

2015-11

现代数据库系统

——GSkyline 算法实现与评估

杭天梦
王需
康荣

2015213527
2015311975
2015311965

hangtianmenglisa@163.com
wangxu.93@icloud.com
kr11@mails.tsinghua.edu.cn

GSkyline 算法实现与评估

在本次作业中，我们小组实现了 *Finding Pareto Optimal Groups: Group-based Skyline* 中提出的 PointWise Algorithm 和 UnitWisePlus Algorithm，并对算法实现的正确性和效率进行了评估，通过比较两种算法在不同数据集的表现，初步提出了进一步改进的思路，具体实现将在第二次作业中完成。

1. 背景

Skyline 的关键是它包含了所能支配其他点的“最好”的点。但是有时候我们想要搜索一组备选数据而不是唯一的一个数据，这时就需要引入 GSkyline 概念。给定组的长度 k ，它能选出点集中最优的组，且该组不会被其他组支配。

1.1 POINTWISE 算法

该算法的主要思想是每次在每一层动态生成候选的排列树，同时也要将非 G-Skyline 的候选树尽可能多的删除。对于每一个节点，我们会为它存储一个尾集(tail set)，尾集用来存储比此节点序号大的所有节点。这样，每次从节点的尾集中挑选一个合适的节点加入到当前的集合中，便可以生成新的集合。在预处理后，根节点包含的集合是一个空集，并且它的尾集是前 k 层所有节点的集合。

1.2 UNITWISEPLUS 算法

THEOREM 2. (Verification of G-Skyline). *Given a group $G = \{p_1, p_2, \dots, p_k\}$, it is a G-Skyline group, if its corresponding unit group set $S = u_1 \cup u_2 \cup \dots \cup u_k$ contains k points, i.e., $|S|_p = k$.*

根据定理 2，我们可以知道 G-Skyline 组可以由单元组（unit group）的集合组成。

前面的 Pwise 算法是一次添加一个点，而 Uwise+ 算法一次添加一个组，接下来的思想和 Pwise 基本相似。值得一提是，Uwise+ 是 Uwise 算法的加强版，它对所有的单元集做了一个逆序，因为节点的序号越大，就意味着其所在的层数越大，那么它就会有很多父亲节点，因此，它的单元组的元素个数就更大。这样的好处是：不符合要求的单元组会在算法执行前期被删掉，因此增加了算法的执行效率。

2. 实验环境

操作系统	Windows8.1
处理器	Intel(R) Core(TM) i7-4790K CPU 4.00GHz
内存	8. 00GB
编译环境	Visual Studio 2013

3. 使用指南

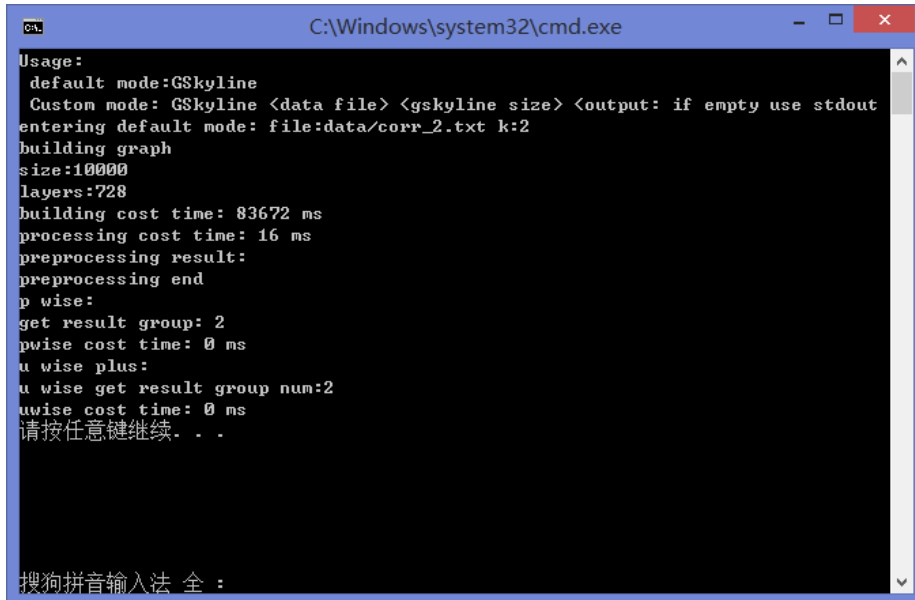
算法使用方法介绍：

1. 在控制台中，输入

“GSkyline.exe 数据文件路径+文件名 k 的值”

如 “GSkyline.exe E:/input.txt 2”,结果会输出到控制台。

2. 输出值解释



```
Usage:
  default mode:GSkyline
  Custom mode: GSkyline <data file> <gskyline size> <output: if empty use stdout>
entering default mode: file:data/corr_2.txt k:2
building graph
size:10000
layers:728
building cost time: 83672 ms
processing cost time: 16 ms
preprocessing result:
preprocessing end
p wise:
get result group: 2
pwise cost time: 0 ms
u wise plus:
u wise get result group num:2
uwise cost time: 0 ms
请按任意键继续. . .
```

Size:数据量大小

Layer: 生成的层数

Building cost time:建立有向图所花的时间

Processing cost time:预处理所花的时间

Get result group: pwise 算法所得的最优组的组数

Pwise cost time: pwise 所花费的总时间

U wise get result group num: Uwise+算法所得最优组的组数

Uwise cost time: Uwise+算法所花的时间

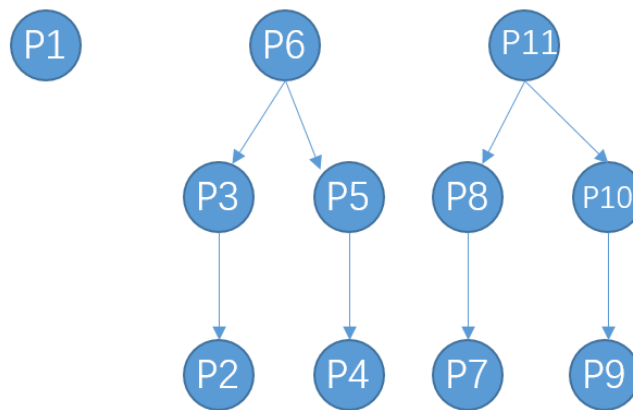
4. 改进详述

4.1 PWISE 方案改进

PointWise 算法的一个很大的缺点就是其在计算大小为 K 的 GSkyline 的时候需要预先计算大小为 $K-1$ 的 GSkyline 的结果，这就导致了该算法的内存消耗巨大，此外，因为要使用如此大的内存，但申请内存消耗的时间都不占少数。为了解决该问题，我们提出基于深度优先搜索的 PointWisePlus 算法。

PointWisePlus 算法的思路首先将 DSG 图进行简化为一个森林。然后对其进行深度优先搜索。为了能够加速运算，本文对 PointWisePlus 进行一些剪枝的运算。

PointWisePlus 算法基于论文作者预处理的基础上，对结果进行第二步预处理。其预处理的作用是针对任意一点的 parent set，只保留其中的一个 parent point，这里定义为 FirstParent。而所有点的 Child 只保留以该点为 FirstParent 为 parent 的点。这样我们就将整个图简化为由多颗多叉树组成的森林，如果创建一个虚拟的根节点，我们则将整个图简化为一颗多叉树。



关于 FirstParent 的选择我们要求当把所有点转化为如上图所示的图时，该点的所有 Parent 必须要是其直接的父亲，要么在该点的右侧。

设原图为 G ，经过上述步骤转化得到的新图为 G' ， G 的 GSkylin 集合为 S ， G' 的

GSkyline 集合为 S' 。

推论 1: $S \subseteq S'$

对于 S 中的任意一个元素 a , a 中包括的点的集合满足 a 的任意点的 parent 都在 a 中。这就意味着 a 的 FirstParent 也在 a 中。所以 a 也是 S' 的一个 GSkyline, 即推论 1 成立。

对于 G 我们给出求其 S' 的算法。

```

Input: stack  $S$  with root,  $k$ 
Output: gskyline
1  $t = S.top$ ;
2 for child point  $e$  in  $t.children$  do
3   | push  $e$  to  $s$ ;
4   | if  $S.size < k$  then
5   |   | PointWisePlus( $S, k$ );
6   | end
7   | else
8   |   | Output  $S$ ;
9   | end
10  | Pop  $e$ ;
11 end
12  $p = t.Parent$ ;
13 while  $p$  is not NULL do
14   | for each right sibling  $e$  of  $t$  in  $p$ 's children do
15   |   | push  $e$  to  $s$ ;
16   |   | if  $S.size < k$  then
17   |   |   | PointWisePlus( $S, k$ );
18   |   | end
19   |   | else
20   |   |   | Output  $S$ ;
21   |   | end
22   |   | Pop  $e$ ;
23   |   |  $t = p$ ;
24   |   |  $p = p.parent$ ;
25   | end
26 end

```

Algorithm 1: PointWisePlus1

该算法保证了如果一个点将被扩展, 其所有的父亲及左兄弟必须已经被扩展。

推论 2: 该算法保证唯一一次遍历 S' 的所有元素。

我们使用先序遍历的方式对该图的所有节点进行标号。算法的第一个 for 循环遍历了节点的所有孩子节点, 所以保证 s 下一个被加入的元素其序号是小于栈顶的元素的。算法的 while 循环自下而上地遍历 t 的右兄弟, t 的父亲右兄弟、 t 的父亲的父亲的右兄弟..., 该过程同样满足先序遍历, 因此最后得到的 s 中的点集必须是从从小到大排列

的。因此该算法最后得到的结果一定不可能重复。

对于 G' 中的任何一个元素，按照先序的顺序排序可以得到一个序列。该序列中每一个点要么是其前一个节点的孩子，要么是其前一个节点或者其所有父亲节点的兄弟。因此该元素一定能够被该算法产生。

剪枝

1. 思路一：对每一个点建立引用记数，如果 e 被压入栈中，那么对 s 及其 parent set 中所有的点建立引用，如果其中的点更新引用前其计算为 0，表明该点将被加入到 s 的 Unit Group 中，更新 s 的 UnitGroup 的大小 size。如果 size 已经超过 k ，则表明该集合不可能成为一个 gskyline，停止加入新的点。
2. 思路二：在思路一的基础上，如果压入 e 时发现 e 已经在 s 的 Unit Group 中。那么扩展 e 后，继续对 e 及其父亲的兄弟节点扩展都是没有意义的，因为 e 是在其右兄弟和父亲的右兄弟的节点的左侧，后面的扩展都不可能再次扩展 e ，也就不可能成为新的 GSkyline。

4.2 UWISE+ 方案改进

这里里共尝试了四种优化策略，分别写在 GSkyline.cpp 中的 uwisepplusplus, uwisepplusplus1, uwisepplusplus2 中。

注：所有的输出函数分别写在 Gskyline:: printAllCombineLayer1(), Gskyline:: printAllLayer1 和 UGroup::printAsc() 中。默认为注释状态，若要输出结果可以去掉注释。

深搜：

三个函数都有用到。其后的三种优化方式都是在此基础上进行的。

本函数包含了两部分优化：1. 将原来的广搜改为深搜；2. 对原算法中只在第一层进行的 GLast 过滤拓展到了所有层。

使用长度为 $k+1$ 的 UGroup 数组 ($0 \sim k$) 记录每层的 Group 状态；

unit 点概念：若点 pi 自身和所有的 parent 组成了一个 unit: ui ，则称 pi 为该 unit 的层点

层的概念是：当已经找到 i 个点作为层点组成了 unit Group，要寻找第 $i+1$ 个层点时，则此时处于第 $i+1$ 层， $layer=i$ 。

使用长度为 k 的 int 数组 tailList 记录每层 UGroup 的 tailSet。由于预处理时已经排序，因此点 pi 的 tailSet 是点 $i-1 \sim 0$ 。若一个 UGroup 有多个层点 $pi1, \dots, pis$ ，则该 UGroup 的 tailSet 范围是 $\min(pi1, \dots, pis)-1 \sim 0$ 。

深搜算法如下：

Algorithm 1 深度搜索算法

Ensure: layer=1; i=allPoints[len-1] (即 index 最大的点) UGroups[0] 为一个空的 UGroup;

- 1: 如果 layer=0, 则退出;
- 2: UGroups[layer-1] 中所有层点的 parent 点组成 PS
- 3: 对于所有小于 i 的点, 如果 p_i 出现在 PS 中, $i=i-1$, 转 6, 否则转 4
- 4: 将 UGroups[layer-1] 赋值给 UGroups[layer], 将 p_i 所在的 unit 插入 UGroups[layer] 中, 此时 UGroups[layer] 新的 tailSet 范围是 $i-1$ 0, 令 $tailList[layer] = i-1$
- 5: 以下三步执行后均转
 - a) 如果 UGroups[layer] 包含点数小于 k, $i = tailList[layer]$, layer++
 - b) 如果 UGroups[layer] 包含点数等于 k, 输出, $i=i-1$
 - c) 如果 UGroups[layer] 包含点数大于 k, $i=i-1$
- 6: 如果 $i < 0$, 则回退, $i = tailList[layer - 1]$, layer=layer-1
- 7: 转 1

以上算法仅仅将原来的广搜改为了深搜, 同时不再保存每一层所有的 group 再进入下一层。这样的改动大大减少了内存的分配和释放, 不仅修复了可能的内存爆掉的 bug, 也使得性能有了较大提升 (相较于第一次原始的算法, 耗时减少了一半)。

多层 GLast 过滤:

写在 UnitWisePlusPlus 中。

可以看到, 在论文算法中, 对于层点数为 1 的 unit (即第一层, unit 由单个点 p_i 及其 parent 构成) 的 GLast 判断, 如果 $GLast \leq k$, 则对任意 $s < i$, 由 ps 作为 layer=1 构成的 unit 都不可能满足 k 条件, 因此直接跳出。

那么在层点数 > 1 时, 也有类似的处理可以剪枝。

定理 1: 在第 i 层, 已存在的 Group 的层点为 $p_{r_1}, p_{r_2} \dots p_{r_{i-1}}$, 其中 $r_1 > r_2 > \dots r_{i-1}$, 所有层点在 $r_s \sim r_{s+1}$ 之间的 parent 点数量记为 $parent_s$, 则对新层点 p_{r_i} , 定义 $GLast_i = i + r_i + \sum_{j=0}^{i-1} parent_s$, 则若 $Glast_i \leq k$, 则接下来的任意点 $p_{rr}, rr > r_i$, 与已有的 $i-1$ 个层点组成的 Group 均不可能满足 k。

以上描述的通俗解释就是, 已有的 $i-1$ 个层点, 他们的 parent 可能分布在 p_{r_i} 之前或之后, 对于大于 r_i 的 parent 点, 已经在之前的遍历中出现并被记录。而 index 小于 r_i 的点必然在 p_{r_i} 的 tailSet, 即 $r_{i-1} \sim 0$ 中。因此包括 i 在内的 i 个层点及其 parent 构成的 group, 最大点数记为 $Glast_i$, 如果 $Glast_i$ 不超过 k, 则比 r_i 更小的点作为第 i 层层点, 自然更不可能满足 k, 因此不必再找新的 i 层点, 可以直接回退到上一层, 寻找下一个 $i-1$ 层层点。

代码实现中采用了 parentNumList 数组。对所有出现在 r_i 到 r_{i+1} 之间的 parent 点, Algorithm 1 第 3 步可以识别出来并将 $parentNumList[i]$ 加 1, 当返回上一层时, $parentNumList[i]$ 清零。

这个优化显然在 k 比较大的时候效果较为显著。然而考虑执行时间, 我们的实验通常选 $k < 10$, 而每一层的点常常超过 100, 这样一来, 对于 k 较小的时候, 该方法的剪枝收益会被

每次的 parentNumList, GLast 判断等多步操作抵消掉。

GSkyline-1 排列组合:

写在 UnitWisePlusPlus1 中。

相较于优化 1, 本层优化非常容易理解。由于 UWise 极好的顺序性, 所有 Gskyline=0 的点 (即第一层 Skyline 点) 满足: 这些点互相之间不被支配, 没有 parent, 即相互独立。

当 layer=1 向后取点取到第一个 Skyline-1 点 p_{r_1} 时, 意味着之后的所有点组成的 unit 只有他自己, 因此只需要在 $r_1 \sim 0$ 之间任意选取 k 个点即可。

在 UnitWisePlusPlus 的第 1 步之后增加以下判断:

如果 layer==1 且 $p_i \cdot \text{layer} = 0$, 则直接输出 $i \sim 0$ 点中的 k 个点的所有组合。

组合生成函数为 printAllLayer1, 只是 k 个坐标的依次移动, 不作详述。

在之前的算法中, 每新选一个点都会带来 tailList 标记、parentSet 合并, 插入等等操作, 耗费大量时间。同时实验所给数据的第一层点数量较大, 采用这个优化效果非常好, 对于原来需要 20,000ms 左右的操作, 优化到了 60ms 左右。

GSkyline-2 排列组合:

写在 UnitWisePlusPlus2 中, 在 **GSkyline-1 排列组合优化** 的基础上实现。

设 Gskyline-1 点范围是 $t \sim 0$

此优化同样容易理解。**GSkyline-1 排列组合优化** 已经表明, 不进行点的插入、删除判断而直接利用排列组合输出, 可以大大提高效率。除了所有点均为 Gskyline-1 的优化之外, 本优化针对的是: 已经有 m 个点后, 寻找第 $m+1$ 层点时到达点 p_i 后, 从 $t \sim 0$ 中任选 $k-m$ 个点。

注意, 由于之前的 group 中的层点的 parent 可能包含 Gskyline-1 中的点, 因此选取 $k-m$ 个点时需要去掉这些 parent 点。假设已有的 ugroup 中包含了 Gskyline-1 中的一些点 p_{r_1}, \dots, p_{r_s} , 则 Gskyline-1 的所有点被划分为 $s+1$ 段, 分别为 $t \sim r_s+1, r_s-1 \sim r_{s-1}+1, \dots, r_1-1 \sim 0$ 。

考虑效率, 这里没有采用从 $t \sim 0$ 中任取 $k-m$ 个点, 然后判断是否有点已经被包含在 Ugroup 的 parent 集合中的方法, 而是将问题转化为从 $s+1$ 个数据集中选取 $k-m$ 个不同的点的排列, 相较于优化 1 要复杂许多。

在 UnitWisePlusPlus1 添加的步骤之后增加以下判断:

如果 layer!=1 且 $p_i \cdot \text{layer} = 0$, 则在剩下的 $t \sim 0$ 集合中任取 $k-m$ 个点 (这些点没有在 ugroup 中出现过), 输出, 并返回上一层。

具体排列组合方法写在 printAllCombineLayer1 中, 过程如下:

在第 i 个集合中任选 s_i 个点, 进入下一个集合中, 递归实现, 直到在某个集合中选够了

$k-m$ 个点，则输出。

已经在 $ugroup$ 中的 $h-1$ 个 GSkyline-1 点将 GSkyline-1 集合分割成 h 个部分，在第 i 部分中选择时，数量有以下限制：

最少应选够 $k-m-\sum_{i=1}^h size(s)$ ，否则即使接下来的部分全部选满都无法满足 $k-m$ 的条件。

如果 $k-m-\sum_{i=1}^h size(s) < 0$ ，则最少数量设为 0，这意味着即使本部分什么都不选，接下来的部分也可以选够 $k-m$ 个点。

最多不应超过 $\min(k-\text{sum}, \text{size}(i))$ ，其中 sum 为之前 $i-1$ 个部分已经选的点数。这很容易理解，不能超过 $k-\text{sum}$ 否则总数就超过了 $k-m$ ，也不能超过 $\text{size}(i)$ 因为本部分只有这么多点。

这个优化带来的效果与数据集有关。对于 $\text{anti}_4, k=3$ ， uWisePlusPlus2 只比 uWisePlusPlus1 快了 4 秒左右。但是对于 $\text{core4}, k=8$ 来说， uWisePlusPlus1 需要 17s 左右，而 uWisePlusPlus2 只需要 2.6s，优化效果非常明显。

这里我们并没有追求极度速度优化，如将函数调用改为内联或内嵌，拆分 for 循环，改造递归等等。我们只是证明了这样的改进方式可以提高速度。

以下是 uWisePlusPlus2 相较于 uWisePlusPlus1 的一些优化。

```
Usage:
  default mode:GSkyline
  Custom mode: GSkyline <data file> <gskyline size> <output: if empty use stdout>
entering default mode: file:data/corr_4.txt k:5
layers:41
data/corr_4.txt 5      983      0      556525  202      438012  16
请按任意键继续. . .
```

```
layers:6
data/inde_8.txt 3      796      15      3634022839  7816      3634022839
6350
请按任意键继续. . .
```

```
Usage:
  default mode:GSkyline
  Custom mode: GSkyline <data file> <gskyline size> <output: if empty use stdout>
entering default mode: file:data/anti_4.txt k:3
layers:3
data/anti_4.txt 3      515      0      119854013077  202988  119854013077
198075
请按任意键继续. . .
```

```
data/corr_1.txt 0      1105
u++
3723233 2060      u++2
3723233 124
请按任意键继续. . .
```

```
layers:41
data/corr_4.txt 7      1170      0      p
u++
22134368      17847      u++2
22134368      624
请按任意键继续. . .
```

5. 性能比较

如下表所示，Pwise+算法为 Pwise 的改进算法，Uwise++为 Uwise+算法的改进算法。测试时，如果算法花的时间过长，我们使用 T = -1 表示。

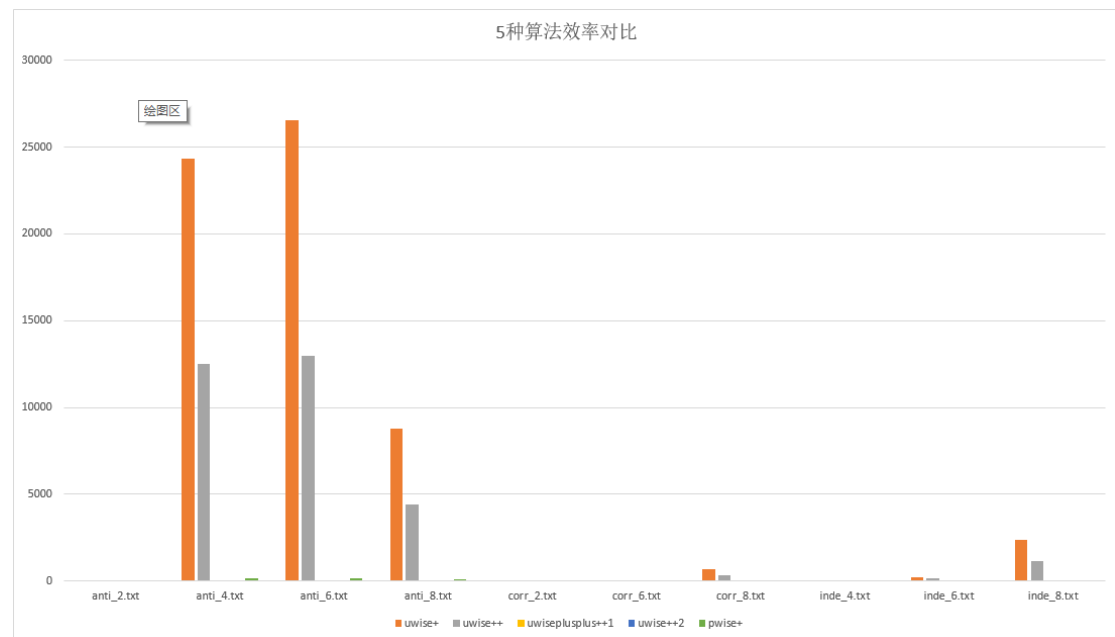


图 1 不同数据集五种算法的运行时间

以下是三个数据集在同一维度下，不同算法 k 与时间之间的关系。由图 2 可以看出，uwise+,uwise++和 uwiseplusplus1 的性能明显弱于 uwise++2 和 pwise+，于是其当 k 的数量增加到 5 的时候。因此改进后的两个算法体现出了较好的性能。

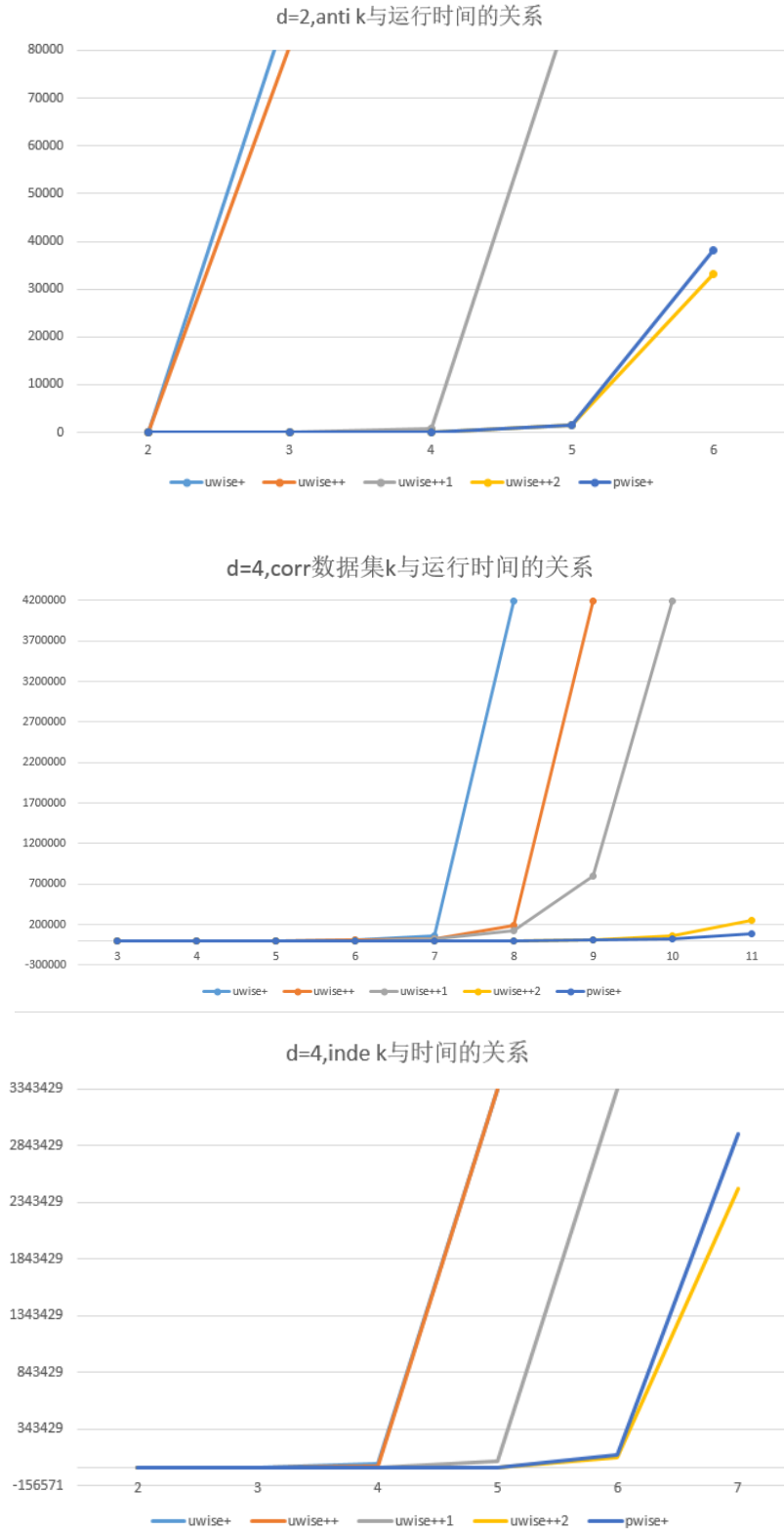


图 2

通过比较维度与时间之间的关系。在 anti 数据集中，uwise++ 和 uwise+ 算法运行效率太慢，因此忽略。由图可知，在同等 k 值得情况下，维度越高，所花的时间越长。但 pwise+ 和 uwise++2 算法依然体现出很好的性能。



图 3

这三个数据集总体来说，anti 花费的时间最长，这也和它的点的特征有关。在进行预处理时裁去一些无用的点，anti 裁剪后剩余的点最多。

6. 结论

上一阶段，我们实现了 Pwise 算法的改进与 Uwise+算法。Pwise+算法（改进后的 Pwise 算法）的性能明显优于朴素 Uwise+算法。

这一阶段，我们实现了 Uwise+算法的改进。Uwise++算法（改进后的 Uwise+算法）在运行效率上有了显著的提升，并优于 Pwise+算法。

对于 Uwise+算法来说，前三个优化实现起来都比较顺利，只需要用 test 的例子测过了，修一些 bug，那么对于大数据量就全都正确。但这只是运气好，因为第四个优化的排列组合比较复杂，调了一下午的错误只是由于一行的考虑不周。但总算调出来了。

从 uWisePlus 到 uWisePlusPlus (改为深搜) 到 uWisePlusPlus (GSkyline-1 优化)，再到 uWisePlus (GSkyline-2 优化)，每一次优化都会让效率提高几倍甚至上百倍，这一方面说明原来的代码问题重重，但另一方面也可以看出，我们的努力得到了回报。看到每次优化之后意料之外的运行时间大幅减少，没有比这种追求极限的感觉更棒的了。

整个过程大家均付出了努力，在讨论过程中碰撞出了思想的火花。大家也提出了很多新的思路与想法。

感谢助教批阅！