sad-L4-hw

Yixiao Feng

17/06/2023

1 实现 NEARBY14

NEARBY14 有多种实现方式，其中的一种实现方式如图 1：

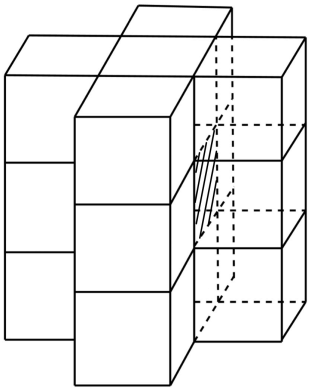

图 1: NEARBY14 实现方式之一

代码实现如下：

```
// enum声明 (gridnn.hpp 28)
enum class NearbyType {
       CENTER, // 只考虑中心
       // for 2D
       NEARBY4, // 上下左右
       NEARBY8, // 上下左右+四角

       // for 3D
       NEARBY6, // 上下左右前后
       // TODO: 第四章习题1, 实现NEARBY14
       NEARBY14, // doc中附有图示
};
// ----------------------------------

// 构造函数改动 (gridnn.hpp 45)
/**
 * 构造函数
 * @param resolution 分辨率
 * @param nearby_type 近邻判定方法
 */
explicit GridNN(float resolution = 0.1, NearbyType nearby_type = NearbyType::NEARBY4)
    : resolution_(resolution), nearby_type_(nearby_type) {
    inv_resolution_ = 1.0 / resolution_;

    // check dim and nearby
    if (dim == 2 && nearby_type_ == NearbyType::NEARBY6) {
        LOG(INFO) << "2D grid does not support nearby6, using nearby4 instead.";
        nearby_type_ = NearbyType::NEARBY4;
    } else if (dim == 3 && (nearby_type != NearbyType::NEARBY14 && nearby_type_ != NearbyType::NEARBY6 &&
         nearby_type_ != NearbyType::CENTER)) {
        LOG(INFO) << "3D grid does not support nearby4/8, using nearby14 instead.";
        nearby_type_ = NearbyType::NEARBY14;
    }

    GenerateNearbyGrids();
}
```

```cpp
// ----------------------------------

// NEARBY14 nearby_grids_实现 (gridnn.hpp 129)
template <>
void GridNN<3>::GenerateNearbyGrids() {
    if (nearby_type_ == NearbyType::CENTER) {
        nearby_grids_.emplace_back(KeyType::Zero());
    } else if (nearby_type_ == NearbyType::NEARBY6) {
        nearby_grids_ = {KeyType(0, 0, 0), KeyType(-1, 0, 0), KeyType(1, 0, 0), KeyType(0, 1, 0),
                         KeyType(0, -1, 0), KeyType(0, 0, -1), KeyType(0, 0, 1)};
    } else if (nearby_type_ == NearbyType::NEARBY14) {
        nearby_grids_ = {KeyType(0, 0, 0), KeyType(-1, 0, 0), KeyType(1, 0, 0), KeyType(0, -1, 0),
                         KeyType(0, 1, 0), KeyType(0, 0, -1), KeyType(-1, 0, -1), KeyType(1, 0, -1),
                         KeyType(0, -1, -1), KeyType(0, 1, -1),KeyType(0, 0, 1), KeyType(-1, 0, 1),
                         KeyType(1, 0, 1), KeyType(0, -1, 1), KeyType(0, 1, 1)};
    }
}
// ----------------------------------

// GTEST (test_nn.cc 178)
LOG(INFO) << "===================";
sad::evaluate_and_call(
    [&first, &second, &grid14, &matches](){ grid14.GetClosestPointForCloud(first, second, matches); },
    "Grid 3D NEARBY14 单线程", 10);
EvaluateMatches(truth_matches, matches);

LOG(INFO) << "===================";
sad::evaluate_and_call(
    [&first, &second, &grid14, &matches]() { grid14.GetClosestPointForCloudMT(first, second, matches); },
    "Grid 3D NEARBY14 多线程", 10);
EvaluateMatches(truth_matches, matches);
```

单元测试结果如下：

```
I0617 21:04:05.430716 29049 test_nn.cc:109] points: 18869, 18779
I0617 21:04:06.892983 29049 gridnn.hpp:106] grids: 9707
I0617 21:04:06.893966 29049 gridnn.hpp:106] grids: 9707
I0617 21:04:06.895084 29049 gridnn.hpp:106] grids: 9707
I0617 21:04:06.896422 29049 gridnn.hpp:106] grids: 14079
I0617 21:04:06.897773 29049 gridnn.hpp:106] grids: 14079
I0617 21:04:06.897775 29049 test_nn.cc:129] ===================
I0617 21:04:06.910897 29049 sys_utils.h:32] 方法 Grid0 单线程 平均调用时间/次数: 1.31206/10 毫秒.
I0617 21:04:06.910902 29049 test_nn.cc:66] truth: 18779, esti: 8518
I0617 21:04:06.940757 29049 test_nn.cc:92] precision: 0.486382, recall: 0.220619, fp: 4375, fn: 14636
I0617 21:04:06.940760 29049 test_nn.cc:136] ===================
I0617 21:04:06.942065 29049 sys_utils.h:32] 方法 Grid0 多线程 平均调用时间/次数: 0.130273/10 毫秒.
I0617 21:04:06.942068 29049 test_nn.cc:66] truth: 18779, esti: 18779
I0617 21:04:06.992329 29049 test_nn.cc:92] precision: 0.486382, recall: 0.220619, fp: 4375, fn: 14636
I0617 21:04:06.992332 29049 test_nn.cc:142] ===================
I0617 21:04:07.037885 29049 sys_utils.h:32] 方法 Grid4 单线程 平均调用时间/次数: 4.55516/10 毫秒.
I0617 21:04:07.037890 29049 test_nn.cc:66] truth: 18779, esti: 13272
I0617 21:04:07.079385 29049 test_nn.cc:92] precision: 0.646775, recall: 0.457106, fp: 4688, fn: 10195
I0617 21:04:07.079388 29049 test_nn.cc:148] ===================
I0617 21:04:07.082223 29049 sys_utils.h:32] 方法 Grid4 多线程 平均调用时间/次数: 0.283228/10 毫秒.
I0617 21:04:07.082228 29049 test_nn.cc:66] truth: 18779, esti: 18779
I0617 21:04:07.133111 29049 test_nn.cc:92] precision: 0.646775, recall: 0.457106, fp: 4688, fn: 10195
I0617 21:04:07.133116 29049 test_nn.cc:154] ===================
```

```
I0617 21:04:07.204798 29049 sys_utils.h:32] 方法 Grid8 单线程 平均调用时间/次数: 7.16812/10 毫秒.
I0617 21:04:07.204802 29049 test_nn.cc:66] truth: 18779, esti: 14613
I0617 21:04:07.246197 29049 test_nn.cc:92] precision: 0.728735, recall: 0.56707, fp: 3964, fn: 8130
I0617 21:04:07.246202 29049 test_nn.cc:160] ===================
I0617 21:04:07.250435 29049 sys_utils.h:32] 方法 Grid8 多线程 平均调用时间/次数: 0.423104/10 毫秒.
I0617 21:04:07.250440 29049 test_nn.cc:66] truth: 18779, esti: 18779
I0617 21:04:07.297513 29049 test_nn.cc:92] precision: 0.728735, recall: 0.56707, fp: 3964, fn: 8130
I0617 21:04:07.297518 29049 test_nn.cc:166] ===================
I0617 21:04:07.327119 29049 sys_utils.h:32] 方法 Grid 3D NEARBY6 单线程 平均调用时间/次数: 2.96004/10 毫秒.
I0617 21:04:07.327123 29049 test_nn.cc:66] truth: 18779, esti: 8572
I0617 21:04:07.351140 29049 test_nn.cc:92] precision: 0.911339, recall: 0.415997, fp: 760, fn: 10967
I0617 21:04:07.351145 29049 test_nn.cc:172] ===================
I0617 21:04:07.353319 29049 sys_utils.h:32] 方法 Grid 3D NEARBY6 多线程 平均调用时间/次数: 0.217251/10 毫秒.
I0617 21:04:07.353323 29049 test_nn.cc:66] truth: 18779, esti: 18779
I0617 21:04:07.395004 29049 test_nn.cc:92] precision: 0.911339, recall: 0.415997, fp: 760, fn: 10967
I0617 21:04:07.395006 29049 test_nn.cc:178] ===================
I0617 21:04:07.445885 29049 sys_utils.h:32] 方法 Grid 3D NEARBY14 单线程 平均调用时间/次数: 5.08768/10 毫秒.
I0617 21:04:07.445888 29049 test_nn.cc:66] truth: 18779, esti: 8933
I0617 21:04:07.470857 29049 test_nn.cc:92] precision: 0.908653, recall: 0.432238, fp: 816, fn: 10662
I0617 21:04:07.470860 29049 test_nn.cc:184] ===================
I0617 21:04:07.473814 29049 sys_utils.h:32] 方法 Grid 3D NEARBY14 多线程 平均调用时间/次数: 0.295033/10 毫秒.
I0617 21:04:07.473821 29049 test_nn.cc:66] truth: 18779, esti: 18779
I0617 21:04:07.513700 29049 test_nn.cc:92] precision: 0.908653, recall: 0.432238, fp: 816, fn: 10662
```

从结果中不难发现，NEARBY14 相比与 NEARBY6，召回率有所上升，但准确率稍有下降，计算效率也稍有下降。这是合理的，因为搜索范围的增加会增加参与计算 nn 的点，降低效率，提高召回率。

2 推导题 - 特征值解法

$$\boldsymbol{d}^* = \arg\max_{\boldsymbol{d}} ||\boldsymbol{A}\boldsymbol{d}||_2^2 = \arg\max_{\boldsymbol{d}} \boldsymbol{d}^\top \boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{d}. \tag{2.1}$$

同样也对 $\boldsymbol{A}^\top\boldsymbol{A}$ 进行对角化：

$$\boldsymbol{A}^\top\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{-1}. \tag{2.2}$$

同样 $\boldsymbol{V}$ 为正交矩阵，它的列向量 $\boldsymbol{v}_1, \cdots \boldsymbol{v}_n$ 构成一组单位正交基。d 可以被这组正交基表示出来：

$$\boldsymbol{d} = \alpha_1\boldsymbol{v}_1 + \cdots + \alpha_n\boldsymbol{v}_n. \tag{2.3}$$

不难看出：

$$\boldsymbol{V}^{-1}\boldsymbol{d} = \boldsymbol{V}^\top\boldsymbol{d} = \left[\alpha_1, \cdots, \alpha_n\right]^\top. \tag{2.4}$$

式(2.1)可以改写成：

$$\boldsymbol{d}^* = \arg\max_{\boldsymbol{d}} ||\boldsymbol{A}\boldsymbol{d}||_2^2 = \arg\max_{\boldsymbol{d}} (\left[\alpha_1, \cdots, \alpha_n\right] \boldsymbol{\Lambda} \left[\alpha_1, \cdots, \alpha_n\right]^\top) = \arg\max_{\boldsymbol{d}} \sum_{k=1}^{n} \lambda_k \alpha_k^2. \tag{2.5}$$

由于 $||\boldsymbol{d}|| = 1$，$\alpha_1^2 + \cdots + \alpha_n^2 = 1$，为了让目标函数最大，且特征值是降序排列的，所以取 $\alpha_1 = 1, \alpha_2 = 0, \cdots, \alpha_n = 0$，即 $\boldsymbol{d}^* = \boldsymbol{v}_1$，也就是最大特征值对应的特征向量。

3 比较本节最近邻算法与 nanoflann [1]

实现了一个新的头文件 nanoflann_utils.h，用来做 nanoflann 两个点云间的 KNN，据我了解 nanoflann 只实现了单个 query 点和点云间的 KNN。代码实现如下：

```
//
// Created by yixfeng on 19/06/23.
//

#ifndef SLAM_IN_AUTO_DRIVING_NANOFLANN_UTILS_H
#define SLAM_IN_AUTO_DRIVING_NANOFLANN_UTILS_H

#include <nanoflann.hpp>
#include "common/point_cloud_utils.h"
#include "common/point_types.h"
#include "common/sys_utils.h"
#include "common/math_utils.h"

// Copied from nanoflann utils.h
template <typename T>
struct PointCloud
{
   struct Point
   {
      T x, y, z;
   };

   using coord_t = T; //!< The type of each coordinate

   std::vector<Point> pts;

   // Must return the number of data points
   inline size_t kdtree_get_point_count() const { return pts.size(); }

   // Returns the dim'th component of the idx'th point in the class:
   // Since this is inlined and the "dim" argument is typically an immediate
   // value, the
   // "if/else's" are actually solved at compile time.
   inline T kdtree_get_pt(const size_t idx, const size_t dim) const
   {
      if (dim == 0)
         return pts[idx].x;
      else if (dim == 1)
         return pts[idx].y;
      else
         return pts[idx].z;
   }

   // Optional bounding-box computation: return false to default to a standard
   // bbox computation loop.
   // Return true if the BBOX was already computed by the class and returned
   // in "bb" so it can be avoided to redo it again. Look at bb.size() to
   // find out the expected dimensionality (e.g. 2 or 3 for point clouds)
   template <class BBOX>
   bool kdtree_get_bbox(BBOX& /* bb */) const
```

[1] https://github.com/jlblancoc/nanoflann

```
    {
        return false;
    }
};

using nano_flann_tree = nanoflann::KDTreeSingleIndexAdaptor<nanoflann::L2_Simple_Adaptor<float, PointCloud<
     float>>,
                    PointCloud<float>, 3, size_t>;

bool GetClosestPointNanoFlann(nano_flann_tree &tree, const sad::CloudPtr &query, std::vector<std::pair<size_t,
      size_t>> &matches, size_t k) {
    matches.resize(query->size() * k);

    std::vector<size_t> index(query->size());
    for (size_t i = 0; i < query->points.size(); i++) {
        index[i] = i;
    }

    std::for_each(std::execution::seq, index.begin(), index.end(), [&tree, &query, &matches, &k](size_t idx) {
        size_t num_results = k;
        std::vector<size_t> ret_index(num_results);
        std::vector<float> out_dist_sqr(num_results);
        auto pt = query->points[idx];
        float query_pt[3] = {pt.x, pt.y, pt.z};
        num_results = tree.knnSearch(&query_pt[0], num_results, &ret_index[0], &out_dist_sqr[0]);
        for (size_t i = 0; i < k; i++) {
            matches[idx * k + i].second = idx;
            if (i < num_results) {
                matches[idx * k + i].first = ret_index[i];
            } else {
                matches[idx * k + i].first = sad::math::kINVALID_ID;
            }
        }
    });

    return true;
}

bool GetClosestPointNanoFlannMT(nano_flann_tree &tree, const sad::CloudPtr &query, std::vector<std::pair<
     size_t, size_t>> &matches, size_t k) {
    matches.resize(query->size() * k);

    std::vector<size_t> index(query->size());
    for (size_t i = 0; i < query->points.size(); i++) {
        index[i] = i;
    }

    std::for_each(std::execution::par_unseq, index.begin(), index.end(), [&tree, &query, &matches, &k](size_t
         idx) {
        size_t num_results = k;
        std::vector<size_t> ret_index(num_results);
        std::vector<float> out_dist_sqr(num_results);
        auto pt = query->points[idx];
        float query_pt[3] = {pt.x, pt.y, pt.z};
        num_results = tree.knnSearch(&query_pt[0], num_results, &ret_index[0], &out_dist_sqr[0]);
        for (size_t i = 0; i < k; i++) {
            matches[idx * k + i].second = idx;
```

```
            if (i < num_results) {
                matches[idx * k + i].first = ret_index[i];
            } else {
                matches[idx * k + i].first = sad::math::kINVALID_ID;
            }
        }
    });

    return true;
}

#endif // SLAM_IN_AUTO_DRIVING_NANOFLANN_UTILS_H
```

其实就是仿照本节 KD Tree 的实现，实现了下单线程和多线程的版本。除此之外，GTest 也添加了一些，如下：

```
// GTEST (test_nn.cc 283)
LOG(INFO) << "building nanoflann tree";
PointCloud<float> first_cloud;
first_cloud.pts.resize(first->size());
for (int i = 0; i < first->size(); i++) {
    first_cloud.pts[i].x = first->points[i].x;
    first_cloud.pts[i].y = first->points[i].y;
    first_cloud.pts[i].z = first->points[i].z;
}

sad::evaluate_and_call([&first_cloud](){ nano_flann_tree nanotree(3, first_cloud, 1 /* max leaf size */); },
                       "NanoFlann Tree build", 1); // 不想单独再写个时间评估函数了，这个nanotree的build是通过构造函
                             数构建的，以lambda传入，外部使用需要再写一遍
nano_flann_tree nanotree(3, first_cloud, 1 /* max leaf size */);

LOG(INFO) << "searching nanoflann";
matches.clear();
sad::evaluate_and_call([&nanotree, &second, &matches]() {
    GetClosestPointNanoFlann(nanotree, second, matches, 5);
}, "NanoFlann Tree 5NN 单线程", 1);
EvaluateMatches(true_matches, matches);

matches.clear();
sad::evaluate_and_call([&nanotree, &second, &matches]() {
    GetClosestPointNanoFlannMT(nanotree, second, matches, 5);
}, "NanoFlann Tree 5NN 多线程", 1);
EvaluateMatches(true_matches, matches);
```

最终，单元测试的结果如下：

```
I0619 04:32:00.350667 38554 sys_utils.h:32] 方法 Kd Tree build 平均调用时间/次数: 3.45961/1 毫秒.
I0619 04:32:00.350672 38554 test_nn.cc:244] Kd tree leaves: 18869, points: 18869
I0619 04:32:00.816596 38554 sys_utils.h:32] 方法 Kd Tree 5NN 多线程 平均调用时间/次数: 2.68522/1 毫秒.
I0619 04:32:00.816609 38554 test_nn.cc:67] truth: 93895, esti: 93895
I0619 04:32:01.879731 38554 test_nn.cc:93] precision: 1, recall: 1, fp: 0, fn: 0
I0619 04:32:01.879741 38554 test_nn.cc:256] building kdtree pcl
I0619 04:32:01.881284 38554 sys_utils.h:32] 方法 Kd Tree build 平均调用时间/次数: 1.49931/1 毫秒.
I0619 04:32:01.881287 38554 test_nn.cc:261] searching pcl
I0619 04:32:01.893106 38554 sys_utils.h:32] 方法 Kd Tree 5NN in PCL 平均调用时间/次数: 11.8074/1 毫秒.
I0619 04:32:01.893211 38554 test_nn.cc:67] truth: 93895, esti: 93895
I0619 04:32:02.975641 38554 test_nn.cc:93] precision: 1, recall: 1, fp: 0, fn: 0
I0619 04:32:02.975651 38554 test_nn.cc:283] building nanoflann tree
```

```
I0619 04:32:02.978137 38554 sys_utils.h:32] 方法 NanoFlann Tree build 平均调用时间/次数: 2.36883/1 毫秒.
I0619 04:32:02.980242 38554 test_nn.cc:296] searching nanoflann
I0619 04:32:02.988370 38554 sys_utils.h:32] 方法 NanoFlann Tree 5NN 单线程 平均调用时间/次数: 8.12593/1 毫秒.
I0619 04:32:02.988373 38554 test_nn.cc:67] truth: 93895, esti: 93895
I0619 04:32:04.044631 38554 test_nn.cc:93] precision: 1, recall: 1, fp: 0, fn: 0
I0619 04:32:04.045312 38554 sys_utils.h:32] 方法 NanoFlann Tree 5NN 多线程 平均调用时间/次数: 0.672774/1 毫秒.
I0619 04:32:04.045318 38554 test_nn.cc:67] truth: 93895, esti: 93895
I0619 04:32:05.102789 38554 test_nn.cc:93] precision: 1, recall: 1, fp: 0, fn: 0
I0619 04:32:05.102797 38554 test_nn.cc:309] done.
```

由此可见，nanoflann 的实现单线程的版本比 PCL 的实现搜索时间效率更高，nanoflann 构建 KD Tree 的时间比课程实现的版本要短，多线程 nanoflann 的版本搜索时间效率最高。在 leafsize 固定为 1，且不调整比例因子，即不使用近似最近邻的情况下，三种方法的准召率都为 1。