Low-Memory Spatial Indexing

JÖRG ROTH

Department of Computer Science, Nuremberg Institute of Technology

Kesslerplatz 12, 90489 Nuremberg, Germany

Joerg.Roth@th-nuernberg.de

Abstract. Spatial indexes provide a basic mechanism to efficiently execute geometric queries in, e.g., geo databases. They return a small list of candidates that *may* hold a certain geometric condition. Current spatial indexes are optimized to run in execution environments with large memories and are able to re-arrange their structures if object geometries change. For end-user devices with small amounts of memory these indexes are not appropriate. As mobile geo data repositories are often static, an index does not have to support runtime changes of geometries. In this paper we present a new spatial index that is optimized to run on small devices. The entire indexing structure is stored in a single file. At runtime, only a small amount of data is transferred into runtime memory. A single query usually requires only a small number of file operations.

1 Introduction

In the area of location-based services, a typical basic operation is to retrieve a set of objects that hold a certain geometric property, e.g., *all objects at a certain position*. To avoid costly checks of millions of geometries, geo databases typically use spatial indexes. They quickly retrieve a set of candidates from a huge set of possible objects, whereas the majority of objects are filtered out due to quick plausibility checks. Even though considered candidates still have to undergo exact geometric checks, spatial indexes speed up the execution of geometric queries by several magnitudes.

Typical spatial indexes are optimized for the queries '*overlaps with*' or '*covers a given geometry*'. As most spatial indexes are integrated into spatial databases, they furthermore are efficient regarding insert and removal of objects and modifications of object geometries. Usually, object geometries are approximated by bounding boxes.

Spatial indexes are usually based on data structures such as trees. To fully take advantage of these structures, they either have to reside in memory or have to efficiently be accessed on mass storage. For runtime environments with low memory this is a problem. For example, smart phones or smart watches have big flash memories (in the order of GB), but the amount of runtime memory is considerably small for a certain application (in the order of KB to some MBs). In addition, communication between flash memory and RAM often is a bottle-

neck. Nevertheless, it could be interesting for these devices to run location-based applications that have to lookup geo objects by geometric queries.

In this paper we present the *Low-Memory Point Index* (*LoMPI*) that requires only a low amount of memory for queries. It fully operates on a single random access file (called *index file*) that contains both indexing structures and object content data. During a query, an application moves a file access pointer among the index file to find the required list of geo objects. As each movement issues a file operation, our approach minimizes the average amount of movements per query.

2 Related Work

A first step to reduce the costs for geometric queries is to approximate geometries by simple bounding geometries such as rectangles, circles, ellipses or convex hulls [1, 3, 5, 13, 14]. The main goal is to provide a quick a priori test for spatial conditions. Only if this a priori test is successful, an additional (costly) exact check is reasonable. The simplest bounding shape is the bounding box, also called *Axis-Aligned Bounding Box* or *Minimal Bounding Rectangle*. In [8, 10] a much better approximation based on two bounding boxes for a single geometry is proposed.

Bounding geometries are only the first step. A spatial index usually contains a structure that tracks down to a small list of candidates without even considering most of the bounding shapes. Such structures are, e.g., Quadtrees [2] or variations of R-Trees [4]. For the full benefit, these structures have to hold certain criteria (e.g. have to be balanced). Indexes are able to quickly restore these criteria if geometries are inserted or removed.

In [6] we suggested a special structure called *Extended Split Index* that generates a structure optimized for queries in standard databases. In [9] we transferred this approach to smart phone databases. However, it requires a certain amount of runtime memory to directly access some structures. In [11] we analysed the benefit, if we sort geometries according to locality criteria to reduce the transfers from data file to memory.

3 The Approach

In our HomeRun project [7] we had the goal to generate a textual description of the current physical position on a smart phone without the need of a network service [12]. Our approach to access these geo data was heavily based on preliminary work in this area. Existing spatial indexes fail in low-memory environments, especially if we want to access millions of geo objects. This was why we introduced a new approach. To achieve our objectives, we have to impose some limitations:

- Our geo data is static. The index file is created *once* from a data source by an explicit export operation that may take some time (e.g. some hours). During this operation, the index structure is optimized according to our demands. This operation is complex and not executed on the target device. It is not intended to change the geometry data after an export.
- Our index only supports *point-queries*. This means: the geometry, to which the geo objects' geometries are compared, is a single point. Respective queries are: *get all objects that cover a point, get all objects that are nearer than x meters to a point* or *get the nearest object*.

There exist a lot of scenarios where only point-queries are required. E.g. a route planning application wants to retrieve the nearest street for the current position, a tour guide wants to present the nearest touristic sight, or a camera wants to tag an image with the current postal address.

For a certain query, the index returns two lists of results. First, a *Hit List* contains geo objects that definitely hold the geometric condition (without the need of exact checks). Second, the *Check List*, contains geo objects that *may* hold the geometric condition. The application has thus to execute an additional geometric check to ensure whether they are real hits. The index ensures that all matching geo objects appear in either of the two lists. The index tries to maximize the probability for an object to appear in the Hit List rather than in the Check List. The export operation can be configured to increase this probability with the cost of a larger index file.

3.1 Tiles, Hit Entries and Check Entries

To heavily reduce the number of candidates from many millions to, e.g., ten in a first step, we use a grid-based approach: we split the covered area of geometries to tiles of same size (initially). Note that the actual geo coordinate system is not of interest as long as it maps positions of the Earth's surface to two dimensions in a continuous manner.

For each tile we organize two lists:

- *Hit Entries* are those objects that fully enclose the tile. If we query a given position inside a tile, all Hit Entries must be a final hit without further checks.
- *Check Entries* are those objects that overlap with the tile. If we query a given position inside a tile, Check Entries *may* be a hit. To reduce the number of additional checks, the minimal bounding rectangle (*MBR*) of the geometric intersection between tile and geometry is stored.

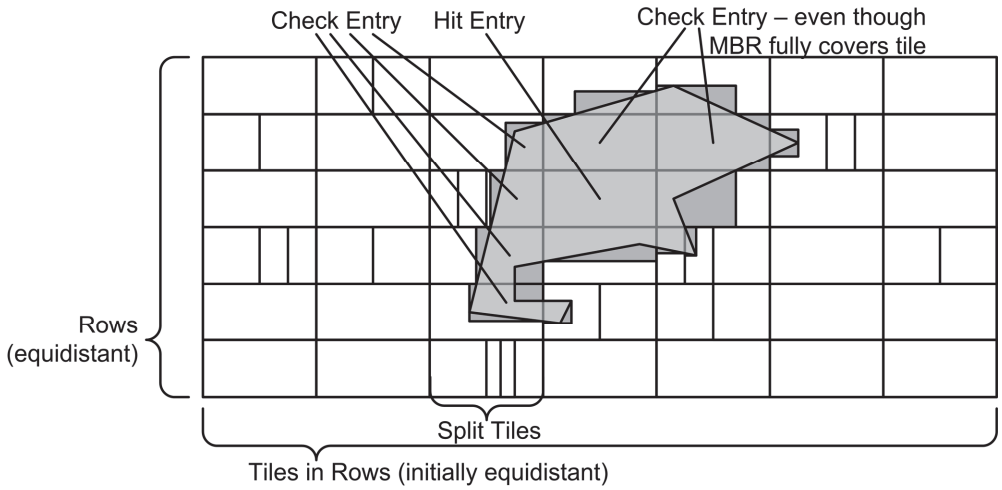

Fig. 1. The basic idea of tiles, Hit Entries and Check Entries

Fig. 1 shows the basic idea. The grid is initially equidistant, but tiles may be split (see later). A given geometry is intersected with all tiles. Only if it fully covers a tile, it is stored in the list of Hit Entries.

If a geometry intersects without covering a tile, it is more complex: we store the MBR of the *intersection* in the tile structure, in particular *not* the actual geometry's MBR. We get different cases (Fig. 2):

- If a geometry is not entirely inside a tile and crosses a border, the stored MBR only considers the part inside the tile (Fig. 2a).
- The geometry's MBR may enclose the tile, but only a small part intersects with the tile: then the stored MBR only covers the intersection (Fig. 2b).
- The MBR of the intersection may also be the entire tile. But even then the object is not stored in the Hit List if the tile is not fully covered (Fig. 2c).

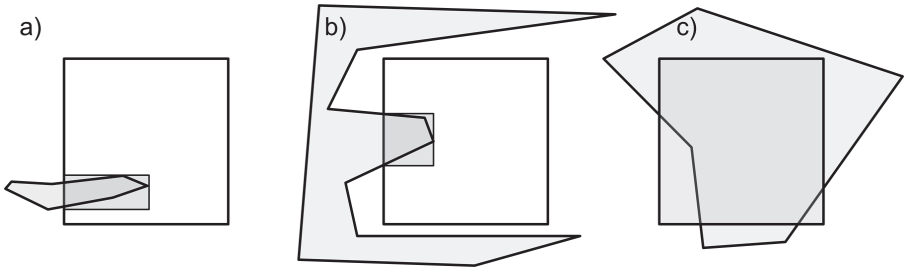

Fig. 2. Check Entries and stored MBRs

3.2 Assigning geo objects to tiles, splitting tiles

In a certain tile, an object can only appear either in Hit or Check List. However an object may appear in multiple tiles, if it crosses a tile border or is large enough to cover multiple tiles. The benefit of our approach: very large geome-

tries (e.g. country borders) usually appear in Hit Lists. Thus, exact checks are avoided in most cases. This is especially a benefit as exact checks of large geometries is costly (e.g. the Bavarian border is defined by 112 000 polygon points).

If an object appears in multiple tiles, the object's content is duplicated. The application developer is responsible to balance between redundancy and index file size. Large contents should be shifted from the index file to another file or database table. Very often, only the object's ID is stored in the index file.

At runtime, the number of Check Entries is a crucial point. In dense regions (e.g. city centres), we have a large number of geo objects. We thus reduce the tile size in such regions. The mechanism works as follows:

- Initially we have a fixed grid of tiles. The grid in latitude direction remains constant, only the tile size in longitude direction may be adapted.
- If the number of Check Entries exceeds a given limit, it is split into two halves of identical size. The Hit and Check Entries are re-organized and assigned to the new tiles (Fig. 3).
- The process is recursively executed until the number of Check Entries is low enough or if a minimal tile size is reached.

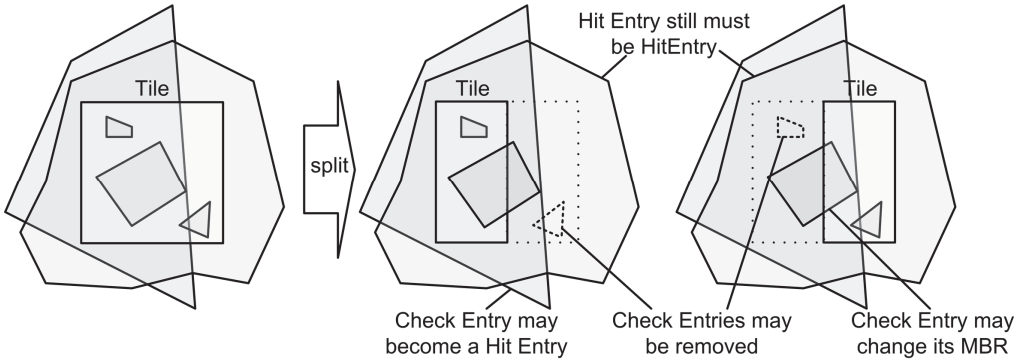

Fig. 3. Re-assignment of objects to split tiles

For a split, all Hit Entries must appear in both halves without any further considerations. For Check Entries the following cases may occur:

- it may become a Hit Entry,
- it may not appear at all in a split tile,
- it may still appear in the split tile, but the MBR may change.

Note that tiles are only split in longitude direction. This is because we want to keep the row size in latitude direction constant, i.e. we are able to compute a row that covers a certain coordinate in O(1). In a row, all tiles are ordered according to their longitude coordinate, thus we can use binary search to identify the actual tile for a coordinate.

3.3 Ordering objects in a tile

Once we located a tile in the index file, all Hit Entries of this tile are directly returned as Hit List. The second result is the Check List that contains the tile's Check Entries whose MBR cover the given position. This means, we cannot directly return *all* Check Entries, but have to check their bounding boxes. To avoid iterating through all Check Entries of a tile, we organize them as follows:

- The entries are ordered by the MBRs' northern borders (*MaxLat*) in ascending order.
- Each entry contains a field *MinSuccLat* that holds the southern bound of all MBRs of trailing entries, including the entry itself.

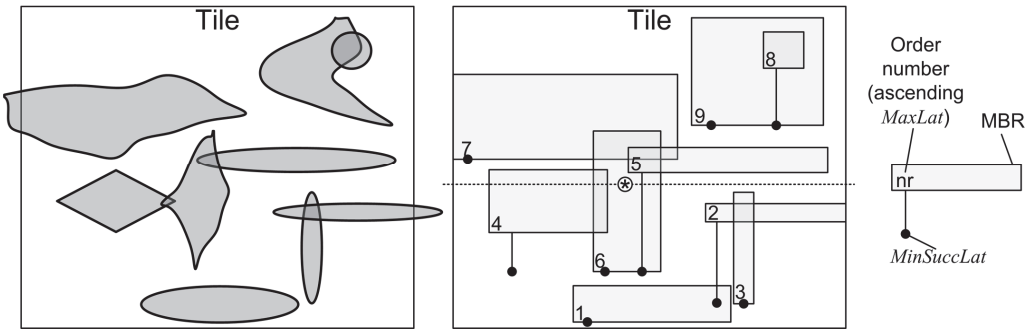

Fig. 4. Organisation of Check Entries in a tile

Fig. 4 shows an example. We look at the position marked by *. As the MBRs are ordered by their northern border (*MaxLat*), the first candidate is object 4 – 1, 2, 3 are too far south. We then check the objects 4, 5 and 6 and identify object 6 as a possible check candidate as its MBR encloses the position.

When we examined object 7 we can stop our search. As the southern bound (*MinSuccLat*) of all objects from 7 (i.e. 7, 8 and 9) is north of the given position, no further object can be a candidate.

3.4 The index file

The *Header*, *Row List*, *Tile List* and *Object List* are consecutively stored in a single file. Fig. 5 shows the structure of the index file. The Header allows to simply compute the locations of all lists using the fields *Row Count* and *Tile Count*. A *Total MBR* is used to quickly answer point queries outside of any geometry in the file. In addition, we can compute the fix south-north extend of a row with the help of *Total MBR*.

The *Row List* indicates the number of the first tile in the row. With this, we easily are able to get the tile indexes *from…to* that belong to a row. As the first tile number of the first row always is 0, we actually do not store this entry in the file – it is only shown for better clearness.

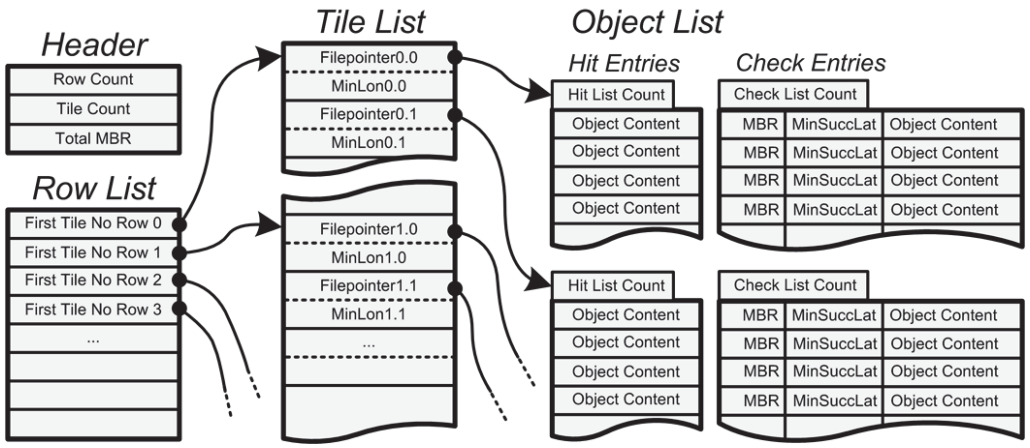

Fig. 5. Structure of the index file

The *Tile List* consecutively stores the start position in the file (in terms of bytes), where the tile content starts. Furthermore the longitude of the western border (*MinLon*) is stored.

All tiles of a certain row are ordered by *MinLon* in ascending order. Thus, we can easily find the tile in a row that covers a given longitude coordinate.

The Object List contains all tiles' content. Each of it contains the Hit and Check Entries whereas the latter is organized according to Section 3.3.

3.5 Queries at runtime

When opening an index file, the application developer can decide, how much data is loaded into memory. As a minimum requirement, the *Header* and *Row List* should reside in runtime memory. This even is possible for very small devices, as the memory requirement is in the order of 100 bytes to kBs.

We can decide whether the *Tile List* is also copied to memory. It typically has a size between some kBs up to some MBs. It depends on the runtime environment and tile size, whether it is reasonable to store the *Tile List* in memory. If not, the search of the actual tile in a row is performed by binary search on the index file.

The application developer can furthermore decide to use a row or tile cache. This is useful, if the runtime system offers memory without further runtime drawbacks (e.g. swapping of virtual memory), and if consecutive point queries usually are 'local'. This means, point queries are not randomly distributed among the area but we have a higher probability of two queries to address the same row or tile.

Given a position latitude/longitude, the following steps are performed to compute the Hit and Check List results:

– Identify the row from the position's latitude in O(1).

- Identify the *tile* in that row that covers the position's longitude. As the tiles are ordered by *MinLon*, we can use a binary search approach that needs O(log(*Tile Count*)) steps.
- From the tile, we return all Hit Entries.
- From the tile, we identify the first Check Entry candidate using a binary search on *MaxLat*. This requires O(log(|*Check Entries*|) steps.
- Starting from the first Check Entry we iterate through the list of Check Entries and test if the MBR covers latitude/longitude of the given position.
- We stop, if *MinSuccLat>latitude*.

The index file structure enables to linearly read from the file to produce the Hit and Check Lists. The Hit List is copied without any further computation. For the Check List we have to move within the file, performed by a function usually called **`seek`**. These jumps are controlled by a binary search. As usual file systems directly load a cluster of bytes from a file, the binary search does not track down to the respective Check Entry but stops, if the Check Entry is inside the currently loaded cluster.

4 Evaluation

To evaluate the efficiency of our approach we exported 21.9 million geo objects with area geometries (i.e. no lines, no points) of the file 'Germany' (July 2015) from Open Street Map. We selected as number of tiles per latitude and longitude one of 25, 50, 100, 200, 400, 800 or 1600. For the maximum number of Check Entries per tile we selected one of 500, 2000 or 20000 elements. *Header*, *RowList* and *TileList* are loaded to memory. No caching was used.

Fig. 6 shows the results. The runtime costs for a query mainly are affected by the number of **`seek`** operations and scanned Check Entries. Both can be controlled in a fine granular manner by adapting the grid size and upper bound for Check Entries. As lower bound for the number of seek operations we get 1, i.e. the respective start file cluster of the check candidates is instantly found in a single step.

The costs to reduce query time are larger index files and higher runtime memory requirements. Whereas the memory requirement follows a simple relation to grid size and max Check Entries, the index file size is not obvious. This is because the maximum check count per tile has a higher influence on the size than the initial grid size.

The ratio of scanned Check Entries to total Check Entries indicates the efficiency of our ordering approach of Check Entries. For low number of Check Entries, this ratio tends to 100% as all entries are available in a single cluster.

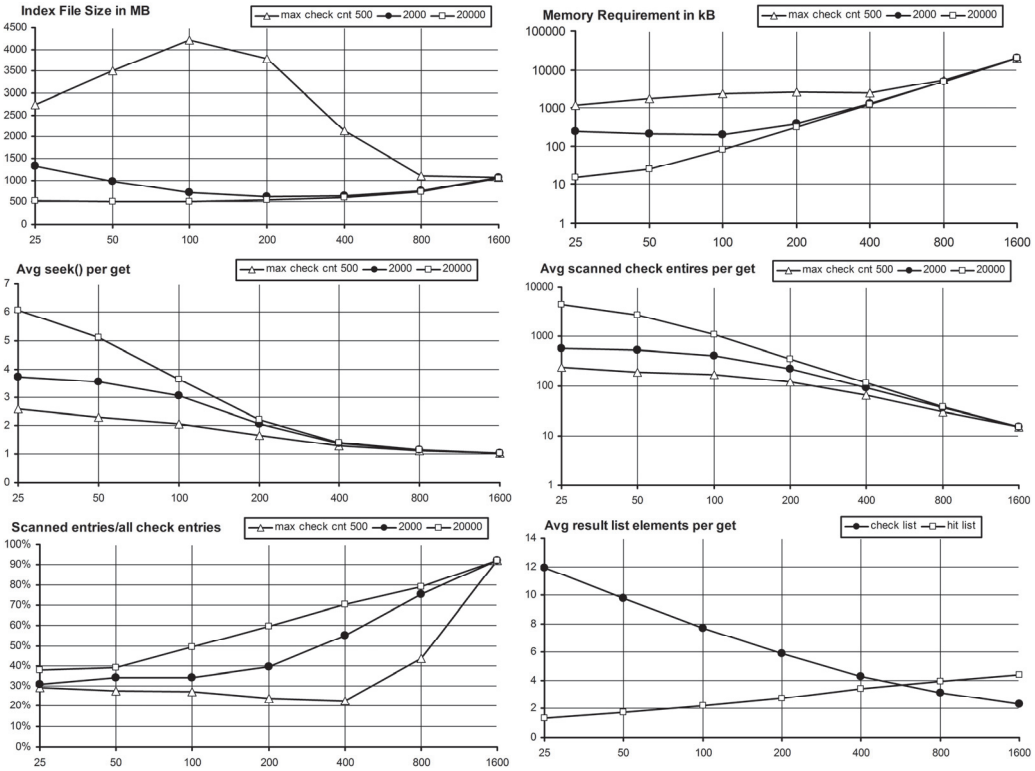

Fig. 6. Evaluation results (x-axis: initial tiles in one dimension)

For higher numbers, the ratio tends below 50% (50% is the theoretical value solely calculated from our binary search approach). Any ratio below 50% is a result of our *MinSuccLat* field that avoids iterations to the end of the list. The effect of *MinSuccLat* was below 10% on average, which was disappointing.

The last chart shows we measured the number of Hit and Check results. Not surprisingly, the Hit Lists get larger for smaller tiles. However, in our case we only have few geo objects that are large enough to cover tiles (usually borders of countries, states and large regions). For small tiles (1600x1600 initially), we get Hit Lists of 4 and Check Lists of 2 on average. This means, for an average query only two geometries had to be checked exactly, whereas four results are final results without further checks.

5 Conclusion

With our approach we can store large sets of geo data and execute point queries on the end-user devices. It is optimized for small amounts of runtime memory and comparably large mass storages. This is the usual case for current mobile devices.

With the help of export parameters, we can control runtime requirements in a fine-granular manner. In simple terms: we may safe memory with the cost of

longer runtime. The approach thus can be adapted to different runtime scenarios. The mechanism can be used stand-alone, but also could be integrated into embedded databases such as SQLite to support geometric queries.

References

[1] Barequet, G.; Har-Peled, S.: Efficiently Approximating the Minimum-Volume Bounding Box of a Point Set in 3D, Proc. 10th ACM-SIAM Sympos. Discrete Algorithms (1999), 82-91

[2] Finkel, R.; Bentley; J.L.: Quad Trees: A Data Structure for Retrieval on Composite Keys. Acta Informatica 4 (1): 1974; pp. 1–9.

[3] Gartner, B; Schonherr. S.: Smallest enclosing ellipses - fast and exact, Tech. Report B 97-03, Free Univ. Berlin, Germany (1997)

[4] Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. Proc. of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, June 1984; pp. 47-57.

[5] Hill, L. L.: Georeferencing: The Geographic Associations of Information MIT Press, Cambridge, MA

[6] Roth, J.: The Extended Split Index to Efficiently Store and Retrieve Spatial Data With Standard Databases, IADIS Intern. Conf. Applied Computing 2009, Rome (Italy), Nov. 19-21 2009; 85-92.

[7] Roth, J.: "Die HomeRun-Plattform für ortsbezogene Dienste außerhalb des Massenmarktes", in A. Zipf, S. Lanig, M. Bauer (eds.) 6. GI/ITG KuVS Workshop Location based services and applications, Heidelberger Geographische Bausteine Heft 18, 2010 (in German)

[8] Roth, J.: The Approximation of Two-Dimensional Spatial Objects by Two Bounding Rectangles, Spatial Cognition & Computation: An Interdisciplinary Journal, Volume 11, Issue 2, 2011, 129-152

[9] Roth J.: Moving Geo Databases to Smart Phones – An Approach for Offline Location-based Applications, Innovative Internet Computing Systems (I2CS), Berlin, June 15-17, 2011, GI Lecture Notes in Informatics, Vol. P-186, 228-238

[10] Roth, J.: An O(n) approximation for the double bounding box problem, IADIS International Conference Informatics 2011, Rome (Italy), July 20-22, 2011, 27-34

[11] Roth J.: A Spatial Hashtable Optimized for Mobile Storage on Smart Phones, in Werner M., Haustein M. (Hrsg.): 9. GI/ITG KuVS Fachgespräch "Ortsbezogene Anwendungen und Dienste", Sept. 13-14, 2012, TU Chemnitz, Universitätsverlag Chemnitz, 2013, 71-84

[12] Jörg Roth: Generating Meaningful Location Descriptions, International Conference on Innovations for Community Services (I4CS), July 8-10, 2015, Nuremberg (Germany)

[13] Yang, M., Kpalma, K.; Ronsin, J.: A Survey of Shape Feature Extraction Techniques. In Pattern Recognition Techniques, Technology and Applications, Peng-Yeng Yin (ed.), Nov. 2008, I-Tech, Vienna, Austria, pp. 626

[14] Welzl, E.: Smallest enclosing disks (balls and ellipsoids) in New Results and New Trends in Computer Science, Lecture Notes in Computer Science, Vol. 555 (1991), 359-370