**细粒度的精确的源代码差分**

**摘要：**软件演化的核心是对源代码文件进行的一系列编辑操作，称为编辑脚本。由于软件系统按版本存储，编辑脚本必须从这些版本计算得出，这是一个复杂的任务。现有的方法通常只用添加行和删除行来计算文本粒度的编辑脚本。然而，从这样的编辑脚本推断语法上的变化是很困难的。由于移动代码是编辑代码时常有的操作，因此也应该考虑进去。在本文中，我们通过引入一种算法，以抽象语法树粒度（包括移动动作）来计算编辑脚本来解决这些问题。我们的目标是计算编辑脚本，这些编辑脚本很短，而且与原始开发人员的意图相近。我们的算法是在一个经过深入验证的免费可用和可扩展的工具中实现的。

**关键词：**软件演化, 程序理解, 树差分, 抽象语法树

1. **介绍**

软件演化的第一法则就是几乎所有的软件系统都必须演化到令人满意[19]。自从制定这一法则以来，已经进行了许多相关研究，以更好地了解软件系统的演化，并形成了软件演化这一研究领域[21]。

软件演化分为全局软件演化（例如需求，执行环境的演化，...）和本地软件演化（源代码文件的演化）。在本文中，我们关注后者，也就是了解源代码文件的演化。特别是，我们专注于编辑脚本，这是编辑动作到源代码文件的序列。通常，由于软件存储在版本控制系统中，因此在同一文件的两个版本之间计算编辑脚本。编辑脚本的目标是准确反映对文件的实际更改。

编辑脚本是提供给开发人员日常使用的。例如，Unix diff工具将两个版本的源代码文件作为输入，并以文本行粒度执行Myers算法[24]，并返回一个编辑脚本，指示哪些行已添加或删除。然而，差异的局限性是双重的。首先，它仅计算添加和删除，并不考虑其他类型的编辑操作，如更新和移动。其次，它的粒度（文本行）既粗糙又不与源代码结构对齐：抽象语法树）。

为了克服这个主要问题，有一些算法可以用在抽象语法树（AST）层面上[13]。使用AST粒度的主要优点是编辑脚本直接引用代码的结构。例如，如果编辑操作是添加一个新的功能节点，这显然意味着在代码中添加了一个新功能。尽管有一些的主要文献（例如[13]），计算AST编辑脚本的问题仍然是开放的，存在两个主要挑战：处理移动动作，并扩展到具有数千个节点的细粒度AST。这也是本文作出贡献的地方。

为了设计出一个新颖的算法，我们考虑开发者的观点：对理论上最短的编辑脚本从不感兴趣。她对反映发生的实际变化的编辑脚本感兴趣。因此，我们的目标不是找到两个版本之间最短的动作序列，而是反映开发者意图的序列。因此，我们设计了一种基于启发式的算法，其中包含了关于什么是好的编辑脚本的务实规则，并且很重要的一点是，这是高效的，并可扩展到大型AST。该算法已经在可自由使用和可扩展的工具中实现。

总之，我们的贡献如下：

* 一种考虑到移动动作及其实现的新型高效AST差分算法;
* 对实际数据的实施性能进行自动评估;
* 通过手动评估144个差异场景，对算法的结果进行手动评估;
* 对12，792个差异场景的大规模自动评估，表明我们的算法的结果比相关工作更准确，即使是细粒度的AST。

本文的其余部分结构如下：第2节介绍什么是AST差异。第3节描述了我们的新的AST差分算法。第4节介绍了实现这一新算法及其性能的工具。第5节介绍了我们工具的经验评估。 第6节介绍相关工作。最后，第7节总结并介绍了未来的工作。

1. **抽象语法树差分**

在介绍AST差分之前，我们简要介绍定义AST结构的主要概念。我们认为AST是一个带有标记的有根树，其中节点可能有一个字符串值。节点标签对应于语法中其生产规则的名称，即编码结构。节点的值对应于代码中的实际符号。

更正式地，设*T*为一个AST。*T*是一系列节点的集合。树*T*有一个根节点（用*root（T）*表示）。每一个节点*t∈T*都有一个父节点*p∈T∪*Ф。唯一一个父节点为Ф的是根节点。节点*t*的父节点用*parent（t）*表示。每个节点*t∈T*都有孩子节点（*children（t）*）。每个节点都有一个标签用字母*l∈∑*表示（*label（t）=l*）。每个节点有一个字符串值*v∈String*，有可能为空。（value(t)=v∨ε）。

例如，我们考虑一个简单的Java源代码及其相应的AST（见图1左下角）。该Java源代码的AST包含19个对应于Java编程语言结构的节点。因此，AST的每个节点都有一个标签，它映射到源代码的结构元素（例如MethodDeclaration或NumberLiteral），以及与代码中的实际符号（例如与1相关联的NumberLiteral）的值。一些值不对信息进行编码，因此被丢弃，例如MethodDeclaration没有与之相关联的符号，因此没有值。

AST可以具有不同的粒度，节点可以编码整个指令或更精细的粒子表达式。我们相信，细粒度的AST最适合开发者。例如返回“Foo！”; 语句，可以使用一个类型为Statement的单个节点进行编码，并返回“Foo！”，或者在我们的示例中使用两个节点（参见节点b）。如果这个声明改为返回“Foo！” + i;只有细粒度的表示可以看到添加了InfixExpression：+和SimpleName：i节点。

AST差分基于AST编辑动作的概念。它旨在计算将AST转换为另一个的一系列编辑动作。 此序列称为编辑脚本。关于我们对AST的定义，我们考虑以下编辑操作：

* *updateValue(t,vn)*表示用旧的值t替代新的值vn
* *add(t,tp,I,l,v)*表示在AST中添加一个新的节点*t*如果*tp*非空而且指定了*i*，那么*t*是*tp*的第*i*个孩子。否则*t*是一个新的根节点，并且有先前的根节点作为它的唯一孩子。最后，*l*是*t*的标签，*v*是*t*的值。
* *delete(t)*移除AST的一个叶子结点*t*。
* *move(t,tp,i)*移动一个节点*t*使其成为*tp*的第*i*个孩子。注意，*t*的所有孩子都被移动，所以这个操作移动了整个子树。

由于有许多执行相同的转换的编辑脚本有很多，编辑脚本的质量取决于它的长度：变换越短，质量越好。注意，考虑到移动动作时，找到最短的变换是NP难的。

在本文中，我们考虑AST差分问题有两个AST作为输入，目的在于识别将第一个AST（t1）转换为第二个（t2）的编辑动作（包括移动）的短编辑脚本σ。执行这种AST差分的现有算法使用启发式方法返回一个短的编辑脚本σ。而且，他们通常会采取两个步骤。首先，它们在两个AST的相似节点之间建立映射（节点对）。这些映射有两个约束：给定节点只能属于一个映射，映射涉及两个具有相同标签的节点。第二，基于这些映射，他们推导出必须在第一个AST上执行的编辑脚本来获得第二个。第一步是最重要的，因为第二步存在二次最优算法[6,15]。在下一节中，我们将介绍一种新算法来计算两个AST之间的映射。

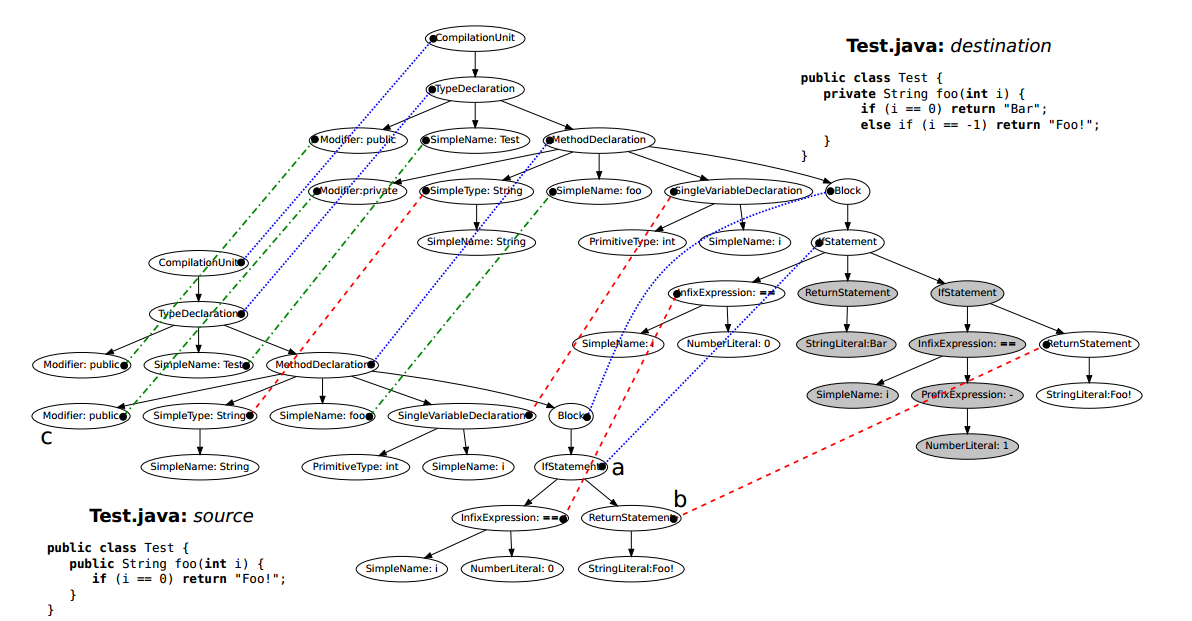
1. **GUMTREE算法**

如上一节所述，AST差分算法分为两个步骤：建立映射，然后推导编辑脚本。由于第二步已经开发了一个二次最优化算法[6]，我们只在本节中解释我们如何寻找两个AST之间的映射。该算法的输出可以由Chawathe等人的算法[6]用于计算实际的编辑脚本。我们用于计算两个AST之间的映射的算法由两个连续的阶段组成：

* 一个贪心的自上而下的算法，找到减少高度的同构子树。在这些同构子树的节点之间建立连接。它们被称为锚点映射。
* 自下而上算法中，如果两个节点的后代（节点的子节点及其子节点等）包含大量公共锚点，则两个节点匹配（称为容器映射）。当两个节点匹配时，我们最终应用最优算法来搜索其后代中的其他映射（称为恢复映射）。

该算法的灵感来自于开发人员手动查看文件之间的更改。首先，他们搜索最大的未修改的代码段。然后他们推断哪些容器的代码可以映射到一起。最后，他们看看每个容器剩下部分的精确差异。为了更好地说明我们的算法，我们将介绍如图1所示的示例。

**3.1 自上而下阶段**

1. **工具**
2. **评估**
3. **相关工作**
4. **结论和未来工作**
5. **引用**