華中科技大學

2024

人工智能导论实验报告

专　　业：计算机科学与技术（图灵班）

班　　级：图灵 2301

学　　号：U202312454

姓　　名：万睿朋

电　　话：18873328139

邮　　件：2167306203@qq.com

完成日期：2024 年 1 月 18 日

摘要：

随着信息技术的发展，海量高维数据的高效存储与快速检索成为关键问题。近似最近邻搜索（ANNS）在图像识别、自然语言处理等领域应用广泛。

本报告重点探讨了 HNSW 算法的理论基础、实现细节及其在实际应用中的表现，给出了可以简单支持在线删除的 HNSW 算法，并将 HNSW 算法应用于实际项目。

本文深入探讨了分层可导航小世界图（HNSW）算法的理论基础、实现细节及其性能。通过 C++实现 HNSW 算法，并与传统枚举算法进行对比测试。除此以外，基于 HNSW 算法，开发了本地文件检索软件 Vectri，实现了文件名向量化、模糊匹配和语义检索，显著提升了检索效率和用户体验。最后，分析了 HNSW 算法的优势与不足，并展望了其在算法优化、分布式计算及跨领域应用等方面的发展方向。

关键词 ：

近似最近邻搜索（ANNS）、HNSW 算法、小世界图、向量检索、文件检索软件、C++

目　录

1　引言：数据的快速检索与 ANNS 问题

1.1 问题背景

在当今信息爆炸的时代，海量数据的高效存储与快速检索成为各类应用系统面临的核心挑战之一。特别是在诸如图像识别、自然语言处理、推荐系统以及生物信息学等领域，快速查找的需求日益增长。近似最近邻搜索（Approximate Nearest Neighbor, ANN）作为一种关键技术，广泛应用于高维数据的相似性检索与分析。

朴素的最近邻查询算法需要与数据量成线性关系的时间复杂度进行单次查询， 这在数据量越来越大的当下是难以接受的， 这对我们探寻单次查询时间复杂度更低的算法提出了强烈的需求。

1.2 HNSW 算法的引入

图结构作为一种灵活且高效的数据组织方式，在 ANN 算法中展现出显著的优势。尤其是小世界图（Small World Graph）因其高连接性和短路径特性，成为构建高效检索系统的重要基础。然而，传统的小世界图在处理大规模高维数据时，仍面临构建复杂度高和查询效率低下的问题。为此，Hierarchical Navigable Small World（HNSW）算法应运而生，作为一种基于分层小世界图的新型 ANN 算法，HNSW 在提高检索速度和精度的同时，有效降低了图的构建与维护成本。

HNSW 算法通过引入多层次的图结构，结合贪心搜索策略，实现了在高维空间中快速且准确的最近邻搜索。其独特的层次化设计不仅优化了数据的组织方式，还增强了算法的可扩展性和适应性，使其在大规模数据集上的表现尤为突出。自 HNSW 算法提出以来，已在多个实际应用场景中展现出卓越的性能，如图像检索系统中的实时相似图像搜索、推荐系统中的个性化内容推荐以及自然语言处理中的语义相似度计算等。

1.3 报告结构

本报告旨在全面探讨 HNSW 算法的理论基础、实现细节及其在实际应用中的表现。首先，将介绍近似最近邻搜索的基本概念及其在各领域的重要性，随后深入解析 HNSW 算法的核心算法， 然后使用 C++ 语言对算法进行简单的实现、优化与测试。我们将在测试中展示 HNSW 算法与传统枚举算法的性能、效率上的优势。之后我们将给出 HNSW 算法支持在线删除的扩展方法。最后我们将把 HNSW 算法实际运用到工程中，实现一个简单的可以支持模糊匹配等效果更好的文件检索程序。

2 ANNS 问题简述

2.1 问题定义

在高维空间中，最近邻搜索（Nearest Neighbor Search, NNS） 是指在给定的数据集中，找到与查询点在某种度量下距离最近的点。 具体而言，也就是给定一个包含 n 个向量的集合 S， 我们需要实现的查询目标就是找到集合中与查询向量 q 距离最小的向量 x*。

然而， 当数据量极为庞大的今天， 精确搜索的效率往往是不可接受的， 为此，近似最近邻搜索（Approximate Nearest Neighbor Search, ANNS） 应运而生，其目标是在保证一定精度的前提下，显著提升搜索效率。ANNS 允许搜索结果为最近邻的近似值，而非精确值，从而在时间和空间复杂度上取得更优的平衡。具体而言， 我们的查询目标变更为**尽量**找到集合中与查询向量 q 距离最小的向量 x*， 同时应该保持有**较快**的速度。

2.2 ANNS 问题的重要性

随着大数据和高维数据在各个领域的广泛应用，ANNS 在信息检索， 推荐系统实现， 计算机视觉， 自然语言处理， 生物信息学等方面都非常重要。

例如， 在信息检索系统中， 如果 ANNS 准确率太低， 则会导致无法检索到需要检索的内容。而如果 ANNS 速度太慢，则会出现查询一次等待十几秒的尴尬情况，这都是产品的使用者无法接受的。

再如， 在计算机视觉与自然语言处理领域， 向量的相似性计算是用于语义分析或者是特征向量匹配的重要一环， 这将显著影响相关机器学习、深度学习算法的运行效率与学习结果。

可见， ANNS 的高效性直接影响到这些应用的性能和用户体验，研究高效的 ANNS 算法具有极其重要的实际意义。

2.3 ANNS 的典型方法

数年来，科学工作者们提供了若干 ANNS 问题的处理方法。这些方法主要可以分成如下几类 ：

2.3.1 基于空间划分的方法

这一类方法的为人熟知的典型代表是 KD-树（k-Dimensional Tree）， KD-树是一种二叉树数据结构，通过在每个节点上选择一个维度进行划分，将空间递归地划分为更小的子空间。这种方法在维度比较低的时候有极高的效率， 如 2～3 维的时候表现都相当优秀，但在高维空间中性能下降显著。

除此以外， 球树（Ball Tree）也是一种效率不错的方法， 但其在非常高维的空间中仍然面临性能问题。

2.3.2 基于哈希的方法

这些方法利用哈希函数将相似的数据点映射到相同的桶中，以减少搜索空间。例如，局部敏感哈希（Locality-Sensitive Hashing, LSH）是一个常见的算法。 这个方法通过设计特定的哈希函数，使得相似的数据点更可能被映射到相同的桶中，然后在哈希桶内进行精确搜索来完成问题。这是一个很有意思的想法， 在高维空间中依然能保持较高的搜索效率，适用于大规模数据集， 但需要多组哈希函数以提高搜索覆盖率，大幅增加了存储和计算成本。

类似的， 也存在超球面哈希（Hyperplane LSH）的方法， 利用超平面将空间划分为不同的区域，通过计算数据点相对于多个超平面的投影位置进行哈希划分， 但这同样需要多组哈希函数， 增加了存储和计算成本。

2.3.3 基于图的方法

这些方法通过构建图结构，将数据点作为节点，边表示数据点之间的相似性，通过图遍历实现高效的近似搜索。其中， Navigable Small World (NSW) 算法是一种基于图结构的近似最近邻搜索（Approximate Nearest Neighbor Search, ANNS）算法。NSW 利

用小世界图（Small World Graph）的特性，通过构建高效的图结构，实现高维空间中数据点的快速相似性检索。但这种方法对内存要求高， 搜索效率和精度依赖于图结构的质量，不同数据集可能需要不同的图构建策略， 且动态更新困难， 具有较大的局限性。本报告将要分析的 HNSW 算法即为 NSW 算法的分层优化， 在 NSW 的基础上显著提升了性能，使其在更大规模和更高维度的数据集上表现优异。

2.3.4 混合方法

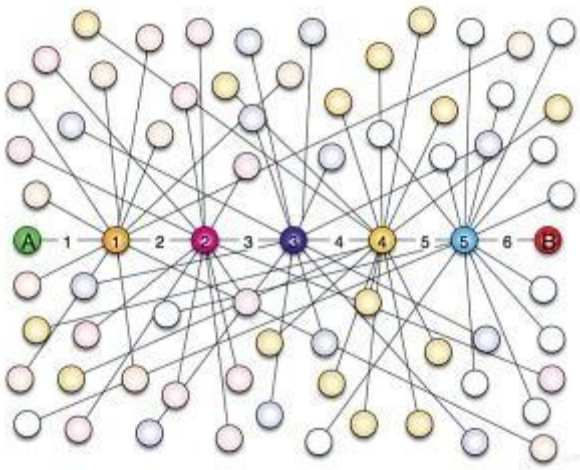
由于方法各有所长， 在实际运用中， 我们常常将各类算法结合使用。例如， 我们先进行一个小范围的哈希算法， 再将哈希算法中的精确搜索替换为 HNSW 算法，这都是可行的。

3　NSW 算法

3.1 NSW 算法的基本原理-“六度分离理论”

NSW 算法的核心思想基于“小世界图“的特性， 即通过少量的随机长距离连接和大量的局部短距离连接，形成一个高效的网络结构。

在讨论算法设计之前， 我们先来讨论所谓的 “六度分离理论”。



图表 1 六度分离理论示意图

1967 年， Stanley Milgram 基于他的实验提出了著名的**六度分离理论**，这个理论指出:

1．现实世界中的短路径是普遍存在的。

2. 人们可以有效地找到并且利用这些短路径。

通过模拟现实世界的情况，我们设计“小世界”网络，把点和点的关系分为两种，分别是“同质性”和“弱连接”。其中，同质性也就是相似的点会聚集到一起，相互连接具有邻接边。而弱连接是指从每一个节点上，会有一些随机的边随机连接到网络中的节点上，这些节点是随机均匀的。

这样， 可以发现我们的“小世界”与现实是类似的。也就是说， 理论上如果我们的图建得还不错， 也只需要走几步路就可以找到离询问点最近的点。

3.2 NSW 算法过程

我们的**构建算法**有如下两个步骤：

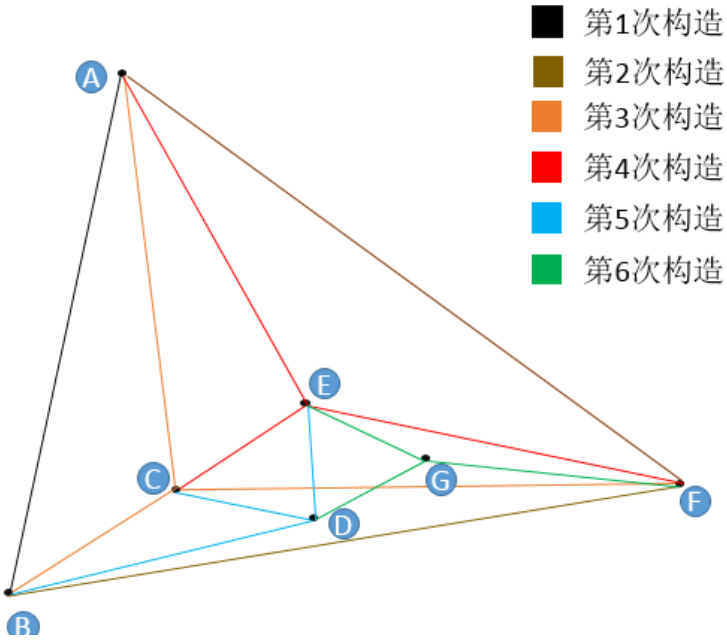
1. 首先， 把所有点的顺序打乱， 依次插入图。

2. 每当插入一个点时， 我们使用查找算法找到离这个点最近的 m 个点， 并把这个点和这 m 个点连边。

可以发现一个问题是， 这个“查找算法”应该如何选取呢？事实上， 这个“查找算法”一般就使用 NSW 算法的查找算法， 这和我们查询向量的最近邻是一样的。而 NSW 的查找算法可以采用如下的朴素 DFS 算法 ：

1. 从随机一个点出发， 把这个点和与这个点有连边的点存在集合 S 中， 并计算其与我们要查询的向量的距离。

2. 接下来走到和这个点有连边的距离查询向量最近的点， 执行步骤 1， 如果 S 集合的距离前 k 小没有发生变化， 则停止当前点的 DFS， 否则继续步骤 2。其中 k 是一个可以指定的参数。



图表 2 增量式朴素构造示意图

可以发现， 这个算法有很大的优化空间， 比如利用数组标记重复走过的点， 或者利用数据结构求前 k 大。这些优化都是可行的， 我们将在 HNSW 算法的构建中使用， 不在这里进行详细讨论。

3.3 NSW 算法的优点与局限

可以发现，NSW 算法的实现实际上非常简单，这给并行计算带来了很大的优势，但是这个算法有很多难以克服的问题。

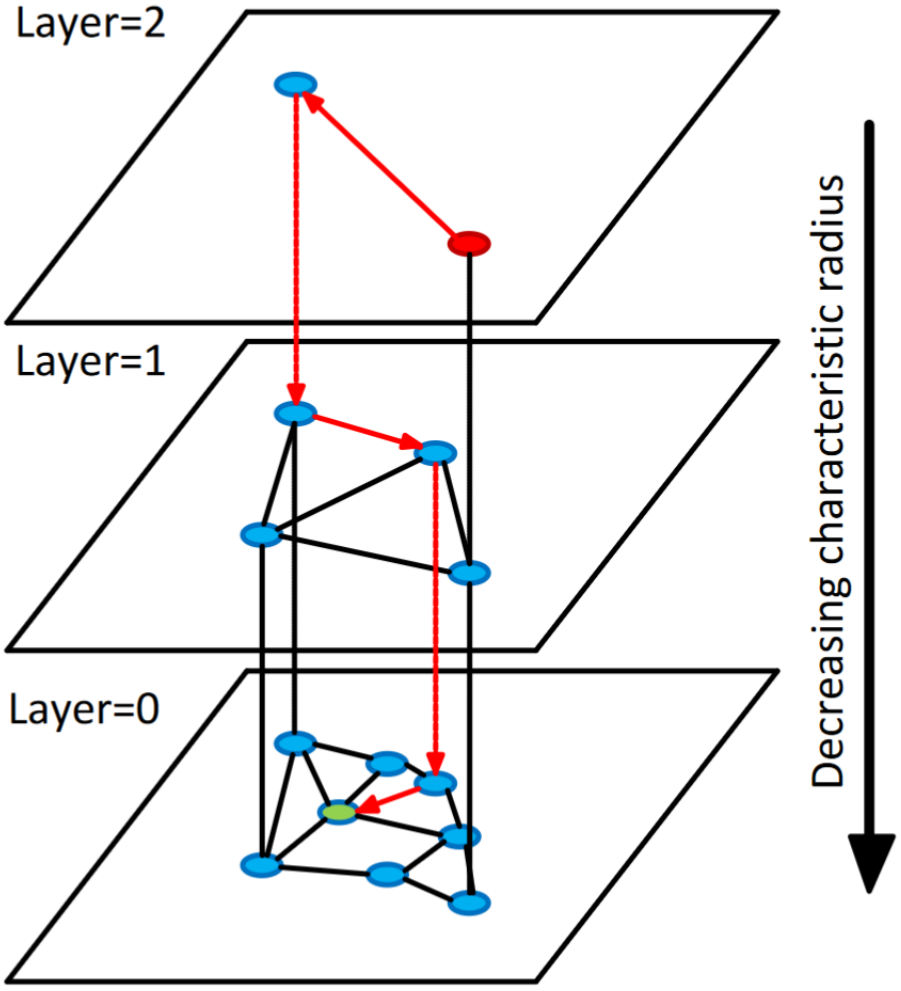
首先可以发现， 搜索效率和精度**依赖于图结构的质量**，不同数据集可能需要**不同**的图构建策略， 图结构不完善可能导致搜索路径较长或搜索失败。除此以外， 在数据频繁更新的场景下，保持图的高质量和高连接性较为困难， 因为我们的算法依赖于插入顺序的随机性。除此以外， 可以发现我们的搜索算法很有可能在时间复杂度上发生严重的退化。 一个例子是， 我们的数据集中的向量构成二维平面上的一圈点， 那么我们单次查询的时间复杂度将会**退化**为 $O(n)$ 甚至是 $O(n \log k)$，这是我们所无法接受的。为了解决这些问题， HNSW 算法应运而生。

4 HNSW 算法

4.1 HNSW 算法的基本原理

HNSW 算法基于 NSW 算法进行改造。观察到在 NSW 算法中，我们的只有一个图，我们算法的性能高度依赖于这个生成的图，图结构不完善可能导致搜索路径较长或搜索失败。但在实际应用中，过度依赖于单个图可能在某些数据集下产生严重的性能问题。基于此，我们基于“跳表”的想法，对 NSW 算法进行改造。

在 HNSW 算法中，我们建立“分层图”，顶层只有少量节点，而底层包含所有数据点。每一层的节点是从底层中选取的一部分节点，通常采用随机分配的层数来决定节点所属的层。而每个节点在其所在层中与若干近邻节点相连，这些连接基于某种近似距离度量。连接的数量通常由参数 M 决定，控制每个节点的最大出边数。



图表 3 HNSW 算法分层图结构

基于这样的结构， 在每次查询中， 我们将从最定层开始往下搜索， 这将是一个“先大致定位， 再仔细搜索”的过程， 大大减小了对数据集的依赖性。与此同时，通过限制搜索次数和点度， 我们把搜索复杂度控制在 $O(K \times M \log_2 N)$ 的级别， 其中 K, M 均为可指定的参数。

4.2 HNSW 算法描述

HNSW 算法主要由 INSERT， SEARCH-LAYER， SELECT-NEIGHBORS-SIMPLE， SELECT-NEIGHBORS-HEURISTIC， K-NN-SEARCH 五个子算法组成，我们将分别讨论各个子算法的过程。以下函数均以伪代码的形式给出， C++ 实现在代码附件中给出。

4.2.1 SEARCH-LAYER

该函数的作用是在同一层中， 找到查询向量的近邻。

函数原型 SEARCH-LAYER(q, ep, ef, lc)：其中 q 代表查询向量； ep 代表该层入口节点； ef 代表一个参数， 表示保留的近似最近邻数量； lc 表示当前层的标号。

可以看到， 在伪代码中， 我们使用了简单的贪心算法和循环找寻最近邻点。在本报告的 C++ 代码中， 我们使用堆来加速了这个过程， 以达到更加优秀的效率。

```
v ← ep // set of visited elements
C ← ep // set of candidates
W ← ep // dynamic list of found nearest neighbors

while |C| > 0
  c ← extract nearest element from C to q
  f ← get furthest element from W to q

  if distance(c, q) > distance(f, q)
        break // all elements in W are evaluated
```

```
  for each e ∈ neighbourhood(c) at layer lc // update C and W
        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
          if |W| > ef
                remove furthest element from W to q
return W
```

4.2.2 INSERT

这个函数的功能是实现单个向量的插入。

函数原型 INSERT(hnsw, q, M, Mmax, efConstruction, mL)：其中 hnsw 即为当前处理问题的 hnsw 类；q 为要插入的向量， M, Mmax, efConstruction, mL 均为可指定的参数。 其中 M, Mmax 用于限制点的最大连接度数，efConstruction 用于限制要保留的最近邻点的个数， mL 表示用于随机确定节点所在层的参数。

在这个插入算法中， 我们先通过参数 mL 随机决定当前插入点的层，然后将这个点插入图中， 使用 HNSW 的层内搜索算法找到这个点的邻居进行连边， 最后更新层的入口节点。

```
W ← ∅ // list for the currently found nearest elements
ep ← get enter point for hnsw
L ← level of ep // top layer for hnsw
l ← ⌊-ln(unif(0..1))·mL⌋ // new element’s level

for lc ← L … l+1
```

```
  W ← SEARCH-LAYER(q, ep, ef=1, lc)
  ep ← get the nearest element from W to q
for lc ← min(L, l) … 0
  W ← SEARCH-LAYER(q, ep, efConstruction, lc)
  neighbors ← SELECT-NEIGHBORS(q, W, M, lc) // alg. 3 or alg. 4
         add bidirectionall connectionts from neighbors to q at layer lc
         for each e ∈ neighbors // shrink connections if needed
           eConn ← neighbourhood(e) at layer lc
           if |eConn| > Mmax // shrink connections of e
                  if lc = 0 then Mmax = Mmax0
                  eNewConn ← SELECT-NEIGHBORS(e, eConn, Mmax, lc)
           // alg. 3 or alg. 4
           set neighbourhood(e) at layer lc to eNewConn
         ep ← W

if l > L
  set enter point for hnsw to q
```

4.2.3 K-NN-SEARCH

这个函数的功能是实现 K 近似最近邻的查询。

函数原型为 K-NN-SEARCH(hnsw, q, K, ef) ：hnsw 为算法类， q 为我们要传入的查询向量， K 表示我们要求近似前 K 大， ef 为可指定的保留最近邻个数。

可以发现， 这个函数简单实现了从上往下逐个搜索的过程。我们的查询从最上面一层开始。首先，找到当前层的“入口点”， 然后在每一层，基于当前层的邻居关系，逐步向下推进， 最终到达最底层时，执行具体的最近邻查找。

```
W ← ∅ // set for the current nearest elements
ep ← get enter point for hnsw
```

```
L ← level of ep // top layer for hnsw
for lc ← L … 1
  W ← SEARCH-LAYER(q, ep, ef=1, lc)
  ep ← get nearest element from W to q

W ← SEARCH-LAYER(q, ep, ef, lc =0)
return K nearest elements from W to q
```

4.2.4 SELECT-NEIGHBORS-SIMPLE

这个函数用于在给定的候选点集合里面选出前若干近的元素。

函数原型 SELECT-NEIGHBORS-SIMPLE(q, C, M) ：q 表示查询向量， C 表示候选点集合， M 表示返回的元素不应当超过多少个。

根据朴素的实现， 使用优先队列取出前 M 小值即可。在本报告的 C++ 代码实现中， 使用 nth_element 函数在线性的时间求出了第 M 小， 在线性的时间中完成了这个问题， 成功将算法的复杂度降低了一个 log N 的级别。

4.2.5 SELECT-NEIGHBORS-HEURISTIC

这个函数同样用于在给定的候选点集合里面选出前若干近的元素。

该函数原型为 SELECT-NEIGHBORS-HEURISTIC(q, C, M, lc, extendCandidates, keepPrunedConnections) ： q, C, M 定义同上， lc 表示当次查询所在层， extendCandidates 和 keepPrunedConnections 是两个参数，分别表示是否要拓展 extendCandidates 集合以及是否添加被抛弃的点。

在实际运用中，使用 SELECT-NEIGHBORS-SIMPLE 函数可能导致在某些具有特殊性质的数据集中连边不够随机， 过于集中而导致查询效率下降。 通过这个启发式的搜索函数，我们进一步使得连边效果更加随机， 增强算法的鲁棒性。

```
R ← ∅ W ← C // working queue for the candidates
if extendCandidates // extend candidates by their neighbors
  for each e ∈ C
    for each eadj ∈ neighbourhood(e) at layer lc
      if eadj ∉ W
            W ← W ∪ eadj

Wd ← ∅ // queue for the discarded candidates
while |W| > 0 and |R|< M
  e ← extract nearest element from W to q
    if e is closer to q compared to any element from R
          R ← R ∪ e
        else
          Wd ← Wd ∪ e

if keepPrunedConnections // add some of the discarded
  // connections from Wd
  while |Wd|> 0 and |R|< M
    R ← R ∪ extract nearest element from Wd to q

return R
```

4.3 对 HNSW 算法的简单测试

4.3.1 测试问题定义

给有 N 个向量，每个向量维度为 dim 的数据集。由于 HNSW 算法的随机性较好， 本问题中向量随机生成， 不会大幅度影响 HNSW 算法的表现。

一共给出 Q 次询问，每次给出一个随机向量，算法需要求出距离这个向量前 K

近的向量。

4.3.2 测试问题评分标准

由于 HNSW 是近似算法，为了评估结果，我们应当引入正确性分数判定算法的正确性。定义评分算法如下：

设单次测试中有 Q 次询问，询问的问题是求前 K 近的向量，输出前 K 近的距离。此处的距离定义为向量的内积。那么每次询问有分数 100 / Q，我们将 HNSW 算法的输出结果与朴素枚举算法的结果进行比对，设 HNSW 算法的结果中有 t 个出现在枚举算法的结果中，则该询问得分 $\frac{100}{Q} \times \frac{t}{K}$，把各个询问的得分求和即得到本次测试的算法得分。

4.3.3 测试环境

测试在一台 Huawei Matebook X Pro 笔记本上进行。

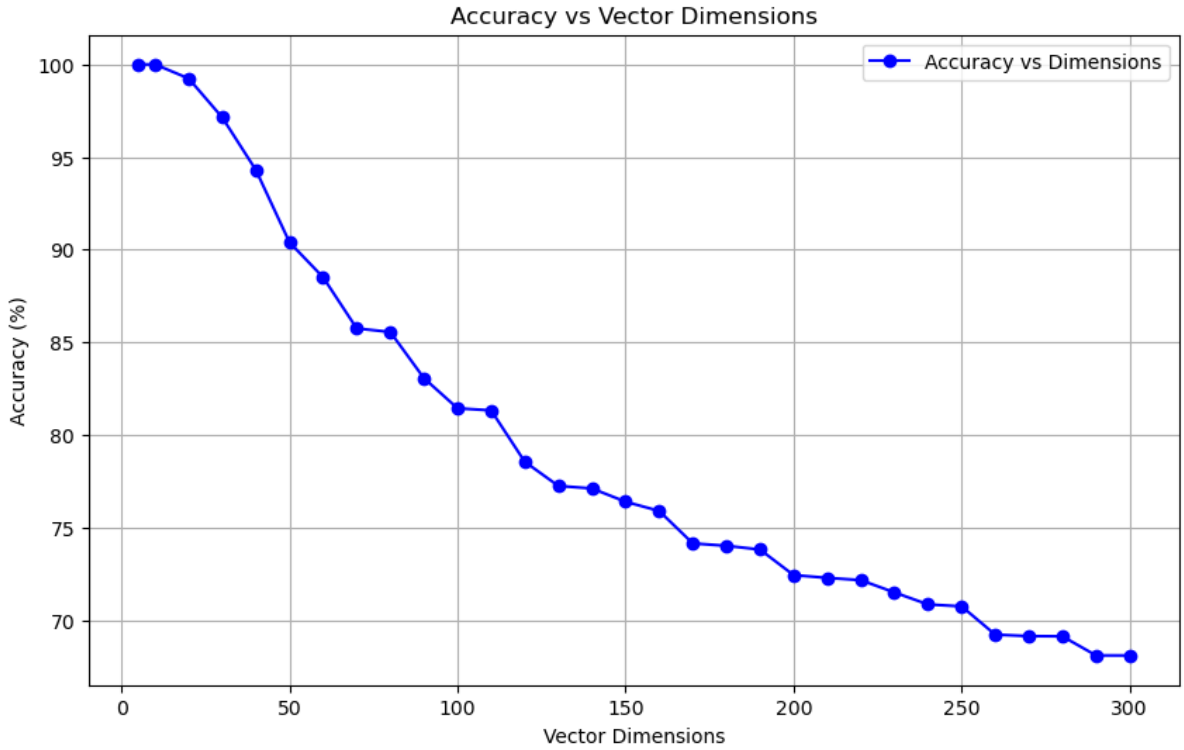
处理器：Intel(R) Core(TM) Ultra 7 155H　3.80 GHz　24 核

内存 ：32 GB

系统 ：Ubuntu 20.04

4.3.4 算法准确性测试

首先为测试“维度灾难”在该算法下的情况，首先对维度不断增大时，算法的“正确性得分”进行讨论。以下在参数 M = 30, Mmax = M, Mmax0 = M * 2, Ml = 1 / log(M), efConstruction = 50,　N = 5000 的情况下进行测试：

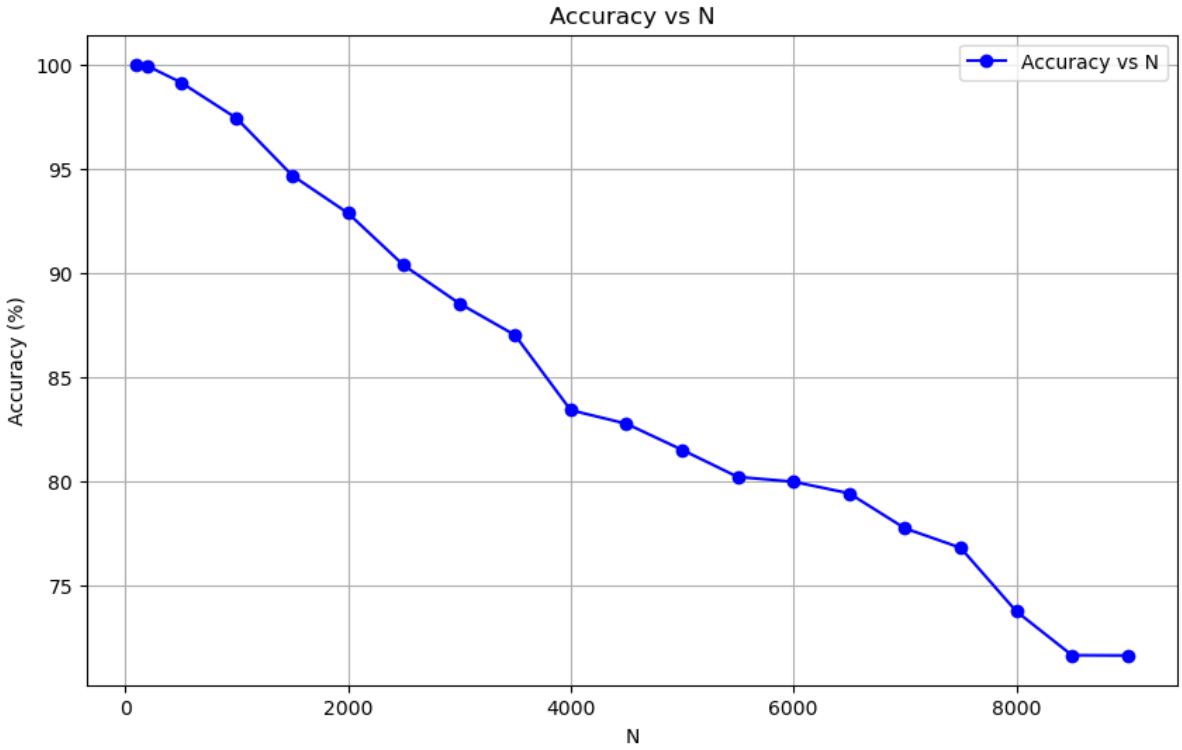


图表 4 维度-正确性得分

可以看到， 随着维度的增加， 算法的准确性显著下降。虽然 HNSW 算法在维度较大的时候仍然有较高的效率， 但算法准确性的下降还是难以避免的。

除此以外， 可以注意到， 算法的准确性在下降到 70 分左右时下降速度明显变慢， 这可能是由于本机算力并不强， 仅在 N = 5000 的条件下测试导致求出接近向量概率比较大， 实际上当 N 进一步增大，可能该部分的下降速度还会加快。

除了维度增加以外， 向量总个数的增加也会给算法准确性带来较大的影响。我们在参数 M = 30, Mmax = M, Mmax0 = M * 2, Ml = 1 / log(M), efConstruction = 50, dim = 100 下进行测试 ：



图表 5 向量个数-正确性得分

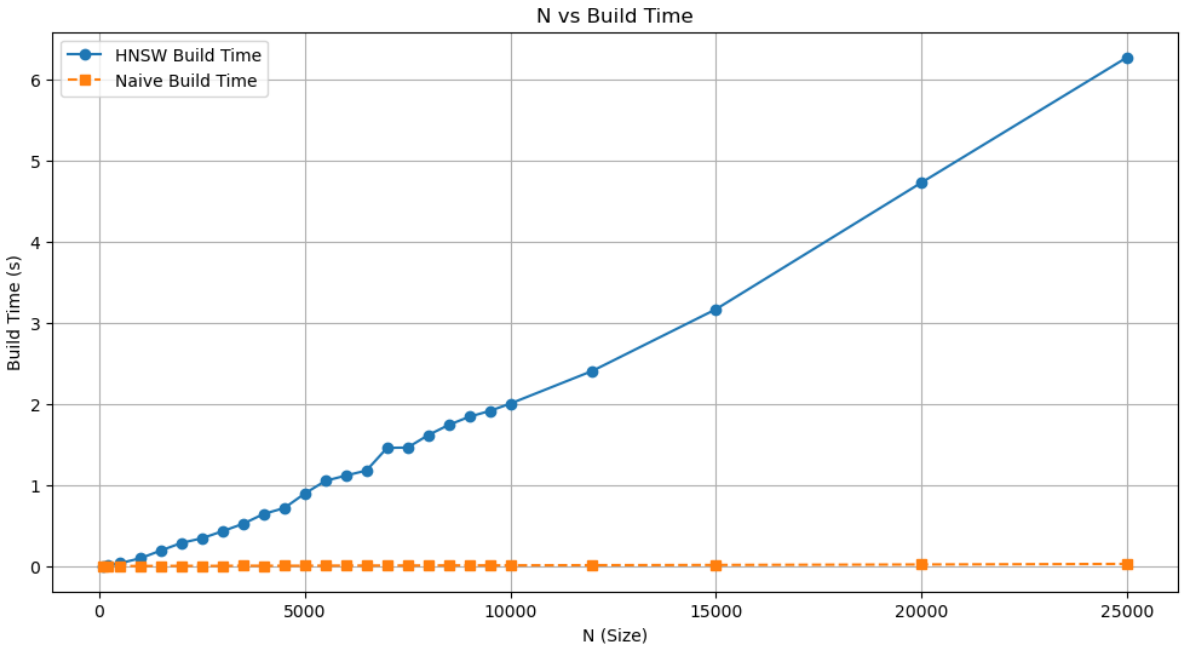
可以看到， 随着 N 增大， 算法准确性下降也是很明显的。因此， **采用分布式系统存储信息**， 在每个子系统**分别**进行向量查询算法是必要的。

4.3.5 算法速度测试

本次测试只对朴素的枚举算法进行测试。为了缩短测试时间， 本处的朴素枚举算法采用了循环展开优化。

测试参数 ： M = 30, Mmax = M, Mmax0 = M * 2, Ml = 1 / log(M), efConstruction = 50, dim = 20。

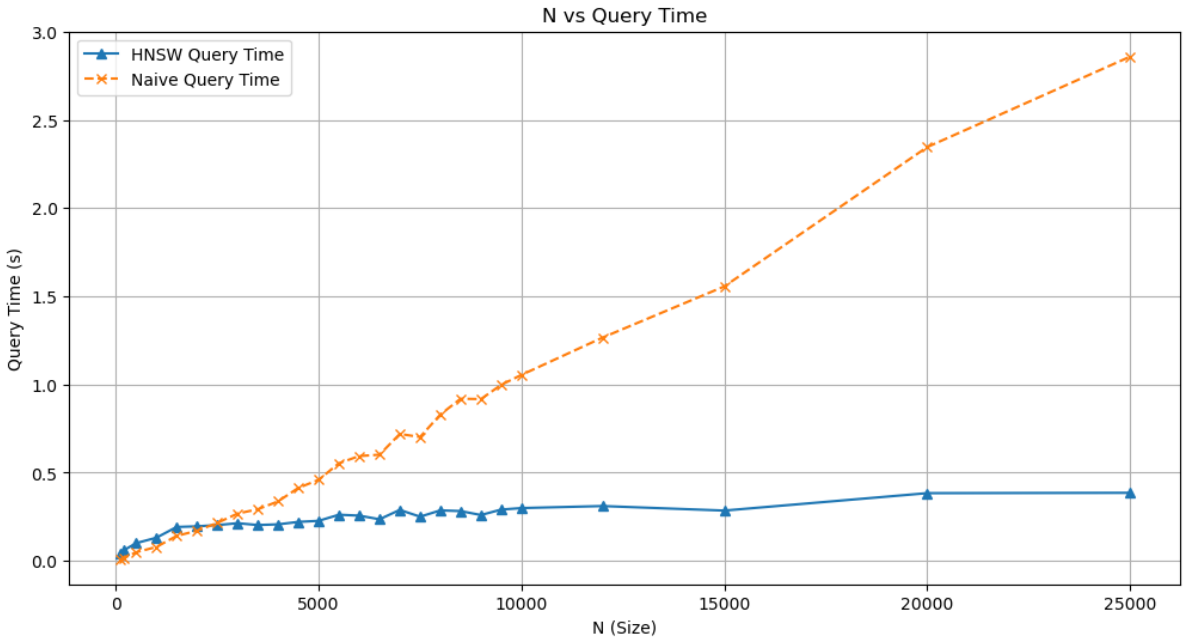
算法构造时间测试结果 ：



图表 6 向量个数-构造时间

可以看到，随着 N 的增大，HNSW 算法的构造时间显著增加，这是因为 HNSW 算法的构造复杂度和 N 是成线性关系的。

查询时间测试结果 ：



图表 7 向量个数-查询时间

可以看到， HNSW 算法在数据量较大的时候， 其查询效率是显著优于朴素的枚

举算法的， 且随着 N 增大， 优势还会越来越大。

4.4 小结

综上， HNSW 是一种优秀的查询最近邻查询算法， 本报告使用简单的 C++ 实现了该算法并进行了简单的测试、优化和分析。

可以看到， HNSW 算法精度高， 效率高， 且算法在图中搜索时可以很容易地并行实现，适用于大规模数据集。但算法也存在构造耗时长，内存需求高的问题， 可以和基于哈希的算法一并使用来进行效率上的平衡。

5 使用 HNSW 算法的本地文件检索软件设计

5.1 动机

在平常使用华为默认的文件检索系统或者 Listary 之类的软件的时候， 我们常常会由于打错字， 或者没有办法想起来精确的文件名来检索文件的位置。我们期望， 借助向量相似性的比较， 可以把文件名转化为一个向量， 利用高速的向量检索算法进行基于词义的文件检索， 以此实现文件名的模糊匹配以及基于“意思”的文件查询。

5.2 支持删除的 HNSW 算法实现

由第 4 节的内容，本报告给出了 HNSW 算法的简单介绍。但是可以发现，HNSW 算法本身是只能在线插入向量，而不支持在线删除的，因此，我们需要对 HNSW 算法进行**可删除性改造**。

虽然 HNSW 算法是不能直接删除的， 但是我们可以给节点打上懒惰删除标记。在查询的时候，我们假设每个节点没有被删除， 如果搜索的时候返回的没有被删除的向量数不够 K 个， 我们就查询 2K 个。如果还不够， 查询 4K 个， 依次类推。

但是有个问题在于， 如果删除的点太多， 这样做会严重影响我们的查询效率。因此， 我们引入重构常数 $t(0 < t < 1)$， 如果删除的点所占的比例大于 t， 我们就把整个查询网络全部重构。在本次项目中， t 的取值为 0.3。

同时， 为了支持删除操作， 我们需要对向量生成一个唯一的哈希值用于标识每个向量。在本项目的实现中， 这个哈希值取为向量被插入的次序。

除此以外， 为了加速查询过程， 代码还进行了一些简单数据结构上的优化， 具体详见代码。

5.3 HNSW 算法的工程化改造

首先， 由于在实际使用中， 我们往往需要支持删除操作， 所以我们应当对每个向量生成一个唯一的哈希值并构建哈希表， 这在 5.2 节中已经提到。

除此之外， 由于 HNSW 算法需要花费大量构造时间， 所以我们期望在一次构造

以后就尽量减少重新构造。于是在算法启动并成功构造以后， 我将程序中的所有信息存放至一个临时文件中，在下次启动时， 可以直接从文件中读取数据， 这样大大减少了重复的构造时间。

5.4 系统流程概述与软件的设计开发

软件前端以及与系统交互的部分由李嘉明同学完成， 由于查重率的要求， 该部分的详细介绍见李嘉明同学的课程报告。除此以外， 也可见 Github 仓库：https://github.com/FutureHasTomorrow/Vectri/tree/main

6　软件功能展示与对比

我们开发的软件名为 Vectri，以下展示软件的使用流程，并简单描述该软件的优势。

同样的， 由于这部分内容在李嘉明同学的报告中有所展示， 此处只进行简单的叙述， 具体可见李嘉明同学的课程报告。

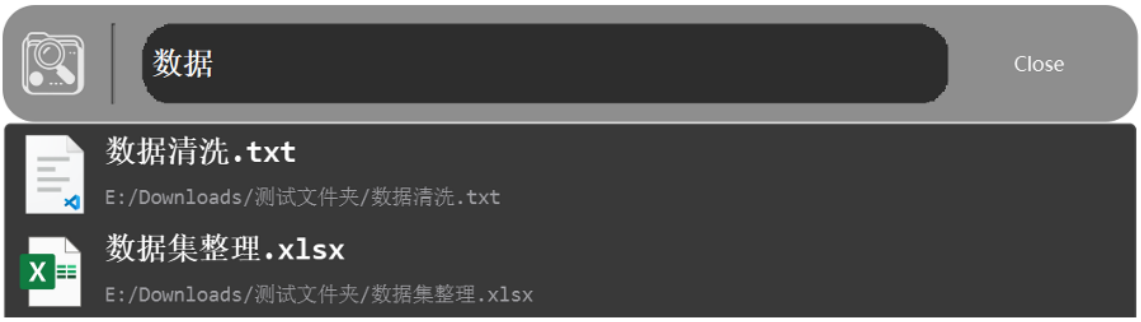
6.1 软件使用流程

6.1.1 选择文件夹并扫描

在软件启动后，用户首先选择需要进行搜索的文件夹。通过点击界面上的“选择文件夹”按钮，用户可以浏览并选择目标文件夹。一旦文件夹被选中，系统会自动扫描该文件夹中的所有文件。文件的基本信息（如文件名、文件路径、文件图标）会被提取并保存。

6.1.2 输入查询并进行搜索

当用户在搜索框中输入查询字符串时，系统将自动将查询词转化为向量，并与已存储的文件信息向量进行比较。系统会利用 HNSW 算法高效地找到与查询词最相似的文件，并按照相似度排序展示在搜索结果中。



图表 8 软件使用示意图

6.1.3 查看与打开文件

用户可以通过点击搜索结果中的文件名或按下回车键来打开文件。系统将快速响应，并自动打开对应的文件。

6.1.4 关闭并再次使用

在使用过程中，用户可以点击按钮关闭主窗口，软件会最小化到系统托盘中继续运行。

6.2 软件优势

在 Everything 等软件中， 利用传统的字符串匹配方法仅能搜索到完全一致的文件名；但在我们的方法中， 利用利用字符级卷积神经网络，能够捕捉到一些语义级的信息， 从而实现中文简写匹配、英文错拼匹配、连词匹配、中文拼音匹配、中英文混合错序匹配等等更加强大的匹配。这使得我们**无需输入完全精确的文件名**也能高效获取文件位置及信息， 这在我们不完全记得文件名称的时候是极其有效的。

7　总结与思考

7.1 总结

本报告系统地探讨了近似最近邻搜索（Approximate Nearest Neighbor Search, ANNS）问题及其在高维数据检索中的应用，重点介绍了 HNSW（Hierarchical Navigable Small World）算法的理论基础、实现细节及其性能表现。通过对 HNSW 算法的深入分析与实验测试，我们验证了其在大规模高维数据集上的高效性与准确性。

同时，报告还展示了基于 HNSW 算法开发的本地文件检索软件 Vectri，说明了其在实际工程应用中的可行性与优势。

7.2 主要贡献

ANNS 问题的深入理解：通过对 ANNS 问题的定义及其重要性的阐述，明确了高效数据检索在现代信息系统中的关键地位。分析了国内外在 ANNS 领域的发展现状，指出了现有方法的优缺点。

提供了 HNSW 算法的 C++ 实现：在网上可以找到很多关于 HNSW 算法的介绍，但无一提供了具体的实现， 仅仅只有伪代码或者是寥寥几行跑不通的 python 代码。而著名的 hnswlib 库的实现， 由于注重性能， 可读性并不好， 对初学者并不友好。本人使用 C++ 简单实现了原论文的方法并进行了一些测试， 代码正确性有保证且可读性强。目前代码已经在 Github 上**开源**供大家参考：https://github.com/lightfom/ann

设计测试方法对 HNSW 算法进行测试与评估：通过设计测试和朴素算法进行比对， 验证了“维度灾难”在 HNSW 算法上的体现，同时通过对结果的分析认识到了一些处理手法的必要性。例如， 由于向量总数的增大会显著降低正确率， 在分布式系统上分别运行该算法是有必要的。

给出可支持在线删除的 HNSW 算法：由于在实际运用中， HNSW 算法往往需要面临支持删除操作的局面， 但原论文并没有给出相关的想法， 在网络上也难以找到 HNSW 支持删除的改造， 实现基本较复杂。在本报告中， 我在 5.2 节中给出了 HNSW 算法支持删除的简单实现， 以应对 Vectri 软件添加和删除文件的需求。具体代码参见附加文件的 hnsw.h。

7.3 思考与展望

尽管 HNSW 算法在 ANNS 领域表现出色，但其在构建阶段的时间复杂度和内存消耗仍是亟待优化的问题。

可以观察到的是，为了平衡时间和内存的消耗，与其余算法结合是很有必要的。并且，除了与其余算法结合以外，可以观察到 HNSW 算法只是基于“跳表”的想法来实现小范围的估计，这个想法看起来技术含量并不高，或许可以构造更加精细的“逼近”手法来加速这个算法，我认为这是一个合理的研究方向。

除此以外，还可以注意到 HNSW 算法的结构并不复杂，并且每部分的处理手法都是经典的。所以如果我们给问题加上一定的限制，HNSW 算法的很多处理部分可以利用传统算法技术进行较大的优化，例如采用线段树一类的结构或者利用 2 的幂次进行分组来优化最近邻的查询应当都是可能奏效的。

总的来说，HNSW 算法作为一种高效的近似最近邻搜索方法，展示了其在处理大规模高维数据中的卓越性能。通过本报告的研究与实践，验证了其在实际应用中的可行性与优势。未来，随着算法的不断优化和技术的进步，HNSW 及其衍生方法将在更多领域发挥重要作用，为信息检索和数据分析提供强有力的支持。

8　参考文献

【1】 [1603.09320] Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

【2】 Approximate nearest neighbor algorithm based on navigable small world graphs - ScienceDirect

【3】 nmslib/hnswlib: Header-only C++/python library for fast approximate nearest neighbors

9 附录

9.1 hnsw.h

```
#include <bits/stdc++.h>

namespace Hnsw {

#define debug(...) fprintf(stderr, __VA_ARGS__), fflush(stderr)

using i64 = long long;
using std ::cin;
using std ::cout;
using std ::pair;
using std ::vector;
using std ::mt19937;
using std ::string;

const int efConstruction = 30;
const double alpha = 0.3;

template <typename ValueType = int>
class hnsw {
  private:
    using T = ValueType;
    using ull = unsigned long long;
    using IT = typename std ::vector<T> ::iterator;
    const int M = 200;
    const int Mmax = M;
    const int Mmax0 = M * 2;
```

```
27.     const double Ml = 1 / log(M);
28.     int enter_point;
29.     int tot;
30.     std ::mt19937 rng;
31.
32.     struct Layer {
33.         vector<vector<int>> e;
34.         void test(int x) {
35.             if (e.size() <= x)
36.                 e.resize((x + 1));
37.         }
38.     };
39.     vector<Layer> layer;
40.     vector<pair<vector<T>, ull> > VectorPool;
41.     std :: unordered_map<ull, int> id2pos;
42.     std :: set<ull> ErasedCookie;
43.
44.     inline int GetLayer() {
45.         double r = -log((double)rng() / rng.max());
46.         assert(Ml * r >= 0);
47.         return (int)(Ml * r);
48.     }
49.     inline long long dist(const vector<T> &a, const vector<T> &b) {
50.         auto Ap = a.data(), Bp = b.data(); int len = a.size();
51.         long long res = 0;
52.         //#pragma GCC unroll 16
53.         for (int i = 0; i < len; i ++)
54.             res += 1LL * (Ap[i] - Bp[i]) * (Ap[i] - Bp[i]);
55.         return res;
```

```
56.     }
57.
58.     inline bool IsErased(ull cookie) {
59.         return ErasedCookie.find(cookie) != ErasedCookie.end();
60.     }
61.
62.     inline void EraseIt(ull cookie) {
63.         ErasedCookie.insert(cookie);
64.     }
65.
66.     vector<int> search_layer(const vector<T> &QueryVector, int EnterPoint,
int ef, int LayerIndex) {
67.         std ::unordered_map<int, int> Visited;
68.         vector<pair<double, int>> Candidator;
69.         std ::priority_queue<pair<double, int>> NearestNeighbors;
70.         Visited[EnterPoint] = 1;
71.         double init_len = dist(QueryVector, VectorPool[EnterPoint].first);
72.         NearestNeighbors.push({init_len, EnterPoint});
73.         Candidator.push_back({init_len, EnterPoint});
74.
75.         while (Candidator.size() > 0) {
76.             auto cit = std ::min_element(Candidator.begin(),
Candidator.end());
77.             int c = cit->second;
78.             Candidator.erase(cit);
79.             int f = NearestNeighbors.top().second;
80.             if (dist(QueryVector, VectorPool[c].first) > dist(QueryVector,
VectorPool[f].first)) {
81.                 break;
```

```
            }

            layer[LayerIndex].test(c);
            for (int e : layer[LayerIndex].e[c]) {
                if (Visited[e] == 0) {
                    Visited[e] = 1;
                    f = NearestNeighbors.top().second;
                    double d = dist(QueryVector, VectorPool[e].first);
                    if (d < dist(QueryVector, VectorPool[f].first) ||
NearestNeighbors.size() < ef) {
                        Candidator.push_back({d, e});
                        NearestNeighbors.push({d, e});
                        if (NearestNeighbors.size() > ef) {
                            NearestNeighbors.pop();
                        }
                    }
                }
            }
        }
        vector<int> res;
        while (NearestNeighbors.size()) {
            res.push_back(NearestNeighbors.top().second);
            NearestNeighbors.pop();
        }
        reverse(res.begin(), res.end());
        return res;
    }

```

```
109.     vector<int> select_neighbors_simple(const vector<T> &QueryVector,
vector<int> &Candidator, int m) {
110.         if (Candidator.size() <= m) {
111.             return Candidator;
112.         }
113.         vector<pair<double, int>> D;
114.
115.         for (int x : Candidator) {
116.             D.push_back({dist(QueryVector, VectorPool[x].first), x});
117.         }
118.
119.         std ::nth_element(D.begin(), D.begin() + m, D.end());
120.         vector<int> res;
121.
122.         for (int i = 0; i < m; i++) {
123.             res.push_back(D[i].second);
124.         }
125.
126.         return res; // 返回可能小于 m 个数 ！！！
127.     }
128.
129.     vector<int> select_neighbors_heuristic(vector<T> QueryVector,
vector<int> &Candidator, int m,
130.                                            int LayerIndex, int extendCandidates,
131.                                            int keepPrunedConnections) {
132.         vector<int> R;
133.         std ::set<int> NearestNeighbors;
134.         for (int x : Candidator) {
135.             NearestNeighbors.insert(x);
```

```
136.         }
137.         if (extendCandidates) {
138.             for (int x : Candidator) {
139.                 layer[LayerIndex].test(x);
140.                 for (int y : layer[LayerIndex].e[x]) {
141.                     if (NearestNeighbors.find(y) == NearestNeighbors.end()) {
142.                         NearestNeighbors.insert(y);
143.                     }
144.                 }
145.             }
146.         }
147.         std ::set<int> W0;
148.         while (NearestNeighbors.size() > 0 && R.size() < m) {
149.             int e = *NearestNeighbors.begin();
150.             for (int x : NearestNeighbors)
151.                 if (x != e) {
152.                     double d = dist(VectorPool[x].first, QueryVector);
153.                     if (d < dist(VectorPool[e].first, QueryVector)) {
154.                         e = x;
155.                     }
156.                 }
157.             NearestNeighbors.erase(e);
158.
159.             bool small_all = 0;
160.             for (int x : R) {
161.                 if (dist(VectorPool[x].first, QueryVector) >
dist(VectorPool[e].first, QueryVector)) {
162.                     small_all = 1;
163.                     break;
```

```
164.                 }
165.             }
166.
167.             if (small_all) {
168.                 R.push_back(e);
169.             } else {
170.                 W0.insert(e);
171.             }
172.         }
173.
174.         if (keepPrunedConnections) {
175.             while (W0.size() > 0 && R.size() < m) {
176.                 int e = *W0.begin();
177.                 for (int x : W0)
178.                     if (x != e) {
179.                         double d = dist(VectorPool[x].first, QueryVector);
180.                         if (d < dist(VectorPool[e].first, QueryVector)) {
181.                             e = x;
182.                         }
183.                     }
184.                 W0.erase(e);
185.                 R.push_back(e);
186.             }
187.         }
188.
189.         return R;
190.     }
191.
192.     bool CheckRebuild() {
```

```
193.         if (ErasedCookie.size() > alpha * VectorPool.size()) {
194.             return 1;
195.         }
196.         return 0;
197.     }
198.
199.     void rebuild() {
200.         debug("rebuilding !\n");
201.         layer.clear();
202.         id2pos.clear();
203.         auto rec = VectorPool;
204.         VectorPool.clear();
205.         enter_point = 0;
206.         layer.resize(1);
207.
208.         for (auto &x : rec) {
209.             if (! IsErased(x.second)) {
210.                 insert(x.first, x.second);
211.             }
212.         }
213.
214.         ErasedCookie.clear();
215.         debug("rebuild finished !\n");
216.     }
217.
218.     vector<vector<T>> k_nn_search(const vector<T> &QueryVector, int K, int
ef = efConstruction) {
219.         vector<vector<T>> res;
220.         int nowcnt = K;
```

```
221.         if (VectorPool.size() < K) {
222.             for (auto &x : VectorPool) {
223.                 if (! IsErased(x.second)) res.push_back(x.first);
224.             }
225.             return res;
226.         }
227.         while (1) {
228.             auto res2 = k_nn_search_no_erase(QueryVector, nowcnt, ef);
229.
230.             int cnt = 0;
231.             for (auto x : res2) {
232.                 if (! IsErased(VectorPool[x].second)) {
233.                     res.push_back(x);
234.                     cnt++;
235.                 }
236.                 if (cnt == K) {
237.                     break;
238.                 }
239.             }
240.
241.             if (cnt == K) {
242.                 break;
243.             }
244.
245.             nowcnt += K;
246.         }
247.         return res;
248.     }
249.
```

```
250.     vector<vector<T>> k_nn_search_no_erase(const vector<T> &QueryVector, int
K, int ef = efConstruction) {
251.         vector<int> NearestNeighbors;
252.         int EnterPoint = 0;
253.         for (int i = layer.size() - 1; i >= 1; i--) {
254.             NearestNeighbors = search_layer(QueryVector, EnterPoint, ef, i);
255.             EnterPoint = NearestNeighbors[0];
256.         }
257.         NearestNeighbors = search_layer(QueryVector, EnterPoint, ef, 0);
258.         NearestNeighbors.resize(K);
259.         vector<vector<T>> res;
260.         for (int x : NearestNeighbors) {
261.             res.push_back(VectorPool[x].first);
262.         }
263.         return res;
264.     }
265.
266.     vector<ull> k_nn_search_cookie_no_erase(const vector<T> &QueryVector,
int K, int ef = efConstruction) {
267.         vector<int> NearestNeighbors;
268.         int EnterPoint = 0;
269.         for (int i = layer.size() - 1; i >= 1; i--) {
270.             NearestNeighbors = search_layer(QueryVector, EnterPoint, ef, i);
271.             EnterPoint = NearestNeighbors[0];
272.         }
273.         NearestNeighbors = search_layer(QueryVector, EnterPoint, ef, 0);
274.         NearestNeighbors.resize(K);
275.         vector<ull> res;
276.         for (int x : NearestNeighbors) {
```

```
277.             res.push_back(VectorPool[x].second);
278.         }
279.         return res;
280.     }
281.
282.     vector<double> get_min_kth_dist_no_erase(const vector<T> &QueryVector,
int K) {
283.         auto res = k_nn_search_cookie(QueryVector, K, efConstruction);
284.         vector<double> res2;
285.         for (auto x : res) {
286.             res2.push_back(dist(QueryVector, VectorPool[id2pos[x]].first));
287.         }
288.         return res2;
289.     }
290.
291.   public:
292.
293.     //-------------------------------------------------------------------
----------
294.
295.
296.     hnsw() {
297.         tot = 0;
298.         rng.seed(
299.             std ::chrono ::steady_clock ::now().time_since_epoch().count());
300.         enter_point = 0;
301.         layer.resize(1);
302.     }
303.
```

```
304.     hnsw(int len) {
305.         tot = 0;
306.         rng.seed(
307.             std ::chrono ::steady_clock ::now().time_since_epoch().count());
308.         enter_point = 0;
309.         layer.resize(1);
310.         VectorPool.reserve(len);
311.     }
312.
313.     ~hnsw() {
314.         //save_data();
315.     }
316.
317.     bool erase(const ull id) {
318.
319.         if (IsErased(id)) {
320.             return 0;
321.         }
322.         ErasedCookie.insert(id);
323.         if (CheckRebuild()) {
324.             rebuild();
325.         }
326.         return true;
327.     }
328.
329.     vector<ull> k_nn_search_cookie(const vector<T> &QueryVector, int K, int
ef = efConstruction) {
330.         vector<ull> res;
331.         if (VectorPool.size() < K) {
```

```
332.             for (auto &x : VectorPool) {
333.                 if (! IsErased(x.second)) res.push_back(x.second);
334.             }
335.             return res;
336.         }
337.         int nowcnt = K;
338.         while (1) {
339.             auto res2 = k_nn_search_cookie_no_erase(QueryVector, nowcnt, ef);
340.
341.             int cnt = 0;
342.             for (auto x : res2) {
343.                 if (! IsErased(x)) {
344.                     res.push_back(x);
345.                     cnt++;
346.                 }
347.                 if (cnt == K) {
348.                     break;
349.                 }
350.             }
351.
352.             if (cnt == K) {
353.                 res = std ::move(res2);
354.                 break;
355.             }
356.
357.             nowcnt += K;
358.         }
359.         return res;
360.     }
```

```
361.
362.
363.
364.     vector<double> get_min_kth_dist(const vector<T> &QueryVector, int K) {
365.         int nowcnt = K;
366.         while (1) {
367.             auto res = k_nn_search_cookie(QueryVector, K, efConstruction);
368.             int cnt = 0;
369.             vector<double> res2;
370.             for (auto x : res) {
371.                 if (! IsErased(x)) {
372.                     res2.push_back(dist(QueryVector,
VectorPool[id2pos[x]].first));
373.                     cnt++;
374.                 }
375.                 if (cnt == K) {
376.                     break;
377.                 }
378.             }
379.             if (cnt == K) {
380.                 return res2;
381.             }
382.             nowcnt += K;
383.
384.         }
385.         assert(0);
386.         return {};
387.     }
388.
```

```
389.     void save_data(string FILE_NAME) {
390.         debug("Saving Data...\n");
391.         std ::ofstream out(FILE_NAME);
392.         out << VectorPool.size() << " " << VectorPool[0].first.size() <<
"\n";
393.         for (auto &x : VectorPool) {
394.             for (auto y : x.first) {
395.                 out << y << " ";
396.             }
397.             out << x.second << "\n";
398.         }
399.         out << ErasedCookie.size() << "\n";
400.         for (auto x : ErasedCookie) {
401.             out << x << "\n";
402.         }
403.         out << layer.size() << "\n";
404.         for (auto &x : layer) {
405.             out << x.e.size() << "\n";
406.             for (auto &y : x.e) {
407.                 out << y.size() << "\n";
408.                 for (auto z : y) {
409.                     out << z << " ";
410.                 }
411.                 out << "\n";
412.             }
413.         }
414.         out << enter_point << " " << tot << "\n";
415.         out << id2pos.size() << "\n";
416.         for (auto &x : id2pos) {
```

```
417.             out << x.first << " " << x.second << "\n";
418.         }
419.
420.         out.close();
421.
422.         debug("Data Saved!\n");
423.         return void();
424.     }
425.
426.     void read_data(string FILE_NAME) {
427.         debug("Reading Data...\n");
428.         std ::ifstream in(FILE_NAME);
429.         int n, dim;
430.         in >> n >> dim;
431.         VectorPool.resize(n);
432.         for (int i = 0; i < n; i++) {
433.             VectorPool[i].first.resize(dim);
434.             for (int j = 0; j < dim; j++) {
435.                 in >> VectorPool[i].first[j];
436.             }
437.             in >> VectorPool[i].second;
438.             id2pos[VectorPool[i].second] = i;
439.         }
440.         int m;
441.         in >> m;
442.         for (int i = 0; i < m; i++) {
443.             ull x;
444.             in >> x;
445.             ErasedCookie.insert(x);
```

```
446.         }
447.         int l;
448.         in >> l;
449.         layer.resize(l);
450.         for (int i = 0; i < l; i++) {
451.             int s;
452.             in >> s;
453.             layer[i].e.resize(s);
454.             for (int j = 0; j < s; j++) {
455.                 int t;
456.                 in >> t;
457.                 layer[i].e[j].resize(t);
458.                 for (int k = 0; k < t; k++) {
459.                     in >> layer[i].e[j][k];
460.                 }
461.             }
462.         }
463.         in >> enter_point >> tot;
464.         int sz;
465.         in >> sz;
466.         for (int i = 0; i < sz; i++) {
467.             ull x;
468.             int y;
469.             in >> x >> y;
470.             id2pos[x] = y;
471.         }
472.
473.         in.close();
474.         debug("Data Read!\n");
```

```
475.     }
476.
477.     ull insert(const vector<T> &QueryVector, ull Cookie = 0) {  // cookie = 0
表示 cookie 未知
478.         ull cookie;
479.         if (! Cookie)
480.             cookie = ++ tot;
481.         else
482.             cookie = Cookie;
483.         VectorPool.push_back( {QueryVector, cookie} );
484.
485.         id2pos[cookie] = VectorPool.size() - 1;
486.
487.         if (VectorPool.size() == 1) {
488.
489.             return cookie;
490.         }
491.
492.         vector<int> NearestNeighbors;
493.         int EnterPoint = enter_point;
494.         int L = layer.size() - 1;
495.         int l = GetLayer();
496.         if (l > L) {
497.             layer.resize(l + 1);
498.         }
499.
500.         for (int LayerIndex = L; LayerIndex >= l + 1; LayerIndex--) {
501.             NearestNeighbors = search_layer(QueryVector, EnterPoint,
efConstruction, LayerIndex);
```

```
502.             EnterPoint = NearestNeighbors[0];
503.         }
504.
505.         for (int LayerIndex = std ::min(L, l); LayerIndex >= 0; LayerIndex--)
{
506.             NearestNeighbors = search_layer(QueryVector, EnterPoint,
efConstruction, LayerIndex);
507.
508.             vector<int> neighbors =
509.                 std ::move(select_neighbors_simple(QueryVector,
NearestNeighbors, Mmax));
510.
511.             for (int x : neighbors) {
512.                 layer[LayerIndex].test(x);
513.                 layer[LayerIndex].test(VectorPool.size() - 1);
514.                 layer[LayerIndex].e[x].push_back(VectorPool.size() - 1);
515.                 layer[LayerIndex].e[VectorPool.size() - 1].push_back(x);
516.             }
517.
518.             for (int e : neighbors) {
519.
520.                 layer[LayerIndex].test(e);
521.                 auto &eConn = layer[LayerIndex].e[e];
522.                 if (eConn.size() > Mmax) {
523.                     eConn =
524.
std ::move(select_neighbors_simple(VectorPool[e].first, eConn, Mmax));
525.                 }
526.             }
```

```
527.             EnterPoint = NearestNeighbors[0];
528.         }
529.         if (l > L) {
530.             enter_point = VectorPool.size() - 1;
531.         }
532.
533.         return cookie;
534.     }
535. };
536.
537. } // namespace Hnsw
538.
539. using Hnsw :: hnsw;
540.
```

9.2 brute.h

```
1. #include <bits/stdc++.h>
2. namespace BruteForce {
3.     #define debug(...) fprintf(stderr, __VA_ARGS__), fflush(stderr)
4.
5.     using i64 = long long;
6.     using namespace std;
7.
8.     class brute {
9.     public:
10.         vector<vector<int> > A;
11.         void insert(vector<int> q) {
12.             A.push_back(q);
```

```
13.         }
14.         vector<i64> get_min_kth_dist(vector<int> q, int K) {
15.             vector<pair<i64, int> > res;
16.
17.             for (int i = 0; i < (int) A.size(); i ++) {
18.                 i64 dist = 0;
19.                 for (int j = 0; j < (int) q.size(); j ++) {
20.                     dist += 1LL * (q[j] - A[i][j]) * (q[j] - A[i][j]);
21.                 }
22.                 res.push_back({dist, i});
23.             }
24.             sort(res.begin(), res.end());
25.             vector<i64> res2;
26.             for (int i = 0; i < K; i ++) {
27.                 res2.push_back(res[i].first);
28.             }
29.             return res2;
30.         }
31.     } ;
32. }
33.
34. using BruteForce :: brute;
```

9.3 tester.cpp

```
1. #include <bits/stdc++.h>
2. #include "../hnsw/hnsw.h"
3. #include "../brute/brute.h"
4.
```

```
#define debug(...) fprintf(stderr, __VA_ARGS__), fflush(stderr)

const double divclk = CLOCKS_PER_SEC;

int main() {
    std :: ios :: sync_with_stdio(0);
    std :: cin.tie(0); std :: cout.tie(0);

    hnsw<int> hnsw_sol;
    brute brute_sol;
    int n, dim;
    std :: cin >> n >> dim;
    const std :: string FILE_NAME = "data.txt";

    clock_t start, end;
    clock_t bf_building_time = 0, hnsw_building_time = 0;

    for (int i = 0; i < n; i ++) {
        std :: vector<int> a(dim);
        for (int j = 0; j < dim; j ++) std :: cin >> a[j];

        start = clock(); hnsw_sol.insert(a); end = clock();
        hnsw_building_time += end - start;

        start = clock(); brute_sol.insert(a); end = clock();
        bf_building_time += end - start;

    }

```

```
34.     debug ("Finished Building! \n HNSW Time: %lf\n Brute Time: %lf\n", 1.0 *
hnsw_building_time / CLOCKS_PER_SEC, 1.0 * bf_building_time / CLOCKS_PER_SEC);
35.     std :: cout << n <<  ' ';
36.     std :: cout << hnsw_building_time / divclk << ' ' << bf_building_time /
divclk << ' ';
37.     int q;
38.     std :: cin >> q;
39.     double score = 0, per = 100.0 / q;
40.     double query_time = 0;
41.
42.     debug("Start Query!\n");
43.
44.     clock_t hnsw_query_time = 0, bf_query_time = 0;
45.
46.     for (int i = 0; i < q; i ++) {
47.         int k;
48.         std :: vector<int> a(dim);
49.         std :: cin >> k;
50.         for (int j = 0; j < dim; j ++) std :: cin >> a[j];
51.         start = clock();
52.         auto res = hnsw_sol.get_min_kth_dist(a, k);
53.         end = clock();
54.         hnsw_query_time += end - start;
55.
56.         start = clock();
57.         auto res2 = brute_sol.get_min_kth_dist(a, k);
58.         end = clock();
59.         bf_query_time += end - start;
60.
```

```
61.         // differ
62.         assert(res.size() == res2.size());
63.         assert(res.size() == k);
64.
65.         double pnt = per / k;
66.         for (auto x : res2) {
67.             bool ok = 0;
68.             for (auto y : res) {
69.                 if (x == (long long) y) {
70.                     ok = 1;
71.                     break;
72.                 }
73.             }
74.             if (ok) {
75.                 score += pnt;
76.             }
77.         }
78.
79.         if (i % 1000 == 0) {
80.             debug("Progress: %.2lf%%\n", i * per);
81.             debug ("HNSW Time: %lf\n Brute Time: %lf\n", 1.0 * hnsw_query_time
/ CLOCKS_PER_SEC, 1.0 * bf_query_time / CLOCKS_PER_SEC);
82.         }
83.
84.     }
85.
86.     debug ("Finished Query! \n HNSW Time: %lf\n Brute Time: %lf\n", 1.0 *
hnsw_query_time / CLOCKS_PER_SEC, 1.0 * bf_query_time / CLOCKS_PER_SEC);
87.
```

```
88.     std :: cout << hnsw_query_time / divclk<< ' ' << bf_query_time/ divclk <<
std :: endl;
89.     return 0;
90. }
91.
```

9.4 generator.cpp

```
 1. #include <bits/stdc++.h>
 2.
 3. using namespace std;
 4.
 5. mt19937 myrand(114514);
 6. #define endl '\n'
 7.
 8. const int dim = 5; // 维度
 9. const int N = 5000; // 向量数
10. const int MaxVal = 1e6;
11. const int Max_Testcase = 500;
12. const int Max_K = 20;
13.
14. int main() {
15.     ios :: sync_with_stdio(0);
16.     cin.tie(0); cout.tie(0);
17.     cout << N << " " << dim << endl;
18.     for (int i = 0; i < N; i++) {
19.         for (int j = 0; j < dim; j++) {
20.             cout << myrand() % MaxVal << " ";
21.         }
```

```
        cout << endl;
    }
    cout << Max_Testcase << endl;
    for (int i = 0; i < Max_Testcase; i++) {
        int k = myrand() % Max_K + 1;
        cout << k << endl;
        for (int j = 0; j < dim; j++) {
            cout << myrand() % MaxVal << " ";
        }
        cout << endl;
    }
    return 0;
}
```

9.5 experimentor.cpp

```
#include <bits/stdc++.h>

using namespace std;

mt19937 myrand(114514);
#define endl '\n'

namespace Generator {

const int dim = 10; // 维度
int N = 10000; // 向量数
const int MaxVal = 1e6;
const int Max_Testcase = 1000;
```

```
14. const int Max_K = 10;
15.
16. int main() {
17.     freopen("input.txt", "w", stdout);
18.     cout << N << " " << dim << endl;
19.     for (int i = 0; i < N; i++) {
20.         for (int j = 0; j < dim; j++) {
21.             cout << myrand() % MaxVal << " ";
22.         }
23.         cout << endl;
24.     }
25.     cout << Max_Testcase << endl;
26.     for (int i = 0; i < Max_Testcase; i++) {
27.         int k = myrand() % Max_K + 1;
28.         cout << k << endl;
29.         for (int j = 0; j < dim; j++) {
30.             cout << myrand() % MaxVal << " ";
31.         }
32.         cout << endl;
33.     }
34.     fclose(stdout);
35.     return 0;
36. }
37.
38. }
39.
40. const int para[] = {100, 200, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000,
4500, 5000, 5500, 6000, 6500, 7000, 7500, 8000, 8500, 9000, 9500, 10000, 12000,
15000, 20000, 25000};
```

```cpp
const int testcase = sizeof(para) / sizeof(para[0]);

int main() {
    system("g++ -o tester tester.cpp -Ofast");
    for (int i = 0; i < 10; i++) {
        Generator :: N = para[i];
        Generator :: main();
        cerr << "Testcase " << i << " finished!" << endl;
        system("./tester < input.txt >> output.txt");
    }
    return 0;
}
```