search on graph 系列算法笔记

日期：June 25, 2020

目录

今天阅读一篇需要复现的论文, 发现自己对"search on graph" 一直没有足够的了解. 因此专门抽出一天时间学习这一类解决 ANNS 问题的方法, 算是为以后的工作铺平道路吧.

1 search on graph 问题的两个步骤

一般而言, 所有的 search on graph 算法都包含两个基本的环节:graph construction 与 graph search. 正如 LSH 等算法需要建设一个向量点的 sketch 一样,search on graph 的算法侧重于捕捉节点与节点之间的连接关系, 即通过一张" 图" 去存储点集的预信息, 而非存储哈希表. 因此, 这两个环节的基本功能是:

1. 图的构建. 对于一个点集合, 通过增量式的点添加去生成一个张 Delaunay Graph.
2. 图的搜索. 对于一个被建了图的点集合, 每个查询 q 其本身可以通过一种贪婪的迭代算法进行快速地 K 个最近邻选取.

为了澄清上述描述中的一些问题, 需要对 Delaunay graph 进行一个简单的说明.

1.1 Delaunay graph 与 voronoi Diagram

你可以在bilibili-delaunay 三角找到对 delaunay 划分的一些直观理解. 当然, 这个视频里仅仅给出了二维空间里点集合的基本形态, 因而会通过" 三角形" 实现最少连接同时最近连接.

直观上讲,Delaunay graph 是一种特殊的图结构. 对于搜索来说, 这种图结构的特殊性在于:

1. 图是连通的. 也就是说, 任意两点之间都存在一条通路;
2. 图中每个节点的最近邻们为其一阶邻居节点. 这个性质大大有益于最近邻搜索.
3. 图中每个节点的度被严格控制. 这样就保证了搜索的效率.

当然, 如果从数学的角度看, 还有其他性质:

1. 不相交性. 得劳内图中边与边只在节点处相交, 这是由于它总是将最近的点连接造成的.
2. 唯一性. 只要满足得劳内图的要求 (后面会提到), 那么对于一个固定的点集将产生唯一的 Delaunay graph
3. 最优性. 点与点之间的边连接永远都是最优的.
4. 规则性. 平缓性. 得劳内图中, 在每个节点处, 边与边构成的夹角, 总体来看其方差是最小的, 相比较于其他的图连接生成方法.
5. 区域性. 增, 删, 移动节点仅仅会对周围节点的连接关系产生影响.
6. 凸外壳. 外面是一个凸包.

那么, 这样的一种得劳内图, 它的定义是什么呢?

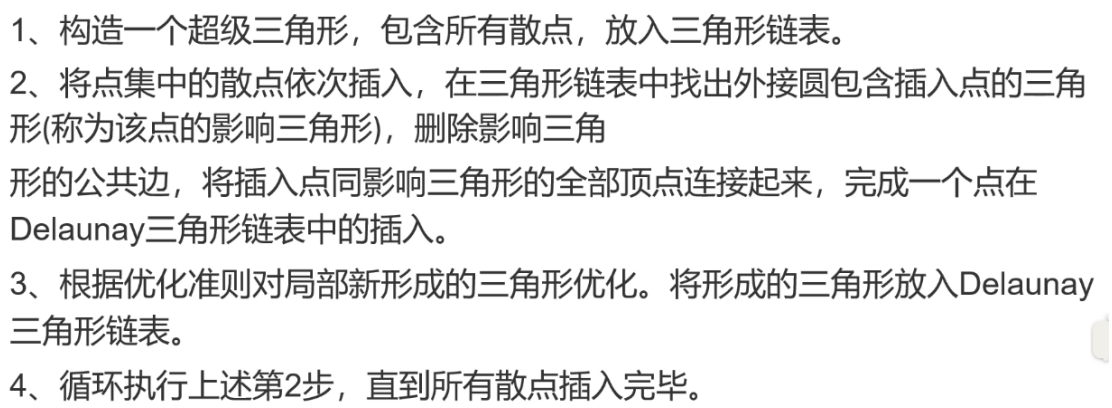
我们大可以把这些性质看作某类定义, 不过他们太过冗余了. 二维平面的得劳三角化生成的图完全可以通过所谓" 空圆特性" 进行表达.

引理: 三点可以唯一确定一个圆.

如果一张图完全由三角形构成, 并且里面每个三角形的外接圆内都不存在该图中的任何节点, 那么这张图被称为是这些节点的一个得劳内三角化了的图.

一般来说, 当构建完成一个二维的 delaunay 图之后, 随机落入的一个 query 点, 它的最近邻就是它所在的三角形的顶点.

建立 delaunay 图比较严格的方法是 Bowyer-Watson 算法. 该算法可以通过下图的计算步骤表示.



关于三角链表的数据结构问题, 可以在本文开头给出的视频中获得详细的说明. 由于这些都不是本文的重点, 因此不给予特殊的关注.

总之,delaunay 图是一种很严格同时很复杂的东西, 现实中一般都会采用随机采样的方式去规避这种复杂性, 同时放宽这种严格的风气. 这类算法才是需要重点关注的东西.

1.2 图的贪婪搜索与度量函数的凸特性

2 NSW Navigation Small World

那么, 怎么去做这件事呢?! 常见的一个思路是,NSW. 在这一章, 将从直观上先说一说 NSW 的基本思路, 而后再给出该算法详细的伪代码.

基本假设:Small world. small word 是进行社交网络研究时发现的一个有趣现象. 其大致意思是: 自然界社会界很多的图都是聚类性质 + 类与类之间连接性质的一张庞大的连通图. 这种现象来自于一个实验, 该实验是

从广东的一些人开始, 这些人通过社交关系的连接, 总能找到一条通往黑龙江地区的人的社交关系链条. 这种无论如何都能找到通路的特性, 给与了基于 SW 的各类算法一种灵感. 也让随机找到一个进入点 (enter point or entry point) 的思路变得可行.

NSW 基本想法: 在进行 delaunay graph construction 时, 对于需要插入的点 q, 随机选择一个入口 c, 然后对比入口以及其邻居节点到插入点 q 的距离, 从而沿着贪婪的" 最小距离" 形成一条通路, 最后得到结果. 这个" 距离", 便起到了所谓"navigation(导航)" 的作用.

NSW 算法自然语言描述:

1. 首先, 初始化三个空的集合, 分别是 candidate, visitedset, 以及 results.
   - candidate. 候选的点构成的集合. 候选的点就是需要考察和插入点"q" 的距离以及其邻居性质, 从而确定是否可以进入 results 的哪些点.
   - visited set. 被计算过的点构成的集合. 顾名思义, 如果一个点已经被算过距离了, 那么它就会落到这里面. 这个集合是为了节点间的邻居节点的重合现象定义的.
   - results. 这个集合里面存储着对于插入点 q 的 x 个最近邻. 一般, 对于 K-ANNS 问题,x>k.
2. 随机在当前的图上选取一个节点作为 entry point. 将其加入 candidate 中, 加入 visited set 中.
3. 遍历所有的 candidates, 计算该集合内所有节点与插入点 q 的距离, 并找到具有最小的距离的点 c.
4. 对于 c, 进行如下判定: 如果 d(c,q) 比 results 中的点和 q 最大的距离还要大, 那么就停止算法, 输出 results.[这是因为本算法是完全贪婪的, 因而只要算法的效果开始变差, 就意味着寻找的结束. 当然,candidates 为空时, 也是表示算法结束] 否则, 继续.
5. 从 candidate 中删除点 c, 将 c 加入到 results 里面.
6. 对于 c 的所有邻居节点, 将其并入集合 visitedset 里面. 计算 c 的所有邻居节点 e 与 q 的距离, 对于距离 q 比 results 中的最远距离要近的节点, 将之加入到 candidates,results 里面.

这个描述或与真实的 NSW 算法有所不同, 不同主要主要体现在 5 中是否将 c 加入到 results 里面. 在第一次循环时, 我觉得将之加入是必要的.(欢迎讨论)

这部分伪代码可以表示为: (感谢这个笔记)

```
K-NNSearch(object q, integer: m, k)
TreeSet [object] tempRes, candidates, visitedSet, result
// 进行m次循环, 避免随机性
for (i 0; i < m; i++) do:
    put random entry point in candidates
    tempRes null
    repeat:
  // 利用上述提到的贪婪搜索算法找到距离q最近的点c
  get element c closest from candidates to q
  remove c from candidates
  // 判断结束条件
  if c is further than k-th element from result then
      break repeat
  // 更新后选择列表
  for every element e from friends of c do:
      if e is not in visitedSet then
    add e to visitedSet, candidates, tempRes
    end repeat
    // 汇总结果
    add objects from tempRes to result
```

```
end for
return best k elements from result
```

可以看出, 为了减弱采样的随机性带来的影响, 算法中进行了多次重复.

3 HNSW Hierarchical NSW

学长就让我看到 HNSW. 或许这就是目前比较好的算法了吧! 其实,HNSW 本质上就是对 NSW 的增量研究, 因而解释起来应该比较简单.

直观上讲,HNSW 主要做了以下改进以提升算法效率. 以前 (也就是 NSW) 是在一张 Delaunay 图上, 选取 m 个随机的 entry point 进行最近邻的查找, 最后取一个并集, 然后选出最小的 k 个节点作为结果输出; 而 HNSW, 则是生成了多张图 (当然图的节点个数随着层数的增加而减小), 而后在每张图上都选取 1 个 enter point(这个 enter point 在顶层是随机的, 而下面的每层都是上一层的最近邻结果) 进行最近邻的查找, 最后取并集, 选出最小的 k 个接待你作为输出. 由于这多张图中, 仅仅是底层图具有和 NSW 一样的节点个数, 而越往上节点的数量就越小, 所以 HNSW 通过这种方式减少了计算量, 实现了算法的性能提升. 除此之外,enter point 的非随机性, 也使得每次的迭代速度快了很多. 当然, 这种性能提升的代价就是, 生成这样一种层次的多, 图比以往简单生成一张图, 要复杂一些. 换句话说,HNSW 通过加大了 graph construction 的负担来提升 graph search 的性能. 所以还是比较靠谱的.

下文就重点对 graph constrcution 流程进行介绍. 也就是, 探究如何去生成这种层次的图.

首先, 下图是 HNSW 生成的层次图的示意.

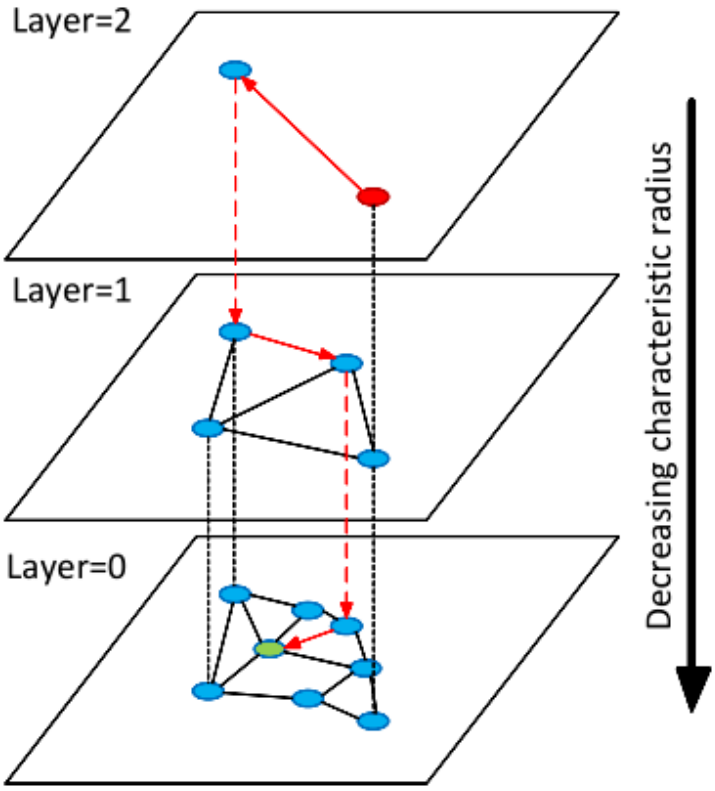


Fig. 1. Illustration of the Hierarchical NSW idea. The search starts from an element from the top layer (shown red). Red arrows show direction of the greedy algorithm from the entry point to the query (shown green).

建议不要按照图例中的描述去理解, 因为它的讲述方式本身不太好理解. 从上图中, 可以发现: 第 0 层的图, 是完整的, 也就是包含所有节点. 随着层数的增加, 节点的数量越来越少 (实际上, 这种减少遵循指数衰减).

生成这样一种层次图的基本步骤是:

1. 对于一个新插入的节点 q, 分析 q 可以最高在第 l=floor(-ln(uniform(0,1)*ml)) 层中出现. 其中,floor 是向下取整.uniform 用于生成 (0,1) 的随机数, 该随机数通过负对数函数可以生成 (0,inf) 的任何数, 而 ml 是确定一个固定的上界, 本质上也就是圈定了层次图的层数.
2. 从最高层依次往下, 一直到 l+1, 每一层中执行操作: 选取 enter point(如果是最高层, 那么随机从选择; 如果是其他层, 那么就是上一层的最近邻), 然后在当前层寻找和 q 最近的 1 个最近邻, 将之加入到候选集合 W 中.
3. 从第 l 层一直下降到底层, 每一层中执行操作: 通过 enter point(不是随机的, 而是步骤 2 中最后的最近邻) 找到一个最近邻集合 W 作为 candidate, 在当前层找到这些 candidate 下的对于节点 q 可能的 M 个最近邻. 并在当前层构筑 q 和这 M 个最近邻的连接.
4. 如果连接过多, 剪掉一些.

这个思路可以用下面的伪代码进行表达:

Output: update *hnsw* inserting element q

$W \leftarrow \emptyset$ // list for the currently found nearest elements
$ep \leftarrow$ get enter point for *hnsw*
$L \leftarrow$ level of ep // top layer for *hnsw*
$l \leftarrow [-\ln(unif(0..1)) \cdot m_L]$ // new element's level
for $l_c \leftarrow L \ldots l+1$
  $W \leftarrow$ SEARCH-LAYER(q, ep, ef=1, l_c)
  $ep \leftarrow$ get the nearest element from W to q
for $l_c \leftarrow \min(L, l) \ldots 0$
  $W \leftarrow$ SEARCH-LAYER(q, ep, $efConstruction$, l_c)
  $neighbors \leftarrow$ SELECT-NEIGHBORS(q, W, M, l_c) // alg. 3 or alg. 4
  add bidirectionall connectionts from $neighbors$ to q at layer l_c
  for each $e \in neighbors$ // shrink connections if needed
    $eConn \leftarrow neighbourhood(e)$ at layer l_c
    if $|eConn| > M_{max}$ // shrink connections of e
                // if $l_c = 0$ then $M_{max} = M_{max0}$
      $eNewConn \leftarrow$ SELECT-NEIGHBORS(e, $eConn$, M_{max}, l_c)
                // alg. 3 or alg. 4
      set $neighbourhood(e)$ at layer l_c to $eNewConn$
  $ep \leftarrow W$
if $l > L$
  set enter point for *hnsw* to q

其中, 搜索的算法为:

Algorithm 2

SEARCH-LAYER(q, ep, ef, l_c)

Input: query element q, enter points ep, number of nearest to q elements to return ef, layer number l_c

Output: ef closest neighbors to q

```
1  v ← ep      // set of visited elements
2  C ← ep      // set of candidates
3  W ← ep      // dynamic list of found nearest neighbors
4  while |C| > 0
5    c ← extract nearest element from C to q
6    f ← get furthest element from W to q
7    if distance(c, q) > distance(f, q)
8      break   // all elements in W are evaluated
9    for each e ∈ neighbourhood(c) at layer l_c   // update C and W
10     if e ∉ v
11       v ← v ⋃ e
12       f ← get furthest element from W to q
13       if distance(e, q) < distance(f, q) or |W| < ef
14         C ← C ⋃ e
15         W ← W ⋃ e
16         if |W| > ef
17           remove furthest element from W to q
18 return W
```

选择最近邻的方法有两种, 分别是简单粗暴的和启发式的, 分别为:

Algorithm 3

SELECT-NEIGHBORS-SIMPLE(q, C, M)

Input: base element q, candidate elements C, number of neighbors to return M

Output: M nearest elements to q

```
return M nearest elements from C to q
```

Algorithm 4

SELECT-NEIGHBORS-HEURISTIC(q, C, M, l_c, *extendCandidates*, *keepPrunedConnections*)

Input: base element q, candidate elements C, number of neighbors to return M, layer number l_c, flag indicating whether or not to extend candidate list *extendCandidates*, flag indicating whether or not to add discarded elements *keepPrunedConnections*

Output: M elements selected by the heuristic

```
1  R ← ∅
2  W ← C   // working queue for the candidates
3  if extendCandidates   // extend candidates by their neighbors
4   for each e ∈ C
5    for each e_adj ∈ neighbourhood(e) at layer l_c
6     if e_adj ∉ W
7      W ← W ⋃ e_adj
8  W_d ← ∅   // queue for the discarded candidates
9  while |W| > 0 and |R| < M
10   e ← extract nearest element from W to q
11   if e is closer to q compared to any element from R
12    R ← R ⋃ e
13   else
14    W_d ← W_d ⋃ e
15  if keepPrunedConnections   // add some of the discarded
                               // connections from W_d
16   while |W_d| > 0 and |R| < M
17    R ← R ⋃ extract nearest element from W_d to q
18 return R
```

在进行 search 时, 方法就简单了许多:

Algorithm 5
K-NN-SEARCH(*hnsw*, *q*, *K*, *ef*)
Input: multilayer graph *hnsw*, query element *q*, number of nearest neighbors to return *K*, size of the dynamic candidate list *ef*
Output: *K* nearest elements to *q*
1 $W \leftarrow \emptyset$ // set for the current nearest elements
2 $ep \leftarrow$ get enter point for *hnsw*
3 $L \leftarrow$ level of ep // top layer for *hnsw*
4 **for** $l_c \leftarrow L \ldots 1$
5 $W \leftarrow$ SEARCH-LAYER(q, ep, ef=1, l_c)
6 $ep \leftarrow$ get nearest element from W to q
7 $W \leftarrow$ SEARCH-LAYER(q, ep, ef, l_c =0)
8 **return** K nearest elements from W to q

4 BFSG Binary Function Search on Graph

Binary function 是指以两个输入为自变量的神经网络函数. 从某意义上讲, 这种函数也可以被认为是一种勉强而特殊的距离度量. 当然, 与 L2,consine 等不同的是, 这种距离度量是非凸的, 因此直接的 search on graph 方法或许无法使用.

WSDM2020 中的 Fast item ranking 一文中给出了一种基于 search on graph 的中和性质的算法,BFSG. 该算法的基本思路非常简单:

1. 在 graph 的构建环节, 不使用神经网络, 而是直接采用 L2 距离进行 HNSW 图的构建, 这个过程与之前介绍的 HNSW 算法并无区别.
2. 在 graph 的 search 环节, 使用神经网络 (也就是所谓的度量函数 f) 进行一切距离度量的准则, 以之作为 Navigation 的基准.

那么针对这种非凸的度量函数, 能否寻找到全局的最优解呢? 作者认为下面两点使得" 摆脱局部最优" 变得有效.

1. 多个 candidate 节点的寻找, 而非一次寻找, 因此可以找到众多的局部最优.
2. HNSW 作为 small word 的性质, 可以供其产生"long-range edges", 这种长程的连接可以减小局部最优的出现概率.

下面是 BFSG 论文中给出的算法伪代码:

1. construciton:

Algorithm 1 Building Index Graphs for SL2G

Input: the data set S, the maximum vertex degree M, and the search depth k.
Initialize graph $G = \emptyset$.
for $x_i \in S$ **do**
    $A \leftarrow$ Search_on_Graph$(x_i, G, k, -\ell_2)$.
    if $|A| \leq M$ **then**
        Connect x_i to vertices in A.
    else
        $m \leftarrow |A|$. $B \leftarrow \emptyset$.
        Order $y_j \in A$ in descending order of $-\ell_2(x_i, y_j)$.
        for $j = 1$ to m **do**
            if $\|x_i - y_j\| \leq \min_{z \in B} \|z - y_j\|$ **then**
                $B \leftarrow B \cup \{y_j\}$.
            if $|B| = M$ **then**
                Break.
        Connect x_i to vertices in B.
Output: G.

1. search:

Algorithm 2 Search_on_Graph (q, G, k, f)

Input: the query element q, the graph $G = (V, E)$, the search depth k, and the searching measure f.
Randomly choose a vertex $p \in V$. $A \leftarrow \{p\}$.
Set p as checked and the rest of vertices as unchecked.
while A does not converge **do**
    Add unchecked neighbors of vertices in A to A.
    Set vertices in A as checked.
    $A \leftarrow$ top-k elements of $v \in A$ in descending order of $f(v, q)$.
Output: A.

5 our method NISG: Neural interface metric search on Graph

在进行 graph construction 的过程中, 单纯使用 L2 距离并不是一个美妙的决定, 同时, 使用神经网络作为 search 函数又难以满足众多非凸的性质. 为了解决这个问题, 我们采用了一种新的方法,