Assignment2 - Nearest Neighbor Problem

Ji

Dec 2022

Contents

1 Algorithms for finding nearest neighbors **2**

2 Algorithms' correctness **2**

3 Algorithms' efficiency **3**

1 Algorithms for finding nearest neighbors

In the benchmark, there are six algorithms that have been implemented and tested:

1. **Numpy brute-force**. A brute force algorithm finds k nearest neighbors of a query through exhaustion. It first calculates the distances between the query and all points and then sorts them in ascending order by the distances. The first k points of the order are k nearest neighbors of the query.

2. **Kdtree-knn_search**. Based on the point cloud data, it first builds a k-d tree and then find k nearest neighbors of a query by the k-d tree.

3. **Kdtree-radius_search**. Based on the point cloud data, it first builds a k-d tree and then find all neighbors of a query within its query ball with a specified radius.

4. **Octree-knn_search**. Based on the point cloud data, it first builds an octree and then find k nearest neighbors of a query by the octree.

5. **Octree-radius_search**. Based on the point cloud data, it first builds an octree and then find all neighbors of a query within its query ball with a specified radius.

6. **Octree-radius_search_fast**. It is almost same as the octree-radius_search, but have an additional mechanism for checking whether the search can stop early.

2 Algorithms' correctness

The first experiment is to verify the correctness of the algorithms described in Section 1. The experiment is designed as the following:

1. The point cloud data is provided by the file 000000.bin.

2. The query is the first point of the point cloud data whose value is $[52.898, 0.023, 1.998]$.

3. The algorithm named numpy brute-force is the baseline which provides the answer. The indexes of the eight nearest neighbors are $[0, 3943, 5884, 3944, 5885, 1, 5883, 1966]$, their distances to the query are $[0.00, 0.38, 0.63, 0.79, 0.82, 0.87, 0.88, 0.91]$.

4. For the remaining algorithms, they share two configurations: leaf_size = 32 and min_extent = 0.0001. The hyperparameter k is set to 8 for knn search, while the hyperparameter radius is set to 1 for raidus search. If other algorithms can obtain the same answer, we can prove that they have been implemented correctly.

algorithm	nearest neighbors sorted by their distance to the query
kdtree-knn_search	$[0, 3943, 5884, 3944, 5885, 1, 5883, 1966]$
kdtree-radius_search	$[0, 3943, 5884, 3944, 5885, 1, 5883, 1966, 1967, 2]$
octree-knn_search	$[0, 3943, 5884, 3944, 5885, 1, 5883, 1966]$
octree-radius_search	$[0, 3943, 5884, 3944, 5885, 1, 5883, 1966, 1967, 2]$
octree-radius_search_fast	$[0, 3943, 5884, 3944, 5885, 1, 5883, 1966, 1967, 2]$

Table 1: Results of nearest neighbors found by five algorithms.

As shown in Table 1, the eight nearest neighbors obtained by kdtree-knn_search and octree-knn_search are the same as that of the baseline. Thus, we have proved the correctness of kdtree-knn_search and octree-knn_search.

As for kdtree-radius_search, octree-radius_search, and octree-radius_search_fast, they found the ten nearest neighbors of the query. This is because they aimed at finding all neighbors within a query ball. Nevertheless, the first eight neighbors are the same as those of the baseline, so their correctness has also been verified.

3 Algorithms' efficiency

In the experiment of comparing the algorithms' efficiency, the benchmark program generates 100 queries randomly. Each algorithm finds the nearest neighbors of those 100 queries in turn, and then we can get the average time of processing one query by each algorithm.

algorithm	average time (ms)	improvement
numpy brute-force	10.393	
kdtree-knn_search	9.929	1.047
kdtree-radius_search	0.168	61.863
octree-knn_search	1.054	9.861
octree-radius_search	0.209	49.727
octree-radius_search_fast	0.239	43.485

Table 2: Algorithms' efficiency compared to the baseline.

As shown in Table 2, there is improvement for the five algorithms. However, their increases are quite different. For example, the efficiency of kdtree-knn_search is only 4.7% better than that of the baseline, while kdtree-radius_search is more than 60 times faster than the baseline.

Notably, octree-radius_search_fast is slower than octree-radius_search, which is caused by the value of radius. If the radius of the query ball is very small, the search will probably not stop earlier but waste additional time on checking the stopping criteria. In a new experiment, I increased the value of radius from 1 to 5. Since the query ball becomes bigger, it is easier to completely contain an octant and thus stop earlier. As a result, octree-radius_search_fast cost 17.848 ms, octree-radius_search costed 22.021 ms. In this case, octree-radius_search_fast is more efficient than octree-radius_search.