实现自适应空间文本分割树

王梓涵　周昕逸　刘权

1　理论介绍

在日常的信息传播中，常常需要处理用户所提供的一系列包含空间和关键词的查询项。例如：一些基于位置信息的推荐系统可根据用户的需求推送相关产品的信息，Twitter 等社交网络可分析用户发布的内容及附带的地理位置标签推送他们感兴趣的内容。本文实现的自适应空间文本分割树（Adaptive spatial-textual Partition Tree，以下简称 AP 树）可以通过自动计算关键词和空间因子两种分割方式的代价选择合适的匹配方式。

AP 树的实现有三大挑战。第一，在实际应用中的数据量极大，算法效率的提升可以节省很多成本。第二，算法吞吐量要大以应对源源不断的包含空间文本信息的对象流。第三，需要建立新的空间因子和关键词的索引机制用于不同的分配方法。

AP 树的基本运算包含三个部分。匹配对象和查询项的算法，计算关键词和空间因子分割代价的模型和建立索引的算法。

AP 树包含三种类型的结点：关键词结点（k-node），空间结点（s-node）和查询项结点（q-node）。一棵 AP 树的叶子结点都是 q-node，每一个查询项都可根据其关键词和空间因子被分配入一个或多个叶子结点中。运用关键词分割方法的结点称为 k-node。我们认为所有的关键词都包含于一个有序的词典中，它们的相对位置是确定的，因此可以比较大小。每一层的 k-node 中包含了其叶子结点对应的查询项的第结点层次个关键词的划分（cuts）。运用空间面积分割方法的结点称为 s-node。每一层的 s-node 中包含了其叶子结点对应的查询项的面积划分（cells）。如果一个查询项没有足够的关键词，无法寻找到一个对应的 cut，就用一个 dummy cut 保存；如果一个查询项的覆盖面积已经包含一个 s-node 的覆盖面积，就用一个 dummy cell 保存。

在匹配对象时，使用深度优先搜索法。如果当前结点是 q-node，则判断该对象是否与 q-node 中的查询项匹配，如果匹配则可等待输出。如果当前结点是 s-node，则访问它的 cell 和 dummy cell。如果当前结点是 k-node，则访问它的 cut 和 dummy cut。

AP 树还提供了一个计算两种分割匹配算法的代价的模型。匹配代价 $C(P)$ 的定义为：

$$C(P) = \sum_{i=1}^{f} \mathbf{w}(B_i) \times \mathbf{p}(B_i)$$

其中 B 是一种分割方式，$\mathbf{w}(B)$ 是与 B 有关的查询项的个数，$\mathbf{p}(B)$ 是 B 的击中概率，即在对象匹配过程中 B 被访问的可能性。在关键词分割法中，$\mathbf{p}(B) = \sum_{w \in B} \mathbf{p}(w)$，其中 $\mathbf{p}(w) = \frac{freq(w)}{\sum_{w \in P} freq(w)}$，

$freq(w)$ 为关键词 w 在所有查询项中出现的频率。在空间分割法中 $\mathbf{p}(B) = \frac{Area(B)}{Area(N)}$，$Area(B)$ 是 B 的面积，$Area(N)$ 是结点 N 的区域面积。接着使用启发式算法寻找使得匹配代价最小的关键词的分割和空间的分割。

AP 树还提供了建立索引的方法。使用两个标志 kP 和 sP 来表示所有的查询项是否可以进一步地进行关键词的分割和空间的分割，并通过计算两种分割方式的代价 C_k 和 C_s 确定当前结点使用哪种分割方式。

2 代码实现

2.1 接口设计

首先定义所涉及的两个结构体：空间文本对象 `STObject` 以及查询项 `Query`。`STObject` 有两个成员：`Pointf` 类型的位置 `location` 以及 `std::set<std::string>` 类型的关键词集合 `keywords`。`STObject` 也有两个成员：`Boundf` 类型的方形区域 `region` 和 `std::set<std::string>` 类型的关键词集合 `keywords`，其中 `Boundf` 为由两个 `Pointf` 确定的轴对齐包围区域。类的使用者需要通过这两个结构来和类进行交互。

本文所实现的 AP 树，是一种空间文本分割树，定义抽象的空间文本分割树类 `STTree`，其有两个纯虚函数 `std::vector<Query> Match(const STObject &) const` 和 `void Register(const std::vector<Query> &)`。AP 树类 `APTree` 继承自 `STTree`。`APTree` 的构造函数声明为 `APTree(const std::vector<std::string> &vocab, const std::vector<Query> &queries, size_t f, size_t theta_Q, size_t theta_KL)`,其中 `vocab` 为所有的关键词,`queries` 为需要注册的查询项,`f`、`theta_Q` 和 `theta_KL` 均为论文中明确给出的参数。

2.2 存储实现

由于 AP 树存储的数据体量大，树的构建和查询算法较为复杂，为了能够发挥 AP 树的优势，在内部存储方式的实现上需要一定的构思。

首先介绍 AP 树内部对于空间文本对象和查询项的存储。在 `APTree` 内部，使用 `STObjectNested` 和 `QueryNested` 来分别表示这两种结构。在两种结构中，位置信息，即 `location` 和 `region` 保持原样，而 `keywords` 变为 `std::vector<size_t>` 类型，其中 `std::vector` 所存储的是该关键词在关键词词汇 `vocab` 中的索引。考虑到字符串的比较较为耗时，而整数的比较效率较高，在建树和查找的过程中可以直接对序号进行操作，而非对字符串。同时，由于 AP 树算法中有对 `keywords` 随机访问的需求，应该将其从原来的 `std::set` 转为 `std::vector` 以实现高效的随机访问。

然后介绍关键词分割方案和空间分割方案的存储表示。关键词分割方案用 `KeywordPartition` 结构体表示,其成员包括:类型为 `std::vector<KeywordCut>` 的分割序列 `cuts`,其中 `KeywordCut` 为关键词序号的闭区间;类型为 `std::map<KeywordCut, std::vector<QueryNested *>>` 的查询项查找表 `queries`，用来表示每个关键词闭区间对应哪些查询项，注意到这里使用的是 `QueryNested` 的指针

类型,这是为了节约运行时内存占用,避免 QueryNested 的多次复制;类型为 std::vector<QueryNested *> 的 dummy cut 表 dummy；还有一个表示该分割方案代价的浮点数 cost。空间分割方案用 SpatialPartition 结构体表示,其成员包括:类型为 size_t,在 X、Y 轴上个各自的分割数目 nPartX、nPartY；类型为 std::vector<double> 的分割位置 partX，partY，注意这两个 std::vector 的 size() 各自比 nPartX，nPartY 多一个，为了代码清晰，不使用它们的 size() 信息；类型为 std::unique_ptr<std::vector<QueryNested *>> 的单元格查询项序列 cells，注意这是一个一维的数组，一个二维的单元格索引需要通过运算转为一维的数组索引；剩下的 dummy 和 cost 与 KeywordPartition 相同，不作赘述。

最后介绍 AP 树的结点类型 Node,由于 AP 树有三种结点类型,而每个结点的子节点的类型父节点是不清楚的，这里使用一个 NodeType 枚举来标记每个子节点的类型。同时每个结点需要存储结点的关键词偏置 offset、区域 bound,以及是否从关键词和空间信息来划分结点 useKw、useSp。对于三种结点各自的信息，定义了 Node 的内部结构体来储存。为了节约结点空间，使用 std::unique_ptr 来实现较为灵活的管理，默认状况下为 nullptr，当结点信息完备时，再根据结点类型动态地分配相应的存储空间。q-node 类型 QueryNode 内只需要存储一个 std::vector<QueryNested> 即可。k-node 类型 KeywordNode 和 s-node 类型 SpatialNode 的成员分别与 KeywordPartition 和 SpatialPartition 成员类似,不同之处在于:KeywordNode 和 SpatialNode 内不需要存储 cost 信息,各自的查找表内存储的是 std::unique_ptr<Node> 的数组,而非 QueryNested *。同时 KeywordNode 和 SpatialNode 还存有各自所包含的查询项个数 nQry,便于重构时计算 KL 散度。由于 k-node 和 s-node 内共有 dummy cut，考虑将其放在 Node 的一个 std::unique_ptr 中，避免重复定义。这样一来，APTree 析构的时候所有内存空间可以自动回收，不需要手动 delete，也不会造成内存泄漏。

2.3 运算实现

AP 树的构建和查询算法基本根据原论文所给的伪代码来实现，以下的介绍只作大致思路的梳理和部分要点的提示。

在构造函数里，首先根据参数 vocab 将 Query 转成 QueryNested，并取址，建立 std::vector<QueryNode *>，然后从根节点开始调用私有 build 函数。在 build 函数内，若当前结点判定为 q-node，则将传入的 std::vector<QueryNode *> 内的指针全部取值储存在 Node::QueryNode 中；不然则调用私有的 keywordHeuristic 和 spatialHeuristic 尝试关键词和空间分割，取代价小的分割方式确定节点类型，将相关信息赋予结点，继续下一层递归。和原论文中算法不同之处在于，l、kP 和 sP 是作为 Node 的成员来传递的，而非通过函数的实参传递，这是为了便于动态重构。要注意，Node 的内存空间要在主调用的 build 函数内分配，其构造的提示信息 offset、bound、useKw 和 useSp 也要在 Node 构造时赋值；其成员的具体类型，以及每种类型下的数据，要在被调用的 build 函数内分配和赋值。

下面介绍 keywordHeuristic 和 spatialHeuristic 内的实现。由于它们均为启发式算法，有相似的思路。首先，对传入的查询项进行**统计**，关键词分割时分别统计当前 offset 以及所有 offset 的词频，空间分割时统计边界位置。然后，对统计的对象进行**均分**，关键词分割时均分关键词到每个 cut，空间分割时均分边界到每个 cell。接着尝试改变划分的边界，找到局部最优的划分，**改善**整体的代价。最

后，利用找到的最佳划分，将查询项**分配**到每个 cut 或者 cell 内，算法结束。

在 `Match` 函数内，首先将 `STObject` 转成 `STObjectNested`，调用私有的 `match` 函数进行匹配，匹配算法完全按照论文中的伪代码实现。将得到的 `std::set` 内的 `QueryNested` 转成 `Query`，返回给类的使用者。

2.4 查询项维护

`APTree` 还实现了查询项的动态注册。原文中只对其做了简单的介绍，没有进行详细的阐述。本文尝试给出其完整的算法伪代码，并对其中的重点问题进行探讨。

Algorithm 1: 动态注册查询项

Input: 结点 N，新增查询项 Q
Output: 新结点 N'

if N 为 *q-node* **then**
  if $|N.Q.\psi| + |Q.\psi| < \theta_q$ **then**
    $N.Q \leftarrow$ 归并 $N.Q$ 和 Q
    返回 N
  else
    $Q' \leftarrow$ 归并 $N.Q$ 和 Q
    构建索引 (N', Q')
    返回 N'
  end
else
  统计 Q 的信息
  $D_{KL} \leftarrow$ 计算 KL 散度
  if $D_{KL} > \theta_{KL}$ **then**
    $Q' \leftarrow$ 归并 $N.Q$ 和 Q
    构建索引 (N', Q')
    返回 N'
  else
    $Q_D \leftarrow$ 寻找 Q 中的 dummy 查询项
    if Q_D 非空 **then**
      if N 有 *dummy cut/cell* N_D **then**
        $N'_D \leftarrow$ 注册 (N_D, Q_D)
        重置 N 的 dummy 结点为 N'_D
      else
        构建索引 (N_D, Q_D)
      end
    end
    foreach N 中分割 B_i **do**
      $Q_{B_i} \leftarrow$ 在 Q 中寻找匹配 B_i 的查询项
      $N'_{B_i} \leftarrow$ 注册 (N_{B_i}, Q_{B_i})
      重置 N_{B_i} 的结点为 N'_{B_i}
    end
    返回 N
  end
end

动态注册查询项的算法如上述伪代码所示。由于现有的函数已经能够根据一个结点的初始信息和待构建的查询项构建一棵子树，该算法主要解决的是一个结点要不要重构，以及重构后如何让父结点指向重构后的结点的问题。

首先要回答的是要不要重构的问题。原论文中，作者提到要用分割的 KL 散度 $D_{KL}(\mathbf{w}_{old}|\mathbf{w})$ 是否超过阈值 θ_{KL} 来判断。对于定义在相同概率空间上的离散概率分布 P 和 Q，它们的 KL 散度定义为

$$D_{KL}(P|Q) = -\sum_{x\in X} P(x)\log(\frac{Q(x)}{P(x)})$$

在 AP 树的理论中，对每个结点的分割 B，则有

$$D_{KL}(\mathbf{w}_{old}|\mathbf{w}) = -\sum_{i=1}^{f} \mathbf{w}_{old}(B_i)\log(\frac{\mathbf{w}(B_i)}{\mathbf{w}_{old}(B_i)})$$

实际实现中，要对上述公式变形。注意到分割 B_i 的新权值（即查询项数目）为原先的权值和新增权值之和 $\mathbf{w}(B_i) = \mathbf{w}_{old}(B_i) + \mathbf{w}_{add}(B_i)$。取上式的绝对值，并将 $\mathbf{w}(B_i)$ 的表达式代入，可得

$$\begin{aligned}|D_{KL}(\mathbf{w}_{old}|\mathbf{w})| &= \sum_{i=1}^{f} \mathbf{w}_{old}(B_i)\log(\frac{\mathbf{w}_{old}(B_i)+\mathbf{w}_{add}(B_i)}{\mathbf{w}_{old}(B_i)}) \\ &= \sum_{i=1}^{f} \mathbf{w}_{old}(B_i)\log(1+\frac{\mathbf{w}_{add}(B_i)}{\mathbf{w}_{old}(B_i)})\end{aligned}$$

在上述公式中，在 $\mathbf{w}_{old}B_i$ 确定的情况下，如果 $\mathbf{w}_{add}(B_i)$ 越大，$|D_{KL}(\mathbf{w}_{old}|\mathbf{w})|$ 就会越大，那么该结点就更有可能被重构，这是符合我们计算 KL 散度的目的的。

然后解决如何重构后父子结点关联的问题。这里利用动态注册函数能够返回一个新的节点指针来实现：如果节点需要重构，则返回重构和新的结点的指针，否则则返回原子结点的指针。在父结点的注册函数中，如果发现子结点的注册函数返回了和之前子结点不同的指针，则更新新的子结点指针，否则则保持原指针。

在具体实现中,为了节省内存占用,传入注册函数的 Q 实际上是 `const std::vector<QueryNested *> &`,该算法一定能保证存储新的查询项时,这些指针都能顺利取值。原因要从 `Node` 和 `QueryNested` 的生命周期来找。将 `QueryNested` 的指针“落实”为实际的值只有两种途径：一是动态注册函数里将 $N.Q$ 和 Q 归并，二是在构建索引函数里将 Q 存进 q-node。根据上述算法，这两个动作均发生在重置结点之前。那么在执行重置子结点之前，可以保证原先的子树保持原样，而新的子树已经建立完毕。这样只要调用 `std::unique_ptr` 里的 `reset` 方法，即可将原先的子树完全析构，并指向新的子结点。

3 性能测试

4 评价

4.1 优势

原论文中也提了几种其他用于解决类似问题的结构，如 $\mathrm{R^t}$ 树和 IQ 树。它们有一个共通点：无论查询项中关键词和空间信息有何特点，在建立索引时空间因素始终优先于关键词因素。相比而言，AP

树有以下几点优势：

1. AP 树在处理不同的数据集时匹配对象的速度快。因为对有些数据集来说关键词和空间两种匹配模式的效率差别很大，AP 树可以很好地利用它的自适应性选择最优的匹配方式。
2. 当 AP 树应对较多查询项时，依然能保证较高的效率。
3. AP 树可以自动维护它的结构。当工作量发生变化时，它可以根据阈值重构自己的结点，体现出它的自适应性。

4.2 不足

1. AP 树对于空间的需求量较大。在存储实现中可以看到，尽管对于结点的存储方案已经做了一定优化，但是单个结点占用空间依然较大。在树的构建过程中，每一层递归都需要储存对应结点查询项的统计信息。当查询项的总数较大时，势必造成内存占用的激增。在我们的测试中，在注册了 5000 个查询项后，内存占用就超过了 90MB。如果是实际应用中上百万的查询项，其内存占用是相当巨大的。
2. AP 树的构造较为复杂，建树的耗时较长。AP 树由于要对两种分割模式进行混合，就要比其它分割树多消耗一倍以上的计算时间。如果一个数据库系统需要快速地对内部的元素进行迭代、更新，那么 AP 树显然不适合这样的场景。

5 未来工作

由于时间和开发者能力所限，该项目还有许多不完善的地方，在将来可能会对以下方面进行改进或补充。

1. 我们的实验中只实现了 AP 树一种结构，并未实现论文中提到的其他空间文本分割结构，所以无法进行比较，不能体现 AP 树的优势。今后考虑实现 IQ 树、R^t 树、顺序关键词字典树等结构，进行横向比较。
2. 当前的 `APTree` 尚不具备删除结点的能力，不过在已有函数的基础上可以相对便利地实现这一功能，在今后考虑补充这一功能。
3. 我们当前的测试只考察了 `APTree` 的执行时间，分析了时间复杂度方面的性质，对于其内存占用的情况只做了定性的分析与简单的记录。今后考虑直接使用或设计相应的测试工具，对其内存的使用情况进行具体的跟踪。

A 源代码

本文所实现的 AP 树源代码仓库在 GitHub 上进行托管：`https://github.com/wzh99/AP-Tree`，采用 MIT 许可证。

B 数据集来源

本文测试所使用的关键词数据集来自 US Board on Geographic Names（`http://geonames.usgs.gov`）

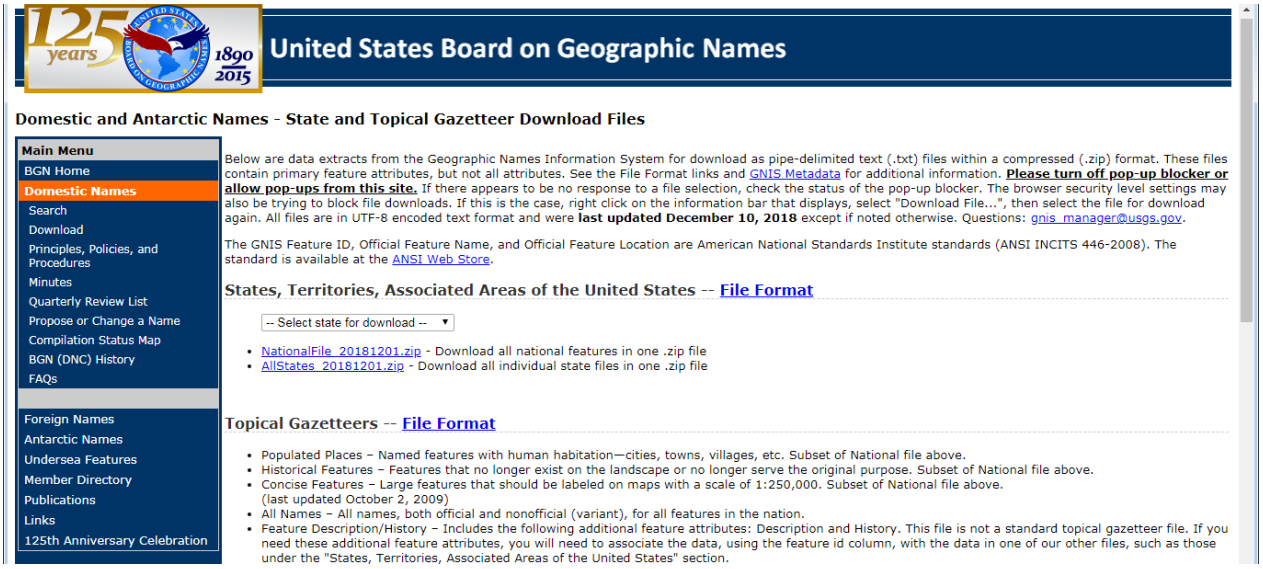

图 1: US Board on Geographic Names 网站

C 分工

- **代码实现**：王梓涵
- **数据测试**：刘权
- **报告撰写**：周昕逸　王梓涵　刘权