通过指令子集缩小归纳式编程搜索空间

Edward Mc Daid[1] [a] 和 Sarah Mc Daid[1] [b]
Zoea Ltd.，英国伦敦 *Wenlock Road 20-22*号，*N1 7GU {edward.mcdaid, sarah.mcdaid}@zoea.co.uk*

关键词：归纳式编程、状态空间搜索、知识表示。

摘要：归纳式编程常常依赖某种形式的搜索来识别候选解。然而，搜索空间的规模限制了归纳式编程的应用，使其只能用于生成相对较小的程序。如果我们能够以某种方式正确预测给定问题所需的指令子集，那么归纳式编程就会更易于实现。我们将证明，在很大比例的情况下，这一目标是可以实现的。本文提出了一种新颖的编程语言指令共现模型，旨在支持 Zoea 分布式归纳式编程系统中的搜索空间划分。该模型由一组相互交叠的指令子集构成，这些子集源自大量开源代码。采用这种方法，可以并行探索搜索空间的不同部分。生成这些子集所用代码的数量增加时，所需子集数量并不会线性增长；只需数量适中的子集，就足以覆盖很大比例的未见代码。这种方法还能显著缩小整体搜索空间，通常可缩小多个数量级。

[a] https://orcid.org/0000-0001-8684-0868
[b] https://orcid.org/0000-0001-7643-6722

1 引言

利用人工智能辅助甚至自动生成计算机软件，是一个活跃的研究领域（例如，Xu 等人，2022，Nguyen 和 Nadi，2022）。许多现有系统基于深度学习，近期研究还包括使用 GPT-3（Brown 等人，2020）等大语言模型。这些方法需要使用大量代码进行训练，但也可能引发对透明性和伦理问题的担忧（Lemley 和 Casey，2021）。

其他一些通常不与训练联系在一起的方法也在持续发展（Cropper、Dumancic 和 Muggleton，2020年；Petke 等，2018年）。归纳式编程旨在直接根据某种形式的规格说明生成代码——通常采用输入输出示例或测试用例的形式（Flener 和 Schmid，2008年）。归纳式编程已经发展出多种技术，但从根本上说，其中许多技术都依赖某种形式的搜索（Kitzelmann，2010年）。

除非是简单的情况，否则无法仅根据源代码直接确定计算结果，而不执行程序。因此，某种形式的生成与测试方法是不可避免的。

在计算机程序中，大量语言元素可以通过多种方式组合，从而迅速产生数量庞大的候选解。搜索空间的规模限制了归纳式编程系统，使其只能生成相对较小的程序（Galwani 等，2015年）。

归纳式编程中组合增长的一个主要来源，是指令的数量，其中包括核心语言、标准库函数以及运算符。这个数量因编程语言而异，但通常约为 200 个，甚至更多。有人提出，如果我们能够以某种方式正确预测给定程序实际所需的指令子集，那么归纳式编程问题将更容易解决，并且在给定资源条件下可以生成更大的程序（McDaid，2019年）。

在给定的时间内生成稍大一些的程序，一种方法是将工作分配给多台计算机。为此，我们需要划分搜索空间。按指令集划分颇具吸引力，因为大多数程序使用的指令子集相对较小。那么，我们该如何定义要使用的子集呢?

本文介绍了一项研究的结果，该研究旨在为 Zoea 归纳式编程系统（McDaid 2021）中指令子集的定义提供依据。开头部分介绍了我们生成和评估指令子集的方法。随后，我们进一步讨论了一些只有在这项工作完成后才显现出来的重要发现。

2 预备知识

设C是高级命令式编程语言L中的一段源代码程序。C由一个或多个程序单元U构成，分别对应过程、函数或方法。L提供一组指令IL，包括内置运算符、核心函数和标准库函数。每个U都包含一组指令IU，其中$IU \subseteq IL \wedge IU \neq \emptyset$。

给定一组程序$SP1$，我们可以枚举相应的程序单元指令子集族SIU，其中$SIU \in \mathrm{P}(IL) \wedge SIU \neq \emptyset$。（此处P表示幂集。）

若IU是U的指令子集，或者IU是另一个U的指令子集，且是U对应IU的超集，则称IU覆盖U。每个IU显然都会覆盖相应的U。我们也可以说SIU覆盖$SP1$。一组程序的覆盖率按被覆盖的程序单元数除以程序单元总数计算，并以百分比表示。

任何作为另一个指令单元子集的指令单元，都可以从SIU中移除，而不会影响SIU相对于$SP1$的总体覆盖率。两个或更多指令单元可以组合形成新的派生指令子集标识符。标识符相对于$SP1$提供的总体覆盖率，与其组成指令单元提供的覆盖率相同。

SID是一个指令子集族，由所有指令单元（尚未被移除或合并）与所有标识符的并集构成。SID相对于$SP1$提供的覆盖率，与原始SIU相同。在创建SID期间，我们可以对|指令单元| 设置上限$M1$，并对|标识符| 设置上限$M2$。任何满足|指令单元| > M1的U都会被静默忽略。$M2$限制了SIU（指令单元）中的哪些子集可以组合形成标识符。

可以创建任意数量的标识符，前提是也移除各自对应的组件指令单元和 |标识符| $<= M2$。创建后，便可根据SID针对另一组程序$SP2$所提供的覆盖率对其进行评估。

3 方法

3.1 目标

这项工作的主要目标是定义一组指令子集，以支持 Zoea 归纳式编程系统（McDaid 2021）中的高效聚类。Zoea 采用分布式黑板架构，其中包含许多并行运行的知识源。知识源的激活也可以根据指令子集进行划分。

目前，Zoea 可以高效利用约 100 个核心，并在解决问题时创建数百万个激活。要高效利用更多核心，需要对指令集进行更细粒度的划分。这将使 Zoea 能够更好地利用基于云的部署。

事先并不清楚，要为各种各样的程序提供足够的覆盖率需要多少个子集。根据对当前系统进行调优的经验，数百到数千个子集可能就能带来可接受的性能。

这项工作源自一项更早的随机子集定义工作，并最终取代了该工作；此处不再详细介绍那项工作。人们认为，从现有代码中推导子集，可能有助于使子集更能代表人类编写的软件。此外，显然这些子集之间需要在一定程度上相互重叠。

指令子集的目标大小是一项重要考量。较小的子集有助于在更短时间内于搜索空间中找到指令数量较少的程序。然而，我们也需要能够生成包含更多指令的程序。这意味着可以采用多组不同大小的指令子集。在这种方案中，子集大小限定的是唯一指令数量的上限，而非能够找到的程序大小。实际操作中，可以先尝试寻找唯一指令较少的程序，也可以同时进行搜索。评估的目标子集大小为 10 到 100，增量为 10。

子集创建过程的输入是一组大小各异、非空的程序单元指令子集。这些子集按照其在源代码中出现的顺序排列。不提供任何其他相关元数据。输入子集中往往有许多重复项。

根据所使用的代码量，程序子集的数量可能很多，而我们希望派生子集的数量要少得多。从广义上说，这可以视为一种聚类。为了对程序子集进行聚类，派生子集的大小允许通过可配置的幅度略大一些。在生成指定大小的派生指令子集时，我们只需忽略较大的程序子集。

对于给定的派生子集大小，每个大小不超过该值的程序单元子集，本身都将是至少一个派生子集的子集。这意味着，每个最大的程序单元子集实际上都会构成一个派生子集的核心。抽屉原理表明，仅仅为了容纳较大的程序子集，就至少需要一定数量的派生子集。这也说明，我们应当先处理较大的程序子集。如果按大小降序处理输入子集，那么在某个阶段，已部分填充的派生子集中的剩余空间将足以容纳较小的程序子集。

可以通过统计由一组派生指令子集所覆盖的、唯一指令数量相同或更少的未见程序，来衡量这组派生指令子集的性能。这里所说的“未见”，仅指未用于生成这些子集的程序。如果某个程序的所有指令都出现在至少一个子集中，那么该程序就被这组子集覆盖。这意味着，对这些子集所定义的空间进行任何穷举搜索，最终都能找到一个被覆盖的程序。

总而言之，主要目标是：为不同的最大子集大小创建指令子集集合；了解不同最大子集大小分别需要多少个子集；并确定这些子集对未见代码的覆盖率。

3.2 方法

指令子集需要能够代表真实世界软件，这意味着它们也应从真实世界软件中提取。为确保有足够的多样性，需要大量代码；理想情况下，这些代码应由来自不同背景的众多开发者编写。此外，这些代码还必须能够合法且合乎伦理地获取。

GitHub（微软，2022）被确定为合适的软件来源，过去也曾用于类似的代码分析（Ray 等人，2014）。我们使用了 GitHub 上最大的 1,000 个代码仓库（截至 2022年5月13日），并将分析范围限定为 Python 程序（ Martelli 、 Ravenscroft 和 Holden，2017）。选择 Python 的依据是

它是一种相当流行的语言，而且其现有指令能够代表类似语言中的指令。

在每种情况下，完整的代码仓库都以 zip 文件的形式下载、解压，并删除其中非 Python 文件。每个 Python 程序都被拆分为程序单元（类、方法、函数和主程序），并使用基于正则表达式的简单自定义代码进行分词。由于解析错误，其中两个代码仓库被排除。根据识别出的词元，统计特定指令集合中每条指令的出现次数。对于 Zoea 中没有意义的指令，例如与变量赋值有关的指令，如果可能，则映射到等效指令；否则予以排除。（例如，'+=' 映射为 '+'，而 '=' 则被排除。）没有以任何其他方式使用代码或其他信息。此步骤的输出是每个程序单元对应的一份按字母顺序排列的唯一指令名称列表。

分析共涵盖 71,972 个 Python 文件，包含 15,749,416 行代码，即 580,476,516 个字符。从中识别出 886,421 个程序单元。系统记录了每个程序单元所包含的指令子集。由此识别出的许多指令子集存在重复；去重后，剩下 345,120 个唯一指令子集。

在处理过程中，对指令子集进行筛选，删除任何属于另一个指令子集真子集的子集。也就是说，如果子集 A 中的所有指令也都出现在子集 B 中，就删除子集 A。重复此过程，直到不再有子集被删除。最终剩下 33,823 个指令子集。它们的大小从 1 条到 74 条指令不等（中位数：3，标准差：3.67）。许多独特的指令子集彼此非常相似，仅相差一两条指令。

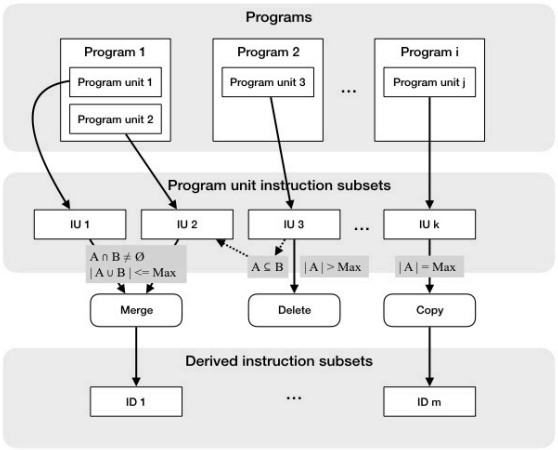

图1：子集创建概览。

3.3 聚类算法

图1从概念上概述了子集的创建过程。从程序单元指令子集（IU）生成派生指令子集（标识符），本质上是一个聚类问题。我们开发并评估了多种不同的聚类算法。最终方案结合了其中表现最好的两种算法，以及一些前处理和后处理。该软件的伪代码见算法 1。

创建一组派生指令子集时，始终会指定一个最大子集大小。此大小限制也决定了派生子集的目标数量，详情将在本节后文进一步说明。

删除所有重复的指令单元；扩增指令单元（见第 3.4 节）；删除作为其他指令单元子集的指令单元；对于每个指令单元，查找 Instr、Subsets，其中 Instr 不是 IU 的子集，且 length(IU ∪ Instr) 最大；令 IU = IU ∪ { Instr }；删除 Subsets 中的所有指令单元；结束循环；创建 NumIDs 个空标识符；对于每个指令单元，如果存在（空标识符），则令 ID = IU；否则，查找使 max(| ID ∩ IU |) 最大的标识符；选择使 min(| ID |) 最小的标识符；令 ID = ID ∪ IU；结束条件判断；结束循环；合并 size <= MaxIdSize 的标识符；返回标识符集合

算法 1：聚类算法。

输入子集按大小递减的顺序处理。为提高性能，所有指令子集始终在内部保持排序。

预处理包括去重、放大和子集移除。下一节将介绍放大。对于重复的多个输入子集，只保留其中一个。任何完全包含于另一个输入子集中的输入子集也会被移除。

第一个聚类阶段尝试通过依次向每个指令单元添加一条额外指令，来涵盖尽可能多的近似子集。选择要添加的指令时，算法会确定添加该指令后，还有多少其他指令单元将成为当前指令单元的子集

。随后选取能使其他指令单元数量减少最多的那条指令。

第二个聚类阶段会尝试将每个指令单元与交集最大的标识符合并。具体做法是预先创建指定数量的空标识符，然后将指令单元填入空标识符，或者将其并入交集最大且剩余容量最多的标识符。如果所有标识符都已达到最大容量，则会创建额外的空标识符。

后处理会将剩余的小型指令单元与未满的标识符合并。此过程完全依据子集大小进行，不考虑相似度。

如前所述，该算法允许指定最大子集大小和目标子集数量。然而，如果无法满足所请求的子集数量，则不一定会照此执行；在这种情况下，会额外创建最少数量的子集。对于每种子集大小，所需子集的大致数量会由一段代码预先确定，该代码按升序遍历所有可能的取值。当创建的子集数量等于所请求的数量时，即可确定所需的子集数量。由于这一过程耗时较长，因此仅按增量进行，所得数值是近似值，误差范围在增量大小以内。

3.4 放大

最初，合并子集的过程仅被视为一种减少派生子集数量并帮助统一其大小的方法。然而，观察发现，合并后对未见代码的覆盖率显著提高。部分原因是，与较小子集相比，较大的子集能提供更好的覆盖率，因为它们包含更多子集。此外，将来自无关代码的不同指令子集组合起来，与软件组合有相似之处。合并输入子集会增加子集内部指令共现的多样性，而这种多样性原本并不存在。值得注意的是，添加数量相当的随机指令，而不是进行合并，并未带来任何可检测到的收益。

为了更充分地利用这一现象，我们引入了放大步骤：在聚类之前，将较小的输入子集合并，以创建更多人工程序子集。我们并未对此进行系统研究，但其收益似乎不会随着子集数量增加超过 50% 而继续提升。

这一领域的进一步研究或许会有所收获。

4 结果

4.1 输入数据

图 2 显示了程序单元指令子集的大小频率分布，其中既包含不同大小子集的数量，也包含每种子集大小对应的程序单元数量的累计百分比。从中可以看出，约 90% 的程序单元所含的指令子集至多包含 10 条唯一指令。此外，只有约 2% 的程序单元子集包含 20 条或更多指令。

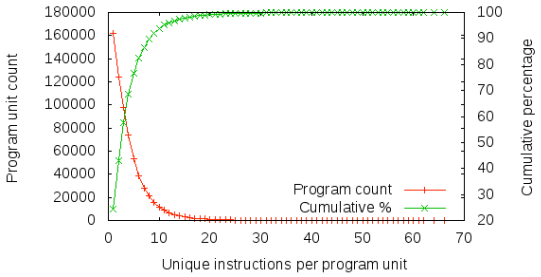

图2：程序单元指令子集大小频率分布。

图3展示了所用全部代码中各条指令的频率分布。这里的指令按频率从高到低排列。这表明，少数指令使用频率很高，而许多指令很少使用。这类似于常与人类语言和人工语言相关联的齐夫分布（Louridas、Spinellis 和 Vlachos，2008年）。

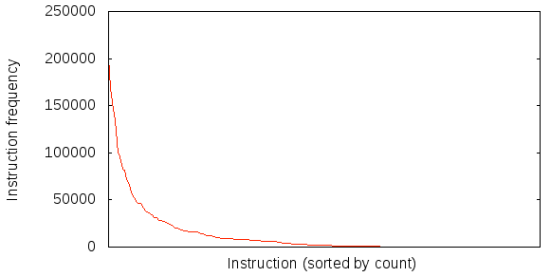

图3：指令频率分布。

共现指令对的排序频率分布与指令的排序频率分布相似，但这种特征更为明显。只有少得多的

指令对会频繁共现，而绝大多数共现频率很低，或者根本不会共现。同样，尾部又长又细。

4.2 未见代码覆盖率

根据定义，派生子集始终能够对用于创建它们的代码实现 100% 的覆盖率。换句话说，每个程序单元指令子集中的所有指令，至少会在一个派生指令子集中同时出现。然而，创建派生子集的目的并非如此。

要发挥作用，派生子集还应能为未用于创建它们的代码提供较高的覆盖率。未见代码的覆盖率是通过指定输入代码的一定比例作为训练集，再据此生成派生子集来确定的。随后，可以使用以这种方式生成的派生子集，对输入代码库中的另一部分代码进行测试，也可以对整个代码库进行测试。

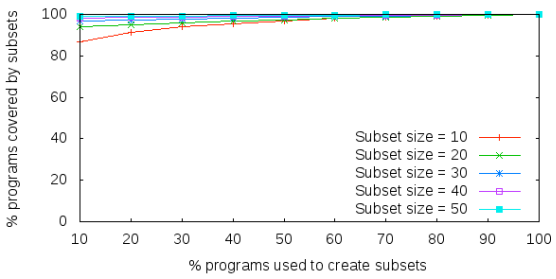

图 4：不同训练集比例和子集大小下的未见代码覆盖率。

为了解使用不同大小训练集所达到的相对覆盖率，我们以 10% 为增量，使用可用代码的 10% 至 100% 生成子集。每种情况下，随后都使用整个代码库对派生子集进行评估。结果如图 4 所示。这些测试执行了 50 次，很早便可以看出，使用代码库中相对较小比例的代码生成的子集也能实现较高的覆盖率。对于子集大小为 10 的情况，仅使用 1% 的代码生成的子集就能达到 77.31% 的覆盖率。

我们还进行了其他测试（此处未报告），以确保提取训练集的那段特定代码不会对结果产生不利影响。

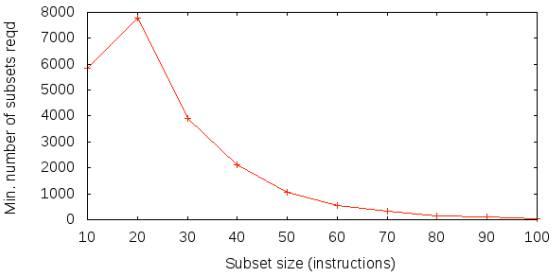

图 5：按子集大小划分所需的子集数量。

4.3 子集数量

正如我们之前指出的，所需的派生子集数量在很大程度上取决于派生子集的最大大小。图 5 展示了不同最大大小对应的派生子集数量。总体而言，随着最大子集大小的增加，子集数量呈指数级减少。最大大小为 10 时数量少于 20，这可能是由于派生子集预留空间的大小固定，再加上子集大小分布偏斜所致。

派生子集的数量处于我们认为可接受的范围内，足以支持 Zoea 聚类。通过各种方式还可以进一步减少子集数量，但这也会降低未见代码的部分覆盖率。例如，仅从训练集覆盖率的角度考虑时，其中一些子集是冗余的。此外，许多单独的指令也可以基于同样的原则从子集中移除。

另一个重要因素是，随着训练集规模扩大，所需子集的数量如何增长。这种增长并非线性增长，而是随着每次增量而逐渐放缓。增长率的下降表明，子集大小最终会趋于稳定，而不是无限增长。

4.4 搜索空间缩减

使用指令子集的最初动机，是将工作分配到集群中的多个工作节点上。一个起初未曾预料到的额外好处是，搜索空间的规模也得到了显著缩减。其原因很容易理解。

代码的搜索空间可近似视为一棵具有给定深度的树，其分支因子主要由指令数量决定。已有多种方法用于估算

此类搜索树的大小（Kilby 等人，2006年）。然而，归纳式编程的实际情况要复杂一些。不同指令的参数数量各不相同，数据流也可能跨越任意多个层级，因此形成的是图而非树。

通过考虑随着指令逐层加入而生成的值的数量，可以更准确地估算累积搜索空间大小。输入被视为位于第零层。如果每一层都应用所有指令，那么单参数指令必须从上一层获取输入，而双参数指令只需从上一层获取一个值，另一个值则可来自任意层。在这种方法中，给定层级的搜索空间节点数等于当前值的总数减去输入值的数量。

图6展示了不同子集大小对搜索空间大小的影响。结果表明，搜索空间大小大幅缩减，尤其是子集大小为10时。这主要是因为分支因子从约200降至10。请注意，图6中的结果已乘以相应的子集数量，不过这对结果的相对尺度影响不大。

显然，以这种方式将工作分配到多个节点，不仅能让工作更快完成，还能有效减少需要完成的总体工作量。

如果我们不够了解情况，可能会认为人类开发者编写代码时会使用所有可能的指令子集。结果表明事实并非如此。正如我们所见，人类编码者使用的指令分布高度偏斜，而指令对的分布偏斜程度更甚。这意味着，在人工编写的代码中，大多数指令对出现的概率微乎其微。由代码导出的任何指令子集中，指令对的分布也必然如此。

我们还可能认为，要编写任何可能的代码，都需要用到所有指令子集。实际上是否如此尚不得而知。使用不同的指令子集，往往有许多种方式可以编写功能等价程序。

无论如何，人们似乎只用所有可能的指令子集中的相对较小一部分，就能应付自如。这意味着许多子集很少被使用，甚至完全不会用到。每个不包含全部指令的指令子集，都对应于完整搜索空间中一个不同且部分重叠的子集。实际上，正是那些未被使用的指令子集，使搜索空间的规模得以缩小。

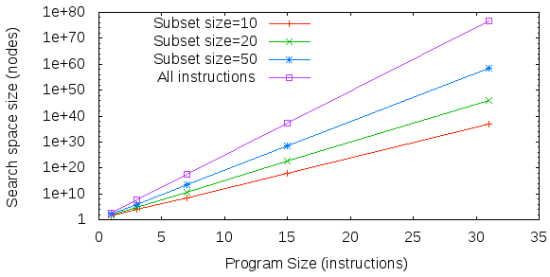

图6：不同指令子集大小下的搜索空间缩减。

4.5 子集重叠

指令子集之间往往会有重叠。这是有意为之，也反映出不同程序单元中使用的指令经常有所交集。这在一定程度上也是聚类过程造成的。子集大小为10时，重叠中位数约为20%；大小为50时，约为40%。因此，使用这些子集来划分工作时，可能会有人担心会浪费过多精力。

只有在我们要搜索的程序非常短时，这一点才成立。然而，随着程序越来越长，搜索空间也随之增大，子集之间共享的指令所产生的影响会迅速减弱，如图7所示。

重复工作量的估算采用了前文用于估算搜索空间大小的相同方法。在某一层级，重叠指令所对应的搜索空间子树大小除以指令数量为子集大小时树的大小，即为该层级的重复工作量。

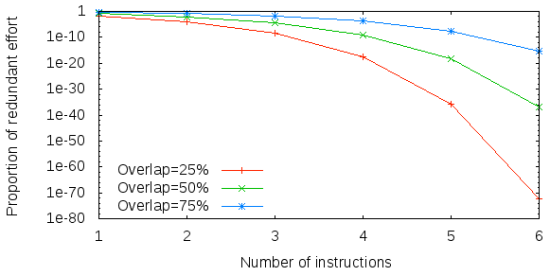

图7：不同子集重叠百分比下冗余活动的比例。

随着搜索树不断扩展，使用任何非重复指令生成的值都彼此不同

因此，每一层中仅由重复指令生成的值所占比例会迅速变得微不足道。在其他技术中也常常容忍类似的重复工作，例如迭代加深深度优先搜索，因为其相对开销被视为很小。

5 讨论与未来工作

单条指令的频率分布可以视为软件开发知识的一种基本形式。这种分布高度倾斜，但我们尚不清楚其成因，也不清楚它是否必然如此。许多软件开发者至少经过反思后会意识到这一点。然而，这似乎并非一个受到科学界广泛关注的课题。

类似地，指令子集中指令的共同出现情况，可以视为另一种非常基本的编码知识。鉴于此类知识有可能显著缩小搜索空间，因此可以将其视为一种重要的隐性知识。尽管如此，这一课题受到的关注甚至更少。这并不完全令人意外，因为从表面上看，这似乎不是一个有前景的研究方向——毕竟涉及的子集数量庞大，而且彼此之间存在大量重叠。

开展这项工作后，我们清楚地认识到，聚类与合并/扩增之间存在一项关键权衡。尽管通过更激进的聚类确实可以大幅减少子集数量，但代价是降低对未见代码的覆盖率。我们并不声称已经找到了这一连续区间上的最优位置，进一步研究这一领域会很有意义。不过，目前的结果足以支持在 Zoea 中初步实际部署这一方法，而相关工作目前仍在进行中。

承认候选解只会来自预先定义的指令子集，就意味着我们接受了一种折衷。我们愿意接受任何解，因为这种方法可能会快得多，但在少数情况下，它可能无法奏效。我们还需要进一步开展工作，以量化生成的解符合规格说明的实际运行成功率。

还可以采用其他生成指令子集的方法和替代聚类算法其中一些方法可能会生成更少的子集，和/或实现更高的覆盖率。

我们假定这些结果同样适用于其他命令式编程语言。大多数主流语言在指令层面都非常相似，尽管某些指令的名称和可用性可能有所不同。目前尚不清楚，这种方法是否有益于逻辑编程等截然不同的软件开发范式。

我们还假定所用的代码样本能够代表其他代码。所用代码来自一个规模庞大且广受欢迎的源代码库。为缓解这一问题，我们使用了来自许多不同代码仓库的大量样本。不过，来自较小代码仓库和其他托管网站的代码可能会得出不同结果。

某些指令在指令子集中出现得非常频繁。一种做法是从子集中移除出现频率最高的指令，并直接假定这些指令始终适用。这样会显著影响子集的规模和数量。然而，频繁指令与不频繁指令之间并没有明确的界限。

值得注意的是，指令子集能够生成比其派生来源多得多的程序。此外，覆盖率不足并不意味着无法生成等效程序。生成一个功能等价程序有许多不同的方法——有时会使用不同的指令。

某些指令子集提供的程序覆盖率远高于其他子集。可以利用这些信息，优先分配集群作业，从而提高及早找到解决方案的概率。

这种方法应适用于任何涉及搜索程序配置空间的问题。对于任何使用预定义指令列表的软件而言，集成起来都很简单。此外，也值得考虑类似方法是否适用于归纳式编程以外的领域。组合问题十分常见，而对工作进行划分以更好地利用大量核心，既是一项优先任务，也颇具难度。

当前工作将子集构造视为一项离线活动。在实际部署中，这也可以作为一个持续进行的过程。类似地，目前的重点是与人工编写的代码相对应的子集。未来，机器生成的代码可能会更加普遍，由此得到的指令子集也可能有所不同。无论如何，这

对 Zoea 来说，子集的来源和创建方式都没有影响。

长期以来，人们一直认为，应该将人工智能的神经方法与符号方法结合起来（Kautz 2022）。这也需要改变这些技术的使用方式。指令子集方法并非深度学习，但它朝着这一方向迈出了一步。虽然本文使用了“训练”一词，但这实际上是一种引出大量隐性知识的练习。从代码样本中提取的数据既通用又非常稀疏，经过的处理也很少。指令子集完全可供人类阅读，因此整个过程是透明的。该方法与深度学习共有的特点是：规则并非由人编写，而是通过机械方式大批量生成。

最后，令人有些意外的是，这一领域竟然还有新知可学。从某种意义上说，这表明我们仍然并不真正理解软件是什么。

6 伦理考量

从输入源代码中提取的信息仅限于每个程序单元内指令的共现情况。这是一份按字母顺序排列的指令名称列表。

需要注意的是，在大多数情况下，这些指令列表并非其来源代码所独有。相反，无论是作为字面复制内容，还是作为彼此的子集，它们都极为常见。否则，几乎所有现存代码都已经构成侵权，因此不能将使用指令子集视为权利侵害。

所提取的信息均不代表或包含任何可执行代码，也不包含有关任何程序、算法或其片段的信息。此外，除非纯属偶然，关于如何使用这些信息，也不存在任何可能导致生成相似或相同代码的因素。因此，无论是相关作者的版权或知识产权，还是任何许可协议的条款，都不可能以任何方式遭到违反。

7 结论

我们概述了一种利用指令子集划分归纳式编程搜索空间的技术。这种方法使我们能够通过为多个计算机核心分别分配彼此不同但有所重叠的指令子集，将归纳式编程任务分配给这些核心。测试表明，这些子集能够快速泛化，尤其是在合并后。这说明，即使面对完全不同的代码，它们也应能继续发挥良好作用。我们还展示了，使用指令子集能够缩小搜索空间，有时甚至能将其缩小数个数量级。最后，我们证明，随着程序规模增大，指令子集重叠所导致的重复工作很快就会变得微不足道。我们也认为，我们的方法符合伦理，因为它不会侵犯开源开发者的知识产权。

致谢

本项工作完全由 Zoea Ltd. 资助。Zoea 是 Zoea Ltd. 的商标。任何其他商标均归其各自所有者所有。作者承认并尊重开源开发者的权利。

参考文献

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; Amodei, D. (2020年). 语言模型是少样本学习者。arXiv预印本。arXiv:2005.14165 [cs.CL]. 纽约州伊萨卡：康奈尔大学图书馆。Cropper, A.; Dumancic, S.; Muggleton, S. H. (2020年). 三十而立：归纳逻辑编程的新思路。载于第二十九届国际联合人工智能会议论文集，IJCAI 2020。维也纳：ijcai.org。第4833–4839页。Flener, P., Schmid, U. (2008年). 归纳式编程简介。《人工智能评论》29(1), 45-62。doi.org/ 10.1007/s10462-009-9108-7。Galwani, S.; Hernandez-Orallo, J.; Kitzelmann, E.; Muggleton, S. H.; Schmid, U.; Zorn, B. (2015年). 归纳编程与现实世界相遇。《美国计算机协会通讯》58(11), 90–99。doi.org/ 10.1145/2736282。Kautz, H. (2022). 第三次人工智能浪潮：AAAI罗伯特·S·恩格尔莫尔纪念讲座。《人工智能杂志》。43(1): 93–104。doi.org/10.1609/aimag.v43i1.19122。

Kilby, P.; Slaney, J. K.; Thiébaux, S.; Walsh, T. (2006). 估算搜索树大小。载于第二十一届全国人工智能会议论文集，AAAI 2006。波士顿：AAAI出版社。1014-1019。Kitzelmann, E. (2010). 归纳式编程：程序综合技术综述。归纳式编程的方法与应用。计算机科学讲义5812, 50–73。柏林：施普林格出版社。Lemley, M. A., Casey B. (2021). 公平学习。《德克萨斯法律评论》99(4): 743-785。Martelli, A.; Ravenscroft, A.; Holden, S. (2017). Python袖珍指南，第3版。塞巴斯托波尔：O'Reilly Media, Inc. 国际标准书号：9781449392925。微软。(2022)。GitHub。https://www.github.com。访问日期：2022-11-06。Nguyen, N., Nadi, S. (2022). 对GitHub Copilot代码建议的实证评估。载于第19届IEEE/ACM国际软件仓库挖掘会议（MSR）论文集，2022年，第1-5页。doi.org/10.1145/3524842.3528470。Louridas, P.; Spinellis, D.; Vlachos. V. (2008年). 软件中的幂律。ACM软件工程与方法学汇刊18(1): 1-26。doi.org/10.1145/1391984.1391986。McDaid, E., McDaid, S. (2019). Zoea——无界可组合归纳式编程。arXiv预印本。arXiv:1911.08286 [cs.PL]. 纽约州伊萨卡：康奈尔大学图书馆。McDaid, E., McDaid, S. (2021). 基于知识的可组合归纳式编程。载于人工智能XXXVIII论文集：第41届SGAI国际人工智能会议，AI 2021，英国剑桥，2021年12月14–16日，施普林格出版社。doi.org/10.1007/978-3-030-91100-3 13.¯Petke, J.; Haraldsson, S.; Harman, M.; Langdon, W. B.; White, D.; Woodward, J. (2018). 软件遗传改进：综合调查。IEEE进化计算汇刊。22(3): 415-432。doi.org/10.1109/TEVC.2017.2693219。Ray, B.; Posnett, D.; Filkov, V.; Devanbu, P. (2014). GitHub上的编程语言与代码质量大规模研究。载于第22届ACM SIGSOFT软件工程基础国际研讨会（FSE 2014）论文集。纽约：计算机协会。第155–165页。doi.org/10.1145/2635868.2635922。Xu, F. F.; Alon, U.; Neubig, G.; Hellendoorn, V. J. (2022). 对代码大语言模型的系统评估。载于第6届ACM SIGPLAN国际机器编程研讨会论文集。纽约：计算机协会。第1–10页。doi.org/10.1145/3520312.3534862。