2015-11

现代数据库系统

——GSkyline算法实现与评估

杭天梦 2015213527 hangtianmenglisa@163.com

王需 2015311975 wangxu.93@icloud.com

康荣 2015311965 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。

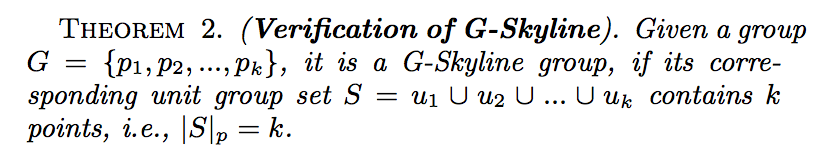
# 背景

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。

## 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。

## UnitWisePlus算法



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。

# 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 |

# 使用指南

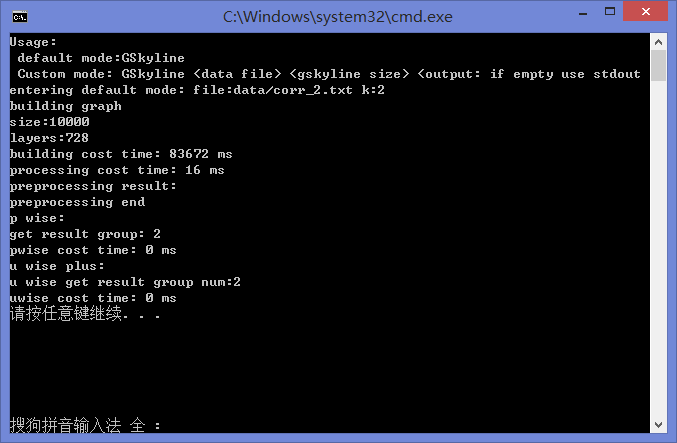
算法使用方法介绍：

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。

1. 输出值解释



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

# 改进详述

## PWise方案改进

1. 改宽度优先搜索为深度优先搜索

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，但但申请内存消耗的时间都不占少数。为了解决该问题，我们首先将PointWise算法改为深度优先搜索。

具体过程如下：

1. 首先，对于一个预处理后的DSG，对每一个点，只保留一条指向其父亲的边，这条边的选择是多样的，我们使用该边上一图层的第一个能domain该点的边，这样就把复杂的DSG图转化为一个森林。森林中每颗树的根就是一个Skyline点。为一个Skyline点加一个虚拟的根节点，这样就将该森林转化为了一颗树。下面的任务便是深度优先搜索这颗多叉树。搜索过程如下：
2. 先将跟节点压入栈中。
3. 对栈中最后一个节点进行扩展，首先扩展该节点的所有下层节点，其次扩展该节点的搜有右下层节点。
4. 递归地扩展栈顶的节点，如果栈的大小为K，则输出，不再扩展该节点。

可以知道该种方法能够遍历所有Size为k的group的组合。

1. 剪枝

2 中描述的过程如果不进行剪枝的话，其效率是很低的，这里介绍剪枝的思想。对于一个Group集合，首先应确定该Group 的Unit Group的大小，这里我们并不保存Unit Group的所有点的集合，而但但保存Unit Group的大小，同时对点的引用次数进行计数。当向group中加入一个新的点的时候，如果该点的引用次数为0，则说明该点还不在Unit Group中，将其引用次数+1，Unit Group的大小同样加1，同时将该点的Parent Set中的点也以类似的方式更新引用次数和Unit Group的大小，如果将一个点删除出Group，只需要将其引用次数减1，且其所有的Parent 的引用次数减1。当获得一个新的Group的Unit Group的大小如果已经超过K，该枝应该被剪掉，当获得一个新的Group的Unit Group的大小已经为K，但是其本身的大小仍然不为K，该情况也应该继续被扩展，此外，如果新的Group的大小和Unit Group的大小都为K，表明该集合正是我们需要的结果，如果新的Group的大小和Unit Group的大小都小于K，则继续向该Group中加入点。

## 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uwiseplusplus，uwiseplusplus1，uwiseplusplus2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~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-1~0。若一个UGroup有多个层点pi1,…pis，则该UGroup的tailSet范围是min(pi1,…pis)-1~0.

深搜算法如下：

Algorithm 1：

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. UGoups[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pi出现在PS中，i=i-1,转6，否则转4
4. 将UGoups[layer-1]赋值给UGoups[layer]，将pi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
   1. 如果UGroups[layer]包含点数小于k，i = tailList[layer]，layer++
   2. 如果UGroups[layer]包含点数等于k，输出，i=i-1
   3. 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--
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pi及其parent构成）的GLast判断，如果GLast<=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pr1,pr2…pri-1，其中r1>r2>…ri-1，所有层点在rs~rs+1之间的parent点数量记为parents，则对新层点pri，定义GLasti=i+ri+Σi-10 parents，则若Glasti<=k，则接下来的任意点prr,rr>ri,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pri之前或之后，对于大于ri的parent点，已经在之前的遍历中出现并被记录。而index小于ri的点必然在pri的tailSet，即ri-1~0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Glasti，如果Glasti不超过k，则比ri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ri到ri+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pr1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r1~0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pi.layer = 0,则直接输出i~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~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层点时到达到达点pt后，从t~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pr1,…prs，则Gskyline-1的所有点被划分为了s+1段，分别为t~rs+1, rs-1~ rs-1+1… r1-1~0。

考虑效率，这里没有采用从t~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pt.layer = 0,则在剩下的t~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si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-Σhisize(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。

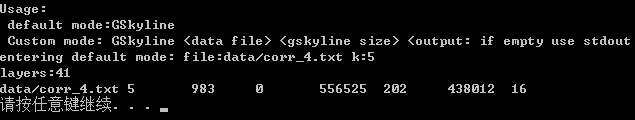
如果k-m-Σhisize(s)<0，则最少数量设为0，这意味着即使本部分什么都不选，接下来的部分也可以选够k-m个点。

最多不应超过min(k-sum,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-sum否则总数就超过了k-m，也不能超过size(i)因为本部分只有这么多点。

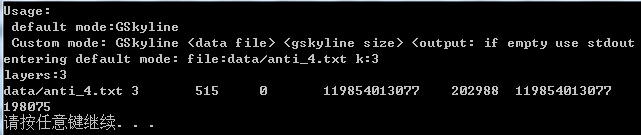
这个优化带来的效果与数据集有关。对于anti\_4,k=3，uWisePlusPlus2只比uWisePlusPlus1快了4秒左右。但是对于core4，k=8来说，uWisePlusPlus1需要17s左右，而uWisePlusPlus2只需要2.6s，优化效果非常明显。

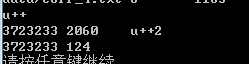
这里我们并没有追求极度速度优化，如将函数调用改为内联或内嵌，拆分for循环，改造递归等等。我们只是证明了这样的改进方式可以提高速度。

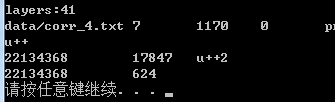
以下是uWisePlusPlus2相较于uWisePlusPlus1的一些优化。











# 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。

表1-算法运行表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 文件 | k | Pwise+运行时间(ms) | Uwise+运行时间(ms) | Uwise++运行时间(ms) | 建图时间(ms) | 预处理时间(ms) | 组数 |
| anti\_2.txt | 1 | 0 | 0 | 0 | 484 | 0 | 139 |
| anti\_2.txt | 2 | 0 | 0 | 0 | 484 | 0 | 9717 |
| anti\_4.txt | 1 | 0 | 0 | 0 | 421 | 15 | 8960 |
| anti\_4.txt | 2 | 9765 | 12870 | 63 | 422 | 0 | 40137097 |
| anti\_6.txt | 1 | 0 | 0 | 0 | 421 | 16 | 9386 |
| anti\_6.txt | 2 | 10686 | 14196 | 63 | 421 | 0 | 44044277 |
| anti\_8.txt | 1 | 0 | 0 | 0 | 734 | 15 | 5378 |
| anti\_8.txt | 2 | 3432 | 4477 | 31 | 764 | 0 | 14460071 |
| corr\_2.txt | 1 | 0 | 0 | 0 | 1373 | 0 | 1 |
| corr\_2.txt | 2 | 0 | 0 | 0 | 1139 | 0 | 2 |
| corr\_4.txt | 1 | 0 | 0 | 0 | 905 | 0 | 35 |
| corr\_4.txt | 2 | 0 | 0 | 0 | 983 | 0 | 625 |
| corr\_6.txt | 1 | 0 | 0 | 0 | 811 | 0 | 404 |
| corr\_6.txt | 2 | 16 | 31 | 0 | 796 | 0 | 81639 |
| corr\_8.txt | 1 | 0 | 0 | 0 | 749 | 0 | 1520 |
| corr\_8.txt | 2 | 265 | 359 | 0 | 749 | 0 | 1155102 |
| inde\_2.txt | 1 | 0 | 0 | 0 | 842 | 0 | 10 |
| inde\_2.txt | 2 | 0 | 0 | 0 | 796 | 0 | 47 |
| inde\_4.txt | 1 | 0 | 0 | 0 | 811 | 0 | 172 |
| inde\_4.txt | 2 | 0 | 0 | 0 | 780 | 0 | 14814 |
| inde\_6.txt | 1 | 0 | 0 | 0 | 687 | 0 | 887 |
| inde\_6.txt | 2 | 94 | 109 | 16 | 702 | 0 | 393417 |
| inde\_8.txt | 1 | 0 | 0 | 0 | 733 | 0 | 2794 |
| inde\_8.txt | 2 | 905 | 1170 | 0 | 687 | 0 | 3902795 |

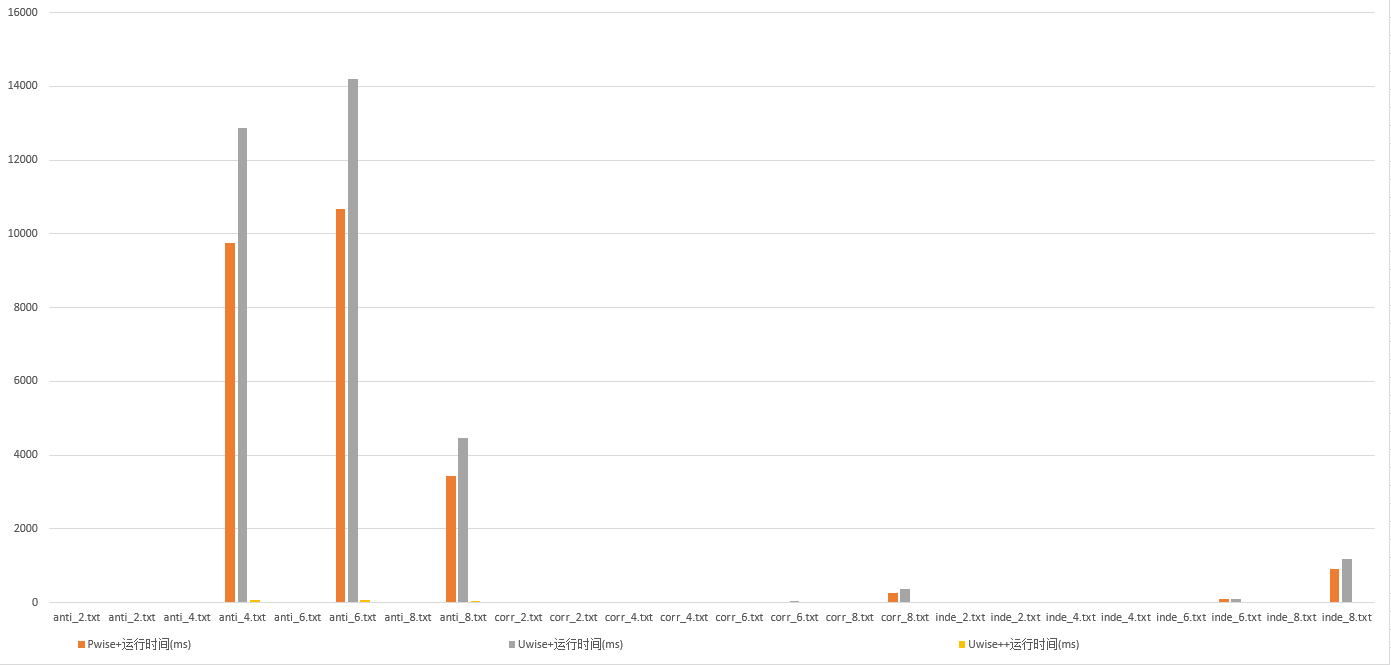


图1 不同数据集三种算法的运行时间

如图1所示，x轴为文件名，y轴为三种算法运行所花的时间。由图可得，UWise++花费的时间均为最小，Pwise+所花的时间小于UWise+时间，PWise算法的优化方案也有了明显的效果。

从图1也可得，对于不同的数据集，算法花费的平均时间也不同。当数据维度相同的时候，anti数据集花费的时间普遍高于corr和inde数据集。由图可知，当数据集的维度等于8时，所花费的时间为anti>index>corr。尽快如此，Uwise++算法也体现出很出色的效果。

当k=1时，不同数据集上所有的算法执行所花时间均小于1ms,当k=2时，不同数据集上各个算法的效率开始拉开了差距。当k≥3时，所花的时间较多，但UWise++算法依然具有优势，如，当数据集为anti\_4，k=4时，T=188371ms。

# 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！