Efficient Radius Neighbor Search in Three-dimensional Point Clouds

Jens Behley, Volker Steinhage, and Armin B. Cremers

***Abstract*— Finding all neighbors of a point inside a given radius is an integral part in many approaches using three-dimensional laser range data. We present novel insights to significantly improve the runtime performance of radius neighbor search using octrees. Our contributions are as follows: (1) We propose an index-based organization of the point cloud such that we can efficiently store start and end indexes of points inside every octant and (2) exploiting this representation, we can use pruning of irrelevant subtrees in the traversal to facilitate highly efficient radius neighbor search. We show significant runtime improvements of our proposed octree representation over state-of-the-art neighbor search implementations on three different urban datasets.**

I. INTRODUCTION

In this paper, we investigate radius neighbor search in three-dimensional point clouds $\mathcal{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N\}, \boldsymbol{p}_i \in \mathbb{R}^3$, *i.e.*, we are interested in finding all neighbors

$$\mathcal{N}(\boldsymbol{q}, r) = \{\boldsymbol{p} \in \mathcal{P} \mid ||\boldsymbol{p} - \boldsymbol{q}|| < r\}, \tag{1}$$

inside an arbitrary radius $r \in \mathbb{R}$ of a query point $\boldsymbol{q} \in \mathbb{R}^3$. Such queries are a cornerstone of many components in laser-based perception approaches, such as feature computation [1], [2] and normal estimation [3], [4]. Despite its importance, it is remarkable that there is virtually no progress towards faster radius neighbor search in the last decade.

Most work concentrated on more efficient retrieval of the nearest neighbor, either in high-dimensional data [5] or metric spaces [6], investigated approximate nearest neighbor search [7], or exploited graphics processing units [8] for faster retrieval. In robotics, kD-trees [9] and octrees [10] are widely adopted to search for nearest neighbors in three-dimensional data and research concentrated on memory-efficient implementations [11], [12] for large-scale datasets.

We propose improvements of octrees enabling faster radius neighbor search. A sparing representation of points inside each octant is our main contribution. Exploiting this information, we propose to prune subtrees of the octree if an octant is completely inside the query region. The early pruning enables the addition of points to the result set without explicitly computing distances $||\boldsymbol{p} - \boldsymbol{q}||$ of points $\boldsymbol{p}$ to queries $\boldsymbol{q}$, which improves the search time significantly. Our experimental results on urban point cloud datasets demonstrate the significant efficiency gains – factor 1.2 to 5.8 depending on the radius – over other publicly available nearest neighbor implementations supporting radius neighbor search. Our proposed octree implementation is available at http://www.iai.uni-bonn.de/∼behley/octree.

J. Behley, V. Steinhage, and A. B. Cremers are with the Department of Computer Science III, University of Bonn, 53117 Bonn, Germany.
{behley,steinhag,abc}@iai.uni-bonn.de

II. BACKGROUND

Before we discuss our contributions, we recapitulate the commonly used recursive construction of *leaf-based* octrees and the search for radius neighbors therein. Based on these principles, we explain our improvements in the next section.

A. Octree Construction

To accelerate the neighbor search in three-dimensional point sets, a commonly used strategy is to use a regular space partitioning – the octree. Starting with an axis-aligned bounding box with center $\boldsymbol{c} \in \mathbb{R}^3$ and equal extents $e \in \mathbb{R}$, which we call *octant* in the following discussion (see also Figure 1 (a)), we subdivide the octant recursively into smaller octants of extent $\frac{1}{2} \cdot e$ until an octant contains less than a given number of points – the *bucket size* b. In each tree level, the spatial subdivision partitions the points $\mathcal{P}_O \subseteq \mathcal{P}$ inside an octant O into disjoint subsets $\mathcal{P}_k \subseteq \mathcal{P}_O$,

$$\mathcal{P}_O = \bigcup_k \mathcal{P}_k, \text{ s.t. } \mathcal{P}_i \cap \mathcal{P}_j = \emptyset, i \neq j. \tag{2}$$

If an octant contains less than b points, we stop the subdivision and store a list of points contained in the leaf octant. Hence, a point $\boldsymbol{p}_i \in \mathcal{P}$ is inside the leaf octant L with center $\boldsymbol{c}_L$ and extent e_L, if and only if

$$\max_j |\boldsymbol{p}_i^{(j)} - \boldsymbol{c}_L^{(j)}| < e_L, \tag{3}$$

where $\boldsymbol{v}^{(j)}$ denotes the j-th component of vector $\boldsymbol{v}$.

We can determine the partition of points in relation to the center $\boldsymbol{c}_O$ of octant O in linear time and therefore build an octree in $O(d \cdot N)$, where $N = |\mathcal{P}|$ refers to the number of points and d is the tree depth. Since every point belongs exactly to a single leaf octant, $O(N)$ additional space is needed to store the points inside the leaf octants.

B. Naïve Radius Neighbor Search in Octrees

Using the octree, we can retrieve all radius neighbors for an arbitrary query point $\boldsymbol{q} \in \mathbb{R}^3$ and radius $r \in \mathbb{R}$ with respect to a norm $||\cdot||$, as follows.

Starting at the root, we traverse the octree recursively and investigate octants overlapping the search ball $S(\boldsymbol{q}, r)$ defined by the query $\boldsymbol{q}$ and radius r, since only these could potentially contain points that are also inside the desired neighborhood. When we reach an overlapping leaf octant, we check all points $\boldsymbol{p}$ inside the octant whether they are also inside $S(\boldsymbol{q}, r)$, *i.e.*, $||\boldsymbol{p} - \boldsymbol{q}|| < r$. All points inside the search ball $S(\boldsymbol{q}, r)$ are also inside the set of radius neighbors $\mathcal{N}(\boldsymbol{q}, r)$. Thus, we only need to compare points inside overlapping leaf octants and consequently may discard large subsets of points that are irrelevant for the query.

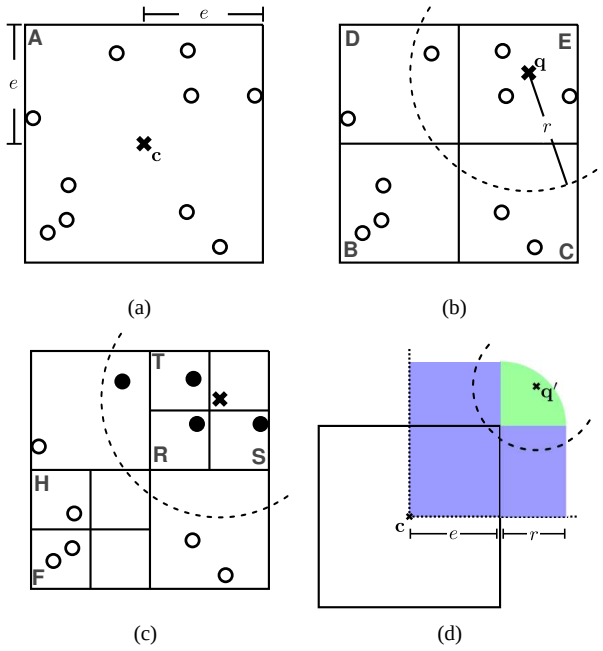

Fig. 1. (a) Representation of an octant by its center $\boldsymbol{c}$ and extent e, *i.e.*, half of the side length. (b) Starting at the root, octants overlapping the search ball $S(\boldsymbol{q}, r)$ (indicated by the dashed line) are visited. (c) Only points in overlapping leaf nodes are tested for inclusion in $S(\boldsymbol{q}, r)$. (d) The overlap test can be simplified if we exploit the axis-symmetry with respect to the octants center $\boldsymbol{c}$. We only need to consider two regions, shown in blue and green, to determine if an octant overlaps $S(\boldsymbol{q}', r)$.

Figure 1 (b) and (c) show an example query. In this example, octant H and F can be ignored in the search traversal, as they do not overlap with the search ball.

To test whether an octant with center $\boldsymbol{c}$ and extent e overlaps $S(\boldsymbol{q}, r)$, we can exploit the axis-symmetry of the octant with respect to its center. We transform the query point $\boldsymbol{q}$ into the local coordinate system of the octant, resulting in the transformed query point $\boldsymbol{q}'$ with $\boldsymbol{q}'^{(j)} = |\boldsymbol{q}^{(j)} - \boldsymbol{c}^{(j)}|$. The transformed search ball $S(\boldsymbol{q}', r)$ now overlaps with the octant if its midpoint $\boldsymbol{q}'$ lies either in the blue or green region depicted in Figure 1 (d).

This directly results in the following two conditions for the overlap test if the transformed query point fulfills $\max_j \boldsymbol{q}'^{(j)} < e + r$:

$$\min_j \boldsymbol{q}'^{(j)} < e \tag{4}$$

$$||\boldsymbol{q}' - \mathbf{1} \cdot e|| < r, \tag{5}$$

where $\mathbf{1}$ is the vector $(1, \ldots, 1)^T$ containing only ones.

Equation 4 corresponds to the blue and Equation 5 to the green region in Figure 1 (d). If one of these conditions holds, the transformed search ball at least touches the given octant.

III. Improving the Radius Neighbor Search

In the leaf-based octree implementation, we always have to traverse the tree until we reach a leaf octant and each point inside the leaf octant needs to be tested for inclusion in the search ball $S(\boldsymbol{q}, r)$. But when we take a closer look at the example shown in Figure 1 (b), we can see that the dashed search ball already completely contains octant E and therefore each point inside the octant must be also inside of $S(\boldsymbol{q}, r)$ due to the inclusion property (Equation 3). Consequently, we can stop the traversal of the octree in a subtree as soon as we find an octant that is completely inside of $S(\boldsymbol{q}, r)$ and simply add all points inside the contained octant to the result set $\mathcal{R}$.

Algorithm 1: Improved **radiusNeighbors**

Input: Octant O, query point $\boldsymbol{q}$, radius r, result set $\mathcal{R}$
Result: $\mathcal{R}$ contains all radius neighbors $\mathcal{N}(\boldsymbol{q}, r)$

```
if O is inside S(q, r) then
    Add all points inside O to R
    return
end
if O is leaf octant then
    foreach point p inside O do
        Add p to R if ||p − q|| < r holds
    end
    return
end
foreach child octant C of O do
    if S(q, r) overlaps C: radiusNeighbors(C, q, r, R)
end
```

We propose to extend the naïve radius neighbor search of the last section by an inclusion check, which is performed before child octants are recursively investigated. If we find an octant that is contained in $S(\boldsymbol{q}, r)$, we just add all points inside the octant without any computation of distances and stop the traversal in this subtree (*cf*. Algorithm 1). As we will see later, this subtle change in the tree traversal has significant impact on the radius neighbor search time.

However, the described leaf-based implementation does not facilitate this idea, since the information which points are inside an octant is only stored in leaf octants. We could naïvely store a list of points in each octant of the search tree. However, this would considerably increase the required space to store the data structure to $O(N^2)$ instead of $O(N)$, which is problematic if we have to deal with large datasets.

We will now discuss how to efficiently store the points and link every octant with the contained points with minimal overhead in the representation of an octant, which is the foundation for the proposed pruning strategy to speedup the radius neighbor search. After this we will discuss an efficient inclusion test to determine if an octant is inside the search ball $S(\boldsymbol{q}, r)$, which is the second contribution of this paper.

A. Index-based Octree

As discussed earlier in Section II, each subdivision partitions the points inside an octant into disjoint subsets (*cf*. Equation 2). Each further subdivision again results in disjoint subsets of the points inside the parent octant. As long as we keep points inside an octant together, we can reorder the subsets arbitrarily in every level of the octree. This insight is the key to a representation that enables us to link each octant with the points inside it.

Algorithm 2: Octant creation with **createOctant**

Input: octant center $\boldsymbol{c}_O$, extent e_O, start index s_O, end index t_O, number of points M_O
Result: Octant O

1 Initialize octant O with $\boldsymbol{c}_O$, e_O, s_O, t_O, and M_O.
2 Let s_k be the start index, t_k the end index, and M_k the number of points of the k-th child.
3 Let $\boldsymbol{c}_k$ be the child center and $e_k = \frac{1}{2}e_O$ its extent.
4 **if** $M_O > b$ **then**
5 Set $i = s_O$ and $j = 0$
6 **while** $j < M_O$ **do**
7 Determine Morton code k of $\boldsymbol{p}_i$
8 **if** $M_k = 0$: $s_k = i$
9 **if** $M_k > 0$: $\mathrm{succ}(t_k) = i$
10 Update $t_k = i$ and $M_k = M_k + 1$
11 Get next point index $i = \mathrm{succ}(i)$ and $j = j + 1$
12 **end**
13 Let l be the Morton code of the last child octant.
14 **foreach** k, *where* $\mathrm{M}_k > 0$ **do**
15 C_k = **createOctant**($\boldsymbol{c}_k$, e_k, s_k, t_k, M_k)
16 Update s_k and t_k with new start/end of C_k
17 **if** C_k *is first child* **then**
18 Update octant's start $s_O = s_k$
19 **else**
20 Update last child's end $\mathrm{succ}(t_l) = s_k$
21 **end**
22 Update octant's end $t_O = t_k$ and set $l = k$
23 **end**
24 **end**
25 **return** O

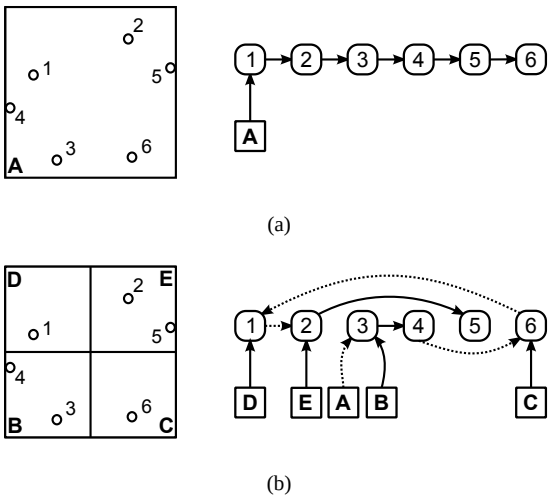

Fig. 2. Example for the update of the successor relation while constructing the octree. Also shown is the link from the octant (square) to the start point s inside the list.

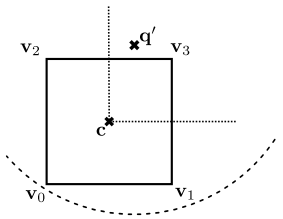

Fig. 3. To check if an octant is completely inside the search ball $S(\boldsymbol{q}', r)$, we only need to test whether the farthest octant corner $\boldsymbol{v}_0$ is inside.

Starting at the root, we subdivide the points into subsets and reorder them such that the points of each child octant are sequenced together. For this purpose, we use an array of successors succ, which is altered in the octree construction such that $\mathrm{succ}(i) = n$ maps to the next point $\boldsymbol{p}_n$ that is inside an octant or the first point of the next child octant. The successor relation represents a single-connected list, where we have random access to individual list items.

In our octree representation, each octant O stores its center $\boldsymbol{c}_O$, its extent e_O, the indexes s_O and t_O of the first and last point inside it, and the number of points M_O inside it. To enumerate all points inside O, we start with index s_O and use succ to access the remaining $M_O - 1$ points.

Algorithm 2 summarizes the recursive creation of octants and update of succ. Lines 6–12 subdivide the points into subsets using Morton codes [13] to index the child octants and update the successors such that it can be used to initialize the child octants. In line 15, the child octants are recursively created, which might alter the start and end index of the child octant due to changes in the successor relation. Therefore, we have to update the start and end index in Lines 18 and 22 such that it points to the correct position in the successors.

Figure 2 shows an example for the relinking of the points while building the octree. Here, we visualized $\mathrm{succ}(i)$ as directed edges between nodes that correspond to points. Also shown is the link from the octant (square node) to the point s, which links to the remaining points inside it.

Initially, all points $\boldsymbol{p}_i$ are linked to the following point $\boldsymbol{p}_{i+1}$ regardless of which octant they belong to, *i.e.*, $\mathrm{succ}(i) = i + 1$, shown in Figure 2 (a).

As long as an octant contains more than b points, we subdivide the points into subsets corresponding to the child octants. As now points with non-sequential indexes might fall into the same octant, we have to update the successor relation, which is depicted in Figure 2 (b). The solid links are caused by Lines 6–12 of the algorithm and ensure that points of the child octants can be used in the recursive construction of the child octants. The dotted links are inserted after creation of the child octants, which is necessary to ensure that we can iterate over all points using succ in the parent octant. First, we have to update the start of octant A in Line 18. But also between the subsets of children, we have to relink the points (see Line 20).

B. Inclusion test

Since we augmented our octree with enough information to determine for each octant which points are inside it, we now have to discuss how to efficiently determine if an octant is completely inside a given search ball $\mathcal{S}(\boldsymbol{q}, r)$.

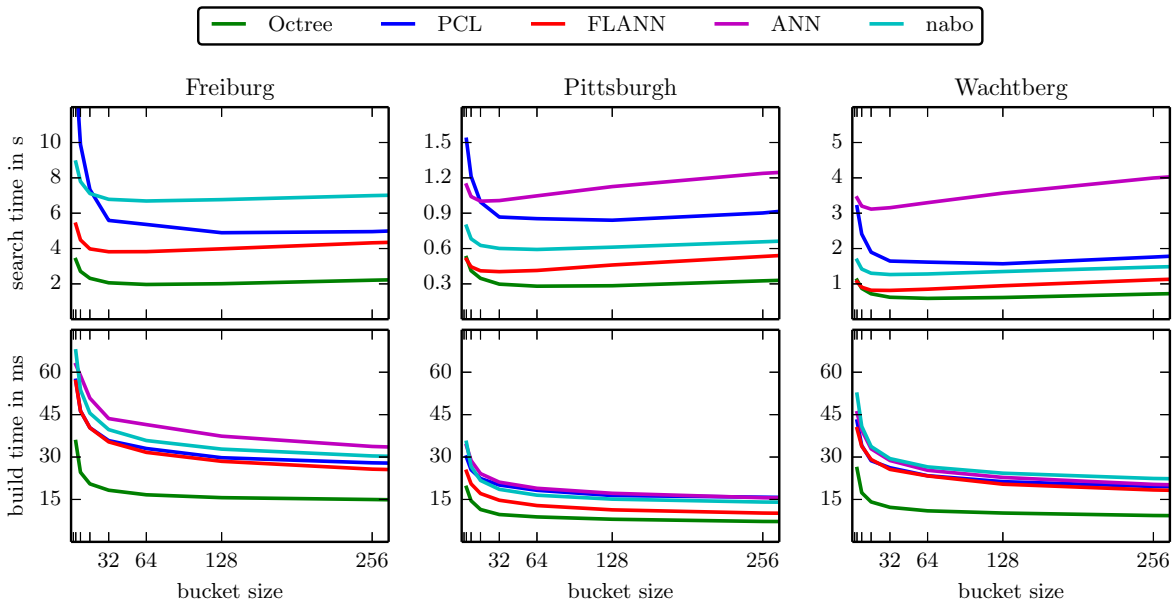

Fig. 4. The upper row shows the complete search time for radius neighbor queries with radius $r = 0.5\,\mathrm{m}$ depending on the bucket size b and the lower row shows the time needed to build the evaluated search tree implementations depending on the bucket size b.

A simple solution would be to check if every corner $\boldsymbol{v}_i$ of the octant is inside $S(\boldsymbol{q}, r)$, since every point inside the cube spanned by the octant must be also inside $\mathcal{S}(\boldsymbol{q}, r)$.

When we take a closer look at the problem, we see that we can again exploit the axis-symmetry of the octant and use the transformed $\boldsymbol{q}'$ as earlier with the overlap test (see also Figure 3). We can simplify the inclusion test by simply checking if $\boldsymbol{v}_0 = -e \cdot \mathbf{1}$ fulfills

$$||\boldsymbol{q}' - \boldsymbol{v}_0|| < r. \tag{6}$$

Intuitively, if corner $\boldsymbol{v}_0$ is inside $S(\boldsymbol{q}', r)$ every other corner must be also inside $S(\boldsymbol{q}', r)$, since $||\boldsymbol{v}_0 - \boldsymbol{q}'|| \geq ||\boldsymbol{v}_i - \boldsymbol{q}'||$.

This results in an efficient test, which only needs a single norm computation with the transformed query point $\boldsymbol{q}'$ and only one comparison in contrast to the the naïve test, which needs eight norm computations and comparisons.

IV. Experimental Evaluation

We compare our proposed octree implementation to publicly available implementations[1] of kD-trees, libnabo [14], ANN [7], FLANN [15], and the octree offered by the Point Cloud Library (PCL) [16], where we used a resolution of 0.01 for the PCL octree, but enabled it to dynamically grow until the desired bucket size b is reached. For all experiments, we report the overall time needed to determine for each point in the point cloud the desired radius neighborhood.

The FLANN kD-tree and PCL octree implementations explicitly offer methods for radius neighbor search. With ANN and libnabo, we can only implicitly search via k-nearest neighbor queries for radius neighbors by setting $k = N$ and the maximal radius of reported neighbors to our desired radius r. However, both implementations report for all k points a distance and therefore set a default value for points outside the radius. Thus, the query time is usually dominated by zeroing the whole result set. To allow a fairer comparison, we modified the ANN and libnabo implementation to only report distances of points inside the query radius.

TABLE I
Dataset statistics

Dataset	No. of points	Dimensions (in x, y, z) [m]
Freiburg	176.250	97.8, 47.8, 8.2
Pittsburgh	100.000	121.0, 72.9, 29.7
Wachtberg	131.807	97.0, 97.6, 5.1

We compiled all implementations with gcc 4.8.1 using `-O3 -march=native -DNDEBUG -UDEBUG` to enable all compiler optimizations. All experiments were performed single-threaded using an Intel Xeon X5550 with 2.67 GHz.

A. Datasets

As the performance of the data structures naturally depends on the point cloud data, we evaluated them using real-world data generated by different sensor setups. The first dataset was recorded at the University of *Freiburg*, Germany, using a SICK LMS laser rangefinder mounted on a pan-tilt unit [17]. The second dataset was acquired at the Carnegie Mellon University in *Pittsburgh* with a Jeep equipped with SICK laser scanners facing sideways [18]. The third dataset was recorded at the Fraunhofer FKIE in *Wachtberg*, Germany, using a Velodyne HDL-64E S2 laser range scanner mounted on an Opel Vectra [1]. A more detailed discussion of the datasets can be found in [1] and we used always the first scan from the available data sets[2]. Table I summarizes statistics of the laser scans used in the experiments: the

[1]We used libnabo 1.0.4, ANN 1.1.2, FLANN 1.8.4, and PCL 1.7.1.

[2]The data can be downloaded at http://www.iai.uni-bonn.de/∼behley/data/.

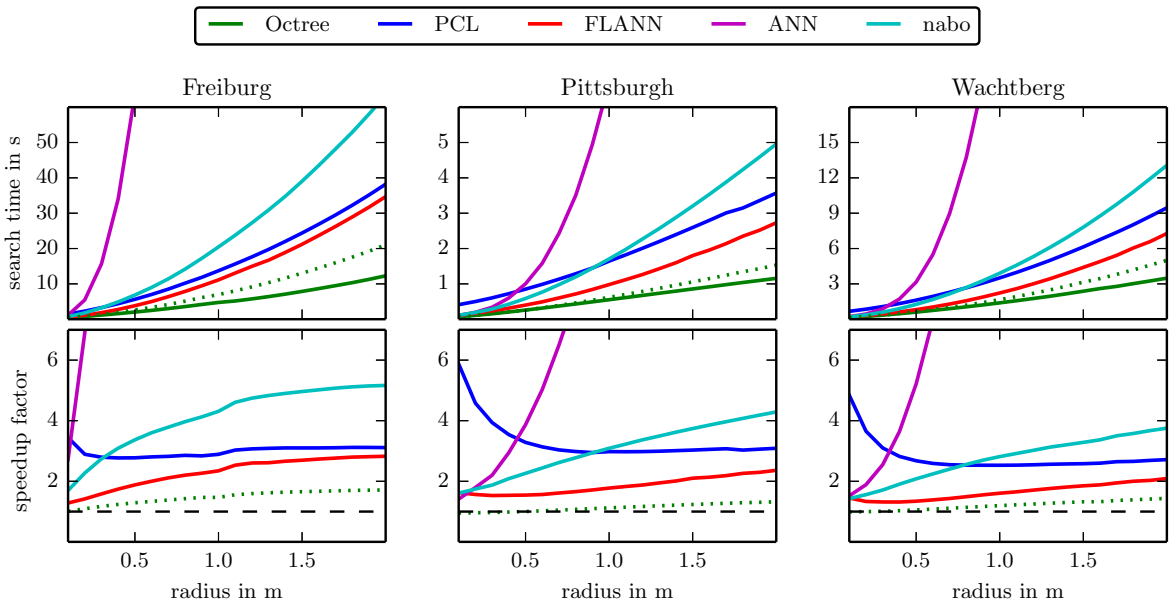

Fig. 5. Search time and speedup depending on the radius of the radius neighborhood. The upper row shows the complete time needed to perform a radius neighbor search for every point in the dataset. The lower row shows the speedup of the proposed octree in comparison to the other search tree implementations. The dashed black line indicates equal performance of the implementations. The dotted green line shows the performance of our octree implementation without pruning. In comparison to the other implementations, our proposed octree implementation shows a significant speedup.

number of points and the dimensions $\boldsymbol{d}$ of an axis-aligned bounding box, *i.e.*, $\boldsymbol{d}^{(j)} = \max_i \boldsymbol{p}_i^{(j)} - \min_i \boldsymbol{p}_i^{(j)}$.

B. Bucket Size and Construction Time

The bucket size b is an important parameter that must be selected in advance. It influences the depth of the search tree and determines how many points have to be compared with the query point $\boldsymbol{q}$ in the leaf nodes. Figure 4 (upper row) shows the radius neighbor search time for $r = 0.5\,\mathrm{m}$ depending on the bucket size $b \in \{2^i | 2 \leq i \leq 8\}$.

Interestingly, all search tree implementations showed a similar behavior depending on the bucket size: (1) If we use a bucket size $b < 16$, the search time increased significantly as the tests in inner nodes dominated the final comparison in the leaf nodes, (2) using a bucket size in the range $32 \leq b \leq 64$ resulted in similar minimal search times for the given implementations, and (3) larger bucket sizes $b > 64$ increased the search time again, since this leads to many comparisons in the leaf nodes and therefore lessens the computational advantage over simple linear search. We also evaluated other radii, but the general influence of the bucket size on the search time was essentially the same. Hence, we fixed the bucket size to $b = 32$ for the following experiments.

The lower row of Figure 4 shows the time needed to construct the search trees depending on the bucket size. Larger bucket sizes reduce the depth of the search trees, which consequently reduces also the number of nodes in the tree. Thus, we can observe a reduction of the build time with increasing bucket size.

Figure 4 also shows that the proposed index-based octree representation clearly outperforms the other implementations in terms of search time and also build time. An explanation for the significant difference in build time might be that we avoid copying the point data and just need to initialize and modify the successors while building the octree. Furthermore, the update of the successors in each level is done in a single pass over the relevant subset of points and only needs the computation of Morton codes.

C. Radius Neighbor Search Time

We also evaluated the search time depending on the query radius r, $r \in \{0.1, 0.2, \ldots, 2.0\}$, and all implementations used a bucket size $b = 32$ based on the earlier results. Figure 5 shows the results of these experiments: the upper row depicts the sum over the individual search times and the lower row shows the speedup, i.e., $S = T_\mathrm{X}/T_\mathrm{ours}$, over the other implementations X, where the dashed black line indicates equal performance.

These results show that our octree implementation significantly outperforms the other implementations also with other radii. The advantage of the pruning strategy increases with larger radii, since octants are more likely to be completely inside the search ball. In comparison to the best other kD-tree implementation, we can observe a speedup of 1.2–2.7 over the different datasets. Especially in outdoor environments, larger neighbor radii are often used for feature computation to capture context of large-scale object classes such as cars, trees, or buildings [1], [2].

To show the impact of the early pruning strategy enabled by the efficient inclusion test, we also show the time needed without pruning (dotted green line in Figure 5). This comparison confirms our hypothesis that the proposed early pruning of subtrees improves the radius neighbors performance, up to 1.7 speedup, in contrast to leaf-based octrees (dotted line), which always have to descend to a leaf node and compare the contained points to the query point.

V. Related Work

Closely related to the (general) radius neighbor search is the fixed-radius neighbor search problem. Here one knows the query radius r in advance and can therefore exploit this information in the construction of the data structure. In such settings, grid-based methods are quite effective for three-dimensional data, since they allow the point location in a grid cell in constant time with minimal overhead to investigate neighboring grid cells [19]. Such grid-based representations are often applied in GPU-based implementations of fixed-radius neighbors. Bentley *et al*. [20], [21] discussed different methods for fixed-radius neighbor search and pointed out that the kD-tree is the most flexible and efficient data structure for arbitrary dimensions. In contrast to their results, we could show that (carefully implemented) octrees can be significantly faster than kD-trees, like the FLANN, if we prune subtrees completely inside the search region.

As stated earlier, most work in three-dimensional neighbor search concentrated on nearest neighbor search. In this context, Elseberg *et al*. [14] evaluated different implementations of octrees and kD-trees for iterative closest points (ICP) [22]. Hornung *et al*. [11] proposed a probabilistic and memory-efficient implementation of octrees in context of mapping applications. Elseberg *et al*. [12] also reorder points to efficiently address points in their implementation of an octree, but use this only in leaf nodes and furthermore use multiple quicksort-like passes over the points in each level to reorder the points. Compared to their approach, our approach with a single pass over the points is certainly faster.

VI. Conclusion

In this paper, we proposed an octree implementation that significantly improves the radius neighbor search in three-dimensional data. Cornerstone of this implementation is an elegant organization of the points enabling the storage of indexes inside each octant and the efficient retrieval of points inside every octant. The experimental evaluation showed that our algorithmic improvements lead to a significant speedup over other state-of-the-art implementations.

The mindful reader might have noticed that we never made any assumption about the distance $||\cdot||$ in the introduction (Equation 1), the derivation of the overlap test (Equations 4 and 5), or the inclusion test (Equation 6). Hence, we can directly use every p-norm instead of the standard L_2 norm without any change in the radius neighbor procedure or the construction of the octree.

In our current implementation, we are quite wasteful in terms of the memory usage of the octants, since this saves computation of the center and extent while traversing the octree. Nevertheless, this relatively large size of octants compared to other implementations (93 Bytes vs. 8 Bytes [12]) might be cumbersome for larger three-dimensional datasets and could be improved by a more memory-efficient representation [11], [12], which only stores the necessary data and computes other information while traversing the tree.

We furthermore investigated here only exact radius neighbor search and showed that this can be sped up using our index-based representation and the proposed pruning strategy. An interesting avenue would be to investigate if this is still true when searching for approximate radius neighbors and whether similar simple geometrical tests could be used to accelerate the k-nearest neighbor search using octrees.

Lastly, the overlap and inclusion tests are directly applicable to octants with unequal extents, which enables the investigation of more data-sensitive octree variants that could represent the data by a more balanced and smaller tree.

VII. Acknowledgments

Thanks to Marcell Missura, Florian Schöler, and Jenny Balfer for fruitful discussions and many corrections to early drafts of the paper.

References

[1] J. Behley, V. Steinhage, and A. B. Cremers, "Performance of Histogram Descriptors for the Classification of 3D Laser Range Data in Urban Environments," in *ICRA*, 2012, pp. 4391–4398.

[2] X. Xiong, D. Munoz, J. A. Bagnell, and M. Hebert, "3-D Scene Analysis via Sequenced Predictions over Points and Regions," in *ICRA*, 2011, pp. 2609–2616.

[3] N. J. Mitra and A. Nguyen, "Estimating Surface Normals in Noisy Point Cloud Data," in *SCG*, 2003, pp. 322–328.

[4] K. Klasing, D. Wollherr, and M. Buss, "A Clustering Method for Efficient Segmentation of 3d Laser Data," in *ICRA*, 2008, pp. 4043–4048.

[5] M. Muja and D. G. Lowe, "Scalable Nearest Neighbor Algorithms for High Dimensional Data," *TPAMI*, 2014.

[6] T. Liu, A. W. Moore, K. Yang, and A. G. Gray, "An Investigation of Practical Approximate Nearest Neighbor Algorithms," in *NIPS*, 2004, pp. 825–832.

[7] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *JACM*, vol. 45, no. 6, pp. 891–923, 1998.

[8] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs," in *ICML*, 2014, pp. 172–180.

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *CACM*, vol. 18, no. 9, pp. 509–517, 1975.

[10] D. Meagher, "Geometric Modeling Using Octree Encoding," *Comp. Graph. and Image Proc.*, vol. 19, pp. 129–147, 1982.

[11] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees," *AURO*, vol. 34, no. 3, pp. 189–206, 2013.

[12] J. Elseberg, D. Borrmann, and A. Nüchter, "One billion points in the cloud – an octree for efficient processing of 3D laser scans," *ISPRS J. of Photogramm. and Rem. Sens.*, vol. 76, pp. 76–88, 2013.

[13] G. M. Morton, "A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing," IBM, Tech. Rep., 1966.

[14] J. Elseberg, S. Magnenat, R. Siegwart, and A. Nüchter, "Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration," *JOSER*, vol. 3, no. 1, pp. 2–12, 2012.

[15] M. Muja and D. G. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration," in *VISAPP*, 2009.

[16] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *ICRA*, 2011.

[17] B. Steder, G. Grisetti, and W. Burgard, "Robust Place Recognition for 3D Range Data based on Point Features," in *ICRA*, 2010, pp. 1400–1405.

[18] D. Munoz, J. A. D. Bagnell, N. Vandapel, and M. Hebert, "Contextual Classification with Functional Max-Margin Markov Networks," in *CVPR*, 2009, pp. 975–982.

[19] R. C. Hoetzlein, "Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids," in *GPU Tech. Conf.*, 2014.

[20] J. L. Bentley, "A Survey of Techniques for Fixed Radius Near Neighbor Searching," Stanford University, Tech. Rep., 1975.

[21] J. L. Bentley and J. H. Friedman, "Data Structures for Range Searching," *CSUR*, vol. 11, no. 4, pp. 397–409, 1979.

[22] P. J. Besl and N. D. McKay, "A Method for Registration of 3-D Shapes," *TPAMI*, vol. 14, no. 2, pp. 239–256, 1992.