**Group skyline 实验报告**

丁霄汉 2017312365 吴超月 2017213865 罗  瑶  2017213866

# 实验目的

实现论文《Finding Pareto Optimal Groups: Group-based Skyline》中的G-Skyline算法。

# 实验任务

主要包括实现文中生成skyline layers的算法并对结果集进行预处理生成directed skyline graph、point-wise算法的实现、unit group-wise+算法的实现。

注：由于unit group-wise+在unit group-wise算法基础之上进行了进一步的剪枝，因此在实验中我们直接实现了unit group-wise+算法。

## 2.1生成DSG

生成skyline layers并构造后续算法所需的数据结构directed skyline graph。

### 2.1.1基本思路

主要分为以下几个步骤：

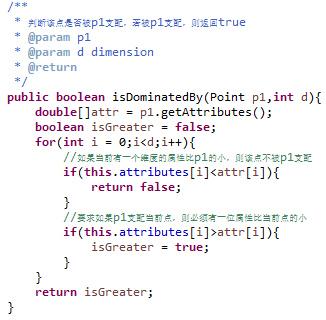
1. 获得给定的数据集后，按照第一维度属性值升序排序；
2. 判断属性维度，如果等于2则直接使用二维下的方法进行skyline layers生成，否则进入高维属性下的skyline layers生成方法；
3. 获得相应的k条skyline后，对每个节点利用论文中的预处理思想去掉不可能出现在group skyline中的点，生成预处理后的DSG。

### 2.1.2代码实现

1. **Point类**

Point类用于存放从文件中读入的每条数据，成员变量包括用于标记当前数据的标号以及存放各个维度数据的double类型数组。Point类实现了Comparable的compareTo方法，使得Point可以按照其第一维度属性值进行升序排序。

此外，Point类包含一个isDominatedBy方法利用定义判断当前点是否被给定的点所支配：如果当前点某个维度上的属性值比给定的点小，则不被其支配；如果所有维度上的属性值都不比给定的点相应的值小，且至少存在一个维度上的属性值比给定点的大，则该点被给定的点所支配：



1. **DSGNode类**

DSGNode类用于表示DSG中的每一个节点，成员变量包括用于标记当前数据标号的pointIndex、记录数据所在的skyline层号的layerIndex、存放各个维度数据的double类型数组attributes、记录其所有父节点的列表parents、记录其所有子节点的列表children以及记录属性维度个数的整形变量d。

1. **ProcessResult类**

ProcessResult类保存程序执行过程中需要记录的数据，包括一个存放所有供后续算法使用的DSGNode列表、存放在预处理中已经找到的符合要求的group以及存放最终结果集的列表。

1. **DSGGenerator类**

DSGGenerator类为生成directed skyline graph的主要处理类，成员变量包括所生成的group skyline中的成员个数k、传入的待分组数据集、记录生成的k条skyline的列表以及每条数据的维度个数d。

在调用generateDSG方法生成DSG时，该方法首先调用成员方法generateSkylines生成k条skyline。在generateSkylines方法中，首先对数据集按照第一维度属性升序排序，然后判断属性的总维度，若为2，则调用generate2D方法利用论文中的algorithm1生成k条skyline，否则调用generateHighDimension方法使用简单的遍历生成k条skyline。在获得k条skyline后，利用论文中的方法对数据集进行预处理，并最终生成供后续使用的DSG。

在调用generateSkylines方法生成k条skyline后，对每个节点进行预处理，筛选掉不可能出现在group skyline中的点，即判断其unit group的大小是否大于k，如果大于则抛弃这些节点；再对每一个点遍历在其所在skyline之前的所有skyline上的点，判断这些点是否支配该点，若支配，则将该点加入对应点的children列表中，相应的将这些点加入该点的parents列表中，最终生成DSG。

generate2D方法即对论文中algorithm1的实现，在其中调用了binarySearchLayer方法加快对给定点所属skyline层号的判断。此外，在生成新的skyline时判断当前skyline的层数，若大于k，则不再继续生成。

generateHighDimension方法即将论文中的algorithm1算法应用到高维属性中，此时无法利用二维情况下的单调性，即需要遍历每个已加入skyline的节点进行判断新加入的节点属于哪个skyline。

## 2.2 point-wise算法实现

从预处理后得到的DSG出发，DSG中的DSGNode已经按照index由小到大排列，用point-wise方法计算group skyline

### 2.2.1基本思路

point-wise方法的基本思路就是从空集开始，逐层生成大小为1，2…k-1，k的G-skyline groups。当已经得到大小为i的G-skyline groups之后，遍历每个大小为i的G-skyline group，每个G-skyline group都有一个尾集合，尾集合包含所有index大于G-skyline group中所有点的index的点，遍历尾集合中的点，向G-skyline group中增加尾集合中的点构成大小为i+1的candidate group，然后检查该candidate group是否是G-skyline group，如果是，就将candidate group加入大小为i+1的G-skyline groups。point-wise尽可能的在每一步剪枝。以下是point-wise的几个剪枝方法：

1. Subtree Pruning：一旦某个大小为i的candidate group被检查之后发现不是G-skyline group，那么往其中加入任何点都不能构成大小为i+1的G-skyline group.所以我们可以直接舍弃这个点不存储
2. Tail Set Pruning：

A、G-skyline group中如果包含某个点p，那么点p的父母节点一定也在这个G-skyline group中。所以我们可以对G-skyline group的tail set进行删减，我们首先计算得到G-skyline group中所有节点的孩子节点集合，tail set中的点如果不在孩子节点集合中或者属于第一层skyline则直接从tail set中删除该点。这样可以减少candidate groups的大小。

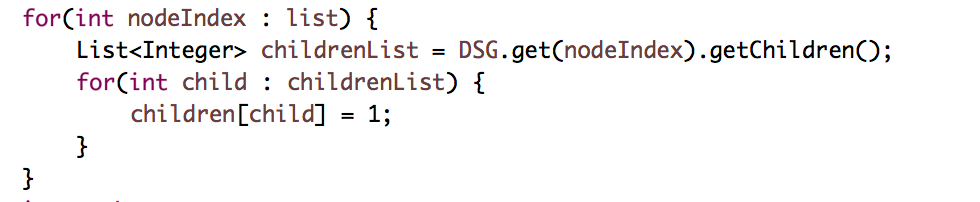
B、假设G-skyline group中节点的最大层数为i，则加入的点p必须来自前i+1层。所以tail set中的点如果层数大于i+1也被直接删除。

总的来说，point-wise包含三个循环，最外的循环每循环一次生成一层的所有G-skyline groups，层数从1到k；在生成每一层的G-skyline groups时，循环遍历所有上一层得到的的G-skyline groups，遍历每一个上一层的G-skyline group时，对tail set进行剪枝；然后遍历剪枝后的tail set，每次遍历得到一个candidate group，检查是否为G-skyline group，如果是，加入集合，否则直接舍弃。

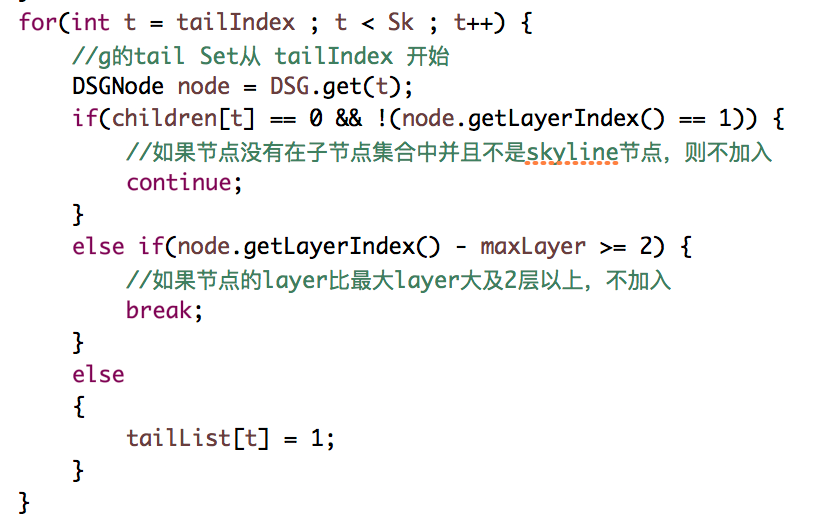
### 2.2.2代码实现

代码使用java实现。

1. 每一个G-skyline group使用List<Integer>表示，每个节点使用其index代表。最后得到的大小为k的G-skyline groups用List<List<Integer>> groupListNew表示。
2. 使用int[] children来统计每个G-skyline group的孩子集合，下标i中的整数为1代表index为i的节点是G-skyline group的孩子，为0代表不是孩子。具体代码实现如下，遍历G-skyline group的每个节点的孩子节点列表，将children数组对应于该孩子节点的下标index的整数设为1。



1. 使用int[] tailList来统计删减过后的尾集合。下标i中的整数为1代表index为i的节点是没有删减的节点。具体实现如下：



1. 遍历tailList，如果下标为i的位置整数为1，将index为i的节点加入G-skyline group，检查是否为新的G-skyline group，这里使用了一个检查是否为G-skyline group的检查方法：检查candidate group的unit group（自己的父母节点加上自身的集合）的大小是否等于candidate group自身的大小，这里使用java的数据结构Set／HashSet来去重检查。具体代码实现如下：



最后groupListNew中存储所有大小为k的G-skyline groups，groupListNew加上DSG预处理阶段得到的unit大小为k的节点集合perfectNodeList就是所有的解。

## 2.3 unit-wise+算法实现

从DSG出发，用unit-wise方法计算group skyline。

### 2.3.1基本思路

主要分为以下几个步骤：

1. 以上一步骤输出的DSG为输入，简化数据结构，构造RefDSG，即Reference-based DSG。这是因为上一步输出的DSG的节点DSGNode是通过List中的下标来存储其父母节点和孩子节点的，而在unit-wise方法中，在涉及unit group的操作时需要频繁查询和合并许多节点的父母节点和孩子节点。将DSGNode转换为基于引用的RefNode，用HashSet保存节点的父母节点和孩子节点的引用，并抛弃无用属性，可以提高效率。另一方面，将排序后的unit group和node作为RefDSG的成员变量，可以提高抽象层次，提高可维护性。
2. 按照论文中的实现方式，将nodes和unit groups按照原本的索引反向排序。
3. 构造仅由单一unit group组成的初始groups列表，即singleUnitGroups。
4. 对每一个初始group：
5. 检查Glast。
6. 以该group为初始节点，构建枚举树，并按照论文中的方法进行剪枝。在这里我们结合java的语言特性做了一些改进。

最终输出结果。

### 2.3.2代码实现

1. **RefNode：**

基于引用的DSG Node。用originPointIdx表示原本的节点索引，parents表示其所有的祖先节点，children表示其所有的后代节点。这里沿用了论文中parent和children的称谓，虽然这个命名不是很精确。这里使用HashSet是为了提高查询和合并的效率。

1. **Unit类**

单个unit group，记录其起始节点和所有长辈节点。

1. **UnitGroup类**

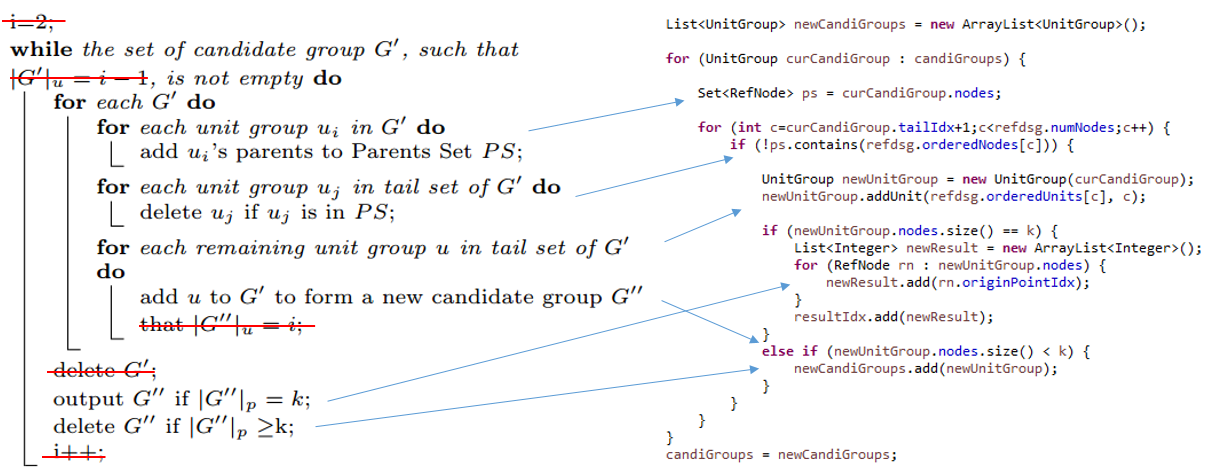
由unit group组合而成的group。记录其在枚举树tail list中的位置和包含的节点。

1. **RefDSG**

基于引用的DSG。保存排序的nodes和units。

1. **UnitWise类**

负责应用论文中描述的unit-wise方法进行计算并输出结果。

在具体的实现上，我们结合java的语言特性对论文中的方法进行了优化。以下面这段代码为例。

论文中所说的一个node的parent set其实可以等价替换为对应unit group的nodes集合。这两个集合的差集就是这个node本身，这一差别对剪枝没有影响。这个集合其实我们已经在之前求出来了，因而大大提高了效率。

在求tail set的时候，我们没有像论文中一样“删去在parent set中的unit group”，而是“只添加不在parent set中的unit group”。考虑到java中ArrayList的删除操作效率低下，LinkedList的插入操作效率又低于ArrayList，这样做可以使我们既使用ArrayList又避免删除操作，提高了效率。

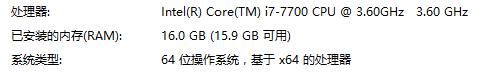
文中使用变量i来索引不同深度的枚举集合。考虑到java的回收机制，我们直接用新的枚举集合替代旧的，以使得旧的集合占据的资源可以被释放和回收。

我们对论文提出算法的若干循环做了最大限度的合并，以提高效率。

# 实验结果

 实验在给定的三种数据分布（inde、corr、anti）的4种维度（2、4、6、8）下共12组数据上进行。实验的group size k取值分别为2、4、6。在上面的两种算法实现描述中，均将所有的结果放入内存中，但是我们在实验中发现，当结果数目较大（上十亿）时，会发生内存溢出，而在实际的应用中，结果集也不是存在内存中的，所以我们在两个算法中在构造出每个结果后，没有将其加入结果集，而只是统计了结果数。

实验环境：





## 3.1生成skyline layers实验结果

图表1、图表2、图表3分别展示了k取2、4、6时在上述12种数据集上生成skyline layers的耗时。在维度为2时采取二分查找的方式进行插入，而在高维度时采取简单的遍历方式。通过图表可以发现，在k取不同值、在具有不同分布的数据集上生成skyline layers时，维度为2时的耗时要明显小于高维情况，也就是说，论文中针对维度为2时的二分处理具有较好的性能优势。

图表1

图表2

图表3

## 3.2 group size k=2时实验结果

表1展示当k=2时在12组数据上使用point-wise和unit-wise+算法计算group skyline的结果。图表4、5、6分别展示两种算法在三种不同分布的数据集上随着维度变化相应时间的变化。

从表1中可以看出，g-skyline size和算法运行时间与预处理之后的点数指数级相关。一般情况下，随着维度的增加，预处理之后的点数相应明显增加，因此最终体现为随着维度的增加，算法运行时间呈现指数级增长（见图表5、图表6）；而anti数据集在维度为4、6、8时，预处理之后的点数集合都非常大并且也与维度不呈现正相关，所以体现为anti数据集在维度为4、6、8时处理时间都非常长，甚至维度为8时算法运行时间比维度为4和6的情况下短（见图表4）。

同时，k=2时，三个数据集12组数据中point-wise算法都比unit-wise+表现好。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **point-wise/μs** | **unit-wise+/μs** | **前k层天际线点数** | **预处理之后的点数** | **g-skyline size** |
| **anti\_2** | 13208 | 27812 | 359 | 129 | 9717 |
| **anti\_4** | 7495227 | 14810862 | 10000 | 8960 | 40137105 |
| **anti\_6** | 12994720 | 16038475 | 10000 | 9386 | 44044277 |
| **anti\_8** | 3073420 | 5300289 | 10000 | 5378 | 14460073 |
| **corr\_2** | 838 | 2108 | 3 | 1 | 2 |
| **corr\_4** | 4569 | 8273 | 10000 | 35 | 632 |
| **corr\_6** | 41852 | 75426 | 10000 | 404 | 81648 |
| **corr\_8** | 259113 | 479645 | 10000 | 1520 | 1155104 |
| **inde\_2** | 1012 | 2844 | 31 | 10 | 47 |
| **inde\_4** | 21518 | 33213 | 10000 | 172 | 14828 |
| **inde\_6** | 114242 | 246345 | 10000 | 887 | 393429 |
| **inde\_8** | 679000 | 1578361 | 10000 | 2794 | 3902800 |

表1

图表4

图表5

图表6、

## 3.3 group size k=4和6时实验结果

表2和表3分别展示k取4和6时的实验结果。

当k取4时，如表2所示，预处理之后的点数较小（约在400以下）的数据集能够顺利地在较短的时间内得出结果；数据集corr\_6预处理之后的点的数目到达844时，处理时间需要两分钟，结果集大小到达十亿，数据集inde\_6预处理之后的点的数目到达1682时，处理时间则达到了半个小时，结果集数目达到\*\*；而当预处理之后的点数更大的时候（超过2000），结果数据集的大小只能更大，两种算法在有限的计算时间都无法计算出结果集，所以我们没有收集到运行时间。

我们以point-wise算法为例，point-wise算法需要同时存储第k层之前的每一层的所有G-groups，所以我们统计了Sk数目从390到2651的数据集在计算过程中产生的中间结果集的大小。

当Sk比较大（844），第三层的所有G-groups数目到千万级，point-wise算法占用的内存特别大。

当Sk更大一点（1682），第二层的所有G-groups数目就有几十万，对应的第三层的G-groups数目上亿，最终结果集的大小到达\*\*\*。

当Sk非常大（2000以上）时，内存已经不足以能够分配出能够装下中间结果的大小，此时point-wise算法就失效了。我们可以看到corr\_8的第三层的结果集就已经超过了内存大小。

我们也可以看到当k=6时，Sk到达200以上point-wise算法需要存储的中间结果的数目就已经超过了内存可以分配的极限，point-wise算法失效。此时unit-wise+可以比较好的工作。

但是当Sk比较大时，即使利用unit-wise+计算得到了一个包含着几十亿上百亿结果也是没有任何实际意义的，这是本文算法极大的局限性。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **point-wise/μs** | **unit-wise+/μs** | **前k层天际线点数** | **预处理之后的点数** | **g-skyline size** |
| **anti\_2** | 3163941 | 1992135 | 918 | 390 | 16124810 |
| **anti\_4** |  |  | 10000 | 9915 |  |
| **anti\_6** |  |  | 10000 | 9951 |  |
| **anti\_8** |  |  | 10000 | 7382 |  |
| **corr\_2** | 507 | 2067 | 9 | 6 | 7 |
| **corr\_4** | 33849 | 42707 | 10000 | 90 | 71701 |
| **corr\_6** | 140062003 | 125487122 | 10000 | 844 | 1112974784 |
| **corr\_8** |  |  | 10000 | 2651 |  |
| **inde\_2** | 2845 | 3039 | 77 | 29 | 440 |
| **inde\_4** | 5889611 | 4160674 | 10000 | 383 | 36861708 |
| **inde\_6** | 2924445368 |  | 10000 | 1682 |  |
| **inde\_8** |  |  | 10000 | 4374 |  |

表2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **point-wise/μs** | **unit-wise+/μs** | **前k层天际线点数** | **预处理之后的点数** | **g-skyline size** |
| **anti\_2** |  |  | 1502 | 575 |  |
| **anti\_4** |  |  | 10000 | 9981 |  |
| **anti\_6** |  |  | 10000 | 9992 |  |
| **anti\_8** |  |  | 10000 | 8124 |  |
| **corr\_2** | 979 | 2334 | 17 | 8 | 8 |
| **corr\_4** | 5708625 | 3570728 | 10000 | 162 | 3723233 |
| **corr\_6** |  |  | 10000 | 1166 |  |
| **corr\_8** |  |  | 10000 | 3310 |  |
| **inde\_2** | 23603 | 13648 | 128 | 47 | 2930 |
| **inde\_4** |  |  | 10000 | 550 |  |
| **inde\_6** |  |  | 10000 | 2210 |  |
| **inde\_8** |  |  | 10000 | 5144 |  |

表3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **预处理之后的点个数Sk** | **第一层** | **第二层** | **第三层** | **第四层** |
| **anti\_2** | 390 | 139 | 9717 | 455623 | 14513512 |
| **corr\_6** | 844 | 404 | 81639 | 11003536 | 1112974784 |
| **inde\_6** | 1682 | 887 | 393417 | 116343425 |  |
| **corr\_8** | 2651 | 1520 | 1155102 | —— | —— |

表4