Distributed Ranking Methods for Geographic Information Retrieval

Marc van Kreveld *Iris Reinbacher*
Avi Arampatzis *Roelof van Zwol*

institute of information and computing sciences, utrecht university
technical report UU-CS-2004-27
www.cs.uu.nl

Distributed Ranking Methods for Geographic Information Retrieval*

Marc van Kreveld
Iris Reinbacher
Avi Arampatzis
Roelof van Zwol

{marc, iris, avgerino, roelof}@cs.uu.nl

Abstract

Geographic Information Retrieval is concerned with retrieving documents in response to a spatially related query. This paper addresses the ranking of documents by both textual and spatial relevance. To this end, we introduce *distributed ranking*, where similar documents are ranked spread in the list instead of consecutively. The effect of this is that documents close together in the ranked list have less redundant information. We present various ranking methods, efficient algorithms to implement them, and experiments to show the outcome of the methods.

1 Introduction

The most common way to return a set of documents obtained from a Web query is by a ranked list. The search engine attempts to determine which document seems to be the most relevant to the user and will put it first in the list. In short, every document receives a *score*, or *distance to the query*, and the returned documents are sorted by this score or distance.

There are situations where the sorting by score may not be the most useful one. When a more complex query is done, composed of more than one query term or aspect, documents can also be returned with two or more scores instead of one. This is particularly useful in *geographic information retrieval* [13, 17, 21]. For example, the Web search could be for campings in the neighborhood of Neuschwanstein, and the documents returned ideally have a score for the query term "camping" and a score for the proximity to Neuschwanstein. This implies that a Web document resulting from this query can be mapped to a point in the 2-dimensional plane, where both axes represent a score. In Figure 1, the map (taken from [1]) indicates campings near the castle Neuschwanstein, which is situated close to Schwangau, with the distance to the castle on the x-axis and the rating given by the ANWB on the y-axis.

Another query is for example "castles near Koblenz". When mapping to the plane, a cluster of points could be several documents about the same castle. If this castle is in the immediate vicinity of Koblenz, all of these documents would be ranked high, provided that they also have a high score on the term "castle". However, the user probably also wants documents about other castles that may be a bit further away, especially when these documents are more relevant for the term "castle". To incorporate this idea in the ranking, we introduce *distributed ranking* in this paper. We present various models that generate ranked lists where closely ranked documents are dissimilar. We also present efficient algorithms that compute the distributed rankings, which is important to keep server load low.

Distributed ranking requires a spatial score and a term score. Term scores are standard in information retrieval and these methods can be used without adaptation. Defining appropriate spatial scores is a different problem; there are several papers discussing this issue [2, 15, 20]. For

*This research is supported by the EU-IST Project No. IST-2001-35047 (SPIRIT).

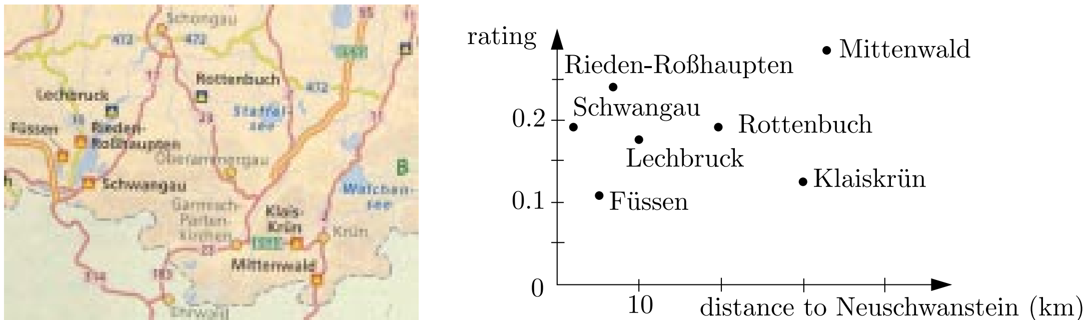

Figure 1: Campings near Neuschwanstein mapped to points in the plane.

this paper it is sufficient to know that a spatial score can be defined and we assume that it is given.

We can also use distributed ranking to obtain *geographically distributed* ranked lists of documents. In this case we use the geographic locations associated with Web documents explicitly in two coordinates. If we also also have a term score, each document has three values (scores) that can be used for ranking the relevant documents. Using our example, two documents referring to two castles at the same distance from Koblenz, but in opposite directions, may now be ranked consecutively. The geographically distributed ranking problem is similar to the so-called *settlement selection problem*, which appears in map generalization [14, 19].

There are several other reasons to rank documents according to more than one score. For example we could distinguish between the scores of two textual terms, or a textual term and a spatial term, or a textual term and metadata information, and so on. A common example of metadata for a web document is the number of hyperlinks that link to it; a document is probably more relevant when more links point to it. In all of these cases we get two or more scores which need to be combined for the ranking.

In traditional information retrieval, the separate scores of each document would be combined into a single score (e.g., by a weighted sum or product) which produces the ranked list by sorting. Besides the problem that it is unclear how the scores should be combined, it also makes a distributed ranking impossible. Two documents with the same combined score could be similar documents or quite different. If two documents have two or more scores that are the same, one has more reason to suspect that the documents themselves are similar than when two documents have one (combined) score that is the same.

The topic of geographic information retrieval is studied, for example, in the SPIRIT project [13]. The idea is to build a search engine that has spatial intelligence since it will understand spatial relationships like *close to*, *North of*, *adjacent to*, and *inside*, for example. The core search engine will process a user query in such a way that both the textual and the spatial relevance of a document are obtained in a score. This is possible because the search engine will not only have a term index, but also a spatial index. Those two indices provide the two scores that are needed to obtain a distributed ranking. The ranking study presented here is part of the geographic search engine under development for the SPIRIT project. Several of the relevance ranking methods discussed in this paper are available in the prototype of the spatial search engine. Another project dealing with geographic information retrieval is the BUSTER project [21].

Related research has been conducted in [17], which focuses on disambiguating geographic terms of a user query. The disambiguation of the geographic location is done by combining textual information, spatial patterns of other geographic references, relative geographic references from the document itself, and population heuristics from a gazetteer. This gives the final value for the geoconfidence. The georelevance is composed of the geoconfidence and the emphasis of the place name in the document. The textual relevance of a document is computed as usual in information retrieval. Once both textual and geographic relevance are computed, they are combined by a

weighted sum.

Finding relevant information and at the same time trying to avoid redundancy has so far mainly been addressed in producing summaries of one or more documents. Carbonell and Goldstein use in [4] the maximal marginal relevance (MMR), which is a linear combination of the relevance of the document to the user query and its independence of already selected documents. MMR is used for the reordering of documents. A user study has been performed, showing that users preferred MMR ranking to the usual ranking of documents. The paper [4] contains no algorithm how to (efficiently) compute the MMR. Following up on this, a Novelty Track of TREC [10] discusses experimenting with rankings of textual documents such that every next document has as much additional information as possible.

Goldstein et al. propose in [8] another scoring function for summarizing text documents. Every sentence is assigned a score combined of the occurrence of statistical and linguistic features. They are combined linearly with a weighting function. In [9] MMR is refined and used to summarize multiple documents. Instead of full documents, different passages or sentences are assigned a score.

The remainder of this paper is organized as follows. In Sections 2 and 3 we introduce several different basic ranking methods and efficient algorithms to compute them. In Section 4 we present several extensions of the basic ranking methods. In Section 5 we show how the different ranking methods behave on real-world data.

2 Basic Distributed Ranking Methods and Algorithms

In this section we present two basic distributed ranking methods. Like in traditional information retrieval, we want the most relevant documents to appear high in the ranking, while at the same time avoiding that documents with similar information appear close to documents already ranked. We will focus on the two dimensional case only, although in principle the ideas and formulas apply in higher dimensions too. We will discuss extensions to higher dimensions explicitly in Section 4.3.

We assume that a Web query has been conducted and a number of relevant documents were found. Each document is associated with two scores, for example a textual and a spatial score (which is the case in the SPIRIT search engine). The relevant documents and the query are mapped to points in the plane. We perform the mapping in such a way that the query is a point Q at the origin, and the documents are mapped to a set of points $P = \{p_1, \ldots, p_n\}$ in the upper right quadrant, such that the documents with high scores are the points close to Q. We can now formulate the two main objectives for our ranking procedure:

1. **Proximity to query:** Points close to the query Q are favored.

2. **High spreading:** Points farther away from already ranked points are favored.

A ranking that simply sorts all points in the representation plane by distance to Q is optimal with respect to the first objective. However, it can perform badly with respect to the second. Selecting a highly distributed subset of points is good with respect to the second objective, but the ranked list would contain too many documents with little relevance early in the list. We therefore seek a compromise where both criteria are considered simultaneously. Note that the use of a weighted sum to combine the two scores into one as in [17] makes it impossible to obtain a distributed ranking.

The point with the smallest Euclidean distance to the query is considered the most relevant and is always first in any ranking. The remaining points are ranked with respect to already ranked points. At any moment during the ranking, we have a subset $R \subset P$ of points that have already been ranked, and a subset $U \subset P$ of points that are not ranked yet. We choose from U the "best" point to rank next, where "best" is determined by a *scoring function* that depends on both the distance to the query Q and the set R of ranked points. Intuitively, an unranked point has a higher added value or relevance if it is not close to any ranked points.

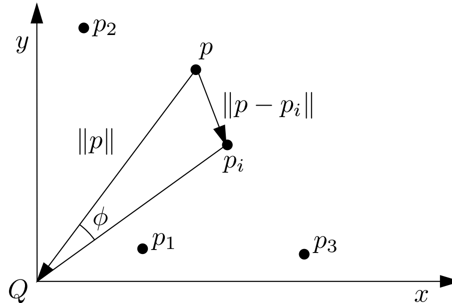

Figure 2: An unranked point p amidst ranked points p_1, p_2, p_3, p_i, where p is closest to p_i by distance and by angle.

For every unranked point p, we consider only the closest point $p_i \in R$, where closeness is measured either in the Euclidean sense, or by angle with respect to the query point Q. This is illustrated by $\|p - p_i\|$ and ϕ, respectively, in Figure 2. Using the angle to evaluate the similarity of p and p_i seems less precise than using the Euclidean distance, but it allows more efficient algorithms, and certain extensions of angle-based ranking methods give well-distributed results. See the experiments in Section 5 for this. We will first present the basic models, and then give the algorithms in Section 2.3.

2.1 Distance to query and angle to ranked

Our first ranking method uses the angle measure to obtain the similarity between an unranked and a ranked point. In the triangle $\triangle pQp_i$ (see Figure 2) consider the angle $\phi = \phi(p, p_i)$ and rank according to the score $S(p, R) \in [0, 1]$, which can be derived from the following normalized equation:

$$S(p, R) = \min_{p_i \in R} \left(\frac{2(\phi(p, p_i) + c)}{\pi + 2c} \cdot \left(\frac{1}{1 + \|p\|} \right)^k \right) \tag{1}$$

Here, $k > 0$ denotes a constant. If k is small, the emphasis lies on the distribution, and if k is large, we give a large importance to the proximity to the query. The additive constant $c > 0$ ensures that all unranked points $p \in U$ are assigned an angle dependent factor greater than 0. This is necessary if several points lie on the same halfline originating in Q. The score $S(p, R)$ necessarily lies between 0 and 1, and is appropriate if we do not have a natural upper bound on the maximum distance of unranked points to the query. If such an upper bound were available, there are other formulas that give normalized scores.

2.2 Distance to query and distance to ranked

In the previous section we ranked by angle to the closest ranked point. It may be more natural to consider the Euclidean distance to the closest ranked point instead. In Figure 2, consider the distance $\|p - p_i\|$ from p to the closest ranked point p_i and rank according to the outcome of the following equation:

$$S(p, R) = \min_{p_i \in R} \left(\frac{\|p - p_i\|}{\|p\|^2} \right) \tag{2}$$

The denominator needs a squaring of $\|p\|$ (or another power > 1) to assure that documents far from Q do not appear too early in the ranking, which would conflict with the proximity to query requirement. A normalized equation such that $S(p, R) \in [0, 1]$ is the following:

$$S(p, R) = \min_{p_i \in R} \left((1 - \mathrm{e}^{-\lambda \cdot \|p - p_i\|}) \cdot \frac{1}{1 + \|p\|} \right) \tag{3}$$

Here, $\lambda > 0$ is a constant that defines the base e^λ of the exponential function.

2.3 Basic algorithms

For both methods described above, during the ranking algorithms we always choose the one unranked point p that has the highest score $S(p, R)$ and rank it next. This implies an addition to the set R and hence, recomputation of the scores of the other unranked points may be necessary.

In this section we describe two simple algorithms that can be applied for the basic methods as well as for the other methods we present in the next section.

The most straightforward algorithm one can think of takes $O(n^3)$ time. By keeping the closest ranked point for each unranked point in evidence, we can create a very simple, generic algorithm which has a running time of $O(n^2)$. The distance referred to in steps 2(a) and 4 can be interpreted either as angle or as Euclidean distance in the plane.

Algorithm 1: Given: A set P with n points in the plane.

1. Rank the point r closest to the query Q first. Add it to R and delete it from P.
2. For every unranked point $p \in P$ do
   (a) Store with p the point $r \in R$ as the closest point
   (b) Compute the score $S(p, R) = S(p, r)$ and store it with p
3. Determine and choose the point p with the highest score $S(p, R)$ to be next in the ranking; add it to R and delete it from P.
4. Compute for every point $p' \in P$ the distance to the last ranked point p. If it is smaller than the distance to the point stored with p', then store p with p' and update the score $S(p', R)$.
5. Continue at step 3 if there are still unranked points.

The first four steps of this algorithm all take linear time. As we need to repeat steps 3 and 4 until all points are ranked, the overall running time of this algorithm is $O(n^2)$. If we are only interested in the top 10 documents of the ranking, we only need linear time for the computation. More generally, the top t documents are determined in $O(tn)$ time.

The next algorithm is a variation of the first, and computes the same ranking. It uses a Voronoi diagram to find the closest unranked points to a ranked point p. Its worst case running time is the same as of Algorithm 1, namely $O(n^2)$; however, a typical case analysis shows that it generally runs in $O(n \log n)$ time in practice.

Algorithm 2: Given: A set P with n points in the plane.

1. Rank the point r closest to the query Q first. Add it to R and delete it from P. Initialize a list with all unranked points and store it with r. Determine the point p in this list with the highest score and insert it in an initially empty priority queue H.
2. Choose the point p with the best overall score from the priority queue H as next in the ranking; add it to R, delete it from P, and update the structures:
   (a) Delete p from the priority queue H.
   (b) Insert p as a new site in the Voronoi diagram of R.
   (c) Create for the newly created Voronoi cell $V(p)$ a list of unranked points that lie in $V(p)$ by traversing the list of each ranked point p' whose Voronoi cell $V(p')$ neighbors $V(p)$, removing the points from it that are closer to p than to p', and adding these points to the list of p.

(d) Compute the point with the best score for the newly created Voronoi cell $V(p)$ and insert it in a priority queue H. For all Voronoi cells whose lists changed, recompute the unranked point with the best score and update the priority queue H accordingly.

3. Continue at step 2 if the priority queue is non-empty.

For the efficiency analysis, assume first that we used the distance to ranked version for assigning scores. Then the Voronoi diagram is the usual planar subdivision, and the average degree of a Voronoi cell is almost six. One can expect that a typical addition of a point p to the ranked points involves a set of neighbors $R' \subseteq R$ with not many more than six ranked points. If we also assume that, typically, a point in R' loses a constant fraction of the unranked points in its list, we can prove an $O(n \log n)$ time bound for the whole ranking algorithm. The analysis is the same as in [11, 19]. In the angle to ranked version of assigning scores, the set of neighbors R' will have only two ranked points.

We conclude with the following theorem:

Theorem 1 *A set of n points in the plane can be ranked according to the basic distributed models in $O(n^2)$ time in the worst case. By maintaining a Voronoi diagram of the closest points, the points can be ranked in $O(n \log n)$ time under certain natural assumptions.*

3 Other Distributed Ranking Methods and Algorithms

In this section we present more basic ranking methods, namely addition methods and a wavefront approach. The addition methods are simple variations of the ideas of the methods from Section 2. Instead of dividing the angle (or distance) of a point p to the closest ranked point by the distance of p to the query, we add those two values (one of them inversed). We can adapt both algorithms given above. However, as we will see in Section 3.1, for one of the methods we can give an algorithm with better running time. The wavefront method is in a way an inversion of the ideas of the basic methods. In the last section, we ranked the point with the highest score according to some scoring function next. For a fixed set of ranked points, the subset of the plane with the same score is a set consisting of curved pieces. When decreasing the score, the curved pieces move and change shape. The next point to be ranked is the one encountered first by these curves when decreasing the score. In Section 3.2 we will use this process instead of a scoring function to define a ranking: we predefine the shape of a wavefront and move it until it hits a point, which will be ranked next. As before, we first introduce the methods, and then present the algorithms in Section 3.3.

3.1 Addition methods

So far, both our distributed methods were based on a scoring function that essentially is a division of the angle or distance to the closest ranked point by the distance to the query. In this way, points closer to the query get a higher relevance. We can obtain a similar effect, but a different ranking, by adding up two terms, obtained from the angle or distance to closest ranked point, and the distance to the query. The term depending on the distance to the query should be such that a larger distance gives a lower score. Although it is unusual to add angles and distances, it is not clear beforehand which method will be more satisfactory for users, so we analyse these methods as well. If the results are satisfactory, this method may be the one of choice, since it allows a very efficient algorithm.

$$S(p, R) = \min_{p_i \in R} \left(\alpha \cdot (1 - \mathrm{e}^{-\lambda \cdot (\|p\| / \|p_{max}\|)}) + (1 - \alpha) \cdot \phi(p, p_i) \cdot \frac{2}{\pi} \right) \quad (4)$$

In this equation, p_{max} is the point with maximum distance to the query, $\alpha \in [0, 1]$ denotes a variable which is used to put an emphasis on either distance or angle, and λ is a constant that defines the base of the exponential function.

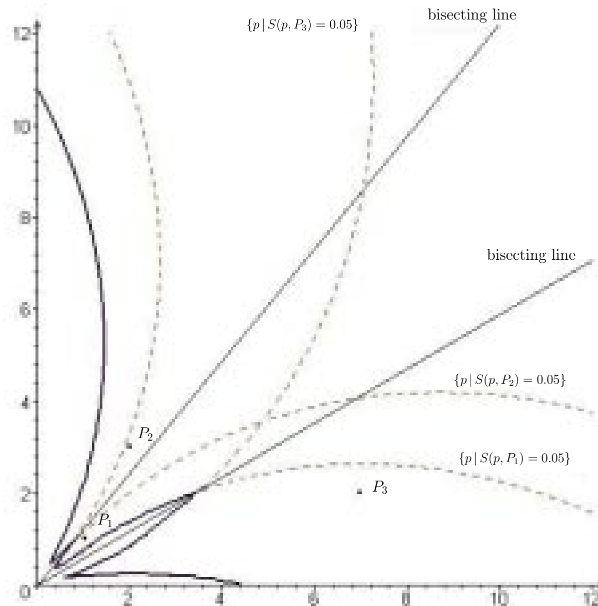

Figure 3: Wavefront of the angle method of Section 2.1. The solid line is the locus of all points p, such that the score $S(p, \{p_1, p_1, p_3\}) = a, \quad a = 0.05$. The wavefront moves when a is increased.

Another, similar, addition method adds the distance to the query and the distance to the closest ranked point:

$$S(p, R) = \min_{p_i \in R} \left(\alpha \cdot (1 - \mathrm{e}^{-\lambda_1 \cdot (\|p\| / \|p_{max}\|)}) + (1 - \alpha)(1 - \mathrm{e}^{-\lambda_2 \cdot \|p - p_i\|}) \right) \tag{5}$$

Again, p_{max} is the point with maximum distance to the query, $\alpha \in [0, 1]$ is a variable used to influence the weight given to the distance to the query (proximity to query) or to the distance to the closest point in the ranking (high spreading), and λ_1 and λ_2 are constants that define the base of the exponential functions.

3.2 The wavefront approach

The ranking methods presented so far can also be described visually. For example, consider the distance to query and angle to ranked method of Section 2.1. At any moment in the algorithm, some selection has been made, and the next point for the ranking must be found. This point is the first point hit by a *wavefront* of a particular shape, as shown in Figure 3. From the shape of the wavefront it is clear that points that are not closest to the query could easily be chosen next, if the angle of the vector p (with the positive x-axis) is not close to the angle of p_i for any previously ranked point p_i. This illustrates the desired behavior of trying to choose a point next that is different from already ranked points. We can derive new ranking methods by specifying the shape of the wavefront instead of a scoring function. At the same time, the idea gives rise to other ways to compute the ranking.

Figure 3 depicts the positions in space where the score values are the same, namely 0.05, for the three ranked reference points $p_1 = (1, 1)$, $p_2 = (2, 3)$, and $p_3 = (7, 2)$. The functions themselves are $\arccos(\text{dist to closest}/\text{dist to query})/\text{dist to query}$ (constants omitted). They can easily be derived from Equation 1. Note that the curves are symmetric with respect to the line starting at the origin and passing through the ranked point they correspond to, and that the intersection with this line lies on a circle with the same radius r for all three curves. Furthermore, two curves of neighboring sectors meet exactly at the bisecting lines between the ranked points.

3.3 Efficient algorithms for addition and wavefront model

For both methods described above, Algorithms 1 and 2 can be applied, which gives a worst case running time of $O(n^2)$ in both cases. But in fact, we can seriously improve the worst case running time with a different algorithm. As the angle $\phi(p, p_i)$ is an additive and not a multiplicative part of the score equation, we can give an algorithm with a worst case running time of $O(n \log n)$ for the angle-distance addition method. We will also present an algorithm for the piecewise linear wavefront method which computes such a ranking in $O(n \log^2 n)$ time in the worst case.

3.3.1 The angle-distance addition algorithm

To initialize for the ranking algorithm, we select the point r from P that is closest to the query and rank it as the first point. Then we build two augmented binary search trees, one for the subset of points$P_0 \subseteq P\backslash\{r\}$ that are below the line through Q and r and one for the subset $P_1 = P\backslash\{r\} - P_0$ of points above or on this line.

The point set P_0 is stored in the leaves of a binary tree T_0, sorted by counterclockwise (ccw) angle to the y-axis. In every leaf of the tree we also store: (i) ccw and clockwise (cw) angle to r and to the x-axis respectively; (ii) the distance to the query; (iii) ccw and cw score, where the angles used are taken from (i). We augment T_0 as follows (see e.g. [5]) for augmenting data structures): In every internal node we store the best cw and the best ccw score per subtree. We additionally store for the whole tree T_0 that its closest ranked point, r, is counterclockwise, and correction values, one cw and one ccw, to be used later. They are additive corrections for all cw and ccw scores of points in the whole tree. Note that the augmentation allows us to descend in a tree towards the point with the best score along a single path; at any node we simply proceed in the subtree that contains the point with the higher score (cw or ccw score, depending on the information stored with the tree).

The point set P_1 is stored in the same way in a tree T_1, with the following differences. The cw angle used in the leaves is to r and the ccw angle is taken to the y-axis, and we store for the whole tree that its closest ranked point, r, is clockwise. Finally, we initialize a priority queue with the point from T_0 with the best score and the point from T_1 with the best score.

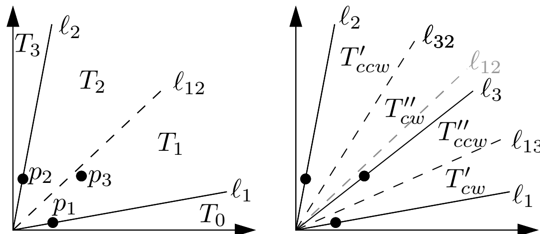

Figure 4: The split and concatenate of trees in Algorithm 3.

During the algorithm, for every ranked point two trees are present. It is easy to see that all trees T_{2i+1}, $i = 0 \ldots m-1$ have their closest ranked point in cw direction, whereas all trees T_{2i}, $i = 0 \ldots m-1$, have their closest ranked point in ccw direction. As shown left in Figure 4, between two already ranked points p_1 and p_2, indicated by ℓ_1 and ℓ_2, there are two binary trees, T_1 cw and T_2 ccw of the bisecting line ℓ_{12}. All the points in T_1 are closer in angle to p_1 and all the points in T_2 are closer in angle to p_2. If we insert a new point p_3 to the ranking, this means we insert a new imaginary line ℓ_3 through p_3 and we need to perform the following operations on the trees:

1. Split T_1 and T_2 at the bisecting lines ℓ_{32} and ℓ_{13}, creating the new trees T'_{cw} and T'_{ccw} and two intermediate trees $\overline{T}_{cw}$ and $\overline{T}_{ccw}$
2. Concatenate the intermediate trees from (1), creating one tree $\overline{T}$.
3. Split $\overline{T}$ at the newly ranked point p_3, creating T''_{cw} and T''_{ccw}.

Figure 4 right, shows the outcome of these operations. Whenever we split or concatenate the binary trees we need to make sure that the augmentation remains correct. In our case, this is no problem, as we only store the best initial scores in the inner leaves. However, we need to update the information in the root of each tree about the closest cw and ccw ranked point and the correction values. The former is easy to update. For the correction values, note that all scores for points in the same tree are calculated with respect to the same ranked point and they change with the same additive amount in the addition model. Therefore, we should simply subtract $(1-\alpha)\cdot\phi'\cdot 2/\pi$ from the scores, where ϕ' denotes the angle between the previously closest ranked point cw (ccw)and the newly closest ranked point cw (ccw, respectively). For efficiency reasons, we do not do this explicitly but in once using the cw (or ccw) correction value. So we subtract $(1-\alpha)\cdot\phi'\cdot 2/\pi$ from the appropriate correction value. If there is no previously closest ranked point cw or ccw (first and last trees), then we take the angle between the newly ranked point and the x-axis or y-axis instead.

Furthermore we need to update the score information in the priority queue. This involves deleting scores that are no longer valid, and inserting scores that are new best scores in a tree. Now we can formulate an algorithm for the addition method that runs in optimal $O(n \log n)$ time.

Algorithm 3: Given: A set P with n points in the plane.

1. Determine the closest point r and remove it from P, split $P\backslash\{r\}$ into sets P_0 and P_1 as described, and initialize the augmented trees T_0 and T_1. Determine the points in T_0 and T_1 with the best score an store them in a priority queue H.

2. Choose the point p with the highest score as next in the ranking by deleting the best one from the priority queue H.

3. For every last ranked point p do:

   (a) Split and concatenate the binary trees as described above and update the information in their roots.

   (b) Update the best score information in the priority queue H:

      i. Delete the best score of the old tree T_1 or T_2 that did not contain p.

      ii. Find the four best scores of the new trees $T'_{cw}, T'_{ccw}, T''_{cw}$, and T''_{ccw} and insert them in the priority queue H.

4. Continue at step 2 if the priority queue is non-empty.

The initialization (step 1) takes $O(n \log n)$ time. Step 2 takes $O(\log n)$ time for each execution. In step 3, the split and concatenate of at most four binary trees, takes $O(\log n)$ time for each tree. Updating the best score information in the priority queue also takes $O(\log n)$ time. So, overall, the algorithm takes $O(n \log n)$ time in the worst case.

Note that this algorithm is not applicable for the second addition method, where we add up the distance to closest and the distance to the query. This is easy to see, since the distance to the closest ranked point does not change by the same amount for a group of points. This implies that the score for every unranked point needs to be adjusted individually when adding a point to R, which is done by the two basic algorithms.

Theorem 2 *A set of n points in the plane can be ranked according to the angle-distance addition method in $O(n \log n)$ time.*

3.3.2 The wavefront algorithm

The linear wavefront method generates a ranking as follows (see Figure 5). Assume a set P of n points in the plane is given, and also an angle ρ which, intuitively, captures how much preference should be given to distribution. It is the angle between a segment of the wavefront and the nearest

(by angle) line through Q and a ranked point. Let $p_1, \ldots, p_i$ be the points ranked so far. Draw lines $\ell_1, \ldots, \ell_i$ where ℓ_j passes through Q and p_j, where $1 \leq j \leq i$ and remove duplicates, if any. Assume without loss of generality that the lines $\ell_1, \ldots, \ell_i$ are sorted by slope in non-increasing order (steepest first). Consider a circle C_s with radius s centered at Q. The wavefront for $p_1, \ldots, p_i$ and s (Figure 5) is defined as the polygonal line $v_1, v_2, \ldots, v_{2i+1}$, where v_{2j} is the intersection point of ℓ_j and C_s for $1 \leq j \leq i$, and the edges $v_{2j-1}v_{2j}$ and $v_{2j}v_{2j+1}$ make an angle ρ with the line ℓ_j. This defines the position of the vertices $v_3, v_5, \ldots, v_{2j-1}$. Vertex v_1 is on the y-axis such that v_1v_2 includes an angle ρ with ℓ_1, and vertex v_{2j+1} is on the x-axis such that $v_{2j}v_{2j+1}$ includes an angle ρ with ℓ_i. When s is increased, the segments of the wavefront move away from Q and they get longer, but they keep their orientation.

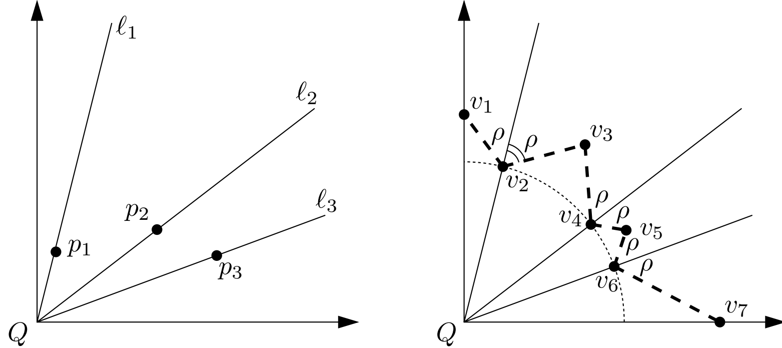

Figure 5: Illustration of the linear wavefront method.

Algorithm 4: The first point in the ranking is the point of P that is closest to the query Q. The next point p in the ranking is determined as follows. Start with $s = 0$ and increase s continuously until some segment of the linear wavefront hits the first non-ranked point. This point p is ranked next. Then restart (or continue) the growing of s with the new wavefront, which has two more vertices and two more segments.

To implement this ranking scheme efficiently we use a geometric data structure that allows us to find the next point p in $O(\log^2 n)$ time. Since we will perform $n-1$ queries, the total time to rank will be $O(n \log^2 n)$. The data structure to be designed stores all unranked points and must be able to answer queries SegmentDrag(ϕ, ψ, ρ), where the first point hit by a moving and growing line segment must be found. The moving and growing line segment is specified by the three angles ϕ, ψ, and ρ, and is defined by two consecutive vertices of the wavefront. Let ℓ_ϕ and ℓ_ψ be the lines through Q that have angle ϕ, respectively ψ with the positive x-axis. The line segment that is moved and grows has its endpoints on ℓ_ϕ and ℓ_ψ, and makes an angle ρ with ℓ_ϕ (if the endpoint on ℓ_ϕ is closer to Q than the endpoint on ℓ_ψ, or with ℓ_ψ otherwise).

Assume that all points of P lie between the lines ℓ_ϕ and ℓ_ψ. Then the query problem reduces to querying with a full line that lies outside the convex hull of P and is translated towards P. The data structures of Hershberger and Suri [12] and Brodal and Jacob [3] have linear size, $O(\log n)$ deletion time, and answer queries in $O(\log n)$ time amortized. These data structures are dynamic convex hull query structures.

For a data structure that can answer SegmentDrag(ϕ, ψ, ρ), we store the set P sorted by angle in the leaves of a binary search tree T. For an internal node ν, denote by T_ν the subtree of T rooted at ν, and by $P_\nu \subseteq P$, the subset of points that are stored in the leaves of T_ν. For any internal node ν, we store a pointer to an associated structure T'_ν which stores the points P_ν in a data structure for dynamic convex hull queries [3, 12]. The technique of extending a data structure by adding a one-dimensional range restriction (here on angle) is quite standard [16, 22] and used for instance for orthogonal range queries [6]. The storage requirements, preprocessing time, query

time, and update time all increase by a factor of $O(\log n)$. This implies that a SegmentDrag query can be answered in $O(\log^2 n)$ time with a data structure that has size $O(n \log n)$, construction time $O(n \log^2 n)$, and deletion time $O(\log^2 n)$.

To obtain an $O(n \log^2 n)$ time algorithm for ranking in the linear wavefront method, we maintain a set of candidate points to be ranked next in a priority queue. We store one point for each sector defined by lines ℓ_j and ℓ_{j+1}, where ℓ_j (and ℓ_{j+1}) is the line through Q and the point v_j (resp. v_{j+1}) on the linear wavefront, where $1 \leq j \leq 2i$. The point we store in a sector is the one that is hit for the lowest value of s, the radius of the circle that defines the wavefront. The overall best candidate, and hence the point to be ranked next, is the one with the lowest value of s among all candidates. Since the candidates are stored in a priority queue H, the best one can be found and extracted in $O(\log n)$ time. After selecting and ranking a point p we must update the priority queue H and the data structure for SegmentDrag queries. The first update is easy; it is the deletion of point p. To update H further, we must first delete the best candidate in every sector that has changed due to the ranking of p. There can be at most two such sectors, so we delete two points from the priority queue. Then we must find the new best candidate in every new sector arising due to the ranking of p. There are at most four new sectors, and we find the new candidates by four SegmentDrag queries. These new candidates are then inserted in the priority queue H, which prepares it for ranking the next point.

We can conclude with the following theorem:

Theorem 3 *A set of n points in the plane can be ranked according to the linear wavefront method in $O(n \log^2 n)$ time.*

4 Extensions of the Basic Ranking Methods

In this section we present extensions of the basic ranking methods introduced in Sections 2 and 3. We divided the given point set P into the set of already ranked points R and the set of unranked points U during the algorithms. We chose the candidates to be ranked next from all points in U and computed the score functions with respect to all ranked points in R. In this section we present two extensions where this is not the case anymore. The staircase enforcement extension limits the set of candidates to be ranked next to the points in U that lie on the staircase of the point set P. The limited windows method limits the set of ranked reference points to be used for the score computation to only the last k ranked points. Furthermore, we will describe the necessary adaptations of the algorithms in higher dimensions.

4.1 Staircase enforcement

In the basic methods, every unranked point was eligible to be next in the ranking, if it had the highest score. This can lead to a ranking where a point is ranked before other points that are better in both aspects. Often this is an undesirable outcome. As an alternative, we choose the candidates to be ranked next only from the points p that lie on the staircase of the point set. A point p is on the lower left staircase of the point set P if and only if for all $p' \in P \setminus \{p\}$, we have $p_x < p'_x$, or $p_y < p'_y$, or $p_x = p'_x$ and $p_y = p'_y$. If we always select from the staircase of unranked points, we automatically obtain the property that any ranked document is more relevant in at least one aspect than all documents that are ranked later.

We can easily adapt the basic ranking algorithms for staircase enforcement. We consider as eligible candidates for next in the ranking only the points on the staircase of unranked points, which we maintain throughout the algorithm. When a point is ranked, the staircase has to be updated, and points that are new on it become eligible too. Computing the staircase of a point set with n points takes $O(n \log n)$ time, insertion or deletion of one point takes $O(\log n)$ time per point. In every update of the staircase, we delete one point from it. There can be as many as a linear number of points that need to be inserted in one update, so a single update can have a running time of $O(n \log n)$. However, every point of P is inserted to the staircase and deleted from

it exactly once. This means that maintaining the staircase takes only $O(n \log n)$ time during the entire execution of the algorithm.

In the following, a brief description of the necessary changes to the basic algorithms is given.

In the generic Algorithm 1, which can be used for all models, we compute the staircase at the beginning, and then we only need to change step 3 as follows:

3. Update the staircase of P and choose from the points on the staircase the one with highest score $S(p, R)$ as next in the ranking; add it to R and delete it from P and from the staircase.

Computing the staircase in the beginning takes $O(n \log n)$ time. We only need to do this once. Updating the staircase in step 3 takes $O(\log n)$ time per insertion and deletion, and the other computations can be performed in constant time. As this step needs to be repeated n times, it takes $O(n \log n)$ time overall, and hence, staircase enforcement does not influence the total running time of $O(n^2)$.

In Algorithm 2 for the angle and distance models, we maintain the Voronoi diagram of all ranked points, and for each Voronoi cell, a list of all unranked points in that cell. The main adaptation is that only points on the staircase of unranked points are stored in the priority queue. That is, for every list of a Voronoi cell the point with highest score and that lies on the staircase will be in the priority queue. We must also maintain the staircase of unranked points throughout the algorithm to have this information. Neither the worst case nor the typical time analysis change in this case, so the worst case time bound is still $O(n^2)$, and the typical running time remains $O(n \log n)$.

For Algorithm 3 we change the algorithm as follows: We start by computing the staircase of the point set $P \backslash \{r\}$ and create the initial, augmented trees T_0 and T_1 only for the points from the staircase. The priority queue is initialized with the best points from T_0 and T_1. We rank the next point p, update the corresponding trees and the priority queue as before, and delete p from the staircase. Step 4 needs to be modified as follows:

4. Update the staircase to incorporate the deletion of p. Insert the new points on the staircase into the binary trees while keeping the augmentation correct. Update the best-score information in the priority queue. Continue with step 2 if the priority queue is non-empty.

The new version of step 4 takes $O(n \log n)$ time overall, so the total running time of $O(n \log n)$ is not affected.

Algorithm 4 for the wavefront model is also easy to adapt. In the original model, we move a line segment towards the convex hull of the whole point set P and rank the first point that is hit next. Here, the point set is restricted to the points of P that lie on the staircase of P. We do not need to change any of the data structures described in Section 3, but we need to precompute the staircase once ($O(n \log n)$ time) and maintain it through the algorithm ($O(\log n)$ time per insertion and deletion). This leaves the asymptotic running time of the wavefront algorithm unchanged.

4.2 Limited windows

In all the previously presented basic ranking methods as well as in the staircase enforced extensions, the closest point from the set R of ranked points is used to determine the score of an unranked point. However, if the second best point lies very close to the best point, it should not be late in the ranking just because it is very similar to the best point. We would like the influence of a ranked point to stop after, say, another ten points have been ranked after it. This is captured in the limited windows extension, where only the k latest ranked points are kept in evidence for the future ranking. The value of k is fixed throughout the algorithm.

The general idea is as follows. We store the set of ranked points that we consider in a queue W, which we will call the window. The basic algorithms remain exactly the same as long as the

size of W does not exceed k. When the $(k+1)$-th point is ranked, we add it at the tail of the queue and delete the point p at the head of it. For all unranked points that had the deleted point p as their closest, we need to find the new closest point among the k ranked points in W and recompute their score.

We give a brief description of the necessary adaptations of the basic algorithms in the next paragraphs. An adaptation of the staircase enforced methods to get a combination of the two extensions is straightforward and therefore not given here. We assume that k is fixed.

The adaptation of Algorithm 1 is straightforward. We now determine in step 4 for all unranked points p' the smallest distance to the k last ranked points in R to determine the score of each unranked point. This clearly takes $O(kn)$ time for each next ranked point, so the overall running time of the algorithm is $O(kn^2)$. If we allow $O(kn)$ extra storage, we can store with every unranked point all k last ranked points sorted by distance. We can keep the sequence of all k points sorted in $O(n^2 \log k)$ time in total.

Algorithm 2 for the angle or distance model is adapted as follows. We keep all ranked points in a queue W. When W contains more than k points, we delete the first point p from the queue and from the Voronoi diagram of R. All unranked points that lie in the Voronoi cell of p need to be redistributed to the at most k neighboring cells, which means that their lists of unranked points need to be updated. Furthermore, we need to recompute the highest score value from these k lists and update the priority queue H accordingly.

The worst case time analysis for deleting one point from the window is as follows. The operations on W only take constant time each. Deleting the point p from the Voronoi diagram can be done in $O(k)$ time. There can be a linear number of unranked points that need to be redistributed, which takes $O(kn)$ time in the worst case. Updating the k lists can be done in $O(n)$ time in total and updating the heap takes $O(k \log k)$ time. This leads to an overall worst case running time of $O(kn^2)$. This can be improved to $O(n^2 \log k)$ time by doing the redistribution more cleverly.

In a typical time analysis we can delete a point p from the queue and the Voronoi diagram in $O(1)$ time. The list of p contains—on the average and once k points are ranked— $O(n/k)$ points, and redistributing them to the $O(1)$ neighboring cells takes $O(n/k)$ time. Updating the priority queue H takes $O(\log k)$ time typically. Overall we therefore get a typical running time of $O(n^2/k + n \log k)$.

Algorithm 3 for the addition model can be easily adapted. As before, every ranked point is added to a queue W. When it contains more than k points, the first element p is deleted from the head of the queue, which means that we delete the imaginary line ℓ that passes through p. We have to concatenate the four binary trees that lie between p and its closest clockwise neighbor p' and counterclockwise neighbor p'', thus creating one tree $\overline{T}$. We then split $\overline{T}$ at the angle of the bisecting barrier line $\ell_{p'p''}$. During the concatenate and split operations we need to keep the augmentation correct. Finally, we need to update the best score information in the priority queue H, which means that we delete the best scores of the old trees and insert the best scores of the new trees. The analysis of the running time is similar as before, and the asymptotical running time does not change.

For the wavefront method, Algorithm 4 can be adapted in a similar manner. We add every ranked point to a queue W, and if it contains more than k points, the first element p is deleted from the head of the queue, which means that we delete the line ℓ_p that passes through p. Thus, we create one new sector out of two old ones. We delete the three vertices v_{p-1}, v_p and v_{p+1} of the wavefront and add one new vertex, namely the intersection of the two edges that make an angle ρ with the bounding lines of the new sector. Finally, we need to delete the two old candidate points of the old sectors from the priority queue and insert the new candidate point of the new sector into it. The overall running time remains $O(n \log^2 n)$ in the worst case.

4.3 Higher dimensions

So far, we have considered only two possible scores that are combined to give a distributed ranking. The applications from the introduction show that dealing with more than two scores for each point the point set P can also be useful. For example, for the query "castles near Koblenz", there could be two castles with the same distance to Koblenz, one north and one south of it. If the documents about them have a similar textual score, they will be mapped to points in the plane which are very close to each other, and therefore they will be ranked apart. However, in this example, this outcome is unsatisfactory, and it would be better if we could distinguish the relative position of the castles to Koblenz by using two spatial dimensions (scores) and one textual score.

The four presented methods can be used for any number of scores. Computing the ranking requires extending the basic algorithms to higher dimensions. However, not all of the algorithms presented in Sections 2 and 3 can be extended to work as efficiently in higher dimensions. In this section we will briefly discuss the extensions of the presented algorithms to higher dimensions, where possible, or otherwise, state why this cannot be done.

The generic Algorithm 1, which can be applied to all four presented models, works as presented in any dimension d. Therefore, we conclude that the worst case running time for the generic algorithm is $O(n^2)$ in any dimension.

Algorithm 2 for the basic models can be extended to higher dimensions $d \geq 3$, but the worst case running time will go up with the dimension. We maintain the d-dimensional Voronoi diagram for the ranked points, and for each cell we maintain the unranked points in that cell in a list. The point with the highest score value per list is stored in a priority queue, where the next point to be ranked is chosen from.

For $d \geq 3$, the Voronoi diagram has complexity $O(n^{\lceil d/2 \rceil})$ in the worst case, and it can be constructed in $O(n^{\lceil d/2 \rceil})$ time [7]. It can also be constructed in $O(N \log n)$ time, where N is the actual complexity of the Voronoi diagram [18]. This leads to $O(n^{\lceil d/2 \rceil})$ time to rank a set of n points in d dimensions, $d \geq 3$, in the distance model. The Voronoi diagram for the angle model in three dimensions can be determined as follows: we project the ranked points to the surface of the unit sphere and compute the Voronoi diagram on this surface, which essentially is a 2-dimensional Voronoi diagram. In general, the Voronoi diagram needed for the angle model for point sets in d-dimensional space has dimension $d-1$. Consequently, in the angle model, ranking can be done in $O(n^{\lceil (d-1)/2 \rceil}) = O(n^{\lfloor d/2 \rfloor})$ time for $d \geq 4$, and in $O(n^2)$ time for $d = 3$.

To analyze the typical running time for the algorithm in higher dimensions, we will make some assumptions and show time bounds under these assumptions. Whether the assumptions hold in practice cannot be verified without implementation. We will assume that any newly ranked point only has a constant number of neighbors in the Voronoi diagram, and every list has length $O(n/r)$, where r is the number of ranked points so far. Note that the former assumption is, in a sense, stronger than in the planar case, because there are point sets where the average number of neighbors of a cell in the Voronoi diagram is linear, when $d \geq 3$. However, for uniformly distributed point sets, a higher-dimensional Voronoi diagram has linear complexity [7], and the average number of neighbors of a cell is indeed constant. Under these two assumptions, we can obtain a running time of $O(n \log n)$, as in the planar case.

Algorithm 3 for the addition model cannot be extended to higher dimensions $d \geq 3$. In the plane we have a linear ordering by angle on the point set P. This ordering is crucial for the functioning of the algorithm, because the split and concatenate operations are with respect to this ordering. In three or more dimensions the approach does not apply anymore.

It is possible to extend the wavefront model to higher dimensions, but the higher-dimensional version of Algorithm 4 would be complex and considerably less efficient. It would require linearization and higher-dimensional partition trees. The highly increased problem complexity in higher dimensions makes it not worthwhile to discuss it any further.

5 Experiments

We implemented the generic ranking Algorithm 1 for the basic ranking methods described in Sections 2.1, 2.2, and 3.1. Furthermore we implemented the extensions for staircase enforcement (Section 4.1) and limited windows (Section 4.2). We compare the outcomes of these algorithms for the two different point sets shown in Figure 6. The point set on the left consists of 20 uniformly distributed points, and the point set on the right shows the 15 highest ranked documents for the query ‘safari africa’ which was performed on a data set consisting of 6,500 Lonely Planet web pages. The small size of the point sets was chosen out of readability considerations.

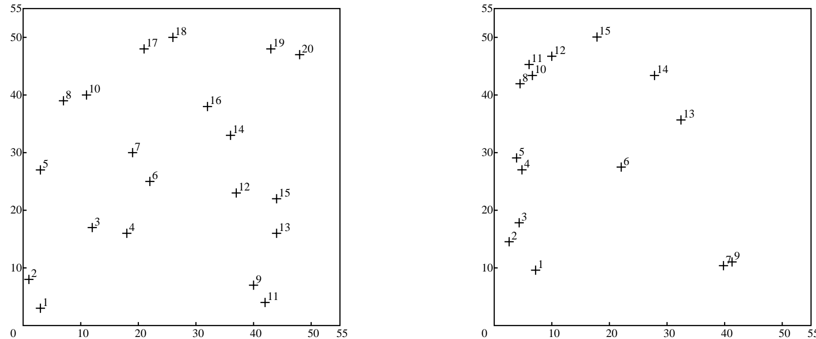

Figure 6: Ranking by distance to origin only.

5.1 Basic ranking algorithms

Figure 6 shows the output of a ranking by distance to query only. It is useful as a reference when comparing with the other rankings.

In the other basic ranking methods, shown in Figure 7, points close together in the plane are not necessarily close in the ranking sequences. This is visible in the ‘breaking up’ of the cluster of four points in the upper left corner of the Lonely Planet point set rankings. Note also that the points ranked last by the simple distance ranking are always ranked earlier by the other methods. This is because we enforced higher spreading over proximity to the query by the choice of parameters. The rankings are severely influenced by this choice. In our choice of parameters we did not attempt to obtain a “best ranking”. We used the same parameters in all three ranking methods presented here, to simplify qualitative comparison.

5.2 Staircase enforced ranking algorithms

In the staircase enforced methods, shown in Figure 8, the candidates to be ranked next are only those points that lie on the (lower left) staircase of the unranked points. The scoring functions and parameters are as before. With this adaptation, proximity to the query gets a higher importance than before. This is clearly visible in the figures, as the points farthest away from the query are almost always ranked last. Still, spreading in the rankings is present. This can for instance be seen from the two points in the lower right corner of both point sets: they are the closest pair of points in space, but their ranking differs significantly. In fact, this is just the behavior we expected from our distributed ranking.

5.3 Ranking algorithms with limited windows

In the last experiments, shown in Figure 9, the already ranked reference point we need to compute angle or distance to, is not the closest of all ranked, but the closest among the five last ranked points. This way we want to avoid that the angle or distance to the closest becomes too small to

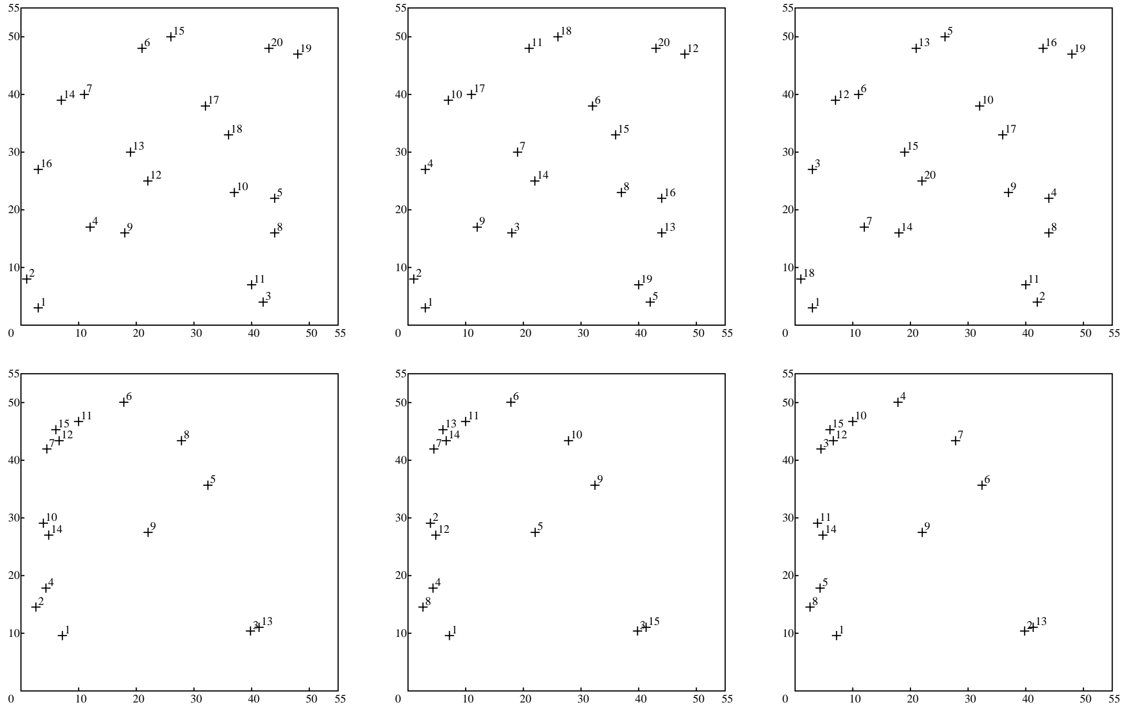

Figure 7: Left: Ranking by distance to origin and angle to closest ($k = 1$, $c = 0.1$). Middle: Ranking by distance to origin and distance to closest (Equation 3, $\lambda = 0.05$). Right: Ranking by additive distance to origin and angle to closest ($\alpha = 0.4$, $\lambda = 0.05$).

ensure a good distribution. The scoring functions and parameters are as before. With a window size of five, the first six ranked points are the same as with the basic models. It can be seen that points that lie very close to early ranked points are now ranked higher than with the basic methods, although the overall ranking remains well spread.

6 Conclusions

This paper introduced distributed relevance ranking for documents that have two or more scores. It is particularly useful for geographic information retrieval, where documents have both a textual and a spatial score. Distributed ranking combines high relevance of the query while avoiding redundancy. To our knowledge, it is the first time that these issues have been addressed with a purely geometric approach.

We have presented several methods for distributed ranking that are mainly based on a combination of the distance to the query and the distance to the closest already ranked point. Furthermore, we presented two extensions of the given basic methods, by staircase enforcement and limited windows. We first presented a generic algorithm with quadratic running time, which can be used for any of the presented methods and in any dimension. For some of the methods we can give efficient algorithms with better worst case running time.

The conducted experiments indicate that both requirements for a good ranking, small distance to query and high spreading, can be obtained simultaneously. Especially the staircase enforced methods and the methods with limited windows seem to perform well.

However, our reasoning is based on visual inspection of the outcome of our experiments only. Still, a user evaluation is needed to discover which ranking method is preferred, and which specific parameters should be used. The implementations of our distributed ranking methods are part of

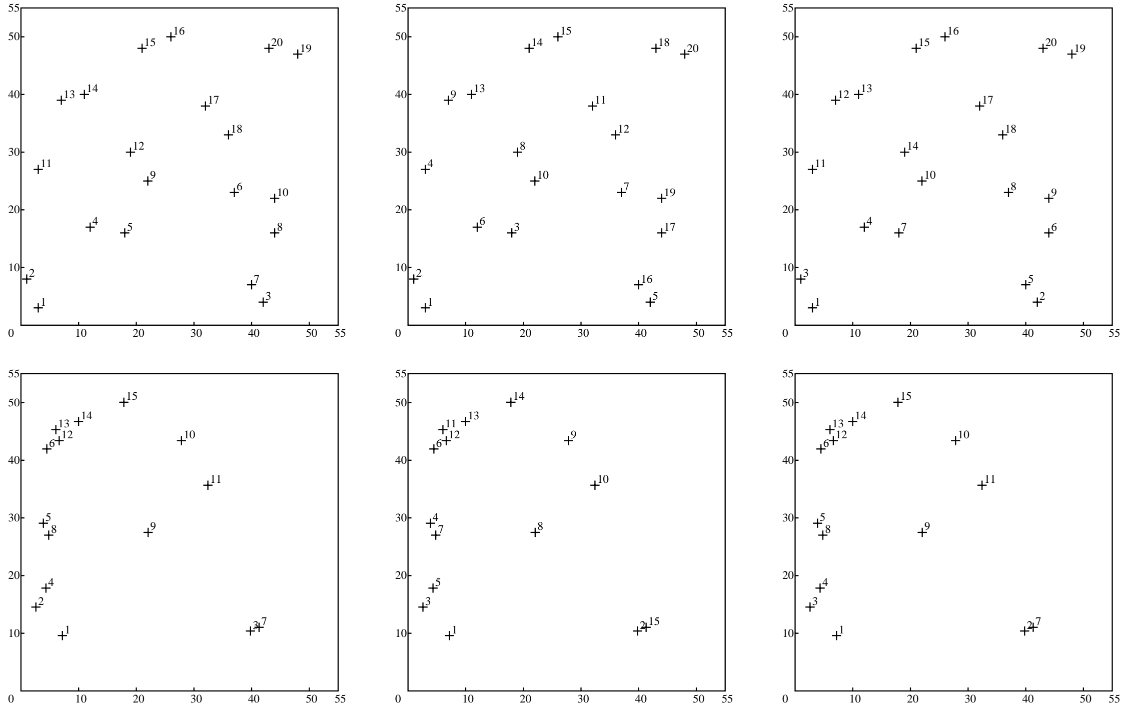

Figure 8: Same as Figure 7, but now staircase enforced.

the prototype of the spatial search engine under development within the SPIRIT project. Experiments to be done after completion of the prototype and the test collection will reveal whether users appreciate the distributed ranking option when results are presented.

References

[1] ANWB. Campinggids 2. 2002.

[2] A. Arampatzis and M. van Kreveld. Implementation of simple geographic similarity measures. Report 5102 of the SPIRIT project, 2003.

[3] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 617–626, 2002.

[4] J. G. Carbonell and J. Goldstein. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Research and Development in Information Retrieval*, pages 335–336, 1998.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[6] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[7] S. Fortune. Voronoi diagrams and Delaunay triangulations. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 225–265. World Scientific, Singapore, 2nd edition, 1995.

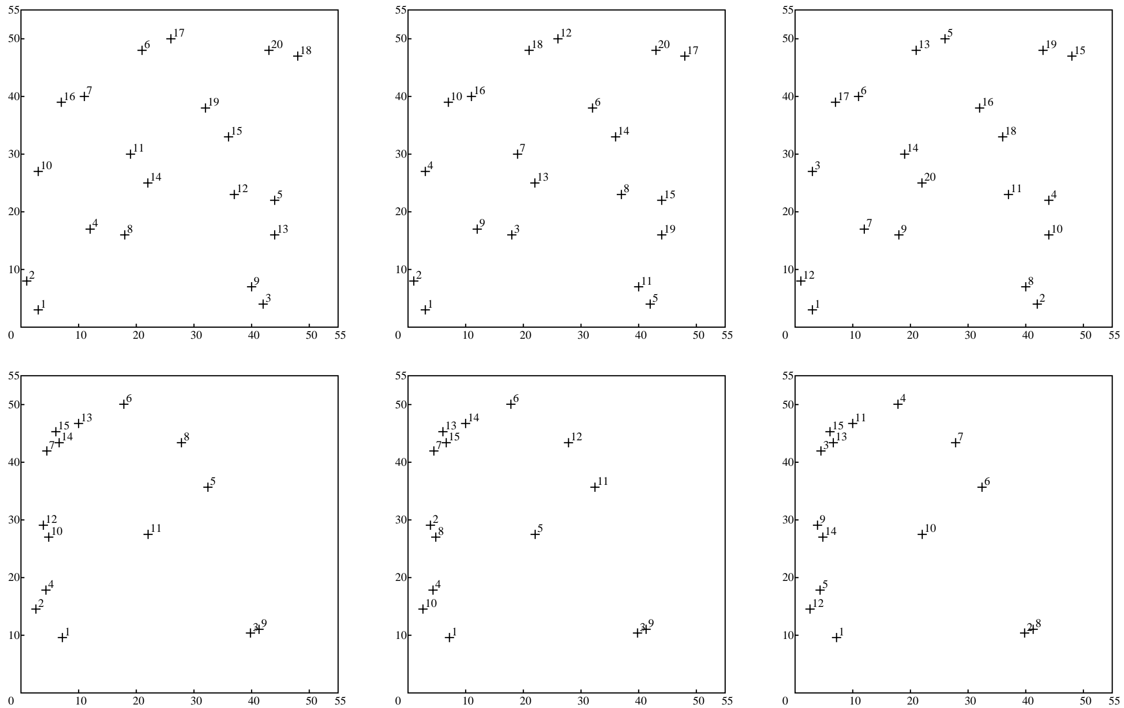

Figure 9: Same as Figure 7, but now with a limited window of size 5.

[8] J. Goldstein, M. Kantrowitz, V. O. Mittal, and J. G. Carbonell. Summarizing text documents: Sentence selection and evaluation metrics. In *Research and Development in Information Retrieval*, pages 121–128, 1999.

[9] J. Goldstein, V. O. Mittal, J. G. Carbonell, and J. P. Callan. Creating and evaluating multi-document sentence extract summaries. In *CIKM*, pages 165–172, 2000.

[10] D. Harman. Overview of the TREC 2002 novelty track. In *NISI Special Publication 500-251: Proc. 11th Text Retrieval Conference (TREC 2002)*, 2002.

[11] P. S. Heckbert and M. Garland. Fast polygonal approximation of terrains and height fields. Report CMU-CS-95-181, Carnegie Mellon University, 1995.

[12] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992.

[13] C.B. Jones, R. Purves, A. Ruas, M. Sanderson, M.Sester, M.J. van Kreveld, and R. Weibel. Spatial information retrieval and geographical ontologies – an overview of the spirit project. In *Proc. 25th Annu. Int. Conf. on Research and Development in Information Retrieval (SIGIR 2002)*, pages 387–388, 2002.

[14] G.E. Langran and T.K. Poiker. Integration of name selection and name placement. In *Proc. 2nd Int. Symp. on Spatial Data Handling*, pages 50–64, 1986.

[15] R.R. Larson and P. Frontiera. Ranking and representation for geographic information retrieval. In *Workshop on Geographic Information Retrieval at SIGIR*, 2004.

[16] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes Comput. Sci.* Springer-Verlag, Heidelberg, West Germany, 1983.

[17] E. Rauch, M. Bukatin, and K. Baker. A confidence-based framework for disambiguating geographic terms. In András Kornai and Beth Sundheim, editors, *HLT-NAACL 2003 Workshop: Analysis of Geographic References*, pages 50–54, Edmonton, Alberta, Canada, May 31 2003. Association for Computational Linguistics.

[18] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proc. 18th Annu. ACM Sympos. Theory Comput.*, pages 404–413, 1986.

[19] M. van Kreveld, R. van Oostrum, and J. Snoeyink. Efficient settlement selection for interactive display. In *Proc. Auto-Carto 13: ACSM/ASPRS Annual Convention Technical Papers*, pages 287–296, 1997.

[20] R. van Zwol, C. Jones, and M. van Kreveld. Aspects of spatial similatity measures. Report 5101 of the SPIRIT project, 2003.

[21] U. Visser, T. Vögele, and C. Schlieder. Spatio-terminological information retrieval using the BUSTER system. In *Proc. of the EnviroInfo*, pages 93–100, 2002.

[22] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32:597–617, 1985.