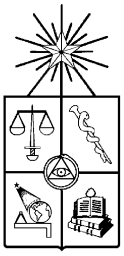


Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Escuela de Postgrado

Graphs for Metric Space Searching

by

Rodrigo Paredes

Submitted to the Universidad de Chile in fulfillment
of the thesis requirement to obtain the degree of
Ph.D. in Computer Science

Advisor	:	**Gonzalo Navarro**
Committee	:	Nancy Hitschfeld
	:	Marcos Kiwi
	:	Patricio Poblete
	:	Peter Sanders (External Professor, Universität Karlsruhe, Germany)

This work has been supported in part by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile & Yahoo! Research Center Latin America

Departamento de Ciencias de la Computación - Universidad de Chile
Santiago - Chile
April 2008

Abstract

من لا يري من الغربال فهو أعمي

[Who doesn't understand a glance, won't understand a long explanation either.]

– Arab proverb

The problem of *Similarity Searching* consists in finding the elements from a set which are similar to a given query under some criterion. If the similarity is expressed by means of a metric, the problem is called *Metric Space Searching*. In this thesis we present new methodologies to solve this problem using graphs $G(V, E)$ to represent the metric database. In G, the set V corresponds to the objects from the metric space and E to a small subset of edges from $V \times V$, whose weights are computed according to the metric of the space under consideration.

In particular, we study *k-nearest neighbor graphs* (kNNGs). The kNNG is a weighted graph connecting each element from V —or equivalently, each object from the metric space— to its k nearest neighbors.

We develop algorithms both to construct kNNGs in general metric spaces, and to use them for proximity searching. These results allow us to use a graph to index a metric space, requiring a moderate amount of memory, with better search performance than that of classical pivot-based algorithms.

Finally, we show that the graph-based approach offers a great potential for improvements, ranging from fully dynamic graph-based indices to optimizations tuned for metric space searching.

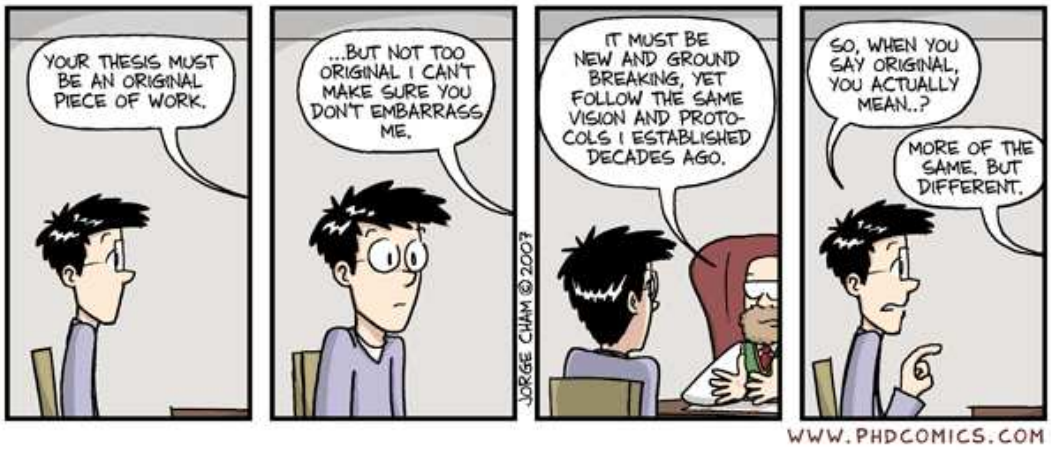
The high amount of computational resources required to build and traverse these graphs also motivated us to research on fundamental algorithmic problems, such as *incremental sorting* and *priority queues*. We propose new algorithms for these problems which, in practice, improve upon the state-of-the-art solutions in many cases, and are useful in many other algorithmic scenarios. As a matter of fact, they yield one of the fastest Minimum Spanning Tree (MST) construction algorithms for random graphs.

These basic algorithms not only open new research lines, for instance, MST construction algorithms for arbitrary graphs; but also they turn out to be appealing to be applied directly in production environments.

Contents

List of Figures

List of Tables



– *Piled Higher & Deeper*, #921, by Jorge Cham

Chapter 1

Introduction

Make your choice, adventurous Stranger;
Strike the bell and bide the danger,
Or wonder, till it drives you mad,
What would have followed if you had.

– *The Magician's Nephew*,
by C. S. Lewis

Searching is a fundamental problem in Computer Science, present in virtually every application. The traditional notion of *exact search* consists in finding an element whose identifier corresponds exactly to a given search key. The natural extension of this notion is *similarity searching*, or *proximity searching*, which aims at finding an element from a database which is close enough to a search key.

Traditional databases are designed and optimized to solve exact queries efficiently by performing comparisons among simple elements, such as numbers or strings. Typical examples are, given a registration number, obtaining the car's information (manufacturer, model, color, year, engine number, current owner, and so on) from the National Registry; or obtaining the definition of a given word from the dictionary. Nowadays, non-traditional applications have given rise to databases containing unstructured data. In these new databases, it is not always possible to define a small, meaningful search key for each database element. In many cases we must use the whole element as the search key, requiring many arithmetical and I/O operations to process a query (think, for instance, in the case where we are looking for similar tunes in a audio database). Moreover, we might want to search the database for elements *similar enough* to a query object, even if the query itself does not belong to the database. This scenario arises in a vast number of applications. Some examples are:

- *Non-traditional databases.* New so-called *multimedia* data types such as images, audio and video, cannot be meaningfully queried in the classical sense. In multimedia

applications, all the queries ask for objects *relevant* —that is, similar— to a given one, whereas comparison for exact equality is very rare. Some example applications are image, audio or video databases, face recognition, fingerprint matching, voice recognition, medical databases, and so on.

- *Text retrieval.* Huge text databases with low quality control have emerged (being the Web the most prominent example), and typing, spelling or OCR (optical character recognition) errors are commonplace in both the text and the queries. Documents which contain a misspelled word are no longer retrievable by a correctly written query or vice versa. Thus, many text search engines aim to find text passages containing close variants of the query words. There exist several models of similarity among words (variants of the "edit distance" [NR02]) which capture very well those kinds of errors. Another related application is spelling checkers, where we look for close variants of a misspelled word in a dictionary.

- *Information retrieval.* Although not considered as a multimedia data type, unstructured text retrieval poses problems similar to multimedia retrieval. This is because textual documents are in general not structured to easily provide the desired information. Although text documents may be searched for strings that are present or not, in many cases it is more useful to search them for semantic concepts of interest. The problem is basically solved by retrieving documents similar to a given query [BYRN99], where the query can be a small set of words or even another document. Some similarity approaches are based on mapping a document to a vector of real values, so that each dimension is a vocabulary word and the relevance of the word to the document (computed using some formula) is the coordinate of the document along that dimension. Similarity functions are then defined on that space. Notice, however, as the vocabulary can be arbitrarily large, the dimensionality of this space is usually very high (thousands of coordinates).

- *Computational biology.* DNA and protein sequences are basic objects of study in molecular biology. They can be modeled as strings (symbol sequences), and in this case many biological quests translate into finding local or global similarities between such sequences, in order to detect homologous regions that permit predicting functionality, structure or evolutionary distance. An exact match is unlikely to occur because of measurement errors, minor differences in genetic streams with similar functionality, and evolution. The measure of similarity used is related to the probability of mutations such as reversals of pieces of the sequences and other rearrangements (global similarity), or variants of edit distance (local similarity).

- There are many other applications, such as *machine learning and classification*, where a new element must be classified according to its closest existing element; *image quantization and compression*, where only some vectors can be represented and those that cannot must be coded as their closest representable point; *function prediction*, where we want to search for the most similar behavior of a function in the past so as to predict its probable future behavior; and so on.

All those applications have some common characteristics, captured under the *metric space model* [CNBYM01, HS03, ZADB06, Sam06]. There is a universe $\mathbb{X}$ of *objects*, and a nonnegative *distance function* d defined among them. The distance function gives us a dissimilarity criterion to compare objects from the database. Thus, the smaller the distance between two objects, the more "similar" they are. This distance satisfies the properties that make $(\mathbb{X}, d)$ a *metric space*, that is, strict positiveness, symmetry, reflexivity and the triangle inequality.

We have a finite *database* $\mathbb{U} \subseteq \mathbb{X}$, of size $n = |\mathbb{U}|$, which is a subset of the universe of objects. Later, given a new object from the universe, a *query* $q \in \mathbb{X}$, we must retrieve similar elements found in the database. There are two typical similarity queries: (i) *Range query* (q, r), which retrieves all elements that are within distance r to q; and (ii) *k-Nearest neighbor query* $NN_k(q)$, which retrieves the k elements from $\mathbb{U}$ that are closest to q.

The distance is assumed to be expensive to compute (as is the case in most of the applications mentioned). Hence, it is customary to define the complexity of the search as the number of distance evaluations performed.

Note that, if we can translate $\mathbb{X}$ to a 1-dimensional space still preserving the distances among objects, then there is a simple way to solve the problem. A 1-dimensional space, for instance $\mathbb{R}$, admits total order, thus, it is enough to index objects in $\mathbb{U}$ by sorting them. Later, given a query $q \in \mathbb{X}$, in this case $q \in \mathbb{R}$, we use binary searching to retrieve the object u_q in $\mathbb{U}$ closest to q. Finally, we perform a sequential scanning around u_q until fulfilling the query. Else, if it is possible to translate $\mathbb{X}$ to a vector space of low dimensionality $D \in [1, 8]$, we can take advantage of the coordinates by using spatial search methods such as *kd*-trees [Ben75, Ben79], *R*-trees [Gut84], quad-trees [Sam84] and *X*-trees [BKK96]. For further references to these techniques see [Sam84, WJ96, GG98, BBK01, Sam06]. Otherwise, for medium and higher dimensionality ($D \leq 8$), or when it is not possible to map $\mathbb{X}$ to a vector space, we are left with the metric space search approach, where we only have a distance function to compare the objects.

Given the metric database $\mathbb{U}$, proximity queries can be trivially answered by performing n evaluations of the distance function. However, this mechanism is unacceptable in most applications, as the distance function is usually expensive to compute. Therefore, metric space search algorithms are devoted to minimizing the number of computations of the distance function when solving a similarity query.

A natural methodology to face the problem consists in building *offline* an *index* $\mathcal{I}$ so as to improve the performance of *online* queries. Current solutions consider two kinds of indices [CNBYM01]: pivot based and compact-partition based. As we can only compare objects by computing the distance between them, the index can be seen, in abstract terms, as a subset of cells from the full distance matrix $\mathbb{U} \times \mathbb{U}$, or, more generally, as somehow summarizing the distance information of cells from $\mathbb{U} \times \mathbb{U}$. Then, given a query object, we can use the index and the triangle inequality in order to avoid some distance computations to solve the proximity query.

In this thesis we present new methodologies to solve the metric space search problem using graphs $G(V, E)$ as the representation of the metric database $\mathbb{U}$. In G, each vertex of V represents an object of $\mathbb{U}$, and the edge set E corresponds to a small subset of weighted edges from the full set $V \times V$. That is, E is a subset from $\mathbb{U} \times \mathbb{U}$, and the edge weights are the distances between the connected nodes according to function d. We emphasize that, for practical reasons concerning memory and CPU time, we will use graphs with small edge sets.

The earliest work in graphs for metric space searching we are aware of used arbitrary graphs [SW90], and the results were not promising. Much better results were achieved by designing the graphs so that they ensure some desiderable properties: our MSc Thesis explored the use of t-spanners for metric space searching, with very encouraging results [Par02, NP03, NPC02, NPC07]. Still, however, the development on using a graph as the representation of the metric database is under-explored and has not reached its full potential. This is the main subject of this thesis.

We exhaustively explore *k-nearest neighbor graphs* (kNNGs). The kNNG is a directed weighted graph connecting each element to its k nearest neighbors. We develop algorithms both to construct kNNGs and to use them to solve proximity queries. kNNGs offer an indexing alternative which requires a moderately amount of memory ($O(kn)$ space) obtaining reasonably good performance in the search process. In fact, in low-memory scenarios, which only allow small values of k (that is, considering few neighbors per element), the search performance of kNNG-based proximity query algorithms in real-world metric spaces is better than using the classical pivot-based indexing alternative.

The performance of kNNG-based proximity query algorithms improves as we have more space to index the dataset, which means considering more and more neighbors in the graph. However, there is an optimal value k^* of neighbors per node which offers the best search performance. Nevertheless, if we have more space we can still use graph-based metric space indices: The t-spanner-based approach always improves the search performance as we allow it to use more memory in the index [NPC02, NPC07].

Graph algorithms make heavy use of fundamental algorithms and data structures, and we have faced several basic algorithmic challenges throughout this work. Apart from the cost of computing distances, further so-called side computations cannot be neglected in graph-based approaches. It is important to keep the extra CPU time under control, for instance, when computing shortest paths over a graph index. Therefore, we also researched on fundamental algorithmic problems, such as *incremental sorting* and *priority queues*.

These basic algorithms help us to improve the CPU performance of side computations of our graph-based approach. On the other hand, these basic algorithms also improve upon the current state of the art on many other algorithmic scenarios. For instance, we plug our basic algorithms in the classic Minimum Spanning Tree (MST) techniques [Wei99, CLRS01], obtaining two solutions that are competitive with the best (and much more sophisticated) current implementations: We use the incremental sorting technique to boost Kruskal's MST algorithm [Kru56], and the priority queue to boost Prim's MST algorithm [Pri57].

1.1 Contributions of the Thesis

> Never follow the beaten track, it leads only where others have been before.
>
> – Alexander Graham Bell

As we said above, this thesis is devoted to the development of metric space search techniques based on graphs. We have been concerned not only about reducing the number of distance computations, but also in keeping under control the CPU time of side computations when using a graph as the underlying index structure. Our specific contributions are as follows:

1. Design, implementation and experimental evaluation of kNNG construction algorithms. This has already been published in the *5th Workshop of Experimental Algorithms* (*WEA'06*) [PCFN06].
2. Design, implementation and experimental evaluation of kNNG-based algorithms to solve proximity queries. This has already been published in the *12th International conference on String Processing and Information Retrieval* (*SPIRE'05*) [PC05].
3. Design, implementation and experimental evaluation of a new incremental sorting algorithm, coined Incremental Quicksort (IQS). This has already been published in the *8th Workshop on Algorithm Engineering and Experiments* (*ALENEX'06*) [PN06].
4. Design, implementation and experimental evaluation of a priority queue for both main and secondary memory based on our incremental sorting algorithm, coined Quickheap.
5. Design, implementation and experimental evaluation of MST construction algorithms based on IQS and Quickheaps. Part of this research has been published in [PN06].

Our contributions widen the spectrum of tools for metric space searching, approaching the problem from a novel and successful point of view consisting in using kNNGs to speed up metric queries. For this sake, we give not only kNNG constructing algorithms, but also several search algorithms that exploit both the metric properties and the features of the kNNG index.

We also contribute with fundamental algorithmic tools. As a matter of fact, in this work we give several simple, efficient and general-purpose algorithms and data structures for priority queues and incremental sorting. For instance, using these ideas we have obtained two MST construction algorithms which are competitive with the state of the art. Indeed, on random graphs our MST algorithms display the best performance we are aware of.

1.2 Thesis Organization

> La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever. [Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.]
>
> – Antoine de Saint-Exupéry

This document is divided into five chapters. A summary of the content of the rest of chapters follows.

In chapter *Basic Concepts*, we introduce the background necessary to follow the thesis and a brief survey of related work that is relevant to our approach. We develop four content lines. The first is devoted to basic algorithmic issues, the second to concepts and tools from graph theory, the third to metric spaces, and the fourth to the relation between metric spaces and graphs.

In chapter *Fundamental Algorithms*, we introduce a novel incremental sorting technique and its applications. Among them, a new priority queue implementation is obtained, as well as algorithms to build graph MSTs.

In chapter *k-Nearest Neighbor Graphs*, we present a general methodology to construct kNNGs, and to use them for proximity searching. We develop two concrete kNNG construction algorithms following our general methodology.

In chapter *Conclusions*, we review our results with a more global perspective. We finish with some directions for further work.

Finally, the bibliography includes over 100 references to relevant publications.

Chapter 2

Basic Concepts

"The time has come," the Walrus said,
"To talk of many things:
Of shoes – and ships – and sealing-wax –
Of cabbages – and kings –
And why the sea is boiling hot –
And whether pigs have wings."

– L. Carroll

In this chapter we give the background needed to read this thesis. We start in Section 2.1 by reviewing fundamental algorithmic aspects. We continue in Section 2.2 by introducing some definitions, data structures and algorithms from graph theory. Next, in Section 2.3, we describe the metric space model and current solutions for the metric space search problem. Finally, in Section 2.4, we explain how to use graph concepts in metric space searching, and the related work on this subject.

2.1 Basic Algorithms

Classic: a book which people praise and don't read.

– Mark Twain

In this section we review some basic algorithmic concepts used throughout the thesis. These consider algorithms for sorting, incremental sorting, priority queues and the potential method for amortized analysis.

2.1.1 Quicksort and Quickselect

Quicksort [Hoa62] —one the most famous computer algorithms— is an in-place randomized sorting method with worst-case time $O(m^2)$, when working over an input array of m numbers. Nevertheless, its expected complexity is $O(m \log m)$ (more precisely, it performs $2(m+1)H_m - 4m$ element comparisons [GBY91], where H_m corresponds to the m-th harmonic number $H_m = \sum_{i=1}^{m} \frac{1}{i} = \ln n + \gamma + O(1/m)$). In practice Quicksort is the most efficient algorithm to sort a given set in main memory.

Quicksort works as follows. Given an array A of m numbers and the indices $first$ and $last$ bounding the array segment where it will work, Quicksort chooses a position $pidx$ at random between $first$ and $last$, reads $p = A[pidx]$, and partitions $A[first \ldots last]$ so that elements smaller than p are allocated to the left-side partition, and the others to the right side. After the partitioning, p is placed in its correct position $pidx'$. Next, Quicksort performs recursive calls on the left and right partitions, that is, in segments $A[first \ldots pidx' - 1]$ and $A[pidx' + 1 \ldots last]$. The recursion stops when $first \geq last$. Figure 2.1 shows the algorithm. The first call is **Quicksort**$(A, 0, m-1)$.

Quicksort (Array A, Index $first$, Index $last$)
1. **If** $first \geq last$ **Then Return**
2. $pidx \leftarrow$ **random**$(first, last)$
3. $pidx' \leftarrow$ **partition**$(A, pidx, first, last)$
4. **Quicksort**$(A, first, pidx' - 1)$
5. **Quicksort**$(A, pidx' + 1, last)$

Figure 2.1: Quicksort algorithm. A is modified during the algorithm. Procedure **partition** returns the resulting position $pidx'$ of pivot $p = A[pidx]$ after the partition completes. Note that the tail recursion can be easily removed.

Based on Quicksort, it is easy to obtain a linear expected-time selection algorithm. Given an integer k, a selection algorithm returns the k-th smallest element from an unsorted array A. The selection algorithm based on Quicksort is known as *Quickselect* [Hoa61]. It also has worst-case time $O(m^2)$, but good (and linear-time) performance in practice.

Quickselect works as follows. Assume we have a set A of m numbers and an integer $k \geq 0$. Once again, let $first$ and $last$ be the bounds of the array segment where Quickselect will operate. Quickselect starts by choosing a position $pidx$ at random between $first$ and $last$, and partitions $A[first \ldots last]$ just like Quicksort, placing the chosen pivot p at its correct position $pidx'$. Then, if $pidx'$ is equal to k, Quickselect returns $p = A[pidx']$ and finishes. Otherwise, if $k < pidx'$ it recursively processes the left partition $A[first \ldots pidx' - 1]$, else the right partition $A[pidx' + 1 \ldots last]$. In the second recursive call it looks for the $(k - pidx' - 1)$-th element of array $A[pidx' + 1 \ldots last]$. Figure 2.2 shows the algorithm. The first call is **Quickselect**$(A, k, 0, m-1)$.

Quickselect (Array A, Index k, Index $first$, Index $last$)
1. **If** $first > last$ **Then Return** NULL // $k > m$
2. $pidx \leftarrow$ **random**$(first, last)$
3. $pidx' \leftarrow$ **partition**$(A, pidx, first, last)$
4. **If** $pidx' = k$ **Then Return** $A[pidx']$
5. **Else If** $pidx' < k$ **Then Return Quickselect**$(A, k, first, pidx' - 1)$
6. **Else Return Quickselect**$(A, k, pidx' + 1, last)$

Figure 2.2: Quickselect. A is modified during the algorithm. Procedure **partition** returns the resulting position $pidx'$ of the pivot $p = A[pidx]$ after the partition completes. Note that the recursion can be easily removed.

2.1.2 Incremental Sorting

There are cases where we need to obtain the smallest elements from a fixed set without knowing how many elements we will end up needing. Prominent examples are Kruskal's Minimum Spanning Tree (MST) algorithm [Kru56] and ranking by Web search engines [BYRN99]. Given a graph, Kruskal's MST algorithm processes the edges one by one, from smallest to largest, until it forms the MST. At this point, remaining edges are not considered. Web search engines display a very small sorted subset of the most relevant documents among all those satisfying the query. Later, if the user wants more results, the search engine displays the next group of most relevant documents, and so on. In both cases, we could sort the whole set and later return the desired objects, but obviously this is more work than necessary.

This problem can be called *Incremental Sorting*. It can be stated as follows: Given a set A of m numbers, output the elements of A from smallest to largest, so that the process can be stopped after k elements have been output, for any k that is unknown to the algorithm. Therefore, *Incremental Sorting* is the online version of a variant of the *Partial Sorting* problem: Given a set A of m numbers and an integer $k \leq m$, output the smallest k elements of A in ascending order.

2.1.2.1 Related Work on Incremental Sorting

In 1971, J. Chambers introduced the general notion of Partial Sorting [Cha71]: Given an array A of m numbers, and a fixed, sorted set of indices $I = i_0 < i_1 < \ldots < i_{k-1}$ of size $k \leq m$, arrange in-place the elements of A so that $A[0, i_0 - 1] \leq A[i_0] \leq A[i_0 + 1, i_1 - 1] \leq A[i_1] \leq \ldots \leq A[i_{k-2} + 1, i_{k-1} - 1] \leq A[i_{k-1}] \leq A[i_{k-1} + 1, m - 1]$. This property is equivalent to the statement that $A[i]$ is the i-th order statistic of A for all $i \in I$.

We are interested in the particular case of finding the first k order statistics of a given set A of size $m > k$, that is, $i_j = j$. This can be easily solved by first finding the k-th smallest element of A using $O(m)$ time *Select* algorithm [BFP+73], and then collecting

and sorting the elements smaller than the k-th element. The resulting complexity, $O(m + k \log k)$, is optimal under the comparison model, as there are $m^{\underline{k}} = m!/(m-k)!$ possible answers and $\log\left(m^{\underline{k}}\right) = \Omega(m + k \log k)$.

A practical version of the above method uses Quickselect and Quicksort as the selection and sorting algorithms, obtaining $O(m + k \log k)$ expected complexity. Recently, it has been shown that the selection and sorting steps can be interleaved. The result has the same average complexity but smaller constant terms [Mar04].

To solve the online problem (incremental sort), we must select the smallest element, then the second smallest, and so on until the process finishes at some unknown value $k \in [0, m-1]$. One can do this by using Select to find each of the first k elements, for an overall cost of $O(km)$. This complexity can be improved by transforming A into a min-heap [Wil64] in time $O(m)$ [Flo64] (see Section 2.1.3.1), and then performing k extractions. This premature cut-off of the heapsort algorithm [Wil64] has $O(m + k \log m)$ worst-case complexity. Note that $m + k \log m = O(m + k \log k)$, as they can differ only if $k = o\left(m^c\right)$ for any $c > 0$, and then m dominates $k \log m$. However, according to experiments this scheme is much slower than the offline practical algorithm [Mar04] if a classical heap is used.

P. Sanders [San00] proposes *sequence heaps*, a cache-aware priority queue, to solve the online problem. Sequence heaps are optimized to insert and extract *all the elements* in the priority queue at a small amortized cost. Even though the total CPU time used for this algorithm in the whole process of inserting and extracting all the m elements is pretty close to the time of running Quicksort, this scheme is not so efficient when we want to sort just a small fraction of the set.

2.1.3 Priority Queues

A *priority queue* is a data structure which allows maintaining a set of elements in a partially ordered way, enabling efficient insertions and extractions of particular elements as well as obtaining the minimum (or alternatively the maximum) of the set. For simplicity we focus on obtaining minima and suppose that elements are of the form $(key, item)$, where the keys can be ordered. The basic priority queue supports the following operations:

- **insert**$(key, item)$: inserts element $(key, item)$ in the queue.
- **findMin**(): returns the element of the queue with the lowest key value.
- **extractMin**(): removes and returns the element of the queue with the lowest key value.

The set of operations can be extended to construct a priority queue from a given array A (**heapify**), increment or decrement a key (**increaseKey** and **decreaseKey**, respectively), answer whether an arbitrary element belongs to the element set (**find**),

delete an arbitrary element from the priority queue (**delete**), retrieve the successor or predecessor of a given *key* (**successor** and **predecessor**, respectively), merge priority queues (**merge**), and a long so on.

2.1.3.1 Binary Heaps

The classic implementation of a priority queue uses a binary heap [Wil64], which is an array A satisfying some properties. A can be viewed as a binary tree with all its levels complete except the bottom level, which can be incomplete. The bottom level is filled from the leftmost to rightmost leaf. All the elements inside the binary heap are stored contiguously and levelwise, starting from cell $A[1]$, which is the root of the tree. Figure 2.3 illustrates. Using the array representation, it is easy to obtain the left and right children of a given node with index x: its left child is at cell $2x$, and its right child at cell $2x + 1$. Naturally, x's parent is at cell $\lfloor \frac{x}{2} \rfloor$.

In min-order binary heaps, any parent is smaller than or equal to both of its children. This implies that the smallest element in a min-order binary heap is stored at the root of the tree, and recursively, the subtree rooted by any node contains values not smaller than that of the node itself. Figure 2.3 shows an example of a binary heap in min-order.

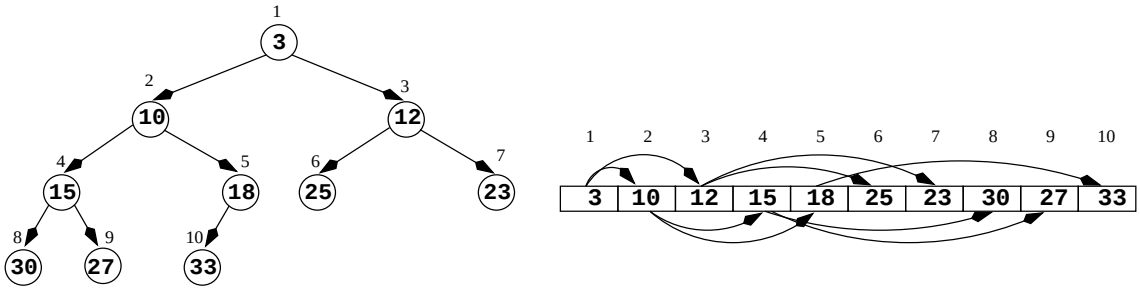


Figure 2.3: Binary heap example.

In the following we explain several min-order binary heap operations; all of them are depicted in Figure 2.4. To implement a min-order binary heap we need an array A to store the elements. We also use the auxiliary variable *last* to indicate at which cell is placed the last element of the heap (the rightmost of the bottom level). Since we are considering min-order, we initialize $A[0]$ with a fake element $(-\infty, \text{NULL})$. This is useful to simplify the following procedures. Note that true elements are stored starting from cell 1. As there are no elements, *last* is initialized to 0.

To insert a new element *elem* we start by incrementing $last \leftarrow last + 1$, and placing *elem* at cell $A[last]$. Next we restore the min-order invariant. For this sake, we need an auxiliary variable *aux* initialized at *last*. We compare the element at cell *aux* with its parent $\lfloor \frac{aux}{2} \rfloor$. If the key at the parent is greater than that of $A[aux]$, we swap the cell contents and repeat the procedure from $aux \leftarrow \lfloor \frac{aux}{2} \rfloor$ towards the root until we restore the invariant. The fake value $A[0] = (-\infty, \text{NULL})$ avoids checking border conditions explicitly.

As objects within the heap already satisfy the min-order property, obtaining the minimum of the binary heap (**findMin**) is as simple as returning $A[1]$.

To extract the minimum of the heap, we store the element in $A[1]$ in an auxiliary variable, pick the rightmost element of the heap ($A[last]$), place it in cell $A[1]$, and restore the min-order invariant by percolating down the element in $A[1]$ until it gets a place in fulfillment with the min-order invariant. Finally, we return the element stored in the auxiliary variable. Procedure **percolateDown** compares the element at position *pos* with the smaller of its children (at positions $2 \cdot pos$ and $2 \cdot pos + 1$). If the smaller child is smaller than the key at $A[pos]$, the elements are swapped and the procedure continues from the position of the child, until the min-order property is fulfilled.

In order to heapify a given array L of m elements, we can insert one-by-one its elements into a binary heap. Thus, for each inserted element, we restore the invariant by moving it upwards. This is an $O(m \log m)$ worst-case alternative to heapify the array. However, Floyd gives a linear-time algorithm to heapify an array [Flo64]. The idea is to traverse the array right to left from the position $\lfloor \frac{aux}{2} \rfloor$ (that is, traverse the heap levelwise bottom to top), and at each position use **percolateDown** to restitute the invariant downwards the binary heap (where the min-order property already holds). The sum of all those costs is just $O(m)$.

2.1.3.2 Related Work on Priority Queues

Wegener [Weg93] proposes a *bottom-up deletion* algorithm, which addresses operation **extractMin** performing only $\log_2 m + O(1)$ key comparisons per extraction on average, in heaps of m elements. Wegener's algorithm saves up to half of the comparisons used by a straightforward implementation taken from textbooks (see previous section) [CLRS01, Wei99]. It works as follows. When the minimum is extracted, it lifts up elements on a min-path from the root to a leaf in the bottom level. Then, it places the rightmost element (the last of the heap) into the free leaf, and bubbles it up to restore the min-heap condition.

We have also mentioned P. Sanders' *sequence heaps* [San00], a cache-aware priority queue tuned to efficiently solve operations **insert** and **extractMin**. Other well-known related works on priority queues are *binomial queues* [Vui78], *Fibonacci heaps* [FT87], *leftist heaps* [Cra72, Knu98], *min-max heaps* [ASSS86], *pairing heaps* [FSST86], *skew heaps* [ST86], and *van Emde Boas queues* [vEBKZ77]. They are summarized in Table 2.1.

Finally, a practical, interesting application of priority queues arises when there are many **decreaseKey** operations for each **extractMin** operation. We have this case in Dijkstra's Shortest Path algorithm [Dij59] or Prim's Minimum Spanning Tree algorithm [Pri57] (we explain these algorithms in detail in the next section). In this application, it seems that implementations of pairing heaps yield the best performance as suggested in [MS91, KST03].

BinaryHeap(Integer N)
// constructor of an empty binary heap, $elem$ is of the form $(key, item)$
1. $A \leftarrow$ new Array$[N+1]$, $last \leftarrow 0$, $A[0] \leftarrow (-\infty, \text{NULL})$

---

insert(Elem $elem$)
1. $last \leftarrow last + 1$, $aux \leftarrow last$, $parent \leftarrow \lfloor \frac{aux}{2} \rfloor$
2. **While** $elem.key < A[parent].key$ **Do**
3. $A[aux] \leftarrow A[parent]$, $aux \leftarrow parent$, $parent \leftarrow \lfloor \frac{aux}{2} \rfloor$
4. $A[aux] \leftarrow elem$

---

findMin()
1. **If** $last = 0$ **Then Return** NULL // there are no elements in the binary heap
2. **Return** $A[1]$

---

extractMin()
1. **If** $last = 0$ **Then Return** NULL // there are no elements in the binary heap
2. $elem \leftarrow A[1]$, $A[1] \leftarrow A[last]$, $last \leftarrow last - 1$
3. **percolateDown**(1)
4. **Return** $elem$

---

percolateDown(Position pos)
1. $aux \leftarrow A[pos]$
2. **While** TRUE **Do**
3. $leftChild \leftarrow 2 \cdot pos$, $rightChild \leftarrow 2 \cdot pos + 1$
4. **If** $leftChild > last$ **Then Break**
5. **Else If** $(leftChild = last)$ OR $(A[leftChild].key < A[rightChild].key)$ **Then**
6. $smaller \leftarrow leftChild$
7. **Else** $smaller \leftarrow rightChild$
8. **If** $A[smaller].key < aux.key$ **Then**
9. $A[pos] \leftarrow A[smaller]$, $pos \leftarrow smaller$
10. **Else Break**
11. $A[pos] \leftarrow aux$

---

heapify(Array L)
1. **If** $|L| = 0$ **Then Return** // there are no elements in the array
2. **For** $i \leftarrow 0, \ldots, |L| - 1$ **Do** $A[last + i + 1] \leftarrow L[i]$ // copying L into A
3. $last \leftarrow last + |L|$
4. **For** $i \leftarrow \lfloor \frac{last}{2} \rfloor$ **downto** 1 **Do**
5. **percolateDown**(i)

Figure 2.4: Basic binary heap operations. Note that we heapify array L into a previously initialized binary heap (which either is empty, $last = 0$, or has elements, $last > 0$). We also need that $|L| + last < N$, else the binary heap is overflowed.

heap	**insert**	**findMin**	**extractMin**	**merge**	**decreaseKey**	**increaseKey**
binary (∗)	$O(\log m)$	$O(1)$	$O(\log m)$	$O(m)$	$O(\log m)$	$O(\log m)$
binomial (∘)	$O(\log m)$	$O(\log m)$	$O(\log m)$	$O(\log m)$	$O(\log m)$	$O(\log m)$
Fibonacci	$O(1)$	$O(1)$	$O(\log m)$ (⋆)	$O(1)$	$O(1)$	
leftist	$O(\log m)$	$O(1)$	$O(\log m)$	$O(\log m)$	$O(\log m)$	$O(\log m)$
min-max (†)	$O(\log m)$	$O(1)$	$O(\log m)$	$O(m)$	$O(\log m)$	$O(\log m)$
pairing (‡)	$O(1)$ (⋆)	$O(1)$	$O(\log m)$ (⋆)	$O(1)$ (⋆)	$O(1)$ (⋆)	
skew	$O(\log m)$ (⋆)	$O(1)$	$O(\log m)$ (⋆)	$O(\log m)$ (⋆)	$O(\log m)$ (⋆)	$O(\log m)$ (⋆)
vEB (•)	$O(\log\log C)$	$O(\log\log C)$	$O(\log\log C)$		$O(\log\log C)$	$O(\log\log C)$

Table 2.1: Operation costs of some priority queues satisfying the min-order property. We also mention some other nice features of some priority queues. All the times are worst case unless we specify otherwise. All the heaps in the table can be created as an empty heap in $O(1)$ time. (∗) Binary heaps allow **heapify** in $O(m)$. (∘) Binomial heaps allow **insert** in $O(1)$ time on average. (⋆) Amortized time. (†) Min-max heaps also implement **findMax** in time $O(1)$, **extractMax** in time $O(\log m)$, and **heapify** in time $O(m)$. (‡) For pairing heaps, it is conjectured that operations **insert**, **decreaseKey** and **merge** have $O(1)$ amortized complexity but this has not yet been proven. (•) The van Emde Boas heaps also support operations **findMax**, **extractMax**, **delete**, **find**, **predecessor** and **successor** in worst-case time $O(\log\log C)$, and are subject to the restriction that the universe of keys is the set $\{1,\ldots,C\}$.

2.1.4 External Memory Priority Queues

The priority queues mentioned in Section 2.1.3.2 assume that the whole element set fits in main memory. Unfortunately, as soon as the size of the element set overreaches the available main memory, the performance of those *internal-memory* priority queues dramatically slows down due to paging in virtual memory. In fact, if the priority queue does not fit in the internal (or main) memory of the computer, necessarily some portions of the queue must be maintained in the computer *external-memory* (for instance, on the hard disk). Then, if the accesses to elements within the queue are random and do not exhibit any locality of reference, we will observe a heavy paging activity, which translates into a sharp decrease in the I/O-performance. In practice, considering the speed ratio between secondary and main memory, the paging activity can generate a slowdown factor of around 10^5 in the performance of the priority queue operations.

Nowadays, there are several ranking applications that must handle huge datasets, being Web search engines [BYRN99] a well-known example. Hence, the design of external memory priority queues that achieve efficient I/O-performance —which implies that, even though the data are retrieved from the hard disk, the overall performance of priority queue operations remains reasonably good— is an extremely appealing problem.

Many classical data structures have been adapted to work efficiently on secondary memory [Vit01], and priority queues are not an exception. However, in order to guarantee good locality of reference, external memory priority queues usually offer just the basic operations, namely, **insert**, **findMin** and **extractMin**. This is because others, like **find** or **decreaseKey** need at least one random access to the queue.

When working in the secondary memory scenario, we assume that we have M bytes of fast-access internal memory and an arbitrary large slow-access external memory located in one or more independent disks. Data between the internal memory and the disks is transfered in blocks of size B, called *disk pages*.

In this model, the algorithmic performance is measured by counting the number of disk access performed, which we call I/Os. Some authors also suggest to measure the internal CPU time, or account for the number of occupied pages. However, the dominant term is the number of I/Os performed.

2.1.4.1 Related Work on External Memory Priority Queues

Some examples of external memory priority queues are *buffer trees* [Arg95, HMSV97], *M/B-ary heaps* [KS96, FJKT99], and *Array Heaps* [BK98], all of which achieve the lower bound of $\Theta((1/B)\log_{M/B}(m/B))$ amortized I/Os per operation [Vit01]. Those structures, however, are rather complex to implement and heavyweight in practice (in extra space and time) [BCFM00]. Other techniques are simple but do not perform so well (in theory or in practice), for example those using B-trees [BM72].

A practical comparison of existing secondary memory priority queues was carried out by Brengel et al. [BCFM00], where in addition they adapt two-level radix heaps [AMOT90] to secondary memory (*R-Heaps*), and also simplify *Array-Heaps* [BK98]. The latter stays optimal in the amortized sense and becomes simple to implement. The experiments in [BCFM00] show that *R-Heaps* and *Array-Heaps* are by far the best choices for secondary memory. In the same issue, Sanders introduced *sequence heaps* [San00], which can be seen as a simplification of the improved *Array-Heaps* of [BCFM00]. Sanders reports that *sequence heaps* are faster than the improved *Array-Heaps*, yet the experiments only consider caching in main memory.

2.1.5 The Potential Method for Amortized Analysis

In Section 3.4 we use a variation of the *potential method* ([Tar85] and [CLRS01, Chapter 17]). In this section we describe the standard potential method, which is a technique used in amortized analysis.

In an amortized analysis, the time required to perform a sequence of operations over a data structure is shared among all the operations performed. The idea is to show that the amortized cost of an operation is small if one considers a sequence of operations, even though a single operation within the sequence might be expensive.

The common techniques for amortized analysis are (i) aggregate analysis, (ii) the accounting method, and (iii) the potential method. In the first, the idea is to determine an upper bound $T(n)$ for a sequence of n operations, and then compute the amortized cost of each operation as $\frac{T(n)}{n}$. In this case all the operations are assigned the same cost.

In the second, the total cost of the process is distributed among operations and data structure objects. The total cost is obtained by summing up all the distributed costs.

In the potential method, the idea is to determine an amortized cost for each operation type. The potential method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on the data structure. The sum of all the prepaid credit is called the *potential* of the data structure. The potential is used later in the sequence to pay for operations that are charged less than what they actually cost.

The potential method works as follows. It starts with an initial data structure D_0 on which operations are performed. Let c_i be the actual cost of the i-th operation and D_i the data structure that results after applying the i-th operation to D_{i-1}. A *potential function* Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i. The *amortized cost* $\widehat{c_i}$ of the i-th operation with respect to potential function Φ is defined by

$$\widehat{c_i} \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;. \tag{2.1}$$

Therefore, the amortized cost of the i-th operation is the actual cost plus the potential variation due to the operation. Thus, the total amortized cost for n operations is

$$\sum_{i=1}^{n} \widehat{c_i} \;=\; \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \;=\; \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \;. \tag{2.2}$$

If we define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, for all n, then the total amortized cost $\sum_{i=1}^{n} \widehat{c_i}$ is an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the i-th operation is positive, then the amortized cost $\widehat{c_i}$ represents an overcharge to the i-th operation, and the potential of the data structure increases. On the other hand, if the potential difference is negative, then the amortized cost represents an undercharge to the i-th operation, and the actual cost of the operation is paid by the decrease in the potential.

2.2 Graph Tools and Definitions

> I married a widow who had a grown up daughter. My father visited our house very often, fell in love with my stepdaughter, and married her. So my father became my son-in-law and my stepdaughter my mother because she was my father's wife. Sometime afterward my wife had a son; he was my father's brother-in-law and my uncle, for he was the brother of my stepmother. My father's wife —i.e., my stepdaughter— had also a son; he was, of course, my brother, and in the meantime my grandchild, for he was the son of my daughter. My wife was my grandmother, because she was my mother's mother. I was my wife's husband and grand child at the same time. And as the husband of a person's grandmother is his grandfather, I was my own grandfather.
>
> – Unknown author

The concept of a graph is present in a vast number of fields in Computer Science. The algorithms and data structures involved in graph theory are fundamental for this discipline, and they have been described in general textbooks [CLRS01, HS76, Wei99], and also in specialized ones, for example [Bol98, Wes01]. Nevertheless, graphs are an extremely active research field, both on development of theory and on applications in Computer Science and other branches in Science. In the following sections we give a glance over graph theory. For further information we encourage the reader to check the references.

2.2.1 Basic Definitions

A graph G consists of a set of *vertices* V and a set of *edges* E, and we usually write it as $G(V, E)$. For simplicity, we use natural numbers (including zero) as vertex identifiers. The vertices are also called *nodes*. Each edge is a pair (u, v), where $u, v \in V$. In an *undirected graph* the pair of vertices representing any edge is unordered, so (u, v) and (v, u) represent the same edge. Instead, in a *directed graph* the pair is ordered, so (u, v) and (v, u) represent two different edges. In this thesis we do not consider edges of the form (u, u).

The vertex v is *adjacent* to u, or a *neighbor* of u, if and only if $(u, v) \in E$. So, in undirected graphs, given an edge (u, v), v is adjacent to u, and symmetrically u is adjacent to v. For the same example, in directed graphs, we only have that v is adjacent to u. In many cases, edges also have a third component, called a *weight* or *cost*. In such cases we speak of *weighted graphs*. In this thesis we only consider nonnegative costs. For simplicity, in order to refer the weight of the edge $e = (u, v)$ we write $weight_e$ or $weight_{u,v}$. For technical reasons, we define the weight $weight_{u,u}$ as zero.

A *path* $u_1 \rightsquigarrow u_l$ in a graph is a vertex sequence $u_1, u_2, \ldots, u_l$ such that $(u_i, u_{i+1}) \in E$, for $1 \leq i < l$. In non-weighted graphs, the *length* of such path is the number of edges composing it, that is, $l - 1$. On the other hand, in weighted graphs, the *length* of a path is the sum of the costs of the traversed edges, that is, $\sum_{1 \leq i < l} weight_{u_i, u_{i+1}}$. We define the length of a path $u \rightsquigarrow u$ as zero.

A *cycle* is a path with no repeated nodes except the first and last ones, which must be the same.

In an undirected graph G, two vertices u and v are said to be *connected* if there is a path in G from u to v (as G is undirected, there also is a path from v to u). A graph is called connected if for every pair of vertices u and v there exists a path connecting them. A graph is *full* if for all pairs $u, v \in V$ there exists edge $(u, v) \in E$.

It is customary to denote by n the size of V, and m the size of E. Note that in this document we also use n as the size of $\mathbb{U}$, but as we explain in Section 2.4, every node of V will represent a single object in $\mathbb{U}$, so there is no possible confusion.

The *adjacency* of a node $u \in V$ is the set of its neighbors, formally $adjacency(u) = \{v \in V, (u, v) \in E\}$. Note that the adjacency of a node u does not contain the node u itself. Analogously, the adjacency of a node set $U \subseteq V$, is the set of all the neighbors of U, formally $adjacency(U) = \{v \in V, \exists\ u \in U, (u, v) \in E\} - U$.

2.2.2 Graph Representations

Let us consider the undirected graph of Figure 2.5, composed by 8 vertices and 13 edges where nodes are numbered starting from 0.

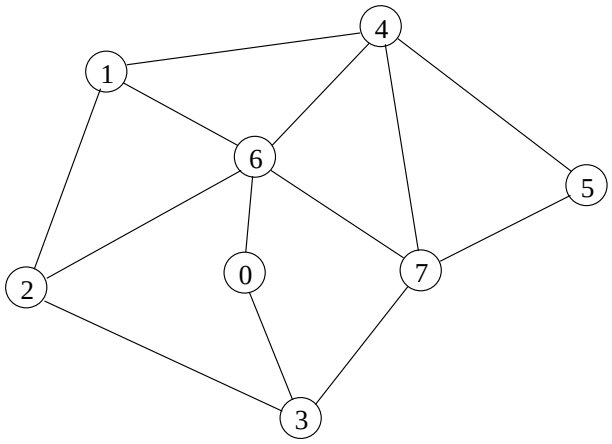


Figure 2.5: Undirected weighted graph. The weights are indicated in the adjacency matrix of Table 2.2 and also in the adjacency list of Figure 2.6.

A simple way to represent a graph is by using a square matrix A, called *adjacency matrix*. In non-weighted graphs, for each pair (u, v) we set $A_{u,v} = 1$ if $(u, v) \in E$, otherwise

we set $A_{u,v} = 0$. In weighted graphs, for each pair (u, v) we set $A_{u,v} = weight_{u,v}$ if $(u, v) \in E$, otherwise we set $A_{u,v} = \infty$. When the graph is undirected, the adjacency matrix is symmetric, so it is enough to store the upper triangle matrix. This representation is extremely simple, requires $\Theta(n^2)$ space and it is convenient for graphs with more than $\frac{n(n-1)}{2}$ edges (or $\frac{n(n-1)}{4}$ edges in the case of undirected graphs). Table 2.2 shows the adjacency matrix of the undirected weighted graph of Figure 2.5.

	0	1	2	3	4	5	6	7
0		∞	∞	4.7	∞	∞	2.9	∞
1			7.5	∞	6.2	∞	4	∞
2				10	∞	∞	7.5	∞
3					∞	∞	∞	4
4						8.5	5.2	8.3
5							∞	5.5
6								6
7								

Table 2.2: Adjacency matrix of the graph of Figure 2.5.

For sparse graphs (with a small set of edges, for instance, when $|E| = n^{1.5}$), the matrix representation is inappropriate as it allocates space for all the edges, which is an excessive memory requirement. For sparse graphs it is recommended to use an *adjacency list*, which consists in storing the list $adjacency(u)$, for each vertex $u \in V$. This way, the space requirement is $\Theta(n + m)$. In weighted graphs, for each neighbor we also store the edge weight. In the case of undirected graphs, as $\forall\ (u, v) \in E \Rightarrow (v, u) \in E$, for each edge we must also store its symmetric version. Figure 2.6 shows the adjacency list of the undirected weighted graph of Figure 2.5.

```
0 - (3, 4.7) - (6, 2.9)
1 - (2, 7.5) - (4, 6.2) - (6, 4)
2 - (1, 7.5) - (3, 10) - (6, 7.5)
3 - (0, 4.7) - (2, 10) - (7, 4)
4 - (1, 6.2) - (5, 8.5) - (6, 5.2) - (7, 8.3)
5 - (4, 8.5) - (7, 5.5)
6 - (0, 2.9) - (1, 4) - (2, 7.5) - (4, 5.2) - (7, 6)
7 - (3, 4) - (4, 8.3) - (5, 5.5) - (6, 6)
```

Figure 2.6: Adjacency list of the graph of Figure 2.5. The notation is (vertex, weigth).

2.2.3 Shortest Paths

Given a nonnegative-weighted graph $G(V, E)$, the shortest path between vertices $u, v \in V$ is the one minimizing the sum of the costs of the traversed edges. Let us call $d_G(u, v)$ this shortest path cost (or minimum sum). The single-source shortest path problem is that

of computing the shortest path to all nodes in V starting from a particular node. This problem can be solved with Dijkstra's algorithm [Dij59], starting from the desired node. The all-pairs shortest path problem is that of computing the shortest path among every pair of vertices. This can be computed using Floyd's algorithm [Flo62] or by n applications of Dijkstra's [Dij59], taking each vertex as the origin node.

2.2.3.1 Dijkstra's Single-source Shortest Path Algorithm

Given a nonnegative weighted graph $G(V, E)$ and a starting node $s \in V$, we are looking for shortest paths starting from s to every other node in V. To do so, Dijkstra's algorithm uses some auxiliary variables. It maintains a set R of vertices whose final shortest paths from the source s have already been calculated. It uses an array sp to store the shortest path distance from s to every node $u \in V$ passing exclusively over the nodes in R. Finally, it uses an array $from$ to represent a path propagation tree whose root is s, so we can rebuild the shortest path from s to every node $u \in V$ by following the edges in $from$. This way, when R becomes V we will have all the shortest path distances in sp, and if we need, we can rebuild each shortest path by following the edges in $from$. R is initialized to empty, the shortest path estimations sp_u are fixed to ∞ for all $u \in V - \{s\}$ and zero for s, and for every node in V we fill the array $from$ with NULL.

Next, the algorithm repeatedly selects the vertex $u^* \in V - R$ with the minimum shortest path estimation, adds u^* to R, and updates the shortest path estimations for objects in the adjacency of u^* that are still out of R, with the minimum value between their current shortest path estimation (sp_v) and the shortest path from s to u^* plus the distance between u^* and v ($sp_{u^*} + weight_{u^*,v}$). For the updated nodes v, it sets $from_v \leftarrow u^*$. Figure 2.7 shows Dijkstra's algorithm.

```
Dijkstra (Graph G(V,E), Vertex s)
1.  For each u ∈ V Do sp_u ← ∞, from_u ← NULL
2.  sp_s ← 0, R ← ∅
3.  While V ≠ R Do
4.      u* ← argmin_{u∈V−R} sp_u
5.      R ← R ∪ {u*}
6.      For each v ∈ (V − R) ∩ adjacency(u*) Do
7.          If sp_{u*} + weight_{u*,v} < sp_v Then
8.              sp_v ← sp_{u*} + weight_{u*,v}
9.              from_v ← u*
10. Return (sp, from)
```

Figure 2.7: Dijkstra's single-source shortest path algorithm.

Once we have computed the shortest path distances starting from s, we can obtain the path $s \rightsquigarrow v$, for each $v \in V$, by using the following procedure. We start from the last node of the path $v_{-1} = v$. Following $from_{v_{-1}}$ we arrive at node v_{-2}, so we obtain

the edge (v_{-2}, v). Next, following $from_{v_{-2}}$ we arrive at node v_{-3}, so we obtain the edge (v_{-3}, v_{-2}). We repeat the process until we arrive at node $v_{-l} = s$, where we obtain the edge $(s, v_{-(l-1)})$. Finally, the path $s \rightsquigarrow v$ is the vertex sequence $s, v_{-(l-1)}, \ldots, v_{-3}, v_{-2}, v$.

The correctness of Dijkstra's algorithm is based on the fact that sp_{u^*} is already the shortest path cost between s and u^* at the time when u^* is added to set R (and obviously, this property holds from then on). Therefore, when R becomes V, we have the shortest paths from s to all the nodes of G.

At the beginning, $R = \emptyset$, so the invariant is trivially true. Later, in the first execution of the **While** loop, $sp_u = \infty$ for all the nodes, except for sp_s which is 0. As the edges have nonnegative cost, the minimum path cost is 0, thus we add s to R, and the invariant is still true. Now, we update the adjacency of s.

In the following minimum selections, nodes $u^* \leftarrow \text{argmin}_{u \in V-R}\ sp_u$ are chosen in non-decreasing order with respect to values of sp. Since when we pick the node u^* and update its adjacency in $V - R$, $v \in (V - R) \cap adjacency(u^*)$, the values sp_v are greater than sp_{u^*} (note that, if we also have edges with weight 0, then the updated value could be equal to sp_{u^*}), once we pick the node u^* its cost sp_{u^*} cannot decrease (as any other path $s \rightsquigarrow u^*$ will have to get out from R and thus will have a greater or equal cost).

The simplest Dijkstra's implementation requires $O\big(m + n^2\big) = O\big(n^2\big)$ CPU time. It checks every edge once, and it performs a sequential scanning over sp in order to find the minimum value. A practical, fast alternative implements sp using binary heaps, so it finds the minimum in time $O(\log n)$, and updates each distance in sp also in time $O(\log n)$. Thus, we obtain time $O(n \log n + m \log n) = O(m \log n)$, which is an important CPU time reduction in the case when $m = o\big(n^2 / \log n\big)$. In fact, we use this latter alternative in this thesis.

2.2.3.2 Floyd's All-pairs Shortest Path Algorithm

Floyd's algorithm can manage negative-weight edges, but assumes that there are no negative-weight cycles. Recall that the vertices of G are $V = \{0, \ldots, n-1\}$. Let us consider a vertex subset $\{0, \ldots, k\}$ for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices belong to $\{0, \ldots, k\}$, and let $mwp_{i,j}^k$ be the minimum weight path under this restriction and $md_{i,j}^k$ its distance. Floyd's algorithm exploits a relation between $mwp_{i,j}^k$ and $mwp_{i,j}^{k-1}$. The relation depends on whether k is an intermediate vertex of path $mwp_{i,j}^k$:

- If k is not an intermediate vertex of $mwp_{i,j}^k$, then all intermediate vertices of $mwp_{i,j}^k$ belong to $\{0, \ldots, k-1\}$. So, $mwp_{i,j}^k = mwp_{i,j}^{k-1}$, and $md_{i,j}^k = md_{i,j}^{k-1}$.
- If k is an intermediate vertex of $mwp_{i,j}^k$, then we split $i \rightsquigarrow j$ into $i \rightsquigarrow k$ and $k \rightsquigarrow j$, which are the minimum weight subpaths connecting i with k and k with j using

nodes belonging to $\{0,\ldots,k-1\}$. That is, the minimum weight subpaths $i \rightsquigarrow k$ and $k \rightsquigarrow j$ are precisely $mwp_{i,k}^{k-1}$ and $mwp_{k,j}^{k-1}$, whose minimum costs are $md_{i,k}^{k-1}$ and $md_{k,j}^{k-1}$, respectively. Therefore $mwp_{i,j}^{k}$ is the concatenation of $mwp_{i,k}^{k-1}$ and $mwp_{k,j}^{k-1}$ and $md_{i,j}^{k}$ is the sum of $md_{i,k}^{k-1}$ and $md_{k,j}^{k-1}$.

It is easy to see that the shortest path $i \rightsquigarrow j$ contains at most once the node k, as otherwise we should add one or more cycle cost $md_{k,k}^{k-1} \geq 0$, which cannot reduce the cost.

To verify whether k is an intermediate vertex of path $mwp_{i,j}^{k}$, it is enough to check if the sum of the distances $md_{i,k}^{k-1}$ and $md_{k,j}^{k-1}$ is lower than the current distance $md_{i,j}^{k-1}$.

Figure 2.8 shows Floyd's algorithm, where we neglect the superscript k and instead overwrite $md_{i,j}$ and $mwp_{i,j}$. It requires $O(n^3)$ CPU time, and it is recommended for graphs where $m = \Omega(n^2/\log n)$. Otherwise, when $m = o(n^2/\log n)$, it is better to use n times Dijkstra's, with a total time of $O(nm \log n)$.

```
Floyd (Graph G(V,E))
1.  For i,j ← 0,...,n−1 Do
2.      If (i,j) ∉ E Then md_{i,j} ← ∞, mwp_{i,j} ← ∅
3.      Else md_{i,j} ← weight_{i,j}, mwp_{i,j} ← < i,j >
4.  For i ← 0,...,n−1 Do md_{i,i} ← 0, mwp_{i,i} ← < i >
5.  For k ← 0,...,n−1 Do
6.      For i,j ← 0,...,n−1 Do
7.          If md_{i,k} + md_{k,j} < md_{i,j} Then
8.              md_{i,j} ← md_{i,k} + md_{k,j}
9.              mwp_{i,j} ← concatenate(mwp_{i,k}, mwp_{k,j})
10. Return (md, mwp)
```

Figure 2.8: Floyd's all-pairs shortest path algorithm.

2.2.4 Minimum Spanning Trees

Assume that $G(V,E)$ is a connected undirected graph with a nonnegative cost function $weight_e$ assigned to its edges $e \in E$. A minimum spanning tree mst of the graph $G(V,E)$ is a tree composed of $n-1$ edges of E connecting all the vertices of V at the lowest total cost $\sum_{e \in mst} weight_e$. Note that, given a graph, its MST is not necessarily unique.

The most popular algorithms to solve the MST problem are Kruskal's [Kru56] and Prim's [Pri57], whose basic versions have complexity $O(m \log m)$ and $O(n^2)$, respectively. When $m = o(n^2/\log n)$, it is recommended to use Kruskal's algorithm, otherwise Prim's algorithm is recommended [CLRS01, Wei99]. Alternatively, Prim's can be implemented using Fibonacci Heaps [FT87] to obtain $O(m + n \log n)$ complexity.

2.2.4.1 Kruskal's MST Algorithm

Kruskal's algorithm starts with n single-node components, and merges them until producing a sole connected component. To do this, Kruskal's algorithm begins by setting the mst to $(V, \emptyset)$, that is, n single-node trees. Later, in each iteration, it adds to the mst the cheapest edge of E that does not produce a cycle on the mst, that is, it only adds edges whose vertices belong to different connected components. Once the edge is added, both components are merged. When the process ends, the mst is a minimum spanning tree of $G(V, E)$.

To manage the component operations, the *Union-Find* data structure C with path compression is used, see [CLRS01, Wei99] for a comprehensive explanation. Given two vertices u and v, operation **find**(u) computes which component u belongs to, and **union**(u, v) merges the components of u and v. The amortized cost of **find**(u) is $O(\alpha(m, n))$ ($\alpha = \omega(1)$ is the very slowly-growing inverse Ackermann's function) and the cost of **union**(u, v) is constant.

Figure 2.9 depicts the basic Kruskal's MST algorithm. We need $O(n)$ time to initialize both C and mst, and $O(m \log m)$ time to sort the edge set E. Then we make at most $m\, O(\alpha(m, n))$-time iterations of the **While** loop. Therefore, Kruskal's complexity is $O(m \log m)$.

```
Kruskal1 (Graph G(V, E))
1.  UnionFind C ← {{v}, v ∈ V} // the set of all connected components
2.  mst ← ∅ // the growing minimum spanning tree
3.  ascendingSort(E), k ← 0
4.  While |C| > 1 Do
5.      (e = {u, v}) ← E[k], k ← k + 1 // select an edge in ascending order
6.      If C.find(u) ≠ C.find(v) Then
7.          mst ← mst ∪ {e}, C.union(u, v)
8.  Return mst
```

Figure 2.9: The basic version of Kruskal's MST algorithm (**Kruskal1**).

Assuming we are using either full or random graphs whose edge costs are assigned at random independently of the rest (using any continuous distribution), the subgraph composed by V with the edges reviewed by the algorithm is a random graph [JKŁP93]. Therefore, based on [JKŁP93, p. 349], we expect to finish the MST construction (that is, to connect the random subgraph) upon reviewing $\frac{1}{2}n \ln n + \frac{1}{2}\gamma n + \frac{1}{4} + O\left(\frac{1}{n}\right)$ edges, which can be much smaller than m. Note that it is not necessary to sort the whole set E, but it is enough to select and extract one by one the minimum-cost edges until we complete the MST. The common solution of this type consists in min-heapifying the set E, and later performing as many min-extractions of the lowest cost edge as needed (in [MS91], they do this in their Kruskal's demand-sorting version). This is an application of Incremental Sort

(Section 2.1.2). For this sake we just modify lines 3 and 5 of Figure 2.9. Figure 2.10 shows Kruskal's algorithm with incremental sorting.

Kruskal2 (Graph $G(V, E)$)
1. UnionFind $C \leftarrow \{\{v\}, v \in V\}$ // the set of all connected components
2. $mst \leftarrow \emptyset$ // the growing minimum spanning tree
3. **heapify**(E)
4. **While** $|C| > 1$ **Do**
5. $(e = \{u, v\}) \leftarrow E$.**extractMin**() // select an edge in ascending order
6. **If** C.**find**$(u) \neq C$.**find**(v) **Then**
7. $mst \leftarrow mst \cup \{e\}$, C.**union**(u, v)
8. **Return** mst

Figure 2.10: Kruskal's algorithm with incremental sorting (**Kruskal2**). Note the differences when comparing with basic Kruskal's in lines 3 and 5.

Kruskal's algorithm with incremental sorting needs $O(n)$ time to initialize both C and mst, and $O(m)$ time to heapify E. We expect to review $\frac{1}{2}n \ln n + O(n)$ edges in the **While** loop. For each of these edges, we use $O(\log m)$ time to select and extract the minimum element of the heap, and $O(\alpha(m, n))$ time to perform operations **union** and **find**. Therefore, the average complexity is $O(m + n \log n \log m)$. As $n - 1 \leq m \leq n^2$, the average complexity of Kruskal with incremental sorting (also known as Kruskal with demand sorting) can be written as $O\left(m + n \log^2 n\right)$.

2.2.4.2 Prim's MST Algorithm

Prim's algorithm computes the MST incrementally starting from some arbitrary source node s. Thus, the algorithm starts with the single node s and no edges, and adds one-by-one nodes of V to the growing MST, until it connects all the nodes of V at the minimum cost.

To do so, Prim's algorithm maintains a set R of vertices already reachable from s, and two arrays *cost* and $from$ (note the similarity with Dijkstra's, Section 2.2.3.1). R can be seen as a growing MST. $cost_u$ stores the cost to connect the node u to some node in R, and $from_u$ stores which node in R u is connected to. R is initialized to empty, and for all nodes u in V, $cost_u$ is initialized to ∞ and $from_u$ to NULL. Finally, $cost_s$ is initialized to 0.

Next, the algorithm repeatedly selects the vertex $u^* \in V - R$ with the minimum connecting cost, adds u^* to R, and updates the connecting cost for objects in the adjacency of u^* that are still out of R with the minimum value between the current connecting cost ($cost_v$) and the weight of the edge between u^* and v ($weight_{u^*,v}$). For the updated nodes v, it sets $from_v \leftarrow u^*$. Figure 2.11 shows Prim's algorithm.

Prim1 (Graph $G(V,E)$, Vertex s)
1. **For each** $u \in V$ **Do** $cost_u \leftarrow \infty$, $from_u \leftarrow$ NULL
2. $cost_s \leftarrow 0$, $R \leftarrow \emptyset$
3. **While** $V \neq R$ **Do**
4. $u^* \leftarrow \text{argmin}_{u \in V-R}\ cost_u$
5. $R \leftarrow R \cup \{u^*\}$
6. **For each** $v \in (V-R) \cap adjacency(u^*)$ **Do**
7. **If** $weight_{u^*,v} < cost_v$ **Then**
8. $cost_v \leftarrow weight_{u^*,v}$
9. $from_v \leftarrow u^*$
10. **Return** $(cost, from)$

Figure 2.11: The basic version of Prim's MST Algorithm (**Prim1**).

Once we have finished the computation, we can obtain the MST by noting that it is composed by the edges $(u, from_u)$ for all the nodes $u \in V - \{s\}$ (the cost of these edges are the ones indicated in *cost*).

Like Dijkstra's, the simplest Prim's implementation requires $O\big(m+n^2\big) = O\big(n^2\big)$ CPU time. It checks every edge once, and it performs a sequential scanning over *cost* in order to find the minimum value. A practical, fast alternative implements *cost* using binary heaps, so it finds the minimum in time $O(\log n)$, and updates the values in *cost* also in time $O(\log n)$. Thus, we obtain time $O(n \log n + m \log n) = O(m \log n)$, which is an important CPU time reduction in the case when $m = o\big(n^2/\log n\big)$.

2.2.4.3 Further Work on the MST Problem

There are several other MST algorithms compiled by Tarjan [Tar83]. Recently, B. Chazelle [Cha00] gave an $O(m\alpha(m,n))$ time algorithm. Later, S. Pettie and V. Ramachandran [PR02] proposed an algorithm that runs in optimal time $O(\mathcal{T}^*(m,n))$, where $\mathcal{T}^*(m,n)$ is the minimum number of edge-weight comparisons needed to determine the MST of any graph $G(V,E)$ with m edges and n vertices. The best known upper bound of this algorithm is also $O(m\alpha(m,n))$. These algorithms almost reach the lower bound $\Omega(m)$, yet they are so complicated that their interest is mainly theoretical. Furthermore, there is a randomized algorithm [KKT95] that finds the MST in $O(m)$ time with high probability in the restricted RAM model, but it is also considered impractical as it is complicated to implement and the $O(m)$ complexity hides a large constant factor.

Experimental studies on MST are given in [MS91, KST02, KST03]. In [MS91], Moret and Shapiro compare several versions of Kruskal's, Prim's and Tarjan's algorithms, concluding that the best in practice (albeit not in theory) is Prim's using pairing heaps [FSST86]. Their experiments show that neither Cheriton and Tarjan's [CT76] nor Fredman and Tarjan's algorithm [FT87] ever approach the speed of Prim's algorithm using pairing

heaps. On the other hand, they show that the basic Kruskal's algorithm can run very fast when it uses an array of edges that can be overwritten during sorting, instead of an adjacency list. Moreover, they show that it is possible to use heaps to improve Kruskal's algorithm. The result is a rather efficient MST version with complexity $O(m + k \log m)$, being $k \leq m$ the number of edges reviewed by Kruskal's technique. However, they also show that the worst-case behavior of Kruskal's algorithm stays poor: If the graph has two distinct components connected by a single, very costly edge, incremental sorting is forced to process the whole edge set.

In [KST02, KST03], Katriel et al. present the algorithm *iMax*, whose expected complexity is $O(m + n \log n)$. It generates a subgraph G' by selecting $\sqrt{mn}$ edges from E at random. Then, it builds the minimum spanning forest T' of G'. Third, it filters each edge $e \in E$ using the cycle property: discard e if it is the heaviest edge on a cycle in $T' \cup \{e\}$. Finally, it builds the MST of the subgraph that contains the edges of T' and the edges that were not filtered out.

2.2.5 k-Nearest Neighbor Graphs

Let us consider a vertex set V and a nonnegative distance function d defined among the elements of V. Following our metric space query notation, let $NN_k(u)$ be the k elements in $V - \{u\}$ having the smallest distance to u according to function d. The *k-nearest neighbor graph* of V (kNNG) is a weighted directed graph $G(V, E)$ connecting each element to its k nearest neighbors, $E = \{(u, v), v \in NN_k(u)\}$, so that $weight_{u,v} = d(u, v)$. Building the kNNG is a direct generalization of the well-known *all-nearest-neighbor* (ANN) problem, which corresponds to the 1NNG construction problem. kNNGs are central in many applications: cluster and outlier detection [EE94, BCQY96], VLSI design, spin glass and other physical process simulations [CK95], pattern recognition [DH73], query or document recommendation systems [BYHM04a, BYHM04b], similarity self joins [DGSZ03, DGZ03, PR08], and many others.

2.2.5.1 Related Work on kNNG Construction Algorithms

The naive approach to construct kNNGs uses $\frac{n(n-1)}{2} = O(n^2)$ CPU time and $O(kn)$ memory. For each $u \in V$ we compute the distance towards all the other $v \in V - \{u\}$, and select the k lowest-distance objects. However, there are several alternatives to speed up the procedure. The proximity properties of the Voronoi diagram [Aur91] or its dual, the Delaunay triangulation, allow solving the problem more efficiently. The ANN problem can be optimally solved in $O(n \log n)$ time in the plane [Ede87] and in $\mathbb{R}^D$ for any fixed D [Cla83, Vai89], but the constant depends exponentially on D. In $\mathbb{R}^D$, kNNGs can be built in $O(nk \log n)$ time [Vai89] and even in $O(kn + n \log n)$ time [Cal93, CK95, DE96]. Approximation algorithms for the problem have also been proposed [AMN+94]. Nevertheless, none of these alternatives, except the naive one, is suitable for general metric spaces, as they use coordinate information that is not necessarily available in all metric spaces.

2.2.6 t-Spanners

Let $G(V, E)$ be a connected undirected graph with a nonnegative cost function $weight_e$ assigned to its edges $e \in E$. Let us call $d_G(u, v)$ the shortest path cost between nodes u and v. A *t-spanner* is a subgraph $G'(V, E')$, with $E' \subseteq E$, which permits us to compute path costs with *stretch* t, that is, ensuring that for any $u, v \in V$, $d_{G'}(u, v) \leq t \cdot d_G(u, v)$ [PU89, PS89, Epp99]. We call the latter the t-condition. The t-spanner problem has applications in distributed systems, communication networks, architecture of parallel machines, motion planning, robotics, computational geometry, and others.

2.2.6.1 Related Work on t-Spanner Construction Algorithms

It was shown [PS89] that, given a graph G with unitary weight edges and parameters t and m, the problem of determining whether a t-spanner of at most m edges exists is NP-complete. Hence, there is no hope for efficient construction of minimal t-spanners. Furthermore, as far as we know, only in the case $t = 2$ and graphs with unitary weight edges there exist polynomial-time algorithms that guarantee an approximation bound in the number of edges (or in the weight) of the resulting t-spanner [KP94] $\left(\text{the bound is } \log \frac{|E|}{|V|}\right)$.

If we do not force any guarantee on the number of edges of the resulting t-spanner, a simple $O\left(mn^2\right)$-time greedy algorithms exists [Par02, NP03, NPC07], where $n = |V|$ and $m = |E|$ refer to the resulting t-spanner. It was shown [ADDJ90, ADD$^+$93] that these techniques produce t-spanners with $n^{1+O(\frac{1}{t-1})}$ edges on general graphs of n nodes.

More sophisticated algorithms have been proposed by Cohen in [Coh98], producing t-spanners with guaranteed $O\left(n^{1+(2+\varepsilon)(1+\log_n m)/t}\right)$ edges in worst case time $O\left(mn^{(2+\varepsilon)(1+\log_n m)/t}\right)$, where in this case m refers to the original graph. Other recent algorithms [TZ01] work only for $t = 1, 3, 5, \ldots$. Parallel algorithms have been pursued in [LB96], but they do not translate into new sequential algorithms.

With regard to Euclidean t-spanners, that is, the subclass of t-spanners where objects are points in a D-dimensional space with Euclidean distance, much better results exist [Epp99, ADDJ90, ADD$^+$93, Kei88, GLN02, RS91], showing that one can build t-spanners with $O(n)$ edges in $O\left(n \log^{D-1} n\right)$ time. These results, unfortunately, make heavy use of coordinate information and cannot be extended to general metric spaces.

In [Par02, NP03, NPC07] we also propose several algorithms to build t-spanners in the metric space case with $n^{1+O(\frac{1}{t-1})}$ edges, using $O(nm \log n)$ worst-case CPU time and $O\left(n^2\right)$ distance computations. These algorithms consider objects as black boxes, and construct t-spanners by using the distances among them measured with the metric function d. They exploit the fact that, in metric spaces, the shortest path cost between two objects u and v is simply the distance $d(u, v)$. These solutions are also well suited to general graphs, in which case it is necessary a previous step so as to compute the shortest path distances among objects.

2.3 A Summary of Metric Spaces

All models are wrong but some are useful.

– George Box

A metric space is a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is the universe of objects, and $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+ \cup \{0\}$ is a distance function defined among them. The function d can be seen as a measure of object dissimilarity. Therefore, the smaller the distance between two objects, the more "similar" they are. The distance function satisfies the following metric properties:

$\forall\ x, y \in \mathbb{X},$	$x \neq y \Rightarrow d(x, y) > 0$	strict positiveness,
$\forall\ x, y \in \mathbb{X},$	$d(x, y) = d(y, x)$	symmetry,
$\forall\ x \in \mathbb{X},$	$d(x, x) = 0$	reflexivity,
$\forall\ x, y, z \in \mathbb{X},$	$d(x, z) \leq d(x, y) + d(y, z)$	triangle inequality.

These properties hold for many reasonable similarity functions. The distance is assumed to be expensive to compute (think, for instance, in comparing two fingerprints).

The metric space model is useful to model several applications. In this work, we focus on the metric space search problem. So, from now on we consider a finite *database* $\mathbb{U}$, which is a subset of the universe $\mathbb{X}$ composed by the objects we are interested in searching. For instance, $\mathbb{U}$ can be formed by the collection of fingerprints of people working in some institution.

2.3.1 Proximity Queries

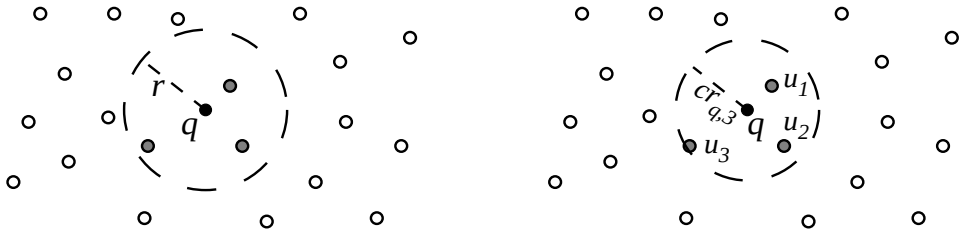
There are two typical proximity queries:

Range query (q, r)**:** Retrieve all elements which are within distance r to q. That is, $(q, r) = \{x \in \mathbb{U},\ d(x, q) \leq r\}$. Figure 2.12(a) illustrates a range query in $\mathbb{R}^2$ under the Euclidean distance. The query outcome is composed by the gray nodes.

k**-Nearest neighbor query** $NN_k(q)$**:** Retrieve the k elements from $\mathbb{U}$ closest to q. That is, $NN_k(q)$ is a set such that $\forall\, x \in NN_k(q)$, $y \in \mathbb{U} - NN_k(q)$, $d(q, x) \leq d(q, y)$, and $|NN_k(q)| = k$. Figure 2.12(b) shows a 3-nearest neighbor query also in $\mathbb{R}^2$ under the Euclidean distance. The query outcome is composed by the nodes $\{u_1, u_2, u_3\}$.

The total time to solve an online proximity query can be split as

$$T = \text{distance evaluations} \times \text{complexity of } d + \text{extra CPU time} + \text{I/O time}.$$



(a) Range query (q, r) (b) 3-Nearest neighbor query $NN_3(q)$

Figure 2.12: In (a), a range query (q, r). In (b), a 3-nearest neighbor query. The covering radius of the query $NN_3(q)$ is the distance from q to u_3, that is, $cr_{q,3} = d(q, u_3)$.

Naturally, we would like to minimize T. In many applications the distance function d is so expensive to compute, that it is customary to define the complexity of the search as the number of distance evaluations performed, disregarding other components such as CPU time for side computations, and even I/O time. This is the complexity model used in this thesis. Nevertheless, we also pay attention to the extra CPU time of side computations when solving proximity queries.

Given a database of $n = |\mathbb{U}|$ objects, proximity queries can be trivially answered by performing n distance evaluations, which is called a *sequential scanning*. However, as computing the distance is costly, the sequential scanning is usually unacceptable. Under this cost model, the ultimate goal of metric space search techniques is to structure the database so as to perform the lowest number of distance computations when solving a similarity query.

Metric space search algorithms preprocess the database $\mathbb{U}$ to build an *index* $\mathcal{I}$. Note that we can only compare objects by using the distance function of the metric space. Thus, to build $\mathcal{I}$ we select a subset of cells from the full distance matrix $\mathbb{U} \times \mathbb{U}$ and store some information on them.

Later, given an object from the universe, a query $q \in \mathbb{X}$, we solve the proximity query as follows. First, we use the index $\mathcal{I}$ and the triangle inequality in order to discard as many objects from $\mathbb{U}$ as possible, and create a (hopefully small) candidate set $\mathcal{C}$ of the objects that could belong to the query outcome. Second, we traverse $\mathcal{C}$ comparing each object directly with q so as to get the final query outcome.

The range query (Figure 2.12(a)) is the basic type of proximity query. In fact, it is possible to implement a k-nearest neighbor query as a succession of range queries of increasing radii until we retrieve an outcome of size k. However, there are specific algorithms to solve $NN_k(q)$ queries more efficiently.

Some observations on k-nearest neighbor queries are in order. In case of ties we choose any k-element set that satisfies the condition. The *covering radius* $cr_{q,k}$ of a query $NN_k(q)$ is the distance from q towards the farthest neighbor in $NN_k(q)$. In Figure 2.12(b), the covering radius $cr_{q,3}$ is $d(q, u_3)$. A $NN_k(q)$ algorithm using an index $\mathcal{I}$ over $\mathbb{U}$ is called

range-optimal if it uses the same number of distance evaluations as the best algorithm solving the range query $(q, cr_{q,k})$ using index $\mathcal{I}$ [HS00]. In Figure 2.12, this corresponds to implementing the query $NN_3(q)$ as a range query $(q, cr_{q,3})$, for $cr_{q,3} = d(q, u_3)$. It is worth to say that there exists range-optimal algorithms for several metric space search indices [HS00], but not for all of them.

2.3.2 Vector Spaces

A particular case of the metric space search problem arises when the space is $\mathbb{R}^D$. There are several distance functions to choose from, but the most widely used belong to the Minkowski distance family L_s, defined as:

$$L_s\left((x_1,\ldots,x_D),(y_1,\ldots,y_D)\right) = \left(\sum_{i=1}^{D}|x_i - y_i|^s\right)^{1/s} .$$

The most common examples of the Minkowski family are L_1, L_2 and L_∞, also known as the "block" or "Manhattan" distance, Euclidean distance, and maximum distance, respectively. We obtain L_∞ by taking the limit of the formula L_s when s goes to infinity:

$$L_\infty\left((x_1,\ldots,x_D),(y_1,\ldots,y_D)\right) = \max_{i=1..D}|x_i - y_i| .$$

In many applications objects are represented as feature vectors, thus the metric space is indeed a vector space, and the similarity can be interpreted geometrically. Note that a vector space grants more freedom than a general metric space when designing search approaches, since it is possible to directly access geometric and coordinate information that is unavailable in general metric spaces. In this thesis we solely use information of the distance among vectors, neglecting any geometric or coordinate information.

In vector spaces there exist optimal algorithms (on the database size) in both the average and the worst case for 1-nearest neighbor searching [BWY80]. Search indices for vector spaces are called *spatial access methods*. Among them we have *kd*-trees [Ben75, Ben79], *R*-trees [Gut84], quad-trees [Sam84] and *X*-trees [BKK96]. These techniques intensively exploit the coordinate information to group and classify vectors. For example, *kd*-trees divide the space using different coordinates through levels of the tree; and *R*-trees group points in hyper-rectangles. Regrettably, the existing solutions are very sensitive to the vector space dimension, so for high dimensions those structures cease to work well. As a matter of fact, range and nearest neighbor queries have an exponential dependence on the dimension of the vector space [Cha94].

Spatial access methods are a research subject by themselves, out of the scope of this thesis. For further reference on vector spaces techniques see [Sam84, WJ96, GG98, BBK01, Sam06]. In this thesis we focus in general metric spaces, although these solutions are also well suited to D-dimensional spaces, especially in medium and high dimensions

($D \geq 8$), where the spatial access methods diminish their performance. One of the reasons of their performance reduction in medium and high dimensional spaces comes form the fact that dimensions are independent each other. So, when we index the space by using one coordinate, we could miss the indexing performed by the others, forcing the backtracking throughout all the index.

2.3.3 A Notion of Dimensionality in Metric Spaces

It is interesting to notice that the concept of "dimensionality" can be translated from vector spaces into metric spaces, where it is called *intrinsic dimensionality*. Although there is not an accepted criterion to measure the intrinsic dimensionality of a given metric space, some empirical approaches based on the histogram of distances among objects have been proposed. In general terms, it is agreed that a metric space is high-dimensional —that is, it has high intrinsic dimensionality— when its histogram of distances is concentrated. As it can empirically be observed that the cost of proximity queries worsens quickly as the histogram concentrates, it is also accepted that a high intrinsic dimension degrades the performance of any similarity search algorithm [CNBYM01]. Indeed, this performance degradation exposes an exponential dependence on the space intrinsic dimensionality. In fact, an efficient method for proximity searching in low dimensions may become painfully slow in high dimensions. For large enough dimensions, no proximity search algorithm can avoid directly comparing the query against all the database. This is called the *curse of dimensionality*.

2.3.4 Current Solutions for Metric Space Searching

Nowadays, there exists several methods to preprocess a metric database in order to reduce the number of distance evaluations at search time. All of them build an index $\mathcal{I}$ over the metric database, so they can discard some elements from consideration by using the triangle inequality. Most of current solutions are grouped into two algorithm families [CNBYM01]. The first, called *pivoting algorithms*, is based on reference objects. The second, called *compact partitioning algorithms*, is based on splitting the dataset into spatially compact subsets.

There are a few other metric space search approaches not clearly fitting into these two main families. In this thesis we are particularly interested in the *graph based approach*. In this approach, the database $\mathbb{U}$ is indexed by using a graph $G(\mathbb{U}, E)$, where E is a subset of the whole weighted edge set $\mathbb{U} \times \mathbb{U}$. The main idea is to use jointly the metric properties and graph features (for instance, upper bounding the distances between two objects with the length of the shortest path between them). We are aware of three previous graph-based techniques: Shasha and Wang's [SW90], this previous work (our MSc thesis) [Par02, NP03, NPC02, NPC07] and Sebastian and Kimia's [SK02].

In the following, we give the underlying ideas of the main families. For a

comprehensive description of these algorithms see [CNBYM01, HS03, ZADB06, Sam06]. The graph based approaches are explained in Section 2.4.

2.3.4.1 Pivot-based Algorithms

We will pay special attention to this class of algorithms because we use them as a baseline to compare the performance of our graph-based approaches.

To build the index, these algorithms select a set $\mathcal{P}$ of distinguished elements, the *pivots* $\mathcal{P} = \{p_1 \dots p_{|\mathcal{P}|}\} \subseteq \mathbb{U}$, and store a table of $|\mathcal{P}|n$ distances $d(u, p_i),\ i \in \{1 \dots |\mathcal{P}|\}, u \in \mathbb{U}$.

Later, to solve a range query (q, r), pivot-based algorithms measure $d(q, p_1)$ and use the fact that, by virtue of the triangle inequality, $d(q, u) \geq |d(q, p_1) - d(u, p_1)|$, so they can discard every $u \in \mathbb{U}$ such that

$$|d(q, p_1) - d(u, p_1)| \quad > \quad r \ , \tag{2.3}$$

since this implies $d(q, u) > r$, without computing explicitly $d(q, u)$.

Once they are done with p_1, they try to discard elements from the remaining set using p_2, and so on, until they use all the pivots in $\mathcal{P}$. The elements u that still cannot be discarded at this point are directly compared against q.

The $|\mathcal{P}|$ distance evaluations computed between q and the pivots are known as the *internal complexity* of the algorithm. If there is a fixed number of pivots, this complexity has a fixed value. On the other hand, the distance evaluations used to compare the query with the objects not discarded by the pivots are known as the *external complexity* of the algorithm. Hence, the total complexity of a pivot-based search algorithm is the sum of both the internal and external complexities.

Since the internal complexity increases and the external complexity decreases with $|\mathcal{P}|$, there is an optimal size of the pivot set $|\mathcal{P}|^*$ that minimizes the total complexity. However, in practice $|\mathcal{P}|^*$ is so large that one cannot store all the $|\mathcal{P}|^* n$ distances, hence the index usually uses as many pivots as memory permits.

Several pivot-based algorithms are almost direct implementations of this idea, and differ essentially in their extra structures used to reduce the CPU time of side computations, but not in the number of distance evaluations performed.

Other algorithms use tree-like data structures, so they implement this idea indirectly: they select a pivot as the root of the tree and divide the space according to the distance to the root. One slice corresponds to each subtree, the number of slices and width of each slice differs from one technique to another. At each subtree, a new pivot is selected, and so on. This way we obtain pivots that are local to the subtree, unlike the previous description where pivots store distances to all the objects in $\mathbb{U}$. These structures differ in the number of distance computations, and also in their extra CPU time for side computations. Their

main drawback is that they do not allow the classical trade-off consisting in using more space for the index in order to compute fewer distances at query time.

Among the pivot-based algorithms we can find structures for discrete or continuous distance functions. In the discrete case we have:

- *Burkhard-Keller Tree* (BKT) [BK73],
- *Fixed Queries Tree* (FQT) [BYCMW94],
- *Fixed-Height FQT* (FHQT) [BYCMW94, BY97], and
- *Fixed Queries Array* (FQA) [CMN01].

In the continuous case we have:

- *Vantage-Point Tree* (VPT) [Yia93, Chi94, Uhl91],
- *Multi-Vantage-Point Tree* (MVPT) [Bri95, BO97],
- *Excluded Middle Vantage Point Forest* (VPF) [Yia98],
- *Approximating Eliminating Search Algorithm* (AESA) [Vid86], and
- *Linear AESA* (LAESA) [MOV94].

2.3.4.2 Compact-Partition based Algorithms

This idea consists in splitting the space into zones as compact as possible and assigning the objects to these zones. See [JD88] for further information about clustering algorithms.

For each zone, the compact-partition based index stores (i) the object identifiers, (ii) a representative point (called the *center*), and (iii) few extra data allowing to quickly discard the zone at query time. After we perform a first division into zones, each zone is recursively subdivided, so that we obtain a hierarchy of zones. There are two basic criteria to delimit a zone.

The first one is the *Voronoi area*, where we select a set of centers $\{c_1, \ldots, c_m\}$ and place each other point in the zone of its closest center. Thus, the areas are limited by hyperplanes, analogously to a Voronoi partition in a vector space. At query time, we evaluate $(d(q, c_1), \ldots, d(q, c_m))$, choose the closest center c and discard every zone whose center c_i satisfies $d(q, c_i) > d(q, c) + 2r$, as its Voronoi area cannot have intersection with the query ball.

The second criterion is the *covering radius* $cr(c_i)$, which is the maximum distance from the center c_i towards any element in its zone. At query time, if $d(q, c_i) - r > cr(c_i)$, then there is no need to consider the i-th zone.

A compact-partition based index using only hyperplanes is:

- *Generalized-Hyperplane Tree* (GHT) [Uhl91].

Others use only covering radii:

- *Bisector Trees* (BST) [KM83],
- *Voronoi Tree* (VT) [DN87].
- *M-tree* (MT) [CPZ97], and
- *List of Clusters* (LC) [CN05].

Finally, some of them use both criteria:

- *Geometric Near-neighbor Access Tree* (GNAT) [Bri95], and
- *Spatial Approximation Tree* (SAT) [Nav02].

2.3.5 Approximating Eliminating Search Algorithm (AESA)

By far, the most successful technique for searching metric spaces ever proposed is AESA [Vid86, CNBYM01]. Its main problem, however, is that it requires precomputing and storing a matrix of all the $n(n-1)/2$ distances among the objects of $\mathbb{U}$. That is, AESA use the full distance matrix as the search index. This huge space requirement makes it unsuitable for most applications.

In AESA the idea of pivots is taken to the extreme $|\mathcal{P}| = n$, that is, every element is a potential pivot and hence we need a matrix with all the $\frac{n(n-1)}{2}$ precomputed distances. Since we are free to choose any pivot, the pivot to use next is chosen from the elements not yet discarded. Additionally, as it is well known that pivots closer to the query are much more effective, the pivot candidates u are ranked according to the sum of their current lower-bound distance estimations to q. That is, if we have used pivots $\{p_1 \dots p_i\}$, we choose the pivot p_{i+1} as the element u minimizing

$$SumLB(u) \quad = \quad \sum_{j=1}^{i} |d(q,p_j) - d(u,p_j)| \ . \tag{2.4}$$

AESA works as follows. It starts with a set of candidate objects $\mathcal{C}$, which is initially $\mathbb{U}$, and initializes $SumLB(u) = 0$ for all $u \in \mathbb{U}$. Then, it chooses an object $p \in \mathcal{C}$ minimizing $SumLB$ (Eq. (2.4)) and removes it from $\mathcal{C}$. Note that the first object p_1 is actually chosen at random. It measures $d_{qp} \leftarrow d(q,p)$ and immediately reports p if $d_{qp} \leq r$. By Eq. (2.3), it removes from $\mathcal{C}$ objects u which satisfy $d(u,p) \notin [d_{qp} - r, d_{qp} + r]$. Recall that $d(u,p)$ is obtained from the precomputed full distance matrix $\mathbb{U} \times \mathbb{U}$. Otherwise, for non-discarded

```
AESA (Query q, ℝ+radius, matrix ℐ)
      // ℐ_{u,v} = d(u,v), u,v ∈ 𝕌
1.    𝒞 ← 𝕌
2.    For each p ∈ 𝒞 Do SumLB(p) ← 0
3.    While 𝒞 ≠ ∅ Do
4.        p ← argmin_{c∈𝒞} SumLB(c), 𝒞 ← 𝒞 − {p}
5.        d_qp ← d(q,p), If d_qp ≤ radius Then Report p
6.        For each u ∈ 𝒞 Do
7.            If ℐ_{u,p} ∉ [d_qp − radius, d_qp + radius] Then 𝒞 ← 𝒞 − {u}
8.            Else SumLB(u) ← SumLB(u) + |d_qp − ℐ_{u,p}|
```

Figure 2.13: Algorithm AESA. $\mathcal{I}$ is the full distance matrix $\mathbb{U} \times \mathbb{U}$, so $\mathcal{I}_{u,p}$ is the distance between u and p.

objects we update $sumLB$ according to Eq (2.4). We repeat these steps until $\mathcal{C} = \emptyset$. Figure 2.13 depicts the algorithm.

AESA has been experimentally shown to have almost constant search cost. Nevertheless, the constant hides an exponential dependence on the dimensionality of the metric space, which once again is due to the curse of dimensionality. Nevertheless, AESA's main problem is that storing $O(n^2)$ distances is impractical for most applications. A recent development [FCNP06] shows that it can slightly improve upon AESA by choosing the next pivot in another way.

2.3.6 Non-exact Algorithms for Proximity Searching

As seen in Section 2.3.3, there are so-called high-dimensional metric spaces where solving proximity queries requires reviewing almost all the database whatever index we use. In these cases, one can consider solving the problem approximately.

Note that, when we design a model to represent real-life objects, we usually lose some information. Think, for instance, in the vector representation of a document. This representation does not consider either positions of the words composing the document, the document structure, or the semantics. Given this level of imprecision, an approximate answer in the modeled space is usually acceptable for the sake of speeding up the query process.

This gives rise to new approaches to the metric space search problem: probabilistic and approximate algorithms. In the first we try to find the objects relevant to a given query with high probability. An intuitive notion of what these algorithms aim to is that they attempt not to miss many relevant objects at query time. In the second we try to report $(1+\epsilon)$-closest objects, which means that, given the query q, they retrieve objects $u \in \mathbb{U}$ such that $d(q,u) \leq (1+\epsilon)d(q,u_{q,k})$ where $u_{q,k}$ is the k-th closest object to q in

$\mathbb{U}$. That is, they aim at finding close enough objects, though not necessarily the k closest ones.

2.4 Graphs and Metric Space Searching

> There exists no separation between gods and men; one blends softly casual into the other.
>
> – *Proverbs of Muad'dib*, from *Dune Messiah*, by Frank Herbert

In this thesis we approach the metric space search problem through graph data structures. Thus, in this section we start by showing the relation between graphs and the metric space search problem. Later, we survey the current state of the art in this matter, explaining how they improve the search process by exploiting the features of the underlying graph index and the triangle inequality.

2.4.1 Graph-based Metric Space Indices

Given a metric space $(\mathbb{X}, d)$ and a database of interest $\mathbb{U} \subseteq \mathbb{X}$, if we identify every object in $\mathbb{U}$ with a node, the full distance matrix can be regarded as a complete graph $G(V = \mathbb{U}, \mathbb{U}\times\mathbb{U})$, where edge weights are distances measured between the edge nodes by using the metric function d, that is $weight_{u,v} = d(u, v)$. Likewise, if we use a reduced set of edges $E \subseteq \mathbb{U}\times\mathbb{U}$, whose weights are also computed by using the distance d of the metric space, we obtain a graph-based index $G(\mathbb{U}, E)$ using less memory than the full distance matrix.

Once we have the graph-based index $G(\mathbb{U}, E)$, by virtue of the triangle inequality, for any $u, v \in \mathbb{U}$ the shortest path between u and v, $d_G(u, v)$, is an upper bound on the real distance, $d(u, v)$, see Figure 2.14. We can use this property to avoid distance computations when solving proximity queries. Moreover, we can select an edge set satisfying some further

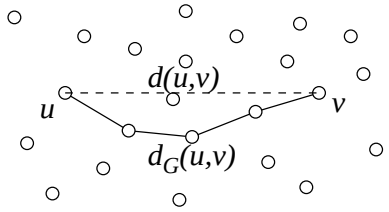


Figure 2.14: The distance $d(u, v)$ is upper bounded by the length of the shortest path $d_G(u, v)$.

properties that can be exploited to improve the search process.

We have found three previous metric space indices based on graphs: one based on arbitrary graphs [SW90], one based on t-spanners [Par02, NP03, NPC02, NPC07], and one

based on kNNGs [SK02]. The latter is a non-exact approach, in the sense that it does not guarantee to find the objects solving the proximity query. The methods are explained in the following sections.

2.4.2 Shasha and Wang's Algorithm

Shasha and Wang [SW90] use a graph whose nodes are the objects of the metric space and whose edges are an *arbitrary collection* of distances among the objects. They only use the triangle inequality to estimate other distances. To calculate the distance estimations, they compute two $n \times n$ matrices. In the first, *ADM* (approximated distance map), they store a lower bound on the distance between two nodes (obtained using the triangle inequality). In the second, *MIN* (minimum path weight), they store an upper bound on the distances (obtained using the shortest path between the nodes). To avoid confusion with the names, let us change the notation from *ADM* and *MIN* to *LB* and *UB*, respectively.

Each cell $LB_{i,j}$ is computed as follows. $LB_{i,j}$ is the maximum, over all possible paths between nodes i and j, of the difference between the longest edge of the path and the sum of all the others, that is $LB_{i,j} = \max_{path\ i \rightsquigarrow j}\{2\max\{edges(path)\} - length(path)\}$. They present an $O\left(n^3\right)$ dynamic programing technique to obtain LB. On the other hand, UB is computed by using Floyd's all-pair shortest path algorithm (see Section 2.2.3.2, page 21).

Given a range query (q, r), matrix UB can discard objects u_j such that $UB_{i,j} < d(u_i, q) - r$, for some u_i. On the other hand, LB can discard objects u_j such that $LB_{i,j} > d(u_i, q) + r$, for some u_i.

The greatest deficiencies of [SW90] are (1) they require $O\left(n^2\right)$ space which is too much memory in most real applications; (2) the selected distances are arbitrary and do not give any guarantee on the quality of their approximation to the real distances. In fact, the index only performs well when distances follow a uniform distribution, which does not occur in practice. Even in $\mathbb{R}$, an extremely easy-to-handle metric space, distances have a triangular distribution, whereas in general metric spaces the distance distribution is usually more concentrated, far from uniform.

2.4.3 t-Spanners and Metric Spaces

As we have said, although AESA is the most successful metric space search algorithm, its $O\left(n^2\right)$ space requirement makes it unsuitable for most applications. Nevertheless, if we reduced the memory requirement of AESA, we could use it in many practical scenarios. For this sake, note that the full AESA distance matrix (see Section 2.3.5) can be regarded as a complete graph $G(\mathbb{U}, \mathbb{U} \times \mathbb{U})$, where $d_G(u, v) = d(u, v)$ is the distance between elements u and v in the metric space. Thus, in order to save memory we can use a t-spanner $G'(\mathbb{U}, E \subseteq \mathbb{U} \times \mathbb{U})$ of G, which permits us estimating upper bounds on the distance between every pair of objects within a factor t, that is $d_{G'}(u, v) \leq t \cdot d(u, v)$, without the need to store $O\left(n^2\right)$ distances but only $|E|$ edges. However, in this case we cannot directly apply

AESA search algorithm over the t-spanner, but we have to take into account the error introduced by the stretch factor t.

Practical results on metric t-spanner construction are given in [Par02, NP03, NPC07]. There are also some related results referring to probabilistic approximations to construct t-spanners for metric spaces using tree metrics [Bar98, CCG+98].

2.4.3.1 Simulating AESA Search over a t-Spanner

Given a t-spanner G' of $G(\mathbb{U}, \mathbb{U}\times\mathbb{U})$, for every $u, v \in \mathbb{U}$ the following property is guaranteed

$$d(u,v) \quad \leq \quad d_{G'}(u,v) \quad \leq \quad t \cdot d(u,v) \ . \tag{2.5}$$

Eq. (2.5) permits us adapting AESA to this distance approximation. According to the stretch factor t, to simulate AESA over a t-spanner it is enough to "extend" the upper bound of the AESA exclusion ring with the associated decrease in discrimination power. More precisely, the condition to be outside the ring, that is, Eq. (2.3), can be rewritten as

$$d(u,p) \quad < \quad d(q,p) - r \qquad \text{or} \qquad d(u,p) \quad > \quad d(q,p) + r \ . \tag{2.6}$$

Since we do not know the real distance $d(u,v)$, but only the approximated distance over the t-spanner, $d_{G'}(p,u)$, we can use Eqs. (2.5) and (2.6) to obtain the new discarding conditions, in Eqs. (2.7) and (2.8):

$$d_{G'}(u,p) \quad < \quad d(q,p) - r \ , \tag{2.7}$$

$$d_{G'}(u,p) \quad > \quad t \cdot (d(q,p) + r) \ . \tag{2.8}$$

What we have obtained is a relaxed version of AESA, which requires less memory $\big(O(|E|)$ instead of $O\big(n^2\big)\big)$ and, in exchange, discards fewer elements per pivot. As t tends to 1, our approximation becomes better but we need more and more edges. Hence we have a space-time trade-off where full AESA is just one extreme.

Let us now consider how to choose the next pivot. Since we have only an approximation to the true distance, we cannot directly use Eq. (2.4). To compensate for the effect of the precision factor t, after some experimental fine-tuning, we chose $\alpha_t = \frac{2/t+1}{3}$, so as to rewrite Eq. (2.4) as follows:

$$sumLB'(u) = \sum_{j=1}^{i} \Big| d(q,p_j) - d_{G'}(u,p_j) \cdot \alpha_t \Big| \ . \tag{2.9}$$

The resulting search algorithm, t-AESA, is quite similar to AESA. So, in Figure 2.15 we give the algorithm without further explanations (for these, we encourage the reader to check [NPC07] :-).

t-AESA (Query q, $\mathbb{R}^+ radius$, t-Spanner G')
1. $\mathcal{C} \leftarrow \mathbb{U}$, $\alpha_t \leftarrow \frac{2/t+1}{3}$
2. **For each** $p \in \mathcal{C}$ **Do** $SumLB'(p) \leftarrow 0$
3. **While** $\mathcal{C} \neq \emptyset$ **Do**
4. $\quad p \leftarrow \text{argmin}_{c \in \mathcal{C}} SumLB'(c)$, $\mathcal{C} \leftarrow \mathcal{C} - \{p\}$
5. $\quad d_{qp} \leftarrow d(q,p)$, **If** $d_{qp} \leq radius$ **Then** Report p
6. $\quad d_{G'} \leftarrow$ **Dijkstra**$(G', p, t(d_{qp} + radius))$
7. $\quad$ **For each** $u \in \mathcal{C}$ **Do**
8. $\quad\quad$ **If** $d_{G'}(u,p) \notin [d_{qp} - radius, t(d_{qp} + radius)]$ **Then** $\mathcal{C} \leftarrow \mathcal{C} - \{u\}$
9. $\quad\quad$ **Else** $SumLB'(u) \leftarrow SumLB'(u) + |d_{qp} - d_{G'}(u,p) \cdot \alpha_t|$

Figure 2.15: Algorithm t-AESA. **Dijkstra**(G', p, x) computes distances over the t-spanner G' from p to all nodes up to distance x, and marks the remaining ones as "farther away".

2.4.4 kNNG and Metric Spaces

The *k-nearest neighbor graph* (kNNG) of the set $\mathbb{U}$ is a weighted directed graph $G(\mathbb{U}, E)$ connecting each element $u \in \mathbb{U}$ to its k nearest neighbors, thus $E = \{(u,v), v \in NN_k(u)\}$. That is, for each element u we store the result of its k-nearest neighbor query $NN_k(u)$.

In this thesis we focus on this kind of graph, and how to use it in the search process. Indeed, kNNGs have several other applications, as we have already said in Section 2.2.5, page 26.

2.4.4.1 Related Work on kNNG Construction Algorithms for Metric Spaces

As we show in Section 2.2.5.1, there are several algorithms to construct geometric kNNGs [AMN+94, Cal93, CK95, Cla83, DE96, Ede87, Vai89]. Unfortunately, all of them assume that nodes are points in a vector space $\mathbb{R}^D$ and that d is the Euclidean or some Minkowski distance. However, this is not the case in several applications where kNNGs are required. For instance, let us consider the collaborative filters for Web search based on query or document recommendation systems [BYHM04a, BYHM04b]. In these applications, kNNGs are used to find clusters of similar queries, to later improve the quality of the results shown to the final user by exploiting cluster properties.

The naive idea to construct kNNGs in metric spaces is iterative: for each $u \in \mathbb{U}$ we compute the distance towards all the others, and select the k smallest-distance objects. Using this procedure we make $O(n^2)$ distance computations.

Clarkson states the first generalization of ANN to metric spaces [Cla99], where the problem is solved using randomization in $O(n \log^2 n \log^2 \Gamma(\mathbb{U}))$ expected time. Here, $\Gamma(\mathbb{U})$ is the distance ratio between the farthest and closest pairs of points in $\mathbb{U}$. The author argues that in practice $\Gamma(\mathbb{U}) = n^{O(1)}$, in which case the approach is $O(n \log^4 n)$ time.

However, the analysis needs a sphere packing bound in the metric space. Otherwise the cost must be multiplied by "sphere volumes", that are also exponential on the dimensionality. Moreover, the algorithm needs $\Omega(n^2)$ space for high dimensions, which is too much for practical applications.

Another way to construct kNNGs in general metric spaces is by indexing the metric space with any metric index (see Section 2.3.4, page 31), and next solving the k-nearest neighbor query for each $u \in \mathbb{U}$. As a matter of fact, in [Fig00], the author used a pivot-based index to solve n range queries of decreasing radii. (Decreasing radius range query is a suboptimal way to solve $NN_k(q)$.) In this thesis, we improve this general idea by giving a methodology that can use any metric space index and exploit both metric and graph properties in order to build kNNGs.

Recently, Karger and Ruhl presented the *metric skip list* [KR02], an index that uses $O(n \log n)$ space and can be constructed with $O(n \log n)$ distance computations. The index answers $NN_1(q)$ queries using $O(\log n)$ distance evaluations with high probability. Later, Krauthgamer and Lee introduce *navigating nets* [KL04], another index that can be constructed also with $O(n \log n)$ distance computations, yet using $O(n)$ space, and which gives an $(1+\epsilon)$-approximation algorithm to solve $NN_1(q)$ queries in time $O(\log n) + (1/\epsilon)^{O(1)}$. Both of them could serve to solve the ANN problem with $O(n \log n)$ distance computations but not to build kNNGs. In addition, the hidden constants are exponential on the intrinsic dimension, which makes these approaches useful only in low-dimensional metric spaces.

2.4.4.2 Related Work on Using the kNNG for Proximity Searching

Sebastian and Kimia [SK02] suggest using the kNNG as an approximation of the Delaunay graph, that is, the graph produced by the Delaunay triangulation. In fact, they give an approach to solve 1-nearest neighbor queries using the kNNG as a navigational device. The idea is to traverse the graph starting from a node, which they call the *seed* node, towards the query q by jumping from one node to its neighbor if the neighbor is closer to the query than the node itself.

If the underlying graph were Delaunay's or a superset of it, this obviously would work. However, it is proved in [Nav02] that it the only superset of the Delaunay graph that works for an arbitrary metric space is the complete graph.

Therefore, Sebastian and Kimia's algorithm is a non-exact approach, in the sense that, given a query element $q \in \mathbb{X}$, it does not guarantee to find the nearest neighbor of q from the elements in $\mathbb{U}$. Indeed, it fails when (i) the kNNG is not connected (lack of connectedness), or (ii) the algorithm arrives at a node which is not the nearest neighbor (false nearest neighbor). In the following we illustrate with Figure 2.16, where we are looking for the nearest neighbor w of q over an Euclidean 3NNG. In the first case, each graph component defines a node cluster in the kNNG. If we do not start inside the cluster the query belongs, we cannot arrive at the nearest neighbor. In the figure, if we start in

node x we arrive at node y. In the second case, we can arrive at a node which is not the nearest neighbor, yet its neighbors are farther from q than the node itself. In the figure, if we start in node u we arrive at node v.

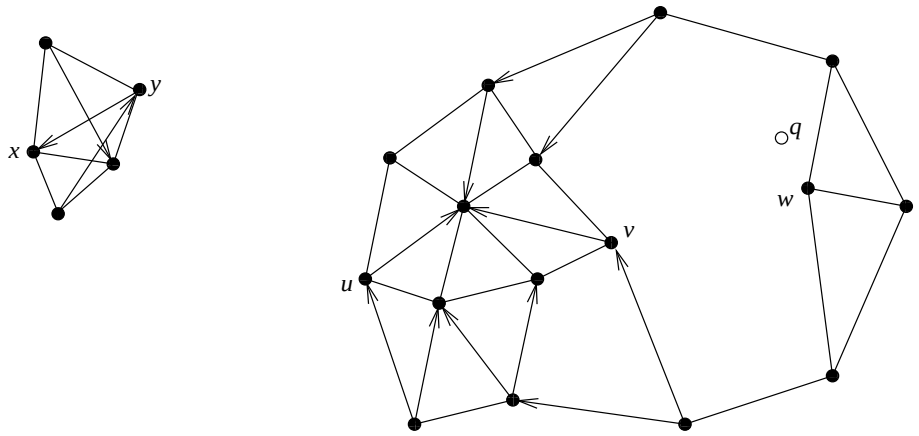


Figure 2.16: Cases where Sebastian and Kimia's algorithm does not work over a 3NNG. Directed edges mean that the target node belongs to the source node 3-nearest neighbors. Undirected edges mean that both vertices are in the 3-nearest neighborhood of each other. We are looking for the nearest neighbor w of q. Suppose we are traversing an Euclidean 3NNG by jumping from one node to its neighbor if the neighbor is closer to the query than the node itself. Due to the lack of connectedness of the kNNG, if we start from node x, which does not belong to the query's connected component, we arrive at node y. On the other hand, even if we start in the component where the query belongs, if we start from node u we arrive at node v which is a false nearest neighbor.

They also give some heuristic techniques to overcome these failures. Among them, they propose to (i) use several (far away) seed nodes to start the navigation, (ii) increase the number of neighbors in the graph, and (iii) start the traversing not only in the seeds, but also in the seed's neighborhood. The former technique tries to deal with the lack of connectedness, the latter with false nearest neighbors, and the second with both failure cases. However, none of these techniques ensures obtaining the real nearest neighbor.

In his encyclopedic book, H. Samet [Sam06] adds two other alternatives to cope with the lack of connectedness of the kNNG. The first is the relative neighborhood graph [Tou80] and the second is the Gabriel graph [GS69]. He also explains how to construct these graphs in the metric space context. Note that using these alternatives we avoid to use several seeds (as the graph becomes connected), but the false nearest neighbor failure still remains, as both relative neighborhood graph and Gabriel graph are subsets of the Delaunay graph.

Chapter 3

Fundamental Algorithms

> And Paul thought: That was in no vision of mine. I did a different thing.
>
> – *Dune*, by Frank Herbert

During this work, we face several fundamental problems, most of which already have efficient solutions. Surprisingly, we found that the incremental sorting problem —that is, obtaining one-by-one the next smallest element of a given unsorted set— still offers interesting research opportunities. This problem can be seen as the online version of the partial sorting problem. It has several applications, being priority queues and the MST construction problem two of the most prominent ones. In Sections 3.1 and 3.2 we present our incremental sorting algorithm. Then, in Sections 3.3 and 3.4 we apply it to priority queues in main memory. Next, in Section 3.5 we show how to adapt our priority queue to work in secondary memory. We go on in Section 3.6 by applying our basic algorithms and structures to boost the construction of the MST of a given graph. Finally, in Section 3.7 we give some experimental results.

3.1 Optimal Incremental Sorting

> If I have seen a little farther than others it is because I have stood on the shoulders of giants.
>
> – Sir Isaac Newton

Let A be a set of size m. Obtaining the first $k \leq m$ elements of A in ascending order can be done in optimal $O(m + k \log k)$ time (Section 2.1.2.1). We present *Incremental Sort* (**IS**), an algorithm (online on k) which incrementally gives the next smallest element of

the set, so that the first k elements are obtained in optimal time for any k. As explained in Section 2.1.2.1, this is not a big achievement because the same can be obtained using a priority queue. However, we also give a practical version of the algorithm, *Incremental Quicksort* (**IQS**), with the same expected complexity, which performs better in practice than the best existing online algorithm.

We start by describing algorithm **IQS**. At the end we show how it can be converted into its worst-case version **IS**. Essentially, to output the k smallest elements, **IQS** calls Quickselect (Section 2.1.1) to find the smallest element of arrays $A[0, m-1]$, $A[1, m-1]$, ..., $A[k-1, m-1]$. This naturally leaves the k smallest elements sorted in $A[0, k-1]$. **IQS** avoids the $O(km)$ complexity by reusing the work across calls to Quickselect.

Let us recall how Quickselect works. Given an integer k, Quickselect aims to find the k-th smallest element from a set A of m numbers. For this sake it chooses an object p (the pivot), and partitions A so that the elements smaller than p are allocated to the left-side partition, and the others to the right side. After the partitioning, p is placed in its correct position i_p. So, if $i_p = k$, Quickselect returns p and finishes. Otherwise, if $k < i_p$ it recursively processes the left partition, else the right partition, with a new $k \leftarrow k - i_p - 1$.

Note that when we incrementally search for the next minimum element of a given set, it is possible to reuse the work made by previous calls to Quickselect. When we call Quickselect on $A[1, m-1]$, a decreasing sequence of pivots has already been used to partially sort A in the previous invocation on $A[0, m-1]$. **IQS** manages this sequence of pivots so as to reuse previous work. Specifically, it uses a stack S of decreasing pivot positions that are relevant for the next calls to Quickselect.

Figure 3.1 shows how **IQS** searches for the smallest element (12) of an array by using a stack initialized with a single value $m = 16$. To find the next minimum, we first check whether p, the top value in S, is the index of the element sought, in which case we pop it and return $A[p]$. Otherwise, because of previous partitionings, it holds that elements in $A[0, p-1]$ are smaller than all the rest, so we run Quickselect on that portion of the array, pushing new pivots into S.

As can be seen in Figure 3.1, the second minimum (18) is the pivot on the top of S, so we pop it and return $A[1]$. Figure 3.2 shows how **IQS** finds the third minimum using the pivot information stored in S. Notice that **IQS** *just works on the current first chunk* ($\{29, 25\}$). In this case it adds one pivot position to S and returns the third element (25) in the next recursive call.

Now, retrieving the fourth and fifth elements is easy since both of them are pivots. Figure 3.3 shows how **IQS** finds the sixth minimum. The current first chunk contains three elements: $\{41, 49, 37\}$. So, **IQS** obtains the next minimum by selecting 41 as pivot, partitioning its chunk and returning the element 37. The incremental sorting process will continue as long as needed, and it can be stopped in any time.

The algorithm is given in Figure 3.4. Stack S is initialized to $S = \{|A|\}$. **IQS**

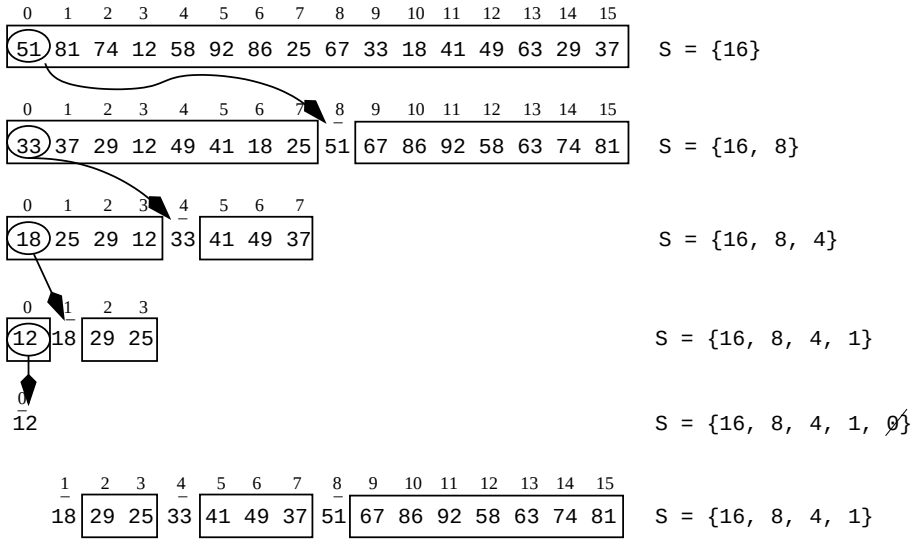


Figure 3.1: Example of how **IQS** finds the first element of an array. Each line corresponds to a new partition of a sub-array. Note that all the pivot positions are stored in stack S. In the example we use the first element of the current partition as the pivot, but it could be any other element. The bottom line shows the array with the three partitions generated by the first call to **IQS**, and the pivot positions stored in S.

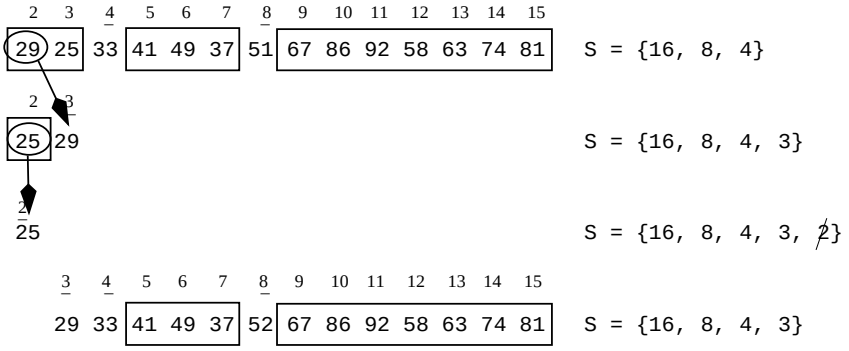


Figure 3.2: Example of how **IQS** finds the third element of the array. Since it starts with pivot information stored in S, it just works on the current first chunk ($\{29, 25\}$).

receives the set A, the index idx[1] of the element sought (that is, we seek the smallest element in $A[idx, m-1]$), and the current stack S (with former pivot positions). First it checks whether the top element of S is the desired index idx, in which case it pops idx and returns $A[idx]$. Otherwise it chooses a random pivot index $pidx$ from $[idx, S.\mathbf{top}()-1]$. Pivot $A[pidx]$ is used to partition $A[idx, S.\mathbf{top}()-1]$. After the partitioning, the pivot has reached its final position $pidx'$, which is pushed in S. Finally, a recursive invocation continues the work on the left hand of the partition. The algorithm can obviously be used to find largest elements instead of the smallest.

[1]Since we start counting array positions from 0, the place of the k-th element is $k-1$, so $idx = k-1$.

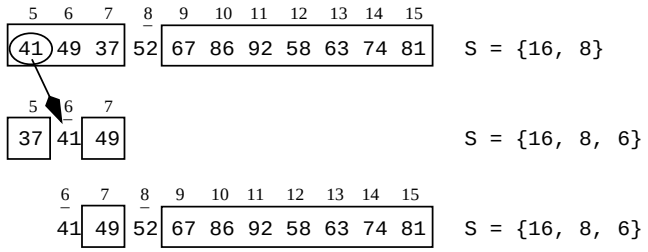


Figure 3.3: Example of how **IQS** finds the sixth element of an array. Since it starts with pivot information stored in S, it just works on the current first chunk ($\{41, 49, 37\}$). We omit the line where element 37 becomes a pivot and is popped from S.

IQS (Set A, Index idx, Stack S)
// Precondition: $idx \leq S.\mathbf{top}()$
1. **If** $idx = S.\mathbf{top}()$ **Then** $S.\mathbf{pop}()$, **Return** $A[idx]$
2. $pidx \leftarrow \mathbf{random}[idx,\ S.\mathbf{top}()-1]$
3. $pidx' \leftarrow \mathbf{partition}(A,\ A[pidx],\ idx,\ S.\mathbf{top}()-1)$
// Invariant: $A[0] \leq \ldots \leq A[idx-1] \leq A[idx, pidx'-1] \leq A[pidx']$
// $\leq A[pidx'+1, S.\mathbf{top}()-1] \leq A[S.\mathbf{top}(), m-1]$
4. $S.\mathbf{push}(pidx')$
5. **Return IQS**$(A,\ idx,\ S)$

Figure 3.4: Algorithm Incremental Quicksort (**IQS**). Stack S is initialized to $S \leftarrow \{|A|\}$. Both S and A are modified and rearranged during the algorithm. Note that the search range is limited to the array segment $A[idx, S.\mathbf{top}()-1]$. Procedure **partition** returns the position of pivot $A[pidx]$ after the partition completes. Note that the tail recursion can be easily removed.

Recall that **partition**$(A,\ A[pidx],\ i,\ j)$ rearranges $A[i, j]$ and returns the new position $pidx'$ of the original element in $A[pidx]$, so that, in the rearranged array, all the elements smaller/larger than $A[pidx']$ appear before/after $pidx'$. Thus, pivot $A[pidx']$ is left at the correct position it would have in the sorted array $A[i, j]$. The next lemma shows that it is correct to search for the minimum just within $A[i, S.\mathbf{top}() - 1]$, from which the correctness of **IQS** immediately follows.

Lemma 3.1 (pivot invariant)**.** *After i minima have been obtained in $A[0, i-1]$, (1) the pivot indices in S are decreasing bottom to top, (2) for each pivot position $p \neq m$ in S, $A[p]$ is not smaller than any element in $A[i, p-1]$ and not larger than any element in $A[p+1, m-1]$.*

Proof. Initially this holds since $i = 0$ and $S = \{m\}$. Assume this is valid before pushing p, when p' was the top of the stack. Since the pivot was chosen from $A[i, p'-1]$ and left at some position $i \leq p \leq p'-1$ after partitioning, property (1) is guaranteed. As for property (2), after the partitioning it still holds for any pivot other than p, as the partitioning rearranged elements at the left of all previous pivots. With respect to p, the partitioning

ensures that elements smaller than p are left at $A[i, p-1]$, while larger elements are left at $A[p+1, p'-1]$. Since $A[p]$ was already not larger than elements in $A[p', m-1]$, the lemma holds. It obviously remains true after removing elements from S. □

The worst-case complexity of **IQS** is $O(m^2)$, but it is easy to derive worst-case optimal **IS** from it. The only change is in line 2 of Figure 3.4, where the random selection of the next pivot position must be changed to choosing the median of $A[idx, S.\mathbf{top}() - 1]$, using the linear-time selection algorithm [BFP+73]. The resulting algorithm, Incremental Sorting, is depicted in Figure 3.5.

IS (Set A, Index idx, Stack S)
// Precondition: $idx \leq S.\mathbf{top}()$
1. **If** $idx = S.\mathbf{top}()$ **Then** $S.\mathbf{pop}()$, **Return** $A[idx]$
2. $pidx \leftarrow$ **median**(A, idx, $S.\mathbf{top}()-1$) // using linear-time section algorithm
3. $pidx' \leftarrow$ **partition**(A, $A[pidx]$, idx, $S.\mathbf{top}()-1$)
// Invariant: $A[0] \leq \ldots \leq A[idx-1] \leq A[idx, pidx'-1] \leq A[pidx']$
// $\leq A[pidx'+1, S.\mathbf{top}()-1] \leq A[S.\mathbf{top}(), m-1]$
4. $S.\mathbf{push}(pidx')$
5. **Return IS**(A, idx, S)

Figure 3.5: Algorithm Incremental Sort (**IS**). Note the difference with **IQS** in line 2.

3.2 Analysis of IQS

> Truth suffers from too much analysis.
>
> – Ancient Fremen Saying, from *Dune Messiah*, by Frank Herbert

In Section 3.2.1 we analyze **IS**, which is not as efficient in practice as **IQS**, but has good worst-case performance. In particular, this analysis serves as a basis for the expected case analysis of **IQS** in Section 3.2.2. Both complexities are $O(m + k \log k)$.

3.2.1 IS Worst-case Complexity

In **IS**, the partition perfectly balances the first chunk since each pivot position is chosen as the median of its array segment.

In this analysis we assume that m is of the form $2^j - 1$. We recall that array indices are in the range $[0, m-1]$. Figure 3.6 illustrates the incremental sorting process when $k = 5$ in a perfect balanced tree of $m = 31$ elements, $j = 5$. Black nodes are the elements

already reported, grey nodes are the pivots that remain in stack S, and white nodes and trees are the other elements of A.

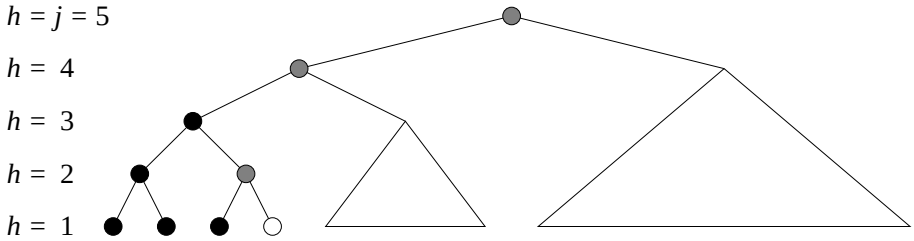


Figure 3.6: **IS** partition tree for incremental sorting when $k = 5$, $m = 31$, $j = 5$.

The pivot at the tree root is the first to be obtained (the median of A), at cost linear in m (both to obtain the median and to partition the array). The two root children are the medians of $A[0, \frac{m-3}{2}]$ and $A[\frac{m+1}{2}, m-1]$. Obtaining those pivots and partitioning with them will cost time linear in $m/2$. The left child of the root will actually be the second pivot to be processed. The right child, on the other hand, will be processed only if $k > \frac{m-1}{2}$, that is, at the moment we ask **IS** to output the $\frac{m+1}{2}$-th minimum. In general, processing the pivots at level h will cost $O(2^h)$, but only some of these will be required for a given k. The maximum level is $j = \log_2(m+1)$.

It is not hard to see that, in order to obtain the k smallest elements of A, we will require $\lceil \frac{k}{2^h} \rceil$ pivots of level h. Adding up their processing cost we get Eq. (3.1), where we split the sum into the cases $\lceil \frac{k}{2^h} \rceil > 1$ and $\lceil \frac{k}{2^h} \rceil = 1$. Only then, in Eq. (3.3), we use $k + 2^h$ to bound the terms of the first sum, and redistribute terms to obtain that **IS** is $O(m + k \log k)$ worst-case time. The extra space used by **IS** is $O(\log m)$ pivot positions.

$$
\begin{aligned}
T(m,k) &= \sum_{h=1}^{\log_2(m+1)} \left\lceil \frac{k}{2^h} \right\rceil 2^h && (3.1)\\
&= \sum_{h=1}^{\lfloor \log_2 k \rfloor} \left\lceil \frac{k}{2^h} \right\rceil 2^h + \sum_{h=\lfloor \log_2 k \rfloor + 1}^{\log_2(m+1)} 2^h && (3.2)\\
&\leq \sum_{h=1}^{\lfloor \log_2 k \rfloor} k + \sum_{h=1}^{\log_2(m+1)} 2^h && (3.3)\\
T(m,k) &= k \lfloor \log_2 k \rfloor + 2m + 1 && (3.4)
\end{aligned}
$$

What we have done is to prove the following theorem.

Theorem 3.1 (**IS**'s worst case complexity)**.** *Given a set A of m numbers,* **IS** *finds the k smallest elements, for any unknown value $k \leq m$, in worst-case time $O(m + k \log k)$.* □

3.2.2 IQS Expected-case Complexity

In this case the final pivot position p after the partitioning of $A[0, m-1]$ distributes uniformly in $[0, m-1]$. Consider again Figure 3.6, where the root is not anymore the middle of A but a random position. Let $T(m,k)$ be the expected number of key comparisons needed to obtain the k smallest elements of $A[0, m-1]$. After the $m-1$ comparisons used in the partitioning, there are three cases depending on p: (1) $k \leq p$, in which case the right subtree will never be expanded, and the total extra cost will be $T(p,k)$ to solve $A[0, p-1]$; (2) $k = p+1$, in which case the left subtree will be fully expanded to obtain its p elements at cost $T(p,p)$; and (3) $k > p+1$, in which case we pay $T(p,p)$ on the left subtree, whereas the right subtree, of size $m-1-p$, will be expanded so as to obtain the remaining $k-p-1$ elements.

Thus **IQS** expected cost follows Eq. (3.5), which is rearranged as Eq. (3.6). It is easy to check that this is exactly the same as Eq. (3.1) in [Mar04], which shows that **IQS** makes exactly the same number of comparisons of the offline version, Partial Quicksort (Section 2.1.2.1). This is $2m+2(m+1)H_m-2(m+3-k)H_{m+1-k}-6k+6$. That analysis [Mar04] is rather sophisticated, resorting to bivariate generating functions. In which follows we give a simple development arriving at a solution of the form $O(m + k \log k)$.

$$\begin{aligned} T(m,k) \;=\; m-1+\frac{1}{m}\left(\sum_{p=k}^{m-1} T(p,k) \quad + \quad T(k-1,k-1)\right. \\ \left. + \sum_{p=0}^{k-2}\Big(T(p,p)+T(m-1-p,k-p-1)\Big)\right) \end{aligned} \tag{3.5}$$

$$= \; m-1+\frac{1}{m}\left(\sum_{p=0}^{k-1} T(p,p)+\sum_{p=0}^{k-2} T(m-1-p,k-p-1)+\sum_{p=k}^{m-1} T(p,k)\right) \tag{3.6}$$

Eq. (3.6) simplifies to Eq. (3.7) by noticing that $T(p,p)$ behaves exactly like Quicksort, $2(p+1)H_p - 4p$ [GBY91] (this can also be seen by writing down $T(p) = T(p,p)$ and noting that the very same Quicksort recurrence is obtained), so that $\sum_{p=0}^{k-1} T(p,p) = k(k+1)H_k - \frac{k}{2}(5k-1)$. We also write p' for $k-p-1$ and rewrite the second sum as $\sum_{p'=1}^{k-1} T(m-k+p', p')$.

$$T(m,k) \;=\; m-1+\frac{1}{m}\left(k(k+1)H_k-\frac{k}{2}(5k-1)+\sum_{p=1}^{k-1} T(m-k+p,p) \;+\; \sum_{p=k}^{m-1} T(p,k)\right) \tag{3.7}$$

We make some pessimistic simplifications now. The first sum governs the dependence on k of the recurrence. To avoid such dependence, we bound the second argument to k and the first to m, as $T(m,k)$ is monotonic on both of its arguments. The new recurrence,

Eq. (3.8), depends only on parameter m and treats k as a constant.

$$T(m) \;=\; m-1+\frac{1}{m}\left(k(k+1)H_k-\frac{k}{2}(5k-1)+(k-1)T(m)+\sum_{p=k}^{m-1}T(p)\right) \tag{3.8}$$

We subtract $m\,T(m)-(m-1)T(m-1)$ using Eq. (3.8), to obtain Eq. (3.9). Since $T(k)$ is actually $T(k,k)$, we use again Quicksort formula in Eq. (3.10). We bound the first part by $2m+2kH_{m-k}$ and the second part by $2kH_k$ to obtain Eq. (3.11).

$$\begin{aligned} T(m) \;&=\; 2\frac{m-1}{m-k+1}+T(m-1) \;=\; 2\sum_{i=k+1}^{m}\left(1+\frac{k-2}{i-k+1}\right)\;+\;T(k) && (3.9)\\ &=\; 2(m-k)+2(k-2)(H_{m-k+1}-1)+(2(k+1)H_k-4k) && (3.10)\\ &<\; 2(m+kH_{m-k}+kH_k) && (3.11)\end{aligned}$$

This result establishes that $T(m,k)<2\,(m+kH_{m-k}+kH_k)$ for any m,k. However, it does not yet look good enough. We plug it again into Eq. (3.7), so that we can bound the sum $\sum_{p=1}^{k-1}T(m-k+p,p)$ with $\sum_{p=1}^{k-1}2\,(m-k+p+pH_{m-k}+pH_p)$ $=(k-1)\left(2m+k\left(H_{m-k}+H_k-\frac{3}{2}\right)\right)$. Upper bounding again and multiplying by m we get a new recurrence in Eq. (3.12). Note that this recurrence only depends on m.

$$\begin{aligned} m\,T(m) \;=\;& m(m-1)+k(k+1)H_k-\frac{k}{2}(5k-1)\\ &+(k-1)\left(2m+k\left(H_{m-k}+H_k-\frac{3}{2}\right)\right)+\sum_{p=k}^{m-1}T(p)\end{aligned} \tag{3.12}$$

Subtracting again $m\,T(m)-(m-1)T(m-1)$ we get Eq. (3.13). Noting that $\frac{(k-1)k}{(m-k)m}=(k-1)\left(\frac{1}{m-k}-\frac{1}{m}\right)$, we get Eq. (3.14), which is solved in Eq. (3.15).

$$\begin{aligned} T(m) \;&=\; 2\frac{m+k-2}{m}+\frac{(k-1)k}{(m-k)m}+T(m-1) && (3.13)\\ &<\; \sum_{i=k+1}^{m}\left(2+2\frac{k-2}{i}+(k-1)\left(\frac{1}{i-k}-\frac{1}{i}\right)\right)+T(k) && (3.14)\\ &=\; 2(m-k)+2(k-2)(H_m-H_k)+(k-1)(H_{m-k}-H_m+H_k)\\ &\qquad +(2(k+1)H_k-4k) && (3.15)\end{aligned}$$

Note that $H_m-H_k\le\frac{m-k}{k+1}$ and thus $(k-2)(H_m-H_k)<m-k$. Also, $H_{m-k}\le H_m$, so collecting terms we obtain Eq. (3.16).

$$T(m,k) \;<\; 4m-8k+(3k+1)H_k \;<\; 4m+3kH_k \tag{3.16}$$

Therefore, **IQS** is also $O(m+k\log k)$ in the expected case, which is stated in the following theorem.

Theorem 3.2 (**IQS**'s expected case complexity)**.** *Given a set A of m numbers* **IQS** *finds the k smallest elements, for any unknown value $k \leq m$, in $O(m + k \log k)$ expected time.* □

As a final remark, we give a simple explanation to the fact that Incremental Quicksort performs fewer comparisons than the classical offline solution of using Quickselect to find the k-th element and then using Quicksort on the left partition. For shortness we call the classical Quickselect + Quicksort solution **QSS**, and the Partial Quicksort algorithm **PQS**. Note that when we use **QSS** a portion of the Quicksort partitioning work repeats the work made in the previous calls to Quickselect. Figure 3.7 illustrates this, showing that, upon finding the k-th element, the Quickselect stage has produced partitions A_1 and A_2, however the Quicksort that follows processes the left partition as a whole ($[A_1 p_1 A_2]$), ignoring the previous partitioning work done over it. On the other hand, **IQS** sorts the left segment by processing each partition independently, because it knows its limits (as they are stored in the stack S). This also applies to **PQS** and it explains the finding of C. Martínez that **PQS**, and thus **IQS**, makes $2k - 4H_k + 2$ fewer comparisons than **QSS** [Mar04].

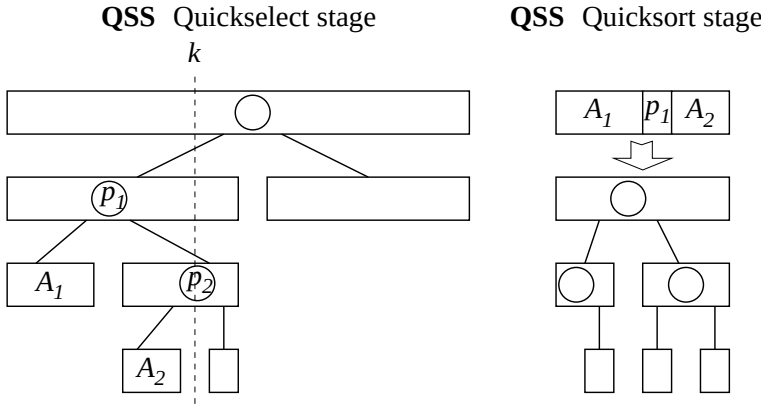


Figure 3.7: Partition work performed by **QSS**. First, **QSS** uses Quickselect for finding the k-th element (left). Then it uses Quicksort on the left array segment as a whole ($[A_1\ p_1\ A_2]$) neglecting the previous partitioning work (right).

3.3 Quickheaps

> On the other hand, we cannot ignore efficiency.
>
> – Jon Bentley

Let us go back to the last line of Figure 3.1, drawn in Figure 3.8, where we add ovals indicating pivots. For the sake of simplifying the following explanation, we also add a ∞ mark signaling a fictitious pivot in the last place of the array.

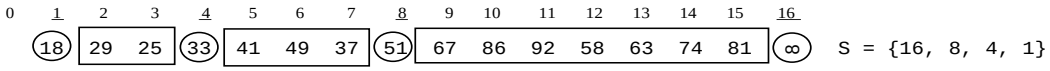


Figure 3.8: Last line of Figure 3.1.

By virtue of the **IQS** invariant (see Lemma 3.1), we see the following structure in the array. If we read the array from right to left, we start with a pivot (the fictitious pivot ∞ at position 16) and at its left side there is a chunk of elements smaller than it. Next, we have another pivot (pivot 51 at position 8) and another chunk. Then, another pivot and another chunk and so on, until we reach the last pivot (pivot 18 at position 1) and a last chunk (in this case, without elements).

This resembles a heap structure, in the sense that objects in the array are semi-ordered. In the following, we exploit this property to implement a priority queue over an array processed with algorithm **IQS**. We call this **IQS**-based priority queue *Quickheap* (**QH**). From now on we explain how to obtain a min-order quickheap. Naturally, we can symmetrically obtain a max-order quickheap.

3.3.1 Data Structures for Quickheaps

To implement a quickheap we need the following structures:

1. An array *heap*, which we use to store the elements. In the example of Figure 3.8, the array *heap* is $\{18, 29, \ldots, 81, \infty\}$.
2. A stack S to store the positions of pivots partitioning *heap*. Recall that the bottom pivot index indicates the fictitious pivot ∞, and the top one the smallest pivot. In the example of Figure 3.8, the stack S is $\{16, 8, 4, 1\}$.
3. An integer *idx* to indicate the first cell of the quickheap. In the example of Figure 3.8, $idx = 1$. Note that it is not necessary to maintain a variable to indicate the last cell of the quickheap (the position of the fictitious pivot ∞), as we have this information in $S[0]$.
4. An integer *capacity* to indicate the size of *heap*. We can store up to $capacity - 1$ elements in the quickheap (as we need a cell for the fictitious pivot ∞). Note that if we use *heap* as a circular array, we can handle arbitrarily long sequences of insertions and deletions as long as we maintain no more than $capacity - 1$ elements simultaneously in the quickheap.

Note that in the case of circular arrays, we must take into account that an object whose position is *pos* is actually located in the cell *pos* mod *capacity* of the circular array *heap*.

Figure 3.9 illustrates the structure. We add elements at the tail of the quickheap (the cell $heap[S[0] \bmod capacity]$), and perform min-extractions from the head of the quickheap

(the cell $heap[idx \bmod capacity]$). So, the quickheap *slides* from left to right over the circular array *heap* as the operation progresses.

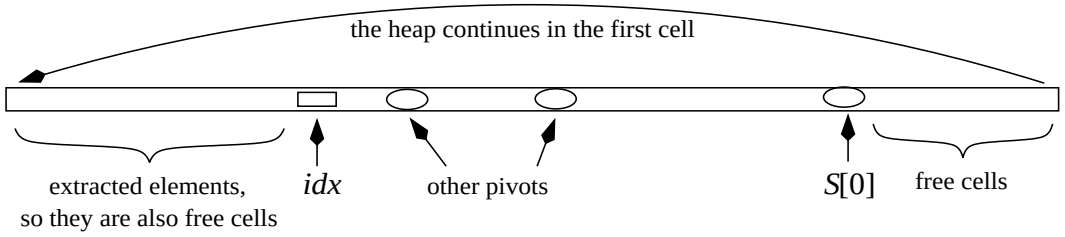


Figure 3.9: A quickheap example. The quickheap is placed over an array *heap* of size *capacity*. The quickheap starts at cell *idx*, there are three pivots, and the last cell of the heap is marked by the fictitious pivot $S[0]$. There are some cells after $S[0]$, which are free cells to store new elements. There are also free cells that correspond to extracted elements, that will be used when the quickheap turns around the circular array.

From now on, we will omit the expression mod *capacity* in order to simplify the reading (but keep it in pseudocodes).

3.3.2 Creation of Empty Quickheaps

The creation of an empty quickheap is rather simple. We create the array *heap* of size *capacity* with no elements, and initialize both $S = \{0\}$ and $idx = 0$. Figure 3.10 gives constructor **Quickheap**. The value of *capacity* must be sufficient to store simultaneously all the elements we need in the array. Note that it is not necessary to place the ∞ mark in $heap[S[0]]$, as we never access the $S[0]$-th cell.

3.3.3 Quick-heapifying an Array

It is also very simple to create a quickheap from an array A. We copy the array A to *heap*, and initialize both $S = |A|$ and $idx = 0$. Figure 3.10 gives constructor **Quickheap** over a given array A. The value of *capacity* must be at least $|A| + 1$.

Note that this operation can be done in time $O(1)$ if we can take array A and use it as array *heap*.

3.3.4 Finding the Minimum

We note that *idx* indicates the first cell used by the quickheap allocated over the array *heap*. The pivots stored in S delimit chunks of semi-ordered elements, in the sense that every object in a chunk is smaller than the pivot at its right, and the pivot is smaller or equal than every object in the next chunk at the right, an so on until the pivot in $S[0]$.

Thus, to find the minimum of the heap, it is enough to focus on the first chunk, that is, the chunk delimited by the cells idx and $S.\mathbf{top}() - 1$. For this sake, we use **IQS** to find the minimum of the first chunk: we just call $\mathbf{IQS}(heap, idx, S)$ and then return the element $heap[idx]$. However, in this case **IQS** does not pop the pivot on top of S. Figure 3.10 gives method **findMin**.

Remember that an element whose position is pos is located at cell pos mod $capacity$, thus we have to slightly change algorithm **IQS** to manage the positions in the circular array.

3.3.5 Extracting the Minimum

To extract the minimum, we first make sure that the minimum is located in the cell $heap[idx]$. (Once again, in this case **IQS** does not pop the pivot on top of S.) Next, we increase idx and pop S. Finally, we return the element $heap[idx - 1]$. Figure 3.10 gives method **extractMin**.

Quickheap(Integer N)
// constructor of an empty quickheap
1. $capacity \leftarrow N + 1$, $heap \leftarrow$ new Array[$capacity$], $S \leftarrow \{0\}$, $idx \leftarrow 0$

Quickheap(Array A, Integer N)
// constructor of a quickheap from an array A
1. $capacity \leftarrow \max\{N, |A|\} + 1$, $heap \leftarrow$ new Array[$capacity$], $S \leftarrow \{|A|\}$, $idx \leftarrow 0$
2. $heap.\mathbf{copy}(A)$

---

findMin()
1. $\mathbf{IQS}(heap, idx, S)$
2. **Return** $heap[idx \bmod capacity]$

---

extractMin()
1. $\mathbf{IQS}(heap, idx, S)$
2. $idx \leftarrow idx + 1$, $S.\mathbf{pop}()$
3. **Return** $heap[(idx - 1) \bmod capacity]$

Figure 3.10: Creation of an empty quickheap, creation of a quickheap from an array, finding the minimum, and extracting the minimum. N is an integer number giving the desired capacity of the heap. In operations **findMin** and **extractMin** we assume **IQS** is slightly modified in order to manage circular arrays, and **IQS** does not pop the pivot on top of S.

3.3.6 Inserting Elements

To insert a new element x into the quickheap we need to find the chunk where we can insert x in fulfillment of the pivot invariant (Lemma 3.1). Thus, we need to create an empty cell within this chunk in the array *heap*. A naive strategy will move every element in the array one position to the right, so the naive insertion time would be $O(m)$ worst-case complexity. Note, however, that it is enough to move some pivots and elements to create an empty cell in the appropriate chunk.

We first move the fictitious pivot, updating its position in S, without comparing it with the new element x, so we have a free cell in the last chunk. Next, we compare x with the pivot at cell $S[1]$. If the pivot is smaller than or equal to x we place x in the free place left by pivot $S[0]$. Otherwise, we move the first element at the right of pivot $S[1]$ to the free place left by pivot $S[0]$, and move the pivot $S[1]$ one place to the right, updating its position in S. We repeat the process with the pivot at $S[2]$, and so on until we find the place where x has to be inserted, or we reach the first chunk. Figure 3.11 illustrates. In the example, we insert element 35 in the quickheap of Figure 3.8, so we first move the fictitious pivot by updating its position in S. Next, we compare 35 with the pivot $heap[S[1]] = 51$. Since $35 < 51$, we move the element 67, which is next to 51, to the end of the chunk, and the pivot 51 one place to the right, updating its position in S. We continue by comparing 35 with the next pivot, 33. Since $35 > 33$, we finish displacing pivots and store 35 at the free position left by pivot 51.

Figure 3.12 shows method **insert**. It uses method **add**, which receives the chunk it must start from with the pivot displacement process to finally insert the element in the proper chunk. In the case of insertions, the pivot displacement process starts from the last chunk, but we will also use method **add** in other operations.

3.3.7 Deleting Arbitrary Elements

Operation **delete** works as follows. Given a position *pos* of some element in the quickheap, this operation deletes the element at cell *pos* from the quickheap. When we delete an element we move some pivots and elements one cell to the left, so this is the dual of operation **insert**.

To delete the element, we first need to find its chunk. Note that each chunk has a pivot at its right, so we reference the chunk by that pivot, *pidx*. Therefore, we traverse the stack S to find the smallest pivot that is larger than or equal to *pos*. We do this in method **findChunk**, depicted in Figure 3.13.

Once we have a pivot *pidx* at a position greater than *pos*, we repeat the following process. We place the element previous to the *pidx*-th pivot in the position *pos*, that is, we move the element $heap[S[pidx]-1]$ to position $heap[pos]$, so we have a free cell at position $S[pidx]-1$. Then, we move the pivot $heap[S[pidx]]$ one place to the left, and update its position in S. Then we update *pos* to the old pivot position, $pos = S[pidx]+1$. Then

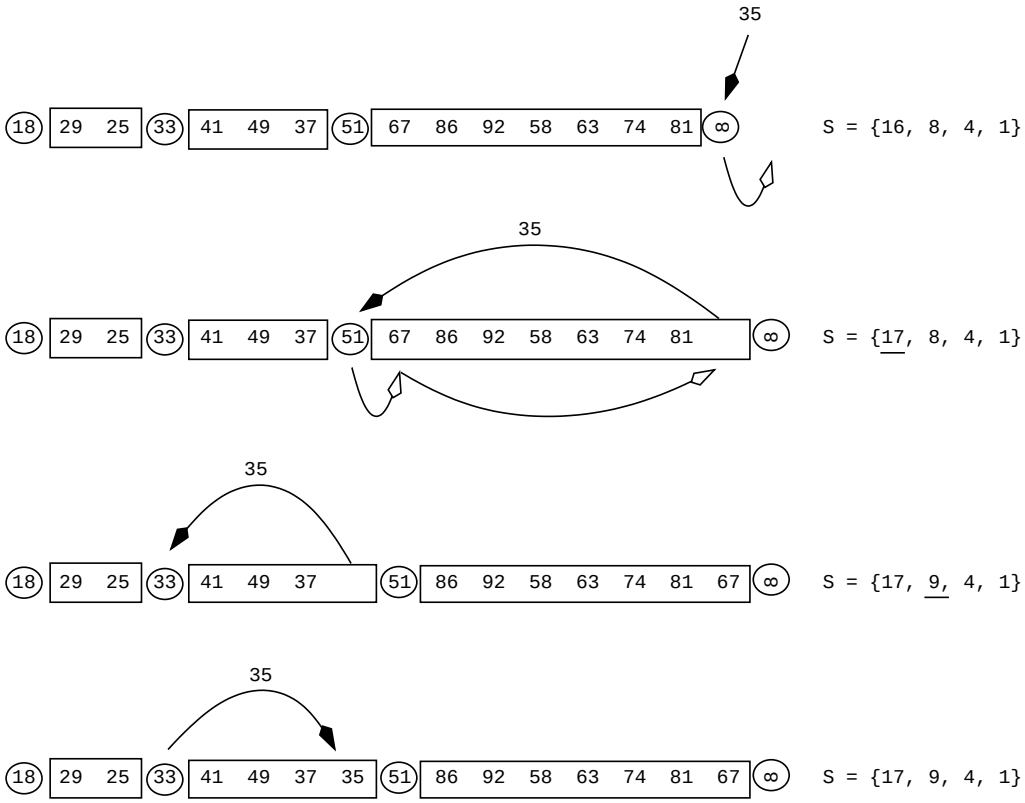


Figure 3.11: Inserting a new element into a quickheap. The figure shows the pivots we have to compare with the new element to insert it into the quickheap, and the elements we have to move to create the free cell to allocate the new element.

---

add(Elem x, Index $pidx$)

1. **While** TRUE **Do** // moving pivots, starting from pivot $S[pidx]$
2. $\quad heap[(S[pidx]+1) \bmod capacity] \leftarrow heap[S[pidx] \bmod capacity]$
3. $\quad S[pidx] \leftarrow S[pidx]+1$
4. $\quad$ **If** $(|S| = pidx+1)$ OR // we are in the first chunk
   $\quad (heap[S[pidx+1] \bmod capacity] \leq x)$ **Then** // we found the chunk
5. $\quad\quad heap[(S[pidx]-1) \bmod capacity] \leftarrow x$, **Return**
6. $\quad$ **Else**
7. $\quad\quad heap[(S[pidx]-1) \bmod capacity] \leftarrow heap[(S[pidx+1]+1) \bmod capacity]$
8. $\quad\quad pidx \leftarrow pidx+1$ // go to next chunk

---

insert(Elem x)

1. **add**(x, 0)

Figure 3.12: Inserting elements to a quickheap.

we process the next chunk at the right. We continue until we reach the fictitious pivot. Figure 3.13 shows the methods **findChunk** and **delete**.

```
findChunk(Index pos)
1.  pidx ← 0 // start in chunk 0
2.  While  pidx < |S| AND S[pidx] ≥ pos Do pidx ← pidx + 1
3.  Return pidx − 1 // come back to the last reviewed pivot
```

```
delete(Index pos)
1.  pidx ← findChunk(pos) // chunk id
2.  If S[pidx] = pos Then
3.      S.extractElement(pidx) // we extract the pidx-th pivot from S
4.      pidx ← pidx − 1 // we go back one pivot, that is, go forward in heap
    // moving pivots and elements to the rear of the quickheap,
    // starting from pivot S[pidx]
5.  For i ← pidx  downto 0 Do
        // moving the last element of the chunk
6.      heap[pos mod capacity] ← heap[(S[i] − 1) mod capacity]
7.      heap[(S[i] − 1) mod capacity] ← heap[(S[i]) mod capacity] // pivot movement
8.      S[i] ← S[i] − 1 // updating pivot position
9.      pos ← S[i] + 1 // updating position pos
```

Figure 3.13: Deleting elements from a quickheap.

Note that, if the element at position *pos* is originally a pivot, we extract it from S —by moving every pivot in the stack one position towards the bottom starting from the deleted pivot to the top of the stack— and go back to the previous pivot, so we always have a pivot at a position greater than *pos*. Thus, extracting a pivot effectively merges the two chunks at the left and right of the removed pivot.

Decreasing and increasing a key can be done via a delete plus insert operations. Nevertheless, in the next two sections we show a more efficient direct implementation.

3.3.8 Decreasing a Key

This operation consists in, given a position *pos* of some element in the quickheap and a value $\delta \geq 0$, changing the value of the element $heap[pos]$ to $heap[pos] - \delta$, and adjusting its position in the quickheap so as to preserve the pivot invariant (Lemma 3.1). As we are decreasing the key, the modified element either stays in its current place or it moves chunk-wise towards position *idx*. Thus operation **decreaseKey** is similar to operation **insert**, in the sense that both of them use the auxiliary method **add**.

To decrease a key, we first need to find the chunk *pidx* of the element to modify. We use procedure **findChunk** for this. If the element at position *pos* is a pivot, we extract it

from S and go back to the previous pivot, so we always have a pivot at a position greater than pos.

Let $newValue = heap[pos] - \delta$ be the resulting value of the modified element. Once we have a pivot $pidx$ at a position greater than pos, we do the following. If we are working in the first chunk, that is $|S| = pidx + 1$, we update the element $heap[pos]$ to $newValue$ and we are done. Otherwise, we check whether $newValue$ is greater than or equal to the preceding pivot ($heap[S[pidx + 1]]$). If so, we update the element $heap[pos]$ to $newValue$ and we have finished. Else, we place the element at the right of the next pivot in the current position of the element. That is, we move the element $heap[S[pidx + 1] + 1]$ to position $heap[pos]$. As we have an empty space next to the pivot delimiting the preceding chunk, we start the pivot movement procedure from that chunk. That is, we call procedure **add**$(newValue, pidx + 1)$. Figure 3.14 gives method **decreaseKey**.

```
decreaseKey(Index pos, Decrement δ)
1.  pidx ← findChunk(pos) // chunk id
2.  If S[pidx] = pos Then
3.      S.extractElement(pidx) // we extract the pidx-th pivot from S
4.      pidx ← pidx − 1 // we go one pivot back
5.  newValue ← heap[pos mod capacity] − δ // computing the new element
6.  If (|S| = pidx + 1) OR // we are in the first chunk
       (heap[S[pidx + 1] mod capacity] ≤ newValue) Then // we found the chunk
7.      heap[pos mod capacity] ← newValue, Return
8.  Else // creating an empty cell next to the preceding pivot
9.      heap[pos mod capacity] ← heap[(S[pidx + 1] + 1) mod capacity]
10.     add(newValue, pidx + 1)
```

Figure 3.14: Decreasing a key in a quickheap.

3.3.9 Increasing a Key

Analogously, given a position pos of some element in the quickheap and a value $\delta \geq 0$, this operation changes the value of the element $heap[pos]$ to $heap[pos] + \delta$, and adjusts its position in the quickheap so as to preserve the pivot invariant. As we are increasing the key, the modified element either stays in its current place or moves chunk-wise towards position $S[0]$. Thus, operation **increaseKey** is similar to operation **delete**, but without removing the element.

To increase the key, we first need to find the chunk $pidx$ of the element to modify. Once again, we use procedure **findChunk**. If the element at position pos is a pivot, we remove it from the stack S and go back to the previous pivot, so we have a pivot in a position greater than pos. The operation is symmetric with **decreaseKey**. Figure 3.15 shows method **increaseKey**.

```
increaseKey(Index pos, Increment δ)
1.  pidx ← findChunk(pos) // chunk id
2.  If S[pidx] = pos Then
3.      S.extractElement(pidx) // we extract the pidx-th pivot from S
4.      pidx ← pidx − 1 // we go one pivot back
5.  newValue ← heap[pos mod capacity] + δ // computing the new element
    // moving pivots and elements to the rear of the quickheap,
    // starting from pivot S[pidx]
6.  While (pidx > 0) AND newValue ≥ heap[S[pidx] mod capacity] Do
        // moving the last element of the chunk
7.      heap[pos mod capacity] ← heap[(S[pidx] − 1) mod capacity]
        // moving the pivot
8.      heap[(S[pidx] − 1) mod capacity] ← heap[(S[pidx]) mod capacity]
9.      S[pidx] ← S[pidx] − 1 // updating pivot position
10.     pos ← S[pidx] + 1 // updating the position pos
11.     pidx ← pidx − 1 // decrementing the chunk id
12. heap[pos mod capacity] ← newValue
```

Figure 3.15: Increasing a key in a quickheap.

3.3.10 Further Comments on Quickheaps

Throughout this section we have assumed that we know beforehand the value of *capacity*, that is, the maximum number of elements we store in the priority queue. However, this is not always possible, but it is not an issue at all. In fact, in order to implement a quickheap with variable capacity, it is enough to implement array *heap* as a *Dynamic Table* [CLRS01, Section 17.4], just adding a constant amortized factor to the cost of quickheap operations.

Finally, we note that by using a variant of **IQS** which performs incremental search from arbitrary elements (not from the first) we can implement operations **successor** and **predecessor** over the quickheap. However, these operations have the inconvenient that they can enlarge the stack out of control, so they can increase the cost of other quickheap operations. In the following we sketch both operations.

Finding the Successor. This operation consists in, given an element x, finding the smallest element y in *heap* such that $y > x$. To do this, given the element x we look for its chunk (by checking decreasingly the pivots in S), and then use x to partition it. Finally, we call **IQS** to get the element immediately larger than x. (If x turns out to be a pivot, we do not partition but just call **IQS** in its right chunk.) If $x \geq \max(heap)$, **successor** answers NULL. Note that, by partitioning the chunk of x, we generate a new stack of pivots S'. We can merge both stacks S and S' to reuse the work for next calls to **successor**, or simply neglect the stack S' (as we could double the size of S if we added the resulting pivots when partitioning a large chunk, and this slows down other operations).

Finding the Predecessor. This operation consists in, given an element x, finding the largest element y in *heap* such that $y < x$. Analogously to operation **succesor**, given the element x we look for its chunk, and then use x to partition it. Let **DQS** (from *Decremental Quicksort*) be the algorithm that mirrors **IQS** finding largest elements. **DQS** can be trivially derived from **IQS**, considering that the stack stores pivots in increasing order and the fictitious pivot is placed just before the array starts. Therefore, we call **DQS** to get the element immediately smaller than x. (If x turns out to be a pivot, we do not partition but just call **DQS** in its left chunk.) If $x \leq \min(heap)$, **predecessor** answers NULL. Once again, this means to partition a chunk and generate a new auxiliary stack S' (which is in reverse order), which we can merge with S or not.

3.4 Analysis of Quickheaps

> Denken ist immer eine schlimme Sache. [Thinking is always a bad thing.]
>
> – Oskar Panizza

This analysis is based on a key observation: quickheaps follow a *self-similar structure*, which means that the distribution of elements within a quickheap seen from the last chunk towards the first chunk is the same as the distribution within such quickheap seen from the second last chunk towards the first chunk, and so on. We start by proving that self-similarity property. Then, we introduce the *potential debt method* for amortized analysis. Finally, exploiting the self-similarity property, we analyze quickheaps using the potential debt method.

3.4.1 The Quickheap's Self-Similarity Property

In this section we introduce a formal notion of the self-similar structure of quickheaps. We show that this property is true at the beginning, and that it holds after extractions of minima, as well as insertions or deletions of elements that fall at independent and uniformly distributed positions in the heap. It follows that the property holds after arbitrary sequences of those operations, yet the positions of insertions and deletions cannot be arbitrary but uniformly distributed.

From now on, we consider that *array segments* are delimited by idx and the cell just before each pivot position $S[pidx]$ ($heap[idx \dots S[pidx] - 1]$, thus segments overlap), and *array chunks* are composed by the elements between two consecutive pivot positions ($heap[S[pidx] + 1 \dots S[pidx - 1] - 1]$) or between idx and the cell preceding the pivot on top of S ($heap[idx \dots S.\mathbf{top}()-1]$). We call $heap[idx \dots S.\mathbf{top}()-1]$ the first chunk, and $heap[S[1] + 1 \dots S[0] - 1]$ the last chunk. Analogously, we call $heap[idx \dots S.\mathbf{top}()-1]$ the first segment, and $heap[idx \dots S[0] - 1]$ the last segment. The *pivot of a segment* will be

the rightmost pivot within such segment (this is the one used to split the segment at the time **partition** was called on it). Thus, the pivot of the last segment is $S[1]$, whereas the first segment is the only one not having a pivot. Figure 3.16 illustrates.

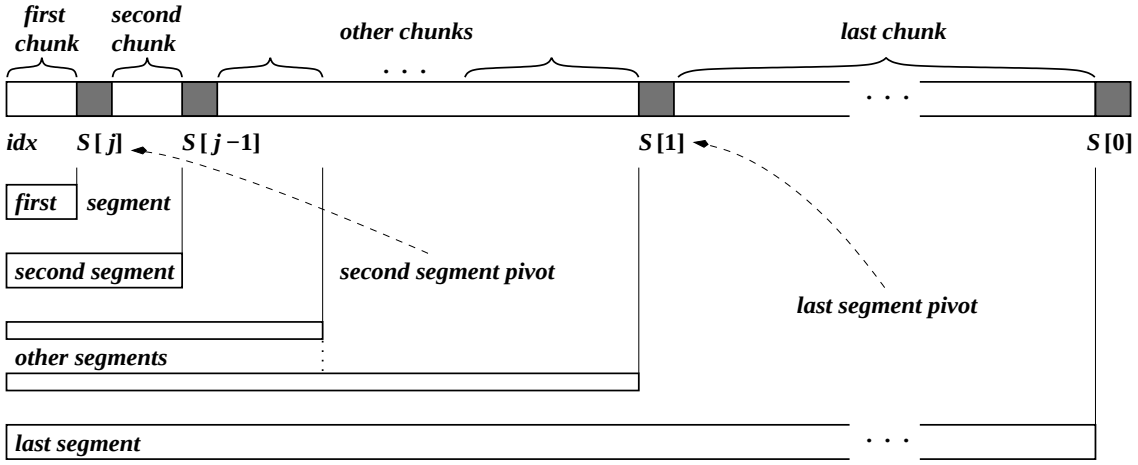


Figure 3.16: Segments and chunks of a quickheap.

Using the traditional definition of the median of a n-element set —if n is odd the median is the $\frac{n+1}{2}$-th element, else it is the average of the values at positions $\frac{n}{2}$ and $\frac{n}{2}+1$ [Ros04, page 20]—, let us call an element not smaller than the median of the array segment $heap[idx \ldots S[pidx]-1]$ a *large element* of such segment. Analogously, let us call an element smaller than the median a *small element*.

The self-similarity property is formally defined as follows:

Definition 3.1 (quickheap's self-similarity property)**.** *The probability that the pivot of each array segment $heap[idx \ldots S[pidx]-1]$ is large in its segment is smaller than or equal to $\frac{1}{2}$. That is, for all the segments $\mathbb{P}(pivot\ is\ large) \leq \frac{1}{2}$.*

To prove the property we need some notation. Let $\mathbb{P}_{i,j,n}$, $1 \leq i \leq n$, $j \geq 0$, $n > 0$, be the probability that the i-th element of a given segment of size n is the pivot of the segment after the j-th operation ($\mathbb{P}_{i,j,n} = 0$ outside bounds). In the following we prove by induction on j that $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$, for all j, n and $2 \leq i \leq n$, after performing any sequence of operations **insert**, **delete**, **findMin** and **extractMin**. That is, the probability of the element at cell i being the pivot is non-increasing from left to right. Later, we use this to prove the self-similar property and some consequences of it.

Note that new segments with pivots are created when operations **extractMin** or **findMin** split the first segment. Note also that, just after a partitioned segment is created, the probabilities are $\mathbb{P}_{i,0,n} = \frac{1}{n}$, because the pivot is chosen at random from it, so we have proved the base case.

Lemma 3.2. *For each segment, the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after inserting a new element x at a position uniformly chosen in $[1, n]$.*

Proof. We suppose that after the $(j-1)$-th operation the segment has $n-1$ elements. As we insert x in the j-th operation, the resulting segment contains n elements. The probability that after the insertion the pivot p is at cell i depends on whether p was at cell $i-1$ and we have inserted x at any of the first $i-1$ positions $1,\ldots,i-1$, so the pivot moved to the right; or the pivot was already at cell i and we have inserted x at any of the last $n-i$ positions $i+1,\ldots,n$. So, we have the recurrence of Eq. (3.17).

$$\mathbb{P}_{i,j,n} \;=\; \mathbb{P}_{i-1,j-1,n-1}\frac{i-1}{n} \;+\; \mathbb{P}_{i,j-1,n-1}\frac{n-i}{n} \tag{3.17}$$

From the inductive hypothesis we have that $\mathbb{P}_{i,j-1,n-1} \leq \mathbb{P}_{i-1,j-1,n-1}$. Multiplying both sides by $\frac{n-i}{n}$, adding $\mathbb{P}_{i-1,j-1,n-1}\frac{i-1}{n}$ and rearranging terms we obtain the inequality of Eq. (3.18), whose left side corresponds to the recurrence of $\mathbb{P}_{i,j,n}$.

$$\mathbb{P}_{i-1,j-1,n-1}\frac{i-1}{n}+\mathbb{P}_{i,j-1,n-1}\frac{n-i}{n} \;\leq\; \mathbb{P}_{i-1,j-1,n-1}\frac{i-2}{n}+\mathbb{P}_{i-1,j-1,n-1}\frac{n+1-i}{n} \tag{3.18}$$

By the inductive hypothesis again, $\mathbb{P}_{i-1,j-1,n-1} \leq \mathbb{P}_{i-2,j-1,n-1}$, for $i > 2$. So, replacing on the right side above we obtain the inequality of Eq. (3.19), where, in the right side we have the recurrence for $\mathbb{P}_{i-1,j,n}$.

$$\mathbb{P}_{i,j,n} \;\leq\; \mathbb{P}_{i-2,j-1,n-1}\frac{i-2}{n}+\mathbb{P}_{i-1,j-1,n-1}\frac{n+1-i}{n} \;=\; \mathbb{P}_{i-1,j,n} \tag{3.19}$$

With respect to $i = 2$, note that the term $\frac{i-2}{n}$ from Eqs. (3.18) and (3.19) vanishes, so the replacement made for $i > 2$ holds anyway. Thus, this equation can be rewritten as $\mathbb{P}_{2,j,n} \leq \mathbb{P}_{1,j-1,n-1}\frac{n-1}{n}$. Note that the right side is exactly $\mathbb{P}_{1,j,n}$ according to the recurrence Eq. (3.17) evaluated for $i = 1$. □

Lemma 3.3. *For each segment, the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after deleting an element chosen uniformly from $[1, n+1]$.*

Proof. Suppose that after the $(j-1)$-th operation the segment has $n+1$ elements. As we delete an element in the j-th operation, the resulting segment contains n elements.

We start by proving the property when the deleted element is not a pivot. The probability that after the deletion the pivot p is at cell i depends on whether p was at cell $i+1$ and we delete an element from positions $1,\ldots,i$, so the pivot moved to the left; or the pivot was already at cell i, and we have deleted from the last $n+1-i$ elements $i+1,\ldots,n+1$. So, we have the recurrence of Eq. (3.20).

$$\mathbb{P}_{i,j,n} \;=\; \mathbb{P}_{i+1,j-1,n+1}\frac{i}{n+1} \;+\; \mathbb{P}_{i,j-1,n+1}\frac{n+1-i}{n+1} \tag{3.20}$$

From the inductive hypothesis we have that $\mathbb{P}_{i,j-1,n+1} \leq \mathbb{P}_{i-1,j-1,n+1}$. Multiplying both sides by $\frac{n+2-i}{n+1}$, adding $\mathbb{P}_{i,j-1,n+1}\frac{i-1}{n+1}$ and rearranging terms we obtain the inequality of Eq. (3.21), whose right side corresponds to the recurrence of $\mathbb{P}_{i-1,j,n}$.

$$\mathbb{P}_{i,j-1,n+1}\frac{i}{n+1}+\mathbb{P}_{i,j-1,n+1}\frac{n+1-i}{n+1} \;\leq\; \mathbb{P}_{i,j-1,n+1}\frac{i-1}{n+1}+\mathbb{P}_{i-1,j-1,n+1}\frac{n+2-i}{n+1} \tag{3.21}$$

By the inductive hypothesis again, $\mathbb{P}_{i+1,j-1,n+1} \leq \mathbb{P}_{i,j-1,n+1}$, so we can replace the first term above to obtain the inequality of Eq. (3.22), where in the left side we have the recurrence for $\mathbb{P}_{i,j,n}$. On the right we have $\mathbb{P}_{i-1,j,n}$.

$$\mathbb{P}_{i+1,j-1,n+1}\frac{i}{n+1} + \mathbb{P}_{i,j-1,n+1}\frac{n+1-i}{n+1} \quad = \quad \mathbb{P}_{i,j,n} \quad \leq \quad \mathbb{P}_{i-1,j,n} \tag{3.22}$$

In the case of deleting a pivot p we have the following. If we delete the pivot on top of S, then the first and the second chunk get merged and the lemma does not apply to the (new) first segment because it has no pivot.

Otherwise, we must have (at least) two pivots p_l and p_r at the left and right of p. Let pos_l, pos and pos_r be the positions of the pivots p_l, p, p_r before deleting p, respectively. Figure 3.17 illustrates. Note that p_l and p are pivots of segments $heap[idx \ldots pos-1]$ and $heap[idx \ldots pos_r - 1]$ with n' and n elements ($n' < n$), respectively.

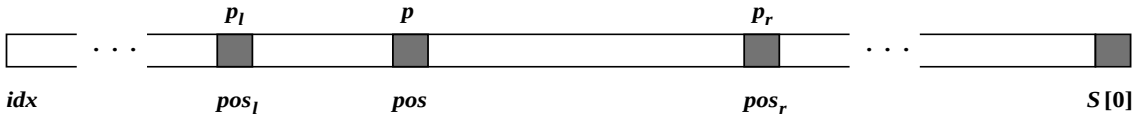


Figure 3.17: Deleting an inner pivot of a quickheap.

Once we delete pivot p, the segment $heap[idx \ldots pos-1]$ is "extended" to position $pos_r - 2$ (as we have one cell less). As the $n - n' - 1$ new elements in the extended segment were outside of the old segment $heap[idx \ldots pos-1]$, they cannot be the pivot in the extended segment. On the other hand, the probabilities of the old segment elements holds in the new extended segment. Therefore, for each $idx \leq i < pos$, $\mathbb{P}_{i,j,n} = \mathbb{P}_{i,j-1,n'}$, and for each $pos \leq i < pos_r - 2$, $\mathbb{P}_{i,j,n} = 0$. Thus the invariant is maintained. □

In order to analyze whether the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ is preserved after operations **findMin** and **extractMin** we need consider how **IQS** operates on the first segment. For this sake we introduce operation **pivoting**, which partitions the first segment with a pivot and pushes it into stack S. We also introduce operation **takeMin**, which increments idx, pops stack S and returns element $heap[idx-1]$.

Using these operations, we rewrite operation **extractMin** as: execute **pivoting** as many times as we need to push idx in stack S and next perform operation **takeMin**. Likewise, we rewrite operation **findMin** as: execute **pivoting** as many times as we need to push idx in stack S and next return element $heap[idx]$.

Operation **pivoting** creates a new segment and converts the previous first segment (with no pivot) into a segment with a pivot, where all the probabilities are $\mathbb{P}_{i,0,n} = \frac{1}{n}$. The next lemma shows that the property also holds after taking the minimum.

Lemma 3.4. *For each segment, the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after taking the minimum element of the quickheap.*

Proof. Due to previous calls to operation **pivoting**, the minimum is the pivot placed in idx. Once we pick it, the first segment vanishes. After that, the new first segment may be empty, but all the others have elements. For the empty segment the property is true by vacuity. Else, within each segment probabilities change as follows: $\mathbb{P}_{i,j,n} = \mathbb{P}_{i+1,j-1,n+1}\frac{n+1}{n}$. □

Finally, we are ready to prove the quickheap's self-similarity property.

Theorem 3.3 (quickheap's self-similarity property)**.** *Given a segment* $heap[idx \dots S[pidx] - 1]$*, the probability of that its pivot is large is smaller than or equal to* $\frac{1}{2}$*, that is,* $\mathbb{P}(pivot\ is\ large) \leq \frac{1}{2}$*.*

Proof. When the segment is created, all the probabilities are $\mathbb{P}_{i,j,n} = \frac{1}{n}$. Lemmas 3.2 to 3.4 guarantee that the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after inserting or deleting elements, or taking the minimum. So, the property is preserved after any sequence of operations **insert**, **delete**, **findMin** and **extractMin**. Therefore, adding up the probabilities $\mathbb{P}_{i,j,n}$ for the large elements, that is, for the $\left(\lceil \frac{n}{2} \rceil + 1\right)$-th to the n-th element, we obtain that $\mathbb{P}(pivot\ is\ large) = \sum_{i=\lceil \frac{n}{2} \rceil + 1}^{n} \mathbb{P}_{i,j,n} \leq \frac{1}{2}$. □

In the following, we use the self-similarity property to show two additional facts we use in the analysis of quickheaps. They are (i) the height of stack S is $O(\log m)$, and (ii) the sum of the size of the array segments is $\Theta(m)$.

Lemma 3.5. *The expected value of the height* $\mathcal{H}$ *of stack* S *is* $O(\log m)$*.*

Proof. Notice that the number $\mathcal{H}$ of pivots in the stack is monotonically nondecreasing with m. Let us make some pessimistic simplifications (that is, leading to larger $\mathcal{H}$). Let us take the largest value of the probability $\mathbb{P}(pivot\ is\ large)$, which is $\frac{1}{2}$. Furthermore, let us assume that if the pivot is taken from the large elements then it is the maximum element. Likewise, if it is taken from the small elements, then it is the element immediately previous to the median.

With these simplifications we have the following. When partitioning, we add one pivot to stack S. Then, with probabilities $\frac{1}{2}$ and $\frac{1}{2}$ the left partition has $m - 1$ or $\lfloor \frac{m}{2} \rfloor$ elements. So, we write the following recurrence: $\mathcal{H} = T(m) = 1 + \frac{1}{2}T(m-1) + \frac{1}{2}T\left(\lfloor \frac{m}{2} \rfloor\right)$, $T(1) = 1$. Once again, using the monotonicity on the number of pivots, the recurrence is simplified to $T(m) \leq 1 + \frac{1}{2}T(m) + \frac{1}{2}T\left(\frac{m}{2}\right)$, which can be rewritten as $T(m) \leq 2 + T\left(\frac{m}{2}\right) \leq \ldots \leq 2j + T\left(\frac{m}{2^j}\right)$. As $T(1) = 1$, choosing $j = \log_2(m)$ we obtain that $\mathcal{H} = T(m) \leq 2\log_2 m + 1$. Finally, adding the fictitious pivot we have that $\mathcal{H} = 2(\log_2 m + 1) = O(\log m)$. □

Lemma 3.6. *The expected value of the sum of the sizes of array segments is* $\Theta(m)$*.*

Proof. Using the same reasoning of Lemma 3.5, but considering that when partitioning the largest segment has m elements, we write the following recurrence:

$T(m) = m + \frac{1}{2}T(m-1) + \frac{1}{2}T\big(\lfloor \frac{m}{2} \rfloor\big)$, $T(1) = 0$. Using the monotonicity of $T(m)$ (which also holds in this case) the recurrence is simplified to $T(m) \leq m + \frac{1}{2}T(m) + \frac{1}{2}T\big(\frac{m}{2}\big)$, which can be rewritten as $T(m) \leq 2m + T\big(\frac{m}{2}\big) \leq \ldots \leq 2m + m + \frac{m}{2} + \frac{m}{2^2} + \ldots + \frac{m}{2^{j-2}} + T\big(\frac{m}{2^j}\big)$. As $T(1) = 0$, choosing $j = \log_2(m)$ we obtain that $T(m) \leq 3m + m\sum_{i=1}^{\infty} \frac{1}{2^i} \leq 4m = \Theta(m)$. Therefore, the expected value of the sum of the array segment sizes is $\Theta(m)$. □

3.4.2 The Potential Debt Method

To carry out the amortized analysis of quickheaps we use a slight variation of the potential method ([Tar85] and [CLRS01, Chapter 17]), which we call the *potential debt method*. In Section 2.1.5 (page 15) we describe the standard potential method.

In the case of the potential debt method, the potential function represents a total cost that has not yet been paid. At the end, this total debt must be split among all the performed operations. The potential debt is associated with the data structure as a whole.

The potential debt method works as follows. It starts with an initial data structure D_0 on which operations are performed. Let c_i be the actual cost of the i-th operation and D_i the data structure that results from applying the i-th operation to D_{i-1}. A *potential debt function* Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential debt associated with data structure D_i up to then. The *amortized cost* $\widetilde{c_i}$ of the i-th operation with respect to potential debt function Φ is defined by

$$\widetilde{c_i} \quad = \quad c_i - \Phi(D_i) + \Phi(D_{i-1}) \; . \tag{3.23}$$

Therefore, the amortized cost of i-th operation is the actual cost minus the increase of potential debt due to the operation. Thus, the total amortized cost for N operations is

$$\sum_{i=1}^{N} \widetilde{c_i} \quad = \quad \sum_{i=1}^{N} (c_i - \Phi(D_i) + \Phi(D_{i-1})) \quad = \quad \sum_{i=1}^{N} c_i - \Phi(D_N) + \Phi(D_0) \; . \tag{3.24}$$

If we define a potential function Φ so that $\Phi(D_N) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{N} \widetilde{c_i}$ is a lower bound on the total actual cost $\sum_{i=1}^{N} c_i$. However, if we sum the positive cost $\Phi(D_N) - \Phi(D_0)$ to the amortized cost $\sum_{i=1}^{N} \widetilde{c_i}$, we compensate for the debt and obtain an upper bound on the actual cost $\sum_{i=1}^{N} c_i$. That is, at the end we share the debt among the operations. Thus, in Eq. (3.25) we write an amortized cost $\widehat{c_i}$ considering the potential debt, by assuming that we perform N operations during the process, and the potential due to these operations is $\Phi(D_N)$.

$$\widehat{c_i} = \widetilde{c_i} + \frac{\Phi(D_N) - \Phi(D_0)}{N} = c_i - \Phi(D_i) + \Phi(D_{i-1}) + \frac{\Phi(D_N) - \Phi(D_0)}{N} \tag{3.25}$$

This way, adding up for all the N operations, we obtain that $\sum_{i=1}^{N} \widehat{c_i} = \sum_{i=1}^{N} \Big(c_i - \Phi(D_i) + \Phi(D_{i-1}) + \frac{\Phi(D_N) - \Phi(D_0)}{N}\Big) = \sum_{i=1}^{N} c_i$.

Intuitively, if the potential debt difference $\Phi(D_{i-1}) - \Phi(D_i)$ of the i-th operation is positive, then the amortized cost $\widehat{c_i}$ represents an overcharge to the i-th operation, as it also pays part of the potential debt of the data structure, so that the potential debt decreases. On the other hand, if the potential debt difference is negative, then the amortized cost represents an undercharge to the i-th operation, as the operation increases the potential debt, which will be paid by future operations.

3.4.3 Expected-case Amortized Analysis of Quickheaps

In this section, we consider that we operate over a quickheap qh with m elements within *heap* and a pivot stack S of expected height $\mathcal{H} = O(\log m)$, see Lemma 3.5.

We define the quickheap potential debt function as the sum of the sizes of the partitioned segments delimited by idx and pivots in $S[0]$ to $S[\mathcal{H}-1]$ (note that the smallest segment is not counted). Eq. (3.26) shows the potential function $\Phi(qh)$. Figure 3.18 illustrates.

$$\Phi(qh) = \sum_{i=0}^{\mathcal{H}-1} (S[i] - idx) = \Theta(m) \text{ expected, by Lemma 3.6} \tag{3.26}$$

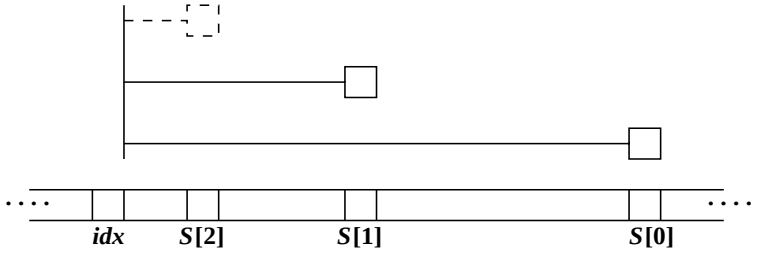


Figure 3.18: The quickheap potential debt function is computed as the sum of the lengths of the partitioned segments (drawn with solid lines) delimited by idx and pivots in $S[0]$ to $S[\mathcal{H}-1]$. In the figure, $\Phi(qh) = S[0] + S[1] - 2idx$.

Thus, the potential debt of an empty quickheap $\Phi(qh_0)$ is zero, and the expected potential debt of a m-elements quickheap is $\Theta(m)$, see Lemma 3.6. Note that if we start from an empty quickheap qh, for each element within qh we have performed at least operation **insert**, so we can assume that there are more operations than elements within the quickheap. Therefore, in the case of quickheaps, the term $\frac{\Phi(qh_N) - \Phi(qh_0)}{N}$ is $O(1)$ expected. So, we can omit this term, writing the amortized costs directly as $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1})$.

Operation insert. The amortized cost of operation **insert** is defined by $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1})$. The difference of the potential debt $\Phi(qh_{i-1}) - \Phi(qh_i) (< 0)$ depends on how many segments are extended (by one cell) due to the insertion. This

means an increase in the potential debt (and also an undercharge in the amortized cost of operation **insert**). Note that for each segment we extend —which increases by 1 the potential debt—, we also pay one key comparison, but there is no increase associated to the last key comparison. Thus, it holds $c_i - \Phi(qh_i) + \Phi(qh_{i-1}) \leq 1$, which means that almost all the cost is absorbed by the increase in the potential debt. Then, the amortized cost of operation **insert** is $O(1)$.

However, we can prove that this is not only the amortized cost, but also the expected (individual) cost of operation **insert**. When inserting an element, we always extend the last segment. Later, with probability $\mathbb{P}_1 \geq \frac{1}{2}$ the position of the inserted element is greater than the position of the pivot $S[1]$ —that is, the element is inserted at the right of the pivot $S[1]$— (from Theorem 3.3), in which case we stop. If not, we compare the pivot of the second last segment, and once again, with probability $\mathbb{P}_2 \geq \frac{1}{2}$ the element is inserted at the right of the pivot $S[2]$, in which case we stop. Else, we compare the third pivot, and this goes on until we stop expanding segments. Thus, the expected number of key comparisons is $1 + (1 - \mathbb{P}_1)(1 + (1 - \mathbb{P}_2)(1 + (1 - \mathbb{P}_3)(1 + \ldots)))$. This sum is upper bounded, by taking the lowest value of $\mathbb{P}_i = \frac{1}{2}$, to $1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \ldots\right)\right)\right) \leq \sum_{i=0}^{\infty}\left(\frac{1}{2}\right)^i = 2 = O(1)$.

Operation delete. The decrease of the potential debt $\Phi(qh_{i-1}) - \Phi(qh_i)(> 0)$ depends on how many segments are contracted (by one cell) due to the deletion. Note that it is also possible to delete a whole segment if we remove a pivot.

The worst case of operation **delete** (without considering pivot deletions) arises when deleting an element in the first chunk. This implies to contract by one cell all the segments, which is implemented by moving all the pivots —whose expected number is $\mathcal{H}$— one cell to the left. So, the actual cost of moving pivots and elements is $\mathcal{H}$. On the other hand, the term $\Phi(qh_{i-1}) - \Phi(qh_i)$, which accounts for the potential decrease due to all the contracted segments, is also $\mathcal{H}$. Thus, the amortized cost is $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1}) = 2\mathcal{H}$. This is $O(\log m)$ expected, see Lemma 3.5.

If, on the other hand, we remove a pivot, we also remove a whole segment, thus decreasing the potential debt. We can delete each of the pivots with probability $\frac{1}{m}$. As the first $\mathcal{H}-1$ pivots in stack S delimit segments which account for the potential debt, we obtain the following sum $\sum_{i=0}^{\mathcal{H}-1} \frac{1}{m}(S[i] - idx) = \frac{1}{m}\Phi(qh)$. As $\Phi(qh) = \Theta(m)$, $\frac{1}{m}\Phi(qh) = \Theta(1)$ expected.

Therefore, the potential debt decrease $\Phi(qh_{i-1}) - \Phi(qh_i)(> 0)$ due to segment contractions and segment deletions is $\Theta(1)$. Considering that every time we contract a segment, we perform $O(1)$ work in pivot and element movements, the expected amortized cost of operation **delete** on pivots is $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1}) \leq 2\mathcal{H} + \Theta(1) = O(\log m)$ expected.

However, we can prove that the individual cost of operation **delete** is actually $O(1)$ expected. We start by analyzing the deletion of non pivot elements with an argument similar to the one used in operation **insert**. When deleting an element, we always contract the last segment. Later, with probability $\mathbb{P}_1 \geq \frac{1}{2}$ the position of the deleted element is

greater than the position of the pivot $S[1]$ (from Theorem 3.3), in which case we stop. If not, we contract the second last segment, and this goes on until we stop contracting segments. Thus, the expected decrement of the potential debt due to segment contractions is $1 + (1 - \mathbb{P}_1)(1 + (1 - \mathbb{P}_2)(1 + (1 - \mathbb{P}_3)(1 + \ldots)))$. This sum is upper bounded, by taking the lowest value of $\mathbb{P}_i = \frac{1}{2}$, to $1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \ldots\right)\right)\right) \leq \sum_{i=0}^{\infty}\left(\frac{1}{2}\right)^i = 2 = O(1)$.

Creation of a quickheap. The amortized cost of constructing a quickheap from scratch is $O(1)$. Instead, the amortized cost of constructing a quickheap from an array A of size m is $O(m)$, as we can see this as a construction from scratch plus a sequence of m element insertions of $O(1)$ amortized cost. Note that the potential debt of the quickheap is zero, as there is only one pivot in S.

If, instead, we do not need to copy the array A but can use it as *heap*, the actual cost c_i is $O(1)$ and the debt is still zero (as there is only one pivot in S). However, this breaks the assumption of having more operations than elements, so even in this case we consider that the creation of a quickheap is a sequence of m insertions, thus it has $O(m)$ amortized cost, yet the potential debt is zero.

Operation extractMin. To analyze this operation, we again use auxiliary operations **pivoting** and **takeMin** (see Section 3.4.1). Thus, we consider that operation **extractMin** is a sequence of zero or more calls to **pivoting**, until pushing *idx* in stack S, and then a single call to **takeMin**.

Each time we call operation **pivoting**, the actual cost corresponds to the size of the first segment, which is not yet accounted in the potential debt. On the other hand, once we push the pivot, the potential debt increases by an amount which is the same of the size of the partitioned segment. Thus, the amortized cost of operation **pivoting** is zero, as all the performed work is absorbed by an increase of the potential debt. With respect to operation **takeMin**, its actual cost is $O(1)$, and the potential debt decreases by $\mathcal{H} - 1$, as all the segments considered in the potential are reduced by one cell after taking the minimum. As the expected value of $\mathcal{H}$ is $O(\log m)$ (see Lemma 3.5), the expected amortized cost of operation **takeMin** is $O(\log m)$. Therefore, adding the amortized cost of **pivoting** and **takeMin** we obtain that the expected amortized cost of operation **extractMin** is $O(\log m)$.

Note that when we extract the minimum from the quickheap, all the segments are reduced by 1, so the variation of the potential is negative. However, we cannot reach a negative potential because we cannot perform many times operation **extractMin** without performing operation **pivoting**, as we need to ensure that the minimum element is in place *idx*, and we call operation **pivoting** for this sake. Note also that operation **pivoting** restitutes the potential as it creates segments.

Operation findMin. Using operation **pivoting**, we rewrite operation **findMin** as: execute **pivoting** as many times as we need to push *idx* in stack S (with amortized

cost zero) and then return element $heap[idx]$ (with constant cost). Then, the amortized cost of operation **findMin** is $O(1)$.

Operation increaseKey. This operation is special in the sense that it only moves elements to the rear of the quickheap. Fortunately, it preserves Theorem 3.3. In fact, when we increase the key of some element the involved pivots either stay in their cells or they move to the left. So the probability of that the pivot is large holds or diminishes. Note that operation **increaseKey** can be seen as a sequence of two single calls to operations **delete** and **insert**. However, it is not a sequence of independent operations. In fact, even though the deletion occurs at random, the following insertion does not.

Fortunately, we can still use the argument that in the worst case we increase the key of an element in the first chunk; which implies at most $\mathcal{H}$ movements of elements and pivots. So the actual cost of operation **increaseKey** is $O(\log m)$ expected. On the other hand, the potential variation $\Phi(qh_{i-1}) - \Phi(qh_i) (> 0)$ depends on how many segments are contracted when moving the modified element, which is also at most $\mathcal{H}$. Thus, the amortized cost is $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1}) = 2\mathcal{H}$. This is $O(\log m)$ expected, see Lemma 3.5.

Operation decreaseKey. This is also a special operation. Regrettably, it does not preserve Theorem 3.3. In fact, each time we decrease some key the involved pivots either stay in their cells or they move to the right. Thus, when we perform operation **decreaseKey** the probability of a pivot being large holds or increases, so it could go beyond $\frac{1}{2}$. However, in practice, this operation performs reasonably well as is shown in Sections 3.7.2 and 3.7.4.

To sum up, we have proved the following theorem.

Theorem 3.4 (quickheap's complexity)**.** *The expected amortized cost of any sequence of m operations* **insert***,* **delete***,* **findMin***,* **extractMin** *and* **increaseKey** *over an initially empty quickheap is $O(\log m)$ per operation, assuming that insertions and deletions occur at uniformly random positions. Actually, the individual expected cost of operations* **insert** *and* **delete** *is $O(1)$.*

3.5 Quickheaps in External Memory

On a clear disk you can seek forever.

– P. Denning

Quickheaps exhibit a local access pattern, which makes them excellent candidates to reside on secondary memory. Note, however, that our algorithms are unaware of the disk transfers, so the result of blindly using them on disk is cache-oblivious. Cache obliviousness

[FLPR99, BF03] means that the algorithm is designed for the RAM model but analyzed under the I/O model, assuming an optimal offline page replacement strategy. (This is not unrealistic because there are well-known 2-competitive page replacement strategies.) Cache-oblivious algorithms for secondary memory are not only easier to program than their cache-aware counterparts, but they adapt better to arbitrary memory hierarchies.

The resulting external quickheap allows performing operations **insert**, **findMin** and **extractMin** in expected amortized I/O cost $O((1/B)\log(m/M))$, where B is the block size, M is the total available main memory, and m is the maximum heap size along the process. This result is close to the lower bounds given in [BF03], $\Theta\Big((1/B)\log_{M/B}(m/B)\Big)$, for cache-oblivious sorting. Although there exist optimal cache-oblivious priority queues [Arg95, HMSV97, KS96, FJKT99, BK98, BCFM00] (see Section 2.1.4.1, page 15), quickheaps are, again, a simple and practical alternative.

3.5.1 Adapting Quickheap Operations to External Memory

When considering the basic priority queue operations —namely, **findMin**, **extractMin** and **insert**— one can realize that quickheaps exhibit high locality of reference: First, the stack S is small and accessed sequentially. Second, each pivot in S points to a position in the array *heap*. Array *heap* is only modified at those positions, and the positions themselves increase at most by one at each insertion. Third, **IQS** sequentially accesses the elements of the first chunk. Thus, under the cache-oblivious assumption, we will consider that our page replacement strategy keeps in main memory (Figure 3.19 illustrates):

(i) the stack S and integers *idx* and *capacity*;

(ii) for each pivot in S, the disk block containing its current position in *heap*; and

(iii) the longest possible prefix of $heap[idx, N]$, containing at least two disk blocks.

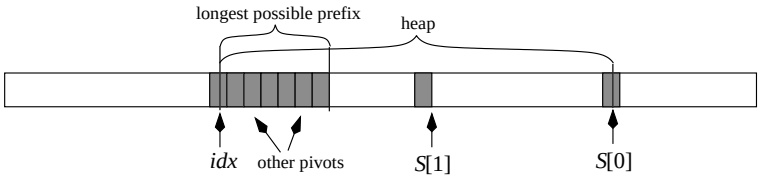


Figure 3.19: External quickheap. The available memory is used both to store the longest possible prefix of *heap* and pages for each pivot not in the prefix.

According to Lemma 3.5, all this requires on average to hold $M = \Omega(B \log m)$ integers in main memory. Say that we have twice the main memory required for (i) and (ii), so that we still have $\Theta(M)$ cells for (iii).

In order to support operations **delete**, **increaseKey** and **decreaseKey** we need a dictionary to know the position of the element to modify. Note that in these operations,

due to element movements, pivot positions can increase or decrease after each operation. To maintain our results on this extended set of operations, it is enough to keep in main memory two blocks per pivot: (1) the block containing the current position of the pivot in *heap*, and (2) the one that previously contained it. This way, even though a pivot moves forward and backward from one page towards the other (note that both pages are consecutive), this does not produce an uncontrolled increase of I/Os. This kind of pivot movement can be produced by, for instance, a sequence of insertions and deletions of a single key.

3.5.2 Analysis of External Memory Quickheaps

We first show a simple approach just considering that we keep in RAM the first two disk blocks of the array. Later, we analyze the effect of a larger prefix of disk blocks cached in internal memory.

3.5.2.1 A Simple Approach

Let us first consider operation **insert**. Assume that entry $heap[i]$ is stored at disk block $\lceil i/B \rceil$. Note that once a disk page is loaded because a pivot position is incremented from $i = B \cdot j$ to $i + 1 = B \cdot j + 1$, we have the disk page $j + 1$ in main memory. From then, at least B increments of pivot position i are necessary before loading another disk page due to that pivot. Therefore, as there are $\mathcal{H}$ pivots, the amortized cost of an element insertion is $\mathcal{H}/B$. According to the results of the previous section, this is $O(\log(m)/B)$ expected.

Operations **findMin** and **extractMin** essentially translate into a sequence of **pivoting** actions. Each such action sequentially traverses $heap[idx, S.\mathbf{top}() - 1]$. Let $\ell = S.\mathbf{top}() - idx$ be the length of the area to traverse. The traversed area spans $1 + \lceil \ell/B \rceil$ disk blocks. As we have in main memory the first two blocks of $heap[idx, N]$, we have to load at most $1 + \lceil \ell/B \rceil - 2 \leq \ell/B$ disk blocks. On the other hand, the CPU cost of such traversal is $\Theta(\ell)$. Hence, each comparison made has an amortized I/O cost of $O(1/B)$. According to the previous section, all those traversals cost $O(\log m)$ amortized expected comparisons. Hence, the amortized I/O cost is $O(\log(m)/B)$ expected. Maintaining this prefix of a given size in main memory is easily done in $O(1/B)$ amortized time per operation, since idx grows by one upon calls to **extractMin**.

To implement operations **delete**, **increaseKey** and **decreaseKey**, we use an auxiliary dictionary to obtain the location of the element we modify. (We are not considering the I/O cost of the dictionary, as it depends on its implementation.) Recall that, in order to support these operations we must maintain two disk blocks per pivot.

Now, let us analyze operation **delete**. This analysis is analogous to the one performed in Section 3.4.3. Assume that we delete the entry x. Using the dictionary we determine that x is stored at $heap[i]$, and hence at block $j' = \lceil i/B \rceil$. This x can be a regular element or a pivot. In the second case we are removing a pivot from S (and deleting a whole

segment), which is I/O-free. If the page j' is not in cache we need an additional I/O when accessing the element at cell i. Then, we move the element preceding the first pivot at the right of entry $heap[i]$ to cell i. This is I/O free, as we already have both blocks in cache (the block just accessed for i and that containing the pivot). Next, the element and pivot movement process repeats towards the end of the heap, moving at most $\mathcal{H}$ pivots one cell to the left. As for **insert**, once we load a disk block because a pivot moved, B further pivot movements are needed to produce a new I/O. Therefore, we conclude that the amortized I/O cost of operation **delete** is $1 + \mathcal{H}/B$, which is $1 + O(\log(m)/B))$ expected. The "1+" can be dropped when deleting elements that are in the prefix, for instance **delete**(idx).

Operation **increaseKey** can be seen as two single calls to operations **delete** and **insert**. Thus, its expected amortized I/O cost is $1+O(\log(m)/B))$. Finally, operation **decreaseKey** does not preserve the quickheap's self-similarity property, so we do not analyze it.

Overall, we achieve $O(\log m/B)$ expected amortized I/O cost, plus an extra access for operation **delete** and **increaseKey**. We now get better bounds by considering an arbitrary size $\Theta(M)$ for the *heap*'s prefix cached in internal memory.

3.5.2.2 Considering the Effect of the Prefix

Let us consider that we have $M' = \Theta(M)$ cells of main memory to store a prefix of array *heap*. Thus, accessing these cells is I/O-free, both when moving pivots or when partitioning segments.

We start by using the potential debt method to analyze the number of key comparisons computed for elements *outside* the prefix for each quickheap operation. Then, we will derive the I/O cost from those values by reusing the arguments of Section 3.5.2.1.

Let us define the potential function $\Psi(qh)$ as *the sum of the sizes of segments minus M' for those longer than M', excepting the shortest segment larger than M'*. Figure 3.20 illustrates. This potential function $\Psi(qh)$ represents the number of cells placed in secondary memory that we have to traverse in future operations.

In this case, the potential debt $\Psi(qh)$ of an empty quickheap $\Psi(qh_0)$ is zero, the potential debt $\Psi(qh)$ of a small heap (that is, where $m \leq M'$) is also zero, and when the heap is big enough (that is, where $m \gg M'$), the expected potential debt $\Psi(qh)$ of an m-element quickheap is $\Theta(m)$, see Lemma 3.6.

Once again, if we start from an empty quickheap qh, for each element within qh we have performed at least operation **insert**, so we can assume that there are more operations than elements within the quickheap. Following the potential debt method, we must share the total debt among all of the operations. However, just as in the previous amortized analysis, the term $\frac{\Psi(qh_N)-\Psi(qh_0)}{N}$ is $O(1)$. Thus, we can omit this term, writing the amortized costs directly as $\widehat{c_i} = c_i - \Psi(qh_i) + \Psi(qh_{i-1})$.

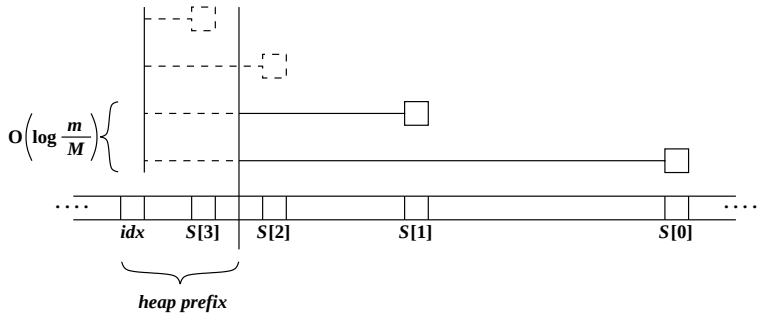


Figure 3.20: The external quickheap I/O potential debt function is computed as the sum of the lengths of the portions outside the *heap* prefix of the partitioned segments (drawn with solid lines). Note that if a segment fits in the prefix it does not contribute to the potential. The same holds, exceptionally, with the shortest segment exceeding the prefix. In the figure, $\Psi(qh) = S[0] + S[1] - 2(idx + M')$.

Due to the self-similarity property (Theorem 3.3), it is not hard to see that on average at least the first $\log_2 M' = \Theta(\log M)$ pivots will be in main memory, and therefore accessing them will be I/O-free. A direct consequence of this is that there are only $O(\log m - \log M) = O(\log(m/M))$ pivots outside the prefix, and only these pivots delimit segments which account in the potential debt $\Psi(qh)$.

Operation insert. The amortized cost of operation **insert** is defined by $\widehat{c_i} = c_i - \Psi(qh_i) + \Psi(qh_{i-1})$. The potential debt variation $\Psi(qh_{i-1}) - \Psi(qh_i)(< 0)$ depends on how many segments larger than M' are extended due to the insertion, which is $O(\log(m/M))$ expected. For each of these segments we extend —increasing by 1 the potential debt—, we also pay one pivot comparison, plus possibly one final comparison that does not expand a segment. Thus, it holds $c_i - \Psi(qh_i) + \Psi(qh_{i-1}) \leq 1$, which means that most of the cost is absorbed by an increase in the potential debt $\Psi(qh)$.

Operation extractMin. Using auxiliary operations **pivoting** and **takeMin** (see Section 3.4.1), we split operation **extractMin** into a sequence of zero or more calls to **pivoting**, and a single call to **takeMin**. Note that operation **pivoting** is I/O-free over the first M' elements of *heap* (as they are cached in main memory).

Each time we call operation **pivoting**, we only consider the key comparisons computed outside the prefix. Note that they correspond exactly with the ones performed on the first segment, which is not included in $\Psi(qh)$, and consequently, those comparisons are not yet accounted for in the potential debt. On the other hand, once we push the pivot, the potential debt $\Psi(qh)$ increases by an amount which is the same as the size of the partitioned segment minus M'. Thus, the amortized cost of the key comparisons performed outside the prefix for operation **pivoting** is zero, as all the performed work is absorbed by an increase of the potential debt $\Psi(qh)$.

With respect to operation **takeMin**, it takes no comparison outside the cached prefix. On the other hand, the potential debt $\Psi(qh)$ decreases by $O(\log(m/M))$, as all the segments considered in the potential are reduced by one cell after taking the minimum. Therefore, the amortized cost of operation **takeMin** is $O(\log(m/M))$.

Operation findMin. We consider that operation **findMin** is a sequence of as many calls to operation **pivoting** as we need to push idx in stack S (with amortized cost zero) and later return element $heap[idx]$ (also with cost zero). Therefore, the amortized cost of operation **findMin** is zero.

Creation of a quickheap. The amortized cost of constructing a quickheap on disk from scratch is $O(1)$. Instead, the amortized cost of constructing a quickheap on disk from an array A of size m is $O(m)$, as we can see this construction as a construction from scratch plus a sequence of m $O(1)$ amortized cost element insertions. Note that the potential debt $\Psi(qh)$ of the quickheap is zero, as there is only one pivot in S.

Operation delete. The variation of the potential debt $\Psi(qh_{i-1}) - \Psi(qh_i)(> 0)$ depends on how many segments outside the cache are contracted due to the deletion, which is $O(\log(m/M))$ expected. For each of these segments we contract —decreasing by 1 the potential debt—, we also pay one pivot comparison, plus possibly one extra comparison. Thus the number of key comparisons performed outside the prefix is $\widehat{c_i} = c_i - \Psi(qh_i) + \Psi(qh_{i-1}) = O(\log(m/M))$.

Operation increaseKey. This operation moves elements to the rear of the quickheap preserving Theorem 3.3. Thus, for the sake of analysis, operation **increaseKey** can be seen as a sequence of two single calls to operations **delete** and **insert**. Thus, the number of comparisons computed outside the cached prefix is $O(\log(m/M))$.

Operation decreaseKey. This operation moves elements to the front of the quickheap, so it does not preserve Theorem 3.3. Thus, we do not analyze it.

Obtaining the I/O costs. Up to this point we have computed the amortized number of key comparisons performed by the quickheap operations outside the cached prefix. Thus, to compute the amortized I/O costs of quickheap operations, we have to take into account how many of those can produce an additional I/O. According to to the analysis of Section 3.5.2.1, there are three cases:

1. When moving a pivot from a disk page which is already in memory towards another page which is not, at least B further pivot movements are necessary before loading another disk page due to that pivot. Note also that the additional movement of the

element beside the pivot is I/O-free, as when the element is moved, both source and target pages reside in main memory.

2. The access to elements inside the segments is sequential. So, the work performed by both element insertions or deletions and operation **partition** is amortized among B consecutive disk accesses. For this to remain true we need, just as in Section 3.5.2.1, that at least two blocks of the prefix are cached, that is, $M' \geq 2B$.

3. When deleting or modifying an element, we have to pay an extra I/O to access the disk page containing it, unless the element is residing inside the cached prefix or any of the page where pivots are placed, for instance, when calling **delete**(idx).

Therefore we have proved the following theorem.

Theorem 3.5 (external quickheap's complexity)**.** *If the* Quickheap *is operated in external memory using an optimal page replacement strategy and holding $M = \Omega(B \log m)$ integers in main memory, where B is the disk block size and m is the maximum heap size along the process; then the expected amortized I/O cost of any sequence of m operations* **insert**, **findMin**, **extractMin**, **delete** *and* **increaseKey** *over an initially empty quickheap is $O((1/B)\log(m/M))$ per operation, assuming that insertions and deletions occur at uniformly random positions. However, operations* **delete** *and* **increaseKey** *need one extra I/O in order to access the element to delete or modify, when its page is not residing in main memory.* □

3.6 Boosting the MST Construction

> In theory, there is no difference between theory and practice; in practice, there is.
>
> – Chuck Reid

As a direct application of **IQS**, we use it to implement Kruskal's MST algorithm. Later, we use **QH**s to implement Prim's MST algorithm. The solutions obtained are competitive with the best current implementations, as we show in Section 3.7.

3.6.1 IQS-based Implementation of Kruskal's MST Algorithm

Recall Section 2.2.4.1 where we explain Kruskal's algorithm. We can use **IQS** in order to incrementally sort E. After initializing C and mst, we create the stack S, and push m into S. Later, inside the **While** loop, we call **IQS** in order to obtain the k-th edge of E incrementally. Figure 3.21 shows our Kruskal's MST variant. Note that the expected number of pivots we store in S is $O(\log m)$ (Section 3.4).

```
Kruskal3 (Graph G(V, E))
1.  UnionFind C ← {{v}, v ∈ V} // the set of all connected components
2.  mst ← ∅ // the growing minimum spanning tree
3.  Stack S, S.push(|E|), k ← 0
4.  While |C| > 1 Do
5.      (e = {u, v}) ← IQS(E, k, S), k ← k + 1 // select the lowest edge incrementally
6.      If C.find(u) ≠ C.find(v) Then
7.          mst ← mst ∪ {e}, C.union(u, v)
8.  Return mst
```

Figure 3.21: Our Kruskal's MST variant (**Kruskal3**). Note the changes in lines 3 and 5 with respect to the basic Kruskal's MST algorithm (Figure 2.9, page 23).

We need $O(n)$ time to initialize both C and mst, and constant time for S. Considering that in the general case we review m' edges, the expected cost of our Kruskal variant is $O(m + m' \log n)$ (Section 3.2.2).

According to the results of [JKŁP93, p. 349], we expect to review $m' = \frac{1}{2} n \ln n + O(n)$ edges within the **While** loop when working on random graphs. Thus we need $O(m + n \log^2 n)$ overall expected time for **IQS** and $O(n\alpha(m, n) \log n)$ time for all the **union** and **find** operations. Therefore, the expected complexity of our Kruskal variant on random graphs is $O(m + n \log^2 n)$, just as Kruskal's algorithm with demand sorting.

3.6.2 Quickheap-based Implementation of Prim's MST Algorithm

Recall Section 2.2.4.2 where we explain Prims's algorithm. We can use a quickheap qh in order to find the minimum node u^* which we add to R and then extract u^* from qh. Next, we check whether we update the values of *cost* for each u^*'s neighbor, and for these nodes we update their values in qh. For the sake of a fast access to the elements within the quickheap, we have to augment the quickheap structure with a dictionary managing the positions of the elements. Figure 3.22 shows our Prim's MST variant.

We need $O(n)$ time to initialize both *cost* and $from$, and constant time to initialize qh. Each call to **insert** and **extactMin** uses $O(1)$ and $O(\log n)$ expected amortized time, respectively. Thus, the n calls to **insert** and **extractMin** use $O(n)$ and $O(n \log n)$ expected time, respectively. Finally, we perform at most m calls to operation **decreaseKey**. Regrettably, we cannot prove an upper bound for operation **decreaseKey**. However, our experimental results suggest that operation **decreaseKey** behaves roughly as $O(\log n)$, so the whole construction process is of the form form $O(m \log n)$.

We have tested **Prim3** on graph with random weights, which is a case where we can improve the obtained bound. Note that we only call operation **decreaseKey** when the new cost $weight_{u^*,v}$ is smaller than $cost_v$. Considering graphs with random weights, for each node, the probability of a fresh random edge being smaller than the current minimum

Prim3 (Graph $G(V, E)$, Vertex s)
1. **For each** $u \in V$ **Do** $cost_u \leftarrow \infty$, $from_u \leftarrow$ NULL
2. $cost_s \leftarrow 0$, $R \leftarrow \emptyset$
   // creating a quickheap qh, elements are of the form (key = cost, $item$ = nodeId),
   // qh is ordered by increasing cost
3. Quickheap $qh(n)$, qh.**insert**$(0, s)$ // $n = |V|$
4. **While** $V \neq R$ **Do**
5. $u^* \leftarrow qh$.**extractMin**().nodeId
6. $R \leftarrow R \cup \{u^*\}$
7. **For each** $v \in (V - R) \cap adjacency(u^*)$ **Do**
8. **If** $weight_{u^*,v} < cost_v$ **Then**
9. **If** $cost_v = \infty$ **Then** qh.**insert**$(weight_{u^*,v}, v)$
10. **Else** qh.**decreaseKey**$(cost_v - weight_{u^*,v}, v)$
11. $cost_v \leftarrow weight_{u^*,v}$
12. $from_v \leftarrow u^*$
13. **Return** $(cost, from)$

Figure 3.22: Our Prim's MST variant (**Prim3**). With respect to the basic Prim's MST algorithm (Figure 2.11, page 25), we have added a quickheap to find the node u^*.

after reviewing k edges incident on it is $\frac{1}{k}$. The number of times we find a smaller cost obeys the recurrence $T(1) = 1$ (the base case, when we find v in the first time), and $T(k) = T(k-1) + \frac{1}{k} = \ldots = H_k = O(\log k)$. As each node has $\frac{m}{n}$ neighbors on average and for the convexity of the logarithm, we expect to find $O\left(\log \frac{m}{n}\right)$ minima per node. So, adding for the n nodes, we expect to call $O\left(n \log \frac{m}{n}\right)$ times operation **decreaseKey**.

Assuming that each call to **decreaseKey** has cost $O(\log n)$, we conjecture that this accounts for a total $O\left(n \log n \log \frac{m}{n}\right)$ expected time, adding up a conjectured $O\left(m + n \log n \log \frac{m}{n}\right)$ expected amortized time for **Prim3** on graphs with random weights.

3.7 Experimental Results

> La science ne sert qu'à vérifier les découvertes de l'instinct.
> [Science only serves to verify the discoveries of instinct.]
>
> – Jean Cocteau

We ran four experimental series. In the first we compare **IQS** with other alternatives. In the second we study the empirical behavior of **QH**s. In the third we study the behavior of **QH**s in secondary memory. Finally, in the fourth we evaluate our MST variants. The experiments were run on an Intel Pentium 4 of 3 GHz, 4 GB of RAM and local disk, running Gentoo Linux with kernel version 2.6.13. The algorithms were coded in C++,

and compiled with `g++` version 3.3.6 optimized with `-O3`. For each experimental datum shown, we averaged over 50 repetitions. The weighted least square fittings were performed with R [R D04]. In order to illustrate the precision of our fittings, we also show the average percent error of residuals with respect to real values $\left(\left|\frac{y-\hat{y}}{y}\right| 100\%\right)$ for fittings belonging to around 45% of the largest values[2].

3.7.1 Evaluating IQS

For shortness we have called the classical Quickselect + Quicksort solution **QSS**, and the Partial Quicksort algorithm **PQS** (both of them explained in Section 2.1.2.1).

We compared **IQS** with **PQS**, **QSS**, and two online approaches: the first based on classical heaps [Wil64] (called **HEx**), and the second based on sequence heaps [San00] (called **SH**, obtained from `www.mpi-inf.mpg.de/~sanders/programs/spq/`). The idea is to verify that **IQS** is in practice a competitive algorithm for the *Partial Sorting* problem of finding the smallest elements in ascending order. For this sake, we use random permutations of non-repeated numbers uniformly distributed in $[0, m-1]$, for $m \in \left[10^5, 10^8\right]$, and we select the k first elements with $k = 2^j < m$, for $j \geq 10$. The selection is incremental for **IQS**, **HEx** and **SH**, and in one shot for **PQS** and **QSS**. We measure CPU time and the number of key comparisons, except for **SH** where we only measure CPU time.

As it turned out to be more efficient, we implement **HEx** by using the bottom-up deletion algorithm [Weg93] for **extractMin** (see Section 2.1.3.2, page 12).

We summarize the experimental results in Figures 3.23, 3.24 and 3.25, and Table 3.1. As can be seen from the least square fittings of Table 3.1, **IQS** CPU time performance is only 2.99% slower than that of its offline version **PQS**. The number of key comparisons is exactly the same, as we expected from Section 3.2.2. This is an extremely small price for permitting incremental sorting without knowing in advance how many elements we wish to retrieve, and shows that **IQS** is practical. Moreover, as the pivots in the stack help us reuse the partitioning work, our online **IQS** uses only 1.33% more CPU time and 4.20% fewer key comparisons than the offline **QSS**. This is illustrated in Figure 3.23(a), where the plots of **PQS**, **IQS** and **QSS** are superimposed. A detail of the previous is shown in Figure 3.23(b), where we appreciate that **PQS** is the fastest algorithm when sorting a small fraction of the set, but **IQS** and **QSS** have rather similar behavior.

On the other hand, Table 3.1 shows large improvements with respect to online alternatives. According to the insertion and deletion strategy of sequence heaps, we compute its CPU time least squares fitting by noticing that we can split the experiment into

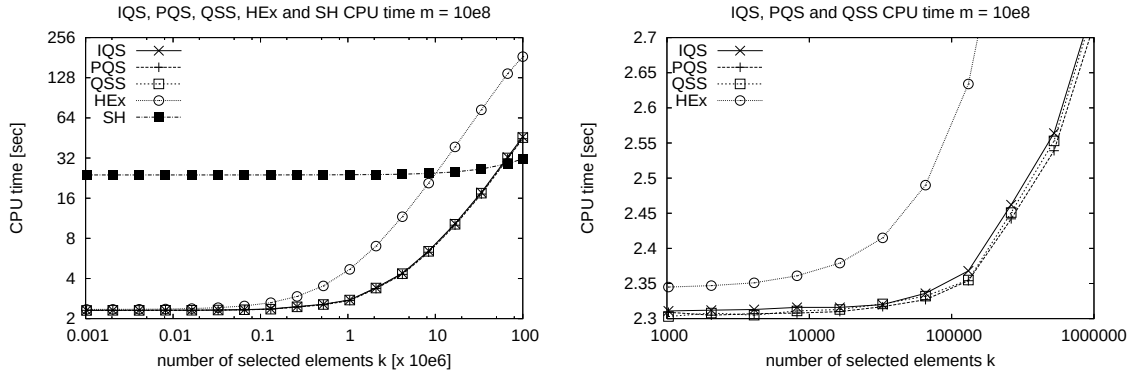
[2]Our fittings are too pessimistic for small permutations or edge sets, so we intend to show that they are asymptotically good. In the first series we compute the percent error for permutations of length $m \in [10^7, 10^8]$ for all the k values, which is approximately 45.4% of the measures. In the second series we compute the percent error for edge densities in [16%, 100%] for all values of $|V|$, which is approximately 44.4% of the measures.

	CPU time	Error	Key comparisons	Error
PQS	$25.79m + 16.87k \log_2 k$	6.77%	$2.138m + 1.232k \log_2 k$	5.54%
IQS	$25.81m + 17.44k \log_2 k$	6.82%	$2.138m + 1.232k \log_2 k$	5.54%
QSS	$25.82m + 17.20k \log_2 k$	6.81%	$2.140m + 1.292k \log_2 k$	5.53%
HEx	$23.85m + 67.89k \log_2 m$	6.11%	$1.904m + 0.967k \log_2 m$	1.20%
SH	$9.165m \log_2 m + 66.16k$	2.20%	—	—

Table 3.1: **IQS**, **PQS**, **QSS**, **HEx** and **SH** weighted least square fittings. For **SH** we only compute the CPU time fitting. CPU time is measured in nanoseconds.

two stages. The first inserts m random elements into the priority queue, and the second extracts the k smallest elements from it. Then, we obtain a simplified $O(m \log(m) + k)$ complexity model that shows that most of the work performed by **SH** comes from the insertion process. This also can be seen in Figure 3.23(a), by noticing that there is little difference between obtaining the first elements of the set, or the whole set. As a matter of fact, we note that if we want a small fraction of the sorted sequence, it is preferable to pay a lower insertion and a higher extraction cost (just like **IQS**) than to perform most of the work in the insertions and little in the extractions.

With respect to the online **HEx** using the bottom-up heuristic, we have the following. Even when it uses at most $2m$ key comparisons to heapify the array, and $\log m + O(1)$ key comparisons on average to extract elements, the poor locality of reference generates numerous cache faults slowing down its performance. In fact, **HEx** uses 3.88 times more CPU time, even using 18.76% fewer key comparisons than **IQS**. This is illustrated in Figure 3.23(a) where we can see that **HEx** has the second worst CPU performance for $k \leq 0.1m$ and the worst for $k \in [0.1m, m]$, despite that it makes fewer key comparisons than others when extracting objects, see Figure 3.24.

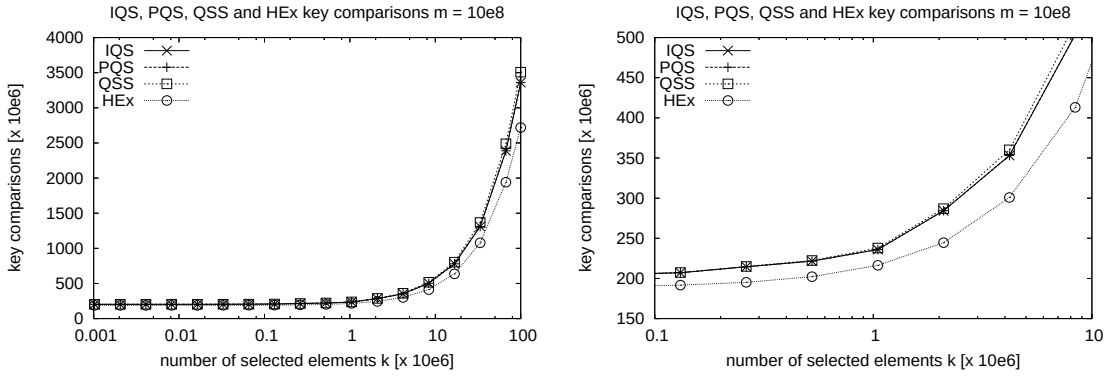


(a) CPU time for the five algorithms.

(b) Detail of CPU time for **IQS**, **PQS**, **QSS** and **HEx**.

Figure 3.23: Performance comparison between **IQS**, **PQS**, **QSS**, **HEx** and **SH** as a function of the amount of searched elements k for set size $m = 10^8$. Note the logscales in the plots.

Finally, Figure 3.25 shows that, as k grows, **IQS**'s behavior changes as follows. When



(a) Key comparisons for the four algorithms. (b) Detail of key comparisons for the four algorithms.

Figure 3.24: Key comparisons for **IQS**, **PQS**, **QSS** and **HEx** for $m = 10^8$ and varying k. Note the logscales in the plots.

$k \leq 0.01m$, there is no difference in the time to obtain either of the first k elements, as the term m dominates the cost. When $0.01m < k \leq 0.04m$, there is a slight increase of both CPU time and key comparisons, that is, both terms m and $k \log k$ take part in the cost. Finally, when $0.04m < k \leq m$, term $k \log k$ leads the cost.

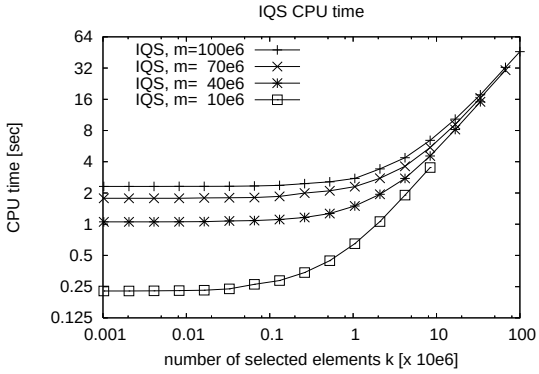


Figure 3.25: **IQS** CPU time as a function of k and m. Note the logscale in the plot.

3.7.2 Evaluating Quickheaps

We start by studying the empirical performance of each operation in isolation. Next, we evaluate two sequences of interleaved operations. The first consists in sequences of insertions and minimum extraction. The second consists in sequences of minimum extractions and key modifications. Each experimental datum shown is averaged over 20 repetitions. For shortness we call operation **insert** ins, operation **extractMin** del, operation **decreaseKey** dk and operation **increaseKey** ik.

In these experiments, inserted elements follow a uniform distribution. On the other

hand, updated keys are chosen uniformly, and they are updated by increasing or decreasing their values by 10%.

3.7.2.1 Isolated Quickheap Operations

We compare the empirical performance of quickheaps (or **QH**s for shortness) with binary heaps [Wil64] —including the improvement of Wegener for **extractMin** [Weg93]— and with paring heaps [FSST86]. We chose binary heaps because they are the canonical implementation of priority queues, they are efficient and easy to program. We also chose pairing heaps because they implement efficiently key update operations. Note that both binary and paring heaps are reported as the most efficient priority queue implementations in practice [MS91]. We obtain the pairing heap implementation, which includes operations **insert**, **extractMin** and **decreaseKey**, from `www.mpi-sb.mpg.de/~sanders/dfg/iMax.tgz`, as it is described in [KST02].

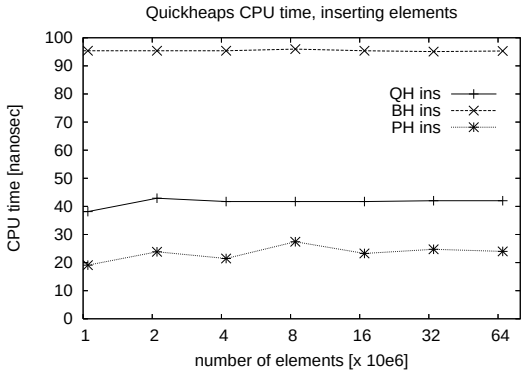
This experiment consists in inserting m elements $\left(m \in \left[2^{17} \approx 0.13\text{e}6, 2^{26} \approx 67\text{e}6\right]\right)$, next performing i times the sequence del-ik^i or del-dk^i, for $i = 10000$, and later extracting the remaining $m-i$ elements. We measured separately the time for each different operation in this sequence. Note that we are testing a combined sequence of one minimum extraction and several element modifications. Recall that both **QH**s and paring heaps actually structure the heap upon minima extractions, so the idea of running an additional minimum extraction is to force both quickheaps and pairing heaps to organize their heap structure.

Figure 3.26 shows the results and Table 3.2 the least-square fittings. It can be seen that the basic quickheap operations perform better than the corresponding binary heap ones. On the other hand, pairing heaps perform better than **QH**s both when inserting elements and decreasing the key, although pairing heap operation **extractMin** is several times costlier than the quickheap one.

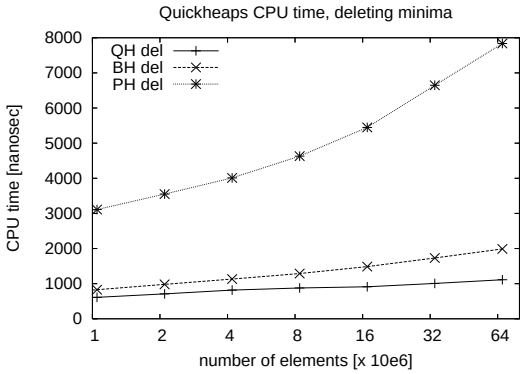
Figure 3.26(a) shows that, as expected from our analysis, the cost of quickheap operation **insert** is constant, and it is the second best time (approximately twice the CPU time of pairing heap and half the time of binary heap). Figure 3.26(b) shows that quickheaps have the best minimum extraction time, approximately two thirds of binary heaps and 6 times faster than pairing heaps. Finally, Figure 3.26(c) shows that quickheap update-key operations perform slightly better than the respective ones for binary heaps. The plot also shows that pairing heap's decrease key operation performs slightly better than quickheaps. Note that the model for quickheaps' decrease key operation was selected by observing the curve, as we could not analyze it.

3.7.2.2 Sequence of Insertions and Minimum Extractions

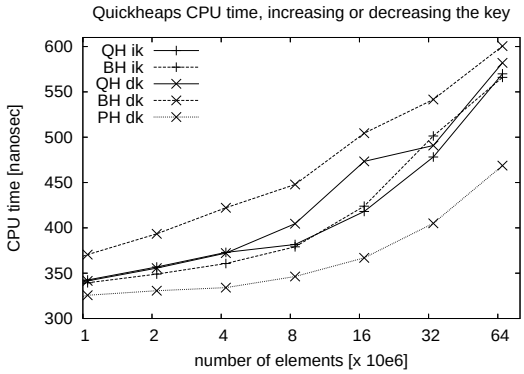
In order to show that quickheaps perform well under arbitrarily long sequences of insertions and minimum extractions we consider the following sequence of operations: $\big($ins-(del-ins)$^i\big)^m\big($del-(ins-del)$^i\big)^m$, for $i = 0$, 1 and 2. Note that for $i = 0$ we obtain algorithm heapsort [Wil64].



(a) Inserting elements.



(b) Extracting minima.



(c) Increasing and decreasing the key.

Figure 3.26: Performance of quickheap operations. Note the logscales in the plots.

	Quickheaps	Binary heaps	Pairing heaps
insert	42	99	26
extractMin	$35 \log_2 m$	$53 \log_2 m$	$201 \log_2 m$
increaseKey	$18 \log_2 m$	$18 \log_2 m$	—
decreaseKey	$18 \log_2 m$	$20 \log_2 m$	$16 \log_2 m$

Table 3.2: Least square fittings for Quickheaps operations. CPU time is measured in nanoseconds.

In this experiment we compare **QH**s with binary and pairing heaps as before, but we also include sequence heaps [San00], which are optimized for this type of sequences. The code for sequence heaps was obtained from `http://www.mpi-sb.mpg.de/~sanders/programs/spq/`. Sequence heaps were excluded from other experiments because they do not implement operations **increaseKey** and **decreaseKey**.

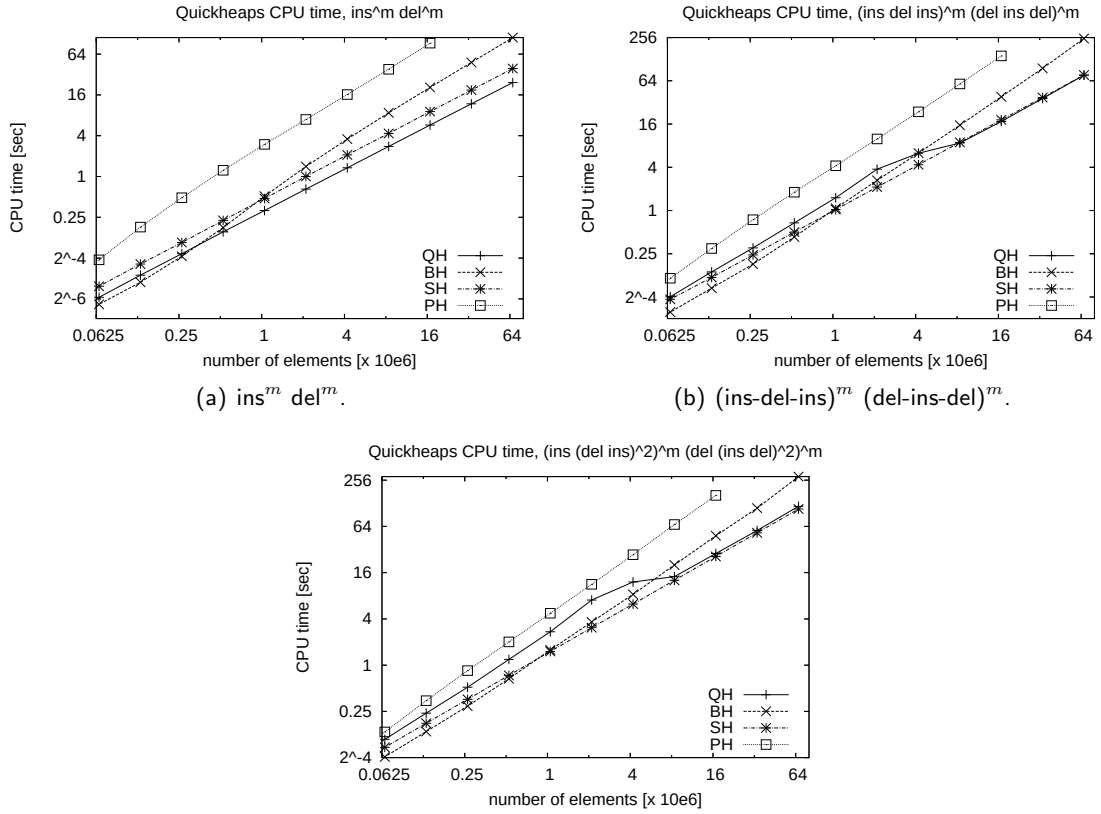
Figure 3.27 shows the results of this experiment. Figure 3.27(a) shows that binary heaps have the best performance for small sets, that is, up to $2^{18} \approx 262e3$ elements. This is expectable for two reasons: the bottom-up algorithm [Weg93] strongly improves the binary heap performance, and the whole heap fits in cache memory. However, as the number of elements in the heap increases, numerous cache misses slow down the performance of binary heaps (these heaps are known to have poor cache locality, since an extraction touches an arbitrary element at each level of the heap, and the lower levels contain many elements). Quickheaps, instead, are more cache-friendly as explained in the previous section. This is confirmed by the fact that quickheaps retain their good performance on large sets, being the fastest for more than $2^{19} \approx 524e3$ elements. In fact, for $m = 2^{26} \approx 67e6$ binary heaps perform 4.6 times slower than **QH**s, and sequence heaps perform 1.6 times slower than **QH**s. On the other hand, the pairing heap is, by far, the slowest contestant in this experiment, as its operation **extractMin** is very costly.

A similar behavior is appreciated for $i = 1$ (Figure 3.27(b)) and $i = 2$ (Figure 3.27(c)). For $i = 1$ binary heaps perform better for $m < 2^{20} \approx 1e6$ elements, then sequence heaps are the fastest until $m < 2^{23} \approx 8.4e6$, and finally quickheaps take over. For $i = 2$ the best behaviour is that of sequence heaps, closely followed by quickheaps. Binary heaps perform up to 2.43 times slower than quickheaps, and sequence heaps perform 8% faster than quickheaps. This is a modest difference considering that quickheaps are much simpler to implement than sequence heaps. Once again, operation **extractMin** leave pairing heaps out of the competitive alternatives for this experiment.

3.7.2.3 Sequence of Minimum Extractions and Key Modifications

Some of the most important priority queue applications are algorithms that use them to find the lowest value of a set, and then update several of the remaining values. This is the case of Dijkstra's Shortest Path algorithm [Dij59] or Prim's Minimum Spanning Tree algorithm [Pri57].

This experiment consists in inserting m elements into a priority queue, and then executing m times the sequence del-ik^{10} or del-dk^{10}. Figure 3.28 shows that quickheaps are consistently faster than binary heaps. It also shows that they are faster than pairing heaps. This is because the performance difference for operation **decreaseKey** between quickheaps and pairing heaps is small, and it is not enough to compensate the costly pairing heap operation **extractMin**.



(a) $\text{ins}^m\ \text{del}^m$. (b) $(\text{ins-del-ins})^m\ (\text{del-ins-del})^m$.

(c) $\left(\text{ins-}(\text{del-ins})^2\right)^m\ \left(\text{del-}(\text{ins-del})^2\right)^m$.

Figure 3.27: Performance of sequences interleaving operations ins and del. Note the logscales in the plots.

3.7.3 Evaluating External Memory Quickheaps

We carry out a brief experimental validation of quickheaps in external memory. It consists in measuring their performance when executing the sequence $\text{ins}^m\ \text{del}^m$ for $m \in [1\text{e}6, 200\text{e}6]$, varying the size of main memory M from 1 to 256 megabytes and disk block size $B = 32$ kilobytes. The inserted elements follow a uniform distribution. We also compare external quickheaps with the results presented in [BCFM00], which report the number of blocks read/written for different sequences of operations on the most promising secondary memory implementations, namely, two-level radix heaps [AMOT90] (*R-Heaps*) and *Array-Heaps* [BK98]. In [BCFM00], authors consider $M = 16$ megabytes of main memory and the same size of disk block as us. Experimental results show that secondary-memory quickheaps are competitive with the best state-of-the-art implementations of secondary memory priority queues.

The results are shown in Figure 3.29. As it can be seen, quickheaps achieve a performance slightly worse than the best alternative structures when using just 4 megabytes of RAM. When using the same 16 megabytes, our structure performs 29%

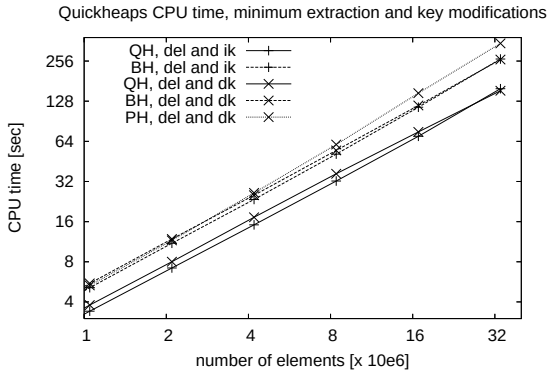


Figure 3.28: Performance of sequences interleaving operations del and ik/dk. Note the logscale in the plot.

to 167% of the I/O accesses of *R-Heaps* (that is, up to 3 times less), which only work if the priorities of the extracted elements form a nondecreasing sequence. If we consider the best alternative that works with no restriction (*Array-Heaps*), external *Quickheaps* perform 17% (up to 5 times less) to 125% of their I/O accesses. We notice that, as the ratio $\frac{m}{M}$ grows, the performance of both *R-Heaps* and *Array-Heaps* improves upon external *Quickheaps*'s. Other tests in [BCFM00] are harder to reproduce [3].

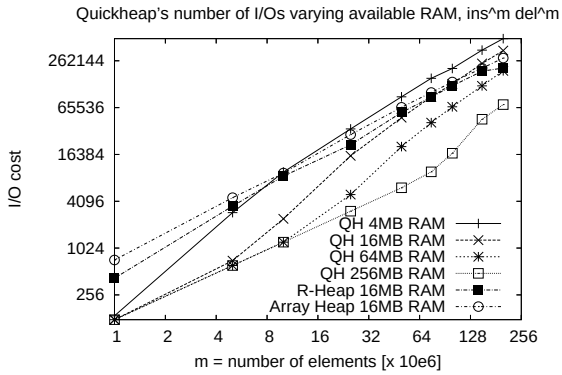


Figure 3.29: I/O cost comparison for the sequence ins^m del^m. Note the logscales in the plot.

Naturally, as more RAM is available, the I/O accesses consistently fall down. In fact, we notice the logarithmic dependence on m and on M (the plots are log-log), as expected from our analysis.

[3]For example, they also report real times, but those should be rerun in our machine and we do not have access to LEDA, which is mandatory to run their code.

3.7.4 Evaluating the MST Construction

MST construction is one of the emblematic applications of partial sorting and priority queues. We now evaluate both how **IQS** improves Kruskal's MST algorithm (Section 2.2.4.1), and the effect of implementing Prim's MST algorithm (Section 2.2.4.2) using **QH**s. Our aim is not to study new MST algorithms but just to demonstrate the practical impact of our new fundamental contributions to existing classical algorithms.

We compare our improved MST construction algorithms with state-of-the-art alternatives. We use synthetic graphs with edges chosen at random, and with edge costs uniformly distributed in $[0, 1]$. We consider graphs with $|V| \in [2000, 26000]$, and graph edge densities $\rho \in [0.5\%, 100\%]$, where $\rho = \frac{2m}{n(n-1)}100\%$.

For shortness we have called the basic Kruskal's MST algorithm **Kruskal1**, Kruskal's with demand sorting **Kruskal2**, our **IQS**-based Kruskal's **Kruskal3**, the basic Prim's MST algorithm **Prim1**[4], Prim's implemented with pairing heaps **Prim2** (Section 2.2.4.2), our Prim's implementation using **QH**s **Prim3** and the iMax algorithm **iMax** (Section 2.2.4.3).

According to the experiments of Section 3.7.1, we preferred classical heaps using the bottom-up heuristic (**HEx**) over sequence heaps (**SH**) to implement **Kruskal2** in these experiments (as we expect to extract $\frac{1}{2}n \ln n + O(n) \ll m$ edges). We obtained both the **iMax** and the optimized **Prim2** implementations from `www.mpi-sb.mpg.de/~sanders/dfg/iMax.tgz`, described in [KST02].

For Kruskal's versions we measure CPU time, memory requirements and the size of the edge subset reviewed during the MST construction. Note that those edges are the ones we incrementally sort. As the three versions run over the same graphs, they review the same subset of edges and use almost the same memory. For Prim's versions and **iMax** we measure CPU time and memory requirements.

We summarize the experimental results in Figures 3.30, 3.31, 3.32 and 3.33, and Table 3.3. Table 3.3 shows our least squares fittings for the MST experiments. First of all, we compute the fitting for the number of lowest-cost edges Kruskal's MST algorithm reviews to build the tree. We obtain $0.524\,|V| \ln |V|$, which is very close to the theoretically expected value $\frac{1}{2}|V| \ln |V|$. Second, we compute fittings for the CPU cost for all the studied versions using their theoretical complexity models. Note that, in terms of CPU time, **Kruskal1** is 17.26 times, and **Kruskal2** is 2.20 times, slower than **Kruskal3**. Likewise, **Prim3** is just 4.6% slower than **Kruskal3**. Finally, **Kruskal3** is around 33% slower than **Prim2** and 2.6 times faster than **iMax**. Note, however, that we cannot finish the experimental series with **Prim2** and **iMax**, as they use too much memory (the memory consumption is shown in Figure 3.31 and discussed soon).

Figure 3.30 compares all the studied versions for $n = 20,000$ and graph edge density $\rho \in [0.5\%, 100\%]$. As can be seen, **Kruskal1** is, by far, the slowest alternative, whereas

[4]That is, without priority queues. This is the best choice to implement Prim in complete graphs.

	Fitting	Error
Edges reviewed in Kruskal's versions	$0.524n \ln n$	2.97%
$\mathbf{Kruskal1}_{cpu}$	$12.87m \log_2 m$	2.31%
$\mathbf{Kruskal2}_{cpu}$	$40.38m + 37.47n \log_2 n \log_2 m$	3.57%
$\mathbf{Kruskal3}_{cpu}$	$20.44m + 9.19n \log_2^2 n$	4.67%
$\mathbf{Prim1}_{cpu}$	$19.08m + 7.24n^2$	1.74%
$\mathbf{Prim2}_{cpu}$	$9.71m + 141.2n \log_2 n$	8.24%
$\mathbf{Prim3}_{cpu}$	$19.81m + 37.56n \log_2 n \log \frac{m}{n}$	3.57%
$\mathbf{iMax}_{cpu}$	$30.44m + 655.1n \log_2 n$	25.83%

Table 3.3: Weighted least-square fittings for MST construction algorithms ($n = |V|$, $m = |E|$). CPU time is measured in nanoseconds.

Kruskal3 shows the best or second best performance for all ρ. **Prim3** also shows good performance, being slightly slower than **Kruskal3** for low densities ($\rho \leq 8\%$), and reaching almost the same time of **Kruskal3** for higher densities. When **Kruskal3** achieves the second best performance, the fastest algorithm is **Prim2**. We also notice that, as ρ increases, the advantage of our Kruskal's MST variant is more remarkable against basic Kruskal's MST algorithm. We could not complete the series for **Prim2** and **iMax**, as their structures require too much space. As a matter of fact, for 20,000 vertices and $\rho \geq 32\%$ these algorithms reach the 3 GB out-of-memory threshold of our machine.

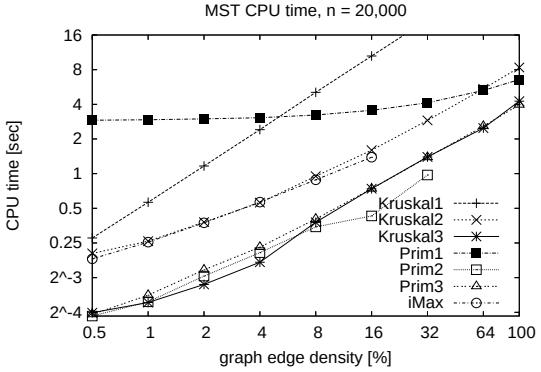


Figure 3.30: MST construction CPU times, for $n = 20,000$ depending on ρ. For $\rho = 100\%$ **Kruskal1** reaches 70.1 seconds. Note the logscale.

Figure 3.31 shows the memory requirements of **Kruskal3**, **iMax**, **Prim2** and **Prim3**, for n = 20,000. Since our Kruskal's implementation sorts the list of edges in place, we require little extra memory to manage the edge incremental sorting. With respect to **Prim3**, as the graph is handled as an adjacency list, it uses more space than the list of edges we use in **Kruskal3**. Nevertheless, the space usage is still manageable, and the extra quickheap structures use little memory. On the other hand, the additional structures of **Prim2** and **iMax** heavily increase the memory consumption of the process. We suspect that these high memory requirements trigger many cache faults and slow down their CPU

performance. As a result, for large graphs, **Prim2** and especially **iMax** become slower than **Kruskal3**, despite their better complexity.

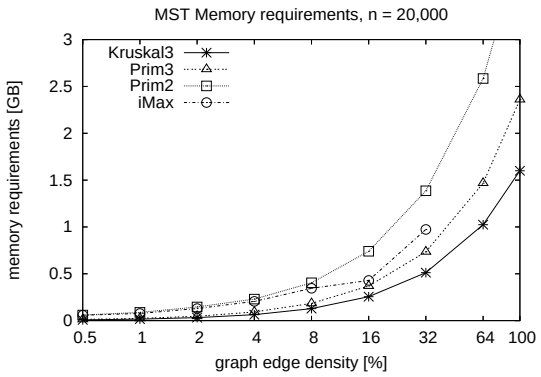
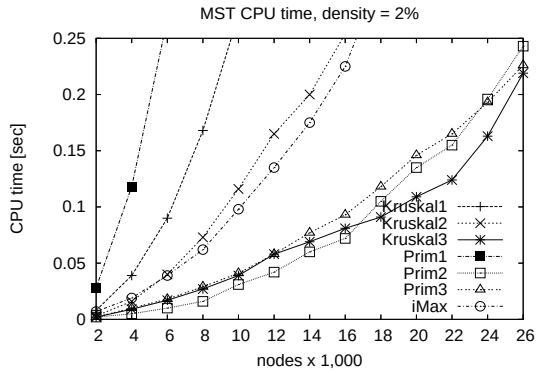


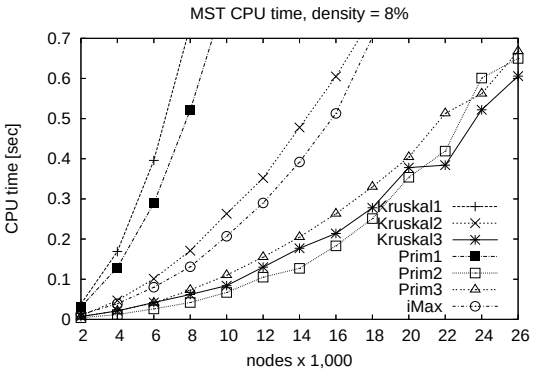
Figure 3.31: Memory used by **Kruskal3**, **iMax**, **Prim2** and **Prim3** for $|V| = 20,000$ nodes, depending on ρ. As can be seen, **iMax** and **Prim2** exhaust the memory for $\rho > 32\%$ and $\rho > 64\%$, respectively. Note the logscale.

Figure 3.32 shows the CPU time comparison for four edge densities $\rho = 2\%$, 8%, 32% and 100%. In the four plots **Kruskal3** is always the best Kruskal's version for all sizes of set V and all edge densities ρ. Moreover, Figure 3.32(d) shows that **Kruskal3** is also better than **Prim1**, even in complete graphs. Once again, **Prim3** shows a performance similar to **Kruskal3**. On the other hand, **Kruskal3** and **Prim3** are better than **iMax** in the four plots, and very competitive against **Prim2**. In fact **Kruskal3** beats **Prim2** in some cases (for $|V| \geq$ 18,000 and 22,000 vertices in $\rho = 2\%$ and 8%, respectively). We suspect that this is due to the high memory usage of **Prim2**, which affects cache efficiency. Note that for $\rho = 64\%$ and 100% we could not finish the series with **Prim2** and **iMax** because of their memory requirements.

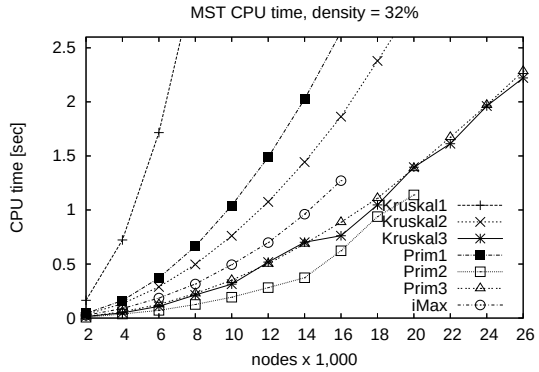
Finally, Figure 3.33 shows the same comparison of previous figure, now considering a *lollipop* graph. Given a random graph $G(V, E)$ we can build a lollipop graph G_l as follows. First we compute the maximum edge weight $weight^*$ of G; and second, we pick a node $u_l \in V$ at random and increase the weight of its edges by $weight^*$. Lollipop graphs are a hard case for Kruskal's algorithms, as they force them to review almost all the edges in G before connecting u_l to the MST of G_l. The figure shows that the MST CPU time of all Kruskal's variant dramatically increases, while **Prim3** preserves its performance. We omit **Prim2** and **iMax** as we do not have the lollipop graph generator for these algorithms. Note, however, that it is also expectable that both **Prim2** and **iMax** will retain their performance. That is, **Prim2** could be the best or second best algorithm, and **iMax** would display the same performance of Figure 3.32, which is not enough to beat **Prim3**. Likewise, it is also expectable they exhaust the main memory in the same cases of previous experiments.



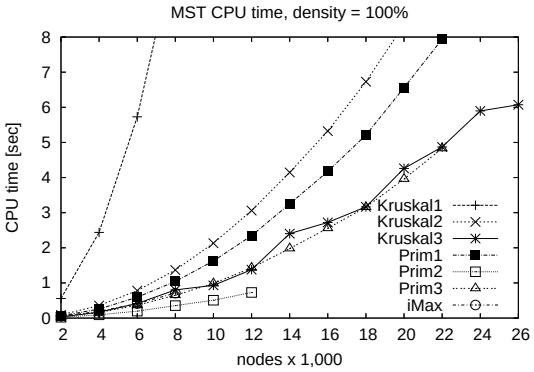
(a) MST CPU time, depending on n.



(b) MST CPU time, depending on n.

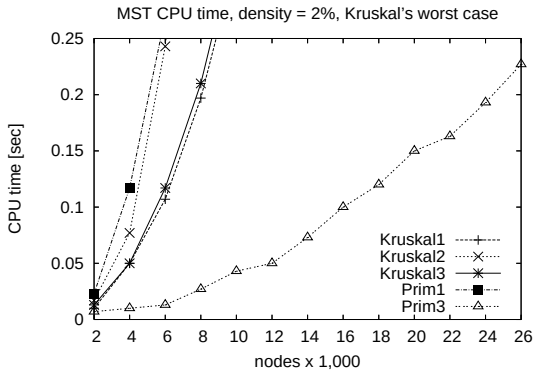


(c) MST CPU time, depending on n.

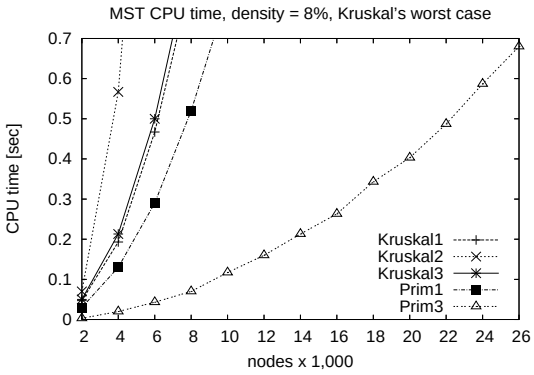


(d) MST CPU time, depending on n.

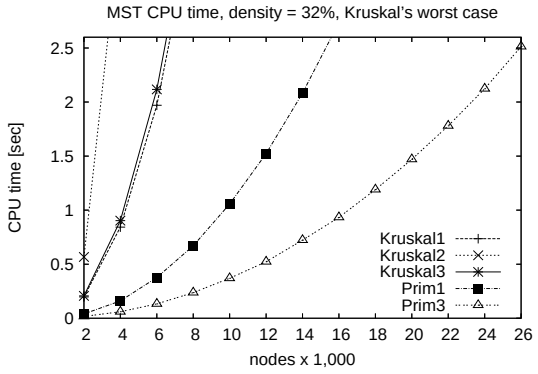
Figure 3.32: Evaluating MST construction algorithms as a function of $n = |V|$ in (a), (b), (c) and (d) for $\rho = 2\%$, 8%, 32% and 100%, respectively. For $n = 26,000$, in (a) **Kruskal1**, **Kruskal2** and **iMax** reach 2.67, 0.76 and 0.62 seconds; in (b) **Kruskal1**, **Kruskal2** and **iMax** reach 9.08, 1.56 and 1.53 seconds; in (c) **Kruskal1** and **Kruskal2** reach 37.02 and 4.82 seconds; in (d) **Kruskal1**, **Kruskal2** and **Prim1** reach 121.14, 13.84 and 25.96 seconds, respectively.



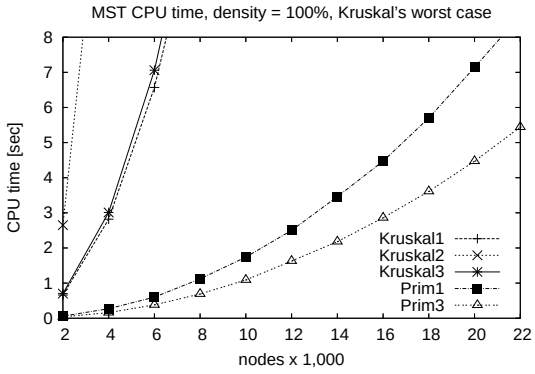
(a) MST CPU time, Kruskal's worst case.



(b) MST CPU time, Kruskal's worst case.



(c) MST CPU time, Kruskal's worst case.



(d) MST CPU time, Kruskal's worst case.

Figure 3.33: Evaluating MST construction algorithms as a function of $n = |V|$ in (a), (b), (c) and (d) for $\rho = 2\%$, 8%, 32% and 100%, respectively. In these series we use a lollipop graph, which is a hard case for Kruskal's algorithm. It can be seen that the MST CPU time of all Kruskal's variant dramatically increases, while **Prim3** preserves its performance.

Chapter 4

k-Nearest Neighbor Graphs

> La visión de tu vecino es tan cierta para él como tu propia visión lo es para ti. [Your neighbor's view is as true for him as your own view is true for you.]
>
> – Miguel de Unamuno

Let us consider a metric space $(\mathbb{X}, d)$, where $\mathbb{X}$ is the universe of objects and d is a distance function defined among them. Let $\mathbb{U}$ be a set of objects in $\mathbb{X}$. The *k-nearest neighbor graph* (kNNG) of the set $\mathbb{U}$ is a weighted directed graph $G(\mathbb{U}, E)$ connecting each element $u \in \mathbb{U}$ to its k nearest neighbors, thus $E = \{(u, v), v \in NN_k(u)\}$. That is, for each element u we store the result of its k-nearest neighbor query $NN_k(u)$. kNNGs themselves can be used for many purposes, for instance, cluster and outlier detection [EE94, BCQY96], VLSI design, spin glass and other physical process simulations [CK95], pattern recognition [DH73], query or document recommendation systems [BYHM04a, BYHM04b], similarity self joins [DGSZ03, DGZ03, PR08], and many others. Hence, their construction is interesting *per se*. For this sake, in Section 4.1 we introduce a general kNNG construction methodology which exploits some metric and graph properties. Next, in Section 4.2 we give two kNNG construction algorithms developed on top of our methodology. In this thesis, we are interested in how to use them to speed up metric queries, which is the main topic of Section 4.3, where we give some algorithms for searching metric spaces using the kNNG as the metric index. Finally, in Section 4.4 we give some experimental results both for construction and for searching algorithms.

4.1 A General kNNG Construction Methodology

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
>
> – C.A.R. Hoare

We are interested in practical kNNG construction algorithms for general metric spaces. This problem is equivalent to solving n $NN_k(u)$ queries for all $u \in \mathbb{U}$. Thus, a straightforward solution has two stages: the first is to build some known metric index $\mathcal{I}$ (there are many indices described in [CNBYM01, HS03, ZADB06, Sam06]), and the second is to use $\mathcal{I}$ to solve the n queries. Naturally, we can try to use range-optimal k-nearest neighbor queries —instead of the classic alternative of performing range queries varying the radius— to avoid extra work that retrieves more objects than necessary. This basic scheme can be improved if we take into account some observations:

- We are solving queries for the elements in $\mathbb{U}$, not for general objects in $\mathbb{X}$. Therefore, the construction of $\mathcal{I}$ itself can give us valuable information for the n queries.
- We are solving queries for *all* the elements in $\mathbb{U}$. If we solve the n queries jointly we can share costs through the whole process. For instance, we can avoid some calculations by using the symmetry of d, or some other heuristic improvements.
- We can upper bound some distances by computing shortest paths over the kNNG under construction, maybe avoiding their actual computation. So, we can use the very kNNG in stepwise refinements to improve the second stage.

Based on these observations, we develop some ingredients to enhance the process of constructing kNNGs. In the following we describe them in detail, and finally show how to use them together in the kNNG recipe.

Since we want to use the kNNG to index the metric space, in order to avoid confusions we call index $\mathcal{I}$ the preindex, and the first stage the preindexing stage.

4.1.1 The Main Data Structure

All along the algorithm, we use a so-called *Neighbor Heap Array* (*NHA*) to store the kNNG under construction. For each object $u \in \mathbb{U}$, we store a k-element *priority queue* NHA_u, and *NHA* can be regarded as the union of all the NHA_u. At any point in the process NHA_u will contain the k elements closest to u known up to then, and their distances to u.

Formally, $NHA_u = \{(x_{i_1}, d(u, x_{i_1})), \ldots, (x_{i_k}, d(u, x_{i_k}))\}$ sorted by decreasing $d(u, x_{i_j})$ (i_j is the j-th neighbor identifier).

For each $u \in \mathbb{U}$, we initialize $NHA_u = \{(\text{NULL}, \infty), \ldots, (\text{NULL}, \infty)\}$, $|NHA_u| = k$. Let $curCR_u = d(u, x_{i_1})$ be the current covering radius of u, that is, the distance from u towards its farthest current neighbor candidate in NHA_u.

4.1.2 Management of *NHA*

In the first stage, every distance computed to build the preindex $\mathcal{I}$ populates *NHA*. In the second, we refine *NHA* with the additional distance computations.

We must ensure that $|NHA_u| = k$ upon successive additions. Hence, if we find some object v such that $d(u, v) < curCR_u$, before adding $(v, d(u, v))$ to NHA_u we extract the farthest candidate from NHA_u. This progressively reduces $curCR_u$ from ∞ to the real covering radius. At the end, each NHA_u stores the k objects satisfying the query $NN_k(u)$, that is, they will become the k closest neighbors of u. Therefore, once the construction finishes *NHA* stores the kNNG of $\mathbb{U}$.

4.1.3 Using *NHA* as a Graph

Once we calculate $d_{uv} = d(u, v)$, if $d_{uv} \geq curCR_u$ we discard v as a candidate for NHA_u. Also, due to the triangle inequality we can discard all objects w such that $d(v, w) \leq d_{uv} - curCR_u$. Unfortunately, we do not necessarily have stored $d(v, w)$. However, we can upper bound $d(v, w)$ with the sum of edge weights traversed in the shortest paths over *NHA* from v to all $w \in \mathbb{U}$, $d_{NHA}(v, w)$. So, if $d_{uv} \geq curCR_u$, we also discard all objects w such that $d_{NHA}(v, w) \leq d_{uv} - curCR_u$. These objects w can be found using a shortest-path expansion from v over the graph *NHA*.

4.1.4 d is Symmetric

Every time a distance $d_{uv} = d(u, v)$ is computed, we check both $d_{uv} < curCR_u$ for adding (v, d_{uv}) to NHA_u, and $d_{uv} < curCR_v$ for adding (u, d_{uv}) to NHA_v. This can also reduce $curCR_v$ and cheapen the future query for v, even when we are solving neighbors for another object.

4.1.5 $\mathbb{U}$ is Fixed

Assume we are solving query $NN_k(u)$, we have to check some already solved object v, and $curCR_u \leq curCR_v$. This means that v already has its definitive k-nearest neighbors and $curCR_v$ achieved its (smallest) final value. Then, if $u \notin NN_k(v) \Rightarrow d(u, v) \geq curCR_v \geq curCR_u$, so necessarily $v \notin NN_k(u)$. Otherwise, if $u \in NN_k(v)$, then we have already computed $d(u, v)$. Then, in those cases we avoid to compute $d(u, v)$. Figure 4.1 illustrates.

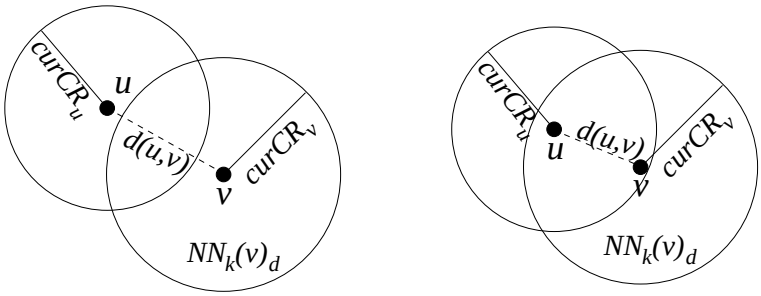


Figure 4.1: Assume we are solving u, v is already solved, and $curCR_u \leq curCR_v$. On the left, if $u \notin NN_k(v) \Rightarrow d(u,v) \geq curCR_v \geq curCR_u$. On the right, if $u \in NN_k(v)$, we have already computed $d(u,v)$. Then, in those cases we avoid computing $d(u,v)$.

4.1.6 Check Order Heap (COH)

We create priority queue $COH = \{(u, curCR_u), u \in \mathbb{U}\}$ to complete $NN_k(u)$ queries for all objects $u \in \mathbb{U}$ as they are extracted from COH in increasing $curCR_u$ order, until COH gets empty. The goal is to solve the easiest queries first, both to reduce the CPU time and, by virtue of the symmetry of d, to increase the chance of reducing other $curCR_v$'s.

Indeed, note that when we are solving the query for an object u, if we have to check an object v whose query is yet unsolved we compute the distance between them ($d_{uv} = d(u,v)$) and by using $curCR_u$ we check whether v gets into the nearest neighbor candidates for u. Nevertheless, using "d is symmetric", we can also check whether v's current covering radius $curCR_v$ improves (that is, diminishes) when considering u within the v neighbor candidates, in which case we update v in COH with its new $curCR_v$. This way, we cheapen the future query for v. This is likely to occur because the query we are currently solving is the one with the smallest candidate radius. Note that a small radius query has better discriminative power and produces candidates that are closer to the query, which is a favorable scenario in order to use "d is symmetric".

4.1.7 The Recipe

We split the process into two stages. The first is to build $\mathcal{I}$ to preindex the objects. The second is to use $\mathcal{I}$ and all the ingredients to solve the $NN_k(u)$ queries for all $u \in \mathbb{U}$. Figure 4.2 depicts the methodology.

For practical reasons, we allow that our algorithms use at most $O(n(k+\log n))$ memory both to preindex $\mathbb{U}$ and to store the kNNG under construction.

---

*K***NN** (Integer k, ObjectSet $\mathbb{U}$)
Stage 1: Initialize *NHA* and construct the preindex $\mathcal{I}$
1. **For each** $u \in \mathbb{U}$ **Do** $NHA_u \leftarrow \{(\text{NULL}, \infty), \ldots, (\text{NULL}, \infty)\}$ // k pairs
2. Create $\mathcal{I}$, all computed distances populate symmetrically *NHA*
Stage 2: Complete the $NN_k(u)$ for all $u \in \mathbb{U}$
3. $COH \leftarrow \{(u, curCR_u), u \in \mathbb{U}\}$
4. **For each** $(u, curCR_u) \in COH$, in increasing $curCR_u$ order **Do**
5. Create the candidate set $\mathcal{C}$ according to $\mathcal{I}$ // exclude NHA_u
6. **While** $\mathcal{C} \neq \emptyset$ **Do**
7. $v \leftarrow$ extract a candidate from $\mathcal{C}$
8. **If** "$\mathbb{U}$ is fixed" does not apply for u and v **Then**
9. $d_{uv} \leftarrow d(u, v)$, try to insert v into NHA_u
10. try to insert u into NHA_v, update v in COH (symmetry)
11. use preindex $\mathcal{I}$ and *NHA* as a graph to filter out objects from $\mathcal{C}$
12. **Return** *NHA* as a graph

Figure 4.2: Sketch of the methodology.

4.2 *k*NNG Construction Algorithms

> Why did Nature create man? Was it to show that she is big enough to make mistakes, or was it pure ignorance?
>
> – Holbrook Jackson

On top of our methodology, we propose two *k*NNG construction algorithms based on small preindices and focused on decreasing the total number of distance computations. These are:

1. Recursive partition based algorithm: In the first stage, we build a preindex by performing a recursive partitioning of the space. In the second stage, we complete the $NN_k(u)$ queries using the order induced by the partitioning.

2. Pivot based algorithm: In the preindexing stage, we build the pivot index. Later, we complete the $NN_k(u)$ queries by performing range-optimal queries, which are additionally improved with metric and graph considerations.

In the next section we show how to use the recipe to implement a basic *k*NNG construction algorithm. Later, we give our two construction algorithms.

4.2.1 Basic kNNG Construction Algorithm

The intuitive idea to solve kNNG is iterative: for each $u \in \mathbb{U}$ we compute the distance towards all the others, and select the k smallest-distance objects. The basic algorithm we present in Figure 4.3 already uses some of the ingredients: the second stage, *NHA*, and the symmetry of d. At the end, in *NHA* we have the kNNG of $\mathbb{U}$.

---

*K***NNb** (Integer k, ObjectSet $\mathbb{U}$)
1. **For each** $u \in \mathbb{U}$ **Do** $NHA_u \leftarrow \{(\text{NULL}, \infty), \ldots, (\text{NULL}, \infty)\}$, $|NHA_u| = k$
2. **For** $i \in [1, |\mathbb{U}| - 1], j \in [0, i-1]$ **Do**
3. $u \leftarrow \mathbb{U}_i$, $v \leftarrow \mathbb{U}_j$, $d_{uv} \leftarrow d(u, v)$ // $\mathbb{U}_i$ refers to the i-th object of $\mathbb{U}$
4. **If** $d_{uv} < curCR_u$ **Then** NHA_u.**extractMax**(), NHA_u.**insert**(v, d_{uv})
5. **If** $d_{uv} < curCR_v$ **Then** NHA_v.**extractMax**(), NHA_v.**insert**(u, d_{uv})
6. **Return** *NHA*

Figure 4.3: Basic kNNG construction algorithm (*K***NNb**). For each $u \in \mathbb{U}$, (i) NHA_u is a priority queue of size k whose elements are of the form (object, $value$) sorted by decreasing $value$, and (ii) $curCR_u$ is an alias for NHA_u.**findMax**().$value$.

The basic kNNG construction algorithm makes $O(n^2)$ distance evaluations, has $O(n^2 \log k)$ worst case extra CPU cost and $O\big(n\left(n + k \log k \log \frac{n}{k}\right)\big)$ expected extra CPU cost, and uses $O(kn)$ memory.

The worst case extra CPU time comes from performing $O(n^2)$ times the pair of operations **extractMax** and **insert** over all the priority queues NHA_u. Using any decent k-element priority queue we need $O(\log k)$ time to perform both operations. This accounts for a total $O(n^2 \log k)$ time.

The expected extra CPU time comes from the expected number of comparisons that occur in *NHA*. Let us considered the classic binary heap. The base case consists in initializing the heap with the first k distances using Floyd's linear time algorithm [Flo64]. Next, during the process, a random distance is greater than the k-th smallest one with probability $\frac{n-k}{n}$, and lower with probability $\frac{k}{n}$ (where n is the number of distances seen up to now). In both cases we pay one comparison against the k-th smallest one, and in the second, we also pay $\log_2 k + O(1)$ comparisons in NHA_u (using Wegener's bottom-up deletion algorithm [Weg93]). Also, the expected number of key comparisons when inserting independent and identically distributed elements in a binary heap is $O(1)$.

So, the recurrence that counts the expected number of comparisons for each NHA_u is $T(n,k) = T(n-1,k) + 1\frac{n-k}{n} + (1 + \log k + O(1))\frac{k}{n}$, $T(k,k) = O(k)$. Collecting terms and replacing the $O(1)$ term by some constant c we obtain $T(n,k) = T(n-1,k) + 1 + (c + \log k)\frac{k}{n} = T(n-1,k) + 1 + k(c + \log k)(H_n - H_{n-1}) = \ldots = O(k) + n - k + k(c + \log k)(H_n - H_k) = O\big(n + k \log k \log \frac{n}{k}\big)$. As we have n queues, we obtain $O\big(n\left(n + k \log k \log \frac{n}{k}\right)\big)$ expected extra CPU time.

Note that using a similar analysis, we can prove that using our Quickheaps (Section 3.3) we obtain the same $O\big(n\left(n + k\log k\log\frac{n}{k}\right)\big)$ expected extra CPU time.

4.2.2 Recursive-Partition-Based Algorithm

This algorithm is based on using a preindex similar to the *Bisector Tree* (BST) [KM83]. We call our modified BST the *Division Control Tree* (DCT), which is a binary tree representing the shape of the partitioning. The DCT node structure is $\{p, l, r, pr\}$, which represents the parent, the left and right child, and the partition radius of the node, respectively. The partition radius is the distance from the node towards the farthest node of its partition. (With respect to the original BST structure, we have added the pointer p to easily navigate trough the tree.)

For simplicity we use the same name for the node and for its representative in the DCT. Then, given a node $u \in \mathbb{U}$, u_p, u_l, and u_r, refer to nodes that are the parent, left child, and right child of u in the DCT, respectively, and also to their representative elements in $\mathbb{U}$. Finally, u_{pr} refers to the partition radius of u.

In this algorithm, we use $O(kn)$ space to store the *NHA* and $O(n)$ to store the DCT. The remaining memory is used as a cache of computed distances, *CD*, whose size is limited to $O(n \log n)$. Thus, every time we need to compute a distance, we check if it is already present in *CD*, in which case we just return the stored value. Note that the $CD \subset \mathbb{U}^2 \times \mathbb{R}^+$ can also be seen as graph of all stored distances. The criterion to insert distances into *CD* depends on the stage of the algorithm (see later). Once we complete the $NN_k(u)$, we remove its adjacency list from *CD*.

4.2.2.1 First Stage: Construction of DCT

We partition the space recursively to construct the DCT, and populate symmetrically *NHA* and *CD* with all the distances computed. Note that during the first stage, *CD* is an undirected graph since for each distance we store both directed edges.

The DCT is built as follows. Given the node *root* and the set S, we choose two far away objects, the children l and r, from S. To do so, we take a sample of $|S|/2$ object pairs from S at random and pick the farthest pair. Then, we generate two subsets: S_l, objects nearer to l, and S_r, objects nearer to r. Finally, for both children we update their parents (with *root*) and compute both partition radii in DCT. The recursion follows with (l, S_l) and (r, S_r), finishing when $|S| < 2$. DCT is constructed by procedure **division**, depicted in Figure 4.4.

Once we finish the division, leaves in the DCT have partition radii 0. The DCT root is the only fictitious node without an equivalent in $\mathbb{U}$. Its partition radius is ∞, and its children are the two nodes of the first division.

division(Object R, ObjectSet S)
1. **If** $|S| = 0$ **Then Return**
2. **If** $|S| = 1$ **Then** $R_l \leftarrow o$, $o_p \leftarrow R$, **Return** // let $S = \{o\}$
3. $(u,\ v,\ d_{uv}) \leftarrow$ two far away objects, and the distance between them
4. $R_l \leftarrow u$, $R_r \leftarrow v$
5. $S \leftarrow S - \{u, v\}$
6. $(S_u, maxd_u, S_v, maxd_v) \leftarrow S$ divided according to distances towards u and v, and the respective partition radii
7. $u_p \leftarrow R, u_{pr} \leftarrow maxd_u, v_p \leftarrow R, v_{pr} \leftarrow maxd_v$
8. **division**(u, S_u), **division**(v, S_v)

Figure 4.4: Auxiliary procedure **division**. All the distances computed in lines 3 and 6 populate *NHA* and are stored in *CD* symmetrically. To improve readability, we use subscripts to reference the parent, left child, right child and partition radius of node u in the DCT, which is managed as a global structure. R is the root of the current partitioning of set S, in the first division is a fictitious node.

It is easy to see that the expected number of distances computed when building the DCT, and thus stored in *CD*, in the first stage is $O(n \log n)$. Note that the DCT is a binary tree composed by $n + 1$ nodes. Let us consider a path starting from any node u towards the DCT root. In this path, node u has computed the distance towards each node in the path, and its respective sibling in DCT, except for the fictitious node *root*. Therefore, the number of computed distances is twice the sum of internal path lengths minus $2n$ (so as to discount all the times we visit node *root* when adding up the n paths).

Now we have to realize that the expected sum of internal path lengths follows the same expected analysis of Quicksort. To do so, we notice that taking two nodes l and r to recursively split the set S according to the distances towards them is equivalent to recursively producing two array partitions using a random pivot. So, the partition tree represented by the DCT obeys the same shape of the partition tree produced by the pivots of Quicksort. Therefore, using the Quicksort formula [GBY91], the expected sum of the internal path lengths of an $(n + 1)$-node tree is $2(n + 2)H_{n+1} - 4(n + 1)$.

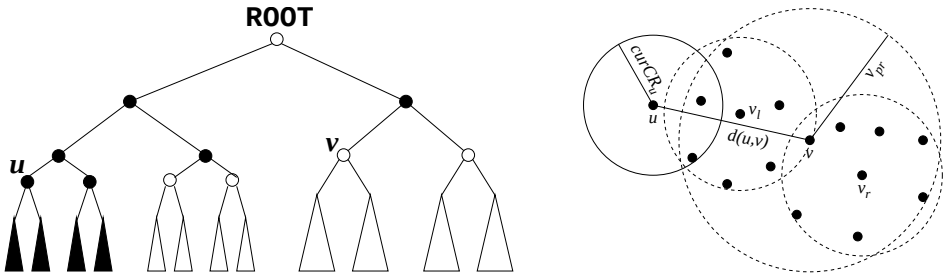
Putting all together, during the construction of the DCT we compute $4(n + 2)H_{n+1} - 8(n + 1) - 2n$ distances. As we store 2 edges per distance, we expect to store $8(n + 2)H_{n+1} - 20(n + 1) + 4 < 8n \ln n$, for $n > 1$. Hence, we fix the space of *CD* as $8n \ln n = O(n \log n)$.

During the first stage we store *all* the computed distances, even if they are more than $8n \ln n$, yet it is very unlikely that the size of *CD* exceeds the limit in this stage. For the second stage, our caching policy enforces the limit of storing at most $8n \ln n$ distances.

4.2.2.2 Second Stage: Computing the kNNG

We first explain how to solve a single $NN_k(u)$ query using only the DCT structure, and next how to use the ingredients to share costs, and take benefit of the whole kNNG construction process.

Solving a Single $NN_k(u)$ Query with the DCT. To solve the query we create the candidate set $\mathcal{C}$ according to the DCT. Note that the construction of DCT ensures that every node has already computed distances to all of its ancestors, its ancestor's siblings, and its parent descent. Then, to finish the $NN_k(u)$ query, it is enough to check whether there are relevant objects in all the descendants of u's ancestors' siblings; as other nodes (ancestors, ancestor's siblings and parent descent) have already been checked. In Figure 4.5(a), we represent the non-checked objects as white nodes and subtrees. So, the candidate set $\mathcal{C}$ should be the set of *all* the nodes and (the nodes inside the) subtrees we are going to review.



(a) Induced check sequence. (b) DCT can discard partitions.

Figure 4.5: Using the DCT to solve $NN_k(q)$ queries. In (a), u has been compared with all black nodes and all the descent of its parent. To finish the query, we only process white nodes and subtrees. In (b), we show how to use DCT to avoid checking some partitions by only reviewing the ones intersecting with the ball $(u, curCR_u)$. We start by checking the v's partition, so we recursively descend to v's children; thus, using the radii $v_{l_{pr}}$ and $v_{r_{pr}}$, we check the partition of v_l, and discard the v_r's.

Nevertheless, the DCT allows us to avoid some work. Assume we are checking whether v is relevant to u. If $d(u,v) \geq curCR_u + v_{pr}$ we can discard v and its whole partition, because the balls $(u, curCR_u)$ and (v, v_{pr}) do not intersect. Otherwise, we recursively check children v_l and v_r. Figure 4.5(b) illustrates. Hence, to solve the query for u, it suffices to initialize the set $\mathcal{C}$ with all u's ancestors' siblings. Next, we pick a node v from $\mathcal{C}$ and check whether its partition (v, v_{pr}) does not intersect with the ball containing u's k-nearest neighbor candidates $(u, curCR_u)$, in which case we discard the whole v's partition. Otherwise, we add v's children (nodes v_l and v_r) to the set $\mathcal{C}$. This way, we only manage nodes, not trees, in $\mathcal{C}$.

Finally, since it is more likely to discard small partitions, we process $\mathcal{C}$ in order of increasing radius. This agrees with the intuition that the partition radius of u's parent's

sibling is likely to be the smallest of $\mathcal{C}$, and that some of u's parent's sibling's descendants could be relevant to u, since these nodes share most of u's branch in DCT, and the partitioning is made according to node closeness. Furthermore, note that finding close neighbors fast helps reducing $curCR_u$ early, which speeds up the process by permitting more pruning of subtrees.

Solving All the Queries Jointly by Using the Ingredients. As CD can be seen as a graph, we use $NHA \cup CD$ to upper bound distances: when $d(u,v) \geq curCR_u$, we discard objects w such that their shortest path $d_{NHA \cup CD}(v,w) \leq d(u,v) - curCR_u$, avoiding the direct comparison. We do this by adding edges (u,w) to CD marked as `EXTRACTED`. This task is performed by **extractFrom**, which is a variant of Dijkstra's shortest-path algorithm with propagation limited to $d(u,v) - curCR_u$. Note that an edge (u,w) could belong to CD in two cases: (i) w has been marked as `EXTRACTED`, and (ii) we have the real value of the distance between u and w. In both cases we stop the propagation (which also ensures us to avoid exponential complexity due to multiple checking), but in the second case we do not mark the node w as `EXTRACTED`, as we can use later this already computed distance.

Note that we can save some CPU time in **extractFrom** if we use the distance to the current nearest neighbor of each node w, NND_w, to know whether it is worthless propagating paths from w because we exceed the discarding limit even with its nearest neighbor. Figure 4.6 depicts procedure **extractFrom**.

```
extractFrom(Object v, Object u, ℝ+ limit)
1.  minHeap.insert(v, 0) // ordered by the second component
2.  While |minHeap| > 0 Do
3.      (c, d_vc) ← minHeap.extractMin()
4.      For each (w, d_cw) ∈ NHA_c ∪ CD_c Do
5.          If (u, w) ∈ CD Then continue // with the next element in minHeap
6.          If d_vc + d_cw ≤ limit Then
7.              CD_u.insert(w, EXTRACTED) // we discard w
8.              If d_vc + d_cw + NND_w ≤ limit Then minHeap.insert(w, d_vc + d_cw)
```

Figure 4.6: Auxiliary procedure **extractFrom**. **extractFrom** is a shortest-path algorithm limited to $limit = d(u,v) - curCR_u$. It discards objects w from v by marking them as `EXTRACTED` (that is, performing CD_u.**insert**(w, `EXTRACTED`)) only if we do not have already stored its value in CD. NND_w is the distance from w towards its current nearest neighbor. CD_c refers to the adjacency list of c in CD.

In this stage, if we have available space in CD —that is, if $|CD| < 8n \ln n$—, we cache all the computed distances towards unsolved nodes that are small enough to get into their respective queues in NHA, since these distances can be used in future symmetric queries. We perform this check in line 13 of algorithm K**NNrp**, see Figure 4.7. If the space checking permits, we instruct auxiliary method **finish**k**NNQ** to cache all the computed distances

for unsolved nodes. Otherwise, unfortunately we have to drop all of them when we finish to solve the query.

Note that adding distances to *CD* without considering the space limitation could increase its size beyond control, as it is shown by the expected case analysis given in the next paragraph. Then, the limit $|CD| = O(n \log n)$ becomes relevant at this stage. In order to have space to store new computed distances, as soon as we finish the query for some node, we delete its adjacency list from *CD*. Therefore, in the second stage *CD* becomes a directed graph.

The expected case analysis for the size of *CD* follows. With probability $\frac{n-k}{n}$ —where n is the number of distances seen up to now—, a random distance is greater than the k-th smallest one (thus, not stored), and with probability $\frac{k}{n}$ it is lower, thus it is stored in *CD* using one cell. The base case uses k cells for the first k distances. Then, the recurrence for the expected case of edge insertions for each NHA_u is: $T(n,k) = T(n-1,k) + \frac{k}{n}$, $T(k,k) = k$. We obtain $T(n,k) = k(H_n - H_k + 1) = O\big(k \log \frac{n}{k}\big)$. As we have n priority queues, if we do not consider the limitation, we could use $O\big(nk \log \frac{n}{k}\big)$ memory cells, which can be an impractical memory requirement.

We combine all of these ideas to complete the $NN_k(u)$ queries for all nodes in $\mathbb{U}$. To do so, we begin by creating the priority queue *COH* where we store all the nodes $u \in \mathbb{U}$ and their current covering radii $curCR_u$. Then, for each node u picked from *COH* in increasing $curCR_u$ order, we do the following. We add the edges of NHA_u to CD_u, where CD_u refers to the adjacency list of u in *CD*. (Due to the size limitation it is likely that some of the u's current neighbors do not belong to CD_u.) Then, we compute shortest paths from all u's ancestors to discard objects. We do this by calling **extractFrom** for all u's ancestors, marking as `EXTRACTED` as many objects as we can (this way we reuse these already computed distances in order to discard other objects). Then, we finish the query by using the procedure **finish$k$NNQ**. Finally, we delete CD_u. Figure 4.7 shows the recursive-partition-based algorithm (**KNNrp**).

Procedure **finish$k$NNQ** (see Figure 4.8) receives the node u to be processed. First, it adds all u's ancestor siblings to $\mathcal{C}$, and it manages $\mathcal{C}$ as a priority queue. Later, it takes objects w from $\mathcal{C}$ in increasing w_{pr} order, and processes w according to the following rules:

1. If w was already marked as `EXTRACTED`, we add its children $\{w_l, w_r\}$ to $\mathcal{C}$;
2. If "$\mathbb{U}$ is fixed" applies for w and u, and $d(u,w) \notin CD$, we add $\{w_l, w_r\}$ to $\mathcal{C}$; or
3. If we have $d(u,w)$ stored in *CD*, we retrieve it, else we compute it and use "d is symmetric". Then, if $d(u,w) < curCR_u + w_{pr}$, we have region intersection between $(u, curCR_u)$ and (w, w_{pr}), thus we add $\{w_l, w_r\}$ to $\mathcal{C}$. Next, if $d(u,w) > curCR_u + NND_w$, we use **extractFrom** considering $NHA \cup CD$ as a graph for computing shortest paths from w limited to $d(u,w) - curCR_u$, so as to discard as many objects as we can by marking them as `EXTRACTED`. Note that we call **extractFrom** for all u's ancestors' siblings.

*K*NNrp (Integer k, ObjectSet $\mathbb{U}$)
1. **For each** $u \in \mathbb{U}$ **Do** // initializations of
2. $NHA_u \leftarrow \{(\text{NULL}, \infty), \ldots, (\text{NULL}, \infty)\}$ // k pairs
3. $(u_p, u_l, u_r, u_{pr}) \leftarrow (\text{NULL}, \text{NULL}, \text{NULL}, 0)$ // the tree DCT
4. $CD_u \leftarrow \emptyset$ // the cache of distances
5. $(\texttt{ROOT}_p, \texttt{ROOT}_l, \texttt{ROOT}_r, \texttt{ROOT}_{pr}) \leftarrow (\text{NULL}, \text{NULL}, \text{NULL}, \infty)$ // the fictitious node
6. **division**(ROOT, $\mathbb{U}$) // populates CD and NHA, see Figure 4.4
7. **For each** $u \in \mathbb{U}$ **Do** COH.**insert**$(u, curCR_u)$
8. **While** $|COH| > 0$ **Do**
9. $(u, curCR_u) \leftarrow COH$.**extractMin**()
10. $CD_u \leftarrow CD_u \cup NHA_u$
11. **For each** $a \in$ **ancestors**(u) **Do**
12. **If** $CD_{ua} > curCR_u + NND_a$ **Then extractFrom**$(a,\ u,\ CD_{ua} - curCR_u)$
13. **finish**k**NNQ**$(u,\ \mathbb{U},\ |CD| < 8n \ln n)$
14. $CD_u \leftarrow \emptyset$ // we free the adjacency list of u in CD
15. **Return** NHA

Figure 4.7: Recursive-partition-based algorithm (*K*NNrp). **ancestors**(u) refers to the DCT tree. $n = |\mathbb{U}|$. DCT is managed as a global structure. For each $u \in \mathbb{U}$, (i) NHA_u is a priority queue of size k whose elements are of the form (object, $value$) sorted by decreasing $value$, (ii) $curCR_u$ is an alias for NHA_u.**findMax**().$value$, and (iii) NND_u is the distance from u towards its current nearest neighbor. $COH = \{(u, curCR_u), u \in \mathbb{U}\}$ is a priority queue sorted by ascending values of $curCR_u$. $CD \subset \mathbb{U}^2 \times \mathbb{R}^+$ is the cache of computed distances.

4.2.3 Pivot-based Algorithm

Pivot-based algorithms have good performance in low-dimensional spaces, but worsen quickly as the dimension grows. However, our methodology compensates this failure in medium and high dimensions. In this algorithm we use $O(kn)$ space in NHA and $O(n \log n)$ space to store the pivot index.

4.2.3.1 First Stage: Construction of the Pivot Index

We select at random a set of pivots $\mathcal{P} = \{p_1, \ldots, p_{|\mathcal{P}|}\} \subseteq \mathbb{U}$, and store a table of $|\mathcal{P}|n$ distances $d(p_j, u),\ j \in \{1, \ldots, |\mathcal{P}|\},\ u \in \mathbb{U}$. We give the same space in bytes to the table as that of the cache of distances plus the division control tree of the recursive-partition-based algorithm, so $|\mathcal{P}|n = |CD| + |DCT|$. For example, in a medium-sized database $(n < 2^{16})$ using single-precision floating point numbers to store the distances in CD, we need 6 bytes (2 for the node id plus 4 for the distance), so for each distance we store 1.5 cells in the table. Each node in DCT uses 10 bytes (6 for the three node ids plus 4 for the partition radius), so for each node we store 2.5 cells in the table. Then, in bytes, we have $4|\mathcal{P}|n = 6 \cdot 8n \ln n + 10n$, that is, $|\mathcal{P}| = 12 \ln n + 2.5 = O(\log n)$.

finish$k$NNQ(Object u, ObjectSet $\mathbb{U}$, Boolean *addCD*)
1. $\mathcal{C} \leftarrow \{(c, c_{pr}), c \in$ **sibling**(**ancestors**$(u))\}$
2. **While** $|\mathcal{C}| > 0$ **Do**
3. $(w, w_{pr}) \leftarrow \mathcal{C}$.**extractMin**()
4. $l \leftarrow w_l$, $r \leftarrow w_r$ // children of w
5. **If** $CD_{uw} =$ `EXTRACTED` **Then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{(l, l_{pr}), (r, r_{pr})\}$
6. **Else If** $w \notin COH$ AND $curCR_u \leq curCR_w$ AND $(u, w) \notin CD$ **Then**
7. CD.**insert**(u, w, `EXTRACTED`), $\mathcal{C} \leftarrow \mathcal{C} \cup \{(l, l_{pr}), (r, r_{pr})\}$ // $\mathbb{U}$ is fixed
8. **Else**
9. $d_{uw} \leftarrow$ **computeDistance**(u, w, *addCD*)
10. **If** $d_{uw} < curCR_u + w_{pr}$ **Then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{(l, l_{pr}), (r, r_{pr})\}$
11. **If** $d_{uw} > curCR_u + NND_w$ **Then extractFrom**(w, u, $d_{uw} - curCR_u$)

computeDistance(Object u, Object w, Boolean *addCD*)
1. **If** $(u, w) \in CD$ **Then Return** CD_{uw}
2. **Else** $d_{uw} \leftarrow d(u, w)$, CD.**insert**(u, w, d_{uw})
3. $NND_u \leftarrow \min(NND_u, d_{uw})$, $NND_w \leftarrow \min(NND_w, d_{uw})$
4. **If** $d_{uw} < curCR_u$ **Then** NHA_u.**extractMax**(), NHA_u.**insert**(w, d_{uw})
5. **If** $d_{uw} < curCR_w$ **Then**
6. $oldCR \leftarrow curCR_w$
7. NHA_w.**extractMax**(), NHA_w.**insert**(u, d_{uw})
8. COH.**decreaseKey**$(w, oldCR - curCR_w)$
9. **If** *addCD* **Then**
10. CD.**insert**(w, u, d_{wu}) // we store the symmetric distance if we can
11. **Return** CD_{uw}

Figure 4.8: Procedures **finish$k$NNQ** and **computeDistance**. **ancestors**(u) and **sibling**(a) refer to the DCT tree, which is managed as a global structure. CD_{uw} refers to the cached distance between u and w. $curCR_u$ is an alias for NHA_u.**findMax**().$value$. NND_u is the distance from u towards its current nearest neighbor.

Note that, since pivots $p \in \mathcal{P}$ compute distances towards all the other objects in $\mathbb{U}$, once we compute the pivot table they have already solved their respective k-nearest neighbors. Likewise, using the symmetry of d, all the non-pivot objects (in $\mathbb{U} - \mathcal{P}$) have pivots in their respective queues in *NHA*.

4.2.3.2 Second Stage: Computing the kNNG

We start by explaining how to solve a single $NN_k(u)$ query by using the pivot table in a range-optimal fashion. Next we show how to use the ingredients to share costs, and take benefit of the whole kNNG construction process.

Solving a Single $NN_k(u)$ Query with the Pivot Table. Because of the triangle inequality, for each $v \in \mathbb{U}$ and $p \in \mathcal{P}$, $|d(v,p) - d(u,p)|$ is a lower bound on the distance $d(u,v)$. Then, we use the candidate set $\mathcal{C}$ to store the maximum lower bound of $d(u,v)$ for each element v using all the pivots, that is $\mathcal{C}_v = \max_{p\in\mathcal{P}}\{|d(v,p) - d(u,p)|\}$. Therefore, we can discard non-relevant objects v such that $\mathcal{C}_v \geq curCR_u$. Figure 4.9 shows the concept graphically. In this case $\mathcal{C}_v = \max\{|d(v,p_1) - d(u,p_1)|, |d(v,p_2) - d(u,p_2)|\} = |d(v,p_2) - d(u,p_2)| \geq curCR_u$, thus v is discarded by p_2. Thus, we store the $\mathcal{C}_v$ values in a priority queue

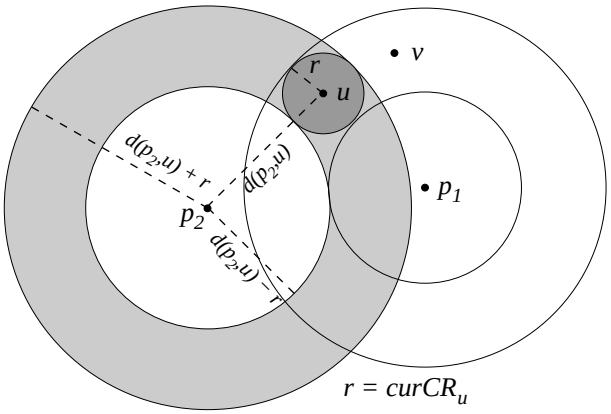


Figure 4.9: Solving queries with pivot-based indices. Despite pivot p_1 cannot discard object v, p_2 can because v is outside its ring.

$SortedC = \{(v, \mathcal{C}_v), v \in \mathbb{U} - (\mathcal{P} \cup NHA_u \cup \{u\})\}$ sorted by increasing $\mathcal{C}_v$ order, and review $SortedC$ until we reach an object v such that $\mathcal{C}_v \geq curCR_u$.

Solving all the Queries Jointly by Using the Ingredients. As pivots have already solved their k-nearest neighbors, we only have to complete $n - |\mathcal{P}|$ queries for objects $u \in \mathbb{U} - \mathcal{P}$. These objects already have candidates in their respective queues in NHA, the k-nearest pivots. (And these candidates are refined as long as the process progresses.) Note that, if we solve the $NN_k(u)$ query for an object u whose $curCR_u$ is small, we have higher chance both of performing a moderate work with the current query and also of improving future queries, as small $curCR_u$ queries has more selective power and they should compute distances towards close objects (since far away objects should be easily discarded). So, after computing the table, we sort objects $u \in \mathbb{U} - \mathcal{P}$ in COH by increasing $curCR_u$ order.

Even though each $NN_k(u)$ query is solved in a range-optimal fashion, the ingredients allow us to avoid some distance computations. In fact, each time we compute the distance from the current node u towards an object v we use all the ingredients both to discard objects during the computation of the current k-nearest neighbor query for u, and to verify whether the symmetric query for v can be improved (and then reducing $curCR_v$).

Note that, for each $NN_k(u)$ query for objects $u \in \mathbb{U} - \mathcal{P}$, the distance calculations we can avoid are the ones computed when we review $SortedC$ until we get an object v such

that $\mathcal{C}_v > curCR_u$ or $Sorted\mathcal{C}$ gets empty. So, when we pick an object v from $Sorted\mathcal{C}$ in ascending $\mathcal{C}_v$ order, we start by checking if "$\mathbb{U}$ is fixed" applies for u and v. In such case, we avoid the distance computation and process the next node.

Otherwise, we compute the distance $d_{uv} = d(u, v)$, and if $d_{uv} < curCR_u$ we add v to NHA_u (this could reduce $curCR_u$). Also, using "d is symmetric", if $d_{uv} < curCR_v$ we refine NHA_v. The latter means that we insert u into v's k-nearest neighbor candidate set, $curCR_v$ can reduce, and consequently we reposition v in COH if necessary. Naturally, this may change the order in which queries are solved. Note that, if we only were solving range-optimal $NN_k(q)$ queries, the order in which nodes in COH are processed would not necessary reduce the final number of distance computations (but it can reduce the CPU time). However, this is not the case when considering the effect of the ingredients. For instance, they allow us to avoid some symmetric distance computations in future queries, and also avoid distance computations in the current query using the graph distance.

Finally, we use NHA as a graph to delete from $Sorted\mathcal{C}$ all the objects w such that $d_{NHA}(v, w) \leq d(u, v) - curCR_u$ avoiding the direct comparison. We perform this in procedure **extractFrom**, depicted in Figure 4.10, which is a shortest-path algorithm with propagation limited to $d(u, v) - curCR_u$. To avoid exponential complexity due to multiple checking, we only propagate distances through objects in $Sorted\mathcal{C}$.

```
extractFrom(Object v, Object u, ℝ+ limit, PriorityQueue SortedC)
1.  minHeap.insert(v, 0) // ordered by the second component
2.  While |minHeap| > 0 Do
3.      (c, d_vc) ← minHeap.extractMin()
4.      For each (w, d_cw) ∈ NHA_c Do
5.          If w ∈ SortedC AND d_vc + d_cw ≤ limit Then
6.              SortedC.delete(w)
7.              If d_vc + d_cw + NND_w ≤ limit Then minHeap.insert(w, d_vc + d_cw)
```

Figure 4.10: Auxiliary procedure **extractFrom**. **extractFrom** is a shortest-path algorithm limited to $limit = d(u, v) - curCR_u$. It discards objects w from v by deleting them from $Sorted\mathcal{C}$. NND_w is the distance from w towards its current nearest neighbor.

We can reduce the CPU time if we take into account three facts. First, it is not always necessary to calculate the maximum difference C_v for each node v. In practice, to discard v it is enough to find *some* lower bound greater than $curCR_u$, not the maximum. Thus we can learn that $\mathcal{C}_v \geq curCR_u$ without fully computing the maximum of $\mathcal{C}_v$ formula. Second, it is not necessary to add all objects in $\mathbb{U} - (\mathcal{P} \cup NHA_u \cup \{u\})$ to $Sorted\mathcal{C}$: if $\mathcal{C}_v$ is already greater or equal to $curCR_u$ then v will not be reviewed. Third, when we compute shortest paths from v, we can save some CPU time if we use the distance of the current v's nearest neighbor, NND_v, to know whether it is worthless propagating paths from it. Figure 4.11 depicts the pivot-based algorithm (K**NNpiv**) and its auxiliary functions.

```
KNNpiv (Integer k, ObjectSet 𝕌)
 1.  For each u ∈ 𝕌 Do NHA_u ← {(NULL, ∞), ..., (NULL, ∞)} // k pairs
 2.  For each u ∈ 𝕌 Do NND_u ← ∞
 3.  For each u ∈ 𝕌 − 𝒫 Do COH.insert(u, ∞)
 4.  For each p ∈ 𝒫, u ∈ 𝕌 Do T[p][u] ← d(p, u), useDistance(p, u, T[p][u])
 5.  While |COH| > 0 Do
 6.      (u, curCR_u) ← COH.extractMin()
 7.      Sorted𝒞 ← heapify(calc𝒞(u, 𝕌, 𝒫, T, curCR_u))
 8.      While |Sorted𝒞| > 0 Do
 9.          (v, 𝒞_v) ← Sorted𝒞.extractMin()
10.          If 𝒞_v ≥ curCR_u Then Break // we have finished the current query
11.          If v ∈ COH OR curCR_u > curCR_v Then
                 // as "𝕌 is fixed" did not apply, we compute the distance
12.              d_uv ← d(u, v), useDistance(u, v, d_uv), limit ← d_uv − curCR_u
13.              If limit ≥ NND_v Then extractFrom(v, u, limit, Sorted𝒞)
14.  Return NHA
```

```
useDistance(Object u, Object v, ℝ⁺ d_uv)
1.  NND_u ← min(NND_u, d_uv), NND_v ← min(NND_v, d_uv)
2.  If d_uv < curCR_u Then NHA_u.extractMax(), NHA_u.insert(v, d_uv)
3.  If d_uv < curCR_v Then
4.      oldCR ← curCR_w, NHA_v.extractMax()
5.      NHA_v.insert(u, d_uv), COH.decreaseKey(v, oldCR − curCR_v)
```

```
calc𝒞(Object u, ObjectSet 𝕌, ObjectSet 𝒫, Table T, ℝ⁺ curCR_u)
1.  For each v ∈ 𝕌 − 𝒫 Do
2.      𝒞_v ← 0
3.      For each p ∈ 𝒫 Do
4.          If |T[v][p] − T[u][p]| > 𝒞_v Then
5.              𝒞_v ← |T[v][p] − T[u][p]|
6.              If 𝒞_v ≥ curCR_u Then Break
    // marking the node u and its current neighbor candidates with ∞
7.  𝒞_u ← ∞, For each v ∈ NHA_u Do 𝒞_v ← ∞
8.  Return {(v, 𝒞_v), 𝒞_v < curCR_u} // returning objects v with small enough 𝒞_v
```

Figure 4.11: Pivot-based algorithm (**KNNpiv**) and its auxiliary procedures. For each $u \in \mathbb{U}$, (i) NHA_u is a priority queue of size k whose elements are of the form $(id, value)$ sorted by decreasing $value$, (ii) $curCR_u$ is an alias for NHA_u.**findMax**().$value$, and (iii) NND_u is the distance from u towards its current nearest neighbor. $COH = \{(u, curCR_u), u \in \mathbb{U}\}$ is a priority queue sorted by ascending values of $curCR_u$. T is the pivot table, $\mathcal{P} \subset \mathbb{U}$, $n = |\mathbb{U}|$, $|\mathcal{P}| = 12 \ln n + 2.5$. **useDistance** exploits the symmetry of d. $CD \subset \mathbb{U}^2 \times \mathbb{R}^+$ is the cache of computed distances.

4.3 Using the kNNG for Proximity Searching

> In search of my mother's garden I found my own.
>
> – Alice Walker

As we said in Section 2.2.5, page 26, kNNGs can be used for many purposes. From now on, we show how to use the kNNG to speed up metric queries. That is, we propose a new class of proximity searching algorithms using the kNNG as the data structure for searching $\mathbb{U}$. To the best of our knowledge, this is the first approach using the kNNG for proximity searching purposes[1].

The core idea is to use the kNNG to estimate both an upper and a lower bound of distances from the metric database elements towards the query. Once we compute $d(q, u)$ for some u we can upper bound the distance from q to all database objects (in its connected component). We can also lower bound the distance from the query to the neighbors of u (using it as a pivot). The upper bound allows the elimination of elements far from the query, whilst the lower bound can be used to test if an element is in the query outcome.

As we explain later (Sections 4.3.1 and 4.3.2), this family of algorithms has a large number of design parameters affecting its efficiency (not its correctness). We tried to explore all the parameters experimentally in Section 4.4.

The guidelines for this new class of algorithms are the following:

- Use the kNNG as a navigational device.
- Try different criteria to select the next object to review.
- Use the graph to estimate distances.
- Avoid to concentrate efforts in the same graph zone.

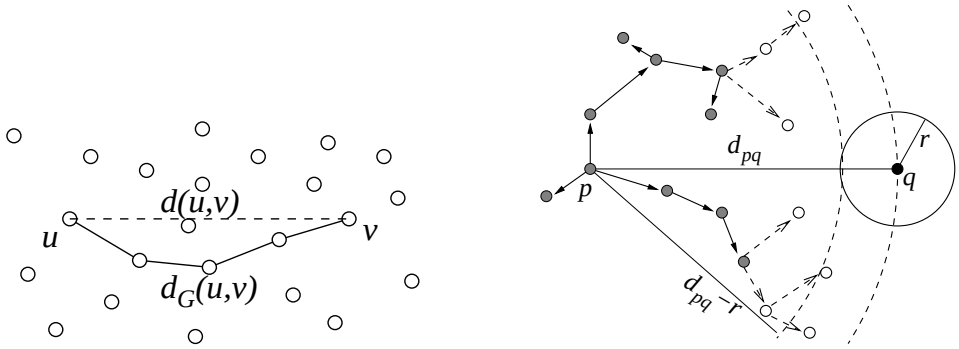
We choose two sets of heuristics giving rise to two range query algorithms, which differ in the criterion to select objects to review. The first aims to select objects with the most non-discarded neighbors. The second aims to select far apart objects. Based on each technique we propose two algorithms for solving k-nearest neighbor queries.

4.3.1 kNNG-based Range Query Algorithms

Assume that we are solving the range query (q, r). Our underlying index is a graph G, whose edge weights correspond to true distances between object pairs. Thus, just as in

[1] Sebastian and Kimia [SK02] give a metric searching algorithm based on kNNGs, however their approach is just a heuristic, since it does not guarantee to find the correct answers. See Section 2.4.4.2 for further details.

kNNG construction algorithms, we use the sum of edge weights of the shortest path between two objects u and v, $d_G(u,v)$, as an upper bound of the true distance between them, $d(u,v)$. Figure 4.12(a) illustrates.



(a) We upper bound the distance $d(u,v)$ with the length of the shortest path $d_G(u,v)$.

(b) Propagating shortest path computations over the graph.

Figure 4.12: Approximating the distances in an arbitrary graph. In (a), using the kNNG to upper bound the distances. In (b), discarding gray objects which have a distance upper bound lower than $d_{pq} - r$.

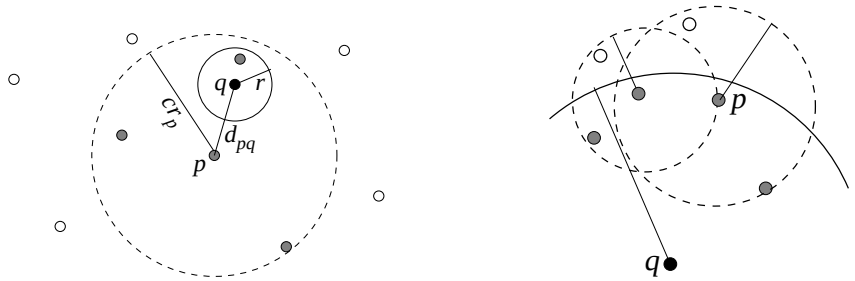
A generic graph-based approach for solving range queries consists in starting with a set of candidate nodes $\mathcal{C}$ of the smallest set provably containing (q,r). A fair choice for an initial $\mathcal{C}$ is the whole database $\mathbb{U}$. Later, we iteratively extract an object p from $\mathcal{C}$ and if $d(p,q) \leq r$ we report p as part of the query outcome. Otherwise, we delete all objects v such that $d_G(p,v) < d(p,q) - r$. Figure 4.12(b) illustrates this: we discard all the gray nodes because their distance estimations are small enough. We repeat the above procedure as long as $\mathcal{C}$ has candidate objects. In the following, we improve this generic approach by using the kNNG properties.

4.3.1.1 Using Covering Radii

Each node p in the kNNG has a covering radius cr_p (which is the distance towards its k-th neighbor). Let (p, cr_p) be the ball centered at p with radius cr_p. If the query ball (q,r) is contained in (p, cr_p) we can make $\mathcal{C} \leftarrow \mathcal{C} \cap (p, cr_p)$, drastically reducing the candidate set. We call this object a *container*. Figure 4.13(a) illustrates.

Actually, we can keep track of the best fitted container, considering both its distance to the query and its covering radius using the value $cr_p - d(p,q)$. The best fitted object will be the one having the largest difference, as this maximizes the chance of containing the query ball. On the other hand, this may favor objects $u \in \mathbb{U}$ with larger balls (u, cr_u), which might seem disadvantageous. However, all these balls have $k+1$ objects inside, and hence are equally good in this sense.

For this sake, we define the container (c_{id}, c_{value}), which is initialized to $(\text{NULL}, -\infty)$, and every time we find an object u such that $cr_u - d(u,q)$ is greater than the current c_{value},



(a) If the ball (p, cr_p) covers (q, r), we make $\mathcal{C} \leftarrow \mathcal{C} \cap (p, cr_p)$.

(b) If we find an object $p \in (q, r)$, we check its neighborhood.

Figure 4.13: Using the kNNG features. In (a), using the container. In (b), checking the neighborhood.

we update the container to $(u, cr_u - d(u, q))$. So, if the container covers the whole query ball, that is $r < c_{value}$, we make $\mathcal{C} \leftarrow \mathcal{C} \cap (u, cr_u)$ as stated above. We do this in procedures **useContainerRQ** for range queries (see Figure 4.14) and **useContainerNNQ** for k-nearest neighbor queries (see Figure 4.20). The difference between them comes from the fact that in range queries we must report all the objects inside the query ball, thus we test with the strict inequality; whilst in k-nearest neighbor queries we need any set satisfying the query, thus we test whether the current query radius is lower than or equal to c_{value}.

The probability of hitting a case to apply this property is low; but it is simple to check and the low success rate is compensated with the dramatic shrink of $\mathcal{C}$ when applied. Note that the container is useful not only in k-nearest neighbor queries, where the query covering radius reduces as long as the query progresses; but also in range queries, since the container value c_{value} increases while the query is being solved, improving the chances of covering the query ball.

---

useContainerRQ (Object $p, \mathbb{R}^+ d_{pq}, \mathbb{R}^+ radius$)
// kNNG G, MinHeap $\mathcal{C}$ and Container (c_{id}, c_{value}) are global variables
1. **If** $c_{value} < cr_p - d_{pq}$ **Then** $(c_{id}, c_{value}) \leftarrow (p, cr_p - d_{pq})$
2. **If** $radius < c_{value}$ **Then**
3. **For each** $u \in \mathcal{C} - NN_k(p)$ **Do extractNode**(u) // $\mathcal{C} \leftarrow \mathcal{C} \cap (p, cr_p)$

Figure 4.14: Auxiliary procedure **useContainerRQ**. $NN_k(u)$ returns the adjacency list (the k nearest neighbors) of u in the format (v, d_{uv}), where v is the node and d_{uv} is the distance between them.

4.3.1.2 Propagating in the Neighborhood of the Nodes

Since we are working over a graph built by an object closeness criterion, if an object p is in (q, r), it is likely that some of its neighbors are also in (q, r). Moreover, since the out-degree

of a kNNG is a small constant, spending some extra distance evaluations on neighbors of processed nodes does not add a large overhead to the whole process. So, when we find an object belonging to (q, r), it is worth to examine its neighbors. Figure 4.13(b) illustrates. Yet, we must take care not to repeat distance calculations.

For this sake, we define a priority queue l to manage the objects. Suppose we start from node p, as shown in the figure. We begin by storing $(p, d(p, q))$ in l, and repeat the following until l gets empty: We extract the minimum element (u, d_{uq}) from l. If $d_{uq} < r$ we report u. Then, we update the container with (u, d_{uq}) by calling **useContainerRQ**. Next, we use u as a pivot to discard some of its neighbors avoiding the direct comparison. With respect to the non-discarded ones, we compute the distance from them to the query, and store the pairs (object, distance) in l. To avoid cycles we only propagate over nodes in $\mathcal{C}$, and each time we compute a distance we discard the node from $\mathcal{C}$. Note that every time we hit an answer we recursively check all of its neighbors (so, we stop the propagation for nodes out of the query outcome).

We implement this in procedure **checkNeighborhood**, depicted in Figure 4.15. Both of our range query algorithms use this idea, yet the first one introduce a small variation we describe soon.

```
checkNeighborhood (Object p, ℝ+ d_pq)
// kNNG G, ℝ+ radius, MinHeap C and Container (c_id, c_value) are global variables
1.  l ← ∅, l.insert(p, d_pq) // sorted by increasing distance
2.  While |l| > 0 Do
3.      (u, d_uq) ← l.extractMin()
4.      If d_uq ≤ radius Then Report u
5.      useContainerRQ(u, d_uq, radius)
6.      For each (v, d_uv) ∈ NN_k(u) ∩ C Do // we check non discarded neighbors
7.          If d_uv ∉ [d_uq − radius, d_uq + radius] Then
8.              extractNode(v) // using u as a pivot
9.          Else If d_uq ≤ radius Then l.insert(v, d(v, q)), extractNode(v)
```

Figure 4.15: Auxiliary procedure **checkNeighborhood**. $NN_k(u)$ returns the adjacency list (the k nearest neighbors) of u in the format (v, d_{uv}), where v is the node and d_{uv} is the distance between them.

4.3.1.3 Working Evenly in All Graph Regions

Since we use path expansions from some nodes it is important to choose them scattered in the graph to avoid concentrating all the efforts in few graph regions. A good idea is to select elements far apart from q and from the previous selected nodes, because these elements would have more potential for discarding non-relevant objects. Unfortunately, the selection of distant objects cannot be done by directly computing the distance to q.

Instead, we can estimate *how much visited* is some region. In fact, our two range query algorithms differ essentially in the way we select the next node to review.

4.3.1.4 First Heuristic for Range Queries (kNNG**RQ1**)

In this heuristic we prefer to start shortest path computations from nodes with few discarded neighbors. We also consider two tie-breaking criteria. The first is to prefer nodes with smaller covering radius. The second is to prefer the least visited nodes. More precisely, to select a node, we consider the following:

1. The number of discarded neighbors. Nodes with few discarded neighbors have larger discarding potential, so they can effectively reduce the number of distance computations performed to solve the query.
2. The size of the covering radius. Objects having small covering radius, that is, very close neighbors, have high chances of discarding them (note that if $cr_u < d(u,q) - r$, all the neighbors are discarded). It is also likely that distance estimations computed from these objects would have tighter upper bounds.
3. The number of times the node was traversed in a path expansion when computing shortest paths. Note that once we compute shortest paths from node u, we discard all the nodes whose upper bound of real distances are lower than $d_{uq} - r$. But nodes whose shortest path distances are in the range $[d_{uq} - r, d_{uq})$ are not so far from u. Thus, starting a new shortest path expansion from them would have lower chance of discarding objects that survived the previous expansion. Therefore, we prefer a node that has been checked few times to scatter the search effort on the whole graph.

The above measures are combined in Eq. (4.1):

$$p = \text{argmin}_{u \in \mathcal{C}}\{|\mathbb{U}| \cdot (dn_u + f(u)) + \#visits\}, \tag{4.1}$$

with $f(u) = \frac{cr_u - cr_{min}}{cr_{max} - cr_{min}} \in [0,1]$ ($cr_{min} = \min_{u \in \mathbb{U}}\{cr_u\}$ and $cr_{max} = \max_{u \in \mathbb{U}}\{cr_u\}$), and dn_u represents the number of discarded neighbors of u. Note that Eq. (4.1) mainly selects nodes with few discarded neighbors and uses both the covering radius and the number of visits as tie-breaking rules.

The equation is computed iteratively for every node in the kNNG. The candidates are stored in a priority queue $\mathcal{C}$ in the form (object,*value*), sorted by increasing *value*, where *value* is computed using Eq. (4.1). We initialize $\mathcal{C}$ as $\big\{\big(u, |\mathbb{U}|f(u)\big), u \in \mathbb{U}\big\}$. Then, when solving the query, each time we discard an object v, we call procedure **extractNode**, which does the following: (1) it deletes v from $\mathcal{C}$ by calling procedure **delete**, and (2) for non-discarded v's neighbors, it increases their values by $|\mathbb{U}|$. Finally, we slightly modify procedure **checkNeighborhood** to take into account the number of visits for each non-discarded node outside of the query outcome, by increasing its *value* in 1. The modified

procedure **checkNeighberhoodRQ1** is depicted in Figure 4.16, where we add a 10-th line with the following pseudocode: **Else** $\mathcal{C}$.**increaseKey**(v, 1).

Therefore, to obtain the next node to be reviewed we call procedure **nextToReview**, which computes $\mathcal{C}$.**findMin**(). Figure 4.16 depicts procedures **extractNode** and **nextToReview**.

```
checkNeighborhoodRQ1 (Object p, ℝ+ d_pq)
// kNNG G, ℝ+ radius, MinHeap C and Container (c_id, c_value) are global variables
1.  l ← ∅, l.insert(p, d_pq) // sorted by increasing distance
2.  While |l| > 0 Do
3.      (u, d_uq) ← l.extractMin()
4.      If d_uq ≤ radius Then Report u
5.      useContainerRQ(u, d_uq, radius)
6.      For each (v, d_uv) ∈ NN_k(u) ∩ C Do // we check non discarded neighbors
7.          If d_uv ∉ [d_uq − radius, d_uq + radius] Then
8.              extractNode(v) // using u as a pivot
9.          Else If d_uq ≤ radius Then l.insert(v, d(v,q)), extractNode(v)
10.         Else C.increaseKey(v, 1)
```

```
nextToReview ()
1.  Return C.findMin() // MinHeap C is a global variable
```

```
extractNode (Object v)
// kNNG G, MinHeap C and ObjectSet 𝕌 are global variables
1.  C.delete(v)
2.  For each w ∈ NN_k(v) Do C.increaseKey(w, |𝕌|)
```

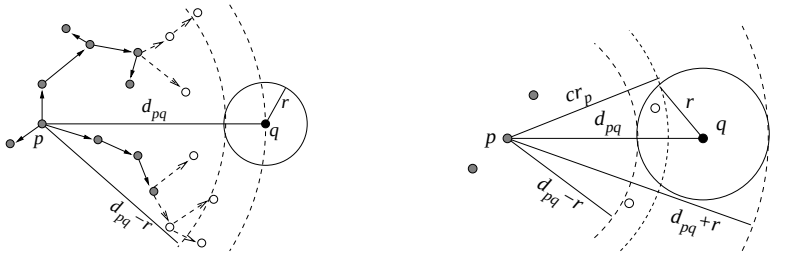
Figure 4.16: *k*NNG**RQ1**'s auxiliary procedures **checkNeighborhoodRQ1**, **nextToReview** and **extractNode**. $NN_k(u)$ returns the adjacency list (the k nearest neighbors) of u in the format (v, d_{uv}), where v is the node and d_{uv} is the distance between them.

As usual, to compute the graph distance we use Dijkstra's all-shortest-path algorithm with limited propagation. This time, the threshold is slightly larger that $d(u, q)$, since spending time computing distance estimations beyond that bound cannot be used to discard nodes, but just to account for node visits.

Now we describe algorithm *k*NNG**RQ1**. It begins by initializing the container as (NULL, $-\infty$). Then, it finds the maximum and minimum covering radius cr_{max} and cr_{min}, and uses them to initialize $\mathcal{C}$ as $\{(u, |\mathbb{U}|f(u)), u \in \mathbb{U}\}$.

After initializations, *k*NNG**RQ1** calls **nextToReview** to get the next node p to review, computes the distance $d_{pq} \leftarrow d(p, q)$, and extracts p from $\mathcal{C}$ with **extractNode**. If $d_{pq} \leq r$ it starts to navigate finding other objects relevant to q by calling **checkNeighborhoodRQ1** from p. Then, it calls procedure **useContainerRQ**. Finally, if there is no intersection between the ball (p, cr_p) and the query ball, it uses

Dijkstra's shortest-path algorithm to compute distance estimations from p towards all the other objects in $\mathbb{U}$, $d_G = \{d_G(p,v), v \in V\}$, limiting the expansion just to past d_{pq}. So it uses **extractNode** to discard objects v such that $d_G[v] < d_{pq} - r$ and increases *value* in $\mathcal{C}$ by 1 for the objects w such that $d_{pq} - r \leq d_G[w] < d_{pq}$. Figure 4.17(a) illustrates. Otherwise, there is intersection, thus it uses p as a pivot to discard neighbors $v \in NN_k(p)$ such that $d(p,v) \notin [d_{pq} - r, d_{pq} + r]$ (by calling **extractNode**), and increases *value* in $\mathcal{C}$ by 1 for the other neighbors. Figure 4.17(b) illustrates, and Figure 4.18 depicts the whole algorithm.



(a) The balls do not intersect each other. (b) The balls intersect each other.

Figure 4.17: Implementing heuristics for kNNG**RQ1**. In (a), we extract gray objects which have a distance estimation lower that $d_{pq} - r$ and count visits for the white ones, which have estimations lower than d_{pq}. In (b), we use p as a pivot discarding its gray neighbors when the distance from p towards them is not in $[d_{pq} - r, d_{pq} + r]$, else, we count the visit for the white nodes.

4.3.1.5 Second Heuristic for Range Queries (kNNG**RQ2**)

A different way to select a scattered element set is by using the distance estimations due to shortest path computations. More precisely, we assume that if two nodes are far apart according to the distance estimation measured over the graph, then they are also far apart using the original metric space distance. Therefore, the idea is to select the object with the largest sum of distance estimations to all the previously selected objects. Thus, kNNG**RQ2** can be seen as a heuristic trying to select outliers for starting shortest path computations.

To do this, we modify procedure **nextToReview** so that it considers the sum of the shortest paths towards previous selected nodes (as we do not have the whole $\mathbb{U} \times \mathbb{U}$ distance matrix). Thus the candidate set $\mathcal{C}$ is now a max-heap whose elements are of the form (object, *value*), sorted by decreasing *value*. Note that it is possible that the kNNG has several connected components. Then, at the beginning we start by picking objects in every component (in fact, for each component, we pick the object having the smallest covering radius).

Upon some preliminary experiments we verified that the order in which elements are selected when calling procedure **nextToReview** is mainly determined by the first shortest path computations. As a matter of fact, after computing approximately 100 times the shortest paths, the order in which nodes are extracted from $\mathcal{C}$ remains almost changeless.

kNNG**RQ1** (Object q, $\mathbb{R}^+ radius$, kNNG G, ObjectSet $\mathbb{U}$)
1. Container $(c_{id}, c_{value}) \leftarrow$ (NULL, $-\infty$), MinHeap $\mathcal{C} \leftarrow \emptyset$
   // elements of $\mathcal{C}$ are of the form (object, $value$), sorted by increasing $value$
2. $cr_{max} \leftarrow \max_{u \in \mathbb{U}}\{cr_u\}$, $cr_{min} \leftarrow \min_{u \in \mathbb{U}}\{cr_u\}$, $cr_{diff} \leftarrow cr_{max} - cr_{min}$
3. **For each** $u \in \mathbb{U}$ **Do** $\mathcal{C}$.**insert**$(u, |\mathbb{U}|\,(cr_u - cr_{min})/cr_{diff})$
4. **While** $|\mathcal{C}| > 0$ **Do**
5. $(p, value) \leftarrow$ **nextToReview**()
6. $d_{pq} \leftarrow d(p, q)$, **extractNode**$(p)$
7. **If** $d_{pq} \leq radius$ **Then checkNeighborhoodRQ1**(p, d_{pq})
8. **useContainerRQ**$(p, d_{pq}, radius)$
9. **If** $cr_p < d_{pq} - radius$ **Then** // propagating distances through the kNNG
10. $d_G \leftarrow$ **Dijkstra**$(G, p, (1+\epsilon)d_{pq}))$ // $|d_G| = |\mathbb{U}|$
11. **For each** $u \in \mathcal{C}$ **Do**
12. **If** $d_G[u] < d_{pq} - radius$ **Then extractNode**(u)
13. **Else If** $d_G[u] < d_{pq}$ **Then** $\mathcal{C}$.**increaseKey**$(u, 1)$
14. **Else**
15. **For each** $(u, d_{pu}) \in NN_k(p) \cap \mathcal{C}$ **Do** // using p as a pivot
16. **If** $d_{pu} \notin [d_{pq} - radius, d_{pq} + radius]$ **Then extractNode**(u)
17. **Else** $\mathcal{C}$.**increaseKey**$(u,1)$

Figure 4.18: Our first range query algorithm (kNNG**RQ1**). **Dijkstra**(G, p, x) computes distances over the kNNG G from p to all nodes up to distance x. $NN_k(p)$ returns the kNNG adjacency list of p (its k-nearest neighbors) in the format (u, d_{pu}) where u is the neighbor, and d_{pu} is the distance between them.

Therefore, for the sake of CPU time, in early iterations of the algorithm we compute full shortest-paths, but after it^* iterations (in our experiment we found that it is enough with $it^* = 100$ iterations) we come back to the limited propagation Dijkstra's version, where, this time, the propagation threshold is slightly larger than $d(u, q) - r$.

In kNNG**RQ2**, procedure **extractNode** simply extracts the node from $\mathcal{C}$. On the other hand, procedure **nextToReview** begins by choosing one non-discarded object from each component, and later it continues by selecting the element from $\mathcal{C}$ maximizing the sum of shortest path distances. The body of kNNG**RQ2** is almost the same of the previous algorithm. Indeed, the major differences come from implementing the mechanism to select the next node to review. Figure 4.19 depicts this algorithm.

4.3.2 kNNG-based Query Algorithms for Nearest Neighbors

Range query algorithms naturally induce nearest neighbor searching algorithms. (Since the size of the query outcome can be specified as an argument, in this section we omit the k from the k-nearest neighbor queries to avoid confusions.) For this sake, we use the following ideas:

```
kNNGRQ2 (Query q, ℝ+radius, kNNG G, ObjectSet 𝕌)
1.  Container (c_id, c_value) ← (NULL, −∞), Integer it ← 0
2.  MaxHeap 𝒞 ← ∅, MinHeap radii ← ∅ // elements of 𝒞 and radii are of the form
    // (object, value), sorted by decreasing/increasing value, respectively
3.  For each u ∈ 𝕌 Do 𝒞.insert(u, 0), radii.insert(u, cr_u)
4.  While |𝒞| > 0 Do
5.      (p, value) ← nextToReview()
6.      d_pq ← d(p, q), extractNode(p), it ← it + 1
7.      If d_pq ≤ radius Then checkNeighborhood(p, d_pq)
8.      useContainerRQ(p, d_pq, radius)
9.      If cr_p < d_pq − radius Then // propagating distances through the kNNG
10.         If (|radii| = 0) AND (it > it*) Then r_exp ← (1 + ε)(d_pq − radius)
11.         Else r_exp ← ∞
12.         d_G ← Dijkstra(G, p, r_exp) // |d_G| = |𝕌|
13.         For each u ∈ 𝒞 Do
14.             If d_G[u] < d_pq − radius Then extractNode(u)
15.             Else If (r_exp = ∞) AND (d_G[u] < ∞) Then
16.                 𝒞.increaseKey(u, d_G[u])
17.     Else
18.         For each (u, d_pu) ∈ NN_k(p) ∩ 𝒞 Do // using p as a pivot
19.             If d_pu ∉ [d_pq − radius, d_pq + radius] Then extractNode(u)
20.             Else 𝒞.increaseKey(u, d_pu)
```

```
nextToReview ()
    // MaxHeap 𝒞 and MinHeap radii are global variables
1.  While |radii| > 0 Do // first, we obtain pivots from different graph components
2.      cand ← radii.extractMin()
3.      If 𝒞.find(cand).value = 0 Then Return cand
4.  Return 𝒞.extractMax() // later, we return the current farthest object
```

```
extractNode (Object v)
1.  𝒞.delete(v) // MaxHeap 𝒞 is a global variable
```

Figure 4.19: Our second range query algorithm (kNNG**RQ2**) and auxiliary procedures **extractNode** and **nextToReview**. *it* is an auxiliary variable we use to account the iterations. **Dijkstra**(G, p, x) computes distances over the kNNG G from p to all nodes up to distance x. Since the kNNG can have more than one connected component, we start by selecting one node from each component. Also, as we explain in the Section 4.3.1.4, it is better to use small covering radius nodes. Thus, we define heap *radii* to order nodes by its covering radius, so that in **nextToReview** we start by choosing the node with the smallest covering radius form each component. Then, we pick the node with the largest sum of distance estimations.

- We manage an auxiliary priority queue nnc of nearest neighbor candidates of q known up to now. So, the radius cr_q is the distance from q to the farthest object in nnc.
- We simulate the nearest neighbor query $NN(q)$ using a range query (q, cr_q) of decreasing radius, whose initial radius cr_q is ∞. Note that, each time we find an object u such that $d_{uq} = d(u,q) < cr_q$, we extract from nnc its farthest object, and then add (u, d_{uq}) to the set nnc. This can reduce cr_q.
- Each non-discarded object u remembers its own lower bound $LB[u]$ of the distance from itself to the query. For each node its initial lower bound is 0.

According to the definition of nearest neighbor queries, in case of ties we can select any element set satisfying the query. So, in order to verify that the container covers the query, instead of using the strict inequality, in method **useContainerNNQ** we relax the condition to $radius \leq c_{value}$ (see Figure 4.20). This can be especially useful with discrete distance functions.

```
useContainerNNQ (Object p, ℝ+ d_pq, ℝ+ radius)
// kNNG G, MinHeap C and Container (c_id, c_value) are global variables
1.  If c_value < cr_p − d_pq Then (c_id, c_value) ← (p, cr_p − d_pq)
2.  If radius ≤ c_value Then
3.      For each u ∈ C − NN_k(p) Do extractNode(u) // C ← C ∩ (p, cr_p)
```

```
traverse (Object p, ℝ+ d_pq)
// kNNG G, MaxHeap nnc, MinHeap C and Container (c_id, c_value) are global variables
1.  (u, nearest_id, farthest_id) ← (NULL, p, p)
2.  (nearest_dist, farthest_dist) ← (d_pq, d_pq)
3.  nnc.extractMax(), nnc.insert(p, d_pq)
4.  While nearest_id ≠ NULL Do
5.      u ← nearest_id, nearest_id ← NULL
6.      For each v ∈ NN_k(u)_d ∩ C Do // we check non discarded neighbors
7.          d_vq ← d(v, q), extractNode(v)
8.          If d_vq ≤ cr_q Then nnc.extractMax(), nnc.insert(v, d_vq)
9.          If d_vq ≤ nearest_dist Then (nearest_id, nearest_dist) ← (v, d_vq)
10.         If d_vq ≥ farthest_dist Then (farthest_id, farthest_dist) ← (v, d_vq)
11.         If c_value ≤ cr_v − d_vq Then (c_id, c_value) ← (v, cr_v − d_vq)
12. Return (farthest_id, farthest_dist)
```

Figure 4.20: Auxiliary procedures **useContainerNNQ** and **traverse**. cr_q refers to the current query covering radius, that is, it is an alias for nnc.**findMax**().$value$. cr_v refers to the covering radius of the k-nearest neighbors of v in kNNG.

Note also that, if $d(u,q) < cr_q$, it is likely that some of the u's neighbors can also be relevant to the query, so we check all of them. However, since the initial radius is ∞ we change a bit the navigational scheme. In this case, instead of propagating through the

neighborhood, we start the navigation from the node to the query by jumping from one node to its neighbor only when the neighbor is closer to the query than the node itself. The underlying idea is to get close to the query as soon as possible. At the same time, we also remember the farthest object we check during the graph traversal. This way, we can use the farthest one to discard objects by computing shortest paths over the kNNG. Figure 4.21 illustrates. In the figure, we start at node p, and navigate towards q until we reach p_c. The node p_f was marked as the farthest object of this traversal. This is implemented in method **traverse** (see Figure 4.20), which returns p_f and the distance from p_f to q.

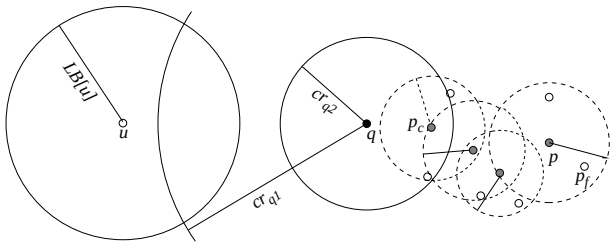


Figure 4.21: If we find an object $p \in NN(q)$ we traverse the graph towards q. Later, as cr_q decreases, it is possible to discard the node u when $LB[u] \geq cr_q$.

To solve the query, we split the discarding process into two stages. In the first, we compute the distance lower bounds for all the non-discarded nodes, by using the following formula. For each non-discarded node u, $LB[u]$ is computed as $\max_p\{d_{pu} - d(p,q), d(p,q) - d_G(p,u)\}$, where p is any of the previously selected nodes, d_{pu} is the distance between p and u stored in kNNG, and $d_G(p,u)$ is the distance estimation computed over the kNNG. In the formula, the term $d(p,q) - d_G(p,u)$ considers using the graph distance to upper bound the real distance between p an u, thus lower bounding the distance between q and u. On the other hand, using the real distances between neighbors stored in kNNG, we can also lower bound the distance between q and u by using the term $d_{pu} - d(p,q)$.

In the second stage, we extract objects whose distance lower bounds are large enough, that is, if $LB[u] > cr_q$ we discard u. Note that, in the beginning, it is possible that the distance lower bound is not large enough to discard the node (as the initial query covering radius is ∞). However, we reduce the query covering radius as the process goes on. So, $LB[u]$ allows us to delay the discarding of node u until cr_q is small enough, even if we only update $LB[u]$ once. This is also illustrated in Figure 4.21. Note that when we start in p the covering radius is cr_{q1}. However, once we reach p_c the covering radius has been reduced to $cr_{q2} < LB[u]$, and we can then discard u.

With these considerations we design two nearest neighbor algorithms. The first is based on algorithm kNNG**RQ1**, and we call it kNNG**NNQ1**. It selects the next node to review according to Eq. (4.1). To update the value of Eq. (4.1), in procedure **extractNode** we increase by $\mathbb{U}$ the values for non-discarded neighbors of extracted nodes, and in line 16 we account for the number of times we visit a node. Figure 4.22 depicts algorithm kNNG**NNQ1**. The second, depicted in Figure 4.23, is based on algorithm kNNG**RQ2**, and

we call it kNNG**NNQ2**. It selects nodes far apart from each other. To do so, in line 17 we add the graph distance computed in the current iteration. Once again, for the sake of CPU time, we update the selection criterion until we reach the it^*-th iteration.

---

kNNG**NNQ1** (Query q, Integer $querySize$, kNNG G, ObjectSet $\mathbb{U}$)
1. MaxHeap $nnc \leftarrow \{(\text{NULL}, \infty), \ldots, (\text{NULL}, \infty)\}$, $|nnc| = querySize$
   // elements of nnc are of the form (object, distance), sorted by decreasing distance
2. Container $(c_{id}, c_{value}) \leftarrow (\text{NULL}, -\infty)$, MinHeap $\mathcal{C} \leftarrow \emptyset$
   // elements of $\mathcal{C}$ are of the form (object, $value$), sorted by increasing $value$
3. $cr_{max} \leftarrow \max_{u \in \mathbb{U}}\{cr_u\}$, $cr_{min} \leftarrow \min_{u \in \mathbb{U}}\{cr_u\}$, $cr_{diff} \leftarrow cr_{max} - cr_{min}$
4. **For each** $u \in \mathbb{U}$ **Do** $LB[u] \leftarrow 0$, $\mathcal{C}$.**insert**$(u, |\mathbb{U}|(cr_u - cr_{min})/cr_{diff})$
5. **While** $|\mathcal{C}| > 0$ **Do**
6. $(p, value) \leftarrow$ **nextToReview**()
7. $d_{pq} \leftarrow d(p, q)$, **extractNode**$(p)$
8. **If** $d_{pq} < cr_q$ **Then** $(p,\ d_{pq}) \leftarrow$ **traverse**$(p,\ d_{pq})$
9. **useContainerNNQ**(p, d_{pq}, cr_q)
10. **For each** $(u, d_{pu}) \in NN_k(p) \cap \mathcal{C}$ **Do**
11. **If** $d_{pu} - d_{pq} > LB[u]$ **Then** $LB[u] \leftarrow d_{pu} - d_{pq}$
12. $d_G \leftarrow$ **Dijkstra**$(G, p, (1+\epsilon)d_{pq})$ // $|d_G| = |\mathbb{U}|$
13. **For each** $u \in \mathcal{C}$ **Do**
14. **If** $d_{pq} - d_G[u] > LB[u]$ **Then** $LB[u] \leftarrow d_{pq} - d_G[u]$
15. **If** $LB[u] \geq cr_q$ **Then extractNode**(u)
16. **Else If** $d_G[u] < d_{pq}$ **Then** $\mathcal{C}$.**increaseKey**$(u, 1)$

Figure 4.22: Our first nearest neighbor query algorithm (kNNG**NNQ1**). We reuse auxiliary procedures **nextToReview** and **extractNode** from kNNG**RQ1**. cr_q refers to the current query covering radius, that is, is an alias for nnc.**findMax**().$value$. $NN_k(p)$ returns the kNNG adjacency list of p (its k-nearest neighbors) in the format (u, d_{pu}), where u is the neighbor and d_{pu} is the distance between them. **Dijkstra**(G, p, x) computes distances over the kNNG G from p to all nodes up to distance x.

4.4 Experimental Results

> The only statistics you can trust are those you falsified yourself.
>
> – Winston Churchill

We have tested our construction and search algorithms on spaces of vectors, strings and documents (these last two are of interest to Information Retrieval applications [BYRN99]). The idea is to measure the behavior of our technique considering several metric spaces of different searching complexity, that is, different intrinsic dimensionality; different sizes

kNNG**NNQ2** (Query q, Integer $querySize$, kNNG G, ObjectSet $\mathbb{U}$)
1. MaxHeap $nnc \leftarrow \{(\text{NULL}, \infty), \ldots, (\text{NULL}, \infty)\}$, $|nnc| = querySize$
   // elements of nnc are of the form (object, distance), sorted by decreasing distance
2. Container $(c_{id}, c_{value}) \leftarrow (\text{NULL}, -\infty)$, Integer $it \leftarrow 0$
3. MaxHeap $\mathcal{C} \leftarrow \emptyset$, MinHeap $radii \leftarrow \emptyset$ // elements of $\mathcal{C}$ and $radii$ are of the form
   // (object, $value$), sorted by decreasing/increasing $value$, respectively
4. **For each** $u \in \mathbb{U}$ **Do** $LB[u] \leftarrow 0$, $\mathcal{C}$.**insert**$(u, 0)$, $radii$.**insert**(u, cr_u)
5. **While** $|\mathcal{C}| > 0$ **Do**
6. $(p, value) \leftarrow$ **nextToReview**()
7. $d_{pq} \leftarrow d(p, q)$, **extractNode**$(p)$, $it \leftarrow it + 1$
8. **If** $d_{pq} < cr_q$ **Then** $(p, d_{pq}) \leftarrow$ **traverse**(p, d_{pq})
9. **useContainerNNQ**(p, d_{pq}, cr_q)
10. **For each** $(u, d_{pu}) \in NN_k(p) \cap \mathcal{C}$ **Do**
11. **If** $d_{pu} - d_{pq} > LB[u]$ **Then** $LB[u] \leftarrow d_{pu} - d_{pq}$
12. **If** $(|radii| = 0)$ AND $(it > it^*)$ **Then** $r_{exp} \leftarrow (1 + \epsilon)d_{pq}$ **Else** $r_{exp} \leftarrow \infty$
13. $d_G \leftarrow$ **Dijkstra**(G, p, r_{exp}) // $|d_G| = |\mathbb{U}|$
14. **For each** $u \in \mathcal{C}$ **Do**
15. **If** $d_{pq} - d_G[u] > LB[u]$ **Then** $LB[u] \leftarrow d_{pq} - d_G[u]$
16. **If** $LB[u] \geq cr_q$ **Then extractNode**(u)
17. **Else If** $(r_{exp} = \infty)$ AND $(d_G[u] < \infty)$ **Then** C.**increaseKey**$(u, d_G[u])$

Figure 4.23: Our second nearest neighbor query algorithm (kNNG**NNQ2**). it is an auxiliary variable we use to account the iterations. Since the kNNG can have more than one connected component, we start by selecting one node from each component. As in kNNG**NNQ2**, we define heap $radii$ to order nodes by their covering radii, so that in **nextToReview** we start by choosing the node with the smallest covering radius form each component. Then, we pick the node with the largest sum of distance estimations. We reuse the auxiliary procedures **extractNode** and **nextToReview** from kNNG**RQ2**. cr_q refers to the current query covering radius, that is, it is an alias for nnc.**findMax**().$value$. $NN_k(p)$ returns the kNNG adjacency list of p (its k nearest neighbors) in the format (u, d_{pu}), where u is the neighbor and d_{pu} is the distance between them. **Dijkstra**(G, p, x) computes distances over the kNNG G from p to all nodes up to distance x.

of the query outcome, that is, range query radii to recover 1 or 10 objects or nearest neighbor queries to recover from 1 to 16 neighbors; and the size of the kNNG indexing the space considering values of k from 2 to 64 neighbors per object. We are not aware of any published kNNG practical implementation for general metric spaces.

In construction algorithms, we measure both the number of distance computations and the CPU time needed by each construction algorithm and the basic one. For shortness we have called the basic kNNG construction algorithm K**NNb**, the recursive partition based algorithm K**NNrp**, and the pivot based algorithm K**NNpiv**.

For the search algorithms, we are interested in measuring the number of distance

evaluations performed for retrieval. Each point in the search plots represents the average of 50 queries for objects randomly chosen from the metric database not included in the index. For shortness they are called the same way as in the previous section, namely, we have called the range query algorithm of Section 4.3.1.4 kNNG**RQ1** and the one of Section 4.3.1.5 kNNG**RQ2**. Likewise, their respective nearest neighbor algorithms are called kNNG**NNQ1** and kNNG**NNQ2**.

We have compared our search algorithms with AESA and a pivot-based algorithm where pivots are chosen at random. As we cannot always manage the AESA's full distance matrix index in main memory, we actually simulate AESA algorithm by computing each matrix cell on the fly when we need it. Of course, we do not count these evaluations when solving the similarity query. Since the matrix is symmetric, in the experiments we will give the size of the upper triangle distance matrix. In the case of nearest neighbor queries we use a range-optimal pivot-based algorithm. For a fair comparison, we provided the same amount of memory for the pivot index and for our kNNG index (that is, we compare a kNNG index against a $1.5k$ pivot set size).

The experiments were run on an Intel Pentium IV of 2 GHz, with 2.0 GB of RAM, with local disk, under SuSE Linux 7.3 operating system, with kernel 2.4.10-4GB i686, using `g++` compiler version 2.95.3 with optimization option `-O9`, and the processing time measured user time.

4.4.1 Uniformly distributed Vectors under Euclidean Distance

We start our experimental study with the space of vectors uniformly distributed in the unitary real D-dimensional cube under the Euclidean distance, that is, $\left([0,1]^D, L_2\right)$, for $D \in [4, 24]$. This metric space allows us to measure the effect of the space dimension D on our algorithms. We have not explored larger D values because $D = 24$ is already too high-dimensional for any known (exact) indexing algorithm. High dimensional metric spaces are best suited for non-exact approaches.

Note that we have not used the fact that this space has coordinates, but have rather treated points as abstract objects in an unknown metric space. Computing a single distance takes from 0.893 microseconds in the 4-dimensional space, to 1.479 microseconds in the 24-dimensional space.

In the construction experiments, we use uniform datasets of varying size $n \in [2{,}048; 65{,}536]$. We first compare the construction algorithms, in order to evaluate the behaviour of our heuristics. Later, search experiments are carried out over kNNGs indexing the dataset of 65,536 objects, where we select 50 random queries not included in the index. We measure range queries using search radii retrieving 1 and 10 objects on average, and also performing nearest neighbor queries retrieving from 1 to 16 relevant objects. The radii used in range queries to retrieve 1 and 10 objects are $r = 0.047$ and 0.077 for $D = 4$, $r = 0.221$ and 0.303 for $D = 8$, $r = 0.420$ and 0.525 for $D = 12$, $r = 0.630$ and 0.738 for $D = 16$, $r = 0.806$ and 0.924 for $D = 20$, and $r = 0.966$ and 1.095 for $D = 24$, respectively.

4.4.1.1 Construction

We summarize our experimental results in Figure 4.24, where we show distance computations per element and CPU time for the whole construction process, and Table 4.1 for the least square fittings computed with R [R D04]. As the dependence on k turns out to be too mild, we neglect k in the fittings, thus costs have the form cn^α. Even though in Table 4.1 we make the constant c explicit, from now on we will only refer to the exponent α. We will write $O(n^\alpha)$ as an abuse of notation, to disregard the constant c.

Space	K**NNrp**		K**NNpiv**	
	Dist. evals	CPU time	Dist. evals	CPU time
$[0,1]^4$	$10.0n^{1.32}$	$0.311n^{2.24}$	$56.1n^{1.09}$	$0.787n^{2.01}$
$[0,1]^8$	$32.8n^{1.38}$	$0.642n^{2.11}$	$168n^{1.06}$	$15.5n^{1.69}$
$[0,1]^{12}$	$15.1n^{1.59}$	$1.71n^{2.03}$	$116n^{1.27}$	$20.1n^{1.79}$
$[0,1]^{16}$	$5.06n^{1.77}$	$0.732n^{2.14}$	$12.1n^{1.64}$	$6.87n^{1.97}$
$[0,1]^{20}$	$2.32n^{1.88}$	$0.546n^{2.18}$	$2.48n^{1.87}$	$2.77n^{2.10}$
$[0,1]^{24}$	$1.34n^{1.96}$	$0.656n^{2.16}$	$1.23n^{1.96}$	$1.29n^{2.16}$
$[0,1]^D$	$0.455e^{0.19D}n^{1.65}$	$0.571e^{0.01D}n^{2.14}$	$0.685e^{0.24D}n^{1.48}$	$0.858e^{0.11D}n^{1.95}$

Table 4.1: K**NNrp** and K**NNpiv** least square fittings for distance evaluations and CPU time for vector metric spaces. e^x refers to the exponential function. CPU time measured in microseconds.

Table 4.1 shows that both of our construction algorithms are subquadratic in distance computations, and slightly superquadratic in CPU time, when considering each dimension independently. It also shows that the best of our construction algorithms is K**NNpiv**, yet it is also more sensitive to the space dimension. The rows for dimensions $D \leq 12$, show that K**NNpiv** is slightly superlinear in distance evaluations, and later it becomes subquadratic. An interesting fact is that for medium dimensions (that is, D in [8,16]) K**NNpiv** is also subquadratic in CPU time. To understand this, we have to take into account that the distance distribution concentrates as the dimension increases [CNBYM01]. The direct consequence is that the difference between the minimum and maximum distances decreases as the dimension grows. Likewise, this also implies that the shortest path propagations stop earlier and earlier as the dimension grows, as it is required to add fewer edge weights in the path so as to reach the propagation limit. This translates into the fact that method **extractFrom** performs less work as the dimension grows. If we consider both the superlinear number of distance computations and the reduced work performed by method **extractFrom** for values of D in [8,16] we obtain that the CPU time turns out to be subquadratic.

The last line of Table 4.1 shows the usual exponential dependence on the space dimension for both of our construction algorithms, a phenomenon known as the *curse of dimensionality*, see Section 2.3.3 in page 31. Note also that this line clearly reveals the subquadratic performance on distance evaluations for both of our construction algorithms when considering the exponential dependence on D. We remark that, in the metric

space context, superquadratic CPU time in side computations is not as important as a subquadratic number of computed distances. This can be appreciated in the experimental results on the document space, shown in Section 4.4.4.

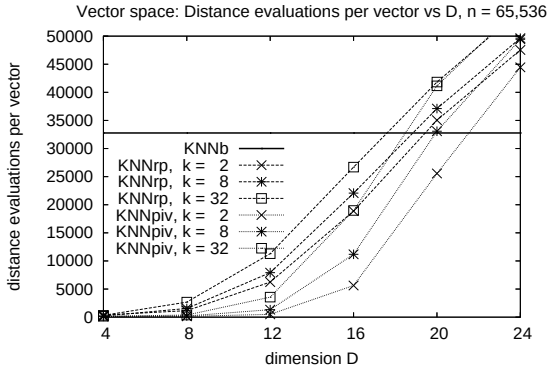
Figures 4.24(a) and (b) are in strong agreement with the last line of Table 4.1. In fact, Figure 4.24(a) shows that, as D grows, the distance evaluation performance of our algorithms degrade. For instance, for $D = 4$, K**NNpiv** uses $O(n^{1.10})$ distance evaluations, but for $D = 24$, it is $O(n^{1.96})$ distance evaluations. Notice that a metric space with dimensionality $D > 20$ is considered as intractable [CNBYM01]. On the other hand, Figure 4.24(b) shows that K**NNrp** is less sensitive to the dimension than K**NNpiv**. Finally, both figures confirm that the latter has better performance for small values of k.

With respect to CPU time, K**NNb** is usually faster than our algorithms, yet Figure 4.24(b) shows that K**NNpiv** beats K**NNb** for values of $D \leq 8$. This is remarkable in this vector space as the Euclidean distance is very cheap to compute, thus a significant fraction of the CPU time comes from shortest path computations. Note that K**NNb**'s CPU time increases with D, but the increase is very mild. On the other hand, the plot also shows that K**NNpr** is more resistant to the curse of dimensionality than K**NNpiv**. Finally, for $n = 65{,}536$, the plot shows that the construction time is better for $D = 8$ than for $D = 4$. This is because as the dimension grows shortest path computations perform less work.

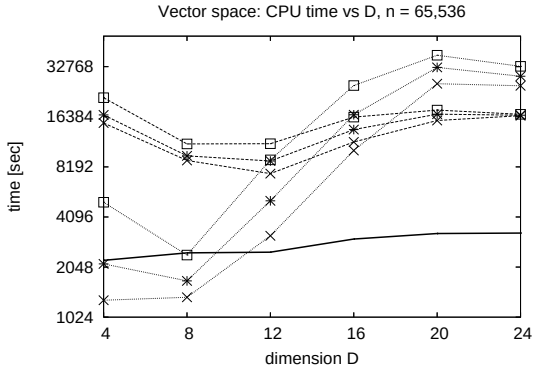
Figure 4.24(c) shows that our algorithms are subquadratic in distance evaluations, instead of K**NNb** that it is always $O(n^2)$ whichever the dimension is. The plot shows results for $D \leq 16$. Subquadraticity can also be verified for higher dimensions, however it is not so visually evident from the plots. For values of $D \leq 16$, our construction algorithms have better performance in distance computations than K**NNb**, being K**NNpiv** the best, which confirms the results obtained from Table 4.1. Moreover, for lower dimensions ($D \leq 8$) ours are only slightly superlinear.

Figure 4.24(d) is in strong agreement with results in plot (b). For instance, it shows that K**NNrp** is more resistant to the dimensionality effect than K**NNpiv**, as the four K**NNrp** curves are closer than the respective ones for K**NNpiv**. The plot also shows that the best CPU time is obtained for $D = 8$.

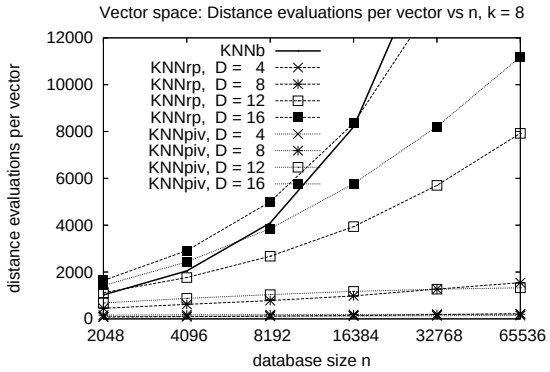
Figures 4.24(e) and (f) show a sublinear dependence on k for all dimensions both in distance computations and CPU time, however, K**NNpiv** is more sensitive to k than K**NNrp**. Also, the dependence on k diminishes as D grows, although it always increases monotonically on k. In terms of distance computations, Figure 4.24(e) shows that our algorithms behave better than K**NNb** for $D \leq 16$. This is also verified in higher dimensional spaces (K**NNpiv** in $D = 20$) for values of $k \leq 8$, but we omit this curve in the plots to improve their readability. Finally, in terms of CPU time, Figure 4.24(f) shows that K**NNpiv** is faster than K**NNb** for $D \leq 8$ and small values of k.



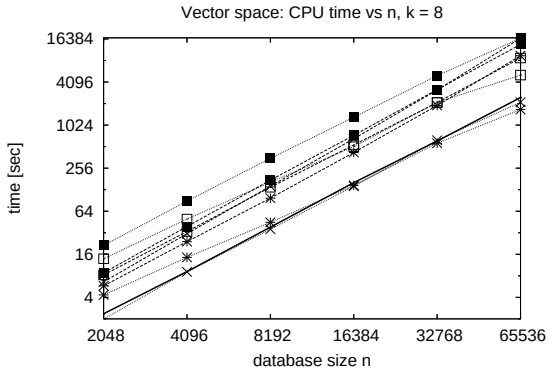
(a) Dist. evals. per vector, dependence on D.



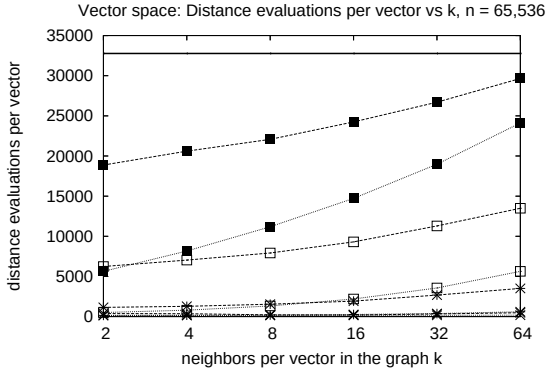
(b) CPU time, dependence on D.



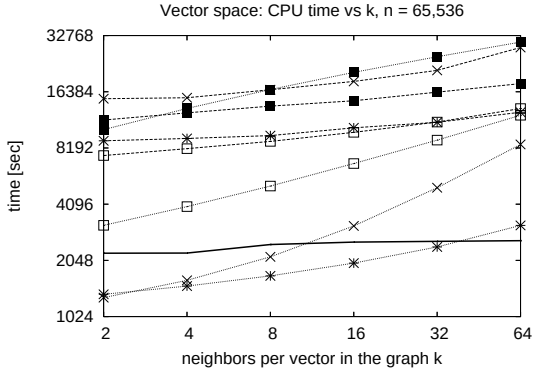
(c) Dist. evals. per vector, dependence on n.



(d) CPU time, dependence on n.



(e) Dist. evals. per vector, dependence on k.

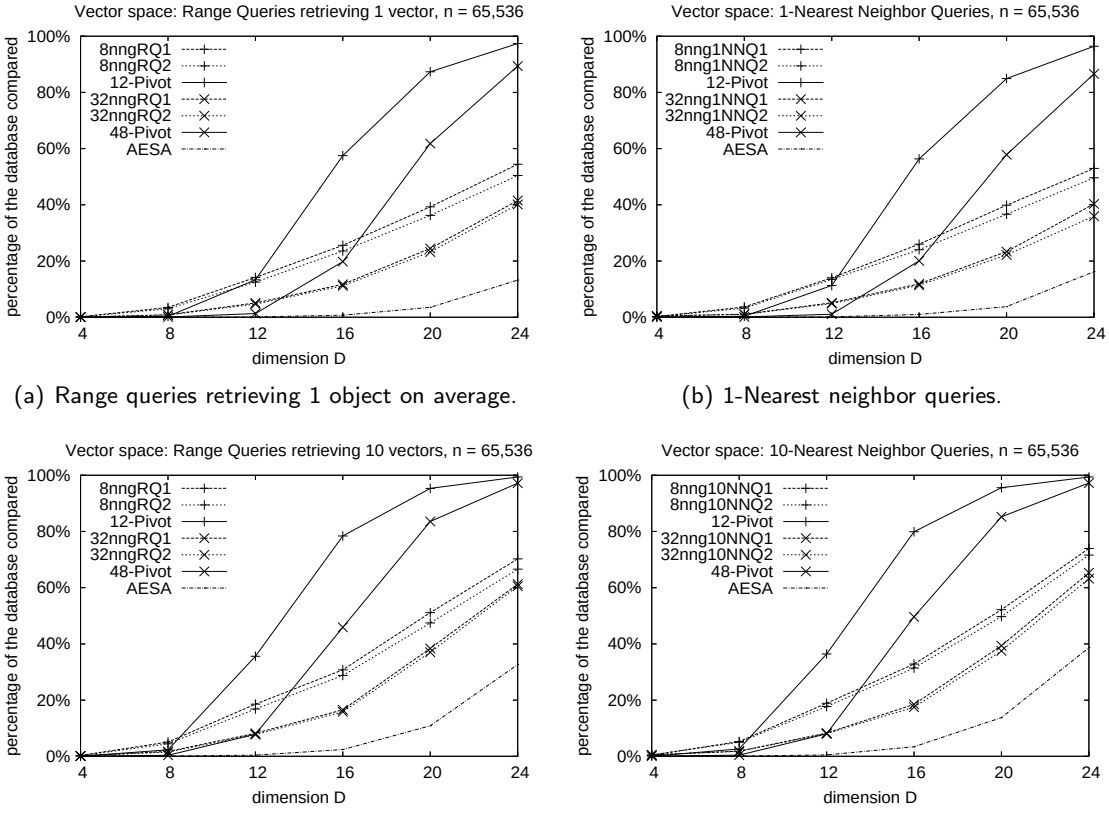


(f) CPU time, dependence on k.

Figure 4.24: Evaluating kNNG construction algorithms in vector spaces. On the left, distance evaluations per element during kNNG construction. On the right, CPU time for the whole process. Note the logscale. Figure (b) follows the legend of Figure (a). Figures (d), (e) and (f) follow the legend of Figure (c). In Figure (a), K**NNrp** and K**NNpiv** (both for $k = 32$) reach 54,569 and 55,292 distance evaluations per node, respectively. In Figure (c), K**NNb** and K**NNrp** (for $D = 16$) reach 32,768 and 22,071 distance evaluations per node, respectively.

4.4.1.2 Searching

Figures 4.25 and 4.26 show results in the vector space. We explore several parameters such as: different values of D, different size of the query outcome (by using two radii or varying the number of closest objects to recover), and different index size (that is, varying the number of neighbor per object in the kNNG).



(a) Range queries retrieving 1 object on average.

(b) 1-Nearest neighbor queries.

(c) Range queries retrieving 10 objects on average.

(d) 10-Nearest neighbor queries.

Figure 4.25: Evaluating percentage of database compared by our kNNG based search algorithms in vector spaces, dependence on dimension D. On the left, range queries. On the right, nearest neighbor queries. We compare the search performance of our algorithms with a pivoting algorithm using $1.5k$ pivots.

Figure 4.25(a) shows range queries using radii that retrieve 1 object per query on average, indexing the space with 8NNG and 32NNG graphs, for varying dimension. On the other hand, Figure 4.25(b) shows the equivalent experiment for nearest neighbor queries retrieving the closest neighbor. As can be seen from these plots, even though our nearest neighbor query algorithms are not range-optimal *per se*, they behave as if they were. Also, both plots show that the classic pivot algorithm has better performance for dimension $D \leq 12$, but later its performance degrades painfully. On the other hand, our kNNG based approach also degrades with the dimension, as expected, but the degradation is smoother,

being ours a competitive alternative for $D > 12$. Finally, the performance of AESA is unbeatable —in fact, for $D = 16$, 20 and 24 AESA is 15, 6.5 and 3 times faster that ours—, however, it needs 8 GB of main memory for the $O(n^2)$ index (the upper triangle distance matrix), whilst the 32NNG only uses 12 MB of main memory, scarcely 0.146% of the AESA space requirement.

Figures 4.25(c) and (d) show the same experiments as above for less discriminative queries, that is, range queries using radii that retrieve 10 objects per query on average and 10-nearest neighbor queries. Again, the behavior of our $NN_k(q)$ algorithms looks like range-optimal, and the classic pivot algorithm has better performance for low dimensions. However, this time our algorithms become competitive with the pivoting algorithm earlier with respect to the dimension. This is because the performance degradation of the pivot alternative is even sharper when we consider queries retrieving more elements, whilst the degradation of our algorithms is smoother, even smoother than the degradation of AESA.

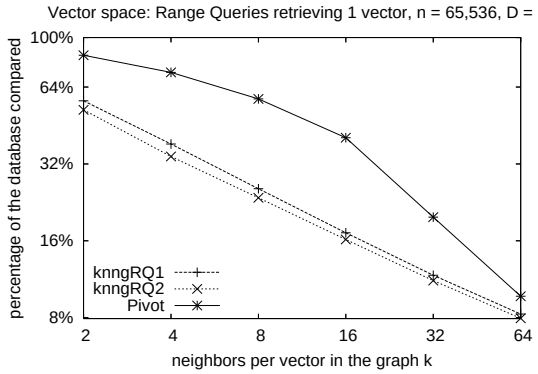
Figures 4.26(a) and (b) show range queries retrieving respectively 1 and 10 vectors on average per query versus the index size (that is, the number of neighbors k per vector in the kNNG index), for dimension $D = 16$. As it is expected, the bigger the size of the kNNG index, the better the searching performance of our technique. The same behavior can be observed for the pivot technique, however its performance is worse (that is, it computes more distances) than ours for the range of values of k under consideration. As we work with small values of k, this can be interpreted as that our technique behaves better than pivots in low-memory scenarios. Again, AESA performance is better by far than ours, however our technique only uses from 0.009% (for a 2NNG) to 0.293% (for a 64NNG) of AESA's space requirement.

Figures 4.26(c) and (d) show nearest neighbor queries over a 32NNG in dimension $D = 16$ and 24 respectively, varying the size of the query outcome. In accordance to the previous plots, these show that, as the query outcome size grows, it is more difficult to solve the proximity query, but the performance degradation of our technique is reasonably mild.

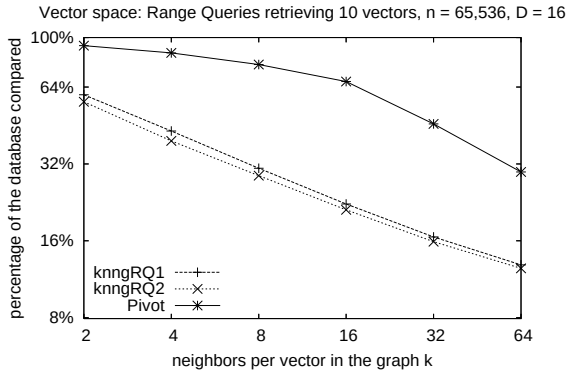
From this experimental series we remark that kNNG based algorithms are more resistant to both the dimension effect (Figure 4.25) and the query outcome size (Figure 4.26) than the classic pivot alternative. Furthermore, all the plots in Figures 4.25 and 4.26 show that our search algorithms have better performance than the classic pivot based approach for dimension $D > 12$. Finally, we verify that both of our kNNG based approaches perform rather similarly, being kNNG**RQ2**, and its induced nearest neighbor query algorithm, slightly better than kNNG**RQ1**.

4.4.2 Gaussian-distributed Vectors under Euclidean Distance

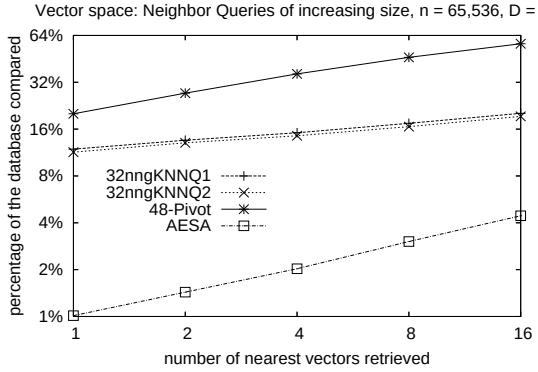
Real-life metric spaces have regions called *clusters*, that is, compact zones of the space where similar objects accumulate. With the Gaussian vector space we attempt to simulate a real-world space. The dataset is formed by points in a 20-dimensional space under the



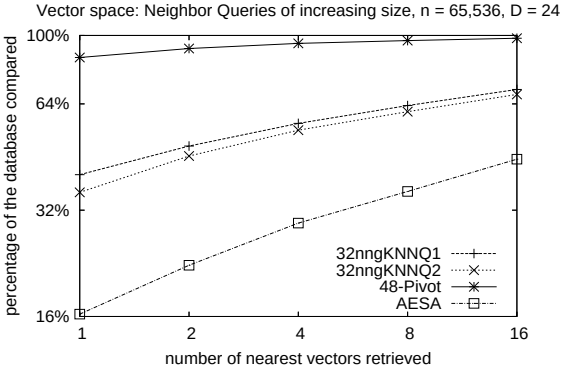
(a) Range queries retrieving 1 object on average, varying index size.



(b) Range queries retrieving 10 objects on average, varying index size.



(c) $NN_k(q)$ in dim 16, varying the query size outcome.



(d) $NN_k(q)$ in dim 24, varying the query size outcome.

Figure 4.26: Evaluating percentage of database compared by our kNNG based search algorithms in vector spaces. On the first row, range queries varying the index size. On the second row, nearest neighbor queries varying the query outcome size. Note the logscale. We compare the search performance of our algorithms with a pivoting algorithm using $1.5k$ pivots. In (a) and (b), AESA performs 487 and 1,590 distance comparisons on average, that is, it compares around 0.74% and 2.42% of the database, respectively.

Euclidean distance with Gaussian distribution forming 256 clusters randomly centered in $[0,1]^{20}$. The generator of Gaussian vectors was obtained from [GBC+99]. We consider three different standard deviations to make more crisp or more fuzzy clusters ($\sigma = 0.1$, 0.2 and 0.3). Of course, we have not used the fact that the space has coordinates, rather we have treated the points as abstract objects in an unknown metric space.

Computing a single distance takes 1.281 microseconds in our machine. Note that a 20-dimensional space already has a high representational dimensionality. However, as long as the standard deviation decreases, the intrinsic dimensionality of the space also diminishes.

In the construction experiments, we use Gaussian datasets of varying size $n \in$

[2,048; 65,536]. Later, search experiments are carried out over kNNGs indexing Gaussian datasets formed by 65,536 objects. We select 50 random queries not included in the index, using search radii that on average retrieve 1 and 10 objects ($r = 0.385$ and 0.458 for $\sigma = 0.1$, $r = 0.770$ and 0.914 for $\sigma = 0.2$ and $r = 1.0865$ and 1.256 for $\sigma = 0.3$, respectively), and also performing nearest neighbor queries retrieving from 1 to 16 relevant objects.

4.4.2.1 Construction

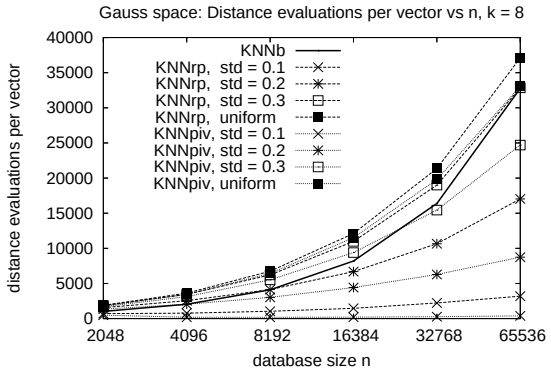
We summarize our experimental results in Figure 4.27, where we show distance computations per element and CPU time for the whole construction process, and in Table 4.2 for the least square fittings computed with R [R D04]. Once again, we neglect k from the fittings, as its influence turns out to be very mild.

Space	K**NNrp**		K**NNpiv**	
	Dist. evals	CPU time	Dist. evals	CPU time
Gaussian $\sigma = 0.1$	$74.7n^{1.33}$	$1.13n^{2.07}$	$1260n^{0.91}$	$63.5n^{1.63}$
Gaussian $\sigma = 0.2$	$7.82n^{1.71}$	$1.13n^{2.09}$	$16.3n^{1.60}$	$8.70n^{1.94}$
Gaussian $\sigma = 0.3$	$2.97n^{1.85}$	$0.620n^{2.17}$	$3.86n^{1.81}$	$3.78n^{2.06}$

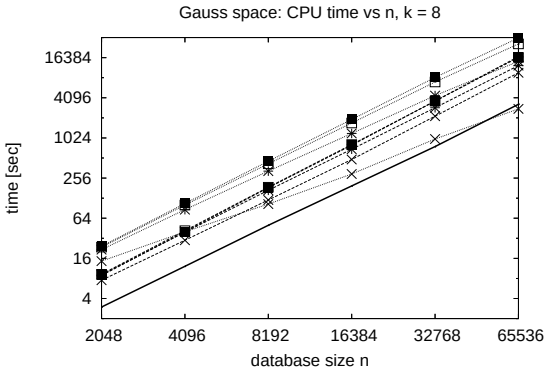
Table 4.2: K**NNrp** and K**NNpiv** least square fittings for distance evaluations and CPU time for 20-dimensional Gaussian metric spaces. CPU time measured in microseconds.

Table 4.2 shows that both of our construction algorithms are subquadratic in distance computations, and slightly superquadratic in CPU time, when considering each standard deviation σ separately. It also shows that as σ increases, the exponents grow. This is expected since the larger the deviation, the more overlapped the clusters. Indeed, the rows for $\sigma = 0.1$, 0.2 and 0.3 exhibit exponent values similar to the ones for 8, 16 and 20 dimensions in the uniformly distributed vector space in Table 4.1, respectively. Thus, the intrinsic dimensionality increases with σ, as the space becomes more uniform.

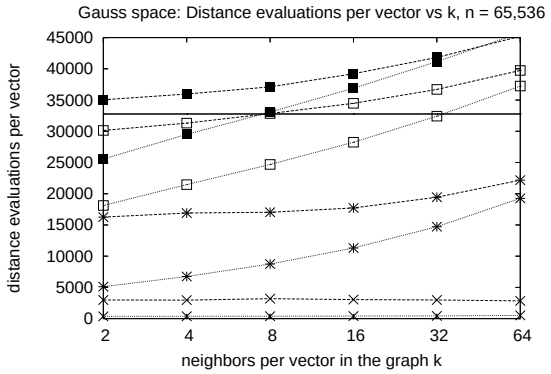
Once again, the table shows that the best of our construction algorithms is K**NNpiv**, yet it is also more sensitive to the intrinsic dimension of the space. K**NNrp** is subquadratic in distance evaluations and slightly superquadratic in CPU time for the three deviations. K**NNpiv** is sublinear for $\sigma = 0.1$ and subquadratic for $\sigma = 0.2$ and 0.3 with respect to distance computations. This particular behavior is explained because for $\sigma = 0.1$ the clusters are so crisp (that is, they have small radius when compared with the average distance between any two objects in the space) that the pivoting preindex detects the appropriate cluster and filters out most of the objects when computing the nearest neighbors for each object in the kNNG. However, as the deviation enlarges the clusters overlap more and more each other, reducing the filtering power of the pivoting preindex. A similar effect can be seen from the K**NNpiv** CPU fittings, where we verify that it is subquadratic for $\sigma = 0.1$ and 0.2, and slightly superquadratic for $\sigma = 0.3$. This comes both from the cluster crispness and from the fact that, since the dimensionality is high, the shortest path computations performed by **extractFrom** stop after few propagations.



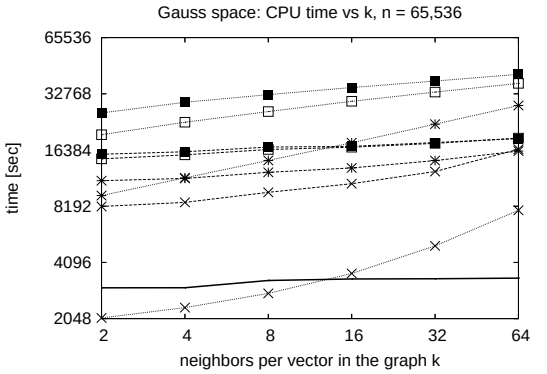
(a) Dist. evals. per vector, dependence on n.



(b) CPU time, dependence on n.



(c) Dist. evals. per vector, dependence on k.



(d) Dist. evals. per vector, dependence on k.

Figure 4.27: Evaluating kNNG construction algorithms in 20-dimensional Gaussian spaces. On the left, distance evaluations per element during kNNG construction. On the right, CPU time for the whole process. Note the logscale. Figures (b), (c) and (d) follow the legend of Figure (a). In Figure (c) K**NNrp** and K**NNpiv** reach 45,210 and 45,749 distance evaluations per element where vectors are uniformly distributed, respectively.

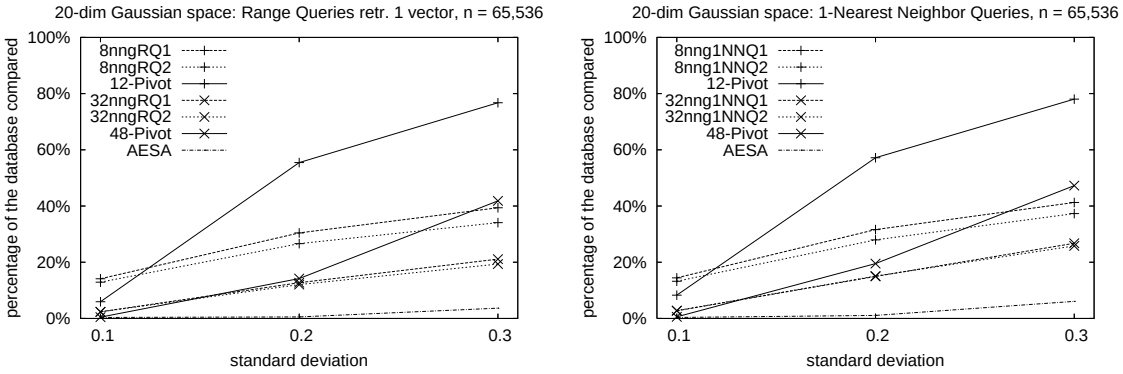
Figure 4.27(a) shows the subquadratic performance in distance evaluations of our approach, and Figure (b) the superquadratic CPU time. Figures 4.27(c) and (d) show that the dependence on k is mild. It is interesting that for crisp clusters ($\sigma = 0.1$) the distance computation performance of our algorithms improves significantly, even for high values of k. Also, in crisp clusters, K**NNpiv** is faster in CPU time than K**NNb** for $k < 16$. We also verify improvements in distance computations for $\sigma = 0.2$. For $\sigma = 0.3$ our construction algorithms behave better that K**NNb** for small values of k. Note that for $k \leq 8$ our algorithms are more efficient in distance computations than K**NNb** for the three variances. Again, KNNpiv has the best performance.

Finally, in the plots of Figure 4.27, we also draw the construction results for the 20-dimensional uniformly distributed vector space. As can be seen, there is a performance improvement even with fuzzy clusters. Once again, we insist that in the metric space context, superquadratic CPU time in side computations is tolerable when this allows

reducing the number of distance computations.

4.4.2.2 Searching

Figures 4.28 and 4.29 show searching results in the 20-dimensional Gaussian space, where we explore the effect of the cluster sizes, the query outcome size and the index size.



(a) Range queries retrieving 1 object on average.

(b) 1-Nearest neighbor queries.

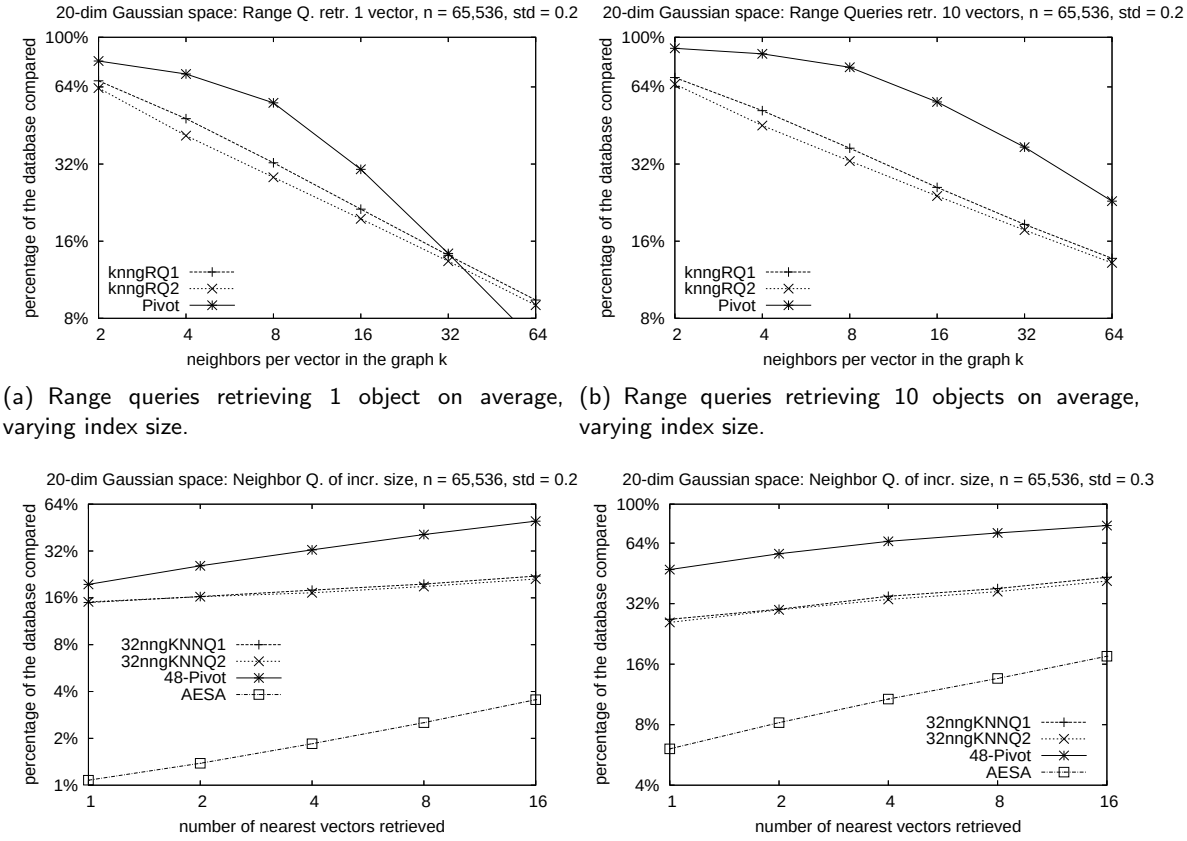
(c) Range queries retrieving 10 objects on average.

(d) 10-Nearest neighbor queries.

Figure 4.28: Evaluating percentage of database compared by our kNNG based search algorithms in 20-dimensional Gaussian spaces, dependence on the standard deviation. On the left, range queries. On the right, nearest neighbor queries. We compare the search performance of our algorithms with a pivoting algorithm using $1.5k$ pivots.

In Figure 4.28 we perform range queries recovering 1 and 10 elements, and compare these results with the ones obtained when performing the equivalent query for nearest neighbors. This figure also confirms that the kNNG based searching algorithms perform in a range-optimal fashion. Furthermore, the plots show that the pivot algorithm behaves better when the Gaussian space is composed by crisp clusters, but as the clusters get fuzzier our techniques behave better in relative terms, as their degradation is smoother than in the case of the pivoting algorithm. Note that this agrees with the previous section, where

we suggest that a 20-dimensional Gaussian space with crisp clusters ($\sigma = 0.1$) looks like a 8-dimensional uniformly distributed space. We also observe that the performance is better when using the 32NNG graph as the metric index instead of using a 8NNG, as expected. Finally, the figure shows that, unlike the pivoting algorithms, ours degrade gradually as the query loses discrimination power.



(a) Range queries retrieving 1 object on average, varying index size.

(b) Range queries retrieving 10 objects on average, varying index size.

(c) $NN_k(q)$ in $\sigma = 0.2$, varying the query size outcome.

(d) $NN_k(q)$ in $\sigma = 0.2$, varying the query size outcome.

Figure 4.29: Evaluating percentage of database compared by our kNNG based search algorithms in Gaussian spaces. On the first row, range queries varying the index size. On the second row, nearest neighbor queries varying the query outcome size. Note the logscale. We compare the search performance of our algorithms with a pivoting algorithm using $1.5k$ pivots. In (a) and (b), AESA performs 359 and 903 distance comparisons on average, that is, it compares around 0.55% and 1.38% of the database, respectively. In (a), for 96 pivots (equivalent to a 64NNG), the pivoting algorithm requires 4247 distance comparisons, around 6.48%.

Figures 4.29(a) and (b) show range queries retrieving respectively 1 and 10 vectors on average per query versus the index size, for $\sigma = 0.2$. As expected, the larger the kNNG index size, the better the searching performance of our technique. The same behavior can be observed for the pivot technique, yet its performance is worse than ours for all the range of k values, with two exceptions in range queries retrieving 1 object and no exceptions when retrieving 10 objects. In fact, over a 32NNG our algorithms behave rather similarly

to the equivalent 48-pivot index, and the 96-pivot index behaves better that ours over the equivalent 64NNG index. Once again, this can be interpreted as that our technique behaves better than pivots in low-memory scenarios. On the other hand, AESA performance is better by far than ours yet it uses $O(n^2)$ space for the index (in these experiments, AESA requires 8 GB of main memory).

Figures 4.29(c) and (d) show nearest neighbor queries over a 32NNG for $\sigma = 0.2$ and 0.3 respectively, versus the size of the query outcome. In accordance to the previous plots, these show that as the query outcome size grows, it becomes more difficult to solve the proximity query, but the performance degradation of our technique is reasonably mild.

As can be seen, the conclusions of this experimental series are similar to that obtained in Section 4.4.1.2, that is, kNNG based algorithms are resistant to both the intrinsic dimensionality effect (Figure 4.28) and the query outcome size (Figures 4.29); and both of our kNNG based approach perform rather similarly, being kNNG**RQ2** and kNNG**NNQ2** slightly better than kNNG**RQ1** and kNNG**NNQ1**.

4.4.3 Strings under Edit Distance

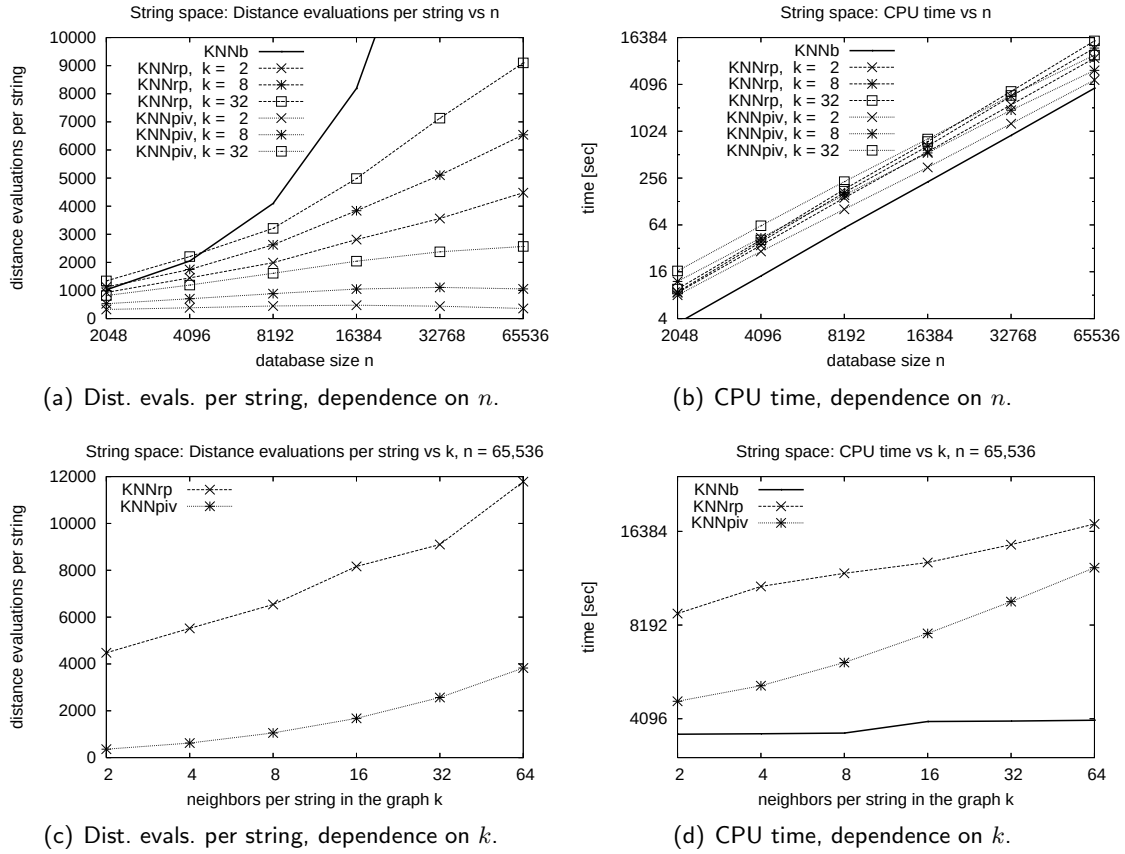
The string metric space under the edit distance has no coordinates. The edit distance is a discrete function that, given two strings, measures the minimum number of character insertions, deletions and substitutions needed to transform one string to the other [NR02]. Our database is an English dictionary, where we index a subset of $n = 65{,}536$ randomly chosen words. On average, a distance computation takes 1.632 microseconds.

In the search experiments, we select 50 queries at random from dictionary words not included in the index. We search with radii $r = 1$, 2 and 3, which return on average 0.003%, 0.044% and 0.373% of the database, that is, approximately 2, 29 and 244 words of the English dictionary, respectively.

4.4.3.1 Construction

Figure 4.30 shows results for strings. As can be seen, both K**NNrp** and K**NNpiv** require a subquadratic amount of distance computations for all $k \in [2, 64]$. This is also verified when checking the least square fitting for distance evaluations, as they are $21.4n^{1.54}$ and $99.9n^{1.26}$ for K**NNrp** and K**NNpiv**, respectively. Note that these exponents are similar to the ones for 12 dimensions in the uniformly distributed vector space in Table 4.1. In order to illustrate the improvements in distance computations of our algorithms, we can show that for $n = 65,536$, K**NNrp** costs 28%, and K**NNpiv** just 8%, of K**NNb** to build the 32NNG.

With respect to CPU time, the recursive algorithm is slightly superquadratic ($1.09n^{2.09}$ microseconds) and the pivoting one is slightly subquadratic ($10.8n^{1.85}$



(a) Dist. evals. per string, dependence on n.

(b) CPU time, dependence on n.

(c) Dist. evals. per string, dependence on k.

(d) CPU time, dependence on k.

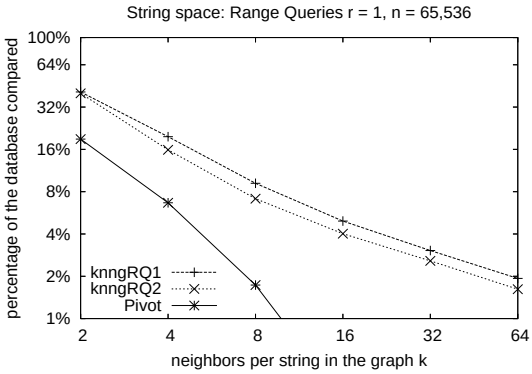
Figure 4.30: Evaluating kNNG construction algorithms in the string space. On the left, distance evaluations per element during kNNG construction. On the right, CPU time for the whole process. Note the logscale. In Figure (a), and (c), K**NNb** reaches 32,768 distance evaluations per node. In Figure (b), K**NNb** reaches 3.5 seconds for n = 2,048.

microseconds). This also can be seen from Figure 4.30(b), as the slope is flatter than the slope of K**NNb**.

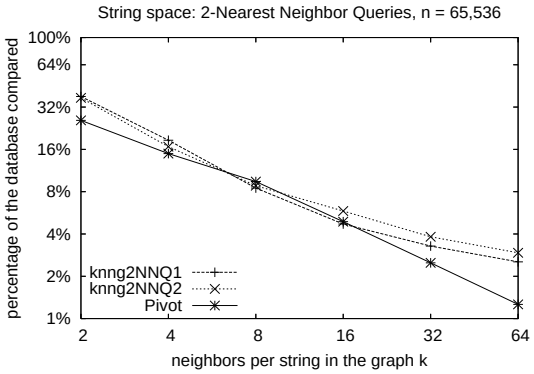
Once again, Figures 4.30(c) and (d) show that the dependence on k is mild for both of our algorithms.

4.4.3.2 Searching

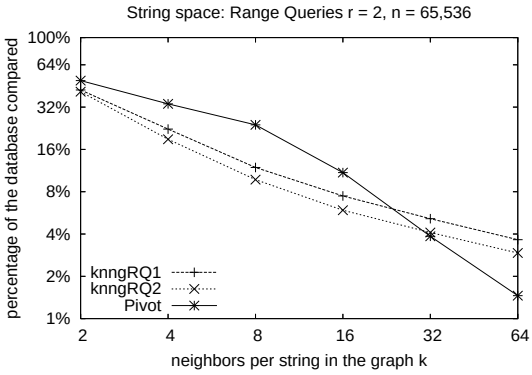
Figure 4.31 shows searching results in the string space. Figure 4.31(a) shows range queries using radius $r = 1$, which retrieve approximately 2 objects per query on average, and Figure (b) shows the equivalent 2-nearest neighbor query experiment. These plots also show that our approach behaves in a range-optimal fashion, which is in agreement with the results shown in synthetic vector spaces.



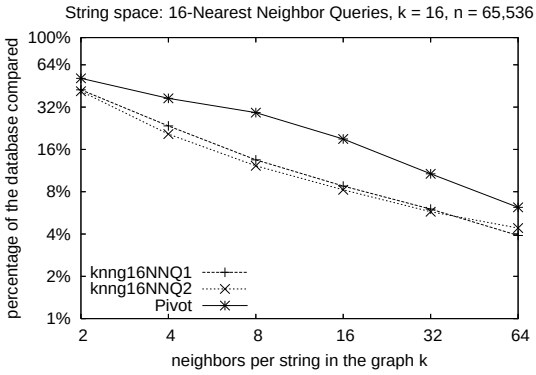
(a) Range queries, radius $r = 1$, varying index size.



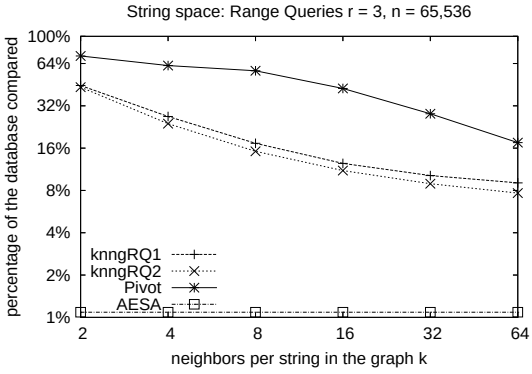
(b) 2-Nearest neighbor queries, varying index size.



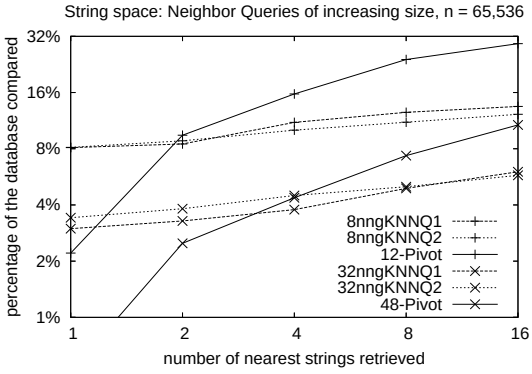
(c) Range queries, radius $r = 2$, varying index size.



(d) 16-Nearest neighbor queries, varying index size.



(e) Range queries, radius $r = 3$, varying index size.



(f) $NN_k(q)$ varying the query size outcome.

Figure 4.31: Evaluating percentage of database compared by our kNNG based search algorithms in the string space. On the left, range queries. On the right, nearest neighbor queries. We compare the search performance of our algorithms with a pivoting algorithm using $1.5k$ pivots. In (a), (b), (c) and (d), AESA performs 25, 106, 42 and 147 distance comparisons on average, that is, it compares around 0.04%, 0.16%, 0.06% and 0.22% of the database, respectively. In (f), AESA perform from 32, 41, 79, 100 and 146 distance evaluations, that is, 0.05%, 0.06%, 0.12%, 0.15% and 0.22% to retrieve 1, 2, 4, 8 and 16 relevant objects, respectively. In (a) and (f), for 96 pivots (equivalent to a 64NNG), the pivoting algorithm requires 74 and 334 distance comparisons, around 0.11% and 0.51% of the metric database, respectively.

On the other hand, all the plots in Figure 4.31 confirm that the kNNG based search algorithms are resistant to the query outcome size, as expected from the previous searching experiments in synthetic spaces. Indeed, with radii $r = 1$, 2 and 3, we retrieve approximately 2, 29 and 244 strings per query on average, yet the performance of our algorithms does not degrade as sharply as the pivot-based one. With radius 1 the pivot based technique has better performance than our algorithms. However, with radius $r = 2$ and 3 our algorithms outperform the pivot-based algorithm, which increases the number of distance computations fast as the search radius grows. The same behavior can be observed for nearest neighbor queries; our algorithms beat the pivot-based one when retrieving 4 or more nearest neighbors. AESA uses very few distances evaluations, however its index uses $O(n^2)$ memory, which is impractical in most of the scenarios (in fact, in this one AESA uses 8 GB of space).

Note that the difference between pivot range queries of radius 1 and the 2-nearest neighbor queries arises because there are strings that have much more than 2 neighbors at distance 1, for example the query word "cams" retrieves "jams", "crams", "cam" and seventeen others, so these words distort the average for radius 1. We also verify that, the bigger the index size, the better the performance.

Once again, our kNNG based approach perform rather similarly, and kNNG**RQ2** is better than kNNG**RQ1**. However, this time kNNG**NNQ1** turns out to be better than kNNG**RQ2** when the kNNG uses more memory.

4.4.4 Documents under Cosine Distance

In Information Retrieval, documents are usually represented as unitary vectors of high dimensionality [BYRN99]. The idea consists in mapping the document to a vector of real values, so that each dimension is a vocabulary word and the relevance of the word to the document (computed using some formula) is the coordinate of the document along that dimension.

With this model, document similarity is assessed through the inner product between the vectors. Note that the inner product between two exactly equal documents is one, since both documents are represented by the same unitary vector. As the documents are more different, the inner product between the vectors representing them goes to zero. As we are looking for a distance, we consider the angle between these vectors. This way, the cosine distance is simply the arc cosine of the inner product between the vectors [BYRN99], and this satisfies the triangle inequality. Similarity under this model is very expensive to calculate.

We use a dataset composed by 20,913 medium size documents, each of them of 200 KB approximately, obtained by splitting documents from the whole TREC-3 collection [Har95]. Given the documents, we synthesize the vectors representing them by using the program machinery provided in the *Metric Spaces Library* (`http://sisap.org/?f=library`) [FNC07]. The resulting medium size document space has high intrinsic dimensionality.

In this dataset we verify a good performance of both kNNG construction and kNNG-based searching algorithms. We have left this dataset available in the Library (`http://www.sisap.org/library/dbs/documents/`).

In the construction experiments, we use document datasets of varying size $n \in [2{,}000; 20{,}000]$. Later, search experiments are carried out over kNNGs indexing a dataset formed by 20,000 documents. We select 50 random queries not included in the index, using search radii that on average retrieve 1 and 10 documents ($r = 0.2842$ and 0.3319, respectively), and also performing nearest neighbor queries retrieving from 1 to 16 relevant documents.

In the construction experiments, we maintain the whole document set in main memory in order to avoid the disk time. In this scenario, a distance computation takes 182 microseconds on average. In real applications, however, this may be unfeasible, since the dataset can be arbitrarily large. So, in search experiments we manage the documents on disk to demonstrate the effect of such an expensive distance function. That is, we show a case where the distance is expensive enough to absorb the extra CPU time we incur with the graph traversals.

4.4.4.1 Construction

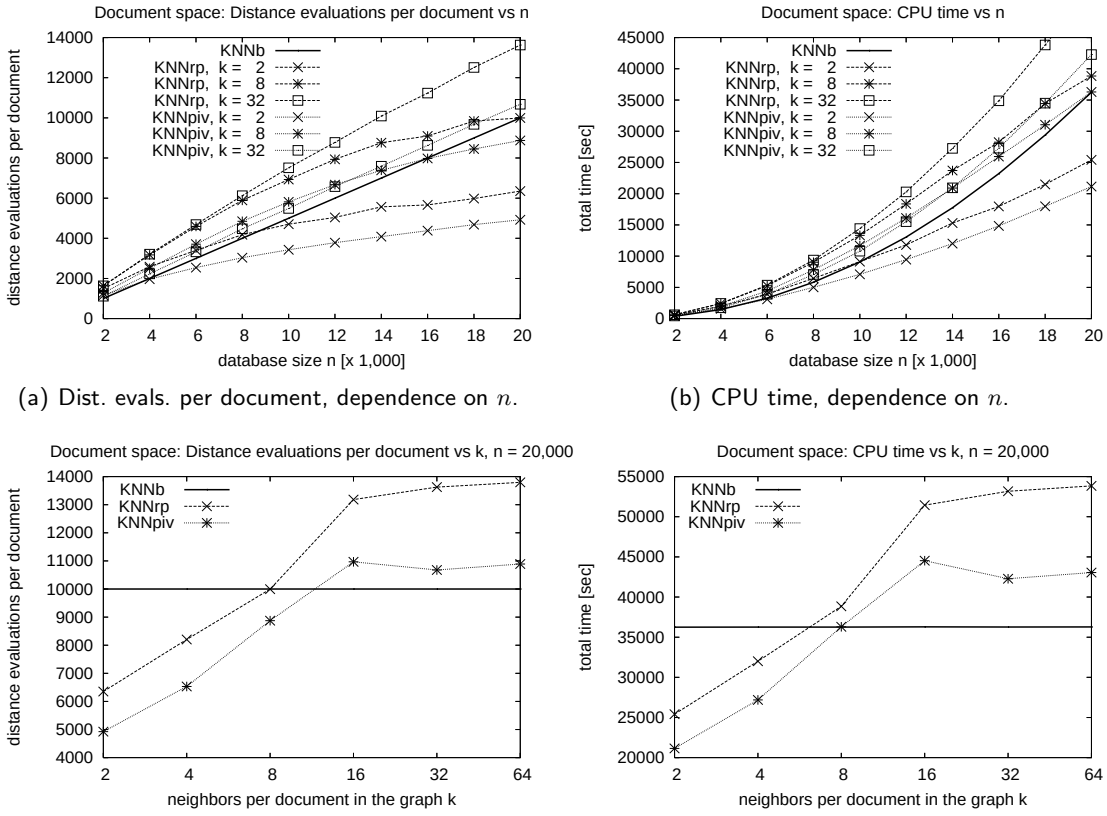
Figure 4.32 shows that our methodology requires a subquadratic amount of distance computations for small values of k ($\leq$ 8). We compute the least square fittings for distance evaluations obtaining $3.47n^{1.81}$ evaluations for K**NNrp** and $2.02n^{1.85}$ evaluations for K**NNpiv**. Note that this values are similar to the ones for 20 dimensions in the uniformly distributed vector spaces in Table 4.1. As can be seen, for $n = 20{,}000$ documents, K**NNrp** costs 64% and K**NNpiv** costs 49% of K**NNb** to build the 2NNG.

With respect to CPU time the least square fittings are $572n^{1.83}$ microseconds for K**NNrp** and $334n^{1.87}$ microseconds for K**NNpiv**. The exponents of these fittings differ marginally from the ones for distance evaluations. This shows that in practice the leading complexity (computing distances) is several orders of magnitude larger than other side computations such as traversing pointers, scanning the pivot table or computing distance through the graph.

Finally, Figures 4.32(c) and (d) confirm that the dependence on k is mild for both of our algorithms.

4.4.4.2 Searching

Figures 4.33 and 4.34 show searching results in the space of documents for distance evaluations and elapsed time, respectively. Figures 4.33(a) and (b) show the performance of range queries retrieving 1 and 10 documents varying the index size. Note that kNNG**RQ2** requires approximately a 29% of the distance evaluations of the equivalent pivot based



(a) Dist. evals. per document, dependence on n.

(b) CPU time, dependence on n.

(c) Dist. evals. per document, dependence on k.

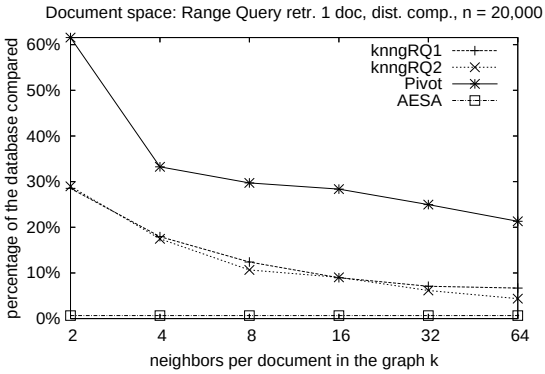
(d) CPU time, dependence on k.

Figure 4.32: Evaluating kNNG construction algorithms in the space of documents. On the left, distance evaluations per element during kNNG construction. On the right, CPU time for the whole process. Note the logscale. In Figure (b), K**NNrp** reaches 53,000 seconds.

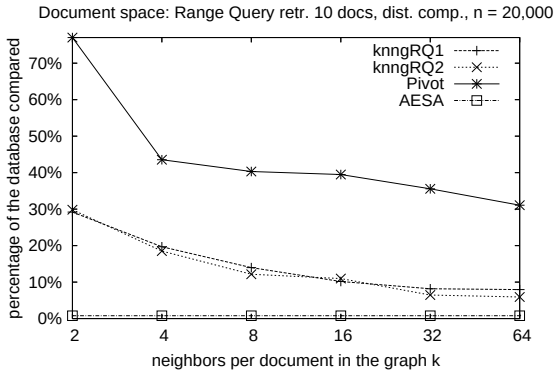
algorithm. These experiments also confirm that the greater the kNNG index size, the better the behavior of our search algorithms, especially when we add more space to kNNG graphs with few neighbors per node. With respect to AESA, in a range query to recover 1 document on average over a 64NNG index, kNNG**RQ2** requires 6.7 times the AESA distance computations, yet using only 0.48% of its space (for this space the full distance matrix requires 763 MB of main memory).

Figures 4.33(c) and (d) show nearest neighbor queries of increasing query outcome size over an 8NNG and a 32NNG. With respect to AESA, using a 32NNG index, kNNG**NNQ1** uses 14.7 times the AESA distance computations to find the closest document (yet using only 0.24% of its space). If we want to retrieve the two closest documents, this ratio reduces to 6.9 times AESA. Note that as the query size outcome increases, the performance ratio between our searching algorithms and AESA decreases.

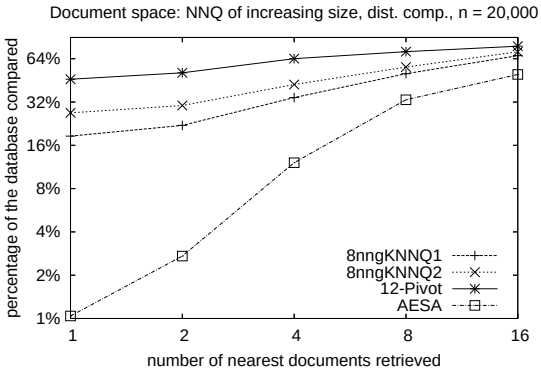
Finally, the elapsed times of Figure 4.34 show similar results with respect to Figure 4.33. This is because the cosine distance is so expensive to compute that it absorbs other



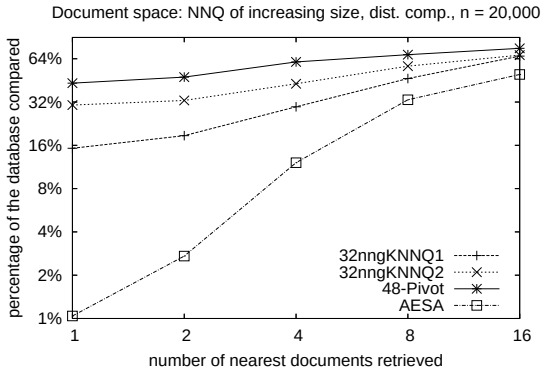
(a) Range queries retrieving 1 object on average, varying index size.



(b) Range queries retrieving 10 objects on average, varying index size.



(c) $NN_k(q)$ varing the query outcome size over an 8NNG.



(d) $NN_k(q)$ varing the query outcome size over a 32NNG.

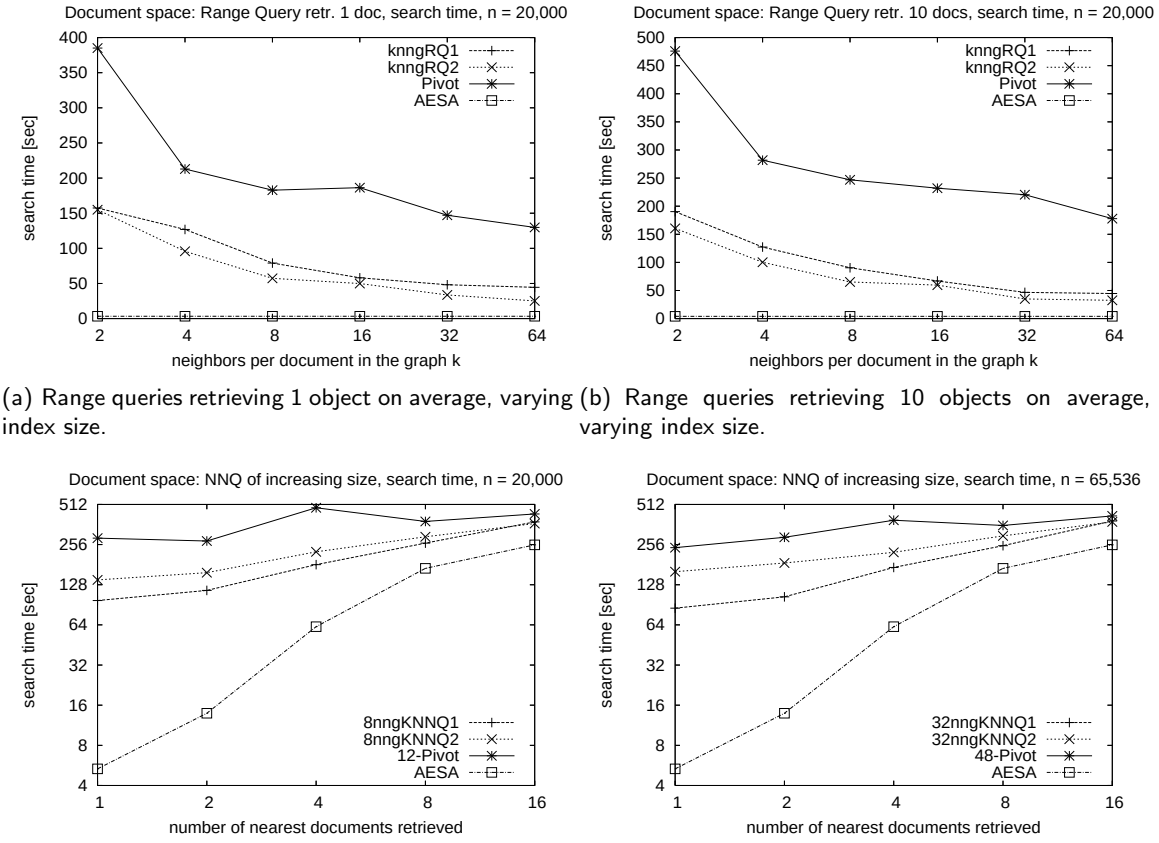
Figure 4.33: Evaluating percentage of database compared by our kNNG based search algorithms in the space of documents. On the first row, range queries varying the index size. On the second row, nearest neighbor queries varying the query outcome size. Note the logscale. We compare the search performance of our algorithms with a pivoting algorithm using $1.5k$ pivots.

components in the total CPU time in the search process, such as traversing the graph or scanning the pivot table. The elapsed time values of AESA do not consider the distance computations performed to simulate the full distance matrix.

4.4.5 Discussion of the Experimental Results

We have carried out several experiments using both synthetic and real-world data, and several testing scenarios in order to understand with reasonable deepness the performance of our proposals. In the following we summarize the experimental results.

With respect to the construction of kNNGs, we have obtained the following experimental results in the high-dimensional metric space of documents. K**NNpiv** requires $2.02n^{1.85}$ distance computations and $334n^{1.87}$ microseconds, which is clearly



(a) Range queries retrieving 1 object on average, varying index size.

(b) Range queries retrieving 10 objects on average, varying index size.

(c) $NN_k(q)$ varing the query outcome size over an 8NNG.

(d) $NN_k(q)$ varing the query outcome size over a 32NNG.

Figure 4.34: Evaluating elapsed time required by our kNNG based search algorithms in the space of documents. On the first row, range queries varying the index size. On the second row, nearest neighbor queries varying the query outcome size. Note the logscale. We compare the search performance of our algorithms with a pivoting algorithm using $1.5k$ pivots.

subquadratic. Note that the exponent values shows that most of the time required in the construction comes from computing the cosine distance. In low-dimensional metric spaces, our algorithms behave even better. For instance, in the string space, K**NNpiv** achieves an empirical CPU time $10.8n^{1.85}$ microseconds, and $99.9n^{1.26}$ distance computations.

The experimental results on vectors with uniform distribution show the usual exponential dependence on the dimensionality of the space. In fact, K**NNrp** requires $0.455e^{0.19D}n^{1.65}$ distance evaluations and $0.571e^{0.01D}n^{2.14}$ microseconds of CPU time. On the other hand, K**NNpiv** requires $0.685e^{0.24D}n^{1.48}$ distance computations and $0.858e^{0.11D}n^{1.95}$ microseconds of CPU time. Note that Euclidean distance is very cheap to compute, thus, CPU times are governed by the time needed to compute shortest paths through the graph. We remark that, in many metric space applications with costly distance functions, superquadratic CPU time in side computations is not as important as

a subquadratic number of computed distances, as can be appreciated in the experimental results on the document space. Finally, the construction experiments on Gaussian dataset has shown that our methodology profits from the clusters usually present in metric spaces.

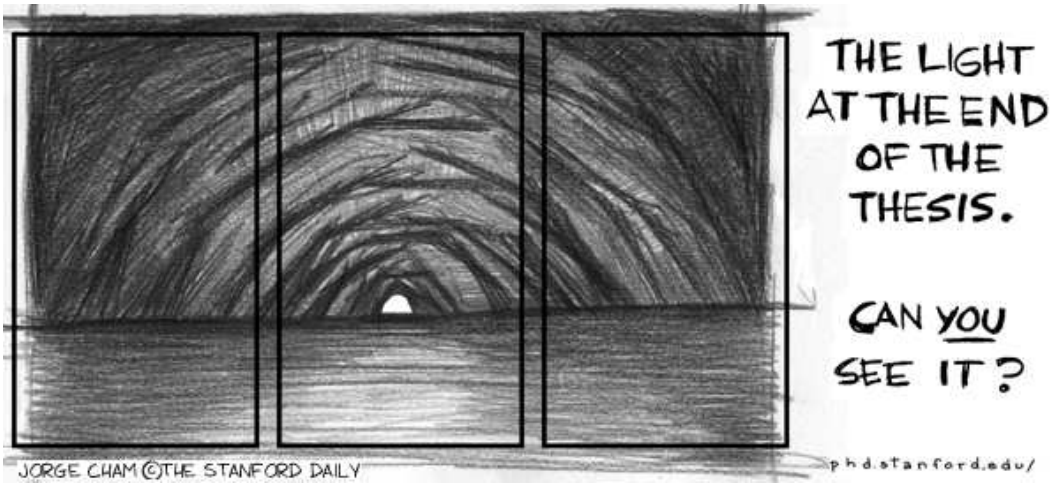
With respect to our kNNG based search algorithms, we have shown that they have practical applicability in low-memory scenarios —that is, with small values of k— for metric spaces of medium or high dimensionality. This can be appreciated throughout all the search experiments performed for kNNGs, yet it is especially noticeable in the space of uniformly distributed vectors, where we can see that for $D \geq 12$ our search performance is better than the classic pivot based alternative.

Note that our kNNG based approach is less sensitive both to the dimensionality of the space and to the selectivity of the query. This is shown by the fact that when we tested with vector spaces with larger dimensionality, or when we increased the query radius, the performance degradation of our algorithms was smother than that of the classical pivot approach using the same memory for the index, and for AESA[Vid86] (the best algorithm for metric space searching).

These search results were verified in the other metric spaces. We want to remark the experimental results in the space of documents: our search algorithms retrieve objects using approximately 29% of the distance evaluations (and also elapsed time) of the equivalent pivot based algorithm for a range of values of $k \in [2, 64]$. Moreover, our nearest neighbor algorithm uses just 30% more distance computations than AESA only using 0.25% of its space requirement.

Chapter 5

Conclusions



– *Piled Higher & Deeper*, #188, by Jorge Cham

Given a metric space $(\mathbb{X}, d)$ and a dataset $\mathbb{U} \subset \mathbb{X}$, we can solve similarity queries over the set $\mathbb{U}$ using the following procedure. We start by building offline an index $\mathcal{I}$ for the dataset $\mathbb{U}$ using any of the current metric space indices [CNBYM01, HS03, ZADB06, Sam06] and later compute online queries with the search algorithms appropriated for $\mathcal{I}$.

The state of the art of the metric indices can be divided into two main families [CNBYM01]: pivot based and compact-partition based. As we can only compare objects by computing the distance between them, the index can be seen, in abstract terms, as a subset of cells from the full distance matrix $\mathbb{U} \times \mathbb{U}$, or, more generally, as somehow summarizing the distance information of cells from $\mathbb{U} \times \mathbb{U}$.

In this thesis we have approached the metric space search problem from a different perspective, which consists in using graphs $G(V, E)$ to represent the metric database $\mathbb{U}$. In

this new model, each vertex of V represents an object of $\mathbb{U}$, and the edge set E corresponds to a small subset of weighted edges from the full set $V \times V$; or equivalently, E is a subset from $\mathbb{U} \times \mathbb{U}$, where the edge weights are the distances between the connected nodes according to the distance function d.

In the following we summarize the main contributions of this thesis. Finally, we give some directions for further work.

5.1 Contributions of this Thesis

> Many have marked the speed with which Muad'Dib learned the necessities of Arrakis. The Bene Gesserit, of course, know the basis of this speed. For the others, we can say that Muad'Dib learned rapidly because his first training was in how to learn. And the first lesson of all was the basic trust that he could learn. It's shocking to find how many people do not believe they can learn, and how many more believe learning to be difficult. Muad'Dib knew that every experience carries its lesson.
>
> – from *The Humanity of Muad'Dib* by the Princess Irulan, taken from *Dune*, by Frank Herbert

We have exhaustively explored the *k-nearest neighbor graph* (kNNG), which is a directed weighted graph connecting each element to its k nearest neighbors. For this sake, we have proposed algorithms both to construct kNNGs in general metric spaces and to use them to solve proximity queries. We are not aware of any other previous practical result published on the topic.

Motivated by the fact that graph algorithms make heavy use of fundamental algorithms and data structures, we have also explored some basic algorithmic problems throughout this work, such as *incremental sorting* and *priority queues*.

5.1.1 Fundamental Algorithms

We have presented *Incremental Quicksort* (**IQS**), an algorithm to incrementally retrieve the next smallest element from a set. **IQS** has the same expected complexity of existing solutions, but it is considerably faster in practice. It is nearly as fast as the best algorithm that knows beforehand the number of elements to retrieve. As a matter of fact, **IQS** is just 3% slower than *Partial Quicksort* [Mar04], the best offline alternative, yet **IQS** requires only 26% of the CPU time of the classical online alternative, consisting in heapifying the set and then performing k minimum extractions.

Based on the Incremental Quicksort algorithm, we have introduced *Quickheaps*, a simple and efficient data structure which implements priority queues. Quickheaps enable efficient element insertion, minimum finding, minimum extraction, deletion of arbitrary elements and modification of the priority of elements within the heap. We proved that the expected amortized cost per operation is $O(\log m)$, for a quickheap containing m elements. Quickheaps are as simple to implement as classical binary heaps, need almost no extra space, are efficient in practice, and exhibit high locality of reference. In fact, our experimental results show that in some scenarios quickheaps perform up to four times faster than binary heaps. They also show that quickheaps can outperform more complex implementations such as sequence heaps [San00], even in the scenarios where sequence heaps were designed for.

Exploiting the high locality of reference of Quickheaps, we have designed a cache-oblivious version, *External Quickheap*, that performs nearly optimally on secondary memory. The external quickheap implements efficient element insertion, minimum finding and minimum extraction. We proved that the amortized cost per operation in secondary memory is $O((1/B)\log(m/M))$ on disk, where m is the maximum heap size achieved, B the block size, and M the main memory size. For other operations like element deletion or modification of the priority, we need an extra dictionary to manage element positions and one extra random access to secondary memory. Our experimental results show that *Quickheaps* are extremely competitive in practice: using the same amount of memory, they perform up to 3 times fewer I/O accesses than *R-Heaps* [AMOT90] and up to 5 times fewer than *Array-Heaps* [BK98], which are the best alternatives tested in the survey by Brengel et al. [BCFM00]. In [OS02], authors show that despite in theory cache-oblivious priority queues have a good performance, in practice they have not. Nevertheless, our cache-oblivious *External Quickheap* is competitive with (and sometimes, more efficient than) the state of the art.

In order to analyze quickheaps we introduce a slight variation of the potential method [Tar85] (and [CLRS01, Chapter 17]) which we call the *potential debt method*. In this case, the potential debt (which is associated with the data structure as a whole) represents a cost that has not yet been paid. Thus, at the end, this total debt must be split among all the operations performed.

Both the algorithm **IQS** and the quickheap improve upon the current state of the art on many algorithmic scenarios. For instance, we plug our basic algorithms into classical Minimum Spanning Tree (MST) techniques [Wei99, CLRS01], obtaining two solutions that are competitive with the best (and much more sophisticated) current implementations: We use the incremental sorting technique to boost Kruskal's MST algorithm [Kru56], and the priority queue to boost Prim's MST algorithm [Pri57]. In the case of random graphs the expected complexities of the resulting MST versions are $O\big(m + n\log^2 n\big)$, where n is the number of nodes and m the number of edges.

5.1.2 k-Nearest Neighbor Graphs

We have presented a general methodology to construct k-nearest neighbor graphs in general metric spaces. On top of our methodology we give two construction algorithms. The first is based on a recursive partitioning of the space (K**NNrp**), and the second on the classical pivot technique (K**NNpiv**), using range-optimal queries when solving nearest neighbor queries. As usual in the metric space context, our methodology considers two stages: the first preindexes the space and the second completes the kNNG by performing several similarity queries. Note that in the second stage we not only use the preindex and metric properties to discard objects, but also several graph optimizations. To the best of our knowledge, there is no other practical algorithm to construct kNNGs in general metric spaces. So we compare our results with the basic construction algorithm requiring a quadratic number of distance computations.

kNNGs themselves can be used for many purposes, for instance, cluster and outlier detection [EE94, BCQY96], VLSI design, spin glass and other physical process simulations [CK95], pattern recognition [DH73], query or document recommendation systems [BYHM04a, BYHM04b], similarity self joins [DGSZ03, DGZ03, PR08], and many others. Hence, their construction is interesting *per se*.

In this thesis, we are in particular interested in how to use them to speed up metric queries. Thus, we have presented two metric space search algorithms that use the k-nearest neighbor graph as a metric index in order to solve range queries. Next, we obtain two nearest neighbor search algorithms induced by either of the range query ones. To do so, we manage the nearest neighbor query as a range query of decreasing radius.

We have carried out several experiments using both synthetic and real-world data, and several testing scenarios in order to understand with reasonable deepness the performance of our proposals. The synthetic spaces correspond to vectors with uniform and Gaussian distribution under Euclidean distance. With respect to real-world data, we have tested with the space of strings under the edit distance, and the space of documents under the cosine distance. Both real-world spaces are of interest to Information Retrieval [BYRN99]. We remark that the cosine distance is known for its high computation cost.

In all the spaces tested, we verify that our construction algorithms are subquadratic in distance evaluations. For instance, in the high-dimensional document space, K**NNpiv** requires $2.02n^{1.85}$ distance computations and $334n^{1.87}$ microseconds. Note that the exponent values shows that most of the time required in the construction comes from computing the cosine distance. In terms of percentage, K**NNpiv** requires 50% of the distance computations and 60% of the CPU time of the basic construction alternative in order to construct the 2NNG graph. Our construction algorithms behave even better in low- and medium-dimensional metric spaces. For instance, in the space of strings K**NNrp** requires 14% and K**NNpiv** only 1.1% of the distance evaluations of the basic algorithm to build the 64NNG.

From the experiments we conclude that K**NNpiv** is in general better than K**NNrp**

for small and moderate k values, yet K**NNrp** is less sensitive to larger k values or higher dimensional spaces.

With respect to our kNNG based search algorithms, we have shown that they have practical applicability in low-memory scenarios —that is, with small values of k— for metric spaces of medium or high dimensionality. Note that our kNNG based approach is less sensitive both to the dimensionality of the space and to the selectivity of the query. We want to remark the experimental results in the space of documents: our search algorithms retrieve objects using approximately 29% of the distance evaluations (and also elapsed time) of the equivalent pivot based algorithm for a range of values of $k \in [2, 64]$. Moreover, our nearest neighbor algorithm uses just 30% more distance computations than AESA only using 0.25% of its space requirement. We have experimentally shown that kNNGs offer an indexing alternative which requires a moderate amount of memory ($O(kn)$ space) obtaining reasonably good performance in the search process.

5.2 Further Work

– *Piled Higher & Deeper*, #844, by Jorge Cham

This research opens several research directions both in fundamental algorithms and on k-nearest neighbor graphs.

5.2.1 Fundamental Algorithms

IQS can be used for other more specialized purposes. For example, we can use the **IQS** stack-of-pivots underlying idea to partially sort in increasing/decreasing order starting from any place of the array. For instance, if we want to perform an incremental sorting in

increasing order (with a stack which previously stores the set size), we first use Quickselect to find the first element we want, storing in the stack all the pivots larger than the first element. Later we use **IQS** with that stack to search for the next elements (the other pivots, instead, would be useful to sort in decreasing order, by initializing the stack with -1). Moreover, with two stacks we can make centered searching, namely, finding the k-th element, the $(k+1)$-th and $(k-1)$-th, the $(k+2)$-th and $(k-2)$-th, and so on.

Similarly, research on extensions to quickheaps is interesting, such as implementing a kind of min-max-heap [ASSS86] by joining two quickheaps back-to-back, or quickheap variants, such as a non-sliding quickheap. We can also consider whether our quickheaps can improve the performance of other algorithms using priority queues.

There are also many research possibilities with external quickheaps. For instance, we can experiment with other sequences of accesses to the elements, and also with real CPU and I/O times. On the theoretical side, we also plan to achieve amortized worst-case guarantees for the data structure, for example by replacing the randomized pivoting by an order statistic on the first chunk.

Furthermore, we can consider studying the behaviour of our **IQS**-based Kruskal on different graph classes, and also research in variants tuned for secondary memory. This includes the study of variants of our approach focused not only on random graphs, but also on efficiently computing the MST of an arbitrary graph [KST03]. As such, Kruskal is very sensitive to the shape of graph it runs on.

Our research has demonstrated that, even in foundational and well-known problems, which are completely solved in the theoretical sense, algorithm engineering still has a lot to offer to improve the practical performance of algorithms and data structures. This is still more striking in the case of cache oblivious algorithms, which is a relatively recent trend mostly in a theoretical stage. Despite some practical studies showing that in some cases cache-oblivious algorithms are significantly inferior to their cache-aware counterparts [OS02], external quickheaps witness that in other cases cache-oblivious data structures can be extremely competitive. Again, there is much research ahead in algorithm engineering.

5.2.2 k-Nearest Neighbor Graphs

Future work involves developing another kNNG construction algorithm based on the list of clusters [CN05] as the preindex, so that we can also obtain good construction performance in higher dimensional metric spaces.

Our nearest neighbor search algorithms behave close to range-optimally. However, we can also consider the development of truly range-optimal nearest neighbor queries.

We also can research on kNNG optimizations tuned for our metric applications. For instance, we want to explore other local graphs, like the *all range-r graph*, where we assign to each node all the nodes within distance r. This way also allow us to control the size of the neighbor ball. Moreover, we can mix the kNNG with the *all range-r graph* in a graph

where each node is connected to its k-closest neighbors only if the distance between the node and the neighbor is smaller than r.

We are also researching on how to enhance the data structure to allow dynamic insertions/deletions in reasonable time, so as to maintain an up-to-date set of k-nearest neighbors for each element in the database. This problem is particularly appealing, especially when we face practical real-world problems where objects are not known from the beginning, but they arrive at any time.

Note that, for insertions, using the nearest neighbor search algorithms we can find the current k-nearest neighbor of a new object. The real problem is to efficiently solve the reverse nearest neighbor query; that is, to determine which objects in the graph have the new object inside its k-nearest neighbors. Computing reverse neighbors is still open, yet a simple approach is to solve a range query $(x, \max_{u \in k\text{NNG}}\{cr(u)\})$. That is, a range query using the maximum covering radius of any object within the graph. How to delete an object from the kNNG is completely open. The trivial solution is to use lazy deletion or to remove the victim object and leave its neighbors with one object less in their respective adjacency lists. However, we can also try to maintain an up-to-date kNNG by completing the neighbors using nearest neighbor queries, yet this could imply heavy deletion costs.

Another appealing question is how to manage the graph in secondary memory. In general, managing graphs in secondary memory is a subject of current active research [Vit01, ABT04]. In kNNGs, we suspect that we can exploit the fact that adjacency lists have limited size. We also notice that the on-disk graph representation has to be appropriate to compute shortest paths without performing many I/Os.

Another extremely relevant problem is that of computing the similarity join between two metric datasets A and B, $A \bowtie_r B$. That is, given both datasets and a threshold r computing all the object pairs (one from either set) at a distance at most r [DGSZ03, DGZ03, PR08]. The simplest solution translates the similarity join into indexing one of the sets (for instance, indexing dataset A) and later computing one range query per object from the other dataset (B in the example) in order to retrieve relevant objects from the indexed dataset (A in the example). However, we can also try with variants of the kNNG which index both sets jointly, as we have already done in [PR08] with the *list of twin clusters* (a metric index based on Chávez and Navarro *list of cluster* [CN05] specially focused on the similarity join primitive). Furthermore, we can research on how to solve other similarity join variants, for instance, finding the k-closest pairs of objects from the datasets A and B (one for each set), $A \bowtie_k B$; or similarity self joins, that is, when both datasets coincide, $A \bowtie_r A$.

Our research on kNNGs for metric spaces is a step towards implementing fully functional metric databases, that is, adding metric objects to the relational model [Cod70, GMUW02]. This translates into allowing the database manager system to natively support metric objects, which involves many appealing challenges. The first problem is to allow efficient database updating, which translates into efficient object insertion and deletion. The second problem is how to manage the metric index in secondary memory. The third is to solving joins among metric datasets. Fortunately, the query language would

have minor modifications, as we should only add primitives to handle the new metric data type. We also have to take into account transactions and concurrency, so that it can be possible to perform simultaneous insertion, deletion and search operations by several users without jeopardizing the database integrity. This implies to guarantee the atomicity, consistency, isolation and durability of transactions. The basic approach is to lock the whole index when performing operations, however this could be too restrictive in real work environments. Much better would be to design index access and locking methods that permit simultaneous operations over the index. This is a extremely relevant and challenging problem for future work.

> Un poème n'est jamais terminé, il est seulement abandonné.
> [A poem is never finished, only abandoned.]
>
> – Paul Valéry

Bibliography

> We do not write because we want to; we write because we have to.
>
> – William Somerset Maugham

[ABT04] L. Arge, G. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004.

[ADD+93] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, 9:81–100, 1993.

[ADDJ90] I. Althöfer, G. Das, D. Dobkin, and D. Joseph. Generating sparse spanners for weighted graphs. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT'90)*, LNCS 447, pages 26–37, 1990.

[AMN+94] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms (SODA'94)*, pages 573–583, 1994.

[AMOT90] R. Ahuja, K. Mehlhorn, J. Orlin, and R. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.

[Arg95] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In *Proc. 4th Intl. Workshop on Algorithms and Data Structures (WADS'95)*, LNCS 995, pages 334–345, 1995.

[ASSS86] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Comm. of the ACM*, 29(10):996–1000, 1986.

[Aur91] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[Bar98] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proc. 30th ACM Symp. on the Theory of Computing (STOC'98)*, pages 161–168, 1998.

[BBK01] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[BCFM00] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues in external memory. *ACM Journal of Experimental Algorithmics*, 5(17), 2000.

[BCQY96] M. Brito, E. Chávez, A. Quiroz, and J. Yukich. Connectivity of the mutual k-nearest neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35:33–42, 1996.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Comm. of the ACM*, 18(9):509–517, 1975.

[Ben79] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.

[BF03] G. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th ACM Symp. on Theory of Computing (STOC'03)*, pages 307–315, 2003.

[BFP+73] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[BK73] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Comm. of the ACM*, 16(4):230–236, 1973.

[BK98] G. Brodal and J. Katajainen. Worst-case external-memory priority queues. In *Proc. 6th Scandinavian Workshop on Algorithm Theory (SWAT'98)*, LNCS 1432, pages 107–118, 1998.

[BKK96] S. Berchtold, D. Keim, and H. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proc. 22nd Conf. on Very Large Databases (VLDB'96)*, pages 28–39, 1996.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[BO97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 357–368, 1997.

[Bol98] B. Bollobás. *Modern Graph Theory*. Springer, 1998.

[Bri95] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conf. on Very Large Databases (VLDB'95)*, pages 574–584. Morgan Kaufmann, 1995.

[BWY80] J. Bentley, B. Weide, and A. Yao. Optimal expected-time algotithms for closest point problems. *ACM Trans. on Mathematical Software*, 6(4):563–580, 1980.

[BY97] R. Baeza-Yates. Searching: An algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker, New York, 1997.

[BYCMW94] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[BYHM04a] R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query clustering for boosting Web page ranking. In *Proc. 2nd Atlantic Web Intelligence Conference (AWIC'04)*, LNCS 3034, pages 164–175, 2004.

[BYHM04b] R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query recommendation using query logs in search engines. In *Proc. Current Trends in Database Technology (EDBT Workshops'04)*, LNCS 3268, pages 588–596, 2004.

[BYRN99] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[Cal93] P. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proc. 34th IEEE Symp. on Foundations of Computer Science (FOCS'93)*, pages 332–340, 1993.

[CCG+98] M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin. Approximating a finite metric by a small number of tree metrics. In *Proc. 39th IEEE Symp. on Foundations of Computer Science (FOCS'98)*, pages 379–388, 1998.

[Cha71] J. M. Chambers. Algorithm 410 (PARTIAL SORTING). *Comm. of the ACM*, 14(5):357–358, 1971.

[Cha94] B. Chazelle. Computational geometry: a retrospective. In *Proc. 26th ACM Symp. on the Theory of Computing (STOC'94)*, pages 75–94, 1994.

[Cha00] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[Chi94] T. Chiueh. Content-based image indexing. In *Proc. 20th Conf. on Very Large Databases (VLDB'94)*, pages 582–593, 1994.

[CK95] P. Callahan and R. Kosaraju. A decomposition of multidimensional point sets with applications to k nearest neighbors and n body potential fields. *Journal of the ACM*, 42(1):67–90, 1995.

[Cla83] K. Clarkson. Fast algorithms for the all-nearest-neighbors problem. In *Proc. 24th IEEE Symp. on Foundations of Computer Science (FOCS'83)*, pages 226–232, 1983.

[Cla99] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[CMN01] E. Chávez, J. L. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001.

[CN05] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[CNBYM01] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Comm. of the ACM*, 13(6):377–387, 1970.

[Coh98] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. 23rd Conf. on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

[Cra72] C. A. Crane. Linear lists and priority queues as balanced binary tree. Technical Report STAN-CS-72259, Stanford University, 1972.

[CT76] D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5:724–742, 1976.

[DE96] M. Dickerson and D. Eppstein. Algorithms for proximity problems in higher dimensions. *Computational Geometry Theory and Applications*, 5:277–291, 1996.

[DGSZ03] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. Similarity join in metric spaces. In *Proc. 25th European Conf. on IR Research (ECIR'03)*, LNCS 2633, pages 452–467, 2003.

[DGZ03] V. Dohnal, C. Gennaro, and P. Zezula. Similarity join in metric spaces using eD-index. In *Proc. 14th Intl. Conf. on Database and Expert Systems Applications (DEXA'03)*, LNCS 2736, pages 484–493, 2003.

[DH73] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. Mathematisch Centrum, Amsterdam, The Netherlands, 1959.

[DN87] F. Dehne and H. Noltemeier. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[EE94] D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete & Computational Geometry*, 11:321–350, 1994.

[Epp99] D. Eppstein. Spanning trees and spanners. In *Handbook of Computational Geometry*, pages 425–461. Elsevier, 1999.

[FCNP06] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes. On the least cost for proximity searching in metric spaces. In *Proc. 5th Intl. Workshop on Experimental Algorithms (WEA'06)*, LNCS 4007, pages 279–290, 2006.

[Fig00] K. Figueroa. Un algoritmo eficiente para el problema de todos los k vecinos más cercanos en espacios métricos (An efficient algorithm for all k nearest neighbor problem in metric spaces). Master's thesis, Universidad Michoacana, Mexico, 2000. In Spanish.

[FJKT99] R. Fadel, K. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[Flo62] R. W. Floyd. Algorithm 97: Shortest path. *Comm. of the ACM*, 5(6):345, 1962.

[Flo64] R. W. Floyd. Algorithm 245 (TREESORT). *Comm. of the ACM*, 7:701, 1964.

[FLPR99] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Symp. on Foundations on Computer Science (FOCS'99)*, pages 285–297, 1999.

[FNC07] K. Figueroa, G. Navarro, and E. Chávez. Metric spaces library, 2007. Available at `http://sisap.org/?f=library`.

[FSST86] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[GBC+99] M. Goldwasser, J. Bentley, K. Clarkson, D. S. Johnson, C. C. McGeoch, and R. Sedgewick. The sixth dimacs implementation challenge: Near neighbor searches, January 1999. Available at `http://dimacs.rutgers.edu/Challenges/Sixth/`.

[GBY91] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 2nd edition, 1991.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GLN02] J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Fast greedy algorithms for constructing sparse geometric spanners. *SIAM Journal on Computing*, 31(5):1479–1500, 2002.

[GMUW02] H. Garcia-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2002.

[GS69] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation. *Systematic Zoology*, 18:259–278, 1969.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1984.

[Har95] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conf. (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[HMSV97] D. Hutchinson, A. Maheshwari, J. Sack, and R. Velicescu. Early experiences in implementing buffer trees. In *Proc. 2nd Intl. Workshop on Algorithmic Engineering (WAE'97)*, pages 92–103, 1997.

[Hoa61] C. A. R. Hoare. Algorithm 65 (FIND). *Comm. of the ACM*, 4(7):321–322, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[HS76] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Science Press, 1976.

[HS00] G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report TR 4199, Dept. of Comp. Sci. Univ. of Maryland, Nov 2000.

[HS03] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. on Database Systems*, 28(4):517–580, 2003.

[JD88] A.K. Jain and R.C. Dubes. *Algorithms For Clustering Data*. Prentice-Hall, Englewood Cliffs, 1988.

[JKŁP93] S. Janson, D. Knuth, T. Łuczak, and B. Pittel. The birth of the giant component. *Random Structures & Algorithms*, 4(3):233–358, 1993.

[Kei88] J. M. Keil. Approximating the complete Euclidean graph. In *Proc. 1st Scandinavian Workshop in Algorithm Theory (SWAT'88)*, LNCS 318, pages 208–213, 1988.

[KKT95] D. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[KL04] R. Krauthgamer and J. Lee. Navigating nets: simple algorithms for proximity search. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms (SODA'04)*, pages 798–807, 2004.

[KM83] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Trans. on Software Engineering*, 9(5), 1983.

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[KP94] G. Kortsarz and D. Peleg. Generating sparse 2-spanners. *Journal of Algorithms*, 17(2):222–236, 1994.

[KR02] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proc. 34th ACM Symp. on the Theory of Computing (STOC'02)*, pages 741–750, 2002.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[KS96] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE Symp. on Parallel and Distributed Processing (SPDP'96)*, page 169, 1996.

[KST02] I. Katriel, P. Sanders, and J. Träff. A practical minimum spanning tree algorithm using the cycle property. Research Report MPI-I-2002-1-003, Max-Planck-Institut für Informatik, October 2002.

[KST03] I. Katriel, P. Sanders, and J. Träff. A practical minimum spanning tree algorithm using the cycle property. In *Proc. 11th European Symp. on Algorithms (ESA'03)*, LNCS 2832, pages 679–690, 2003.

[LB96] W. Liang and R. Brent. Constructing the spanners of graphs in parallel. Technical Report TR-CS-96-01, Dept. of CS and CS Lab, The Australian National University, January 1996.

[Mar04] C. Martínez. Partial quicksort. In *Proc. 6th ACM-SIAM Workshop on Algorithm Engineering and Experiments and 1st ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics (ALENEX-ANALCO'04)*, pages 224–228, 2004.

[MOV94] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[MS91] B. Moret and H. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Proc. 2nd Workshop Algorithms and Data Structures (WADS'91)*, LNCS 519, pages 400–411, 1991.

[Nav02] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[NP03] G. Navarro and R. Paredes. Practical construction of metric t-spanners. In *Proc. 5th Workshop on Algorithm Engineering and Experiments (ALENEX'03)*, pages 69–81, 2003.

[NPC02] G. Navarro, R. Paredes, and E. Chávez. t-Spanners as a data structure for metric space searching. In *Proc. 9th Intl. Symp. on String Processing and Information Retrieval (SPIRE'02)*, LNCS 2476, pages 298–309, 2002.

[NPC07] G. Navarro, R. Paredes, and E. Chávez. t-spanners for metric space searching. *Data & Knowledge Engineering*, 63(3):818–852, 2007.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[OS02] J. Olsen and S. Skov. Cache-oblivious algorithms in practice. Master's thesis, University of Copenhagen, 2002.

[Par02] R. Paredes. Uso de t-spanners para búsqueda en espacios métricos (Using t-spanners for metric space searching). Master's thesis, Universidad de Chile, 2002. In Spanish.

[PC05] R. Paredes and E. Chávez. Using the k-nearest neighbor graph for proximity searching in metric spaces. In *Proc. 12th Intl. Symp. on String Processing and Information Retrieval (SPIRE'05)*, LNCS 3772, pages 127–138. Springer, 2005.

[PCFN06] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Proc. 5th Intl. Workshop on Experimental Algorithms (WEA'06)*, LNCS 4007, pages 85–97, 2006.

[PN06] R. Paredes and G. Navarro. Optimal incremental sorting. In *Proc. 8th Workshop on Algorithm Engineering and Experiments and 3rd Workshop on Analytic Algorithmics and Combinatorics (ALENEX-ANALCO'06)*, pages 171–182. SIAM Press, 2006.

[PR02] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[PR08] R. Paredes and N. Reyes. List of twin clusters: a data structure for similarity joins in metric spaces. In *Proc. 1st Intl. Workshop on Similarity Search and Applications (SISAP'08)*, page to appear, 2008.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[PS89] D. Peleg and A. Schaffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[PU89] D. Peleg and J. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:740–747, 1989.

[R D04] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004.

[Ros04] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Academic Press - Elsevier, 3rd edition, 2004. Indian reprint, 2005.

[RS91] J. Ruppert and R. Seidel. Approximating the d-dimensional complete Euclidean graph. In *Proc. 3rd Canadian Conf. on Computational Geometry*, pages 207–210, 1991.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, New York, 2006.

[San00] P. Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics*, 5:7, 2000.

[SK02] T. B. Sebastian and B. B. Kimia. Metric-based shape retrieval in large databases. In *Proc. 16th Intl. Conf. on Patter Recognition*, volume 3, pages 291–296, 2002.

[ST86] D. D. Sleator and R. E. Tarjan. Self adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, 1986.

[SW90] D. Shasha and T. Wang. New techniques for best-match retrieval. *ACM Trans. on Information Systems*, 8(2):140–158, 1990.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[Tar85] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[Tou80] G. T. Toussaint. The relative neighborhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

[TZ01] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. 33rd ACM Symp. on Theory of Computing (STOC'01)*, pages 183–192, 2001.

[Uhl91] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[Vai89] P. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. *Discrete & Computational Geometry*, 4:101–115, 1989.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Vid86] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[Vit01] J. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001. Version revised at 2007 from `http://www.cs.duke.edu/~jsv/Papers/Vit.IO_survey.pdf`.

[Vui78] J. Vuillemin. A data structure for manipulating priority queues. *Comm. of the ACM*, 21(4):309–315, 1978.

[Weg93] I. Wegener. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118(1):81–98, 1993.

[Wei99] M. A. Weiss. *Data Structures & Algorithm Analysis in C++*. Addison-Wesley, 2nd edition, 1999.

[Wes01] D. B. West. *Introduction to Graph Theory*. Prentice-Hall, 2nd edition, 2001.

[Wil64] J. Williams. Algorithm 232 (HEAPSORT). *Comm. of the ACM*, 7(6):347–348, 1964.

[WJ96] D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, La Jolla, California, July 1996.

[Yia93] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms (SODA'93)*, pages 311–321. SIAM Press, 1993.

[Yia98] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. Technical report, NEC Research Institute, 1998. In *6th DIMACS Implementation Challenge: Near Neighbor Searches Workshop, ALENEX'99*.

[ZADB06] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.