Lab2： HNSW (Hierarchical-Navigable-Small-World)

难点以及一些实现细节：

1. 在此次编写过程中，函数传入的各参数中不包括数组的长度，这就要求我们自己在 hnsw.cpp 这个文件中去分析输入的具体长度，这是一个有点卡住我的点。但是在分析后就容易发现长度都是固定的 128，问题解决。

2. 在编写的过程中，发现如果给每一个节点都分配最大层数，最大相邻节点个数条边，会导致 malloc 失败，因此使用一个 unordered map 与 vector 的结合来存每一个节点的不同层上的边的指向节点，从而减少储存空间。此外，在上面观察 hnsw.cpp 的过程中，发现可以直接使用传入的指针作为存一个结点具体向量，进一步节省了空间。

   最终的每个节点的储存结构如下：

   ```
   struct node{
       int label;
       const int *item;
       int level;
       unordered_map<int, vector<node*> > edges;
   };
   ```

3. 一开始，由于未考虑并发安全方面的情况，将每次计算距离的结果直接存入一个结点中，导致在尝试实现并发查询时的 average recall 的值与串行实现时有差异。后改为只在局部变量中存储，解决问题。

4. 在测试并发性能时，发现多核的表现不如使用双线程的情况，经过检查，发现给 Linux 虚拟机分配的 CPU 核的个数只有两个，是一个意料之外的错误。在分配了 8 个核后性能的提升就比较明显了。

在实现过程中，最重要的一个函数是 search_layer, 这个函数在插入与查询函数中都要用到，用于在一层中找到要求个数的最近的结点的集合。以下是代码及部分注释。

```
std::vector<HNSW::node*> HNSW::search_layer(node *q, node *ep, int ef, int lc)
{
    // 用于记录节点是否已经访问过
    unordered_set<node *> visited;
    visited.insert(ep);
    // C 为小根堆（通过存相反数实现）， W 为大根堆。
    priority_queue< pair<long, node*> > C;
    priority_queue< pair<long, node*> > W;
    long dis = l2distance(q->item, ep->item, len);
    // 将 entrance point 加入优先队列
    C.push(make_pair(-dis, ep));
    W.push(make_pair(dis, ep));

    pair<long, node*> now;
    while (C.size() > 0) {
        now = C.top(); C.pop();
        // 如果当前最近的节点已经比存入W中最远的结点更远，则以后的结点都会比最远的结点远。
        if (!W.empty() && W.top().first < -now.first)
            break;
```

```
        // 枚举每一个相邻的结点
        for (int i = 0; i < now.second->edges[lc].size(); i++) {
            node* cur = now.second->edges[lc][i];
            auto it = visited.find(cur);
            if (it != visited.end()) continue;
            visited.insert(cur);
            dis = l2distance(cur->item, q->item, len);
            if (dis < W.top().first || W.size() < ef_search) {
                C.push(make_pair(-dis, cur));
                W.push(make_pair(dis, cur));
                // 只取最近的 ef_search 个结点
                if (W.size() > ef_search)
                    W.pop();
            }
        }
    }
    vector<node*> w;
    while (!W.empty())
    {
        w.push_back(W.top().second);
        W.pop();
    }
    return w;
}
```

各参数的影响

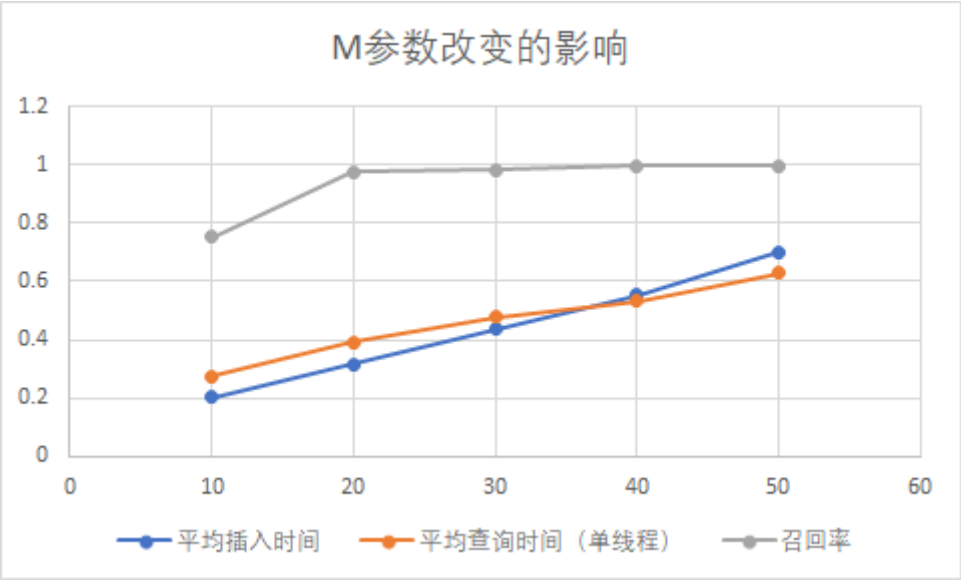
各参数的定义

1. M 初次建立的边数
2. M_max 最大建立边数
3. ef_construction, ef_search 建立图与查找时要找的最大结点数
4. threadNum 多线程查询时的线程数

参数 M 的影响

在 M = M_max 的情况下，分别取 M = 10, 20, 30, 40, 50 进行测试（查询时间按串行计算），测试的结果如下。

M的取值	10	20	30	40	50
平均插入时间(ms)	0.202	0.317	0.437	0.554	0.702
平均查询时间(ms)	0.274	0.390	0.476	0.531	0.627
召回率	0.751	0.976	0.983	0.996	0.996



结果分析：

测试结果与预期的保持一致，随着 M 的增加，平均所需要的插入和查询时间都在增加，且基本呈线性增长。同时，随着 M 的增大，召回率则会增大，但是在 M 增大到 20 以后，就稳定在 95% 以上了。

此外，从图中，我们也可以发现平均插入时间与平均查询时间是差不多的，这有点出乎我的意料，因为在要添加的边超过 M 的时候，我们需要从已有的边中找一条最远的（而且要比新加入结点远）边删除，有一个额外的复杂度，但是在这里似乎影响不大。

参数 M_max 的影响

除了 M 之外，M_max 参数对召回率和时间的影响也是非常大的。统计了召回率如下：

M\M_max	20	30	40	50
M * 0.25	0.991	0.996	0.998	1.000
M * 0.33	0.993	0.998	1.000	1.000
M * 0.50	0.998	1.000	1.000	1.000
M * 0.75	0.992	0.998	1.000	1.000
M * 1.00	0.976	0.983	0.996	0.996

串行情况下查询时间（按串行计算）如下表：

M\M_max	20	30	40	50
M * 0.25	0.451	0.486	0.544	0.589
M * 0.33	0.509	0.532	0.531	0.600
M * 0.50	0.449	0.509	0.566	0.613
M * 0.75	0.402	0.504	0.533	0.720
M * 1.00	0.356	0.443	0.518	0.619

可以发现，当取初始的 M 为 M_max 约一半时，可以让召回率达到最大，尽管同时耗时也有所上升，但是幅度不大。经测试，取 M = 15，M = 30 可以取到很好的性能和正确性。（实际上 M = 12, M_max = 25 时更好）

性能测试

测试类型

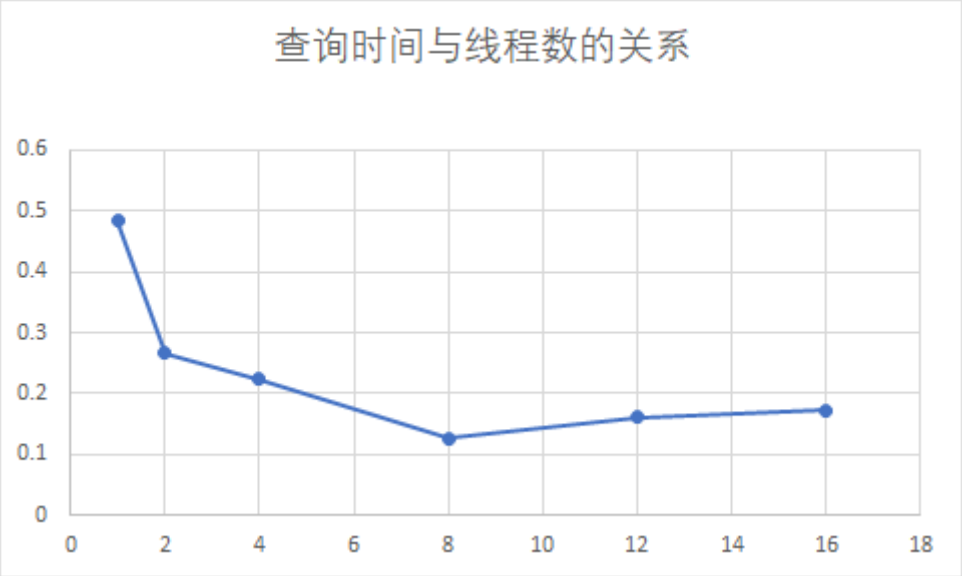
在 Linux 虚拟环境下进行测试，CPU 核数为 8， 内存大小为 8GB。

在此次测试中，取 M = 15, M_max = 30, ef_construction = ef_search = 100.

分别测试了线程数为 1, 2, 4, 8, 12, 16 的各运行时间情况

测试结果

线程数	1	2	4	8	12	16
平均查询时间	0.484	0.267	0.224	0.127	0.161	0.172
相对于单线程的优化比例	1	0.55	0.46	0.26	0.33	0.36



结果分析

测试结果与预期的基本保持一致。多线程的情况下，平均单次查询时间均比单线程的情况下更短，且在使用 8 线程时达到最优。在从单线程变为双线程时，查询时间降低了近一半，但是在后面分出更多线程时下降的程度逐渐减小，且在线程数比 8 大以后时间反而上升了。

出现性能提升并非线性的可能原因有：

1. 在8线程时，性能提升最为显著，这是因为8线程恰好匹配8核机器的核心数，此时并行度达到最佳。
2. 当线程数超过8时，线程之间的上下文切换、缓存一致性维护等开销增加，导致性能下降。在我的实现中，由于没用共享的资源会被多个线程同时修改，因此不存在一些冲突的问题，因此没有加锁也不需要考虑锁带来的性能问题。