ANNS 问题中 random kd-tree、LSH 和 PQ 算法比较实验结果

朱正茂

2017 年 6 月 1 日

1 算法描述补充

在实验方案设计中，我描述的是最基础的 kdtree 算法（MATLAB 自带 KNN 算法，recall 可达到 100%，但速度极慢，不予比较），之后又使用了 VLFeat 包（我在配置 FLANN 的过程中遇到了一些障碍，所幸搜索后发现 VLFeat 用的 random kd-tree 算法和 FLANN 中是一样的，就直接拿来用了）中的函数来实现 random kd-tree 算法。下面我补充一下试验方案设计中没有介绍的 random kd-tree 算法。

random kd-tree 算法相比于 kd-tree 算法的改进主要在分支维度的选择上，不是只考虑方差最大的维度，而是考虑方差前 d 大的维度，再从中随机选取一个维度作为分支维度，其余不变。

2 实验方案调整

在实际进行实验的时候，我发现对 recall@（k,R）中的 k，R 参数进行调整意义不大。因为对于 k 参数，PQ 算法的论文中提到“We do not include these measures in the paper, as we observed that the conclusions for K = 1 remain valid for K >1.”。而对于 R 参数，仅有 PQ 算法随之变化，

而其余算法的 recall 在 R=100 以后基本都已经稳定，故比较意义不大。所以在实验报告中，我统一使用 recall 来指代 recall@（1，100）。

3 实验结果

如实验方案设计所言，我们建立精度（recall）和查询所耗时间（time/for 10000 queries）的图表。（均使用 sift 数据集）

而鉴于各算法均有参数可以调整，我们改变其中一个变量来绘制折线图来展现各算法的精度、速度关系。同时，LSH 算法和 PQ 算法均有超过一个参数可以调整，这就给绘制折线图增加了复杂性。对于这两个算法，我的处理方法是设置多个参数组，绘制散点图，再在散点图的基础上绘制一条使得所有散点都在其下方的折线图，视为其最优的 recall-time 折线。

在设置参数组的时候，由于 LSH 有三个参数，PQ 算法有四个参数，如果盲目地枚举，制造三重、四重循环，是非常耗时且不必要的。我在处理的时候先以论文中的建议参数设置了初始参数，然后再依次选择一个参数变动，固定其他参数，找到满足“精度不再随时间增长而明显增加”的阈值（这个“明显”非常主观，但我一时也找不到一个超参数来 λ 来构造形如 $f(recall, time) = time * \lambda - recall$ 的代价函数，我主观上是在比较 $\frac{\nabla recall}{\nabla time}$，如果减小了数倍则认为可以停止）。

参数说明：

kd-tree:

d : 回溯次数 (在 BBF 查找中探查路径数)；

LSH:

L : 函数组数目

k : 单个函数组内 hash 函数数目

r :hash 函数分母

PQ:

x : 粗聚类数

m : 子量化单元数

k^*: 子量化单元聚类数

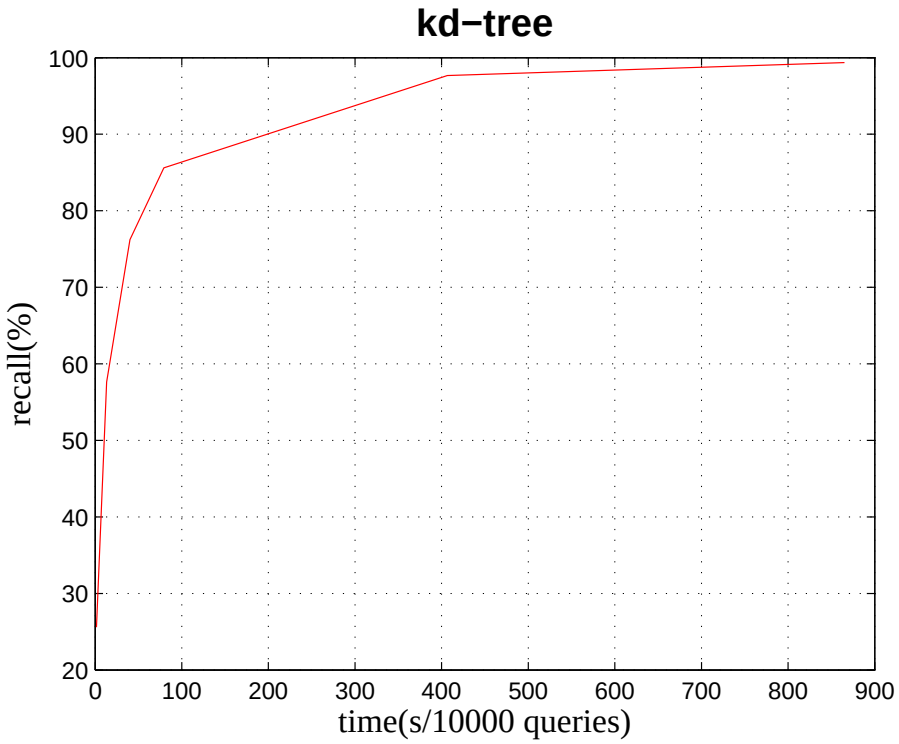
w : 考虑子量化单元中最近的类别数

3.1 kd-tree

对于 kd-tree 算法，只有 d 一个参数，我们直接可以绘制折线图。

表 1: kd-tree

time(s)	1.484	13.203	40.063	79.328	406.719	865.016
recall(%)	25.6	57.62	76.22	85.61	97.68	99.37

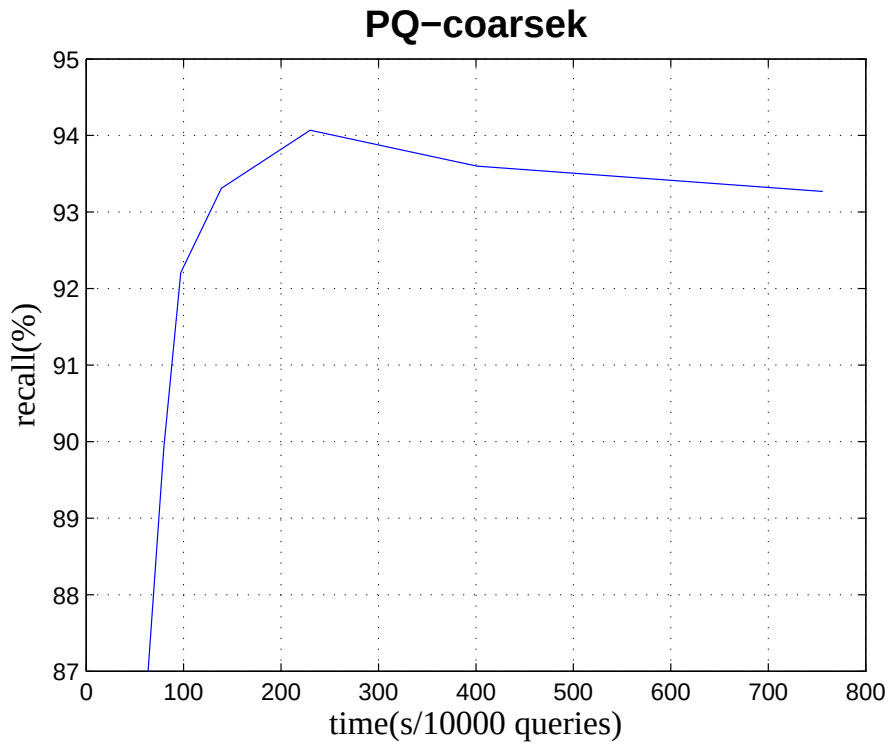


3.2 PQ

对于 PQ 算法，我们先设置其初始参数 $x = 64$,$m = 8$,$k^* = 2560$,$w = 8$, 然后改变参数 $x = \{1024, 512, 256, 128, 64, 32, 16\}$, 固定其他参数，得到下图，发现 $x = 128$ 时（直观上）最好。

表 2: PQ-x

time(s)	63.578	79.641	97.094	138.875	229.828	401.016	755.859
recall(%)	87	89.92	92.21	93.31	94.07	93.6	93.27

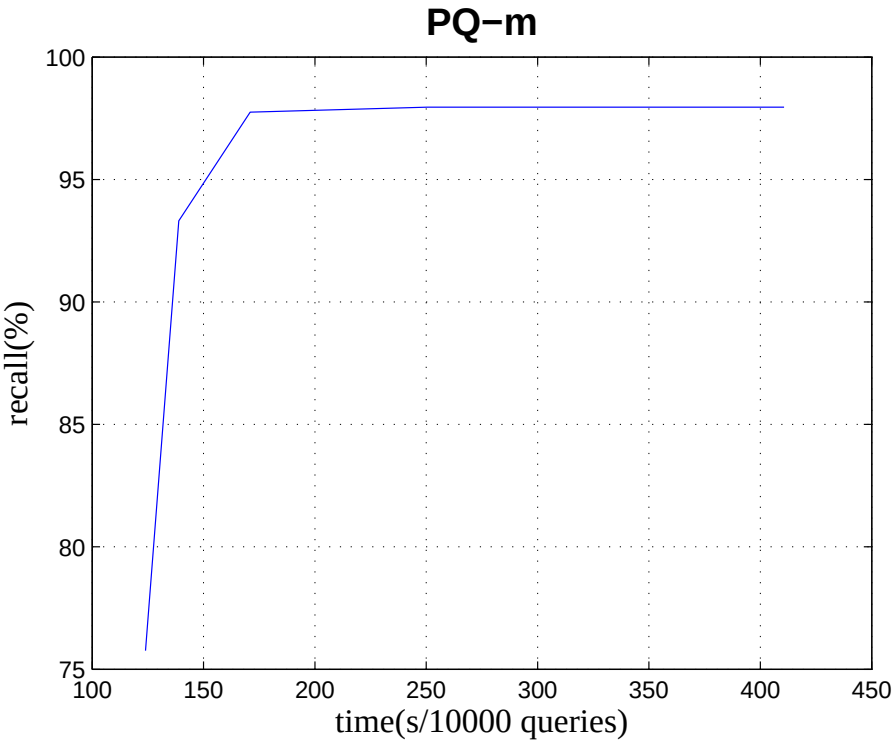


之后固定参数 $x = 128$,$k^* = 2560$,$w = 8$，改变参数 $m = \{4, 8, 16, 32, 64\}$,

得到下图，发现 $x = 32$ 时最好。

表 3: pq-m

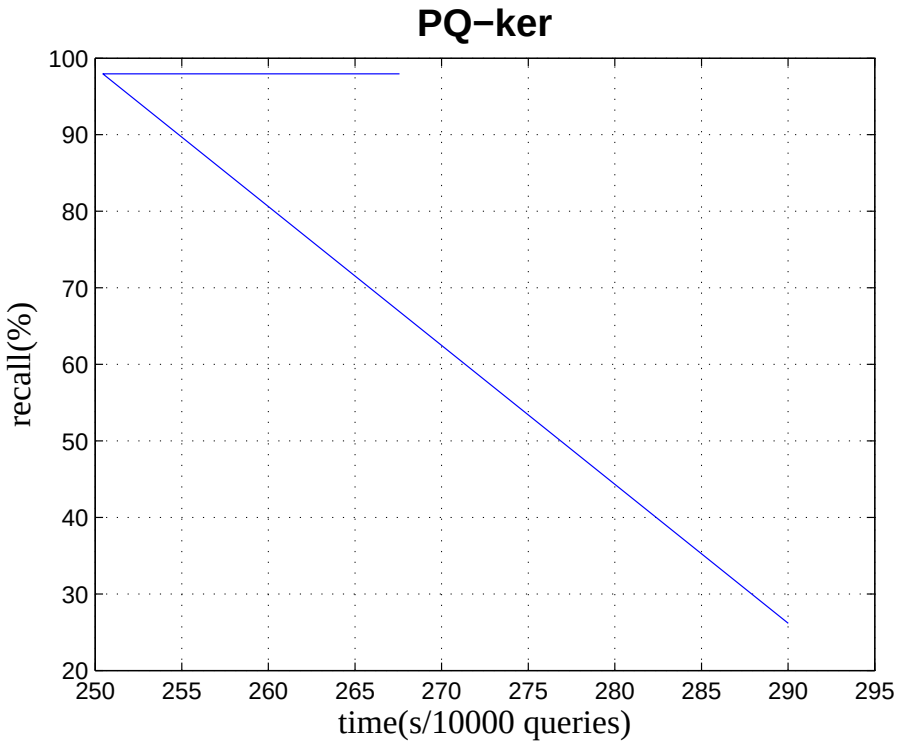
time(s)	123.906	138.875	170.922	250.438	410.656
recall(%)	75.76	93.31	97.75	97.95	97.95



然后固定参数 $x = 128$,$m = 32$,$w = 8$，改变参数 $k^* = \{512, 256, 128\}$，得到下图，发现 $x = 256$ 时最好。

表 4: pq-k^*

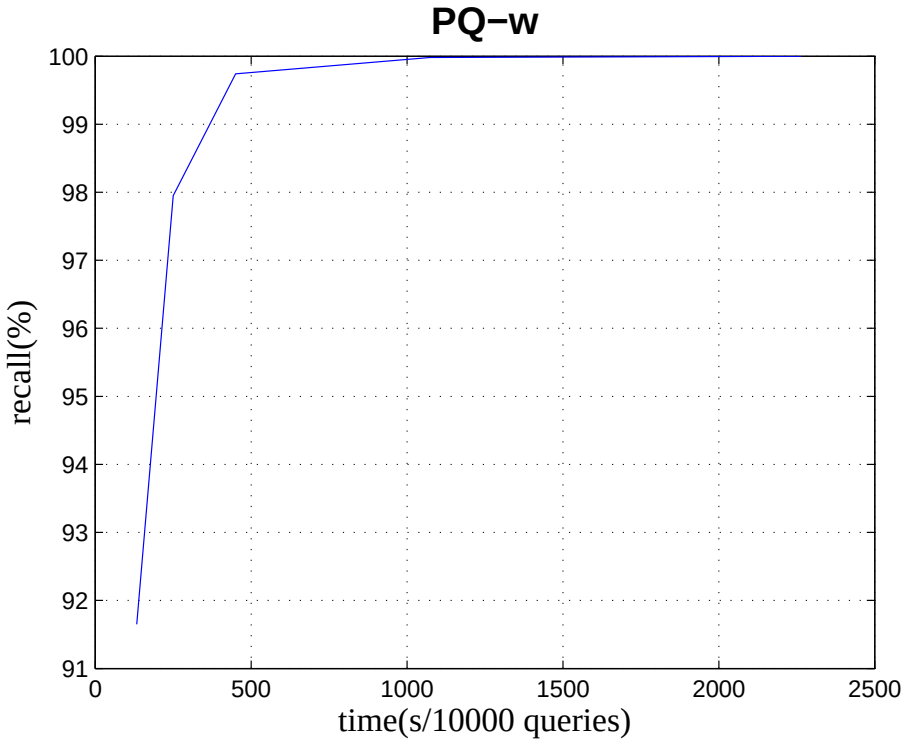
time(s)	267.563	250.438	290.016
recall(%)	97.95	97.95	26.16



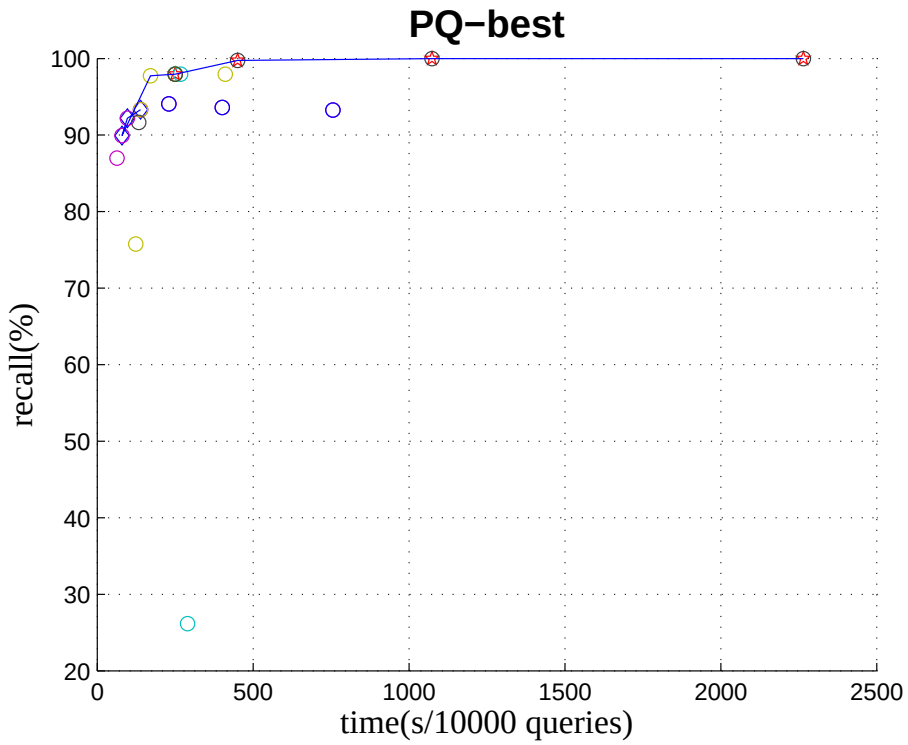
最后固定参数 $x = 128$,$m = 32$,$k^* = 256$，改变参数 $w = \{4, 8, 16, 32, 64\}$，得到下图，发现 $w * 16$ 时效果最好。(其中 $w = 64$ 时 recall 甚至达到了 100%，但耗时太多)

表 5: pq-w

time(s)	133.531	250.438	450.227	1073.7	2265.2
recall(%)	91.65	97.95	99.74	99.98	100



最后结合上面的所有情况，给出近似最佳折线图。



3.3 LSH

对于 LSH 算法，先固定初始参数 $L = 10$,$k = 5$,$w = 300$，然后改变参数 $K = \{4, 5, 6\}$, 固定其他参数，得到 $k = 5$ 时最好。

表 6: lsh-k

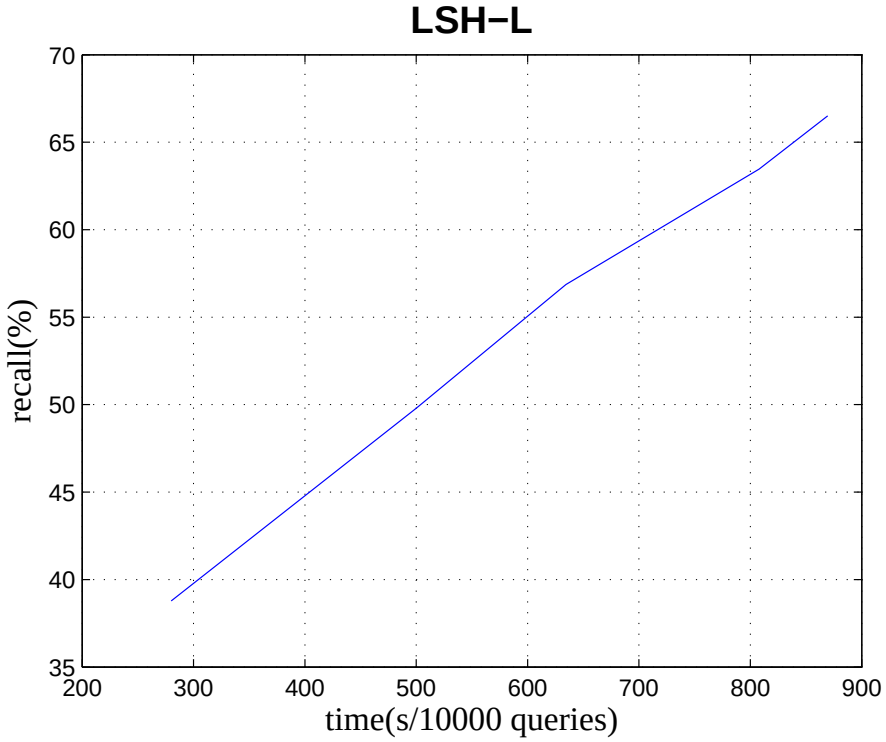
time(s)	822.25	279.875	99.063
recall(%)	56.81	38.78	26.82

然后固定 $k = 5$,$w = 300$，改变参数 $L = \{10, 15, 20, 25, 30\}$, 得到下图，发现 $L = 25$ 时最好（这里结合了上面两个算法的最大耗时考虑，盲目为

了提高 recall 而继续取更大的 L 来试验，导致耗时超过 900s，无益于算法比较）。

表 7: lsh-L

time(s)	279.875	500.313	634.438	807.828	869.422
recall(%)	38.78	49.81	56.87	63.46	66.51

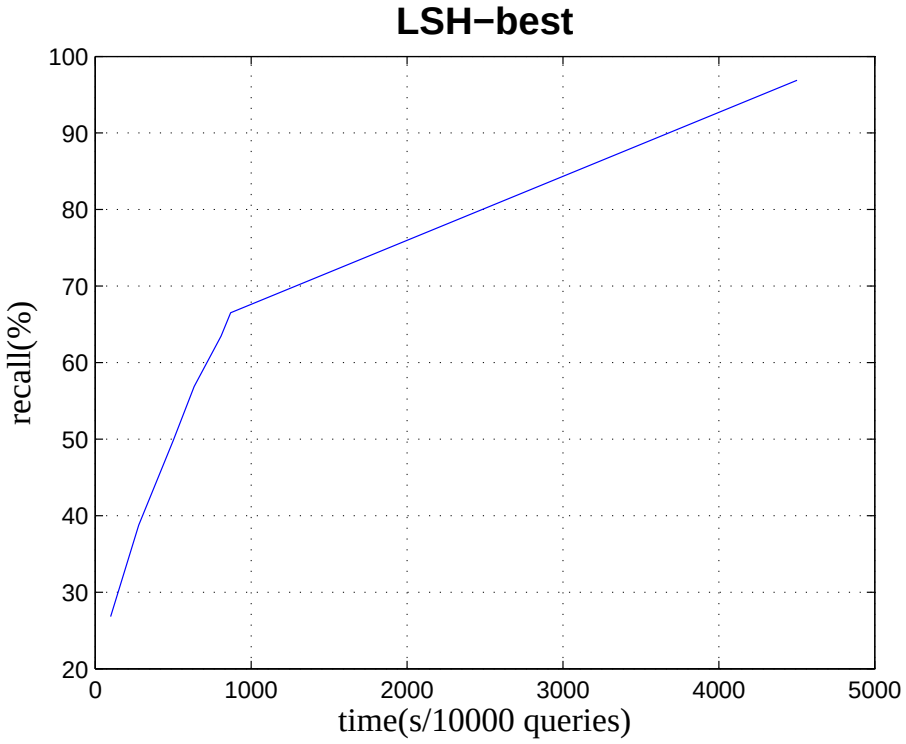


然后固定 $k = 5$,$L = 25$, 改变参数 $w = \{300, 500\}$。

表 8: lsh-w

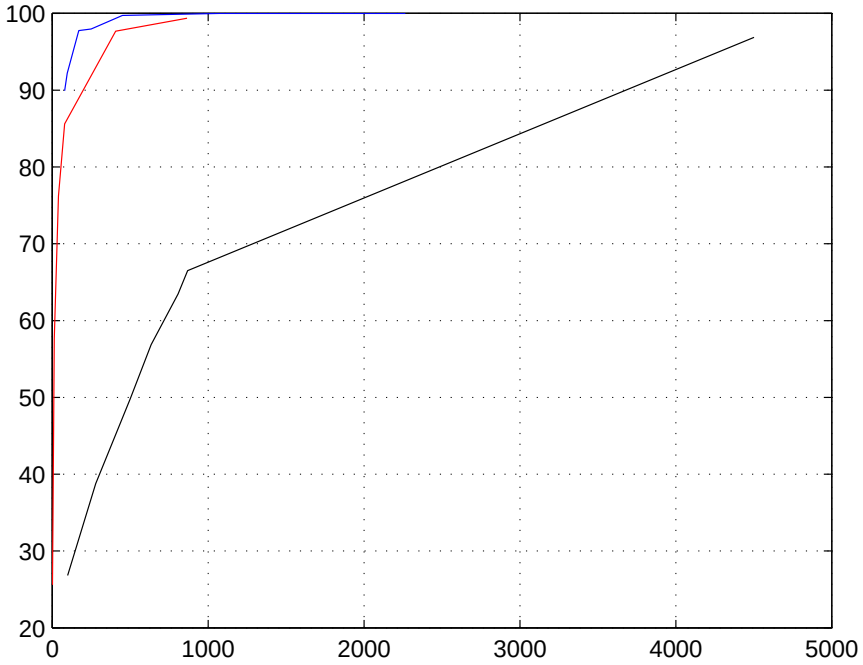
time(s)	869.422	4502.047
recall(%)	66.510	96.880

最后结合上面的所有情况，给出近似最佳折线图。

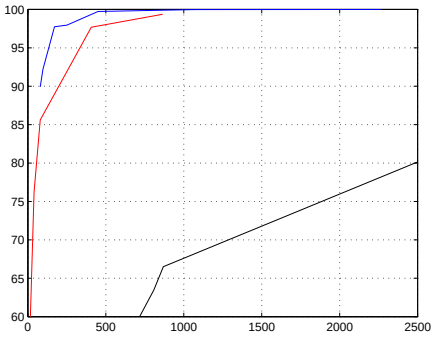


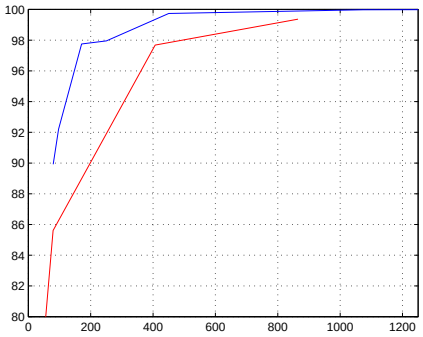
3.4 算法比较

我们把三个算法的折线图放在一起比较，random kd-tree 算法是红线，PQ 算法是蓝线，LSH 算法是黑线，得到



可以看出 LSH 算法明显落后于另外两个算法。我们接下来放大左上角。





可以明显看出 PQ 算法还是略优于 random kd-tree 算法的。不过同时，PQ 算法有众多参数要调整，为了到达这条上界线，在调参上要花费不少时间。而 random kd-tree 算法参数很少，非常容易实现这条上界限。

总而言之，我们如果要在短时间内实现 ANNS，则 random kd-tree 是

个不错的选择，而如果时间相对充沛，且对速度、精度均有一定要求，则可以尝试 PQ 算法。

3.5 gist 数据集

对于 gist 数据集，我猜测结果应该是大同小异的，不过我还是用类似的方法做了一遍实验。下面分别给出 random kd-tree、PQ 的图表和他们放在一起比较的图表。而 LSH 算法我只跑了一次，参数是 $l = 10$,$k = 4$,$w = 300$, 耗时 4 个多小时，recall 达到 100%(补充里对此有所说明)。

表 9: pq-gist-best

coarsek	128	64	64	128	64	128	64	64
m	16	16	32	16	16	32	32	32
k^*	256	256	256	256	256	128	128	128
w	8	8	8	16	16	16	16	32
time(s)	176	207	274	360	427	504	554	1066
recall(%)	97.75	99.11	99.35	99.51	99.69	99.74	99.95	100

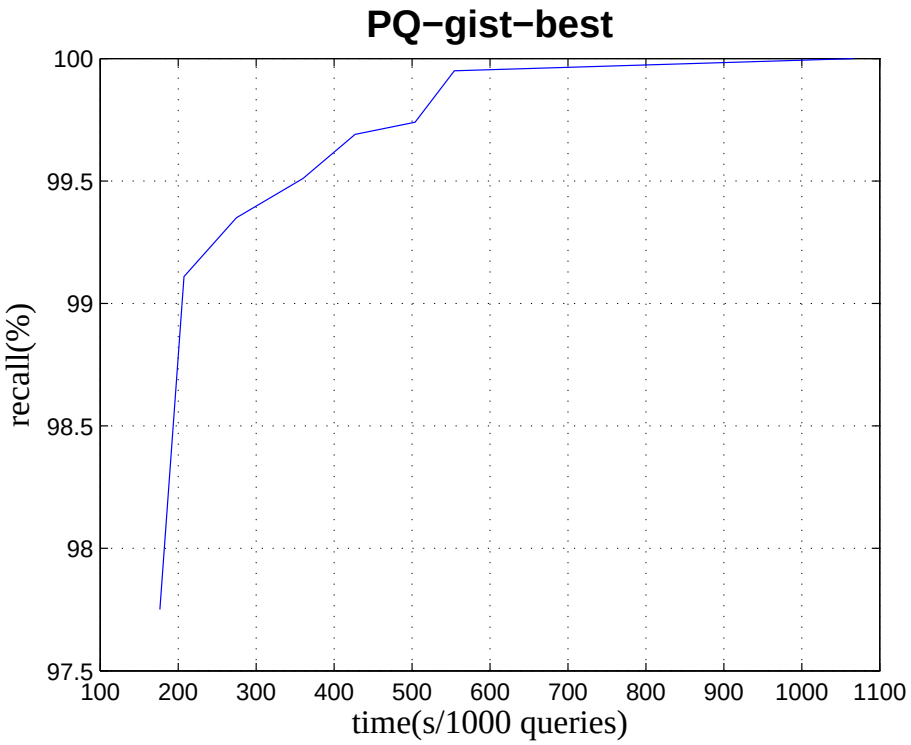
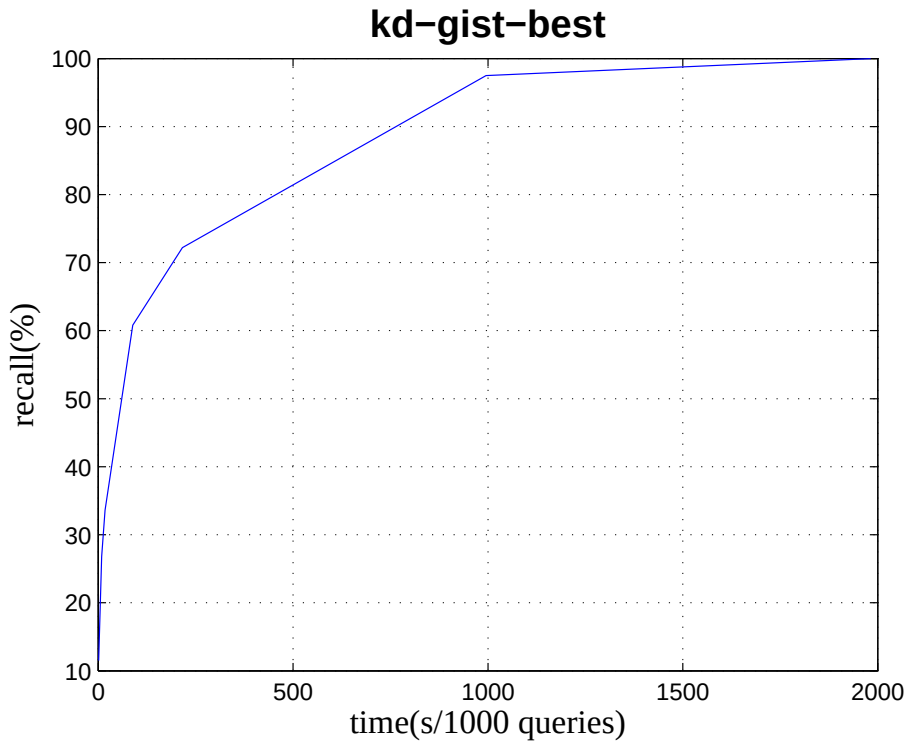


表 10: kd-tree

d	1000	5000	10000	50000	100000	500000	1000000
time(s)	1.188	8.781	17.42	89	216	995	1982
recall(%)	11.5	26.8	33.6	60.8	72.2	97.5	100



其中蓝线是 PQ 算法，红线是 random kd-tree 算法，可以看出和 sift 的结果非常类似。

4 一些补充

1. 代码已上传到 github。阅读 README 可复现我的实验。

https://github.com/siqili1230/ANNS.git

2. 第三部分中讲到的避免三重、四重循环的方法理论上是可能有问题的，这有点像在一个不一定凸的地方试图找到一个局部极小值来代替全局极小值，而且我还没有用迭代法找到局部极小值。但我有两个理由这么做。第一是可以看到 PQ 算法的精度已经很高了，即使我找到的不是全局极小值，可优化的空间也不大了。第二是 LSH 算法的速度确实很慢，和另外两个算法差距太大，我认为不需要找到全局极小值也可以判断优劣性。

3. 我怀疑我用的 LSH 算法可能是落后的（已经是从正规地方找的算法了），毕竟差距太大了。其实我在同一个地方好像有看到某些改进，但没仔细看论文，并没有深入研究。

4. 在用 gist 数据集时，我猜测是由于 gist 中的参数是实数，所以 hash 函数的效果受到影响。根据以往经验，增大 w 和减小 k 可以加速查询过程，我试了很多次参数，包括把 w 增加到 200000，速度都没有明显的提高，所以我就没有继续尝试调整参数，也就没有画相应的图表，但我觉得这应该不影响做出 LSH 是三者中效果最弱的判断。

5. 代码与论文来源:

kd-tree 代码来源：VLFeat 函数包 http://www.vlfeat.org/ 加自己手写

PQ 论文：Jégou H, Douze M, Schmid C. Product quantization for nearest neighbor search.[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2011, 33(1):117.

PQ 代码来源：http://people.rennes.inria.fr/Herve.Jegou/projects/ann.html

LSH 论文：Datar M, Immorlica N, Indyk P, et al. Locality-sensitive hashing scheme based on p-stable distributions[C]// Twentieth Symposium on Computational Geometry. ACM, 2004:253-262.

LSH 代码来源：http://ttic.uchicago.edu/ gregory/download.html