HolisticTwigStack数据查询算法的实验研究

李希萌

摘　要

采用XML语言对数据进行描述会形成一种较为灵活和复杂的树形结构，查询其中存在的特定结构模式的问题是一类重要问题，即小枝查询问题。由于采用朴素的算法会导致巨大的计算量，解决此类问题需要的高效算法正处于广泛研究中。HolisticTwig-Stack算法便是效率较高的小枝查询算法中的一种。

本文探讨了基于HolisticTwigStack算法的小枝查询系统的设计和实现方法，为支持系统功能对该算法进行了部分改进，并论证了改进后的算法的正确性。此外，在HolisticTwigStack算法的基础上提出了一种XML流数据小枝查询算法，对后者的正确性进行了论证，并探讨了基于该算法的流数据小枝查询系统的设计和实现方法。

本文分析了小枝查询系统的性能测试结果，用该结果支持了系统内部算法设计和改进的有效性。

关键词 小枝查询；HolisticTwigStack；流处理

Abstract

The adoption of Extensible Markup Language (XML) in the description of data leads to a flexible and complex tree structure. A typical problem is to find all occurrences of some structural pattern in such trees, which is called the twig query problem. Naive approaches will result in a drain on computational power when some complexity involves in such problems. Therefore twig query algorithms are extensively studied, among which HolisticTwigStack is a relatively efficient one.

In this paper, the design and implementation of a twig query system based on HolisticTwgiStack algorithm is presented. To support some of the system's functionalities, some refinements are made on the original algorithm, of which the proof can be found in the corresponding chapter. Besides, based on the HolisticTwigStack algorithm, a new algorithm with the ability to perform twig queries on streamed data is proposed and proved. And the design and implementation of a stream-processing system on the basis of that algorithm is discussed.

Experimental results concerning the performance of the two systems support the validity of the algorithmic refinements and the effectiveness of the proposed stream-processing algorithm.

Keywords Twig Query; HolisticTwigStack; Stream-processing

目 录

1. 引言

XML语言全名为“可扩展标记语言”(Extensible Markup Language)[8]。它与HTML语言一样是一种标记语言。两种语言中的标记均是用来描述数据的。但HTML语言中的标记对数据的描述仅仅以按照特定的格式显示数据为最终目的。由于用途固定，HTML语言的语法也是固定的。相比之下，XML语言本身没有限定其中的标记对数据进行描述的目的，因而可以适用于几乎任何用途的信息交换。XML语言在语法上有着一些基本的限定，如标记必须严格嵌套等。在满足语法上的基本规定的前提下，只要交互的双方（应用程序或人）对某一组标记和某些关于这组标记之间组合方式的约束有着共同的理解，就可以在此基础上进行信息交换。这时，这组特定标记以及它们相互之间组合方式的约束条件就构成了一种新的语言。该语言可视为在XML语言的基础上派生得到。因而XML语言的名称中包含“可扩展”的意思。这种“可扩展”性使得这种语言的应用非常广泛，如数据库、应用间的通信、程序规格和用户界面的描述等等。

由于XML语言在描述数据方面应用的广泛性，在XML数据中查询目标数据的技术处于广泛研究中。类似于在关系型数据库中常用于表示查询请求SQL语言，在XML数据库中，用户往往采用XPath[6]、XQuery[7]等语言进行查询。由这两种语言所指定的查询要进行转化和优化，以提高查询的执行在时间和空间上的效率。转化得到的中间表示将交给核心的数据查询算法去执行。这时就会涉及到小枝查询问题和小枝查询算法。

1.1 小枝查询问题及相关算法研究

1.1.1 小枝查询问题

小枝查询问题的提出根源于XML语言描述数据的特点。在关系型数据库中，对于确定的二维表，其中的数据可以划分为元组。而元组中各个域所表示的意义都是预先由关系模式确定的。任何两个元组，相同位置的字段具有相同的意义。

在XML数据库中，XML文档具有树形结构。图1-1分别在(a)和(b)中表现了某个XML文档及其所具有的树形结构(没有在树中画出标记中的文本所对应的孩子)。后者称为XML文档树。像在关系型数据库中将二维表视为由元组组成那样，在XML数据库中可将XML文档视为由子树组成。不过，子树中既存在数据元素(在图1-1中没有画出)，又存在描述数据的元素。也就是说，在XML数据库中，描述数据的信息并不能完全独立地抽取出来，形成像关系模式那样的元数据，而是与其所描述的数据并存于文档中。

图1-1中的XML文档描述了一些关于“论文”的数据。其中，temperature and growth of vegetables被描述为某篇论文的标题。Bob和Mike被描述为该论文的作者。根据描述，另一篇标题为environmental factors in the growth of vegetables的论文引用了第一篇论文。该论文的唯一作者为Jack。

可以看出，描述数据的文档元素本身存在于一种树形结构中，但该结构不像关系数据库中的关系模式那样是完全固定的。譬如，未必每一篇论文都被其他论文所引用，相

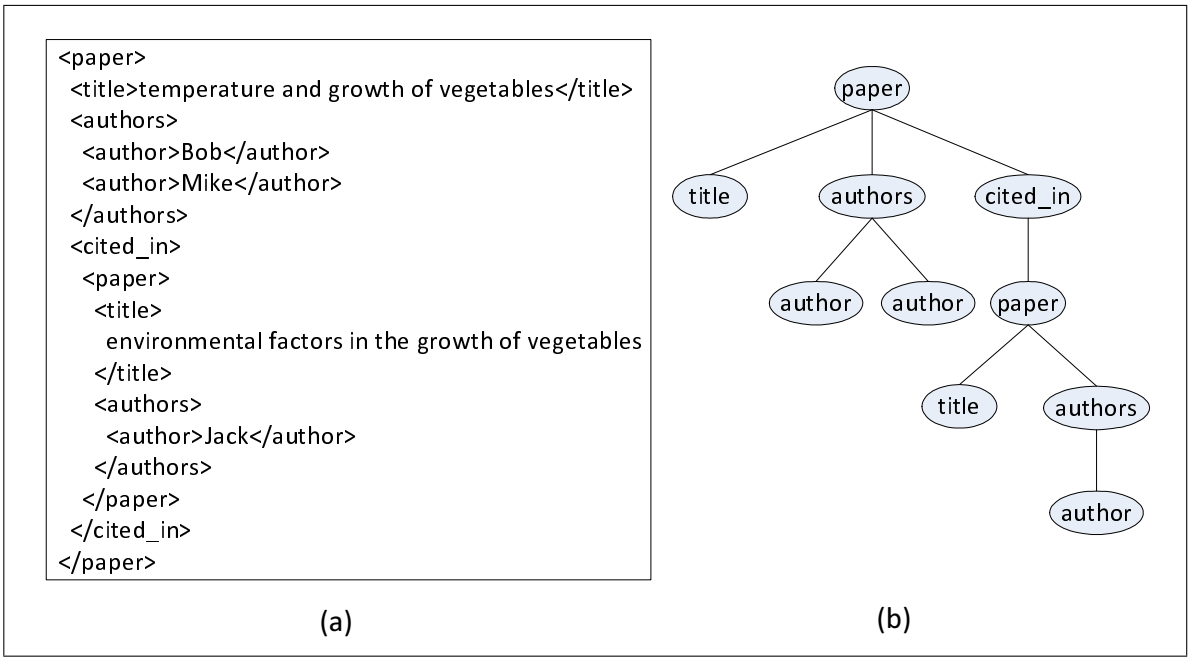

图 1-1　XML 文档和文档树

应地，paper 不一定有孩子 cited_in。因此，这种描述数据意义的树形结构本身便可以成为查询的对象。经常可以遇到与下面类似的 XQuery 查询程序：

```
declare variable $doc = document(“Papers.xml”)
for $pa in $doc/paper
   where $pa[title]/cited_in/paper/title
   return <titles>
              <title1>$pa/title</title1><title2>$pa/cited_in/paper/title</title2>
          </titles>
```

意思是查询论文标题及引用该论文的论文标题的组合。这个查询进行转化后可以抽取出一种结构上的模式，就是——标签名为 paper 的元素内部需要嵌套标签名为 title 的元素、以及标签名为 cited_in 的元素，后者内部又要嵌套标签名为 paper 的元素，后一个 paper 内部又要嵌套标签名为 title 的元素。注意到这种模式本身也可以被表达为一种树形结构(如图1-2)，因而称为小枝模式。这种在 XML 文档描述数据的树形结构中查询指定的小枝模式的问题就是小枝查询问题。

小枝模式可以用模式树表示。譬如，对上面的问题，小枝模式可由图1-2(a)的模式树表示。在模式树中，若父子结点之间由单线连接，表示以子结点名称为标签名的 XML 文档树元素 f 在文档树中必须为以父结点名称为标签名的 XML 文档树元素 e 的孩子，两者在文档树中的层数必须只差1。若允许放宽层数条件的限制，只要求 f 为 e 的后代，则在模式树中用双线连接父子结点。称单线表示的关系为 AD 关系(ancestor-descendant relationship)[5]；而称双线表示的关系为 PC 关系(parent-

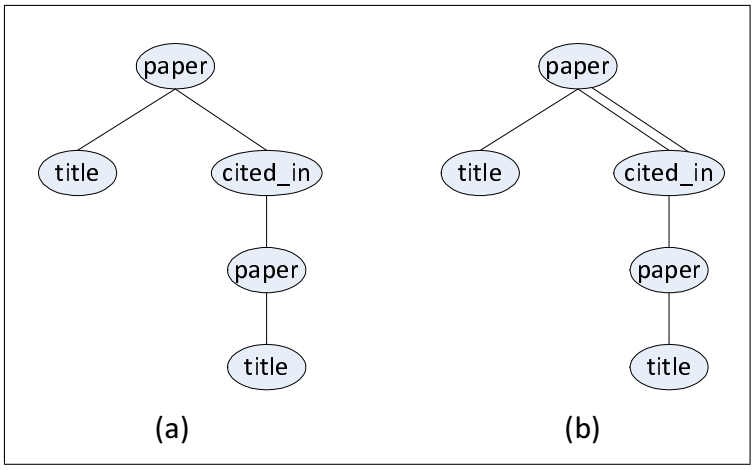

图 1-2　小枝查询中的模式树

child relationship)[5]。如果查询目标变为——论文标题与该论文的被引用链上的某篇论文标题的组合，那么能够表示该目标模式的模式树中，paper 结点和 cited_in 结点就会由 PC 关系变为 AD 关系。如图1-2(b)所示。对本例而言，这两个查询均只能返回一组结果(其中包括了所有的标签名为 papaer、title、cited_in 的元素)，因为对最外层的 paper 而言，引用它的论文不再被另外的论文引用了。

1.1.2 小枝查询算法

由于在 XML 文档中大量的查询问题可以归结为小枝查询问题或者类似的问题。返回小枝模式在文档中匹配结果的小枝查询算法处于广泛研究之中。早期的比较经典的算法为 TwigStack 算法[3]，该算法将小枝模式分解为从根到叶的许多不同路径，在一种栈结构中保存这些不同路径在文档中的匹配结果。该算法将这些路径匹配结果作为中间结果输出，再将它们进行组合，得到最终的小枝匹配结果。在 TwigStack 算法中，先通过过程 getNext(Q)对能够进入栈结构的元素进行“筛选”，防止了大量无用元素入栈。但由于该算法是先生成模式中单个路径的匹配结果，在进行这些中间结果的存储和组合时存在巨大的时间和空间开销。为避免中间结果及其归并过程，研究者提出了 Twig^{2}Stack 算法[5]，该算法在一种层级栈结构（hierarchical stack）中保存参与最终匹配结果的候选栈元素。具体说，该算法为每一个模式树结点都建立一个栈树，模式树结点 Q 的栈树记为 HS[Q]。若 Q 在模式树中有孩子 R，则 HS[Q]在层级栈结构中有子栈树 HS[R]。从栈树 HS(Q)开始沿着栈树间的连接进行枚举所得到的结果是模式树中以 Q 为根的子树在文档中的匹配结果。Twig^{2}Stack 算法的特点在于，HS(Q)中的元素 e_Q必定参与模式树中 Q 为根的子树在文档中的匹配结果，但不一定存在参与完整匹配结果的父类型元素作为 e_Q的双亲或祖先(也称为 covers，即盖住)。这样，层级栈结构中仍会有许多不参与最终解的无用元素。另外，在该种栈结构中，存在一些空栈，用来作为栈树的根，使栈树能被其他栈树中的元素所引用。这些空栈本身及它们的构造又导致了空间和时间上的开销。在 Twig^{2}Stack 算法之后出现的 TwigList 算法[4]同样保证了在存

储结构中存放的元素必定参与模式树中该类型结点的子树在文档中的匹配结果(但未必参与最终结果)。该算法对存储结构做了较大改进，不再为每个模式树结点设立一个栈树，而是仅仅建立一个线性表。在其中存放候选解元素。TwigList 算法及从该算法派生的算法是目前查询速度最快的一类算法。但在存储空间使用上，TwigList 没有保证存入其存储结构的元素参与整个小枝模式的匹配结果，因而在一定程度上浪费了存储空间。本文标题中的HolisticTwigStack 算法[1]是在TwigStack 和Twig^{2}Stack 算法之后，与TwigList 同一时期提出的小枝查询算法。

1.2 区域编码

小枝查询算法需要寻找相互位置关系满足模式树中规定的XML 元素集合，为此就需要对文档元素位置关系进行判断。为了提高判断速度，往往在读入文档元素时对其进行反映位置的编码。其后在算法运行过程中，完全通过两个元素的编码判断它们的位置关系。

常用的一种编码是“区域编码”。具体地说，每个文档元素关联一个三元组(LeftPos, RightPos, Level)，代表元素在文档树中的左位置、右位置和所在层数。“左位置”和“右位置”这两个概念可以较严格地做如下定义。若设置一个计数器，令其初值为0，之后对XML 文档树进行先序遍历，当进入以任何文档元素e 为根的子树时，令计数器自增1，当访问完e 的子树，要离开该子树时再令计数器自增1，则进入时计数器自增后的值和离开时计数器自增后的值就分别是e 的左、右位置。

区域编码中左位置、右位置的生成如图1-3所示。文档树中，元素左边的数字为其左位置的值，元素右边的数字为其右位置的值。

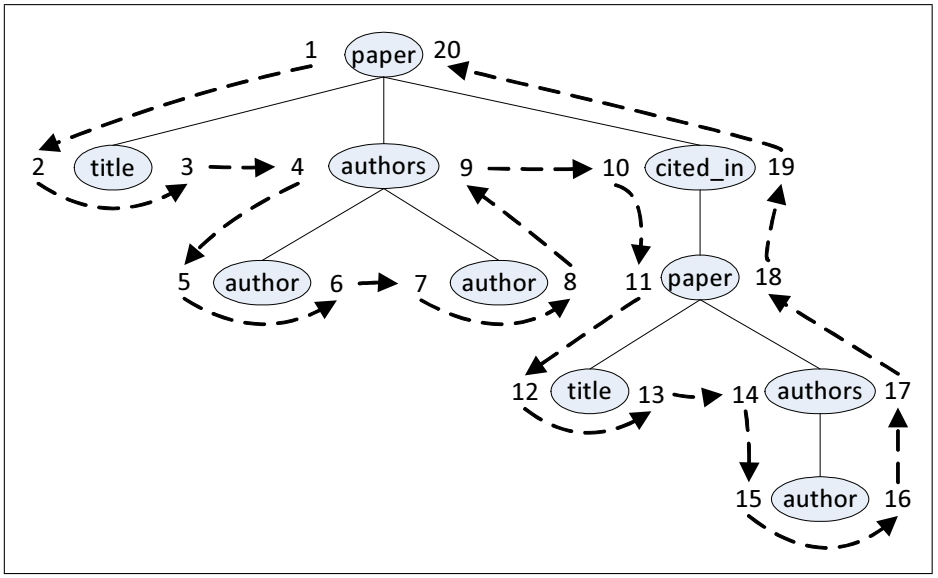

图 1-3　小枝查询中的模式树

通过区域编码可以判断两个元素是否具有“盖住”关系以及两个元素盖住的区

域是否相交。如果是“盖住”关系，还可以进一步判定它们是父子关系还是仅仅是更宽泛的祖先——后代关系。元素e_1盖住e_2当且仅当$L(e_1)<L(e_2)<R(e_2)<R(e_1)$。而元素$e_1$是$e_2$的父元素当且仅当$L(e_1)<L(e_2)<R(e_2)<R(e_1)$且$Level(e_1)+1=Level(e_2)$。其中L()表示取得左位置，R()表示取得右位置，Level()表示取得层数。值得注意的是，元素e_1盖住e_2还有另外一个等价条件：$L(e_1)<L(e_2)<R(e_1)$。这个等价条件无需用到e_2的右位置，在某些情况下会带来方便。

1.3 HolisticTwigStack算法及其特点

HolisticTwigStack算法[1]是本文工作的基础，该算法采用一种栈结构作为小枝匹配结果的压缩表示。像TwigStack算法一样，本算法使用过程getNext(Q)筛选可以进入栈结构的元素。但在本算法中栈结构中存放的并非路径匹配结果。在模式树中不存在PC关系的情况下，可以保证栈结构中存放的元素均为参与最终匹配结果的元素。与Twig^{2}Stack和TwigList算法相比，HolisticTwigStack算法在中间存储结构中存放的元素更少。不过getNext(Q)过程[3]本身具有递归结构，并且为了取出每一个文档元素都需要运行一次该过程，导致HolisticTwigStack算法在该过程内部的时间开销较大。

1.4 本文主要工作

本文作者基于HolisticTwigStack算法实现了小枝查询系统TwigMiner(其中的Miner取“挖掘者”的意思)。为支持该系统的功能，对算法做了一些改进，并对改进后的算法的正确性进行了论证。此外，基于HolisticTwigStack算法提出了一种在XML流数据上运行的小枝查询算法——SHolisticTwigStack(其中“S”是stream的第一个字母，代表流)，对该算法的正确性进行了论证。作者基于SHolisticTwigStack实现了流数据小枝查询系统STwigMiner。

1.5 本文结构

第2章比较详细地介绍了HolisticTwigStack算法。第3章讨论了为实现TwigMiner系统对该算法进行的改进，并对改进算法的正确性进行了论证。第4章提出了流数据小枝查询算法SHolisticTwigStack，对该算法正确性进行了论证。第5章介绍了基于HolisticTwigStack算法和SHolisticTwigStack算法实现的小枝查询系统TwigMiner和STwigMiner的设计和实现。第6章介绍了系统的测试工作。

2. HolisticTwigStack 算法

HolisticTwigStack 算法是一种在 getNext(Q)[3]过程的帮助下从文档元素序列中取出元素并使用栈结构保存解元素的整体性小枝匹配算法。本章对这一算法做简要介绍，因为它是小枝查询系统和后文中提出的流数据小枝查询算法的基础。

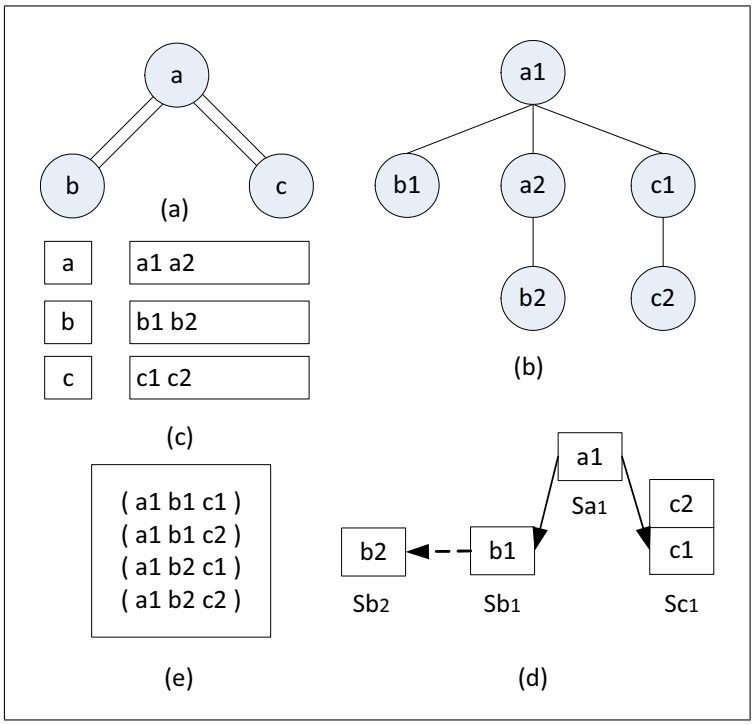

图 2-1 HolisticTwigStack 算法示例

2.1 文档元素序列

HolisticTwigStack 算法的运行需要建立在将原始的 XML 文档数据转化为多个文档元素序列的基础上。每个文档元素序列关联一个模式树结点。起初，与模式树结点 a 关联的文档元素序列中容纳标签名为 a 的所有文档元素。这些文档元素按照在文档中出现的先序顺序存放在序列中。随着算法的运行，序列中的元素或被取出、或被跳过。被取出的元素可能进入栈结构，也可能被舍弃。图2-1中对应于(a)的模式树和(b)的文档树的文档元素序列如(c)中所示。下面，用 Seq(Q)表示模式树结点 Q 所关联的文档元素序列。用 First(seq)表示序列 seq 的首元素。

2.2 getNext(Q)

HolisticTwigStack 算法以模式树的根结点为参数运行 getNext(Q)，该过程会返回一个模式树结点，之后算法从该结点的文档元素序列中取出其首元素。

getNext(Q)返回的模式树结点保证：(1)对于从该模式树结点的文档元素序列头部中取出的元素而言，它在文档树中拥有模式树中指出的各类型的孩子元素，且那些孩子元素出现在它们所属序列的头部；(2)那些孩子元素递归地满足条件(1)。这在[1]中称为具有“最小后代扩展”(minimal descendent extension)。元素具有“最小后代扩展”是元素参与解的必要条件。用getNext(Q)保证取出的元素满足这个条件就避免了对很大一部分不参与解的元素进行入栈操作的开销。

若在模式树中，Q∈Children(P)，那么getNext(P)将对seq_1=Seq(Q)和seq_2=Seq(P)的首元素进行考察。若∀Q∈Children(P)，First(seq_2)盖住First(seq_1)，那么getNext(P)将会返回结点P。若∃Q∈Children(P)使得L(First(seq_1))<=L(First(seq_2))，那么返回所有P的孩子Q中其序列首元素的左位置最小的那一个。若∃Q∈Children(P)使得L(First(seq_1))>R(First(seq_2))，就跳过seq_2头部的元素，直到该不等式不再成立为止。

整个模式树可以看成许多这样的具有一层父子关系的子树在一些公共结点上连接而形成的。getNext(Q)处理以Q为根的模式树时将递归地运行，在每一个这样的子树中进行这种判断，以决定向上一层返回父结点还是子结点。如果决定返回子结点，则在所有的上层子树中不再进行判断，直接将该子结点层层向上返回。

图2-1的例子中，第1次调用getNext(a)时，Seq(a)、Seq(b)和Seq(c)的首元素有关系a_1盖住b_1和c_1。因此该调用返回a。First(Seq(a))=a_1将被从序列中取出。第2次再调用getNext(a)时，Seq(a)中的首元素变为a_2。这时对c_1和a_2有关系L(c_1)>R(a_2)。这导致a_2被跳过。这样Seq(a)变空，其下一个元素的左右位置可以认为均是∞。这样，a的孩子结点b和c的序列中首元素的左位置均小于∞。由于L(b_1)<L(c_1)，本次getNext(a)返回结点b。First(Seq(b))=b_1将被取出。第3次调用getNext(a)，Seq(a)、Seq(b)、Seq(c)的下一个元素的左位置分别为：∞、L(b_2)和L(c_1)，由于L(b_2)<L(c_1)，再次返回结点b。元素b_2从序列中取出。第4次调用getNext(a)，Seq(b)下一个元素的左位置也变为∞，易知本次及再下一次调用getNext(a)都将返回c结点，元素c_1和c_2依次被取出。

可以发现，a_2缺失c类型的孩子元素，因而不具有“最小后代扩展”(当然也就不参与任何解)，它恰恰在getNext(a)中被跳过了。这防止了该元素被取出，作为无用元素进入栈结构导致的时间和空间上的浪费。

2.3 栈结构和moveElementToStack(E, e)

HolisticTwigStack算法所采用的栈结构中，栈也是与模式树结点相关联的，与栈关联的模式树结点也可称为栈的类型。如图2-1中有三种类型的栈，每种类型的栈有一个或多个。Q类型栈中存放的是Q类型的解元素。且同一个栈中靠近栈底的元素是靠近栈顶的元素的祖先(如c_1是c_2的祖先)。栈元素间有连接。总共有两种类型的连接，一是父类型栈元素到子类型栈元素的连接(在图中以实线箭头表示)，二是同种类型的兄弟栈元素之间的连接(在图中用虚线箭头表示)。由第二类连接连通的子类型栈

中的元素通过实线箭头连接到的父类型栈元素称作它们的“最近模式祖先”(closest pattern ancestor)[1]，意思是“根据查询模式，元素e在文档树中的最近祖先”[1]。如在图2-1(b)中，b_1和b_2的最近模式祖先均为a_1。但如果a_2下面增加一个c类型孩子元素，使a_2变为解元素，则b_1的最近模式祖先仍然为a_1，而b_2的最近模式祖先将会变成a_2。

在这样的栈结构中，栈元素将继承(inherit)其后代所拥有的指向各个类型孩子元素的连接。如在图2-1(d)中，若a_1上方还有元素a_2，则a_2的指向孩子的连接将被a_1继承。这是因为a_2的后代也必然是其祖先a_1的后代。

算法moveElementToStack(E, e)[1][2]用来使从文档元素序列中取出的元素e∈Seq(E)入栈。

从文档元素序列中取出元素后，运行入栈算法之前，需要对一个条件进行判断，那就是：父类型栈中是否有元素盖住当前元素。在图2-1的例子中，b_2取出后，要判断a类型栈中是否有元素盖住它。事实上，a_1就是这样的元素。这样b_2将可以进入栈结构。试想，若a_2为根的子树变为a_1为根的子树的邻居，b_2取出后将不被a类型栈中的唯一元素a_1盖住，这样b_2将不会进入栈结构。在这种情况下，可以发现b_2恰好缺少一个参与解的父类型元素盖住它。b_2的子树虽然与模式树中b结点为根的子树匹配，它却缺少合适的父元素，以形成更大范围内的匹配。这个判定同样可以防止一部分非解元素进入栈结构。

moveElementToStack(E, e)的设计思想是：首先检查e是否为E类型最后一个栈的栈顶元素的后代，如果是，则直接将e压入该栈。如$e=c_2$时就是这种情况。否则计算出e的最近模式祖先CPA(e)。如果CPA(e)还没有E类型的子栈，就新建一个包含e的栈，并建立由CPA(e)到e的连接；否则须将e放到CPA(e)拥有的由第二类连接所连通的一个或多个子栈所组成的结构中的合适位置，很可能同样需要为e建立新栈。如$e=b_2$时就是这种情况。

2.4 结果枚举算法

结果的枚举是沿着栈间的两种连接，以及同一个栈中的祖先后代关系进行的。在图2-1的例子中，为了枚举a_1参与的结果，首先枚举其孩子栈元素参与的部分结果，再将a_1与部分结果(b_1, b_2)和(c_1, c_2)进行组合。另外如果a_1在栈中还有后代元素或者在栈结构中还有兄弟元素，要对它们运行同样的算法，将所得到的解合在一起，才是完整的小枝匹配解。如果a_1在它的栈中具有后代元素a_X，则a_1本身也将具有a_X所具有的各类型孩子元素。因而那些类型的孩子解也需要与a_1进行组合。由上面的描述知结果枚举算法具有递归结构，并且比较复杂。这里不再详细地写出该算法，请参见[2]。

3. HolisticTwigStack 算法的改进

在提出HolisticTwigStack 算法的参考文献[1]和[2]中并未讨论处理含有PC 关系的小枝查询的方法、高效地寻找最近模式祖先的方法、以及按顺序输出小枝匹配结果的方法等。另外，事实上[1]和[2]中栈结构构造算法使用的一个优化仍存在着问题。为实现一个完整、健壮的小枝查询系统，须对这些问题进行讨论和解决。为此，本章对该算法进行扩展和改进。

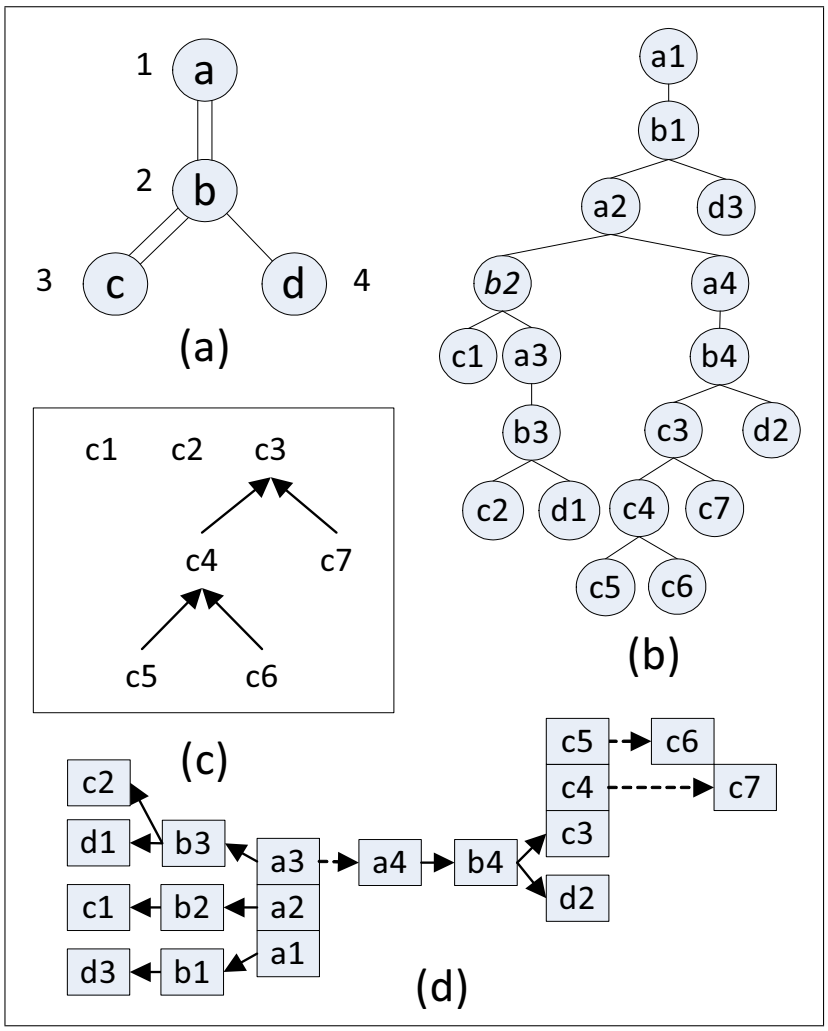

图 3-1 HolisticTwigStack 算法的改进

3.1 改进算法的输入和输出

改进的HolisticTwigStack 算法以模式树PT、XML 文档元素序列的集合$\mathbb{S}_{seq}$作为输入。模式树中允许父子结点之间具有AD 关系或PC 关系。

该算法按顺序输出PT在$\mathbb{S}_{seq}$上的所有正确的小枝匹配结果。每个结果Res形如(m_1, m_2, ..., m_k)，其中k与PT中结点个数相等。该结果中某个确定位置上的元素与PT中确定位置上的结点匹配。对相邻两个匹配结果(m_1, m_2, …, m_{k-1}, m_k)和(m'_1, m'_2, … m'_{k-1}, m'_k)，若令m_i<m'_i当且仅当Fir(m_i)<Fir(m'_i)，其中Fir()表示取得文档元素的先序位置，那么这两个匹配在系统输出中的先后顺序与串$m_1m_2\cdots m_{k-1}m_k$和$m'_1\ m'_2\cdots m'_{k-1}\ m'_k$按字典序的排列顺序一致。

3.2 栈结构构造算法的改进

在[2]中提出的栈结构构造算法和结果枚举算法并不十分一致，而后者暗示另一种栈结构的可能性。在2.3节中，由于没有涉及到太多细节，这个问题没有显现。本节先讨论在两种不同栈结构中的取舍。由于栈结构决定了其构造算法，栈结构的选择是本算法一切改进的基础。之后讨论PC关系的处理、高效寻找最近栈内祖先的方法、以及对原来的栈结构构造算法中所用优化的改进。而后给出改进的算法，并对算法正确性进行论证。

3.2.1 两种不同的栈结构

HolisticTwigStack算法中，需要采用某种机制，为某个栈元素及其在解中的父元素和子元素建立通路。首先，我们可以由元素e_Q引出到其在解中先序位置最小的第一个子元素$e_{Children(Q)}$的连接，或到该子元素所在栈的连接。这种连接是由栈元素引出的，而具体终止于子栈的栈底元素还是终止于子栈，则对正确性没有影响。因为由子栈可以找到其栈底元素，而如果在栈元素中添加一个记录其所属栈的域stack，则反过来也可以从栈元素找到其所属的栈。其次，要找到元素e_Q的后续子元素，需要将那些子元素与e_Q相连。或者将所有子元素彼此相连，使得包括第一个子元素在内的所有子元素或它们的栈互相连通。[1]和[2]中采取的都是后一种做法。然而，关于同种类型的元素之间的具体连接方式却大体可以有两种，如下：

1. 将同种元素之间的连接处理成栈和栈之间的连接。

2. 将同种元素之间的连接处理成栈和栈元素之间的连接。

采用第一种方法时，若进行图3-1中所示的查询，所有以b_4为最近模式祖先的c_i的栈可以组成一个线性表，而不是(d)中像图一样的结构。即只需记录：栈(c_3, c_4, c_5)的后继为栈(c_6)，而后者的后继又是栈(c_7)。一旦发现最近从文档元素序列取出的元素与之前的一个元素盖住的区域不相交时，就为新元素构造一个新的栈，并将该栈附在线性表尾部即可。

而采用第二种方法时，需要由具体的栈元素引出指向新栈的连接，使得引出连接的元素与新栈的栈底元素在文档树中是兄弟。这里所说的“兄弟”的意思是，在文档树中只保留该种元素及其相互间的盖住关系后所形成的森林中的兄弟关系。如在图

3-1(b)所示的文档树中，仅仅保留所有Q_a，则可以发现a_4是a_3的兄弟，它们有共同的父元素a_2。因而在栈结构中由栈元素a_3引出连接，指向a_4。

易见，采用第一种方法时，入栈操作会比较高效。c_6、c_7到来时，只需将新建的栈连到已有栈组成的线性表尾部。不必再计算其具体需要连接到哪个元素(如c_7连接到c_4而不是c_6)。然而，采用这种连接方式会给枚举结果的算法带来负担。事实上，在判断属于结果的元素组合时，需要考虑的不仅仅是某个新栈的栈底元素是否与上一个栈的栈顶元素盖住的区域不相交。如图3-1所示的例子中，c_5与c_4均不盖住c_7，而更靠近栈底的c_3盖住c_7，这直接导致了c_7的孩子（虽然在本例中不存在，但完全有可能存在）可以作为c_3的孩子，但却不能作为c_4和c_5的孩子。这一点仅仅由c_7与c_5盖住的区域不相交是无法判断的。因而若采用第一种方法，虽然在入栈阶段不必计算同种类型元素内部的兄弟关系，但在结果枚举算法中还要计算，复杂性只是转移了，而没有减少。而且在3.3节将会看到，按顺序进行结果枚举的算法已经相当复杂。因此，在小枝查询中采用第二种栈结构，也就是图示的那一种。

在这样的栈结构中，每个栈都具有与其关联的模式树结点，栈间通过具体的栈元素相互连接。栈元素间的连接有两类，一类是父元素与其第一个子元素间的连接，另一类是同一序列中取出并入栈的不同子元素之间的连接。这两类连接在图3-1(d)中分别用实线和虚线箭头表示。完全通过第二类连接相互可达的元素（如图3-1(d)中的c_3和c_7）是具有相同“最近模式祖先”的元素。这些元素组织在一个或多个栈中，如果某两个元素之间存在“兄弟”关系，则它们之间存在显式连接。

这样连接后形成的类似“图”的结构实际上是森林的孩子-兄弟表示法的一种变形。其中，由栈元素e开始，沿着栈的方向向上找到的第一个元素是e的第一个孩子。若e还有其他孩子(即第一个孩子的兄弟)，则那些孩子可以沿着由第一个孩子引出的第二类连接找到。除此以外，不同的树的根之间也可能具有第二类连接。这里所说的孩子，兄弟和树都是将整个文档树中某种类型的栈元素单独抽取出来后，在所形成的森林中出现的概念。如从图3-1(b)所示的文档树中单独抽取出a类型的元素所得到的森林如(c)所示(确切地说(c)中两个元素之间的箭头去掉后才表示森林，箭头是为了说明另外一个概念，与这里讨论的内容无关，如后文所述)。 在(c)所示的森林中，a_2的第一个孩子是a_3，与此相应，在栈结构中，a_2上方的第一个元素就是a_3。而a_2还有另外的孩子，即a_3的兄弟——a_4。在栈结构中，表现为由a_3引出了指向a_4的第二类连接。另外，若在文档树中存在a_5，使得a_1和a_5是森林中两棵不同树的根，它们也具有“兄弟”关系，在栈结构中反映为存在由a_1引出，指向a_5的第二类连接。由于这种同种类型且具有相同“最近模式祖先”的栈元素的相互连接实质上是在反映它们在文档中的树形结构(实际上为森林)，故在下文中把它们连接后形成的结构称作“栈树”。如a_1、a_2、a_3和a_4组成一棵栈树。事实上c_3也可能存在兄弟c_X，这时c_3-c_7和c_X将组成森林，不过这时也称它们组成的栈结构为栈树。

3.2.2 模式树中的PC关系

如果查询模式树中的所有非扩展结点之间的关系均为AD关系，getNext(Q)[3]和入栈算法moveElementToStack(Q, e)[2]一起，可以保证：

1. 栈结构中保存的元素集合和参与解的元素集合相等；
2. 除根栈元素以外的其他元素直接地或者通过同类型的其他栈元素间接地连接到其“最近模式祖先”。

如果允许模式树中PC关系的存在，上面的两个规律均会在某些情况下被打破。

首先，如果某个新取出元素对应的模式树结点与其父结点是PC关系，则确定了该元素在栈树中的位置后，它仍然未必需要入栈。可以在此时先检查它与其“最近栈内祖先”所在的层数差，如果该差值大于1则丢弃该元素。这样可以将栈结构所占空间进一步减少。不过，还是有部分元素与其子元素的层次关系不符合PC关系，但仍然会进入栈结构中，无法在事先判定需要丢弃。如图3-1(d)中的栈元素b_2，它不具有d类型PC关系的子元素，但无法避免b_2的入栈。不过结果枚举算法可以发现栈元素b_2并没有d类型的子栈元素，从而不会生成包含b_2的解。

此外，并不能保证最近栈内祖先就是最近模式祖先。如图3-1(b)中，由上段中分析，b_2不参与解，因而就不可能是任何元素的最近模式祖先。然而b_2是存在于栈中、盖住c_1且离c_1最近的元素，即c_1的最近栈内祖先。事实上c_1真正的最近模式祖先是b_1，而本算法仅仅保证它被连接到其“最近栈内祖先”b_2。可以看出，虽然b_1与b_2不在同一个栈中，但b_1盖住b_2。

不过这两个规律的打破并不影响解的正确输出。如果采用3.3节中的结果枚举算法，解的正确输出仅仅依赖于比上述两个条件更弱的条件。而后者可以由改进的入栈算法加以保证。

3.2.3 算法运行的先决条件和“最远祖先”

经由getNext(Q)取出的元素是具有“最小后代扩展”的，但并不能保证其参与解(即便模式树中的关系全部为AD关系)。因为元素e_Q具有“最小后代扩展”仅仅保证它具有一棵子树，该子树与模式树中以Q为根的子树匹配。要使得e_Q参与构成解，还要保证有一个参与解的父元素盖住它。只有在这两个条件都满足时，才能保证整棵模式树与包括e_Q在内的某个文档元素子集及其元素之间的关系匹配。在HolisticTwigStack算法中，针对后一个条件需要判断：取出的非根元素是否被父类型栈中已经存在的某个元素盖住。只有存在盖住它的父类型栈元素时，才能使那个新取出的元素进入栈结构。

然而，这个判断并不适合直接进行。在栈结构中，很可能某类型的栈树有很多棵，而每棵栈树中又有许多栈。如果直接遍历每个栈的栈底元素，判断在其中是否存在可以盖住当前新取出元素的元素，会有比较大的开销。如算法对图3-1的输入运行时，当经由getNext(Q)取出了元素c_7后，若要遍历已经存在与栈结构中的b类型栈底元素b_1、b_3、b_4，看其中是否有盖住c_7的元素，则需要进行很多次比较。不过，任何文档元素序

列中元素的取出都是在序列头部进行的，而序列中的元素是按照先序顺序排列的。故元素的取出和入栈的顺序也遵循其在文档中出现的先序顺序。如对与模式树结点a关联的序列中的元素，入栈的顺序是a_1、a_2、a_3、a_4。即便不是所有元素都可以入栈，这种先序顺序也不会受到影响。这样，在某一时刻，从某个序列中已经取出并成功进入栈结构的所有元素将盖住一系列不相交的区域。如在本例中从a对应的序列中的元素全部取出时已经取出并入栈的a_1、a_2、a_3、a_4盖住了两个不相交的区域，分别是a_1和a_4盖住的区域。被这些已经入栈的元素盖住的子类型元素必须在这些区域之一中。而经过进一步分析可以发现，它只能处于最后一个区域中。这是由于，当经由getNext(Q)取出目前的最后一个区域的根元素e（本例中的a_4）并且e成功进入栈结构时，e必定具有“最小后代扩展”，因而该元素是盖住其各个孩子序列首元素的。设这时某孩子序列首元素的左位置为l_0，则L(e)<l_0。对于从此时开始从该子序列取出的元素来说，其左位置l>l_0。因而有l>L(e)。就是说从最后一个区域的根元素取出开始，再取出的子序列元素f都将具有比该根元素左位置更大的左位置。因此，如果f存在于上述各不相交区域中的某一个，则只能是最后一个。反之，如果f确实存在于最后一个区域中，显然其父类型栈元素中存在盖住它的元素，至少最后一个区域的根元素就是其中一个。基于上述讨论，可以将对f的父类型栈中是否存在元素盖住f的判断，转化为对f的父类型栈元素盖住的最后一个区域的根元素是否盖住f的判断。设有模式树结点Q，记：到当前为止E的序列中取出的栈元素盖住的所有不相交区域中的最后一个的根元素为FA(Q)，或称为到目前为止Q类型栈元素的“最远祖先”。只要能以较低代价维护FA(Q)，就可以通过判断Q的子类型P的序列中新取出的元素e_P是否被FA(Q)盖住来判断其是否可以进入栈结构。这很容易实现，只要对各个模式树结点Q，记录当前的FA(Q)，并且当Q关联的序列中新的栈元素离开当前记录中的FA(Q)盖住的区域时，用该元素来更新记录即可。

3.2.4 新元素在栈结构中的定位和“最近祖先”

在确定元素可以进入栈结构之后，就需要确定其在栈结构中的位置。按照本节开始处的讨论，栈元素需要直接或间接地连接到其“最近栈内祖先”。当直接连接时，由“最近栈内祖先”引出连接指向入栈元素即可。而当间接连接时，又需要找到与之具有相同“最近栈内祖先”的元素组成的栈树，之后连接到那棵栈树中的合适位置——或者是某个栈的栈顶；或者是某个栈元素的兄弟。因而在这个阶段面临的任务是：

1. 寻找新取出元素的“最近栈内祖先”；
2. 寻找新取出元素在其栈树内的兄弟元素。

这两个任务有一定的相似性：都是寻找盖住某个元素e的元素，并且如果有多个这样的元素(它们必同在e的双亲链上)，所需要得到的都是其中距离e最近的那一个元素。只是盖住e的元素的寻找范围不同：在第一个任务中是在e的同类型栈元素中寻找，而在第二个任务中是在e的父类型栈元素中寻找。

为了完成这两个任务，需要检查元素间的盖住关系，并在具有盖住关系的情况下计算元素间的距离(也就是元素所处层数的差)。这在区域编码的基础上都很容易实现。只是，如果分别采用遍历所有新取出元素的父类型栈元素、遍历新取出元素应该进入的栈树中的所有栈元素的方法来完成这两个任务，开销仍将很大。

注意到，如果构造一个到目前为止从某序列中取出并已经入栈的栈元素集合$\mathbb{S}_{path}$，当该序列中一个新元素e入栈时就将e加入$\mathbb{S}_{path}$，并将原来$\mathbb{S}_{path}$中盖不住e的所有元素移除。这样$\mathbb{S}_{path}$中的元素必定始终在一条路径上，不会有分枝。在图3-1(c)中表示了模式树结点c所关联的文档元素序列中取出并入栈的元素形成这种路径的情况。随着c_1至c_7依次进入栈结构，$\mathbb{S}_{path}$的内容依次为：c_1、c_2、c_3、c_3，c_4、c_3，c_4，c_5、c_3，c_4，c_6、c_3，c_7。注意c_7入栈时，c_6、c_4已经从整个路径中去掉了，并代之以c_7，因为c_6和c_4盖不住c_7。由于任何序列中的元素按照先序顺序进入栈结构，这样的路径是不断向右移动的，这种移动由路径中靠下面的元素不断向上“传递”。在这样形成的路径称之为“活动路径”。模式树结点Q类型的“活动路径”记做AP(Q)。AP(Q)=($\mathbb{V}$, $\mathbb{E}$)，其中$\mathbb{V}$是路径上的栈元素的集合，$\mathbb{V}$=$\mathbb{S}_{path}$，而$\mathbb{E}$是相邻元素之间边的集合，可由$\mathbb{V}$确定。在“活动路径”中，存在“最近祖先”的概念。设路径上某个元素为e，则e上方距离e最近的元素称作e的最近祖先，记为CA(e)。我们可以在每一个栈元素中记录其“最近祖先”，这样就维护了所有当前的和历史的“活动路径”信息。

通过这种在算法中维护的由“最近祖先”关系形成的路径，可以更高效地完成前述的两个任务。

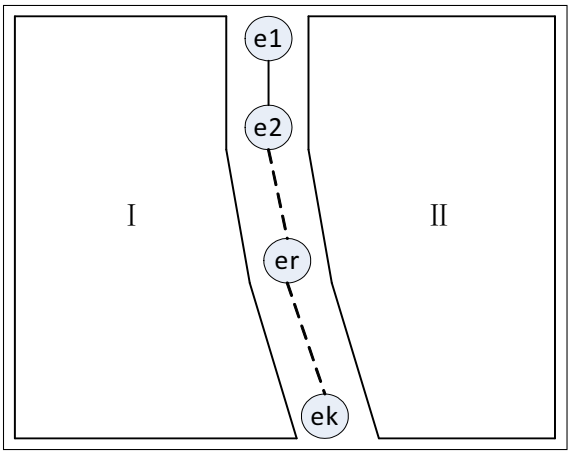

图 3-2　寻找“最近栈内祖先”方法的原理

在第一个任务，也就是寻找“最近栈内祖先”的任务中，设某新取出的元素f_F的父类型“活动路径”为AP(E)(如图3-2所示)，其中E为F在模式树中的父结点。AP(E)将整个仅由E类型元素组成的森林划分出的其余两部分编号为Ⅰ、Ⅱ。区域Ⅰ、Ⅱ中均包括一部分以AP(E)上元素为祖先的元素，以及一部分与以e_1为根的子树不想交的子树中的元素。

由活动路径的“移动”规律可以看出，AP(E)上的最后一个元素e_k应为E类型文档元素序列中到目前为止的最后一个入栈元素(事实上，需要记录该元素，并由元素中关

于“最近祖先”的记录找到“活动路径”上的其他元素)。而AP(E)上的第一个元素实际上是FA(E)。目前为止入栈的所有E类型元素或者在AP(E)上，或者在区域Ⅰ中，但是不可能在区域Ⅱ中。因为假设在区域Ⅱ中，则$\mathbb{S}_{path}$中必然会包括区域Ⅱ中的元素。这样，我们将在区域Ⅰ和AP(E)中寻找盖住f的E类型栈元素。而更进一步的分析显示在区域Ⅰ中是不可能存在这样的元素的。在该区域中任取一个栈元素e，如果它不在e1为根的树中，则其右位置小于所有AP(E)上元素的左位置；若它在e1为根的树中，还是可以在AP(E)上找到栈元素e_i，使得R(e)<L(e_i)，事实上，只要令e的兄弟元素为e_i即可。也就是说：

$$\forall e \in Region(\text{Ⅰ}),\ \exists e' \in \mathbb{S}_{path},\ \text{使得} R(e) < L(e') \tag{3.1}$$

元素e′已经经由getNext(Q)取出，它具有“最小后代扩展”，这意味着对其后取出的F类型元素f，L(e’)<L(f)。再结合式3.1可得：L(f)>R(e)。这样就不可能有e盖住f了。又因为这个e是在区域Ⅰ中任取的，这就说明在整个区域Ⅰ中都不可能有盖住f的元素了。盖住f的E类型元素仅仅存在于AP(E)上(或$\mathbb{S}_{path}$中)。又e_1=FA(E)是盖住f的，否则f不能进入栈结构。这样，由e_k开始沿着AP(E)向上，将会先连续遇到盖不住f的元素，而后连续遇到盖住f的元素，直到e_1。而在分界处的那个盖住f的元素就是f的最近栈内祖先了。

基于上述讨论，为找到某个元素的“最近栈内祖先”，只需从下端开始遍历其父类型的“活动路径”，到分界处停止。之所以从下端开始向上遍历，是由于在栈元素中维护的是其“最近祖先”。这种寻找“最近栈内祖先”的方法在效率上可以带来一定的改善。

在栈结构构造算法需要完成的第二个任务，也就是寻找新取出元素在其栈树内的兄弟元素的任务中，采用的方法与上面类似。只是为新取出的F类型的元素f′寻找其在栈树中的位置时需要考察的是F而不是其父类型的“活动路径”。首先，同样需要找到F类型最后一个栈的栈顶元素，之后向上遍历以该元素结束的“活动路径”AP(F)。直至找到第一个盖住f′的元素，那就是f′在栈树中的父元素。找到父元素后，如果该父元素就是其所在栈的栈顶元素，则f′为其第一个孩子，将f′直接压入该栈即可。否则f′不能够进入与其父元素相同的栈，而需要将其放入新建的栈，并连接到其兄弟元素。那个兄弟元素实际上就是AP(F)中位于那个父元素下方的元素。但是存在一种例外情况：如果f′的“最近栈内祖先”尚无任何F类型的孩子，此时f′并不是连接到栈树上的，而是直接连接到其“最近栈内祖先”上。因此这个情况是要优先判断的。

在确定新取出的E类型元素e在栈结构中位置的同时，还要计算出CA(e)，以维护“活动路径”信息。另外，要同时检测FA(E)是否发生变化，如果是，还要更新FA(E)的记录。在按照上面方法计算出e在E类型元素的森林中的父元素e′之后，有CA(e)=e′。如果“活动路径”中所有元素都无法盖住e，则令CA(e)=NULL，表示e已经是该森林中某一棵树的根元素了。

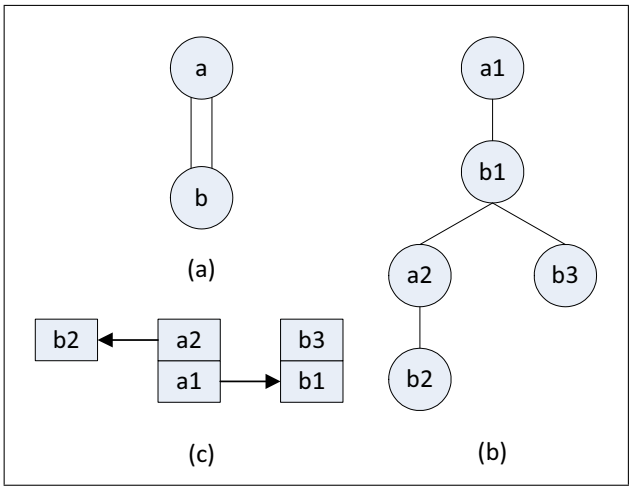

图 3-3　栈结构构造算法优化问题

3.2.5 关于优化

值得注意的是，[1]和[2]中对入栈算法做了一个优化：如果新取出的元素被其所属类型的最后一个栈的栈顶元素盖住，且两个元素具有相同的最近模式祖先，那么直接将该元素压入该栈即可。由于只是入栈，不进行新栈创建的开销很小，将这个情况放在最开始考虑可以提高效率。不过，如果此处的“最后一个栈”是指所有该类型栈中的最后一个，则该条件并没有涵盖所有的只需入栈而不需要新建栈的情况。图3-3中的情况下，当b_3到来时，最后一个b类型的栈是$_2$所在栈，其栈顶元素b2并不盖住b3，然而仍然不需创建栈。由于b_1盖住b_3，但b_1在栈中上方尚无元素，因而可以直接将b3压入该栈。但是，[2]中给出的栈结构构造算法中，只要该条件不成立，就要建立新栈。这样将会创建多余的栈。

如果将条件改为——新取出的元素是否被其最近栈内祖先的最后一个子栈的栈顶元素盖住，并在该条件成立时直接将新元素压入该栈，否则再构造新栈，则创建多余的栈的问题将可以得到解决，其开销亦可以省去。不过，若要判断该条件，就不仅仅需要像[1]和[2]中所说的那样需要记录每种类型的最后一个栈，而是需要在栈元素e中记录以e为“最近栈内祖先”的各种类型的最后一个栈。这样，对空间的需求将会增加。

3.2.6 改进的栈结构构造算法

综合上面的讨论，经过改进的栈结构构造算法如算法1所示。算法对从模式树结点Q关联的文档元素序列中取出的元素e进行操作。其中，NewStack(e)表示建立新栈，并将e入栈后返回该栈。Push(S, e)表示将元素e压入栈S。Covers(e, f)判断元素e是否盖住元素f。LevelDifferenceSatisfied(e, f)判断元素e和f层数之差是否与他们对应的模式树结点E和F之间的AD或PC关系匹配。UpdateFAWhenNeeded(Q, e)在元素e已经离开当前FA(Q)的范围时自动用元素e更新Q类型的最远祖先。UpdateFA(Q, e)将元素e设为Q类型栈元素的最远祖先。SetCA(f, e)将元素f的“最近祖先”设

算法 1 moveElementToStack(Q, e)

Require: e被FA(P)盖住，P=Parent(Q)；栈结构中已经存在“总根栈”。
Ensure: 若e参与采用宽松模式树后的小枝匹配结果，则e进入栈结构；
进入栈结构的e直接地或通过Q类型栈树间接地与其“最近栈内祖先”相连。

```
1: csa ← CSA(Q, e)
2: if not LevelDifferenceSatisfied(e, csa) then
3:   return
4: end if
5: if LastChildStack(csa, Q) ≠ NULL then
6:   if Covers(Top(LastChildStack(csa, Q)), e) then
7:     SetCA(e, Top(LastChildStack(csa, Q)))
8:     Push(e, LastChildStack(csa, Q))
9:     return
10:  end if
11:  S ← NewStack(e)
12:  UpdateLastStack(Q, S)
13:  UpdateLastChildStack(e, Q, S)
14:  p ← Top(S)
15:  q ← CA(Top(S))
16:  while q ≠ NULL and not Covers(q, e) do
17:    p ← q
18:    q ← CA(q)
19:  end while
20:  SetCA(e, q)
21:  UpdateFAWhenNeeded(Q, e)
22:  SetSibling(p, e)
23: else
24:  S ← NewStack(e)
25:  UpdateLastStack(Q, S)
26:  UpdateLastChildStack(e, Q, S)
27:  if LastStack(Q) ≠ NULL then
28:    p ← Top(LastStack(Q))
29:    while p ≠ NULL and not Covers(p, e) do
30:      p ← CA(p)
31:    end while
32:    SetCA(e, p)
33:    UpdateFAWhenNeeded(Q, e)
34:  else
35:    SetCA(e, NULL)
36:    UpdateFA(Q, e)
37:  end if
38: end if
```

算法 2 CSA(Q, e)

Require: Q结点在模式树中，e为Q类型元素。
Ensure: 返回元素e的最近栈内祖先。
1: $csa \leftarrow Top(LastStack(Parent(Q)))$
2: **while** $csa \neq NULL$ **and not** $Covers(csa,e)$ **do**
3: $csa \leftarrow CA(csa)$
4: **end while**
5: **return** csa

定为元素e。LastChildStack(e, Q)获得当前元素e的最后一个Q类型孩子栈。Update-LastChildStack(e, Q, S)用栈S更新该记录。LastStack(Q)获得当前所有Q类型栈中的最后一个栈。UpdateLastStack(Q, S)用栈S更新该记录。SetChild(e, f)将f设为e的孩子元素(建立由e到f的第一类连接)。SetSibling(e, f)将f设为e的兄弟(建立由e到f的第二类连接)。CSA(Q, e)取得e的“最近栈内祖先”。

此外，本算法要求：如果需要入栈的元素为根类型元素，则栈结构中已经单独建立了一个“总根栈”，其中只有一个元素，就是代表整个文档的元素(盖住任何其他元素)。这样，栈结构构造算法内部就不用像[2]中那样单独考虑根类型元素的入栈了，因为这时栈结构中已经存在其父类型栈，就是“总根栈”。

3.2.7 改进的栈结构构造算法分析

本小节对前面论述的栈结构构造算法所构造的栈结构中的元素及元素之间关系的性质做一总结和证明，证明的性质是结果生成算法由栈结构计算出全部正确小枝匹配结果的基础和前提。在本部分论述中，记Seq(Q)为与模式树结点Q关联的文档元素序列，First(seq)为序列seq的首元素。ST(Q)为Q类型的栈树集合。CSA(e)为元素e的“最近栈内祖先”。$FRST_Q$为将所有Seq(Q)中的栈元素单独放在一起，保留它们在原文档树中的关系而形成的森林。

由于栈结构构造算法的正确运行依赖于过程getNext(Q)为其指明下一个元素的来源，故首先说明getNext(Q)所具有的几个性质。

引理 1 若getNext(Q)返回P，First(Seq(P))具有“最小后代扩展”[3]。

引理 2 若e∈Seq(E)，f∈Seq(F)，E=Parent(F)，e盖住f，则f不可能在e尚存在于其文档元素序列中时被取出。其中e、f为文档元素，E、F为模式树结点。

引理 3 在getNext(Q)内部被跳过的元素不参与任何解。

引理 4 所有序列中的元素最终或在getNext(Q)外部被取出，或在getNext(Q)内部被跳过。

由于getNext(Q)已在[3]中详细说明，此处不再对这4个引理进行证明。下面给出描述栈结构中元素及元素相互关系的定理，并加以证明。

定理 1 设有集合$\mathbb{M}$，$\forall Q \in$模式树PT，$\mathbb{M}$中存在唯一的$e \in Seq(Q)$。则$\mathbb{M}$是一个小枝匹配解中的元素集合，当且仅当：

1. $\mathbb{M}$中所有元素存在于栈结构中；
2. 对$\mathbb{M}$中元素p和q，$p \in Seq(P)$，$q \in Seq(Q)$，P=Parent(Q)，则或p=CSA(q)，或p盖住CSA(q)，且当P与Q为PC关系时，必定是相等关系成立。

证明：

先证必要性。采用数学归纳法进行，对$\mathbb{M}$中元素对应的模式树结点所在层数lev进行归纳。

1. 对$\mathbb{M}$中lev=1的元素，即$e \in Seq(Q)$，Q为模式树根结点。由引理3和4知，因为e是解元素，故e必定在某时刻在getNext(Q)外部被取出。而e作为解元素，与整个文档树的根必定满足模式树根结点与总根结点间的AD、PC关系。因而e会进入栈结构。
2. 假设$\mathbb{M}$中lev=k的元素具有定理中叙述的两个性质。则对lev=k+1的元素f，设它被$\mathbb{M}$中lev=k的元素e盖住。由引理3和4知，e和f均是在getNext(Q)外部被取出的。再由引理2知，f取出时e已经取出。在由归纳假设，e作为lev=k的元素能够进入栈结构。并且由算法结构可知，e取出后，在下个元素取出前便进入了栈结构。故f取出时，e已经在栈结构中了。这样，f的父类型栈中必定至少存在元素e盖住f。满足了栈结构构造算法要求的前提条件。当e与f对应的模式结点E与F是PC关系时，算法中将对f和CSA(f)的层数关系进行检查。由e与f在同一个解中的关系知，此时有Level(f)=Level(e)+1，因此这时找到的CSA(f)必为e(因为e盖住f，且在栈中不可能再有同样盖住f同时距离f更近的元素)。因而层数关系的检查可以通过。又，作为解元素，f的谓词肯定满足。因此算法中关于谓词是否满足的检查也可以通过。这样f必定会进入栈结构。这样就证明了第一个性质。而关于第二个性质，前面已经证明了当模式结点E与F是PC关系时，必定有e=CSA(f)。若不是这种情况，由于e是栈结构中盖住f的元素之一，而CSA(f)是栈结构中盖住f且距离f最近的元素，故e与CSA(f)必定同在f的双亲链上，或者e=CSA(f)，或者e盖住CSA(f)。

再证充分性。在$\mathbb{M}$中存在模式树中各类型元素的前提下，只需证明它们的位置关系符合模式树中的规定，以及如果有谓词约束，被约束的元素满足约束它的谓词即可。如果元素在栈结构中，说明在栈结构构造算法中，关于其谓词的检查已经通过。所以下面只需证明元素间的位置关系与模式树中相应结点之间的AD、PC关系匹配即可。

若对$\mathbb{M}$中元素$e \in Seq(E)$，$f \in Seq(F)$，且E=Parent(F)，有e=CSA(f)或e盖住CSA(f)，则说明e可以盖住f。如果E与F之间为AD关系，则这种“盖住”关系已经与AD关系匹配了。而如果E与F之间为PC关系，由f在栈结构中可知，算法检查CSA(f)与f的

层次关系的结果为：Level(f)=Level(CSA(f))+1。而由已知条件e=CSA(f)。故Level(f)=Level(e)+1。又e盖住f，因此e为f的父结点。这种关系同样与E和F之间的PC关系匹配。又因为这里e和f是在$\mathbb{M}$中任取的，因此已经可以证明$\mathbb{M}$中元素关系与它们在模式树中相应结点的AD、PC关系匹配了。□

定理 2 在某个栈树ST∈ST(E)中，若在栈元素e所在栈中，e上方的元素为e_{UP}，则在$FRST_E$中，e为e_{UP}的孩子；若由栈元素e存在指向栈元素$e_{SIBLING}$的第二类连接，则在$FRST_E$中，$e_{SIBLING}$为e的兄弟元素。

由入栈算法的描述，本定理的结论较为明显，此处不再证明。

3.3 结果枚举算法的改进

结果枚举算法的改进主要是针对原算法不能按3.1节中规定的顺序枚举结果的问题进行的。同时也注意到仅仅依靠存在于栈中的祖先-后代关系进行结果的枚举可能会产生丢解的问题，而依靠前面讨论的“最近祖先”关系可以解决该问题。本节先对算法思想进行分析，再给出算法本身，最后对其正确性进行论证。

在本节的讨论中，用记号Sol(e)表示在栈结构中e与其各类型孩子解的所有组合的按序排列，这里所说的“孩子”包括连接到e上的子栈树中的元素，也包括连接到e的同类型后代元素上的子栈树中的元素。如图3-1的例子中，$Sol(b_3)=(b_3c_2d_1)$。另外，用记号EnumAlg(e)表示以元素e为参数调用结果枚举算法，以返回E类型栈树中以e为根的子栈树中的所有解。如$EnumAlg(a_2)$将返回a_2、a_3和a_4参与的所有解。EnumAlg(e)的返回值用AccuSol(e)表示。

3.3.1 结果的枚举顺序和组合顺序

记元素e在栈中的“直接后代”(direct descendent element，“与当前元素在同一个栈中，在其上方，且位置与其相差1的元素”)[2]为e_{UP}，e引出的第二类连接指向的兄弟元素为$e_{SIBLING}$，则有：

$$AccuSol(e) = Sol(e) \bullet AccuSol(e_{UP}) \bullet AccuSol(e_{SIBLING}),\ \bullet \text{为连接运算符} \quad (3.2)$$

等式3.2表示：EnumAlg(e)的返回值可以由Sol(e)和$EnumAlg(e_{UP})$、$EnumAlg(e_{SIBLING})$的返回值按序组合而成。即需要在EnumAlg(e)中调用$EnumAlg(e_{UP})$和$EnumAlg(e_{SIBLING})$，并计算出Sol(e)。若e为叶元素，则Sol(e)=e，否则Sol(e)是元素e与所有其孩子解的组合，因而要得到Sol(e)须先枚举e的孩子解。即需要对所有的f调用EnumAlg(f)，其中f为e经由第一类连接指向的某类型的孩子元素。在EnumAlg(e)中调用$EnumAlg(e_{UP})$、$EnumAlg(e_{SIBLING})$和EnumAlg(f)，并将e与各个EnumAlg(f)的返回结果进行组合，这就是结果枚举算法的基本框架。

若要使EnumAlg(f)返回AccuSol(f)，需要使$EnumAlg(e_{UP})$先于EnumAlg(f)被调用。如图3-1(d)所示，设$e=a_1$，则$e_{UP}=a_2$，$f=b_1$。有(b_1, c_1, d_3)∈AccuSol(f)。但若

不调用EnumAlg(a_2)即EnumAlg(e_{UP})，c_1是无法被访问到的。因此需要先调用EnumAlg(e_{UP})。而EnumAlg($e_{SIBLING}$)在何时调用均可，只要按照式3.2右端将其返回值放在总返回值中的合适位置即可，规定EnumAlg($e_{SIBLING}$)在EnumAlg(e_{UP})之后调用。

3.3.2 最近祖先关系的利用

图3-1的查询例子中，($a_1b_1c_1d_3$)是最终解中的一个元素。然而从(d)的栈结构可以看出，b_1与c_1并不连通。事实上，新元素取出后，在栈结构构造算法中将会面临两种约束，其一是要连接到其最近栈内祖先，其二是要进入栈顶元素盖住它的栈。这两个约束有时是矛盾的，这时将优先满足前者。在本例中，按照第二个约束条件，b_2本可进入b_1所在的栈，那样将导致c_1与b_1连通。然而，那样b_2也将通过b_1连接到a_1，但b_2的最近栈内祖先不是a_1，而是a_2。这样，b_2还是会被连接到a_2，从而导致了其与b_1的隔离，进而导致c_1与b_1的隔离。

按照[1]和[2]中所述的结果枚举算法，由于a_2是a_1的直接后代，故a_2的孩子解将同时可以成为a_1的孩子解。但是b_2不是b_1的直接后代，如果不使用额外的信息，无法使两者发生关联，并使b_2的孩子解“传到”b_1，从而形成b_1与c_1的组合。

注意到若要关联b_2和b_1，3.2.4小节提出的最近祖先关系正是一种“额外的信息”，因为b_1=CA(b_2)。如果在元素e的孩子解计算完毕后，将其“传到”CA(e)处，那么就不必再依靠栈中的直接后代关系(因为若在某个栈中e_2是e_1的直接后代，同时也必有e_1=CA(e_2))进行孩子解得传递，而且原来靠直接后代关系无法形成的组合也可以形成。这样，同一个栈中的直接后代关系就仅仅充当枚举结果时递归调用运行的线索，而不再具有其他用处。

3.3.3 segmentsList

采用向最近祖先传送孩子解的策略时，孩子解可能有多个来源。由某个新的来源传来的孩子解是不能直接连接到已经传来的孩子解的末尾的。假设图3-1的例子中a_2在b_2和a_4的中间有另外一个孩子元素b_X，且b_X具有最小后代扩展。由3.3.1小节中的规定，在EnumAlg(a_2)的调用中先调用EnumAlg(a_3)，后调用EnumAlg(a_4)。在EnumAlg(a_3)中，AccuSol(b_3)将传至a_2处，而在EnumAlg(a_4)中，AccuSol(b_4)亦将传至a_2处。然而，AccuSol(b_4)是不能立即附加到AccuSol(b_3)的末尾的，因为在解中AccuSol(b_3)和AccuSol(b_4)中间还隔着 AccuSol(b_X)。可以想见，若a_4右侧存在兄弟b_Y、a_5、b_Z、a_6……在这种情况下在EnumAlg(a_i)($i\geq3$)中传至a_2的孩子解就会被AccuSol(b_X)、AccuSol(b_Y)、AccuSol(b_Z)等诸多部分解隔开。这样就产生了采用数据结构segmentsList来存放各个传上来的孩子解的需要。segmentsList内的每个元素是一组上传的孩子解。在本例中a_2的segmentsList为(AccuSol(b_3), AccuSol(b_4))。

值得注意的是，各个孩子解在最终解中出现的顺序未必是它们被上传的顺序。如图3-1的查询中，由3.3.1小节的讨论，在EnumAlg(a_2)中，EnumAlg(a_3)应该先于EnumAlg(b_2)进行调用。在EnumAlg(a_3)中会调用EnumAlg(b_3)，在其中会将d_1传至CA(b_3)

=b_2。因为a_3有指向a_4的第二类连接，在EnumAlg(a_3)中会调用EnumAlg(a_4)。在其中会上传d_2至CA(b_4)=b_1。而这时，EnumAlg(a_3)尚未结束，因而EnumAlg(b_2)尚未开始。而已经传至b_2的d_1要进一步传至b_1则要在EnumAlg(b_2)中进行。也就是说，d_2比d_1更早地传至b_1。然而在最终的解中，组合b_1d_1却应出现在b_1d_2前面。这说明应该对各个传上来的孩子解进行排序，或者在将某组孩子解加入到segmentsList中时找到按照顺序它应该插入的位置(可以采用二分查找法以提高效率)。

在将某组孩子解AccuSol(e)按序插入segmentsList时，插入位置的实际依据是L(e)。令L(e)为一组孩子解AccuSol(e)的左位置，记为L(AccuSol(e))。事实上，L(e)也就是该组孩子解中第一个匹配结果的第一个元素的左位置。在内层递归调用保证每一组孩子解AccuSol(e)内部各个匹配结果的有序性的前提下，只需依据L(AccuSol(e))对各组孩子解进行排序，就可以保证segmentsList中所有孩子解出现的顺序与它们在最后的解中出现的顺序相同。

在EnumAlg(b_2)中需要将上传至a_2的b类型孩子解整合到a_2本已具有的孩子解中。这时a_2的segmentsList为 (AccuSol(b_3), AccuSol(b_4))。对EnumAlg(b_2)有L(b_2)<L(b_3)，因此AccuSol(b_3)尚在AccuSol(b_2)之后。这样EnumAlg(b_2)中就不必将segmentsList中的任何孩子解放到计算出的Sol(b_2)之前，可直接调用EnumAlg(b_X)。在EnumAlg(b_X)中，由于L(b_3)<L(b_X)，需要将segmentsList中的首元素AccuSol(b_3)放到Sol(b_X)之前。之后将AccuSol(b_3)从segmentsList中移除。在EnumAlg(b_2)返回后，将发现segmentsList中仍有剩余元素AccuSol(b_4)。须将剩余元素均附加在EnumAlg(b_2)返回值的末尾，构成完整的b类型孩子解。

3.3.4 改进的结果枚举算法

本小节给出改进后的结果枚举算法，其中GetStackEle(stack, pos)表示获得栈stack中位置为pos的元素，约定栈底的位置为0。Size(stack)表示栈stack的大小。Leaf(stack)判断栈stack是否为叶类型栈。Stack(e)获得元素e所在的栈。QueryNode(e)获得e对应的模式树结点。Children(e)获得栈元素e引出的所有第一类连接指向的孩子元素(每个子类型一个孩子元素)。Sibling(e)和DirectDesc(e)分别获得栈元素e的兄弟元素和直接后代元素。AddChildListToSegList(e, childList)表示将孩子解childList传到元素e处。GetSegList(e, Q)表示获得上传到e处的Q类型的segmentsList。Append(matchList$_1$, matchList$_2$)表示对两个解l_1和l_2进行连接，将结果放入返回。CurMatchList(segmentsList)用来获得segmentsList中当前的第一组孩子解。RemoveHeadMatchList(segmentsList)移除segmentsList中当前的第一组孩子解。Empty(segmentsList)判断当前segmentsList是否为空。L(matchList)用来获得部分解matchList的左位置。AddChildList(childLists, childList)将某一组孩子解childList加入到各个类型的孩子解集合childLists中。Combine(e, childLists)用来形成元素e与其所有类型孩子解的组合(childLists为不同类型的孩子解的集合)。PC(e)判断元素e对应的模式树结点与其父结点之间的关系是否为PC。

改进后的结果枚举算法如算法3所示。

算法 3 computeTwigSolution(stack, pos, segmentsList)

Require: 栈 stack 在位置 pos 处存在元素；
segmentsList 为上传到栈元素 GetStackEle(stack,pos)的孩子解组成的表。
Ensure: 返回 AccuSol(GetStackEle(stack,pos))；
在枚举过程中形成的所有的某个元素 e 的孩子解上传到 CA(e)处。

1: $matchList \leftarrow ()$
2: $curEle \leftarrow GetStackEle(stack, pos)$
3: **while** !$Empty(segmentsList)$ **and** $L(CurMatchList(segmentsList))<L(curEle)$ **do**
4: $matchList \leftarrow Append(matchList, CurMatchList(segmentsList))$
5: RemoveHeadMatchList(segmentsList)
6: **end while**
7: **if** $pos + 1 < Size(stack)$ **then**
8: $dMatchList \leftarrow computeTwigSolution(stack, pos + 1, segmentsList)$
9: **end if**
10: **if** !$Leaf(stack)$ **then**
11: $childLists \leftarrow \{\}$
12: **for all** e_{CHILD} *in* $Children(curEle)$ **do**
13: $childList \leftarrow ()$
14: **if** $e_{CHILD} \neq NULL$ **then**
15: $childList \leftarrow computeTwigSolution(Stack(e_{CHILD}), 0, segmentsList)$
16: **end if**
17: $remSegs \leftarrow GetSegList(QueryNode(e_{CHILD}))$
18: **while** !$Empty(remSegs)$ **do**
19: $Append(childList, CurMatchList(remSegs))$
20: $RemoveHeadMatchList(remSegs)$
21: **end while**
22: $AddChildList(childLists, childList)$
23: **if** !$PC(curEle)$ **and** $CA(curEle) \neq NULL$ **then**
24: $AddChildListToSegList(CA(curEle), childList)$
25: **end if**
26: **end for**
27: $Append(matchList, Combine(curEle, childLists))$
28: **else**
29: $Append(matchList, (curEle))$
30: **end if**
31: $Pop(stack)$
32: **if** $Sibling(curEle) \neq NULL$ **then**
33: $sMatchList \leftarrow computeTwigSolution(Stack(Sibling(curEle)), 0, segmentsList)$
34: **end if**
35: $Append(matchList, dMatchList)$
36: $Append(matchList, sMatchList)$
37: **return** $matchList$

3.3.5 改进的结果枚举算法分析

本节论证结果枚举算法的正确性和其所输出的结果的有序性。首先介绍栈结构构造算法所构造的栈结构的一种树形表示，这与后面定理的证明证明有关。

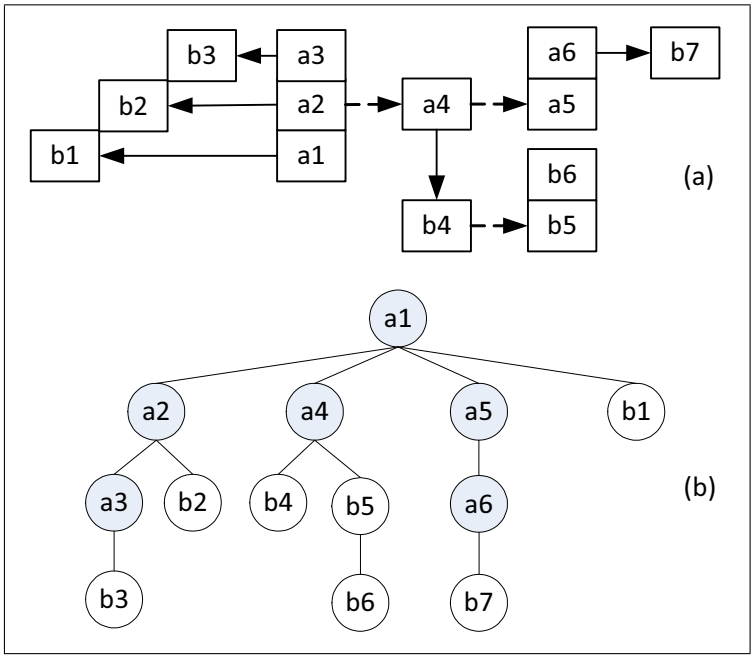

图 3-4　栈结构整体的树形等价表示

图3-4(a)所示的栈结构可以表示成(b)中的等价形式。其中，将某元素e在其所在栈中的直接后代表示成其第1个孩子，该直接后代的兄弟（由第二类连接所指向）表示成e的后续孩子，而e的子类型的后代（由第一类连接所指向）表示成e的其他孩子，放在所有与e同类型的孩子之后。这样形成的栈结构的树形等价表示称为栈结构的等价树。

由算法中的递归调用可以看出，结果枚举算法实际上完成了对等价树的遍历。由于在EnumAlg(e)中，先对其同类型直接后代，也就是在等价树中e的第1个孩子e_{UP}调用EnumAlg(e_{UP})，而在EnumAlg(eUP)中会对e的其余的与之同类型的孩子，也就是e_{UP}的兄弟$e_{UP_SIBLING}$调用EnumAlg($e_{UP\_SIBLING}$)。而在EnumAlg(e_{UP})之后才会开始对e的不同类型孩子e_Q调用EnumAlg(e_Q)。这时，与e的同类型孩子已经在EnumAlg(e_{UP})中被访问完。由此可知，对任何一个元素e，其各个孩子是按照从左到右的顺序访问的。进而，在整个遍历过程中，访问算法是按照先序顺序进入等价树各个元素的。这对应到栈结构中，表明结果枚举算法按照某种次序遍历了栈结构中的所有元素，对每个栈元素e都调用了EnumAlg(e)一次且仅一次。

在上述讨论以及如下引理的基础上，将可以证明结果枚举算法的正确性定理。

引理 5 若栈元素e′为e的后代，f为e的子栈元素。则EnumAlg(e′)结束后，EnumAlg(f)才开始运行。

证明：采用数学归纳法，对栈元素e所对应的模式树结点在模式树中所处的层数lev进行归纳。

1. 当lev=0时，e为根类型元素，此时e与e′必然同在根类型的唯一栈树中。EnumAlg(e)的第8行调用了EnumAlg(e″)，e″为e的直接后代。e′=e″或者e′为e″的后代。由结果枚举算法在同一棵栈树中的访问顺序知EnumAlg(e′)就是EnumAlg(e″)，或者前者在后者内部调用。因此，无论如何，第8行的调用返回后，EnumAlg(e′)已经返回。而EnumAlg(f)是在第15行进行的，其所在循环位于第8行之后。因此lev=0时，引理的结论成立。

2. 假设当lev=k(k≥0)时引理的结论成立，往证lev=k+1时结论成立。lev=k+1时，或者CSA(e)=CSA(e′)，或者CSA(e′)为CSA(e)的后代。当CSA(e)=CSA(e′)时，e与e′连接到相同的父类型元素，因而处于相同的栈树中。这样，采用基本情况的论证中所采用的逻辑即可证明EnumAlg(e′)结束后，EnumAlg(f)才开始运行。若CSA(e′)为CSA(e)的后代，由于两者是e和e′的父类型元素，故其对应的模式树结点层数为k。由归纳假设知，EnumAlg(CSA(e′))结束后，EnumAlg(e)才开始运行(因为e为CSA(e)子类型的孩子)。又EnumAlg(CSA(e′))结束前，EnumAlg(e′)就已经结束，EnumAlg(e)刚开始运行时，EnumAlg(f)尚未开始运行。因此EnumAlg(e′)结束后，EnumAlg(f)才开始运行。即lev=k+1时，引理的结论同样成立。

由数学归纳法，本引理的结论成立。□

定理 3 元素集合$\mathbb{M}$是小枝匹配结果当且仅当结果枚举算法将$\mathbb{M}$作为一个结果输出。

证明：

先证必要性，采用数学归纳法。注意到解$\mathbb{M}$与模式树匹配，因而它本身存在一种树形结构。对该树中元素e为根的子树$\text{SubTree}_{\mathbb{M}}$(e)的树深dep进行归纳。

1. 当dep=1时，也就是e为叶元素时，由定理1知，由于e是解元素，因而e在栈结构中。再由上面的讨论知，EnumAlg(e)必会获得运行机会。由算法知EnumAlg(e)返回的AccuSol(e)必定包含e本身(算法中处理叶元素的部分比较简单，不再具体分析)。也就是说，$\text{SubTree}_{\mathbb{M}}$(e)中的元素集合会被EnumAlg(e)当做一个结果返回。

2. 假设当dep≤k(k≥1)时，EnumAlg(e)会将$\text{SubTree}_{\mathbb{M}}$(e)中的元素集合作为一个结果返回，往证当dep=k+1时也是这样。dep=k+1时，考察e的孩子e_{CHILD}，由定理1，e_{CHILD}在栈结构中所直接或间接连接到的与e同类型的元素或者为e，或者为e盖住的某个栈元素e′。又$\text{SubTree}_{\mathbb{M}}(e_{CHILD})$的树深至多为k。由归纳假设，EnumAlg($e_{CHILD}$)返回的结果AccuSol($e_{CHILD}$)中包含$\text{SubTree}_{\mathbb{M}}(e_{CHILD})$中的元素集合。由此可知，EnumAlg(e)或者EnumAlg(e’)中，第15行的childList被赋值后将包含$\text{SubTree}_{\mathbb{M}}(e_{CHILD})$中的元素集合。如果是在EnumAlg(e)中计算出的这

些孩子解集合，易知在第27行处，e将获得与所有这些$SubTree_{\mathbb{M}}(e_{CHILD})$中的元素组合的机会，而组合的结果就是$SubTree_{\mathbb{M}}(e)$。如果是在EnumAlg(e′)中计算出的这些孩子解集合，设$SubTree_{\mathbb{M}}(e_{CHILD})$中的元素的集合为$\mathbb{N}$。考察e到e′的路径，设其中的E类型元素依次为为e、e_1、e_2、……、e_{k-1}、e_k、e′。则e′为e_k的后代，由引理5知当EnumAlg(e′)结束后才开始枚举e_k的孩子解。当EnumAlg(e′)结束后，e′的孩子解中所包含的$\mathbb{N}$已经作为孩子解上传至e_k处了。因此，$\mathbb{N}$也将包含在此后计算出的e_k的孩子解中。而EnumAlg(e_k)结束后，才开始枚举e_{k-1}的孩子解，同理，$\mathbb{N}$也将包含在e_{k-1}的孩子解中。以此类推。所有的$\mathbb{N}$将包含在e的孩子解中，并在第27行处与e组合。组合的结果就是$SubTree_{\mathbb{M}}(e)$。因此，当dep=k+1时，EnumAlg(e)会将$SubTree_{\mathbb{M}}(e)$中的元素集合作为一个结果返回。

由数学归纳法知，当dep等于$\mathbb{M}$所对应的整棵树的深度时，$SubTree_{\mathbb{M}}(rt)$会被EnumAlg(rt)返回，其中rt为小枝匹配$\mathbb{M}$所对应的树的树根。而$SubTree_{\mathbb{M}}(rt)$中的元素集合正是$\mathbb{M}$。rt是根类型栈树中的元素，因此EnumAlg(rt)所返回的结果必定包含在最终结果中。这也就证明了：元素集合$\mathbb{M}$是小枝匹配结果，则结果枚举算法将$\mathbb{M}$作为一个结果输出。

再证充分性。即证明：若元素集合$\mathbb{M}$不是小枝匹配结果，则结果枚举算法不会将$\mathbb{M}$作为一个结果输出。

元素集合$\mathbb{M}$不是小枝匹配结果有两种情况。一种可能是：$\mathbb{M}$不满足对模式树中的每个类型，存在唯一该类型元素的条件。但是，因为结果的形成过程中，每个类型的元素以父元素的身份参与与各类型孩子的组合都有唯一的一次，在该次组合中该元素进入结果。故结果枚举算法输出的所有结果中都满足该条件，即相应于每个模式树结点，结果中都存在唯一的该类型的元素。因此在这种情况下结论是正确的。

另一种可能是：对模式树中的某两个结点P和Q，P=Parent(Q)，$\mathbb{M}$中这两个类型的元素e_P和e_Q不满足P与Q之间的AD或PC关系。假设在这种情况下欲证之结论不成立，即$\mathbb{M}$会被输出。在此假设下，必有CSA(e_Q)=e_P，或者e_P盖住CSA(e_Q)，否则两个分别属于模式树中父子类型的元素不可能获得组合机会。无论是那种情况，必有e_P盖住e_Q。因此e_P与e_Q间必定满足AD关系。到此，只可能e_P与e_Q间不满足PC关系，即Level(e_P)+1≠Level(e_Q)。不过在结果枚举算法返回的结果中包括e_P与e_Q组合的情况下，若P与Q是PC关系，一定有CSA(e_Q)=e_P。否则，由于算法第27行阻止后代元素的孩子解向最近祖先传递，e_Q所在的孩子解将无法传到e_P处，不会有机会与e_P组合。但在CSA(e_Q)=e_P的情况下，如果e_P与e_Q的层数差1的关系不满足，在栈结构构造算法中第2行处将阻止e_Q进入栈结构。因此e_P与e_Q间不满足PC关系的可能性也排除了。这样，关于$\mathbb{M}$会被输出的假设与“对模式树中的某两个结点P和Q，P=Parent(Q)，$\mathbb{M}$中这两个类型的元素e_P和e_Q不满足P与Q之间的AD或PC关系”的前提矛盾。也就是说，在这种情况下，$\mathbb{M}$同样是不会被结果枚举算法输出的。

综合这两种情况下的讨论，充分性成立。□

定理 4 结果枚举算法所生成的小枝匹配结果的顺序与3.1节所述之顺序相同。

本定理的完整证明比较琐细，此处仅仅阐述其证明思想。首先，容易证明枚举叶元素所产生的matchList中的元素是满足3.1节所述之顺序的。由孩子解上传到“最近祖先”时“按序插入”segmentsList的机制可知，当前元素自身的孩子、加上其后代的孩子所形成的孩子解(segmentsList中元素)所形成的childList是满足该顺序的。接着，如果元素e的各个childList内的各个部分解的排列是满足该顺序的，则容易写出算法，进行e与childList中部分解的各种组合，并且使组合结果的排列满足该顺序。此外，若式3.2中的Sol(e)、AccuSol(e_{UP})和AccuSol($e_{SIBLING}$)已经计算出，且在这三部分内部的各个元素之间满足该顺序，则由于这三部分中的每一个部分匹配结果的首元素分别为e、e_{UP}和$e_{SIBLING}$，且由栈结构特点知：e→e_{UP}→$e_{SIBLING}$的顺序是先序顺序，故这三部分中的元素按照式3.2中的顺序放在一起后形成的大的线性表中元素也满足3.1节中所述顺序。按照这个思想可以使用数学归纳法写出完整的证明。

4. 一种基于HolisticTwigStack的流数据小枝查询算法

本章基于HolisticTwigStack算法提出XML流数据上的小枝查询算法SHolistic-TwigStack，并对算法正确性进行论证。首先介绍XML文档数据流的概念以及流数据小枝查询算法的意义。之后提出SHolisticTwigStack流数据小枝查询算法。接着对算法正确性进行论证。论证时遵循的逻辑如下。HolisticTwigStack算法所构造的栈结构的性质（见3.2.7）保证了结果枚举算法可以从中枚举出全部正确的小枝匹配结果。SHolisticTwigStack算法中的栈结构构造算法以及结果枚举算法与HolisticTwigStack算法中相同。因此，只要在流处理算法中也保证可以构造出具有该性质的栈结构即可。在3.2.7节中已经找到了getNext(Q)的一些性质，它们作为充分条件保证了由文档元素序列取出的元素传递给栈结构构造算法后就可以构造出具有该性质的栈结构。故只需证明在前者中地位与getNext(Q)等同的getNextFromStream(Q)与doSkip(Q)保持了getNext(Q)所具有的性质，即可证明SHolisticTwigStack算法的正确性。

4.1 XML流数据小枝查询算法概述

XML流数据查询算法是指以XML字节流或记号流（token stream）[10]作为输入进行查询的算法。字节流粒度太小、层次太低，往往需要先在系统内部转化为记号流或类似的表示，再传给系统所采用的流数据查询算法。记号流是由许多代表XML文档和文档元素的开始和结束的记号组成的。在代表元素开始的记号中有关于该元素的标签名和属性的信息，而在代表结束的记号中有结束元素的标签名。由于XML文档的严格嵌套结构，这保证了结束元素的唯一确定。

XML流数据小枝查询算法往往需要在内存中缓存部分数据，将其作为查询对象。缓存中不可能包括尚未到达的数据，也不应包括距流的当前内容太远的历史数据。

在许多应用中，数据不断产生，或者虽然已经完全产生，但总量巨大。在这两种情况下，须采用流数据查询算法进行数据查询[9]。

首先，不断产生中的、本身具有“流”属性的数据是很常见的。如金融领域、工业领域的各种监测数据——股价信息、各种软硬件系统运行状况数据等。这些数据源源不断地产生，何时停止往往无法预料。只要尚未停止，就必有一部分数据是不可预知的。另外如果当前数据中已经包含结果，往往需要即刻获得该结果。而不是当前数据已经变为历史数据很长时间后再产生这些结果。也就是说需要把处理所依赖的数据范围限制在不很久远的历史直到当前时刻的一个较窄的窗口内。这种情况下采用流处理算法是很自然的。

另外，如果数据总量很大，出于硬件条件的限制，可能无法将所有数据全部读入内存，再行处理。然而，由于效率上的需要，又不能直接处理磁盘数据。最好可以对磁盘数据进行一遍顺序扫描，并同时对缓存在内存中的当前扫描位置的数据片段进行查询，在扫描结束后就产生所有查询结果。这比多遍扫描，或者根据处理需要随时从磁盘各处

进行不规则的数据读取往往效率更高。也就是说，这种情况下需要从数据源刻意产生流数据。流处理算法解决了空间紧张的问题，并满足了效率上的需要。

由这两类应用可见，衡量XML流数据查询算法性能的两个最重要的指标可以归结为：响应时间和缓存量。即是否可以迅速给出包含在已经到达或已经读到的数据中的结果（上面第一种应用中可能更加需要响应迅速），和是否历史数据的存储空间可以更快、更多地释放。

目前，XML流数据查询技术，或更一般的XML流管理技术，在国内外均被广泛研究[10]。一些经典XML数据查询算法，如YFilter，本身就是为流数据查询任务提出的。在YFilter中，新的XML文档元素的到来引起自动机的状态转移，自动机到达接受态意味着部分查询结果的产生。这相当于历史数据已经转化并储存在了自动机的状态中，因而其丢弃并不影响查询的进行。而包括HolisticTwigStack在内的另一些算法，则需要进行一些改变才能胜任流处理任务。下文提出一种经过改进，在HolicticTwigStack算法基础上实现的流处理算法SHolisticTwigStack，并对利用此算法实现的查询系统进行介绍。为此，先说明作为本算法的XML文档数据流的类型及限制。之后提出SHolisticTwigStack算法，并对其正确性及效率做一分析。最后介绍基于SHolisticTwigStack算法的流处理系统的设计和实现。

4.2 XML数据流需要满足的限定条件

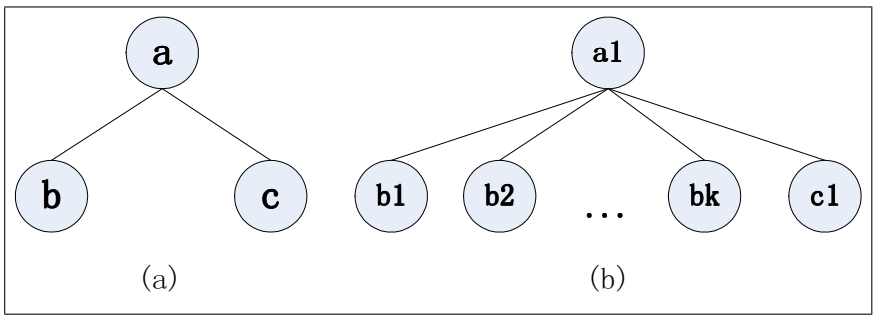

图 4-1　使用流处理算法进行查询也无法减小缓存的流数据

在内容上，XML流数据查询算法对流数据均会有一定的假设或限制条件。如果条件不满足，算法性能就会大大下降，甚至无法再完成流数据查询。例如在下面的情形下，任何算法均需缓存整个XML文档，之后才能给出查询结果。如图4-1所示，(a)为查询模式树，(b)为文档树。假设此文档非常大，因而需要使用流处理算法，以避免将整个文档加载到内存中。然而，由于内容上的特殊性，这是无法避免的。此情形对应的记号流为：

STARTDOCUMENT STARTELEMENT(a_1) STARTELEMENT(b_1) ENDELEMENT(b_1) STARTELEMENT(b_2) ENDELEMENT(b_2) ... STARTELEMENT(b_k) ENDELEMENT(b_k) STARTELEMENT(c_1) ENDELEMENT(c_1) ENDELEMENT(a_1) ENDDOCUMENT

其中STARTDOCUMENT为表示文档开始的记号。STARTELEMENT为表示文档元素开始的记号，括号内为开始元素的标签名。ENDELEMENT为表示文档元素结束的记号，括号内为结束元素的标签名。ENDDOCUMENT为表示文档结束的记号。容易发现，STARTELEMENT(c_1)之前到来的记号所表示的文档元素是否属于解，在STARTELEMENT(c_1)记号到来之前是无法判断的。也就是说必须对那些记号或者它们所表示的文档元素进行缓存。结果几乎整个文档都需要缓存。这就使流处理算法蜕化成了非流处理算法。而在文档非常大的情况下，这种缓存可能无法实现，这便导致了算法的失效。

SHolisticTwigStack算法同样不能避免这个问题。事实上，这个算法能够达到预期的流处理效果的前提是：小枝查询的任何一个解中的元素在XML文档（或数据流）中分布的跨度相对于整个文档（或数据流）的跨度不能过大。此处解的跨度定义为该解中先序位置最大和最小的元素的先序位置之差，而整个文档的跨度定义为文档中数据元素的个数（不包括文档根元素DOCUMENTELEMENT）。

4.3 SHolisticTwigStack算法的基本思想

SHolisticTwigStack算法是本文提出的在XML记号流上执行小枝查询的算法。该算法保持了HolisticTwigStack算法对小枝模式的完整性处理，即在栈中保存的不是模式树中单个路径（由根到叶）的匹配结果，而已经是整个模式树在文档中的匹配结果。避免了不参与解的无用结点对内存的消耗和归并中间结果（即路径匹配结果）造成的开销[1]。此外，就流处理算法而言，SHolisticTwigStack保证了较短的响应时间和较低的内存消耗。

在SHolisticTwigStack算法中，根据流处理的需要，为查询模式树中的每一个结点分配一个队列，而不再只是分配一个线性表。与HolisticTwigStack中一样，模式树中可能有多个结点是具有相同类型的，这样该类型将被分配多个相互独立的队列。而具有该类型的元素在到来后将进入该类型的所有这些队列中。不再需要的文档元素随时出队。仍然按照前文所述的方法判断文档元素的类型，从而决定其需要进入的队列。另外对文档元素进行区域编码的方法也不变。

在HolisticTwigStack算法中，用getNext(Q)过程[3]获得下一个需要从文档数据序列中取出的元素对应的模式树结点，以保证从该模式树结点的队列中取出的队头元素具有“最小后代扩展”(见5.2.3节)。从而避免了对很大一部分不参与解的元素进行入栈操作的开销。这和采用的栈结构一起，构成了HolisticTwigStack算法的两大特征。

SHolisticTwigStack基于该算法提出，也具有这两个特征。在这两者中，产生栈结构的算法并不需要直接对文档数据操作，而是依靠getNext(Q)为其提供合适的文档元素。后者才需要直接对文档数据（存在于多个队列中）进行操作。而流处理和非流处理的关键区别就在于文档数据的呈现方式不一样。在非流处理中，任何时刻都可以在内存中访问到全部的文档数据；而流处理中可以访问的数据限制在一个“窗口”中。由此，为使SHolisticTwigStack具有上述两个特征，主要的改动应存在于getNext(Q)中。

流处理版本的getNext(Q)所起的作用由两个过程分担，分别为getNextFromStream

(Q)和doSkip(Q)。它们在记号流中的元素开始标记（STARTELEMENT）到来时运行。

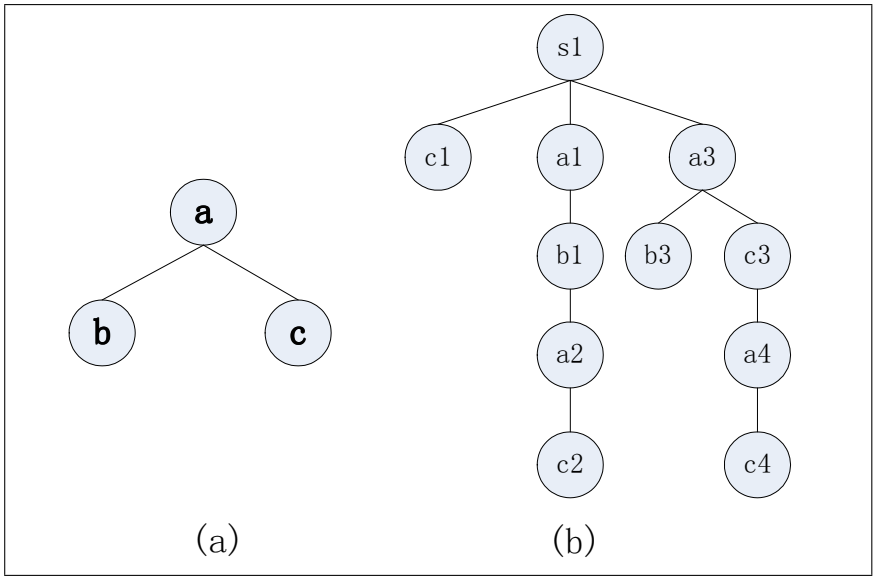

图 4-2　流处理示例

像HolisticTwigStack算法中的getNext(Q)一样，getNextFromStream(Q)在SHolisticTwigStack中保证传给入栈算法的元素都是具有“最小后代扩展”的。不过getNextFromStream(Q)不会像getNext(Q)那样，每次调用都会返回一个确定的模式树结点，而是有时只返回NULL。这表示由目前已经到达的流元素，还无法判断缓存中哪个模式树结点对应的队列的队首元素具有“最小后代扩展”，可以取出。如图4-2中的情况所示，图中(a)为模式树，(b)为文档树。在STARTELEMENT(b_1)到来时，在缓存中可以发现a_1具有b类型孩子b_1，但此时尚不具有任何c类型孩子。这时调用的getNextFromStream(Q)就会返回NULL。这是因为那个等待的c类型孩子“此时没有”，但并不能说明“根本没有”，可能只是“尚未到来”。在本例中就是后一种情况，a_1的那个在等待中但是尚未到来的孩子就是c_2。但在HolisticTwigStack中，getNext(Q)在这种情况下已经可以确定a_1是没有c类型孩子的。这是因为在HolisticTwigStack中，所有元素都读到内存中后，才开始调用getNext(Q)。这样getNext(Q)发现某个需要的元素“还没有出现”就说明该元素“根本没有”。

此外，为了让更多的无用元素尽快跳过，也出于效率上的考虑，SHolisticTwigStack中将跳过无用元素的算法独立出来，并且变为一种非递归的写法。也就是doSkip(Q)。

本算法中，getNextFromStream(Q)、doSkip(Q)和入栈算法是在记号流中的元素开始标记到来时运行的。那时该元素的右位置尚不确定，但关于元素的“盖住”关系等的判断需要比较左右位置的大小关系。故在本算法中需要使得不确定的右位置也可以与某个已到来元素的左位置比较大小。这是另外一处需要改动的地方。

4.4 记号

在算法描述中，用记号RPOS_PENDING表示元素尚不确定的右位置。用记号INFINITE表示某类型的下一个元素尚未到来，因而该类型的队列目前为空。另外，用NULL表示getNextFromStream(Q)返回的不确定类型。

由于模式树中可能有多个结点共享相同类型，但各自独立地拥有一个队列。故用Que(Q)表示模式树结点Q所对应的队列。QueOffer(que,e)表示元素e入队que。QuePeek(que)表示获得que队头的第一个元素，First(que)也表示该元素。QueRemove(que)表示使que队头的第一个元素出队。QueRemove(que,e)表示移除队que中的元素e（为了算法实现方便，允许对队列中间的元素进行移除操作）。QueEmpty(que)判断队que是否为空。Size(que)表示取得队que中的元素个数。

此外PT表示模式树。Parent(PT, Q)结点Q在模式树中的父结点或双亲结点。Children(PT, Q)表示结点Q在模式树中的子结点集合。此外，用Nodes(PT)表示模式树中所有结点的集合。Leaf(PT, Q)判断Q在模式树中是否为叶结点。Types(PT)表示模式树中的类型集合。NodesOfType(PT, type)表示模式树中类型为type的结点集。Type(e)表示元素e的类型。

4.5 SHolisticTwigStack

SHolisticTwigStack算法见算法4和5。其中需要的子过程见算法6和7。

这些算法的运行时机有两个：一是元素开始标记到来时，二是文档结束标记到来时。大部分的处理发生在元素开始标记到来时。不过在最后一个开始标记的处理结束后，队列中仍有可能还有剩余元素。故在文档结束标记到来时，还要追加第二阶段的处理，采用原HolisticTwigStack算法找出剩余的参与解的元素并入栈。算法的意义详见下面的几个小节。

4.5.1 SHolisticTwigStack

算法4在有元素开始标记（STARTELEMENT）从流中到来时运行。由4.2节中对记号流的描述可知，元素开始标记带有元素标签名及属性信息。这样就可以确定该元素所属类型，如第1行所示。第2行指出，只有该类型在模式树中，也就是说到来的元素在查询范围内，才考虑后续的缓存和其他处理；否则新到来的元素可直接抛弃。如果元素类型在模式树中出现，则需要缓存该元素。在模式树中该类型的结点可能不止一个，每个结点关联一个队列，需要把e放入所有这些队列中。元素的缓存在第5行处进行。

由于getNextFromStream(Q)运行时只关注各个队列的队头，因此如果e放入那些队列前，队不空，则队头元素不变，那么getNextFromStream(Q)在此情况下的运行将与上一次运行结果相同。getNextFromStream(Q)的上一次运行是在算法4的上一次运行中进行的。从14行知getNextFromStream(Q)的调用是循环进行的。而从16、17行知循环之所以结束原因是getNextFromStream(Q)返回了NULL（即不确定的模式树结点）。由此可

知，getNextFromStream(Q)的上一次运行结果是NULL，这才导致上一次算法4的运行结束了。因此如果e的到来并没有改变任何队列的队头元素，本次getNextFromStream(Q)仍将返回NULL。没有元素可以被取出，各队列内容不会改变。因而在这种情况下可以不必再调用getNextFromStream(Q)，而是直接返回即可。这就是3-12行的作用之一。

与此同时，在某个队列Que(Q)由空变为不空时，新的队头元素（也是队中唯一的元素）亦可作为丢弃祖先类型队（指所有的Que(P)，其中P为模式树PT中Q结点的祖先结点）中元素的依据。这在原理上与原getNext(Q)相似，但不全同。算法4的第7行就是为此而设。在后文对此进行详述。

算法14-30行所做的事情是这样的：只要不是所有队列都变空，就循环运行算法getNextFromStream(Q)。getNextFromStream(Q)每次运行都会返回一个模式树结点Q，或者NULL。如果返回NULL，就结束循环，进而结束算法4的本次运行。这是因为，在这种情况下无法确定哪个元素具有“最小后代扩展”，并因而具有成解的可能。这样也就无法从队中取出任何元素，对其进行入栈。进而各队列内容不会变化。这样在下一个元素开始标记到来前再调用getNextFromStream(Q)，返回的也将是一样的结果——NULL。而如果getNextFromStream(Q)返回了确定的模式树结点，则说明该结点所关联的队列的队头元素具有“最小后代扩展”，可以取出。事实上，与HolisticTwigStack算法中一样，对于这样取出的元素e，只要父类型栈中有元素盖住它，就可以将e传给入栈算法。见27-29行。而24-26行的意义是，如果新取出的元素已经离开了当前根栈栈底元素盖住的区域，说明该区域下的解已经完全包含在当前的栈结构中了。因此可以输出这些解。

如果根据getNextFromStream(Q)指出的模式树结点Q，从队列Que(Q)的队头取出了元素，则队头元素有所改变。若新的队头元素为e1，这时同样需要考察各祖先队列中是否有元素不具有“最小后代扩展”，而且已经无法盖住e1（因为e1的左位置太大）。这样的祖先类型元素可以丢弃。故在第22行处与在第7行处一样运行doSkip(Q)，以图从缓存中丢弃一批已确定不参与解的元素。

算法5在文档结束标记（ENDDOCUMENT）从流中到来时运行。因为单靠算法4在很多情形下并不能处理掉流尾部的所有元素，所以在整个文档读完时还要运行原Holistic-TwigStack算法来达到这个目的。算法5就是为找出存在于流尾部的解元素而设的。

4.5.2 getNextFromStream(Q)

getNextFromStream(Q)在算法6中给出。如上文所述，该算法的目的是给出队头元素具有“最小后代扩展”的一个队列对应的模式树结点。根据其返回值的指示，就可以取出具有“最小后代扩展”的下一个元素，它很可能就是构成解的元素。这样此后就可以将取出的元素传给入栈算法了。

具有“最小后代扩展”的充要条件已在4.3节中给出。假设一个元素e具有“最小后代扩展”，e对应的模式树结点为E，E在模式树中的孩子结点为F_i（i=1,2,...,n, n为孩子结点个数），这些模式树结点的队列的下一个元素为f_i。这意味着所有这些f_i都是e的孩子元素。进而e盖住所有的f_i。另外这些f_i与它们所属队列的子队列队头元素

算法 4 SHolisticTwigStack

Require: 记号流中元素e的开始标记到来。
Ensure: 流中一批元素被正确入栈或丢弃；
若栈结构中已经产生一组完整的解，全部已经产生的解被输出。

```
1: type ← Type(e)
2: if type ∈ Types(PT) then
3:   headChanged ← false
4:   for all Q ∈ NodesOfType(PT, type) do
5:     QueOffer(Que(Q), e)
6:     if Size(Que(Q)) == 1 then
7:       doSkip(Q)
8:       headChanged ← true
9:     end if
10:  end for
11:  if headChanged == false then
12:    return
13:  end if
14:  while not ∀Q ∈ Nodes(PT), QueEmpty(Que(Q)) do
15:    Q ← getNextFromStream(Root) {Root为模式树的根}
16:    if Q == NULL then
17:      return
18:    end if
19:    E ← QuePeek(Que(Q))
20:    QueRemove(Que(Q))
21:    if not Empty(Que(Q)) then
22:      doSkip(Q)
23:    end if
24:    if E为根类型元素 and E离开了根栈栈底元素盖住的区域 then
25:      showTwigSolution(RootStack, 0) {RootStack为根栈，0为栈底元素下标}
26:    end if
27:    if E为根类型元素 or E的父类型栈中有元素盖住E then
28:      moveElementToStack(Q, E)
29:    end if
30:  end while
31: end if
```

算法 5 SHolisticTwigStack

Require: 记号流中文档结束标记到来。
Ensure: 流中最后一批元素被正确入栈或丢弃；
若栈结构中产生一组完整的解，最后一组解被输出。
{运行非流处理版本的算法以处理剩余元素}

```
1: HoisticTwigStack()
```

又递归地满足这个条件。由1.2小节的讨论知有：

1. 对元素e而言：

$$\forall f_i, L(e) < L(f_i) \wedge R(e) > L(f_i) \tag{4.1}$$

2. 对每个这样的f_i而言，它们与其所属队列的子队列的队头元素递归地满足条件1。

在原HolisticTwigStack算法中，$L(e) < L(f_i)$和$R(e) > L(f_i)$这两部分均是在getNext(Q)中加以判断并获得保证的。然而在本算法中，getNextFromStream(Q)本身只保证取出的元素及其后代元素满足前者，而后者则由doSkip(Q)帮助确认。也就是说，当getNextFromStream(Q)运行之前运行了doSkip(Q)时，经由getNextFromStream(Q)指出的队头元素e已经满足全部的两个条件。并且由于getNextFromStream(Q)的递归结构，e孩子又递归地满足这两个条件。这样就可以保证e是具有“最小后代扩展”的。

保证第一个条件满足是getNextFromStream(Q)自身的作用，在本小节说明。而保证第二个条件满足作为doSkip(Q)的目的，将在4.5.3小节说明。

在算法6中，第1-7行处理递归的基本情形——Q在模式树中为叶结点的情形。由于getNextFromStream(Q)会对Q的孩子调用该算法自身，故最终在各条路径上都会递归下降到模式树的叶结点。对叶元素来说，由于其对应的模式树结点没有孩子结点，故其自身就是具有“最小后代扩展”的，当然也满足式4.1中的第一个条件。因此只要Q的队列不空，就将其直接返回。否则，不能确定Q类型队列中的下一个元素是否具有“最小后代扩展”，因为该元素尚未到来，也可能永远也不会到来了。故当叶结点Q的队列为空时返回NULL。

第8-26行处理非基本情形，如果Q的所有孩子都已经确定满足式4.1中的第一个条件（getNextFromStream(child)返回child本身），而Que(Q)的队头元素又分别与各个孩子队列的队头元素满足该条件，就说明Q及其后代递归地满足了该条件。这时可以返回Q本身（第25行）。

除此以外还有其他情况。由于是流处理，某个孩子队列可能在某时刻为空，这时其中的下一个元素的存在性和位置都无法确定。可能会导致无法确定某个模式树结点的队列的队头元素及其孩子递归满足式4.1中的第一个条件。这会导致某个getNextFromStream(Q1)向上一层返回NULL。如果某个孩子child使得getNextFromStream(child)返回了NULL，就置标记notSureExist为true，以表示孩子状态的不确定性。

这时，如果另外一个孩子e2递归满足了该条件，又从左侧跃出了Que(Q)队头元素盖住区域的界限，仍然可以将该孩子元素对应的模式树结点child返回。见第18-20行。这是因为Q已经不可能满足该条件，而其孩子child却满足。需要注意的是，对两个父子关系的队列que1=Que(Q)、que2=Que(child)，若que1、que2均为空队，则从下个左位置都是INFINITE的角度上看两个左位置是相等的。但并不能判定这违背了式4.1中的第一个条件，因为那里所取的都是确定元素的确定的左位置。而que1和que2的下个元素还不能确定。因而不能以此为依据将child返回。这就是第18行中ln≠INFINITE条件的作用。但如果que1的下个左位置为INFINITE，而que2的下

个左位置是确定的，则可以判定两个左位置违背了该条件。因为队列的下个左位置为INFINITE意味着其下一个元素尚未从流中到来，又由于在流中开始标记是按照元素的先序顺序到来的，尚未到来的元素的左位置必然大于已经到来的元素的左位置。子元素从左侧跃出父元素盖住范围的界限导致其模式树结点被返回，这样返回的模式树结点会在上一层的第10行处被截获。之后如第11行所示，该模式树结点将层层向上返回，并最终成为getNextFromStream(Root)的返回值。

同样在notSureExist被置为true时，如果没有孩子元素从左侧跃出Que(Q)队头元素范围的界限，则getNextFromStream(Q)需要返回NULL。这是第22、23行所表示的动作。这时Que(Q)队头元素自身所处的不确定状态是某些孩子元素的不确定状态所导致的。这正是因为“最小后代扩展”要求式4.1的第一个条件被层层递归满足，而不是只在一层父子关系中满足。

算法 6 getNextFromStream(Q)

```
Require: Q为某个模式树结点。
Ensure: 若返回类型为Q，则Que(Q)的队头元素具有“最小后代扩展”；
        若返回NULL，则当前尚无法判断是否有队头元素具有“最小后代扩展”。
 1: if Leaf(Q) then
 2:   if not Empty(Que(Q)) then
 3:     return Q
 4:   else
 5:     return NULL
 6:   end if
 7: end if
 8: for all child ∈ Children(Q) do
 9:   n ← getNextFromStream(child)
10:   if n ≠ NULL and n ≠ child then
11:     return n
12:   end if
13:   if n == NULL then
14:     notSureExist ← true
15:   else
16:     ln ← L(QuePeek(Que(n)))
17:     lq ← L(QuePeek(Que(Q)))
18:     if ln ≤ lq and ln ≠ INFINITE then
19:       return n
20:     end if
21:   end if
22:   if notSureExist then
23:     return NULL
24:   else
25:     return Q
26:   end if
27: end for
```

4.5.3 doSkip(Q)

如第4.5.2小节所述，getNextFromStream(Q)自身可以保证取出的元素及其后代队列的队头元素递归地满足式4.1中的第一个条件。为了保证其具有“最小后代扩展”，尚需保证第二个条件的递归满足。这由doSkip(Q)实现。如算法7所示。

算法的基本思想是，若队列que1、que2的模式树结点具有祖先－后代关系，而que1中的某元素e与que2的队头元素f并不满足该条件。也就是说，e与f的关系是R(e)<L(f)。这时就将e从队que1中移除。注意，如果在进行此判断时ENDELEMENT(e)尚未到来，R(e)是不确定的，此时R(e)与L(f)的比较方法在下一小节中详述。

maxl用来保存Q的双亲链上的所有模式树结点所对应队列中最大的队头左位置。在第1行先将其置为Que(Q)队头元素的左位置。之后，在第2-8行沿着Q的双亲链向上迭代。移除Q的祖先结点所对应队列中右位置小于当前已经获得的最大左位置的元素。每移除一个元素，如果队不空，就要在remL记录中记录剩余的队头元素左位置。这样，最终remL中保存的应该是队中剩余元素的最小左位置；或者，在队已经空的情况下，保存的是队空之前队中最后一个已到来元素的左位置。如果remL已经大于当前的maxl，则用其值更新maxl。注意，如果移除元素之后队已经变空，不能用INFINITE来更新maxl。因为INFINITE并不是一个确定的左位置值，仅仅表示下一个元素尚未到来。从祖先队列中移除元素的操作在第4行。更新maxl的操作在第5-7行。

值得注意的是，只能以后代队列的队头元素左位置为依据移除祖先队列中的元素，后代队列中的其他元素不能作为依据。以图4-3为例说明这一点。图中(a)为模式树，(b)为文档树，(c)表示STARTELEMENT(b2)到来之后各个队列的情况。易知此时缓存中所有元素都参与成解，所有元素都都要取出，而不能移除。但如果以Que(b)的非队头元素b2为依据来移除Que(a)中的元素，则由于R(a1)<L(b2)，a1将被错误地移除。这将会导致丢解。而仅仅以Que(b)的队头元素b1作为依据则不会有任何问题。

另外，被移除的元素却并不仅限于队首元素。在任何队列中，由于元素从流中按先序顺序到来，故元素的左位置必定是递增的。然而右位置并不具有递增规律。若某个队列中两个相邻元素依次为e1和e2(e1为队头元素)，而e1是e2的祖先，则它们右位置的关系反而是R(e1)>R(e2)。因此有可能出现e1不被移除，而e2被移除的情况。这也就是第4行并不写作

```
while(R(QuePeek(Que(Q)))<lmax)
  QueRemove(Que(Q))
end while
```

的原因。

事实上，在HolisticTwigStack算法中调用的getNext(Q)采用的正是后一种做法——仅仅移除队头元素。这在正确性上是没有问题的。在上一段所描述的情形中，如果此时不移除e2，当以后e1作为队头元素被移除时，由R(e1)>R(e2)可知，e2也将作为那时的队头元素被移除。两种做法的区别在于移除时机。SHolisticTwigStack算法中采取的方法保证e2这样的无用元素可以在更少的后续标记到来之后就被移除。而这以移除算法运行时间的增加为代价。因此，如果流的速度较慢，即相邻两个开始标记到来

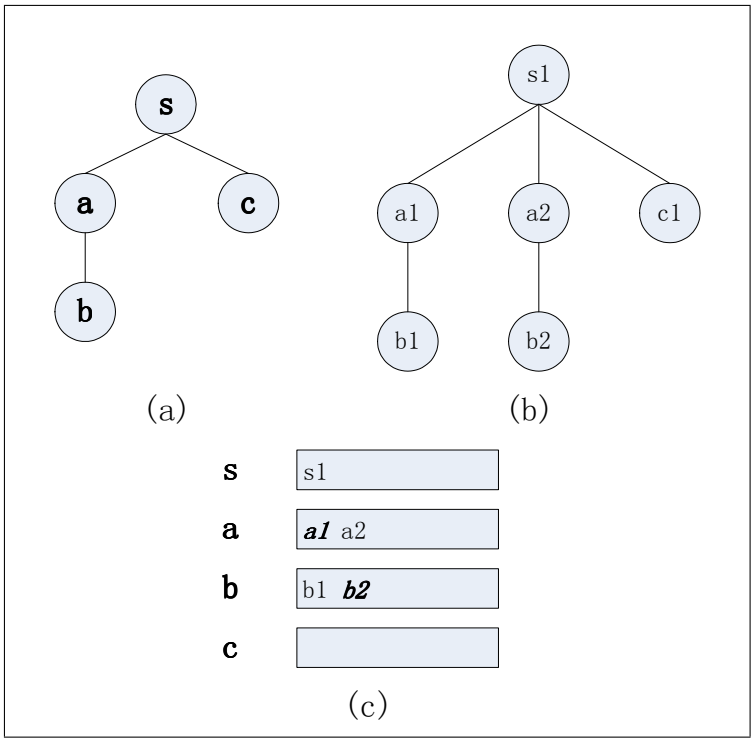

图 4-3　移除祖先元素只能以后代队头元素为依据

的时间间隔长于处理时间时，SHolisticTwigStack算法采取的不仅限于移除队头元素的策略将具有优越性。因为此时是流本身的速度、而非处理速度，在更多地决定响应时间。这样doSkip(Q)的运行时间稍长对给出解的响应时间并无太大影响。而它对减小缓存则有明确的作用。而当流的速度快于处理速度时，必然只能对不连续的数据片段进行查询。如果这是有意义的，则如果能使doSkip(Q)的运行时间缩短，将有助于缩短整个处理时间，进而扩大查询所能覆盖的数据范围。因此，对于快速的流，在内存并不紧张时，原HolisticTwigStack算法中的策略具有优越性。

若模式树中某结点Q的父结点P的队列Que(P)的首元素被移除，将使得Que(P)队头元素左位置变大，超过其父结点（Q的祖先结点）R的队列Que(R)的某些元素的右位置。进而，这将导致Q的祖先结点队列中元素被移除。此外，即便Que(P)中没有任何元素需要移除，以Que(Q)队头元素为依据，还是有可能跨过Que(P)，发现Que(R)中需要移除的元素。以图4-4为例说明这一点。图中(a)为模式树，(b)为文档树。STARTELEMENT(c2)到来时，Que(a)和Que(b)中分别有元素a1和b1。这两个元素尚未取出是因为它们尚缺少必要的孩子元素。而STARTELEMENT(c2)到来后，由于Que(c2)由空变为不空，将运行doSkip(c)。c的父结点为b，然而Que(b)中的唯一元素b1并不需要移除。倒是c的祖先a的队列Que(a)中元素a1可以在此时移除，因为L(c2)>R(a1)。因此，即使父类型队列中没有任何元素需要移除，使算法doSkip(Q)中的循环继续也仍然是有意义的。

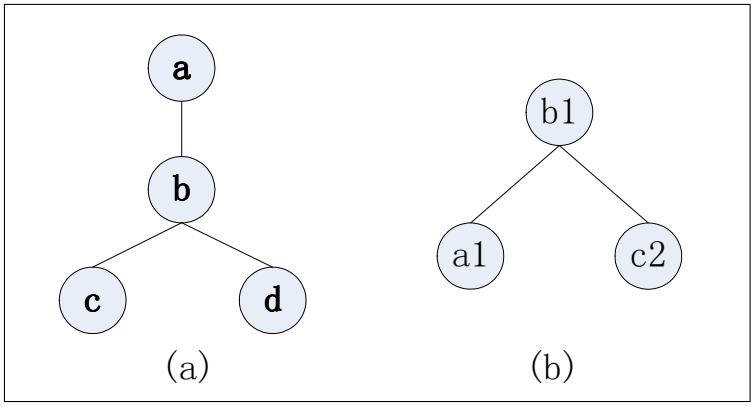

图 4-4　父队列不变，而祖先队列需要移除元素

算法 7　doSkip(Q)

Require: Q为某个模式树结点。
Ensure:　若Q的双亲链上有结点P，$e_p \in$ Que(P)元素为，
　　Q到P的所有结点对应的队列中最大的队头左位置为lpos，则lpos<R(e_p)。

```
1:  maxl ← L(QuePeek(Que(Q)))
2:  repeat
3:    Q ← Parent(Q)
      remL ← L(QuePeek(Que(Q)))
4:    for all e ∈ Que(Q) do
5:      if R(e) < maxl then
6:        QueRemove(Que(Q), e)
7:        if not QueEmpty(Que(Q)) then
8:          remL ← L(QuePeek(Que(Q)))
9:        end if
10:     end if
11:   end for
12:   maxl ← Max(remL, maxl)
13: until Q == Root
```

4.5.4 不确定的右位置与左位置的比较

上一小节中介绍的算法doSkip(Q)中涉及两个不同元素右位置与左位置的比较。doSkip(Q)的运行发生在某元素开始标记到来时。那时，所有已经从流中到来的元素的左位置都是确定的，而凡是结束标记没有到来的元素，其右位置的值均没有被确定下来。因此这个问题的实质是已经确定下来的左位置与尚未确定下来的右位置的比较。下面的定理解决了这个问题。

定理 5 具有未确定的右位置值的元素，其右位置的实际值大于当前所有已确定的左位置的值。

证明：设e为具有已确定的左位置值的元素，而f为右位置尚未确定的元素。
则STARTELEMENT(f)已经到来，而ENDELEMENT(f)尚未到来。因此，当前已经到来（至少其开始标记已经到来）的元素中有一部分在f的子树中，记这部分元素的集合为$\mathbb{U}$。由于元素是按先序顺序到来的，因而f子树中的元素是连续到来的，中间不会夹杂此范围之外的元素。而已经到来的另外一部分元素则是f子树外的元素，其左位置小于$\mathbb{U}$中元素的左位置，记这部分元素的集合为$\mathbb{V}$。对$e\in\mathbb{U}$，显然R(f)>L(e)。而对$e\in\mathbb{V}$，有L(e)<L(e1)，其中$e1\in\mathbb{U}$。又R(f)>L(e1)。故亦有R(f)>L(e)。□

4.6 算法运行示例

本节以图4-2中的查询为例，说明本算法的运行过程。

算法的运行是由记号流中标记的到来驱动的。首先到来的是整个文档的开始标记STARTDOCUMENT，这时在实际的系统中可以进行一些初始化工作，在算法本身的讨论中略去这些内容。之后到来的标记是STARTELEMENT(s1)，由于模式树中没有s结点，元素s1将被算法忽略，不会缓存在任何队列中。下一个标记是ENDELEMENT(s1)，在元素的结束标记到来时，算法不进行任何处理，因此下面就将所有元素结束标记的到来略去不提。下一个标记是STARTELEMENT(c1)，元素c1将入队Que(c)。这使该队列由空变为不空，队头元素发生变化。因此，将运行doSkip(c)。但是c的唯一祖先a的队列Que(a)为空，自然没有元素需要移除，结果本次doSkip(c)运行不会产生任何效果。此后要运行算法getNextFromStream(a)。此时Que(a)为空，因而其下一个元素的左位置肯定大于L(c1)。也就是说，c1从左侧跃出了Que(a)中下一个元素盖住的范围。因此本次getNextFromStream(a)将返回模式结点c。元素c1被取出，但不满足入栈条件，因而被舍弃。此时亦没有可输出的解，不会运行showTwigSolution(Root-Stack, 0)。紧接着会再一次运行getNextFromStream(a)，这时所有队列为空，无法确定哪个队列的下一个元素会具有“最小后代扩展”，返回NULL。相应于STARTELEMENT(c1)的所有处理结束。之后Que(c)队头元素元素又发生变化(由“有”变为“无”)。但由于Que(c)变空，无法预料下一个元素的位置，也就无法将那个元素作为移除祖先队列中元素的依据。因此不再运行doSkip(c)。下一个到来的标记是STARTELEMENT(a1)。元素a1入队Que(a)，导致其队头元素发生变化。运行doSkip(a)。本次不会产

生任何效果，原因不再赘述。事实上，实现算法时，作为优化，当首元素变化的队列属于根结点时，可以不再运行doSkip(Q)。原因是根结点已经没有祖先结点。之后将运行getNextFromStream(a)。由于此刻Que(b)和Que(c)为空，不能确定a1是否具有这两类孩子元素。故本次getNextFromStream(a)返回NULL。相应于STARTELEMENT(a1)的所有处理结束。由此刻直到STARTELEMENT(c2)到来的过程中，所有到来的元素都会被缓存。不会有被移除的元素。而所有getNextFromStream(a)均会返回NULL。STARTELEMENT(c2)到来时，已经可以确定Que(a)队头元素a1具有“最小后代扩展”。这时getNextFromStream(a)会返回结点a。a1被取出，入栈。之后还将继续运行getNextFromStream(a)，依次返回b和c和NULL。相应地元素b1和c2被取出并入栈。最后相应于STARTELEMENT(c2)的处理结束。此时所有队列中只有Que(a)尚不空，内有唯一的元素a2。此后STARTELEMENT(a3)到来，但仅仅会使队列Que(a)中增加该元素。STARTELEMENT(b3)到来，可使a2被移除，a3成为新的队头元素，并使Que(b)中增加元素b3。之后STARTELEMENT(c3)到来，这时getNextFromStream(a)的第一次运行将导致a3被取出。由于a3已经与此时根栈S_a的栈底元素a1盖住不相交的区域，故在a3入栈之前，将会输出栈结构中的解(a1、b1、c2)。之后继续运行的getNextFromStream(a)将依次返回b、c和NULL。导致在b3、c3取出（并入栈）后相应于STARTELEMENT(c3)的处理结束。此后文档中剩下的STARTELEMENT(a4)和STARTELEMENT(c4)相继从流中到来。但在这两次处理中没有元素可以取出或者移除。这是因为元素开始标记到来时运行的算法无法识别出已经没有更多元素会进入Que(b)。这样它无法确定a4是否具有b类型的孩子。因此在最后的标记ENDDOCUMENT到来时，还要运行原HolisticTwigStack算法来处理所有的剩余元素，在本例中为a4和c4。在转换算法时，第一阶段SHolisticTwigStack所构造的栈结构不应销毁。这是因为可能某些解的一部分构成元素是在第一阶段找出的，而另一部分需要在第二阶段补充到栈结构上去。如本例中解(a3、b3、c4)中的元素a3和c3就是在第一阶段由SHolisticTwigStack算法找出的，而元素c4则是在第二阶段由HolisticTwigStack算法找出的。这三个元素要存在于同一个栈结构中才不致丢解。HolisticTwigStack算法运行后将输出剩余的解(a3、b3、c3)和(a3、b3、c4)。之后本查询彻底执行完毕。

4.7 算法分析

本部分证明SHolisticTwigStack算法的正确性。由于SHolisticTwigStack算法中采用的构建栈结构的算法除去要考虑“不确定右位置”的问题以外与HolisticTwigStack算法中的相同。而两个算法枚举小枝查询结果的算法完全相同。不同之处在于，HolisticTwigStack算法中，元素是经由getNext(Q)返回的模式树结点从其队列取出的；而在SHolisticTwigStack算法中，元素是在getNextFromStream(Q)和doSkip(Q)配合下，依据前者指出的模式树结点从其队列取出的。HolisticTwigStack算法的正确性在前一章中已经说明。如3.2.7小节所述，getNext(Q)向栈结构构造算法提供的元素、及提供元素的顺序所满足的四个引理作为充分条件，保证了后续算法可以得出全部正确

的小枝匹配结果。既然SHolisticTwigStack中的“后续算法”与HolisticTwigStack中的相同，只要证明getNextFromStream(Q)和doSkip(Q)一起，为栈结构构造算法提供的元素、及提供元素的顺序同样满足那4个引理，也就证明了SHolisticTwigStack算法的正确性。

引理 6 设有模式树结点P=Parent(Q)，若Que(P)不为空，p∈Que(P)，First(Que(Q))=q，则在算法4运行期间，除去doSkip(Q)正在运行时的任何时刻，有R(p)>L(q)。

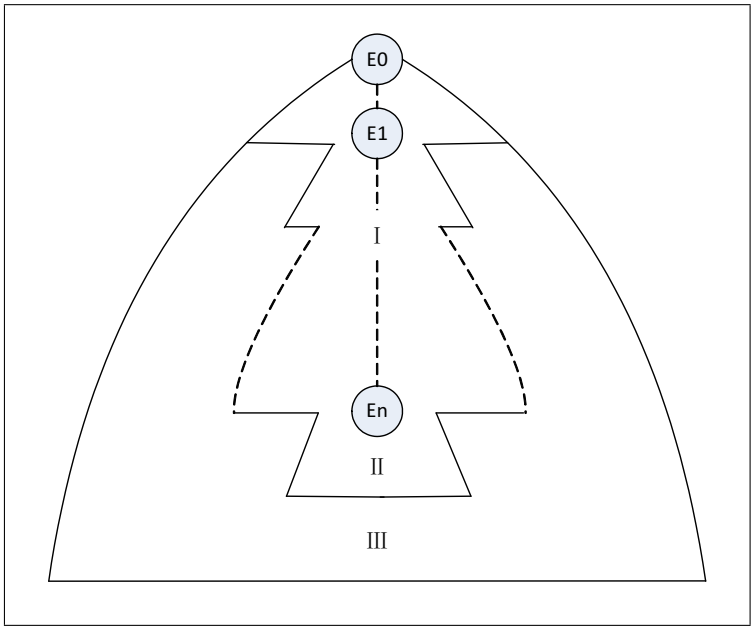

图 4-5 doSkip(Q)所起作用的证明

证明：

在SHolisticTwigStack算法中，凡是有队头元素First(Que(Q))改变时(不包括在doSkip(Q)内部队头元素的改变)，就会调用doSkip(Q)一次。下面采用数学归纳法进行引理的证明，对队头元素在doSkip(Q)过程外部发生改变的次数进行归纳。

1. 在第0次队头元素发生改变，并且运行了0次doSkip(Q)时，也就是还没有队头元素发生改变时，所有队列为空，引理6的前件为假，故该引理成立。

2. 假设第k次队头元素发生了改变，并运行了doSkip(Q)后，引理成立。往证第k+1次队头元素改变，并运行了doSkip(Q)后，引理成立。

   如图4-5，设发生队头元素改变的队列是Que(E_n)。整个模式树分为三个区域，区域Ⅰ是结点E_n的双亲链，区域Ⅱ由区域Ⅰ中结点的孩子结点组成，而区域Ⅲ为模式树剩下的部分。对于引理中所述的P和Q，由于是父子关系，只可能P、Q均处于Ⅰ中，或P∈Ⅰ而Q∈Ⅱ，或P∈Ⅱ而Q∈Ⅲ，或P、Q均处于区域Ⅲ中。

结点P和Q均处于区域III中，或者P处于区域II中，而Q处于区域III中，那么由于doSkip(Q)运行过程中不会改变任何该区域中II、III中结点所关联队列中的元素，因而在doSkip(E_n)结束后p和q应当仍然满足引理中的关系。

若P、Q均处于区域I中，则可以找到一个i，使得P、Q在图中分别为E_i和E_{i+1}。若i+1=n，doSkip(E_n)显然可以在第一次循环结束之后保证$\forall e \in Que(E_{n-1})$，R(e)>L(First(Que($E_n$)))。若i+1<n，当doSkip($E_p$)运行到关于Que($E_i$)和Que($E_{i+1}$)的那一次循环时，如果两个队列中有任何一个在该循环中为空，则本次doSkip(E_n)结束后该队列仍然为空，这样又导致引理的前件为假，引理仍然成立。如果两个队列都不空，在该次循环开始时，remL已经保存了First(Que(E_{i+1}))的左位置。易知，在本次循环结束后，Que(E_i)中任何元素的右位置将必定大于remL。而从本次循环结束直到整个doSkip(E_n)运行结束，这两个队列中元素不再变化，引理中的关系R(p)>L(q)将保持成立。

现只剩P∈I而Q∈II一种情况。首先若P为E_n而Q∈Children(E_n)，队列Que(E_n)发生的改变可能会有两种，一种是由空变为不空，这种情况下因为p是新到来的元素，而q是之前到来的元素，L(p)尚且大于L(q)，就更有R(p)>L(q)了。另一种情况是从Que(P)中取出了元素(传给了栈结构构造算法)，不过由于在此之前L(q)小于Que(P)中所有元素的右位置，现对Que(P)剩下的所有元素的右位置也仍然是这样。其次，若P不为E_n，而是P=E_i，i<n，而Q∈Children(E_i)且Q≠E_{i+1}。这种情况下，从Que(E_n)队头元素改变直到当前，只可能Que(E_P)中的元素有所改变(是在本次doSkip(E_n)运行期间因为移除了某些元素)，而Que(E_Q)中的元素不会有变。由于在doSkip(E_n)中移除某个队列中的元素只能导致队中的最小右位置增大或不变，而在此之前L(q)小于所有Que(P)中元素的右位置，故此后对Que(P)剩下的所有元素的右位置也仍然是这样。

到此证明了第k+1次队头元素改变，并运行了doSkip(Q)后，仍然有R(p)>L(q)。

由数学归纳法，本引理成立。□

定理 6 SHolisticTwigStack算法中，若getNextFromStream(Q)或者getNext(Q)返回模式树结点P，则那时Que(P)的队头元素具有“最小后代扩展”。

算法getNext(Q)具有该性质，已在[3]中给出，此处不再进行证明。下面仅证明算法getNextFromStream(Q)具有该性质。该算法较为复杂，需要对其中具体某行进行分析，下面的证明中出现的行号请参见算法6。

证明：

采用数学归纳法，对以返回值P为根的子树SubTree(P)的树深进行归纳。

1. 如果SubTree(P)树深为1，即P为叶结点。由于P没有孩子结点，First(Que(P))必然具有“最小后代扩展”。另外，观察到整个getNextFromStream(Q)的返回值

必定来源于第3行和第25行的返回语句。如果最终该算法的返回值是叶结点，则它必定来源于第3行的返回语句。因此第二行的条件成立，即同时我们可以得到Que(P)不为空。

2. 假设SubTree(P)树深不大于k(k≥1)时，若getNextFromStream(Q)返回P，则Que(P)队头元素具有“最小后代扩展”，且Que(P)不为空。则SubTree(P)树深为k+1时，往证该结论依然成立。此时P不再是叶结点。考察Q=P的那次递归调用。在该递归调用中，第7行和它以前的部分不可能获得执行机会。又由于最终返回的P必定来源于第3行或第25行，故该次第25行一定被运行。为此，第11行不能被运行。即第10行的条件不成立——n为NULL或者n等于child。又由第13、14行知，若n为NULL，则notSureExist会被置为TRUE，这样即便没有在19行处返回，也会在23行处返回，不会有机会运行第25行。因此n不能为NULL，getNextFromStream(child)返回了child，n==child。而以该返回值child为根的模式树子树的树深至多为k，由归纳假设知P的所有孩子child是具有“最小后代扩展”的，且Que(child)不为空。由于第19行不能被执行，故第18行的条件不成立，即$l_n>l_q$或l_n为INFINITE。后者等价于Que(n)为空。又前面已经说明n==child，故又等价于Que(child)为空。这不成立。故l_n不为INFINITE，只能是$l_n>l_q$。到此，已经得到：

   (a) ∀CH∈Children(P)，First(Que(CH))具有“最小后代扩展”；

   (b) ∀CH∈Children(P)，L(First(Que(CH)))>L(First(Que(P)))

   再由引理6知，∀CH∈Children(P)，L(First(Que(CH)))<R(First(Que(P)))。故P队头元素盖住其所有孩子队列中的队头元素。当SubTree(P)的树深为k+1时，Que(P)的队头元素具有“最小后代扩展”。

由数学归纳法知，算法getNextFromStream(Q)具有定理中陈述的性质。□

引理 7 getNextFromStream(Q)的一次运行可以在有限次的对自身的递归调用后结束。

证明：

采用数学归纳法，对参数Q为根的树的树深dep进行归纳。

1. 当dep=1时，Q为根的树中只包含它一个结点，Q为叶结点。只有第1-7行的部分有运行机会。在其中没有出现更多对自身的调用。就是说这时getNextFromStream(Q)在0次对自身的递归调用后就可以结束。是有限次。

2. 假设dep≤k(k≥1)时，结论成立，往证dep=k+1时结论保持成立。那时，dep≥2，在最外层的调用中，Q不是叶结点，故只可能第8-27行有机会运行。其中，在第9行进行了对自身的递归调用，其中每个调用都是对Q的孩子结点child进行的。

孩子有有限多个，故本层中调用的getNextFromStream(child)亦只有有限个。又以child为根的子树树深至多为k。由归纳假设，那些以child为参数的调用均可以在它们内部的有限次递归调用后结束。故总体上还是有限次。而这已经是dep=k+1时的结论了。

由数学归纳法，本引理成立。□

定理 7 若P、R为模式树结点，P=Parent(R)，p∈Que(P)，r∈Que(R)，p盖住r，则当p尚存在于其队列中时，r不可能被取出。

证明：

首先，如果r是在getNext(Q)的某次运行后取出的，那时根据getNext(Q)的性质，p必定尚存在于队列中。如果不是这样，说明r是在某次getNextFromStream(Q)运行后取出的。

用反证法。假设r在p尚存在于其队列中时取出。取出r之前那一次getNextFromStream(Q)的运行必定返回模式树结点R。R有父结点P，而在SHolisticTwigStack中进行的调用为getNextFromStream(Root)，Root为模式树根结点。故R≠Root。getNextFromStream(Q)的最终返回值如果不是NULL，则它在其最外层调用及其引发的所有递归调用中，一定在第3、11、19、25行中某些行的Q或n的位置出现过。如果该最终返回值仅仅在第3、25行出现过，那么它应该与最外层调用的参数相同。这样，由于getNextFromStream(Q)的最外层调用返回值R与其参数Root不同，且不是NULL，R一定在第11或19行的某次运行时出现在n的位置。

问题在于，是否可能R从未出现在第19行，仅仅在第11行被返回过。如果那样，说明在第9行处的getNextFromStream(child)的返回值R与其参数不同。但是在第19行没有运行机会的前提下，可以返回与调用参数不同值（不是NULL）的位置只有第11行。而在第11行处返回又再次说明在第9行处的getNextFromStream(child)的返回值与其参数不同……这样在第9行进行的递归调用将深入到无限层。这与引理7矛盾。

因此R必定曾经在第19行处返回过。16-20行说明Que(R)的队头元素，即定理中的r的左位置小于等于其父队Que(P)的队头元素p′的左位置。L(r)≤L(p′)。那时p尚存在于其队列中，因此L(p′)≤L(p)。故L(r)≤L(p)。但是由p盖住r可知，L(r)>L(p)。矛盾。

因此假设不成立，在r于getNextFromStream(Q)的某次运行之后取出的情况下，p不可能尚存在于其队列中。这样，在该情况下，定理的结论同样成立。□

定理 8 所有序列中的元素最终或在getNextFromStream(Q)、getNext(Q)和doSkip(Q)外部被取出，或在getNext(Q)和doSkip(Q)内部被跳过。

首先，若运行一系列getNext(Q)，每次都根据其返回的模式树结点从从其文档元素序列的头部取出一个元素，最终是可以穷尽任何XML文档元素序列的。也就是说

序列中的元素或者在某个getNext(Q)内部被跳过，或者在其外部被取出。否则将影响到HolisticTwigStack算法的正确性。在SHolisticTwigStack中，第二阶段开始运行getNext(Q)时所面对的文档元素序列（确切地说在流处理算法中是队列）的内容是经过第一阶段在其中取出和移除元素后形成的。然而那些剩余元素本身组成的也同样是合法的XML文档。因此可以被一系列getNext(Q)穷尽。

到此已经说明了这个定理的正确性，因为只要getNext(Q)具有其中描述的性质，该定理就必定成立。至于第一阶段发生的元素在getNextFromStream(Q)和doSkip(Q)外部被取出，或在后者内部被移除的情况可以在4.6小节的例子中看到。此处不再从理论上证明该情况必定对每一个查询模式和文档的组合都会发生。然而在下文所述的实验结果中可以看到，对通常的XML文档，在第一阶段取出的元素在占绝大多数，一般不会出现大量元素均在流中最后一个元素已经到达后，才经由getNext(Q)指出的模式树结点取出的情况。否则将大大影响SHolisticTwigStack算法的流处理能力。

定理 9 在doSkip(Q)或getNext(Q)中被移除的元素不参与任何解。

证明：

在getNext(Q)中移除的元素不可能参与任何解，这是HolisticTwigStack算法正确性的基础之一。在此处不作证明。下面仅对doSkip(Q)具有该性质进行证明。采用数学归纳法，来证明一个等价的命题——参与某个解的元素e不可能在doSkip(Q)中跳过。在下面的证明中，对e所属的队列所对应的模式树结点为根的子模式树的树深dep进行归纳。

1. dep=1时，e为Que(E)中参与解的元素，其中E在模式树中为叶结点。这样E不可能存在于某个模式树结点的双亲链上，e是不可能在doSkip(Q)内部被移除的。

2. 假设dep≤k(k≥1)时，e不可能在doSkip(Q)中移除。dep=k+1时，考察E的孩子结点为根的子模式树。设e参与的解为$\mathbb{M}$，在那些子模式树中的结点P所关联的队列中存在于$\mathbb{M}$中的某个元素为e_P。由归纳假设，由于P为根的子模式树的最大树深为k，e_P不可能在doSkip(Q)中被移除，而根据getNext(Q)本身的性质，e_P也不可能在该过程中移除。这样，而根据定理9，e_P若要离开其队列，必定是被取出的。由于e_P和e同在解$\mathbb{M}$中，而E=Ancestor(P)，因此e_P盖住e。这样，由定理7知，在e尚存在于其队列中时，e_P也不可能被取出。故在e尚存在于其队列中时，e_P也必定存在于其队列中。

   由于doSkip(Q)的运行只影响Q的双亲链上的结点所关联队列中的元素，要使得e被移除，须运行doSkip(P)，P为以E的某个孩子结点为根的子模式树中结点。doSkip(P)沿着P的双亲链向上迭代，在达到关于Child(E)和E的那一层时，是e被移除的唯一机会。但由于在P的双亲链上，P到Child(E)的所有结点所关联的队列中，都存在某个解$\mathbb{M}$中的元素e′，在到达哪一层时第5行中的maxl不可能大于最大的e′(记为e_{MAX})，即maxl≤e_{MAX}。而e盖住e_{MAX}，有L(e_{MAX})<R

(e)。因此，maxl<R(e)。故Que(E)中的e不可能被移除。这就证明了dep=k+1时解元素e不会被移除。

由数学归纳法，参与某个解的元素e不可能在doSkip(Q)中跳过。□

5. 小枝查询系统的设计与实现

本章通过介绍系统中的模块和模块中的类来说明TwigMiner和STwigMiner系统的设计和实现。

5.1 小枝查询系统的功能

TwigMiner和STwigMiner系统以XPath表达式P和XML文档D作为输入。支持的X-Path语言特性包括名称结点测试、含/和//的路径表达式，包括这种路径表达式以及关于属性的关系表达式、逻辑表达式的单个和连续多个谓词。系统的输出为按照3.1节中所述顺序排列的小枝匹配结果。

5.2 小枝查询系统的模块设计

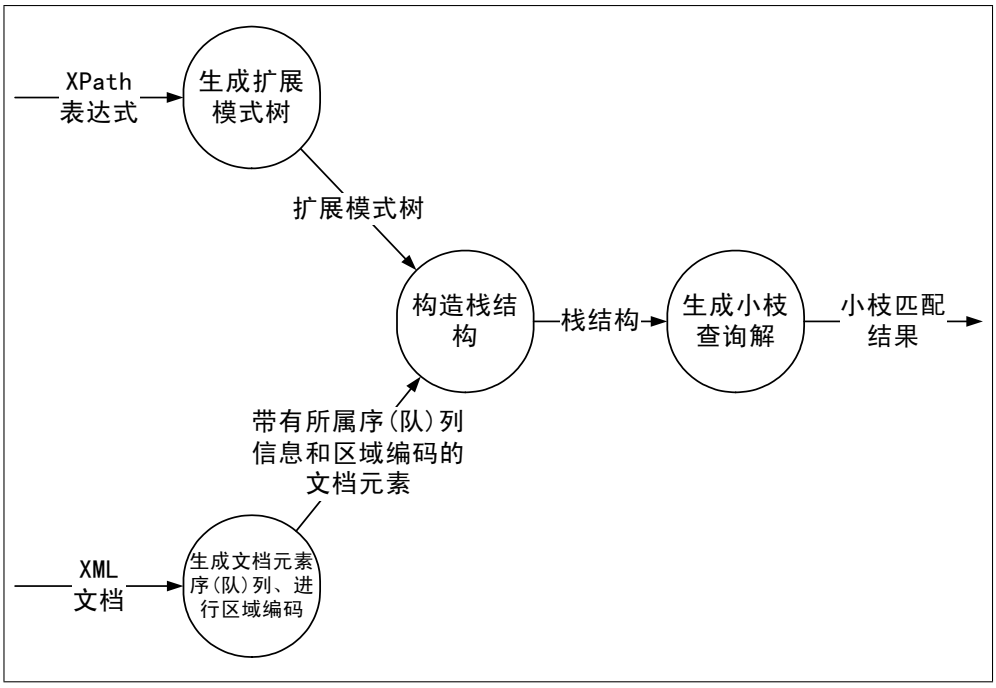

图 5-1　小枝查询系统的结构

TwigMiner和STwigMiner系统的整体设计思想是：将代表查询任务的用户输入转化成HolisticTwigStack和SHolisticTwigStack算法执行查询任务所需要的表示形式，以转化后的数据作为输入，运行HolisticTwigStack或SHolisticTwigStack算法，得到小枝匹配结果。

基于这样的设计思想，两个系统均由四个模块组成，即：扩展模式树生成模块、文档元素序(队)列生成模块、栈结构构造模块、结果生成模块。这四个模块所完成的数据

处理和转换如图5-1中的4个加工所示。其中，模式树生成模块接受用户输入的XPath查询表达式，将其翻译成扩展模式树。扩展模式树的概念见5.2.1小节。文档元素序列生成模块以XML文档作为其输入，之后将文档元素按照其标签名归入不同序列中。同时对序列中的元素应用一种“区域编码”，以在后续处理中更快地计算元素之间的位置关系。栈结构构造模块以前一阶段处理得到的查询模式树和文档元素序列作为输入，在栈结构中存放模式在数据中匹配结果中的那部分元素。这样，栈结构成为了最终小枝匹配结果的压缩表示。结果生成模块以构建好的栈结构作为输入，将其中压缩表示的结果“展开”成最终结果，并按照5.1节所述的顺序输出。其中A、B对应于输入数据的初级转化，而C、D对应于核心的HolisticTwigStack算法的执行。

5.2.1 扩展模式树的生成

由于TwigMiner和STwigMiner系统除去支持能够产生所有小枝模式的XPath语言特性外，进行了少量扩展，主要包括对谓词中的属性、逻辑表达式和关系表达式的支持。相应地，对小枝查询问题中的模式树进行了扩展，使其可以体现谓词中的属性、逻辑和关系表达式。在查询过程中，当需要判断某个此类谓词是否满足时，就访问扩展模式树中与此类谓词相关的子树，同时运行算法，计算出表示谓词判断是否通过的布尔值。扩展模式树构造模块在TwigMiner和STwigMiner系统中是完全相同的。本小节说明扩展模式树概念及其在这两个系统中的构造过程。

扩展模式树概念

小枝查询问题中的查询模式树可用二元组(𝕍, 𝔼)表示。𝕍表示模式树结点集合，每个结点V∈𝕍关联一个标签名，XML文档D中具有该标签名的所有元素与该结点V匹配。𝔼表示模式树中两个结点之间关系的集合，关系分为“AD”和“BC”两种，前者表示具有该关系的两个模式树结点所匹配的文档元素间只能是父子关系，后者表示它们只要是祖先——后代关系即可。

现使得：若XPath表达式中与结点V∈𝕍对应的类型受谓词（关于属性的算术、逻辑表达式）约束，则使结点V具有子树ST，ST为该谓词中表达式的抽象语法树。则经过这一修饰后形成的就是扩展的模式树。

如图5-2中(a)、(b)所示，结点a、b、c及它们之间的关系是属于基本的小枝查询问题的模式树的。由于XPath表达式中的b具有无法并入该模式树的谓词，它是一个关于属性（@id、@num）的逻辑表达式，因此将该表达式的抽象语法树（以or为根结点）作为b的子树。这就形成了扩展模式树。

在执行查询时，如果按照HolisticTwigStack算法，某个与模式树结点b（a的孩子）匹配的文档元素可以入栈，则再找到结点b下面表示该谓词的子树，通过访问该子树就可以判断该谓词是否满足。如果不满足，则仍然舍弃该文档元素即可。

另外，模式树中的非扩展结点可能会共享同一个类型，如图5-2的(b)中存在两个b类型的结点。为区分这样的结点，为扩展模式树中所有的非扩展结点分配一个ID号。

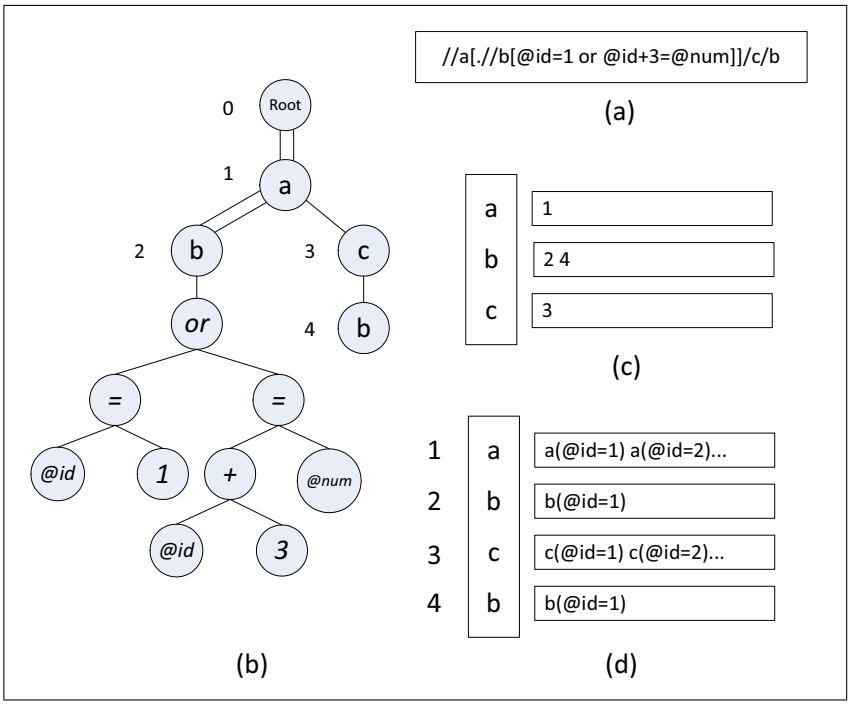

图 5-2　扩展模式树及文档元素序列示例

某个结点得到的ID号就是该结点的先序位置(以0表示根结点Root的先序位置)。在图5-2的(b)中非扩展结点Root、a、c分别具有ID号0、1、3，而两个共享类型b的结点分别具有不同ID号2、4。

此外，XPath表达式中除去第一个“/”或“//”关系，其他的关系都明确指明了其所约束的父、子结点。只有第一个“/”或“//”没有显式地指明父结点。为了处理的一致性，人为在扩展模式树中加入一个总的根结点，使原“根结点”与新加入的根结点具有第一个“/”或“//”所指出的PC或AD关系。令总的根结点的ID号为0。

为了下面讨论方便，在扩展模式树PT中，记结点Q的父结点或双亲结点为Parent(Q)，孩子结点集合为Children(Q)，祖先结点集合为Ancestors(Q)。此外，在下文中出现的所有“模式树”也均指本小节所述的“扩展模式树”。

扩展模式树的生成

由XPath表达式可首先生成其语法树。采用编译器前端生成工具SableCC来完整语法树的生成。为此只需编写需要支持的XPath语法规则，之后运行SableCC解析这些语法规则，即可生成将XPath表达式转化为语法树的类。

在由XPath表达式所生成的语法树的基础上，可设定合适的语义规则，应用VISITOR模式按先序顺序访问该语法树，生成扩展模式树。为应用VISITOR模式，可以设置一个记录栈RS来记录翻译所处的状态，为访问下一个语法树结点时决定所做的动作提供上下文信息。

能够产生模式树中非扩展部分结点的XPath语法规则形如：

$$path\_expr\_part \ \rightarrow \ '/' \ step\_expr \mid '//' \ step\_expr \tag{5.1}$$

$$step\_expr \ \rightarrow \ id \ \ predicate* \tag{5.2}$$

其中“id”代表XPath表达式中的名字结点测试，也就是与模式树结点关联的需要与文档元素名称匹配的标签名。在访问规则5.2右侧的id对应的语法树结点之前，需保证RS栈顶存有id在模式树中的父结点及其与id的关系(/或//)。由于语法树是先序访问的，id在模式树中的父结点所对应的语法树结构已经被访问过，在那时就可以将为那个父结点建立的数据结构入栈。而它们之间的关系可以在访问规则5.1右侧对应的语法树结构时获得并入栈。这样在访问到id时(访问其子语法树之前)，就可以构造表示id结点的数据结构，并将其作为表示id在模式树中父结点的数据结构的孩子，挂接于其下，同时设定它们之间的关系。如果当前结点（非扩展结点）是谓词中的第一个结点，则之前最靠近栈顶的非扩展结点V_0还不能出栈，因为谓词对应的语法树结构结束后可能还会遇到表示V_0在模式树中的孩子结点的语法树结构；否则需要将栈顶的刚刚使用过的结点和关系出栈。为了向后面的处理传递“不应出栈”的信息，在进入谓词时需要向RS中压入标记PR(PREDICATE的缩写)。

前段所述为生成模式树的非扩展部分的方法，而生成扩展部分时只要采用生成逻辑、关系表达式的抽象语法树的方法即可。要保证该抽象语法树生成之后马上作为子树挂接到其所约束的非扩展模式树结点之下。

对于图5-2中的查询，生成扩展模式树时记录栈的变化如图5-3所示。图中Rt表示ID号为0的总的根结点。最终生成的扩展模式树见图5-2的子图(b)。

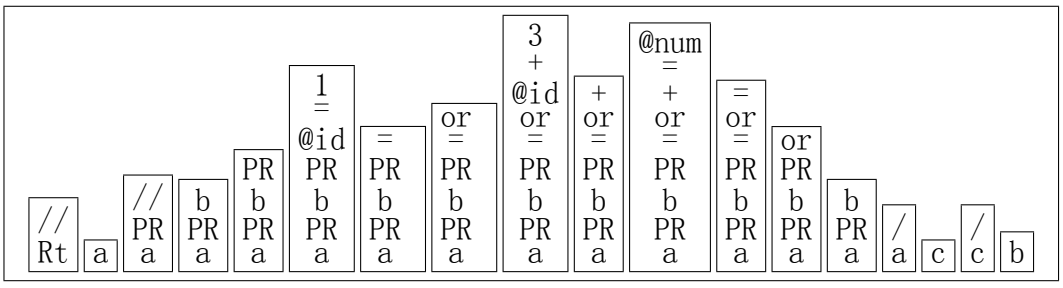

图 5-3　构造扩展模式树过程中记录栈的变化

5.2.2 带区域编码的文档元素序列或队列的生成

在TwigMiner和STwigMiner系统中，核心算法所需要的输入分别为来自文档元素序列和队列的XML数据元素。为此分别需要由XML文档构造文档元素序列和队列。另外，对进入序列和队列的元素均需要进行区域编码。本小节讨论序（队）列及区域编码的具体实现机制。

文档元素序列或队列的ID号

由于每个文档元素序列或队列关联一个模式树结点，令该模式树结点的ID号为该文档元素序列或队列的ID号。在XPath表达式中以及扩展模式树中，每个类型可能出现多次，如图5-2的(a)和(b)中类型b就出现了多次。相应地该类型的文档元素序列或队列就有多个。这些序列虽然共享相同的类型，但它们是各自独立的。元素从这些序列或队列中的一个取出时不影响其他序列或队列的内容。图5-2的(d)中有两个以b为类型的序列或队列。分别关联(b)中ID号为2、4的模式树结点，因而这两个序列或队列的ID号分别为2和4。

将文档元素放入某个序列或队列

每个文档元素有一个标签名，它需要放入所有以该标签名为其类型的序列或队列中。如果为每个序列或队列保存其类型，再将元素标签名依次与这些类型比较，如果与某个类型相同，就将元素放入对应的序列或队列中，这种做法需要的比较次数过多。不过注意到文档元素序列或队列是具有与模式树结点相应的ID号的，该ID号是连续的整数，因此可以把各个序列组织在一个数组中。如果能用较少的时间由标签名（也就是类型）获得具有该类型的序列或队列在数组中的下标，之后就可以用随机访问从数组中找出元素需要放入的序列或队列了。为此采用散列表。图5-2中(c)就是一个为此目的生成的散列表。散列表中，由某个标签名（或类型）经过hash将得到一个存有一个或多个整数的表numList，该表中存放的整数就是序列或队列的数组（如(d)所示，其中元素的id与序列的ID无关，是用来标示具有相同标签名的不同元素的）中当前元素需要放入的序列或队列的下标。如标签名为b的元素需要放入的就是下标为2、4的序列或队列。

下面说明生成该散列表的方法。我们只需在生成某个模式树结点时，对该结点的类型进行hash，找到其需要映射到的numList，之后向该numList中追加其ID号即可。如在生成ID号为4的那个结点b时，对b这个字符串进行hash，找到的numList中已经有一个2，再将4加入进去即可。

在getNext(Q)、getNextFromStream(Q)和doSkip(Q)中，要由模式树结点找到其所关联的文档元素序列或队列。这是很简单的，因为模式树结点与其所关联的文档元素序列或队列具有相同的ID号。

给文档元素加入区域编码

事实上，1.2小节中说明区域编码概念的同时就已经给出了生成区域编码采用的思想。只不过那里是以XML文档树为基础讨论的；而本系统中并未构造文档树，是用工具将XML文档以从前到后扫描时遇到的一系列开始和结束标记的形式呈现给后续处理工序。容易想到，元素开始和结束标记的出现顺序与先序遍历文档树时进出元素子树的顺序是一致的。

但还是有其他的不同之处。采用递归过程先序遍历文档树，在离开某元素的子树

时，直接将产生的右位置赋给该元素即可(只要序列中的元素和文档树中的元素在内存中具有相同位置)。而如果没有文档树，由元素的结束标记本身只能访问到该元素的标签名，而无法直接知道该元素的内存位置。一种可能的解决方案是从后向前扫描该元素所在序列，直到找到第一个右位置未曾赋值的元素，将当前计数器的值作为右位置进行赋值，之后立即结束循环。但这样做可能会有比较大的开销，因为当结束标记所对应的元素在文档中是某个该类型元素的多层嵌套结构的外层(或在文档树中是一个层次较深的子树的根，而该子树中有多个该类型元素在一条双亲链上)时，本次右位置所属于的元素将远离其序列的末尾。图5-4就是一个例子。(a)为XML文档，其中有很多层嵌套的标签名为a的元素，并且假设没有任何模式树结点是b类型的。(c)为a类型的一个序列。当最外层的到来时，若采用上述方法给序列头部的那个a的右位置赋值，就要扫描整个序列。

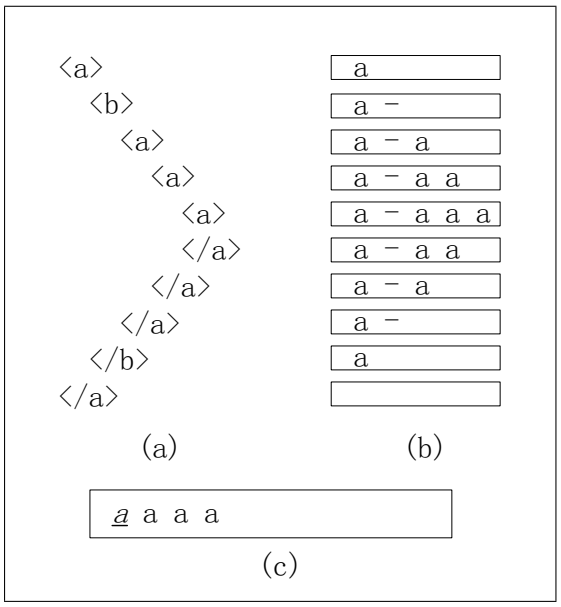

图 5-4　右位置所属元素的定位

为了避免这种开销，采用一个记录栈，在其中存放所有结束标记尚未到来的元素。每读到一个开始标记，若在模式树中存在结点具有标记的标签名，就将为该元素建立的数据结构中左位置赋值，并将该数据结构压入栈中；否则不必为该元素建立数据结构，也不必增加计数器的值，将一个特殊记号（就是图(b)中的-）压入栈中占位即可。当元素结束标记到来时，由XML文档中标记的嵌套性可知，对栈顶元素出栈，出栈的数据结构就是之前为拥有该结束标记的文档元素建立的。只要出栈的不是用来占位的记号，就可以增加计数器的值并赋值给该数据结构中的右位置。图(b)表示了记录栈的变化(从上到下与读到的标记对应)，每个栈的增长方向是从左到右的。

5.2.3 栈结构的构造

栈结构构造模块包括两个子部分，第一部分提供从文档元素序列或队列中取出元素的功能；第二部分提供决定取出的元素能否进入栈结构，并在可以进入栈结构时将其放入栈结构中合适位置的功能。

在TwigMiner和STwigMiner系统中，本模块的运行时机不同。在TwigMiner中，当全部属于模式树中某类型的文档元素都已经进入相应的文档元素序列后，才开始运行本模块。而在STwigMiner中，每一个属于模式树中某类型的文档元素开始标记到来时，就需要考察是否需要运行本模块。此外，在这两个系统中，本模块的第一部分内部的实现也非常不同。在TwigMiner系统中，该部分使用算法HolisticTwigStack中的getNext(Q)；而在STwigMiner系统中，该部分主要使用算法SHolisticTwigStack中的getNextFromStream(Q)和doSkip(Q)。本模块的第二部分在两个系统中的实现几乎完全相同，除了在STwigMiner中构造栈结构时可能需要处理元素的不确定右位置，而在TwigMiner中不需要该处理以外。

本模块采用的主要算法已经在前文中进行了详细的讨论。在实现中，仍然存在的问题主要在于为栈结构设计合理的数据结构。

为栈结构设计具体的数据结构的难点不在于栈本身，而在于由栈引出的连接。对于2.3节所述的第二类连接，由每个栈元素只引出1条或0条。这是因为这种连接指向同种元素，也就是说只能是单一种类的元素。这是很容易实现的。而第一种连接，即父-子类型的连接，却往往会有多个。因为栈元素往往会具有多个类型的子元素。如图3-1的(d)中，栈元素b5就同时具有c和d类型的子元素，因而需要引出两条连接。由此，连接数将与子元素类型数相同。然而，模式树中每个结点的子结点数目是不同的，导致每种栈元素的子元素类型数也不相同。如果为栈元素中的父-子连接分配固定大小的空间，则必须计算模式树中结点的最大扇出，才能知道分配多大空间可以避免越界访问。即便这样，空间浪费又成为一个问题。当扇出数参差不齐时，随着栈结构的增长，浪费的空间也同时增长。最好可以根据栈元素对应的模式树结点找到其子元素的类型个数，之后动态分配相应的空间。该空间可以以动态数组的形式存在。事实上，我们只需在栈元素的数据结构中保存记录其所对应的模式树结点的域，即可实现为不同类型栈元素分配不同大小的空间，以保存其所具有的父-子连接。

譬如为图3-1(d)中的b_5分配了具有两个元素的动态数组来容纳指向c_2和d_3的连接，算法需要知道指向c_2和d_3的连接分别用该数组中的哪个元素存放。首先，为了保证结果枚举的顺序，最好在栈结构中的各处，这种动态数组中的位置到其所容纳的连接指向的元素类型的映射是固定的。此外，这种映射最好在初始化阶段就计算好，在需要进行新的连接时只需查询那时的计算结果，而不是每次都进行单独的计算。基于这样的想法，设计数据结构childTypeMap用来保存这种映射关系。为图3-1中的查询模式所计算的childTypeMap如图5-5所示。

图5-5中的行左侧的数字代表引出连接的父元素所对应模式树结点的ID号，列上方的数字代表连接指向的子元素所对应模式树结点的ID号。行列交叉处如果有数字，该数字代表该连接在父元素里动态数组中的下标，否则代表父元素没有该类型的子元

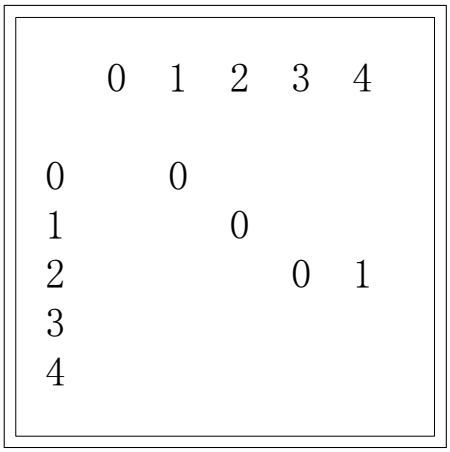

图 5-5　保存栈间连接到其所使用内存位置的映射的数据结构

素。如第三行的意义是：ID号为2的模式树结点b具有ID号为3、4的孩子结点c和d，因而b类型的栈元素具有c和d类型子元素，且指向这两类子元素的连接分别使用b的动态数组中的下标为0和1的元素存放。第一行是关于人为添加的模式树根结点的，该结点在XPath表达式中并不出现，其ID号为0，具有唯一的子结点，也就是XPath表达式中出现的查询模式的根结点。在需要建立新的连接时，查询该childTypeMap获得连接存放位置是非常快的操作。然而，省去的时间是childTypeMap本身占据的空间换取的。不过，该空间是固定的，只与模式树大小有关(而在大多数情况下模式树都不会太大)，不会随着栈结构的增长而增长。故到此已经比较好地解决了栈元素间连接的问题。

5.2.4 小枝查询解的生成

在TwigMiner和STwigMiner系统中，由栈结构生成小枝查询解的模块几乎完全相同，内部均使用HolisticTwigStack的结果枚举算法实现。不同之处仅仅为：前者将所有结果输出到内存，并使结果集可以由程序引用和迭代；而后者只在内存中保留产生后续结果需要的部分已经产生的结果(以防止结果耗尽内存空间，因为需要使用STwigMiner在超大数据集中查询)。

本部分所需要的主要算法已在上文中详细讨论，在实现中仍然面临的问题主要是数据结构的设计。两个主要的数据结构是保存匹配结果的matchList和保存孩子解的segmentsList。matchList中的每一个元素是一个小枝匹配结果。

由于经常需要将两组部分结果连接起来(如算法3中第4、19、27、29、35、36行中的连接操作)，应对matchList使用链式存储结构以避免连接两个表时重新分配空间和拷贝元素的时空开销。值得注意的是，若要对两个链表进行连接，需要直接操作元素指向前驱、后继的previous、next引用，改变它们的指向。但Java本身的LinkedList类将每个元素及其previous、next引用封装在了其私有内部类Entry中，外界无法直接对其进行操作，而LinkedList类本身又没有单独提供不进行拷贝而连接两个链表的方法。

故只能使用自定的链表数据结构。

segmentsList 中保存着1个或多个matchList，其中的每个matchList代表由当前元素的后代传送至当前元素的孩子解。由于要保持其中的matchList 的有序性，需要对 segmentsList 进行快速排序或者对需要插入 segmentsList 的matchList 在 segments-List 中的插入位置进行二分查找。故 segmentsList 需要采用顺序存储结构实现。

5.3 小枝查询系统的类结构及类间协作

本节说明小枝查询系统中实现各模块的主要类，各个类的职责，以及系统运行时类间的动态协作关系。

类结构

TwigMiner 系统的类图见图5-6，STwigMiner 系统的类图见图5-7。两个系统中，属于模式树生成模块和结果生成模块的类是基本相同的。下面逐个说明实现各个模块的类。介绍两个系统中都具有的、职责和接口相同的类时，不再交待具体的系统。

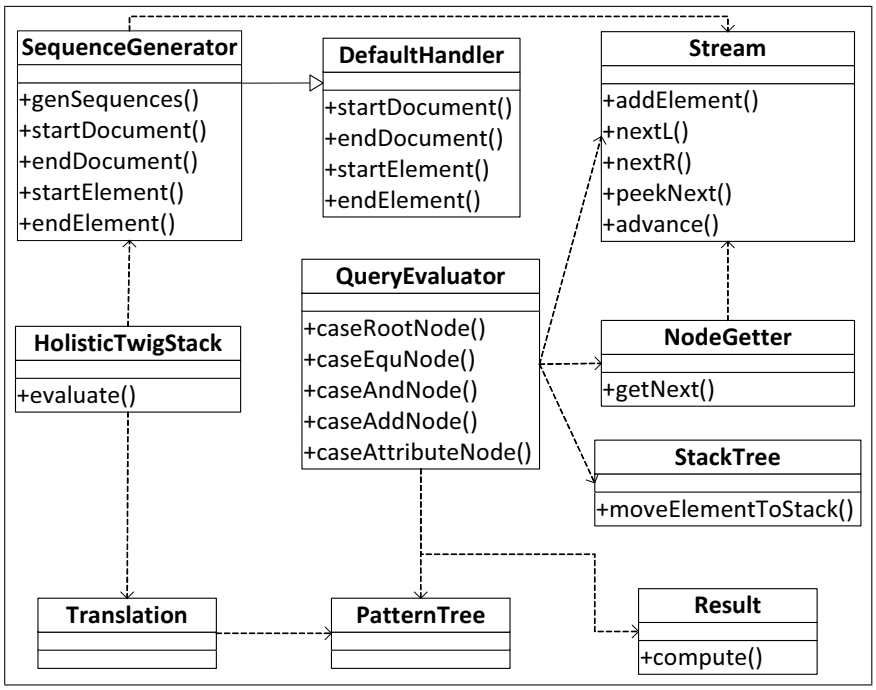

图 5-6 TwigMiner 系统的类图

模式树生成模块中的类

Translation：XPath 表达式语法树的VISITOR，用来生成模式树和算法运行所需要的其他一些数据结构。

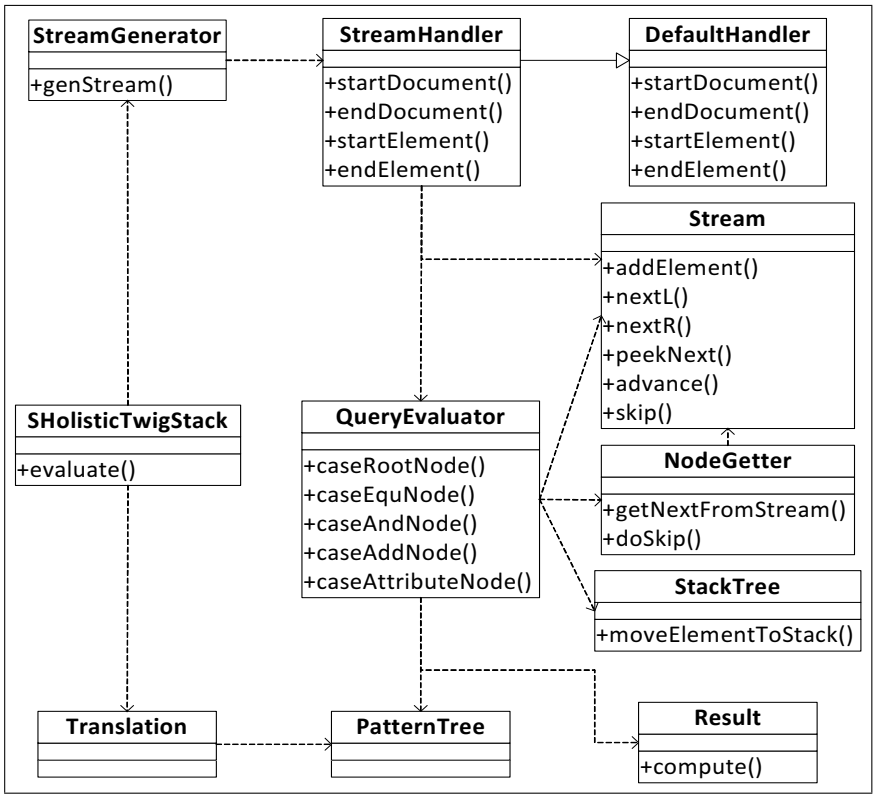

图 5-7　STwigMiner 系统的类图

PatternTree：代表扩展模式树，由该类对象可以访问到整棵扩展模式树的总根结点，进而访问其他结点。

文档元素序列生成模块中的类

SequenceGenerator：属于 TwigMiner 系统的类。其方法 genSequences()用来触发 SAX对XML文档的解析。该类同时为 SAX 事件的处理器，在 SAX API 的回调方法中将新解析出的文档元素加入某个文档元素序列中，并对其进行区域编码。

StreamGenerator：属于 STwigMiner 系统的类。其方法 genStream()用来触发SAX对XML文档的解析。

StreamHandler：属于 STwigMiner 系统的类。实现了 SAX 事件处理器。在回调方法中将新解析出的文档元素加入某个文档元素序列中，并对其进行区域编码。而后判断是否有必要运行 doSkip(Q)、getNextFromStream(Q)和其他后续算法，并运行有必要运行的算法。

Steam：在 TwigMiner 系统中用来管理单个文档元素序列，在 STwigMiner 系统中用来管理单个队列。在两个系统中均提供加入新元素、获得首元素及其左右位置、跳过首元素等接口。在 TwigMiner 系统中，跳过的首元素可以不从序列中移除；而在 TwigMiner 系统中，跳过的首元素当即从队列中移除。此外，在后者中，Stream 类提供专门的方法 skip()，用来依据某个后代的左位置移除队列中的无用元素。

DataElement：存放XML数据元素的信息，包括标签名、属性，还包括元素的左、右位置和层数。

栈结构构造模块中的类

NodeGetter：在TwigMiner系统中封装算法getNext(Q)，负责按照其要求给出栈结构构造算法需要的下一个元素；在STwigMiner系统中封装算法getNextFromStream(Q)和doSkip(Q)，按照前者的指示给出栈结构构造算法需要的下一个元素。

TwigStack：类图中未画出。代表算法中的栈，记录了栈所对应的模式结点，以StackElement类对象作为栈元素。关于连接的信息记录在栈元素中而不是栈中。

StackTree：代表栈结构。由此可以访问到根栈，以及每种类型的最后一个栈。在其中封装了栈结构构造算法。

StackElement：类图中未画出。代表栈元素。维护栈结构中的第一类和第二类连接。同时还维护该栈元素所属的栈，该栈元素的最后一个子栈，以及其segmentsList。

小枝查询结果生成模块中的类

Result：封装了结果枚举算法，负责由栈结构枚举小枝匹配结果。

MatchList：表示结果枚举算法形成的部分或最终解。其中的每一个元素为MatchedElement对象的一个数组，表示一个匹配。其中维护了第一个匹配中第一个MatchedElement的左位置，这是维护MatchList对象在SegmentsList对象中顺序的依据。

SegmentsList：表示后代元素“上传”至当前元素的全部孩子解。其中的每一个元素为一个MatchList对象，代表一组由后代元素“上传”到此的孩子解。本类对象用来收集某个栈元素与后代元素共享的孩子解。

MatchedElement：代表解元素。其中维护MatchedElement对象的引用，由该引用可以访问到解中需要提供的元素标签名和属性等信息。一个小枝匹配包括属于模式树中各个类型的各一个解元素，用MatchedElement对象的一个数组表示。

同时属于多个模块的类

QueryEvaluator： 模式树的Visitor。它首先访问模式树的根结点，在那里利用NodeGetter对象获得下一个算法需要的XML结点，再利用StackTree对象的入栈方法建立栈结构，并在适当时候调用Result对象的相关方法产生最终结果。QueryEvaluator在遇到不适宜归入主体小枝模式的谓词时访问相应的模式树（事实上是扩展了的模式树）结点（如表示and、or、not、=、>=、<=、>、<、+、-、*、idiv、mod、@的结点）对谓词是否满足进行计算。

类协作

TwigMiner和STwigMiner系统的顺序图分别如图5-8和图5-9所示。空间所限，没有画出所有的对象和对象间所有可能的协作关系，而只画出了典型的交互中对象间所发送

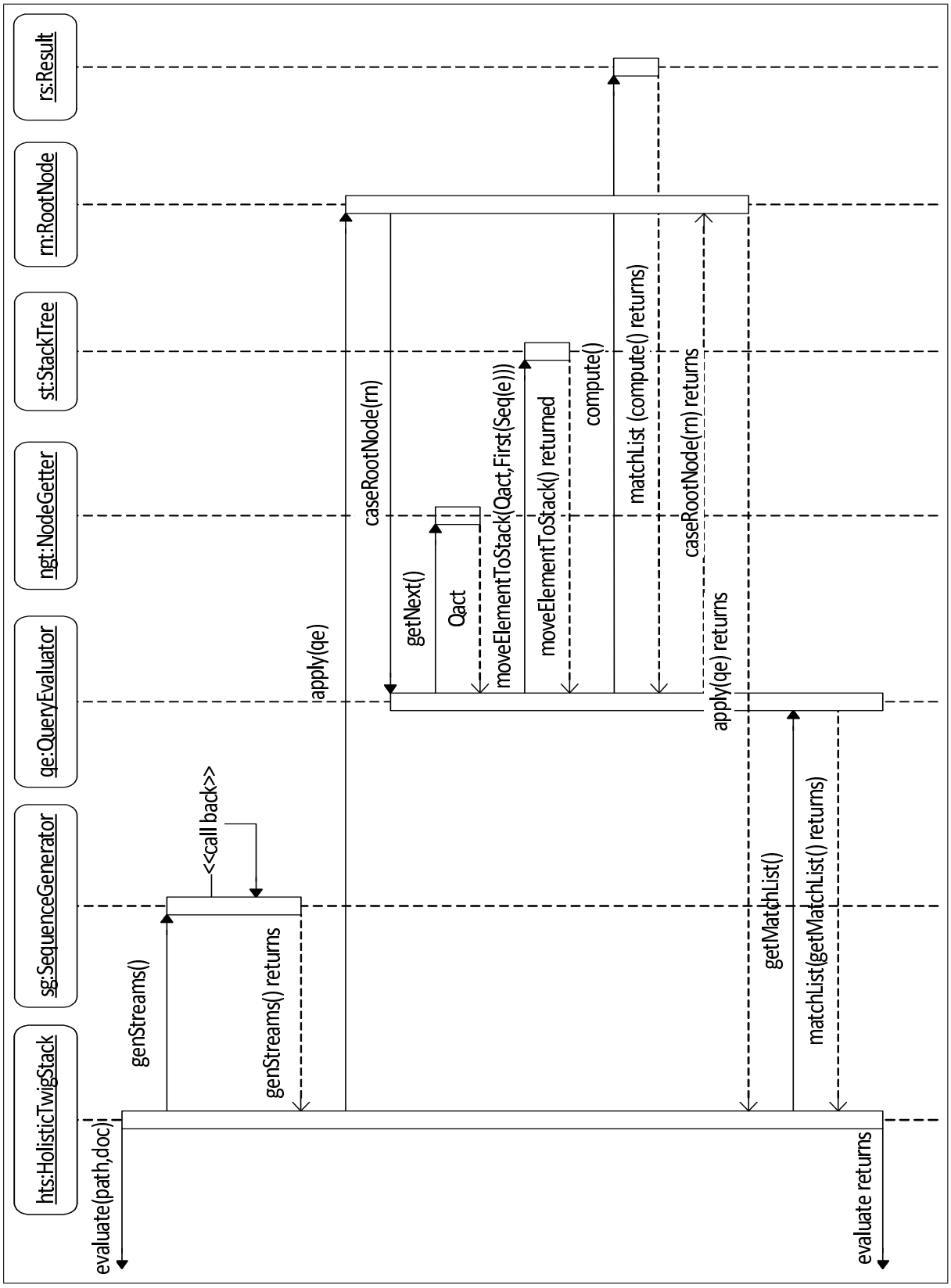

图 5-8 TwigMiner 系统的顺序图

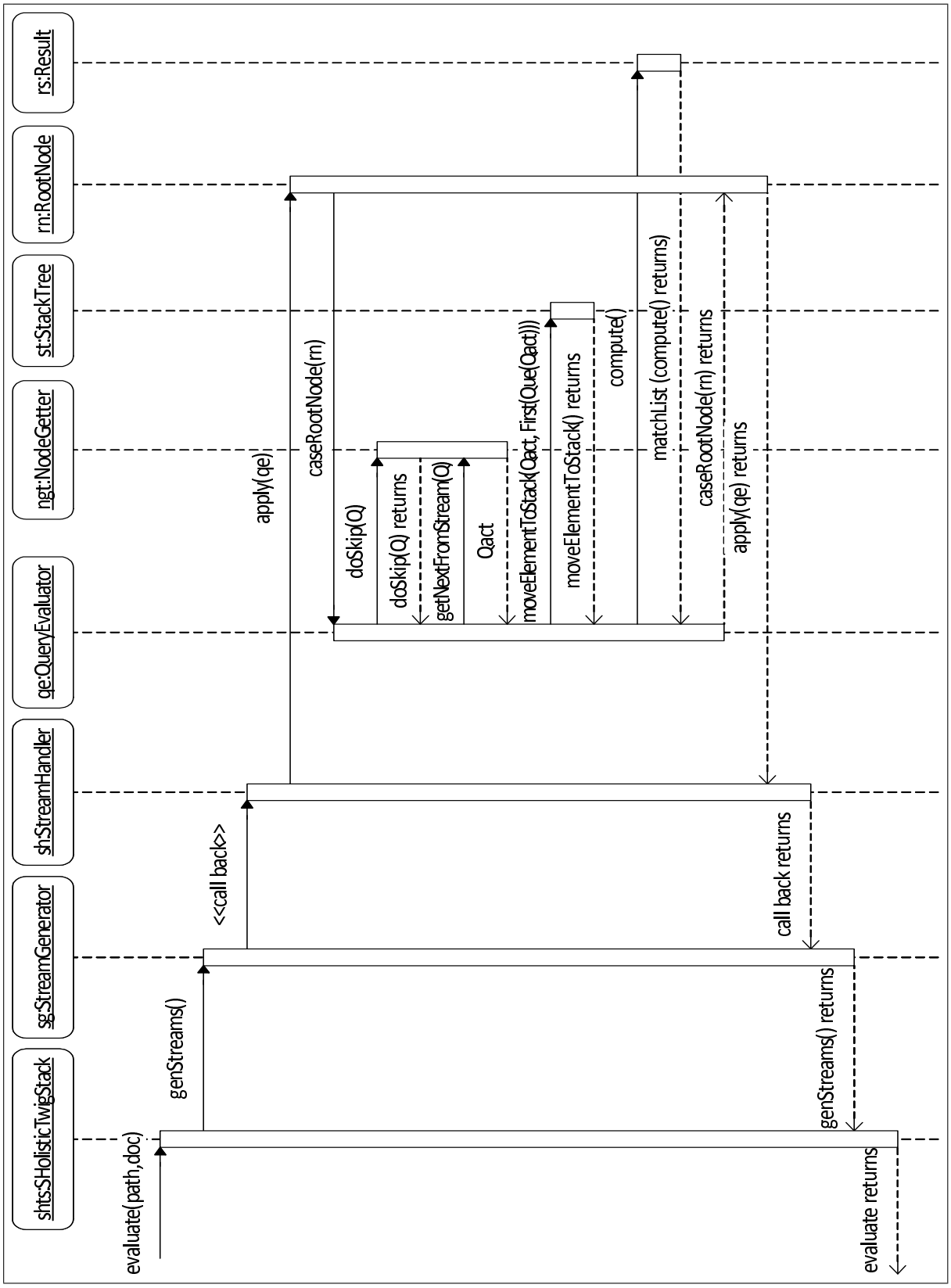

图 5-9 STwigMiner 系统的顺序图

的消息；另外，对象的构造均未画出。

两个系统都从主类对象收到消息evaluate(path, doc)时开始运行。主类对象会构造后续处理所需要的对象，包括文档元素序列产生器、流数据产生器、流数据处理器、模式树访问者、NodeGetter类对象等。

在TwigMiner系统中，主类对象会向文档元素序列产生器发送消息genStreams()，后者收到消息后开始解析路径为doc的XML文档。在完成某个开始或结束标记的解析后，文档元素序列产生器会向自身发送消息«call back»，导致其中的回调方法、也就是处理相应SAX事件的方法被调用，具体调用的方法名称可能为startDocument、endDocument、startElement、endElement。图5-8中只画出了其中的一次回调。事实上则应为一系列对上述4个方法的回调。在回调方法中将建立数据元素对象（DataElement类对象），并将这些对象进行区域编码，放入相应的文档元素序列（Stream类对象）中。在TwigMiner系统中，当所有的回调结束后，也就是整个文档中的相关类型元素都在文档元素序列中进行缓存之后，才开始运行后续算法。在顺序图中表现为genStreams()方法返回后，才开始向模式树的总根结点rn发送消息apply(qe)，请求其接受访问者qe的访问。

而在STwigMiner系统中，主类对象会向流数据产生器发送消息genStreams()，导致同名方法的调用。收到消息后流数据产生器同样开始解析路径为doc的XML文档，而完成文档标记的解析后向流数据处理器发送消息«call back»，调用在后者中注册的回调方法。每次开始标记解析完成所引起的回调中都会对是否需要运行后续算法进行评估，并在需要时运行之。这在顺序图中表现为QueryEvaluator类对象qe、NodeGetter类对象ngt和StackTree类对象st和Result类对象rs的一系列交互完成之后，回调方法才返回。

QueryEvaluator类对象qe、NodeGetter类对象ngt和StackTree类对象st和Result类对象rs的一系列交互表现核心算法的运行。由于在TwigMiner和STwigMinere系统中采用的部分算法不同，在这两个系统中，发送的消息也略有不同。在TwigMiner中，qe向ngt发送getNext(Q)，以请求后者通过算法getNext(Q)给出下一个需要取出的元素所来自的文档元素序列。而在STwigMiner中，qe向ngt发送getNextFromStream(e)和doSkip(Q)，以请求后者通过算法getNextFromStream(Q)和doSkip(Q)给出下一个需要取出的元素所来自的队列。注意，事实上这些交互是反复进行的，直到一些条件不再成立。在顺序图中，空间所限，没有表现出相关条件、及在那些条件下消息的多次发送。图中仅仅表现出了一次典型的交互中可能出现的主要消息及其顺序。

最后，在TwigMiner系统中，当查询完成时，主类可以向qe发送getMatchList()消息，请求后者返回内存中存放全部匹配结果的对象的引用。而在STwigMiner系统中，全部结果已经在查询运行的同时输出到终端。出于空间上的考虑，没有在内存中保留结果的副本，因此也无法再获取全部结果的内存引用。

6. 小枝查询系统的测试

本章介绍对两个小枝查询系统TwigMiner和STwigMiner进行的测试工作。

首先，在前文中对系统采用的主要算法进行了大量的证明，但是由算法到系统中的程序代码仍有一定距离。程序代码中存在大量的细节性处理，要从理论上论证稍微复杂的程序的正确性非常难。为了尽可能保证程序正确实现了算法，使得系统功能与约定的一致，对这两个系统进行了功能测试。

其次，XML数据查询算法的提出目的就是为了提高查询执行的性能，因此需要对系统进行性能测试。然而，虽然课题组已经实现了某些其他小枝查询算法，如Twig²Stack、TwigList、TwigFast等，但实现这些算法的小枝查询系统由不同人编写，往往对输入、输出具有不同的约定(譬如支持的XPath表达式的语法特性不同，输出结果时遵循的顺序不同等)。因此并不容易使用这些系统进行算法查询性能的对比。本章主要通过系统运行时某些部分对时间和内存的使用情况来说明两个系统中对HolisticTwigStack算法改动的部分不会成为算法执行时间的瓶颈，另外流处理算法在通常的数据集上是有效的(不会退化为非流处理算法)。

6.1 功能测试

为了验证系统具有目标功能，将TwigMiner和STwigMiner系统的输出与一段在功能上与之等价的XQuery程序的执行结果进行对比。为了增强测试的自动化程度，XQuery程序由一段翻译程序从XPath表达式所指定的小枝查询目标自动生成。

Qizx[11]是一种XQuery引擎，支持使用XQuery语言进行XML文档或数据库中的数据操作。它包括一组API，可以在Java语言程序中使用，执行内嵌的XQuery查询。当等价的XQuery程序生成之后，在测试驱动程序中调用相应的API，执行XQuery程序所表示的查询，并返回结果。

为了区分结果中具有相同标签名的不同元素(如<a/>和<a/>)，为每一个文档元素添加一个属性@id，使得那些具有标签名的不同元素具有按自然数递增的@id值(如<a @id=1/>,<a @id=2/>,...)。当XML文档稍大时，使用基于DOM或StAX(两种XML解析器)的小程序自动向文档中加入@id属性。在此基础上，为了使TwigMiner和STwigMiner系统生成的结果与XQuery程序生成的结果具有可比性，在程序中将两个结果都转化为一系列@id值组合的形式。

最终，只要向某个指定目录中放入某个XML文档和代表在其上的查询目标的XPath表达式，并对文档运行产生@id的程序向其中的元素加入@id属性值。就可以运行测试驱动程序，返回关于小枝查询系统在该文档上的查询结果是否正确的信息。

实现自动化的功能测试的框架见图6-1。

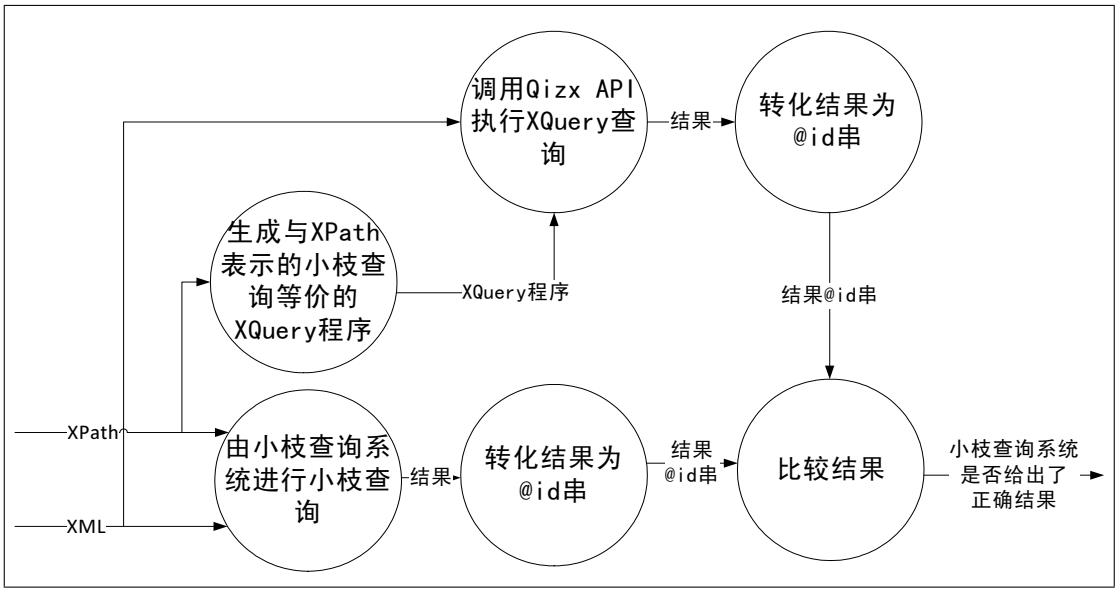

图 6-1　自动化的功能测试框架

表 6-1　性能测试中使用的查询模式

名称	数据集	小枝模式
TreeBank-Q1	TreeBank	//S[.//VP][VBZ]//NP[NP//PP[NP]//PP] //PP[.//IN]//NP[DT]//NP//NNP
TreeBank-Q2	TreeBank	//S/VP//PP[.//NP/VBN]/IN
TreeBank-Q3	TreeBank	//S[VBZ][.//VBP]/VP/PP[NP]/IN
DBLP-Q1	DBLP	//inproceedings[@id mod 1000 = 0][title]/author
DBLP-Q2	DBLP	//article[@id < 20 or @id mod 100 = 0][author] [.//title]//year
DBLP-Q3	DBLP	//inproceedings[@id mod 100 = 0][author] [.//title]//booktitle[@id mod 50 = 0]

6.2 性能测试

性能测试使用的硬件环境包括英特尔1.6GHz主频CPU、1.5GB内存，软件环境为Java VM 1.6。使用的测试数据集包括TreeBank和DBLP，这两个测试数据集可以在[12]中找到。前者是一种用计算机程序生成的树结构较深、较不规则的数据集。而后者是真实的数据集，并且树结构较浅、较规则。在每个测试数据集上进行三个不同的小枝查询，表示小枝模式的XPath表达式见表6-1。

性能测试分为如下三个部分，其中文字和图表所述的结果枚举算法的运行时间均仅仅包括CPU时间，而不包括IO时间。事实上，后者往往要远长于前者，且是由问题本身而非算法决定的——查询问题决定了结果大小，进而决定输出需要的IO时间。

6.2.1 TwigMiner系统中各核心子过程使用CPU时间的比例

根据3.2.1小节中的讨论，TwigMiner系统中采用的栈结构中同种类型且具有相同“最近栈内祖先”的栈组成树形结构，而不是线性表。这样栈结构构造算法往往需要寻找某个新取出的元素在栈结构内的兄弟，从而确定其连接位置。这可以减小结果枚举算法的复杂性，并提高其效率，但会导致栈结构构造算法的运行比采用线性表时更加费时。不过从本测试中可以看出，整个算法的瓶颈过程仍然不在栈结构构造算法上，getNext(Q)的执行所消耗的CPU时间远多于栈结构构造算法和结果枚举算法。

图6-2为本测试的结果。图中蓝色、红色、绿色分别表示getNext(Q)、栈结构构造算法和结果枚举算法消耗的CPU时间。其上的数字是以毫秒为单位的绝对时间值。注意到DBLP上的三个查询中栈结构构造时间相对较长，这是因为该数据集的树形结构较复杂，相应地在其上的查询模式也较复杂，这导致栈结构比较庞大和复杂。然而，栈结构构造算法运行时间所占比例最大时，也只达到了getNext(Q)过程的三分之一。

因此，有本测试可以看出：

1. TwigMiner系统中采用的栈结构及其构造算法没有对总体性能产生过大影响，是相对合理的。

2. 过程getNext(Q)的执行是费时的，可能成为进一步改进的入手点。

6.2.2 SHolisticTwigStack算法运行的两个阶段所用CPU时间的比例

由上文中的讨论可知，SHolisticTwigStack算法的运行分为两个阶段。在算法的第二阶段，实际上运行了原始的HolisticTwigStack算法，对文档元素队列中的剩余元素进行处理。

因此，若在第二阶段中的处理时间过长，甚至于占到了整个算法执行时间的多半部分，SHolisticTwigStack算法就开始向非流数据小枝查询算法“退化”了。因此在通常遇到的大多数数据集上，这两个阶段执行时间的比例是决定SHolisticTwigStack算法是否能有效地进行流处理的重要条件。这就是本测试的着眼点。

本测试在STwigMiner系统中进行，测试结果如表6-2所示。表中getNext1一栏显示了算法第一阶段的getNextFromStream(Q)和doSkip(Q)的总运行时间；getNext2一栏显示了第二阶段getNext(Q)的运行时间。后面4栏分别显示了第一阶段和第二阶段中栈结构构造算法和结果枚举算法的运行时间。所有时间的单位为毫秒。

由该结果可以看出，各个子过程的第二阶段执行时间与第一阶段执行时间相比，可以忽略不计。这体现出了SHolisticTwigStack算法在通常的流数据处理方面的有效性。

另外，有本结果亦可以看出，在流处理算法中与getNext(Q)具有同等地位的getNextFromStream(Q)和doSkip(Q)仍然是整个算法的瓶颈过程，其所消耗的时间远远长于栈结构构造算法和结果枚举算法。

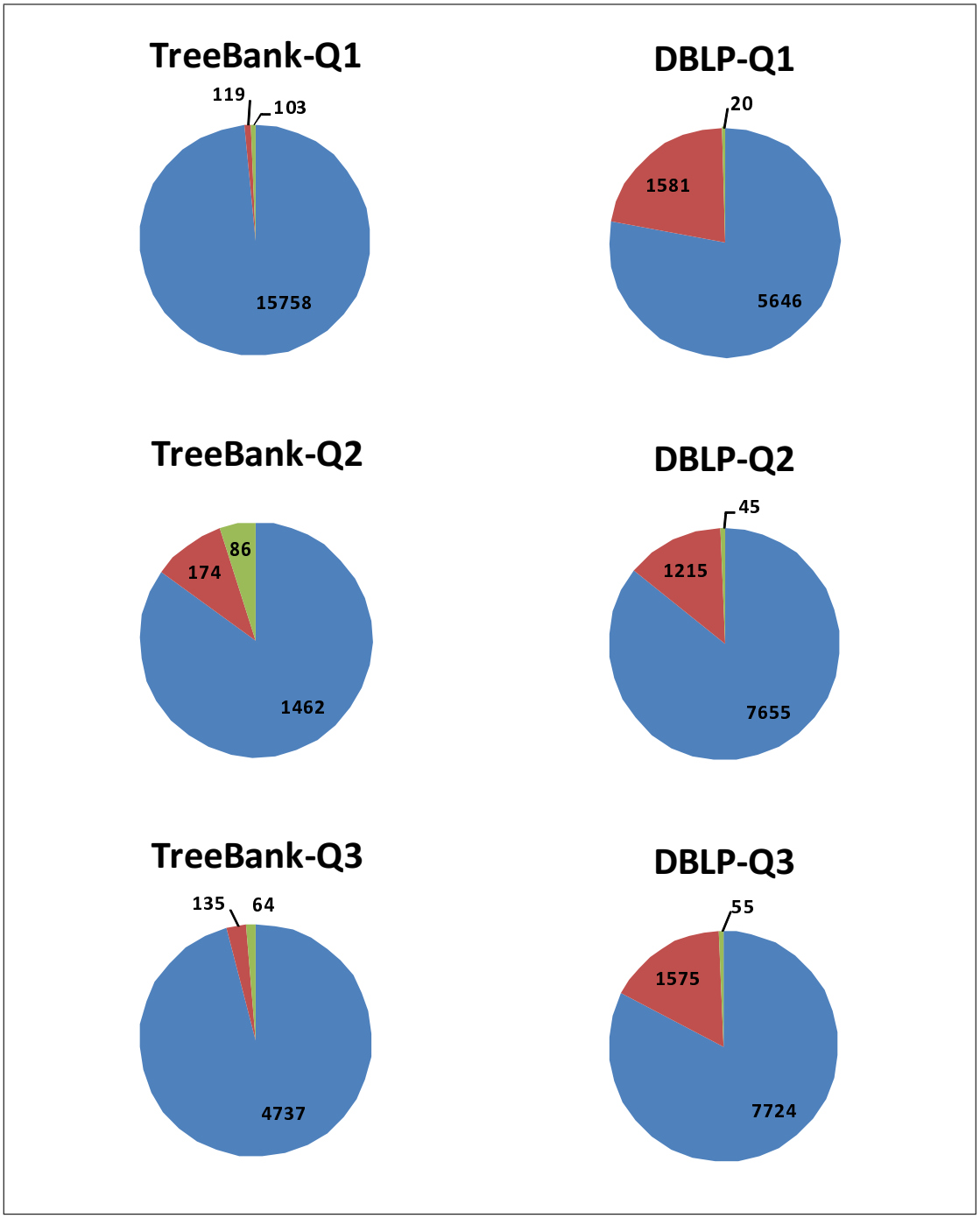

图 6-2 getNext(Q)、moveElementToStack(Q,e)和computeResult()在TwigMiner中使用CPU时间的比例。

表 6-2 STwigMiner系统中算法SHolisticTwigStack执行的第一、二阶段所需CPU时间的分配（表中时间值以毫秒为单位）

	getNext1	getNext2	stack1	stack2	result1	result2
TreeBank-Q1	12612	0	143	0	146	0
TreeBank-Q2	1662	1	242	0	152	0
TreeBank-Q3	5519	7	203	0	134	0
DBLP-Q1	6010	0	1751	0	29	0
DBLP-Q2	6633	0	1345	0	83	0
DBLP-Q3	7844	0	1762	0	120	0

6.2.3 STwigMiner系统运行时文档元素队列中缓存的元素数目

流数据小枝查询算法的一个前进方向就是用更少的缓存实现查询功能。本测试在一个均匀分布于整个算法执行过程的方法调用中取得文档元素队列的总大小，并每隔一定次数将改值输出到文件。而后使用得到的一系列数据，绘出从算法开始运行到结束，文档元素队列的总大小及其变化情况。结果如图6-3所示。

注意到两个测试数据集的大小均上百兆字节，其中各个类型标记的平均数量也至少在10^4级，然而在DBLP上进行查询时所有文档元素队列中元素的总数为个位数(这也与文档和模式结构简单有很大关系)。在TreeBank上进行查询时，队列中文档元素的数量最多也只有几百，而平均为几十。在图6-3中可以发现，在DBLP上的查询运行过程中有大段时间队列为空，事实上这是该数据集中查询模式的分布具有分段的连贯性导致的。

本测试的结果表现出使用SHolisticTwigStack算法处理超大数据集是比较有效的，可以显著减少内存空间用量。这样在数据集过大，导致无法全部读入内存时，采用SHolisticTwigStack算法很可能能够正常地执行查询。

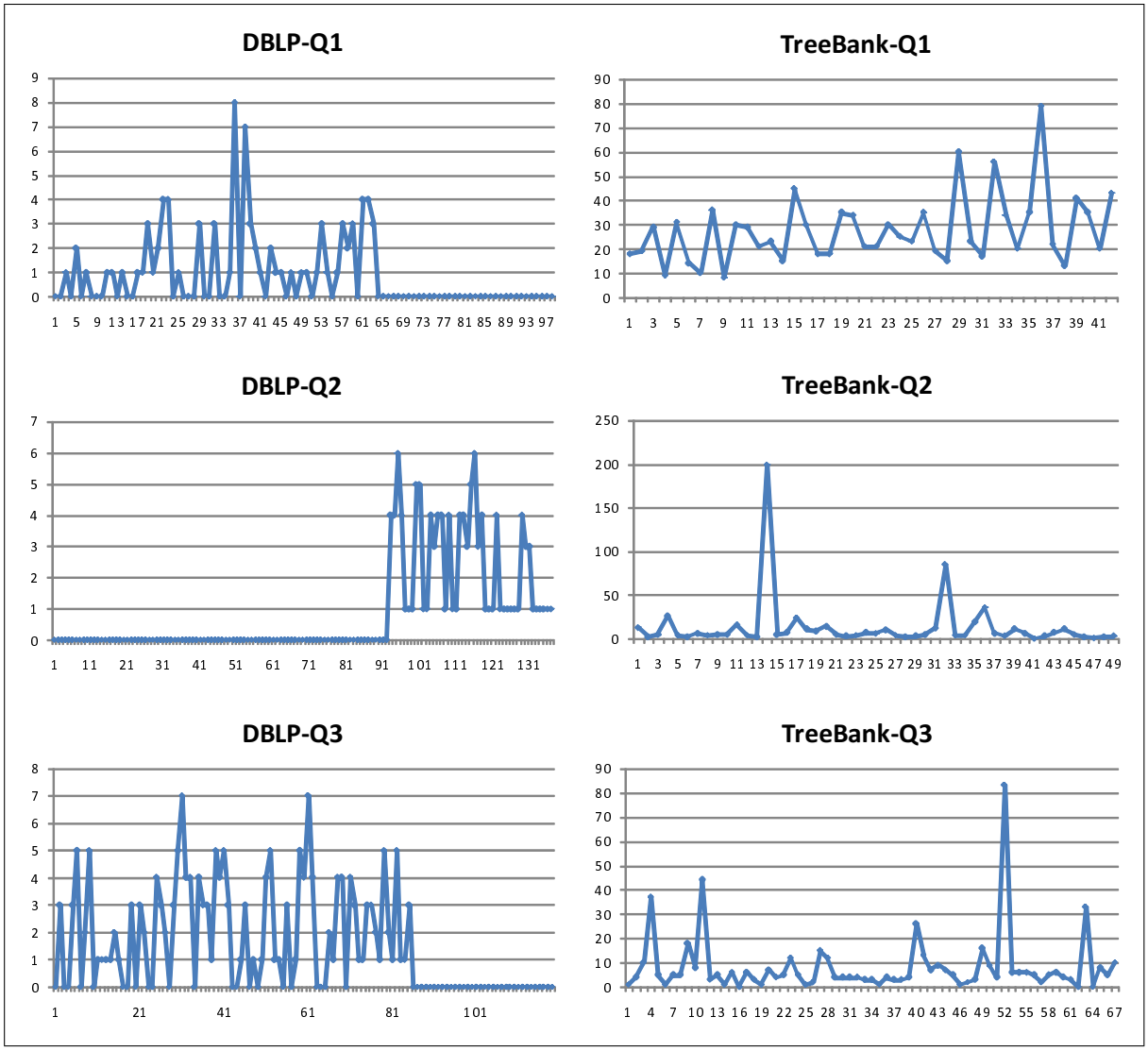

图 6-3　随着 STwigMiner 系统的运行，队列中缓存的元素数目的变化

结 论

本文在对XML数据的小枝查询问题及小枝查询算法进行调研的基础上，着重分析了HolisticTwigStack算法。并对该算法进行了改进，使其输出的小枝匹配结果不经过单独的排序操作，就能够按照“字典序”排列。在对改进的算法进行了正确性论证的基础上，实现了小枝查询系统TwigMiner。

本文研究了XML流数据查询问题，基于HolisticTwigStack算法提出了流数据小枝查询算法SHolisticTwigStack，对该算法的正确性进行了论证，并基于该算法实现了流数据小枝查询系统STwigMiner。

在实现两个小枝查询系统的基础上，使用它们进行了一些性能测试。性能测试显示，对原HolisticTwigStack算法改动后，改动部分没有成为瓶颈子过程，影响整个系统的执行效率。另外，SHolisticTwigStack算法可以比较有效地处理通常的流数据，较显著地减少需要缓存的元素数量。

以下问题有待进一步研究：首先，性能测试显示，HolisticTwigStack算法中的过程getNext(Q)，以及SHolisticTwigStack算法中具有同等地位的getNextFromStream(Q)和doSkip(Q)在整个算法运行过程中耗用了大量的CPU时间，它们成为了整个算法的瓶颈子过程。若能提高这些递归过程的执行效率，则非常有助于提高HolisticTwigStack算法的性能。其次，在SHolisticTwigStack算法中，由于存在部分getNextFromStream(Q)的调用会返回不确定状态，无法指示下一个可以取出的元素的位置，降低了效率。如果可以避免不确定状态的返回，将会在一定程度上使流处理算法的运行更加流畅。维护一些数据以帮助getNext(Q)和getNextFromStream(Q)进行元素关系的判断可能有助于解决这两个问题。然而，要使维护的数据简单且易于更新，则是难点。

致 谢

本次的毕业设计和本论文是在许多人的指导、帮助和鼓励下得以完成的。在此，本文作者作为受益人，向所有提供过帮助和提出过意见、建议者表示衷心的感谢。

高红雨老师全程对本毕业设计进行了各个方面的指导。首先，在战略上，高老师多次提供了关于调整工作重心的有益建议，使我避免了在最主要工作完成之前，在一些扩展性问题上耗费过多的时间和精力。其次，在战术上，高老师提供了系统功能和性能测试方面的一些有效方法和工具，使得相关的测试工作可以顺利进行。在治学态度上，他注重身教——耐心细致地与我进行算法地讨论，在思考的基础上为我指出某种方案的不足之处；也注重言传——严谨地为我找出在设计报告和论文中的各种问题——从语言上的到结构上的，再到思想上的。虽然高老师并不完全了解本工作的细节，但在与他的不断交流中，我在各个方面收益匪浅。

本课题组的王磊、高增琦、邓自强、张晓博学长提供了重要帮助。王磊师兄几次与我讨论算法上的细节问题，在讨论中某些问题的实质变得更加清晰。另外他还为我提供了多篇有用的算法文献。高增琦师兄在如何减少系统内存使用和如何进行性能测试方面传授过他自身的经验，提出过宝贵建议。邓自强师兄在本工作开始时提供了关于学习编译器自动生成工具的建议，并将他自己实现的 $Twig^2Stack$ 算法的代码与我共享，使我了解到这种类型工作的大体框架，有一个好的开端。张晓博师兄曾耐心为我讲解课题组在 XQuery 语言的编译方面实现的一种中间语言，以及在其中小枝模式的表示。

最后，感谢父母和朋友在精神上提供的支持和鼓励，这是我在遇到困难时继续坚持探索算法和调试程序的一大动力源泉。

参 考 文 献

[1] Zhewei Jiang, Cheng Luo, Wen-Chi Hou, et al. Efficient Processing of XML Twig Pattern: A Novel One-Phase Holistic Solution. In: DEXA 2007, LNCS 4653, 2007. 87–97. (2007)

[2] Zhewei Jiang, Cheng Luo, Wen-Chi Hou, et al. Efficient XML Tree Pattern Query Evaluation using a Novel One-Phase Holistic Twig Join Scheme. (2008) http://www.cs.siu.edu/~dche/pdf_files/ijwis09.pdf.

[3] Bruno N, Koudas N, Srivastava D. Holistic Twig Joins: Optimal XML Pattern Matching. In: SIGMOD Conference, 310–321. (2002)

[4] Lu Qin, Jeffrey Xu Yu, Bolin Ding. TwigList: Make Twig Pattern Matching Fast. IN: LNCS 4443, 2010. 850-862. (2007)

[5] Chen, S., Li, H., Tatemura, J., et al. Twig^{2}Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents. In: SIGMOD Conference, pp. 283–294 (2006)

[6] Anders B, et al.XML Path Language (XPath) 2.0. (2007) http://www.w3.org/TR/2007/REC-xpath20-20070123/

[7] Scott B, et al. XQuery 1.0: An XML Query Language. (2007) http://www.w3.org/TR/2007/REC-xquery-20070123/

[8] Tim Bray, Jean Paoli, C.M.Sperberg. Extensible Markup Language (XML) 1.0. (1998) http://www.w3.org/TR/1998/REC-xml-19980210/

[9] Feng Peng, Sudarshan C. XPath queries on streaming data. In: SIGMOD Conference, 431-442. (2003)

[10] 杨卫东, 施伯乐. XML 流管理研究综述. 计算机研究与发展, 46(10). 1721-1728. (2009)

[11] http://www.xmlmind.com/qizx/

[12] U. of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/.