华东师范大学软件工程学院实验报告

实验课程: 数据结构实践	年级: 2023 级	实验成绩:
实验名称: 向量搜索算法	姓名:	
实验编号:	学号:	实验日期: 2024 年 5 月 29 日
指导老师: 王丽萍	组号:	

目录

1 实验简介

本次作业要求我们实现向量的搜索算法, 即给定一个向量的集合 $X \in \mathbb{R}^{n\times d}$, 其中包含了 n 个 d 维的向量. 对于一个查询向量 $\mathbf{q}$, 向量最近邻搜索的目的是从 X 中找到一个向量 $\mathbf{x}^*$, 满足

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in X} \delta(\mathbf{x}, \mathbf{q})$$

即寻找 $\mathbf{q}$ 最近邻的向量. 其中 $\delta(\cdot,\cdot)$ 表示向量之间的距离函数, 它的数学定义如下 (欧几里得距离):

$$\delta(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d} (x_i - y_i)^2$$

其中, x_i 和 y_i 分别表示 $\mathbf{x}$ 和 $\mathbf{y}$ 的第 i 维分量.

如上定义可以非常简单地扩展到 K-最近邻搜索 (K-Nearest Neighbor Search), 即搜索 K 个最近邻的向量. 但是当 n 和 d 的数目显著增加的时候, 精确最近邻搜索往往不再适用 (搜索成本太高), 我们往往会在牺牲准确率的情况下, 提高搜索的速度, 这就是相似最近邻搜索 (Approximate Nearest Neighbor Search, ANN). 相似最近邻搜索的衡量指标 (召回率) 定义为

$$\text{Recall@k} = \frac{|R \cap \hat{R}|}{k}$$

其中 R 是 $\mathbf{q}$ 真正的 k 个最近邻向量的集合, $\hat{R}$ 是通过算法得到的近似最近邻向量的集合.

2 Baseline

首先我制作了 `baselineSearch` 函数, 其在测试集 8000 上的效率是约每秒 100 次搜索.

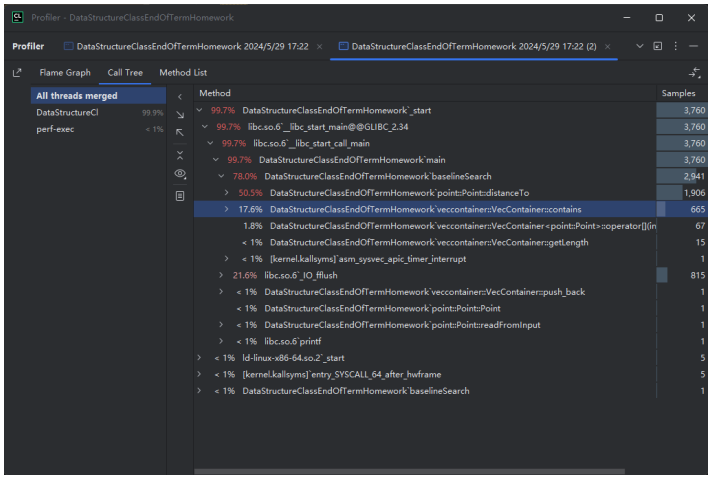

图 1: Baseline Profile

发现选中项 `contains` 有不必要的时间占用.

3 Baseline With HashSet

于是我将 `HashSet` 应用到 `baselineSearch` 索引的保存中, 并将 `hashSet` 设置为 1000.

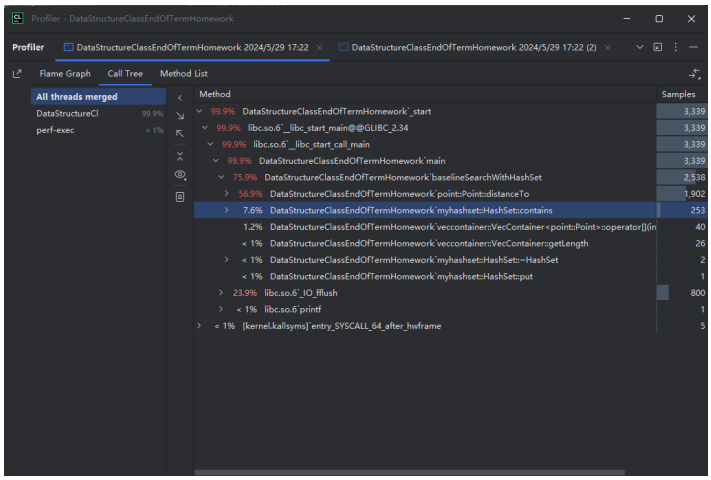

图 2: Baseline with HashSet Profile

得到了小许优化.

3.1 结果

3679149	2024-05-31 20:47:20	2024-05-31 20:47:30	C++17	Wrong answer: 7	0.729	8.602

图 3: Baseline Score

惨不忍睹.

4 HNSW

通过向量数据库视频, 我了解到了 HNSW 算法, 用于查找与目标向量最接近的向量.
其大致的思路是:

1. 创建多个层, 不同的层从稀疏到密集.
2. 算法分为两个步骤, 我称之为 Build 和 Search:
   (a) Build: 创建不同层次的图, 以不同的频率添加向量到不同的层之中, 添加向量时让其和原先在图中和其最靠近的 N_LINK 个向量建立连接.
   (b) Search:
      i. 先在最稀疏的层中选一个节点开始进行贪婪地查找最接近的向量, 直到图中节点没有邻居比其更加靠近目标向量, 得到局部最小值节点.

ii. 从上一层的局部最小值节点开始, 在更加密集的下一层中, 重复上述步骤, 直到最底层, 得到近似的和目标向量最接近的节点.

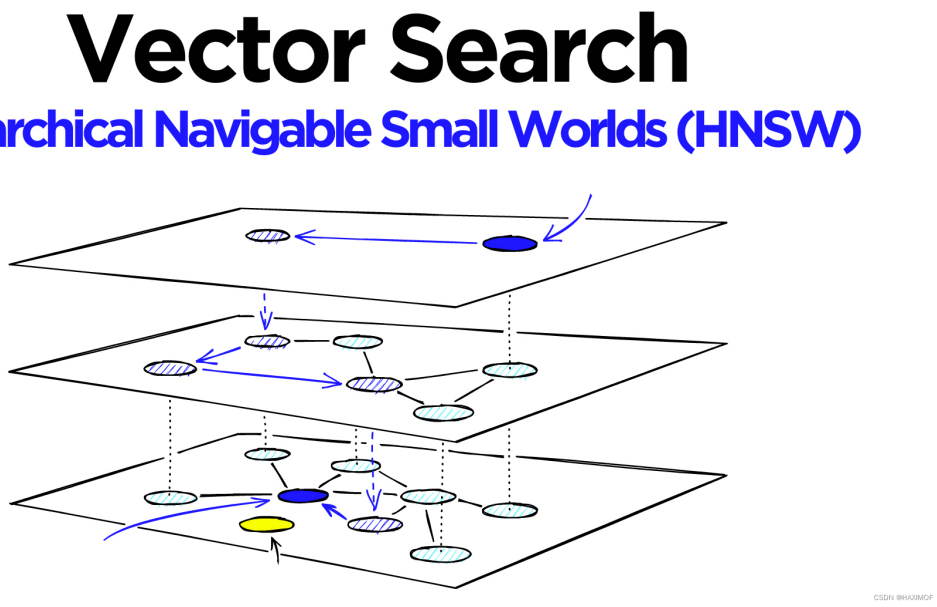

图 4: HNSW Introduction

在我的理解中, 此算法在 Build 中也需要使用 Search 中的部分步骤, 其中最困扰我的是如何进行 KNN 搜索.

4.1 HNSW-KNN

但是我第一眼看去 HNSW 只能用于查找最近的一个向量, 并没有直接看出如何使用 HNSW 应用于 KNN 问题上.

4.1.1 优先级队列

于是我问 GPT, 它说用优先级队列来实现最近 k 个节点的获取, 但是它给出的具体解决方案近乎胡扯, 于是我只能自己想具体实现.

在我最初的设想中: 优先级队列中队列首位元素是距离目标点最远的点, 探索时每次循环弹出队列首位元素, 然后如果弹出的节点有比此节点距离目标向量更近的邻居节点, 且此邻居节点不在优先级队列内, 则将邻居节点添加进优先级队列中. 总结就是: 弹出最远的, 放入更近的.

4.1.2 优先级队列特殊情况

如果探索完, 结果没有 k 个节点, 可能的情况为:

1. 层中没有足够的点.
2. 搜索的起始点和目标点足够近.

4.1.3 优先级队列特殊情况的解决思路

对于第一点, 我的解决思路是直接返回层中的所有节点而不进行搜索.

对于第二点, 我自己想的思路是, 如果优先级队列的元素数量没有达到 k 那就不检查邻居节点与目标向量的距离是否更近, 直接加入队列, 但重复性还是要检查的.

4.1.4 解决优先级队列比较里的重复计算和重复性检查

要把优先级队列应用下来, 还需要一个步骤: 缓冲每个节点和目标向量的距离. 原因是在优先级队列中, 需要进行多次比较, 若不缓冲距离则会造成大量的重复的运算.

对此我的设计是: 每个 Layer 维护一个独特的 `searchBatch:int`, 用于记录搜索批次, 每当搜索的目标节点或使用的优先级队列发生变化, 此值自增. Layer 中的每个 GraphNode 添加三个字段: `batch:int`, `inQueue:bool`, `distance:double`, 分别代表缓冲的距离所属的搜索批次, 是否在队列里和缓冲的距离.

这三个字段不仅可以减少重复的计算, 还能用于重复性检查, 如果 `batch != searchBatch`, 就说明此节点在此次搜索中没有被涉及到, 此时 `inQueue` 和 `distance` 的值需要重新设置和计算. 如果 `batch == searchBatch`, 说明此次搜索层涉及过此节点, 那么 `distance` 字段可以直接使用, `inQueue` 字段用来判断此节点是否在优先级队列里, 防止反复添加.

最后放进优先级队列里的是 GraphNode 的指针, 同时传入一个比较函数给优先级队列, 用于取得 `distance` 并比较. 此处的设计一开始困扰了我一会, 我的几个原有的想法是:

1. 优先级队列里存放 GraphNode 的包裹类, 重载比较运算符.
2. 优先级队列里存放 GraphNode 在 Layer 中的索引.

这几个想法由于以下缺陷而取消了:

1. 包裹类内存开销太大, 每次搜索都要进行多次内存的分配, 且查找值时不方便.
2. 如果只传入索引, 那么在比较时获取 `distance` 的值就需要一个 Layer 的引用, 于是在比较函数中就需要拥有 Layer 的指针, 而获取 Layer 的引用比较困难, 会使代码的耦合性变高.

4.1.5 自己想的优先级队列搜索步骤

注意: 以下步骤中,

1. 每次使用节点的缓冲距离时, 需要提前计算.
2. 每次从优先级队列中放入值前需要设置 `inQueue = true`.
3. 每次从优先级队列中弹出值时需要设置 `inQueue = false`.

步骤为:

1. 如果 Layer 的节点数量小于等于 k, 直接把 Layer 中所有的节点索引返回.

2. 从 Layer 中的前 k 个节点开始, 把这些节点放入优先级队列 (pq: `PriorityQueue<GraphNode *>`, 堆顶存放着距离最远的节点).

3. 循环, 直到 pq 为空 (pq 为空估计不会出现):

   (a) 弹出 pq 顶上的元素 `node`.

   (b) 遍历 `node` 的每个邻居, 当邻居的距离比 `node` 近时才添加此邻居到 pq 内.

   (c) 如果此 `node` 没有邻居被添加, 结束循环.

4. 丢弃 pq 中的多余元素, 直到 pq 剩下 k 个元素.

5. 将这 k 个元素倒序放在 Vec 中 (distance 字段最小的放在 Vec 0 号位置), 返回.

4.1.6 错误, 推倒重来

在编写完上述思路的代码, 准备写层间搜索的衔接时, 我发现以上方法的一个致命的错误:

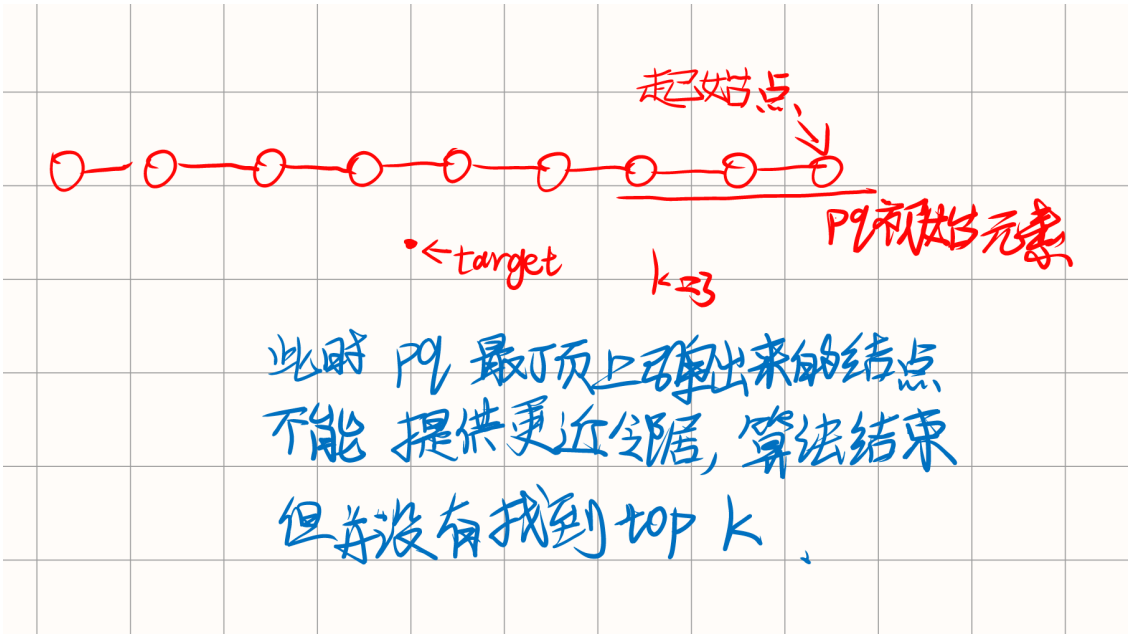

图 5: Miscase

4.2 靠近后拓展

上述方法失败之后, 我想出了另外一个方法: 先在层内找到距离目标向量最接近的节点, 然后以此节点为起始节点, 拓展 k 个节点, 以此实现 KNN 问题.

拓展的方式是给定一个节点, 从此节点开始以广度优先的方式增添节点, 直到增添到了 k 个节点, 返回这些节点所代表的值.

HNSW 算法实践后, 速度相比 Baseline 有大幅提升, 再把链表的取值用自制迭代器的形式进行优化, 减少冗余的指针游走, 最终在测试集 8000 上的总搜索速度可以到 1 秒左右 (N_LINK 为 3), 但准确率不尽人意.

速度 (avg 表示每秒进行的搜索次数):

```
D:\Program_Projects\Cpp_Projects\DataStructureClassFina
44999, time(s): 1, time(ms): 15, avg: 44333.985566 Hz

Process finished with exit code 0
```

图 6: HNSW Top-K Efficiency

准确率样本 (左边是当前方法搜索出来的向量和目标向量的距离, 右边是 Baseline 结果):

```
D:\Program_Projects\Cpp_Projects
Distance: 3.072600, 2.327806
Distance: 4.874210, 2.510265
Distance: 6.455727, 2.635693
Distance: 6.766348, 2.969491
Distance: 7.255120, 3.072600
Distance: 7.366715, 3.223833
Distance: 7.505963, 3.325562
Distance: 7.651447, 3.325731
Distance: 7.908677, 3.738243
Distance: 8.262122, 4.033727
Distance: 8.377673, 4.123006
Distance: 8.687564, 4.130256
Distance: 8.787715, 4.158977
Distance: 9.778990, 4.204982
Distance: 10.844708, 4.210038
Distance: 11.042339, 4.237144
Distance: 11.489833, 4.268434
Distance: 12.110540, 4.282012
Distance: 12.596525, 4.331919
Distance: 12.781760, 4.376074
Distance: 12.933412, 4.410323
Distance: 13.073381, 4.419710
Distance: 13.656811, 4.509404
Distance: 16.142036, 4.532644
Distance: 17.656423, 4.552542
Distance: 19.923192, 4.557267
Distance: 22.387020, 4.577296
Distance: 26.088641, 4.613334
Distance: 29.033342, 4.626596
Distance: 35.457167, 4.629465
```

图 7: HNSW Top-K Accuracy Sample

由图可知, 此拓展方式产生的 KNN 召回率结果太差, 可以说就是 0%.

4.3 查阅论文和开源项目

论文: Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

开源项目: hnswlib

浏览了论文中对 HNSW 算法的描述后, 我一层一层地分析项目中的代码, 最终找到了此项目中的 knn 搜索方法: searchBaseLayerST, 把此方法的几个分支简化 (假定 `bare_bone_search` 为 true) 后, 发现方法在搜索时使用了两个优先级队列, 一个优先级队列是正序的, 另一个是倒序的.

这个分支的搜索思路是:

1. 同时维护一个结果优先级队列 rq 和一个候选者优先级队列 cq, rq 就是返回值对应的队列.
2. 把起始节点放入 rq 和 cq.
3. 循环, 直到 cq 为空:
   (a) 从 cq 中取出距离目标向量最短的节点 nn.
   (b) 如果 nn 比下界还要大 (下界指的是 rq 中距离目标向量最远的节点对应的距离) 则结束循环.
   (c) 遍历所有 nn 的没被探索过的邻居 ng: (被加入到 rq 或者 cq 过则说明被探索过)
      i. 如果 ng 的距离比下界小则把 ng 加入 cq 和 rq.
      ii. 删除 rq 中的距离最远的几个节点, 直到 rq 尺寸小于等于 k.
4. 取出 rq 的节点作为返回值 (循环中已经保证其尺寸小于等于 k).

此算法相比我自己想的优先级队列搜索, 令我为之震惊的点在于其使用两个优先级队列, 每次循环弹出队列中最近的节点进行探索而不是最远的节点, 还有其使用下界进行判断的方式也令我大开眼界.

我将这个算法和结合 searchNearestNode 使用, 形成了 HNSW 算法 KNN 搜索部分.

4.4 结果

3679154	2024-05-31 20:51:39	2024-05-31 20:51:49	C++17	Wrong answer: 7	0.726	8.344

图 8: HNSW KNN Score

第一次提交时, 此分数让我有点失望, 7 分的分数驱使我不断反复创建二维数据集反复测试准确性, 在各种测试集上, 其显示的结果大致都如以下输出, 召回率十分低:

```
           HNSW   Baseline                  HNSW  Baseline
Node:      7399:      6164,  Distance: 2.969491, 2.327806
Node:       271:      3151,  Distance: 4.204982, 2.510265
Node:      2427:      7996,  Distance: 4.331919, 2.635693
Node:      7636:      7399,  Distance: 4.670664, 2.969491
Node:      7415:      5568,  Distance: 5.209835, 3.072600
Node:      5968:      7346,  Distance: 5.450563, 3.223833
Node:      7769:       974,  Distance: 5.527832, 3.325562
Node:       407:      5588,  Distance: 5.820290, 3.325731
```

```
Node:       2921:        766,  Distance: 5.860978, 3.738243
Node:       3653:       4222,  Distance: 5.956301, 4.033727
Node:         56:       6309,  Distance: 5.979393, 4.123006
Node:       7477:       3586,  Distance: 6.249059, 4.130256
Node:       6353:       3771,  Distance: 6.266310, 4.158977
Node:        199:        271,  Distance: 6.335025, 4.204982
Node:        143:       6780,  Distance: 6.611314, 4.210038
Node:       5940:       5613,  Distance: 6.762243, 4.237144
Node:       5657:       4479,  Distance: 6.766348, 4.268434
Node:       6398:       3673,  Distance: 6.857042, 4.282012
Node:       5173:       2427,  Distance: 6.887219, 4.331919
Node:       4997:       4762,  Distance: 7.156574, 4.376074
Node:       7049:       3239,  Distance: 7.348852, 4.410323
Node:       3473:       3915,  Distance: 7.376329, 4.419710
Node:       7749:       2220,  Distance: 7.424413, 4.509404
Node:       7734:       2714,  Distance: 7.467890, 4.532644
Node:       1175:       2070,  Distance: 7.702319, 4.552542
Node:       3026:       1260,  Distance: 7.765935, 4.557267
Node:       6934:       1736,  Distance: 7.768849, 4.577296
Node:        146:       1164,  Distance: 7.918658, 4.613334
Node:       4823:       6265,  Distance: 8.089707, 4.626596
Node:       7600:       6202,  Distance: 8.109049, 4.629465
```

4.5 陷入局部最小值

在分析画图过程中，我发现我写的算法里的问题：容易陷入局部最小值.

下图显示了为何会陷入局部最小值.

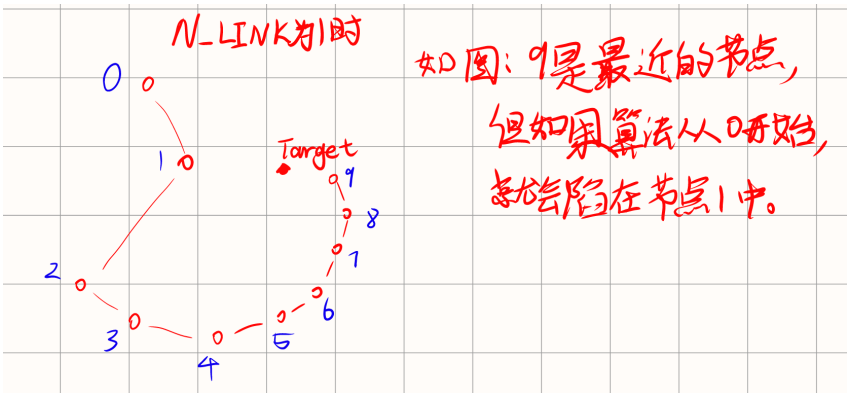

图 9: Local Minimum

从图中也能看出，陷入局部最小值的一个显著影响因素就是 N_LINK 参数，我在第一次提交时设置了 N_LINK

为 2. 增加 N_LINK 的值可以减小陷入局部最小值的可能性, 但是会导致每个节点和其他节点的连接变多，会增大内存的占用和时间消耗.

4.6 修改参数

于是我斗胆一试，把 N_LINK 逐渐增加，令我意外的是，结果不断变好，一路冲上 93 分，算法终于发挥了显著的速度优势.

评测结果	耗时	内存	N_LINK
Time limit exceeded: 93	1.559	11.805	10
Wrong answer: 93	1.431	11.273	9
Wrong answer: 86	1.342	10.641	8
Wrong answer: 86	1.267	10.277	7
Wrong answer: 64	1.221	9.824	
Wrong answer: 29	1.100	9.449	
Wrong answer: 7	0.730	8.348	
Wrong answer: 7	0.843	8.719	
Wrong answer: 7	0.726	8.344	
Wrong answer: 7	0.729	8.602	2

图 10: HNSW Score Increasing

4.7 AC

1.559，很接近 1.5s 的时间限制，抱着撞撞运气的想法，以 N_LINK 为 10 提交了几次，竟然真的给我撞出 AC 了 (代码中没有任何算法依靠随机生成).

评测结果	耗时	内存	
Accepted: 100	1.493	11.715	N_LINK = 10

图 11: AC

4.8 测试集时间性能分析

在测试集 b8000_q45000_k30_d8_1.in 作为输入的情况下，时间性能分析如下:

```
Parameters: { DK_LAYER1: 100, DK_LAYER2: 500, DK_LAYER3: 1000, N_LINK: 10 }
Building Result { cnt: 8000, time(ms): 250, avg: 32000.000000 Hz }
Searching Result { cnt: 45000, time(ms) : 6032, avg: 7460.212202 Hz }
Total: 1350000, Correct: 1316681, Recall: 97.53%
```

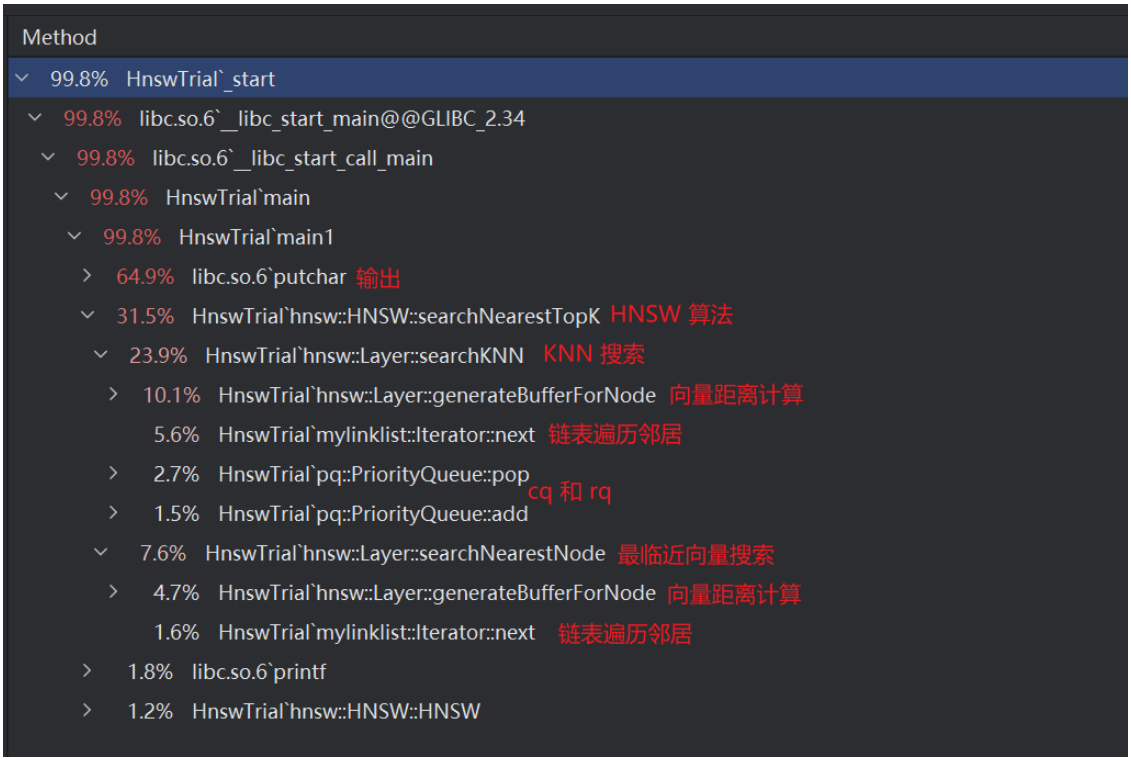

图 12: HNSW Test Dataset Profile

4.9 中等数据集上的表现

此算法在 sift.in 为输入时，表现如下:

```
Parameters: { DK_LAYER1: 100, DK_LAYER2: 500, DK_LAYER3: 1000, N_LINK: 10 }
Building Result { cnt: 1000000, time(ms): 471019, avg: 2123.056607 Hz }
Searching Result { cnt: 10000, time(ms) : 18920, avg: 528.541226 Hz }
Total: 1000000, Correct: 775206, Recall: 77.52%
```

5 项目结构

此次作业可在本人的 github 仓库: DataStructureClassFinalAssignment 上找到项目源码.

- 项目根目录 `*Trial.cpp` 存放了我的不同的尝试代码, `MergedResult.cpp` 是所有代码的整合, 可读性不强, 是交到 EOJ 上的代码.
- `src` 目录下存放了我自己写的基本数据结构.
- `src/hnsw` 目录下存放的是 hnsw 代码.

编写此算法时, 我的代码大致框架是:

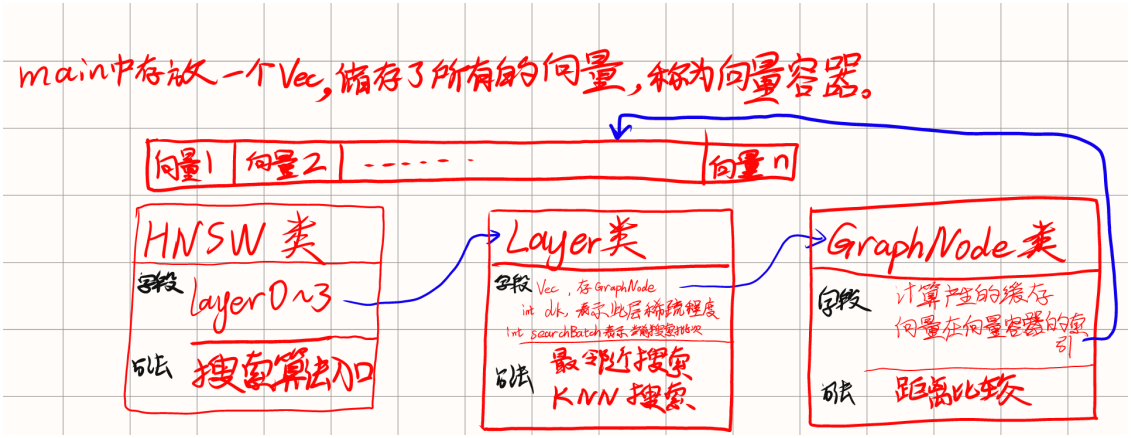

图 13: Code Structure

另: 代码内有详细的注释.