# 在 GP 2 中实现二叉搜索树（扩展摘要）

ZiadIsmaili Alaoui Detlef Plump利物浦大学，英国 约克大学，英国

## 1 引言

我们介绍一种在基于规则的图编程语言 GP 2 中实现二叉搜索树的方法。（关于 GP 2 的简要介绍，请参见 [4]。）我们的实现利用 GP 2 的根节点图转换规则来提高速度 [1]，并支持插入、删除和查询操作。我们认为，对于包含 $n$ 个节点的树，每种操作的最坏情况运行时间均为 $O(n)$。此外，我们预计这些操作的平均情况运行时间为 $O(\log n)$[1]。因此，该实现的时间复杂度应与命令式语言中二叉搜索树实现的时间复杂度相当（例如，参见 [6]）。

## 2 二叉搜索树

二叉搜索树（BST）是一种二叉树，其中每个节点都标有一个不同的键（即整数值）。对于每个节点$v$，其左子树中的所有键都严格小于$v$的键，而其右子树中的所有键都严格大于$v$的键。[2] 图 1 展示了一个包含 6 个节点的二叉搜索树示例。二叉搜索树通常支持三种操作：插入、查询、

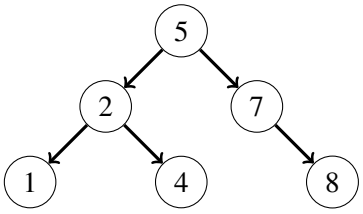


图 1：二叉搜索树示例。

以及删除。下面将简要说明这些操作的工作方式。

**插入**。 向二叉搜索树中插入节点时，必须保持其性质不变。要插入一个新键，我们从根节点开始递归遍历这棵树：在每个节点处，如果新键小于或等于当前节点的键，就移至左子节点；如果更大，则移至右子节点。这个过程会持续进行，直到到达空子节点位置，并在此处插入新节点。这样可以确保插入后仍满足二叉搜索树性质。

[1]此处及全文中，$\log = \log_2$.[2]注意，这意味着二叉搜索树中不会有重复的条目。

L. Lambers 和 O. Semeráth（编）：GCM 2025 EPTCS 440，2026年，第 6–12 页，doi:10.4204/EPTCS.440.2

© Z. Ismaili Alaoui 和 D. Plump 本作品依据知识共享署名许可协议授权。

**查询**。 在二叉搜索树中搜索键的过程与插入类似。从根节点开始，我们递归比较目标键与当前节点的键：如果两者相等，则搜索成功；如果目标键较小，则搜索左子树；如果较大，则搜索右子树。搜索会持续进行，直到找到该键，或无法继续搜索，这表明该键不在树中。

**删除**。 与插入类似，从二叉搜索树中删除一个节点也必须保持其性质。需要处理三种可能的情况：

(1) 如果要删除的节点是叶节点，则可以直接将其移除，使其父节点少一个子节点。(2) 如果它有一个子节点，则移除该节点，并将其子节点直接连接到其父节点。
(3) 如果它有两个子节点，则用其左子树中的最大键替换其键，然后删除该替换节点（该节点始终是叶节点或只有一个子节点）。要找到左子树中的最大键，我们从左子节点开始，反复移动到右子节点，直到不存在更右侧的子节点为止；即使存在重复元素，这也能确保替换节点不会有两个子节点。此操作自然保持了树中元素的顺序。

在传统编程语言中，对于具有 $n$ 个节点的二叉搜索树，这些操作在最坏情况下可以在 $O(n)$ 时间内实现。（此时树会退化为链表。）如果二叉搜索树是平衡的，也就是说，任意节点的子树高度之差最多为 1，那么这些操作可在 $O(\log n)$ 时间内运行（见 [6]）。

## 3 程序

GP 2 编程语言采用所谓根节点的概念，这些根节点是与非根节点分开存储的特殊节点。为避免混淆，本节将二叉树的根节点（即顶部元素）称为顶部。

程序 bst（图 2 和图 3）在 GP2 中实现了一棵二叉搜索树，其顶部由一个绿色节点指向，该节点充当顶部的父节点，并遵循以下规范。

**输入：** 由未标记的节点和边组成的链表，且满足

1. 每个节点标签恰好表示一条用户指令（插入、查询和删除）；2. 标记为“i”：$n$ 的节点（其中 $n$ 为某个整数）表示将$n$ 插入二叉搜索树的指令；3. 标记为“s”：$n$的节点（其中 $n$ 为某个整数）表示在二叉搜索树中搜索标签为$n$的节点（若存在），并从指令节点到二叉搜索树中找到的节点创建一条虚线边；4. 标记为“d”：$n$ 的节点（其中 $n$ 为某个整数）表示将二叉搜索树中标签为$n$ 的节点（若存在）变为垃圾节点[3]；5. 第一条指令是链表的头节点，且为根节点；6. 当且仅当$u$表示第$i$条指令，且$v$表示第（$i+1$）条指令时，才存在一条从$u$到$v$的边。

**输出：** 该图由用户提供的链表（其尾节点改为根节点）、一个绿色节点，以及一棵二叉搜索树组成。二叉搜索树的节点为灰色，边没有标记；其顶部由绿色节点指向，并按照用户指定的指令顺序及第 2 节所述的过程构造。

[3]一个垃圾节点是一个无法从绿色节点到达的非指令节点。

```
Main    = 创建_根节点; （尝试插入，然后执行 Insert，否则（
                        尝试搜索，然后执行 Search，否则（
                        尝试删除，然后执行 Delete，否则失败）；
                        尝试 next_op，否则中断）！
Insert  =尝试根节点   然后（{go_right1, go_left1}！；
                      如果匹配则中断，否则添加_叶节点）否则添加_根节点
搜索    =尝试根节点   否则中断；{go_right1, go_left1}！；尝试匹配
删除    =尝试根节点   否则中断；{go_right1, go_left1}！；
          尝试 情况1，否则（尝试 情况2，否则 情况3）
情况1   = 尝试 删除_叶节点，否则失败
情况2   = 尝试 删除_midl，否则失败
情况3   = 保存_节点；转到_左侧2；如果转到_右侧2
          然后（转到_右侧2！；{交换1, 交换2, 交换3, 交换4}）
          否则{交换5, 交换6}；尝试情况1，否则情况2
```

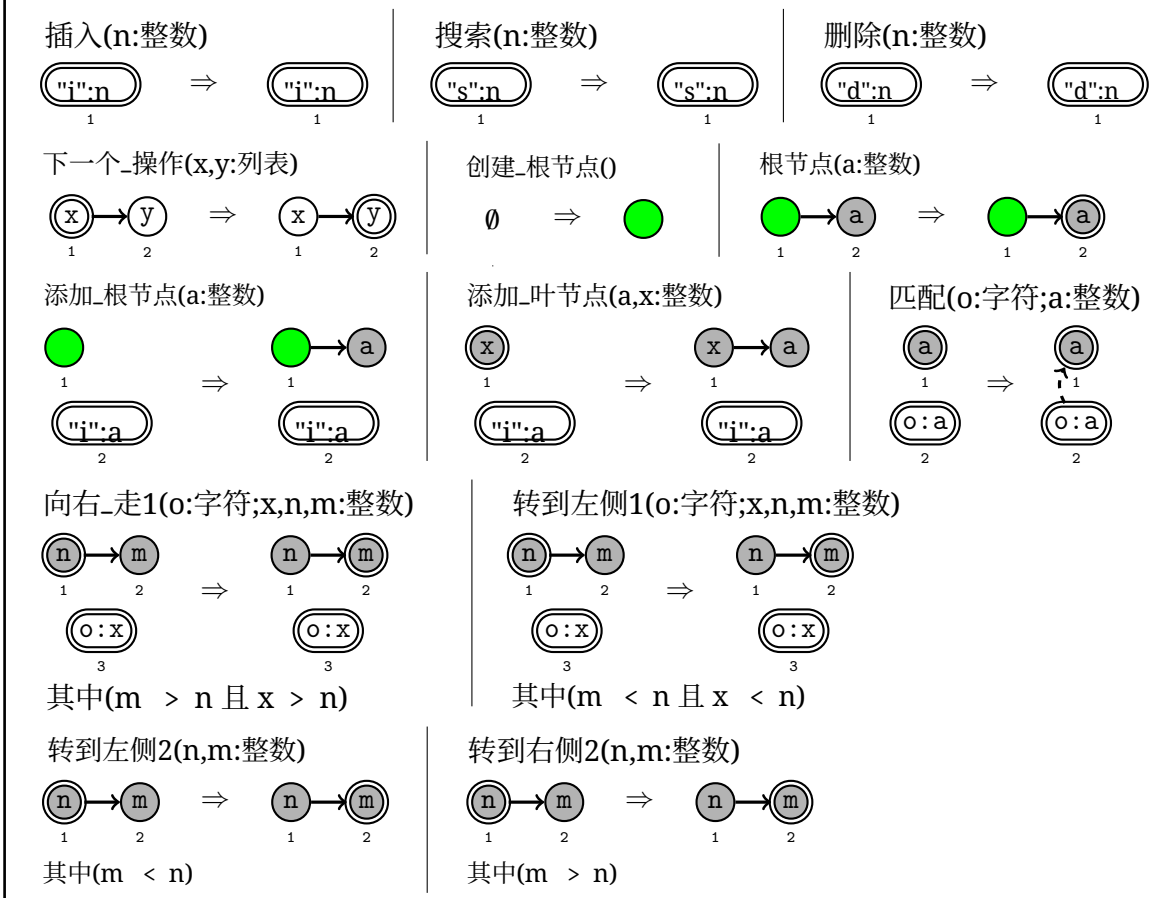


图2：程序 bst；其余规则以图形方式描绘于图3中。

我们将用户指定的节点称为指令节点。程序首先应用规则 make_root，创建一个绿色哑节点，其作用是作为二叉搜索树顶部的父节点；

这种技术使规则设计更加简单。程序通过遍历作为输入提供的链表，依次执行用户给定的指令。在程序执行过程中，当前正在执行其指令的指令节点会被设为根节点；否则，它不会被设为根节点。下面我们将按操作分别说明程序的行为，并将其模块化为相应的过程（Insert、Search 和 Delete）。

**插入**（Insert）。程序首先调用 root 规则，尝试将二叉搜索树的顶部设为根节点。如果不存在（即树为空），该规则将无法应用，此时改为应用 add_root，从而创建树的顶部。如果 root 规则成功应用（即树中至少有一个节点），则尽可能多次调用规则集 {go_right1, go_left1}。[4] 本质上，每条规则都会沿树向下移动根节点，直到无法再应用为止；当且仅当根节点到达一个子节点少于两个的节点，并且根节点指令节点中的新键可以作为二叉搜索树中该根节点的子节点添加时，就会发生这种情况。通过应用 add_leaf 规则完成插入。由于二叉搜索树不能存储重复元素，每次根节点移动到某个位置时都会尝试应用 match 规则；一旦树中存在具有相同键的节点，该规则便会终止循环。

**查询**（搜索）。该过程的工作方式与插入类似。程序首先调用 root，尝试将二叉搜索树顶端的节点设为根节点。如果该规则失败，过程便会终止，因为树中没有节点。遍历过程也类似：一旦规则集 {go_right1, go_left1} 中没有任何规则适用，就说明已找到该元素，或到达了死路。随后调用 match规则；如果匹配成功，它会在已设为根节点的指令节点与二叉搜索树中存有待查键的已设为根节点的节点之间创建一条虚线边。

**删除**（删除）。同样，程序会尝试应用 root规则；如果失败，过程便会终止，因为没有可删除的节点。随后，若存有待删除键的节点存在，就沿树向下进行与先前过程类似的搜索，并将该节点设为根节点。如果找到了该节点，则按第 2 节所述，删除分为三种不同情况。如果待删除节点是叶节点，则应用规则 delete_leaf。如果它是单子节点，则应用规则 delete_midl。否则（即该节点有两个子节点），应用规则 save_node 以记录该节点；然后将根节点移至左子节点，并尽可能一直向下遍历右子节点。最终，树中被设为根节点的节点将包含待删除节点的左子树中的最大键（LKCLS）。通过重新连接边来交换这些键；这由以 swap 开头的规则完成（根据 LKCLS 向下距离多少层，共有 6 种不同情况），随后适用另外两种情况之一来处理待删除节点。请注意，节点并不会从树中实际删除；实际上，它们只是与二叉搜索树断开连接，成为垃圾节点。插入先前已删除的键不会影响垃圾节点（也就是说，可能存在多个键相同的垃圾节点）。

**声明 1** 插入、删除和查询操作的运行时间为$O(n)$，其中 $n$ 是二叉搜索树节点的数量。

上述声明源于以下事实：所有规则都能在常数时间内匹配并应用，并且二叉搜索树无环，因此循环 {go_right1, go_left1}! 总会在至多$n$ 次迭代后终止，其中$n$ 是二叉搜索树节点的数量。

[4]二叉搜索树的边不必进行显式标记，因为通过比较端点节点的值，即可直接推断每条边的方向（左或右）。这不会影响声明 1 中所述的复杂度。

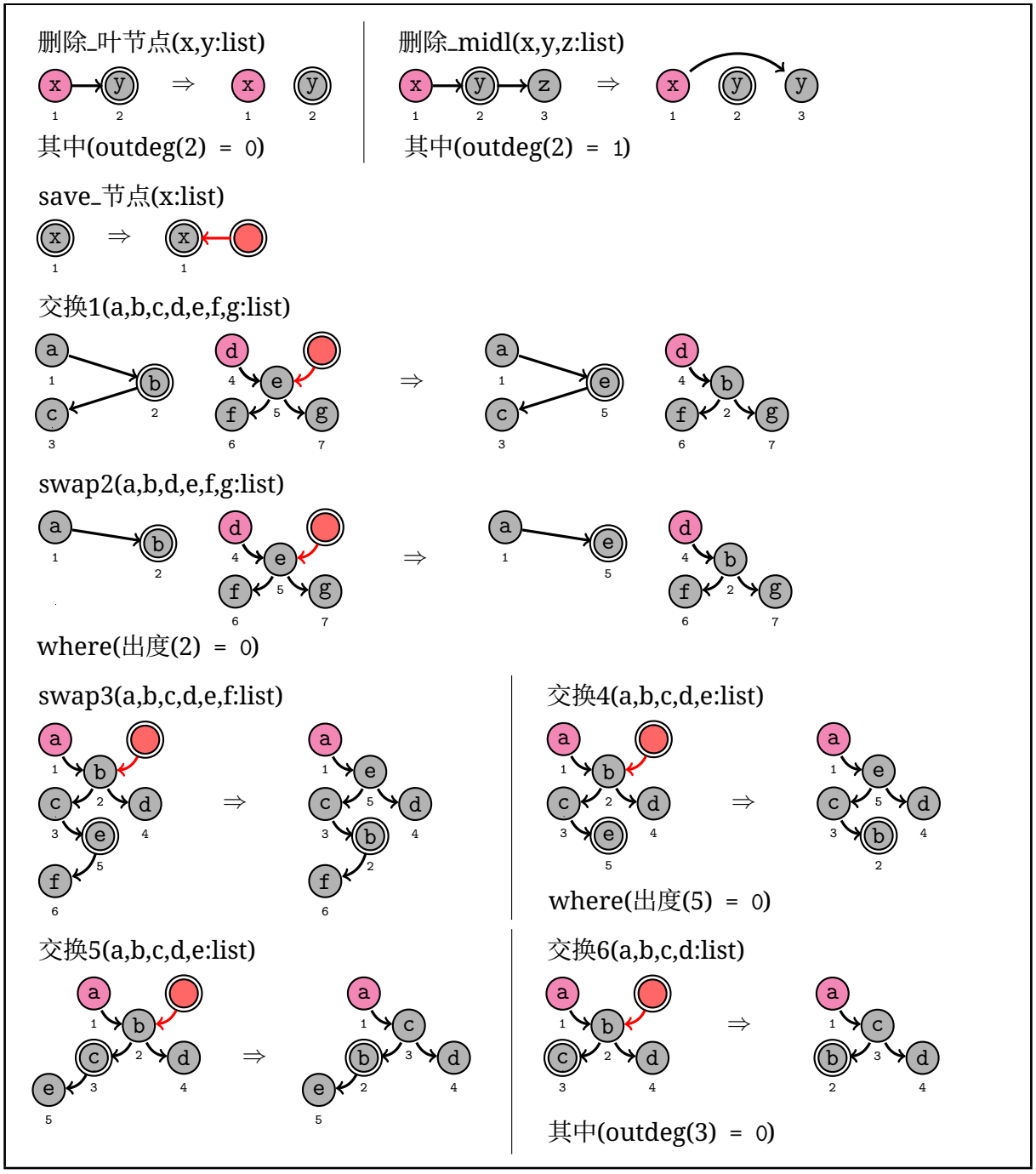


图3：程序 bst 的其余规则。

## 4 运行时测量

我们在图4和图5中给出了程序 bst[5] 的运行时间测量结果（测量使用一台四核英特尔 i5 处理器，主频为 2.4 GHz，配备 16 GB 内存，运行 64 位 Ubuntu 22.04）。在图4中，我们只考察退化树（即链表），并记录执行单次操作所需的时间。对于每种树大小，操作均执行 300 次，并对结果取平均值。在链表上执行每项操作所需的时间更长，因为任何更新或搜索都需要遍历所有已有顶点。相比之下，对于大小相同的平衡树，此类操作最多只需访问 $\log n$ 个顶点，数量显著更少。这一差异解释了为什么第二幅图中的操作快得多。在图5中，我们只考察完全平衡的树。每次测量都记录向树的一个叶节点插入元素、搜索该元素，随后将其删除所需的时间。由于平衡树上每项操作的运行时间都太短，无法可靠测量，因此我们将这三项操作合并测量，以获得更稳定且更具代表性的结果；与退化树的情况一样，每组操作序列重复执行 300 次，并对结果取平均值。

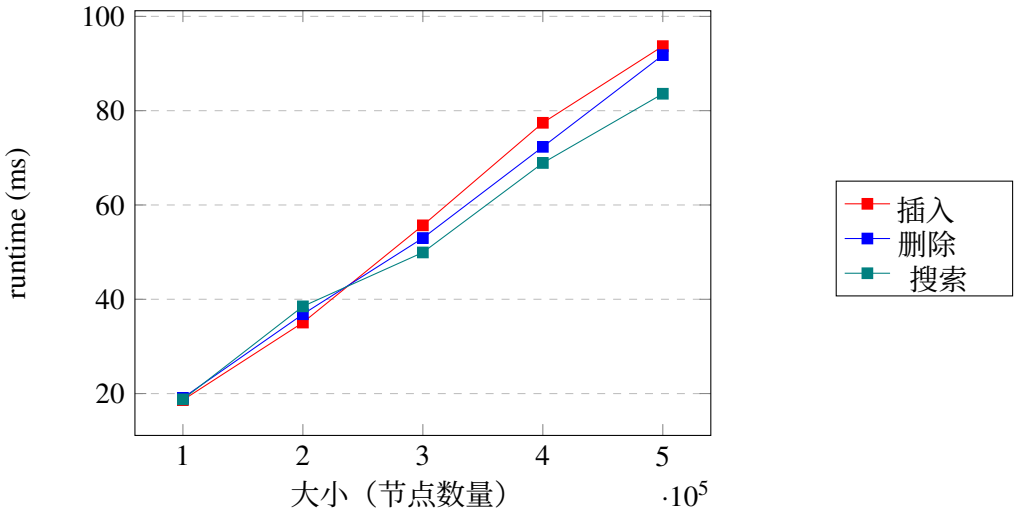


图4：退化树上每次操作的测量运行时间。

## 5 结论与未来工作

我们展示了如何在基于规则的语言 GP 2 中实现二叉搜索树，使得在最坏情况下，搜索、插入和删除条目都需要线性时间。下一步将通过实验和分析证实，这些操作的平均运行时间为 $O(\log n)$。

未来工作的另一个主题是实现某种形式的平衡 二叉搜索树，例如红黑树，在这类树中，操作能够保持平衡。对于这类结构，查询、插入和删除操作的运行时间都能保证为 $O(\log n)$。 [2]，讨论了在图变换语言 PROGRES 中实现红黑树的方法，其中允许节点的左、右子树高度相差最多 2 倍。我们还提及了 [3]，这是一种图转换方法，用于形式化验证红黑树的插入操作。

[5]具体 语法见：https://github.com/UoYCS-plasma/GP2/blob/master/programs/bst.gp 2.

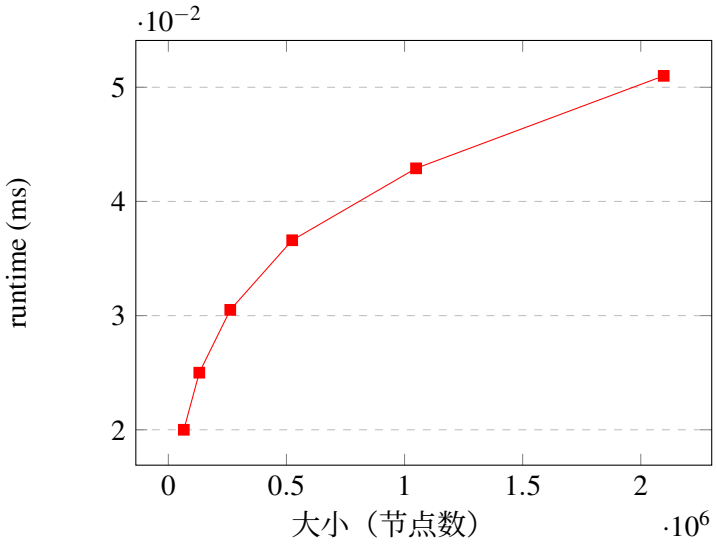


图5：在平衡树上执行一次插入、随后进行搜索和删除所测得的平均运行时间。

在 GP2 中实现数据结构的另一种方法，是使用 C 等命令式语言高效地实现它们，并在 GP 2 中提供访问操作。不过，我们认为，对于二叉搜索树这类图结构，使用基于规则的图变换语言编写操作代码，更便于阅读和理解。此外，在 GP 2 中同时编写应用程序和数据结构实现，还可以进行形式化程序验证，例如采用 [5, 7] 中描述的方法。

## 参考文献

[1] Ziad Imaili Alaoui & Detlef Plump（2025年）：与命令式算法时间复杂度相匹配的基于规则的图程序。技术报告，约克大学，doi:10.48550/ARXIV.2501.09144。40页。[2] Marc Andries、Gregor Engels、Annegret Habel、Berthold Hoffmann、Hans-Jörg Kreowski、Sabine Kuske、Detlef Plump、Andy Schürr & Gabriele Taentzer（1999年）：用于规约与编程的图转换。《计算机程序科学》34(1)，第1–54页，doi:10.1016/S0167-6423(98)00023-9。[3] Paolo Baldan、Andrea Corradini、Javier Esparza、Tobias Heindel、Barbara König & Vitali Kozioura（2005年）：验证红黑树。载于：动态分配堆并发系统验证研讨会（*COSMICAH 2005*），伦敦玛丽女王大学，第1–15页。非正式论文集。[4] Graham Campbell、Brian Courtehoute & Detlef Plump（2022年）：快速基于规则的图程序。《计算机程序科学》214，文章编号102727，doi:10.1016/j.scico.2021.102727。[5] Christopher M. Poskitt & Detlef Plump（2023年）：*GP 2* 的单子二阶错误逻辑。《逻辑与代数编程方法杂志》130，文章编号100825，doi:10.1016/j.jlamp.2022.100825。[6] Steven S. Skiena（2020年）：《算法设计手册》，第三版。Springer，doi:10.1007/978-3-030-54256-6。[7] Gia Wulandari & Detlef Plump（2021年）：使用单子二阶逻辑验证图程序。载于：第*14*届国际图转换会议（*ICGT 2021*）论文集，《计算机科学讲义》12741，Springer，第240–261页，doi:10.1007/978-3-030-78946-6_13。