Kd-Trees for Document Layout Analysis*

Christoph Dalitz

Hochschule Niederrhein

Fachbereich Elektrotechnik und Informatik

Reinarzstr. 49, 47805 Krefeld, Germany

Abstract

Kd-trees allow for efficient nearest neighbor searches and can therefore be useful for layout analysis problems in document image analysis. Two alternative customizations of the k nearest neighbor search in kd-trees are presented, such that it returns only within line or between line neighbors. One customization is based on a modified distance metric, which does not entirely suppress unwanted neighbors, but makes them less likely to be returned. The other customization is based on a search predicate. Both customizations have been implemented in the Gamera framework for document analysis and recognition. In experiments performed on the UW dataset, the probability for an unwanted neighbor with the modified distance metric turned out to be low, while the runtime of the search was considerably less than that of the search with a predicate.

1 Introduction

Many algorithms for document layout analysis are of a "bottom-up" type: they start from *connected components* (CCs) and subsequently build therefrom larger units. The first step in these algorithms usually consists in building clusters of neighbors among the connected components. Prominent examples are the determination of the skew angle from the histogram of angles between neighboring CCs [1] [2], text line detection by finding paths in the neighborship graph [3], or O'Gorman's "docstrum" [4].

A naive implementation for finding all nearest neighbor pairs from a set of n two dimensional points takes $O(n^2)$ time. O'Gorman suggested to sort the points in one of the two dimensions [4], which happens to be an algorithm proposed about 20 years earlier by Friedman et al. [5], who have shown that this algorithm has $O(n^{3/2})$ runtime complexity on average. Its worst case runtime is still $O(n^2)$. To further reduce the runtime, two common geometric data structures can be used: *Voronoi diagrams* or *kd-trees*.
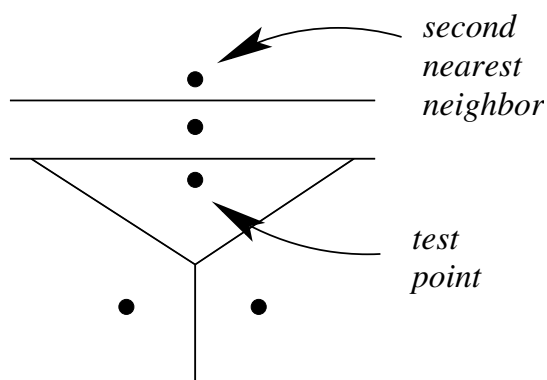

Figure 1: Example for a Voronoi diagram in which the second nearest neighbors is not in one of the neighboring cells.

In a Voronoi diagram, each of the points creates a cell, in which every element of that cell is closer to the cell defining point than to any other point. Thus, a Voronoi diagram is not actually a neighborship graph based on distance, but on adjacency [6], see Fig. 1. If only the nearest neighbor is searched, this must also be one of the adjacent neighbors, and, consequently, the Voronoi diagram can also be used to find the nearest neighbor.

Even though a Voronoi diagram can be built in $O(n \log n)$ time [7], the worst case run time for a single nearest neighbor search in a Voronoi diagram is still $O(n)$[1]. When adjacency, however, is more important than distance, the Voronoi diagram can be a powerful tool for document layout analysis, especially when each CC is not only represented by a single point, but its shape is taken into account by using a generalization of the Voronoi diagram, the *area Voronoi* diagram [8] [9].

When the layout analysis algorithm requires not adjacent neighbors, but the actual k nearest neighbors of a CC (like O'Gorman's "docstrum" [4]), a kd-tree is the appropriate data structure for searching. A kd-tree can be built in $O(n \log n)$ time and a single k nearest neighbor search can be done in $O(\log n)$ expected time, resulting in $O(n \log n)$ time for finding all k nearest neighbor sets. Kd-trees were originally introduced by Bentley [10]; a modern text book introduction can be found in [7]. Neither of these references covers nearest neighbor searches in kd-trees; these are described in [11].

It should be noted that the worst case runtime for a nearest neighbor search in a kd-tree is still $O(n)$. Consider, e.g., the case of all points arranged on a circle and seeking the nearest neighbor to the circle center. Recently, more sophisticated data structures have been proposed that even have a guaranteed runtime of $O(\log n)$ [12]. Typical document layout analysis problems however not require a single nearest neighbor query, but, rather, several queries, a problem also

---

[1]The worst case occurs when the cell of the test point shares edges with all other points.

known as the *all-k-nearest-neighbor* problem. This has the effect that pathological situations cannot dominate the worst case search runtime, and it is indeed possible to obtain worst case $O(n \log n)$ runtime for this problem with kd-trees [13].

For document layout analysis, not only the distance between CCs is of interest, but also their relative position. In skew angle estimation for instance, only neighbor pairs within the same text line are of interest. As the distance between neighboring CCs within a line is generally smaller than the distance between lines, the nearest neighbor pairs will be dominated by within-line pairs. When between-line pairs are needed, a plain nearest neighbor search is insufficient. A workaround could be to to select k nearest nearest neighbors for each CC, in the hope that among them there might also be some from a different text line. O'Gorman, for instance, used $k = 5$ [4]. It would be however more satisfying to have an option to directly search for within-line or between-line neighbors.

In this paper, two customizations of the nearest neighbor search are presented that are especially tailored to this problem. One is the obvious approach to allow for an optional search predicate. This ensures strictly that only neighbors fulfilling some condition are returned, but it has the disadvantage of possibly increasing the runtime complexity. The other customization consists in modifying the distance metric in such a way that neighbors of a specific type are returned more often. When this is acceptable, this approach has the advantage of maintaining the runtime complexity of kd-trees.

This paper is organized as follows: section 2 gives an introduction to kd-trees and section 3 describes a straightforward C++ implementation of a kd-tree that the author recently has added to the Gamera framework for document analysis[2]. Section 4 describes search modifications for within- and between-line searching, and an interface for their implementation based on C++ function objects. Section 5 presents the results of an experiment, in which the two search modifications were compared to each other on the UW-I image dataset of the University of Washington.

2   Kd-Tree Basics

A kd-tree is a binary tree whose nodes represent points $\{\mathbf{x}_i\}_{i=1}^{n}$ in $\mathbb{R}^d$. Each node has assigned a cutting dimension *cutdim*, such that all points in a left

---

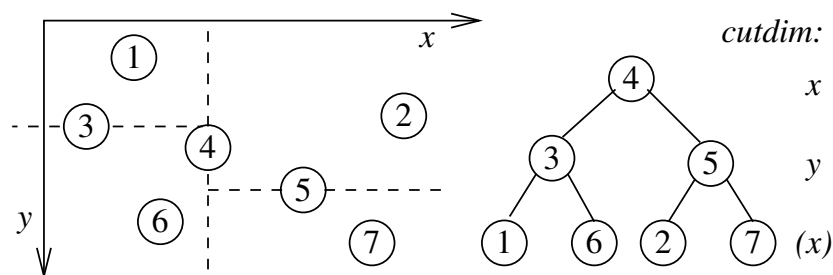[2]Freely available from `http://gamera.informatik.hsnr.de/`

Figure 2: Example of a kd-tree built from seven points in $\mathbb{R}^2$. The dashed lines show how the plane is cut by the node along *cutdim*.

subtree have a smaller coordinate value in this dimension than the parent point and vice versa for the right subtree (see Fig. 2). This has the effect that all nodes in a subtree are bound to a specific region in $\mathbb{R}^d$, which can be utilized to speed up k nearest neighbor searches because regions farer away than the k-th nearest neighbor found so far need not be searched. In a detailed analysis, Friedman et al. have shown that the expected runtime for a k nearest neighbor search is $O(\log n)$, provided the tree is balanced [11].

While a balanced kd-tree can be built in $O(n \log n)$ time, keeping a kd-tree balanced on insertions and deletions can be expensive. Therefore, kd-trees are usually used as static data structures. This is no restriction for layout analysis problems, because the location of the CCs is fixed and on these fixed data many queries are to be done. Friedman et al. give algorithms both for building a balanced kd-tree and for a k-nearest neighbor search [11]. Their k nearest neighbor search algorithm assumes that the distance measure is of the form

$$d(\mathbf{x}, \mathbf{y}) = F\left(\sum_{i=1}^{d} f_i(x_i, y_i)\right) \tag{1}$$

where F is monotonous, and all f_i are monotonous in both arguments and symmetric. Typical examples for distance measures fulfilling (1) are the Minkowski distances of order p

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{d} |x_i - y_i|^p\right)^{1/p} \tag{2}$$

The search algorithm uses two utility functions to determine whether a search is finished or whether a subtree can contain nearest neighbor candidates:

- *ball_within_bounds* tests whether the circle around the search point with the radius of the current k-th farest nearest neighbor lies entirely in the subtree; if this is the case the search is finished.

```
                                    1..*
┌──────────────────────┐ ┌─────────────────────────────────────────────┐
│       KdNode         │ │                                             │
├──────────────────────┤ │  ┌───────────────────────────────────────┐  │
│  point : CoordPoint  │ │  │  KdTree                               │  │
│  data : void*        │ │  ├───────────────────────────────────────┤  │
└──────────────────────┘ │  │  dimension: int                       │  │
                         │  │  allnodes : vector<KdNode>            │  │
┌──────────────────────┐ │  │  root : kdtree_node*                  │  │
│    kdtree_node       │ │  │  distance : DistanceMeasure*          │  │
├──────────────────────┤ │  ├───────────────────────────────────────┤  │
│  point : CoordPoint  │ │  │ + KdTree(nodes : vector<KdNodes>)     │  │
│  cutdim : int        │ │  │ + k_nearest_neighbors(point : CoordPoint, │
│  hison, loson : kdtree_node* │ │ k : int, result : vector<KdNode>*) │  │
│  hibound, lobound: CoordPoint │ │ − neighbor_search(point : CoordPoint, │
│  dataindex : int     │ │  │      node : kdtree_node*, k : int)    │  │
└──────────────────────┘ │  │ − ball_within_bounds(point : CoordPoint, │
                         │  │      dist : double, node : kdtree_node*) │
┌──────────────────────┐ │  │ − bounds_overlap_ball(point : CoordPoint, │
│   DistanceMeasure    │ │  │      dist : double, node : kdtree_node*) │
├──────────────────────┤ │  │ − build_tree(depth : int, index_a : int, │
│ distance(p,q : CoordPoint) │ │      index_b : int) : kdtree_node* │  │
│      : double        │ │  └───────────────────────────────────────┘  │
│ coord_distance(x,y : double, │ └─────────────────────────────────────────────┘
│      dim : int) : double │
└──────────────────────┘
```
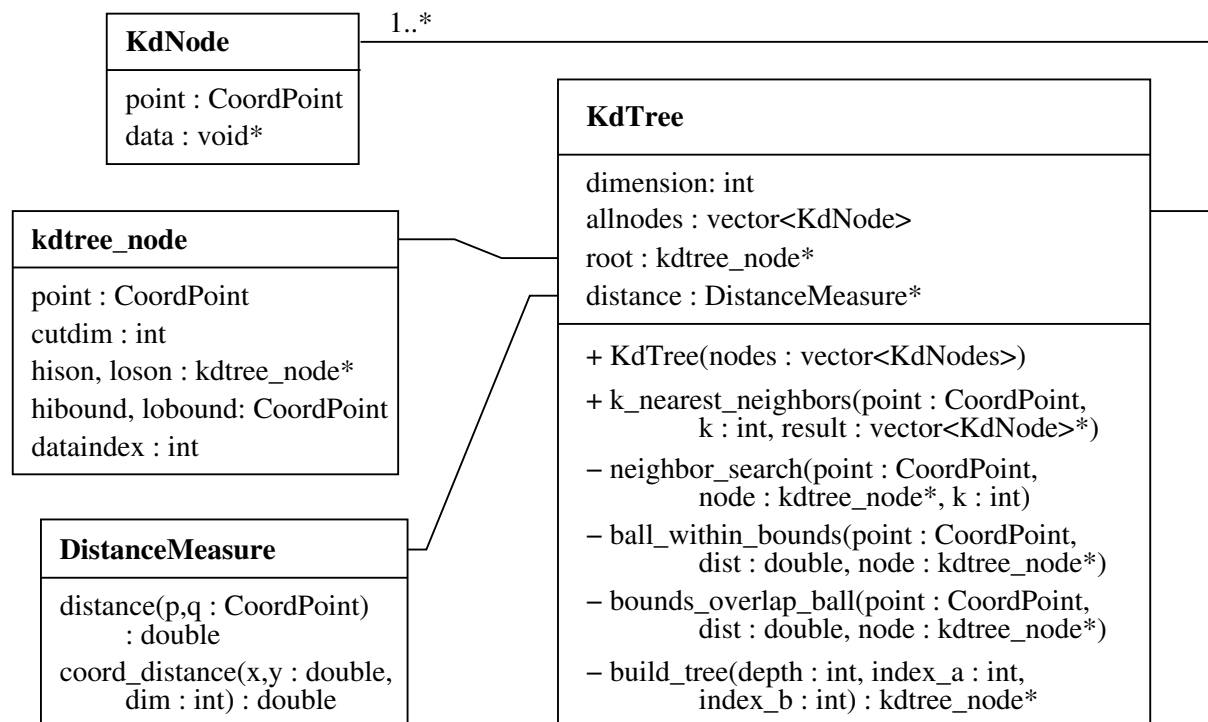
Figure 3: Class diagram of the classes involved in the kd-tree implementation. *CoordPoint* is a typedef for *vector<double>*. The private functions of *KdTree* are the functions given in the appendix of reference [11].

- *bounds_overlap_ball* tests whether the circle mentioned above overlaps with a subtree region; if this is not the case, the subtree needs not to be searched.

In the original algorithm by Friedman et al., the bounds for the subtree regions were computed on the fly during the search. When many queries are done on the same tree, it is however more efficient to store the bounds with each node while building the tree.

3   Kd-Tree Implementation

In this section, a kd-tree implementation is described that the author has written for the Gamera framework for document analysis and recognition [14]. As the source code of this implementation is freely available[3], I give here only the main ideas. An overview over the classes involved can be seen in Fig. 3.

When applying kd-trees to document layout analysis problems, we must distin-

---

[3]See the files *include/kdtree.hpp* and *src/kdtree/kdtree.cpp* in the Gamera source code distribution.

```
from gamera.kdtree import *

ccs = image.cc_analysis()
nodes = [KdNode([cc.center.x,cc.center.y], cc) for cc in ccs]
tree = KdTree(nodes)

nn_pairs = []
for cc in ccs:
   knn = tree.k_nearest_neighbors([cc.center.x,cc.center.y], 2)
   nn_pairs.append([cc, knn[1].data])
```

Listing 1: Python code for finding all nearest neighbor pairs among the connected components of an image.

guish between the *internal* and *external* representation of a node. The *external* representation is the class *KdNode*, which is a node from the user's point of view. Beside the point in $\mathbb{R}^d$, it can store an arbitrary object (in typical use cases a connected component) in its property *data*. A vector of *KdNode*'s must be passed to the kd-tree constructor.

The *internal* representation is the class *kdtree_node*, which is only needed within the kd-tree data structure. Beside the point location, the cut dimension, pointers to its sons, and the bounds of its subtree's region, it also stores the index of the corresponding *KdNode* from *KdTree.allnodes* in its property *dataindex*.

To allow for flexible distance measures, the *KdTree* class does not use a fixed function for computing the distance between two points, but stores a pointer *distance* to a class *DistanceMeasure*. This is a class that implements the functions $d(\mathbf{x}, \mathbf{y})$ and $f_i(x_i, y_i)$ from Eq. (1) as member functions *distance()* and *coordinate_distance()*. Both member functions are defined to be *virtual* in the base class *DistanceMeasure* so that custom metrics can be implemented as derived classes that are stored in *KdTree.distance*. As the building of the tree is independent from the distance measure, the distance metric may be changed between subsequent calls to *KdTree.k_nearest_neighbors()*.

The kd-tree implementation in Gamera closely follows [11], with additional utilization of the C++ Standard Template Library (STL) [15]: the optimal split point for a balanced tree is found with the *nth_element* algorithm, which finds the median in linear time, and the current neighbor candidates in the k nearest neighbor search are stored in a *priority_queue*.

One of the strengths of the Gamera framework lies in its ease of use as a Python library. Therefore the kd-tree implementation has also been made available on
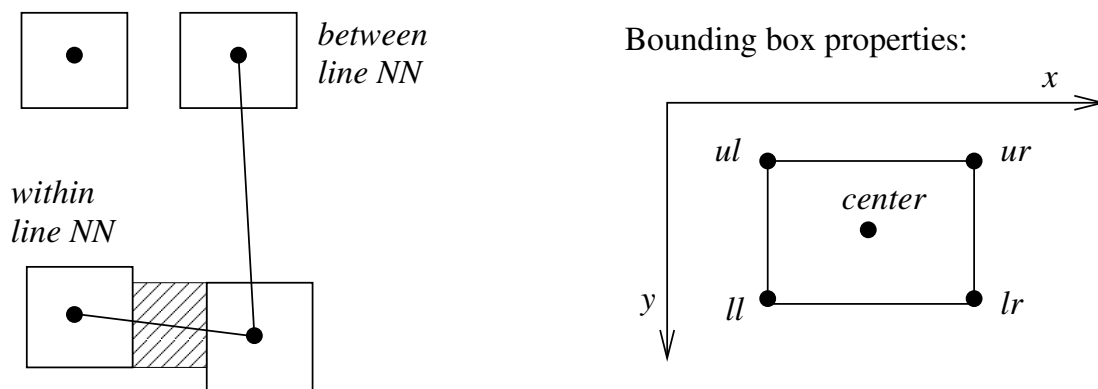
Figure 4: The bounding boxes of neighbors within the same line have overlapping y dimensions (shaded region in left figure). This criterion can be computed from the bounding box properties shown in the right figure.

the Python side with only the the classes *KdNode* and *KdTree* made public. An example for the use of the kd-tree API can be seen in listing 1.

4   Within-Line and Between-Line Neighbors

Under the assumption that text lines are not too strongly skewed, we can define two neighboring components a and b to belong to the same text line when the following property holds (see Fig. 4):

$$[a.ur_y, a.lr_y] \; \cap \; [b.ur_y, b.lr_y] \; \neq \; \emptyset \tag{3}$$

where ur_y and lr_y stand for the y-component of the upper right and lower right corner of the bounding box, respectively, and the square brackets denote closed intervals. Let us call neighbors fulfilling (3) *within-line neighbors*, and neighbors not fulfilling this condition *between-line neighbors*.

To use a kd-tree for finding only one specific type of neighbors, the Gamera kd-tree API adds an optional search predicate to the arguments of the method *KdTree.k_nearest_neighbors* (see Fig. 3). For passing the search predicate, a *function object* (also known as *callable class* or *functor* [15]) is the appropriate programming construct. In the C++ API, the search predicate must be derived from *KdNodePredicate*, while it can be any callable class in the Python API. The call of the class may only take a single argument, the *KdNode*, and returns true when this node is an admissible search result. If this criterion depends on some information from the actual search point (like the bounding box dimension $[ur_y, lr_y]$ in (3)), this information can be passed to the class constructor and stored within the class. Listing 2 gives an example both for Python and C++.

45

```python
# defining a search predicate in Python
#----------------------------------------
# predicate definition as callable class
class predicate(object):
    def __init__(self, cc):
        self.top = cc.ur.y
        self.bot = cc.lr.y
    def __call__(self, node):
        cc = node.data
        if (self.top >= cc.ur_y and self.top <= cc.lr_y) \
                or (cc.ur_y >= self.top and cc.ur_y <= self.bot):

# find three nearest neighbors of a CC fulfilling predicate
knn = tree.k_nearest_neighbors([cc.center.x, cc.center.y], \
                               3, predicate(cc))
```

```cpp
// defining a search predicate in C++
//-----------------------------------
// predicate definition as a functor
struct MyPredicate : public KdNodePredicate {
  size_t top, bot;
  MyPredicate(size_t t, size_tb) {
    top = t; bot = b;
  }
  bool operator()(const KdNode& kn) const {
    size_t cctop = kn.data->offset_y();
    size_t ccbot = kn.data->offset_y() + kn.data->nrows();
    return ((top >= cctop && top <= ccbot) ||
            (cctop >= top && cctop <= bot));
  };
};

// find three nearest neighbors of a CC fulfilling predicate
MyPredicate predicate(cc.offset_y(),cc.offset_y()+cc.nrows());
CoordPoint center(2);
center[0] = (double)(cc.offset_x() + cc.ncols()/2)
center[1] = (double)(cc.offset_y() + cc.nrows()/2)
tree.k_nearest_neighbors(center, 3, &neighbors, &predicate);
```

Listing 2: Defining condition (3) as a search predicate for the Gamera kd-tree API in Python (top) or in C++ (bottom). In both cases, it is assumed that *KdNode.data* stores the CC associated with the point.

An alternative way to obtain only a specific type of neighbors can be devised from the observation that within-line neighbors typically have a much larger
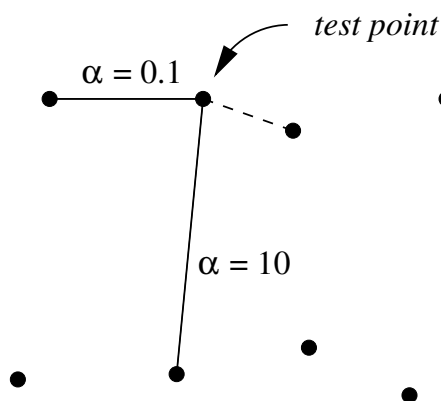
Figure 5: By modifying the factor α in the distance measure (4), different points become the nearest neighbor (NN) of the test point. The dashed line shows the Euclidean NN, while $\alpha \ll 1$ leads to a better vertically aligned neighbor, and $\alpha \gg 1$ to a better horizontally aligned neighbor (between-line neighbor). Here $p = 2$ is used, which is the exponent used in the Euclidean distance.

distance in the x-direction than in the y-direction:

$$|a.center_y - b.center_y| \ll |a.center_x - b.center_x|$$

and vice versa for between-line neighbors. It is therefore possible to make neighbors of one kind "closer" than neighbors of the other kind by giving different weights to the two directions in the distance metrics (2):

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{d} w_i |x_i - y_i|^p \right)^{1/p}$$

As we only have two dimensions, and only the relative values of distances are of importance for nearest neighbor searches, we can express the two weights (w_x, w_y) by a single number, their ratio $\alpha = w_x / w_y$. Then the weighted distance reads

$$d(\mathbf{a}, \mathbf{b}) = \left(\alpha |a_x - b_x|^p + |a_y - b_y|^p \right)^{1/p} \tag{4}$$

A value of $\alpha < 1$ means that the y-distance has a higher penalty than the x-distance. As shown in Fig. 5, we can then find within-line or between-line neighbors for appropriate choices of α.

This method is not fool-proof, but depends on the choice for α. Moreover, even decent choices for α can occasionally return neighbors of the wrong kind. This leads us to two questions:

- What are good choices for the weight α and what is the error rate on realistic document images?

- Does the distance weighted approach have an edge in terms of runtime performance?

Both questions will be answered in the following section.

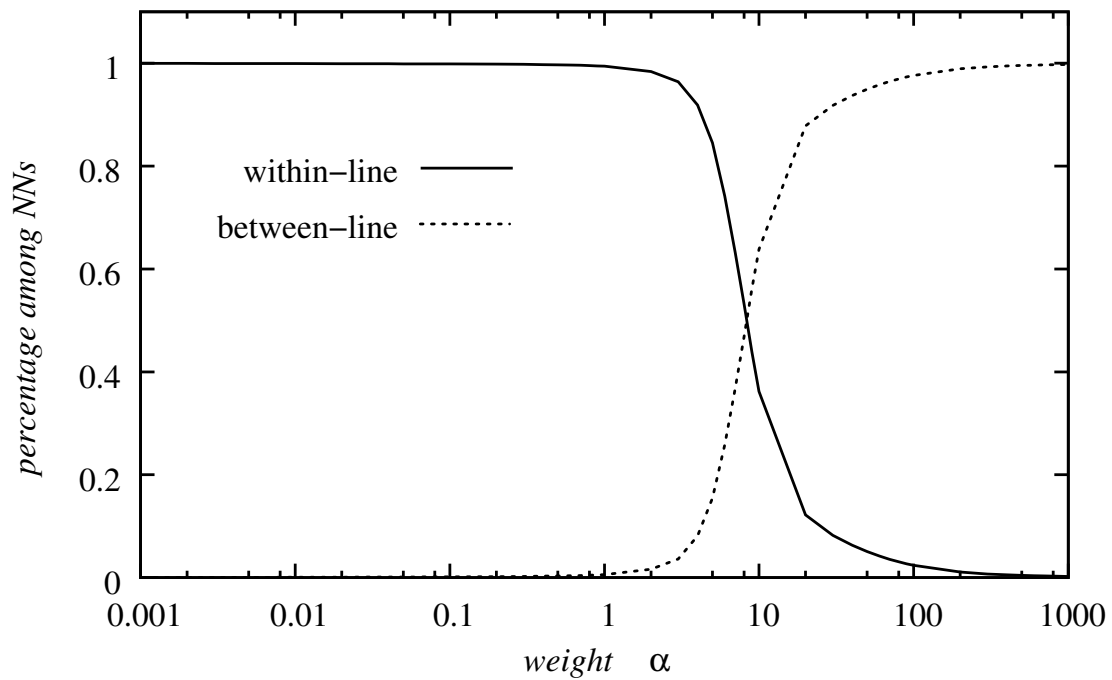# 5   Experiments on the UW-I Image Dataset

To examine the questions raised in the preceding section, I have measured the numbers of within-line neighbors obtained for weighted metrics on the *UW English Document Image Database I*, which was released by Haralick et al. in 1993 at the University of Washington [16]. This dataset contains both scans from machine printed journal articles and synthetic images generated from LaTeX sources. From these, I have only used the real scans as provided in the UW-I dataset as binarized images. As many images contained considerable noise like speckles or copy margins, all images have been pre-processed with the following filters:

- All CCs consisting of less than eight pixels were removed. This was necessary to avoid that "pepper noise" from the binarization of halftone figures spoilt the subsequent measurement of the character size.

- The median CC area (*width × height*) was measured as an estimator for the character size m_{area}, and all CCs with an area less than $0.25 \cdot m_{area}$ (representing noise and diacritical signs[4]) and greater than $8 \cdot m_{area}$ (representing fragments from copy margins and figures) were removed.

- Some images only contained figures and almost no text. To filter these out, only images with more than thousand CCs were examined.
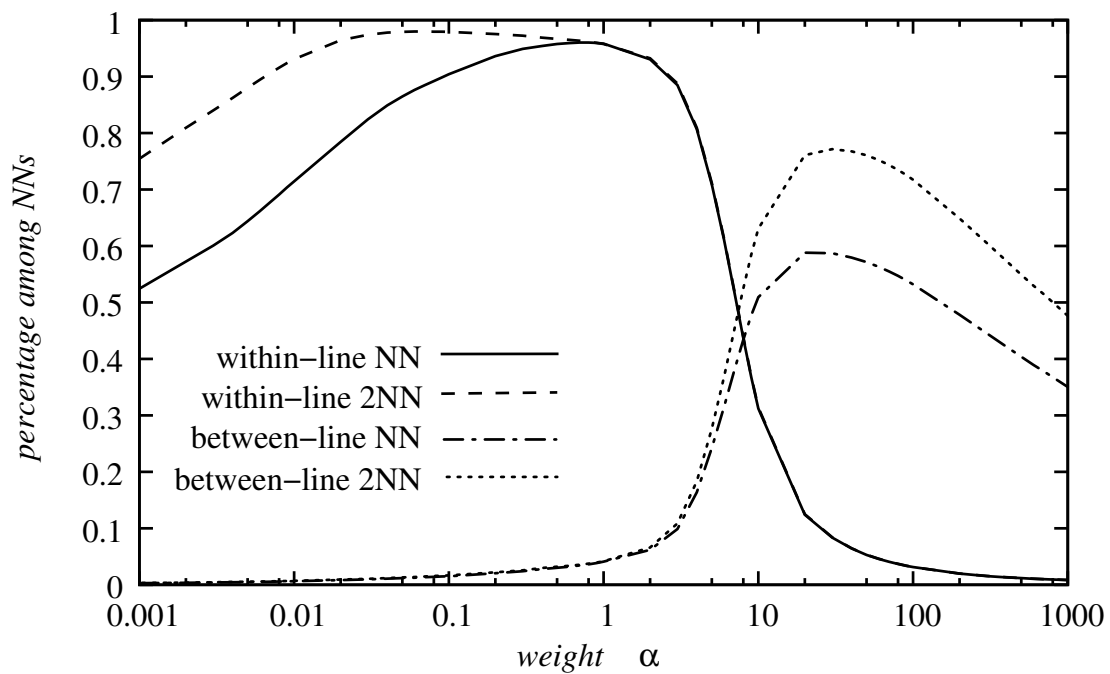
The last criterion sorted out 47 from the 979 images in the UW-I dataset, which left 913 images for the experiment. On each of these images, the following numbers were measured for $p = 2$ (Euclidean metric) and different values of α, ranging between 10^{-3} and 10^{3}:

- the percentage of within-line neighbors among all nearest neighbor pairs with the α-weighted distance (4)

- how many of the α-weighted neighbors actually were the nearest neighbors within the same line, and the same for between-line neighbors, respectively

- the runtime for finding all α-weighted nearest-neighbor pairs, compared to the runtime of finding only within/between-line neighbors with a search predicate

---

[4]The removal of small sized components is a typical step in CC based bottom up layout analysis like O'Gorman's "docstrum" [4]. Even though O'Gorman did not give an actual algorithm for small component removal, he said that it "can be done by peak detection" from the CC size histogram [4].

(a) Percentage of nearest neighbors fulfilling ("within-line") and violating ("between-line") property (3)

(b) Percentage of nearest neighbors matching the actual within-line/between-line nearest neighbor (NN) or one of the two within-line/between-line nearest neighbors (2NN)

Figure 6: Percentage of within-line and between-line nearest neighbors for different values of the weight α in Eq. (4) with $p = 2$.

Fig. 6a shows clearly how small or large values of α suppress neighbors of a specific type, in accordance with the conjecture from section 4. The turnover point occurs for $\alpha \approx 7$, which is greater than one because, for text documents, within-line neighbors tend to be closer than between-line neighbors.

One might draw the conclusion from Fig. 6a that setting α as low as possible is the best for searching within-line nearest neighbors, and vice versa for between-line neighbors. This does however not hold, because the returned α-weighted neighbor needs not to be the actual nearest neighbor under the predicate (3). An example can be seen in Fig. 5: setting $\alpha = 0.1$ favors the neighbor on the left of the test point because it is better vertically aligned than the actual nearest neighbor within the same line (dashed line).

Therefore, I have also measured how many of the α-weighted neighbors actually match the nearest or second nearest neighbor under the within/between-line search predicate. Fig. 6a shows that these numbers decrease when α becomes too small or too large, and that there is an optimum at about $\alpha \in (0.1, 1)$ for within-line neighbors and about $\alpha \in (20, 90)$ for between-line neighbors. This behavior was universal for all documents, except for only 11 images, all of which were either in landscape format, or contained a large number of randomly arranged CCs stemming from figures or halftone images. Hence, it can be concluded that for finding within-line neighbors in text documents, $\alpha \approx 0.5$ is a good choice, and $\alpha \approx 50$ for finding between-line neighbors.

Concerning the runtime, the neighbor search with predicate (3) or its complement was on average about seven times slower than the distance weighted nearest neighbor search. This may be however not necessarily due to a higher runtime complexity, but because the test of the predicate requires calling a separate function on the Python side and by querying CC information. The distance weighting in contrast adds no overhead and does not need to access any external information stored in the connected component data structure. It is likely that the runtime difference can be made smaller when special purpose data structures for storing CC bounding box information are added directly to the kd-tree data structure. Nevertheless, this shows that, in the general implementation of kd-trees in the Gamera framework, a search predicate based on CC dimensions adds considerable runtime overhead compared to a weighted distance approach.

6   Conclusion

The Gamera kd-tree library provides a versatile and easy to use tool for k nearest neighbor searches. The optional search predicate makes it applicable for searching special kind of neighbors, like those within or between lines.

When no strict search predicate is required, like in statistical measurements of rotation angle or page orientation, a weighted distance nearest neighbor search is a faster method for obtaining mostly within- or between-line neighbors. For within-line neighbor searches, the distance weight ratio $\alpha = w_x/w_y$ between the x-weight w_x and the y-weight w_y should be set to $\alpha \approx 0.5$, and for between-line searches to $\alpha \approx 50$ in an Euclidean metric.

Both a conditional nearest neighbor search and a weighted distance nearest neighbor search are made very easy with the Gamera kd-tree API. Hopefully, this paper will help practitioners in the field of document image analysis to make effective use of this new kd-tree library.

References

[1] A. Hashizume, P.S. Yeh, A. Rosenfeld: *A method of detecting the orientation of aligned components.* Pattern Recognition Letters 4, pp. 125-132 (1986)

[2] Y. Lu, C.L. Tan: *Improved Nearest Neighbor Based Approach to Accurate Document Skew Estimation.* 7th International Conference on Document Analysis and Recognition (ICDAR), pp. 503-507 (2003)

[3] A. Simon, J.C. Pret, A.P. Johnson: *A Fast Algorithm for Bottom-Up Document Layout Analysis.* IEEE Transactions on Pattern Analysis and Machine Intelligence 19, pp. 273-277 (1997)

[4] L. O'Gorman: *The Document Spectrum for Page Layout Analysis.* IEEE Transactions on Pattern Analysis and Machine Intelligence 15, pp. 1162-1173 (1993)

[5] J.H. Friedman, F. Baskett, L.J. Shustek: *An algorithm for finding nearest neighbors.* IEEE Transactions on Computers C-24, pp. 1000-1006 (1975)

[6] C.M. Gold: *The meaning of 'neighbour'.* In "Theories and Methods of Spatio-Temporal Reasoning in Geographic Space", pp. 220-235, LNCS 639, Springer (1992)

[7] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf: *Computational Geometry.* Second edition, Springer (2000)

[8] K. Kise, A. Sato, M. Iwata: *Segmentation of page images using the area Voronoi diagram.* Computer Vision and Image Understanding 70, pp. 370-382 (1998)

[9] F. Shafait, D. Keysers, T. Breuel: *Performance Comparison of Six Algorithms for Page Segmentation.* 7th IAPR Workshop on Document Analysis Systems (DAS), pp. 368-379, LNCS 3872, Springer (2006)

[10] J.L. Bentley: *Multidimensional Binary Search Trees Used for Associative Searching.* Communications of the ACM 18, pp. 509-517 (1975)

[11] J.H. Friedman, J.L. Bentley, R.A. Finkel: *An Algorithm for Finding Best Matches in Logarithmic Expected Time.* ACM Transactions on Mathematical Software 3, pp. 209-226 (1977)

[12] A. Beygelzimer, S. Kakade, J. Langford: *Cover Trees for Nearest Neighbors.* 23rd International Conference on Machine Learning, pp. 97-104 (2006)

[13] P.M. Vaidya: *An $O(n \log n)$ Algorithm for the All-Nearest-Neighbor Problem.* Discrete and Computational Geometry 4, pp. 101-115 (1989)

[14] M. Droettboom, K. MacMillan, I. Fujinaga: *The Gamera framework for building custom recognition systems.* Symposium on Document Image Understanding Technologies, pp. 275-286 (2003) (see also `http://gamera.informatik.hsnr.de/`)

[15] B. Stroustrup: *The C++ Programming Language.* Third Edition. Reading, MA: Addison-Wesley. (1997)

[16] I. Guyon, R.M. Haralick, J.J. Hull, I.T. Phillips: *Data Sets for OCR and Document Image Understanding Research.* Handbook of Character Recognition and Document Image Analysis, pp. 779-799, Eds. H. Bunke and P.S.P. Wang, World Scientific (1997)