors in memory. Also, we investigate heuristics and learning schemes for pruning edges at graph construction time, aiming to generate sparse NN graphs that demand a reasonable storage cost.

1.2 Hypotheses and Research Questions

This thesis introduces a novel scheme for efficient construction of NN graphs over large scale datasets, taking into account computational resources limitations. Also, we introduce new approaches for improving the query time–accuracy tradeoff of NN searches conduced over NN graphs. The thesis hypotheses are presented in the following, with the respective research questions that allow us to address our hypotheses:

Hypothesis 1: *The use of multiple clustering leads to the construction of NN graphs, which are faster to traverse at search time.*

Q1.1 What is the best way to connect points inside clusters?

Q1.2 How can sub-graphs created by clusters be merged?

Q1.3 Which clustering algorithm should be employed?

Hypothesis 2: *The use of classical tree structures for indexing improves the NN search results on NN graphs, with no significant extra cost.*

Q2.1 How KD-Trees can be employed to improve the starting vertex selection on NN graph searches?

Q2.2 How KD-Tree-like structures can be employed to avoid visit unnecessary vertices at graph navigation process?

Hypothesis 3: *The use of topological information of vertices (along with the distance to query) through a learning scheme leads to a better selection of next vertex (in each step of NN graph search), which fosters the earlier discovering of the true NN.*

Q3.1 How to combine the topological properties with the distance?

Q3.2 Which learning technique can be used to find near optimal combinations of topological properties?

Q3.3 Which topological properties should be considered?

Q3.4 Which function should be optimized in the learning process?

Hypothesis 4: *Compressing vectors via quantization schemes and the adoption of suitable pruning strategies at construction time allow the construction of NN graphs on billion-size datasets with a reasonable memory consumption and time construction.*

Q4.1 Does the compression of original vectors affects to the accuracy of search on NN graphs?

Q4.2 Which heuristics can be used for pruning edges at graph construction?

Q4.3 Can we obtain search performance improvements by exploiting the topological properties of vertices for pruning edges at graph construction?

Q4.4 Does our NN graph technique still maintain its top search performance at NN search when compared to the state-of-the-art schemes for billion-scale ANNS?

Q4.5 How much resources need the proposed NN graph technique in comparison with the state of the art?

1.3 Key Contributions

We envision this research contributing to the areas of Machine Learning, Computer Vision, and Information Retrieval. We can summarize our key contributions as follows:

1. A novel approach to efficient construction of NN graphs, with comparable search performance than state-of-the-art techniques for ANNS (Chapter 3).

2. Two novel heuristics to improve search on NN graphs using classical KD-Trees as auxiliary data structures: one for initial vertex selection, and the other for avoiding exhaustive exploration of vertices' neighbors (Chapter 3).

3. A Genetic Programming (GP) framework that aims to discover a near-optimal combination of local topological features of vertices along with the classical *distance-to-the-query*, that improves the criterion for selection of the next vertex to be explored in the search algorithm (Chapter 4).

4. The first reported memory-aware technique for creation of sparse NN graphs on billion-size datasets (Chapter 5).

5. The extension of the GP framework above applied to the selection of vertices' neighbors at NN graph construction stage, in scenarios with restricted degree of vertices (as in the billion-size datasets, Chapter 5).

1.4 Text Organization

This thesis is organized as follows. Chapter 2 describes general related work, including classic approaches for ANNS, NN graphs, and Quantization-based techniques. In the last part of that chapter, we also include a detailed description of Genetic Programming-based learning process, which will be used in Chapter 4 and Chapter 5 to describe our proposals. In Chapter 3, we introduce our technique for efficient construction of NN graphs, and the two heuristics that uses classical KD-Trees for improving search performance on NN graphs. Chapter 4 introduces the GP-based framework to discover a near-optimal combination of local topological features of vertices to improve the criterion for selection of the next vertex. Chapter 5 goes beyond and extend approaches presented in Chapter 3 and Chapter 4, to propose an approach for creation of NN graph on billion-size datasets. Finally, Chapter 6 summarize the discoveries of this work, presents the conclusions, and draws future research directions.

Chapter 2

Related Work and Related Concepts

We divided the related work and related concepts into four sections. Section 2.1 presents a revision of the classic approaches in the literature for ANNS. Section 2.2 describes briefly several state-of-the-art NN graph-based approaches for ANNS, including the description of the methods for NN graph construction and the search algorithm employed on these graphs. Section 2.3 presents a literature review of quantization-based method for compression of high dimensional data, and the state-of-the-art data structures, for indexing these compressed vectors, that support billion scale ANNS. Finally, Section 2.4 introduces Genetic Programming, a learning technique that will be employed in next chapters.

2.1 Classic Schemes for Nearest Neighbor Search

The problem of NN search has been broadly studied in the literature for many decades. Two classic schemes were proposed to address this problem: space partitioning trees and hashing-based schemes. The advantage of these approaches relies on the cheap cost for index construction. In the following sections, we present a literature review of techniques proposed in each group of approaches.

2.1.1 Space Partitioning Trees

A strategy broadly studied in the literature for NN search is to organize data in a tree data structure. Data are partitioned into subsets, based on some criteria, at each tree level until some stop condition is satisfied, e.g., a minimum size is reached. When a search is performed, the tree is traversed from the root to the leaves, which probably contain the closest point. A backtracking approach can be used to further explore other nodes and increase the probability of finding the most similar objects.

Tree structures for exact search (e.g., KD-Tree [11], Ball Tree [53], and Cover Tree [12]) are very efficient for low dimensional data, but their performance decreases quickly when dimensionality increases. KD-Tree is one of the most cited tree structures for exact search. In the algorithm of KD-Tree index construction, at each level and in a sequential order, a dimension is selected to split the data and a point is used to better balance the division. Figure 2.1 shows an example of a KD-Tree created over a set of 2D points. At root level the set of points are split by L_1, placing the points located in the negative side of L_1 to

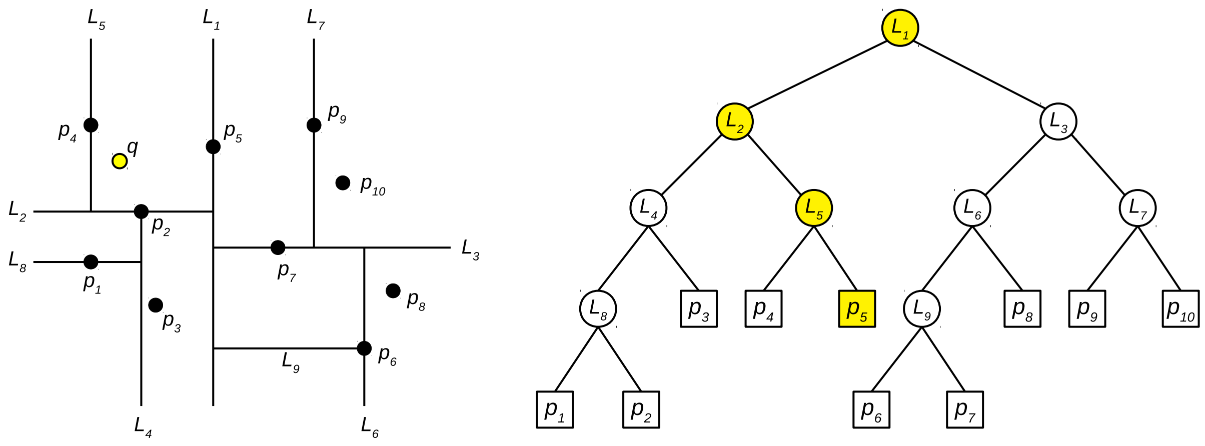


Figure 2.1: Example of a KD-Tree in a synthetic 2D dataset.

the left sub-tree, and those on the positive to the right sub-tree. This division continues in each level, alternating the axis, until just one point is left. When the query q is issued, the tree is traversed starting in the root L_1 and, in each level, selecting the sub-tree to which q corresponds, according to the current line. Thus, the search will traverse the path $L_1 \rightarrow L_2 \rightarrow L_5 \rightarrow p_5$. A backtracking approach can be used to explore other leafs and increase the probability of finding the true NN. A well-known variant of KD-Trees for ANNS is named Randomized KD-Trees [61]. In their construction, several KD-Trees are created, where the dimension to split the data is selected randomly. Then, at query time, search is performed on all the trees, and is stopped when a fixed number of leaves is explored. This method is implemented in the widely used Fast Library for Approximate Nearest Neighbors (FLANN) [49].

Muja and Lowe [47] proposed the hierarchical k-means tree, where, at each level, the data are split using the k-means algorithm into K subsets, and then the same algorithm is applied recursively to the subsets generated. The recursion is stopped when the size of the subset is less than K. In the traverse from the root, at each level, the branch with centroid closest to the query is taken. The exploration is stopped when a fixed number of nodes are visited.

Methods for tree partitioning can be roughly divided into two groups. Techniques that divide data points with respect to hyperplanes and clustering based. From the first group, we can mention KD-Tree [11,61], PCA-Tree [62], and Random Projection tree [18]. In the second, we can find hierarchical clustering tree [49], hierarchical k-means tree [47], Ball tree [53], Cover tree [12], Geometric Near-neighbor Access tree [13], MDF-tree [25], Lower Bound tree [14], and Vantage Point tree [70].

2.1.2 Hashing

The principle of hashing-based techniques is to use a hash function to generate binary signatures for original vectors, which are then used as keys for buckets in a hash table. The hash functions are intended to map nearest vectors to the same bucket, allowing an efficient search when a query is issued by exploring only the vectors contained on the bucket with nearest key to the query's key. Differently from conventional hashing where a minimum of collisions is desired, techniques based on hashing for NN search do the contrary, aim to maximize the collisions of nearest vectors.

One of the seminal works was presented by Indyk and Motwani [32]. Currently, the most cited method in this family is probably Locality Sensitive Hashing [2] (LSH). In this technique, multiple hash tables are created and used at the same time to obtain a reduced list of candidates for each function. Next, an exhaustive search is performed in all the list of candidates to find the nearest neighbors. Thus, the more hash tables are created, the higher the probability to determine the nearest points, but also, this increases linearly the memory consumption.

Lv *et al.* proposed Multi-Probe LSH [44], a method that reduces the high storage requirement by reducing the number of hash tables. The authors' idea is based on the supposition that if a nearest neighbor is not in the same bucket as the query, then is highly probable that it is contained in close buckets. In this way, the algorithm makes a harder exploration of closest buckets, reducing the number of hash tables needed to achieve high recall values. Bawa *et al.* [10] presented a variant of LSH, which self tunes its parameters to the data. On the other hand, many supervised approaches have been proposed in the literature [21,40,42,60,66] which aim to learn hash functions that encode original vectors in such a way that allow effective and efficient search in the coding space. However, as it is common in supervised approaches, these methods demand a extra learning step (sometimes very costly), increasing the time for index construction. For a detailed revision of the literature on supervised approaches, see the survey by Wang *et al.* [69].

2.2 Nearest Neighbor Graphs

NN graphs present an easy-to-understand scheme for multimedia indexing. Each multimedia collection object becomes a vertex on the graph, and each of them are connected to other more similar ones, i.e., to those objects with the lower distance between their feature vectors. The naïve algorithm for construction of NN graphs consists in, for each vertex, scanning the whole collection and select the K-nearest vertices as their neighbors. Therefore, resulting in a quadratic solution at the collection size. In real scenarios with millions or billions of feature vectors, this naïve algorithm is unpractical. In the following, we describe some efficient approaches that have been proposed to speed up creation of approximate NN graphs. Also, details on how searches are performed over those graphs are provided.

2.2.1 Creation of NN Graphs

Many approaches have been proposed to construct approximate nearest neighbors graphs; here we briefly describe some of them. Since the idea of construction of NN graphs over multimedia collections were employed also in other tasks, such as image annotation [31, 41, 63, 64], we only considered those techniques that were employed in the task of NN search.

Harwood and Drummond [27] proposed an incremental algorithm to create an NN graph, called Fast Approximate Nearest Neighbour Graphs (FANNG). Initially, the set of vertices of the graph are composed of all objects in the collection (represented by their feature vectors), and it is fully disconnected (no edges). Then, in each iteration, two vertices v_1 and v_2 are selected randomly, and a naïve greedy search is performed using v_1 as starting vertex and v_2 as query. If the search fails to arrive at v_2, an edge is added between the last node visited and v_2. This process is repeated until enough edges are created to connect properly the graph, about $50N$ times in reported results for million-size datasets. An important strategy employed refers to the deletion of *occluding* edges. This makes the graph to preserve the closest and the spreadest neighbors of each vertex, leading to a more efficient graph traversal.

Malkov *et al.* [45] introduced the Small World Graphs (SW-graph). Differently from the strategy of FANNG construction algorithm, initially, it is created an empty graph (no vertices or edges). Then, at each iteration, a new vertex (object collection) is selected and a search over the current graph to find a fixed number of its nearest neighbors is performed. The new vertex is added to the graph and non-directed edges are created between this vertex and the set of nearest neighbors found. This is repeated until all collection vectors are included in the graph. Their objective is to create an approximation to the Delaunay graph [5], and, at the same time, maintain “long” edges to allow logarithmic navigation on the graph. This property is known as *Small World* [37]. However, in the final graph, many vertices end with high degree, increasing the number of distance calculations to reach the nearest neighbors.

A recent proposed approach, Hierarchical Navigable Small World (HNSW) [46], creates a hierarchy of NN graphs, in which each collection vector is assigned randomly to a maximum hierarchy level. Long edges (edges that connect distant vectors) are presented in top layers, and the short ones in bottom layers (edges that connect nearest vectors). The construction of graphs is analog to SW-graph. Incrementally, vectors are added into the graphs, starting in the graph at the vector’s maximum level and descending up to the graph in the ground layer, linking them to the nearest ones in each level. The search algorithm proposed by the authors is quite different from search algorithms employed from most of NN graph-based techniques (which will be described in next section), since it is created a hierarchy of graphs and not just a global one. At query time, search starts in some vertex in the top layer’s graph and traverse the graph to find the closest ones to the query. When a local optimum is reached, one level is descended in the hierarchy, and search is started using as starting points the nearest vertices found at the above level. This is repeated until the ground layer is reached. This method showed to be one of the most competitive baselines in our experiments across all datasets.

Unlike the incremental strategy used by the aforementioned initiatives, the algorithm for graph construction proposed by Dong *et al.* [22] (KGraph) initially assumes a random set of neighbors for every vertex. In each iteration, based on the heuristic – *a neighbor of a neighbor is also likely to be a neighbor* –, these sets are updated by selecting the nearest points from the actual set and the neighbors of neighbors. The process is repeated until the sets of neighbors of each vertex do not change significantly. Although the construction algorithm converges fast and approximates with high recall the real nearest neighbors of each vector from the collection, the resulting graph does not present good properties for search, as it aims to construct an exact NN graph and experimental results showed low performance for search on these graphs (see Section 3.3.6).

The idea of creating an NN graph using the results of multiple clustering executions was explored previously by Wang et al. [67]. Their proposed method aims to create efficiently a near-exact NN graph, maintaining a list of the k-NN for each point. These lists are updated in each clustering execution by creating a complete subgraph for each cluster and, then, maintaining for each vertex, those new neighbors that are closer than any of their previously found k-NN. The execution of clustering procedures is stopped when there is not a significant modification of k-NN lists in the current iteration. Then, aiming to discover and include even more true NN on those lists, these are improved through a neighbor propagation phase. Similar to KGraph, this method also aims to create an exact NN graph, and, consequently, experimental results showed a poor search performance compared to the other NN graph-based approaches.

2.2.2 Search in NN Graphs

As seen in previous section, techniques for NN graph construction are based on conceptually different strategies. However, most of search algorithms used on these NN graphs rely on roughly the same strategy. This strategy is detailed in Algorithm 1, which considers a given NN graph G, a query point q, and a maximum number of distance computations T. The traversing starts by initializing the global minimum (nearest neighbor) at some vertex on the graph (lines 2-3), and a priority queue to help in selecting the next vertex to be explored (line 4) at each step (an iteration of loop in line 5). The vertex in the queue to be selected will be the one with the minimum distance to the query (Euclidean distance between their corresponding feature vectors). In each step of traversal, after the next vertex is selected (line 6), the neighbors of this vertex are scanned (line 7). For any neighbor, if it was not visited previously (line 8), then it is pushed to the queue (lines 9-10). Also, if any neighbor is closer to the query than the global minimum, then it is updated (lines 12-14). When the maximum number of vertices to be explored is reached, the global minimum discovered is returned (line 15).

An example is illustrated in Figure 2.2, with starting vertex v_0 and query point q (red point). The first vertex is taken from the queue (v_0), then, their neighbors are explored and pushed to the queue. In the next step, from the vertices on the queue, the closest to the query is taken and explored (v_1), and so on until the maximum number of distance calculation is reached. Note that, the next vertex to be explored does not depend only on the neighbors of the current vertex, as all vertices on the queue are candidates (blue vertices, after the 3rd step).

Malkov *et al.* [45] presented a slightly variation of this algorithm, changing the stop condition. According to their proposal, a graph is traversed until a set of K-nearest neighbors remain unchanged at a given iteration. Also, they proposed to perform multiple searches, with different starting vertices and then combine the results of such searches to return the best-k vertices. Harwood and Drummond [27] also proposed to use the nearest vertex to the data mass center as starting vertex for search.

```
1  Function SearchNN(G, q, T)
       Data: NN graph $G$, query point $q$
       Data: maximum distance calculations $T$
       Result: nearest vertex $n$, nearest vertex distance $d$
2      $n \leftarrow$ some vertex in $G$
3      $d \leftarrow distance(n, q)$
4      $Q \leftarrow$ initialize priority queue with tuple $[n, d]$
5      while $T > 0$ do
6          $v \leftarrow Q.pop()$
7          foreach $u \in Neighbors(v)$ do
8              if $T > 0$ and $u$ not visited then
9                  $d^* \leftarrow distance(u, q)$
10                 $Q.push([u, d^*])$
11                 $T \leftarrow T - 1$
12                 if $d^* < d$ then
13                     $n \leftarrow u$
14                     $d \leftarrow d^*$
15     return $n, d$
```

Algorithm 1: Algorithm for search in NN graphs.

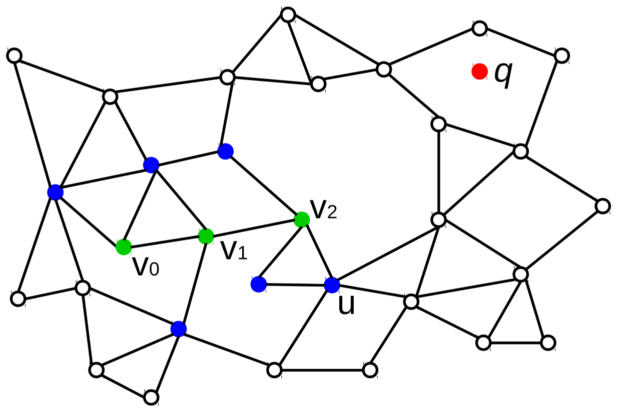


Figure 2.2: Example of search using Algorithm 1.

2.3 Quantization-Based Indexing

Most of techniques for ANNS mentioned before, except for a couple of hashing-based techniques, only reported their experiments on million-size datasets. However, at present there are already many multimedia collections that reached the scale of billions. The application of those techniques for billion-size datasets would require very costly infrastructures, especially, with hundreds or thousands of gigabytes of RAM to load all feature vectors in memory. Quantization-based schemes for lossy data compression addressed with success the memory restrictions presented by most of techniques described before, employing just a couple of dozens of bits per feature vector. In addition, many quantization-based indexing structures were proposed to support efficient searches over the compressed vectors. We present in the following a literature review of the quantization-based approaches to vector compression and indexing structures.

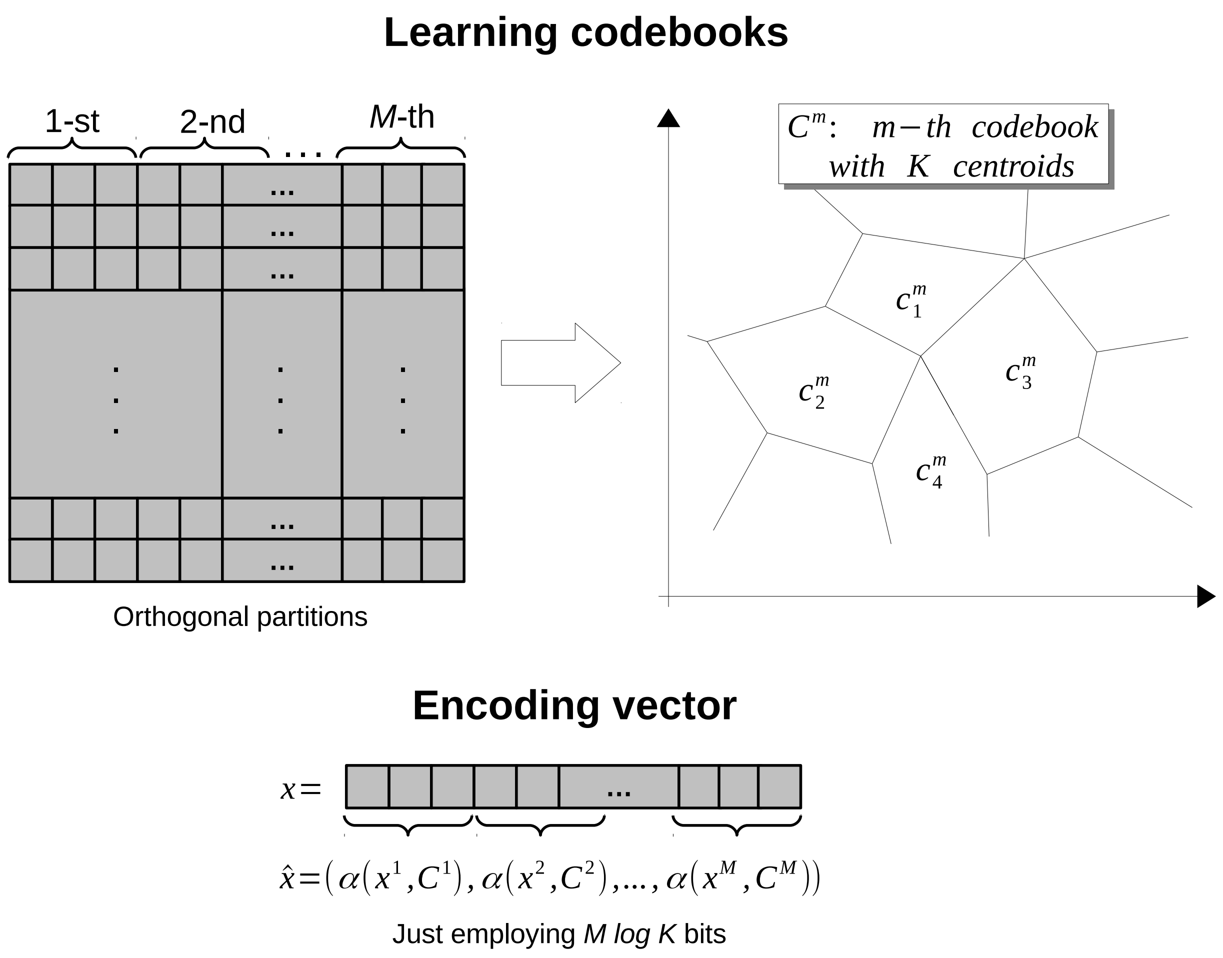
2.3.1 Vector Compression

Jegou et al. [34] reported one of the first results of NN searches on a billion-scale dataset. They introduced the Product Quantization (PQ) scheme, in which the original vector space is partitioned orthogonaly and each partition is quantized independently via the K-means algorithm. Thus, original vectors are encoded by the tuple of the nearest centroids' indices in each codebook, using just a couple of dozen of bits. This scheme is illustrated in Figure 2.3. The orthogonal partition is performed by splitting the original vectors dimensions into M-disjoint subsets with equal size. Then, a set of codebooks $C = \{C^1, C^2, ..., C^M\}$ is learned on the partitions, each one consisting of the set of K-cluster's centroids $C^m = \{c_1^m, c_2^m, ..., c_K^m\}$ found by the K-means algorithm in their respective partition. The compressed representation for given vector x is computed after applying the same space partition to x, and is given by the tuple of values $\hat{\mathbf{x}} = (\alpha(x^1, C^1), \alpha(x^2, C^2), ..., \alpha(x^M, C^M))$, where $\alpha(x^m, C^m) = \arg\max_i d(x^m, c_i^m)$ computes the index of nearest centroid in C^m for the sub-vector x^m and a given distance function d. Therefore, the compressed representations just need $M \log K$ bits of storage per vector. Since PQ [34], many works have dedicated efforts in order to develop techniques that create compact representations that approximate original vectors with less quantization error. A common approach employed to improve the coding accuracy is to rotate original vectors by a matrix learned together with codebooks [23, 52], also known as Optimized Product Quantization (OPQ).

A generalization of PQ with non-orthogonal partitions were proposed in [6, 71], in which original vectors are approximated through the sum of centroids selected from different codebooks learned on the original vector space. Non-ortogonal partitions showed to improve consistently coding accuracy, compared to PQ. This is explained by the fact that non-orthogonal partitions do not assume the independency between the data distribution of different subspaces (created by the partitions). However, some of these approaches sacrifice encoding efficiency in favor of this coding accuracy, as in the case of Additive Quantization (AQ) [6]. Tree Quantization (TQ) [8] presented a slightly non-orthogonal partition of vector space, since any dimension could be quantized by more than one codebook, getting a comparable coding accuracy with AQ, but maintaining the low cost for coding as in PQ. Also, the idea of creating local codebooks was explored in [36], by clustering original vectors and applying independent OPQ over vectors in each cluster.

2.3.2 Data Structures for Compressed Vectors

Although quantization-based techniques for vector compression deal successfully with the memory issues presented by classic approaches for NN search on billion-size datasets, it is still very time consuming to perform an exhaustive exploration to find the NN over these collections, even more with a high concurrency of queries. Jegou et al. [34] took advantage of a well-known data structure from the information retrieval field, known as *inverted index*, and employed it in their IVFADC system. First, original vectors are clustered to



Learning codebooks
1-st
2-nd
...
M-th
Orthogonal partitions
C^m: m−th codebook with K centroids
c_1^m
c_2^m
c_3^m
c_4^m
Encoding vector
$x=$
$\hat{x}=(\alpha(x^1,C^1),\alpha(x^2,C^2),\dots,\alpha(x^M,C^M))$
Just employing M log K bits

Figure 2.3: Illustration of Product Quantization-like schemes.

find the centroids that will be used as keys for the inverted lists. Database vectors are indexed by adding them to the list with the nearest key. Only the residuals between the keys and the original vectors are encoded via PQ. At search time, lists with the closest keys to the query are scanned first, since they are more likely to contain the true NN. A scanning rate can be set to avoid exhaustive exploration. Inverted Multi-Index (IMI) was introduced by Babenko and Lempitsky [7]. This approach led to significant improvements in terms of search accuracy with respect to IVFADC. IMI also indexes vectors into lists, but the original space is split orthogonally and each partition is quantized separately, then, keys for lists are obtained by the cartesian product of codebooks' centroids. A recent work [9] introduced GNO-IMI, a generalization of IMI in which lists' keys are also composed of the cartesian product of two codebooks. However, differently from IMI, the first is learned over original vectors and the second over the residuals w.r.t. the first codebook. This non-orthogonal partition for indexing, as in the case of quantization techniques for compression, obtained significant improvements in terms of search accuracy, specially in cases where there is correlation between subspaces.

All above indexing structures could be easily adapted to be used along with almost any encoding technique. For example, IMI was used previously for indexing vectors encoded with PQ [7], OPQ [24], and LOPQ [36]. Search on these structures follow roughly the

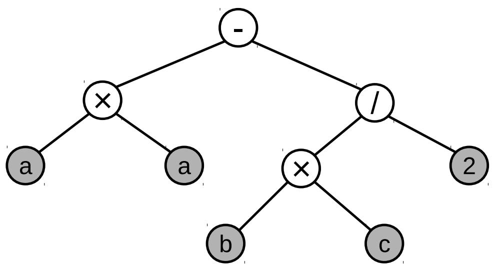


Figure 2.4: Example of tree representation of a GP individual.

same idea. First, the index is used to obtain a small set of candidates by scanning the lists whose indices are closer to the query. Then, this candidates list is re-ranked by computing the approximate Euclidean distance to the query, according to the encoding technique chosen.

Johnson *et al.* [35] proposed a data structure with different idea than inverted indices, that also employed quantization-based compressed vectors to deal with memory restrictions. First, authors created an IVFADC index as an auxiliary data structure to help in the construction of an NN graph. Then, for all vertices, to determine their k-neighbors, a k-NN search was performed over the IVFADC index. To the best of our knowledge, it is the only technique in the literature based on the NN graph idea that scaled up to billion-size datasets. However, the NN graphs created by this method were not evaluated on the NN search task. Also, no comparisons with other indexing structures were provided. Authors only evaluated how accurate they approximated their graph to an exact NN graph. Although the reported time for NN graph construction is reasonable, they employed a costly multi GPU architecture for performing experiments.

2.4 Genetic Programming

Genetic Programming (GP), introduced first by Koza [38], is a problem solving technique inspired in the biological inheritance and evolution. Each potential solution is called *individual*, and is commonly represented by a tree. Figure 2.4 presents a tree representation of the mathematical function $f(a, b, c) = a \times a - ((b \times c)/2)$. The whole GP process to discovery near-optimal solutions is detailed in the following steps:

1. **Create initial population**: initially, a population of fixed size is created, where each tree (individual) is defined randomly. A tree is composed of functions (white nodes in Figure 2.4) and terminals (gray nodes). Functions are the internal nodes, employed to combine the terminals, usually mathematical functions. Terminals are the leaf nodes, employed as inputs of individuals.

2. **Evolve population**: through a number of generations, individuals are evolved by employing genetic operators. The following steps are repeated at each generation.

   2.1. **Compute fitness of individuals**: at the beginning of every generation, the fitness value is computed for all individuals. The fitness value is an indicator

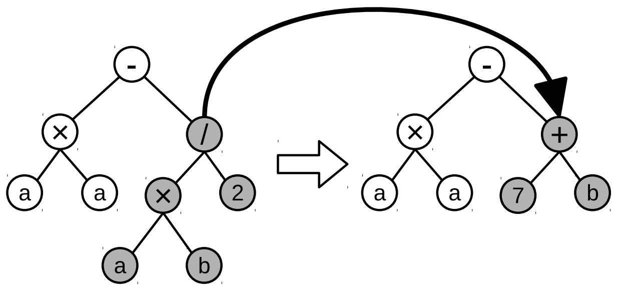


Figure 2.5: Example of the mutation operator.

of how well an individual performs in a given task. Therefore, this function is application dependent.

2.2. **Select individuals for genetic operators**: an important factor towards the convergence of the evolution process is the technique employed for selecting individuals that will be used for genetic operators. There are many techniques proposed in the literature [33]. Usually, these approaches are based on the fitness value of individuals or on their relative order in the whole population (rank-based).

2.3. **Apply genetic operators**: genetic operators are applied on selected individuals to create individuals for the next generation. The following operators are commonly used:

   i. **Reproduction**: this operator takes a percentage of the individuals with best fitness values and copies them directly to the next generation. This operator guaranties that the best individual of a generation will be at least as good as best individual of the previous generation.

   ii. **Mutation**: this operator is applied on a individual by taking a randomly selected subtree and replacing it with a randomly generated tree. Figure 2.5 illustrates this operator. The purpose of mutation is to add a minimum of diversity to the population.

   iii. **Crossover**: this operator takes two individuals and exchanges two randomly selected subtrees. An example is shown in Figure 2.6. The objective of crossover is to create new diverse individuals by exchanging genetic information from parents.

2.4. **Replace old population**: the population of the next generation is composed of individuals created in step 2.3, then, the evolutionary process returns to step 2.1.

3. **Select best individuals**: the fitness of individuals of the last generation is computed, and, then, the best individuals are returned.

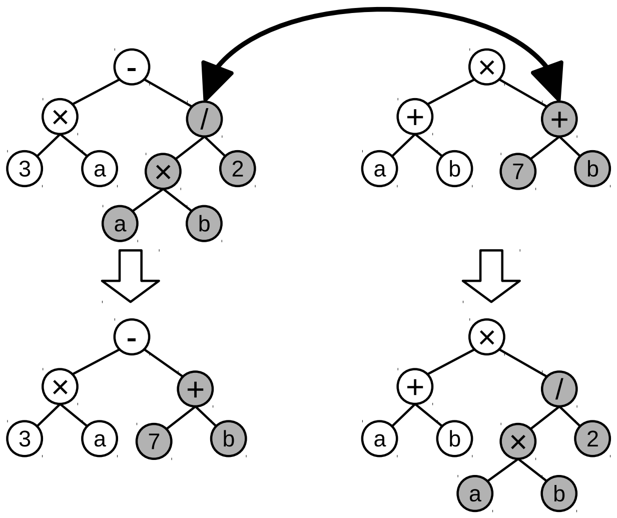


Figure 2.6: Example of the crossover operator.

Chapter 3

Hierarchical Clustering-Based Nearest Neighbor Graphs

This chapter introduces the proposed framework for ANNS, the **H**ierarchical **C**lustering-based **N**earest **N**eighbor **G**raphs, called HCNNG from now on, which constructs efficiently NN graphs based on the clusters information obtained after performing multiple clustering procedures (Section 3.1). Also, as a part of this framework, we introduce two novel techniques that employ classic KD-Tree-like as auxiliary data structures to help on two different stages of NN graph search: the selection of starting vertex (Section 3.2.1), and the pruning of unnecessary vertices at each step of navigation (Section 3.2.2). Later in this chapter, we describe the experiments conduced to validate the proposed HCNNG framework on million-size datasets (Section 3.3).

3.1 Graph Construction

The graph construction process relies on three steps: the execution of multiple clustering procedures, the creation of sub-graphs to connect the vectors contained on the clusters, and finally, the fusion of the sub-graphs to compose a global graph. The objective is to generate a graph with a good connectivity among data points. Figure 3.1 illustrates the proposed framework for NN graph construction. We need to create enough connections among the vertices for supporting effective searches, but, at the same time, without unnecessary multiple paths, which may lead to inefficient processing time. One important aspect that we considered at the moment of selecting the clustering technique to be employed, was the time complexity, since it need to scale to very large collections. Following this reasoning, we selected the hierarchical clustering, which has a $O(N \log N)$ bounded time complexity. Other clustering approaches may generate clusters that fit better to the data distribution (quality of clusters), but since we only are interested in knowing which points are close enough to later create edges between them, we can disregard this aspect.

The hierarchical clustering performed over a set of points defines an implicit relationship of proximity between the points in each cluster. We explore this idea to create an NN graph. Algorithm 2 describes the hierarchical clustering procedure, returning the set of edges created over the points. Given a set of points $\mathcal{P}$, a distance function d, and a

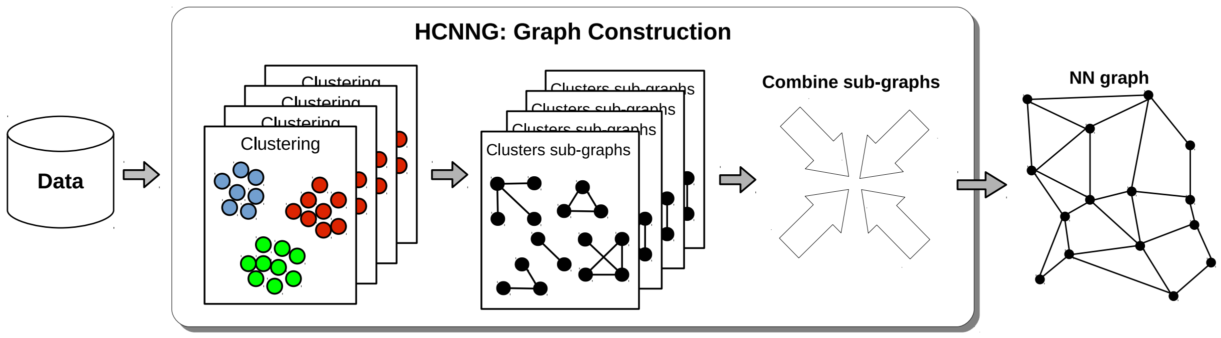


Figure 3.1: HCNNG framework for NN graph construction.

minimum size of clusters n, if the size of $\mathcal{P}$ is less than n, then, points contained in current $\mathcal{P}$ are connected by means of the function $CreateSubGraph$. Otherwise, two points are selected randomly (lines 7 and 8) and $\mathcal{P}$ is divided into two subsets ($\mathcal{P}_1$ and $\mathcal{P}_2$) based on the proximity to the points randomly selected. Then, the hierarchical clustering procedure is applied recursively over both subsets (lines 9 and 10) to, finally, join the edges created on each one of the partitions.

There are many ways to connect points inside of clusters (function $CreateSubGraph$). We explored four common structures employed in the graph theory field: a Complete graph, Stars, Hamiltonian paths, and Minimum Spanning Trees (MST). Figure 3.2 illustrates these structures using the same 2-dimensional points (assuming that those points are in the same cluster). The use of complete graphs leads to the construction of dense graphs, where every vertex ended with high degree. Stars, in the other hand, led to the constitution of many hubs (vertices with high number of connections) in the resulting graph, which caused poor search results when the graph navigation reached one or more of them. Hamiltonian paths presented a clear improvement in the final degree of vertices, reducing the mean value and the standard deviation. Finally, the use of MST presented a slightly greater degree of vertices than Hamiltonian paths, but the search results obtained on these graphs were the best among all explored structures. Since MST algorithms do not limit the maximum degree of vertices, they could produce vertices with high degree, specially in high dimensional data. Therefore, we limited the maximum degree of vertices in MST algorithm to 3 (called $MST3$ from now on). The maximum degree of 3 was determined experimentally, we tested with values ranging from 2 to 10, and observed that search results do not change significantly for values greater than 3, thus, to guarantee lower vertices degree we used 3 for all the experiments presented in Section 3.3.

By applying the clustering procedure just once, the resulting graph will be highly disconnected. This is better illustrated in Figure 3.3 ((a), (b), and (c)), using a synthetic 2-Dimensional dataset of points. Since $MST3$ just connects points inside the same cluster, there is no way to reach points from other clusters. Also, results of different clustering executions (using different seeds for the random number generator) showed that edges between nearby points that were not found in a given execution could be discovered by another execution. We address the problem of disconnected graphs by performing multiple random clustering procedures and combining their resulting graphs. Figure 3.3d shows an example of the combined graph obtained from the fusion of 15 disconnected graphs

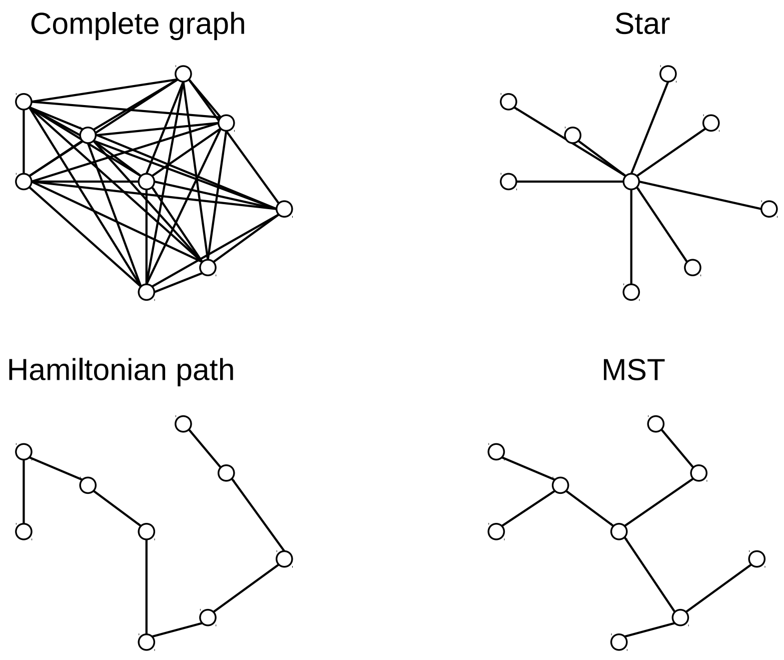


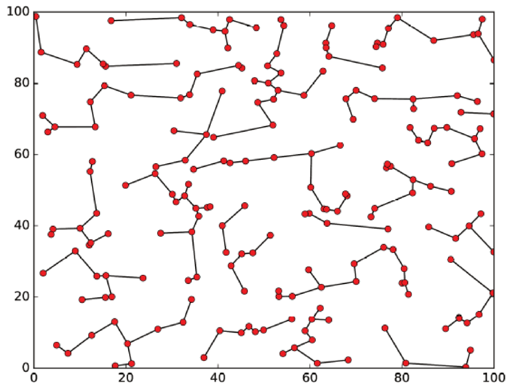
Figure 3.2: Graph structures for connecting vectors in clusters.

```
Function $HierarchicalClustering(\mathcal{P}, n)$
    Data: data points $\mathcal{P}$, min size of clusters $n$
    Result: graph edges $E$
    $E \leftarrow \phi$
    if $|\mathcal{P}| < n$ then
        $E \leftarrow CreateSubGraph(\mathcal{P})$
    else
        select randomly $x_1$ and $x_2$ points from $\mathcal{P}$
        $\mathcal{P}_1 \leftarrow \{x \in \mathcal{P} \mid d(x, x_1) < d(x, x_2)\}$
        $\mathcal{P}_2 \leftarrow \{x \in \mathcal{P} \mid d(x, x_1) \geq d(x, x_2)\}$
        $E_1 \leftarrow HierarchicalClustering(\mathcal{P}_1, n)$
        $E_2 \leftarrow HierarchicalClustering(\mathcal{P}_2, n)$
        $E \leftarrow E_1 \cup E_2$
    return $E$
```

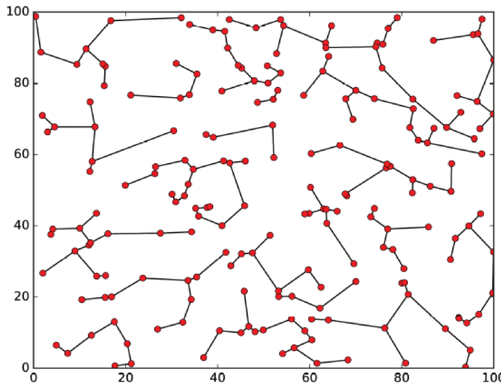
Algorithm 2: Hierarchical clustering procedure.

(obtained from 15 different clustering executions), where, it can be observed that the final graph contains paths from one point to any other. This also solves the problem associated with points located on the border of two or more clusters, i.e., these points may belong to different clusters despite being close in the feature space. The process to create the final proximity graph is outlined in Algorithm 3.

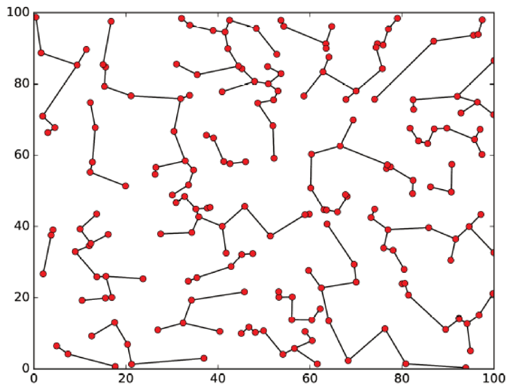
More formally, the graph fusion procedure can be defined as follows. Let $\mathcal{C} = \{c_1, c_2, \ldots, c_S\}$ be a set with S clusters defined after performing multiple hierarchical clustering procedures. For each cluster $c_i \in \mathcal{C}$, we can create a proximity graph $G_i = (V_i, E_i)$, where V_i is a set of vertices defined by the points belonging to c_i, and E_i is the set of edges generated by $MST3(V_i)$ for a given distance function $d : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^+$. The fusion



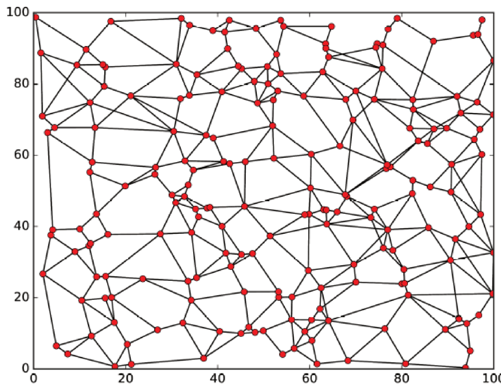
(a) 1st execution



(b) 2nd execution



(c) 3rd execution



(d) Combined graph from 15 executions

Figure 3.3: Examples of graphs obtained after different clustering executions. (a), (b), and (c) from isolated executions. (d) from combination of multiple executions.

```
1  Function CreateHCNNG($\mathcal{P}$, $m$, $n$)
       Data: data points $\mathcal{P}$, number of random clusterings $m$, min size of clusters $n$
       Result: graph edges $E$
2      $E \leftarrow \phi$
3      for $i \leftarrow 1$ to $m$ do
4          $E_i \leftarrow HierarchicalClustering(\mathcal{P}, n)$
5          $E \leftarrow E \cup E_i$
6      return $E$
```

Algorithm 3: Fusion of graphs.

graph G' is defined as $G' = (V', E')$, such that $V' = \cup_{i=1}^{n} V_i$ and $E' = \cup_{i=1}^{n} E_i$.

Another reason to choose this hierarchical-clustering-based approach instead of other methods like k-means, because it is hard to estimate the initial number of clusters and the number of iterations, without previous knowledge of the data distribution. The expected time complexity of each clustering execution scales in a logarithmic factor to the collection size leading to a total of $O(ND\log(N/n))$, plus $O(NDn\log(n))$ to construct $MST3$'s for each cluster, where N is the size of the collection, n the estimated size of clusters, and D

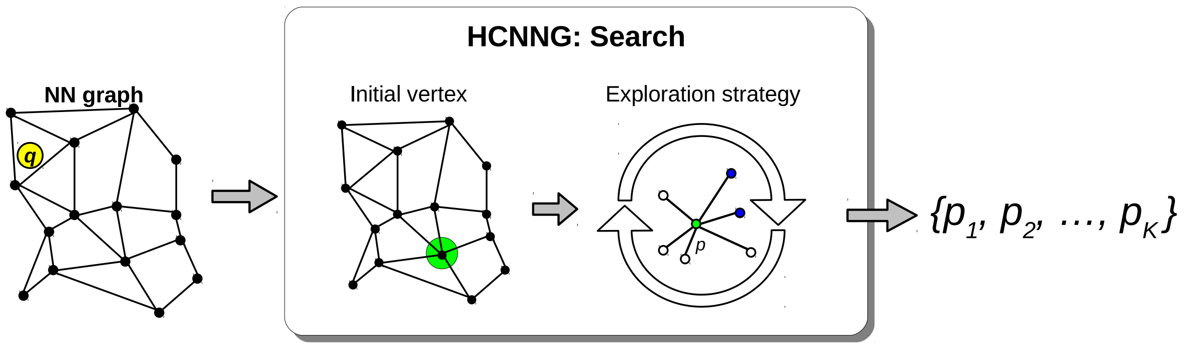


Figure 3.4: HCNNG framework for search on NN graphs.

represents the cost of each distance calculation (equal to the data dimensionality for the Euclidean distance). Therefore, for h clustering executions the expected complexity of graph construction is $O(h \times (ND\log(N/n) + NDn\log(n)))$. With respect to the vertices degree, as we limited the vertex degree to 3 in each cluster, all vertices will have no more than $3 \times h$ neighbors in final graph, then the final number of edges complexity will be $O(Nh)$.

3.2 Search on Graphs

As described in Section 2.2.2, the search strategies for NN graphs proposed in the literature follow the general idea of Algorithm 1. We also employed a similar search strategy over the graph constructed by the proposed algorithm described in the previous section, but differently from this search strategy, we propose two heuristics to accelerate the discovering o the true NN: one for the selection of the initial vertex, and the other for improving the navigation on the graph by pruning some edges at search. These heuristics, that are part of our HCNNG framework, are illustrated in Figure 3.4 and are detailed in the next sections.

3.2.1 Non-randomic Selection for Starting Vertex

A good non-randomized starting vertex selection for search could help to reduce the number of distance calculations. We experimented creating a KD-Tree over all the collection and taking as starting point the leaf that contains the query. This idea was explored in a previous work [3]. Also, we do not actually perform the search on the KD-Tree, we just perform a sequence of comparison of values from the root to the leaf that contains the query, which is limited by the height of the KD-Tree (scales logarithmically with dataset size), without spending any distance calculation in determining the initial point. Figure 3.5 shows an example for initial vertex selection based on a KD-Tree using a synthetic 2-dimensional dataset. The paths traversed in the KD-Tree for queries q_1, q_2 and q_3 (performing just one comparison per level) are $L_1 \rightarrow L_2 \rightarrow L_5 \rightarrow p_5$, $L_1 \rightarrow L_3 \rightarrow L_6 \rightarrow p_8$ and $L_1 \rightarrow L_3 \rightarrow L_7 \rightarrow p_{10}$, respectively. Similar to the idea of picking multiple random starting points, we decided to create multiple global KD-Tree's and pick the closest point to

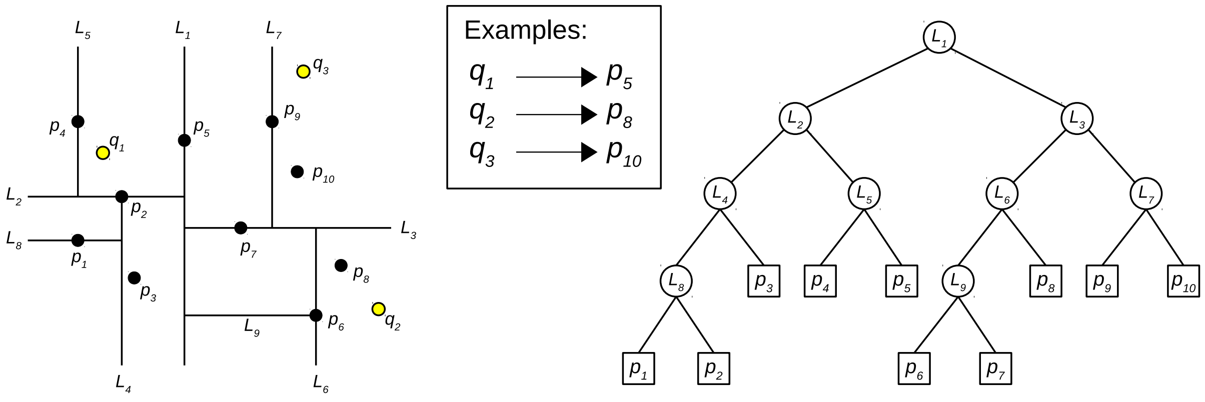


Figure 3.5: Starting vertex selection example based on a KD-Tree.

the query among the ones returned by the KD-Tree's. This value was set experimentally and is detailed in the next section. At this extra step, we just spend as many distance calculations as the number of KD-Tree's created (to determine which one is closer). A similar idea was explored by Wang and Li [68], to select initial candidates based on multiple KD-Tree's for search on NN graphs, but differently from our approach, they actually perform the search on KD-Tree's (which involves extra costs), and also employed them again in the traversal graph process. Finally, the creation of these KD-Tree's can be done offline, along with the creation of the NN graph, thus, the unique extra cost at search time for the proposed approach is the simple traversal on the KD-Tree's, which is negligible, as we detailed above.

3.2.2 Pruning Edges to Accelerate Searches

As detailed in Algorithm 1, for searching in NN graphs, when a new vertex is taken from the queue, all its neighbours are fully explored. This carries the cost of computing the distance from all neighbors (of current vertex) to the query. When the path followed by the search algorithm reach some hubs, this fully exploration strategy will increase significantly the database scan rate to find the nearest neighbors. We propose a simple heuristic, called *guided search* from now on, to avoid exhaustively computing the distance from the query to all neighbors in each step of the search by pruning some edges. Our objective is to focus the graph traversal process on a reduced set of neighbors that are probably in the *direction* to the query. Figure 3.6 illustrates our idea. At any point in the navigation (say p), we will just compute the distances from the query (q) to those neighbors (v_1 and v_2) located at the same quadrant as the query (quadrant highlighted in the figure), in reference to the actual vertex.

Figure 3.7 shows an example of multiple steps of a guided search from the starting point (green vertex) to the query point (yellow star), where the vertices explored by guided search are colored in blue and green (total 5), and vertices explored by classical greedy approach are those colored in gray in addition to the blue and green ones (total 10). The guided search starts at vertex 11 and, in the first step, it is computed the distance from the query to just vertices 6 and 8. Vertex 8 is then selected for the next step. In the

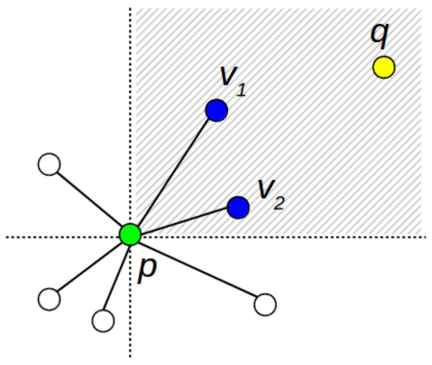


Figure 3.6: Example of directed selection of neighbors.

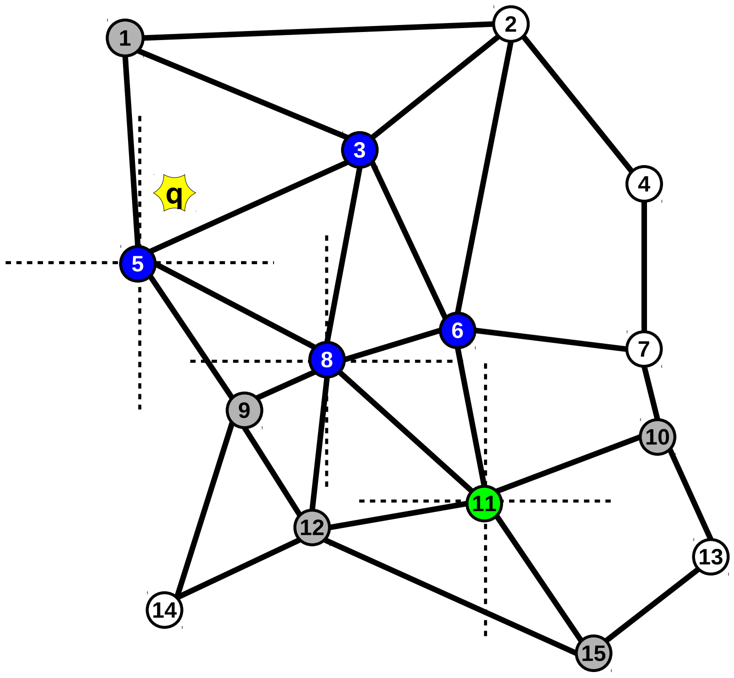


Figure 3.7: Example of multiple steps in the guided search.

second step, it is explored vertex 5, which is selected for the next step, as it is closer to the query than vertex 8. Finally, vertex 3 is explored, and as it is not closer to the query than the actual vertex, the search ends. As we can observe, the number of distance computations is decreased in half compared to classical greedy approach, in this example.

Determining the neighbors located in the same quadrant, at query time, is almost as expensive as computing all the distances, so we propose a preprocessing stage, to be applied on each vertex neighborhood N_p, with the objective of organizing the neighbors of each vertex p in quadrants (a subspace), taking as origin the vertex itself. Our idea is inspired in the space partition procedure used by KD-Trees. We also used a tree-like structure to store the neighbors in each subspace, where each internal node keeps the dimension used to partition the space, and two sub-trees to store the neighbors located in the positive and negative subspaces defined by the dimension chosen and an origin of reference (the vertex itself p). Leaf nodes contain the neighbors located in the subspace

```
1  Function DivideSubspace(p, $N_p$)
       Data: point of graph $p$, neighbors points $N_p$
       Result: space partition tree $T$
2      $dim \leftarrow$ select dimension based on $N_p$
3      $neg \leftarrow \{v \in N_p | v[dim] < p[dim]\}$
4      $pos \leftarrow \{v \in N_p | v[dim] \geq p[dim]\}$
5      if $|neg| = 0$ or $|pos| = 0$ then
6          $T.points \leftarrow N_p$
7      else
8          $T.dim \leftarrow dim$
9          $T.neg \leftarrow DivideSubspace(p, neg)$
10         $T.pos \leftarrow DivideSubspace(p, pos)$
11     return $T$
```

Algorithm 4: Creation of a tree structure to search in subspaces.

determined by the successive space partitions performed since the root up to the leaf. This process is outlined in Algorithm 4.

For each call to the *DivideSubspace* function, first a new dimension (dim) is selected to split the subspace. To maintain the tree balanced, we select the dimension that keeps the best balance between the number of neighbors in the negative and positive side (line 2). Then, we divide the neighbors into two subsets (*neg* and *pos*), using the dimension chosen. If one of these sets ends empty after the subspace division (line 5), a leaf is created to store the actual set of neighbors (N_p) (line 6). Otherwise, an internal node is created to store the dimension used in the division, and then, recursively, two sub-trees ($T.neg$ and $T.pos$) are created (lines 9 and 10) to continue the division of the two subsets defined by neighbors *neg* and *pos*. A naïve algorithm for dimension selection costs $D \times |N_p|$, being D the data dimensionality and $|N_p|$ the vertex degree. As the height of tree scales logarithmically to the vertex degree, and the vertex degree scales linearly to the number of clustering executions (h), then the complexity of Algorithm 4 is $O(Dh\ log(h))$ for each vertex of the graph.

As previously mentioned, the subspace division is applied to every vertex and its neighborhood in a preprocessing phase, along with the NN graph creation. At query time, it is enough to traverse from the root towards the leaf that contains the neighbors located at the same subspace as the query. Algorithm 5 details this traversal procedure. We start at the tree root. If, at some time, we find a leaf, we return the neighbors contained in this subspace (line 4). Otherwise, we check whether the query point belongs to the negative (line 7) or positive (line 9) subspace defined by the dimension stored, and then continue the traversal process following in the direction of the adequate subspace.

The creation of subspace trees is performed in such a way that the resulting tree is balanced and, therefore, the expected height of trees is proportional to the logarithm of the number of neighbors (for a certain vertex). Note, also, that determining the points at the same query's subspace takes as many comparisons (of integers) as the tree height, and, the resulting graph of nearest neighbors remains sparse $O(h)$, as discussed in the previous section. Therefore, the application of the function *GetNeighbors* at each step of

```
1  Function GetNeighbors(q, p, node)
       Data: query point $q$, point of graph $p$, node of space partition tree $node$
       Result: neighbors in query's subspace $N_q$
2      $N_q \leftarrow \phi$
3      if node is a tree leaf then
4          $N_q \leftarrow node.points$
5      else
6          if $q[node.dim] < p[node.dim]$ then
7              $N_q \leftarrow GetNeighbors(q, p, node.neg)$
8          else
9              $N_q \leftarrow GetNeighbors(q, p, node.pos)$
10     return $N_q$
```

Algorithm 5: Identification of neighbors at the same subspace of the query.

the navigation on the graph has a negligible cost.

Finally, for our experiments we employed Algorithm 1 (described in Section 2.2.2) with a few modifications. We start the search at the multiple KD-Tree based vertex selection (line 2), and, in each step, instead of exploring all neighbors, we continue the search through the neighbors returned by the *GetNeighbors* function (line 7). In any step of the search, when it had not found a closer point to the query in the selected subspace, adjacent subspaces are explored, i.e., closer leaves in the subspace partition tree are traversed. Subspaces are explored until a neighbor closer to the query than current vertex is found. Finally, if there is no any neighbor closer to the query in any quadrant, a backtracking strategy is performed using a priority queue similar to Algorithm 1.

3.3 Experiments

This section discusses the performed experiments aimed to validate the proposed method. All experiments were conduced using the Euclidean distance.

3.3.1 Datasets

We experimented with two databases from BIGANN datasets[1] for approximate nearest neighbor search. One collection is composed of 1 million SIFT feature vectors (128 dimensions) to index construction, and 10 thousand of queries to evaluate performance. The other collection contains 1 million GIST feature vectors (960 dimensions) and 1 thousand queries. These datasets were used in previous works [27, 34, 55] to evaluate approximate nearest neighbors search techniques.

We also tested with a collection of Global Vectors [56] of textual features (GloVe, 100 dimensions) extracted from 2 billion of tweets[2]. In this dataset, to maintain a similar size than in previous datasets, we used a randomly selected subset of 1 million vectors to index construction and 10 thousand queries. For all datasets considered, the groundtruth

[1]BIGANN: `http://corpus-texmex.irisa.fr/` (As of January 2020).

[2]GloVe: `https://nlp.stanford.edu/projects/glove/` (As of January 2020).

was pre-computed by performing linear search to find the real nearest neighbors on the whole dataset for each query of the test set.

3.3.2 Evaluation Criteria

We employed a widely-used evaluation metric for ANNS methods, the Speedup × Recall charts. To keep the speedup independent on architecture where experiments were executed, we only consider the number of distance calculations performed by each method. Thus, speedup is defined as:

$$Speedup = \frac{Collection\ size}{Number\ of\ distance\ calculations}$$

The Recall is defined by the fraction of true nearest neighbors that are successfully retrieved. More formally, we used the following expression to evaluate the *recall@K* of an ANNS method for a given query q in the K-NN search task:

$$Recall@K(q) = \frac{GT(q, K) \cap AM(q, K, T)}{K}$$

where the function GT returns the real K-NN for the query q, and AM returns the K-NN discovered by the approximated method being evaluated, limiting to T the maximum number of points explored (or distance computations) in the collection. In the rest of this thesis we report the overall performance of some approximate method as the average recall on a test set of queries. We conduced three experiments to evaluate the methods' performance: the first to search for the closest neighbor (1-NN), the second to search the 10 nearest neighbors (10-NN), and the last to search the 100 nearest neighbors (100-NN).

3.3.3 Parameter Tuning

Two parameters are used in the HCNNG construction algorithm: the number of executions of hierarchical clustering procedures h, and the minimum size of clusters n. We performed an exhaustive parameter search in a synthetic dataset of 100 thousand uniformly sampled random vectors with a dimensionality of 20. We perform this parameter search in this random data, since it is the hardest scenario, when there is no correlation between dimensions. Figure 3.8 shows the heat map obtained by evaluating at Recall@10 search results on graphs created for different values assigned to parameters h and n. To evaluate the recall of each configuration, we fixed the maximum number of distance calculation allowed in search (called as T from now on) to 1000. We can observe in the figure that the convergence of the recall scores is reached for $h \geq 20$ and $n \geq 1000$. We set the number of clustering procedure executions to 20 for all experimental results presented in the following sections. Even though we can get same recall values with a larger h and lower n, it is less costly to choose a lower number of clustering executions. In general, most of our experiments across different datasets suggested an optimal value of $\sqrt{N}$ for the minimum size of clusters, being N the size of the collection, so we use this value in all our experiments.

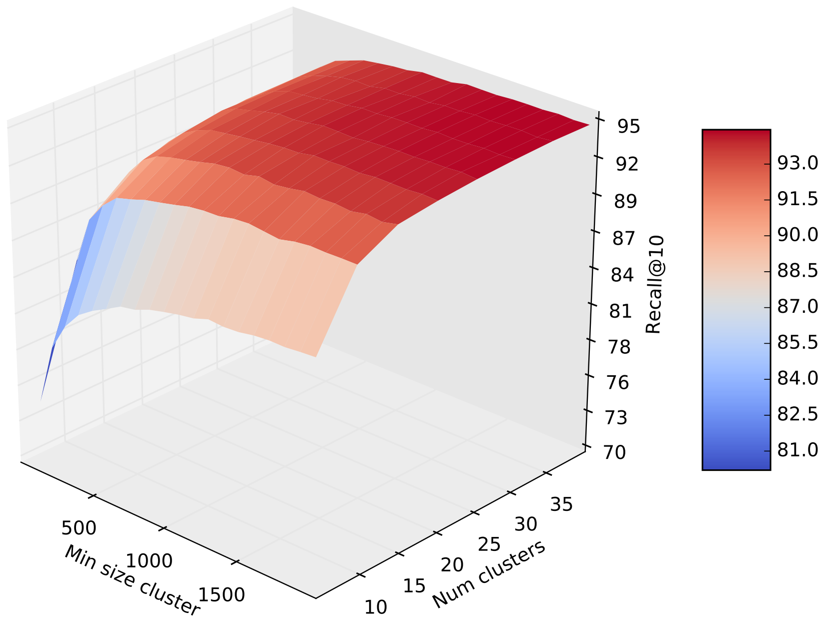


Figure 3.8: Recall@10 for different values of clusters' minimum size and clustering executions.

At search time, to determine the starting vertex, we experimented creating different number of KD-Tree's, from 1 to 20. To evaluate the impact of this parameter, we performed the search (using the proposed guided strategy) on the graph created over SIFT dataset with the parameters suggested above, and measured the Recall@1. Also, we ran different experiments limiting to $\{100, 200, 300, 500, 1000\}$ the value of T. Results for this experiments are shown in Figure 3.9. As it can be observed, for lower values of T (greater speedups), there is a significant gain in recall when the number of KD-Tree's increases. Based on these results, we selected the value of 10 (number of KD-Tree's) for the next experiments, as we wanted our method to yield results with high values of recall for any desired speedup.

Figure 3.10 shows results of proposed framework on SIFT dataset using different configurations for search. The same behavior was observed for the other datasets. It can be observed that the two proposed approaches (initial vertex selection based on multiple KD-Tree's, and guided search) obtain significant gains over a classical approach to search (random initial vertex selection with fully neighbors exploration), and when combined (KD-Tree + Guided), improve even more the results.

3.3.4 Scaling Datasets

We conduced an experiment to measure the average scan rate performed (proportion of the elements explored in some dataset) by 10-NN search (find 10 nearest neighbors) to achieve a 99% of recall in all datasets. We measure this for different dataset sizes and results are presented in Figure 3.11, with both axes in logarithmic scale. The shapes of the curves suggest that the search of proposed method scales polylogarithmicly to the dataset

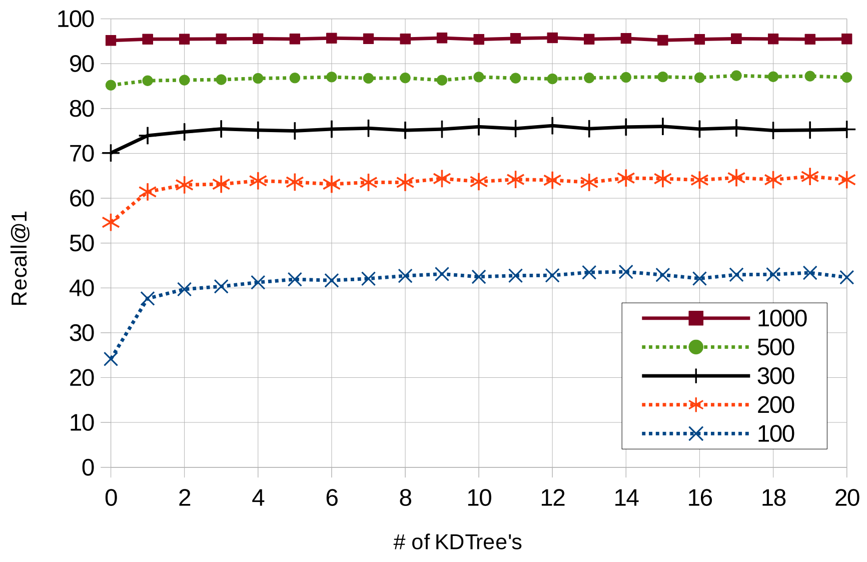


Figure 3.9: Recall@1 for different numbers of KD-Tree's and maximum number of distance calculations (SIFT dataset).

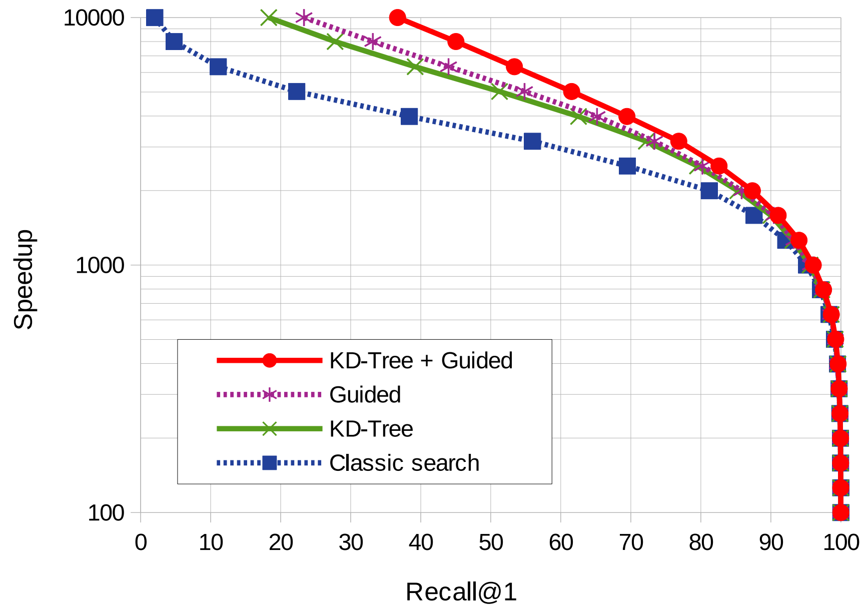


Figure 3.10: Different configurations of search (starting vertex + exploration strategy), on SIFT dataset.

size, independently of the dataset. The proposed search method shows to be very efficient in SIFT dataset by just needing to scan less than 0.4% of all dataset at size 1 million. The slope of GloVe's curve shows a slower decreasing of scan rate than in the other collections. This seems contradictory to the fact that dimensionality in GloVe vectors is far smaller than GIST vectors and slightly different from SIFT vectors. However an analysis of the intrinsic dimensionality of the datasets showed that GloVe has the highest

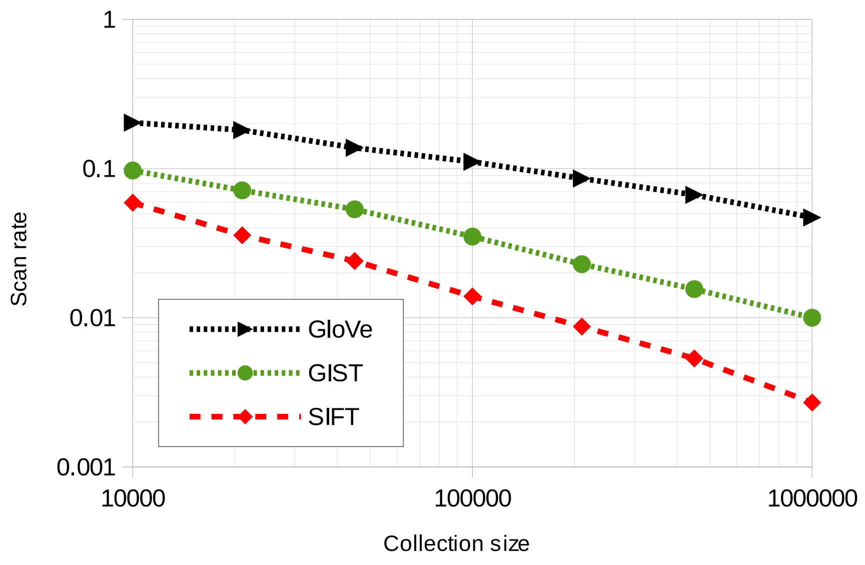


Figure 3.11: Scan rate needed to reach 99% of recall for different collection sizes.

Base	GloVe	GIST	SIFT
Dim	15	9	10

Table 3.1: Intrinsic dimensionality of datasets, estimated through Hausdorff dimension.

value of all 3 datasets. To estimate the intrinsic dimensionality we used a method based on the analysis of fractal dimensions, known as *Hausdorff dimension* [28]. These values are shown in Table 3.1.

3.3.5 Literature Comparison

We compared the proposed approach, henceforth called HCNNG, with several well-known and recent state-of-the-art methods. FLANN library [49] is probably the most known library for approximate nearest neighbors search. There are three main space partitioning trees techniques implemented in FLANN: Randomized KD-Trees [61], K-Means Tree [47], and Hierarchical Clustering Tree [48]. We used the auto-tuned algorithm from FLANN library in our experiments, which selects the best algorithm (included in FLANN) and parameter values for each of the data.

Among the NN-graph-based techniques, we compared our HCNNG with a recently proposed method [27], called FANNG. We used our own implementation, and run the experiments with parameters reported by authors on the same datasets. Other techniques included from this family of methods were SW-graph [45] and HNSW [46]. For both, the experiments were conducted with the implementation found in a recent library called *Non-Metric Space Library* (NMSLIB)[3]. Also, we run the KGraph [22] technique, using

[3]NMSLIB: `https://github.com/searchivarius/nmslib` (As of January 2020).

the implementation provided by authors in their website[4]. Finally, we implemented the method introduced by Wang *et al.* [67] and included it as baseline (denoted as *Wang2012*).

To compare with methods based in Product Quantization, we use a implementation available of the classic method and two more recent optimized versions[5]: Product Quantization (PQ) [34], Optimized Product Quantization (OPQ) [23], and Additive Quantization (AQ) [6].

3.3.6 Experimental results

Search performance results across SIFT, GIST, and GloVe are shown in Figure 3.12. Results for 1-NN, 10-NN, and 100-NN searches are presented in the first, second, and third rows, respectively.

In the case of SW-graph technique, the graph is constructed with the objective of allowing a logarithmic navigation, but this property is negatively affected by the high degree of many vertices, leading to a slow convergence, which is reflected in the results of all experiments. As KGraph and Wang2012 aims to approximate an exact KNN graph, thus its performance should be similar to an exact KNN graph. In all datasets, we added an extra baseline, with the name of *kNN-Graph*, that represents the search performed over an exact KNN graph. As it can be observed, empirical results suggest a considerable margin of difference of kNN-Graph against methods that optimize graphs for NN search (e.g., HCNNG, FANNG, and HNSW). Consequently, KGraph and Wang2012 have a similar behavior. Is worth it to stress out that, although, Wang2012 [67] employs a similar strategy to construct the NN graph, the selection of $MST3$ for connecting the points inside clusters, in our proposed approach, has a great impact on search results, demonstrating a good balancing between graph sparsity and an easy to traverse NN graph.

On SIFT collection, although FLANN starts with one of the highest recall scores (at Speedup of 10^4), it evolves slowly, being quickly outperformed by the proposed method, HCNNG, and most of baselines, for all the three cut-off points considered (1, 10, and 100). In this dataset, HNSW shows to be the most competitive baseline, and FANNG, the second best. Even though FANNG starts with low recall values, it quickly evolves to high scores ($> 90\%$). From the quantization-based techniques, the best method, AQ, shows to be competitive at high recall values at 1-NN search, but it is outperformed by a significant margin by HCNNG for 10-NN and 100-NN searches. In this dataset, HCNNG yielded the best results both in terms of effectiveness and efficiency: it achieves high recall values quickly and yields good recall scores when the maximum number of distance calculation is low (higher speedups).

On the GIST dataset, the performance of all methods is clearly affected by its higher data dimensionality, but, similarly to the SIFT dataset, the relative performance of all methods does not change. Again, HNSW shows to be competitive, at the three cut-off points, but for high recall values when compared with FANNG. For high recall values, FANNG presents similar results as the proposed HCNNG. AQ method shows good performance at first (1-NN search), but it is notably more affected than HCNNG when K

[4]KGraph: `http://www.kgraph.org` (As of January 2020).
[5]Quantizations: `https://github.com/arbabenko/Quantizations` (As of January 2020).

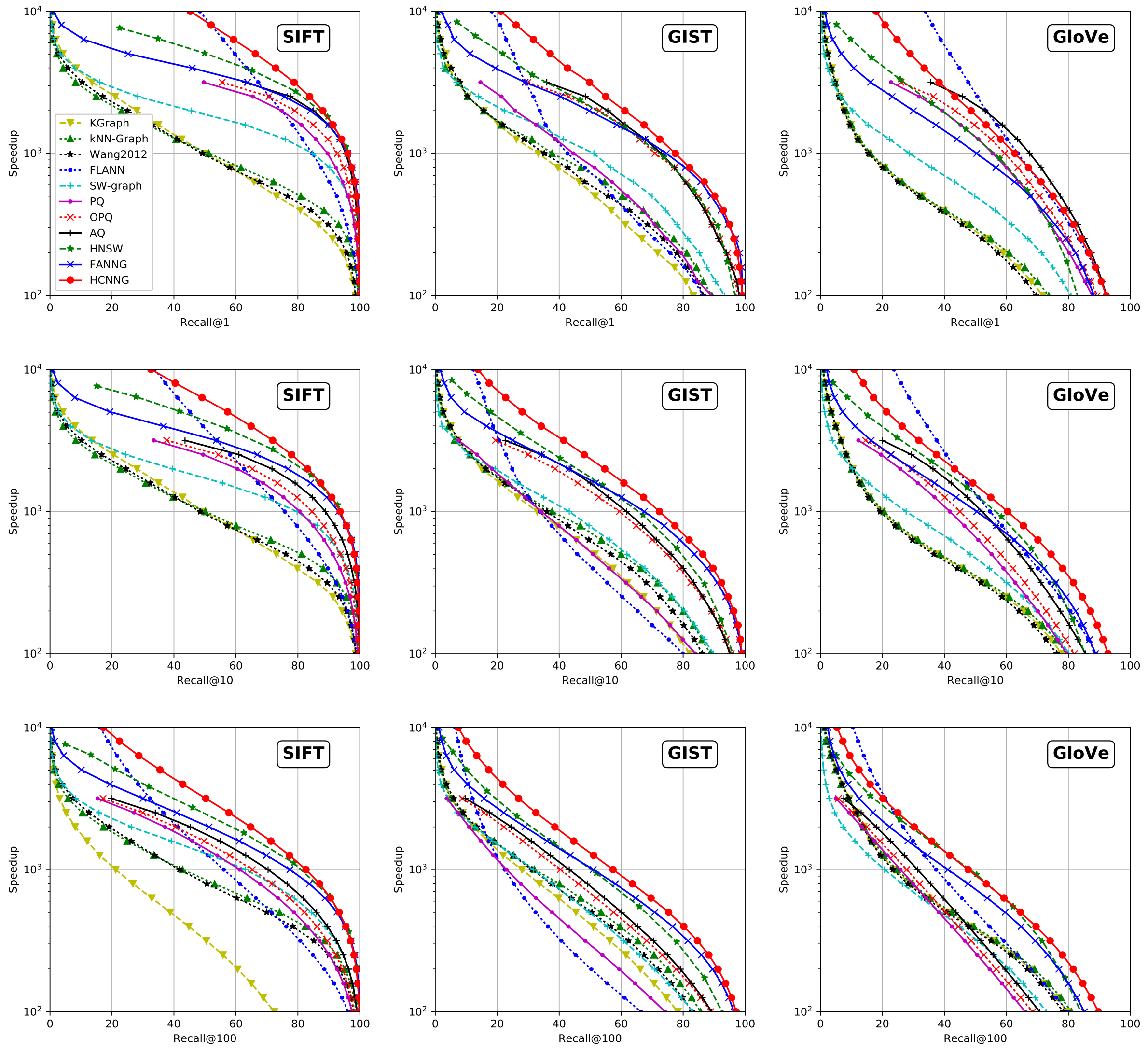


Figure 3.12: Speedup vs recall on SIFT, GIST, and GloVe datasets.

(number of NN to search) increases. As in case of SIFT dataset, the curve of the proposed HCNNG was superior at almost any point for 1-NN, 10-NN, and 100-NN searches on the GIST dataset.

Results obtained by all evaluated methods on GloVe (for all 1-NN, 10-NN, and 100-NN searches) were significantly worse than in SIFT and GIST datasets, considering that they have the same size and GloVe has a lower dimensionality than SIFT and GIST vectors. This fact is explained by the intrinsic dimensionality of datasets. As seen in Section 3.3.4, GloVe has the highest intrinsic dimensionality, therefore evolution to get high recall values will be in fact slower. Although, FLANN and AQ yielded better results than HCNNG at medium recall values for 1-NN search, their performance is affected dramatically when the number of NN to search increases, as it can be observed in 10-NN and 100-NN search results for this dataset.

Summarizing, HNSW and FANNG have shown to be the most competitive methods in all datasets, obtaining comparable speedups to the proposed HCNNG at high recall values, but they are outperformed at medium recall values by HCNNG. The FLANN's

Table 3.2: Total time for 100K queries (in seconds) using classical search approach vs the guided proposed (without compiler optimizations).

	SIFT			GIST			GloVe		
Max. Distance Calculations	Query time (seconds)		Gain (recall)	Query time (seconds)		Gain (recall)	Query time (seconds)		Gain (recall)
	Classic	Guided		Classic	Guided		Classic	Guided	
100	30.412	37.141	21.37%	87.118	91.605	5.60%	26.438	29.346	6.14%
250	78.675	85.869	27.39%	236.031	227.334	20.92%	68.176	72.196	14.27%
500	161.466	171.058	5.16%	449.054	445.305	11.20%	141.542	145.689	10.56%
750	247.852	260.133	1.77%	679.243	687.949	5.22%	218.889	226.076	6.44%
1000	338.050	348.214	0.76%	910.847	921.767	2.70%	294.816	305.367	3.00%

auto-tuned algorithm starts with good recall values, but, in most experiments, speedup tends to decrease quickly as the recall increases. In the case of AQ, although it has some competitive results at 1-NN search, the method performance is very sensitive to the number of NN to search. The proposed method shows the best behavior across all datasets, obtaining both good speedups at medium recall and best speedups at high recall values in the three types of searches performed, demonstrating its applicability in cases where not only the closest neighbor is required (e.g., local image features matching) but also when a set of them are requested (e.g., content-based image retrieval).

3.3.7 Overhead of Auxiliary Local KD-Trees

One important aspect to consider in the proposed HCNNG framework, is the overhead of auxiliary local KD-Trees-like structures employed to improve the search performance of HCNNG, since the use of them is very recurrent at search. Thus, we measured the real overhead of using local KD-Trees in our guided approach, by performing 100 thousand 1-NN searches on each dataset and measured the total time to process these queries using the classical search (Algorithm 1) and the proposed guided approach, limiting both methods to different maximum numbers of distance calculations (number of vertices explored). The resulting times are shown in Table 3.2, which were obtained using a single thread having both methods a g++ implementation. As it can be observed, the use of local KD-Trees in our guided approach add an almost constant extra constant time of ≈ 9.5 seconds for the SIFT dataset. For the GIST dataset, there are two cases (250 and 500) where our approach is even slightly faster than classical search. Finally, for the GloVe dataset, as in the case of SIFT, we can observe an almost constant overhead.

Since in our guided approach, at execution time, multiple recursive calls of Algorithm 5 carries their own overhead (considering the large number of queries performed), we ran again these experiments compiling with the compiler optimization option $-O3$, that performs optimizations on recursive calls. The times measured are shown in Table 3.3. As it can be observed, this optimization reduced considerably the execution time of both methods, and even, it made our approach run slightly faster than classical search (almost by a constant) in the SIFT and GloVe datasets. In the case of the GIST dataset, the time required by our approach was **almost half** of the classical search. In summary, our guided search approach has demonstrated empirically to have no significantly overhead and to run even faster than classical search, when subjected to the same limitations, and at same time obtaining positive recall gains over classical search.

Table 3.3: Total time for 100K queries (in seconds) using classical search approach vs the guided proposed (with -O3 option for compiler optimization).

	SIFT			GIST			GloVe		
Max. Distance Calculations	Query time (seconds)		Gain (recall)	Query time (seconds)		Gain (recall)	Query time (seconds)		Gain (recall)
	Classic	Guided		Classic	Guided		Classic	Guided	
100	10.969	10.797	21.37%	43.558	25.029	5.60%	9.167	9.659	6.14%
250	28.243	25.617	27.39%	110.971	59.117	20.92%	24.036	22.552	14.27%
500	58.703	50.911	5.16%	226.253	115.686	11.20%	50.594	44.631	10.56%
750	91.684	78.329	1.77%	345.229	176.746	5.22%	78.676	68.131	6.44%
1000	125.634	104.463	0.76%	464.278	234.372	2.70%	110.736	90.916	3.00%

Chapter 4

Learning to Navigate on Nearest Neighbor Graphs

In this chapter, we introduce a supervised framework for learning to navigate on NN graphs. For this purpose, we investigate the use of different topological properties of vertices to support the graph navigation through vertices that would lead to an earlier discovering of the true NN. First, in Section 4.1 we present an overview of the proposed framework based on Genetic Programming (GP) to automatically discover near-optimal ways for combining topological properties to conduce the graph navigation. Then, Section 4.2 describes the topological properties considered in the proposed framework. Section 4.3 describes the fitness function employed in the GP-based learning process.

4.1 Genetic Programming Framework for Better Graph Navigation

As shown previously in Algorithm 1 (Section 2.2.2), in classical approach for searching on NN graphs, vertices of graph are scored in the priority queue based only on their distances to the query, so the vertex with minimum score (minimum distance) is taken from the queue in each step and its neighbors visited. However, local topological information of vertices can be used to improve the criterion for selection of the next vertex to be explored (as seen in example shown in Figure 1.2), by better scoring those vertices that could allow a faster discovery of the true nearest neighbors. In the context of networks, there are many metrics [16] broadly studied in the literature that capture different properties of the topology of vertices' neighborhood.

In this work, we propose a GP framework to find a near-optimal scoring function that takes into consideration both the classical distance-to-the-query and local topological properties of vertices. The scores obtained by this function will be used to sort the vertices on the priority queue, aiming to minimize the number of vertices explored to discover the true nearest neighbors. GP has been shown to perform well in optimization scenarios like this [1, 17, 19, 39]. In the context of the GP approach explained above, an individual from the population corresponds to a candidate scoring function. We modeled this scoring function as a combination of the distance and topological properties through

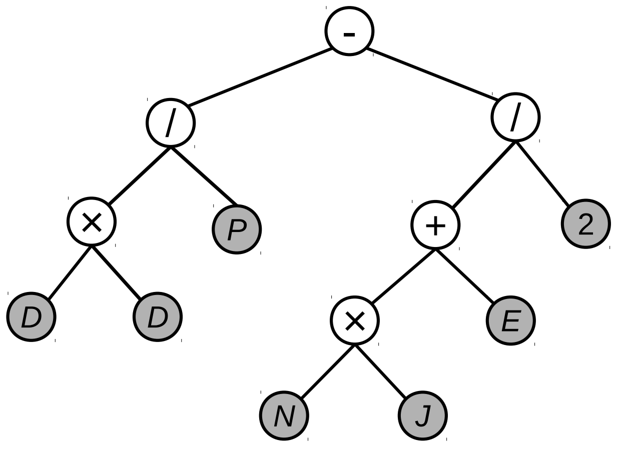


Figure 4.1: Example of individual in the proposed framework.

basic mathematical operators, therefore, we adopted the classical tree representation of GP individuals for mathematical functions, as detailed in Section 2.4. Figure 4.1 shows an example of an hypothetical individual that could be created in the proposed GP-based framework. This tree represents the vertex scoring function $\hat{f}(x) = \frac{(D \times D)}{P} - \frac{((N \times J)+E)}{2}$, that combines the following properties of a given vertex x: distance to the query (D), preferential attachment (P), vertex degree N, Jaccard coefficient (J), and the edge weigh (E). All these properties are described in the next section.

In Figure 4.2, it is shown an overview of the proposed framework for discovery of a more adequate scoring function for search on NN graphs. The whole process is analog to the GP process detailed in Section 2.4. At first, an initial population of candidate solutions is created randomly employing a set of functions composed of mathematical operators, and a set of terminals composed of several Search Dependent (SD) and Search Independent (SI) features – described in the next section. Then, this initial population is evolved over several generations. At the start of each generation, the fitness of each individual is evaluated. As the fitness function depends on the application, we defined the fitness function as the search performance of the scoring functions that represents the GP individuals. Therefore, given an individual $\hat{f}(x)$, to compute its fitness value, we average the search performance on a training set of queries obtained by using $\hat{f}(x)$ function in lines 3 and 9 of Algorithm 1 (replacing the *distance* function). The exact computation of this fitness value is detailed in Section 4.3. After the fitness evaluation, some individuals are selected to be subjected to genetic operators, and, finally, a new population is created, composed of the individuals produced by the genetic operators. At the end of this evolutionary process, the most fitted individual (scoring function) seen through the whole process is selected, and employed in the search algorithm to support the execution of new queries.

4.2 Topological Properties of Vertices

Let v be the current vertex at search that was popped from queue in line 6 of Algorithm 1, and u the neighbor of v being explored (line 7). We considered two types of features to

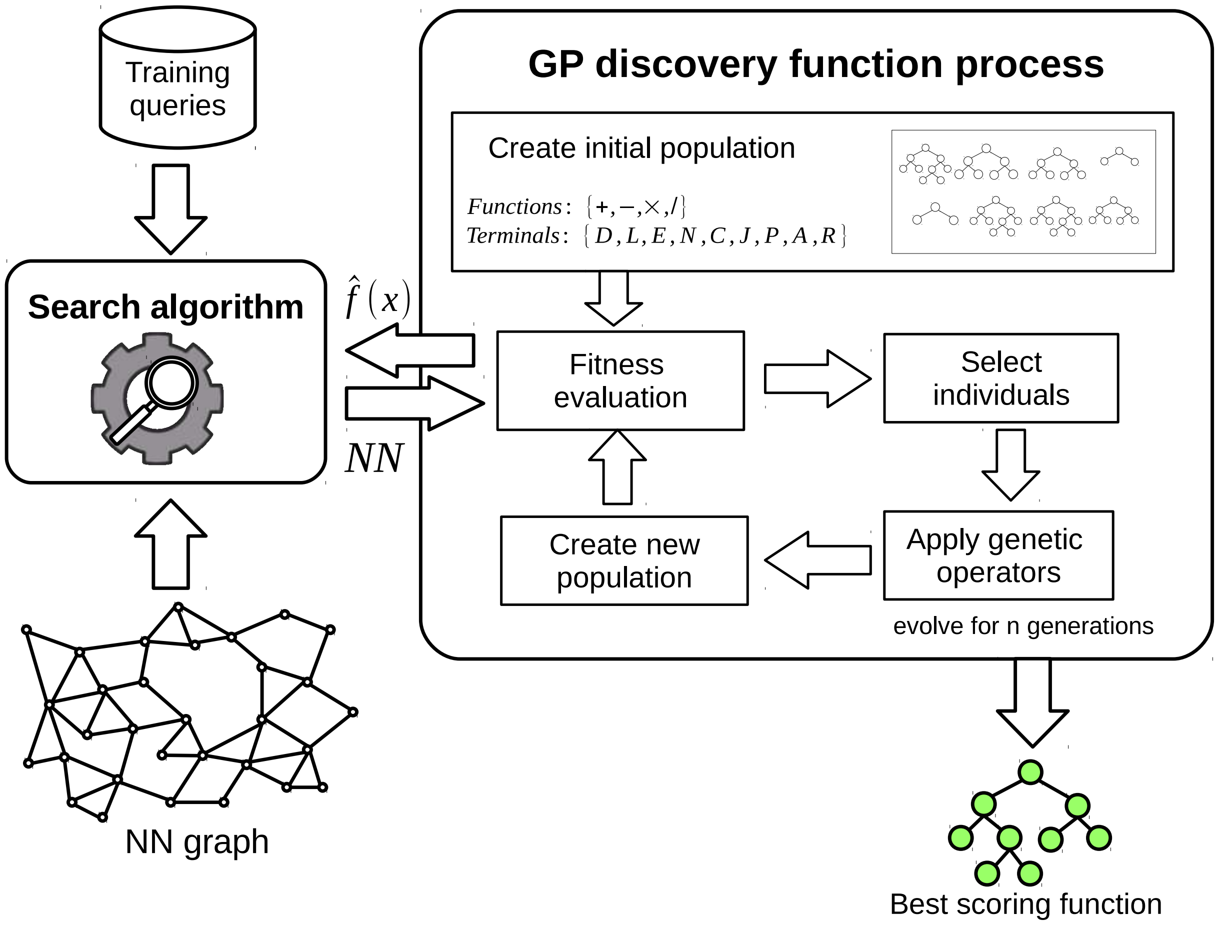


Figure 4.2: GP-based scoring function learning for NN search.

be taken into account by the scoring function discovery process: Search Dependent (SD) and Search Independent (SI).

Features from the first group can only be computed at search time. We included in our method the following search dependent features:

- **Distance (D)**: this is the traditional distance from vertex u to the query point (computed in line 9 of Algorithm 1).

- **Path length (L)**: easily, we can adapt Algorithm 1 to keep track of the length of the path (number of vertices) from the starting vertex to u. In the example shown in Figure 2.2, $L(u) = 3$ (path from v_0 to u).

On the other hand, search independent features depend on the local topology of NN graph's vertices, thus, these features can be pre-computed offline, adding no-extra cost at search time. We considered the following features from this category, most of them from the survey presented by Costa *et al.* [16] about network metrics:

- **Edge weight (E)**: the weight of the edge between v and u. In the example shown in Figure 2.2, $E(v_2, u) = dist(v_2, u)$.

- **Vertex degree (N)**: the degree of u. $N(u) = 5$, in the figure.
- **Common neighbors (C)**: the number of common neighbors between vertices v and u. $C(v_2, u) = 1$, in the figure.
- **Jaccard coefficient (J)**: a broadly used similarity measure commonly employed in information retrieval tasks, defined as:

  $$Jaccard(v, u) = \frac{|\tau(v) \cap \tau(u)|}{|\tau(v) \cup \tau(u)|}$$

  where τ is a function that returns the set of neighbors of a given vertex. For example, $J(v_2, u) = 1/8 = 0.125$, in Figure 2.2.
- **Preferential attachment (P)**: another well-known metric in networks, which serves as an indicator that vertices with many neighbors will create more connections in the future (in dynamic graphs). This metric is given by:

  $$PrefAttach(v, u) = |\tau(v)| \times |\tau(u)|$$

  where τ is the same as in Jaccard coefficient. $P(v_2, u) = 4 \times 5 = 20$, in the example (Figure 2.2).
- **Adamic Adar (A)**: a classical metric employed initially in the context of link prediction that is defined by:

  $$AdamicAdar(v, u) = \sum_{x \in \tau(v) \cap \tau(u)} \frac{1}{\log |\tau(x)|}$$

  thus, in example above, $A(v_2, u) = \frac{1}{\log(2)} = 3.32$.
- **Edge redundancy (R)**: we introduce this binary property that takes the value of 0 if exist any vertex w (different from v and u) that is neighbor of v and u, otherwise, this property takes the value of 1. In Figure 2.2, $R(v_2, u) = 0$.

For example, if the best scoring function found through the evolution process was $\hat{f}(x) = \frac{(D \times D)}{P} - \frac{((N \times J)+E)}{2}$, and considering that v_2, u, and q in Figure 2.2 represent the points $(4, 4)$, $(5, 3)$, and $(10, 10)$, respectively. Then, the score that will be associated with vertex u will be: $\hat{f}(u) = \frac{(\sqrt{72} \times \sqrt{72})}{20} - \frac{((5 \times 0.125)+\sqrt{2})}{2} = 2.58$.

We selected this set of features based on their associated low computational cost, an important requirement, as our method is expected to support searches on large graphs.

4.3 Fitness Function Computation

It is of paramount importance to define an adequate fitness function based on what we are aiming to optimize. As we intend to discover scoring functions that improve search performance on NN graphs, we decided to use a conventional metric on information retrieval known as *recall* to measure the search performance for a given individual $\hat{f}$. Also, in order to have an indicator of the improvement in terms of the *recall* obtained by using $\hat{f}$ instead of the classical approach, we defined the fitness function as the difference of the recall obtained between the search based on $\hat{f}$ and the search based only on the distance to the query. Formally, this fitness function is given by:

$$\mathit{fitnessNN}(\hat{f}) = \frac{1}{|T|} \sum_{t \in T} \Big(g(t, \hat{f}) - g(t, L2) \Big) \tag{4.1}$$

where the function $g(t, s)$ computes the average *recall@1* obtained on a training set of queries Q, by using s as scoring function at search, and limiting the number of vertices explored to t. Also, $L2$ represents the function that computes the Euclidean distance between any vertex and a query. The reason why to average the gain obtained for different values of t is to discover scoring functions that perform well regardless the maximum number of vertices allowed to explore.

4.4 Experiments

This section presents the the experimental protocol employed to validate the proposed technique, and our experimental results.

4.4.1 Datasets

We experimented with three different datasets: GloVe, SIFT, and YFCC100M. The GloVe and SIFT datasets are the same as those employed in experiments conduced to validate the proposed HCNNG (Chapter 3), described in Section 3.3.1. The new dataset YFCC100M of visual features vectors is described below:

- **YFCC100M:**[1] the Yahoo-Flickr Creative Commons 100 Million (YFCC100M) contains 99.2 million photos and 0.8 million videos from Flickr (we only considered the images). We employed the feature vectors provided by Popescu *et al.* [58], representing improved VLAD vectors, in which their initial dimensions (32,768) were reduced with PCA+whitening, maintaining the 128 most significant dimensions for our experiments. We selected randomly a subset of 1 million of images (referenced as YFCC from now on) for graph construction and 10 thousand queries.

4.4.2 NN graph baselines

We considered three of the methods described in Section 2.2.1 for construction of NN graphs: KGraph [22], SW-graph [45], and FANNG [27]. Also, we experimented with the NN graph created using the proposed HCNNG, described in Chapter 3. In the case of KGraph, we performed experiments using the implementation provided by the authors'

[1]YFCC100M: `http://multimedia-commons.s3-website-us-west-2.amazonaws.com` (As of January 2020).

website.[2] For the SW-graph method, we used the implementation included in the Non-Metric Space Library (NMSLIB).[3] For FANNG, we performed experiments with our own implementation. We did not consider the HNSW method, since the search employed by this technique differs significantly from the approach considered by all three methods, due to their hierarchical structure of multiple graphs. We leave the investigation of the use of our method combined with HNSW for future work.

In the case of SW-graph and FANNG, the maximum degree of vertices can not be delimited by any parameter at graph construction phase, thus, some vertices will probably end up with a very high number of neighbors, as we observed for the three dataset considered above. This would affect negatively the convergence on the GP evolution process, since the vertex degree is considered as input in GP individuals and this value could change significantly the value returned by the mathematical expressions represented by individuals. Therefore, we decided to limit the vertices degree in all graphs created. Selecting a low value for maximum degree could lead to an unconnected graph, so we tested with many cut-off values near to the average degree and determined the value of 60 as near-optimal cut-off point. To have a fair comparison between the three methods, we used this value for all of them. Also, we verified that the search performance was not affected significantly at this cut-off value (for all three methods).

4.4.3 GP set-up

The following GP configuration were used in our experiments. We based our choices on related work [1,17,19,39] and empirical results:

- **Population**: we experimented with different sizes of population, starting with a big enough value of 1000, and decreasing it while performance of best individuals were not affected. Thus, for final experiments, we created an initial population of 400 individuals, for all three datasets, using the *ramped-half-and-half* technique, restricting the individuals' tree representation to a maximum depth of 5.

- **Functions**: we employed the set of classical mathematical operators: $\{+, -, \times, /\}$. Additionally, we included the binary operators of max and min.

- **Terminals**: we included all the search dependent and independent features described in Section 4.2: $\{D, L, E, N, C, J, P, A, R\}$. Also, to maintain these values approximately at the same scale, we divided them by their correspondent maximum feature values. We made this aiming to assign the same importance to all features. Finally, we included random real values uniformly selected from the range $[-1, 1]$, as done in other related research initiatives [1,17,19,39], aiming to facilitate the scaling of some features.

- **Genetic operations**: in all experiments, we used the classical operators of reproduction, mutation, and crossover, employing the tournament selection method, with

[2]KGraph: `https://github.com/aaalgo/kgraph` (As of January 2020).
[3]NMSLIB: `https://github.com/nmslib/nmslib` (As of January 2020).

size 6, as criteria to select the individuals. In all experiments, the reproduction, mutation, and crossover operators were applied at a rate of 5%, 10%, and 85% of the population, respectively.

- **Fitness function**: we employed the fitness function given by (4.1). For the set T, we selected the logarithmic scaled values of $\{10^2, 10^{2.1}, 10^{2.2}, ..., 10^{3.9}, 10^4\}$, guarantying that individuals are optimized for very distinct situations, ranging from a restricted number of distance calculations to less limited scenarios.

- **Stopping criterion**: in the parameter exploration phase, we observed that, in most of experiments, after the 100th generation, the fitness value does not change significantly.

At fitness evaluation of individuals, a subset of 1,000 queries were selected randomly from the original test set of 10,000 queries, for each dataset. The remaining 9,000 were used to test the best individual found through the evolution process.

4.4.4 Experimental results

We employed the same Speed × Recall charts, as in previous chapter, to evaluate the performance of the proposed search approach. Results for 1-NN search of the proposed approach on the GloVe, YFCC, and SIFT datasets are presented in first column of Figure 4.3. Dashed curves (with suffix "GP") represent the search performance on the corresponding graphs using the GP-based scoring function to score vertices in priority queue.

For the GloVe dataset, as it can be observed, search performance results at 1-NN case showed a significant gain obtained by using the GP-based scoring function against the usual distance. This gain is observed for all the four methods considered for NN graph construction. In case of the YFCC and SIFT datasets, note that, although, the margin between the two curves (baselines and proposed GP-based search) seems to be small, figures are shown in logarithmic scale. Also, we run statistical per-query paired t-test with 95% confidence, over the 9,000 queries employed as test queries, and considering the same values of speedup used in Figure 4.3. Results of this statistical test are shown in Table 4.1, where "+" symbols means the statistical superiority of our GP-based search approach against classical search (Algorithm 1), symbol "−" means the opposite, and "=" means a statistical tie. The last row of the table presents the results of the statistical tests done over all queries and speedup values, considering all graph construction methods and datasets. These results demonstrate the statistical superiority of our approach against their corresponding baselines.

The following scoring functions were discovered by the proposed GP framework and employed in the final experiments that led to the results described above:

- **GloVe**:
  - **HCNNG**: $min((E + max(A, C)), (D + D))/(0.68 + E) + min((((D/L)/E) \times (D + min(D, L))),$ $((min(D, P)/0.68) + min(0.68, D)))$
  - **FANNG**: $min(L, min(D, (L + N)))/((L \times min(D, E)) \times (L + (-0.94 \times D))) + ((((D - E) - L)/(D \times L)) +$ $((N - 0.94 \times J) + (N + min(C, P))))$

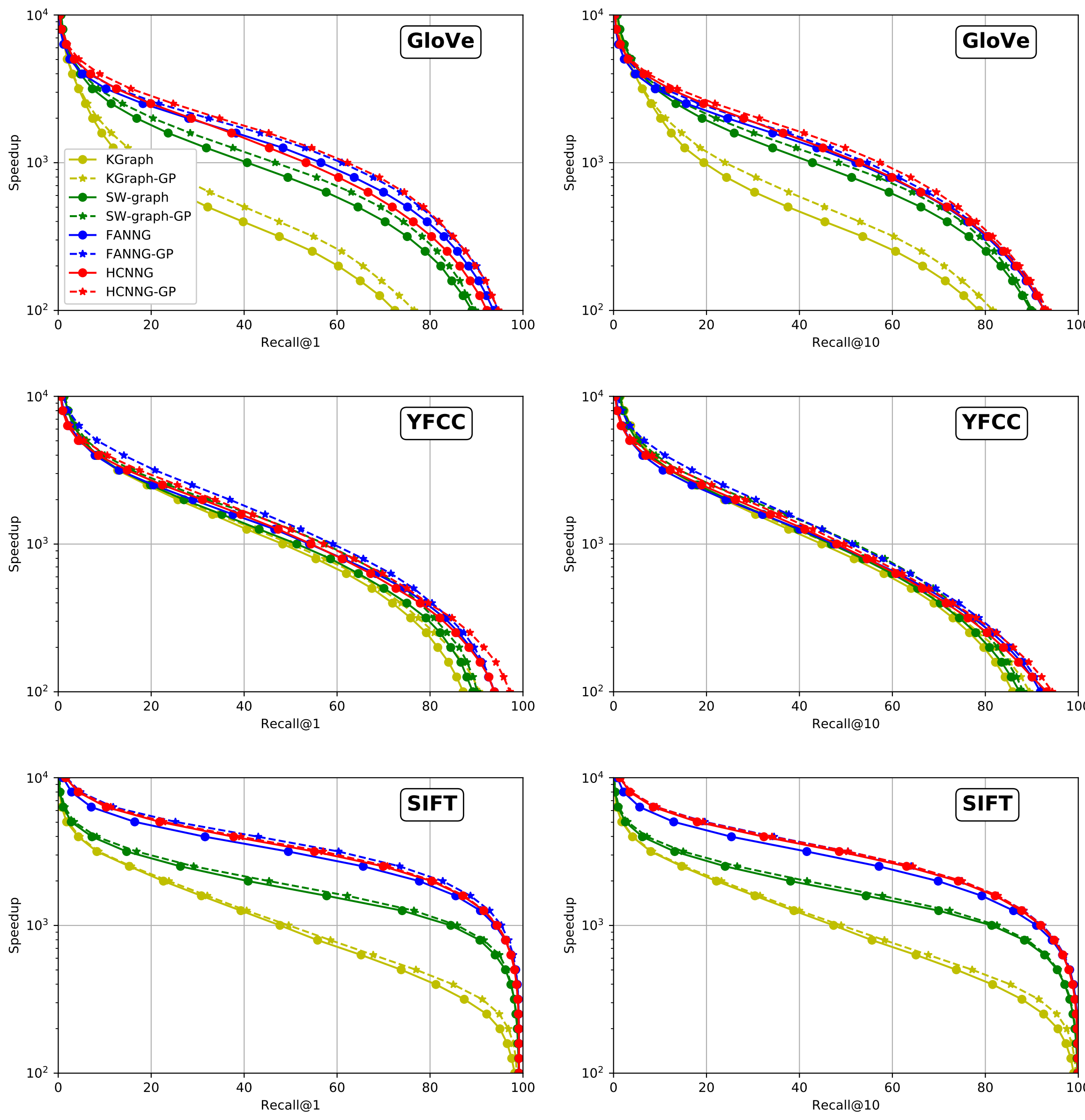


Figure 4.3: Speedup vs recall on GloVe, SIFT, and YFCC datasets.

- **SW-graph**: $min(min(N,(((-0.62/N)/D)/min(N,(D\times D)))),min(max(D,(-0.62/N)/D),max(((-0.62/N)/D),$ $min(D,(-0.62\times N)))))$
- **KGraph**: $(max((N-E)-E,max(D,P)-max(D,J))\times min(D,(P-max(0.96,D))))\times(max(((0.96-E)-E),0.96-L)+(((R-D)-D)\times((R-E)-max(E,P))))$

- YFCC:
  - **HCNNG**: $max((max(A,N)\times(P+(D/L))),(-0.11+(L\times(E\times L))))+((-0.11/min(D,(A+D)))+max((-0.11/min(D,N)),max(E,P)))$
  - **FANNG**: $max(min(min(C,J),max(0.10,J)\times min(L,N)),(N\times min(D,P)+(min(0.61,D)+D)))+min(min(0.61\times E,(D\times N)+(N\times N)),min(min(D,min(D,N)),min(D,(0.61\times N))))$
  - **SW-graph**: $((max(0.73,N)+(N+(D+N)))\times(max(0.73/E,N/L)+max((0.04/0.07),max(E,N))))+((max(0.57,(D+N))\times((N+N)+(N+N)))-(0.73/D))$
  - **KGraph**: $min(((((-0.13+E)/D)-(D+(-0.52\times N)))-(E+(P+0.13))/D),((min(A,N)+(N-R))-(min(A,R)/E))-((E+(P+0.13))/D))$

Table 4.1: Statistical paired t-test for 1-NN search, comparing our GP-based approach vs classical search ("+": gain, "−": lost, "=": tie, H: HCNNG, F: FANNG, S: SW-graph, K: KGraph).

Speedup	GloVe				SIFT				YFCC			
	H	F	S	K	H	F	S	K	H	F	S	K
$10^{2.0}$	+	+	+	+	=	=	=	+	+	=	+	+
$10^{2.1}$	+	+	+	+	=	−	−	+	+	=	+	+
$10^{2.2}$	+	+	+	+	=	−	=	+	+	+	+	+
$10^{2.3}$	+	+	+	+	+	=	+	+	+	+	+	+
$10^{2.4}$	+	+	+	+	=	=	+	+	+	+	+	+
$10^{2.5}$	+	+	+	+	=	=	=	+	+	+	+	+
$10^{2.6}$	+	+	+	+	=	=	=	+	+	+	+	+
$10^{2.7}$	+	+	+	+	+	=	+	+	+	+	+	+
$10^{2.8}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{2.9}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.0}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.1}$	+	+	+	+	=	+	+	+	+	+	+	+
$10^{3.2}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.3}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.4}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.5}$	+	+	+	+	+	+	+	+	+	+	+	=
$10^{3.6}$	+	+	+	=	+	+	+	+	+	+	+	+
$10^{3.7}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.8}$	+	+	+	−	+	+	+	+	+	+	+	=
$10^{3.9}$	+	+	=	=	+	+	+	+	+	+	+	+
$10^{4.0}$	+	+	=	=	=	+	+	+	+	+	+	=
General	+	+	+	+	+	+	+	+	+	+	+	+

- **SIFT**:
  - **HCNNG**: $min(((min(0.42, P) \times (A + D)) \times (E - (0.42/D))), (N - (0.42/D))) + (((min(0.42, P) \times (A + E)) \times (min(0.42, D) + (A + D))) - (E + min(D, min(D, P))))$
  - **FANNG**: $((((C/L) + min(N, P)) - (J - max(E, N))) + (((-0.64/D) + max(C, N)) - (min(P, P) - max(N, P)))) - ((min(P, (C/D)) - max(max(N, P), max(C, D))) - max((-0.64 \times (-0.64/D)), ((D/P) + max(L, N))))$
  - **SW-graph**: $max(((min(D, (E + J)) + ((D + D) - D)) - (((D + E) - N) \times ((D + D) - min(D, E)))), ((D + ((D + D) - (D \times E))) - ((E \times P) \times ((D + E) - D))))$
  - **KGraph**: $min((D - (min(D, E) \times min(0.41, E))), (max(min(C, J), (E - C)) + min(max(E, J), max(0.41, E)))) - (min(max(min(0.41, J), (E - C)), max((R/E), min(C, D))) \times (min(J, (D/R)) \times (max(0.41, R) + min(0.41, E))))$

From the set of scoring functions above, we could observe that the two features that have more influence on the final scores were the distance to the query (D) and the edge weigh (E), since they have higher frequency.

By employing these discovered functions, we also ran experiments for the 10-NN (search of the 10 nearest points). Results of these experiments are shown in second column of Figure 4.3, for the GloVe, YFCC, and SIFT datasets. Similarly to the case of 1-NN search, we also run statistical paired t-tests for 10-NN search. These results are shown in Table 4.2. As it can be observed, the behavior of all methods did not change significantly. The proposed approach yielded superior or equivalent results to their counterparts, in almost all points, demonstrating the effectiveness of discovered functions in

Table 4.2: Statistical paired t-test for 10-NN search, comparing our GP-based approach vs classical search ("+": gain, "−": lost, "=": tie, H: HCNNG, F: FANNG, S: SW-graph, K: KGraph).

Speedup	GloVe				SIFT				YFCC			
	H	F	S	K	H	F	S	K	H	F	S	K
$10^{2.0}$	+	+	+	+	+	=	=	+	+	+	+	+
$10^{2.1}$	+	+	+	+	+	=	=	+	+	+	+	+
$10^{2.2}$	+	+	+	+	+	−	+	+	+	+	+	+
$10^{2.3}$	+	+	+	+	+	=	+	+	+	+	+	+
$10^{2.4}$	+	+	+	+	+	−	+	+	+	+	+	+
$10^{2.5}$	+	+	+	+	+	=	+	+	+	+	+	+
$10^{2.6}$	+	+	+	+	+	=	+	+	+	+	+	+
$10^{2.7}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{2.8}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{2.9}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.0}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.1}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.2}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.3}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.4}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.5}$	+	+	+	+	+	+	+	+	+	+	+	+
$10^{3.6}$	+	+	+	=	+	+	+	+	+	+	+	=
$10^{3.7}$	+	+	+	=	+	+	+	+	+	+	+	=
$10^{3.8}$	+	+	+	−	+	+	+	+	+	+	+	=
$10^{3.9}$	+	+	+	=	+	+	+	+	+	+	+	=
$10^{4.0}$	+	+	+	−	+	+	+	+	+	+	+	=
General	+	+	+	+	+	+	+	+	+	+	+	+

the scenario of K-NN searches, even though these were trained to optimize the 1-NN search.

These experimental results showed the robustness of the proposed supervised search approach, since this can obtain search gains, compared to classical search, independently of the technique employed to construct the NN graphs. Additionally, the using of the propose GP-based scoring functions carry no significant extra cost, since all topological properties considered for vertices can be pre-computed before search.

Chapter 5

Billion-Size Nearest Neighbor Graphs

In this chapter, we introduce the extension of our HCNNG framework, introduced in Chapter 3, to make it scalable to billion-size datasets, employing quantization-based compression techniques to deal with the memory issues presented for most of existing techniques for ANNS. First, in Section 5.1, we present the algorithm for construction of NN graphs using compressed vectors. Next, in Section 5.2, we describe how to adapt the proposed GP-based supervised framework, introduced in Chapter 4, to pruning edges at graph construction phase, aiming to generate very sparse graphs and, therefore, leading to decrease the memory consumption to store the NN graph. Section 5.3 describes the search algorithm employed on the billion-size graphs created over the compressed vector. Finally, in Section 5.4, we describe the experimental protocol employed to validate the proposed techniques and present our results compared with the state-of-the-art techniques for billion-size ANNS.

5.1 Graph Construction

Existing NN graph techniques require to load all feature vectors to main memory, which could demand from hundreds gigabytes to terabytes of memory for billions of high-dimensional vectors, therefore leading to employ very costly architectures to be able to execute these techniques. Given these restrictions, we propose to compress the original feature vectors using a quantization technique. In this way, the memory consumption to load all data in memory scales to no more than a dozen gigabytes (using conventional parameters for quantization). We will introduce our approach based on the OPQ [23] technique to create the compact representations of original vectors. However, any other technique with possibly lower compression error could be used instead. We selected OPQ given its good trade-off between efficient encoding and coding accuracy [23].

More formally, given M orthogonal partitions of the D-dimensional vector space of the original data, the set of codebooks $\mathcal{C} = \{\mathcal{C}^1, \mathcal{C}^2, ..., \mathcal{C}^M\}$ (corresponding to each partition) and the rotation matrix R are learned through OPQ. Each codebook is composed of a set of K codewords (centroids) $\mathcal{C}^m = \{\mathbf{c}_1^m, \mathbf{c}_2^m, ..., \mathbf{c}_K^m\}$. Let $\mathbf{x}$ be a vector $\mathbf{x} \in \mathbb{R}^D$, then after applying the rotation $\mathbf{x}R^T$, and encode it based on $\mathcal{C}$, $\mathbf{x}$ is approximated by $\hat{\mathbf{x}} = [\mathbf{c}_{i_1}^1 \mathbf{c}_{i_2}^2 ... \mathbf{c}_{i_M}^M]$. Also, $\hat{\mathbf{x}}$ could be represented by the compact code $(i_1, i_2, ..., i_M)$

```
1  Function $HCG(\hat{\mathcal{P}}, \hat{f}, k, G)$
2      if $|\hat{\mathcal{P}}| < n$ then
3          $\mathcal{T} \leftarrow MST3(FullGraph(\hat{\mathcal{P}}))$
4          foreach $\hat{\mathbf{x}} \in \hat{\mathcal{P}}$ do
5              $N_{\hat{\mathbf{x}}} \leftarrow \varphi(\hat{\mathbf{x}}, G) \cup \varphi(\hat{\mathbf{x}}, \mathcal{T})$
6              $S_{\hat{\mathbf{x}}} \leftarrow AssignScores(N_{\hat{\mathbf{x}}}, \hat{f})$
7              sort $N_{\hat{\mathbf{x}}}$ with respect to $S_{\hat{\mathbf{x}}}$
8              $\varphi(\hat{\mathbf{x}}, G) \leftarrow Top(N_{\hat{\mathbf{x}}}, k)$
9      else
10         select randomly $\hat{\mathbf{x}}_1$ and $\hat{\mathbf{x}}_2$ from $\hat{\mathcal{P}}$
11         $HCG(\{\hat{\mathbf{x}} \in \hat{\mathcal{P}} \mid \hat{d}(\hat{\mathbf{x}}, \hat{\mathbf{x}}_1) < \hat{d}(\hat{\mathbf{x}}, \hat{\mathbf{x}}_2)\}, \hat{f}, k, G)$
12         $HCG(\{\hat{\mathbf{x}} \in \hat{\mathcal{P}} \mid \hat{d}(\hat{\mathbf{x}}, \hat{\mathbf{x}}_1) \geq \hat{d}(\hat{\mathbf{x}}, \hat{\mathbf{x}}_2)\}, \hat{f}, k, G)$
```

Algorithm 6: Hierarchical clustering procedure for a dataset $\hat{\mathcal{P}}$ with an expected size of clusters n.

employing just $M \log K$ bits.

Another problem with NN graphs approaches is the memory required to store the graph itself, since most of these approaches commonly generate a few hundreds of neighbors for each vertex (according to the results reported on million-scale datasets). The use of adjacency lists, a typical representation, would require storing hundreds of integers for each vertex, which at scale of billions leads to a very costly and impractical solution. The graph construction algorithm of the proposed HCNNG limits the number of neighbors per vertex to just 60. Even with this reduced number of neighbors, HCNNG still would demands a high amount of memory to store the graph for billion-size datasets (≈ 240 GB). In the following, we will describe the extension of the graph construction algorithm of HCNNG that scales up to billions of vectors in both time of execution and memory. The main differences relies on the use of quantization-based compressed vectors, and an edge pruning stage that aims to limit even more the degree of vertices. The new hierarchical clustering algorithm is detailed in Algorithm 6.

Let $\hat{\mathcal{P}} = \{\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, ..., \hat{\mathbf{x}}_N\}$ be a set that contains the compact representations of the original vectors, where N is the size of the dataset. Also, let $\varphi(\hat{\mathbf{x}}, \mathcal{G})$ be the function that returns the set of neighbors of $\hat{\mathbf{x}}$ on graph $\mathcal{G}$. The clustering procedure starts by checking if the size of $\hat{\mathcal{P}}$ has reached the expected size of clusters (n). In the case that the size of $\hat{\mathcal{P}}$ is still larger than the expected size, then the vectors of this set are divided into two subsets, regarding to the distance to two randomly selected vectors in $\hat{\mathcal{P}}$ (line 10). Next, these subsets are recursively clustered (lines 11-12). Otherwise, a fully connected graph is created on the current $\hat{\mathcal{P}}$, where the weight of edges is given by the distance between the vectors. Then, the $MST3$ ($\mathcal{T}$) is computed over the full graph (line 3). At this stage, in the graph construction algorithm of HCNNG, the edges contained in $\mathcal{T}$ would be directly added to the global graph G. Instead, we propose an extra step to select the best k-neighbors (line 8) for each vector in $\hat{\mathcal{P}}$, among those neighbors currently in G and the new ones found in $\mathcal{T}$ (line 5), conducing this selection based on a scoring function $\hat{f}$ (line 6).

As in HCNNG, we perform multiple times the hierarchical clustering procedure of Algorithm 6 in order to discover enough edges and to connect properly all vectors in $\hat{\mathcal{P}}$ through G. Given that in our case the set $\hat{\mathcal{P}}$ only stores the compact codes of original vectors, then the Euclidean distance computation (lines 3, 11 and 12) can be done of two forms: reconstructing the original vectors or using lookup tables. The first case is trivial, and the complexity of Algorithm 6 remains the same as in HCNNG, given by $O(ND\log(N/n) + NDn\log(n))$. For the second case, given two compact vectors $\hat{\mathbf{x}} = (i_1, i_2, ..., i_M)$ and $\hat{\mathbf{y}} = (j_1, j_2, ..., j_M)$, the squared Euclidean distance is computed based on the following expression:

$$\hat{d}(\hat{\mathbf{x}}, \hat{\mathbf{y}}) = \sum_{m=1}^{M} L2(\mathbf{c}^m_{i_m}, \mathbf{c}^m_{j_m})^2 \tag{5.1}$$

thus, if we pre-compute the square Euclidean distance between all the centroids of each codebook in $\mathcal{C}$, and store them in a lookup table, Equation 5.1 could be evaluated using just M additions. Therefore, the complexity of Algorithm 6 in this case decreases to $O(NM\log(N/n) + NMn\log(n))$, replacing D by M ($M < D$). Common values for M ranges from 8 to 16, while in the case of D from hundreds to thousands, thus, we employed in our experiments the last approach, since the memory overhead to store the lookup table $O(K^2M)$ is not significant.

5.2 Learning to Select Vertices' Neighbors at Graph Construction

For the scoring function, we propose two approaches: a naïve greedy approach and a supervised approach. In the first approach, we simply score the neighbors of a given vertex $\hat{\mathbf{x}}$ by their square Euclidean distance to $\hat{\mathbf{x}}$. For the supervised approach, we employ a scoring function discovered by the GP framework proposed in the Chapter 4, that combines the distance (the same as first approach) with topological properties of neighbors. But differently from our previous GP-based learning approach, we optimize this scoring function to select better vertices' neighbors at NN graph construction stage, instead of optimizing it to select vertices that lead to better graph traversal at search stage (as it was done previously).

More formally, for a given vertex $\hat{\mathbf{x}} \in \hat{\mathcal{P}}$ and $\hat{\mathbf{y}} \in \varphi(\hat{\mathbf{x}}, G)$, let $\bar{y}_{\hat{\mathbf{x}}} = \{\bar{y}_{\hat{\mathbf{x}}1}, \bar{y}_{\hat{\mathbf{x}}2}, ..., \bar{y}_{\hat{\mathbf{x}}F}\}$ be the set of F-properties of $\hat{\mathbf{y}}$ with respect to $\hat{\mathbf{x}}$, which may include the distance $\hat{d}(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ and some topological properties (e.g., the number of common neighbors between $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$, the Jaccard coefficient between $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$, and adamic adar metric). We aim to find a scoring function $\hat{f}(\bar{y}_{\hat{\mathbf{x}}})$ that combines through simple mathematical operators the properties' values in $\bar{y}_{\hat{\mathbf{x}}}$ that leads to a better scoring of $\hat{\mathbf{y}}$ with respect to $\hat{\mathbf{x}}$, and, therefore, to a better selection of the k-neighbors for each vertex on the global graph G.

The GP-based scoring function discovery process is outlined in Algorithm 7. This starts by creating an initial population composed of randomly generated candidate solutions (line 1), and creating the NN graph using the distance function (Equation 5.1) as scoring function in Algorithm 6. This initial population is evolved through a number of

```
1  Function LearnScoringFunction($\hat{\mathcal{P}}_t$, $k$, $Q$)
2      $P \leftarrow$ create initial random population
3      $G_{\hat{d}} \leftarrow HCG(\hat{\mathcal{P}}_t, \hat{d}, k)$
4      for $g$ generations do
5          for $f \in P$ do
6              $G_f \leftarrow HCG(\hat{\mathcal{P}}_t, f, k)$
7              fitness($f$) $\leftarrow R@1(Q, G_f) - R@1(Q, G_{\hat{d}})$
8          $S \leftarrow$ select individuals from $P$ based on fitness
9          $P' \leftarrow$ apply genetic operators on $S$
10         $P \leftarrow P'$
11     return $\hat{f} \in P$ with max fitness value
```

Algorithm 7: Genetic programming-based scoring function learning.

generations. At the start of each generation, all individuals (scoring functions) of current population are evaluated (lines 5-7) to know how well they perform at constructing NN graphs (compute their fitness value). To this end, a different NN graph is created using each scoring function (line 6). Then, as we did it previously in Chapter 4, we evaluated the performance of each function f based on the *recall@1* (R@1) improvements obtained (on a set of training queries Q) by using f instead of $\hat{d}$ at graph construction (line 7). After fitness evaluation, a subset of the individuals is selected (line 8) based on their fitness values, to finally apply the genetic operators (e.g., reproduction, mutation and crossover) over this subset (line 9), which will create the new individuals of the population for the next generation. At the end of evolution, the best candidate solution (with maximum fitness value) of last generation is returned (line 11).

This learning process is executed over small subsets ($\hat{\mathcal{P}}_t$) of the billion-size datasets, since it is very costly to create a graph for each individual of the population in each generation. Just the best scoring functions ($\hat{f}$) returned by Algorithm 7 are employed to create the NN graphs on the billion-size datasets.

5.3 Search on Billion-Size Graphs

We adapted the classical algorithm for searching on NN graphs to work with data compressed via OPQ. This is illustrated in Algorithm 8. Assume we have a query q and a maximum number of distance calculations T (scanning rate). To start a search, as it is done in other indexing structures [7,34], a lookup table is pre-computed (line 2) to store the square Euclidean distance from the query to all the centroids of each codebook in $\mathcal{C}$. Then, a random vertex v on graph G is selected to be the initial global minimum (line 3). Next, the distance from vertex v to the query is computed (line 4) through the Asymmetric Distance Computation (ADC, which is described in the next paragraph) [34], by using the lookup table $\mathcal{L}$ and the compressed codes for vertex v ($\hat{\mathcal{P}}_v$). In this way, we evaluate the distance using just M additions. A priority queue is then initialized with the vertex v, scoring it with the value of its distance to the query. The graph is traversed until there is no distance calculation left (line 6). In each step, the vertex with minimum distance is taken from the queue (line 7) to explore its neighbors. If a neighbor was not visited yet, then the distance to query is computed via ADC and it is pushed to queue (lines 10-11). Also, if any visited vertex has a distance lower to the global minimum, then this is updated (lines 13-15). Finally, the nearest visited vertex to the query is returned. This algorithm is easy to generalize for k-NN searches, by adding a list to store the closer k-vertices.

There are two ways for computing the distance from the query vector to the compressed vectors indexed by the NN graph. The first is to encode the query vector, this is, determine the index of nearest centroids in each codebook $\mathcal{C}^m \in \mathcal{C}$ and then compute the distance between the compressed representations in the same way as in Equation 5.1. This approach is known as *Symmetric Distance Computation* (SDC). Also, it can be done efficiently employing just M additions by pre-computing a lookup table to store the distance matrix for each codebook. The second approach to calculate the distance does not encode the query vector q, instead, prior to the search, pre-compute the distance from the original query vector to all the centroids of all codebooks and store them in a lookup table $\mathcal{L} \in R^K \times R^M$ (line 2 in Algorithm 8), where $\mathcal{L}_{i,m}$ stores the distance from q^m (m-th subvector after orthogonal partition) to centroid $c_i^m \in \mathcal{C}^m$. Then, at search time, to calculate the distance with a compressed vector $\hat{\mathbf{x}} = (i_1, i_2, ..., i_M)$, it just performs M additions by computing the next expression:

$$ADC(q, \hat{\mathbf{x}}) = \sum_{m=1}^{M} \mathcal{L}_{i_m,m} \tag{5.2}$$

This last approach is known as *Asymmetric Distance Computation* (ADC). The cost of encoding the query vector in SDC is the same that pre-computing the lookup table in ADC ($O(KD)$), however, experiments performed by Jégou *et. al* [34] evidenced that ADC approximate better than SDC to the euclidean distance between original uncompressed vectors. This is the reason why we use the ADC instead of the SDC.

5.4 Experiments

In this section, we present the experimental protocol, conducted parametric analysis, experimental results, and comparisons of our proposed approach with state-of-the-art indexing structures for billion-scale datasets.

5.4.1 Datasets

To validate the proposed index structure, we performed experiments on two billion-size datasets:

- **SIFT1B**:[1] this dataset is composed of one billion SIFT vectors, and is part of the well-known BIGANN benchmark for nearest neighbor search evaluation. Also, it is

[1]BIGANN: `http://corpus-texmex.irisa.fr` (As of January 2020).

```
1  Function SearchNN($q$, $T$, $\hat{\mathcal{P}}$, $\mathcal{C}$)
2      $\mathcal{L} \leftarrow LookupTable(q, \mathcal{C})$
3      $v \leftarrow$ random vertex in $G$
4      $d \leftarrow ADC(q, \hat{\mathcal{P}}_v)$
5      $\mathcal{Q} \leftarrow$ initialize priority queue with tuple $[n, d]$
6      while $T > 0$ do
7          $\bar{v} \leftarrow \mathcal{Q}.pop()$
8          foreach $u \in \varphi(\bar{v}, G)$ do
9              if $T > 0$ and $u$ not visited then
10                 $d^* \leftarrow ADC(q, \hat{\mathcal{P}}_u)$
11                 $\mathcal{Q}.push([u, d^*])$
12                 $T \leftarrow T - 1$
13                 if $d^* < d$ then
14                     $v \leftarrow u$
15                     $d \leftarrow d^*$
16     return $v, d$
```

Algorithm 8: Search algorithm for billion-scale graphs.

available 10 thousand queries for search evaluation with their respective groundtruth computed via exhaustive exploration.

- **DEEP1B**:[2] this dataset was introduced in a recent work [9], and it is composed of one billion deep learning-based feature vectors with 96 dimensions. For this dataset, authors also made available a set of 10 thousand queries with their groundtruth.

5.4.2 NN Graph Parameter Setting

In order to set the final configuration of parameters for our proposed approach, we extracted a subset of 1 million vectors from DEEP1B, called DEEP1M from now on, and a set of 10 thousand queries (not contained in DEEP1M) to evaluate the different configurations. In all experiments, we employed the same parameters as in HCNNG for creation of the NN graphs. The expected size of clusters n was set to 1000 and were performed 20 executions of the hierarchical clustering procedure to create enough edges for connecting properly the global graph.

On the other hand, in order to encode the vectors, we set the number of partitions of the original vector space to $M = 8$ (a common value used for compression), in all experiments. Also, we experimented encoding vectors with different quantities of bits per sub-quantizer. Figure 5.1 shows the NN search performance on graphs created over DEEP1M, encoded with different number of bits, where the x-axis represents the number of vectors scanned (T) and the y-axis, the average recall@1 obtained on a set of 10 thousand queries. We observed that significant gains in search accuracy can be obtained by increasing the number of bits (compared with the standard value of 8 bits). However,

[2]DEEP1B: `http://sites.skoltech.ru/compvision/noimi` (As of January 2020).

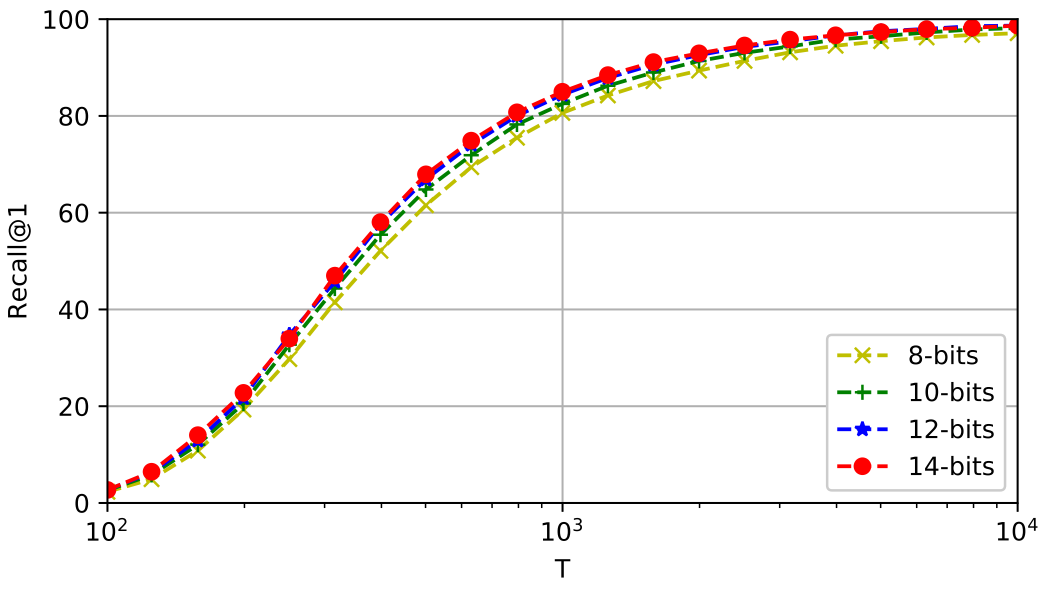


Figure 5.1: Search performance on DEEP1M encoding with different numbers of bits per sub-quantizer.

we also noted that this gain is just noticible up to 12 bits. Therefore, in the final experiments, we encoded the original vectors via OPQ with 12 bits per sub-quantizer. The NN graphs employed for this parametric analysis were created using just the distance from Equation 5.1 as scoring function, and vertices were limited to a maximum degree of $k = 15$ (as it will explained in next section).

5.4.3 Genetic Programming Set-up for Scoring Function

The GP configuration that were employed on final experiments is shown in Table 5.1. We used similar parameters as in previous chapter for learning the scoring function. We employed a reduced population, since it is costly to evaluate the fitness of individuals. We also observed that it was only necessary few generations to converge, differently from experiments reported previously. Most of vertices properties employed in the learning process are the same as those described in Section 4.2: weight edge (E), vertex degree (N), common neighbors (C), Jaccard coefficient (J), preferential attachment (P), Adamic Adar (A), and edge redundancy (R). We did not include the "length path" since that feature does not make sense in this case.

We performed a set of experiments to analyze the gains in search accuracy of the GP-based scoring function against the Euclidean distance. Figure 5.2 shows the search performance on DEEP1M using NN graphs created by limiting the degree of vertices to $k = 5, 10, 15$; and employing both scoring functions, the GP-based (dashed curves) and the Euclidean distance (solid curves). We observed that for a degree of $k = 5$, our GP-based scoring function led to the construction of a graph notably better than the distance-based for the task of NN search. Also, while we increased the vertices' degree, the overall search performance on graphs improved, but the gains for our GP-based scoring function

Table 5.1: GP parameter values.

Parameter	Value
Size population	200
Depth individuals	4
Functions	$\{+, -, /, *, min, max\}$
Terminals	$\{E, N, C, J, P, A, R\} \cup random[-1, 1]$
Num generations	20
Genetic operators	reproduction (5%), mutation (10%), crossover (85%)

decreased as well. It is guaranteed that search accuracy for GP-based function will be at least the same as the distance-based, since this property is also included on the function discovery process. The selection of final vertices' degree (k) depends on the memory available. For the billion-scale datasets considered, it is required approximately 4 GB for storing one neighbor per vertex. On the final NN graphs created over those datasets, we limited the vertices' degrees to $k = 15$. Although, the consumption of memory is high, compared to the other indexing structures, it is compensated with a fast and more accurate search, as we will show in next section.

The learning process was performed over a subset of 200 thousand vectors randomly selected from original datasets and it was executed for each different value of k. The scoring functions discovered on the GP-based learning process that were employed to create the final NN graphs are listed below (for $k = 15$):

- **SIFT1B**: $min(min(min(-0.42, (max(0.88, E) + (P + R))), (max(min(J, R), (P + R)) \times (-0.44 + (P + R)))), (-0.44 - ((0.51 + (P + R))/max(-0.42, E))))$
- **DEEP1B**: $max(((A/((E - 0.35) \times (R - J))) \times max(E, (A \times (E - J)))), (max(E, min(E, (E - R))) - (min(min(E, J), (R \times R))/(E + J))))$

5.4.4 Literature Comparison

We compared the proposed NN graph-based indexing approach to other three indexing schemes:

- **IVFADC** [34]: this system is based on the use of simple inverted indices. We used our own implementation to run this method. For creation of index we employed a coarse codebook with $K = 2^{14}$ centroids. Also, we compressed the residuals vectors though PQ with 12 bits per sub-quantizer, to perform a fair comparison with our proposed approach.

- **Multi-ADC** [7]: this indexing scheme is based on inverted multi indices. In this case, we also run the experiment using our own implementation. For index construction, as it was done in other works [9,24,36], we set the number of centroids for the coarse quantizers to $K = 2^{14}$. We encoded the original vectors via OPQ using also 12 bits per sub-quantizer.

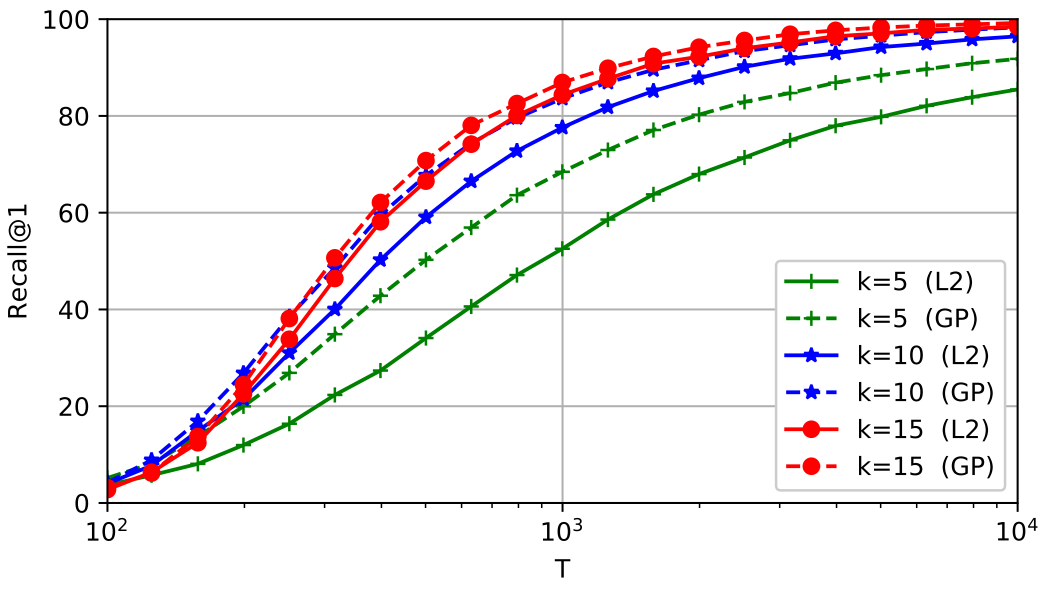


Figure 5.2: Search performance on DEEP1M using graphs with different degrees and scoring functions for neighbors selection.

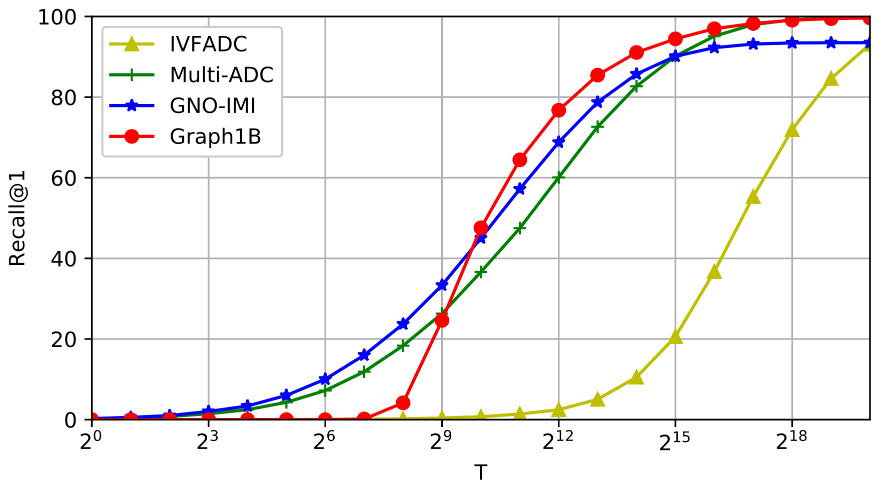
- **GNO-IMI** [9]: this structure is based on a generalization of inverted multi indices. In this case, we found part of the implementation on the authors' repository,[3] therefore, we implemented the remainder of the code in order to compare our approach with this technique. We used the same parameters for index construction reported by authors (for the same datasets). We encoded original vectors as proposed by authors, through local OPQ, however, we were only able to use 8 bits per sub-quantizer, since 2^{14}-product quantizers should be learned, and increasing the number of bits would take from several days to weeks for encoding vectors. We even had to reduce the number of vectors for each local PQ learning to 10 thousand vectors, in order to encode the whole dataset in a reasonable time.

The implementation of the proposed approach and all baselines considered in experiments were CPU-oriented and executed using 50 threads on a Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz.

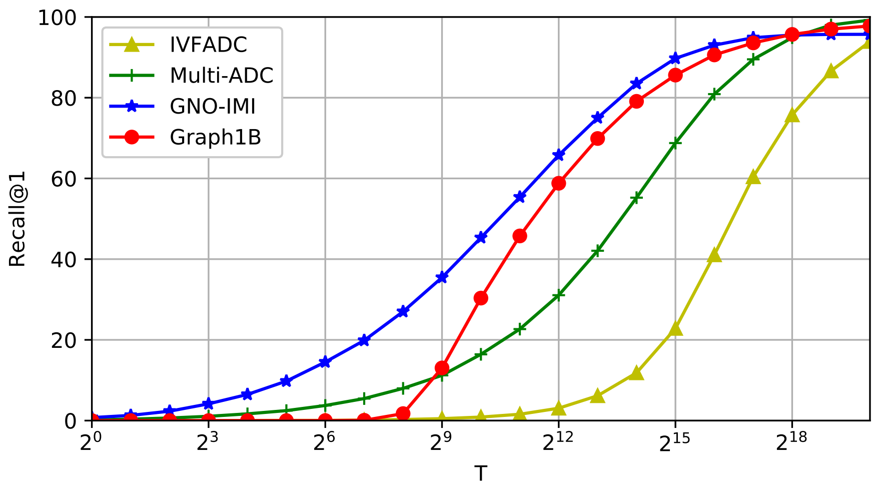
5.4.5 Results on Billion-Size Datasets

To evaluate the search performance of the different indexing structures, we first conduced a common analysis to measure the number of vectors scanned by the indices (T) vs the recall obtained on the search of the 1-NN (only the nearest neighbor). This analysis did not consider the different time overheads of each technique. Figures 5.3a and 5.3b show the results of this analysis for the SIFT1B and DEEP1B datasets, respectively. Our proposed technique is represented by the red curve, called Graph1B from now on. In the case of SITF1B, for a reduced number of scannings, we can observe that the GNO-IMI

[3]GNO-IMI: `https://github.com/arbabenko/GNOIMI` (As of November 2019).



(a) SIFT1B



(b) DEEP1B

Figure 5.3: 1-NN search performance on billion-size datasets.

and Multi-ADC performed better than the proposed Graph1B. This is because they start exploring lists with keys near to the query (which also lead to a time overhead discussed below), and we only use a random initialization on the search algorithm for NN graphs. However, from $T \approx 2^9$ and $T \approx 2^{10}$ on, the proposed Graph1B outperformed Multi-ADC and GNO-IMI, respectively. The IVFADC system demonstrated to need very high number of vector scannings to reach high recall values, obtaining the worst performance among all indexing structures considered.

In the case of DEEP1B dataset, the GNO-IMI and Multi-ADC indices, again, performed slightly better than the proposed Graph1B for a reduced number of vector scannings, but differently from SIFT1B, the GNO-IMI stayed on the top for almost all values of T. The proposed Graph1B outperformed consistently the Multi-ADC from $T \approx 2^9$ on, and it becomes competitive with GNO-IMI from $T \approx 2^{13}$ on. Also, as in the case of SIFT1B, the IVFADC index obtained the worst performance for search.

In order to evaluate the real performance of all indexing structures, we measured the time for processing the 10 thousand queries for both datasets. Figures 5.4 and 5.5 show the time measured in seconds required for processing all queries for the SIFT1B and DEEP1B datasets, respectively. For SIFT1B, we evaluated two types of searches: 1-NN (search of the nearest neighbor) and 100-NN (search of the 100 nearest neighbors). For DEEP1B, we only were able to run 1-NN, since the ground-truth provided for this dataset only contains the true nearest neighbor. For both datasets, the overhead of each method is given by the time in which the curves starts in x-axis. In SIFTB dataset, the proposed Graph1B obtained the lowest overhead, due to the simple random initialization for search, and outperformed consistently all other indexing schemes in almost any point, both in time and search accuracy, for the 1-NN. In the case of inverted multi indices-based techniques, Multi-ADC and GNO-IMI, both showed a high overhead since they perform an expensive preliminary search of the lists with keys closer to the query. The proposed Graph1B performed approximately 2.4× and 3.5× faster than Multi-ADC and GNO-IMI, respectively, at reaching 90% of recall. Finally, the IVFADC showed a overhead comparable with GNO-IMI, but the overall performance was the worst when compared with all indices. For 100-NN experiments, the accuracy of all indices decreased, but their relative performance did not change significantly, standing Graph1B yet as the more fast and accurate.

In the case of DEEP1B dataset, similar overheads as in SIFT1B were observed. Again, the proposed Graph1B obtained the lowest overhead and outperformed all indexing structures considered for almost all points. The high overhead of GNO-IMI index led to a drop on its search performance, even though this showed in previous experiment (Figure 5.3b) to perform slightly better than Graph1B and outperform consistently to Multi-ADC. Considering the results of all analysis performed on these datasets, the proposed Graph1B showed to be the more consistent index in terms of scanning rate, search accuracy, and search time.

5.4.6 Resource Consumption

We performed a final analysis to measure the memory peak at search and the index construction time for each technique and dataset. Table 5.2 presents the values obtained on each measurement. In the case of the memory, this includes the compressed data, the index, and other auxiliary variables (such as lookup tables). There is a few difference in the memory usage between the two datasets (in almost all cases). This is due to the dimensionality of data, given that DEEP1B has 96 and SIFT has 128 dimensions. The impact was not significant since the vectors are already compressed using the same amount of bits for the two datasets. The proposed Graph1B required the highest amount of memory from all indices, since it is costly to store one neighbor per vertex (≈ 4GB). On the other hand, for index construction time, the difference between the measurements on the two datasets are significant. In this case, the dimensionality had more impact, since in order to index a vector, all dimensions need to be processed. Also, we can observe that the GNO-IMI index required the highest amount of time for index construction, among all techniques considered.

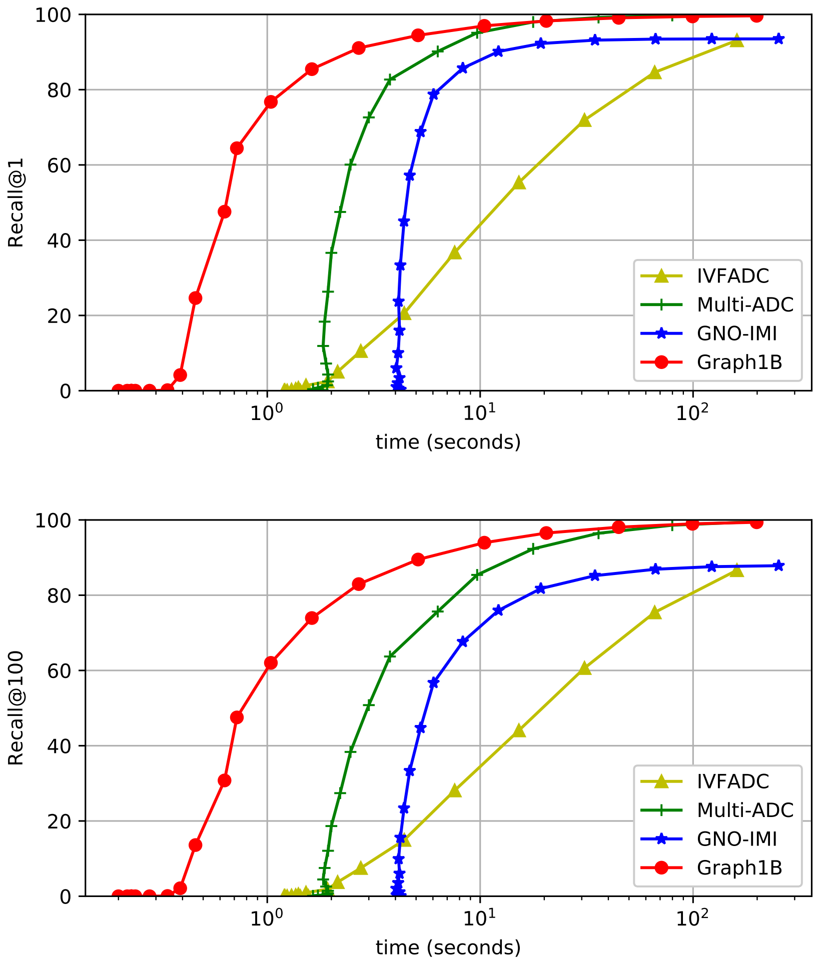


Figure 5.4: Search time for 10 thousand queries on SIFT1B. Results for the 1-NN (top) and 100-NN (bottom) searches.

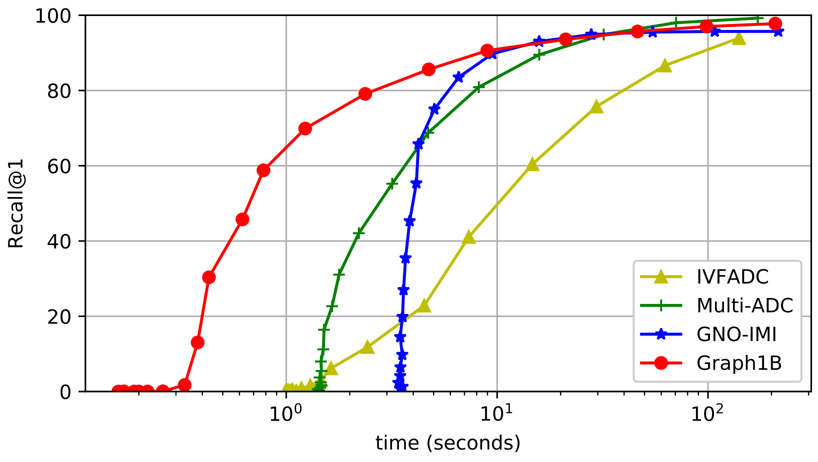


Figure 5.5: Search time for 10 thousand queries on DEEP1B.

Table 5.2: Resources consumption for indexing techniques.

	Search memory peak (GB)		Construction time (hours)	
Method	**SIFT1B**	**DEEP1B**	**SIFT1B**	**DEEP1B**
Graph1B	72.2	72.2	41.3	37.8
GNO-IMI [9]	31.5	30.9	65.1	55.9
Multi-ADC [7]	28.6	27.5	23.2	18.4
IVFADC [34]	20.0	19.7	18.9	14.2

Chapter 6

Conclusions and Future Work

The size of existing multimedia collections and the high dimensionality of vectors employed to represent them, present many challenges at the moment of designing data structures for indexing this data, especially in the task of NN search, since this is a principal routine in many machine learning, computer vision and information retrieval related tasks. We focused on this thesis to the investigation of novel indexing schemes based on NN graphs to support more efficient and accurate NN searches, motivated by previous success achieved by techniques from this group on the NN search task.

In the following sections we will summarize the principal contributions presented along the chapters, describing their relation with the hypotheses and research questions presented in Chapter 1, then, we present future research directions and finalize the thesis with the list of publications resulted from our research work.

6.1 Summary of Contributions

In Chapter 3 we introduced a novel approach for an efficient construction of easy-to-traverse NN graphs. Our idea explores the relationship of proximity between points in the same cluster, performing multiple clustering procedures to reinforce the connectivity of vertices. We addressed this first contribution by answering the following research questions:

Q1.1 *What is the best way to connect points inside clusters?* **Answer:** as we detailed in Section 3.1, connecting the points by minimum spanning trees with maximum degree 3 produced the best results among all structures considered.

Q1.2 *How can sub-graphs created by clusters be merged?* **Answer:** to create the final global graph, we just performed the union of all sets of edges and vertices of sub-graphs.

Q1.3 *Which clustering algorithm should be employed?* **Answer:** we employed the hierarchical clustering due to their low time complexity.

Experimental results presented in Section 3.3.6, conduced over million-size datasets, showed higher recall values than other graph-based techniques for high speedups (most notoriously in 1-NN search); and a comparable convergence to high recalls with better results than those observed for state-of-the-art techniques at searching different number of nearest neighbors. These experimental results led us to corroborate our **Hypothesis 1**: *The use of multiple clustering leads to the construction of NN graphs, which are faster to traverse at search time.*

The second contribution presented on Chapter 3 was the *guided search*, that employs KD-Trees-like auxiliary structures to select a better starting vertex for search, and to avoid exhaustive distance computations to all neighbors of current vertex at any step of search. We addressed this contribution by answering the following research questions:

Q2.1 *How KD-Trees can be employed to improve the starting vertex selection on NN graph searches?* **Answer:** creating multiple global KD-Trees and performing soft searches over them, to finally select the best vertex among those contained in the same leaf as the query.

Q2.2 *How KD-Tree-like structures can be employed to avoid visit unnecessary vertices at graph navigation process?* **Answer:** we employed local KD-Trees-like structures to partition the space with respect to each vertex, and, at search time, just explore the neighbors of vertices that are contained in the same sub-space than the query.

The analysis of time overhead performed in Section 3.3.7, related to the use of auxiliary KD-Trees, demonstrated that the use of these data structures in addition to generate gains in search results, they also carry no significant extra time execution, corroborating our **Hypotesis 2**: *The use of classical tree structures for indexing improves the NN search results on NN graphs, with no significant extra cost.*

In Chapter 4, we introduced a novel learning framework based on genetic programming that explored different topological properties in NN graphs, aiming to improve the order in which vertices are visited at search time, and, therefore, reducing the number of vertices explored to discover the true nearest neighbors. We addressed this contribution by answering the following research questions:

Q3.1 *How to combine the topological properties with the distance?* **Answer:** by means of mathematical operators to compose new scoring functions for the priority queue.

Q3.2 *Which learning technique can be used to find near optimal combinations of topological properties?* **Answer:** we employed genetic programming, since it was employed with success combining different evidences in similar scenarios.

Q3.3 *Which topological properties should be considered?* **Answer:** the list of topological properties considered was listed in Section 4.2. We selected those based on their low time complexity to be computed.

Q3.4 *Which function should be optimized in the learning process?* **Answer:** since we evaluated the search results using the *recall* metric, we also defined the fitness function based on this metric, as it is detailed in Section 4.3.

The experimental results and statistical tests presented in Section 4.4.4 showed that searches on NN graphs can be improved by considering other kinds of vertices' features, besides the distance to the query, combining them by means of the mathematical expression returned by the learning stage of the proposed framework. Also, the proposed technique has shown to obtain improvements against the usual search approach, in terms of recall, independently of the technique employed to construct the NN graph. These results led us to confirm our **Hypothesis 3**: *The use of topological information of vertices (along with the distance to query) through a learning scheme leads to a better selection of next vertex (in each step of NN graph search), which fosters the earlier discovering of the true NN.*

Finally, we presented in Chapter 5 a memory-aware version of our initial HCNNG algorithm for creation of NN graphs on billion-scale datasets. Also, we extended the learning framework proposed in Chapter 4 to, in this case, improving the selection of vertices' neighbors at graph construction phase based on topological properties of vertices, aiming to create graphs with very low vertices' degree. To the best of our knowledge, this is the first reported NN graph technique that scaled up to billion-size datasets in the task of ANNS. We addressed this contribution by answering the following research questions:

Q4.1 *Does the compression of original vectors affects to the accuracy of search on NN graphs?* **Answer:** yes. In the parameter setup presented in Section 5.4.2, it was observed that when the compression error is reduced (by using more bits in the encoding), the search results improved.

Q4.2 *Which heuristics can be used for pruning edges at graph construction?* **Answer:** we experimented with the distance as the unique criteria to select vertices' neighbors, saving for each vertex just the k-nearest vertices found in the whole graph construction stage.

Q4.3 *Can we get search performance improvements by exploiting the topological properties of vertices for pruning edges at graph construction?* **Answer:** yes. Experiments in Section 5.4.3 demonstrated the improvements in search results (compared to naïve approach) by selecting vertices' neighbors based on the topological information of vertices at graph construction phase.

Q4.4 *Does our NN graph technique still maintain its top search performance at NN search when compared to the state-of-the-art schemes for billion-scale ANNS?* **Answer:** yes. Our proposed technique showed best query time/recall trade-off than state-of-the-art techniques considered.

Q4.5 *How much resources need the proposed NN graph technique in comparison with the state of the art?* **Answer:** the proposed technique required approximately the double of memory and index construction time than best baseline, but it is still far lower than the resources that another NN graph-based technique would require to scale up to billion size dataset, which it would be approximately 8 times the memory required for our proposed approach (considering the same datasets).

Graphs created with the proposed approach were compared with state-of-the-art indexing structures on two billion-scale datasets. Our experimental results showed the lowest time overhead at search for the proposed graph-based index and very competitive search performance. Even more, our proposed technique required significant lower index construction time than a state-of-the-art indexing technique based on multi-inverted indices. These results lead us to corroborate **Hypothesis 4**: *Compressing vectors via quantization schemes and the adoption of suitable pruning strategies at construction time allow the construction of NN graphs on billion-size datasets with a reasonable memory consumption and time construction.*

6.2 Future Work

We presented in this thesis novel approaches for large scale indexing of high dimensional data and approximate nearest neighbors search. In addition to the contributions presented, we visualize some future research directions that we discuss in the next paragraphs.

In all experiments conduced to validate the proposed approaches, we only considered the Euclidean distance. However, in many scenarios there are other distance or similarity functions that are more adequate to compare two vectors, for example, in text retrieval the use of the cosine similarity is preferred. Furthermore, most of space partitioning techniques do not support searches on non-metric spaces. The proposed techniques in this thesis do not make assumptions over the space of data, so any distance/similarity function can be used. We leave for future work the evaluation of the proposed techniques on other metric and non-metric spaces.

Concerning to the GP-based learning framework presented in Chapter 4, we believe that the inclusion of other more complex topological properties in the proposed framework could potentially improve the search results of the proposed technique, like, for example, Page-Rank or Random Walks. Future work also comprehends the extension of this framework to support searches on billion-size NN graphs, taking into consideration the possible time/memory overheads that this technique would carry.

Recently, the high computational power of GPU's has been exploited in different applications to accelerate the execution of algorithms. Some works [27, 35] focused on the task of ANNS also have presented GPU's-based implementations, reducing considerable the time for creation of the indexing structures. Possible future work could also encompass the development of a GPU-based implementation of the algorithms presented in this thesis for creation of NN graphs. Another architecture-related optimization that was studied previously in the literature [45], was the use of distributed architectures to deal with high concurrency queries. We also plan to investigate this possibility.

In real scenarios, collections of multimedia data are highly dynamic, which demand that indexing structures to support insertion of new data. Some works [45, 46] have proposed approaches to insert new elements on NN graphs. We intend to investigate in the future the use of these insertion algorithms on the graph structures proposed in this thesis. An important aspect that will be considered is that this insertion should keep the same navigating properties that original NN graphs created by our proposed algorithms.

Finally, NN graphs were also applied with success in the context of image annotation [31,41,63,64] by propagating labels through the neighbors of each vertex. We believe that the techniques proposed in this thesis could help to improve the accuracy of annotations. We left this evaluation as future work.

6.3 Research Outcomes

This section presents the list of the papers that were accepted or submitted during the doctorate period. They are listed in the following.

Related to this thesis:

1. **Javier Vargas Muñoz**, Marcos André Gonçalves, Zanoni Dias, Ricardo da Silva Torres: **Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search**. *Pattern Recognition*, 2019 (Chapter 3)

2. **Javier Vargas Muñoz**, Zanoni Dias, Ricardo da Silva Torres: **A Genetic Programming Approach for Searching on Nearest Neighbors Graphs**. *International Conference on Multimedia Retrieval*, 2019 (Chapter 4)

Other collaborations:

3. Keiller Nogueira, Samuel G. Fadel, Ícaro C. Dourado, Rafael de O. Werneck, **Javier Vargas Muñoz**, Otávio A. B. Penatti, Rodrigo Tripodi Calumby, Lin Tzy Li, Jefersson A. dos Santos, Ricardo da Silva Torres: **Exploiting ConvNet Diversity for Flooding Identification**. *IEEE Geoscience and Remote Sensing Letters* [51].

4. **Javier Vargas Muñoz**, Lin Tzy Li, Ícaro C. Dourado, Keiller Nogueira, Samuel G. Fadel, Otávio Augusto Bizetto Penatti, Jurandy Almeida, Luis A. M. Pereira, Rodrigo Tripodi Calumby, Jefersson A. dos Santos, Ricardo da Silva Torres. RECOD@ Placing Task of MediaEval 2016: **A Ranking Fusion Approach for Geographic-Location Prediction of Multimedia Objects**. *In MediaEval 2016.*

Bibliography

[1] Juan F. Hernández Albarracín, Edemir Ferreira, Jeferson A. dos Santos, and Ricardo da S. Torres. Fusion of genetic-programming-based indices in hyperspectral image classification tasks. In *Proceeding of the IEEE International Geoscience and Remote Sensing Symposium*, pages 554–557, July 2017.

[2] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communication of the ACM*, 51(1):117–122, January 2008.

[3] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1993.

[4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In Christian Beecks, Felix Borutta, Peer Kröger, and Thomas Seidl, editors, *Similarity Search and Applications*, pages 34–49, Cham, 2017. Springer International Publishing.

[5] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[6] Artem Babenko and Victor Lempitsky. Additive quantization for extreme vector compression. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 931–938, June 2014.

[7] Artem Babenko and Victor Lempitsky. The inverted multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(6):1247–1260, June 2015.

[8] Artem Babenko and Victor Lempitsky. Tree quantization for large-scale similarity search and classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4240–4248, June 2015.

[9] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, June 2016.

[10] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH forest: Self-tuning indexes for similarity search. In *Proceedings of the 14th International Conference on World Wide Web*, WWW'2005, pages 651–660, 2005.

[11] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[12] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML'2006, pages 97–104, 2006.

[13] Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB'1995, pages 574–584, 1995.

[14] Yong-Sheng Chen, Yi-Ping Hung, Ting-Fang Yen, and Chiou-Shann Fuh. Fast and versatile algorithm for nearest neighbor search based on a lower bound tree. *Pattern Recognition*, 40(2):360 – 375, 2007.

[15] Jian Cheng, Cong Leng, Jiaxiang Wu, Hainan Cui, and Hanqing Lu. Fast and accurate image matching with cascade hashing for 3d reconstruction. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, page 1–8, USA, 2014. IEEE Computer Society.

[16] Luciano da F. Costa, Francisco A. Rodrigues, Gonzalo Travieso, and Paulino Ribeiro Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in physics*, 56(1):167–242, 2007.

[17] Ricardo da S. Torres, Alexandre X. Falcão, Marcos A. Gonçalves, João P. Papa, Baoping Zhang, Weiguo Fan, and Edward A Fox. A genetic programming framework for content-based image retrieval. *Pattern Recognition*, 42(2):283–292, 2009.

[18] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC'2008, pages 537–546, 2008.

[19] Moises G. de Carvalho, Alberto H. F. Laender, Marcos A. Gonçalves, and Altigran S. da Silva. A genetic programming approach to record deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(3):399–412, March 2012.

[20] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[21] Thanh-Toan Do, Anh-Dzung Doan, Duc-Thanh Nguyen, and Ngai-Man Cheung. Binary hashing with semidefinite relaxation and augmented lagrangian. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *European Conference on Computer Vision*, pages 802–817, Cham, 2016. Springer International Publishing.

[22] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW'2011, pages 577–586, 2011.

[23] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, June 2013.

[24] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):744–755, April 2014.

[25] Eva Gómez-Ballester, Luisa Micó, and Jose Oncina. Some approaches to improve tree-based nearest neighbour search algorithms. *Pattern Recognition*, 39(2):171 – 179, 2006. Part Special Issue: Complexity Reduction.

[26] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD'1984, pages 47–57, New York, NY, USA, 1984. ACM.

[27] Ben Harwood and Tom Drummond. FANNG: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5713–5722, 2016.

[28] Felix Hausdorff. Dimension und äußeres maß. *Mathematische Annalen*, 79(1):157–179, 1918.

[29] James Hays and Alexei A. Efros. Scene completion using millions of photographs. *ACM Transactions on Graphics*, 26(3):4–es, July 2007.

[30] Ran He, Yinghao Cai, Tieniu Tan, and Larry Davis. Learning predictable binary codes for face indexing. *Pattern Recognition*, 48(10):3160–3168, October 2015.

[31] Michael E. Houle, Xiguo Ma, Vincent Oria, and Jichao Sun. Improving the quality of k-nn graphs for image databases through vector sparsification. In *Proceedings of International Conference on Multimedia Retrieval*, ICMR'2014, pages 89:89–89:96. ACM, 2014.

[32] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[33] Khalid Jebari. Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3:333–344, 12 2013.

[34] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[35] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, pages 1–1, 2019.

[36] Yannis Kalantidis and Yannis Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2329–2336, June 2014.

[37] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 163–170, 2000.

[38] John R. Koza. *Genetic Programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[39] Anísio Lacerda, Marco Cristo, Marcos André Gonçalves, Weiguo Fan, Nivio Ziviani, and Berthier Ribeiro-Neto. Learning to advertise. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'2006, pages 549–556, New York, NY, USA, 2006. ACM.

[40] Venice E. Lionga, Jiwen Lu, Gang Wang, Pierre Moulin, and Jie Zhou. Deep hashing for compact binary codes learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2475–2483, June 2015.

[41] Jing Liu, Mingjing Li, Qingshan Liu, Hanqing Lu, and Songde Ma. Image annotation via graph learning. *Pattern Recognition*, 42(2):218–228, February 2009.

[42] Wei Liu, Cun Mu, Sanjiv Kumar, and Shih-Fu Chang. Discrete graph hashing. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, pages 3419–3427. Curran Associates, Inc., 2014.

[43] Qin Lv, Moses Charikar, and Kai Li. Image similarity search with compact data structures. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management*, CIKM'2004, page 208–217, New York, NY, USA, 2004. Association for Computing Machinery.

[44] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB'2007, pages 950–961, 2007.

[45] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[46] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*, 2016.

[47] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proceedings of the International Conference on Computer Vision Theory and Application*, pages 331–340, 2009.

[48] Marius Muja and David G. Lowe. Fast matching of binary features. In *Proceedings of the 9th Conference on Computer and Robot Vision*, pages 404–410, 2012.

[49] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

[50] David Nister and Henrik Stewenius. Scalable recognition with a vocabulary tree. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '06, page 2161–2168, USA, 2006. IEEE Computer Society.

[51] Keiller Nogueira, Samuel G. Fadel, Ícaro C. Dourado, Rafael de O. Werneck, Javier A. Vargas, Octavio A. Penatti, Rodrigo T. Calumby, Lin T. Li, Jefersson A. dos Santos, and Ricardo da S. Torres. Exploiting convnet diversity for flooding identification. *IEEE Geoscience and Remote Sensing Letters*, 15(9):1446–1450, Sep. 2018.

[52] Mohammad Norouzi and David J. Fleet. Cartesian k-means. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3017–3024, June 2013.

[53] Stephen M. Omohundro. *Five balltree construction algorithms.* International Computer Science Institute Berkeley, 1989.

[54] Dimitris Papadias. Hill climbing algorithms for content-based retrieval of similar configurations. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'2000, pages 240–247, New York, NY, USA, 2000. ACM.

[55] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11):1348–1358, 2010.

[56] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1532–1543, 2014.

[57] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

[58] Adrian Popescu, Eleftherios Spyromitros-Xioufis, Symeon Papadopoulos, Hervé Le Borgne, and Ioannis Kompatsiaris. Toward an automatic evaluation of retrieval performance with large scale image collections. In *Proceedings of the 2015 Workshop on Community-Organized Multimodal Mining: Opportunities for Novel Solutions*, MMCommons'2015, pages 7–12, New York, NY, USA, 2015. ACM.

[59] Gregory Shakhnarovich, Paul Viola, and Trevor Darrell. Fast pose estimation with parameter-sensitive hashing. In *Proceedings of the 9th IEEE International Conference on Computer Vision*, ICCV'2003, page 750, USA, 2003. IEEE Computer Society.

[60] Fumin Shen, Xiang Zhou, Yang Yang, Jingkuan Song, Heng T. Shen, and Dacheng Tao. A fast optimization method for general binary code learning. *IEEE Transactions on Image Processing*, 25(12):5610–5621, Dec 2016.

[61] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[62] Robert F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1-6):579–589, 1991.

[63] Feng Su and Like Xue. Graph learning on k nearest neighbours for automatic image annotation. In *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*, ICMR'2015, pages 403–410, New York, NY, USA, 2015. ACM.

[64] Jinhui Tang, Richang Hong, Shuicheng Yan, Tat-Seng Chua, Guo-Jun Qi, and Ramesh Jain. Image annotation by knn-sparse graph-based label propagation over noisily tagged web images. *ACM Transactions on Intelligent Systems and Technology*, 2(2):14:1–14:15, February 2011.

[65] Javier A. Vargas, Ricardo da S. Torres, and Marcos A. Gonçalves. A soft computing approach for learning to aggregate rankings. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM'2015, page 83–92, New York, NY, USA, 2015. Association for Computing Machinery.

[66] Jianfeng Wang, Jingdong Wang, Nenghai Yu, and Shipeng Li. Order preserving hashing for approximate nearest neighbor search. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM'2013, pages 133–142, New York, NY, USA, 2013. ACM.

[67] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR'2012, pages 1106–1113, June 2012.

[68] Jingdong Wang and Shipeng Li. Query-driven iterated neighborhood graph search for large scale indexing. In *Proceedings of the 20th ACM International Conference on Multimedia*, MM'2012, pages 179–188, New York, NY, USA, 2012. ACM.

[69] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng T. Shen. A survey on learning to hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):769–790, April 2018.

[70] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, 1993.

[71] Ting Zhang, Chao Du, and Jingdong Wang. Composite quantization for approximate nearest neighbor search. In *Proceedings of the 31st International Conference on International Conference on Machine Learning*, ICML'2014, pages II–838–II–846. JMLR.org, 2014.