k-d $tree$在传统OI数据结构题中的应用

任之洲

绍兴市第一中学

2015 年 2 月 10 日

应用背景

应用背景

- *k-d tree*（k-dimensional tree）是一种基于空间分割的树形数据结构。

应用背景

- *k-d tree*（k-dimensional tree）是一种基于空间分割的树形数据结构。
- 主要用于多维空间中点集的数据检索，在OI中也可以解决一些传统数据结构题。

SIFT

SIFT

- SIFT（Scale-invariant feature transform）是一种计算机视觉的算法。

SIFT

- SIFT（Scale-invariant feature transform）是一种计算机视觉的算法。
- 在这个算法中，数字图像将被表示成一系列与影像的大小和旋转无关的关键点特征向量。

SIFT

- SIFT（Scale-invariant feature transform）是一种计算机视觉的算法。
- 在这个算法中，数字图像将被表示成一系列与影像的大小和旋转无关的关键点特征向量。
- 采用关键点特征向量的欧式距离来作为两幅图像中关键点的相似性判定度量。

SIFT

- SIFT（Scale-invariant feature transform）是一种计算机视觉的算法。
- 在这个算法中，数字图像将被表示成一系列与影像的大小和旋转无关的关键点特征向量。
- 采用关键点特征向量的欧式距离来作为两幅图像中关键点的相似性判定度量。


SIFT

SIFT

- 特征点的匹配需要进行高维矢量间的近似检索。

SIFT

- 特征点的匹配需要进行高维矢量间的近似检索。
- *k-d tree*就是其中一种高维空间检索的算法。

k-d tree

k-d tree

- *k-d tree*的结构是一棵二叉搜索树，且树深度是平衡的。

k-d tree

- *k-d tree*的结构是一棵二叉搜索树，且树深度是平衡的。
- *k-d tree*中的每棵子树表示一个空间范围，左右子树表示的空间是不交的。

k-d tree

- *k-d tree*的结构是一棵二叉搜索树，且树深度是平衡的。
- *k-d tree*中的每棵子树表示一个空间范围，左右子树表示的空间是不交的。
- 需要支持的主要操作有点的插入、删除，最近点查询，范围内点计数。

构树

构树

- *k-d tree*的构造是基于对K维空间的分割，每次选取其中一维坐标的中位数作为划分界线。

构树

- *k-d tree*的构造是基于对K维空间的分割，每次选取其中一维坐标的中位数作为划分界线。
- 为了更直观地解释*k-d tree*的结构，先考虑二维的情况。

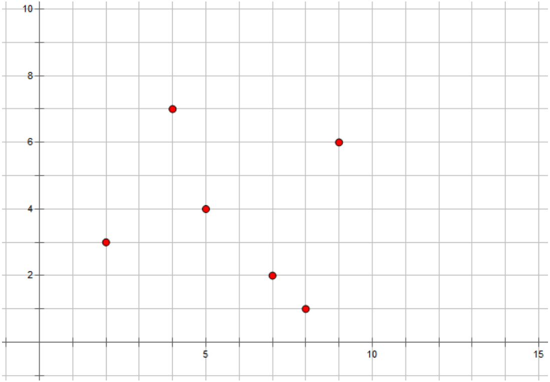
构树

构树

- 可以循环地以每维坐标为划分依据，把中位数所在点作为该子树的根。

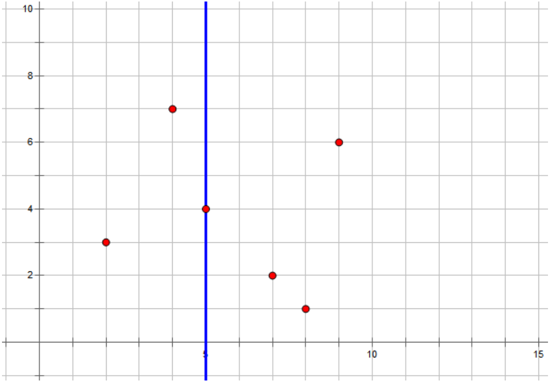
构树

- 可以循环地以每维坐标为划分依据，把中位数所在点作为该子树的根。



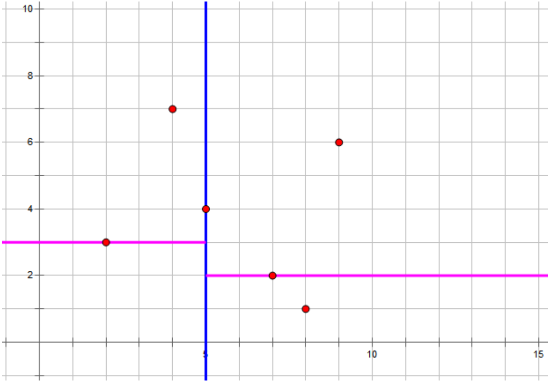
构树

- 可以循环地以每维坐标为划分依据，把中位数所在点作为该子树的根。



构树

- 可以循环地以每维坐标为划分依据，把中位数所在点作为该子树的根。



构树

- 可以循环地以每维坐标为划分依据，把中位数所在点作为该子树的根。

构树

构树

- 三维以上的*k-d tree*构造也可以同样得出。

构树

- 三维以上的*k-d tree*构造也可以同样得出。
- 对于中位数的查找，有许多$O(n)$的算法，也可以直接调用STL的nth_element()。

构树

- 三维以上的k-d $tree$构造也可以同样得出。
- 对于中位数的查找，有许多$O(n)$的算法，也可以直接调用STL的nth_element()。
- 所以建树的复杂度$T(n) = O(n) + 2T(\frac{n}{2}) = O(n \log n)$，且树深度是$O(\log n)$的。

构树

- 三维以上的*k*-*d tree*构造也可以同样得出。
- 对于中位数的查找，有许多$O(n)$的算法，也可以直接调用STL的nth_element()。
- 所以建树的复杂度$T(n) = O(n) + 2T(\frac{n}{2}) = O(n \log n)$，且树深度是$O(\log n)$的。
- 当循环选择划分维度时，*k*-*d tree*的形态就如同离散化后的四分树。

构树

构树

- 循环选择划分维度并不是唯一的方法，可以每次选择方差最大的维度进行划分。

构树

- 循环选择划分维度并不是唯一的方法，可以每次选择方差最大的维度进行划分。
- 但是少量的噪点就可以影响方差。

构树

- 循环选择划分维度并不是唯一的方法，可以每次选择方差最大的维度进行划分。
- 但是少量的噪点就可以影响方差。
- 在一些和“具体距离”无关的问题中，可以把坐标进行离散化处理后再求方差。

插入

插入

- 由于*k-d tree*是二叉搜索树结构，所以插入新的节点可以类比BST的插入进行。

插入

- 由于*k*-*d tree*是二叉搜索树结构，所以插入新的节点可以类比BST的插入进行。
- 随着新的节点的插入，*k*-*d tree*的树深度将不再平衡。

插入

- 由于*k-d tree*是二叉搜索树结构，所以插入新的节点可以类比BST的插入进行。
- 随着新的节点的插入，*k-d tree*的树深度将不再平衡。
- 有一种很经典的解决方法，就是用替罪羊树的方式，在不平衡时重构*k-d tree*的子树。

插入

- 由于k-d $tree$是二叉搜索树结构，所以插入新的节点可以类比BST的插入进行。
- 随着新的节点的插入，k-d $tree$的树深度将不再平衡。
- 有一种很经典的解决方法，就是用替罪羊树的方式，在不平衡时重构k-d $tree$的子树。
- 均摊时间复杂度$O(\log n)$。

删除

删除

- 由于*k-d tree*需要保证结构的空间划分性质，所以不能直接使用一些平衡树的删除方式。

删除

- 由于*k-d tree*需要保证结构的空间划分性质，所以不能直接使用一些平衡树的删除方式。
- 可以在删除的节点上保留一个已删除标记，并在它的祖先中抹除它的信息，在子树进行重构时再真正地删除它。

删除

- 由于k-d $tree$需要保证结构的空间划分性质，所以不能直接使用一些平衡树的删除方式。
- 可以在删除的节点上保留一个已删除标记，并在它的祖先中抹除它的信息，在子树进行重构时再真正地删除它。
- 时间复杂度$O(\log n)$。

最近点查询

最近点查询

- 在最近点查询问题中，使用*k-d tree*将可以大幅度降低运算量。

最近点查询

- 在最近点查询问题中，使用*k-d tree*将可以大幅度降低运算量。
- 从*k-d tree*的根开始进行dfs遍历，每个子树表示一个空间范围，每次选择较近的子树优先搜索。

最近点查询

- 在最近点查询问题中，使用*k-d tree*将可以大幅度降低运算量。
- 从*k-d tree*的根开始进行dfs遍历，每个子树表示一个空间范围，每次选择较近的子树优先搜索。
- 当这个子树表示的空间范围离询问点的最近距离也不能更新答案时就回溯（最优性剪枝）。

最近点查询

- 在最近点查询问题中，使用*k-d tree*将可以大幅度降低运算量。
- 从*k-d tree*的根开始进行dfs遍历，每个子树表示一个空间范围，每次选择较近的子树优先搜索。
- 当这个子树表示的空间范围离询问点的最近距离也不能更新答案时就回溯（最优性剪枝）。
- 当$n \gg 2^k$时，对随机数据的询问的期望效率为$O(\log n)$。

BBF算法

BBF算法

- 在特征向量的匹配中，空间维度将会达到128维甚至更大，k-d $tree$的效率将会退化到接近$O(n)$。

BBF算法

- 在特征向量的匹配中，空间维度将会达到128维甚至更大，*k-d tree*的效率将会退化到接近$O(n)$。
- BBF（Best Bin First）算法可以将*k-d tree*扩展到高维数据集上，在较优的时间内得出一个近似的“最”优解或近似的K邻近。

BBF算法

BBF算法

- 该算法采用best first search思想。

BBF算法

- 该算法采用best first search思想。
- 因为*k*-*d tree*的子树表示一个空间范围，所以可以用询问点到那个空间范围的最短距离作为估价，维护一个优先队列。

BBF算法

- 该算法采用best first search思想。
- 因为*k-d tree*的子树表示一个空间范围，所以可以用询问点到那个空间范围的最短距离作为估价，维护一个优先队列。
- 每次从优先队列中选取一个节点，贪心从该点走到某个叶子节点，将沿途的可能会更新答案的点放入优先队列。

BBF算法

- 该算法采用best first search思想。
- 因为*k-d tree*的子树表示一个空间范围，所以可以用询问点到那个空间范围的最短距离作为估价，维护一个优先队列。
- 每次从优先队列中选取一个节点，贪心从该点走到某个叶子节点，将沿途的可能会更新答案的点放入优先队列。
- 由于只是求近似解，所以可以根据需要设定阈值，当遍历的点数超过阈值后就认为得到的解近似最优。

BBF算法

- 该算法采用best first search思想。
- 因为*k-d tree*的子树表示一个空间范围，所以可以用询问点到那个空间范围的最短距离作为估价，维护一个优先队列。
- 每次从优先队列中选取一个节点，贪心从该点走到某个叶子节点，将沿途的可能会更新答案的点放入优先队列。
- 由于只是求近似解，所以可以根据需要设定阈值，当遍历的点数超过阈值后就认为得到的解近似最优。
- 当优先队列中没有可更新答案的点，或遍历达到阈值时，结束该算法。

OI中的*k-d tree*

OI中的*k-d tree*

- 在OI中，很少会需要查找一个点的K邻近的近似解，更多的可能是在一定空间范围内的计数问题。

OI中的*k-d tree*

- 在OI中，很少会需要查找一个点的K邻近的近似解，更多的可能是在一定空间范围内的计数问题。
- 而且空间维度一般都在2或3，*k-d tree*的运行效率是很可观的。

范围计数

范围计数

- 在*k-d tree*中对一个平行坐标轴的空间范围（axis-parallel range）内点集的求和、求最值的操作很简单。

范围计数

- 在*k-d tree*中对一个平行坐标轴的空间范围（axis-parallel range）内点集的求和、求最值的操作很简单。
- 从根开始遍历，若当前点表示的空间完全在询问范围内，则直接取用该节点上的子树相关信息，若当前点表示的空间与询问范围没有交，则直接回溯，否则将该节点单独计算，递归继续进入子树计算。

范围计数

- 在*k-d tree*中对一个平行坐标轴的空间范围（axis-parallel range）内点集的求和、求最值的操作很简单。
- 从根开始遍历，若当前点表示的空间完全在询问范围内，则直接取用该节点上的子树相关信息，若当前点表示的空间与询问范围没有交，则直接回溯，否则将该节点单独计算，递归继续进入子树计算。
- 考虑该算法的运行效率，造成进入左右子树的条件只有该子树和询问范围相交。

范围计数

- 在*k-d tree*中对一个平行坐标轴的空间范围（axis-parallel range）内点集的求和、求最值的操作很简单。
- 从根开始遍历，若当前点表示的空间完全在询问范围内，则直接取用该节点上的子树相关信息，若当前点表示的空间与询问范围没有交，则直接回溯，否则将该节点单独计算，递归继续进入子树计算。
- 考虑该算法的运行效率，造成进入左右子树的条件只有该子树和询问范围相交。
- 把每一维分开考虑，$T(n) = O(2^k) + 2^{k-1}T(\frac{n}{2^k}) = O(n^{1-\frac{1}{k}})$。

范围计数

- 在*k-d tree*中对一个平行坐标轴的空间范围（axis-parallel range）内点集的求和、求最值的操作很简单。
- 从根开始遍历，若当前点表示的空间完全在询问范围内，则直接取用该节点上的子树相关信息，若当前点表示的空间与询问范围没有交，则直接回溯，否则将该节点单独计算，递归继续进入子树计算。
- 考虑该算法的运行效率，造成进入左右子树的条件只有该子树和询问范围相交。
- 把每一维分开考虑，$T(n) = O(2^k) + 2^{k-1}T(\frac{n}{2^k}) = O(n^{1-\frac{1}{k}})$。
- 由此可得，时间复杂度为$O(kn^{1-\frac{1}{k}})$。

范围计数

范围计数

- 除了解决平行于坐标轴的范围约束，*k-d tree*还可以用于求圆内、球内的点集信息。

范围计数

- 除了解决平行于坐标轴的范围约束，*k-d tree*还可以用于求圆内、球内的点集信息。
- 在求矩形内信息时，*k-d tree*的运行效率有时优于树套树，且空间开销为$O(n)$。

合并

合并

- 由于k-d $tree$的结构比较特殊，所以不能像四分树、线段树那样合并。

合并

- 由于*k-d tree*的结构比较特殊，所以不能像四分树、线段树那样合并。
- 可以采取类似通解的启发式合并策略，把较小的树拆分，逐一插入较大的树中。

合并

- 由于*k-d tree*的结构比较特殊，所以不能像四分树、线段树那样合并。
- 可以采取类似通解的启发式合并策略，把较小的树拆分，逐一插入较大的树中。
- 均摊复杂度$O(n \log^2 n)$。

可持久化

可持久化

- *k-d tree*是简单的BST结构，所以可持久化比较容易完成。

可持久化

- *k-d tree*是简单的BST结构，所以可持久化比较容易完成。
- 应对插入新点而导致的重构操作，可以参考替罪羊树的可持久化来解决。

卡常数

卡常数

- 给出一个n个点的三维点集，m次操作，操作有两种：

卡常数

- 给出一个n个点的三维点集，m次操作，操作有两种：
- 修改一个点的坐标。

卡常数

- 给出一个n个点的三维点集，m次操作，操作有两种：
- 修改一个点的坐标。
- 给定球心和半径，查找恰好在球面上的一个点。

卡常数

- 给出一个n个点的三维点集，m次操作，操作有两种：
- 修改一个点的坐标。
- 给定球心和半径，查找恰好在球面上的一个点。
- $n, m \leq 65536$，坐标随机，强制在线。

卡常数

卡常数

- 由于坐标随机，求恰好在球面上的点，近似于球内点计数。

卡常数

- 由于坐标随机，求恰好在球面上的点，近似于球内点计数。
- 在树中遍历时需要通过球面和长方体判交来判断，从而带来了极大的常数问题。

卡常数

- 由于坐标随机，求恰好在球面上的点，近似于球内点计数。
- 在树中遍历时需要通过球面和长方体判交来判断，从而带来了极大的常数问题。
- 复杂度$O(n^{\frac{5}{3}})$。

卡常数

卡常数

- 看上去已经比较靠谱了，但是为什么还是过不去呢？

卡常数

- 看上去已经比较靠谱了，但是为什么还是过不去呢？
- 主要是球面和长方体判交部分常数较大。

卡常数

- 看上去已经比较靠谱了，但是为什么还是过不去呢?
- 主要是球面和长方体判交部分常数较大。
- 如何快速缩小常数? 注意到下面这个:

卡常数

- 看上去已经比较靠谱了，但是为什么还是过不去呢？
- 主要是球面和长方体判交部分常数较大。
- 如何快速缩小常数？注意到下面这个：

为尽可能减小精度误差，建议 C/C++ 使用long double类型、Pascal 使用 EXTENDED 类型存储实数。

卡常数

- 看上去已经比较靠谱了，但是为什么还是过不去呢？
- 主要是球面和长方体判交部分常数较大。
- 如何快速缩小常数？注意到下面这个：

> 为尽可能减小精度误差，建议 C/C++ 使用long double类型、Pascal 使用 EXTENDED 类型存储实数。

- 只要无视这条温馨提示就可以贴线低飞了。

A simple rmq problem

A simple rmq problem

- 给出一个长度为n的序列a。

A simple rmq problem

- 给出一个长度为n的序列a。
- m次询问，求$[l, r]$之间最大的只出现一次的数。

A simple rmq problem

- 给出一个长度为n的序列a。
- m次询问，求$[l,r]$之间最大的只出现一次的数。
- $n \leq 10^5$，$m \leq 2 * 10^5$，强制在线。

A simple rmq problem

A simple rmq problem

- 设p_i为a_i前一次出现的位置，q_i为a_i后一次出现的位置。

A simple rmq problem

- 设p_i为a_i前一次出现的位置，q_i为a_i后一次出现的位置。
- 那么询问可以转化为$Max(a_i)$，$i \in [l, r]$，$p_i < l$，$q_i > r$。

A simple rmq problem

- 设p_i为a_i前一次出现的位置，q_i为a_i后一次出现的位置。
- 那么询问可以转化为$Max(a_i)$，$i \in [l, r]$，$p_i < l$，$q_i > r$。
- 把序列中的每个数看作三维空间中的点，用*k-d tree*维护，复杂度$O(mn^{\frac{2}{3}})$。

A simple rmq problem

- 设p_i为a_i前一次出现的位置，q_i为a_i后一次出现的位置。
- 那么询问可以转化为$Max(a_i)$，$i \in [l, r]$，$p_i < l$，$q_i > r$。
- 把序列中的每个数看作三维空间中的点，用$k\text{-}d\ tree$维护，复杂度$O(mn^{\frac{2}{3}})$。
- 虽然时间复杂度较高，但是实际运行时间仅为传统树套树算法的一半左右，且空间复杂度降到了$O(n)$。

A simple rmq problem

A simple rmq problem

- 假如不强制在线，可以把询问维护成k-d $tree$。

A simple rmq problem

- 假如不强制在线，可以把询问维护成*k-d tree*。
- 限制可以被写成这样$p_i < l \le i$，$i \le r < q_i$，这样维护的是二维点集。

A simple rmq problem

- 假如不强制在线，可以把询问维护成k-d $tree$。
- 限制可以被写成这样$p_i < l \le i$，$i \le r < q_i$，这样维护的是二维点集。
- 时间复杂度$O(m\sqrt{n})$。

带插入区间K小值

带插入区间K小值

- 在线支持插入，修改，区间K小值询问。

带插入区间K小值

- 在线支持插入，修改，区间K小值询问。
- 原序列长度$\leq$ 35000，插入个数$\leq$ 35000，修改个数$\leq$ 70000，查询个数$\leq$ 70000。

带插入区间K小值

带插入区间K小值

- 先考虑如何用*k-d tree*求区间K小值。

带插入区间K小值

- 先考虑如何用k-d $tree$求区间K小值。
- 把序列中的每个数a_i，看作一个坐标为(i, a_i)的点。

带插入区间K小值

- 先考虑如何用*k-d tree*求区间K小值。
- 把序列中的每个数a_i，看作一个坐标为(i, a_i)的点。
- 二分答案，求$i \in [l, r]$，$a_i \in [0, ans]$内的点数。

带插入区间K小值

- 先考虑如何用*k-d tree*求区间K小值。
- 把序列中的每个数a_i，看作一个坐标为(i, a_i)的点。
- 二分答案，求$i \in [l, r]$，$a_i \in [0, ans]$内的点数。
- 时间复杂度为$O(\sqrt{n}\log n)$。

带插入区间K小值

带插入区间K小值

- 修改操作可以看作先删除原位置再插入。

带插入区间K小值

- 修改操作可以看作先删除原位置再插入。
- 考虑支持插入操作，在插入新的位置时需要维护序列的顺序。

带插入区间K小值

- 修改操作可以看作先删除原位置再插入。
- 考虑支持插入操作，在插入新的位置时需要维护序列的顺序。
- 序列顺序维护问题是经典问题，在陈立杰2013年的国家集训队论文中有提及。

带插入区间K小值

- 修改操作可以看作先删除原位置再插入。
- 考虑支持插入操作，在插入新的位置时需要维护序列的顺序。
- 序列顺序维护问题是经典问题，在陈立杰2013年的国家集训队论文中有提及。
- 由于这里的序列维护不再是瓶颈，所以可以选用较为简单的$O(\log n)$做法。

带插入区间K小值

带插入区间K小值

- 对序列维护一棵替罪羊树，用于寻找修改和插入的具体位置。

带插入区间K小值

- 对序列维护一棵替罪羊树，用于寻找修改和插入的具体位置。
- 通过调整替罪羊树的参数，可以把树高控制在32以内。

带插入区间K小值

- 对序列维护一棵替罪羊树，用于寻找修改和插入的具体位置。
- 通过调整替罪羊树的参数，可以把树高控制在32以内。
- 对于每个点，维护一个二进制标号tag_i，通过比较标号tag_i来确定k-d $tree$的询问区间。

带插入区间K小值

- 对序列维护一棵替罪羊树，用于寻找修改和插入的具体位置。
- 通过调整替罪羊树的参数，可以把树高控制在32以内。
- 对于每个点，维护一个二进制标号tag_i，通过比较标号tag_i来确定*k-d tree*的询问区间。
- *k-d tree*中则用最靠左的点和最靠右的点来描述那一维坐标的范围，这样就不需要因为标号的更改而造成*k-d tree*的形态变化。

带插入区间K小值

带插入区间K小值

- 可惜的是，这个算法实现了一下跑得没有暴力快。

带插入区间K小值

- 可惜的是，这个算法实现了一下跑得没有暴力快。
- 有一个简单的优化。

带插入区间K小值

- 可惜的是，这个算法实现了一下跑得没有暴力快。
- 有一个简单的优化。
- 对于每一次询问，在二分答案时，变动的只有权值那一维坐标。

带插入区间K小值

- 可惜的是，这个算法实现了一下跑得没有暴力快。
- 有一个简单的优化。
- 对于每一次询问，在二分答案时，变动的只有权值那一维坐标。
- 所以可以预先找好$i \in [l, r]$，$a_i \in [0, +\infty]$的子树和路径上单独的点，这样来减少一些多余的计算。

总结

总结

- *k-d tree*适用于低维数据集的维护。

总结

- *k-d tree*适用于低维数据集的维护。
- 在OI中运用*k-d tree*时，需要先把信息构造成坐标系上的点集。

总结

- k-d $tree$适用于低维数据集的维护。
- 在OI中运用k-d $tree$时，需要先把信息构造成坐标系上的点集。
- k-d $tree$是BST结构，所以可以支持一些树套树不能维护的标记。

致谢

- 感谢ccf提供这次营员交流的机会。
- 感谢绍兴一中陈合力老师、董烨华老师长期的付出。
- 感谢俞鼎力、董宏华、何奇正同学对我的帮助。
- 感谢大家的细心聆听。

参考资料

- Wikipedia, k-d tree
- Wikipedia, Best Bin First
- J_Outsider的博客，k-d tree的优化查找算法BBF
- 陈立杰，重量平衡树和后缀平衡树在信息学奥赛中的应用，国家集训队2013论文