摘要

数学公式图像识别是一个具有相当难度的前沿课题，许多处理方法都还在探索和尝试中。为了不断改进各种处理方法，提高公式识别系统的整体性能，需要对公式识别的各个阶段以及整体进行自动的测试和性能评估。然而有关性能评估的研究相当少，在本文开始研究之前，尚没有对大规模图像进行性能评估的实验结果，这严重阻碍了数学公式识别研究的进一步发展。

本文根据数学公式本身的二维特性，首次提出采用带边属性的三叉树结构来表示数学公式的识别结果，并首次提出两种基于树匹配算法（动态规划算法和BUTD算法）的公式识别自动性能评估方法。其中基于BUTD算法的性能评估方法是目前唯一能够发现公式符号切割、识别以及公式分析中产生的任意错误的评估方法。

关键词　数学公式图像识别 数学公式分析 自动性能评估

Abstract

The recognition of mathematical expression image is a difficult subject. Many recognition methods are still tentative. In order to improve these methods and enhance the system performance, it is necessary to evaluate the performance of the entire recognition system, but also the performance of each phase. However, the research on performance evaluation of mathematical expression recognition is very scarce. Before our work, there is no report about performance evaluation in a large variety of images, which has prevented the advance of mathematical expression recognition.

Because of the two-dimensional formatting of mathematical expressions, a triplet-tree with marked edge is presented to represent the analysis result of mathematical expressions. Two automatic performance evaluation methods on the basis of tree matching are proposed, which are based on dynamic programming algorithm and BUTD (Bottom-Up and Top-Down) algorithm, respectively. BUTD is the only known method that can find any errors of symbol segmentation, symbol recognition and expression analysis.

Key Words: Mathematical Expression Image Recognition, Mathematical Expression Analysis, Automatic Performance Evaluation

第一章 引言

1.1 数学公式图像识别的意义

数学是“唯一的国际科学语言”，不受种族、国家、文化和任何方言的限制，不同国籍的科学人员都可以通过这种共通的符号沟通。这是因为“数学提供了一种有力的、简洁的、准确无误的交流信息的手段”，正如《数学算数》一书中所说，“所有数学有用性的理由都根源于数学可用作交流信息的有利手段这个事实”。数学大师华罗庚甚至认为可以使用数形关系图作为与外星人交流的媒介。这是因为在宇宙中，计数的规则应该类似，自然图形所具备的数形关系也是普遍的。魏晋时期的大数学家刘徽不使用任何数学符号和文字，不进行任何运算，只用涂有颜色的“青朱出入图”就把勾股定理清晰地呈现在人们面前。无怪乎华罗庚会如此感慨。

数学在“宇宙之大、粒子之微、火箭之速、化工之巧、地球之变、生物之谜、日用之繁”等各方面都有重要贡献，而数学公式是数学语言的最主要表现形式，所以科技文献中包含大量数学公式。这些文献被扫描仪扫描，作为图像数据保存入计算机，识别其中的数学公式图像，就是为了获取公式中包含的信息，使信息的储存和使用变得容易，实现信息更好的交流和共享。例如，研究人员要求能够轻易重用数学公式；数字图书馆要求能够以便于编辑、便于查找的方式储存数学公式；远程教育要求能够以有效的方式在网络上传输数学公式。现有的文档图像处理系统能够高速、准确的识别文档图像中的普通文本，但是不能正确处理其中的数学公式。大多数情况下，数学公式的识别结果是一些毫无意义的符号，如图1-1。

(a) 公式扫描图像　　(b) 公式识别结果

图1-1：一般文档图像处理系统识别数学公式的结果

不能够正确识别数学公式图像会带来以下问题：

第一，验证与重用。如果研究人员想要验证或重用文档中的数学公式，就只能使用专用的数学计算软件（例如Matlab 或者Maple），或者数学排版软件（例如TEX或者LATEX），按照各自的语法要求重新输入。这个输入过程既漫长又令人生厌，而且还要冒着引入新的

错误的风险。即便是使用新的可视化的公式输入软件（例如Scientific Workplace 或者Science Word），输入速度也不可能加快很多。

第二，编辑与查找。以图像形式保存的数学公式，既无法编辑修改，也无法实现公式的查找和检索。

第三，存储和传输。以图像形式保存的数学公式占据的空间远远大于以其他可编辑格式保存占据的空间。这样既耗费了存储空间，又加大了传输量。

为什么目前的文档图像处理系统不能识别数学公式呢？除了数学公式与普通文本反映的内容不同以外，排版结构上也存在很大差别。表1-1列举了两者在结构上的一些区别。

	数学公式	普通文本
符号间位置关系	二维（2-D）	一维（1-D）
字符集	英文字符、希腊字符、数字、运算符等	文本所属语言的字符集
字号	变化频繁 （上/下标字号小）	同一段落内基本相同
风格	斜体居多	正体居多
基线 (BASELINE)	不对齐 （符号间二维关系）	行内对齐
符号间距	不均匀	均匀，字间距，行间距
符号出现频率	数学运算符号出现频繁	遵循一般文本的统计规律

表1-1：数学公式与普通文本的差异

因此识别数学公式图像有着重要的现实意义。

1.2 数学公式图像识别的基本步骤

自动数学公式图像识别的目的就是从包含有数学公式的扫描图像开始，定位其中的公式，识别公式中的符号，然后根据符号的内容和相互间位置关系，对公式进行分析，并将分析结果按照某种格式保存，最终实现复用。如图 1-2 所示，数学公式识别的基本步骤可以分为数学公式定位、数学公式识别、数学公式分析以及数学公式分析结果的格式转换输出。

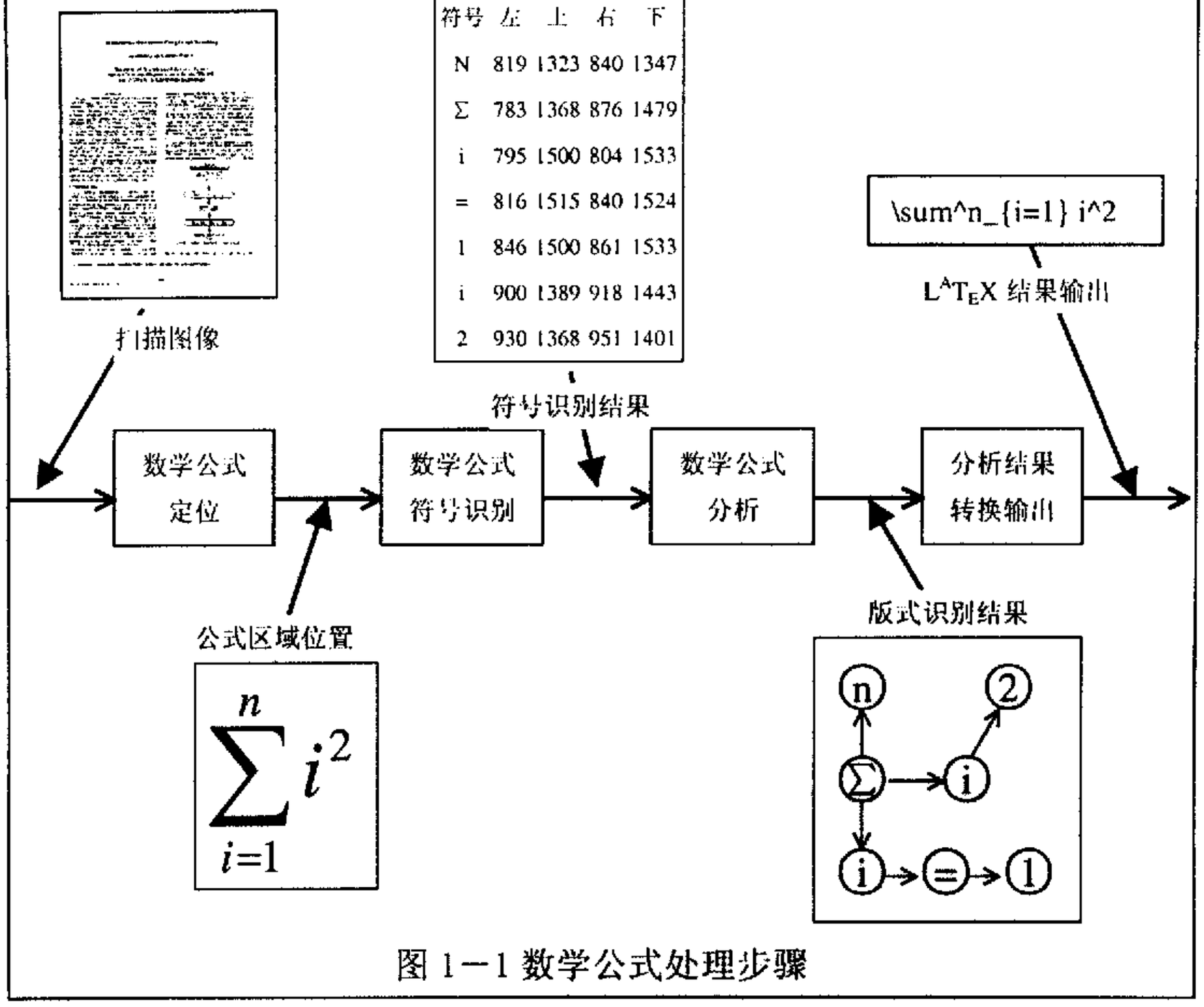


图 1—1 数学公式处理步骤

无论是数学公式定位，数学公式符号识别还是数学公式分析，都需要对其性能进行有效的评估，以便准确地定位系统中出现的各种错误。

1.3 数学公式图像识别自动性能评估的必要性

现实生活中，我们到处都能见到形形色色的“排行榜”，这些“排行榜”实际上就是对事物某些方面的性能或功能的一个评价结果，不可否认“排行榜”对事物发展有着或多或少的影响。同样，对于一个软件系统而言，在设计开发的整个过程中，在每个阶段都需要对各个模块和各个功能进行测试和性能评估。有时候测试和性能评价工具的欠缺甚至成为系统性能提高和改善的制约因素。因为通常设计开发者在对其中的某一个模块进行了一个很小的改动，往往就会影响系统性能的方方面面。他们需要花费大量的时间来分析被修改以后的系统的性能，而且有时即使花费了很多时间对系统性能的评估仍然不准确。因为这些分析和评估只是在实验的基础上，对一个较小的样本集进行，而不可能对大的样本集进行，因此这些分析的结果一般只有理论上的价值，而没有实际的价值。

数学公式图像识别是一个具有相当难度的前沿课题，许多处理方法都还在探索和尝试中。无论是公式的定位，公式符号的识别，还是公式的分析都需要进行测试和性能评估。

有关数学公式识别系统性能的评测问题至今尚无公认的标准，然而性能评估是非常重要的。为了比较处理方法的优劣，就必须在大规模集合上进行测试。如果测试过程不能自动进行，测试规模就只能很小。

数学公式行的二维结构决定了公式识别不仅包含符号识别，更重要的是对其结构的分析，因此数学公式分析的性能评估显得尤为重要。同时，也正因为二维结构，使数学公式分析的性能评估变得更困难，不能像符号识别那样仅仅用符号的识别率就能衡量其性能。

虽然已经有很多数学公式分析的方法，然而这些方法分析得到的结果究竟如何，谁的方法效果好，准确性高，目前尚没有一个公认的评测标准。分析结果的好坏只能由人工评判，所以目前几乎所有的方法都只是在几十甚至十几个公式集上进行测试，这严重阻碍了数学公式识别研究的进一步发展。这就迫切需要开发出一个工具，使其对数学公式分析技术进行自动的性能评估。只有这样，对它们的性能评估才可能在大样本集上进行，得出的评估数值才真正具有实际意义。因此对这方面的研究也变得非常有价值了。

1.4 本文的主要内容及结构

数学公式图像识别的各个阶段都需要进行有效的测试和性能评估。目前出现的对手写体数学公式识别的性能评估都存在有片面性，性能指标太过笼统以及测试规模小等缺点，并不通用。而对于印刷体数学公式识别的性能评估，只有 M.Okamoto 等人在[11]中提出了一种基于 MathML 格式比较的方法，但是他们的方法只能判断出分析结果是否正确，并没有指出错误的原因及修改的方法。由于公式本身的二维特点决定了公式关系用树表示是非常恰当的，所以一般识别结果都采用有层次的树结构表示，因此本文提出了基于树匹配的公式识别自动性能评估的方法。该方法不仅能够找出符号识别和分析中存在的错误，还能指出错误的类型和原因，并以此指导识别方法的修改。作为核心，本文详细介绍了两种树匹配算法：动态规划算法和 BUTD（Bottom-Up and Top-Down，因为每一个树节点都采用先自底向上然后自定向下的比较过程）算法。其中 BUTD 算法不仅能够评估公式分析的性能，还能评估符号切割和识别的性能。最后我们给出了 BUTD 算法的实验结果。

第一章：概述了数学公式图像处理的意义和基本步骤，以及数学公式识别的自动性能评估在数学公式处理中的地位。

第二章：给出性能评估的概念和系统性能评估的一般模型。概述公式识别的结果并提出识别结果的三叉树表示，同时总结了当前公式识别性能评估的现状。就此本文提出一个公式识别的自动性能评估模型和两种基于树匹配的性能评估算法：动态规划算法与 BUTD

算法。

第三章：给出基于动态规划的性能评估算法。由于我们使用带边属性的三叉树与一般带标记的树结构有所不同，而且节点匹配的标准也不一样，因此本文将传统的动态规划匹配算法进行了扩充和改进，即把传统的节点更改，节点插入和节点删除三种基本编辑操作扩充为节点完全替代，节点内容修改，节点类型修改，节点插入，节点删除，边属性修改六种基本编辑操作，以满足公式识别性能评估的特殊需要。对该算法进行了详细的描述和分析，并给出一个实例。

第四章：给出基于 BUTD 的性能评估算法。动态规划算法由于受到节点的兄弟次序不变和祖孙次序不变的约束，在记录错误的时候存在冗余，隐藏了某些错误的根本原因，因此本文根据实际需要，将动态规划中的编辑操作和错误类型进一步扩充，使错误类型能够直接反映出错误的本质原因，并提出新的匹配算法（BUTD）。详细介绍了该算法的流程，同时给出相应的实例。最后从合理性和效率两个角度比较了 BUTD 算法和动态规划算法，证明 BUTD 算法比动态规划算法更具优势。

第五章：给出了基于 BUTD 算法的自动性能评估的实验结果，并对结果进行了分析和总结。

第二章 数学公式图像识别的自动性能评估模型

2.1 性能评估概述[14]

所谓性能评估是指根据一定的基准数据来进行的测试比较。性能评估在比较几个相似的复杂系统时非常重要。此外，当一个系统被修改或者被更新版本时，也需要对修改后的系统的性能做出评估。在这种情况下，性能评估不仅要对整个系统进行，而且需要对系统的各个模块来进行，这样才有机会发现系统中的薄弱环节，并且给出一个直接面向系统目标的改进。

2.1.1 性能评估的一般目标

性能评估基本上追求以下三个目标：

- 提高系统性能并对系统进行质量控制；
- 比较多个复杂系统；
- 评估系统的性能，即定义一个系统性能的绝对值。

性能评估的第一个目标是提高或者维护系统的性能。

性能评估的第二个目标是比较多个不同的系统。这在要从几个系统中选择一个最好的系统去完成某项任务时，是非常关键的。

性能评估的第三个目标是进行性能评估，给出评价系统性能的一个绝对数值，这个数值可以说明系统的当前发展水平。它描述了在不同特征的输入数据情况下，实际系统和理想系统之间的性能差距。

这些情况下，性能评估的执行都是基于系统比较。这种比较可以是两个系统之间的比较，也可以是多个系统之间的比较。如果要评价系统性能的绝对值，可以用该系统与所谓的理想系统进行比较。而所有这些比较的执行都使用参考数据或者基准（Ground Truth 简称 GT）来作为比较的依据。

下面，我们给出一个基于上述目标的关于性能评估的更正规的描述。通常情况下，一个性能评估包含输入数据，一个或多个系统，一个评估函数和一个评价函数。

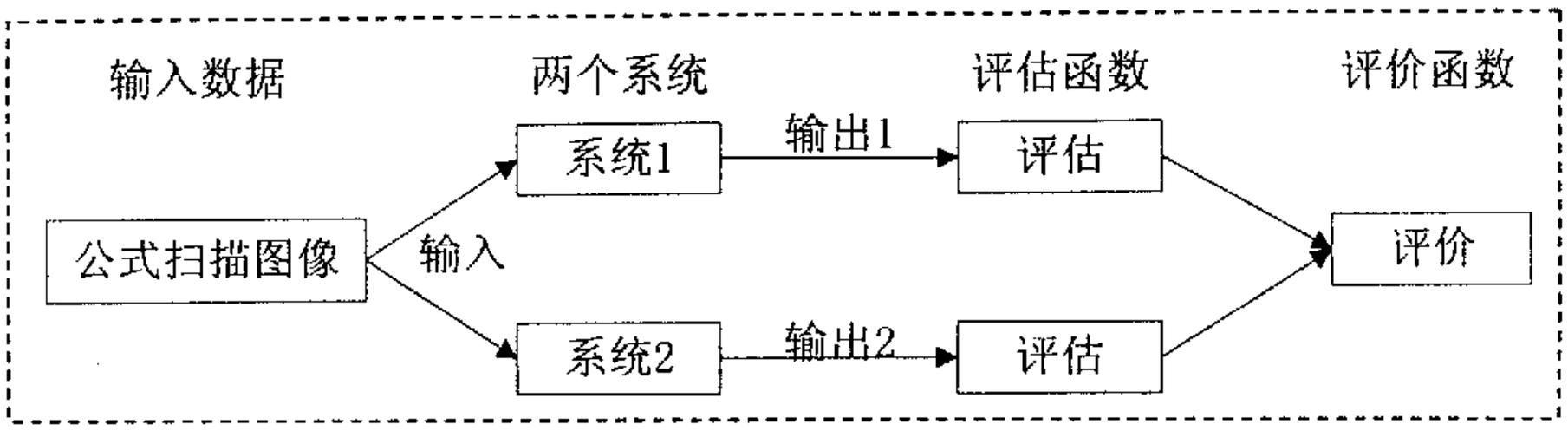


图 2-1：两个系统的性能评估模型

图 2-2：一个系统的性能评估模型

图 2-1 和图 2-2 分别表示对两个系统和一个系统进行比较评估的模型，其中各模块说明如下：

- 输入数据（Input Data）的选择对于性能评估来说是非常重要的。输入数据对于系统处理的一般任务应该具有代表性。有的系统本身对它所要完成的任务描述就不清楚，但是对于一个确定的算法不会出现这种情况。然而这里系统的概念是包含理想系统（见 2.1.2 节）的，这个理想系统是存在于设计者的脑海里的。系统所给出的描述可能是系统的理想输出或者是一种基准（GT）。从实际的角度考虑，这种方法看上去很合理，因为性能评估通常要求系统的输出满足或者达到一定的要求。
- 评估函数（Assessment Function）是性能评估的另一个重要部分。评估函数是根据系统的输出而赋的一个值，这个值可以是基于系统的输出与对应输入的比较所得出的值，它定量地反映了系统输出的质量。显然，不同的评估函数对于系统相同的输出会有不同的结果。而理想的评估函数总是能产生一个最优的值。评估函数对于某些系统是比较确定的，但是对于数学公式处理系统或公式分析模块来说就不是很确定了，因此，评估函数的选择是一个很重要的因素。
- 评价函数（Evaluation Function）用于比较几个不同系统的评估函数的值。这个函数给出的结果是几个系统中哪个好哪个坏，或者哪个更好一些。评价结果应当同时指出对输入数据的分类情况。因此，输出结果就不应当是：“系统 A 比系统 B 好”，而是：“在输入有字符粘连（Touching）的情况下，系统 A 比系统 B 好”。或者输出结果中可以包含假设的测试过程。

如果输入数据、系统、评估函数和评价函数都确定了，则性能评估也就确定了。

2.1.2 理想系统与理想评估

在此，我们给出理想系统与理想评估的概念。

我们所说的性能评估都隐含着：每个实际系统 *Sys* 都对应一个理想系统 *IdSys*。这个理想系统以最优的方式解决了实际系统所要解决的问题。由于理想系统是实际系统所希望达到的目标，实际系统就会试图将它的输出向基准靠近。对应于理想系统，也有一个理想的评估。理想评估总是以期望的方式来评估系统的输出。如果系统的输出是理想输出，则评估结果就应该是一个最优值，否则评估结果就应该差一些。由于实际应用中也许只提供了有限的信息，使实际的评估含有不精确的因素，因此实际的评估只是理想评估值的一个近似。理想评估是一个没有错误的评估，不同的目标应该由不同的评估来完成。

因此，理想系统和理想评估相对于实际系统和实际评估是一个没有错误情况的系统模型和评估模型。我们的目标是要最优地近似这些理想模型。

2.1.3 测试基准（Ground Truth）

这里所提到的测试基准就是前面所介绍的基准（Ground Truth 简称 GT）。在数学公式识别系统中，基准代表系统的最优输出。在性能评估中我们也使用这个词来表示评估所依赖的标准，为了定义基准，我们必须考虑这个最优是相对于一个什么参考标准而言的，即：‘相对于什么最优？’。

例如在数学公式识别系统中，基准主要是为系统的输出定义的。对于数学公式处理系统的输出，基准就是人对于数学公式图像的识别结果。因此，在这种情况下，是人决定什么是最优。例如，图 2-3 所示，也许有人认为‘t’，‘a’，‘c’是同一行的关系，而有人则认为‘a^c’是‘t’的上标，‘c’是‘a’的上标，遇到这种情况，就要考虑是‘相对谁最优？’。当我们要确定一个系统某些内部模块输出的基准，这个最优输出就依赖于该模块之后的其它模块，所以基准可以在系统任何两个模块的接口之间定义。因此，在确定了输入数据，评估函数和评价函数之后，性能评估也可以在系统的任意两个模块之间进行。如图 2-4 所示。

总之，我们需要一个评估函数，这个评估函数可以由人来确定，也可以由系统的后续模块确定。

不失一般性，我们可以认为评估函数是一个计算代价的函数。这个代价就是将系统的输出修改成为最优输出所需要花费的代价。这就意味着评估函数的值总是大于或等于零。

而且系统的输出结果越好，代价值就越小。对于基准来说，该评价函数的值应该等于零。这是一个一般意义上的定义，因为其它函数都可以很容易地转化为这种代价函数的形式。

taC

图 2-3

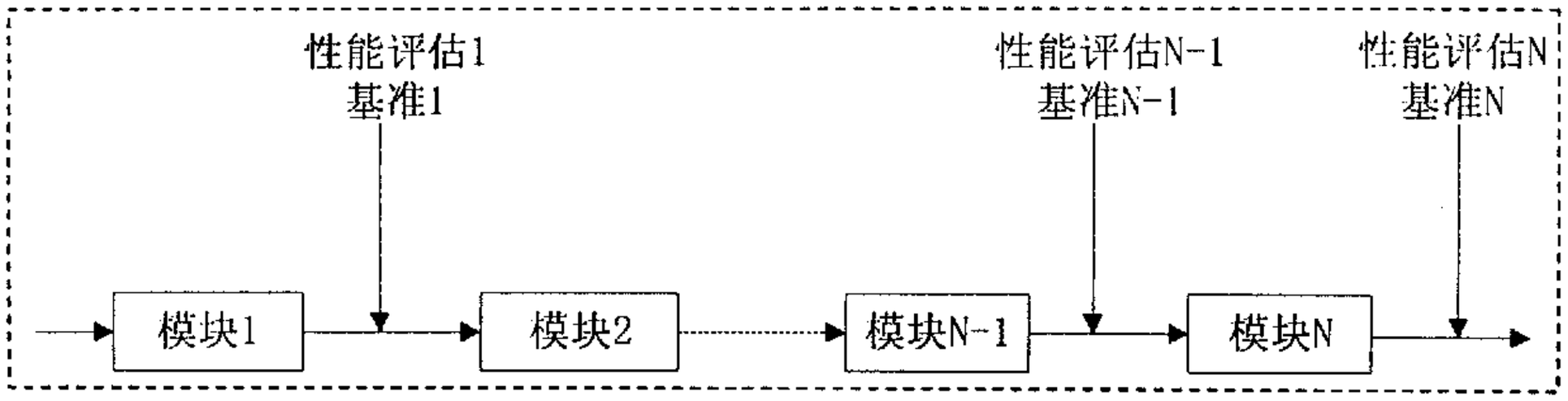


图 2-4：基准可以在系统的各个地方定义

定义 2.1：假设给定系统 *Sys* 和它的输入数据集合 *In*。gt_{in} 是输入数据 *in*（$in \in In$）在基准中的对应结果，系统 *Sys* 对应的理想系统是 *IdSys*，理想的评估函数为 *IdAss*。则：

$$IdSys(in) = gt_{in} \text{ 且 } IdAss(gt_{in}) = 0$$

注：一个输入数据可能对应多个基准。因为可能有几种结果都被认为是最优的。但是通常出现这种情况时，我们就给出一些特殊规定或说明，从而选定这几种结果中的一种作为基准。

2.2 数学公式图像识别结果及其自动性能评估现状

2.2.1 数学公式图像识别的结果

图像中的数学公式经过定位，符号识别和分析之后以某种格式保存即得到公式的识别结果，因此数学公式识别的结果实际上等同于公式分析之后所得的结果。

数学公式分析就是在正确识别公式的每个符号，并且得到包含该符号的最小外接矩形的基础上，分析符号之间的关系并进行组合，理解公式的含义，实现公式版式结构的恢复，甚至表达式的自动计算，最后将分析结果用某种数据结构（一般是图或者树）表示出来。

根据最终分析目标的不同，数学公式分析分为两种层次[13]：

(1)版式识别(Layout recognition)：版式识别要求分析结果足够恢复该公式的排版样式，但是不需要理解整个公式的数学含义。版式识别结果可以使用 $LaTeX$ 等排版语言输出。文献电子化过程中，为了保证版面的一致，就需要识别数学公式的版式。

(2) 语义识别（Semantic recognition）：语义识别要求确切理解整个数学公式的数学含义，即明确表达式的计算顺序和函数的作用域。语义识别结果可以使用 Matlab 等数学计算软件包的语言输出。实现数学公式的自动计算就需要识别公式的语义。

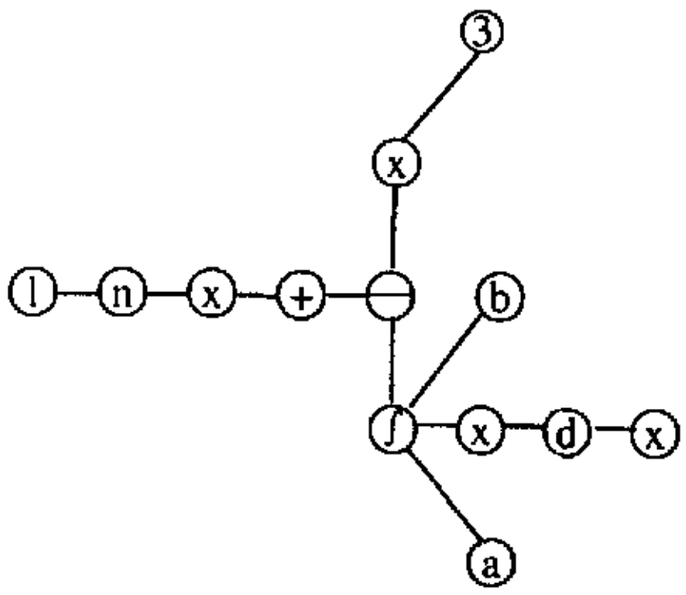
同一公式的版式识别和语义识别的结果并不相同，我们以公式（2-1）为例给出版式识别结果与语义识别结果。图 2-5 是利用树结构表示的版式识别结果。图 2-6 是利用树结构表示的语义识别结果。

由于语义识别的难度远大于版式识别的难度，因此目前绝大部分数学公式分析都是以版式识别为目的。版式识别的分析方法大致可以分为两种：基于结构的分析方法和基于文法的分析方法。

基于结构的分析方法就是直接根据各个符号的内容、大小、相对位置以及符号间的空白等结构信息判断出相邻符号的关系，生成符号组，合并子表达式，从而实现数学公式分析的方法。这种分析方法既可以把握公式的整体结构，也可以把握局部符号之间的关系，能够处理格式复杂，符号数目多的公式。结构分析可以较好的识别公式版式，但语义识别的能力较差。

基于文法的分析方法是通过定义文法来分析数学公式的含义。这种方法具有很强的语义识别能力，但是只能处理格式简单，类型单一，符号数目少的数学公式，根本不能满足实际需要。

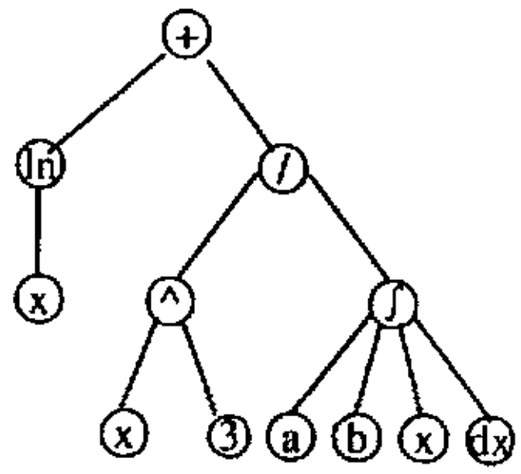
$$\ln x + \frac{x^3}{\int_a^b x dx} \qquad (2\text{-}1)$$



L^AT_EX 结果输出：

\[\ln x + \frac{x^3}{\int_a^b x dx}\]

图 2-5.版面识别结果



MatLab 结果输出：

ln(x)+(x^3)/quad('x', a, b)

图 2-6.语义识别结果

得到公式的分析结果之后，我们就可以视其为基准，基于它来评估整个识别系统的性

能。就目前来说，基于结构的分析方法居多，所以我们后面定义的 11 种公式关系只反映了结构信息，如果定义足够的关系，那么语义信息同样也是可以评估的。

2.2.2 数学公式图像识别结果的表示

L^AT_EX 是一种科技排版软件，常用于数学文献的排版，而且 L^AT_EX 格式的源文件是人可以阅读并理解的文本格式。因此可以将数学公式识别结果保存为 L^AT_EX 格式。以下是数学公式及相应的 L^AT_EX 表达式：

$$\int_0^{\infty} f(x)dx \approx \sum_{i=0}^{n} w_i c^{x_i} f(x_i) \qquad (2\text{-}1)$$

```
\[
\int_{0}^{\infty{}}f\left(x\right)dx
\approx{}\sum^{n}_{i=1}w_{i}c^{x_{i}}
f\left(x_i\right)
\]
```

无论用什么样的分析方法，公式本身的二维特点决定了公式关系用树表示是非常恰当的，所以一般分析结果都采用有层次的树结构表示，另外我们很容易就能将 L^AT_EX 格式转化为树结构。因此，我们提出用一种带边属性的三叉树来表示数学公式的分析结果，从这种三叉树结构很容易就可以得到公式的版式结构，给人一种直观的感觉。

根据人书写、阅读公式的习惯，并参考 L^AT_EX 语言表示数学公式的方法，我们将数学公式划分成 11 种类型，即：多行表达式（Multiline Expression），分式表达式（Fraction Expression），根式表达式（Radical Expression），定界表达式（Delimiter Expression），矩阵表达式（Matrix Expression），组表达式（Group Expression），堆叠表达式（Stack Expression），帽子表达式（Hat Expression），角标表达式（Script Expression），普通表达式（Common Expression）和基元表达式（Primary Expression）。图 2-7 至 2-17 给出了各种表达式的示例。

$$\begin{aligned}(x+y)(x-y) &= x^2 - xy + xy - y^2 \\ &= x^2 - y^2 \\ &\amalg\end{aligned}$$

图 2-7：多行表达式示例

$$\frac{a^2 - b^2}{a + b}$$

图 2-8：分式表达式示例

$$\sqrt{x^2 + y^2 + 2xy}$$

图 2-9：根式表达式示例

$$(8^{2/3} + 1^{2/3})$$

图 2-10：定界表达式示例

$$\sum_{i=1}^{n}$$

图 2-11：组表达式示例

$$\begin{vmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{vmatrix}$$

图 2-12：矩阵表达式示例

$$\underline{\underline{\mathbf{def}}}$$

图 2-13：堆叠表达式示例

$$a + \overbrace{b + \cdots + y}^{} + z$$

图 2-14：帽子表达式示例

$$x^{y^2}$$

图 2-15：角标表达式示例

$$z = 2a + 3y$$

图 2—16：普通表达式示例

$$x$$

图 2-17：基元表达式示例

任意类型的公式都可以用带边属性的三叉树表示，自然整个数学公式也可以表示成一棵三叉树。树中每个节点代表一个数学符号或一个组元（如 sin, log, min, lim 等），节点中给出了相应符号的识别结果，位置信息以及公式类型，并且每个节点至多有三个非空子节点。每条边都有一个属性值，指出所连接的两个节点之间的关系，包括：同行（NEXT），上标（SUPERSCRIPT），下标(SUBSCRIPT)，组表达式上标（LOP UPSCRIPT），组表达式下标(LOP DOWNSCRIPT)，分子（NUMERATOR），分母（DENOMINATOR），帽子（HAT），堆叠（STACK），根式开方数(RADICALEXP)，根式被开方数（RADICALMAIN），矩阵起始元素(MATRIXBEGIN)，矩阵结束元素(MATRIXEND)。对于多行表达式，我们把每一行表达式看成一棵独立的三叉树。这种结构类似于公式的版式结构，但是比版式结构包含更多的语义信息。如图 2-18

所示为公式$\sum_{i=1}^{n} a_i + \frac{bx^3}{3}$的三叉树结构。

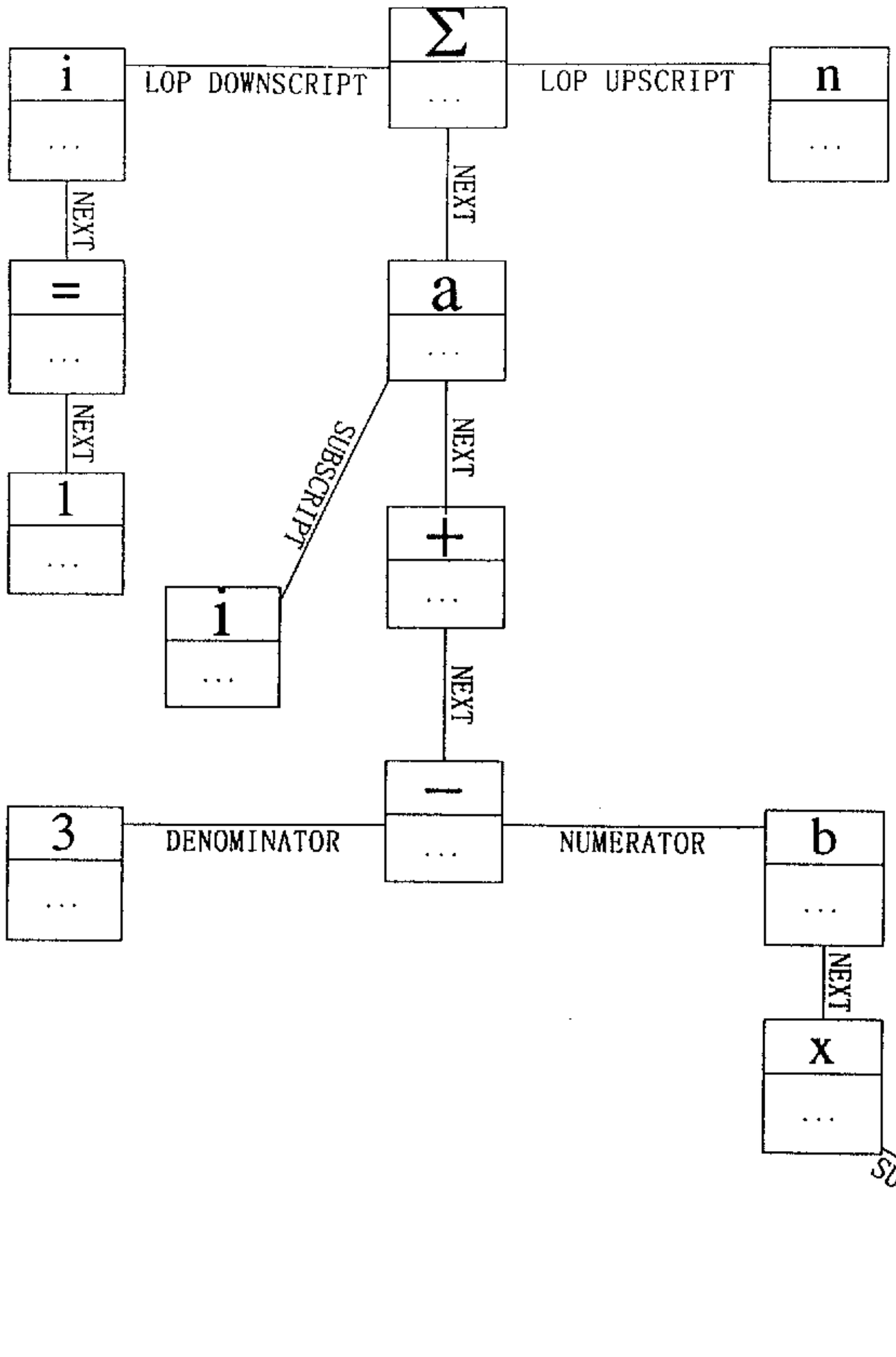


图 2-18：公式$\sum_{i=1}^{n} a_i + \frac{bx^3}{3}$的三叉树结构示例

用带有边属性的三叉树表示数学公式的结构，会使某些公式类型“消失”，如定界表达式（Delimiter Expression），堆叠表达式（Stack Expression），帽子表达式（Cap Expression）以及角标表达式（Script Expression），这些表达式类型信息将保存在相应子表达式的边属性中，而不必在节点中记录。这种节点平衡的三叉树结构极大的方便了后续的基于树匹配算法的性能评估工作。

2.2.3 自动性能评估的现状

1968 年，Anderson 在他的博士论文[1]中最早提出了数学公式的处理问题，在随后的三十多年中，有关数学公式处理的各个方面都有或多或少的研究。由于早期的研究人员主要致力于理论方面的研究，而没有实验结果，因此直到后来才提出如何评价一个数学公式处理系统的性能问题。在 2000 年之前的文献中所提到的评估方法都是针对手写体数学公式的，大致可以归为三类：

（1） 注重于公式的结构，通过测试只给出两种结果，即正确或不正确。

（2） 只考虑符号的识别率。

（3） 只对少数非常典型的公式进行评测，这些公式由少数固定人员书写，而且书写整洁。

这些手写体数学公式的评估方法都具有片面性，性能指标太过笼统以及测试规模小等缺点，并不通用。Kam-Fai Chan 等在[12]中也提到了在线手写体数学公式处理系统的性能评估问题，他们综合了前人的方法，将错误类型分成两种：公式中符号的识别错误和公式本身结构的错误，给出了两种错误率的计算公式，即错误个数除以总数，但是也仅仅是如此而已，并没有说明如何找到那些错误，也未提出有效的评测步骤和过程。

对于印刷体数学公式识别的性能评估，至今只有 M.Okamoto 等人在[11]中提出了一种基于 MathML 格式比较的方法。他们将公式的标准分析结果和实际的分析结果都用 MathML 格式表示，在此基础上进行比较，检查那些典型公式结构（如角标，分式，根式，极限，矩阵等）的分析是否正确。首先，对应的两个公式结构的图像坐标一致时，才会比较两者的结构类型是否相符合，因此将子表达式图像的左上角和右下角坐标作为标记(tags)加入到 MathML 格式中，如图 2-19 所示。其次，用正确或错误表示公式结构比较的结果，即如果对应的两个公式结构类型一致，就认为分析是正确的，否则认为是错误的。考虑到尽管整个公式结构可能不完全相同，但是其中某些子表达式结构类型也许是一致的，所以他们用一种子结构方法来比较两个子结构。主要步骤如下：

1. 从标准分析结果中找每个子表达式最内部的标记；

2. 如果这个标记（比如<mfrac rect=”43,11.87.187”>）在实际分析结果中存在，则检查标记内部子表达式的结构类型是否相同。如果不相同，则记录该子表达式结构类型错误。

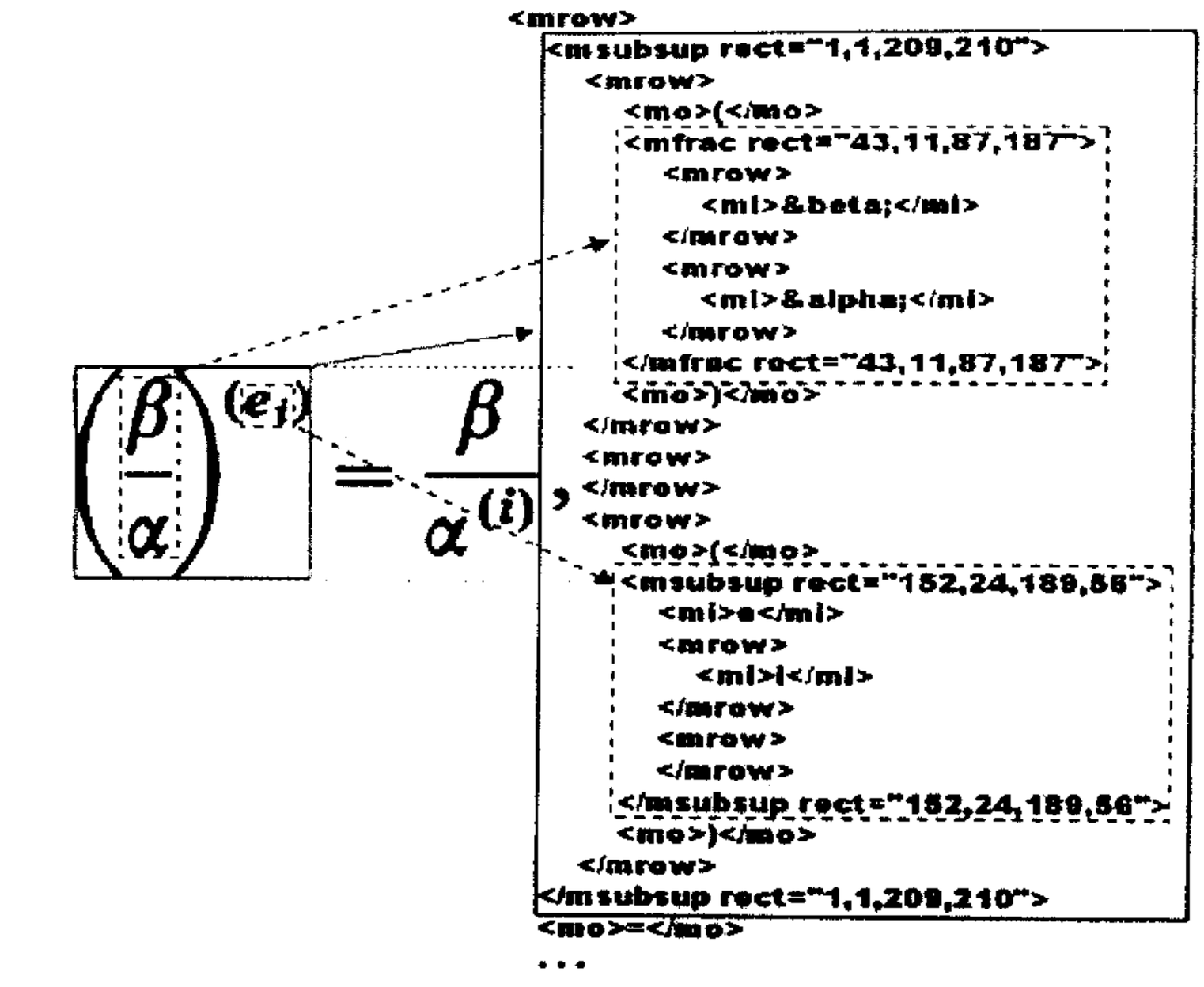


图 2-20：公式结构的比较示例

以图 2-20 所示的公式分析结果为例，首先，对于标准分析结果中的标记<mfrac rect= “43, 11, 87, 187” >，在实际分析结果中也检测到了，因此比较标记中两个子表达式类型，由标记中“mfrac”知道，它们具有相同的类型，都为分式，所以这个子表达式分析正确。

其次，同理可知标记<msubsup rect="152, 24, 189, 56">中的子表达式分析结果也是正确的再次，由于标准分析结果中的标记<msubsup rect="1, 1, 209, 210">在实际分析结果中不存在，因此认为该标记中的角标(subsup)结构类型分析错误。

对于简单的非常典型的公式结构，M.Okamoto 的方法效果比较好，但是对于复杂一点的结构，其比较结果不甚理想，因此该方法没有很好的通用性。此外，他们只是判断出分析结果是否正确，并没有指出错误的原因，这不满足我们评估的宗旨即为系统的设计开发者指出问题所在和改进方向。

基于目前的实际情况，实现数学公式识别的自动性能评估不仅是必要的而且是艰巨的。

2.3 数学公式图像识别的自动性能评估模型

2.3.1 自动性能评估模型

为了提高整个公式处理系统的性能，或者比较各种公式处理方法的效果，需要在大规模集合上进行测试，因此我们需要构建一个自动性能评估系统来定量地评估公式结果，当然这是相当困难的。

从 2.1 的讨论，我们知道为了对数学公式识别系统进行性能评估，我们需要有一个衡量性能的尺度，还需要一些基准数据，以及一个实用的算法来比较公式分析的结果与基准数据，从而得出性能评估的结果数据。对这些结果数据进行统计以后，将最终结果反馈给设计开发者，可以帮助他们很快地知道公式分析算法的效果。这些统计结果之中包括对错误的分类，对错误严重程度的估算，以及对这些错误可能产生的原因的分析。基于上述考虑，我们提出了图 2-21 所示的自动性能评估模型，这实际上是将数学公式识别系统与一个理想系统进行比较。。

下面，对性能评估模型中主要部分作一个简要说明：

- 基准数据是评估实际识别结果的一个参考标准。在我们的自动性能评估系统中，基准数据是每个数学公式的正确的三叉树结构。我们构造一个标准的数据库，用来存放各个数学公式的正确的三叉树格式文件。
- 公式识别的结果数据是系统实际识别所得的结果。
- 格式转换模块用于将实际的分析结果转换为带边属性的三叉树结构，该模块可以内嵌于公式识别系统中，也可以独立运作。
- 转换结果数据是将实际的识别结果通过格式转换所得，是实际识别结果的三叉树结构。

• 性能评估算法是自动性能评估系统的核心。一个有效而实用的性能评估算法是真正做到自动进行性能评估的关键，否则就只能纸上谈兵。有关性能评估算法参见 2.3.2 小节。

• 性能评估的结果是各种错误的描述文件和统计数字文件。

• 性能评估结果的表示工具是性能评估结果的表现形式，我们采用图形，表格，文本相结合的表示方式。

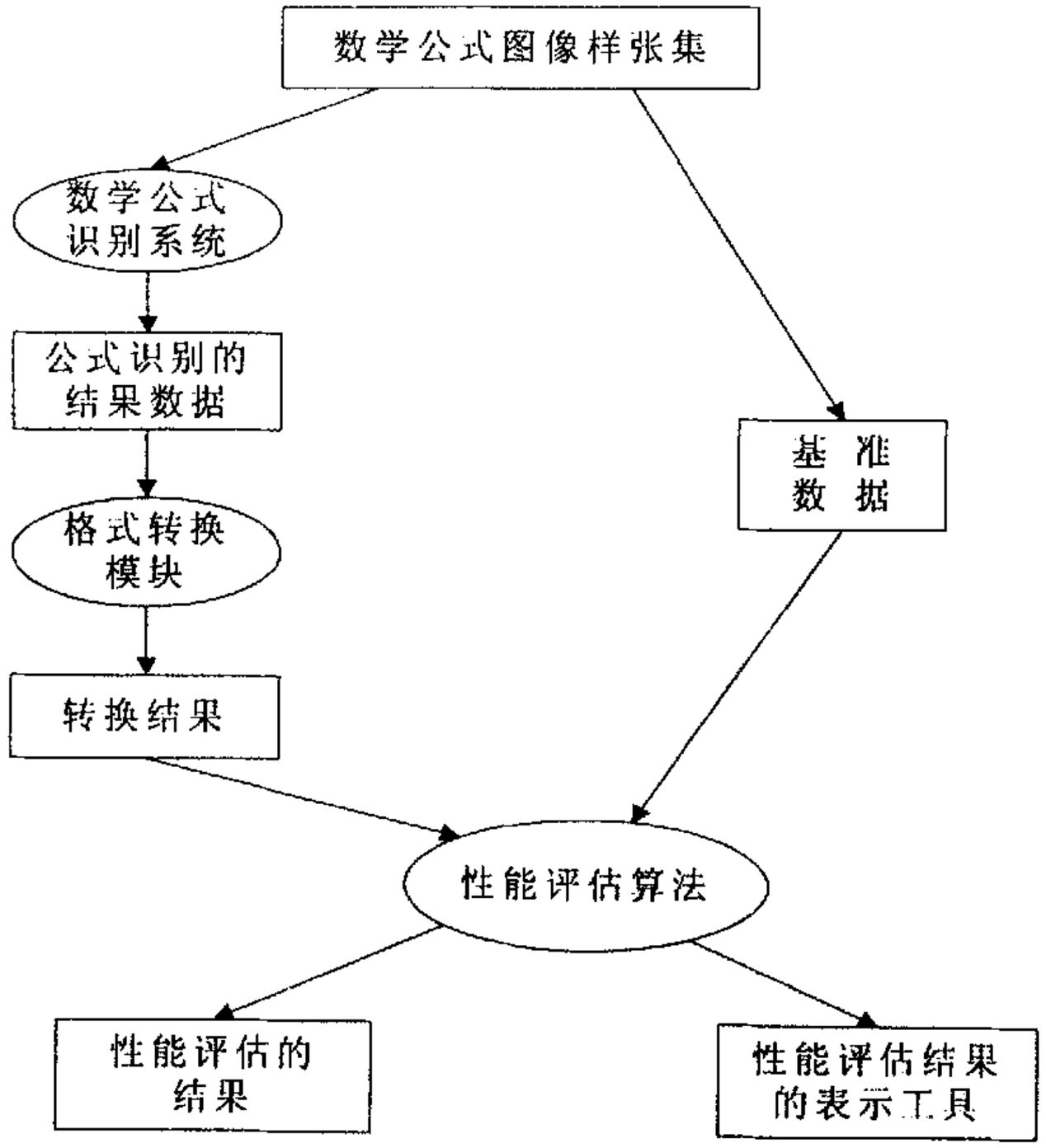


图 2-21：数学公式识别系统的自动性能评估模型

凡是能将实际识别结果通过格式转换模块转化为带边属性的三叉树结构的公式识别系统，都可以用我们基于该模型构建的自动性能评估系统来评测系统的性能。

2.3.2 性能评估算法概述

根据 2.3.1 的自动性能评估模型，我们设计了两个基于树匹配的算法来实现上述的评估方法。一是经过扩充的动态规划算法，二是 BUTD（Bottom-Up and Top-Down）算法。要构建这样一个自动性能评估系统，关键的有两点：

a. 定义性能指标，即从哪些方面来衡量识别结果的好坏。根据上述描述，对公式识别结果的评估，最终是将实际分析得到结果与基准数据进行比较，因此我们采用编辑距离表示

两棵树之间的差异，用该距离中出现的各种编辑操作的个数来定量描述相应的分析错误，并且给出各种统计结果。具体的编辑操作将根据所使用的树匹配算法而定，见 3.1 小节。

b. 设计出有效的树匹配算法。该算法在时间效率上应该是比较快的，否则性能评估占去太多的时间会给人一种本末倒置的感觉。

在动态规划算法中，我们定义了节点完全替代，节点内容修改，节点类型修改，节点插入，节点删除，边属性修改六种基本编辑操作，而在 BUTD 算法中我们定义了节点内容修改，节点类型修改，节点插入，节点删除，节点分裂，节点合并，节点转移和边属性修改八种基本编辑操作，用这些编辑操作的数目和比例来评估公式识别方法的性能。其中的节点内容修改反映了公式符号识别的性能，其它操作则主要反映了公式分析和符号切割的性能。两种树匹配算法的输入与输出分别为：

- 算法的输入

按照自动性能评估模型，算法的输入应该是待比较的公式实际分析结果的转换结果与相应的基准数据，即公式的三叉树。

- 算法的输出

输出是比较后的结果描述文件和性能评估的统计数字文件。结果描述文件中包括对正确情况的描述和对错误情况的描述，错误情况根据不同的编辑操作错误进行归类描述。而性能评估的统计数字文件则包含了公式分析的各种错误类型的数目和它们在整个错误中所占的比例以及各种公式类型的分析错误率。

在后续两章我们会给出两种算法的详细描述及相应的实例。

第三章 基于动态规划算法的自动性能评估

3.1 预备知识

由于我们的自动性能评估系统主要通过树匹配算法实现性能评估，因此先介绍有关树及树匹配的相关基础知识。

3.1.1 树的遍历

本文所涉及到的树都是指带标记的有序树，即树中每个节点都有一个标记(label)，甚至可能有某些附加的信息（这种信息被称作节点的内容），并且如果该节点有子节点，则子节点之间从左至右的顺序是固定的。这里所提到的从左到右的顺序是指按照某种方式对树进行遍历，得到所有节点的一个排序，每个节点在排序中的相对位置。所谓遍历就是将树中每个节点都访问一遍，通常有先根遍历，后根遍历几种方式等。先根遍历就是先访问根结点，再访问它的子节点；反之，先访问子节点再访问根节点则称之为后根遍历。

由于本文后面介绍的两个算法都采用后根遍历，所以在此举例说明。与一般树中的节点有所不同，我们的每个节点代表一种表达式类型，节点本身的结构比较复杂，而且对各种表达式处理也有差异，因此整个遍历过程相对复杂。但是基本的思路仍然与一般的后根遍历相同，只是每次访问到某个节点时，必须先判断它的表达式类型，再确定下一步的遍历方向。如图 3-1 所示为公式$\sum_{i=1}^{n} a_i + \frac{bx^3}{3}$的后根遍历的结果，其中每个节点右侧的(*i*)表示该节点在后根遍历排序中的序号。

3.1.2 树的距离度量

早在 1979 年 K.C.Tai 就在[9]提出了有关树与树之间的比较（或者称之为匹配）问题。至今，已经出现许多解决树匹配的方法，其中大多数都采用动态规划来实现，而动态规划的方法实际上是字符串匹配算法的一个推广。当然，也有一些其他的方法。例如[10]中将树匹配问题看作一个最优化问题，提出一个简单的递归的回溯算法，但是该算法的运行时间是指数级的，相较于动态规划算法它的速度要慢。

无论用什么方法解决树匹配问题，首先都必须定义树之间的距离度量，用以定量地描述出两棵树之间到底相差多少，这是前提条件。得益于字符串匹配问题的解决思路，树之

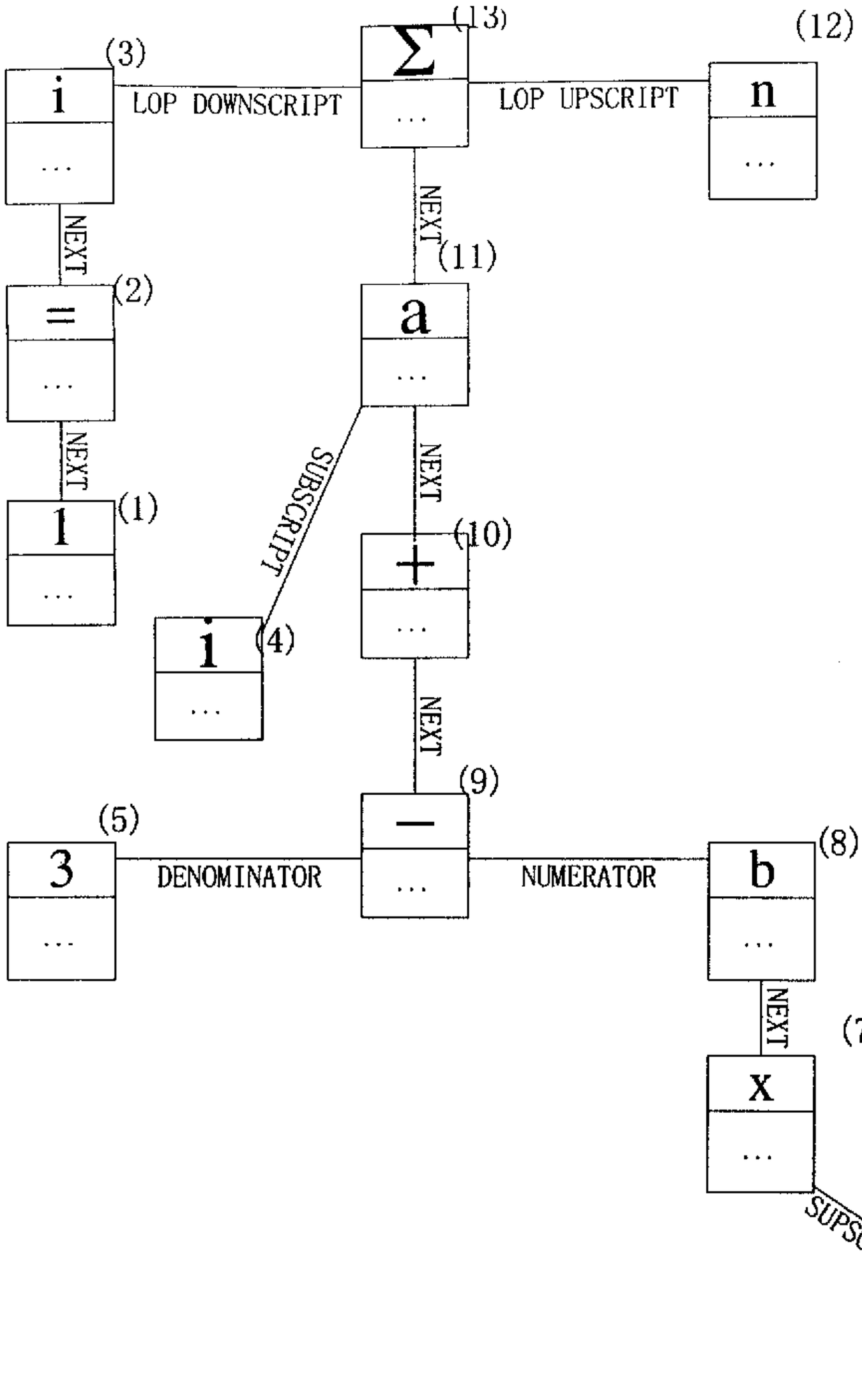


图 3-1：公式 $2\sum_{i=1}^{n} a_i + \frac{bx^3}{3}$ 后根遍历结果

间的距离也可以用编辑距离表示。因此树之间的匹配转化为对两棵树进行比较并且找出一个基本编辑操作（如插入，删除，替代等）的序列，该序列满足以下两个条件：

（1） 按照该序列进行操作能够将一棵树转化成另一棵树；

（2） 在满足（1）的所有基本编辑操作序列中，该序列的操作代价最小。

除了编辑距离之外，诸如对齐距离、独立子树距离、Top-Down 距离以及 Bottom-Up 距离等等都可以用来表示树之间的距离度量。事实上，这些距离度量都是在编辑距离的基

础上附加一些限制条件得来的。本文采用编辑距离来度量树之间的距离，将在后面详细介绍。

3.1.3 编辑操作与编辑距离

对于一般的带标记的有序树，我们通常考虑三种基本的编辑操作：节点的替代(change 或 relabel)，删除（delete）和插入（insert）。节点替代是指改变一个节点 υ 的标记；节点删除是指将一个节点 υ 从树中删除，并且使 υ 的子节点变成 υ 的父节点的子节点；节点插入与删除操作正好相反，插入一个新节点 υ 使之成为已有节点 υ' 的子节点，则 υ' 的某些子节点（可能为空）将成为 υ 的子节点。

按照[9][10]，可以用(a, b)表示一个编辑操作，符号 Λ 表示空节点，则(a, b)≠(Λ, Λ)，a 或者为 Λ 或者是树 T_1 中某个节点的标记，b 或者为空或者是树 T_2 中某个节点的标记。当 a 和 b 都不为 Λ 时，我们称 a→b 是一个替代操作；当 a≠Λ 而 b=Λ 时，我们称 a→b 是一个删除操作；当 a=Λ 而 b≠Λ 时，我们称 a→b 是一个插入操作。图 3-2 显示了各种操作的结果。

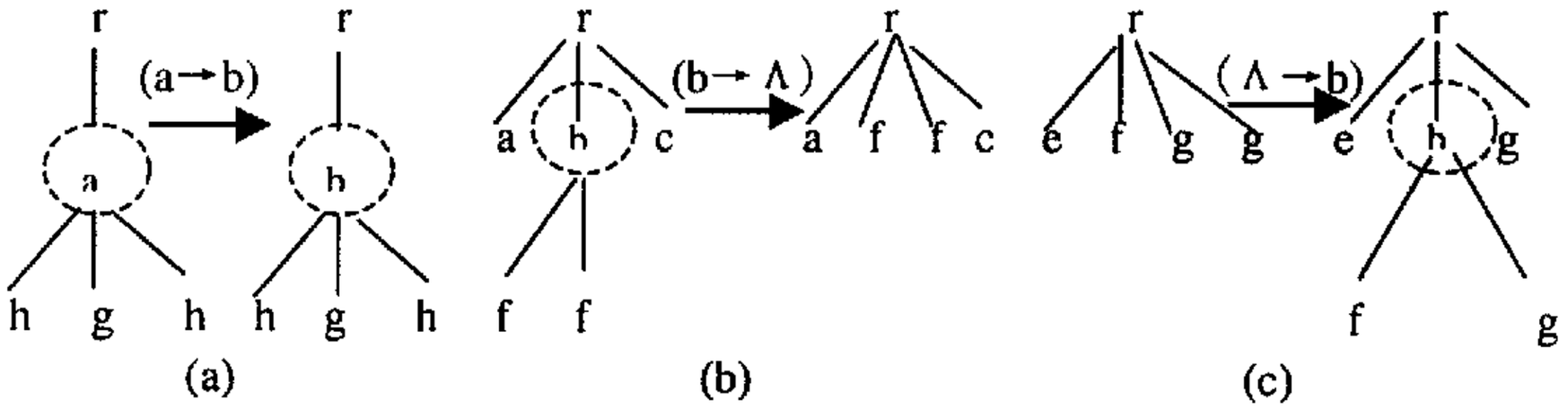


图 3-2：三种操作 (a)替代 (b)删除 (c)插入

定义 E 为一个编辑操作序列 $e_1, \ldots, e_k$，则从 A 到 B 的 E—派生为一个树的序列 $A_0, \ldots, A_k$，其中，$A=A_0$，$B=A_k$，A_{i-1} 通过 e_i 转变为 A_i（$1 \leqslant i \leqslant k$）。定义函数 $\gamma(a \to b)$ 为编辑操作 a→b 的代价函数，取值为非负实数。不同的节点可以赋给它不同的代价函数值（简称代价值），例如，对于树中那些较高的节点可以赋给它们较大的代价值，而对于那些较低的节点可以赋给它们较小的代价值。γ 是一个距离度量，即满足以下条件：

(1). $\gamma(a \to b) \geqslant 0$; $\gamma(a \to a) = 0$;

(2). $\gamma(a \to b) = \gamma(b \to a)$;

(3). $\gamma(a \to c) \leqslant \gamma(a \to b) + \gamma(b \to c)$.

在此，可以将 γ 由编辑操作的代价扩展到编辑操作序列的代价。令

$$\gamma(E) = \sum_{i=1}^{i=\|E|} \gamma(e_i) \quad ,$$

其中，$|E|$ 表示编辑操作序列 E 的长度，即 E 中编辑操作的个数。则树 T_1 与树 T_2 之间的距离定义为：

$$\delta(T_1, T_2) = \min\{\gamma(E) \mid E \text{ 是将 } T_1 \text{ 转变为 } T_2 \text{ 的编辑操作序列}\}$$

由 γ 的定义可以证明 δ 也是一个距离度量。

3.1.4 映射

为了计算 $\delta(T_1, T_2)$，我们考虑从 T_1 到 T_2 的映射。所谓映射就是对作用于 T_1 和 T_2 的每个节点的编辑操作的一个图形化表示，或者理解成一种描述树之间如何通过编辑操作相互转变的工具。假设我们已经知道每棵树的相关信息，用 T[i]表示树 T 的第 i 个节点。则可以定义一个三元组(M, T_1, T_2)，称之为从 T_1 到 T_2 的映射，其中 M 是任意整数对(i, j)的集合，满足：

1. $1 \leqslant i \leqslant |T_1|, 1 \leqslant j \leqslant |T_2|$；$|T_1|$ 和 $|T_2|$ 分别表示树 T_1 和 T_2 的节点数

2. 对于 M 中的任意整数对(i_1, j_1)和(i_2, j_2)，有

(a) $i_1 = i_2$ 当且仅当 $j_1 = j_2$ （一对一）；

(b) $T_1[i_1]$是 $T_1[i_2]$的祖先当且仅当 $T_2[j_1]$是 $T_2[j_2]$的祖先（保持父子或祖孙次序不变）；

(c) $T_1[i_1]$在 $T_1[i_2]$左边当且仅当 $T_2[j_1]$在 $T_2[j_2]$左边（保持兄弟或左右次序不变）。

图 3-3 显示了上述的 2(b)和 2(c)两条映射规则。

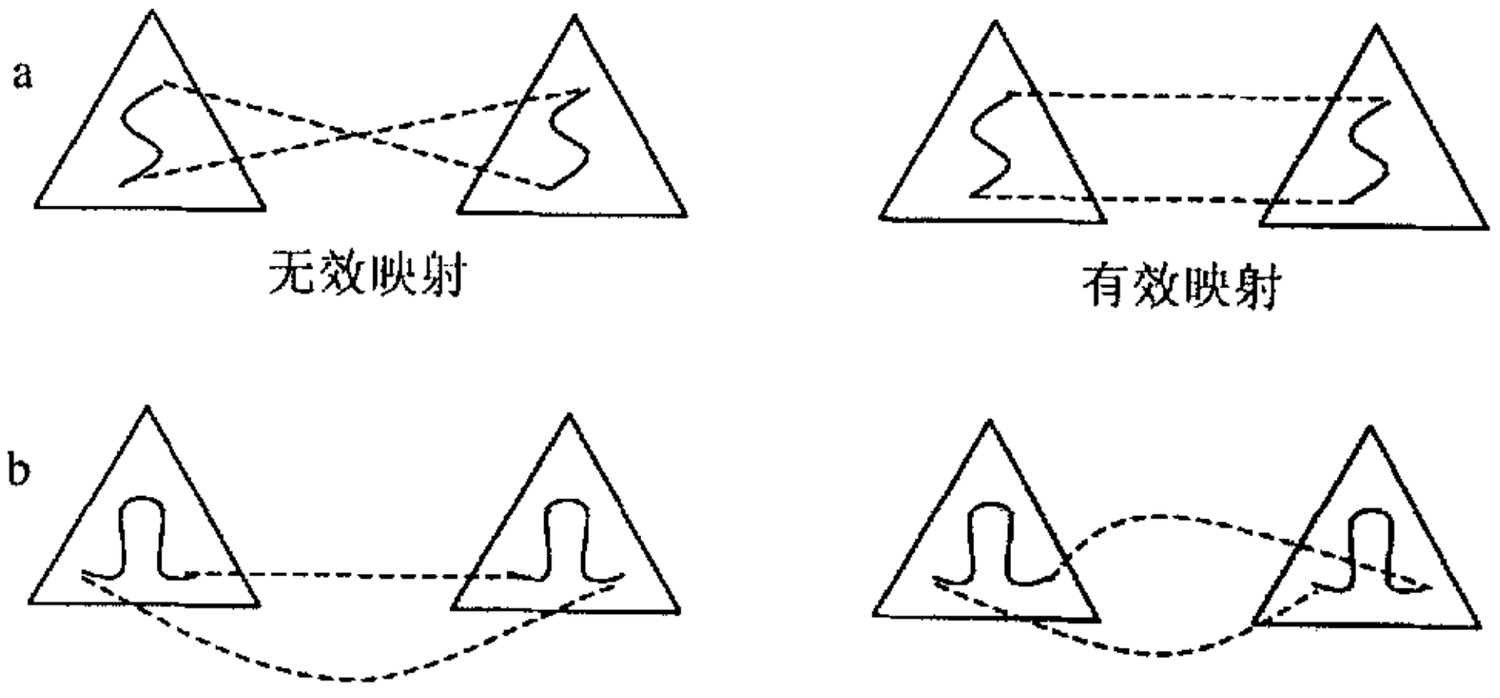


图 3-3 映射规则：a.祖孙次序不变；b.兄弟次序不变

根据映射(M, T_1, T_2)的定义可以得出：对于任一节点 $T_1[i]$，如果不存在整数对 (i, j)

$\in M$，则意味着该节点从 T_1 中删除了；否则表示该节点被另一节点 $T_2[j]$所替代了；对于任一节点 $T_2[j]$，如果不存在整数对 $(i, j) \in M$，则该节点是新插入 T_2 的。其中替代又分成两种：一致替代（对应节点的标记没有改变）和非一致替代（对应节点的标记发生改变）。

由此可以把 T_1 和 T_2 中的所有节点分成三部分：第一部分称为替代节点（由于一致替代相当于没有发生变化，代价为 0，因此我们只考虑非一致替代），用 S 表示；第二部分称为删除节点，用 D 表示；第三部分称为插入节点，用 I 表示。在不会引起歧意的情况下，我们用 M 代替三元组(M, T_1, T_2)。则映射 M 的代价值等于 $|S|p + |D|r + |I|q$，其中 p，r，q 分别是替代，删除，插入操作的代价值。

通过映射我们很容易就能看出树 T_1 和树 T_2 是如何相互转变的并且计算出相应转变过程的代价。如图 3-4 所示，T_1 中的节点 $T_1[1]$，$T_1[7]$，$T_1[8]$，$T_1[9]$，$T_1[10]$与 T_2 中的节点 $T_2[1]$，$T_2[5]$，$T_2[6]$，$T_2[7]$，$T_2[8]$相对应，即由虚线连接的两个节点存在替代关系，而节点 $T_1[2]$，$T_1[3]$，$T_1[4]$，$T_1[5]$，$T_1[6]$从 T_1 中删除了，同时，在 T_2 中插入节点 $T_2[2]$，$T_2[3]$，$T_2[4]$。这里的节点 $T[i]$指的是按先根遍历次序为 i 的节点。用 n_1，n_2分别表示 T_1 和 T_2 的节点数，则删除节点数为 $n_1 - |M| = 10-5 = 5$；插入节点数为 $n_2 - |M| = 8-5 = 3$；非一致替代节点数为 0。因此该映射 M 的代价值为 $5r + 3q$。若假设各种编辑操作的代价值均为 1，则 M 的代价值为 8。

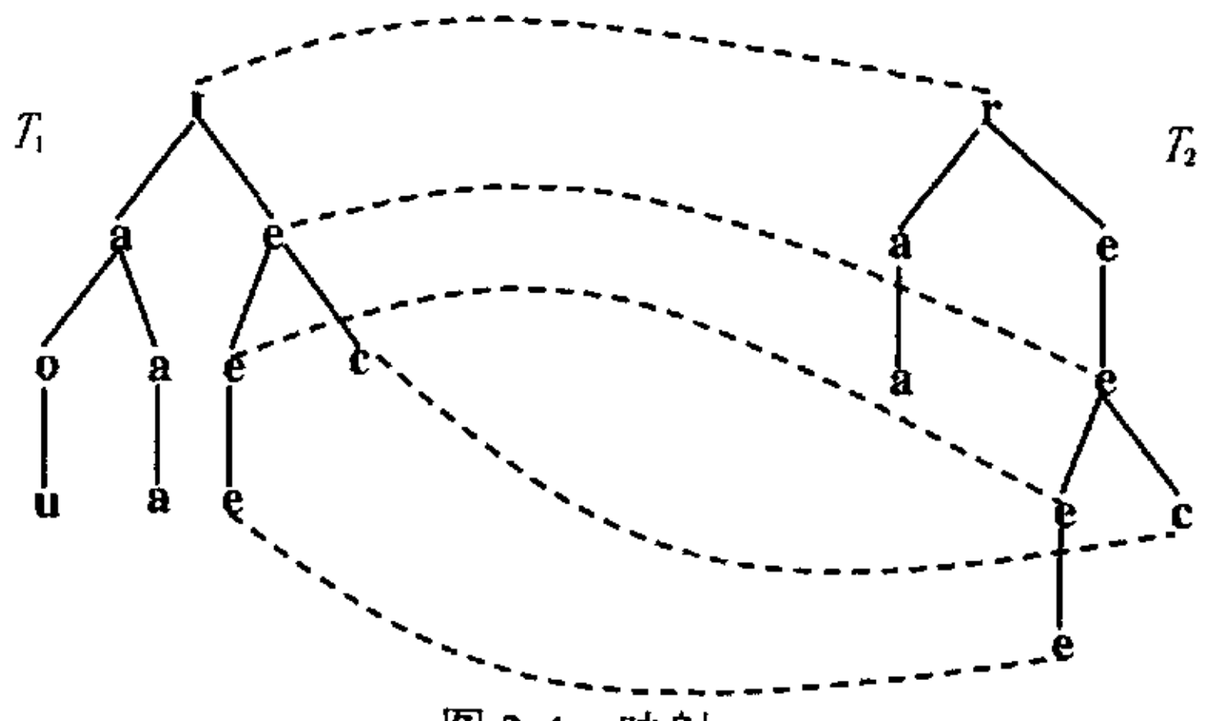


图 3-4：映射

映射与编辑操作序列之间的关系为：给定一个编辑操作序列 E，则存在一个从 T_1 到 T_2 的映射 M 使得 $\gamma(M) \leq \gamma(E)$；反之，对于任意的映射 M，存在一个编辑操作序列 E，使得 $\gamma(M) = \gamma(E)$。因此，从 T_1 到 T_2 的编辑距离

$\delta(T_1, T_1) = \min\{\gamma(M) \mid$M 是从 T_1 到 T_2 的映射$\}$。

3.2 动态规划算法

字符串是树的一个非常重要的特例，许多有关字符串的问题都能用动态规划算法来很好的解决。因此很自然地就会尝试用动态规划来实现树之间的匹配，而且已经有很多文献涉及这个问题。虽然他们所提到的树与我们的树之间存在一些差异，例如节点本身结构比较复杂，并且还涉及到边的顺序和属性等，但是一些概念仍然适用。在动态规划算法中我们会将树的匹配推广到森林的匹配，但是前面所提到的编辑距离和映射同样有效。

3.2.1 概念与符号定义

首先我们要解释节点匹配的概念。设公式 E 的标准分解树是 T_1，分析得到的分解树是 T_2，对任意的两个节点 $T_1[i]$和 $T_2[j]$，存在以下几种情况：

(1) $T_1[i]$和 $T_2[j]$的外接矩形框的坐标值不一致，其中外接矩形框是指节点对应的子表达式图像的外接矩形框，这种情况下，我们不会对两个节点作进一步的比较。

(2) $T_1[i]$和 $T_2[j]$的外接矩形框的坐标值一致，此时我们将比较两个节点的类型及识别结果，根据比较结果的不同又可以分成三种情况：

a. $T_1[i]$和 $T_2[j]$的节点类型不一致，则无需比较两者的识别结果；

b. $T_1[i]$和 $T_2[j]$的节点类型一致，而识别结果不同；

c. $T_1[i]$和 $T_2[j]$的节点类型和识别结果都一致。

我们称(1)和(2)a 两种情况为节点不匹配，(2)b 为半匹配，(2)c 为完全匹配。在后文中所提到匹配指的就是后两种情况。

考虑到实际应用中，我们的树结构与一般带标记的有序树不太一样，因此对三种基本编辑操作进行扩充。我们定义六种基本编辑操作：

1. 节点完全替代，Change(T_1,n_1,n_2)：用节点 n_2 代替树 T_1 的节点 n_1，即上文的情况(1)。

2. 节点类型修改，Type (T_1,n,t_1,t_2)：修改树 T_1 中节点 n 的类型 t_1 为 t_2，即上文的情况(2)a。

3. 节点内容修改，Content(T_1,n,c_1,c_2)：修改树 T_1 中节点 n 的内容 c_1 为 c_2，即上文的情况(2)b。

4. 节点插入，Insert(T_2,n_1,m,n_2)：在树 T_2 的节点 n_1 的第 m 个子节点位置插入节点 n_2，原来 n_1 的第 m 个子节点变为 n_2 的子节点。该节点只出现在 T_2 中，T_1 中的所有节点与它都满足上文的情况(1)。

5. 节点删除，Delete(T_1,n,m)：删除树 T_1 的节点 n 的第 m 个子节点，节点 n 的第 m 个子节点的子节点变为 n 的子节点，即 T_2 中的所有节点与该节点都满足上文的情况(1)。

6. 边属性修改，Edge(T_1,n_1,n_2,e)：修改树 T_1 中连接节点 n_1 和 n_2 的边的属性为 e。指某条边两端的节点都相匹配，而边的属性发生变化，这通常是由于表达式类型的误判导致的，因此在最后的输出阶段我们会把这种错误归类到表达式类型替代中。

对于节点完全替代操作，我们认为这两个节点不匹配，无需再比较它们的类型和内容，即这两个节点之间不存在任何关系的错误，但在后面介绍的动态规划算法中会遇上这种情况，因此我们考虑这种操作，并且赋予该操作较高的代价值。我们把后五种基本操作看作相应的五种错误类型，同时赋予各种基本操作同样的代价值，即单位代价值 1。由定义可知五种错误类型是彼此不重叠的。

根据操作对象的不同，上述五种操作可以分为两类:节点编辑操作和边编辑操作，分别用 a→b 和 E_1→E_2 表示。就节点编辑操作而言，与 3.2.2 中的定义相吻合，只是将节点的修改扩充为 1 和 2 两种情况，因此对于 3.2.2 中的各种定义都适用。此外，对于给定的两棵树 T_1 和 T_2，我们认为两者之间的边属性错误数是定值，而且与边编辑操作的顺序无关，可以通过扫描所有的边得到这个值。定义函数 γ(a→b)为编辑操作 a→b 的代价函数，c(E_1→E_2)为编辑操作 E_1→E_2 的代价函数，取值都为非负实数。令 C 为所有边编辑操作的代价和，E 为一个节点编辑操作序列 e_1，…，e_k，则有 $\gamma(E)=\sum_{i=1}^{i=|E|}\gamma(e_i)$，得到 T_1 和 T_2 的距离为：

$\delta(T_1, T_2)=\min\{\gamma(E)\mid E$ 是将 T_1 转变为 T_2 的编辑操作序列$\}+C$。

因此，求 T_1 和 T_2 之间距离的关键是找到一个代价最小的节点编辑操作序列。后文的动态规划算法主要就是用来求最小节点编辑操作序列的。下面对算法中涉及到的术语和符号作一说明。

我们采用后根遍历的序号来标识树中的每个节点，则有如下定义：

定义 1　令 $T[i]$为树中第 i 个节点，$l(i)$是以 $T[i]$为根的子树的最左边叶子节点的序号。当 $T[i]$为叶子节点时，$l(i)=i$。用 $p(i)$表示 $T[i]$的父节点，定义 $p^0(i)=i$，$p^1(i)=p(i)$，$p^2(i)=p(p^1(i))\ldots$，令 $anc(i)=\{p^k(i)\mid 0\leqslant k\leqslant T[i]$的深度$\}$。

定义 2　$T[i..j]$是由第 i 至第 j 个节点构成的有序子森林，如图 3-5。当 $i>j$ 时，$T[i..j]=\varnothing$。用 $forest(i)$ 表示 $T[1..i]$，$tree(i)$表示 $T[l(i)..i]$，$size(i)$表示子树 $tree(i)$的节点数。

定义 3　用 $forestdist(T_1[i'..i], T_2[j'..j])$表示 $T_1[i'..i]$和 $T_2[j'..j]$之间的距离，简记为 $forestdist(i'..i, j'..j)$。用 $forestdist(i, j)$表示 $T_1[1..i]$和 $T_2[1..j]$之间的距离。用 $treedist(i, j)$表示子树 $T_1[l(i)..i]$和 $T_2[l(j)..j]$之间的距离。

3.2.2 动态规划算法描述

我们将用动态规划算法求解树 T_1 和 T_2 之间的最小节点编辑操作序列，然后加上边编辑操作，即可得到 T_1 和 T_2 的距离，同时给出各种编辑操作的个数。在介绍具体的算法之前，先给出三个与[13]中类似的引理，其中编辑操作 $T_1[i]\rightarrow T_2[j]$包括节点类型修改和节点内容修改两种。

引理 1　令 $anc(i)=\{p^k(i)|0\leqslant k\leqslant T[i]$的深度$\}$，则有

(1) $forestdist(\varnothing,\varnothing)=0$;

(2) $forestdist(T_1[l(i_1)..i],\varnothing)=forestdist(T_1[l(i_1)..i\text{-}1],\varnothing)+\gamma(T_1[i]\rightarrow\Lambda)$;

(3) $forestdist(\varnothing,T_2[l(j_1)..j])=forestdist(\varnothing,T_2[l(j_1)..j\text{-}1])+\gamma(\Lambda\rightarrow T_2[j])$;

其中，$i_1\in anc(i)$，$j_1\in anc(j)$

证明：(1) T_1 和 T_2 都为空，则无需任何编辑操作，因而距离得 0;

(2) T_2 为空，显然 T_1 要转变为 T_2，需要进行$|T_1|$次删除操作;

(3) T_1 为空，要得到 T_2，T_1 需要进行$|T_2|$次插入操作。

这个引理实际上是 T_1 和 T_2 之间距离的边界条件。

引理 2　令 $i_1\in anc(i)$，$j_1\in anc(j)$，则有

$$forestdist(l(i_1)..i,l(j_1)..j)=\min\begin{cases}forestdist(l(i_1)..i-1,l(j_1)..j)+\gamma(T_1[i]\rightarrow\Lambda)\\forestdist(l(i_1)..i,l(j_1)..j-1)+\gamma(\Lambda\rightarrow T_2[j])\\forestdist(l(i_1)..l(i)-1,l(j_1)..l(j)-1)\\+forestdist(l(i)..i-1,l(j)..j)\\+\gamma(T_1[i]\rightarrow T_2[j])\end{cases}$$

证明：考虑 $l(i_1)\leqslant i\leqslant i_1$ 和 $l(j_1)\leqslant j\leqslant j_1$ 的情况，可以通过构造 $forest(l(i_1)..i)$和 $forest(l(j_1)..j)$之间的最小代价映射来计算 $forestdist(l(i_1)..i,\ l(j_1)..j)$。对于 $T_1[i]$和 $T_2[j]$，在映射图中存在三种情况：

(1) 映射 M 中没有虚线连接 $T_1[i]$，则 $T_1[i]$被删除，$(i,\Lambda)\in M$，因此

$forestdist(l(i_1)..i,\ l(j_1)..j)=forestdist(l(i_1)..i\text{-}1,\ l(j_1)..j)+\gamma(T_1[i]\rightarrow\Lambda)$;

(2) 映射 M 中没有虚线连接 $T_2[j]$，则 $T_2[j]$被插入，$(\Lambda,j)\in M$，因此

$forestdist(l(i_1)..i,\ l(j_1)..j)=forestdist(l(i_1)..i,\ l(j_1)..j\text{-}1)+\gamma(\Lambda\rightarrow T_2[i])$;

(3) 映射 M 中有一虚线连接 $T_1[i]$和 $T_2[j]$，则$(i,j)\in M$，因为假设(i,k)和(h,j)都属于 M，如果 $l(i_1)\leqslant h\leqslant l(i)\text{-}1$，则 i 在 h 右边，由映射规则 2(c)知道 k 必定也在 j 右边，但这种情形在 $forest(l(j_1)..j)$中是不可能出现的。同理，如果 i 是 h 的祖先，则由映射规则 2(b)知道 k 必定

也是 j 的祖先，这也是不可能的。所以，$h=i$，由对称性，$k=j$，所以$(i,j) \in M$。

由映射规则 2(b)，以 $T_1[i]$为根的子树中任意节点只能与以 $T_2[j]$为根的子树中的节点互相映射。因此

$$forestdist(l(i_1)..i,\ l(j_1)..j) = forestdist(l(i_1)..l(i)-1,\ l(j_1)..l(j)-1)$$
$$+ forestdist(l(i)..i-1,\ l(j)..j-1) + \gamma(T_1[i] \to T_2[j])。$$

图 3-5 显示了这种情况。

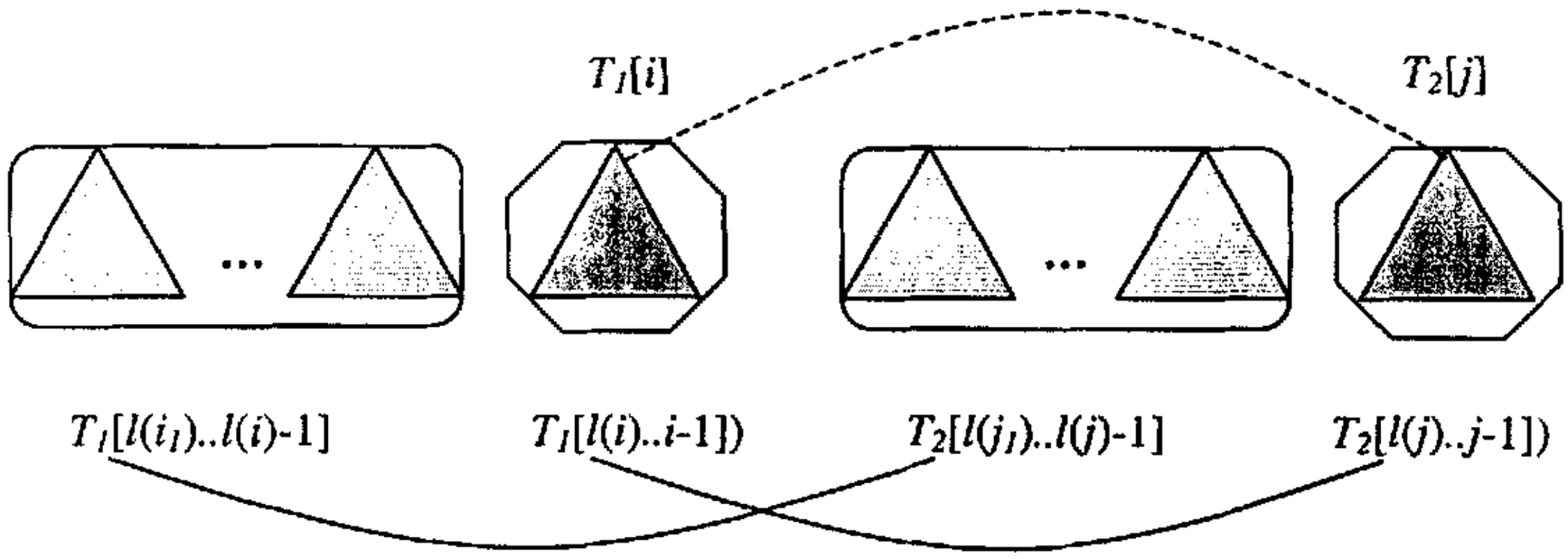


图 3-5：引理 2 中第三种映射情况示例

引理 3　令 $i_1 \in anc(i)$，$j_1 \in anc(j)$，则有

(1) 当 $l(i)=l(i_1)$并且 $l(j)=l(j_1)$时，有

$$forestdist(l(i_1)..i, l(j_1)..j) = \min\begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j]) \\ forestdist(l(i_1)..i-1, l(j_1)..j-1) + \gamma(T_1[i] \to T_2[j]) \end{cases}$$

(2) 当 $l(i)\neq l(i_1)$或者 $l(j)\neq l(j_1)$时，有

$$forestdist(l(i_1)..i, l(j_1)..j) = \min\begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j]) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) \\ + treedist(i,j) \end{cases}$$

证明：(1)当 $l(i)=l(i_1)$和 $l(j)=l(j_1)$时，由引理 2，

$forestdist(l(i_1)..l(i)-1,\ l(j_1)..l(j)-1) = forestdist(\emptyset, \emptyset) = 0$，(1)得证。

(2)由于我们所求的距离应该是最小代价映射的代价值，有

$forestdist(l(i_1)..i,\ l(j_1)..j) \leqslant forestdist(l(i_1)..l(i)-1,\ l(j_1)..l(j)-1) + treedist(i,j)$，因为该不等式右边是 $forest(l(i_1)..i)$到 $forest(l(j_1)..j)$的映射的特殊情况，因此可能是子最有的。同理，$treedist(i,j) \leqslant forestdist(l(i)..i-1,\ l(j)..j-1) + \gamma(T_1[i] \to T_2[i])$。由引理 2 和这两个不等式，可以知道用 $treedist(i,j)$ 代替 $forestdist(l(i)..i-1,\ l(j)..j-1) + \gamma(T_1[i] \to T_2[i])$是正确的，因此(2)得证。图 3-6 给出了这种情况。

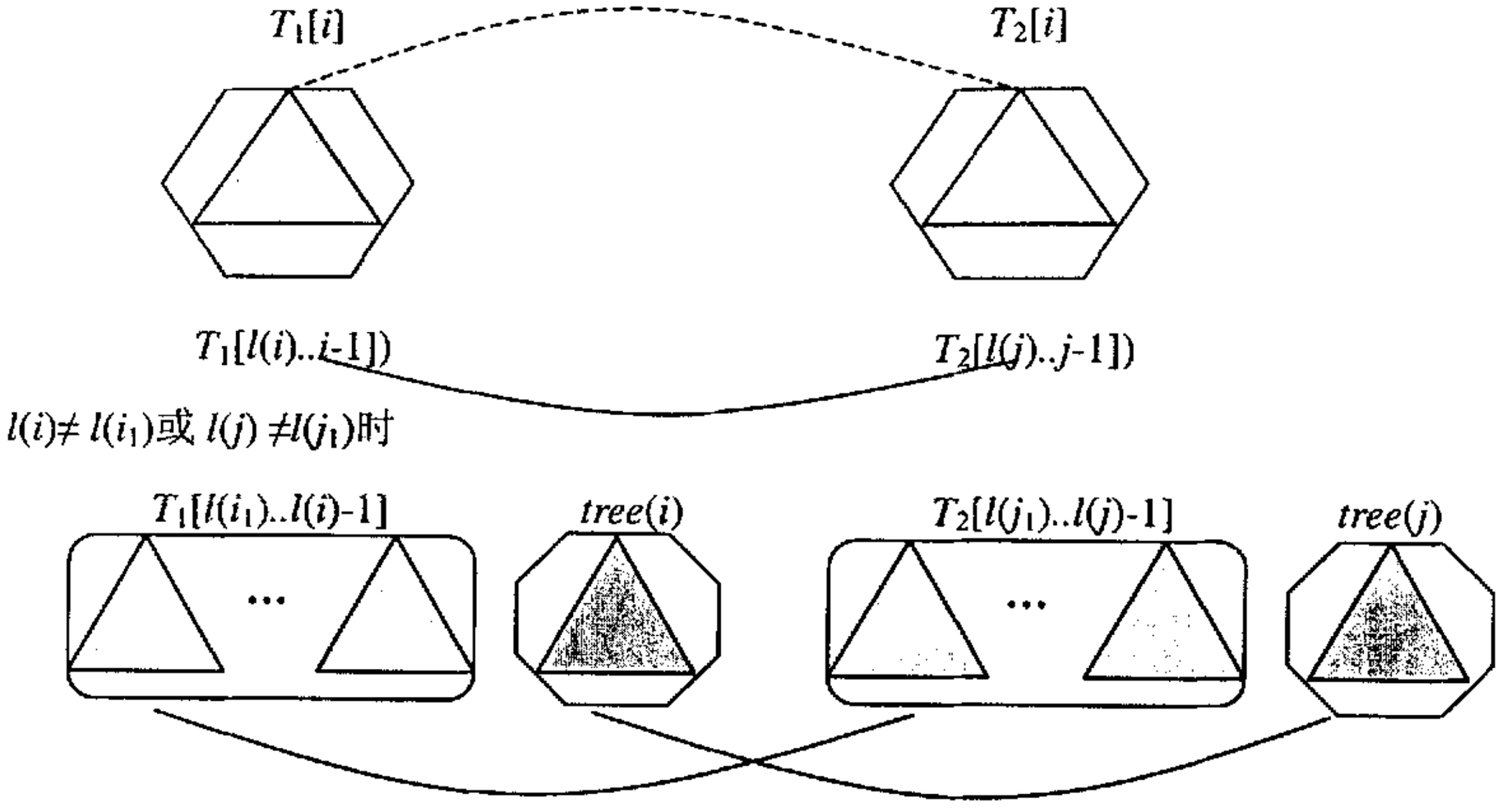


图 3-6：引理 3 两种情况图示

由引理 3 可以得出以下三点：

(1) 距离公式表明可以用动态规划算法来求解；

(2) 引理第二种情况表明，求解 $treedist(i_1, j_1)$的前提是必须已经求得几乎所有的 $treedist(i, j)$，其中 i 是以 i_1 为根的子树的节点，j 是以 j_1 为根的子树的节点，因此可以用自底向上的过程实现。

(3) 引理第一种情况表明，当 i 和 j 分别处在 $l(i_1)$到 i_1 和 $l(j_1)$到 j_1 的路径上时，无需专门计算 $treedist(i, j)$，因为这些子树的距离可以从计算 $treedist(i_1, j_1)$的过程中直接获得。

在此，还需要为树 T 定义一个集合 $LR\_keyroots$：

$LR_keyroots(T) = \{k|$不存在 k'使 $l(k) = l(k')\}$。

对于集合 $LR\_keyroots(T)$，若 k 属于 $LR\_keyroots(T)$，则 k 或者是树 T 的根节点，或者满足 $l(k) \neq l(k')$，例如 k 有一个左兄弟节点。直观上看，集合 $LR_keyroots(T)$是那些需要单独计算距离的子树的根节点。如图 3-7 所示，$LR\_keyroots(T_1) = \{3,5,6\}$，$LR\_keyroots(T_2) = \{2,5,6\}$。

图 3-7：树 T_1 和 T_2

算法详细描述如下：

(1)输入：树 T_1 和 T_2，即表示公式正确结构的三叉树与实际分析所得的三叉树。

(2)后根遍历树 T_1 和 T_2，同时通过函数 $l(T)$和 *LR_keyroots*(T)分别计算 $l(i)$和集合 *LR_keyroots*，将结果保存在数组 *l*1，*l*2，*LR_keyroots*1 和 *LR_keyroots*2 中。

(3)树匹配过程，主要循环为：

FOR i':= 1 to $\|LR\_keyroots(T_1)\|$

FOR j' := 1 to $\|LR\_keyroots(T_2)\|$

$i = LR\_keyroots1[i']$;

$j = LR\_keyroots2[j']$;

计算 *treedist*(i,j);

用动态规划算法计算 *treedist*(i,j)，其中子森林之间的距离保存在一个临时数组中，直到求得相应的子树距离，释放该临时数组，子树距离保存在数组 *treedist* 中。

treedist(i,j)计算过程：

forestdist$(\emptyset,\emptyset) = 0$;

FOR $i_1 := l(i)$ to i

forestdist$(T_1[l(i)..i_1],\emptyset) =$ *forestdist*$(T_1[l(i)..i\text{-}1],\emptyset) + \gamma(T_1[i_1]\rightarrow\Lambda)$;

FOR $j_1 := l(j)$ to j

forestdist$(\emptyset ,T_2[l(j)..j_1]) =$ *forestdist*$(\emptyset ,T_2[l(j)..j_1\text{-}1]) + \gamma(\Lambda\rightarrow T_2[j_1])$;

FOR $i_1 := l(i)$ to i

FOR $j_1 := l(j)$ to j

IF($l(i_1) = l(i)$&& $l(j_1)=l(j)$)

forestdist$(l(i)..i_1,\ l(j)..j_1) = \min\{$

forestdist$(l(i)..i_1\text{-}1,\ l(j)..j_1) + \gamma(T_1[i_1]\rightarrow\Lambda)$,

forestdist$(l(i)..i_1,\ l(j)..j_1\text{-}1) + \gamma(\Lambda\rightarrow T_2[j_1])$,

forestdist$(l(i)..i_1\text{-}1,\ l(j)..j_1\text{-}1) + \gamma(T_1[i_1]\rightarrow T_2[j_1])\}$;

记录最小距离对应的编辑操作；

treedist(i_1,j_1)= *forestdist*$(l(i)..i_1,\ l(j)..j_1)$;

记录子树之间的编辑操作序列；

ELSE

forestdist$(l(i)..i_1,\ l(j)..j_1) = \min\{$

$$forestdist(l(i)..i_1\text{-}1,\ l(j)..j_1)+\gamma(T_1[i_1]\rightarrow\Lambda),$$

$$forestdist(l(i)..i_1,\ l(j)..j_1\text{-}1)+\gamma(\Lambda\rightarrow T_2[j_1]),$$

$$forestdist(l(i)..l(i_1)\text{-}1,\ l(j)..l(j_1)\text{-}1)+treedist(i_1,j_1);$$

记录最小距离对应的编辑操作；

(4) 根据数组 *treedist* 求得各种节点编辑操作数。

(5) 扫描所有的边，计算边编辑操作数。

(6) 输出结果。

3.2.3 动态规划算法的分析

根据 3.2.2 的算法描述，在空间上主要用到两个数组：一个是保存最后计算结果的数组 *treedist*，另一个是计算 *treedist*(*i*,*j*)时所需的临时数组。这两个数组的空间需求分别是 $O(\|T_1\|*\|T_2\|)$。在时间上，可以将主要过程分成两部分：计算节点编辑操作的时间和计算边编辑操作的时间。

1. 先看计算边编辑操作的时间：

由于只需要扫描 T_1 中每条边，与 T_1 中相应的边进行比较，因此这个时间是线性的。

2. 计算节点编辑操作的时间分析如下：

首先，后根遍历同时利用函数 *l*(*T*)和 *LR_keyroots*(*T*)分别计算 *l*(*i*)和集合 *LR_keyroots*，这都是线性时间复杂度的。

其次，在树匹配过程中，每一次用动态规划算法计算 *treedist*(*i*,*j*)所需时间为 (*i*-*l*(*i*)+1)*(*j*-*l*(*j*)+1)，即以 *i* 为根的子树的节点数与以 *j* 为根的子树的节点数之积，分别用 *size*(*i*)和 *size*(*j*)表示，则计算 *treedist*(*i*,*j*)所需时间等于 *size*(*i*)×*size*(*j*)。总共调用动态规划算法若干次，令 $m=\|LR\_keyroots(T_1)\|$，$n=\|LR\_keyroots(T_2)\|$，则整个树匹配所需时间为

$$\sum_{i=1}^{m}\sum_{j=1}^{n} size(i)\times size(j)$$

可以进一步分析该时间复杂度的上限：

(1)令 *leaves*(*T*)为树 *T* 所有叶子节点的集合，则有 $\|LR\_keyroots(T)\|\leqslant\|leaves(T)\|$。

我们很容易就可以证明这个不等式。根据集合 *LR_keyroots*(*T*)的定义可知对于任意的 $i,j\in LR\_keyroots(T)$，必然有 $l(i)\neq l(j)$。*l*(*i*)表示的正是 *i* 的最左叶子节点，因此，*LR_keyroots*(*T*)中每个节点都与一个最左叶子节点唯一对应，而每个叶子结点至多是 *LR_keyroots*(*T*)中一个节点的最左叶子节点，所以 $\|LR\_keyroots(T)\|\leqslant\|leaves(T)\|$ 成立。

(2)因为并非所有子树之间的距离都需要计算的，所以每个节点需要参加距离计算的次数小于它的深度。令 $num(i)$为节点 i 参与计算的次数，则

$num(i) = \| anc(i) \cap LR\_keyroots(T) \|$

定义 $num(T) = \max\{ num(i)\}$，$d(T)$表示树 T 的深度，$l(T) = \| leaves(T) \|$，由定义以及(1)可知 $num(i) \leqslant \min(d(T), l(T))$，因此 $num(T) \leqslant \min(d(T), l(T))$。

(3)等式 $\sum_{i=1}^{m} size\ (i) = \sum_{j=1}^{\|T\|} num\ (j)$ 成立，因为左边表示所有需要专门计算距离的子树的节点数之和，相当于每个节点的计算次数之和。

由(1)，(2)和(3)可以得到

$$\sum_{i=1}^{m}\sum_{j=1}^{n} size(i) \times size(j)$$

$$= \sum_{i=1}^{m} size(i) \times \sum_{j=1}^{n} size(j)$$

$$= \sum_{i=1}^{\|T_1\|} num(i) \times \sum_{j=1}^{\|T_2\|} num(j)$$

$$\leq \|T_1\| \times \|T_2\| \times num(T_1) \times num(T_2)$$

$$\leq \|T_1\| \times \|T_2\| \times \min(d(T_1), l(T_1)) \times \min(d(T_2), l(T_2))$$

由于前处理和后处理部分都只需线性时间，综上所述，该算法的时间复杂度为 $O(\|T_1\| \times \|T_2\| \times \min(d(T_1), l(T_1)) \times \min(d(T_2), l(T_2)))$。

3.3 基于动态规划算法的自动性能评估实例

动态规划算法是解决树匹配问题的经典算法，在此我们以公式 $e^{-u^2/3}$ 的版式结构分析为例来说明该算法。图 3-8 是该公式的正确版式结构表达树，假设我们的系统对该公式的结构分析结果如图 3-9，即分析结果成为公式 $e^{-u^2/3}$ 的版式结构表达树。

令 T_1 和 T_2 分别表示公式的标准三叉树和实际结构分析所得的三叉树，输入 T_1 和 T_2，各步骤主要计算结果如下：

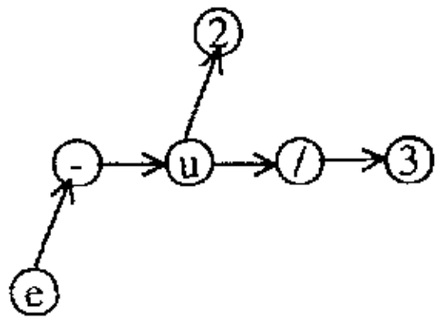


图 3-8：标准版式识别结果

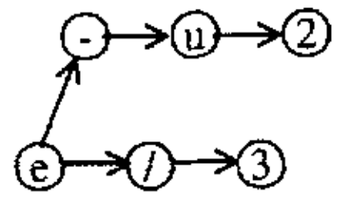


图 3-9：错误版式识别结果

(1) $LR\_keyroots(T_1) = \{3,6\}$;

$LR\_keyroots(T_2) = \{5,6\}$;

(2) 计算各个 *treedist*(*i*,*j*)时，相应的临时数组 *forestdist*(*l*(*i*)..*i*,*l*(*j*)..*j*)和最后所得的 *treedist* 数组的结果由以下各表给出：

forestdist(*l*(3)..3,*l*(5)..5)

	Ø	3	4	5
Ø	0	1	2	3
3	1	0	1	2

forestdist(*l*(3)..3,*l*(6)..6)

	Ø	1	2	3	4	5	6
Ø	0	1	2	3	4	5	6
3	1	1	2	2	3	4	5

forestdist(*l*(6)..6,*l*(5)..5)

	Ø	3	4	5
Ø	0	1	2	3
1	1	1	2	3
2	2	2	2	3
3	3	2	3	4
4	4	3	2	3
5	5	4	3	2
6	6	5	4	3

forestdist(*l*(6)..6,*l*(6)..6)

	Ø	1	2	3	4	5	6
Ø	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	0	1	2	3	4
3	3	2	1	0	1	2	3
4	4	3	2	1	2	3	4
5	5	4	3	2	3	4	4
6	6	5	4	3	4	5	4

treedist(*i*,*j*)

	1	2	3	4	5	6
1	0	1	1	2	3	5
2	1	0	2	2	3	4
3	1	2	0	1	2	5
4	3	2	3	2	3	4
5	4	3	4	3	2	4
6	5	4	5	4	3	4

← T_2 中的后根遍历序号

↑ T_1 中的后根遍历序号

(3) 根据 *treedist*(*i*,*j*)求得 T_1 和 T_2 之间的节点编辑距离为 *treedist*(6,6) = 4，节点编辑操

作序列为：删除 T_1 中第 5 个节点⑤，删除 T_1 中第 4 个节点ⓤ，在 T_2 中插入第 4 个节点ⓤ，在 T_2 中插入第 5 个节点⑤。

(4) 扫描所有的边，得到边ⓤ->②的属性值由 SUPERSCRIPT 变为 NEXT，即存在一个上标表达式分析错误。

第四章 基于BUTD算法的自动性能评估

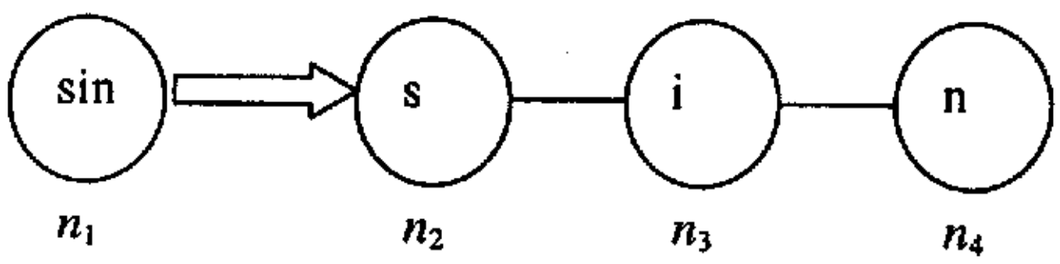
4.1 动态规划算法存在的实际问题

尽管动态规划算法对于我们的树匹配问题已经能够给出相应的距离，并且具有较好的时间复杂度，但是我们知道只有保证兄弟次序不变和祖孙次序不变的条件下才能得出 3.2.2 中引理 3 的距离公式，也因此才能使用动态规划算法来计算距离。

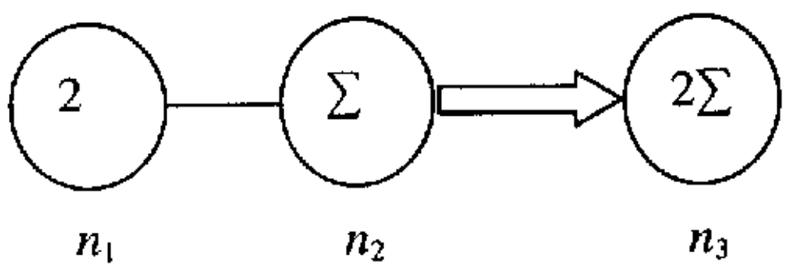
然而有时候不遵循这种条件约束反而更合理，我们仍然以公式$e^{-u^2/3}$的版式识别为例来说明这一问题。按照 3.3 动态规划算法，记录的错误共有 5 个，即先删除节点⊖和ⓤ，然后插入节点ⓤ和⊖，最后修改边ⓤ->②的属性值。如果我们在这里记录一个子树⑴→③从根结点ⓤ转移到ⓔ的错误，代替两个删除和两个插入错误，是不是更合理一些呢？答案是肯定的。但是转移根结点这样一种操作必然会打破祖孙次序不变的约束，而动态规划算法也就无能为力。

此外，作为系统输入的三叉树中每个节点应该代表一个有意义的数学符号或一个组元（如 sin, log, min, lim 等），即由多部分组成的符号如“≤”等以及组元“sin”等作为一个整体应该占据一个节点。如果出现将节点拆分的情形，如图 4－1(a)所示，将“sin”拆分成“s”，“i”，“n”三个节点，则动态规划算法将记录四个错误：删除节点 n_1，分别插入节点 n_2，n_3，n_4。。同理，对于多个节点被合并成一个节点的情况，如图 4-1(b)所示，动态规划算法将记录三个错误：删除节点 n_1 和 n_2，然后插入节点 n_3。实际上，前者是由于字符分割错误或分析过程中语义理解上的不足引起的，将语义上单一的符号划分成版式上的多个字符，而后者完全是由于字符分割错误造成的，为此，我们可以分别定义节点分裂操作和节点合并操作来表示这两种错误，这样只需记录一个错误即可。

因此，基于以上实际情况的考虑，我们又提出了另一种树匹配算法：BUTD 算法。



(a) 组元节点拆分示意图



(b) 节点合并示意图

图 4-1：节点拆分与合并

4.2 BUTD 算法

4.2.1 BUTD 算法概述

BUTD是Bottom-Up and Top-Down的缩写，因为在我们的树比较过程中采用了先自底向上，然后自顶向下的传递过程。该算法可以在最差时间复杂度为$O(n^2)$的情况下完成距离的计算。首先给出相关概念的定义如下：

(1). 距离度量

我们仍然用编辑距离来表示两棵树之间的差异，定义了八种基本编辑操作，对应着八种错误类型。在此我们不再考虑节点完全替代操作，除了定义节点内容修改，节点类型修改，节点插入，节点删除，边属性修改之外，增加了节点分裂操作，节点合并操作和节点转移操作，定义如下：

- 节点类型修改，Type（T_1,n,t_1,t_2）：修改树 T_1 中节点 n 的类型 t_1 为 t_2。
- 节点内容修改，Content(T_1,n,c_1,c_2)：修改树 T_1 中节点 n 的内容 c_1 为 c_2。
- 节点插入，Insert(T_2,n_1,m,n_2)：在树 T_2 的节点 n_1 的第 m 个子节点位置插入节点 n_2，原来 n_1 的第 m 个子节点变为 n_2 的子节点。
- 节点删除，Delete(T_1,n,m)：删除树 T_1 的节点 n 的第 m 个子节点，节点 n 的第 m 个子节点的子节点变为 n 的子节点。
- 节点分裂，Split(T_1,n,n_1...n_k)：指树 T_1 中的节点 n 分裂为 k 个节点 n_1...n_k。
- 节点合并，Merge((T_1,n_1...n_k,n)：指将树 T_1 中 k 个节点 n_1...n_k 合并为一个节点 n。
- 节点转移，Move(T_1,m,n,o,p,q)：即树 T_1 中节点 p 的第 m 个子节点 o，变成节点 q 的第 n 个子节点。
- 边属性修改，Edge(T_1,n_1,n_2,e)：修改树 T_1 中连接节点 n_1 和 n_2 的边的属性为 e。

我们赋予八种基本编辑操作的代价值都为单位值 1，则两棵树 T_1 和 T_2 之间的距离为：

$$\delta(T_1,T_1)=\min\{\gamma(E)\mid E \text{ 是将 } T_1 \text{ 转变为 } T_2 \text{ 的编辑操作序列}\}$$

(2). 节点的唯一 ID

由于数学公式中相同的符号可能会出现多次，为了使它们相互区别，给树中每个节点分配一个唯一的序号，称之为节点的 ID。这个 ID 可以通过很多种方式获得，例如对树进行先根遍历或后根遍历，得到所有节点的一个排序，将节点在该排序中的序号作为它的 ID 即可。在实际运用中，我们采用后根遍历的序号作为节点的 ID。

(3). 节点的权重值

由于树中每个节点并非都是同等重要的，有些节点的改变会对其他节点产生重要的影响，例如某个节点发生类型修改错误，则可能会导致其所连接的边属性的修改，而另一些节点的改变则可能只影响它本身。我们认为任何一个节点都比它的子节点重要，为了区分这些节点地位的不同，赋予每个节点一个权重值。对树中的每个节点 n，定义它的权重值 $weight(n)=1+sum(weight(children))$，即任意节点的权重值等于它所有子节点的权重之和加 1。为了便于计算，我们选择以 $T(i)$为根的子树的节点数作为节点 $T(i)$的权重值。在我们的 BUTD 算法中，首先需要计算树中所有节点的权重值，在树匹配过程中，每次都取当前权重值最大的节点加入队列中进行比较，因此最开始队列中必然只有根节点。

基于上述定义，BUTD 算法的主要匹配过程为：给定树 T_1 和 T_2，首先分别对 T_1 和 T_2 进行后根遍历，赋予每个节点一个唯一的 ID，同时计算每个节点的权重值；然后从 T_1 中选择权重值最大的节点 n 开始比较，如果在 T_2 中找到与之相匹配的节点 m，则比较它们的父节点，若也匹配，则继续比较父节点的其他子节点及所连接的各条边，否则从 T_1 中选择下一个权重值最大的节点开始比较，并记录相应的错误类型。对于每个节点都通过自底向上然后自顶向下这样一个比较过程，并且只比较一次。

4.2.2 BUTD 算法的详细描述

对于输入的两棵树 T_1 和 T_2，该算法主要步骤：

1. 后根遍历树 T_1 和 T_2，赋给每个节点一个唯一的 ID，即后根遍历序号，同时计算每个节点的权重。将遍历的结果分别存在数组 Array1 和 Array2 中，

2. 比较两棵树，同时记录各种分析错误，节点合并错误和节点分裂错误在该过程中并不记录，而是在后处理中分别对记录节点删除和插入错误的数组进行分析与归并得到。具体的比较流程如图 4-2 所示：

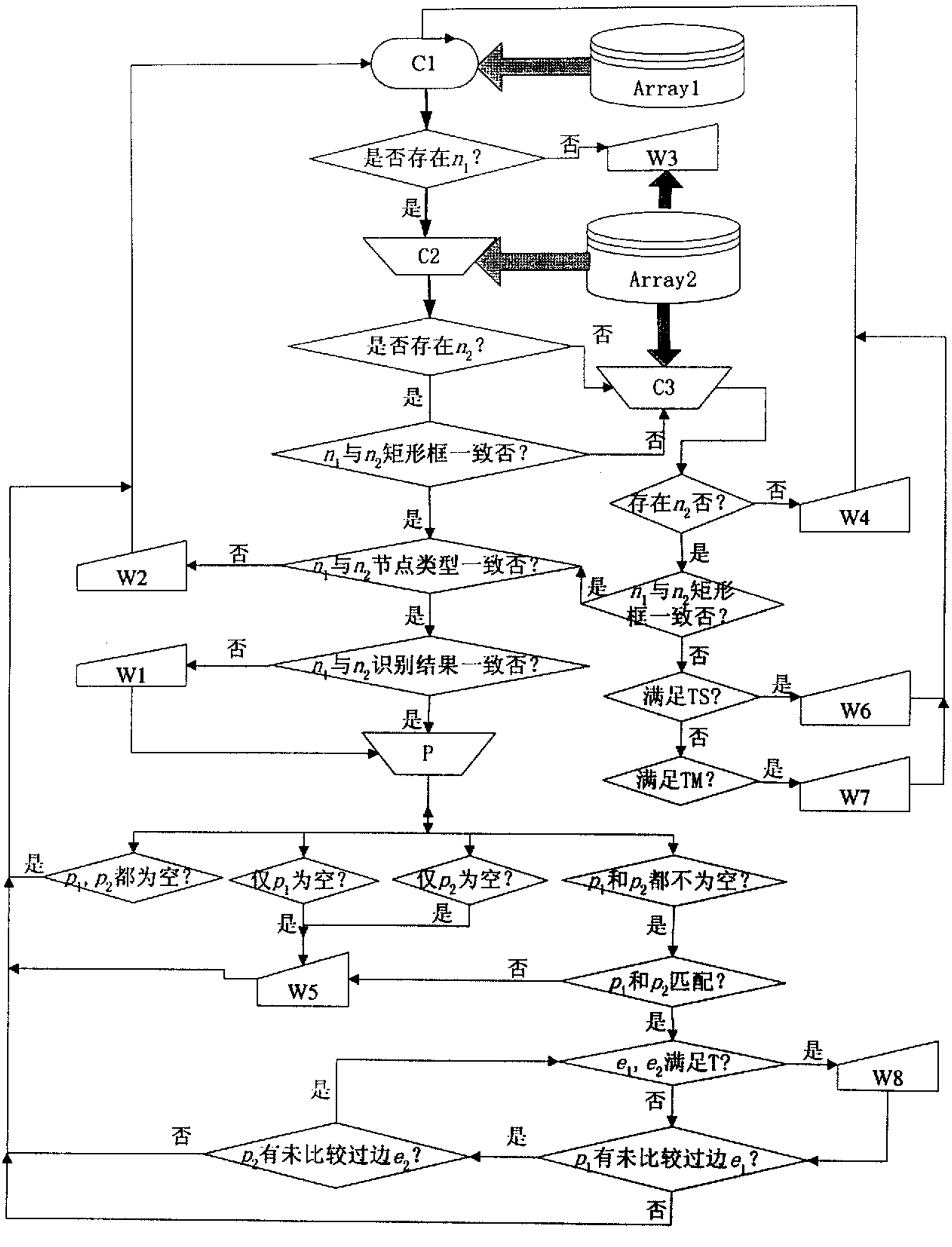


图 4-2：BUTD 算法流程图

其中一些符号定义如下：

C1：从 Array1 中选择权重值最大的尚未比较过的节点 n_1。

C2：从 Array2 中选择与 n_1 具有相同 ID 的节点 n_2。

C3：扫描 Array2，查找节点 n_2，或者与 n_1 的外接矩形框各坐标值相同，或者在 n_1 的外接矩形框内部，或者包含 n_1 的外接矩形框。

W1：记录一个节点内容修改错误，并将节点 n_1 和 n_2 标记为已比较过。

W2：记录一个节点类型修改错误，并将节点 n_1 和 n_2 标记为已比较过。

W3：将 Array2 中那些尚未参加比较的节点标记为已比较过，并且分别记录节点插入错误。

W4：记录一个节点删除错误，并将节点 n_1 标记为已比较过。

W5：记录一个节点移动错误，并将节点 n_1 和 n_2 标记为已比较过。

W6：记录一个节点分裂错误，并将节点 n_1 和 n_2 标记为已比较过。

W7：记录一个节点合并错误，并将节点 n_1 和 n_2 标记为已比较过。

W8：记录一个边属性修改错误，并将边标记为已比较过。

P：取 n_1 和 n_2 的父节点 p_1 和 p_2。

E：取 p_2 尚未比较过的边 e_2。

e_1 和 e_2：边 $p_1 \rightarrow n_1$ 和 $p_1 \rightarrow n_2$。

T：边属性修改条件，即边所连接的两个节点都分别匹配，而边的属性发生变化。

TS：节点分裂条件，即节点 n_2 在节点 n_1 的图像外接矩形框内。

TM：节点合并条件，即节点 n_1 在节点 n_2 的图像外接矩形框内。

比较过程描述如下：

step1：从 Array1 中选择权重值最大的尚未比较过的节点 n_1，若存在转 step2，否则转 step8；

step2：从 Array2 中选择与 n_1 的 ID 一致的节点 n_2，若 n_1 和 n_2 的图像外接矩形框一致，转 step4，否则转 step3；

step3：从 Array2 中搜索节点 n_2，若存在 n_2 与 n_1 的外接矩形框一致的，则转 step4；若 n_2 在节点 n_1 的外接矩形框内，则将 n_1 标记为已比较过，并记录一个节点分裂错误，转 step1；若 n_1 在节点 n_2 的外接矩形框内，则将 n_1 标记为已比较过，并记录一个节点合并错误，转 step1；若不存在 n_2 满足前面三个条件，则将 n_1 标记为已比较过，并记录一个节点删除错误，转 step1；

step4：比较 n_1 和 n_2 的公式类型，若一致则转 step5，否则将 n_1 和 n_2 标记为已比较过，并记录一个节点类型修改错误，转 step1；

setp5：比较 n_1 和 n_2 的识别结果，若一致则转 step6，否则将 n_1 和 n_2 标记为已比较过，并记录一个节点内容修改错误，转 step1；

step6：取 n_1 和 n_2 的父节点 p_1 和 p_2，若 $p_1 \neq \Lambda$ 同时 $p_2 \neq \Lambda$，转 step7，否则记录一个节点转

移错误，转 step1；

step7：比较 p_1 和 p_2 的各条边，记录相应的错误类型，转 step1；

step8：结束比较。

3. 将各种错误按八种编辑操作错误进行归类，并记录每种错误类型的数目以及所占的比例，同时统计各种公式类型的分析错误数目和错误率。将性能评估结果写入相应的结果文件中。

4.2.3 BUTD 算法的分析

我们用 n 表示两棵树的节点数之和，分析 BUTD 算法的时间和空间复杂度。

整个算法的时间包括：

1. 后根遍历两棵树同时计算权重值，所需的时间和空间都是线性的；

2. 选择当前权重值最大的节点加入优先级队列，最差情况是没有任何节点相匹配，因此所有节点都会加入到队列中，时间复杂度为 $n*log(n)$，空间复杂度仍然为线性的；

3. 比较过程中，最好情况是两棵树完全一致，所需时间等于每个节点和每条边分别比较一次，即$(n/2 + n/2 - 1) = n - 1$ 次比较；平均情况下相当于每个节点有 $n/4$ 次的外接矩形框比较和一次内容比较，每条边有一次比较，因此时间复杂度为 $O(n^2)$；最差情况是每个节点有 n 次外接矩形框比较，因此时间复杂度为 $O(n^2)$。

综上所述，在最佳情况下 BUTD 算法的时间复杂度为 $n*log(n)$，在平均和最差情况下时间复杂度都为 $O(n^2)$，空间复杂度是线性的。由于一般的数学公式，其总的符号数相对来说是挺小的，即 n 不会特别大（在测试中，最大的 n 为 280），因此即使是 $O(n^2)$的时间复杂度也是完全可以接受的。

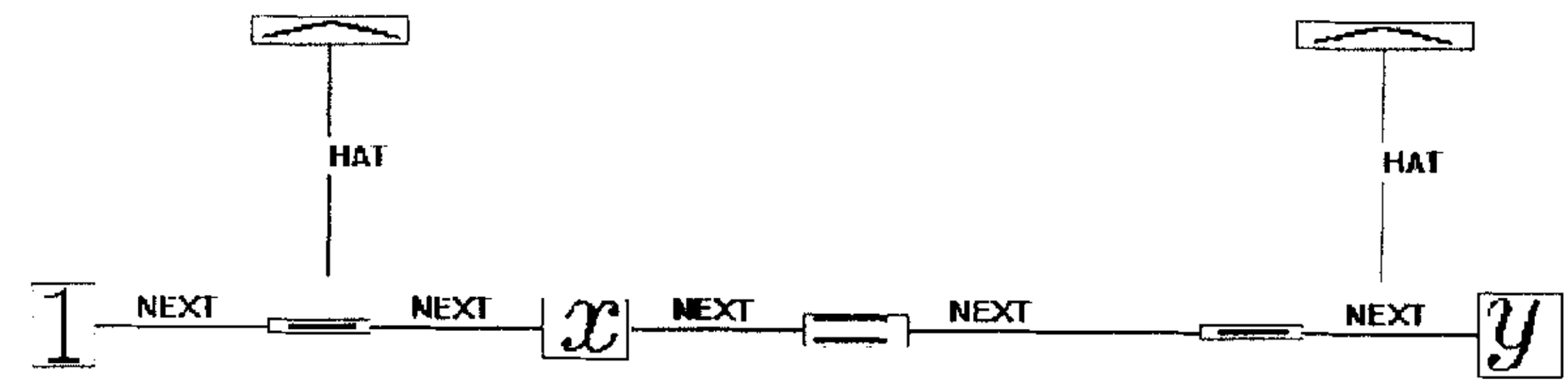
4.3 基于 BUTD 算法的自动性能评估实例

如图 4-4 所示，我们给出利用 BUTD 算法比较公式结构分析结果的例子。4-4(a)是测试的数学公式，4-4(b)是该公式的标准结构树，4-4(c)是系统实际分析结果。应用 BUTD 算法比较之后，得到各种编辑操作错误，我们将这些错误标识在标准结构树中，如图 4-4(d)所示。我们同时将各种错误类型的个数和变化情况记录在一个错误信息文件中，如图 4-4(e)所示。由文件记录可知，该公式分析共有 3 个错误：两个节点内容修改错误（“\widehat”变为“—”，“\widehat”变为“\overline”）和一个节点类型修改错误（帽子表达式变为角标表

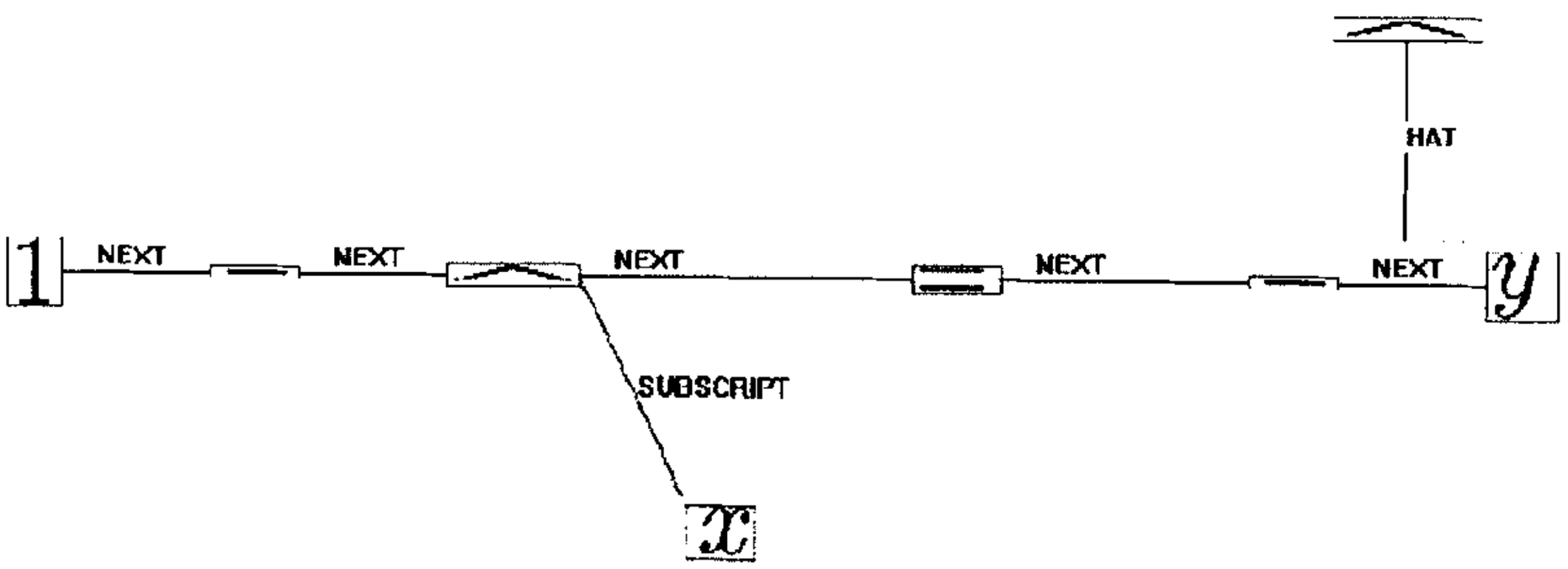
达式），分别对应着 4-4(d)中的 N1，N2 和 T1。

$$\widehat{1-x}=\widehat{-y}$$

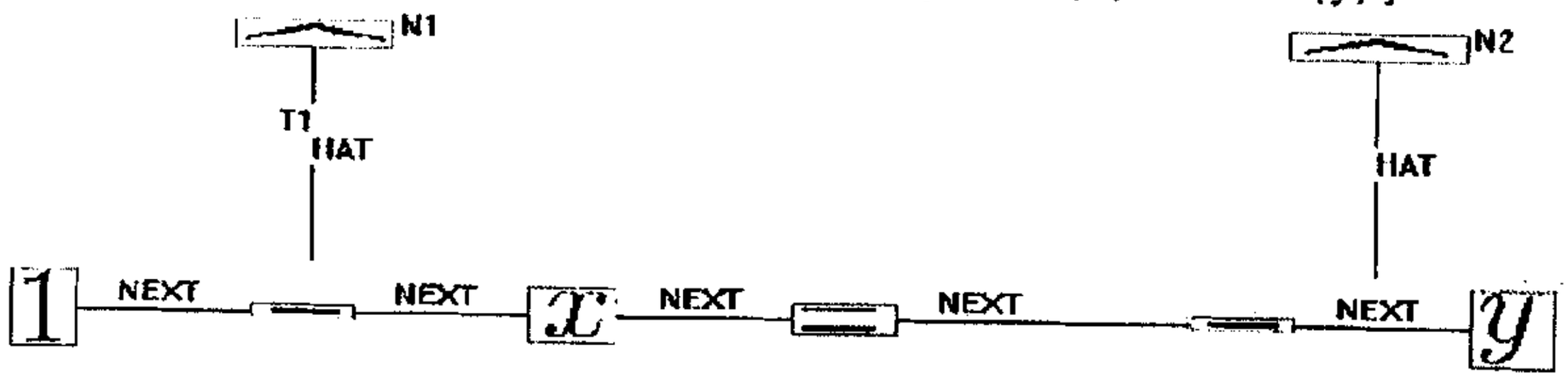
(a)测试公式



(b)标准分析结果：\[\widehat{1－x}=\widehat{－y}\]



(c)实际分析结果：\[1－－_{x}=\overline{y}\]



(d)错误示意图

```
The number of Node-changed error is 2:
1    Node \widehat-->-
2    Node \widehat-->\overline

The number of Type error is 1:
1    Node    HatExpression    changed into    ScriptExpression
```

(e)错误结果描述文件

图4-4：基于BUTD匹配算法的公式分析性能评估示例

4.4 动态规划算法与 BUTD 算法的比较

在数学公式分析阶段，我们提出了基于结构的反复分解的分析方法。由于每个公式经过分析之后都表示成一棵分解树，因此我们构造了一个基于树匹配的自动性能评估系统，以便自动评测这种分析方法的效果。

在我们的自动性能评估系统中，关键是要找到一种既合理又有效的树匹配算法。对于前文给出的两种树匹配算法，可以从合理性和效率两个方面进行比较。

首先，从合理性看，动态规划算法和 BUTD 算法都采用编辑距离来表示两棵树之间的差异，用该距离中出现的各种编辑操作个数来定量描述相应的分析错误。理论和实践都证明这种编辑距离表示方法是完全可行的。动态规划算法定义了节点完全替代，节点内容修改，节点类型修改，节点插入，节点删除，边属性修改六种基本编辑操作，其中后五种操作对应着五种错误类型。该算法能有效地计算出两棵树之间的最短距离，并给出此距离对应的操作序列，由于我们赋予节点完全替代操作很高的代价值，因此实际上该操作不会出现在最短距离操作序列中。针对 4.1 小节中提出的数学公式图像处理的一些实际问题，我们定义了节点内容修改，节点类型修改，节点插入，节点删除，节点分裂，节点合并，节点转移，边属性修改八种基本编辑操作，并给出了基于这些编辑操作的 BUTD 算法。相较于动态规划算法，BUTD 算法既全面反映了各种错误类型，又准确地指出了错误的本质原因，显然更符合我们的实际情况。

其次，从算法效率上看，根据 3.2.3 和 4.2.3 的分析，动态规划算法的时间和空间复杂度分别为 $O(\|T_1\|\times\|T_2\|\times\min(d(T_1),l(T_1))\times\min(d(T_2),l(T_2)))$ 和 $O(\|T_1\|*\|T_2\|)$，BUTD 算法的空间复杂度是线性的，在最佳情况下时间复杂度为 $n*log(n)$，在平均和最差情况下的时间复杂度为 $O(n^2)$。通过对大量数学公式的观察，发现一个特点：公式结构树中节点的个数不是很大，但是树的高度相对比较高。因此 BUTD 算法在时间和空间上都优于动态规划算法，尤其当我们的公式结构分析结果比较好时，这种优势更明显。

因此，无论从算法的合理性还是效率来看，BUTD 算法都比动态规划算法好。

第五章 实验结果与结论

5.1 系统实现环境与测试数据准备

我们的自动性能评估系统在 vc6.0 环境中开发，运行环境是赛扬 850。

测试数据是选自 LaTeX 手册和 IEEE 论文的 144 个数学公式。我们建立一个标准数据库存放这些数学公式的标准三叉树。

5.2 实验结果及分析

由于 BUTD 比动态规划算法更好，因此我们将给出基于 BUTD 算法的自动性能评估系统的实验结果。

如 2.3.2 所述，系统的输入带有边属性的三叉树结构。144 个数学公式的标准三叉树结构中总共包含 4566 个节点（其中最长的公式含有 140 个节点，最短的公式含有 3 个节点），4786 个数学符号，4405 条边。其中所包含的 11 种公式类型的数目统计结果如表 5-1 所示。

公式类型	一般表达式	多行表达式	分式表达式	根式表达式
数目	601	14	157	34
公式类型	定界表达式	矩阵表达式	组表达式	角标表达式
数目	230	15	70	559
公式类型	堆叠表达式	帽子表达式	基元表达式	总数
数目	3	25	3056	4764

表 5-1

用基于 BUTD 算法的自动性能评估系统对公式分析结果进行测试之后，可以给出八种编辑操作错误的数目，并且根据错误信息记录进行自动分析，统计出各种公式类型的错误情况。

下面我们给出 144 个数学公式两次结构分析结果的测试结果，为方便说明，将第一次分析结果简记为 α 版，第二次分析结果简记为 β 版，其中 β 版是根据 α 版的测试结果改进了公式分析方法之后所得的结果。

首先，通过对 α 版的测试，得到每个公式的错误信息报告，进行如下分析：

1．节点修改错误共有 130 个，这类错误反映了符号识别的准确率。若用(A，B)表示将符号 A 识成 B，则 α 版中实际出现的错误情况主要有(=，$\supseteq$)，(0，O)，($\int$，I)，(e，c)还有一

些希腊字符被识错，如(ω，w) ，(α，a)等。

2．节点类型错误共有 50 个，主要集中在组表达式，帽子表达式和矩阵表达式，具体分布见表 5-2，例如，组表达式被分析为一般表达式，则分别称组表达式和一般表达式为正确公式类型和错误公式类型。

正确公式类型	错误公式类型	错误数目
组表达式	角标表达式	3
组表达式	一般表达式	17
帽子表达式	角标表达式	10
帽子表达式	一般表达式	4
帽子表达式	多行表达式	4
矩阵表达式	角标表达式	9
	多行表达式	
	一般表达式	
堆叠表达式	帽子表达式	2
定界表达式	一般表达式	1

表 5-2

其中组表达式分析错误主要是由表达式中的大操作符（如“∫”）的识别错误引起的。帽子表达式分析错误主要是因为把“帽子”识成一般的符号所致。矩阵表达式比较特殊，通常一个矩阵因为被分割为多个子表达式而导致错误，即矩阵表达式类型会分析成多个不同表达式类型。

3．节点删除错误共 3 个，这是由于分析过程中丢失某些节点产生的。

4．节点插入错误共 2 个，都是由于图像中的噪音被当成了数学符号。

5．节点转移错误共 51 个，大致可以分为两类：一是某些角标表达式中的上标前移，这种情况属于角标表达式分析成了一般表达式；二是符号“...”及“,”被移到下标位置，属于一般表达式分析成了角标表达式，这主要是因为这类符号尺寸小，位置偏低，容易与下标混淆。

6．节点分裂错误共 65 个，可以分为两类：一是组元如“sin”，“max”等被分裂为多个字符，二种是由多个连通体构成的符号如“i”，“≤”，“‖”等被分裂为多个符号。这类错误是由于语义理解不足和字符分割错误造成的。由于在实际的分析结果中节点分裂直接导致

节点类型出基元表达式变为一般表达式，帽子表达式，堆叠表达式及分式表达式等，因此在后面的各种公式类型分析错误统计中将节点分裂错误归为基元表达式分析错误。

7．节点合并错误共 5 个，该错误是因为相邻两个字符之间的间距比较小而被切割为一个字符所致。例如 α 版中的一个错误是操作符“$\sum$”与前后的符号进行合并，是一般表达式转化为组表达式。

8．边属性修改主要发生在 NEXT 和 SUBSCRIPT 以及 NEXT 和 SUPERSCRIPT 之间，其实质是上/下标与同行关系即角标表达式与一般表达式之间的混淆，因此可以将这种错误归因于角标表达式与一般表达式的分析错误。在分析结果中实际出现了 72 个边属性修改错误，可以分为四类：①将上标判断为同行关系；②将同行关系判断为上标；③将下标判断为同行关系；④将同行关系判断为下标。各种边属性修改错误见表 5-3。

错误类别	①	②	③	④
错误数目	6	1	33	32

表 5-3

基于上述分析，我们对公式分析方法进行了多次改进，然后对 144 个数学公式进行分析得到 β 版，通过对 β 版的测试得到各种编辑操作错误如下：

1．节点修改错误共 64 个，主要有（0，O），（e，c），及一些希腊字符的识别错误。

2．节点类型错误共 20 个，具体分布情况见表 5-4。

正确公式类型	错误公式类型	错误数目
帽子表达式	角标表达式	10
帽子表达式	一般表达式	3
帽子表达式	多行表达式	2
堆叠表达式	帽子表达式	2
定界表达式	一般表达式	1
多行表达式	一般表达式	1
一般表达式	多行表达式	1

表 5—4

可以看出组表达式和矩阵表达式的处理已经得到改进，但是出现了个别多行表达式与一般表达式之间的分析错误。

3．节点删除错误共 3 个，这是由于分析过程中丢失某些节点产生的。

4. 节点插入错误共 2 个，都是由于图像中的噪音被当成了数学符号

5. 节点转移错误没有发生。

6. 节点分裂错误共 18 个，都是由多个连通体构成的符号被分裂为多个符号的情况。

7. 节点合并错误共 2 个，角标表达式被合并成了根式表达式。

8. 边属性修改共 5 个，其中 3 个是同行判为下标，2 个是上标判为同行。

由以上结果可以看出 β 版与 α 版相比主要存在以下区别：

（1）节点内容修改错误减少了一半，其中的=，∋)和(∫，∫)已经得到校正。

定义符号的识别率＝(正确识别的符号数 / 符号总数)*100%，则 α 版的识别率为 97.28%，β 版的识别率为 98.66%。

（2）节点类型修改错误也减少了一半，其中组表达式和矩阵表达式的分析也不再出错。

（3）节点转移错误不再出现。

（4）α 版中的组元节点分裂错误已经得到有效的解决。

（5）边属性修改错误大幅度下降。

从表 5-5 可以更清楚的看出这些差别。

错误类型	错误数目	
	α 版	β 版
节点内容修改	130	64
节点类型修改	50	20
节点删除	3	3
节点插入	2	2
节点分裂	65	18
节点合并	5	2
节点转移	51	0
边属性修改	72	5
总数	378	114

表 5-5

表 5-6 是我们将除了节点内容修改，节点删除和节点插入错误之外的五种错误按各种公式类型的分析错误进行归类得到的统计结果。

公式类型	总数	α版		β版	
		错误数目	准确率	错误数目	准确率
一般表达式	601	71	88.19%	4	99.33%
多行表达式	14	0	100%	1	92.85%
分式表达式	157	0	100%	0	100%
根式表达式	34	0	100%	0	100%
定界表达式	230	1	99.57	1	99.57%
矩阵表达式	15	9	40%	0	100%
组表达式	70	22	68.57%	0	100%
角标表达式	559	55	90.16%	4	99.28%
堆叠表达式	3	2	33.33%	2	33.33%
帽子表达式	25	18	28%	15	40%
基元表达式	3056	65	97.87%	18	99.41%

表 5-6

此外，我们的自动性能评估系统还进一步统计了表 5-7 的数据，其中完全正确的公式是指没有发生任何编辑操作错误的公式，结构分析正确的公式是指那些最多只包含节点内容修改错误的公式。

公式总数	完全正确的公式				结构分析正确的公式			
	α版		β版		α版		β版	
	数目	正确率	数目	正确率	数目	正确率	数目	正确率
144	29	20.14%	94	65.28%	48	33.33%	120	83.33%

表 5-7

表 5-6 和表 5-7 的统计数据表明改进后的公式结构分析方法在性能上有了很大的提高。

5.3 错误示例

为了对各种错误类型有个直观的概念，下面按公式类型给出测试过程中出现的部分错误示例。由于某些公式比较长，因此只截取其中分析错误的部分子表达式，或者只给出公式本身而不显示其分析结果。

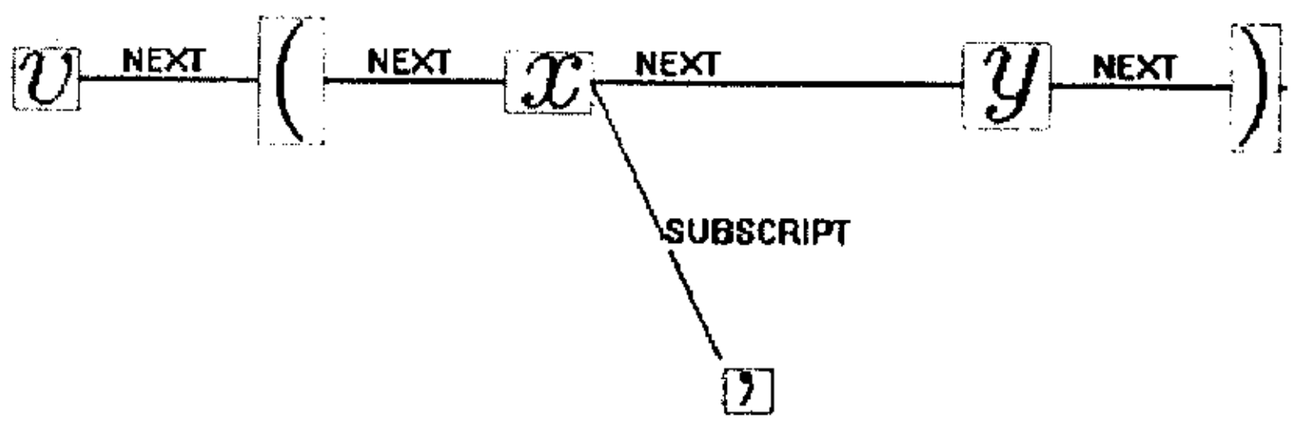
（1）一般表达式分析错误通常是边属性由同行判断为上标或下标所致，其中以图 5-1 所示的错误为主。另外，在β版中出现了将一般表达式分析成多行表达式的错误，如图 5-2

所示的公式。

$$v(x,y)$$

(a) 测试子表达式

(b) 标准分析结果



(c) 实际分析结果

图 5-1：一般表达式分析为角标表达式示例

$$\frac{\partial^2 U}{\partial x^2}+\frac{\partial^2 U}{\partial y^2}=0 \Longrightarrow U_M=\frac{1}{4\pi}\oint_{\Sigma}\frac{1}{r}\frac{\partial U}{\partial n}\,ds-\frac{1}{4\pi}\oint_{\Sigma}\frac{\partial \frac{1}{r}}{\partial n}U\,ds$$

(a) 测试公式：一般表达式

$$\frac{\partial^2 U}{\partial x^2}+\frac{\partial^2 U}{\partial y^2}=0 \Longrightarrow U_M=\frac{1}{4\pi}\oint_{\Sigma}\frac{1}{r}\frac{\partial U}{\partial n}\,ds-\frac{1}{4\pi}\oint_{\Sigma}\frac{\partial \frac{1}{r}}{\partial n}U\,ds$$

(b) 实际分析结果：多行表达式

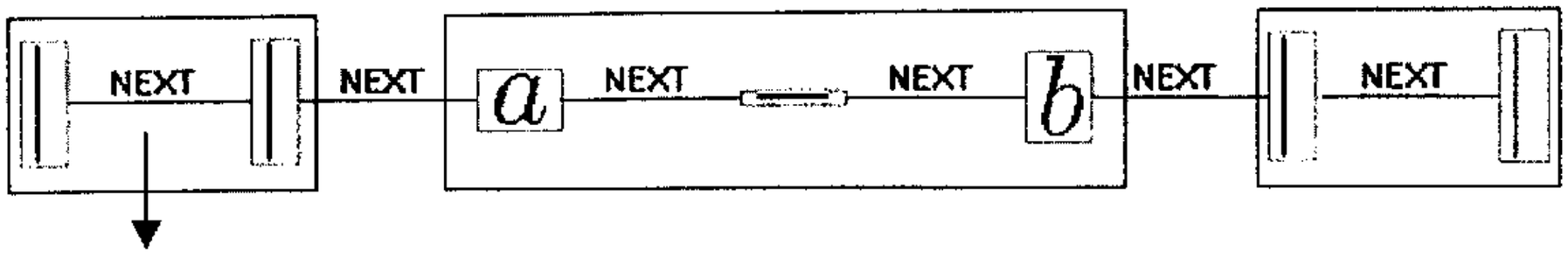
图 5-2：一般表达式分析为多行表达式示例

（2）定界表达式分析错误如图 5-3 所示，定界符“‖”被分裂，使基元表达式成了定界表达式，而整个定界表达式成了一般表达式。

$$h(\mathcal{A},\mathcal{B})=\max_{a\in\mathcal{A}}\min_{b\in\mathcal{B}}\parallel a-b\parallel$$

(a) 测试公式

(b) 定界表达式部分标准分析结果



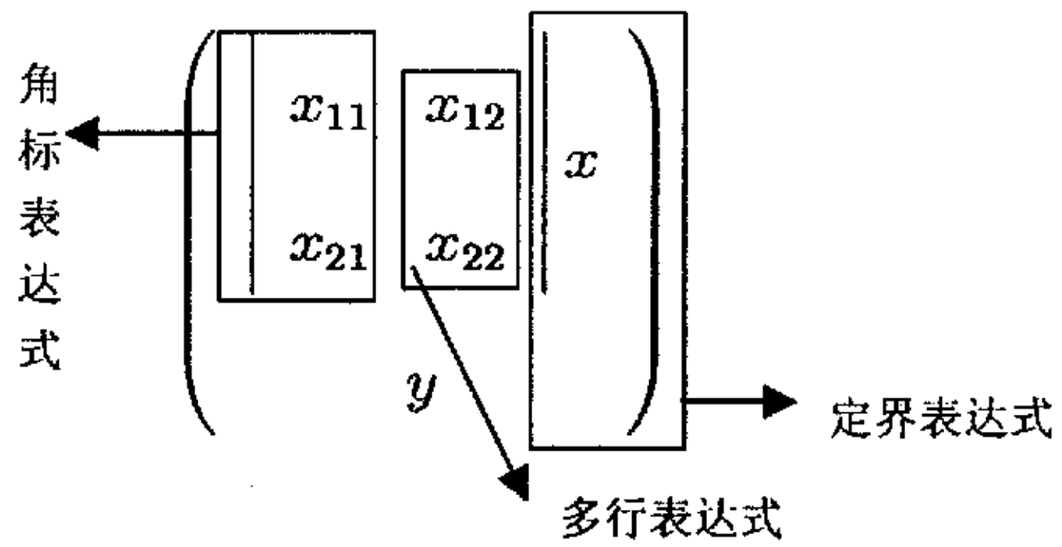
(c) 实际分析结果

图 5-3：定界表达式分析为一般表达式示例

（3）矩阵表达式分析错误在 α 版中出现，如图 5-4 所示。

$$\left(\begin{vmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{vmatrix} x \right)$$
$$y$$

(a) 测试公式



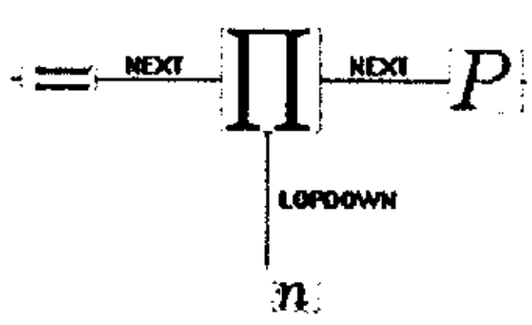
(b) 实际分析结果

图 5-4：矩阵表达式分析错误示例

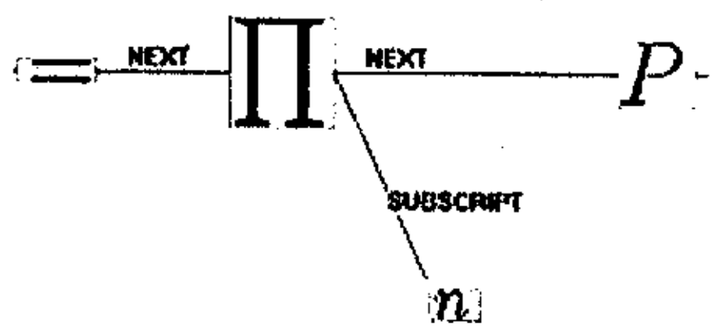
（4）组表达式分析错误主要分为两类，一是分析为角标表达式，二是因为将大操作符“∫”识成“I”而分析成一般表达式，如图 5-5 和图 5-6 所示所示。

$$P(G_i|S_j) = \prod_n P(G_i(n)|S_j(n))$$

(a) 测试公式



(b) 组表达式标准分析结果

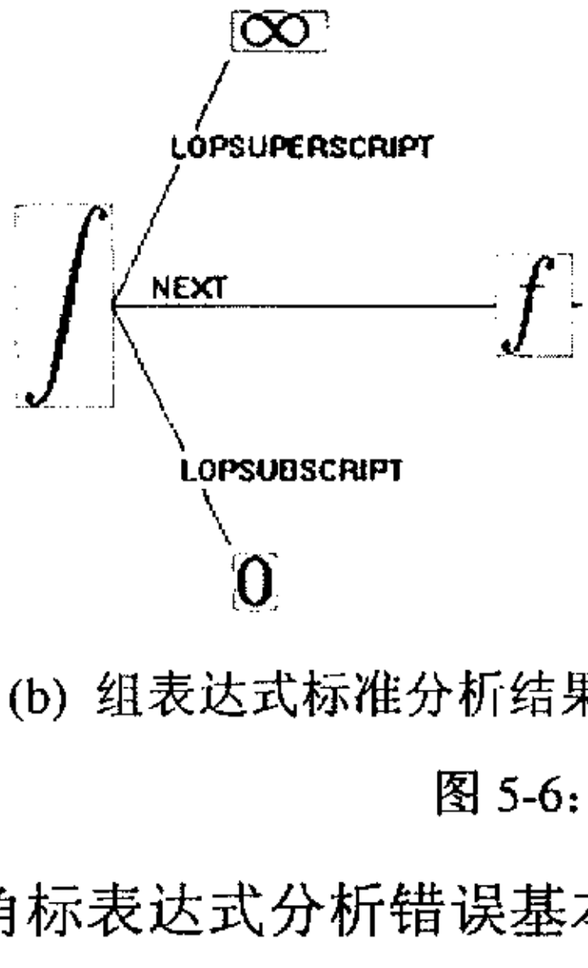


(c) 实际分析结果

图 5-5：组表达式分析为角标表达式

$$\int_0^{\infty} f(x)\,dx \approx \sum_{i=1}^{n} w_i c^{x_i} f(x_i)$$

(a) 测试公式



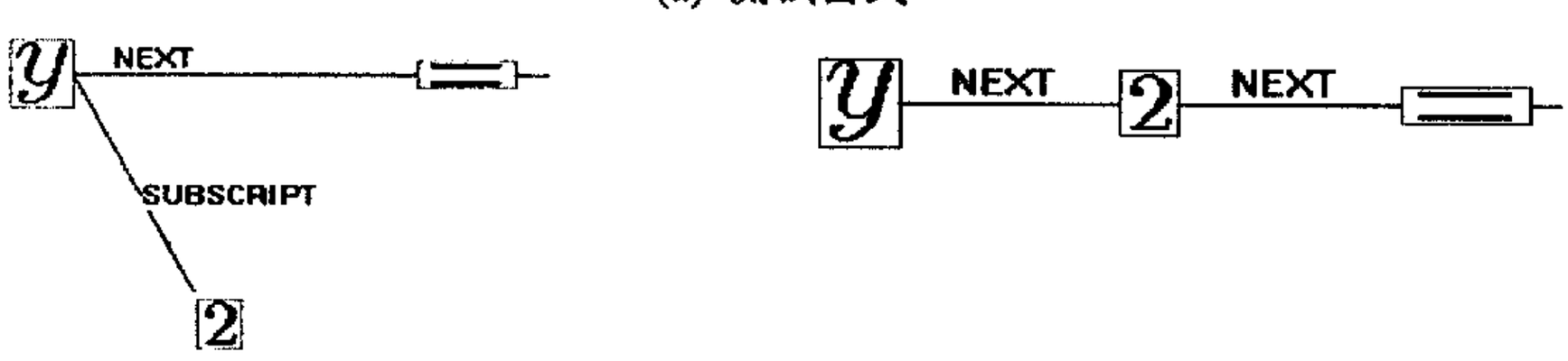
(b) 组表达式标准分析结果　　(c) 实际分析结果

图 5-6：组表达式分析为一般表达式

（5）角标表达式分析错误基本上都是边属性由上标或下标判断为同行所致，如图 5-7。

$$y_2 = -\frac{u+v}{2} + \frac{i}{2}\sqrt{3}(u-v)$$

(a) 测试公式



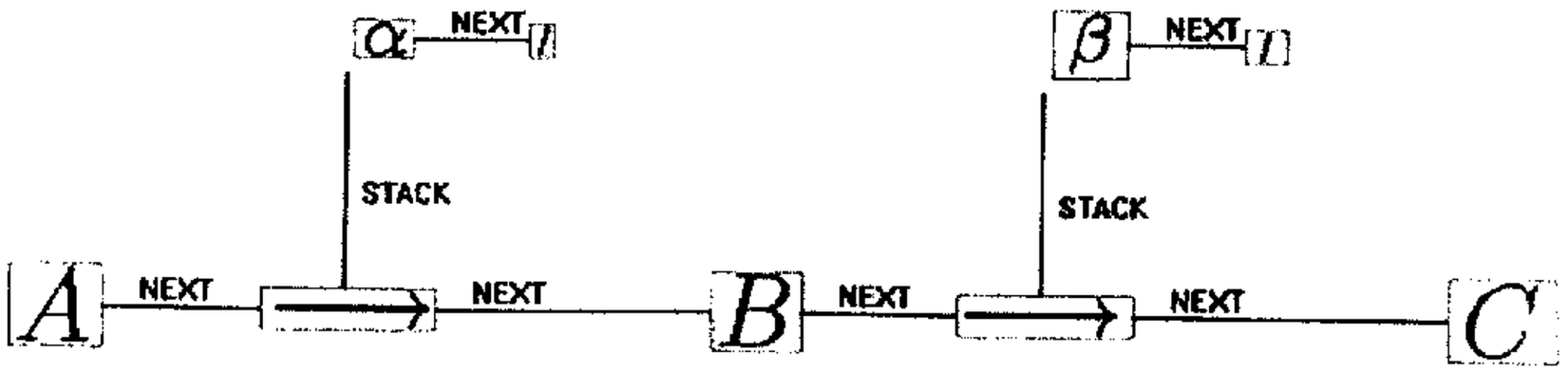
(b) 角标表达式标准分析结果　　(c) 实际分析结果

图 5-7：角标表达式分析为一般表达式

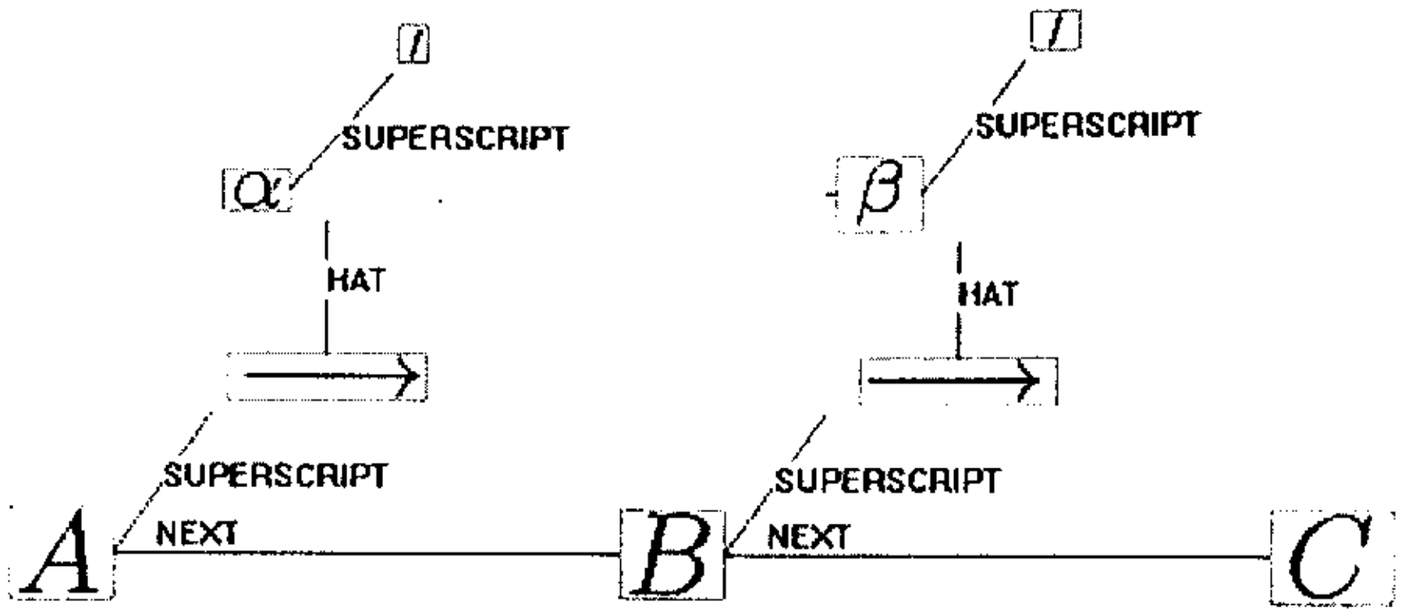
（6）堆叠表达式与帽子表达式形状有些类似，所以容易被当作帽子表达式分析，如图 5-8，包含了堆叠表达式与一般表达式的分析错误。

$$A \xrightarrow{\alpha'} B \xrightarrow{\beta'} C$$

(a) 测试公式



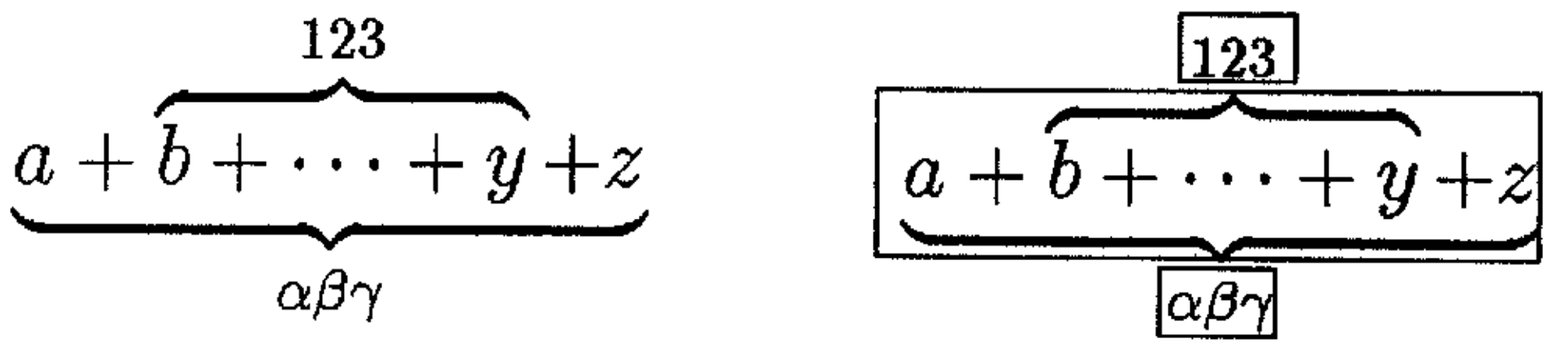
(b)公式标准分析结果



(c)实际分析结果

图 5-8：堆叠表达式分析为帽子表达式示例

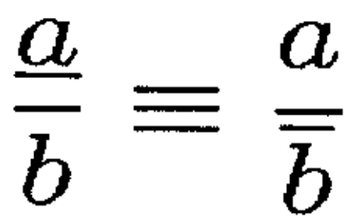
（7）帽子表达式错误主要分为两类，一是被分析为角标表达式（如图 4-4 所示），二是被分析为多行表达式，如图 5-9 所示。



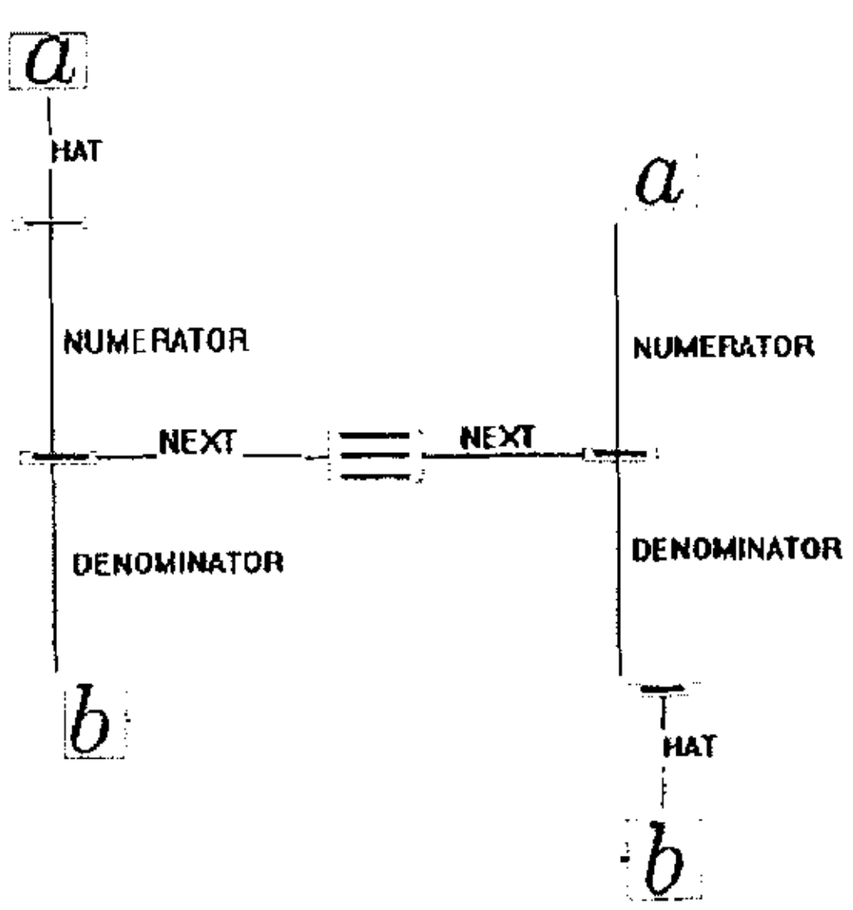
(a) 测试公式　　(b) 实际分析结果：成了三行表达式

图 5-9：帽子表达式分析为多行表达式

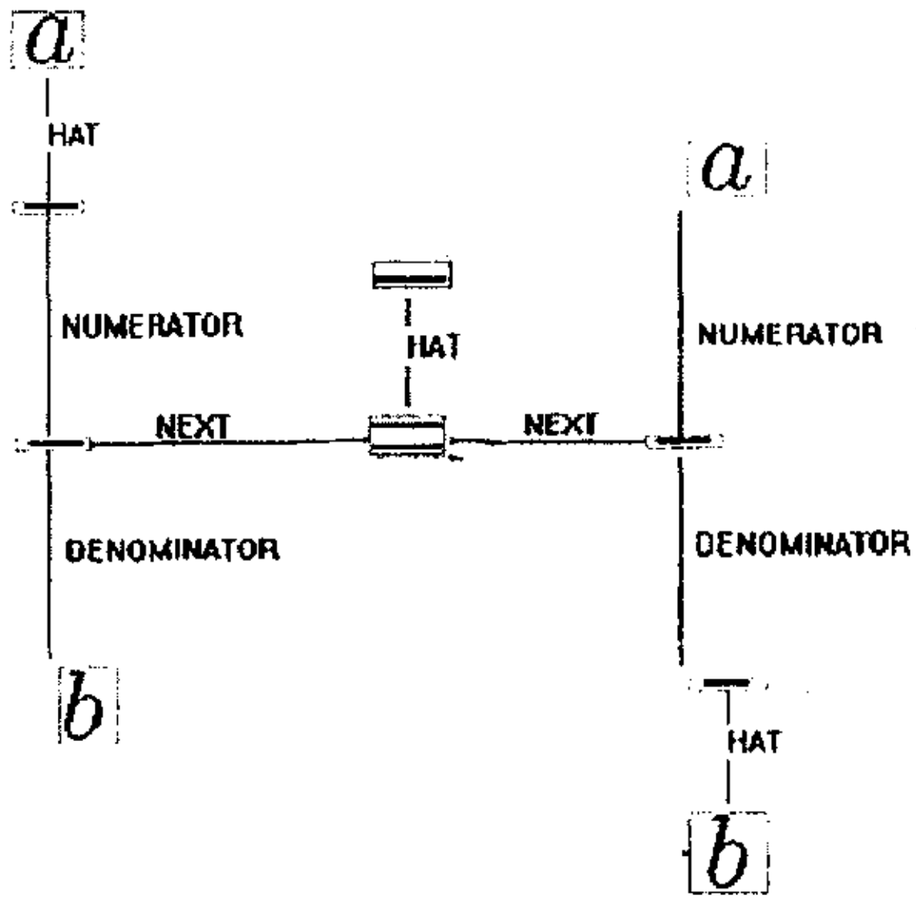
（8）基元表达式分析错误是由节点分裂或节点合并产生的，图 5-3 就是基元表达式分析错误的一个实际例子。此外，基元表达式还被分析为一般表达式，帽子表达式，堆叠表达式等等。如图 5-10 所示，“≡”被分裂为“—”和“=”两部分，使基元表达式成了帽子表达式，这是字符切割时所造成的错误。



(a) 测试公式



(b) 公式标准分析结果



(c) 实际分析结果

图 5-10：基元表达式分析成帽子表达式

5.4 结论

基于 BUTD 算法的自动性能评估系统不仅能够给出公式分析结果的错误类型，还能判断出错误的原因是属于符号切割，符号识别还是结构分析问题，从而指导设计人员对相应处理阶段的方法进行改进，使整个系统不断完善，这种辅助作用从上述的实验结果即可见一斑。

虽然八种基本编辑操作已经能够比较准确地描述各种错误类型，但是实际的分析结果总会出现意想不到的情况，以至于就算是我们人本身也未必能非常精确地判断某些错误类型。如图 5-11 所示，其中子表达式a_i^k,中的“i”被分裂为“.”和“\imath”两部分，同时“.”与其后的“,”被合并成一个符号“\rceil”。同样的，子表达式b_j^k,中的“j”也被分裂为两部分“.”和“\jmath”，同时“.”与其后的“,”被合并成一个符号“\rceil”。对于这种由符号切割引起的错误，从编辑操作的角度来看，我们可以认为存在节点分裂与节点合并错误，但是从公式类型的角度就很难说清楚是哪种表达式变成了哪种表达式。

$$f(a_i^k, a_{i+1}^k, b_j^k, b_{j+1}^k)x_{ij}x_{i+1\,j+1} = 0, i = 1, 2, \ldots, m-1, j = 1, 2, \ldots, n-1$$

(a) 测试公式

(b) 节点合并与分裂

图 5-11：节点合并与分裂错误示例

因此，对于少数类似的错误类型，还需更精确地唯一表示每一种错误情况。但是从找出错误原因以便改进系统性能的角度来说，BUTD 算法完全能够满足要求。

参考文献

[1]R.H.Anderson."Syntax-directed Recognition of Hand-printed Two-dimensional Mathematics", *Interactive Systems for Experimental Applied Mathematics*, Academic Press, New York, 1968.

[2]R.H.Anderson. "Syntax-directed Recognition of Hand-printed Two-dimensional Mathematics". *Ph.D. Dissertation*, Department of Engineering and Applied Physics, Harvard University, Cambridge, Mass., Jan, 1968.

[3]S.K.Chang. "A Method for the Structural Analysis of Two-Dimensional Mathematical Expressions". *Information Sciences* Vol. 2(3), pp. 253-272, 1970.

[4]R.H.Anderson. "Two-dimensional Mathematical Notations". I*n Syntactic Pattern Recognition Applications* (K. S. FU, Ed.), Springer-Verlag, New York, pp. 147-177, 1977.

[5]Abdelwaheb Belaid and Jean-Paul Haton. "A Syntactic Approach for Handwritten Mathematical Formula Recognition". *IEEE Transaction on Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, No. 1, pp. 105-111, January, 1984.

[6]Zi-Xiong Wang and Claudie Faure. "Structural analysis of handwritten mahematical expressions". *Proceedings of 9th International Conference on Pattern Recognition*, ICPR'88, Rome, Italy, pp. 32-34, 1988.

[7]P. A. Chou. "Recognition of Equations Using a Two-Dimensional Stochastic Context-Free Grammar". *Proceedings of SPIE Conference on Visual Communicationa and Image Processing* IV, Philadelphia, PA, Vol. 1199, pp. 852-863, Nov 1989.

[8]Claudie Faure and Zi-Xiong Wang. "Automatic Perception of the Structure of Handwritten Mathematical Expressions". *In Computer Processing of Handwritting*. World Scientific Publishing Co., Singapore, pp. 337-361. 1990.

[9] K.C.Tai, "The tree-to-tree correction problem", J. ACM, 26(3), 422-433, 1979.

[10]R.Wilhelm,"A modified tree-to-tree correction problem",*Information processing letters* 12(2), pp.127-132,1981

[11]Masayuki Okamoto, Hiroki Imai and Kazuhiko Takagi. "Performance Evaluation of a Robust Method for Mathematical Expression Recognition". *Proceedings of 6th International Conference on Document Analysis and Recognition*, ICDAR'01, September 10-13, 2001, Seattle, Washington, USA, pp. 121-128, 2001.

[12]K. F. Chan and D. Y. Yeung. "Error detection, error correction and performance evaluation in on-line mathematical expression recognition". *Pattern Recognition*, Vol. 34(8), pp. 1671-1684, August 2001.

[13]K.Zhang and D.Shasha. "Simple fast algorithms for the editing distance between trees and related problemes".*Siam J.Comput.*18, No.6, pp. 1245-1262, 1989.

[14]V.F.Märger,P.Karcher,A.-K.Pawlowski. "On Benchmarking of Document Analysis Systems", *Proc. of the 4th Int. Conf. on Document Analysis and Recognition*, IEEE Computer Society: Los Alamitos, CA. pp. 331-336,1997.

致谢

衷心感谢我的导师王庆人先生。大三时，他的一次精彩讲座使我对模式识别产生了好奇，并且决定踏入这个神奇的领域。在攻读研究生期间，他深邃的思想，独特的视角引领我叩开科学研究之门，他严谨务实的治学态度鞭策我勤奋学习，他积极乐观的人生信念激发了我对生活的热爱。

衷心感谢韩维桓先生。他在学业上的指导和工作中的诸多建议使我豁然开朗。他的“德、识、才、学”的教诲指明了我前进的方向，让我终身受益。

衷心感谢许静副教授给予我的悉心指导和帮助。

衷心感谢宋洪生老师。他为智能所鞠躬尽瘁，对每位同学给予了无微不至的照顾和关心，使我们能够全身心投入学习和科研。

感谢师兄靳简明博士。在三年的学习、工作和生活中，无数次毫无保留的交流与帮助使我受益匪浅。他学术上的刻苦专研，工作上的踏实认真为我树立了良好的榜样，留下了深刻的印象。

感谢师兄潘武模和史广顺博士对我在各方面的指导和帮助。

感谢同学程蕾、罗华、林乐健、于鹏、杨冰、郑德斌、闫会强、艾涛、吴松涛、蒋学。感谢他们给予我学习和工作上的帮助和交流，给予我生活上的照顾与关心。

感谢师弟师妹们给我带来无穷的欢乐。

感谢所有关心和支持我的人。

特别地，谨以此文献给我的父母，衷心感谢他们含辛茹苦的养育之恩，严格细致的教诲，无时无刻的支持与关心。

江红英