Future Generation Computer Systems 175 (2026) 108060

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Cheesemap: A high-performance point-indexing data structure for neighbor search in LiDAR data

Ruben Laso [a],*, Miguel Yermo [b]

[a] Research Group for Scientific Computing, Faculty of Computer Science, University of Vienna, Währinger Straße 29, Vienna, 1090, Vienna, Austria
[b] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidad de Santiago de Compostela, Rúa de Jenaro de la Fuente Domínguez, Santiago de Compostela, 15782, Galicia, Spain

ARTICLE INFO

Keywords:
Point cloud
Data structure
Nearest neighbors
LiDAR

ABSTRACT

Point-cloud data, as the representation of three-dimensional spatial information, is a fundamental piece of information in various domains where indexing and querying these point clouds efficiently is crucial for tasks such as object recognition, autonomous navigation, and environmental modeling. In this paper, we present a novel data structure, `cheesemap`, designed for fast neighbor search in 3D LiDAR point clouds. Points are indexed using a grid of voxels, which can be organized in three different ways, originating three flavors of the `cheesemap`: dense, sparse, and mixed. The lookup of the voxels is theoretically ensured to be performed in constant or amortized constant time, speeding up the search for neighboring points. Experimental results show that `cheesemap` can outperform, in terms of performance and memory footprint, other state-of-the-art data structures both in region-based and k-NN queries throughout different types of point clouds, particularly for Airborne Laser Scanning (ALS) point clouds.

1. Introduction

Point cloud data, representing three-dimensional information through discrete points, has gained significant prominence across various domains, including computer vision, robotics, archaeology, geospatial mapping, and autonomous navigation. The efficient extraction of meaningful information from point clouds is crucial for tasks such as object recognition, scene understanding, and environmental modeling. One of the fundamental operations in point cloud processing is the search for neighboring points around a specific point, as this operation serves as the basis for tasks like segmentation, feature extraction, data mining, convolutions, and clusterization.

The great diversity of data acquisition methods, mainly using Light Detection and Ranging (LiDAR) technology, such as Airborne Laser Scanning (ALS), Terrestrial Laser Scanning (TLS), Mobile Laser Scanning (MLS), or Unmanned Laser Scanning (ULS), has led to the creation of a wide variety of point cloud types in terms of density and distribution. Depending on these characteristics, not all methods for neighborhood search of a given point and not all data structures used to store the point cloud will be equally suitable.

This paper presents a comprehensive comparative analysis of various data structures combined with neighboring search methods across different types of point clouds. Our goal is to provide insights into the strengths, weaknesses, and applicability of these methods in various scenarios. By understanding the performance of neighboring search techniques under various conditions, researchers and practitioners can make informed decisions when selecting an appropriate method for their specific point-cloud processing tasks. The motivation behind our research stems from the observation that state-of-the-art data structures and neighboring search methods may exhibit varying levels of efficiency depending on the nature of the point-cloud data. Factors such as varying point densities, irregular distributions, and high dimensionalities can significantly impact the performance of these methods. Additionally, we propose a novel data structure that aims to optimize the efficiency of neighboring search operations especially (but not limited) for ALS point clouds.

The remainder of this paper is organized as follows: Section 2 reviews related work on data structures and neighboring search methods in point-cloud processing. Section 3 introduces a novel data structure that aims to improve search performance in ALS point clouds. In Section 4, we present the datasets used in our experiments and the experimental setup. Section 5 presents the results of our comparative analysis, highlighting the performance of each data structure under different conditions. Finally, Section 6 concludes the paper with a summary of key observations and directions for future work.

* Corresponding author.
E-mail addresses: ruben.laso.rodriguez@univie.ac.at (R. Laso), miguel.yermo@usc.es (M. Yermo).

https://doi.org/10.1016/j.future.2025.108060
Received 3 April 2025; Received in revised form 25 July 2025; Accepted 27 July 2025
Available online 6 August 2025

2. Related work

Computing neighborhoods in a point cloud is a non-trivial task due to the irregular nature of the data itself. The neighborhood of a given data point is composed of all the surrounding points meeting a certain condition, and it provides information about the local structure of the point cloud, which is essential in its processing [1]. The most common neighboring methods encountered in the state-of-the-art literature are two: the fixed-radius neighborhood [2] and the k-nearest neighbors (k-NN) [3,4]. While the former queries the point cloud to retrieve all the points inside a previously defined kernel, the latter computes the k-nearest points using any valid metric.

Regarding the fixed-radius neighborhoods, the typical approach is to use a spherical neighborhood, composed of all the points whose Euclidean distance to the queried point is less than R. Note that different distances than the L_2 norm can be used, such as the L_1 norm or the L_∞ norm, for generating cubic neighborhoods, for example. Another variation is to ignore the third spatial dimension (typically the vertical, z), producing a cylindrical neighborhood, often used in point clouds obtained from airborne platforms [5]. While these two kernels are the most common, any custom geometry can be defined and used to carry out the queries: cubes, and their 2D counterpart squares, toroids, and so on.

Concerning the k-nearest neighbors, the method needs the parameter k, which is the number of neighbors to be found, and the neighborhood is composed of the k-closest points to the queried point. If the point being queried is located in a low-density area, the k-closest neighbors may include points too far away to be geometrically meaningful. This could be detrimental to the quality of the local descriptors computed on the point, such as linearity, planarity, or eigenentropy. To overcome this issue, some authors use a mixed approach. For example, [6] employs a fixed-radius sphere as a kernel, but only k randomly selected neighbors are included.

Which type of neighborhood to use depends on the application. When processing point clouds, the neighborhood computation is usually the most computationally expensive operation [7], so choosing the correct approach is crucial in terms of accuracy and saving computing time. Both types of search methods depend on a parameter, which must be chosen carefully. Due to the irregular nature of point clouds, a fixed-radius search can be appropriate for some areas of the cloud but may fail in others. This occurs because of the variations of point densities across a single dataset. To avoid this problem, some authors suggest the use of different scales across the dataset to retrieve the neighborhood. For example, [8] presented a way to adapt the query parameter for each point. An improvement of this technique is proposed in [9,10] for k-NN queries and in [11,12] for fixed-radius neighborhoods. Nevertheless, the computational cost is still significant.

To reduce query times, some authors rely on stochastic sampling methods, such as the Poisson Disk Sampling [13], which produces evenly distributed sets of points in a specific region. This is the case of [14], where the random sampling is also accelerated using GPU support to keep a high number of neighbors while reducing the computation times. To compute multiscale neighborhoods in reasonable times, [7] opted to perform a subsampling of the dataset by using a grid, which allows them to control the size of the neighborhoods. In the realm of real-time applications, [15] used a GPU-accelerated algorithm to detect obstacles for mobile vehicles. The hardware accelerated methods are also useful to process massive point clouds, as is the case of [16], where GPUs are used to accelerate the computation of triangular irregular networks.

Notably, the method used in [17] exploits the geometry of the scanlines to retrieve the spherical neighborhood in $\mathcal{O}(1)$. By knowing the characteristics of the sensor, the point cloud is transformed into a convenient space where each point has two indexes: one for the scanline it belongs to and another for its position inside the scanline. Nevertheless, this method is only valid when the sensor is known and its scanning pattern is regular and predictable.

The naïve method for computing the neighborhoods is checking for every point in the dataset whether it meets the conditions to be part of the neighborhood or not. This method could be suitable if the point clouds are composed of a few hundred points, but modern point clouds have millions of points, making this approach unfeasible.

To overcome this issue, several data structures were created to speed up the query times. The k-d tree [18] is a tree-based data structure in which each node has a maximum of two children. In the k-d tree, the data domain is partitioned by using (k-1)-dimensional planes, and each division created by each plane corresponds to a node of the tree. The Octree [19] is also a tree-based data structure, but each node has eight children. The space partitioning in this case is carried out using voxels: The root of the Octree is a voxel whose size's length is the larger edge of the minimum bounding box of the dataset. Each voxel is then subdivided into eight equal voxels by halving its length. The partitioning is usually stopped when a fixed number of data for a given voxel is reached. Other not-so-common data structures are the Ball Tree [20] and the R-Tree [21]. The Ball Tree can be thought of as the same as a k-d tree, but using (k-1)-dimensional spheres instead of planes. The R-Tree is similar to the Octree, but the space partitioning is carried out by computing smaller bounding boxes for each level of the tree. To check whether two elements are nearby, the collision of the bounding boxes is computed, which usually is a computationally cheap operation.

Among the data structures used for neighboring searches in point clouds, the k-d tree and the Octree are the most popular.

The k-d tree is known for its effectiveness in multidimensional data for nearest-neighbor searches [17], having an average complexity of $\mathcal{O}(\log N)$ for finding the nearest neighbor in a point cloud with N points. Nevertheless, this complexity can be limiting [22].

The Octree is widely utilized for organizing point clouds into voxels, simplifying the exploitation of its structure for neighbor searching [23]. It is also used in indexing structures for massive point clouds, enhancing the efficiency of spatial indexing [24], and neighbor searching methods for 3D point clouds, particularly in the context of LiDAR data [25].

Furthermore, in the context of graph-based networks and point cloud analysis, k-d tree is utilized for conducting point-to-node k-nearest-neighbor searches. This allows for the systematic adjustment of the network's receptive field [26]. Additionally, k-d tree is used in the establishment of a three-dimensional point cloud registration based on entropy and particle swarm optimization, where the relationship of points is established by k-d tree for finding the k-nearest neighbor of a point [27]. Octree, on the other hand, has been used in the context of 3D change detection, where it is employed for adaptive thresholds based on local point cloud density [28].

Both k-d tree and Octree play crucial roles in neighbor searches in point clouds, however, to the best of our knowledge, they have been reviewed only twice in the literature. In the work by Elseberg et al. [29], the performance of five implementations of k-d tree, one Octree, and one R-Tree is compared when used for k-nearest neighbors and fixed-radius queries, in the context of shape registration in synthetic and real point-cloud data. The other review is presented by Lawson et al. [30], where a single unstructured mesh composed of 152.7 million nodes and 152.1 million hexahedral elements is used as a study case. In that review, a deep comparison between 20 open-source C/C++ libraries implementing different search methods is presented. However, the study uses synthetic data with an unknown spatial distribution.

While the aforementioned studies presented are exhaustive, some questions remain unanswered: How do the different data structures perform in real-world point clouds? How do they perform in different types of point clouds? Is the performance different in k-NN and fixed-radius queries? Can a novel data structure outperform current state-of-the-art data structures?

In this work, we aim to answer these questions by comparing different methods for performing neighborhood searches, while proposing a new data structure to perform neighbor searches in three-dimensional point clouds.

3. Cheesemap

As an alternative to the proposals above, we present a data structure known as `cheesemap`.[1] and its flavors: dense, sparse, and mixed.

This structure, particularly in its sparse and mixed forms, aims to take advantage of the features of LiDAR point clouds, in which point density is usually not uniform. In ALS point clouds, the range in the vertical z-axis tends to be significantly smaller than in the x- and y-axis. Thus, when indexing the points, the z-axis can be omitted. In other point clouds, like TLS or MLS, and particularly in urban environments, points do not follow a uniform distribution, with the density varying significantly depending on the distance to the sensor and the presence of obstacles.

Note that the `cheesemap` works aligned with the x, y, and z axes. If the point cloud is not aligned with these axes, a rotation could be desirable for using the `cheesemap` efficiently. This is a common limitation in other data structures, such as the k-d tree and the Octree, which are also axis-aligned.

3.1. Notation convention

In the rest of the paper, we use the x, y, and z subscripts to denote the dimension. For example, p_x would reference the x coordinate of point p.

Additionally, we define p^- and p^+ as the corners of the bounding box of the point cloud P, such as:

$$p^- = \left(\min\left\{p_a \mid p \in P\right\},\ a = \{x, y, z\}\right), \tag{1}$$

$$p^+ = \left(\max\left\{p_a \mid p \in P\right\},\ a = \{x, y, z\}\right). \tag{2}$$

3.2. General idea

Internally, the `cheesemap` is based on a two- or three-dimensional grid of voxels, depending on the coordinates used to index the points. The dimensionality of the grid is determined by a template parameter so, for example, `chs::Dense<2>` and `chs::Dense<3>` are two different instantiations of the `cheesemap`, using 2D and 3D grids, respectively. It is the way that the grid is stored in memory that differentiates the three flavors of the `cheesemap` (dense, sparse, and mixed), as explained in Section 3.3.

For the sake of conciseness, explanations in this paper will assume three-axis indexing. Generally, the same calculations are also valid for 2D by ignoring the third axis. In case that the two- and three-dimensional versions of the `cheesemap` require different computations, it would be explicitly stated.

Let s be the size of the voxel[2] and let $B = \left(p^-, p^+\right)$ be the bounding box of the point cloud. Then, a grid G of $\left(n_x, n_y, n_z\right)$ voxels is generated such that,

$$n_a = \left\lfloor \left(p_a^+ - p_a^-\right)/s \right\rfloor + 1,\ a = \{x, y, z\}, \tag{3}$$

$$G := \left\{v_{ijk},\ i = 1, \ldots, n_x,\ j = 1, \ldots, n_y,\ k = 1, \ldots, n_z\right\}. \tag{4}$$

As a consequence of the structure of the grid, given a point p, the indices of its corresponding voxel v_{ijk} are determined by

$$\{i, j, k\} = \left\{\left\lfloor \left(p_a - p_a^-\right)/s \right\rfloor,\ a = \{x, y, z\}\right\}. \tag{5}$$

For clarification, in a two-dimensional version of the `cheesemap`, the grid would be defined as

$$n_a = \left\lfloor \left(p_a^+ - p_a^-\right)/s \right\rfloor + 1,\ a = \{x, y\}, \tag{6}$$

[1] Available at https://github.com/ruben-laso/cheesemap

[2] In the explanation, we assume that the dimensions of the voxel are the same in all directions for the sake of simplicity. In the actual implementation, the user can specify different sizes for each dimension.

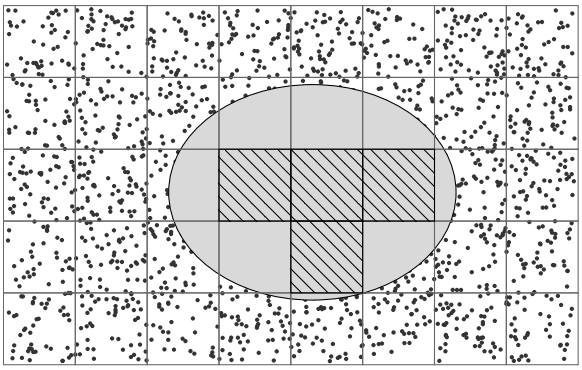

Fig. 1. Example of sparsity in a point cloud. There are no points within the light-gray ellipse (representing, for example, a lake in ALS point clouds), so shaded voxels do not contain any points.

$$G := \left\{v_{ij},\ i = 1, \ldots, n_x,\ j = 1, \ldots, n_y\right\}, \tag{7}$$

and the indices of the voxel v_{ij} containing a point p would be

$$\{i, j\} = \left\{\left\lfloor \left(p_a - p_a^-\right)/s \right\rfloor,\ a = \{x, y\}\right\}. \tag{8}$$

3.3. Flavors of `cheesemap`

The most interesting part of the `cheesemap` lies in the way of storing the voxels and their information. To do so, three variants are proposed: dense, sparse, and mixed.

It is important to note that the three flavors were designed with the objective of providing constant or amortized constant time for the lookup of a voxel.

3.3.1. Dense

The dense representation of `cheesemap` is the most traditional one, where a list of $n_x \times n_y \times n_z$ voxels is built. Storing these voxels is straightforward thanks to the regular structure of the grid. Thus, it is possible to identify voxels by a single global index,

$$g(i, j) = in_y + j, \tag{9}$$

$$g(i, j, k) = i(n_y n_z) + j(n_z) + k, \tag{10}$$

for the two- and three-dimensional cases, respectively.

By implementing the grid G as a one-dimensional array (using `std::vector`), the global index g can be used to access the voxel v_{ijk} in constant time. This is ensured, as the C++ standard library guarantees that a random access to a `std::vector` should be performed with complexity $\mathcal{O}(1)$.

3.3.2. Sparse

The sparse representation stores only non-empty voxels (those containing at least one point).

Imagine, for example, a point cloud like the one shown in Fig. 1 where there is a big empty region such as a mass of water in an ALS point cloud. Those voxels fully contained within that region do not need to be stored as they contain no points.

Therefore, the non-empty voxels are stored in an `std::unordered_map`, which is a collection of key–value pairs. The key is the global index of the voxel, computed using Eqs. (9) and (10) for the two- and three-dimensional cases, respectively, and the value is the voxel itself. Since the C++ Standard Library guarantees that a random access to an `std::unordered_map` should be performed in amortized constant time, $\mathcal{O}(1)$, this allows for efficient lookups of voxels.

3.3.3. Mixed

The mixed representation combines the advantages of both dense and sparse representations by adaptively dividing the point cloud into slices, where each slice can be either dense or sparse depending on its point distribution.

The point cloud is divided into n_z slices through the z-axis. Since the z-axis is typically the smallest dimension, fewer slices are generated by splitting the point cloud in this direction, speeding up the lookup process. Note that a smaller s_z can be used to generate more slices if needed.

Another advantage of slicing along the z-axis is that, in LiDAR point clouds, horizontal planes near the ground typically have a higher density than those at higher elevations. This way, those slices that are closer to the ground could take advantage of a faster lookup with the dense representation, while those above the ground would use a more memory-efficient indexing method with sparse indexing.

At first, all slices are represented in sparse form, in the same way as shown in Section 3.3.2. As points are added to the voxels of each slice, the number of non-empty voxels is counted. Once a slice has enough non-empty voxels, it is turned into a dense slice. By default, this conversion happens when 80% of the voxels have points. This optimizes memory usage for low-density slices, and lookup speed for highly-populated slices.

When looking up for a voxel v_{ijk}, two steps are performed: first, its slice is retrieved using the k index; and second, the voxel is retrieved from the slice using the i and j indices.

In the 3D mixed `cheesemap` implementation, a `std::vector` of slices is used. Therefore, the lookup of a voxel v_{ijk} is still performed in constant or amortized constant time. First, the retrieval of the slice is done with complexity $\mathcal{O}(1)$ (random access to a `std::vector`). Second, the voxel is retrieved in $\mathcal{O}(1)$ from a dense slice, or in amortized $\mathcal{O}(1)$ from a sparse slice, as explained in Sections 3.3.1 and 3.3.2, respectively.

Note the corner case of the 2D mixed `cheesemap`, where a single slice is generated ($n_z = 1$). In this case, the slice follows the same rules as in the 3D case: once it has sufficient non-empty voxels, it is changed from the sparse to the dense representation.

3.4. Neighborhood search

Thanks to the structured grid of the `cheesemap`, queries can be performed efficiently, as the time complexity to find the voxel containing a point is $\mathcal{O}(1)$ for the dense `cheesemap`, and amortized $\mathcal{O}(1)$ for the sparse and mixed versions. In this section, we delve into the implementation of the two most common operations: kernel-based and k-nearest neighbors search.

3.4.1. Kernel-based search

Imagine a spherical kernel[3] defined by the ball $B(c, r)$, centered at c with radius r. The first step is to compute the bounding box of the kernel, $\left(p^-_{\text{ker}}, p^+_{\text{ker}}\right)$. In our example:

$$p^-_{\text{ker}} = \left(c_x - r, c_y - r, c_z - r\right), \tag{11}$$

$$p^+_{\text{ker}} = \left(c_x + r, c_y + r, c_z + r\right). \tag{12}$$

With the bounding box, it is possible to determine which voxels should be explored (as they may intersect with the kernel) using Eq. (5). An example of the intersection region is shown in Fig. 2.

After calculating the intersection region, it is only a matter of traversing through the points in those voxels, and returning the list of points within the search kernel.

Note that the same procedure can be applied to any kernel shape, as long as the bounding box is provided. In the code, the kernel must comply with a C++20 concept that requires two member functions: `box()`, returning the bounding box of the kernel, and `is_inside()`, determining whether a point lies within the kernel.

[3] The same would apply to other kernels like cubic or cylindrical.

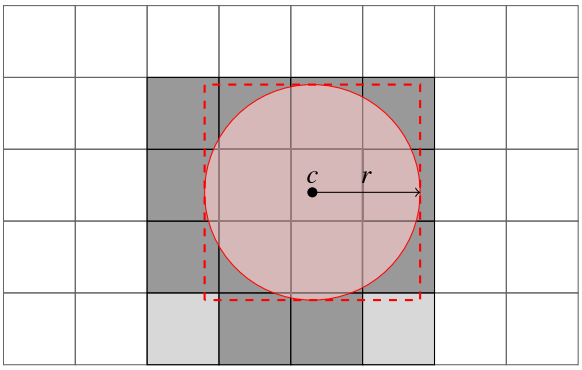

Fig. 2. Example of kernel-based search. The red circle represents the kernel, a sphere centered in c with radius r. The red dashed box represents the bounding box of the sphere. In light gray, the voxels that intersect with the bounding box, and may intersect with the kernel. In dark gray, those voxels that actually intersect with the sphere.

The voxels are implemented as `std::vector` of pointers to points, so the points are not copied when the voxel is created. That means that checking if all the points in a voxel are inside the kernel is done in linear time, $\mathcal{O}\left(N_v\right)$, where N_v is the number of points in the voxel. It is expected that N_v is small, as the cell size s should be chosen appropriately. The impact on memory usage and performance of this choice is discussed in Sections 5.1 and 5.2.

Algorithm 1 provides the pseudo-code of the kernel-based search.

Algorithm 1 Kernel-based neighborhood search

Require: Function `is_inside`(p) returning true is p is inside the kernel.

$P_{\text{neigh}} = \varnothing$ ▷ *List of neighbors.*
$\left(p^-_{\text{ker}}, p^+_{\text{ker}}\right)$ = kernel bounding box
(i^-, j^-, k^-) = indices such $p^-_{\text{ker}} \in v_{ijk}$
$\left(i^+, j^+, k^+\right)$ = indices such $p^+_{\text{ker}} \in v_{ijk}$
for $(i, j, k) = \left(i^-, \dots, i^+\right) \times \left(j^-, \dots, j^+\right) \times \left(k^-, \dots, k^+\right)$ **do**
  if $\exists v_{ijk}$ **then**
    for all $p \in v_{ijk}$ **do**
      if `is_inside`(p) **then**
        $P_{\text{neigh}} := P_{\text{neigh}} \cup \{p\}$
return P_{neigh}

3.4.2. k-NN search

When looking for the k-NN, two kinds of points are considered:

- Candidate points: points whose distance to the center point has already been calculated.
- Non-visited points: points whose distance has not yet been computed.

Thus, a set P_{cand} is used for the candidate points, and a subset of it, $\hat{P}_{\text{cand}}$, which stores the points that are within a distance r from the center point c:

$$\hat{P}_{\text{cand}} = \left\{p \in P_{\text{cand}} \mid d(p, c) \le r\right\}. \tag{13}$$

Additionally, a set of voxels named T is defined and serves as a taboo list of already visited voxels to avoid redundant calculations.

To find the k-nearest neighbors of a point c, a two-phase iterative search is performed where each iteration is similar to a spherical kernel-based search with an increasing search radius r. In each iteration, the voxels that intersect with the search region $B(c, r)$ are visited, the points contained in those voxels are added to P_{cand}, and r is updated. Because

Algorithm 2 k-NN search

Require: Number of neighbors k.

Require: Point c used as center of the k-NN neighborhood.

Require: Function $d(p, q)$ that returns the distance between points p and q.

$P_{\text{cand}} = \varnothing$ ▷ *List of points whose distance to c has been calculated*
$\hat{P}_{\text{cand}} = \varnothing$ ▷ *Points of P_{cand} within the search radius. Implemented as a priority queue.*
$T = \varnothing$ ▷ *Taboo list for the already computed voxels*
$r = \min\left\{|c_a - p^-_{v,a}|, |c_a - p^+_{v,a}|,\ a = \{x, y, z\}\right\}$ ▷ *Initial radius*
while $|\hat{P}_{\text{cand}}| < k$ and $T \neq G$ **do** ▷ *$\hat{P}_{cand}$ defined in Eq. (13)*
  $\left(p^-_{\text{ker}}, p^+_{\text{ker}}\right)$ = bounding box of $B(c, r)$
  (i^-, j^-, k^-) = indices such $p^-_{\text{ker}} \in v_{ijk}$
  $\left(i^+, j^+, k^+\right)$ = indices such $p^+_{\text{ker}} \in v_{ijk}$
  for $(i, j, k) = \left(i^-, \ldots, i^+\right) \times \left(j^-, \ldots, j^+\right) \times \left(k^-, \ldots, k^+\right)$ **do**
    if $\exists v_{ijk}$ and $v_{ijk} \notin T$ **then**
      $T = T \cup \left\{v_{ijk}\right\}$
      for all $p \in v_{ijk}$ **do**
        $P_{\text{cand}} := P_{\text{cand}} \cup \{p\}$
  $\hat{P}_{\text{cand}} = \left\{p \in P_{\text{cand}} \mid d(p, c) \leq r\right\}$ ▷ *Update $\hat{P}_{cand}$*
  if $\hat{P}_{\text{cand}} \neq \varnothing$ **then**
    $\rho = \frac{|\hat{P}_{\text{cand}}|}{(4/3)\pi r^3}$ ▷ *Density-based growth*
    $r = (k/\rho)^{1/3}$
  else
    $r = r + s$ ▷ *Monotonic growth*
return $\left\{p_i \in \hat{P}_{\text{cand}} \mid i \leq \min\left\{|\hat{P}_{\text{cand}}|, k\right\}\right\}$ ▷ *$\hat{P}_{cand}$ is already sorted by distance (closest first).*

of the taboo list T, each voxel is visited only once, so the distances of its points to c are calculated a single time. The search stops when the condition $|\hat{P}_{\text{cand}}| \geq k$ is met.

In the first phase, a monotonic growing search is performed to find the initial neighbors. The starting radius r corresponds to the closest distance from c to the wall of its voxel:

$$r = \min\left\{|c_a - p^-_{v,a}|, |c_a - p^+_{v,a}|,\ a = \{x, y, z\}\right\}, \tag{14}$$

where p^+_v and p^-_v are the delimiting points of the voxel. After the initial iteration, the radius is continuously increased by the size of the voxel s until the search region is not empty and $\hat{P}_{\text{cand}} \neq \varnothing$.

In the second phase, the density of $\hat{P}_{\text{cand}}$ is used to estimate how large r should be to find k neighbors:

$$\rho = \frac{|\hat{P}_{\text{cand}}|}{(4/3)\pi r^3}, \tag{15}$$

$$r = \left(\frac{k}{\rho}\right)^{1/3}. \tag{16}$$

If the density does not change between two consecutive iterations, the radius is updated by $r = r + s$. This second phase is repeated until we find k-nearest neighbors, $|\hat{P}_{\text{cand}}| = k$, or all the voxels have been visited, $T = G$.

Algorithm 2 shows the pseudocode of the k-NN search.

3.5. Further remarks

As part of the design and implementation of the `cheesemap`, some details have been considered to enhance its performance.

Reordering. To enhance the cache-friendliness of the `cheesemap`, the points of the dataset can be reordered according to the voxel they belong to (see Eqs. (9) and (10)). Then, when iterating over the points of a voxel, their memory addresses should be closer or even consecutive, improving the cache hit rate and speeding up the query times.

Voxels in dense `cheesemap`. Since all voxels are created in the dense version of `cheesemap`, it is not necessary to check if a certain voxel exists (Lines ?? and ?? of Algorithms 1 and 2, respectively).

Taboo list in k-NN. As mentioned in Section 3.4.1, the taboo list not only serves to obtain correct results (avoiding duplications) but also affects performance by preventing the repetition of calculations. Because of the structured grid and the growing radius, the implementation of the taboo list can be optimized by storing only the indices (i^-, j^-, k^-) and $\left(i^+, j^+, k^+\right)$ of the voxels that correspond to the bounding box of the last search region. In the next iteration, only the voxels with indices that do not belong to the last $(i^-, i^+) \times (j^-, j^+) \times (k^-, k^+)$ region need to be visited. This optimization reduces both memory consumption and execution time, as only two tuples of indices are stored instead of a (potentially large) list of voxels, and the comparison of the indices is faster than the lookup in a list.

Candidate list in k-NN. In the current implementation, both sets P_{cand} and $\hat{P}_{\text{cand}}$ are stored in the same list, using a custom data structure built on top of a `std::vector`. The container is designed to store pairs of points and distances, $(p, d(p, c))$, and its elements are sorted by distance. When inserting a new point, a binary search is performed to find the correct position in the list. Then, the point is inserted and the elements with a greater distance are shifted one position to the right. When the list is full (it has more than k elements), the last element is removed. Despite the worst-case complexity of this operation being $O(n)$, this is not the case in practice. The reason is that, with an increasing-radius search, the new points to be inserted are further and further away from the center, so their position in the list is closer to the end, and the number of elements to be shifted is small (if any at all).

Density-based growth in k-NN. Experimentally, it has been observed that the growth of the search radius using the density of the neighbors is more efficient than monotonic growth. In a synthetic dataset of 1 million random points, the search radius increased by density reduced the number of explored voxels in 84.5 % of the cases, while a monotonic growth was better in only 3.6 % of the cases. In the remaining 11.9 % of the cases, both methods were equivalent. Consequently, the algorithm is slightly faster when using density-based growth.

Table 1
Characteristics of the datasets used in the comparison. The sensor information used in the Semantic3D dataset is not publicly available.

Name	Type	Scene	Sensor	Points	Extension (m^2)	Density (pts/m^2)	Weighted density (pts/m^2)
5080_54400	ALS	Urban	Riegl Q1560	12 219 779	250 000	48.87	61.00
5110_54320	ALS	Urban	Riegl Q1560	17 747 769	250 000	70.99	69.93
5110_54475	ALS	Rural	Riegl Q1560	11 981 458	250 000	47.93	20.68
5130_54355	ALS	Rural	Riegl Q1560	12 148 800	250 000	48.60	57.90
Lille	MLS	Urban	Velodyne HDL-32E	80 000 000	798 000	100.27	49.13
Paris	MLS	Urban	Velodyne HDL-32E	50 000 000	48 000	1030.83	42.78
Speulderbos	ULS	Forestry	Riegl VUX-1UAV	79 221 314	33 000	2378.46	109.80
hallway6	TLS	Indoor	Matterport Camera	3 840 664	151	25 419.88	120.48
sg28_2	TLS	Rural	–	170 158 281	262 000	649.85	101.67
sg28_4	TLS	Urban	–	258 720 948	50 000	5181.00	73.24

4. Experimental setup

In this section, we describe the datasets used, the machines where the experiments will be conducted, and the metrics used to evaluate the performance of the data structures.

4.1. Datasets

We chose a set of point clouds representative of real-world applications to compare various state-of-the-art data structures and their implementations by different authors. Our goal was to choose datasets that are open, publicly available, and widely used in the literature. For this purpose, we have selected several datasets representing different environments for each type of platform. The density of each dataset is computed as the ratio of the total number of points to the area of the bounding box. For the weighted density, we calculate the local density of each point as the ratio of its number of neighbors to the area of its neighborhood. In this study, a 2D circular neighborhood with a radius of 1 meter is used. A histogram with 256 bins is then created, with each bin representing an integer value of density. The weighted density is obtained by averaging the local densities, where each local density is weighted according to its frequency in the histogram. Table 1 shows a summary of the dataset used in this study.

The DALES [31] dataset is a well-known benchmark for evaluating deep learning classification models acquired using airborne laser scanning. The dataset is split into squared chunks of 250 000 m^2. Four chunks were selected representing different areas: a residential area with low buildings (*5080_54400*), a downtown area with skyscrapers (*5110_54320*), an isolated area with high-voltage powerlines (*5110_54475*), and a rural area with high vegetation (*5130_54355*).

Mobile laser scanning is often used to capture urban environments, which are well represented by the Paris-Lille-3D [32] dataset, from which we used the scenes of *Paris* and *Lille 2* (referred to as *Lille* for simplicity).

To the best of our knowledge, there are no reference benchmark datasets for unmanned laser scanning. We have chosen the *Speulderbos* [33] dataset, which contains around 80 M points, despite covering a relatively small area (0.018 km^2). Its utility lies in analyzing the behavior of data structures in extremely dense point clouds.

Regarding terrestrial laser scanning, two datasets were chosen. As an example of an indoor point cloud taken from the S3DIS [34] dataset, we have chosen the one containing the largest number of LiDAR points: *hallway6*. For outdoor areas, two scenes from the Semantic3D [35] dataset were selected. The *sg28_2* scene corresponds to a farm in a rural area, and *sg28_4* is a point cloud representing an urban environment. Other scenes in this dataset correspond to streets, which are better represented in mobile laser scanning datasets. The sensor information used for the acquisition of this dataset was not publicly disclosed.

The datasets are distributed in LAS, ASCII, and binary PLY formats. All were converted to the standard LAS format.

4.2. Tested libraries

To assess the performance of the `cheesemap`, we have compared it against several state-of-the-art libraries and data structures. These libraries were chosen based on their popularity and the availability of their source code. Below, we present the libraries used in the comparison:

- The `nanoflann` library [36] is a C++ header-only library based on the original FLANN [37], optimizing the performance of queries using a *k*-d tree. In the results, we will refer to this library as `nanoflann::KdTree`.
- The PCL library [38] is a widely-used library for point cloud processing. It includes several data structures for point cloud indexing, such as the *k*-d tree and the Octree, which will be noted as `pcl::kdtree` and `pcl::octree`, respectively.
- The implementation of the Octree by Behley et al. at the University of Bonn [19], which is a reference in the field of point cloud processing. This implementation will be referred to as `unibn::Octree`.
- The implementations of the `cheesemap`, which will be referred to as `chs::Dense`, `chs::Sparse`, and `chs::Mixed`. Dimensionality is noted within angle brackets, e.g., "`chs::Dense<3>`" refers to the 3D dense `cheesemap`.

4.3. Execution environment

The experiments were conducted on a NUMA system with four Intel Xeon E5-4620 v4 processors at 2.1 GHz and 10 physical cores each. Hyperthreading was disabled during the experiments. The total installed memory was 256 GiB DDR4. The operating system used was AlmaLinux 8.6 with kernel 4.18.0.

4.4. Methods

To compare the performance of the different data structures, we designed a benchmark[4] to measure the time needed to perform a spherical-kernel search, a cubic-kernel search, and a *k*-NN search. To that end, we considered the following radii: 0.5 m, 1.0 m, 2.0 m, 3.0 m, 5.0 m, 7.5 m and 10.0 m, and the following number of neighbors: 5, 10, 20, 30, 40 and 50.

Each query consisted of selecting a random point from the point cloud and performing the search with the specified radius or number of neighbors. Using Google's benchmark library, each experiment ran for 1 s, performing as many queries as possible. For example, if a query takes on average 0.1 ms, approximately 10 000 different queries will be performed. The limit of 1 s was deemed sufficient to obtain reliable average times, since each query is expected to take between µs and ms.

[4] Available at https://github.com/ruben-laso/pc-structs-comparison.

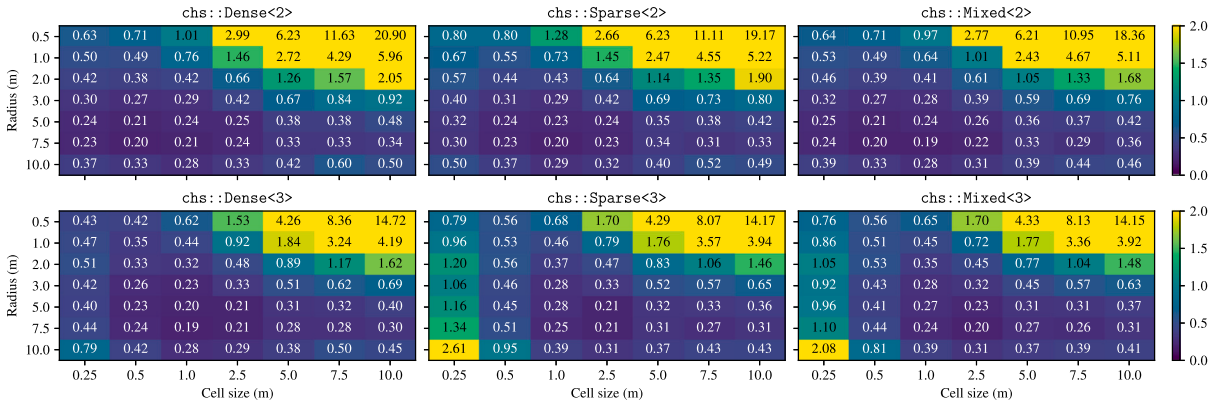

Fig. 3. Geometric average of the normalized search time of the `cheesemap` against `nanoflann::KdTree` in the spherical-kernel search. Lower is better.

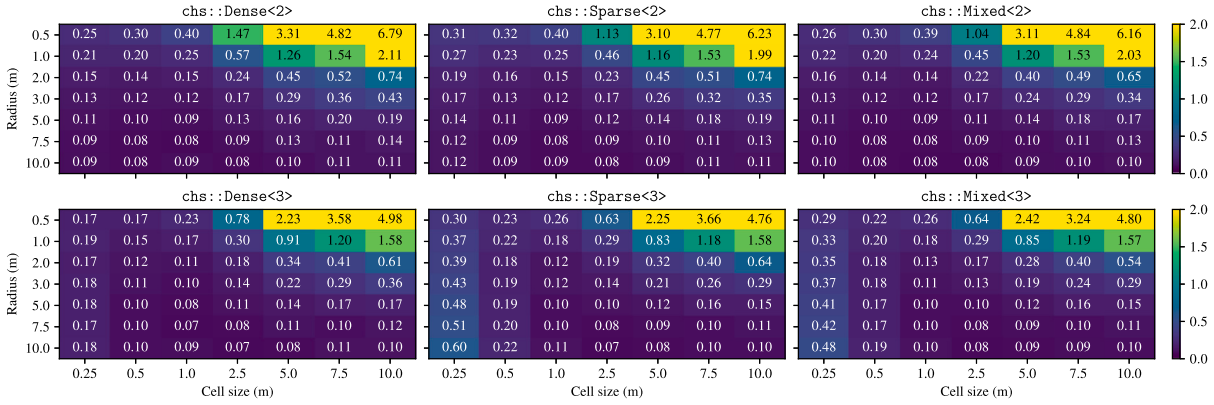

Fig. 4. Geometric average of the normalized search time of the `cheesemap` against `nanoflann::KdTree` in the cube-kernel search. Lower is better.

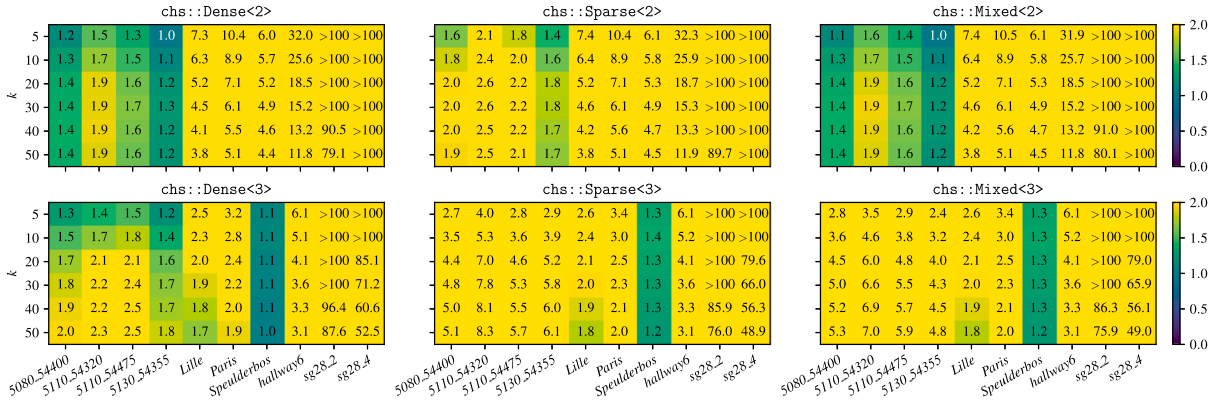

Fig. 5. Normalized search time of the `cheesemap` (with cell size 0.25 m) against `nanoflann::KdTree` in the k-NN search. Lower is better.

The memory footprint of the different data structures was also measured, for which we used Malt [39].

As mentioned in Section 3, some data structures (including our proposal) could benefit from applying certain rotations or transformations to the point cloud. However, we decided to avoid these optimizations to ensure a fair comparison between the different structures.

5. Results and discussion

The results of the experiments are divided into two sections, one analyzing the performance of the `cheesemap` and its parameters, and another comparing the `cheesemap` with other state-of-the-art data structures.

5.1. Performance of `cheesemap`

First, we analyze how the performance of the dense, sparse, and mixed implementations of the `cheesemap` varies across different datasets and parameters.

For the sake of brevity and simplicity, the reported performance metrics correspond to the geometric average of the speedup of the `cheesemap` against `nanoflann::KdTree` across all datasets described in Table 1. The geometric average is used as it is the preferred mean when averaging normalized data [40]. Here, `nanoflann::KdTree` serves as a reference since it is one of the fastest libraries for point cloud indexing and implements all query types

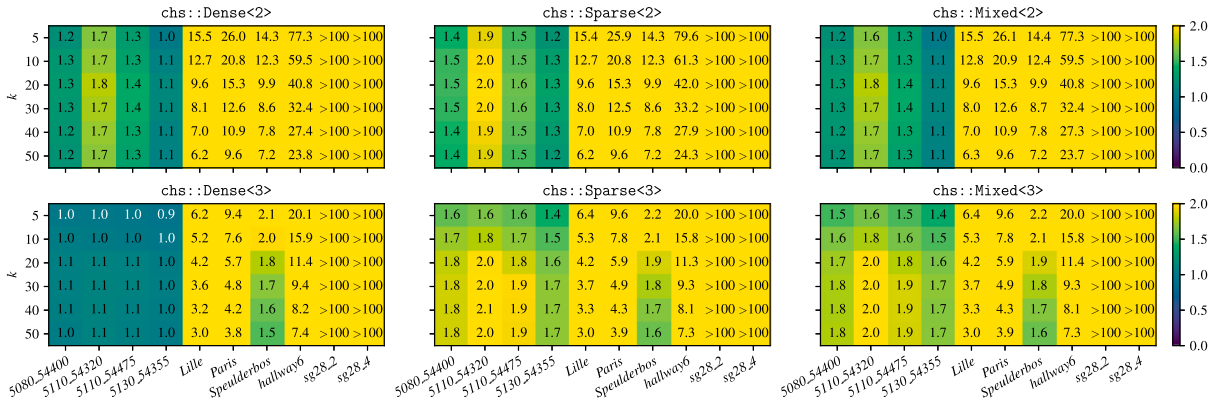

Fig. 6. Normalized search time of the `cheesemap` (with cell size 0.50 m) against `nanoflann::KdTree` in the k-NN search. Lower is better.

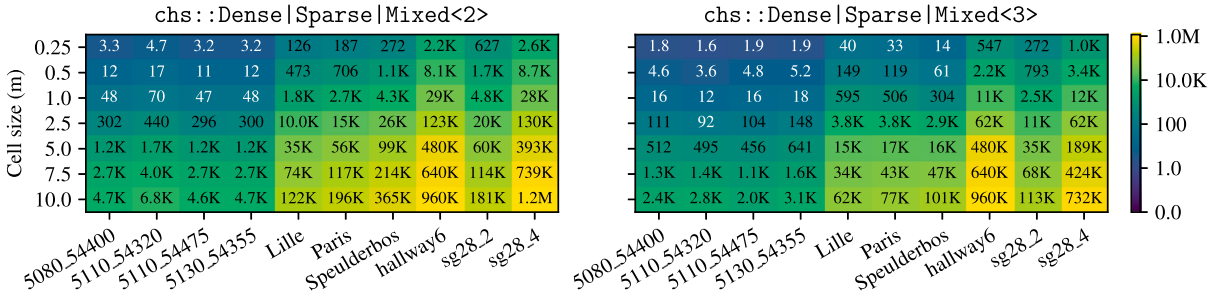

Fig. 7. Points per non-empty cell for several datasets and cell sizes. This value is the same for `chs::Dense`, `chs::Sparse`, and `chs::Mixed`, as it only changes with the cell size, and dimensionality (2D or 3D).

used in this study. Figs. 3 to 6 show the results for the spherical-kernel search, cubic-kernel search, and k-NN search times, respectively. Additional statistics, like number of stored voxels and average number of points per non-empty voxel, are shown in Figs. 7 and 8.

The results show that several parameters affect the performance of the `cheesemap`: the cell size (0.25 m, 0.5 m, 1.0 m, 2.5 m, 5.0 m, 7.5 m, or 10.0 m), the dimensions of the grid (2 or 3), and the type of kernel (spherical, cubic, or k-NN).

Results enabling the reordering of the points were obtained, but not shown in the figures for the sake of brevity.

Cell size. The cell size is a key parameter in the performance of the `cheesemap`.

Smaller cell sizes generally perform better, due to a higher grid resolution (more accurate indexing), which reduces the number of points per voxel, as shown in Fig. 7. This implies a reduction in the time spent computing distances of points that might not be within the search kernel. However, higher resolution comes at the expense of a higher memory footprint, as the number of voxels increases, especially for the `chs::Dense` (see Figs. 8 and 9). According to results, the best performance is found with 10 to 100 points per voxel.

Dimensionality. We observed that the performance of the `cheesemap` is highly dependent on the number of dimensions of the grid. Generally, the performance is better in 3D than in 2D, at the expense of a higher memory footprint. This is especially noticeable when the cell size is small, as the number of voxels increases (see Fig. 8). It is worth noting that the differences are smaller in the ALS datasets, where the z dimension is usually smaller than the x and y dimensions (see Figs. 12 to 14) and it is not usual to have vertical clusters of points.

Reordering of the points. The effect of this operation is highly dependent on the point cloud being used. It was observed that it is slightly beneficial for small point clouds, but its impact becomes negligible for larger point clouds.

Kernel-based search. The type of query does not significantly affect the performance of the `cheesemap`. The results show that query times are similar for both spherical and cubic kernels, with the cubic kernel performing slightly better. Search radius affects the performance, resulting in higher query times as the search radius increases since more voxels need to be visited. It is worth noting that search radii and cell sizes are interrelated, as indexing granularity plays a key role in the `cheesemap`'s performance. For example, performance degradation is observed with small radii and large cell sizes.

k-NN search. The k-NN search exhibits different behavior from the kernel-based search, with point density being a key factor. As shown in Figs. 5 and 6, the `cheesemap` is competitive with (and sometimes superior to) `nanoflann::KdTree` for ALS datasets. However, query times are sensitive to the cell size, requiring fine-tuning for optimal results. This is particularly evident in TLS datasets, where the `cheesemap` performs poorly compared to `nanoflann::KdTree`, necessitating specific cell sizes for better efficiency.

5.2. Memory footprint of `cheesemap`

Besides the execution time, memory footprint is a key aspect to consider in a data structure. We consider the memory footprint as the contribution of two elements: (1) The memory needed to store the pointers to each point of the dataset, which scales with the dataset size; (2) The additional memory required by the point-indexing data structure, which depends on its memory efficiency.

We consider component (1) unavoidable and thus we treat it as the theoretical minimum, calculated assuming an ideal data structure that only needs pointers of b bytes to the N points of the dataset. In a 64-bit system, the expected minimum footprint would be $8N$ bytes. Component (2) measures the overhead of the indexing, which in the case of the `cheesemap` is the memory needed to store the voxels.

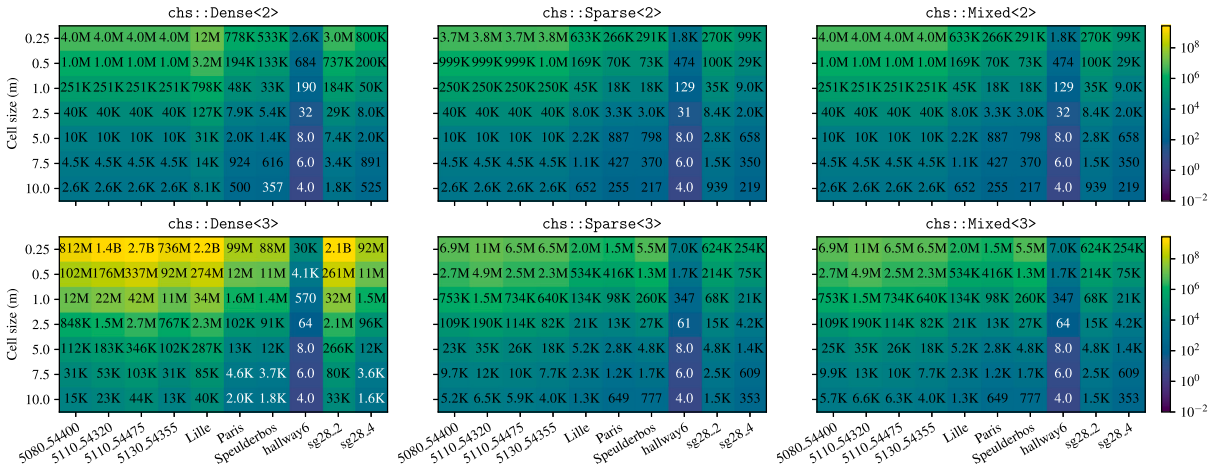

Fig. 8. Number of stored cells per dataset and cell size. For the `chs::Dense<2>` and `chs::Dense<3>`, it corresponds to the total number of voxels of the grid; for the `chs::Sparse<2>` and `chs::Sparse<3>`, to the number of non-empty voxels; the `chs::Mixed<2>` and `chs::Mixed<3>` sit in the middle.

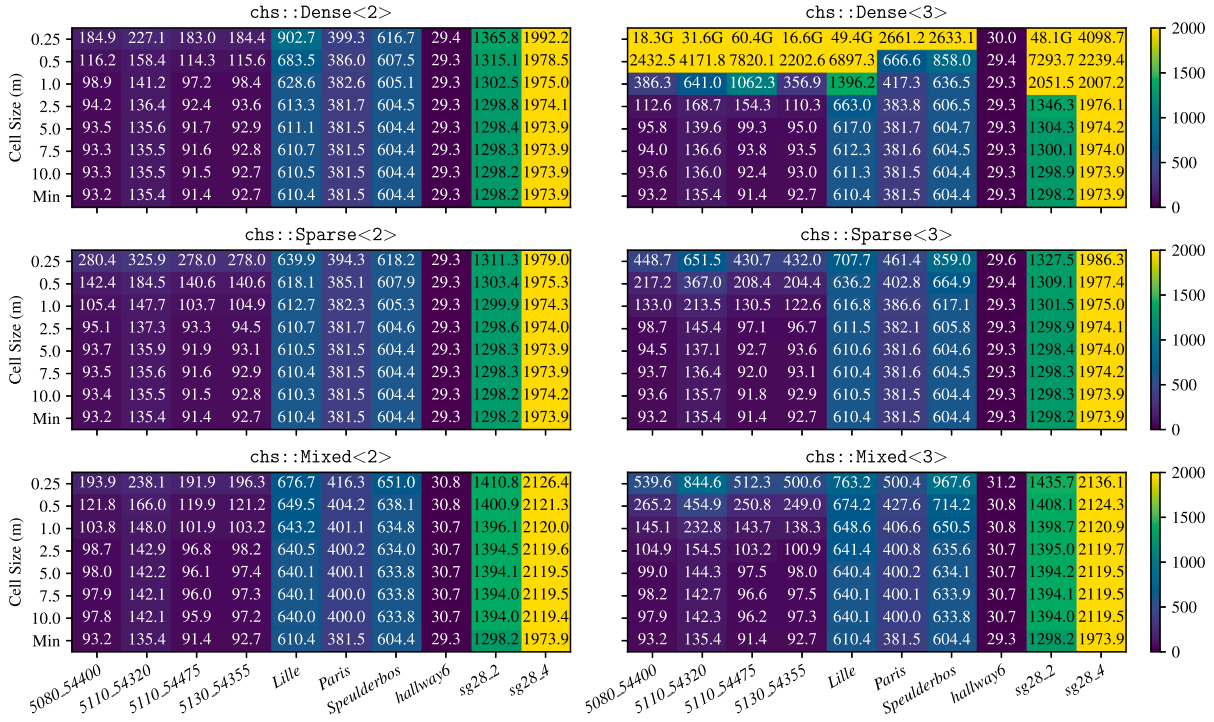

Fig. 9. Memory footprint (MiB) of the different `cheesemap` structures for different cell sizes in each dataset and theoretical minimum calculated as the number of points multiplied by the size of a pointer (assumed 8 B).

Fig. 9 shows the memory footprint of the different types of `cheesemap` and cell sizes for the datasets described in Table 1, as reported by Malt [39], and the theoretical minimum.

Provided that the different types of `cheesemap` have to store the same number of points, the difference in memory footprint is due to the number of stored voxels, as shown in Fig. 8.

The difference between the two- and three-dimensional versions of the `cheesemap` is clear, with the memory footprint increasing as the cell size decreases. This difference is more noticeable in the `chs::Dense` case, since all the bounding box is filled with voxels.

Regarding the different types of `cheesemap`, the largest memory footprint is found in the `chs::Dense` version. This is expected since all the space is filled with voxels, even if they are empty. Specifically, when the cell size is less than 1.0 m, the memory consumption explodes in the three-dimensional version of `chs::Dense`. Nevertheless, the query performance in those cases does not justify the increase in memory consumption, so the use of `chs::Dense<2>`, `chs::Mixed`, or `chs::Sparse`, is recommended for the smallest cell sizes. The `chs::Sparse` version has a lower memory footprint since only the voxels that contain points are created. It is worth noting that the `chs::Mixed` is always close to the most memory-efficient version, meeting its design goal. In some cases, the memory footprint of the `chs::Mixed` is lower than the `chs::Sparse` because the `chs::Mixed` stores some slices of the grid in a dense way. This is better, memory-wise, than using the sparse representation with a high density of non-empty voxels.

The larger the cell size, the smaller the difference in memory footprint between the two- and three-dimensional versions of the `cheesemap`, since typically the extent of the z dimension is negligible compared with the other two dimensions. This is especially true for the ALS datasets.

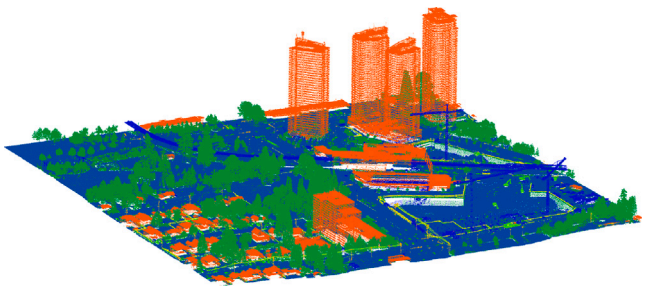

Fig. 10. Point cloud of *5110_54475* dataset.

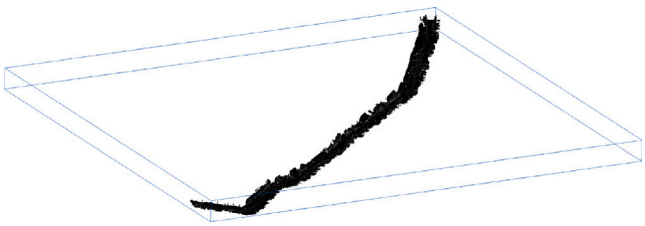

Fig. 11. Point cloud of *Lille* dataset.

There are two datasets where the difference between `chs::Dense<2>` and `chs::Dense<3>` is more evident: *5110_54475* and *Lille*. These datasets show some peculiarities that are worth mentioning.

In the case of *5110_54475*, the memory footprint of the `chs::Dense<3>` case is an order of magnitude higher than the corresponding two-dimensional case. As shown in Fig. 10, the presence of skyscrapers in the scene causes the z dimension of the bounding box to be large compared to the other datasets extracted from DALES. Because of this, the volume of the bounding box is large, and the number of voxels created increases accordingly. Quantitatively, the dimensions of the bounding box are $500\,\text{m} \times 500\,\text{m} \times 168.4\,\text{m}$. When the cell size is 1 m, the `chs::Dense<3>` creates 42.42×10^6 voxels, of which 98.25 % are empty. In the case of `chs::Dense<2>`, the number of created voxels is 251×10^3, of which only 428 (0.17 %) are empty.

As for the *Lille* dataset, the memory footprint in the `chs::Dense<3>` case is about 2.27 times higher compared to the `chs::Dense<2>` case. A representation of the point cloud with the corresponding bounding box is shown in Fig. 11. As we can observe, the *Lille* dataset represents a long, quasi-linear street. The specific alignment of the dataset with the axes results in the largest possible bounding box among all orientations. Consequently, the vast majority of the voxels are empty for this dataset when a cell size of 1.0 m is used. In the `chs::Dense<2>` case, 798.43×10^3 voxels are created, of which 94.31 % are empty. In the `chs::Dense<3>` case, 34.33×10^6 voxels are created, with 34.20×10^6 empty voxels, corresponding to 99.62 % of the total voxels.

5.3. Comparison against other data structures

In this section, we compare the performance of the `cheesemap` against other data structures. Regarding the `cheesemap`, we use a cell size of 1.0 m as it provides a good performance without excessively compromising memory overhead. The rest of the data structures are used with their default parameters. Note that different parameters (e.g., the cell size) could be used for different datasets but, for fairness, we have used the same parameters across all datasets. Similarly, reordering of points is not used in the `cheesemap` to ensure a fair comparison.

Results for spherical-, cubic-kernel search, and k-NN search are shown in Figs. 12–14, respectively. The memory footprint of the different data structures is shown in Table 2.

Kernel search. In the fixed-radius search, two scenarios can be observed. For small radii, the performance is highly dependent on the indexing structure and how fast a structure can retrieve points within a small region. For large radii, the performance depends more on how fast the structure can traverse its internal structure to find points within the radius. The `pcl::octree` exemplifies this phenomenon since for small radii it is the slowest data structure, but as the radius increases it becomes competitive with other structures. In general, most structures perform similarly for small radii, but `cheesemap` variants become more advantageous as the radius increases.

Additionally, some observations can be made regarding different dataset types. For ALS point clouds with spherical- and cubic-kernel search, the `chs::Dense<3>` is the fastest data structure, followed by `chs::Sparse<3>` and `chs::Mixed<3>`. From the other data structures, `unibn::Octree` shows the best query times in most cases, and particularly for big radii.

For MLS point clouds (*Lille* and *Paris*), the `unibn::Octree` and the three-dimensional versions of `cheesemap` are the fastest data structures, showing similar efficiency for small radii. However, the `cheesemap` outperforms for larger radii.

In the *Speulderbos* dataset (ULS), there is a significant difference between two- and three-dimensional `cheesemap` versions, with the latter performing almost on par with `unibn::Octree` and `nanoflann::KdTree`. The z dimension is significant due to tree presence, causing many points to stack along the vertical axis. Therefore, considering the z axis benefits the `cheesemap` for this dataset.

Unlike other datasets, for the *hallway6* dataset (TLS), the `cheesemap` is slower for very small radii queries but becomes the fastest as the radius increases. This results from the small dimensions (and high density) of the dataset, where a 1.0 m cell size is too large for good performance with small radii queries. When the radius exceeds the cell size, the three-dimensional `cheesemap` versions become the fastest structures.

Finally, for the *sg28_2* and *sg28_4* datasets (TLS), the `cheesemap` is the fastest data structure, with the `chs::Mixed<3>` being the most efficient version. These results outline the benefits of the `cheesemap`, considering that these are the largest datasets in terms of the number of points.

k-NN Search. Before commenting on the results, it should be mentioned that the `unibn::Octree` does not support k-NN search, so it is not included in the comparison.

As mentioned in Section 5.1, the `cheesemap` shows notable differences between datasets for k-NN search. For ALS datasets, the `cheesemap` performs on par with `nanoflann::KdTree` and `pcl::kdtree`, with `nanoflann::KdTree` and `chs::Dense<3>` being slightly faster than the others. For the rest of the datasets, `nanoflann::KdTree` is the fastest data structure, closely followed by `pcl::kdtree`. The `cheesemap` is significantly slower, but still faster than the `pcl::octree`, which is the slowest data structure across all point clouds.

These results highlight the importance of point density in the performance of the `cheesemap`, pointing to the need for fine-tuning the cell size for specific kinds of point clouds.

Memory footprint. As mentioned in Section 5.2, the memory footprint of indexing data structures is composed of two elements: the memory needed to store the pointers to the points and the memory needed to store the data structure itself. Table 2 shows the memory footprint of the different data structures as reported by Malt [39] alongside the theoretical minimum.

The results show that two structures have the lowest memory footprint: `nanoflann::KdTree` and `chs::Dense<2>`. All two-dimensional versions of the `cheesemap` have a memory footprint close to the theoretical minimum.

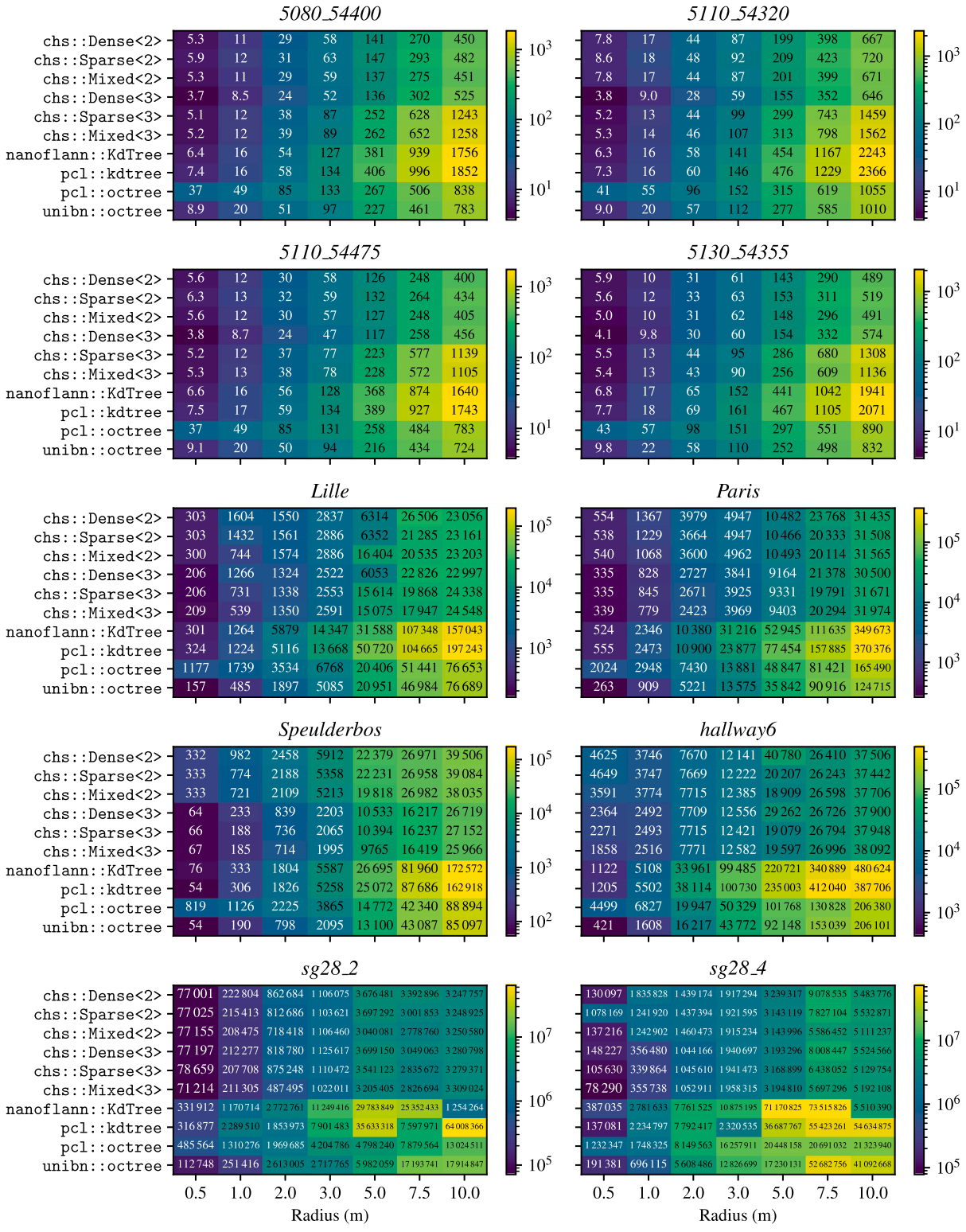

Fig. 12. Search time (in µs) of the different data structures in the spherical-kernel search. Lower is better.

The three-dimensional versions of the `cheesemap` have a slightly higher memory footprint, around 30 % higher than the minimum. Note that memory consumption increases drastically when using `chs::Dense<3>` with a cell size smaller than 1.0 m. Nevertheless, as mentioned in Section 5.2, despite having a higher memory footprint for cell sizes above 1.0 m, `chs::Dense<3>` remains comparable to other state-of-the-art data structures.

Finally, the `pcl::octree` and `unibn::Octree` are generally close to each other but use approximately twice the memory of `nanoflann::KdTree`. The `pcl::kdtree` has the highest memory

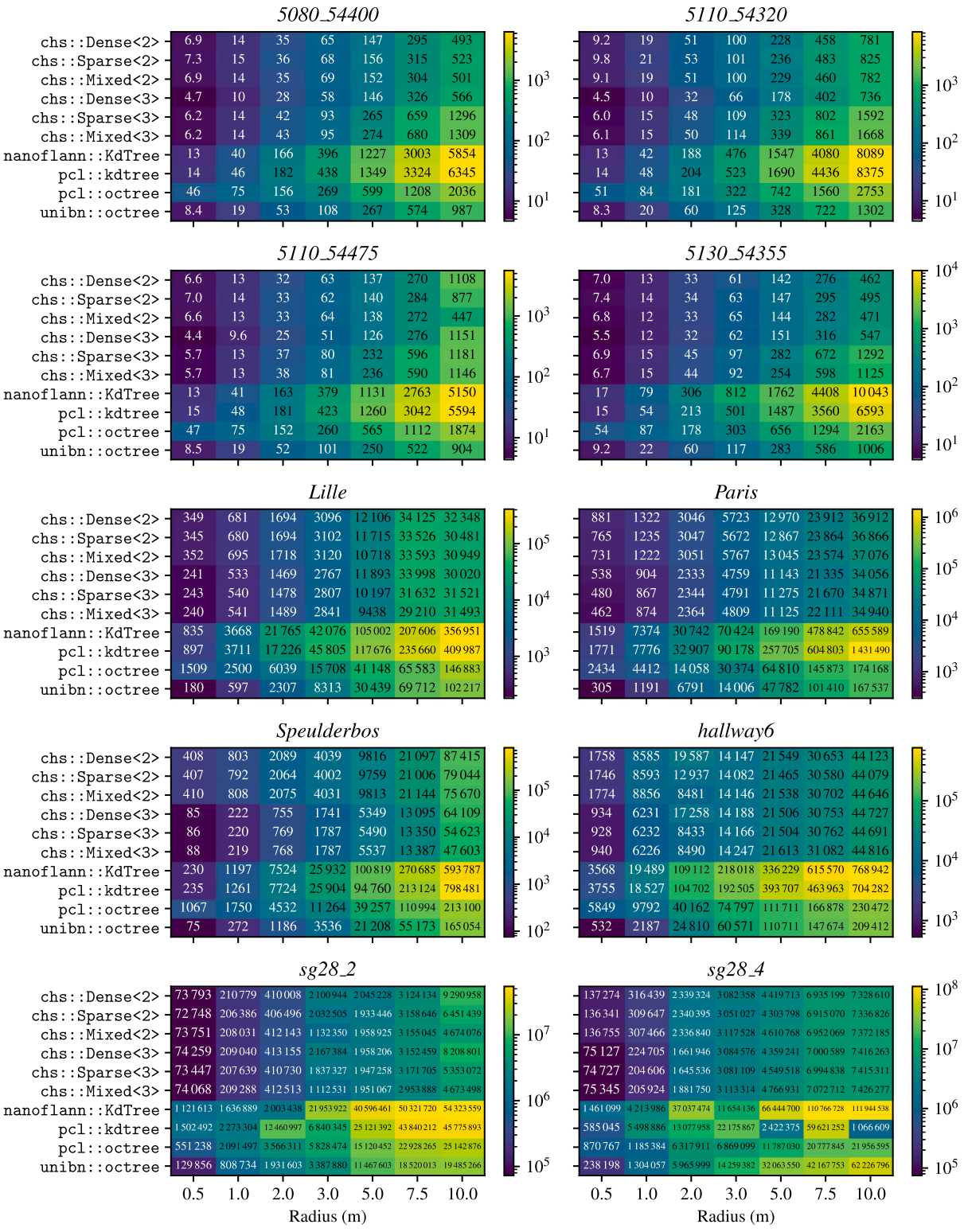

Fig. 13. Search time (in µs) of the different data structures in the cube-kernel search. Lower is better.

footprint, using up to 6 times more memory than `nanoflann::KdTree` or `cheesemap` in the *sg28_2* dataset.

Limitations of the `cheesemap`. While `cheesemap` demonstrates significant advantages in performance and memory efficiency for LiDAR point clouds, it is important to acknowledge certain limitations, particularly concerning specific configurations and use cases.

Firstly, the performance of `cheesemap`, especially for k-NN searches, is highly sensitive to the chosen cell size and the type of the dataset. As observed in our experiments, achieving optimal results often requires fine-tuning this parameter for specific datasets. For example, in TLS datasets, `cheesemap` can perform slower than `nanoflann::KdTree` if an appropriate cell size is not selected. Note

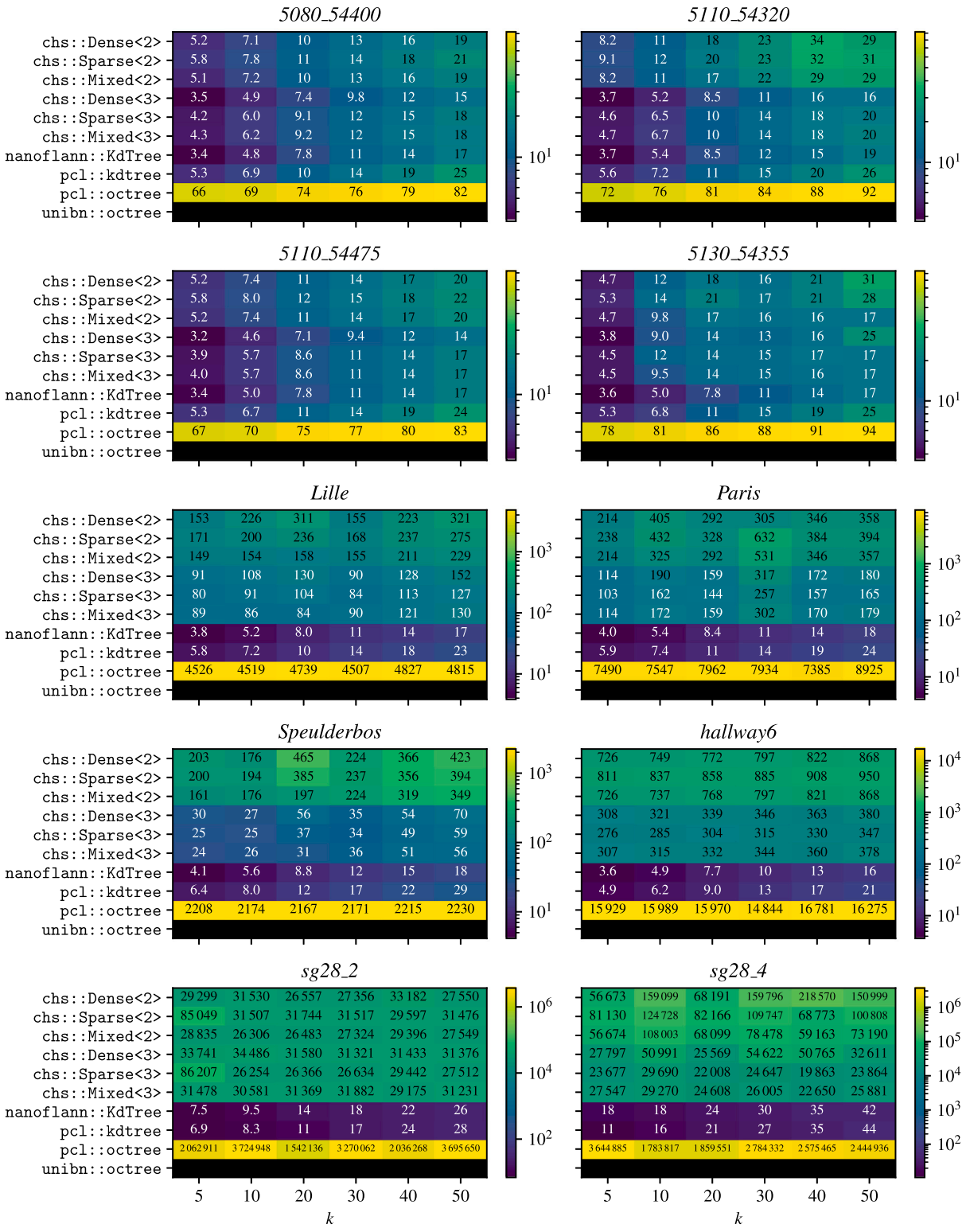

Fig. 14. Search time (in µs) of the different data structures in the k-NN search. `unibn::Octree` does not implement this operation. Lower is better.

that k-d trees exhibit amortized complexity of $\mathcal{O}(\log N)$, while the `cheesemap` relies on a linear radius growth. In cases where the cell size is coarse, the `cheesemap` may compute more distances than needed for points far outside the target radius. In these scenarios, deeper k-d trees with smaller leaves tend to perform better (as their spatial resolution might be higher). This sensitivity highlights the need for user expertise or an automated mechanism to determine the most suitable cell size for the varied characteristics of the point cloud.

Secondly, although the flavors `chs::Mixed` and `chs::Sparse` are designed to adapt to varying point densities, performance might be slightly affected in highly heterogeneous point clouds containing cells with exceptionally high numbers of points. In such cells, the internal linear search could lead to minor performance degradation. Nevertheless, the same limitation is also present in other data structures, such as `pcl::kdtree` and `pcl::octree`, which also rely on linear searches within the leafs.

Table 2
Memory footprint (MiB) for the different data structures in each dataset and theoretical minimum calculated as the number of points multiplied by the size of a pointer (assumed 8B). Lower is better.

Map	*5080_54400*	*5110_54320*	*5110_54475*	*5130_54355*	*Lille*	*Paris*	*Speulderbos*	*hallway6*	*sg28_2*	*sg28_4*
Points (M)	12.2	17.7	11.9	12.1	80.0	50.0	79.2	3.8	170.2	258.7
Theor. minimum	93.2	135.4	91.4	92.7	610.3	381.4	604.4	29.3	1298.2	1973.9
`chs::Dense<2>`	98.9	141.2	97.2	98.4	628.6	382.6	605.1	29.3	1302.5	1975.0
`chs::Dense<3>`	386.3	641.0	1073.5	356.9	1396.2	417.3	636.5	29.3	2051.5	2007.2
`chs::Sparse<2>`	105.4	147.7	103.7	104.9	612.7	382.3	605.3	29.3	1299.9	1974.3
`chs::Sparse<3>`	133.0	213.5	130.5	122.6	616.8	386.6	617.1	29.3	1301.5	1975.0
`chs::Mixed<2>`	103.8	148.0	101.9	103.2	643.2	401.1	634.8	30.7	1396.1	2120.0
`chs::Mixed<3>`	145.1	232.8	413.7	138.3	648.6	406.6	650.5	30.8	1398.7	2120.9
`nanoflann::KdTree`	98.9	141.7	94.9	92.8	672.8	381.5	676.3	29.3	1327.3	1970.8
`pcl::kdtree`	580.4	840.9	569.4	579.1	3779.3	2373.2	3769.3	180.4	8032.2	12 208.8
`pcl::octree`	186.4	270.8	182.8	185.3	1259.7	783.5	1171.8	58.6	2596.4	2817.3
`unibn::Octree`	174.4	254.6	176.4	182.9	1122.6	717.0	1056.9	50.0	2226.1	3381.6

Finally, like other axis-aligned data structures such as k-d trees and Octrees, `cheesemap`'s efficiency can be compromised in point clouds that are not aligned with its primary axes, specially in terms of memory consumption, as shown in Fig. 9. The *Lille* dataset serves as a clear example, where its specific orientation and shape results in a large bounding box with a high percentage of empty voxels, particularly for the `chs::Dense<3>`. In such scenarios, a preliminary rotation of the point cloud to align it with the principal direction of the data could minimize `cheesemap`'s memory footprint.

Applications of the `cheesemap`. The `cheesemap`, with its voxelized grid structure and different flavors, is particularly well-suited for handling dynamic point clouds and time series data. This suitability stems from its superior efficiency in data modification operations, specifically point insertion and deletion. The insertion of new points can be performed in amortized $\mathcal{O}(1)$ time, leveraging the fast lookup and insertion capabilities of the underlying map for voxel indexing. Similarly, point deletion in `cheesemap` involves an amortized $\mathcal{O}(1)$ voxel lookup followed by an $\mathcal{O}(N_v)$ operation to remove the point from the voxel's internal list, where N_v is the number of points in the voxel. Since cell sizes yielding the best performances are small, this implies that typically N_v is a small number. This efficiency contrasts with tree-based structures, where operations often incur higher computational costs due to potential rebalancing or partial reconstructions. Thus, `cheesemap` is a robust choice for applications requiring continuous updates of the dataset, such as change detection in dynamic series, for example for tracking the evolution of an open mining site, where several complete scans are done every day.

6. Conclusions

In this paper, we presented a thorough analysis of several data structures for indexing point clouds and compared them in terms of performance and memory footprint in real-world datasets. We also introduced the `cheesemap`, a novel data structure designed to be fast and memory-efficient for LiDAR point clouds. The `cheesemap` is based on a grid structure that divides the space into voxels, where each voxel lists the points within it. This structure allows for fast queries of points within a certain region (for example, spherical- and cubic-kernel search) and k-NN search.

The novelty of the `cheesemap` lies in its three indexing strategies: dense, sparse, and mixed. The dense representation creates a voxel for each cell, regardless of whether it contains points. The sparse structure only creates voxels for cells containing points, storing them in a hashmap. The mixed strategy is a hybrid approach where the grid is divided into dense or sparse slices depending on their non-empty voxel distribution.

The results show that the `cheesemap`'s performance depends heavily on cell size and dimensionality, being best with small cell sizes and three-dimensional grids at the cost of a slightly higher memory footprint.

Compared to other state-of-the-art data structures, the `cheesemap` shows superior performance for ALS and large point clouds for fixed-radius searches, being up to 40% faster than `nanoflann::KdTree` or `unibn::Octree`. For k-NN, the performance of the `cheesemap` is similar to `nanoflann::KdTree`, the fastest library that implements this type of query. Regarding datasets acquired using other kinds of platforms besides ALS, the performance for fixed-radius searches is on par with that of the fastest libraries, while the go-to option for k-NN searches, in this case, is `nanoflann::KdTree`. Finally, the memory footprint of the `cheesemap` is close to the theoretical minimum overhead, which is shown to be more efficient than `pcl::kdtree`, `pcl::octree`, and `unibn::Octree`.

Regarding future work, several ideas can be explored. For instance, exploring alternatives to `std::unordered_map`, like *flat* hashmaps (e.g., `absl::flat_hash_map`, or C++23's `std::flat_map`), which can improve performance due to better cache usage and lower pointer indirection. However, this comes at the expense of compatibility with the C++ standard library or theoretical guarantees. Another idea is automating cell size selection instead of relying on user input. This would allow the `cheesemap` to be more user-friendly and efficient, given its sensitivity to this parameter, especially for k-NN search. Further gains could be achieved from voxel and/or points reordering (e.g., Morton order) to improve memory locality and cache usage, speeding up intra-voxel operations.

In addition to internal improvements, we also plan to evaluate the `cheesemap` in real-world applications such as powerline detection [41], object classification [8], or change detection [42]. These applications could benefit from the `cheesemap`'s fast queries and efficient memory usage.

CRediT authorship contribution statement

Ruben Laso: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Miguel Yermo:** Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT to improve the language and readability of the text. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work received financial support from Consellería de Cultura, Educación e Ordenación Universitaria (accreditation ED431C-2022/16, ED431G-2019/04) and the European Regional Development Fund (ERDF), which acknowledges CiTIUS-Research Center in Intelligent Technologies as a Research Center of the Galician University System. The Ministry of Economy and Competitiveness, Government of Spain (Grant Number PID2019-104834GB-I00, and PID2022-141623NB-I00) also provided support.

Thanks to Prof. Siegfried Benkner, Prof. Fran F. Rivera, and Prof. José C. Cabaleiro for their valuable feedback and suggestions on the preliminary versions of the manuscript.

The authors also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of Finisterrae III to conduct the initial experiments.

Data availability

Data will be made available on request.

References

[1] Q. Xiang, Y. He, D. Wen, Adaptive deep learning-based neighborhood search method for point cloud, Sci. Rep. (2022) http://dx.doi.org/10.1038/s41598-022-06200-z.

[2] V. Turau, Fixed-radius near neighbors search, Inform. Process. Lett. (1991) http://dx.doi.org/10.1016/0020-0190(91)90180-p.

[3] E. Fix, J.L. Hodges, Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties, Int. Stat. Rev. / Rev. Int. de Stat. 57 (3) (1989) 238–247, http://dx.doi.org/10.2307/1403797.

[4] T. Cover, P. Hart, Nearest neighbor pattern classification, IEEE Trans. Inform. Theory 13 (1) (1967) 21–27, http://dx.doi.org/10.1109/TIT.1967.1053964.

[5] S. Filin, N. Pfeifer, Neighborhood systems for airborne laser data, Photogramm. Eng. Remote Sens. 71 (6) (2005) 743–755, http://dx.doi.org/10.14358/PERS.71.6.743.

[6] L. Wang, Y. Huang, Y. Hou, S. Zhang, J. Shan, Graph attention convolution for point cloud semantic segmentation, in: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR, 2019, pp. 10288–10297, http://dx.doi.org/10.1109/CVPR.2019.01054.

[7] H. Thomas, F. Goulette, J.-E. Deschaud, B. Marcotegui, Y. LeGall, Semantic classification of 3D point clouds with multiscale spherical neighborhoods, in: 2018 International Conference on 3D Vision (3DV), 2018, pp. 390–398, http://dx.doi.org/10.1109/3DV.2018.00052.

[8] M. Weinmann, B. Jutzi, S. Hinz, C. Mallet, Semantic point cloud interpretation based on optimal neighborhoods, relevant features and efficient classifiers, ISPRS J. Photogramm. Remote Sens. 105 (2015) 286–304, http://dx.doi.org/10.1016/j.isprsjprs.2015.01.016.

[9] T. Hackel, J.D. Wegner, K. Schindler, Fast semantic segmentation of 3D point clouds with strongly varying density, ISPRS Ann. Photogramm. Remote. Sens. Spat. Inf. Sci. III-3 (2016) 177–184, http://dx.doi.org/10.5194/isprs-annals-III-3-177-2016.

[10] M. Pauly, R. Keiser, M. Gross, Multi-scale feature extraction on point-sampled surfaces, Comput. Graph. Forum 22 (3) (2003) 281–289, http://dx.doi.org/10.1111/1467-8659.00675.

[11] N. Brodu, D. Lague, 3D terrestrial LiDAR data classification of complex natural scenes using a multi-scale dimensionality criterion: Applications in geomorphology, ISPRS J. Photogramm. Remote Sens. 68 (2012) 121–134, http://dx.doi.org/10.1016/j.isprsjprs.2012.01.006.

[12] J. Niemeyer, F. Rottensteiner, U. Soergel, Contextual classification of lidar data and building object detection in urban areas, ISPRS J. Photogramm. Remote Sens. 87 (2014) 152–165, http://dx.doi.org/10.1016/j.isprsjprs.2013.11.001.

[13] R.L. Cook, Stochastic sampling in computer graphics, ACM Trans. Graph. 5 (1) (1986) 51–72, http://dx.doi.org/10.1145/7529.8927.

[14] E. Sevgen, S. Abdikan, Classification of large-scale mobile laser scanning data in urban area with LightGBM, Remote. Sens. 15 (15) (2023) http://dx.doi.org/10.3390/rs15153787.

[15] Y. Tian, W. Song, L. Chen, Y. Sung, J. Kwak, S. Sun, A fast spatial clustering method for sparse LiDAR point clouds using GPU programming, Sensors 20 (8) (2020) http://dx.doi.org/10.3390/s20082309.

[16] X. Zeng, W. He, GPGPU-based parallel processing of massive LiDAR point cloud, in: F. Zhang, F. Zhang (Eds.), MIPPR 2009: Medical Imaging, Parallel Processing of Images, and Optimization Techniques, vol. 7497, SPIE, 2009, 749716, http://dx.doi.org/10.1117/12.833740.

[17] Q. Li, P. Yuan, Y. Lin, Y. Tong, X. Liu, Pointwise classification of mobile laser scanning point clouds of urban scenes using raw data, J. Appl. Remote. Sens. 15 (2) (2021) 024523, http://dx.doi.org/10.1117/1.JRS.15.024523.

[18] J.L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517, http://dx.doi.org/10.1145/361002.361007.

[19] J. Behley, V. Steinhage, A.B. Cremers, Efficient radius neighbor search in three-dimensional point clouds, in: 2015 IEEE International Conference on Robotics and Automation, ICRA, 2015, pp. 3625–3630, http://dx.doi.org/10.1109/ICRA.2015.7139702.

[20] S.M. Omohundro, Five Balltree Construction Algorithms, International Computer Science Institute Berkeley, 1989, URL https://steveomohundro.com/wp-content/uploads/2009/03/omohundro89_five_balltree_construction_algorithms.pdf.

[21] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD '84, Association for Computing Machinery, New York, NY, USA, 1984, pp. 47–57, http://dx.doi.org/10.1145/602259.602266.

[22] C. Luo, X. Li, N. Cheng, H. Li, S. Lei, P. Li, MVP-Net: Multiple view pointwise semantic segmentation of large-scale point clouds, 2022, arXiv:2201.12769.

[23] E. Che, M.J. Olsen, An efficient framework for mobile lidar trajectory reconstruction and mo-norvana segmentation, Remote. Sens. 11 (7) (2019) http://dx.doi.org/10.3390/rs11070836.

[24] W. Wang, Y. Zhang, G. Ge, Q. Jiang, Y. Wang, L. Hu, A hybrid spatial indexing structure of massive point cloud based on octree and 3D R*-Tree, Appl. Sci. 11 (20) (2021) http://dx.doi.org/10.3390/app11209581.

[25] Y. Feng, M. Cen, T. Zhang, An algorithm of fast index constructing and neighbor searching for 3D LiDAR data, in: H. Tan (Ed.), PIAGENG 2013: Intelligent Information, Control, and Communication Technology for Agricultural Engineering, vol. 8762, SPIE, 2013, p. 87620Y, http://dx.doi.org/10.1117/12.2019658.

[26] J. Li, B.M. Chen, G.H. Lee, SO-Net: Self-organizing network for point cloud analysis, in: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 9397–9406, http://dx.doi.org/10.1109/CVPR.2018.00979.

[27] X. Zhan, Y. Cai, P. He, A three-dimensional point cloud registration based on entropy and particle swarm optimization, Adv. Mech. Eng. 10 (12) (2018) 1687814018814330, http://dx.doi.org/10.1177/1687814018814330.

[28] D. Liu, D. Li, M. Wang, Z. Wang, 3D change detection using adaptive thresholds based on local point cloud density, ISPRS Int. J. Geo- Inf. 10 (3) (2021) http://dx.doi.org/10.3390/ijgi10030127.

[29] J. Elseberg, S. Magnenat, R. Siegwart, A. Nüchter, Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration, J. Softw. Eng. Robot. 3 (1) (2012) 2–12, URL https://robotik.informatik.uni-wuerzburg.de/telematics/download/joser2012.pdf.

[30] M. Lawson, W. Gropp, J. Lofstead, Exploring spatial indexing for accelerated feature retrieval in HPC, in: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2022, pp. 605–614, http://dx.doi.org/10.1109/CCGrid54584.2022.00070.

[31] N. Varney, V.K. Asari, Q. Graehling, DALES: A large-scale aerial LiDAR data set for semantic segmentation, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, 2020, pp. 186–187, http://dx.doi.org/10.1109/CVPRW50498.2020.00101.

[32] X. Roynard, J.-E. Deschaud, F. Goulette, Paris-Lille-3D: A large and high-quality ground-truth urban point cloud dataset for automatic segmentation and classification, Int. J. Robot. Res. 37 (6) (2018) 545–557, http://dx.doi.org/10.1177/0278364918767506.

[33] B. Brede, K. Calders, A. Lau, P. Raumonen, H.M. Bartholomeus, M. Herold, L. Kooistra, Non-destructive tree volume estimation through quantitative structure modelling: Comparing UAV laser scanning with terrestrial LiDAR, Remote Sens. Environ. 233 (2019) 111355, http://dx.doi.org/10.1016/j.rse.2019.111355.

[34] I. Armeni, O. Sener, A.R. Zamir, H. Jiang, I. Brilakis, M. Fischer, S. Savarese, 3D semantic parsing of large-scale indoor spaces, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2016, pp. 1534–1543, http://dx.doi.org/10.1109/CVPR.2016.170.

[35] T. Hackel, N. Savinov, L. Ladicky, J.D. Wegner, K. Schindler, M. Pollefeys, Semantic3D.net: A new large-scale point cloud classification benchmark, ISPRS Ann. Photogramm. Remote. Sens. Spat. Inf. Sci. IV-1/W1 (2017) 91–98, http://dx.doi.org/10.5194/isprs-annals-IV-1-W1-91-2017.

[36] J.L. Blanco, P.K. Rai, nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with KD-trees, 2014, https://github.com/jlblancoc/nanoflann.

[37] M. Muja, D.G. Lowe, Fast approximate nearest neighbors with automatic algorithm configuration, in: Proceedings of the Fourth International Conference on Computer Vision Theory and Applications (VISIGRAPP 2009) - Volume 1: VISAPP, SciTePress, 2009, pp. 331–340, http://dx.doi.org/10.5220/0001787803310340.

[38] R.B. Rusu, S. Cousins, 3D is here: Point cloud library (PCL), in: 2011 IEEE International Conference on Robotics and Automation, 2011, pp. 1–4, http://dx.doi.org/10.1109/ICRA.2011.5980567.

[39] S. Valat, A.S. Charif-Rubial, W. Jalby, MALT: a Malloc tracker, in: Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems, in: SEPS 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–10, http://dx.doi.org/10.1145/3141865.3141867.

[40] P.J. Fleming, J.J. Wallace, How not to lie with statistics: the correct way to summarize benchmark results, Commun. ACM 29 (3) (1986) 218–221, http://dx.doi.org/10.1145/5666.5673.

[41] M. Yermo, R. Laso, O.G. Lorenzo, T.F. Pena, J.C. Cabaleiro, F.F. Rivera, D.L. Vilariño, Powerline detection and characterization in general-purpose airborne LiDAR surveys, IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens. 17 (2024) 10137–10157, http://dx.doi.org/10.1109/JSTARS.2024.3396522.

[42] D. Liu, D. Li, M. Wang, Z. Wang, 3D change detection using adaptive thresholds based on local point cloud density, ISPRS Int. J. Geo- Inf. 10 (3) (2021) http://dx.doi.org/10.3390/ijgi10030127.

Ruben Laso currently working as a postdoctoral researcher at the Research Group for Scientific Computing at the University of Vienna. I earned my PhD in 2023 from the Universidade de Santiago de Compostela on High-Performance and Parallel Computing. Before that, I completed my Master's in Industrial Mathematics in 2019 and my Bachelor's in Computer Science in 2017, both at Universidade de Santiago de Compostela. My research interests include parallel computing, with a particular emphasis on manycore and NUMA systems, as well as implementing performance portability in scientific codes.

Miguel Yermo currently working as a postdoctoral researcher at the University of Santiago de Compostela. I earned my Ph.D. in High-Performance Computing from the Universidade de Santiago de Compostela in 2024. Before that, I completed my Master's in Applied Mathematics and my Bachelor's in Physics. My research interests include parallel computing, remote sensing, and deep learning for point clouds.