DP优化杂谈

——斜率优化升级

By nodgd

上次谈到的一些斜率优化不能解决的问题之一：

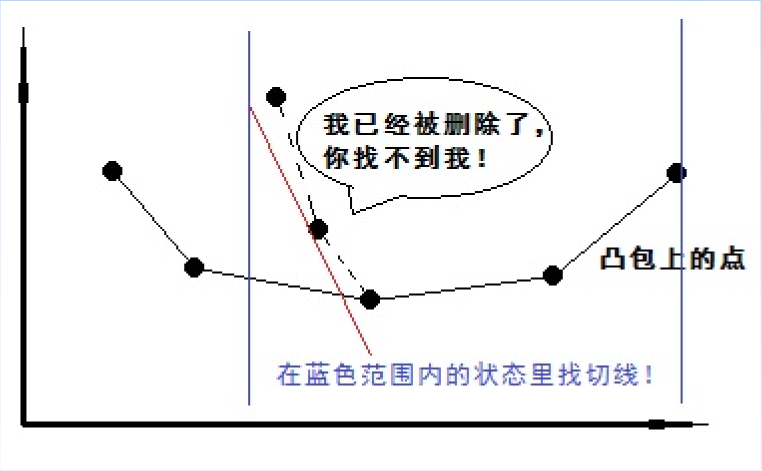
·状态转移$j \to i$中j的范围的左端点必须是1，右端点必须使随着i增大逐渐向右移动的。

·然而很多时候我们状态转移方程并不满足这个条件。例如这个状态转移方程：($NOI2014\ Day2\ T3$ 的弱化版)

$$f[i] = f[j] + p[i] * (dis[i] - dis[j]) + q[i], left[i] \le j \le i - 1$$

其中$p[i], q[i], dis[i], left[i]$是一个输入的值，只有$dis[i]$是单调递增的，其他均不单调。

·这时，显然朴素的斜率优化是不能够解决了，因为它可能会遇到这样的情况：



回顾一下凸包的一个性质——可加不可减

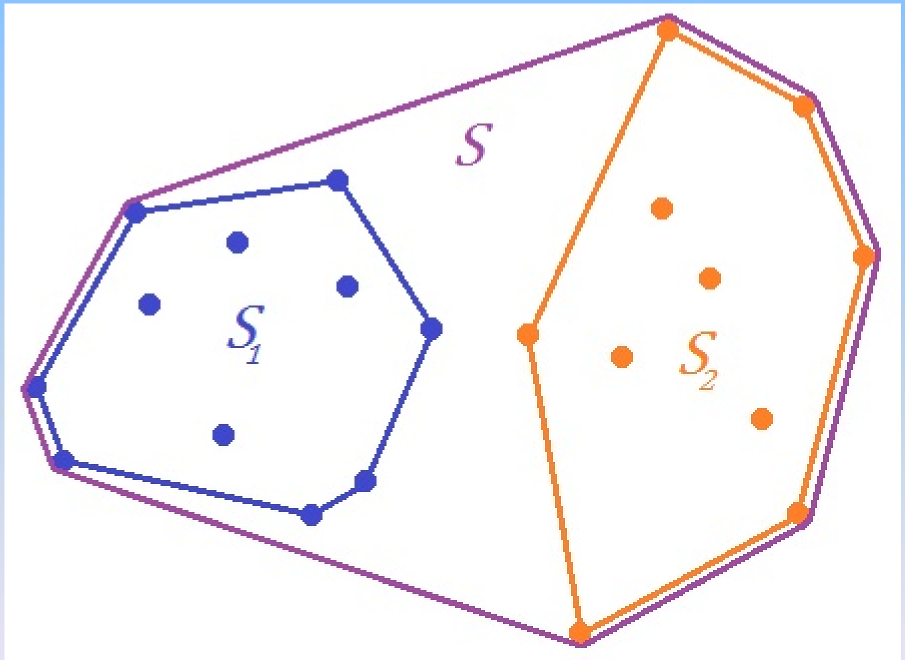
- 平面上两个点集S_1, S_2，设$S = S_1 \cup S_2$，$Convex(S)$表示S集合的凸包上的点组成的集合，那么一定

$$Convex(S) \subseteq Convex(S_1) \cup Concex(S_2)$$

但是不一定满足

$$Concex(S_i) \subseteq Convex(S)$$

- 正是因为这个性质，我们能轻易的维护加点的凸包，却不好维护删点的凸包，从而造成了这道题的困难。



- 怎么办呢？很简单，只要不进行删点操作就搞定了！
- 用一个线段树来维护，线段树的每个节点是一个凸包。这样的话，每次得到一个新的点要插入到凸包中的操作，变成了插入到$O(\log n)$个凸包中；每次需要某段区间的凸包，也可以把这段区间分成$O(\log n)$个小区间，每段小区间的凸包都已经求出。
- 因为p_i不具有单调性，所以需要在每个凸包通过二分查找找到相应的转移位置，然后进行状态转移。整个做法的时间复杂度是$O(n\log^2 n)$，空间复杂度$O(n\log n)$，能够通过本题。

- 总结这道题，斜率优化的推导仍然和常规的斜率优化一样，只是凸包的维护上遇到了常规斜率优化不能解决的问题。这个时候就拿出了数据结构的武器，成功解决了这个问题。

- 接下来我们再来看一个运用数据结构辅助进行斜率优化DP的例子。

上次谈到的一些斜率优化不能解决的问题之二：

- 凸包上新添加的点一定在端点处，那么如果不在端点处会发生什么事情呢？看看这道题：
- 第0天有C元初始金钱，通过通过买卖机器、进行生产获得利润。每台机器只能在第D_i天以P_i的价格购买，可以在任意一天以R_i的价格卖出，从第$D_i + 1$天开始到卖出的前一天每天可以获得G_i的利润。任意时刻只能拥有一台机器，即卖出机器的那一天可以买回一台新机器，第$D + 1$天必须将手上的机器卖出，求此时的最大获利。$(NKOJ\ 3019)$
- $1 \le N \le 100000, 1 \le C \le 10^9, 1 \le D \le 10^9,$
- $1 \le D_i \le D, 1 \le R_i < P_i \le 10^9, 1 \le G \le 10^9.$

谈谈你的看法

- 将所有的机器按照D_i由小到大排序，进行DP：
- $f[i]$表示在第D_i天购买第i台机器之前的最大获利，写出状态转移方程
- $f[i] = \max\limits_{0 \le j < i, f[j] \ge P_j} f[j] - P_j + R_j + G_j \times \left(D_i - D_j - 1\right)$

- 令$U_j = f[j] - P_j + R_j - G_j \times \left(D_j + 1\right)$，转移方程化简为
- $f[i] = \max\limits_{0 \le j < i, f[j] \ge P_j} U_j + G_j \times D_i$

- 然后假设$G_j < G_k$且j比k优，可以得到这样的斜率关系
- $-D_i > \frac{U_k - U_j}{G_k - G_j}$

- 于是把(G_j, U_j)看做平面上的一个点，却惊奇的发现这次并不只是在最右边添加新点了，而是到处都可以添加！

维护凸包的问题变麻烦了。。。

- 也就是说，我们要维护一个可以随便添加点的动态凸包。
- 这是一个经典问题，例如NKOJ2621就是一道这样的裸题，可以用平衡树或块状链表轻易的解决它。
- 为了追求效率，我们采用平衡树。
- 平衡树每个节点表示凸包上一个点，这个节点的前驱后继分别表示凸包上左右相邻的两个点。
- 当需要加入一个新的点时，先按照普通平衡树的规则将其插入到平衡树中，然后进行调整。设新插入的节点为i，我们需要判断$left\left(left(i)\right), left(i), i$三个点的位置关系，考虑是否删除$left(i)$。如果$left(i)$被删除了，需要继续进行判断。同理也需要对$right(i), right\left(right(i)\right)$进行判断。最后再判断$left(i), i, right(i)$的位置关系，考虑是否删除$i$。
- 当需要查询一条斜率为k_l的切线时，从根开始一路向下。走到节点i时，要用$left(i), i, right(i)$的位置关系判断是递归查询左子树，或是递归查询右子树，或是直接返回点i。

代码实现时的一点小技巧

- 因为整个查询过程中需要查询$O(\log n)$次前驱和后继，所以暴力查询会让复杂度提升到$O(\log^2 n)$，这不是我们希望看到的，尝试进行与优化。
- 其实一个简单的处理方法就能够让复杂度回到$O(\log n)$，就是直接把每个节点的前驱后继保存在这个节点上，每次插入删除时更新相邻节点的信息。
- 另外还有一点，注意到每次查询的斜率是单调的，所以每次可以将查询到的节点左边的半棵树全部删除，从而减小树的规模，优化了常数。
- 这样做总的时间复杂度是$O(n\log n)$，空间复杂度$O(n)$，能够轻松的通过。
- 值得一提的是，平面上随机选n个点，凸包的期望点数为$O(\log^2 n)$。所以若不是精心构造的数据，平衡树做法的期望复杂度是$O(n\log\log^2 n)$，块状链表的期望复杂度是$O\left(n\sqrt{\log^2 n}\right)$，平衡树做法反而会在常数上慢于块状链表。

练习题

NKOJ 2878 向量集　　难度 ★ ★ ★
BZOJ 1492 货币兑换　难度 ★ ★ ★
BZOJ 2149 拆迁队　　难度 ★ ★ ★

其实斜率优化挺挺简单的，不是吗？

谢谢！