arXiv:2301.03224v1 [cs.SE] 9 Jan 2023

使用 Dafny 开发经验证程序以进行可访问性评估的案例研究*

João Pascoal Faria[1], [2][0000−0003−3825−3954] 和 Rui Abreu[1], [3][0000−0003−3734−3157]

[1] 波尔图大学工程学院，波尔图，葡萄牙 {jpf,rma}@fe.up.pt[2] INESC TEC - 系统与计算机工程、技术与科学研究所，波尔图，葡萄牙[3] INESC ID，里斯本，葡萄牙

摘要。 形式化验证技术旨在形式化地证明计算机程序相对于形式化规约的正确性，但应用形式化规约与验证技术所需的专业知识和投入，以及可扩展性问题，限制了这些技术的实际应用。近年来，SAT 和 SMT 求解器取得了巨大进展，推动构造出新一代工具；这些工具有望通过自动化大部分乃至全部验证过程，让软件工程师更容易进行形式化验证。Dafny 系统是这一发展趋势的典型代表。然而，目前关于其易用性的证据仍然有限。为弥补这一空白，我们开展了 10 项案例研究，使用 Dafny 开发若干真实世界算法和数据结构的经验证实现，以评估其对软件工程师的易用性。我们发现，平均而言，用于规约与验证的代码量与传统上用于实现与测试的代码量处于同一数量级（比率为 1.14）——对于高完整性软件而言，这种“开销”无疑是值得的。Dafny 验证器的性能令人印象深刻：平均每行代码可生成 2.4 项证明义务，而每项证明义务的生成与验证平均耗时 24 毫秒。不过，我们也发现，编写辅助验证代码所需的人工工作可能相当繁重，且难以预测和掌握。因此，进一步自动化和系统化验证任务，可能是该领域未来发展的方向。

关键词： 形式化验证 · Dafny · 无障碍 · 案例研究。

1 引言

1.1 动机

随着我们的社会日益依赖基于软件的系统，确保它们正确、安全且可靠地运行变得愈发重要，尤其是对于高完整性系统 [1]。由于软件开发是一项知识密集型活动，且基于软件的系统日益复杂，错误

* 这是我们 FSEN 2023 论文的扩展版本，其中包含源代码。

在所难免，因此需要在开发过程中采用多种技术，尽早发现并修复缺陷。

测试和评审是软件行业中应用最广泛的缺陷检测技术。然而，正如“程序测试可以用来证明错误的存在，却永远无法证明错误不存在” [2],，仅靠测试不足以用于高完整性系统。如果应用得当， [3],评审是一种经济高效的缺陷检测和知识共享技术，但与测试一样，无法用来证明错误不存在。

相比之下，形式化验证技术旨在从形式上证明计算机程序的正确性，即证明不存在缺陷。为此，我们需要对程序意图进行形式化规约，并建立一个逻辑推理框架，通常以霍尔逻辑为基础 [4]。然而，应用形式化规约与验证技术所需的专业知识和投入，以及可扩展性问题，限制了这些技术在实践中的应用。近年来，SAT 和 SMT 求解器 [5], （例如 Z3 [6],）取得了巨大进展，推动构造出新一代工具，有望让软件工程师也能使用形式化验证，例如 Dafny [7], 、Frama-C [8] 和 Why3 [9], ，它们能够自动化验证过程中的大部分乃至全部工作。然而，目前几乎没有证据表明这些工具易于使用，也缺乏关于应用它们所需专业知识和投入的相关证据。

作者多年来在软件工程研究、教育和实践中使用形式化规约语言和自动推理工具 [10, 11, 12, 13, 14]。例如，在 [11], 中，Alloy [15]被用于根据泛型的代数规约自动生成 JUnit[4] 单元测试和模拟对象。尽管此类基于模型的测试方法无法保证完全没有错误，但其提供的保障高于手动生成测试，而且目前似乎比形式化验证更容易采用。

从教育角度来看，作者还希望评估在本科课程中嵌入计算机辅助的形式化规约和验证技术是否可行，具体而言，是在专门学习算法和数据结构的课程中引入这些技术。

1.2 目标与方法

为弥补当前技术现状中无障碍性研究的空白，我们开展了一系列案例研究，在 Dafny 中开发了若干复杂度各异的知名算法和数据结构的经验证的实现，旨在确定这些方法对于软件工程从业者、学生和研究人员的易用程度；他们接受形式化方法培训的时间有限。

表 1 展示了案例研究列表。这些案例研究探索了复杂度逐步提高的形式化规约与验证特性。在第 2 节中，我们介绍了部分精选特性的要点。对于每项案例研究，我们收集了一些度量指标和经验教训，以帮助回答有关 Dafny 易用性的核心问题。˙这些度量指标和经验教训汇总并讨论于第 3 节。源代码可在 GitHub 代码仓库[5] 和附录 A 中获取。

[4] https://junit.org/[5]
https://github.com/joaopascoalfariafeup/DafnyProjects

表 1. 案例研究列表。

类别	案例研究
数值算法	○ 整数除法（欧几里得除法） ○ 数的自然数次幂（分治算法）
搜索 & 排序算法	○ 二分查找 ○ 插入排序
集合	○ 使用二叉堆实现的优先队列 ○ 使用哈希表实现的无序集合（哈希集合） ○ 使用二叉搜索树实现的有序集合（树集合）
匹配问题-lems	○ 通过盖尔-沙普利算法解决稳定婚姻问题 ○ 将教师分配问题归约为稳定婚姻问题
图算法	○ 拓扑排序（Khan 算法 [16]） ○ 欧拉回路（希尔霍尔策算法）

1.3 论文结构

第 2 节介绍了案例研究中复杂度逐步提高的规约与验证特性要点。第 3 节汇总了收集到的度量指标和经验教训，并就我们的研究目标得出结论。第 4 节讨论相关工作。第 5 节介绍结论和未来工作。

2 案例研究亮点

2.1 入门示例（整数除法）

图 1 中这个顾名思义的程序探讨了 Dafny 的一些基本特性，并作为我们的第一个案例研究。

Dafny[6] [7] 是一种多范式编程语言及系统，用于开发经验证程序。函数式风格通常用于编写规约，使用值类型、无副作用表达式、函数和谓词。过程式和面向对象风格通常用于编写实现，使用引用类型（数组、类等），以及带有副作用的方法和语句。Dafny 编程系统包含一个验证器（基于 Z3）、可生成多种目标语言代码（C#、Java、JavaScript、Go 和 C++）的编译器，以及 Visual Studio Code 扩展。

方法（此处为 div）的语义通过前置条件和后置条件进行形式化指定，分别由 requires 子句和 ensures 子句表示。Dafny 验证器负责检查这些前置条件和后置条件是否成立（借助 Z3 定理证明器）。如果实现中包含循环，用户必须提供循环不变式（通过

[6] https://github.com/dafny-lang/dafny

```
// 计算整数除法的商 q 和余数 r
// 将（非负）被除数 n 除以（正）除数 d。
方法 div(n: 自然数, d: 自然数) 返回 (q: 自然数, r: 自然数)
  要求 d > 0
  确保 q * d + r == n && r < d
{
  q := 0;
  r := n;
  当 r >= d
    递减 r
    不变式 q * d + r == n
  {
    q := q + 1;
    r := r - d;
  }
}
// 主程序，包含一个简单的测试用例（已通过静态检查！）。
方法 Main() {
    变量 q, r: = 整除(15, 6);
    断言 q == 2 && r == 3;
    打印 "q=", q, " r=", r, "\n";
}
```

图 1. Dafny 中执行整数除法的简单程序。

invariant 子句），并在某些情况下提供循环变式（通过 decreases 子句），以帮助验证器完成其工作。

Main 方法是 Dafny 程序的入口点。在此示例中，它使用一些输入调用 div 方法，并检查（使用 assert）和打印相应的输出。与前置条件和后置条件一样，assert 语句由 Dafny 验证器进行静态检查。在此示例中，验证器将仅根据 div 方法的后置条件尝试证明该断言（即，出于此目的，方法体是不透明的）；这使验证具有模块化和可扩展性。由于断言是在静态分析期间检查的，因此，像所示示例这样的测试用例实际上是在编译前阶段测试规约，而不是在运行时测试实现；此类静态测试用例有助于发现规约中的问题，例如不完整的后置条件。

上述提到的所有规格构造和断言（由 requires、ensures、invariant、decreases 和 assert 子句表示）仅作为验证用途的注解（在静态分析期间使用），不会被编译到可执行程序中，因此不会产生运行时开销。

2.2 引理与 AutomaticInduction（数的幂）

在这个案例研究中，目标是证明一个著名的 $O(log\ n)$分治算法的正确性，该算法用于计算实数的自然数次幂（x^n）。图 2 展示了易于理解的代码片段，完整代码见第 A.2 节。它说明了如何使用引理来指定 Dafny 单靠自身无法推导出的性质，

以及如何使用自动归纳，即 Dafny 通过归纳法自动证明某些性质的能力（指令：induction a）。

```
// 函数式风格的 x^n 递归定义。
函数 power(x: 实数, n: 自然数) : 实数 {
    如果 n == 0 则 1.0 否则 x * power(x, n-1)
}
// 在 O(log n) 时间和空间内计算 x^n。
方法 powerDC(x: 实数, n: 自然数) 返回 (p : 实数)
  确保 p == 幂(x, n)
{...
    如果 n % 2 == 0 {
        幂的乘积(x,  n/2, n/2): // 同想引理
        变量 temp: = powerDC(x, n/2);
        返回 temp * temp;
    }...
}
// 陈述性质 x^a * x^b = x^(a+b)，供"powerDC"使用。
// 该性质通过对"a"进行自动归纳证明。
引理 {:归纳法 a} 幂的乘积(x: 实数, a: 自然数, b: 自然数)
  确保 幂(x, a) * 幂(x, b)  == 幂(x, a + b)
{/*证明应写在这里，但 Dafny 已自动发现！*/}
```

图 2. Dafny 中用于计算一个数的自然数次幂的程序片段。

2.3 模块、可变对象与泛型（插入排序）

在这个案例研究中，我们探索 Dafny 中用于处理可变对象（此处为数组）和泛型的功能，并通过模块分离规约、实现和测试代码。图 3 展示了易于理解的代码片段。

数组排序问题通过抽象模块 Sorting 中不含方法体的排序方法 sort 进行规约，并借助辅助谓词。框架条件"modifies a"表示实现可以修改 a 所引用的内容。后置条件中的"old(a[...])"和"a[..]"分别表示方法执行开始和结束时的数组内容，并将其视为数学序列。Dafny 对泛型谓词、函数和方法提供了一些支持，但遗憾的是，不支持可进行相等以外运算的类型参数（==）；因此，为了演示，我们使用特定的类型定义来声明数组元素的类型。

排序算法可以在细化抽象模块的具体模块中提供，例如 InsertionSort 模块。该模块继承方法契约，并在方法体中提供实际算法（此处省略）。在这种情况下，我们只需提供循环不变式，验证器便能成功检查插入排序算法是否符合规约。

模块 TestSorting 展示了一个排序方法的测试用例示例。为了让 Dafny 验证器成功检查测试结果中的

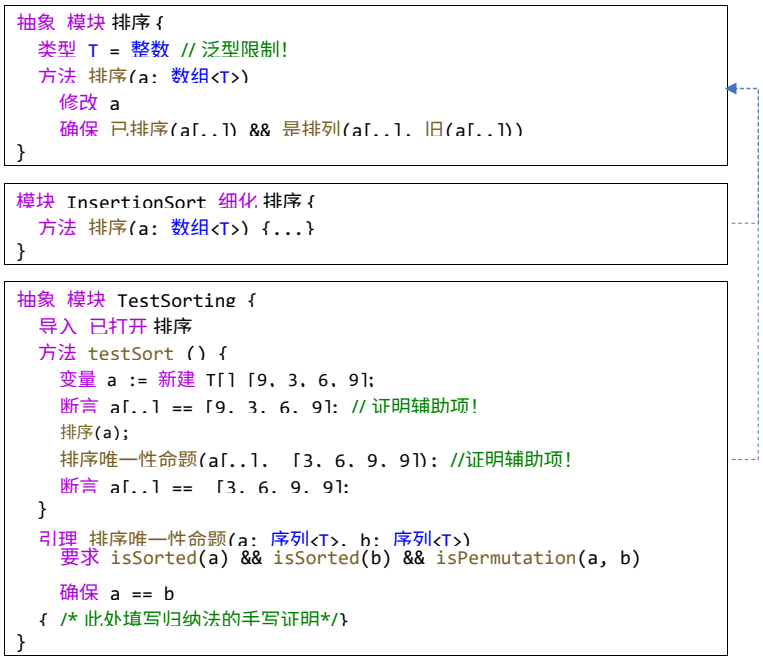


图 3。 使用模块在 Dafny 中组织数组排序程序。

最后一条断言语句，我们必须编写一个辅助引理，说明排序结果是唯一的。令人惊讶的是，为了让代码通过检查，我们还必须提供一些额外的“证明辅助项”断言（作为第一条断言），陈述一些我们原以为理所当然的显而易见的事实。

2.4 状态抽象与自动契约（优先队列）

在本案例研究中，我们探索 Dafny 在分离规约与实现，以及处理面向对象程序中的类不变式方面的特性，遵循契约式设计（DbC）原则。图 3 展示了优先队列的规约及其使用二叉堆实现的摘录。

优先队列（图 3 顶部框）的操作前置条件和后置条件独立于内部状态表示（在此为二叉堆）进行规约，借助的是状态抽象函数 （elems）。该函数将优先队列的内容表示为多重集（允许重复值），仅用于规约与验证（不会生成可执行代码）；为了使规约保持较高层次的抽象，它不说明元素的排序顺序（该顺序由 deleteMax 给出）。

在后续的细化中（图 3 中间的方框），选择了一种内部（具体）状态表示——存储在数组中的二叉堆。还

为每个方法提供了实现（方法体）（图 4 底部的方框）。在 Dafny 中，借助所谓的自动契约（使用“:autocontracts”属性），可以简化类不变式的定义和验证。这些不变式规定了内部状态必须满足的限制，并且必须在方法边界处得到遵守。类不变式通过谓词 Valid 指定；对该谓词的调用以及一些框架条件，会自动插入所有方法的前置条件，以及所有方法和构造函数的后置条件中。



图 4. 优先队列规约（上方）及其使用 Dafny 中的二叉堆实现（中间和下方）的摘录。

得益于状态抽象函数和类不变式，Dafny 验证器能够自动检查方法的实

现（以具体状态定义）是否符合方法的前置条件和后置条件（以抽象状态定义），而无需用户额外付出！我们只需定义一个辅助引理，证明堆不变式（由图 4 中的谓词 Valid 表示）蕴含最大值位于顶部（数组索引 0 处）。

2.5 证明技术（拓扑排序、欧拉回路）

简单的算法可能需要复杂的证明，这一点并不令人意外，拓扑排序案例研究便展示了这一点。事实上，Kahn 算法 [16] 只需 6 行代码（在较高的抽象层次上）即可实现，但为了证明其正确性，我们不得不编写 7 个辅助引理，如图 5 所示。幸运的是，Dafny 支持丰富多样的证明技术，能够补全大部分（甚至全部）证明步骤，因此我们只需提供关键的中间步骤，使得每个引理的手写证明都相当简短。

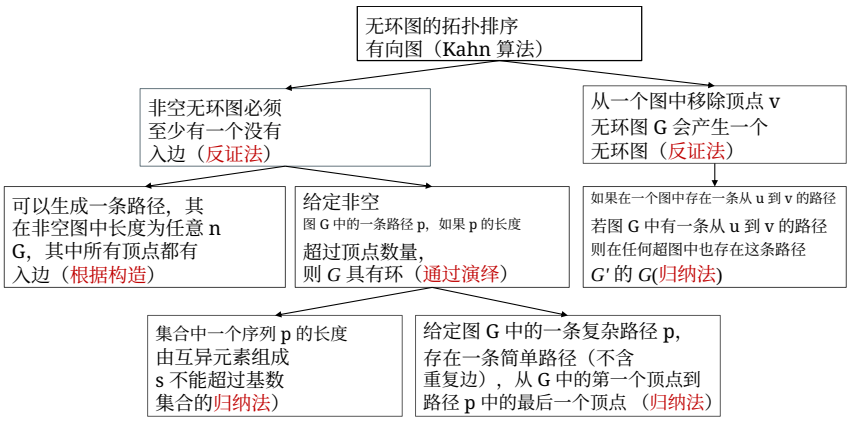


图 5. 用于证明 Kahn 算法正确性的引理和证明技术。

然而，证明步骤的编写方式可能会对验证时间产生重大影响。例如，在欧拉回路案例研究中，为验证一个引理花费了约 20 秒。该引理指出：如果图 G 中存在欧拉迹 r （即一条恰好经过每条边 G 一次的路径），那么除非图 r 中的首尾顶点不同，否则图 G 中每个顶点都有偶数个相邻顶点。该证明采用归纳法。将归纳步骤改写为从 r 和 G 中移除第一条边，而非最后一条边（这可能更符合证明所需的递归定义结构），验证时间便缩短至不到 1 秒！

3 结果与讨论

本节总结所开展案例研究中收集的度量指标以及获得的经验，并围绕我们的研究目标得出一些结论。

3.1 收集的度量指标

表 2 总结了案例研究中收集的度量指标。表 3 所述代码类别的大小以物理代码行数（LOC）衡量，不包括空行和注释。

执行时间是在一台搭载 Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz、配备 6 核处理器和 16 GB RAM、运行 Windows 10 企业版的笔记本电脑上测量的。我们使用了 VS Code 的 Dafny 扩展 v2.1.1 和 Dafny 服务器 3.3.0 版；在某些情况下，由于 Z3 与 Dafny v3[7]存在一个错误，我们使用了 2.3.0 版。

表 2. 案例研究结果（大小、时间和证明义务）。

程序	实现。LOC	Test LOC	规格。LOC	验证 LOC	总计 LOC	(S+V)/(I+T)	证明义务	验证时间（秒）
整数除法	10	5	2	2	19	0.27	15	0.5
数的幂	17	7	4	5	33	0.38	45	0.5
二分查找	15	7	7	3	32	0.45	51	0.5
插入排序	13	13	10	21	57	1.19	90	1
优先队列	74	13	30	35	152	0.75	483	3
哈希集合	86	16	57	38	197	0.93	656	16
树集合	**87**	13	39	38	177	0.77	**809**	18
稳定婚姻	50	**66**	54	10	180	0.55	209	7
拓扑排序	19	18	21	94	152	3.11	157	3
欧拉回路	32	10	**66**	**115**	**223**	**4.31**	407	**19**
总计	**403**	**168**	**290**	**361**	**1222**	**1.14**	**2922**	**69**

表3。 代码类别。

类别	描述
实施—实现	“传统的”、可编译的实现代码（方法签名，方法体、数据定义等）。
Test	测试代码（静态或动态检查），包括断言。
规约	契约的规约，包括 requires 和 ensures 子句、类不变式、框架条件，以及其中使用的辅助定义。
验证	验证辅助代码，例如引理和所有不可编译的代码（方法体中的循环变式、循环不变式、断言、引理调用、幽灵变量操作等）。

[7] https://github.com/dafny-lang/dafny/issues/1498

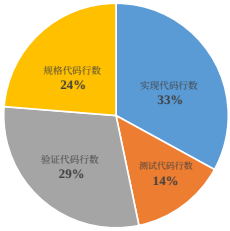


图 6. 代码规模（代码行数）分布。

平均而言，用于形式化规约（S）和验证（V）的代码量，与用于实现（I）和测试（T）的“传统”代码量处于同一数量级——这种“额外开销”显然是值得的，至少对于高完整性软件而言如此。平均比率为 (S+V)/(I+T)=1.14，从最简单情况的 0.27 到最复杂情况的 4.31 不等。图 6 的饼图显示，平均而言，不同代码类别之间的代码量分布较为均衡。

用户所花费时间的开销很难衡量，因为它很大程度上取决于用户经验。公平的评估应在不同的情境下进行（在这些案例研究中，算法是已知的，但在许多情况下，验证策略需要经过探索才能确定）。我们认为，经过适当培训后，在需要设计新算法的情况下，规约与验证所需的工作量可以与设计、实现和测试的工作量处于同一量级。

Dafny 验证器生成并检查的证明义务（PO）数量令人印象深刻：在这些案例研究中，平均每编写 1 行代码就会生成 2.4 个证明义务（表 2 中为 2922 个证明义务/1222 行代码），每行实现代码则对应 7.3 个（表 2 中为 2922 个证明义务/403 行代码）。Dafny 验证器的性能同样令人印象深刻：在这组案例研究中，平均每个生成并验证的证明义务耗时 24 毫秒（表 2 中为 69 秒/292 个证明义务）。

然而，根据案例研究的经验，需要注意的是，某些证明义务的验证可能耗时很长，达到数分钟，甚至无法终止。遇到这种情况时，通过仔细调试和重构（断言、验证代码等），通常可以大幅缩短验证时间（如欧拉回路案例研究所示）。

3.2 经验教训

案例研究的经验教训汇总于表 4 和表 5 中，并采用配色方案突出显示优势和劣势。总体而言，Dafny 语言和验证器表现出了强大的能力，能够自动完成大部分验证工作，但在语言功能上存在一些局限（涉及泛型、自动契约等方面）。对于我们的主要研究问题，我们发现最大的难点在于：对于非平凡程序，人工验证工作可能非常繁重，而且难以预测和掌握。

表 4. 案例研究的经验教训（第一部分）。

类别：经验教训（优点 和 缺点）	
Dafny Lan-言	– **集成式语言** 用于编写 **规约** （方法的前置条件和后置条件）、**实现** （方法体）以及**验证-辅助代码** （例如循环不变式）[例：整数除法]。 – 丰富的 **逻辑量词**（全称量词、存在量词等）以及**数学-数学集合** （序列、集合、多重集、映射等），用于编写规约和断言，并以高层次的抽象描述复杂算法 [例：二分查找、稳定婚姻]。 – **归纳数据类型** 和 **模式匹配** 表达式可用于使代码保持较高层次的抽象 [例如：哈希集合]。 – **空值安全**：引用类型除非标记了“?”后缀，否则不可为空。 [例如：树集合] – 用于指定**框架条件**并**查询对象的旧状态**的构造，适用于处理可变对象 [例如：插入排序]。 – **模块** 实现了规约、实现和测试代码之间的清晰分离 [例如：插入排序]。 – 对**泛型**的支持有限：不支持受以下操作约束的类型参数（相等之外的其他操作） [例如：二分查找]。 – 显式分离规约与实现的支持以及在面向对象程序中隐藏实现细节的支持仍有提升空间以便改进（例如，没有可见性修饰符） [例：树集合]。
Dafny Com-译器	– Dafny 编译器能够生成可执行代码，使用**多种目标语言**（本例中仅探讨 C#）。 – 用于规约和验证的断言及其他构造——证明义务不会被编译，因此意味着**没有运行时开销**。
Dafny 验证器	– 在许多情况下，验证器能够**自动检查实现是否符合规约**，而用户只需提供极少的帮助（可能只需编写循环不变式） [例如：整数除法]。 – Dafny 通常能够发现循环变式 [例如：二分查找]。 – 在方法之外，方法体对于验证而言是不可见的——（只有前置条件和后置条件才重要），这使验证过程具有模块化特性且可扩展。
手动验证评估 Work	– Dafny 有效支持多种**证明技术**(例如**演绎、归纳法、矛盾、构造、计算**[17]) [例：拓扑排序、树集合] – 可能需要用户定义辅助性质（作为**引理**），以帮助验证器，但证明本身可以高度自动化，甚至完全自动化，许多细节会自动补全；发现需要定义哪些性质并非易事，尽管如此 [例如：幂、拓扑排序]。 – 很难**预测何时需要人工处理，以及需要处理什么工作**（除了编写循环不变式之外）对于成功验证所必需 [例如：插入排序、拓扑排序]。

表5. 案例研究的经验教训（第二部分）。

类别 经验教训（优点和缺点）	
自动化 con-契约	– Dafny 支持定义并强制执行**类不变量**，尤其是使用 ":autocontracts" 属性，同时也会处理适当框架条件的生成 [例如：优先队列]。 – 自动契约仍有改进空间；在某些情况下，用户可能需要改用底层功能 [例如：树集合、哈希集合]。 – 为表示自引用数据的类正确编写契约数据结构可能相当棘手 [例如：树集合]。 – 继承与自动契约之间存在明显冲突契约 [例如：优先队列]。
状态抽象-tion	– **状态抽象函数**（幽灵函数）允许指定类所提供服务的语义（前置条件/后置条件）与实现（方法体和内部状态表示）相互独立 [例如：优先队列]。 – 状态抽象也可以通过**抽象状态变量**（幽灵变量）实现，其与具体状态变量之间的抽象关系在类不变式中指定 [例如：哈希集合]。
测试	– 测试仍然相关，但主要用于**对规范进行静态测试分类**，且不对实现进行动态测试（已证明为符合规约）[例如：整数除法，插入排序]。 – 允许多个输出的测试用例可以轻松指定和检查 [例：插入排序]。
调试-以及性能分析	– 当验证失败时，Dafny 语言和 Dafny 验证器提-供了多项便于调试的功能，例如 assume 语句和"/tracePOs"选项 [ex: 欧拉回路]。 – 验证时间较长时，大部分时间可能集中在一两个断言上。通过识别并改写这类断言，验证时间可能会大幅缩短 [例如：欧拉回路]。

3.3 可访问性评估

我们区分了使用 Dafny 开发经过验证的程序所需的三种能力水平，其可及性依次降低：

– **基础**：编写实现和测试代码；– **中级**：编写规约（前置/后置条件、框架条件、类不变量及相关谓词和函数），以及循环变式和不变式；– **高级**：除循环变式和不变式外，还需识别并编写所需的验证代码（辅助引理、断言、幽灵变量等）。

案例研究中总结的经验和收集的度量指标表明，即使面对看似简单的问题，用户也可能需要熟练掌握高级验证功能和技术。

因此，尽管 Dafny 大幅提升了程序自动验证能力，我们仍认为，要实现让软件工程从业者和学生能够轻松开发经验证程序这一目标，“我们已经非常接近，但尚未达到”。进一步实现验证任务的自动化与系统化（包括为常见性质建立可复用库，并编写“操作指南”），以及将其集成到主流语言中，都是该领域未来可能的研究方向。

我们的评估结果也得到了我们教学经验的印证：我们在 2020/21学年开设了“软件工程中的形式化方法”课程[8]，共有 151 名硕士生选修，学生反馈非常积极（平均评分为 6 分，满分 7 分）。期中考试成绩较高（$\geq$ 85%）的学生受邀使用 Dafny 完成一个项目，开发一个中等复杂度算法或数据结构（哈希集合、树集合、稳定婚姻、拓扑排序、欧拉回路和文本压缩）的经验证的实现。符合条件的 28 名学生中，14 名接受了挑战，但只有 9 名提交了项目，且没有人完全达到目标。需要指出的是，关于形式化规约与验证的课程（连续 6 周、每周 4 小时）仅对高级验证技术进行了浅显介绍，而且学生完成项目的时间相对较短（1 个月）。这段经历使我们得出结论：要让有兴趣的学生能够使用 Dafny 或类似系统处理非平凡的规约与验证问题，还需要更深入的培训。

4 相关工作

在 [18], 中，作者报告了他们在 VerifyThis 2021 程序验证竞赛中使用 Dafny 的经验。该竞赛旨在通过受控实验评估基于逻辑的程序验证工具的易用性，并对验证工具及其用户提出挑战。他们应对了提出的两项挑战，由此指出了 Dafny 在验证相对复杂算法方面的优势和不足。作者提到的优势包括：Dafny 能以较少的输入证明终止性和内存安全；内置值类型，如集合、序列、多重集和映射；可使规约更简洁的谓词和引理；自动归纳；幽灵变量和函数。他们发现，验证可能为空的对象的性质等任务颇为困难，因而未能按时完成所有任务。

在 [19] 中，作者认为，形式化验证工具往往由专家为专家开发；因此，对于形式化方法经验有限的程序员而言，这些工具的可用性可能受到严重限制。他们介绍了在两种具有代表性的非专家使用情境中使用 AutoProof（这是一种能够验证 Eiffel 面向对象软件功能正确性的工具）的经验。首先，他们讨论了研究生课程“软件验证”中的学生对该工具的使用情况；这些学生的任务是验证各种排序算法的实现。其次，他们评估了该工具在验证为

[8] https://sigarra.up.pt/feup/en/UCURR_GERAL.FICHA_UC_VIEW?pv_ocorrencia_i d=459493

某本科课程编程作业开发的代码时的可用性。他们报告了使用经验和心得，并据此提出了一些改进验证工具可用性的建议。他们报告称，在两个小型程序中，规约与验证注解中的词元数量与实现代码中的词元数量之比平均为 1.3。尽管情境和计量单位有所不同，这一比例与我们的处于同一数量级。

在 [20] 中，作者指出，学生常因认为形式化方法难度高、数学性强且与实践无关而对其产生抵触。他们重新设计了软件正确性课程，采用编程密集型教学方式，并使用 Dafny 通过自动评估提供即时的形成性反馈，从而提高了学生留存率和课程评价。总体而言，学生的反馈非常积极，但他们发现 Dafny 难以学习和使用。作者的非正式观察表明，许多困难源于 Dafny 工具引入的“意外”复杂性。他们提出对 Dafny 的设计作出一些改进，以解决程序测试、验证调试和类不变式等方面发现的问题。

5 结论与未来工作

我们开展了一系列案例研究，在 Dafny 中开发一些真实世界和知名算法及数据结构的经验证实现，旨在确定这种做法对软件工程专业学生、从业者和研究人员而言是否易于使用。我们得出结论：尽管 Dafny 在自动化程序验证方面取得了显著进展，但编写辅助验证代码所需的人工工作可能相当繁重，且难以预测和掌握。进一步自动化并系统化验证任务（包括常见性质的可复用库和“操作指南”），以及将其集成到主流语言中，都是该领域未来可能的研究方向。我们还计划使用其他验证器和问题开展进一步研究。

致谢

本研究由 NationalFunds 资助，资金通过葡萄牙资助机构 FCT — 科学与技术基金会拨付，项目编号为 EXPL/CCI- COM/1637/2021。

参考文献

[1] Barry Boehm。《系统工程》9.1（2006），第1–19页：“系统与软件工程过程的未来趋势及其影响”。[2] Edsger Wybe Dijkstra 等。《结构化编程笔记》。1970。[3] WattsS Humphrey。《团队软件过程（sm）简介》。Addison-Wesley Professional，2000。

[4] Charles Antony Richard Hoare。《ACM通信》12.10（1969），第576–580页：“计算机编程的公理基础”。[5] Moshe Y Vardi。“自动推理革命：从理论到实践，再回到理论”。发表于《NSF CISE 杰出讲座，春季》（2016）。[6] Leonardo de Moura 和 Nikolaj Bjørner。“Z3：高效的SMT求解器”。发表于《系统构造与分析工具及算法国际会议》。Springer，2008年，第337–340页。[7] K Rustan M Leino。“使用Dafny实现无障碍软件验证”。发表于《IEEE软件》34.6（2017），第94–97页。[8] Pascal Cuoq 等。“Frama-C”。发表于《软件工程与形式化方法国际会议》。Springer，2012年，第233–247页。[9] Jean-Christophe Filliâtre 和 Andrei Paskevich。“Why3——程序与证明器相遇之处”。发表于《欧洲编程研讨会》。Springer，2013年，第125–128页。[10] Rui Abreu 等。“使用约束诊断故障电子表格”。发表于《软件质量期刊》23.2（2015），第297–322页。[11] Francisco Rebello de Andrade 等。“面向Java泛型类的规约驱动单元测试生成”。发表于《集成形式化方法国际会议》。Springer，2012年，第296–311页。[12] José Campos 和 Rui Abreu。“将测试需求编码为约束以实现测试套件最小化”。发表于《2013年第10届信息技术：新一代国际会议》。IEEE，2013年，第317–322页。[13] Alexander Diedrich 等。“将模拟退火应用于基于模型的诊断问题”。发表于《诊断原理国际研讨会：DX-2016》。ARC-E-DAA-TN35662。DX会议系列电子书，2016年。[14] Bruno Lima、João Pascoal Faria 和 Robert Hierons。“具有时间约束的分布式测试中的局部可观测性与可控性分析及其实现”。发表于《IEEE Access》8（2020），第167172–167191页。[15] Daniel Jackson。《软件抽象：逻辑、语言与分析》。MIT出版社，2012。[16] Arthur B Kahn。“大型网络的拓扑排序”。发表于《ACM通信》5.11（1962），第558–562页。[17] K Rustan M Leino 和 Nadia Polikarpova。“经验证计算”。发表于《经验证软件：理论、工具与实验工作会议》。Springer，2013年，第170–190页。[18] Marie Farrell、Conor Reynolds 和 Rosemary Monahan。“使用Dafny解决VerifyThis 2021挑战”。发表于《第23届ACM类Java程序形式化技术国际研讨会论文集》。2021年，第32–38页。[19] Carlo A Furia、Christopher M Poskitt 和 Julian Tschannen。“Auto-Proof验证器：非专家用户的易用性及其在标准代码上的应用”。发表于《arXiv预印本 arXiv:1508.03895》（2015）。[20] James Noble 等。“编程之外的更多编程：在软件工程课程中教授形式化方法”。发表于《NASA形式化方法研讨会》。Springer，2022年，第431–450页。

案例研究的代码

A.1 整数除法

/*
*Dafny“你好，世界！”：一个用于执行*整数除法的简单程序。

*/

// 计算整数// 除法的商“q”和余数“r”：将（非—负）被除数“n”除以（正）// 除数“d”。

方法 div（n：自然数，d：自然数）返回（q：自然数，r：自然数）要求 d > 0确保 q *d +r ==n && r <d { q: = 0; r: = n; 当 r >=d 递减 r 不变量 q *d +r ==n { q: = q + 1; r: = r − d; } }

// 主程序，包含一个简单的测试用例（已静态检查！）方法 Main() { var q, r := div(15, 6); assert q == 2 && r == 3; print "q =", q, " r =", r, "\n "; }

A.2 数的幂

/*
*形式化验证一个 O(log n) 算法，用于计算
*实数的自然数次幂（x^n），展示在 Dafny 中*引理和自动归纳的用法。

*/ // 函数式风格中 x^n 的递归定义，// 时间和空间复杂度为 O(n)。函数 power (x : r e a l , n : 自然数) : r e a l { i f n == 0 then 1 . 0 e l s e x *power (x , n−1) }

```
// 在时间和空间 O(log n) 内计算 x^n。方法 powerDC(x: real, n: 自然数) returns(p:real) ensures p == power(x,n) {if n == 0{return 1.0;}elseif n == 1{return x;}elseif n % 2 == 0{幂的乘积(x,n/2,n/2); // 调用引理 var temp := powerDC(x, n/2); return temp * temp; } else { 幂的乘积(x,(n-1)/2,(n-1)/2); // 调用引理 var temp := powerDC(x,(n-1)/2); return temp * temp * x;}}
```

```
// 表示性质 x^a * x^b = x^(a+b)                    .
// 该性质通过对 'a' 进行自动归纳证明。引理{:induction a} 幂的乘积(x: real, a: 自然数, b: 自然数) 确保 power(x,a) * power(x,b) == power(x, a+b) {}
```

```
// 几个测试案例（由 Dafny 静态检查）                    .
方法 testPowerDC() { var p1 := powerDC(2.0, 5); 断言 p1 == 32.0; var p2 := powerDC(-2.0, 2); 断言 p2 == 4.0; var p3 := powerDC(-2.0, 1); 断言 p3 == -2.0; var p4 := powerDC(-2.0, 0); 断言 p4 == 1.0; var p5 := powerDC(0.0, 0); 断言 p5 == 1.0;}
```

A.3 二分查找

```
/*
 *使用 *Dafny 对二分查找算法进行形式化验证。

 */
```

```
类型 T = int // 用于演示，但也可以是其他类型
```

```
// 检查数组 'a' 是否已排序。
```

谓词 isSorted(a : 数组<T>) 读取 a {对所有 i, j : : 0 <= i < j < a . 长度 ==> a [i] <= a [j]}

// 在已排序的数组 ' a ' 中查找值 ' x ', 并返回// 其索引; 如果未找到, 则返回 −1。方法 b i n a r y S e a r c h (a : 数组<T>, x : T) 返回 (索引 : i n t) 要求 i s S o r t e d (a) 确保 (0 <= 索引 < a . 长度 && a [索引] == x) | | (索引 == −1 && x ! i n a [. .]) { var low , high := 0 , a . 长度 ; w h i l e low < high 不变式 0 <= low <= high <= a . 长度 不变式 x ! i n a [. . low]&& x ! i n a [high . .]{ var mid := low + (high − low) / 2 ; i f { c a s e a [mid] < x => l ow := mid + 1; c a s e a [mid] > x => high := mid ; c a s e a [mid] == x => 返回 mid ; } } 返回 −1; }

// 用于检查后置条件的简单测试用例。方法 testBinarySearch() { var a: new int[]; assert a[..]; // 证明辅助项 var id1: binarySearch(a, 6); assert id; var id2: binarySearch(a, 3); assert id; var id3: binarySearch(a, 4); assert id3 in {1, 2}; }

A.4 插入排序

/*
使用 Dafny 对插入排序算法进行形式化验证。

*/ // 排序算法的契约。抽象模块 Sorting { type T =int // 仅用于演示, 也可以是其他类型

// 定义契约（语义）的抽象方法// 用于数组排序。method sort(a: array<T>) modifies a ensures isSorted(a[..]) ensures multiset(a[..]) == multiset(old(a[..]))// 检查序列“a”是否已排序的辅助谓词。predicate isSorted(s: seq<T>) { forall i, j :: 0 <= i < j < |s| ==> s[i] <= s[j]} }

// 排序契约抽象模块的静态测试module TestSorting { import opened Sorting method testSortSimple() { var a: array T; assert a[..]; // prover helper! sort(a); assert a[..]; } method testSortWithDups() { var a: array T; assert a[..]; // prover helper sort(a); SortingUniquenessProp(a[..]); assert a[..]; // assertion violation (!?) } // 陈述并用归纳法证明以下性质：如果两个序列都已排序，且具有相同的多重集 // 元素，那么它们必定相同（因此排序有唯一解）。 lemma SortingUniquenessProp(a: seq T, b: seq T) requires isSorted(a) && isSorted(b) && multiset(a) == multiset(b) ensures a == b { // 回顾关于序列及其多重集的有用性质 seqProps(a); seqProps(b); // 归纳证明的关键步骤，针对 a 和 b // （其余部分由 Dafny 自动补全） if a == [] { SortingUniquenessProp(a[..], b[..]);

} }// 陈述序列的两个性质（仅由 Dafny 证明）： // — 序列拼接后再按头部和尾部拆分// — 序列的元素属于其多重集。lemma seqProps(a: seq T) ensures a == [a[0]] + a[1..] ensures forall i :: 0 <= i < |a| ==> a[i] in multiset(a) {}}module InsertionSort refines Sorting { // 使用插入排序算法对数组 a 排序。 // 继承 Sorting 中的契约。 method sort(a: array T) { for i := 0 to a.Length invariant isSorted(a[..i]) invariant multiset(a[..]) == multiset(old(a[..])) { var j := i; while j > 0 && a[j] < a[j-1] invariant forall l, r :: 0 <= l < r < i && r != j ==> a[l] <= a[r] invariant multiset(a[..]) == multiset(old(a[..])) { a[j], a[j-1] := a[j-1], a[j]; // 交换（并行赋值。） j := j - 1; } } }}

A.5 优先队列

/*
* 使用堆实现的优先队列的形式化规范与验证。堆是一个部分有序的集合

* 以数组表示，适用于实现优先队列操作 insert 和删除最大值，时间复杂度为 O(堆大小)。*
* 展示了面向对象程序的验证 —
*/

类型 T = 整数 // 仅用于演示，也可以是真实类型等。

c 类 { : 自动契约 } 优先级队列 { // 具体状态表示 var 堆 : 数组<T>; var 大小 : 自然数 ; // 配置参数 static const 初始容量 := 10; // 类不变量 (堆不变量 + 自动生成的内容 // 由 : 自动契约 生成) 谓词 有效 ( ) { 堆不变量 ( ) } // 堆不变量 谓词 { : 自动契约 false } 堆不变量 ( ) 读取 this , 堆 { // 有效大小 大小 <= 堆 . 长度 // 每个节点都小于或等于其父节点 && 对所有 i : : 1 <= i < 大小 ==> 堆 [i] <= 堆 [(i −1)/2]} // 状态抽象函数: 获取堆内容, 作为一个多重集。 函数 元素 ( ) : 多重集 < T>{ 多重集 (堆 [. . 大小]) } // 将堆初始化为空。 构造函数 ( ) 确保 为空 ( ) { 堆 := 新 T[初始容量]; 大小 := 0; } // 检查堆是否为空 谓词 方法 为空 ( ) 确保 为空 ( ) <==> 元素 ( ) == 多重集 {} { // 帮助证明后置条件 assert 元素 ( ) == 多重集 {} <==> |元素 ( ) | == 0; // 实际表达式 大小 == 0} // 在堆中插入一个值 x。

方法insert(x:T)ensures元素() ==old(元素()) +multiset{x}{ifsize ==堆.长度{grow();}// 将元素放在底部 堆[size]:=x;size:=size + 1;// 按需向上移动堆中的元素 向上堆化();}// 用于扩展堆容量的内部方法 方法 grow()requiressize ==堆.长度ensures堆.长度 > sizeensures堆[..size] ==old(堆[..size]){var oldHeap:=堆;堆:=new T[ifsize == 0theninitialCapacityelse 2 *size];foralli | 0 <=i <oldHeap.Length{堆[i]:=oldHeap[i];}}// 将底部的待处理节点向上移动的辅助方法 method{:autocontractsfalse}heapifyUp()requiressize > 0&& heapifyUpInv(size −1)modifies堆ensuresheapInv()ensuresmultiset(堆[..size]) ==old(multiset(堆[..size])){vark:=size − 1;whilek > 0 && 堆[k] >堆[(k − 1)/ 2]invariant 0 <=k <sizeinvariant heapifyUpInv(k)invariantmultiset(堆[..size]) ==old(multiset(堆[..size])){堆[k],堆[(k−1)/ 2]:=堆[(k − 1)/ 2],堆[k];k:=(k − 1)/2;}}// 在向上堆化过程中，当将索引 k 处的节点向上移动时，// 存在一些差异:

// k 的子节点相对于 k 的父节点是有序的，但 k 相对于其父节点并非有序。谓词 {:autocontractsfalse} heapifyUpInv(k: 自然数) reads this, 堆 {size <= 堆.长度 && (forall i:: 1 <= i < size && i != k ==> 堆[i] <= 堆[(i - 1) / 2

])

&& (k > 0 ==> forall i:: 1 <= i < size && (i - 1)/2 == k ==> 堆[i] <= 堆[((i - 1)/2 - 1) / 2])} // 删除并返回堆中的最大值 //（假定堆非空）。方法 deleteMax() 返回 (x: T) 要求 !为空() 确保 是最大值(x, old(元素())) 确保 元素() == old(元素()) - 多重集{x} { // 回忆引理..... maxIsAtTop(); // 从堆顶取出最大值 x := 堆[0]; // 减小大小 size := size - 1; 如果 size > 0 { // 将最后一个元素移到堆顶 heap[0] := 堆[size]; // 根据需要向下调整堆 heapifyDown(); } } // 删除并返回堆中的最大值 //（假定堆非空）。方法 geteMax() 返回 (x: T) 要求 !为空() 确保 是最大值(x, 元素()) { maxIsAtTop(); 返回 heap[0]; } // 用于检查某个值是否为最大值的辅助谓词

n

// 多重集谓词 是最大值(x: T, m: 多重集<T>) { x 属于 m && 对于所有 y:: y 属于 m ==> y <= x }

// 用于将堆顶的脏节点向下移动的辅助方法 // 在堆中 方法 { : 自动合约 false } 向下堆化() 要求 size > 0 && 向下堆化不变式(0) 修改 heap 确保 heapInv() 确保 多重集(heap [..size]) == old(多重集(heap [..size])) { var k:= 0; while true decreases size − k invariant 0 <= k < size invariant 向下堆化不变式(k) invariant 多重集(heap [..size]) == old(多重集(heap [..size])) { var leftChild:= 2 ∗k + 1; // 左子节点的索引 var rightChild:= 2 ∗k + 2; if leftChild >= size { return; // 已到达底部 } var maxChild:= if rightChild < size && heap [rightChild] > heap [leftChild]then rightChild else leftChild; if heap [k] > heap [maxChild] { return; // 已经有序 } // 向上移动并继续 heap [k] , heap [maxChild] := heap [maxChild] , heap [k] ; k:= maxChild; } } // 在向下堆化过程中，当将索引 k 处的节点向下移动时，// 存在一些差异：// k 的子节点相对于 k 的父节点是有序的，而 k 相对于其子节点 // 无序。谓词 { : 自动合约 false } 向下堆化不变式(k: 自然数) 读取 this, heap{size <= heap.Length && (对于所有 i:: 1 <= i < size && (i − 1)/2 != k ==> heap [i] <= heap [(i − 1)/2]) && (k > 0 ==> 对于所有 i:: 1 <= i < size && (i − 1)/2 == k ==> heap [i] <= heap [((i − 1)/2 − 1)/2]) } // 说明最大值位于堆顶（位置 0）的引理。deleteMax 假定此性质，并

// 由堆不变式推得。// 通过对堆的大小进行归纳证明，解释为何// 接收一个表示所考虑堆大小的参数。引理 {:induction n} maxIsAtTop(n: 自然数 := size) requires n <= size ensures forall i :: 0 <= i < n ==> heap[i] <= heap[0]{}}

// 一个简单的测试场景。方法 testPriorityQueue(){ var h := new PriorityQueue(); assert h.为空(); h.insert(2); h.insert(5); h.insert(1); h.insert(1); var x := h.删除最大值(); assert x == 5; x := h.删除最大值(); assert x == 2; x := h.删除最大值(); assert x == 1; x := h.删除最大值(); assert x == 1; assert h.为空();}

A.6 哈希集合

```
/*
 *在 Dafny 中，使用开放寻址法和线性探测实现并验证哈希集合。*

 *提供基本的集合操作（包含、插入、
 *删除），在抽象层面进行规范，并借助抽象状态变量 '元素' 表示集合内容。*

 */ // 哈希表中每个单元格内容的数据类型。// 它存储一个 T 类型的值、Nil（无值）或 Deleted（单元格已标记为删除）。datatype Cell<T> = Nil | Deleted | Some（value: T）// 哈希函数的函数类型 type HashFunction<!T> = (T) -> 自然数 // 表示由 T 类型（可进行相等性比较）的元素构成的哈希集合，即存储在哈希表中的集合。// 使用"autocontracts"属性，自动在方法的前置条件和后置–条件中注入类不变式检查和框架条件。class {:autocontracts} HashSet<T(==)>{
```

// Ghost 变量（抽象状态变量），仅用于规范说明。ghost var elems：集合<T>; // 具有内部表示的具体状态变量。var hashTable：数组 <Cell<T>>; // 要使用的哈希函数（由构造函数提供）。const hash：HashFunction<T>; // 哈希表中已使用（存有某个值）和标记为已删除的位置数。var used：自然数；var deleted：自然数；// 哈希表的初始容量。static const initialCapacity：= 101; // 用于形式化类不变式的 Ghost 谓词。predicate Valid（）{ // 定义抽象状态变量与具体状态变量之间抽象关系的约束 elems == valSet（hashTable，hashTable.Length）// 对内部状态表示的约束 && hashTable.Length > 0&& hashTableInv（hashTable）&& used == |valSet（hashTable，hashTable.Length）|&& deleted == |delSet（hashTable，hashTable.Length）| } // 检查哈希表“t”一致性的 Ghost 谓词。predicate { : autocontracts false } hashTableInv（t：数组<Cell<T>>）reads t { forall i：： 0 <= i <t.Length && t [i].Some? ==> validPos（t [i]. value，i，t）} // 检查‘i’是否为哈希表‘t’中值‘x’的有效位置的 Ghost 谓词。（‘x’当前可能存储在该位置，也可能没有。）predicate { : autocontracts false } validPos（x：T，i：自然数，t：数组<Cell<T>>）requires 0 <= i < t.Length reads t { var h：= hash（x）% t.Length;

h == i||(h <i&&forallj::h <=j <i==>t[j]!=Nil&&t[j]!=Some(x))||(h >i&&forallj::h <=j <t.长度 || 0 <=j <i==>t[j]!=Nil&&t[j]!=Some(x))}

// 用于获取哈希表 't' 前 'n' 个位置中存储的值集合的幽灵函数。function {:autocontracts false} valSet(t: array<Cell<T>>, n: 自然数): set<T> requires 0 <=n <=t.Length reads t {set i | 0 <=i <n&&t[i].Some?::t[i].value} // 用于获取哈希表 't' 前 'n' 个位置中标记为 Deleted 的位置集合的幽灵函数。function {:autocontracts false} delSet(t: array<Cell<T>>, n: 自然数): set<自然数> requires 0 <=n <=t.Length reads t {set i | 0 <=i <n&&t[i].Deleted?} // 用于获取哈希表 't' 前 'n' 个位置中标记为 Nil 的位置集合的幽灵函数。function {:autocontracts false} nilSet(t: array<Cell<T>>, n: 自然数): set<自然数> requires 0 <=n <=t.Length reads t {set i | 0 <=i <n&&t[i].Nil?} // 辅助引理，说明以下性质：对于有效的哈希表，valSet、delSet 和 nilSet 的大小之和等于哈希表（数组）的长度。// 这是因为哈希表不变量蕴含存储的值没有重复。// 证明通过对表的长度进行归纳完成（省略由 Dafny 补全的步骤）。lemma {:autocontracts false} countingLemma(ht: array<Cell<T>>, len: 自然数, v: 自然数, d: 自然数, n: 自然数) requires 0 <=len <=ht.Length requires hashTableInv(ht) requires v ==|valSet(ht, len)|&&d ==|delSet(ht, len)|&&n ==|nilSet(ht, len)|ensures v +d +n ==len{if len >0{

var vs: = valSet (ht, len); var ds: = delSet (ht, len); var ns: = nilSet (ht, len); var vs1: = valSet (ht, len − 1); var ds1: = delSet (ht, len − 1); var ns1: = nilSet (ht, len − 1); // 递归部分 countingLemma (ht, len − 1, | vs 1 | , | ds 1 | , | ns 1 |); // 递增部分 match ht [len − 1] { case Deleted => assert vs == vs1 && ds == ds 1 + {len − 1} && ns == ns1; case Nil => assert vs == vs1 && ds == ds1 && ns == ns 1 + {len − 1}; case Some (x) => assert vs == vs 1 + {x} && ds == ds1 && ns == ns1; } } }

// 内部谓词，用于检查哈希表是否“已满”，也就是说，所有位置都已存有值或标记为已删除（即没有位置为 Nil）。在这种情况下，可能无法插入新值。predicate method full() ensures full() <==> nilSet(hashTable, hashTable.Length)=={} { // 帮助证明后置−条件（等价关系）： countingLemma(hashTable, hashTable.Length, used, deleted, |nilSet(hashTable, hashTable.Length) |); // 实际函数值： used + deleted == hashTable.Length } // 公共方法，检查此集合是否包含值 x。method contains(x: T) returns (res: bool) ensures res <==> x i n elems { var pos := locate(x); return pos != −1 && hashTable [pos] == Some(x); } // 内部方法，用于确定值“x”（已存在或待插入）的位置（“pos”）。// 如果找不到这样的位置（因为表已满），则返回 −1。// 对于新值，尝试重用标记为已删除的位置。method locate(x: T) returns (pos: int)

```
    requires Valid() ensures x in elems ==> 0 <= pos < hashTable.Length && hashTable[pos] == Some(x) ensures x !in elems ==> (pos == -1 && full()) || (0 <= pos < hashTable.Length && !hashTable[pos].Some? && validPos(x, pos, hashTable)) { var h := hash(x) % hashTable.Length; var reuse := -1; for i := h to hashTable.Length invariant forall j :: h <= j < i ==> hashTable[j] != Nil && hashTable[j] != Some(x) invariant reuse == -1 || (h <= reuse < i && hashTable[reuse] == Deleted) { if hashTable[i] == Nil || hashTable[i] == Some(x) { return i; } if hashTable[i] == Deleted && reuse == -1 { reuse := i; } } for i := 0 to h invariant forall j :: 0<=j<i || h<=j<hashTable.Length ==> hashTable[j] != Nil && hashTable[j] != Some(x) invariant reuse == -1 || ((0 <= reuse < i || h <= reuse < hashTable.Length) && hashTable[reuse] == Deleted) { if hashTable[i] == Nil || hashTable[i] == Some(x) { return i; } if hashTable[i] == Deleted && reuse == -1 { reuse := i; } } return reuse; } // 公共构造函数，接收要使用的哈希函数，并将集合初始化为空。constructor(hash: HashFunction<T>) ensures elems == {} { // 初始化具体状态变量 this.hash := hash; hashTable := new Cell<T>[initialCapacity](_ => Nil); used := 0; deleted := 0;
```

// 初始化 ghost/抽象状态变量 elems :={ } ; } // 内部方法，将新值 'x' 插入哈希集合，保证集合未满。方法 insertAux (x : T) 要求 x !在 elems 中 要求 !满 ( ) 确保 elems == old (elems) +{x} 确保 deleted <= old (deleted) // 对重新哈希有用? 确保 hashTable == old (hashTable) // 对重新哈希有用? { var i := 定位 (x) ; 如果 hashTable [i] == Deleted { // 帮助证明 deleted > 0断言 i 在 delSet (hashTable , hashTable . Length) 中 ; // 现在可以递减 deleted :=deleted − 1; } hashTable [i]:=Some (x) ; used := used + 1; elems :=elems +{x } ; // 帮助证明 elems ==valSet ( ) 断言对于所有 k : : 0 <=k <hashTable . Length & & k !=i==>hashTable [k] ==old (hashTable [k]) ; // 帮助证明 deleted == | delSet ( ) |断言 delSet (hashTable , hashTable . Length) ==old (delSet (hashTable , hashTable . Length)) −{ i } ; } // 内部方法，用于扩展并清理哈希表。方法 rehash ( ) 确保 !满 ( ) 确保 elems ==old (elems) { var oldTable := hashTable ; hashTable :=new Cell <T>[hashTable . Length ∗2+1](_ =>Nil) ; deleted := 0; used := 0; elems :={ } ; // 还需要更新由 // : autocontracts 生成的 ghost 变量 'Repr'，以便能在有效状态下调用 InsertAux Repr :={ this , hashTable } ;

使用 Dafny 3 开发经验证程序的案例研究1

对于 i := 0 从 0 到 oldTable . Length 不变式 elems == valSet (oldTable , i) 不变式 deleted == 0// 证明 ! 满 ( ) 不变式 oldTable 不在 Repr 中 // 确保 insertAux 不会改变它 不变式 Valid ( ) // 确保维持一致性 不变式 oldTable . Length < hashTable . Length // 证明 !满 ( ) 不变式 fresh (Repr - old (Repr)) // 使 insertAux 能够修改新的 hashTable 不变式 oldTable . Length < hashTable . Length // 证明 !满 ( ) 不变式对于所有 k : : 0 <= k < oldTable . Length ==> oldTable [k] == old (hashTable [k]) // 证明后置—条件 { // 帮助证明 !满 ( ) countingLemma (oldTable , i , |valSet (oldTable , i) |,|delSet (oldTable , i) |, |nilSet (oldTable , i) |) ; 如果 (oldTable [i]. Some ?) { insertAux (oldTable [i]. value) ; } } // 帮助证明 !满 ( ) var i := oldTable . Length ; countingLemma (oldTable , i , |valSet (oldTable , i) |,|delSet (oldTable , i) |, |nilSet (oldTable , i) |) ; } // 将新值 'x' 插入此哈希集合。方法 insert (x : T) 要求 x !在 elems 中 确保 elems == old (elems) + {x} { 如果满 ( ) { rehash ( ) ; } insertAux (x) ; } // 从此哈希集合中删除已有值 'x'。方法 delete (x : T) 要求 x 在 elems 中 确保 elems == old (elems) - {x} { var h := hash (x) % hashTable . Length ; var i := 定位 (x) ;

元素 := 元素 − {x}; 哈希表 [i]:= 已删除; 已删除 := 已删除 + 1; 已使用 := 已使用 − 1; // 帮助证明元素 == 值集合(哈希表, // 哈希表.长度) 断言 对所有 k:: 0 <= k < 哈希表.长度 && k != i ==> 哈希表 [k] == 旧(哈希表 [k]); // 帮助证明已删除 == |删除集合() |断言 删除集合(哈希表, 哈希表.长度) == 旧(删除集合(哈希表, 哈希表.长度)) + {i}; }}

```
方法 testHashSet(){ var h := new 哈希集合<string >(x => |x |); assert h.元素 == {}; h.insert("Hello"); assert h.元素 == {"Hello"}; h.insert("World"); assert h.元素 == {"Hello", "World"}; var found := h.contains("Hello"); assert found; found := h.contains("ANSI"); assert! found; h.delete("Hello"); assert h.元素 == {"World"}; found := h.contains("Hello"); assert! found; }
```

A.7 树集合

```
/*
*使用二叉搜索树（BST）实现有序集合的规范与验证。

*说明了幽灵变量在数据方面的用法
*抽象化以及规范与*实现的分离。

*使用 {:autocontracts} 属性处理类不变量的强制执行及框架生成（读取/修改）。

*/ 类型 T = int // 用于演示 // 不含重复项的
序列
```

使用 Dafny 3 开发经验证程序的案例研究

类型 useq<T> = s : 序列<T> |! 包含重复项 (s) // 检查序列 ' s ' 是否包含重复项。谓词 hasDuplicates <T>(s : 序列<T>) { 存在 i , j : : 0 <= i < j < |s | && s [i] == s [j]} // 二叉搜索树节点类 { : 自动契约 } 二叉搜索树节点 { // 具体实现变量 var 值 : T // 此节点中的值 var 左 : 二叉搜索树节点? // 小于 ' value ' 的元素 var 右 : 二叉搜索树节点? // 大于 ' value ' 的元素 // (? – 可能为 null) // 用于规范和验证的抽象变量 // 目的 ghost var 元素 : 集合 <T>// 子树中的值集合 // 从此节点开始（包括此值） // 包含上述变量完整性约束的类不变量 谓词 有效 ( ) { (元素 == { 值 } + (如果 左 == null 则 {} 否则 左.元素) + (如果 右 == null 则 {} 否则 右.元素)) && (左 != null ==> 对所有 x : : x 属于 左.元素 ==> x < 值) && (右 != null ==> 对所有 x : : x 属于 右.元素 ==> x > 值) && (右 != null ==> 对所有 x : : x 属于 右.元素 ==> x > 值) && (左 != null ==> 左.有效 ( )) && (右 != null ==> 右.有效 ( )) && (左 != null && 右 != null ==> 左.表示 !! 右.表示) // 左、右子树中的对象集合互不相交 // 确保对一个子树中的节点进行修改 // 不会影响另一个子树 ! } // 使用值 ' x ' 和空的左、右子树初始化新节点。构造函数 (x : T) 确保 元素 == {x} { 值 := x ; 左 := null ; 右 := null ; 元素 := {x } ;

} // 检查以此节点为起点的子树是否包含值 'x'。运行时间为 O(log h)，其中 'h' 是子树的高度。predicate method contains (x : T) ensures contains (x) <==> x in elems { if x == value then true else if x < value && left != null then left . contains (x) else if x > value && right != null then right . contains (x) else false } // 在以此节点为起点的子树中插入值 'x'。如果该值已存在，则不执行任何操作。运行时间为 O(log h)，其中 h 是子树的高度。method insert (x : T) ensures elems == old (elems) + {x} decreases elems { if x == value { return; } else if x < value { if left == null { left := new BSTNode(x); } else { left . insert (x); } } else { if right == null { right := new BSTNode(x); } else { right . insert (x); } } elems := elems + {x }; } // 用于查找此子树中最大值的公共函数。运行时间为 O(log h)，其中 'h' 是子树的高度。function method max () : T ensures max () in elems && forall x :: x in elems ==> x <= max () {

if right == null then value else right . max () } // 用于查找此子树中最小值的公共函数。运行时间为 O(log h)，其中 'h' 是子树的高度。function method min () : T ensures min () in elems && forall x :: x in elems ==> x >= min () { if left == null then value else left . min () } // 从以此节点为起点的子树中删除值 'x'，并返回该子树的新头部（如果 'x' 是子树中的唯一值，则返回 null）。如果该值不存在，则不执行任何操作。目前看来，运行时间为 O(log h log h)，其中 'h' 是子树的高度。method delete (x : T) returns (res : BSTNode?) ensures if old (elems) == {x} then res == null else res != null && res . elems == old (elems)−{x} && res . Valid () // 非自动合约添加..... && res . Repr <= old (Repr) // 以保持互不相交..... decreases elems { if x == value { if left == null { res := right; // 仅更改头部 return; } else if right == null { res := left; // 仅更改头部 return; } else { if */* 非确定性选择 */ { value := left . max (); left := left . delete (value); } else { value := right . min (); right := right . delete (value); } } } else if x > value && right != null { right := right . delete (x);

```
    }elseifx <value&&left!=null{left:=left.
delete(x);}res:=this;元素:=元素 −{x};}方法
asSeq()returns(s:useq<T>)ensuresisSorted(s)
&& asSet(s) == 元素 decreases 元素{var l, m, r:
= [], [value], [];ifleft!=null{l:=left.asSeq()
;}ifright!=null{r:=right.asSeq();}asSetProp(l,
m);//recall引理 asSetProp(l +m, r);//recall引理 r
eturnl +m +r;}}
```

```
// 一个辅助函数，用于获取序列中的元素集合。function asSet(s: seq<T>): set<T>
ensures forall i :: 0 <= i < |s| ==> s[i] in asSet(s)ensures forall x :: x in asSet(s)
==> x in s{ if |s| == 0 then {} else {s[0]} + asSet(s[1..])}// 一个辅助谓词，用于检查
序列是否严格递增。predicate isSorted(s: seq<T>) { forall i, j :: 0 <= i < j < |s|
==> s[i] < s[j]}// 引理通过归纳法陈述并证明以下性质：序列拼接所得的元素集合，等
于各个序列的元素集合的并集。lemma asSetProp(s1: seq<T>, s2: seq<T>)
ensures asSet(s1 +s2) == asSet(s1) + asSet(s2){ if |s1| > 0| > 0 { assert s
1 == [0..1] + s1[1..] == s1] +; assert (s1[0..1] + s1] +[1..]) + s1[1[0..1] +
s|[1..] == s +[0..1] + (s1]+[1..] + (s1[1. + s2)); asSetProp(s1[1[1..] + s2); }
else { assert s [] + == s2; }}
```

```
}

// 引理通过归纳法陈述并证明以下性质：如果两个不含重复元素的序列都是有序的，
且具有相同的元素集合，那么它们必定相同。lemma sortingUniqueness(a: seq<
T>, b: seq<T>)requires isSorted(a) && isSorted(b) && asSet(a) ==
asSet(b)ensures a == b{ if |a| > 0 { sortingUniqueness(a[1.., b[1..]); }}

// 一个简单的测试用例。方法 testSortedSet() { var s :=
new 二叉搜索树节点(2); s.insert(5); s.insert(1);
s.insert(4); s.insert(4); var t := s.asSeq();
sortingUniqueness(t, [1, 2, 4, 5]); // 帮助证明下
一个断言 assert t == [1, 2, 4, 5]; assert s.
min() == 1; assert s.max() == 5; var s2 :=
s.delete(5); assert s2.elems =={1, 2, 4}; }
```

A.8 稳定婚姻

```
/*
* 使用 Dafny 对 Gale–Shapley 算法进行形式化验证
* 以解决稳定婚姻问题，两者均有描述，见
* https://en . 维基百科 . org/wiki/稳定婚姻_问题_ .
* 然后，将该算法应用于解决教师们的问题。
* 分配问题。该问题曾在 2004 年给葡萄牙* 带来严重困扰。

*/ // 无重复元素的序列类型 useq<T> = s : 序列<T> |!
hasDuplicates ( s )

// 注入映射类型 inmap<K, V> = m: map<K, V> | 是注入的 (m)
```

// 检查序列 s 是否包含重复元素。predicate hasDuplicates <T>(s: seq<T>) { 存在 i, j :: 0 <= i < j < |s| && s[i] == s[j] }// 检查映射 m 是否为注入映射，即不同的键映射到不同的值。predicate isInjective <K, V>(m: map<K, V>) { 对于所有 i, j :: i 属于 m && j 属于 m && i != j ==> m[i] != m[j] }// 检查元素 e1 是否在序列 s 中位于 e2 之前。predicate method precedes <T(==)>(e1: T, e2: T, s: seq<T>) { 存在 i, j :: 0 <= i < j < |s| && s[i] == e1 && s[j] == e2 }// 获取序列中元素的集合。function elems<T>(s: useq<T>): set<T>ensures 对于所有 x :: x 属于 elems(s) ==> x 属于 sensures 对于所有 x :: x 属于 s ==> x 属于 elems(s) { set i | 0 <= i < |s| :: s[i] }// 检查情侣匹配是否有效，即只有当男性和女性都出现在彼此的偏好列表中时，他们才能订婚。predicate isValid <Man, Woman> (couples: inmap<Man, Woman>, menPrefs: map<Man, useq<Woman>>, womenPrefs: map<Woman, useq<Man>>) { 对于所有 m :: m 属于 couples ==> var w := couples[m]; m 属于 menPrefs && w 属于 womenPrefs && w 属于 menPrefs[m] && m 属于 womenPrefs[w] }// 检查情侣匹配是否稳定，即不存在一对 (m, w)，彼此都比当前配对更喜欢对方。predicate isStable <Man, Woman> (couples: inmap<Man, Woman>, menPrefs: map<Man, useq<Woman>>, womenPrefs: map<Woman, useq<Man>>) { ! 存在 m, w :: m 属于 menPrefs.Keys && w 属于 womenPrefs.Keys && unstable(m, w, couples, menPrefs, womenPrefs) }

谓词 不稳定 <男性, 女性>(m: 男性, w: 女性, 配对: 映射 <男性, 女性>, 男性偏好 : 映射 < 男性 , 序列 < 女性 >>, 女性偏好 : 映射 < 女性 , 序列 < 男性 >>) 要求 m 属于 男性偏好 . 键 && w 属于 女性偏好 . 键 { w 属于 男性偏好 [m] && m 属于 女性偏好 [w] && (m 属于 配对 ⟹ 先于 (w, 配对 [m] , 男性偏好 [m])) && (对于所有 m' : : m' 属于 配对 && 配对 [m'] == w⟹ 先于 (m, m' , 女性偏好 [w])) }

// 使用 Gale–Shapley 算法进行稳定匹配，列表不完整且不存在平局。// 接收男性和女性的偏好列表，并返回生成的配对。// 时间复杂度（使用合适的数据结构时）为 O(|M| ∗ |W|)，其中 W 是女性集合，M 是男性集合。// Man 和 Woman 类型被定义为类型参数，因为它们的内部结构与此处无关。method stableMatching <Man, Woman>(menPrefs : map<Man, useq <Woman>>, womenPrefs : map < Woman, useq <Man>>) 返回 (couples : inmap <Man, Woman>) // P1：男性偏好中引用的女性必须存在 requires for all m :: m in menPrefs⟹ for all w :: w in menPrefs [m] ⟹ w in womenPrefs // P2：女性偏好中引用的男性必须存在 requires for all w :: w in womenPrefs⟹ for all m :: m in womenPrefs [w] ⟹ m in menPrefs // Q1：只有在双方的偏好列表中都提及对方时，男性和女性才能配对 ensures isValid (couples, menPrefs , womenPrefs) // Q2：稳定婚姻（及极大性） ensures isStable (couples, menPrefs , womenPrefs) { // 将结果初始化为空配对 couples := map [ ] ; // 将已探索的男性偏好列表初始化为空 var menPrefsExplored := map m | m in menPrefs :: [ ] ; // 用于借助 Dafny 证明终止性的 Ghost 变量 //（而非结构过于复杂的 menPrefsExplored） ghost var unexploredPairs := set m, w | m in menPrefs && w in menPrefs [m] :: (m, w) ; // 当存在一个仍有女性 w 可供求婚的自由男性 m 时

```
// 向 w 求婚 while exists m :: m in menPrefs && m !in couples &&
menPrefsExplored [m] < menPrefs [m] decreases unexploredPairs // I1:
menPrefsExplored 与 menPrefs 具有相同的键（男性） invariant
menPrefs.Keys == menPrefsExplored.Keys // I2: menPrefsExplored 中的
列表必须是 menPrefs 中的子列表（前缀） invariant for all m :: m in
menPrefsExplored==> menPrefsExplored [m] <= menPrefs [m] // I3: 为
逐步确保 Q1，使用 menPrefsExplored 而非 menPrefs invariant isValid (
couples, menPrefsExplored , womenPrefs ) // I4: 为逐步确保 Q2，使用
menPrefsExplored 而非 menPrefs invariant isStable ( couples,
menPrefsExplored , womenPrefs ) // I5: 已配对的男性不会再向列表中靠后的
女性求婚（这是保持 isStable 所必需的） invariant for all m :: m in couples
==> couples [m] == last ( menPrefsExplored [m] ) // I6: 定义
unexploredPairs 内容的循环不变式 invariant unexploredPairs == set m,
w | m in menPrefs && w in menPrefs [m]&& w !in menPrefsExplored [
m] :: (m, w) { // 选择一个满足此条件的男性（仍有女性 w 可供求婚的自由男性
m） var m : | m in menPrefs && m !in couples && menPrefsExplored [m
] < menPrefs [m]; // 选择 m 偏好列表中的下一位女性（使用辅助函数绕过
Dafny 的限制） var w := nth ( menPrefs [m] , | menPrefsExplored [m] | ) ;
// 如果 w 并非自由（即已有一对 (m', w) ） if m' : | m' in couples &&
couples [m' ] == w { // 如果 w 更喜欢 m 而非 m' if m in womenPrefs [w]&
& precedes (m, m', womenPrefs [w] ) { // m' 恢复自由 couples := map x |
x in couples && x != m' :: couples [x ]; // (m, w) 配对 couples := couples [
m := w ]; } } else // w 是自由的
```

{ // 如果男性 m 对 womenPrefs 中的 w 感兴趣
[w]{ // （m, w）成为一对 couples:=couples [
m:=w]; } }

// 将此配对标记为已探索 menPrefsExplored:=menPrefsExplored [m:=
menPrefsExplored [m] + [w]]; 未探索的配对: =un explored Pairs −{(m,
w)}; } } /* 稳定婚姻问题的一些测试用例。

*/ 方法 testStableMatching1() { var menPrefs:=映射 [1:= [1, 2], 2 :
= [1, 2]]; var womenPrefs:=映射 [1:= [1], 2:= [2]]; var
expectedCouples:=映射[1:= 1, 2:= 2]; var actualCouples:=
stableMatching(menPrefs, womenPrefs); assert isValid(expectedCouples,
menPrefs, womenPrefs); // 证明辅助程序..... assert actualCouples ==
expectedCouples; }

方法 测 试 稳 定 配 对 2 ( ) { var 男性偏好 :=映射 [1:= [2, 1], 2:= [1, 2]];
var 女性偏好 :=映射 [1:= [1, 2], 2:= [2, 1]]; var 预期配对1 :=映射 [1:
= 2, 2:= 1]; var 预期配对2 :=映射 [1:= 1, 2:= 2]; var 实际配对 :=稳定配对
(男性偏好 , 女性偏好) ; 断言 有效 (预期配对1 , 男性偏好 , 女性偏好) ; // 证明辅助工
具 . . . 断言 实际配对 ==预期配对1| | 实际配对 ==预期配对2 ; }

方法 t e s t S t a b l e M a t c h i n g 3 ( ) { var 男性偏好 :=映射 [1:= [1, 2], 2:
= [1]]; var 女性偏好 :=映射 [1:= [1, 2], 2:= [2, 1]]; var 预期配对 :=
映射 [1:= 2, 2:= 1]; var exp ec ted Cou pl es2 :=映射 [1:= 1]; var a c t u a l
C o u p l e s :=s t a b l e M a t c h i n g (男性偏好 , 女性偏好) ; a s s e r t i s V a l i
d (预期配对 , 男性偏好 , 女性偏好) ; // p r o o f h e l p e r . . .

assertactualCouples == expectedCouples1||actualCouples == expectedCouples2;} /* 解决教师职位分配问题的应用。

*/ 类型 Teacher = 自然数类型 Vacancy = 自然数 // 辅助函数：将序列 's'（不含重复元素）中的元素 'x' 移到序列头部。函数方法 moveToHead<T==(>(s:useq<T>, x:T):useq<T>要求 x 属于 s 确保对于所有 y: y 属于 moveToHead(s,x){ var i:| 0 <= i < |s| && s[i] == x; [s[i]]+s[..i]+s[i+1..]} // 获取序列中的最后一个元素 函数 last<T>(s:seq<T>):T 要求 |s| > 0{s[|s|-1]} // 获取序列中的第 n- 个元素 函数方法 nth<T>(s:seq<T>, n:自然数):T 要求 0 <= n < |s|{s[n]} // 辅助谓词：根据教师排名列表、初始安排和最终安排，检查教师 't' 是否优先于当前占据空缺职位 'v' 的教师（如果有）。// 最初占据 'v' 的教师优先级高于其他所有人；否则，排名较高的教师优先。谓词方法 teacherHasPrecedenceForVacancy(t:教师, v:空缺职位, finalPlacement:inmap<教师, 空缺职位>, teachers:useq<教师>, initialPlacement:inmap<教师, 空缺职位>){ if t2:|t2 in finalPlacement && finalPlacement[t2] == v then t!=t2 && ((t,v) in initialPlacement.Items||((t2,v)!initialPlacement.Items && precedes(t,t2,teachers))) else true // 空缺职位仍然空缺，因此任何教师都比继续空缺要好 }

// 教师安排问题的解法：将其归约为稳定婚姻问题。// 输入参数：// v a c a n c i e s －可用空缺职位集合（包括当前由希望调换职位的教师占据的位置）// t e a c h e r s －按排名顺序排列的教师集合（表示为无重复元素的序列）// p r e f e r e n c e s －映射，指明每位教师按顺序排列的空缺职位偏好列表 // i n i t i a l P l a c e m e n t －映射，表示教师的初始安排 // 输出参数：// f i n a l P l a c e m e n t －教师最终安排 方法 t e a c h e r s P l a c e m e n t (v a c a n c i e s : s e t <空缺职位 >, t e a c h e r s : useq<教师 >, p r e f e r e n c e s : map<教师 , useq<空缺职位 >>, i n i t i a l P l a c e m e n t : inmap <教师 , 空缺职位 >) 返回 (f i n a l P l a c e m e n t : inmap<教师 , 空缺职位 >) // P1：排名序列中的教师与拥有偏好的教师相同 要求 elems (t e a c h e r s) == p r e f e r e n c e s . 键 // P2：教师偏好中提到的空缺职位必须存在于空缺职位集合中 要求对于所有 t：t 属于 p r e f e r e n c e s ==> elems (p r e f e r e n c e s [t]) <= 空缺职位 // P3：初始安排中提及的教师和空缺职位必须分别存在于教师集合和空缺职位集合中 要求对于所有 t：t 属于 i n i t i a l P l a c e m e n t ==> t 属于 t e a c h e r s && i n i t i a l P l a c e m e n t [t] 属于空缺职位 // P4：教师的初始安排必须作为最后一项偏好出现在其偏好列表中 要求对于所有 t：t 属于 i n i t i a l P l a c e m e n t ==> i n i t i a l P l a c e m e n t [t] 属于 p r e f e r e n c e s [t] && i n i t i a l P l a c e m e n t [t] == 等于最后一项 (p r e f e r e n c e s [t])

// Q1：最终分配中提到的教师必须 // 存在于教师集合中

确保最终位置 . 键 <= 元素（教师）
// Q2：教师只能被安排到其偏好列表中列出的空缺职位，这意味着对于所有 t：t 在最终分配中 ==> 最终分配 [t] 在偏好列表中 [t] // Q3：该分配是稳定的，即不存在一对教师 t 和空缺职位 v，使得 t 在其偏好列表中，并且相比当前情况更偏好 v（原因可能是 t 当前空闲，也可能是 t 更偏好 v 而非已分配的职位），同时 v 更偏好 t 而非当前

// 情况（原因可能是 v 空缺，因此相比继续空缺，更偏好任何教师；或者 t 是最初被安排到该职位的教师，目前没有占据 v；或者目前占据 v 的教师 t' 最初并未被安排在那里，且其排名低于 t）

确保！存在 t::t ∈ 教师 && v ∈ 偏好 [t]&& (t ∈ 最终安排 ⟹ 优先于 (v, 最终安排 [t] , 偏好 [t])) // t 偏好 v && teacherHasPrecedenceForVacancy (t , v , 最终安排 , 教师 , 初始安排) // Q4：具有初始职位的教师也必须有一个最终职位确保对于所有 t::t ∈ 初始安排⟹ t ∈ 最终安排 { // 将问题归约为稳定婚姻问题，其中 // 教师作为男性（拥有给定的偏好），// 空缺职位作为女性，而每个空缺职位的偏好 // 由教师的排名列表给出，并将最初安排在该职位的教师（如果有）移至头部最终安排：=稳定婚姻 (偏好 , 空缺职位偏好 (空缺职位 , 教师 , 初始安排))； } // 每个空缺职位的偏好由教师的排名列表给出，// 并将最初安排在该职位的教师（如果有）移至头部函数 方法 空缺职位偏好 (空缺职位：集合 <空缺职位 >, 教师：useq<教师 >, 初始安排：inmap <教师, 空缺职位 >): map<教师, seq<空缺职位>>要求对于所有 t::t ∈ 初始安排 ⟹ t ∈ 教师 && 初始安排 [t] ∈ 空缺职位 { map v |v ∈ 空缺职位::如果 t: | t ∈ 初始安排 && 初始安排 [t] == v 则 moveToHead (教师 , t) 否则 教师 } /* 教师安排问题的一些测试用例。

*/ 方法 test1TP ( ) { var v a c a nc i e s :={1 , 2} ; var t e a c h e r s : = [1 , 2 , 3]; var p r e f e r e n c e s :=映射 [1:= [2 , 1], 2 := [1 , 2], 3 : = [2]]; var i n i t i a l P l a c e m e n t :=映射 [1:= 1];

varexpectedVacanciesPrefs:=映射[1:= [1,2,3],2:
= [1,2,3]];varexpectedFinalPlacement:=映射 [1:= 2,2:
= 1];varactualFinalPlacement:=teachersPlacement(v
acancies,teachers,preferences,initialPlacement);as
sertisValid(expectedFinalPlacement,preferences,expected
VacanciesPrefs);//proofhelper...assert!unstable(1,1,
expectedFinalPlacement,preferences,expectedVacanciesPr
efs);//proofhelper...assertactualFinalPlacement ==e
xpectedFinalPlacement;}

方法 test2TP(){varvacancies:={1,2};varteachers:= [1, 2, 3];
varpreferences:=map [1:= [2, 1],2:= [1, 2],3:= [2, 1]];
varinitialPlacement:=map [3:= 1];varexpectedVacancie
sPrefs:=map[1:= [3, 1, 2],2:= [1, 2, 3]];varexpectedFin
alPlacement:=map [1:= 2,3:= 1];assertmoveToHead(teache
rs,3) == [3, 1, 2];//proofhelpervaractualFinalPlaceme
nt:=teachersPlacement(vacancies,teachers,preferen
ces,initialPlacement);assertisValid(
expectedFinalPlacement,preferences,expectedVacanciesPre
fs);//proofhelper...assert!unstable(1,1,
expectedFinalPlacement,preferences,expectedVacanciesPre
fs);//proofhelper...assertactualFinalPlacement ==ex
pectedFinalPlacement;}

A.9 拓扑排序

/*
经典拓扑排序算法（Kahn 算法）的正确性证明，经过简化，以 Dafny 编写。

*/ // 定义一个有向图，其顶点可以是任意类型 T，// 该图由一对 (V, E) 构成，其中 V 是顶点–集，E 是// 边–集。// 每条有向边都由一对顶点表示。数据类型 Graph< T> =Graph (V: 集合 <T>, E : 集合 <(T,T)>) // 检查 G 是否定义了有效图（检查 E 是否是 V*V 的子集）。谓词 validGraph <T>(G: Graph<T>) { 对所有 e : e 属于 G. E ==> e . 0 属于 G.V && e . 1 属于 G.V

} // 检查图是否无环。谓词 acyclic <T>(G: Graph<T>) { 不存在 v : v 属于 G.V && 存在 SimplePath (G, v , v) } // 检查图 G 中是否存在一条从顶点 u 到// 顶点 v 的非–空简单路径。（目前，“简单”指没有// 重复的边，但也可以指没有重复的顶点。）谓词 existsSimplePath <T>(G: Graph<T>, u : T, v : T) decreases G. E { (u , v) 属于 G. E | | 存在 e : e 属于 G. E && e . 0 == u && 存在 SimplePath (Graph (G. V, G. E–{e }) , e . 1 , v) } // 从图 G 中移除顶点 v 及与其相连的边。函数 method removeVertex< T>(v : T, G: Graph<T>) : Graph<T>{ Graph (G.V –{v } , 集合 e |e 属于 G. E && e . 0 != v && e . 1 != v) } // 检查顶点序列是否是图 G 的顶点的拓扑排序。谓词 isTopSorting <T>(s : 序列<T>, G: Graph<T>) requires validGraph (G) { multiset (s) == multiset (G.V) && 对所有 i, j : 0 <= i <= j < |s | ==> (s [j] , s [i]) ! 属于 G. E } // 检查图 G 中的顶点 v 是否有入边。谓词 method hasIncomingEdges <T>(G: Graph<T>, v : T) requires v 属于 G.V { 存在 u : u 属于 G.V && (u , v) 属于 G. E } // 对无环有向图的顶点进行拓扑排序。返回一个序列（线性排序），其中顶点按拓扑顺序排列。方法 topsort <T>(G: Graph<T>) returns (s : 序列<T>) requires validGraph (G) && acyclic (G) ensures isTopSorting (s, G) { s : = [ ]; var R :=G; // 剩余图

w h i l e R.V !={} // s 与 R 之间的关系（基本上，R ={G −s）不变式 R == Graph（集合 v |v ∈ G.V 且 v !∈ s，集合 e |e ∈ G.E 且 e.0 !∈ s 且 e.1 !∈ s）// s 是 G 的拓扑排序 −R 不变式多重集（s） ==多重集（G.V −R.V）不变式对于所有 i, j :: 0 <= <= < | |i j s（s [j] ,s [i]）!∈ G.E // 不存在从 R 中的顶点指向 s 中顶点的边 不变式对于所有 i :: 0 <= < | |i s 对于所有 v :: v ∈ R.V ⟹（v, s [i]）!∈ G.E 减少 R.V { // 回顾性质：无环图的子图也是 // 无环 lemmaAcyclicSubgraph（R, G）； // 回顾性质：非空无环图中必定 // 存在一个没有入边的顶点 lemmaAcyclicIndegrees（R）； // 选取一个没有入边的顶点 var v : |v ∈ R.V && ! hasIncomingEdges（R, v）； // 将其追加到结果中：=s + [v]； // 从图中移除该顶点及其出边 R: =removeVertex（v, R）； } } /** 第二个引理 ***/ // 陈述并通过反证法证明以下性质：// 非空无环图至少有一个没有入边（入度为 0）的顶点。lemma lemmaAcyclicIndegrees <T>（G: Graph<T>）requires validGraph（G）&& G.V !={} && acyclic（G）ensures 存在 v :: v ∈ G.V && ! hasIncomingEdges（G, v）{ // 为了导出矛盾，假设所有顶点 // 都有入边。若对于所有 v :: v ∈ G.V ⟹hasIncomingEdges（G, v）{ // 那么可以构造任意长度的路径，其中可能 // 重复边和顶点，即一条长度为 // |G.V| + 1的路径 var p :=genPath（G, |G.V| + 1）； // 这样的路径必定包含重复顶点，并且

// 因此至少存在一个环 lemmaPathLen（G, p）； // 所以该图不是无环图，这与 // 前提条件相矛盾 assert ! acyclic（G）； } }

// 在非空图 G 中生成一条长度为 n 的有效路径，其中所有顶点都有入边。// 由于这一性质，可以构造任意长度的路径（边和顶点可能重复）。lemma genPath<T>(G: Graph<T>, n : nat) returns (p : seq<T>) requires validGraph (G) && G.V !={} requires forall v :: v in G.V ==> hasIncomingEdges (G, v) ensures |p| ==n && validPath (p , G) { p := []; while |p| <n invariant |p| <=n && validPath (p , G) { var u : |u in G.V && (p == [] ||(u , p [0]) in G. E) ; p := [u] +p ; } } // 检查顶点序列 p 是否定义了图 G 中的一条有效路径（允许顶点和边重复）。predicate method validPath<T>(p : seq<T>, G: Graph<T>) { forall i :: 0 <=i < |p| ==>p [i]in G.V && (i < |p| - 1 ==>(p [i] ,p [i +1]) in G. E) } // 陈述并证明如下性质：给定图 G 及其中的路径 p，如果 p 的长度超过顶点数，那么 G 中存在环。lemma lemmaPathLen<T>(G: Graph<T>, p : seq<T>) requires validGraph (G) && validPath (p , G) && |p| > |G.V| ensures ! acyclic (G) { // 首先注意，如果所有顶点互不相同，则路径长度不会超过 G 中的顶点数 if nodups (p) { lemmaSeqLen (p , G.V) ; } // 因此，p 中必定存在重复顶点，于是我们选取一条无环（复杂）子路径 var i , j : | 0 <=i <j < |p |&& p [i] ==p [j];

```
var p' := p[i..j+1]; // 回忆一下辅助引理，它保证还存在一个简单环
lemmaComplexPath(G, p'); } // 通过归纳陈述并证明如下性质：给定图 G 中任意一条有效复杂路径 p（边和/或顶点可能重复），在 G 中存在一条从该复杂路径的第一个顶点到最后一个顶点的简单路径（没有重复边）。lemma lemmaComplexPath<T>(G: Graph<T>, p: seq<T>) requires G.V != {} && validPath(p, G) && |p| > 1 ensures exists SimplePath(G, p[0], p[|p|-1]) decreases p { // 处理第一个顶点在中间重复出现的情况 if i: | 1 <= i < |p|-1 && p[i] == p[0] { lemmaComplexPath(G, p[i..]); } // 处理归纳证明的递归情况 else if |p| > 2 { lemmaComplexPath(Graph(G.V, G.E - {(p[0], p[1])}), p[1..]); } } function elems<T>(s: seq<T>): set<T> { set x | x in s } predicate nodups<T>(s: seq<T>) { forall i, j :: 0 <= i < j < |s| ==> s[i] != s[j] } // 通过归纳陈述并证明以下性质：集合 s 中互异元素构成的序列，其长度不会超过集合的基数。lemma lemmaSeqLen<T>(p: seq<T>, s: set<T>) requires nodups(p) && elems(p) <= s ensures |p| <= |s| { if p != [] { lemmaSeqLen(p[1..], s - {p[0]}); } } /** 第一个引理 ***/ // 通过反证陈述并证明以下
```

// 性质：无环图 G' 的子图也是无环的。lemma lemmaAcyclicSubgraph<T>(G: Graph<T>, G' : Graph<T>) 要求 validGraph (G) && validGraph (G') && 无环 (G') && 是子图 (G, G') 保证 无环 (G) { 如果 ! 无环 (G) { var u : |u 在 G.V 中 && 存在量词 简单路径 (G, u , u) ; // 根据无环的定义，存在量词 lemmaExistsSimplePath (G, G' , u , u) ; // 调用引理，说明这样的路径也存在 // 于 G 中 assert ! 无环 (G') ; // 因此 G' 将不是无环的，与前置条件矛盾。 } } // 检查图 G 是否是另一个图 G' 的子图。谓词 是子图<T>(G: Graph<T>, G' : Graph<T>) { G. E <= G' . E && G.V <= G' . V } // 通过归纳陈述并证明以下性质：如果图 G 中 // 存在一条从 u—>v 的（简单）路径，且 G 是 G' 的子图， // 那么 G' 中也存在一条从 u—>v 的路径。lemma lemmaExistsSimplePath<T>(G: Graph<T>, G' : Graph<T>, u : T, v : T) 要求 validGraph (G) && validGraph (G') && 是子图 (G, G') && 存在量词 简单路径 (G, u , v) 保证 存在量词 简单路径 (G' , u , v) 减少 G. E { 如果 (u, v) ! 在 G. E 中 { // 递归情形 var e : |e 在 G. E 中 && e . 0 == u && 存在量词 简单路径 (Graph (G. V, G. E–{e }) , e . 1 , v) ; // 根据存在量词 Path 的定义，这一定存在 lemmaExistsSimplePath (Graph (G. V, G. E–{e }) , Graph (G' . V, G' . E–{e }) , e . 1 , v) ; // 这个引理说明 'e' 在 G' 中也存在 } }

/** 测试 cases ***/ 方法 testTopSortingSingleSolution() { var G: Graph<自然数>:=Graph ({ 1 , 2 , 3} , { (1 , 2) , (2 , 3) }) ; assert 有效图 (G) && acyclic (G) ; var s : 序列<自然数>:= [1, 2, 3]; assertisTopSorting (s , G) ; var t:=topsort(G) ;

assertt ==s;} 方法testTopSortingMultipleSoluti ons(){ var G: Graph<自然数>:=Graph({1,2,3},{(1,2),(1,3)}); assert 有效图(G) && acyclic(G); var s1 : 序列<自然数>: = [1, 2, 3]; var s2 : 序列<自然数>:= [1, 3, 2]; assertisTop Sorting(s1,G); assertisTopSorting(s2,G); var t:=topso rt(G); assertt ==s1 ||t ==s2;}

A.10 欧拉回路

/*
*证明 Hierholzer 算法（1873 年）的正确性，用于
*在欧拉图中查找欧拉回路（方法 *findEulerCircuit）。*参考:
https://en.wikipedia.org/wiki/Eulerian_路径。

*//****图的表示与有效性 ****// 顶点可以是任何支持相等性的类型。type Vertex =自然数 // 或其他类型 // 图表示为从顶点到其相邻顶点集合的映射。type Graph =m: map<Vertex, set<Vertex>> |定义 ValidGraph(m) // 该映射必须是反自反且对称的。predicate 定义 ValidGraph(m: map<Vertex, set<Vertex>>) { forall v, w :: v in m && w in m[v]==> w!= v && w in m && v in m[w]} /****图修改操作 ****// 从图 G 中移除顶点 v（如果存在）。function rmvVertex(v: Vertex, G: Graph): Graph { map u | u in G && u!=v :: G[u] −{v} } // 从图 G 中移除边 (u, v)（如果存在）。function method rmvEdge(u: Vertex, v: Vertex, G: Graph): Graph

ensures var G' : Graph :=rmvEdge(u, v, G); u in G && v in G && v in G[u] ==>hasEvenCard(G'[v])!=hasEvenCard(G[v]) && hasEvenCard(G'[u])!= hasEvenCard(G[u]) { map x | x in G : : if x == u then G[x] -{v} else if x == v then G[x] -{u} else G[x]} // 向图 G 添加边 (u, v)。function addEdge(u: Vertex, v: Vertex, G: Graph): Graph requires u in G && v in G && u!=v { map x | x in G : : if x == u then G[x] +{v} else if x == v then G[x] +{u} else G[x]} /****子图 ****/ // 检查 G1 是否是 G2 的边子图，但顶点集合相同。predicate isSubgraphE(G1: Graph, G2: Graph) { G1.Keys == G2.Keys && forall x :: x in G1 ==> G1[x] <=G2[x]} /****连通性 ****/ // 检查给定的图是否连通，也就是说，任意两个顶点之间都存在一条路径。predicate isConnected(G: Graph) { forall u, v :: u in G && v in G==> connectedVertices(u, v, G) } // 检查图 G 中的顶点 u 和 v 是否连通，也就是说，存在一条连接它们的路径（不重复经过顶点）。predicate connectedVertices(u: Vertex, v: Vertex, G: Graph) requires u in G && v in G decreases G { u == v| | exists w :: w in G[u]&& connectedVertices(w, v, rmvVertex(u, G)) } // 通过归纳法证明：如果顶点 u 属于图 G 中一个在邻接关系下闭合的顶点集合 C，而 v 不属于该集合，那么它们必定不连通。lemma unconnectedVerticesLemma(u: Vertex, v: Vertex, G: Graph,

C: 集合<顶点>) 要求 u∈G && v∈G && u∈C && v∉C 要求对于所有 x: : x∈C && x∈G ==> G[x] <= C // C 是一个封闭顶点—集合 减少 G 确保 !连通顶点(u, v, G) { // 模仿连通顶点的结构 对于所有 w |w∈G[u]{ unconnectedVerticesLemma (w, v, rmvVertex (u, G), C) ; } }

/**** 顶点度数 ****/ // 检查图 G 中所有顶点的度数是否为偶数（偶数 // 条关联边）。谓词 hasEvenDegrees (G: Graph) { f o r a l l v : : v i n G ==> hasEvenCard (G[v]) } // 检查集合 s 中的元素个数是否为偶数（偶数 // 基数）。谓词 hasEvenCard<T>(s: s e t <T>) {| s | % 2 == 0} // 如果从顶点度数均为偶数的图 G 中移除子图 T 的边，且 T 的顶点度数也均为偶数，// 则得到一个顶点度数均为偶数的子图 R。引理 偶数度引理 (G: Graph, R: Graph, T: Graph) r e q u i r e s G. 键 == T. 键 == R. 键 r e q u i r e s f o r a l l x : : x i n G==> T[x] <= G[x]&& R[x] == G[x] − T[x]r e q u i r e s hasEvenDegrees (G) && hasEvenDegrees (T) e n s u r e s hasEvenDegrees (R) { // 感谢 Dafny } /*** 游走、迹、回路与增广性质 ****/ // 检查顶点序列是否定义了图 G 中的有效游走。// 谓词 isValidWalk (s: 序列<Vertex >, G: Graph) { (f o r a l l x : : x i n s ==> x i n G) && (f o r a l l i : : 0 <= i < |s | − 1 ==> s [i +1]i n G[s [i]]) } // 有效游走的实用增广性质。引理 walkAugmentationLemma (s: 序列 <Vertex >, G: Graph,

```
    u: Vertex) r e q u i r e s isValidWalk (s, G) && u i n
G && (|s | == 0 ||u in G[s [|s |-1]]) e n s u r e s
isValidWalk (s + [u ], G) { /* 感谢 Dafny */ }


// 检查顶点序列是否经过边// (u, v)。predicate traversesEdge(s: seq<Vertex>, u:
Vertex, v: Vertex)// 要求 u !=v {exists i :: 1 <=i < |s |&& {s [i -1], s [i ]}
 =={u, v}// 遍历边的有用增广性质。lemma traversesEdgeProp(s: seq<Vertex>,
v: Vertex)requires |s | > 0&& v !=s [|s |-1]ensures traversesEdge(s + [
v] ,s [|s|-1], v); { // Dafny 似乎只需要证明“拼接”断言var s':=s + [v ]; s' [..|
s |] ==s && s' [|s |] ==v; }// 检查顶点序列是否在图 G 中定义了一个有效迹// 即
没有重复边的有效游走。predicate isValidTrail(s: seq<Vertex>, G: Graph) {
isValidWalk(s, G) && forall i :: 1 <= < | |i s! traversesEdge(s [..
i ] ,s [i -1], s [i ]) }// 有效迹的有用增广性质。lemma
trailAugmentationLemma (s: seq<Vertex>, G: Graph, u: Vertex)requires
isValidTrail(s, G) && u in Grequires |s | > 0 ==>u in G[s [|s |-1]]// 使其
成为有效游走requires |s | > 0 ==>! traversesEdge(s, s [|s |-1], u) // 使其
成为有效迹ensures isValidTrail(s + [u ], G) { /* 多亏了 Dafny */ }// 检查顶点
序列是否在图 G 中定义了一个有效回路// 即非空迹，其中第一个和最后一个顶点相同。
predicate isValidCircuit(s: seq<Vertex>, G: Graph) { isValidTrail(s, G) && |s
| > 0&& s [|s |-1] ==s [0]}
```

// 说明回路增广（通过嵌入）蕴含所经过边的并集。lemma circuitAugmentationLemma1 (s1: seq<Vertex>, i: int, s2: seq<Vertex>, s3: seq<Vertex>, G: Graph)requires isValidCircuit(s1, G) && isValidCircuit(s2, G)requires 0 <= i < |s1| && s2[0] == s1[i] && s3 == s1[..i] + s2 + s1[i+1..]ensures forall x, y :: x in G && y in G[x] ==>(traversesEdge(s3, x, y) <==> traversesEdge(s1, x, y)| |traversesEdge(s2, x, y)) { // Dafny 似乎只需要证明“拼接”（切片）断言assert s1[..i] == s3[..i]; assert s2 == s3[i .. i + |s2|]; assert s1[i + 1..] == s3[i + |s2|..]; }

// 通过演绎证明，通过嵌入另一条边不相交的回路来扩充回路，结果仍是有效的

// 回路，且不包含重复的边 .
引理 回路增广引理2（s1：seq<顶点>，i：int，s2：seq<顶点>，s3：seq<顶点>，G：图）要求 isValidCircuit(s1, G) && isValidCircuit(s2, G) 要求 0 <= i < |s1| && s2[0] == s1[i] && s3 == s1[..i] + s2 + s1[i+1..]要求对于所有 k：： 1 <= k < |s1|==>! traversesEdge(s2, s1[k-1], s1[k]) 要求对于所有 k：： 1 <= k < |s2|==>! traversesEdge(s1, s2[k-1], s2[k]) 确保 isValidCircuit(s3, G) { // 模仿检查是否存在重复 // 边的过程，对于所有 j, k | 1 <= j < k < |s3|确保 {s3[k-1], s3[k]} !={s3[j-1], s3[j]} { // 映射到原始序列中的索引 var (j’，sj)：= 如果 j <= i 则 (j, s1) 否则如果 j < i + |s2| 则 (j–i, s2) 否则 (j –(|s2|-1), s1)； var (k’，sk)：= 如果 k <= i 则 (k, s1) 否则如果 k < i + |s2| 则 (k–i, s2) 否则 (k –(|s2|-1), s1)； // 回顾原始序列中的边彼此不同 //（来自前置条件）assert {sk[k’-1], sk[k’]} !={sj[j’-1], sj[j’]}； }

} /*** 欧拉迹与回路 ****/ // 检查顶点序列是否定义了图 G 中的欧拉回路，// 即一条恰好遍历 G 每条边一次的回路。predicate isEulerCircuit(s：seq<顶点 >，G：图) { isValidCircuit(s,G) // 确保不存在重复穿越的边 && 对所有 u, v：：u 属于 G && v 属于 G[u]⟹ traversesEdge(s, u, v) } // 通过反证法证明，在连通图 G 中，不可增广回路 r // 必须覆盖所有边，即必须是欧拉回路。引理 不可增广回路引理(G：图，r：seq<顶点 >) 要求 isConnected(G) && isValidCircuit(r, G) 要求对于所有 x, y：：x 属于 r && y 属于 G[x]⟹ traversesEdge(r, x, y) 确保 isEulerCircuit(r, G) { assert 对所有 x, y：：x 属于 r && y 属于 G[x] ⟹ y 属于 r；// 由第二个前置条件蕴含 if v：|v 属于 G && v ! 属于 r { unconnectedVerticesLemma(r[0] ,v, G, set x |x 属于 r)；// 这与 G 连通的假设矛盾，// 因此 v 不存在；所以 G 的所有顶点都被 r 覆盖，// 因而其关联边也被覆盖（由第二个前置条件）。} } // 检查顶点序列是否定义了图 G 中的欧拉迹，// 即一条恰好遍历 G 每条边一次的迹。predicate isEulerTrail(s：seq<顶点 >，G：图) {|s | > 0 && isValidTrail(s, G) && 对所有 x, y：：x 属于 G && y 属于 G[x]⟹ traversesEdge(s, x, y) } // 欧拉迹中顶点度数的性质：每个顶点上的 // 关联边数量都是偶数，唯有起点和终点不同的情况除外。predicate EulerTrailDegrees(G：图，r：seq<顶点 >) 要求 isEulerTrail(r, G) { var first, last：=r[0] ,r[|r |−1]；对于所有 x：：x 属于 G ⟹((x==first) !=(x==last) /*异或 */ <⟹ ! hasEvenCard(G[x]))

} // 通过归纳法证明欧拉迹中关于顶点度数的上述性质。lemma EulerTrailLemma (G: Graph , r : seq<Vertex >) r e q u i r e s i s E u l e r T r a i l (r , G) e n s u r e s E u l e r T r a i l D e g r e e s (G, r) { i f |r | > 1{ EulerTrailLemma (rmvEdge (r [0] , r [1] , G) , r [1..]) ; } }

/**** 主要算法 ****/

// Hierholzer 算法：基于深度优先搜索，在非空欧拉图 G 中查找欧拉回路。method findEulerCircuit(G: Graph) returns (r: seq<Vertex >) requires isConnected(G) && hasEvenDegrees(G) && |G| > 0 ensures isE ulerCircuit(r, G) { // 构建初始回路，从任意顶点开始，// 并获取剩余图 var v: | v i n G; var R: Graph; r, R := dfs(v, G); // 用于辅助证明终止性的幽灵变量（待探索的顶点）ghost var V := set x |x in R && R[x]!={}; // 尽可能增广 r while exists i :: 0 <=i < |r |&& R[r [i]]!={} // r 是 G 中以 v 为起点的有效回路 invariant isValidCircuit(r, G) && |r | > 0&& r [0] == v // R 是 G 的子图，且顶点度数均为偶数 invariant hasEvenDegrees(R) && isSubgraphE(R, G) // R 包含 G 中未被 r 遍历的边 invariant forall x, y :: x in G && y in G[x]==>(y ! in R[x] <==> traversesEdge(r, x, y)) // V（变元）是这样一个顶点集合：它们有尚未探索的相邻顶点 invariant V == set x |x in R && R[x]!={} decreases V { // 选择 r 中一个有出边可供探索的顶点

var i : | 0 <= i < |r |&& R[r [i]] != {}; var u := r [i]; // 记住稍后需要的旧值 ghost var oldr, oldV := r, V; // 从该顶点开始，在剩余图中执行深度优先搜索，// 获取新的子回路和剩余图 var c: seq<Vertex >; c, R := dfs(u, R); // 将子回路插入主回路 r := r [..i] + c + r [i + 1..]; // 调用回路增广性质以确保// 不变式得以维持 circuitAugmentationLemma1(oldr, i, c, r, G); // 已遍历边的并集 circuitAugmentationLemma2(oldr, i, c, r, G); // 不存在重复边 // 证明变元 V 减少: V := set x |x in R && R[x]!= {}; assert u in oldV && u ! in V; assert V < oldV; } // 由于 G 连通，证明 G 的所有边均已遍历 nonAugmentableCircuitLemma(G, r); } // 通过执行深度优先搜索，在图 G 中生成一条从顶点 v 开始的完整有效路径。// 假设图中所有顶点的度数均为偶数，则生成的路径是闭合的。// 返回闭合路径 r 和剩余图 R（其中包含未探索的边）。method dfs(v: Vertex, G: Graph) returns (r: seq<Vertex >, R: Graph) requires hasEvenDegrees(G) && v in G ensures isValidCircuit(r, G) && |r | > 0&& r [0] == v ensures isSubgraphE(R, G) && hasEvenDegrees(R) ensures forall x, y :: x in G && y in G[x]==> (y ! in R[x] <==> traversesEdge(r, x, y)) ensures R[v] == {} // v 的所有后继顶点均已探索

d

{ R := G; // 尚待访问边的子图 宿主图 var T: Graph := map x |x i n G : : { } ; // 已遍历边的子图

```
// 用于辅助证明终止性的幽灵变量（剩余边）

g ho st var E :=s e t x , y  |x i n G && y i n G[x ]: : ( x , y ) ;

// 用初始顶点开始构造结果（此时也是最后一个顶点）r:=  [v ] ; var u:=  v; // 尽可能增广 rwhile R[u ]!={} // R 是 G 的子图（顶点集相同） invariant isSubgraphE (R, G) // T 是 G 的子图，其边是 G −Rinvariant T.Keys  == G.Keys && forall x :: x in G ⟹ T[x ]  == G[x ]  − R[x ]// E 是 R 中边的集合invariant forall x, y :: x in R && y in R[x ]⟺ (x, y) in E // r 是从 v 开始、以 u 结束的顶点序列invariant  |r | > 0 && r [0]  == v && r [|r |−1]  == u // r 恰好遍历 T 中的边（不重复）invariant isValidTrail(r, T)invariant forall x, y :: x in G && y in G[x ]⟹ (y in T[x ]  <⟹ traversesEdge(r, x, y)) // 用于确保终止的变式：E 递减{ // 选择一条尚未遍历的边，并沿边选取一个相邻顶点var w: |w in R[u ] ; // 回顾迹增广的一些性质 trailAugmentationLemma(r, G, w); // 有效的迹遍历 EdgeProp(r, w); // 已遍历的边// 增广迹，并更新已访问和未访问的边以及最后一个顶点r:=r  +  [w] ; R:=rmvEdge(u, w, R); T:=addEdge(u, w, T); E:=E  −{(u, w), (w, u)}; u:=w;
```

```
} // 证明得到的迹（T 中的欧拉迹）结束于起始顶点assert T[u] == G[u]; // 因为 R[u] == {}assert hasEvenCard(T[u]); // 因为 hasEvenCard(G[u]) EulerTrailLemma(T, r); assert u == v; // 证明在剩余图 (R) 中所有顶点的度都是偶数 assert hasEvenDegrees(T); evenDegreesLemma(G, R, T); assert hasEvenDegrees(R);} method testEulerCircuit() { var G: Graph: = map[1:={2, 3}, 2:={1, 3}, 3:={1, 2, 4, 5}, 4:={3, 5}, 5:={3, 4}]; var c: seq<Vertex>: = [1, 2, 3, 4, 5, 3, 1]; assert c == [c[0],c[1],c[2],c[3],c[4], c[5],c[6]]; // 辅助程序……assert isEulerCircuit(c, G); } method testEulerTrail() { var G: Graph: = map[1:={2, 3}, 2:={1, 3}, 3:={1, 2, 4}, 4:={3, 5}, 5:={4}]; var c: seq<Vertex>:= [3, 2, 1, 3, 4, 5]; assert c == [c[0],c[1],c[2],c[3],c[4],c[5]]; // 辅助程序……assert isEulerTrail(c, G); }
```