Lab2-HNSW report

张越 522031910791

2024 年 5 月 28 日

1 背景介绍: 向量数据库与 ANN 索引

在本次 lab 中，要求自主实现了一种基于图的 ANN 索引 HNSW 算法的整体框架以及 insert、query 功能，HNSW 是基于 NSW（Navigable Small World）进行优化的算法，旨在处理高维向量空间的最近邻搜索问题时，能够提供高效的查询性能，同时保持较低的空间复杂度。

NSW 算法的简介如下：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。

HNSW 算法的简介如下，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。

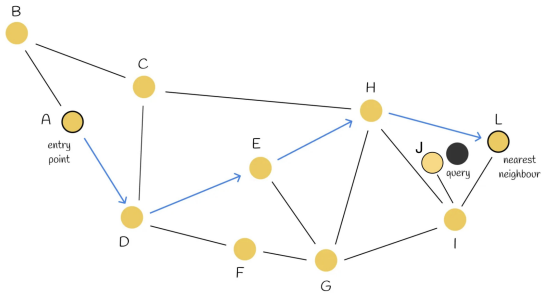

图 1: NSW 示意图

2 系统实现

简要介绍：本次 Lab2 中、HNSW 部分已经提供了伪代码以及主体代码框架，总体实现起来比起 Lab0、Lab1 要简单一些，在实现过程中，也并没有遇到很难实现的地方，主要的问题

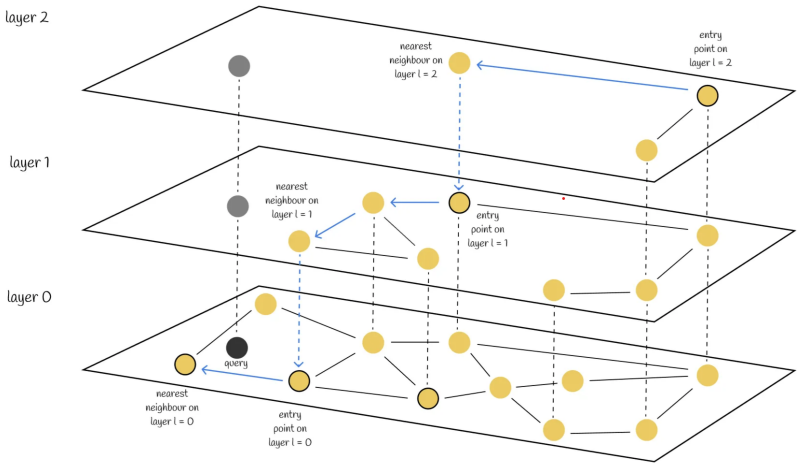

图 2: HNSW 示意图

集中在 HNSW 类的构造，每一个结点的数据结构类型，在本次 Lab 中我构造了 HNSWNode 的类用来存储 HNSW 图里的结点，每个结点存储了 lable 存储其 index 的值、Vector(int*) 存储每个结点的位置、还有 neighbors 保存其各层邻居结点的 index、对于 HNSW 类我主要把每一层的图保存于 Graph(std::vector<HNSWNode>) 中，Graph 的键值为结点的 index，而其值为 HNSWNode 类，保存结点的所有信息。此外我还设计了 max level 用来保存当前 HNSW 最高层级的值、以及 Entry 用于记录进入查找的结点信息。

insert 功能我设计了两个版本实现 (其实区别很小 hhh，但是对于其召回率有一定程度的区别...), 第一个版本是根据伪代码实现的版本，第二个版本大致上于第一个版本并没有任何区别，但是在进入结点与最高层级的更新的策略上有一定的区别——第一个版本是每一个结点随机分配一个 level，然后如果此结点的 level 大于目前的 max level 时对于此 HNSW 类的 max level 与 entry 进行更新：将 entry 更新为当前节点的信息。而第二个版本进行了改变-我在 HNSW 类中保存了节点数量的信息，当第一个节点插入时，将这个节点的层级直接设为 max level，入口也设为此节点，此后不再对 max level 和 Entry 进行改变。经过对于这两个版本的 insert 以及 query 结果的测试后，我发现第二个版本的召回率大于第一个版本的召回率，但性能略差于第一个版本。

——（至于其他逻辑与伪代码一致，首先使用 mL 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）然后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。最后自 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。

而 query 部分、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，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。

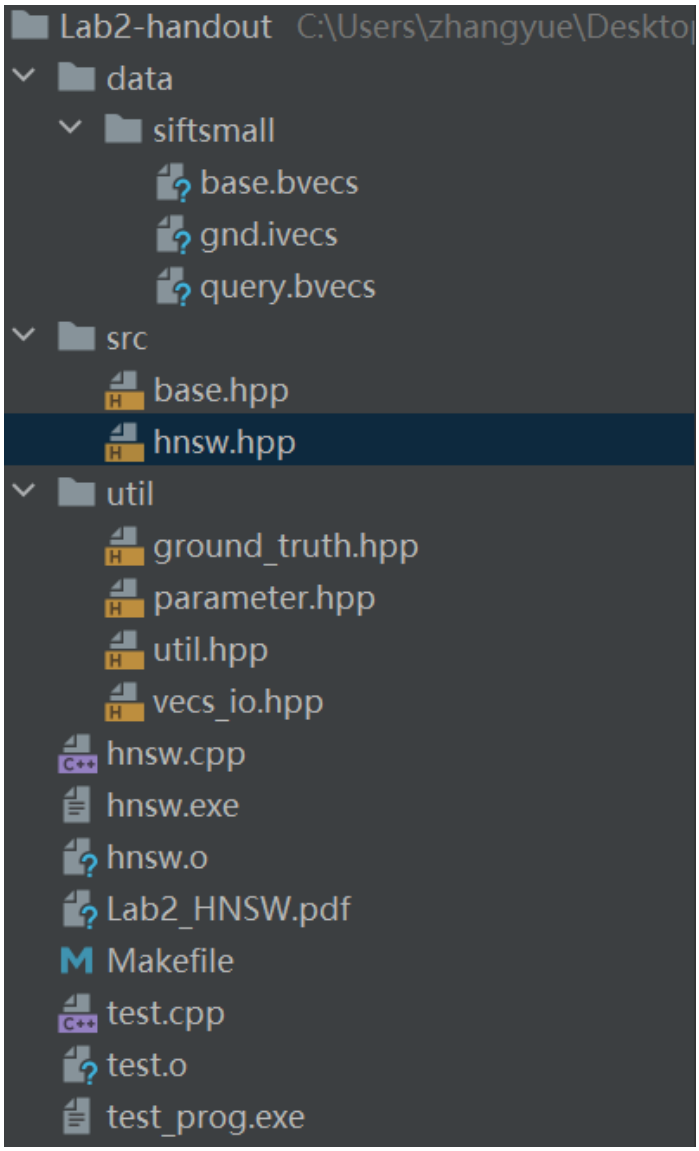

图 3: 代码主体部分框架

3 测试

3.1 正确性测试

```
5439 5460 5773 5810 5924 6047 6745 6882 6906 7419
4767 7857 8082 8782 9163 9240 9338 9846 9942 9993
average recall: 0.942, single query time 2.2 ms
PS C:\Users\zhangyue\Desktop\Lab2-handout1>
```

图 4: version1-test

```
5439 5460 5773 5810 5924 6047 6745 6882 6906 7419
2894 4767 8082 8782 9163 9240 9338 9846 9942 9993
average recall: 0.975, single query time 2.3 ms
```

图 5: version2-test

：可见在参数分别：M max=30,efconstruction = 100 的情况下，两个版本的召回率分别为 0.942 和 0.975，均高于 0.9 的要求、而优化过的版本高于 0.95，满足召回率要求。

3.2 性能测试

3.2.1 测试配置

测试对象：HNSW 串行版本的 insert 性能、query 性能以及召回率大小与 M max 的关系。即 M max 参数对于 HNSW 算法召回率和程序性能的影响。

系统配置: Windows 系统，使用的 IDE 是 Clion , 语言：C++ 17。

机器配置：处理器: 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz

3.2.2 测试结果

```
whole insert time for 10000 times 16099.9 ms
average recall: 0.996, total query time for hundred times 213.4 ms
```

图 6: 部分测试结果

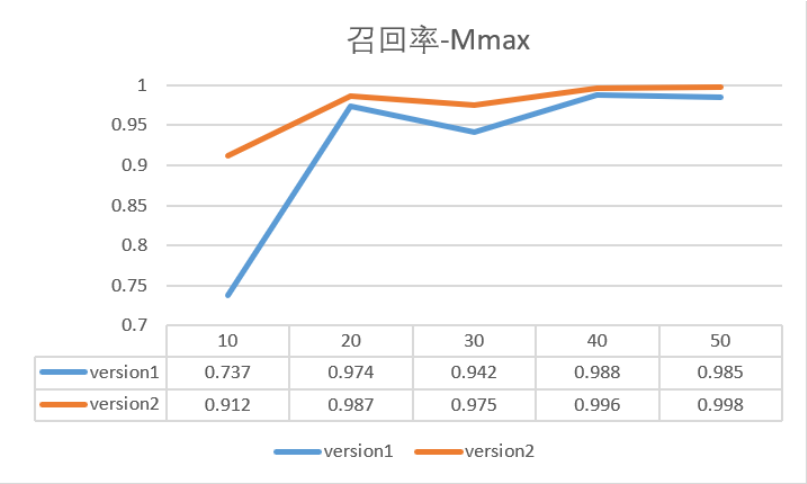

图 7: 测试结果-召回率与 M max

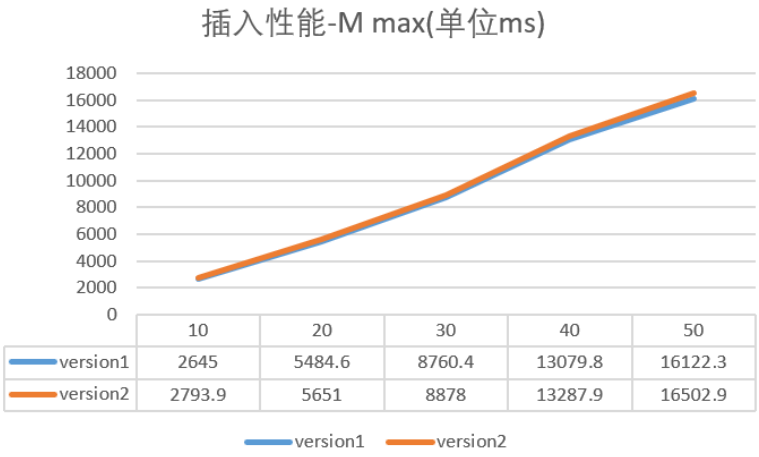

图 8: 测试结果-插入性能与 M max(注：单位 ms, 插入 10000 组数据)

3.2.3 结果分析

首先研究 Mmax 最大层数与各项性能的关系，我发现不论是查询的性能还是插入的性能都随最大层数 Mmax 增加而增加，且在一定程度上可能有线性的关系，原因应该是随着层数增加，从入口开始找到目标节点要经过更多的层数，尽管每一层的节点可能更加稀疏、但总体而言、随着最大层数增加经过节点的数量是线性增加的。

再对于召回率进行研究，我惊讶发现虽然总体而言召回率在随层数增加而增加、但这种规律并不是普适的，我们可以很容易发现召回率在层数从 20 变为 30 时、召回率下降了，这可能

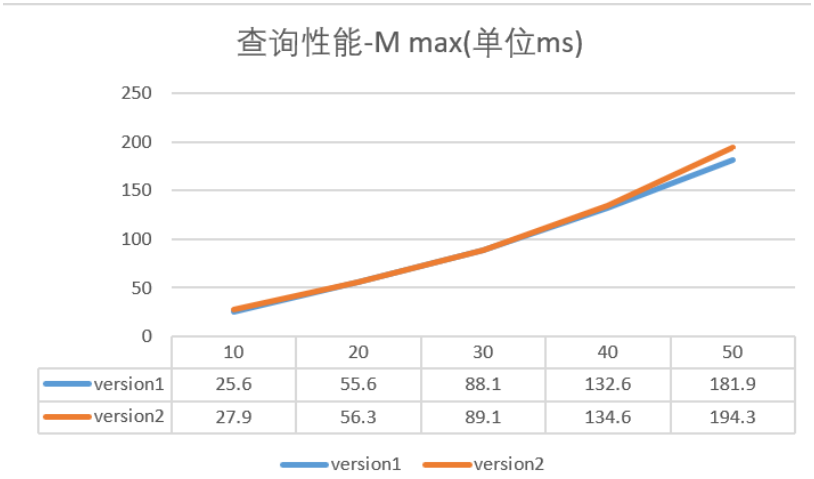

图 9: 测试结果-查询性能与 M max(注：单位 ms, 查询 100 组数据)

有多种原因，当层数增加时，HNSW 算法在粗略层上搜索的范围更广，这有助于在早期排除更多的非最近邻候选，从而提高召回率。但是，如果层数过多，可能会导致在精细层上搜索时错过一些真正的最近邻，因为粗略层的简化可能太大，从而降低召回率，因此召回率不总是随最大层数增加而增加。

最后横向分析两个版本的 HNSW 同层情况下的召回率与各性能的差异、通过对比后我们发现在召回率上、直接将第一个节点设为最高入口节点的策略可以提升整体的召回率、但会降低一定的性能。整体可以将召回率提升 1% 到 3%，但是会使平均查找、插入时延降低 0.01ms-0.05ms(应该也是微乎其微的区别。。。)

最大的差别在于面对高精度要求的向量库需求时，优化过的 HNSW 算法显然能更好完成这样需求，在高层数的 HNSW 算法中第二个 HNSW 显著优于第一个版本，事实上在 M max 大约大于 80 层时、该 HNSW 的召回率可以到达 100%，而在同层级下，未更改的版本的召回率仅为 34-55%，经过思考我认为出现这种情况的可能是因为在不断升高层级时会出现更多的孤立点，而假设我们一开始就标记了最高点并且固定了入口，那么所有后面的点都能经由第一个点进行关联、从而生成关联更加紧密的 HNSW 图，最终获得更高的召回率！

总而言之，选择 HNSW 算法的最大层数 Mmax 时，我们需要在查询性能、插入性能和召回率之间进行权衡。随着层数的增加，查询和插入性能通常会提高，但召回率可能会呈现非单调的变化趋势。为了确定最佳层数，应该根据具体应用的需求进行实验和评估，找到能够满足召回率要求的同时，保持查询和插入性能可接受的平衡点。对于高精度要求的场景，可能需要更多的层数来保证高召回率，同时也应该适当改变启发性策略，尽管这可能会牺牲一些性能。

```
5439 5460 5773 5810 5924 6047 6745 6882 6906 7419
2894 4767 8082 8782 9163 9240 9338 9846 9942 9993
average recall: 1.000, single query time 11.9 ms
```

图 10: Mmax-80 HNSW1 的召回率

```
2309 4024 4997 5184 6198 7095 7881 8024 8279 8400
average recall: 0.340, single query time 8.8 ms
```

图 11: Mmax-100 HNSW2 的召回率

4 结论

在这次 Lab 中，我自主完成了 HNSW 算法基本功能的实现，进行了正确性测试以及各种性能测试，通过更换 Mmax 并对插入查询性能以及召回率进行测试，了解了 HNSW 参数对于性能和正确性的影响。同时也加深了我对于图的理解，也了解了向量数据库 ANN 索引的高效率算法。最有趣的是通过自己的一个小小的改变提升了伪代码里 HNSW 算法的正确性，并且全面的对自己的改进进行全方面的分析，从正确性到性能测试、也可以说把前两次 lab 学习的 YCSB 测试实际应用到了这次的 Lab 中、也让我切身体会到前两次 lab 对于我自身编程水平以及测试水平的实际锻炼！

总体而言，对于功能实现部分，由于吸取了 Lab0-哈夫曼压缩与 Lab1-基数树的经验，总体主体部分的实现比较顺利，也让我对于各种图的算法以及功能有了更深刻的认识、也了解了部分关于向量数据库查询的知识，可谓受益匪浅！

5 建议

本次 Lab 中我觉得我还可以进一步思考 HNSW 算法中建图的过程、并且研究更多启发性的策略优化 Insert、建图的过程。

附言：总体而言感谢这次 Lab 的经历以及助教老师相助，让我对于不同数据结构的认识有了进一步的了解。

——由于本人学疏才浅，做这个 Lab2 时已经绞尽脑汁、同时又由于最近事务繁多、被 LSM-KV Tree 以及各学科大作业的 ddl 所困、实属分身乏术，上述建议中提到进一步的研究恐怕已是无力为之，由于做 LAB 的时间较为紧迫, 程序也未免十分周全，能将所有情况纳入考虑，只得匆匆为之，若有纰漏之处，恳请助教与老师的谅解。

参考文献