最近点对问题

Martin Hitz [1] 和 Michaela Hitz[2]

[1]信息系统系，克拉根福大学，克拉根福，奥地利，ORCID：0000-0002-8581-1720，martin.hitz@aau.at（通讯作者）[2]统计系，克拉根福大学，克拉根福，奥地利，ORCID：0000-0003-2987-2251，michaela.hitz@aau.at

2026年1月12日

arXiv:2601.05681v1 [cs.DS] 9 Jan 2026

摘要

我们基于数学最优装填理论的研究成果，为在点集群中寻找最近点对的问题提出了两种新算法。两种算法均具有确定性，实际运行速度快，且非常易于实现。对于我们的主要算法 cppMM，我们证明了在均匀分布点情况下的 $O(n)$ 时间复杂度。我们的第二种算法 cppAPs 几乎与暴力法一样简单，但表现出极快的实测运行时间，尽管其最坏情况时间复杂度同样为 $O(n^2)$。我们将这些新算法纳入对最具代表性的竞争算法的评述中，并通过实验证明了它们在问题规模最高达 $n = 33{,}554{,}432$ 个点时的运行时间表现，这些数据是在我们的 C++ 测试环境中采集的。对于较大 n问题规模，cppMM 在所研究的其他算法中表现最佳。

关键词：最近点对问题、算法、复杂度。

1 引言

“最近点对问题”（CPP）是计算几何中的一个经典问题，具有重要的实际应用。例如，现代交通控制系统（如自动驾驶汽车所使用的系统）需要知道距离最近的两辆车，以检测潜在的碰撞。

该问题可表述如下：给定平面上的一个紧集中的 $P = \{p_1, \ldots, p_n\} = \{(p_1.x,\ p_1.y)$、.....、$(p_n.x,\ p_n.y)\}$ 个 $n \in \mathbb{N}$ 点（例如，单位正方形），寻找关于度量 $d_k(p,\ q) = \sqrt[k]{|p.x - q.x|^k + |p.y - q.y|^k}$的距离彼此最近的点对，其中任意 $k \geq 1$均满足此条件。下文中，我们始终使用欧几里得距离$d_2(p,\ q) = \|(p,\ q)\| = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2}$。

文献中报道了几种解法：蛮力法比较所有点对之间的距离，其时间复杂度为 $\Theta(n^2)$；分治算法的运行时间为 $O(n \log n)$；还有一种 $O(n \log n)$ 的平面扫描算法 [2]。这些都是计算决策树模型下的最优算法示例；对于这个问题，该模型要求至少 $\Omega(n \log n)$ 步 [5]。Rabin [6] 提出了一种概率算法，后来 Lipton [4]对其进行了评述和略微修改。若使用哈希表模拟一个可能非常庞大的二维矩阵，并保证对矩阵元素的访问时间为 $O(1)$，则后者可在 $O(n)$ 时间内解决该问题。为此，该算法利用拓扑信息，将所有点分配到网格的单元格中；网格大小取决于最短距离的估计值，而该估计值通过对所有输入点进行随机抽样并采用蛮力法计算得出

[1]大多数算法 都可以推广到高维空间中的点。为清晰起见，在表述中， 我们仅考虑二维情形。

另一种由 Khuller 和 Matias 提出的概率 $O(n)$ 算法与 Rabin-Lipton 方法有些相似：它同样使用网格来捕捉点的拓扑分布，网格大小也取决于根据随机样本计算出的最短距离近似值 [3]。

我们重新研究了 CPP 问题，并提出了两种易于实现、具有确定性、非递归且结果极具前景的新算法。我们的主要贡献算法 cppMM 的核心思想基于数学最优装填理论的研究结果。¯具体而言，我们利用了已证明的上界 δ，即规模为 n 的点云中任意两点之间最小距离 δ 的上界，以构建一个类似于上述网格的网格 [8]。这样便省去了近似最小距离的步骤。我们的方法产生了一个明显更粗、因而可以直接实现的大小为 $O(n)$ 的网格，同时在许多情况下保持线性平均情况复杂度。另一种算法 cppAPs 是暴力法的显著加速版本，因而在实践中具有很好的实用性；此外，它还加入了一个类似于 Hinrichs 等人所使用的“活动窗口”的简单停止条件。 [2]

我们对这些算法进行了全面的模拟研究，并将其与文献中的若干算法进行了比较。此外，对于 cppMM，我们证明了在均匀分布点情况下具有线性时间复杂度。所讨论的所有算法均使用 C++实现，并通过使用随机生成的、点数从$2^{10} = 1024$到 $2^{25} = 33\,554\,432$的点云进行了经验运行时间比较。

2 经典算法

比较所有点对 直接算法 cppAP 会比较所有 $\binom{n}{2}$ 对点，并用时 $\Theta(n^2)$，见算法 1。

算法 1 $\mathrm{cppAP}(P) \rightarrow < (p_{min}, q_{min})\ \delta >$

要求： $P = \{p_1, \ldots, p_n\}, n \geq 2$
确保： $\delta =$ 最小值$\{\|(p,q)\| \mid p, q \in P,\ p \neq q\}$

```
1: δ ← ∞
2: 对于 i ← 1 到 n − 1，执行
3:     对于 j ← i + 1 到 n，执行
4:         如果 ‖(p_i, p_j)‖ < δ，则
5:             δ ← ‖(p_i, p_j)‖，最近点对问题 ← (p_i, p_j)
6: 返回 < 最近点对问题 δ >        返回一个由最近的两个点及其距离组成的结构 δ
```

请注意，可以将欧几里得距离 $\|(p,\ q)\|$替换为其平方，从而避免在第 4 行和第 5 行计算平方根，并返回 δ 内联 6.[2] 的平方根，以此线性加速 cppAP

分治 一种更快的方法是分治算法；它的时间复杂度为 $O(n\log\ n)$ [1]。其思路是按照点的 x 坐标将点集划分为两半，递归地求出每一半中的最近点对，并取两个所得距离中的最小值 δ。然后，检查两半之间是否存在一对点，其距离比任一半中的最小距离更小；如果存在，则将该距离作为最终结果。

[2]在我们的实验设置中，这种加速约为 7.5%。这项技巧可应用于所研究的若干种算法。不过，在实证研究中，我们始终使用欧几里得距离本身，因为有些算法需要用到它。

算法 2 a $\mathrm{cppDC}_0(P) \rightarrow \langle (p_{min}, q_{min})\ \delta \rangle$

要求： $P = \{p_1, \ldots, p_n\}, n \geq 2$
确保： $\delta =$最小值$\{\|(p,q)\| \mid p, q \in P, p \neq q\}$

$X \leftarrow$ 排序$_x(P)$
$Y \leftarrow$ 排序$_y(P)$
返回 $\mathrm{cppDC}(X, 1, n, Y)$ 参见算法 *2b*

为了在划分阶段对平面进行分区，准备步骤（cppDC_0）中，输入集 P 按照 x 坐标排序，存入数组 $X = [(x_i,\ y_i) \in P \mid x_i \leq x_j,\ 1 \leq i < j \leq n]$。在每个递归步骤中，划分阶段将 X 中相关部分拆分为两个子数组，分别位于其中点的左侧和右侧。在合并阶段，会搜索以该中点为中心、宽度为 a 2δ 的（竖直）条带，查找可能距离更近的点对。对于该条带左半部分中的每个点，只检查右半部分中与其 y 距离至多为 $\pm\delta$的点。为了支持这一过程，准备步骤中还会生成第二个数组 $Y = [(x_i,\ y_i) \in P \mid y_i \leq y_j,\ 1 \leq i < j \leq n]$，其中包含 P 中的点，并按 y 坐标递增的顺序排列。详情见算法 2。

算法 2 b $\mathrm{cppDC}(X,$ 首个，最后一个 $Y) \rightarrow <(p_{min},\ q_{min})\ \delta >$

要求： $n :=$最后一个 $-$ 首个 $+1,\ n = |Y|,\ n \geq 2$
确保： $\delta =$最小值$\{\|(p,\ q)\| \mid p,\ q \in P,\ p \neq q\}$

如果 $n \leq 3$ **那么**
  $<$ cpp $\delta >\leftarrow$ cppAP(X[首个 : 最后一个]) 蛮力法，参见算法 *1*
否则
  $m \leftarrow \lfloor$(第一个 $+$最后一个)$/2 \rfloor$
  中位数 $\leftarrow (x_m,\ y_m) \in X$
  $Y_l \leftarrow [(x_i,\ y_i) \in Y \mid x_i \leq$中间 $.x]$ 顺序保持
  $Y_r \leftarrow [(x_i, y_i) \in Y \mid x_i >$中 $.x]$ 顺序保持
  $<$最近点对问题$_l,\ d_l >\leftarrow$ cppDC(X，第一个 m，Y_l)
  $<$最近点对问题$_r,\ d_r >\leftarrow$ cppDC(X，$m+1$，最后一个 Y_r)
  如果 $d_l < d_r$ **那么**
    $\delta \leftarrow d_l$，最近点对问题 $\leftarrow$ 最近点对问题$_l$
  else
    $\delta \leftarrow d_r$，最近点对问题 $\leftarrow$ 最近点对问题
  $X' \leftarrow [(x_i,\ y_i) \in X \mid$中点 $_r.x - \delta \leq x_i \leq$中点 $.x]$
  $Y' \leftarrow [(x_i,\ y_i) \in Y_r \mid x_i \leq$中位数 $.x + \delta]$ 顺序保持
  对于 $p_1 \in X'$ **执行**
    对于 $p_2 \in [p \in Y' \mid |p_1.y - p.y| \leq 2\delta]$[3] **执行**
      如果 $\|(p_1,\ p_2)\| < \delta$ **则**
        $\delta \leftarrow \|(p_1,\ p_2)\|$，最近点对问题 $\leftarrow (p_1,\ p_2)$
返回 $<$ 最近点对问题 $\delta >$

平面扫描 Hinrichs 等人的平面扫描方法 [2] 基于 Shamos 和 Hoey [7]提出的一般平面扫描技术。该迭代算法将一条垂直线从平面的左侧（尚未考虑任何点）向右移动，每次处理一个点，直到所有点都经过检查并获得整个点集的最终解。

[3]为了计算这个子集 Y'，我们对不高于 $p_1.y + \delta$ 的最高点进行二分查找，然后向下遍历，直到到达 $p_1.y$ 下方 δ 的高度。

这种对平面的“扫描”由一个按 x 坐标排序的点队列 $xQueue$负责协调。用于与已处理点进行比较的当前相关点被维护在一个基于 y 坐标 $yTable$ 的映射表中，该映射表将点的 y 坐标与各自的点相关联。

$xQueue$ 初始化为按 x 坐标排序的点集。最近点对 cpp 初始化为队列中的前两个点， δ 保存它们之间的距离，也就是目前找到的最短距离。这两个点也会被插入到$yTable$中。在遍历所有点的循环中（ $xQueue$中的第三个点 q_3开始），我们首先从 $yTable$ 中移除所有位于 $current$ 左侧且距离超过 δ 的点 q_t ——也就是说，这些点位于所谓的“活动窗口”之外。然后，将 $yTable$ 中 y 坐标介于 $current.y - \delta$ 和 $current.y + \delta$ 之间的剩余点与 $current$进行比较；如果距离小于 δ，就会得到新的最近点对，并更新 δ。以这种方式检查完 $current$左侧完整的矩形 $\delta \times 2\delta$区域后，将点 $current$ 加入$yTable$，然后循环继续处理下一个点。

算法 3 cppPS(P) $\rightarrow <(p_{min}, q_{min})\ \delta >$

要求： $P = \{p_1, \ldots, p_n\},\ n \geq 2$
确保： $\delta =$最小值$\{\|(p,\ q)\| \mid p,\ q \in P,\ p \neq q\}$

```
1:  xQueue = [q_1, ..., q_n] ← 排序x(P)   xQueue 保存按其 x 坐标 q_i 排序的点
2:  yTable ← []                          yTable 将点的 y 坐标映射到点本身
3:  最近点对问题 ← (q_1, q_2)
4:  δ ← ‖(q_1, q_2)‖
5:  yTable.insert(q_1), yTable.insert(q_2)
6:  t ← 0                                                  ≙ 将指针“尾部”指向 xQueue [2]
7:  对 i ← 3 到 n 执行
8:      当前 ← q_i
9:      当 q_t.x ≤ 当前。x − δ 时
10:         yTable.delete(q_t)
11:         t ← t + 1
12:     对于 p ∈ [p ∈ yTable | |p.y − 当前.y| < δ] 执行
13:         如果 p ≠ 当前 ∧‖(当前 p)‖ < δ 则
14:             δ ← ‖(当前 p)‖, 最近点对问题 ← (当前 p)
15:     yTable.insert(current)
16: 返回 < cpp δ >
```

正如 Hinrichs 等人所解释的， [2], 初始化 $xQueue$ 所需的运行时间为 $O(n\log\ n)$，而向 $yTable$中插入（或删除）一个 y 坐标值所需的运行时间为 $O(\log\ n)$。由于在遍历所有 n 个点的一次迭代中，每个点都要执行一次后一操作，因此总运行时间为 $O(n\log\ n)$。

Rabin-Lipton Rabin [6] 展示了如何使用概率算法在线性时间内求解最近点对问题（CPP）；该算法经 Lipton 评论并稍作改编。Rabin-Lipton 方法 cppRL 的步骤如下：从 P 中随机抽取 $\sqrt{n}$ 个点，并用 cppAP 计算该样本中各点的最小距离 d 。这需要 $O(\sqrt{n^2}) = O(n)$ 步。然后将单位正方形划分为边长为 $d \times d$ 的方形单元格网格。尽管当 d 的值非常小时，单元格数量 $\lceil 1/d \rceil \times \lceil 1/d \rceil$ 可能非常庞大（$\gg n$），但最多只有 n 个单元格会包含 P 中的一个或多个点。我们需要以一种方式组织这些最多 n 个“非空”单元格，使得能够在 $O(n)$ 步内依次访问它们，并在 $O(1)$ 步内单独访问任意一个。大小为 $O(n)$、键由网格索引 i, j 组成的哈希表即可满足要求。利用这一数据结构，对每个点 $p \in P$，检查所有可能与其距离不超过 d 的点。这些点必定位于 p 所在的单元格内，或位于最多八个相邻的非空单元

格中。因此，对于 P 中的每个 n 个点，只需计算它与最多九个单元格中各点之间的距离。[4] [5] 算法 4 介绍了具体细节。正如 Rabin 所示，这可以在常数时间内完成，从而使整体平均时间复杂度达到 $O(n)$。请注意，Lipton 认为，为了证明该结果，应当对距离 而非点 [4]进行采样。在我们的实验中，距离采样在实际测试中也比点采样快，因此下文给出了采用距离采样的 cppRL 运行时间。

算法 4 $\mathrm{cppRL}(P) \rightarrow < (p_{min}, q_{min})\ \delta >$

要求： $P = \{p_1, \ldots, p_n\} \subset [0,1]^2,\ n \geq 2$
确保： $\delta =$ 最小值$\{\|(p,\ q)\| \mid p,\ q \in P,\ p \neq q\}$

$S \leftarrow$ 随机采样$(P,\ \lfloor\sqrt{n}\rfloor)$ 取大小为 $\lfloor\sqrt{n}\rfloor$ 的随机样本
$< $最近点对问题 $d >\leftarrow$ cppAP(S) 蛮力法，见算法 *1*
$g \leftarrow \lceil 1/d \rceil$
$G :=$ $g \times g$ 空桶网格 G 使用大小为 O 的哈希表(n)
</style}对于 $p \in P$，**执行**
  $i \leftarrow \lfloor p.x/d \rfloor + 1,\ j \leftarrow \lfloor p.y/d \rfloor + 1$
  $G_{i,j} \leftarrow G_{i,j} \cup \{p\}$ $G_{i,\ j}$ 是（现在）一个“已填充”的桶
  visited($G_{i,\ j}$) $\leftarrow$ **假** 仅当 *visited*($G_{i,\ j}$) 未在第 *4* 行初始化时才需要
$\delta \leftarrow d$
对 $p \in P$ **执行**
  $i \leftarrow \lfloor p.x/d \rfloor + 1, j \leftarrow \lfloor p.y/d \rfloor + 1$ 找到“已填充”的桶 $G_{i,\ j}$
  如果 ¬已访问($G_{i,\ j}$)**则**
    已访问($G_{i,\ j}$) $\leftarrow$ **为真** 确保仅访问桶一次
    对于 $p_1 \in G_{i,\ j}$，**执行**
      对于 $p_2 \in (G_{i,\ j} \setminus \{p_1\}) \cup G_{i,\ j+1} \cup G_{i+1,\ j-1} \cup G_{i+1,\ j} \cup G_{i+1,\ j+1}$[4] [5] [6] **执行**
        如果 $\|(p_1,\ p_2)\| < \delta$ **则**
          $\delta \leftarrow \|(p_1,\ p_2)\|$, 最近点对问题 $\leftarrow (p_1,\ p_2)$
返回 $< \mathrm{cpp}\ \delta >$

Khuller-Matias Khuller 和 Matias 提出另一种随机化算法，其时间复杂度为 $O(n)$，该算法基于一种过滤方法（筛法） [3]。筛选过程从复制整个点集 P_1 开始，并按以下步骤迭代进行：在第 i 次迭代中，从 P_i 中随机选取一个点 x_i，计算 δ 的近似值，并求出该点到 P_i 中其他点的最小距离$d(x_i)$。然后，构造一个大小为 $d(x_i)/3$ 的网格，并将 P_i 中的点分配到网格单元中。利用该网格，定义 P_{i+1}：从 P_i 中取出所有点，但排除那些恰好独自位于其网格单元、且周围八个相邻单元均为空的点。当 P_{i+1} 变为空时，本次迭代终止。在算法的最后阶段，创建一个大小为最后一次计算所得 $d(x_i)$ 的网格，并将 P 中的点分配到网格单元中。然后，对于每个点，在其所在网格单元及其八个相邻单元中寻找最近的配对点。从这些 n 个“局部”最近点对中，选出距离最小的一对作为最终结果。具体细节见算法 5。与 cppRL 类似，这里使用的网格以哈希表实现，以保证 $O(n)$ 的空间复杂度。

[4]出于对称性考虑，只需检查该单元格及其一半的邻居，详情请参见第3.2节。[5]请注意，原始方法分为两个独立阶段：首先，计算每个非空网格单元中的最近点对；其次，计算每对相邻的非空单元中的最近点对，参见 [4]。尽管我们的实现基本上做的是同样的事，但我们认为，我们的方案“计算该单元格及其非空邻居中的最近点对”更容易理解。[6]用于实现 G 的哈希表可确保对于不存在的网格单元索引，返回空桶，因此这里无需处理边界条件。

算法 5 cppKM(P) $\rightarrow <(p_{min}, q_{min})\ \delta>$

要求： $P = \{p_1, \ldots, p_n\} \subset [0,1]^2, n \geq 2$

确保： $\delta = \min\{\|(p,q)\| \mid p, q \in P, p \neq q\}$

$i \leftarrow 1$, $P_1 \leftarrow P$

当 $P_i \neq \varnothing$ **时**

  $x_i \leftarrow$ 随机采样$(P_i,1)$ 从 P_i 中随机选择一个点

  $d \leftarrow 2$

  对 $p \in P_i \setminus \{x_i\}$ **执行**

    如果 $\|(x_i, p)\| < d$ **则**

      $d \leftarrow \|(x_i, p)\|$

  $g \leftarrow d/3$

  $G :=$ $g \times g$ 个空桶组成的网格 G 使用大小为 $O(n)$ 的哈希表

  对于 $p \in P_i$ **执行**

    $i \leftarrow \lfloor p.x/g \rfloor + 1$, $j \leftarrow \lfloor p.y/g \rfloor + 1$

    $G_{i,j} \leftarrow G_{i,j} \cup \{p\}$

  $P_{i+1} \leftarrow P_i$ 准备下一次迭代

  对于 $p \in P_i$ **执行**

    $i \leftarrow \lfloor p.x/g \rfloor + 1$, $j \leftarrow \lfloor p.y/g \rfloor + 1$

    $N \leftarrow G_{i-1,j-1} \cup G_{i-1,j} \cup G_{i-1,j+1} \cup G_{i,j-1} \cup G_{i,j} \cup G_{i,j+1} \cup G_{i+1,j-1} \cup G_{i+1,j} \cup G_{i+1,\ j+1}$ [6]

    如果 $|N| = 1$ **，则**

      $P_{i+1} \leftarrow P_{i+1} \setminus \{p\}$ 从下一次迭代中移除 p

  $i \leftarrow i + 1$

$G :=$ $d \times d$ 个空桶组成的网格

对于 $p \in P$ **，执行**

  $i \leftarrow \lfloor p.x/d \rfloor + 1$, $j \leftarrow \lfloor p.y/d \rfloor + 1$

  $G_{i,j} \leftarrow G_{i,j} \cup \{p\}$

$\delta \leftarrow d$

对于 $p_1 \in P$ **执行**

  $i \leftarrow \lfloor p_1.x/d \rfloor + 1$, $j \leftarrow \lfloor p_1.y/d \rfloor + 1$

  $N \leftarrow G_{i-1,j-1} \cup G_{i-1,j} \cup G_{i-1,j+1} \cup G_{i,j-1} \cup (G_{i,j} \setminus \{p_1\}) \cup G_{i,j+1} \cup G_{i+1,j-1} \cup G_{i+1,j} \cup G_{i+1,j+1}$ [6]

  对于 $p_2 \in N$ **执行**

    如果 $\|(p_1, p_2)\| < \delta$ **则**

      $\delta \leftarrow \|(p_1, p_2)\|$, 最近点对问题 $\leftarrow (p_1, p_2)$

返回 $<$ 最近点对问题 $\delta >$

3 新算法

本节介绍两种新算法——一种是对 cppAP 的改进，我们称之为 cppAPs（“所有点对排序”的缩写）；另一种是我们的主要成果 cppMM。请注意，这两种算法都是完全确定性的。

3.1 改进全点对比较方法：cppAPs

乍看之下，在存在渐近速度更快的算法时，改进一种 $O(n^2)$ 方法似乎有些奇怪。然而，我们的全点对排序算法 cppAPs 极易实现，而且在我们的实验设置中，它在均匀分布的测试点情况下，对于最多 $n = 2^{21}$ 个点的输入，是所有受测算法中速度最快的（见图 4）；对于截断正态分布的测试点，则在最多 $n = 2^{23}$ 个点的输入下最快（见图 5）。因此，尽管其最坏情况时间复杂度仍为 $O(n^2)$，cppAPs 似乎仍是实用场景下的一个不错选择。

cppAPs 的基本思路如下：首先，按照点的 x 坐标对 P 排序（得到点列表 X）。接着采用与 cppAP 类似的方法，但在两个**for**循环中，按 x 坐标递增的顺序检查序列中所有可能的点对。对于给定的点 $q_i \in X$，如果嵌套的**for**循环（从点 q_{i+1} 开始）到达点q_j 时，其 x 坐标与 q_i 相差超过当前最小距离 d（例如 $q_j.x - q_i.x \leq d$），那么就没有必要检查该点及其后续点，因为当前最小值不会再改变。请注意，这与 Hinrichs 等人 [2] 提出的聚焦于“活动窗口”的方法有关。因此，嵌套的**for**循环可能经过很少的迭代就会终止，但遗憾的是，迭代次数也可能达到 $O(n)$。详细内容见算法 6。

算法 6 $\mathrm{cppAPs}(P) \rightarrow < (p_{min}, q_{min})\ \delta >$

要求： $P = \{p_1, \ldots, p_n\}, n \geq 2$
确保： $\delta = \min\{\|(p,q)\| \mid p, q \in P, p \neq q\}$

```
Q = [q_1, ..., q_n] ← 排序x(P)
δ ← ∞
</style}对 i ← 1 到 n − 1 执行
    对于 j ← i + 1 到 n，当 q_j.x − q_i.x ≤ δ 时执行
        如果 ‖(q_i,q_j)‖ < δ，则
            δ ← ‖(q_i, q_j)‖, 最近点对问题 ← (q_i, q_j)
返回 < cpp δ >
```

为了说明嵌套**for**循环（算法 6 的第 4–6 行和算法 1 的第 3–5 行）的行为，我们比较算法 cppAP 和 cppAPs 中输入规模 n 翻倍时，其迭代次数 $I_2(n)$ 的增长情况。图 1 绘出了 cppAP 和 cppAPs 中因子 $I_2(n)/I_2(n/2)$ 分别从 $n = 2^{11}$ 到 $n = 2^{25}$ 的变化。我们观察到，输入规模翻倍时，cppAP 的嵌套**for**循环的迭代次数总是翻倍，而 cppAPs 只有在非常缓慢地逼近因子 2，且仅当 n 的值非常大时才会如此。因此，尽管 cppAPs 的渐近时间复杂度为 $O(n^2)$，但直到 n 非常大时，其运行时间都接近线性。由此，对于许多实际应用而言，cppAPs 是一个非常不错的选择。

3.2 一个确定性线性复杂度算法：cppMM

为了构造我们的主要算法，我们采用了 Rabin 的思路：将单位正方形划分为由单元格组成的网格，并在这些单元格及其邻近单元格中搜索最近点对。在 cppRL 中，由于网格尺寸过小，无法在线性时间内简单地搜索网格。我们的想法是尽可能增大网格，同时确保网格仍足够小，使得每个单元格所需的比较次数不会太多。因此，我们需要尽可能精确地给出最小距离最大值的上界，

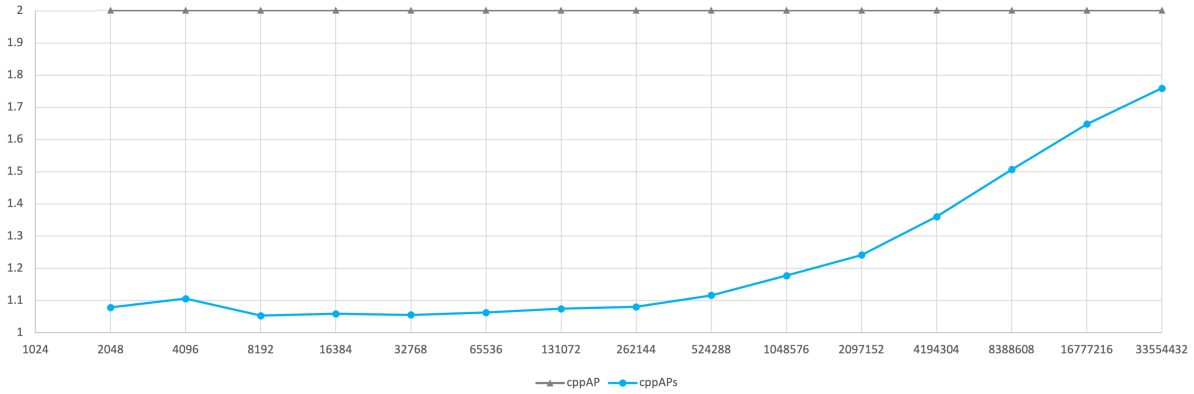

图 1：对于 n 个点，cppAP 和 cppAPs 嵌套 **for** 循环的迭代次数 $I_2(n)$ 除以 $I_2(n/2)$。横坐标采用对数刻度。

我们将利用数学最优装填理论中的以下结果，该结果源自 Szabó 等人 [8]。

定理 3.1 ([8,定理 2])。 最小距离的最大值不超过

$$\bar{\delta} = \min(U_1(n), U_2(n)), \tag{1}$$

with

$$U_1(n) = \frac{2}{\sqrt{n\pi + C_n(\sqrt{3} - \frac{\pi}{2}) + (4\lfloor\sqrt{n}\rfloor - 2)(2 - \frac{\pi}{2})} - 2},$$

其中

$$C_n = \begin{cases} n-2, & \text{if } 3 \le n \le 6, \\ n-1, & \text{if } 7 \le n \le 9, \\ 3\lfloor\frac{n}{2}\rfloor - 5 + n \bmod 2, & \text{otherwise} \end{cases}$$

and

$$U_2(n) = \frac{1 + \sqrt{1 + (n-1)\frac{2}{\sqrt{3}}}}{n-1}.$$

由于 $C_n \in O(n)$，我们有 $U_1(n) \in O(\frac{1}{\sqrt{n}})$，以及 $U_2(n) \in O(\frac{1}{\sqrt{n}})$，这表明 $\bar{\delta} \in O(\frac{1}{\sqrt{n}})$。

我们利用这一结果将单位正方形划分为一个 $\lceil 1/\delta \rceil \times \lceil 1/\delta \rceil$ 网格 G，网格单元大小为 $\delta \times \delta$，并将每个点 $p \in P$ 分配到相应的单元中。这与 cppRL 类似，但采用的网格仅包含 $O(n)$ 个单元，可直接用二维数组实现，且不会影响线性运行时间或线性空间复杂度。我们的算法还有其他优点：省去了 cppRL 中的预计算步骤，并且因此成为确定性算法，而 cppRL 是一种随机化算法。

为了找到最近点对，我们遍历网格中的所有点，检查同一网格单元内所有可能的配对点 $G_{i,j}$、右侧相邻网格单元内的点 $G_{i,j+1}$、左下方对角网格单元内的点 $G_{i+1,j-1}$、正下方网格单元内的点 $G_{i+1,j}$，以及右下方对角网格单元内的点 $G_{i+1,j+1}$（如果这些网格单元存在）。图 2 展示了对网格单元 $G_{2,4}$ 进行的搜索。

其他单元格中的所有配对点，要么已在先前的迭代中检查过，要么由于网格的定义而相距太远，不可能构成满足最小距离要求的有效配对点；也就是说，最终最近点对的距离不会超过单元大小 δ。算法 7 描述了“最大最小距离”算法 cppMM 的详细信息。

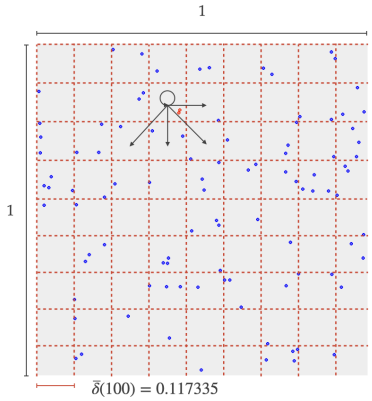

图 2：对于单位正方形中的 $n = 100$ 个点，最小距离的上界为 $\bar{\delta} = 0.117$。由于 $1/\delta = 8.523$，我们构造一个 9×9 网格，正方形右侧和底部边界处的网格单元较小。最近点对要么完全位于同一个网格单元中，要么一个点位于某个单元中，另一个点位于其八个相邻单元之一中。根据对称性，只需检查每个单元的四个相邻单元，如单元 $G_{2,\,4}$的情况所示。

该算法正确性的依据是，它是 cppAP 的直接加速版本：通过使用算法 7 第 14–16 行的“拓扑引导”循环，排除了算法 1（第 3–5 行）嵌套for循环中的大多数不必要比较。

4 性能分析

本节将进行全面的实证分析，比较 cppMM 与其他算法的性能。我们使用 C++.[8] 实现了所有算法。 <time.h> 库函数 clock_gettime() 用于计算运行时间，并在搭载 Apple M3、配备八核和 16 GB 内存的 MacBook Air 上，在 macOS Sequoia 15.3 系统下进行测试。cppRL 和 cppKM 中网格实现所用的哈希表可容纳 $2^{\lceil \log_2 3n \rceil}$ 个条目，并采用外部冲突链；对于非空条目，冲突链的平均长度通常为 0.12。

我们的测试系列所使用的点集，是通过 C++ library（std::default_random_engine 和 std::normal_distribution<double>）中开箱即用的伪随机数生成器生成的。我们分析了单位正方形内均匀分布的点集和截断[9]正态分布的点集。为便于读者理解，现回顾如下内容。

截断正态分布 令 $f_{\mathcal{N}_2}$ 为参数为 (μ, Σ) 的二维正态分布的密度，即对于所有 $x \in \mathbb{R}^2$，

$$f_{\mathcal{N}_2}(x) = \frac{1}{2\pi\sqrt{\det(\Sigma)}} \cdot \exp\Big(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\Big).$$

[7]为简洁起见，本文不在此讨论边界情况。不存在的网格单元定义为空集。[8]审阅时，可从 // www.aau.at/en/isys/ias/research/ 下载代码；论文发表后，代码将以适当的开源许可证发布。[9]由于二维正态分布存在于整个 $\mathbb{R}^2$ 上。

算法 7 $\mathrm{cppMM}(P) \rightarrow < (p_{min}, q_{min})\ \delta >$

要求： $P = \{p_1, \ldots, p_n\} \subset [0,1]^2, n \geq 2$
确保： $\delta = \min\{\|(p,q)\| \mid p, q \in P,\ p \neq q\}$

$\bar{\delta} \leftarrow$ 方程（1）给出的上界
  $w \leftarrow$ 最大值$\{p.x \mid p \in P\} -$ 最小值$\{p.x \mid p \in P\}$
  $h \leftarrow$ 最大值$\{p.y \mid p \in P\} -$ 最小值$\{p.y \mid p \in P\}$
  $\bar{\delta} \leftarrow \bar{\delta} \times \sqrt{w \times h}$ ¯缩放 δ 至实际点云的大小
  $g \leftarrow \lceil 1/\bar{\delta} \rceil$
  $G := g \times g$ 空桶网格
7: </style}对于 $p \in P$ **执行**
    $i \leftarrow \lfloor p.x/\bar{\delta} \rfloor + 1,\ j \leftarrow \lfloor p.y/\bar{\delta} \rfloor + 1$
    $G_{i,j} \leftarrow G_{i,j} \cup \{p\}$
  $\delta \leftarrow \bar{\delta}$
对 $i \leftarrow 1$ **到** g **执行**
  对 $j \leftarrow 1$ **到** g **执行**
    对 $p_1 \in G_{i,\ j}$ **执行**
      对 $p_2 \in G_{i,\ j+1} \cup (G_{i,\ j} \setminus \{p_1\}) \cup G_{i+1,\ j-1} \cup G_{i+1,\ j} \cup G_{i+1,\ j+1}$[7] **执行**
        如果 $\|(p_1, p_2)\| < \delta$ **则**
          $\delta \leftarrow \|(p_1,\ p_2)\|$, 最近点对问题 $\leftarrow (p_1,\ p_2)$
返回 $< \mathrm{cpp}\ \delta >$

其截断版本的密度 $f_{\mathcal{TN}_2}$ 经过设定，使所有样本点都位于单位正方形内，即对于所有 $x \in [0,1]^2$，我们有

$$f_{\mathcal{TN}_2}(x) = \frac{1}{\int_{[0,1]^2} f_{\mathcal{N}_2}(y)dy} \cdot f_{\mathcal{N}_2}(x).$$

请注意，$f_{\mathcal{TN}_2}$ 上方的第一个因子会缩放该函数，使得它与$f_{\mathcal{TN}_2}(x) = 0$的选择相结合后，对于所有 $x \in \mathbb{R}^2 \setminus [0,1]^2$，$f_{\mathcal{TN}_2}$都是一个密度。[10]

我们通过接受-拒绝法从二维截断正态分布 $\mathcal{TN}_2(\mu, \Sigma)$ 中采样。在实现中，我们选择参数 $\mu = (0.5, 0.5)^\top$，使点云位于单位正方形的中心，并选择 $\Sigma = \mathrm{diag}_2(0.04, 0.04)$，以使其与均匀分布有显著差异（参见图 3）。为方便起见，记作 $\sigma = \sqrt{\Sigma_{11}} = \sqrt{\Sigma_{22}}$。请注意，选择 $\sigma = 0.2 = \sqrt{0.04}$ 会使截断正态分布在每个坐标上的方差为 0.036，因此仅略小于 0.04。在这个特殊情况下，由于密度的对称性，其期望与未截断情况下相同。

我们从 $n = 2^{10} = 1024$ 个点开始，将输入规模逐步翻倍，直到达到 $n = 2^{25} = 33\,554\,432$个点[11]。图 3 展示了包含 $n = 1024$的典型问题实例。
表1和表2以及图4和图5展示了针对每个 n值，从十个不同的随机抽取的 CPP问题实例中累积得到的运行时间（秒）。
图 4 和表1考察均匀分布点的情况，而图5和表 2展示了截断正态分布点实验的结果。所有坐标轴均采用对数刻度。

在点均匀分布的情况下（图 4），直到 $n = 2^{21}$，cppAPs 都是实际运行速度最快的算法，cpp MM 紧随其后。只有到了这时，cppAPs 的渐近时间复杂度 $O(n^2)$ 才开始（逐渐）显现，并被 cppMM 超越。

[10]请注意，通常情况下，μ既不是期望，Σ 也不是 $\mathcal{TN}_2$的协方差矩阵（μ，Σ）。[11]由于运行时间过长，我们提前停止了 cppAP，因为它显然已被所有其他算法超越。

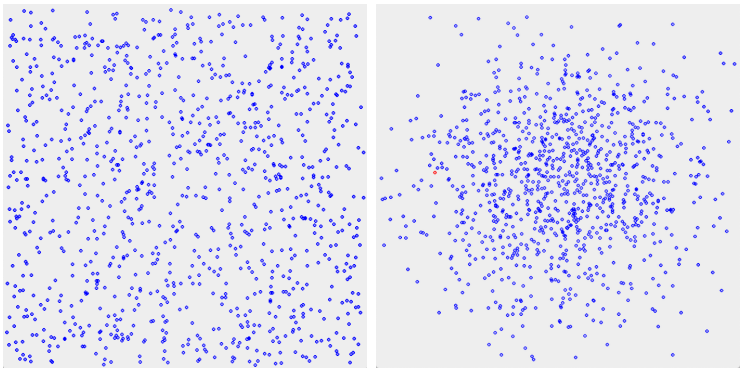

图3：典型测试集，包含1024个伪随机点：均匀分布（左）和截断正态分布，参数为 $\mu = (0.5,0.5)^\top$ 和 $\sigma = 0.2$ （右）。

k	$n = 2^k$	AP	APs	DC	KM	MM	PS	RL
10	1,024	4,711	103	388	529	132	531	172
11	2,048	10,625	112	497	643	168	642	222
12	4,096	42,020	222	1,055	1,268	339	1,283	526
13	8,192	172,488	428	2,271	2,545	666	2,538	1,014
14	16,384	706,161	879	4,840	5,037	1,320	5,058	1,850
15	32,768	2,826,759	1,894	10,754	11,425	2,776	10,857	3,923
16	65,536	11,339,138	3,704	22,701	23,195	5,542	21,104	8,032
17	131,072	45,350,724	7,908	45,989	47,616	11,499	43,420	18,113
18	262,144		17,865	91,294	127,298	25,853	94,709	53,226
19	524,288		43,761	182,940	364,034	59,576	213,286	158,652
20	1,048,576		115,600	369,691	937,741	161,351	499,074	400,559
21	2,097,152		334,576	748,543	2,186,129	421,355	1,200,979	811,589
22	4,194,304		1,092,343	1,532,403	4,772,536	1,016,446	3,013,602	1,791,181
23	8,388,608		3,891,827	3,231,553	9,825,886	2,159,184	8,182,573	4,043,268
24	16,777,216		14,485,259	6,829,380	19,934,263	4,552,791	23,970,723	8,831,863
25	33,554,432		55,736,141	14,728,577	40,933,580	9,533,593	75,796,244	16,009,355

表1：对于每种问题规模 n ，均匀分布点的10个实例中算法1-7的平均运行时间，单位为微秒。

对于 $n > 2^{21}$ ，直到我们实验范围的上限 $n = 2^{25}$，cppMM 的表现最佳。尽管 cppDC 的时间复杂度为 $O(n\log\ n)$，其表现仍与线性时间算法 cppRL、cppKM 和 cppMM 非常接近。

对于截断正态分布点（图5），运行时间测量结果在定性上相似，cppAPs在问题规模不超过 $n = 2^{23}$ 时最快。

实验表明，在两种分布情况下，cppMM的表现都优于其他所有算法，但分别要到 $n = 2^{22}$ 和 2^{24}, 时才超过cppAPs。与总体上速度最快的竞争者cppRL相比，cppMM更易于实现，因为无需借助哈希表来模拟二维网格；cppMM所用的网格大小仅为$O(n)$，因此可以用简单的二维数组实现，从而保证对单个单元和所有单元的访问时间分别为常数时间和线性时间。此外，cppMM是纯确定性算法，而cppRL使用随机近似。

当然，很容易构造出一个最坏情况的问题实例，使 cppMM 的运行时间达到 $\Theta(n^2)$：将大多数点放入 $\bar{\delta}$-网格[12], 中的某一个方格内，这样 cppMM 就不得不比较该方格内所有 $O(n^2)$对点，因此实际上退化为 cppAP。图6展示了这一情形。

接下来，我们研究 cppMM 的平均情况时间复杂度。对于均匀分布点，我们将在第5节证明其时间复杂度为线性；关于截断正态分布点的讨论

[12]如果仍有一些点位于单位正方形的边界附近，如图6所示，那么由于否则网格会相应地细化，参见算法 7中的第2–4行。

k	$n=2k$	AP	APs	DC	KM	MM	PS	RL
10	1,024	2,776	68	257	343	124	346	112
11	2,048	10,962	134	543	694	255	690	225
12	4,096	43,832	271	1,167	1,404	508	1,437	506
13	8,192	176,529	523	2,489	3,112	1,024	2,804	996
14	16,384	708,588	1,031	5,376	6,154	2,082	5,614	2,072
15	32,768	2,838,028	2,016	11,708	12,346	4,149	11,081	3,909
16	65,536	11,346,148	4,201	24,289	23,422	8,321	22,600	8,110
17	131,072	45,399,074	8,721	51,635	48,254	16,864	46,884	17,161
18	262,144		18,577	107,927	127,070	35,729	98,483	48,199
19	524,288		42,053	231,625	371,522	78,180	214,898	149,684
20	1,048,576		102,855	500,531	950,452	199,526	491,998	376,713
21	2,097,152		261,540	1,098,059	2,121,538	511,603	1,132,139	918,130
22	4,194,304		725,205	2,376,220	5,010,292	1,227,949	2,653,479	1,825,625
23	8,388,608		2,300,839	5,179,681	9,617,666	2,669,723	6,756,197	3,799,797
24	16,777,216		7,765,041	11,717,941	21,658,310	5,820,174	17,971,181	7,784,499
25	33,554,432		27,665,930	25,888,110	40,848,654	12,587,923	51,003,423	15,213,277

表 2：截断正态分布点在每种问题规模下进行 10 次实例测试时，算法1-7 的平均运行时间（单位：微秒） n 。

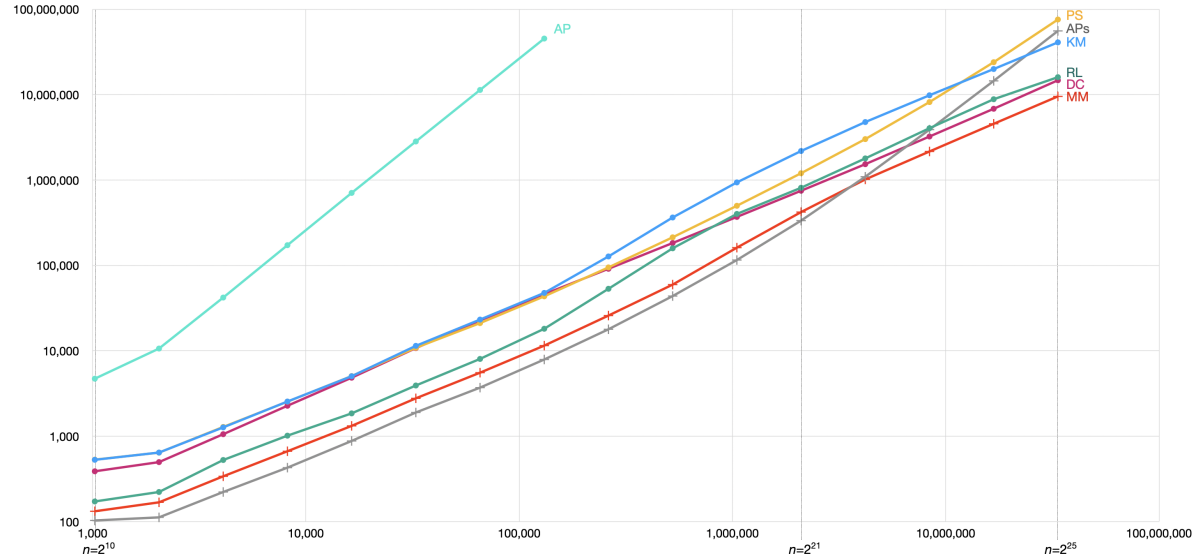

图 4：均匀分布点下算法1-7 的运行时间（单位：微秒，对数刻度）。

将在下文展开。

如图 7 所示，存在一些“无害”的截断正态分布，例如 $\sigma \geq 0.25$；它们与均匀分布足够相似，因此与均匀情形相比，不会显著降低 cppMM 的时间复杂度。然而，图 6 所示的最坏情形很可能也不会在正态分布下出现：考虑图 7 右下方所示的情形 $\sigma = 2^{-5}$ 。虽然这个点云与图 6 一样，大多数点集中在中心，但由于方差较小，边界上出现离群点的概率非常接近于零。具体而言，对于 $\sigma = 0.25$和 $n = 2^{24}$，点落在 $[0, n^{-1/2}] \cup [n^{-1/2}, 1] \times [0, n^{-1/2}] \cup [n^{-1/2}, 1]$[13]中的概率约为 1，而对于 $\sigma = 2^{-5}$，该概率的量级为 10^{-9} 。因此，我们可以相应地调整网格[12]，，所以预计其表现不会比均匀情形差很多。图 8 验证了这一点。

在 Fi 图 8 研究了所考察算法运行时间的稳健性 for

[13]回想一下， $\bar{\delta} \in O\left(n^{-1/2}\right)$ 。

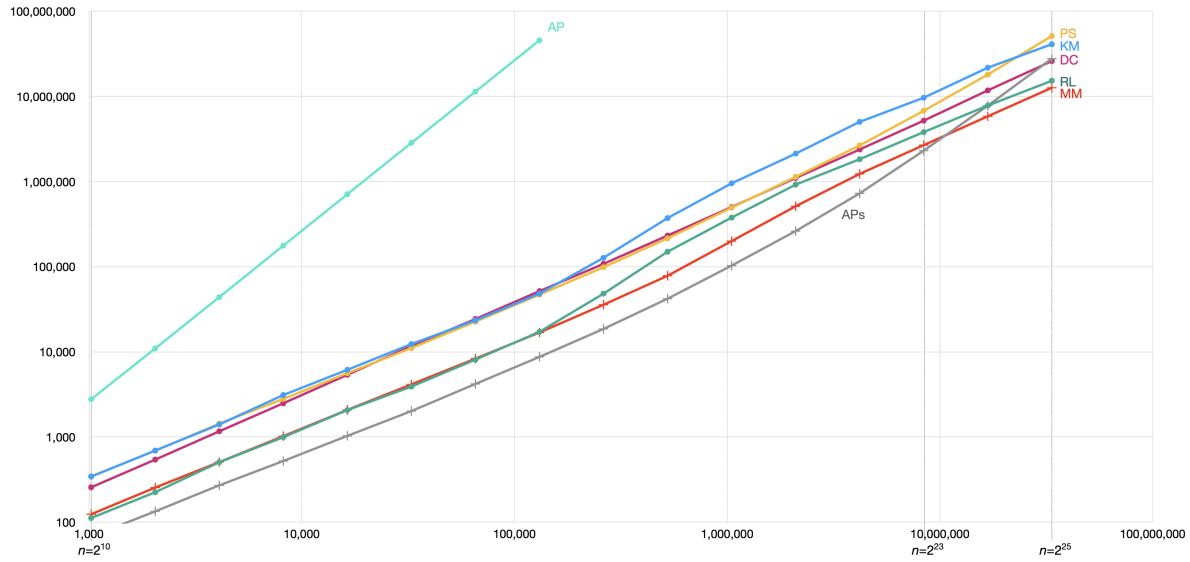

图5：算法1-7处理截断正态分布点的运行时间（μ =(0.5,0.5)$^\top$, $\sigma = 0.2$，微秒，对数刻度）。

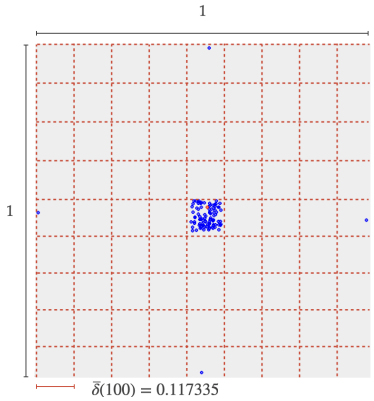

图6：在单位正方形中产生 cppMM 的 $\Theta(n^2)$ 运行时间的 $n = 100$ 点的分布。

递减 σ。对于每个 $\sigma \in \{2^{-k} \mid k \in \{0, \ldots, 6\}\}$ 值，均求解了十个包含 2^{24} 个点的截断正态分布随机点集。就总运行时间而言，我们观察到，尽管 cppMM 在 $\sigma \geq 1/8$ 值时速度最快，但随着 σ 递减，其速度略有下降；不过，它仍然优于 cppDC、cppPS 和 cppKM，且与 cppRL 不相上下。

5 复杂度结果

最后，我们将证明，在均匀分布点集 P的情况下，cppMM 具有线性时间复杂度。

直观地说，构建网格需要对 P 进行一次迭代，因此需要 $O(n)$ 个计算步骤。寻找最小距离的过程如图 2 所示，也就是说，每个点最多与五个相邻网格单元中的配对点进行匹配。对于 n 个点，我们有 $O(n)$ 个网格单元。根据均匀分布假设，每个单元格预计包含 $O(1)$ 个点。因此，整体平均运行时间的量级为 $O(n) + O(n) \times 5 \times O(1) = O(n)$。我们将正式证明这一结果。

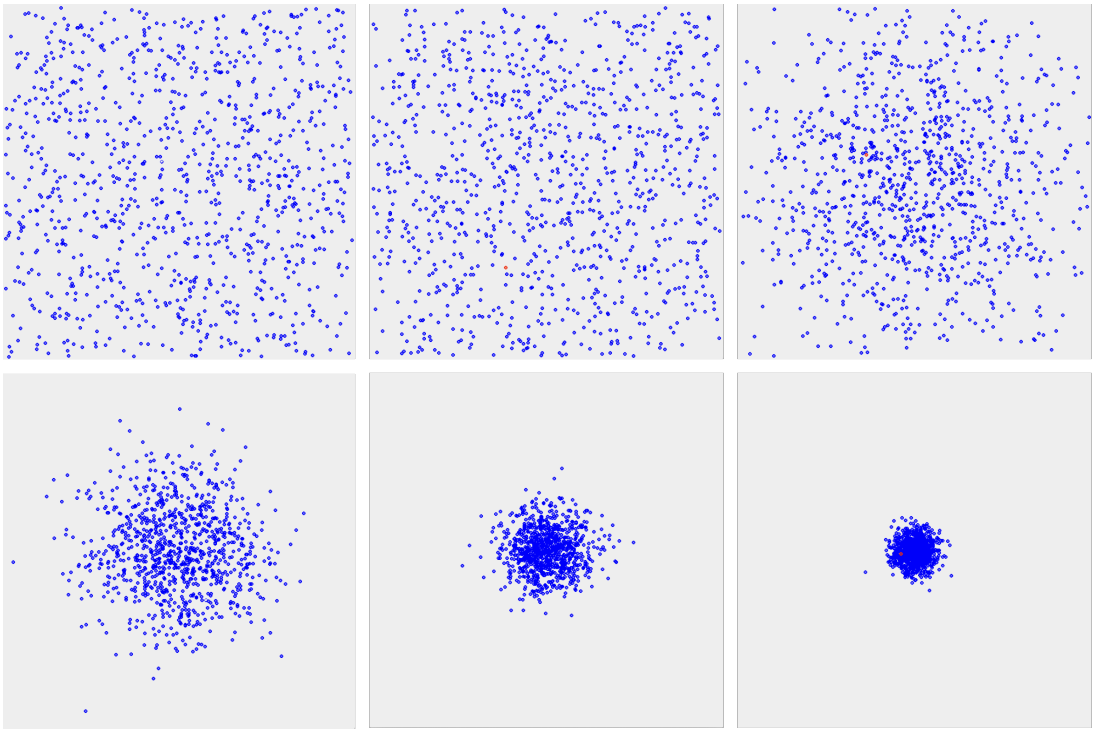

图 7：截断正态分布点集（n $= 1024$），$\mu = (0.5, 0.5)^\top$ 以及 $\sigma = 2^{-k}$（$k \in \{0, \ldots, 5\}$），从图的左上到右下依次排列。

定理5.1. 设概率空间 $(\Omega\ \mathcal{F}, \mathbb{P})$ 包含我们所考虑的所有随机变量。设$n \in \mathbb{N} \setminus \{1\}$ 为单位正方形中的点 $P_1, \ldots, P_n$ 的数量，设 $N = N_1 + N_2$ 为算法 7 的计算步骤总数，其中 N_1 为预计算（第 *1–10* 行）的步骤数，N_2 为比较次数（第 *15–16* 行）。假设随机变量 $P_1, \ldots, P_n$ 相互独立，且均匀分布于单位正方形上。那么算法 7 的计算复杂度为线性量级，即

$$\mathbb{E}[N] \in O(n).$$

证明。 显然，

$$\mathbb{E}[N_1] \in O(n),$$

因为我们遍历了所有 $n = |P|$ 点。第 9 行的时间复杂度为 $O(1)$，因为向桶中添加一个元素$G_{i,\ j}$ 只需常数步。

记作 $M = \lceil 1/\delta \rceil$，并令对所有 $i, j \in \{1, \ldots, M\}$, $N_{i,j}$，网格单元 $G_{i,j}$ 中的点数为。在我们访问的每个网格单元 $G_{i,j}$ 中，将该网格单元内的所有点 $N_{i,j}$ 与该单元中的其他点进行比较，需要 $N_{i,j} \cdot (N_{i,j} - 1)/2$ 次操作；并将它们与四个相邻单元中的所有点进行比较，需要 $N_{i,j} \cdot (N_{i,j+1} + N_{i+1,j-1} + N_{i+1,j} + N_{i+1,j+1})$ 次操作。在网格边界处，所需的比较次数较少。在下式中，第一个双重求和对应于

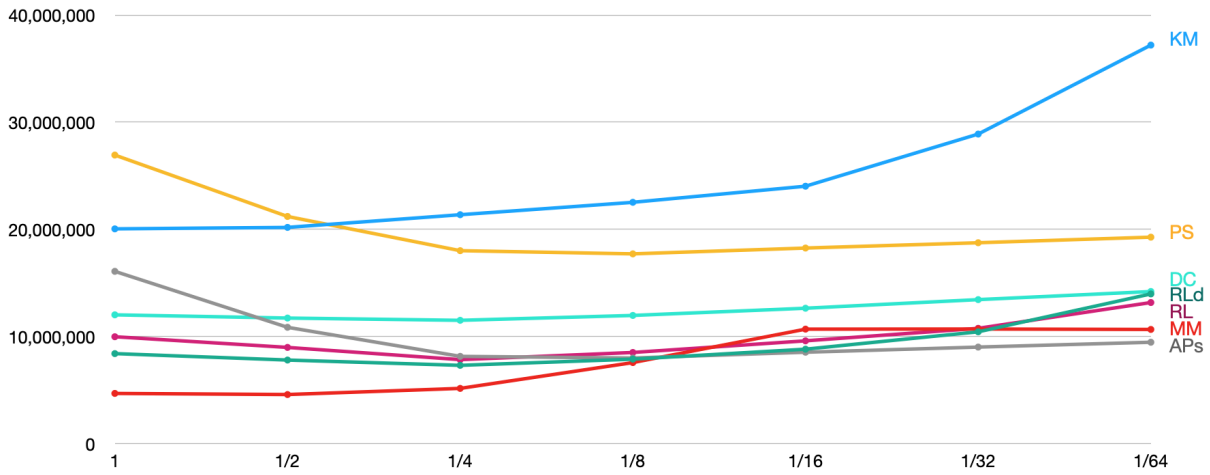

图 8：使用σ 在 1 和 1/64 之间变化、包含 n $= 2^{24} = 16\,777\,216$ 个点的截断正态分布点集时，算法2-7的运行时间（十次重复的耗时，单位为微秒，对数刻度）。

对于内部 网格单元，而单重求和则处理单位正方形的边界。我们有

$$
\begin{aligned}
\mathbb{E}[N_2] = \mathbb{E}\Big[& \sum_{i=1}^{M-1}\sum_{j=2}^{M-1} N_{i,j}\cdot\Big(\frac{1}{2}(N_{i,j}-1) + N_{i,j+1} + N_{i+1,j-1} + N_{i+1,j} + N_{i+1,j+1}\Big) \\
& + \sum_{i=1}^{M-1} N_{i,1}\cdot\Big(\frac{1}{2}(N_{i,1}-1) + N_{i,2} + N_{i+1,1} + N_{i+1,2}\Big) \\
& + \sum_{i=1}^{M-1} N_{i,M}\cdot\Big(\frac{1}{2}(N_{i,M}-1) + N_{i+1,M-1} + N_{i+1,M}\Big) \\
& + \sum_{j=1}^{M-1} N_{M,j}\cdot\Big(\frac{1}{2}(N_{M,j}-1) + N_{M,j+1}\Big) + \frac{1}{2}N_{M,M}(N_{M,M}-1)\Big] \\
= & \sum_{i=1}^{M-1}\sum_{j=2}^{M-1}\Big(\frac{1}{2}(\mathbb{E}[N_{i,j}^2] - \mathbb{E}[N_{i,j}]) + \mathbb{E}[N_{i,j}\cdot N_{i,j+1}] \\
& \qquad + \mathbb{E}[N_{i,j}\cdot N_{i+1,j-1}] + \mathbb{E}[N_{i,j}\cdot N_{i+1,j}] + \mathbb{E}[N_{i,j}\cdot N_{i+1,j+1}]\Big) \\
& + \sum_{i=1}^{M-1}\Big(\frac{1}{2}(\mathbb{E}[N_{i,1}^2] - \mathbb{E}[N_{i,1}]) + \mathbb{E}[N_{i,1}\cdot N_{i,2}] + \mathbb{E}[N_{i,1}\cdot N_{i+1,1}] + \mathbb{E}[N_{i,1}\cdot N_{i+1,2}]\Big) \\
& + \sum_{i=1}^{M-1}\Big(\frac{1}{2}(\mathbb{E}[N_{i,M}^2] - \mathbb{E}[N_{i,M}]) + \mathbb{E}[N_{i,M}\cdot N_{i+1,M-1}] + \mathbb{E}[N_{i,M}\cdot N_{i+1,M}]\Big) \\
& + \sum_{j=1}^{M-1}\Big(\frac{1}{2}(\mathbb{E}[N_{M,j}^2] - \mathbb{E}[N_{M,j}]) + \mathbb{E}[N_{M,j}\cdot N_{M,j+1}]\Big) + \frac{1}{2}(\mathbb{E}[N_{M,M}^2] - \mathbb{E}[N_{M,M}]).
\end{aligned}
\tag{2}
$$

对于所有 $i,j \in \{1,\dots,M\}$，我们有 $N_{i,j} = \sum_{k=1}^{n} 1_{\{P_k \in G_{i,j}\}}$。由于 n 个点在单位正方形上均匀分布，且对于所有 $i,j \in \{1,\dots,M\}$，单元 $G_{i,j}$ 的面积为$\delta^2 \in O(1/n)$，因此我们有

$$
\mathbb{E}[N_{i,j}] = \sum_{k=1}^{n} \mathbb{P}(\{P_k \in G_{i,j}\}) \in n \times O(1/n) = O(1). \tag{3}
$$

接下来，对于所有 $i,j \in \{1,\dots,M\}$,

$$\mathbb{E}[N_{i,j}^2] = \sum_{k=1}^{n}\sum_{\ell=1}^{n}\mathbb{E}[1_{\{P_k\in G_{i,j}\}}\cdot 1_{\{P_\ell\in G_{i,j}\}}] = \sum_{k=1}^{n}\Big(\mathbb{P}(\{P_k\in G_{i,j}\}) + \sum_{\substack{\ell=1\\ \ell\neq k}}^{n}\mathbb{E}[1_{\{P_k\in G_{i,j}\}\cap\{P_\ell\in G_{i,j}\}}]\Big)$$

$$= \mathbb{E}[N_{i,j}] + \sum_{k=1}^{n}\sum_{\substack{\ell=1\\ \ell\neq k}}^{n}\mathbb{P}(\{P_k\in G_{i,j}\}\cap\{P_\ell\in G_{i,j}\}).$$

由于 $\ell\neq k$、 P_ℓ 和 P_k 相互独立，我们有

$$\mathbb{P}(\{P_k\in G_{i,j}\}\cap\{P_\ell\in G_{i,j}\}) = \mathbb{P}(\{P_k\in G_{i,j}\})\cdot\mathbb{P}(\{P_\ell\in G_{i,j}\}) \in O(1/n)\times O(1/n).$$

因此,

$$\mathbb{E}[N_{i,j}^2] \in O(1) + n\times(n-1)\times O(1/n)\times O(1/n) = O(1). \tag{4}$$

式 (2) 中出现的最后一类加项是 $\mathbb{E}[N_{i,j}\cdot N_{\alpha,\beta}]$ 与 $(i,j)\neq(\alpha,\beta)$。对于所有满足 $(i,j)\neq(\alpha,\beta)$ 的$i,j,\alpha,\beta\in\{1,\dots,M\}$，我们有

$$\begin{aligned}\mathbb{E}[N_{i,j}\cdot N_{\alpha,\beta}] &= \sum_{k=1}^{n}\sum_{\ell=1}^{n}\mathbb{E}[1_{\{P_k\in G_{i,j}\}}\cdot 1_{\{P_\ell\in G_{\alpha,\beta}\}}]\\ &= \sum_{k=1}^{n}\Big(\mathbb{E}[1_{\{P_k\in G_{i,j}\}\cap\{P_k\in G_{\alpha,\beta}\}}] + \sum_{\substack{\ell=1\\ \ell\neq k}}^{n}\mathbb{E}[1_{\{P_k\in G_{i,j}\}\cap\{P_\ell\in G_{\alpha,\beta}\}}]\Big)\\ &= 0 + \sum_{k=1}^{n}\sum_{\substack{\ell=1\\ \ell\neq k}}^{n}\mathbb{P}(\{P_k\in G_{i,j}\}\cap\{P_\ell\in G_{\alpha,\beta}\}).\end{aligned}$$

这确保了

$$\mathbb{E}[N_{i,j}\cdot N_{\alpha,\beta}] \in n\times(n-1)\times O(1/n)\times O(1/n) = O(1). \tag{5}$$

¯回忆一下 $M = 1/\lceil 1/\delta\rceil \in O(\sqrt{n})$，并将 (2) 与 (3)、(4) 和 (5) 结合，可得

$$\mathbb{E}[N_2] \in O(\sqrt{n})\times O(\sqrt{n})\times O(1) = O(n).$$

至此证明完成。 □

6 结论

虽然从理论角度来看，CPP 问题的经典解法无法改进，但我们发现仍有充分的机会提升实际算法运行时间。我们基于数学最优装填理论的理论发现，提出了两种确定性且非递归的新算法。对于 cppMM 和均匀分布点，我们证明了$O(n)$ 时间复杂度。我们将其与经典方法进行了比较研究。研究结果汇总于表 3。

我们观察到，就平均情形渐近时间复杂度而言，cppMM（针对均匀分布点集）、cppRL 和 cppKM 处于同一水平，其次是 cppDC 和 cppPS，最后是 cppAPs 和 cppAP。然而，如图 1 所示，直到非常高的

[14] 已针对均匀分布点证明，对于截断正态分布点也可观察到这一点。

表 3：算法1-7总结。

算法	No.	易用性 实现 I=简单 – III=困难	平均 案例耗时 复杂度	最坏 案例耗时 复杂度	敏感 to σ	实测 运行时间 （UD、ND，含 $\sigma \geq 1/4$） I=低 – III=高
cppAP	1	I	$O(n^2)$	$O(n^2)$	no	III
cppDC	2	III	$O(n$对数 $n)$	$O(n$对数 $n)$	no	II
cppPS	3	II	$O(n$对数 $n)$	$O(n$日志 $n)$	no	II
cppRL	4	II	$O(n)$	$O(n)$	no	II
cppKM	5	III	$O(n)$	$O(n)$	从经验来看：是	II
cppAPs	6	I	$O(n^2)$	$O(n^2)$	no	I–II
cppMM	7	II	$O(n)$[14]	$O(n^2)$	在相关程度上并非如此	I

n，cppAPs 的复杂度都更接近于 $O(n)$，而不是 $O(n^2)$，并且只有在 n 取非常大的值时才会缓慢接近渐近值。

就实测运行时间而言，cppMM 优于其他算法，但 cppAPs 除外；在相当大的问题规模范围内，cppAPs 绝对占优。

大多数算法对点集的选择并不敏感。对于 cppMM，实验结果也证实了这一点；不过，也存在一些人为构造的最坏情况示例，在这些示例中，它需要 $O(n^2)$ 的时间。

就实现难易度而言，我们根据所使用的代码行数以及所需的辅助数据结构或算法，将这些算法分为 I 至 III 类。I 类包含两个极其简单的“嵌套循环”算法 cppAP 和 cppAPs；cppPS、cppRL 和 cppMM 属于 II 类，而 cppDC 和 cppKM 则具有最为复杂的实现。

综合来看，我们的算法至少与其他竞争算法一样快，同时实现起来较为容易（cppMM），甚至极其容易（cppAPs）。当然，尽管我们已尽最大努力，仍无法排除测量结果受到实现中某些个别缺陷影响的可能性（源代码见 https://www.aau.at/en/isys/ias/research/）。如有这方面的任何信息，我们将不胜感激。

声明

资助／利益冲突

本研究未获得外部资助。作者声明不存在与本文内容相关的利益冲突。

参考文献

[1] Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clifford Stein。《算法导论》。麻省理工学院出版社，马萨诸塞州剑桥；英格兰伦敦， 2nd 版，2003年。

[2] Klaus Hinrichs、JugNievergelt 和 Peter Schorn。平面扫描优雅地解决了最近点对问题。*Inf. Process. Lett.*，26(5):255–261，1988年1月。ISSN 0020-0190。doi: 10.1016/ 0020-0190(88)90150-0。URL https://doi.org/10.1016/0020-0190(88)90150-0。

[3] S. Khuller 和 Y.Matias。用于最近点对问题的简单随机筛选算法。《信息与计算》，118：34–37，1995年。

[4] Richard J. Lipton。Rabin抛硬币，2009年。https://rjlipton.wpcomstaging.com/2009/03/01/rabin- flips-a-coin/（最后访问：2021年12月10日）。

[5] Franco P. Preparata 和 Michael I. Shamos。计算几何：导论。Springer-Verlag，柏林、海德堡，1985年。ISBN 0387961313。[6] Michael O. Rabin。概率算法。载于 J. F. Traub 编，算法与复杂性：新方向与最新成果。由卡内基梅隆大学计算机科学系于*1976*年*4*月*7*日至*9*日举办的算法与复杂性新方向与最新成果研讨会论文集，第21–38页，纽约、纽约州；旧金山、加利福尼亚州；英格兰伦敦，1976年。学术出版社。ISBN 9780126975406。[7] Michael Ian Shamos 和 Dan Hoey。几何相交问题。载于第*17*届计算机科学基础年度研讨会论文集，SFCS '76，第208–215页，美国，1976年。IEEE计算机学会。doi：10.1109/SFCS.1976.16。URL https://doi.org/10. 1109/SFCS.1976.16。[8] P. G. Szabó、T. Csendes、L. G. Casado 和 I. García。正方形内等圆装填 I——问题设定与最优解界。载于 Franco Giannessi、Panos Pardalos 和 Tamás Rapcsák 编，优化理论：*Mátraháza*的最新进展，第191–206页。Springer美国，波士顿，马萨诸塞州，2001年。ISBN 978-1-4613-0295-7。doi：10.1007/978-1-4613-0295-7_14。URL https://doi.org/10.1007/978-1-4613-0295-7_14。