Lab2+3: HNSW+并行编程

Lab2的主题涉及图相关的数据结构和算法，Lab3则关注并行编程。我们将这两个Lab合并为一个Lab，并在其中涵盖这两个主题。在这个Lab中，你将学习并实现一个名为Hierarchical-Navigable-Small-World（HNSW）的图相关数据结构及其算法，并在此基础上对其实现进行并行化优化。

注意：助教实现的版本大概需要约300行左右的代码（不包括代码框架和并行优化部分）。请同学们合理安排时间和工作进度，以完成Lab的要求。

1 背景介绍：向量数据库与ANN索引

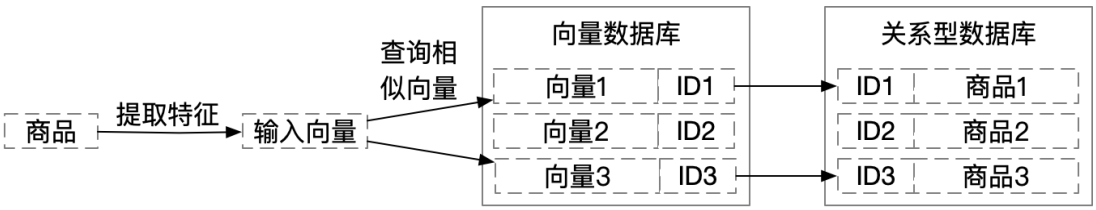
近年来，在人工智能的推动下，推荐系统不断发展。以商城的推荐系统为例，系统中的所有商品都通过机器学习的方法提取其特征，这些特征被表示为特征向量。当需要为用户推荐与某个商品相似的商品时，推荐系统会寻找与该商品特征向量接近的其他商品。在这个实验中，我们使用欧氏距离来衡量向量之间的接近程度。欧氏距离被定义为两个向量之间各个维度差值的平方和的开方，如下所示：

$$d = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}$$

其中，d表示欧氏距离，x和y分别表示两个向量的特征值，n表示向量的维度。欧氏距离越小，表示两个向量越接近。

然而，在所有商品的特征向量中查找输入向量的**最为接近的向量**计算量大、效率低（特别是面临维度灾难的问题，感兴趣的同学可以自行了解）。为了解决这个问题，向量数据库被引入推荐系统中，用来专门存储商品的向量，并建立ANN（Approximate Nearest Neighbor）索引。ANN索引牺牲查询的精确性，选择查找与输入向量**相对接近的向量**，从而提高搜索性能。如下图所示，在查找一个商品的相似商品时，首先会在向量数据库中利用索引查找具有相似特征向量的其他商品的ID，接着再从关系型数据库中通过这些ID读取相应商品。

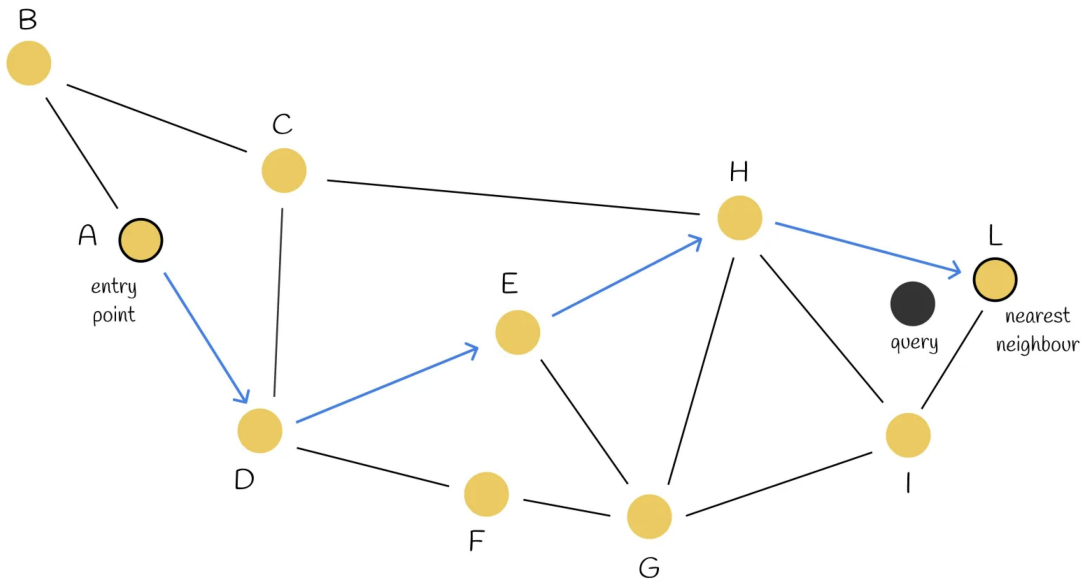
在该Lab中你需要实现的HNSW即为一种基于图的ANN索引。



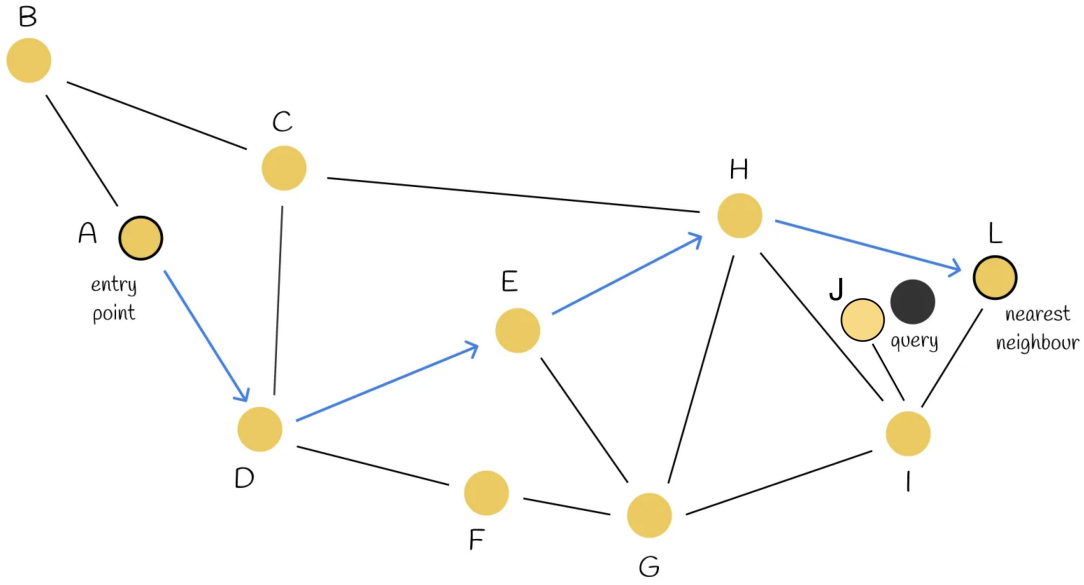
2 基本结构

本节介绍HNSW的基本思想与结构。HNSW是基于NSW（Navigable Small World）进行优化的。为便于理解，我们首先介绍NSW的结构，并以此为基础介绍HNSW的结构。

NSW找到目标节点的临近节点的方法是：将数据库中的向量与接近的向量相连，形成一个连通图。查询过程从这个连通图上的某个起始节点开始，不断跳到更靠近目标节点的邻居节点，直到无法再靠近目标节点为止，得到的终点节点即为查询结果。这个过程被称为导航（Navigation）。下图为NSW的基本结构示意图。每个黄色节点代表一个特征向量，并与较为接近的向量相连。黑色节点表示被查询的目标向量，若要搜索一个目标向量的相似向量，从A向量出发，导航过程分别经过A、D、E、H，最后停在L。由于L的邻边无法再使目标节点更靠近，因此L即为最终的查询结果。

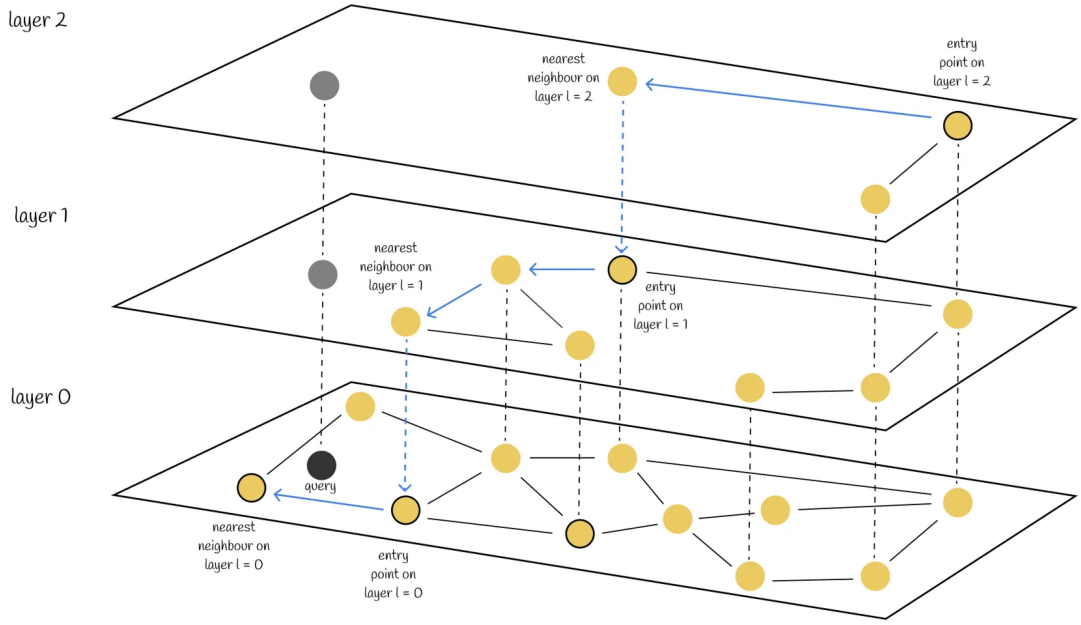


注意：通过这个算法并不一定能找到最接近的向量，例如下图中节点J比L更接近目标节点，然而L与其之间并没有边，则L不是最接近的向量，但仍然是较为接近的向量，这在推荐系统等应用中是可以被接受的。



上述查询方式相比精确地找到最接近的相似向量能够大大提升效率。然而，当数据库中的向量数量特别庞大时，若起始节点距离目标节点很远，导航过程的时延也会显著增加。在这种情况下，HNSW借鉴了跳表的思想，采用分级的方式存储特征向量，在高层级存储较少的向量，这使得导航过程能够在高层级快速地靠近目标节点。

下图展示了HNSW基本结构的示意图。导航过程从入口节点（entry point）开始，在较高层级尽可能向目标节点靠近。如果无法继续靠近，则下降到下一层级的相同节点，直到最终下降到最底层（layer 0）并完成查询。



3 基本操作与算法

本节详细介绍HNSW的两个基本操作及其具体算法。HNSW的基本操作包括：

- insert：用于在建立索引过程中插入新的向量。
- query：用于在索引中查询与目标向量相近的N个向量。

3.1 Insert

本小节从伪代码出发介绍了HNSW插入的流程。首先，介绍HNSW算法中的几个配置参数，这些参数会影响HNSW搜索的精确度和性能。在提供的代码框架中，可以找到这些变量的定义与赋值：

1. M: 在插入过程中，被插入节点需要与图中其他节点建立的连接数
2. M_max: 每个节点与图中其他节点建立的最大连接数。随着新节点的插入，图中旧节点连接数可能会超过该值，需要相应地删去部分边（伪代码中会具体介绍如何选择被删除边）
3. efConstruction: 由于查询时需要搜索一批相似向量，因此在搜索过程中需要维护一个候选节点集合，efConstruction为候选节点集合的数量
4. m_L: 一个正则化参数，用于决定被插入节点最高会被插入到HNSW哪一层级

insert过程主要分为三步：

- 第一步：使用m_L计算被插入节点q最高会被插入到第L层（第三步会将q插入到第0~L层）
- 第二步：自顶层向被插入节点q的层数L逐层搜索，一直到L+1，在每一层导航到与节点q相对接近的节点（没有与节点q更接近的邻居节点），将其加入最近邻元素集合W，并从W中挑选最接近q的节点作为下一层搜索的入口节点，这一过程与第二节中只需查找一个相似向量的图例相同。
- 第三步：自L层向第0层逐层搜索，维护当前层搜索到的与q最近邻的efConstruction个点，并在该层与节点q相连。

以下为insert过程的伪代码及其解释：

```
/**
 * 输入：
 * hnsw：q插入的目标图
```

```
 * q: 插入的新元素
 * 输出: 新的hnsw图
 */
INSERT(hnsw, q, M, M_max, efConstruction, m_L)
W ← ∅  // W: 现在发现的最近邻元素集合
ep ← get enter point of hnsw
maxL ← level of ep
/**
 * 第一步使用类似跳表中的对数方法决定q从哪一层开始插入
 * 该函数被实现为util函数，可以直接使用
 */
L ← ⌊-ln(unif(0..1))•mL⌋
/**
 * 第二步：自顶层向q的层数L逐层搜索，一直到L+1
 * 每层寻找当前层q最近邻的1个点，加入W，
 * 并从W中挑选最接近q的节点作为下一层搜索的入口节点
 * SEARCH_LAYER(q, ep, ef, lc)函数的功能是：
 * 从ep节点开始查找第lc层中与q相近最多的ef个节点，其伪代码在之后展示
 */
for lc ← maxL … L+1
    W ← SEARCH_LAYER(q, ep, ef=1, lc)
    ep ← get the nearest element from W to q

// 第三步：自L层向第0层逐层搜索
for lc ← min(maxL, L) … 0
    // 每层寻找当前层q最近邻的efConstruction个点赋值到集合W
    W ← SEARCH_LAYER(q, ep, efConstruction, lc)
    // 在W中选择q最近邻的M个点作为neighbors双向连接起来
    // neighbors[lc][q]表示第lc层中节点q的邻居
    neighbors[lc][q] ← SELECT_NEIGHBORS(q, W, M, lc)
    // 检查每个neighbors的连接数，
    // 如果大于Mmax，则保留其中长度最短的Mmax个连接，并删去其他边
    for each e ∈ neighbors
        neighbors[lc][e].add(q)
        if |neighbors[lc][e]| > M_max
            neighbors[lc][e] ← SELECT_NEIGHBORS(e, eConn, M_max, lc)
    // 从W中挑选最接近q的节点作为下一层搜索的入口节点
    ep ← W

// 如果q更新了图的最高层数，则将之后插入或查询的入口节点设置为q
if L > maxL
    set enter point for hnsw to q
```

在上述伪代码中直接使用了功能函数SEARCH_LAYER：在lc层查找距离节点q近邻的ef个节点。简单来说，其实现方法是不断探索已访问节点的邻居，并将其中距离目标节点更近的邻居加入进一步探索目标，同时维护目前发现的最近邻节点集合。该函数流程的伪代码如下所示：

```
/**
 * 输入
 * q: 插入的新元素
 * ep: 入口节点 enter point
```

```
 * ef: 需要返回的近邻数量
 * lc: 层数
 * 输出: q的ef个最近邻
 */
SEARCH_LAYER(q, ep, ef, lc)
v ← ep  // v: 访问过的节点集合
C ← ep  // C: 候选节点集合，其邻居将会被探索
W ← ep  // W: 现在发现的最近邻元素集合
// 遍历每一个候选元素，包括遍历过程中不断加入的元素
while |C| > 0
    // 取出C中q的最近邻c
    c ← pop nearest element from C to q
    // 取出W中q的最远点f
    f ← get furthest element from W to q
    // 如果c已经比最近邻所有节点离目标节点更远，则无需探索
    if distance(c, q) > distance(f, q)
        break
    /**
     * 当c比f距离q更近时，则将c的每一个邻居e都进行遍历
     * 如果e比w中距离q最远的f要更接近q，那就把e加入到W和候选元素C中
     * 由此会不断地遍历图，直至达到局部最佳状态，c的所有邻居没有距离更近的了或者所有邻居
都已经被遍历了
     */
    for each e ∈ neighbourhood(c) at layer lc
        if e ∉ v
            v ← v ∪ e
            f ← get furthest element from W to q
            if distance(e, q) < distance(f, q) or |W| < ef
                C ← C ∪ e
                W ← W ∪ e
                // 保证返回的数目不大于ef
                if |W| > ef
                    remove furthest element from W to q
return W
```

3.2 Query

本小节介绍查询目标节点的相邻节点的算法。查询过程与插入过程相近，在此不再以伪代码形式展开描述，query过程同样分为两步：

- 自顶层向第1层逐层搜索,每层寻找当前层与目标节点q最近邻的1个点赋值到集合W（可以复用SEARCH_LAYER函数）然后从集合W中选择最接近q的点作为下一层的搜索入口点。
- 在第0层使用SEARCH_LAYER函数查找与目标节点q临近的efConstruction个节点，并根据需要返回的节点数量从中挑选离目标节点的最近的节点集（在该Lab中efConstruction为100，需要返回的节点数为10个）。

第二步得到的节点即为返回结果。

4 并行优化

在实际应用场景中，向量数据库往往面临着大量的查询压力。简单的串行处理将带来用户不可忍受的时延，特别是在短时间内有大量用户同时发起请求的情况下；另一方面，现代服务器均是多核架构，且一般通过集群来更好地提升应用的性能，串行处理请求的方式无法利用拥有的计算资源，将造成极大的浪费。因此，在实际应用场景下一般采用并行处理查询请求的方式。

在本次Lab的并行优化部分，你需要通过并行编程来处理查询请求。简单起见，你**不需要**考虑查询过程中可能出现的数据更新，例如插入和删除等操作。你可以为每一个查询请求分配一个新的线程，由这些线程彼此独立地处理查询。**本次实验只要求完成上述优化即可**，感兴趣的同学也可以进一步思考：本次实验的查询请求只有100条，那么如果查询请求的数量达到数万甚至数十数百万，该方法是否依然可行？有没有更好更高效的处理办法？以下是一些可能的思路：

- 使用线程池
- 预先确定用于服务查询请求的线程数量，依据线程数量为每个线程分配一定数量的查询请求

如果你使用了其他的并行技巧正确地处理查询请求，同样可以获得该部分的分数。**注意：并行优化是一个独立的Lab，请在报告中详细叙述你的并行优化细节，我们将以此判定你Lab3的分数。**

5 测试

5.1 正确性测试

测试程序使用召回率衡量HNSW的实现正确性及效果。召回率表示：使用ANN索引查询得到的efConstrucion个节点占实际上最邻近的efConstruction个节点的比例。

测试数据集使用siftsmall，其数据格式及含义可见参考资料，测试输入文件包括：

- base.bvecs: 需要被插入并构建ANN索引的向量。
- query.bvecs: 被查询的向量集合。
- gnd.ivecs: groundtruth文件，保存对于每个查询通过暴力搜索出的**最临近的**节点ID，按照与目标节点的欧氏距离按升序排序，测试程序将查询结果与该文件比对从而计算召回率。

在vecs_io.hpp文件中提供了读取上述输入文件的工具函数，供同学们直接使用或参考实现。

测试主函数见hnsw.cpp，主函数首先读取输入文件，构建HNSW索引，执行查询操作，最后与gnd.ivecs文件进行比较。在使用预设好的参数设置（M=30, M_max=30, efConstruction=100）下，如下所示为你的召回率测试结果与相应得分（注：在助教实现的版本中，M=30时，召回率达到98%+，M=10时，召回率达到约90%）：

- 召回率大于80%，得分60%
- 召回率大于90%，得分80%
- 召回率大于95%，得分100%

使用如下命令运行正确性测试：

```
# 编译链接正确性测试文件
make all
# 运行正确性测试
make grade
# 清理编译结果
make clean
```

5.2 参数M的影响

在该测试中你需要修改M以及M_max的值（M=M_max=10,20,30,40,50），记录不同M值下查询召回率以及单次查询时延的变化（测试程序已经实现了对召回率以及单词查询时延的记录与输出），并对测试结果进行观察与分析。

5.3 性能测试

你将在siftsmall数据集的基础上实现测试程序（不修改hnsw.cpp，在test.cpp中实现），你需要衡量HNSW的插入和查询性能，记录HNSW插入和查询的时延，并将串行查询的性能与并行优化后的性能进行比较。感兴趣的同学也可以选择***参考资料***中siftsmall数据集网站上的其他数据集进行测试并比较。一般而言，经过并行优化后的查询性能应优于串行查询，如果你的并行查询性能劣于串行查询，请寻找原因并给出你的分析。**你的测试程序应在运行结束后输出并行查询所需要的时间，同时参照hnsw.cpp中的方式输出召回率。**

6 提交要求

你的提交内容应该包括：

- 所有相关代码
- 一份README：
  - 说明如何编译、运行测试（特别是性能测试）、获取实验数据结果。
- 一份实验报告：
  - 简单列举实现过程中遇到的难点、或是印象较深的细节
  - 5.2与5.3的测试结果及其分析
  - 实验报告篇幅在4页以内

7 建议

- siftsmall测试数据集是包含10000个向量的数据集，不利于debug。在实现过程中，建议自己构建一个只包含少量向量的测试集，并在此基础上按步执行程序，观察程序行为是否正确。
- 正确性测试程序中的计时方式仍使用std::chrono的高精度计时库，未避免计时精度问题，请在linux环境中进行测试。

8 参考资料

- HNSW论文：Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs
- siftsmall数据集