arXiv:2602.17688v1 [cs.LG] 5 Feb 2026

AnCoder：通过离散扩散模型实现锚定代码生成

Anton Xue* **LituRout*** **Constantine Caramanis Sanjay Shakkottai**德克萨斯大学奥斯汀分校 {anton.xue, litu.rout, constantine, sanjay.shakkottai}@utexas.edu

摘要

扩散语言模型为自回归代码生成提供了一种颇具吸引力的替代方案，能够对复杂的程序逻辑进行全局规划和迭代优化。然而，现有方法无法遵循编程语言严格的结构，因此常常生成无法执行的错误程序。为解决这一问题，我们提出了 AnchorTree，这是一种利用代码固有的结构化层次先验来显式锚定扩散过程的框架。具体而言，AnchorTree 利用抽象语法树优先处理语法和语义上显著的词元，例如关键字（如 if、while）和标识符（如变量名），从而建立结构框架，引导其余内容的生成。我们通过 AnCoder（一系列模型）验证了这一框架，并表明结构锚定扩散为高质量代码生成提供了一条参数高效的路径。

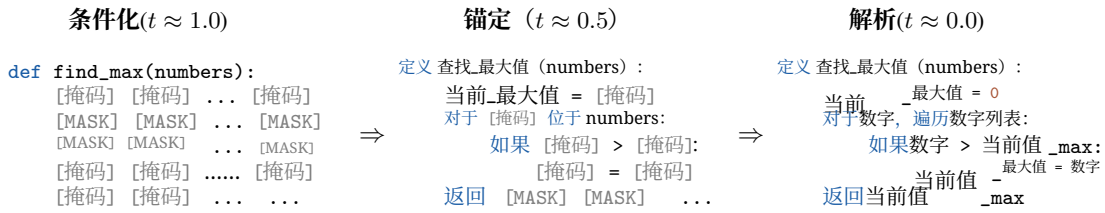

图1：**AnCoder：锚定代码生成**。生成始于条件化，此时输入提示以掩码填充。在锚定阶段，根据句法层级解掩码句法和语义上显著的词元，例如关键词和标识符。借助这些锚点，模型便能在最后的解析 阶段更好地解掩码其余词元。按照这一顺序去噪，生成的代码比扩散语言模型基线具有更高的功能正确性。

1 引言

大语言模型驱动的编码智能体如今已广泛应用于软件工程，任务范围从简单脚本编写到复杂基础设施架构设计 [2, 4, 7, 11, 17, 27, 34, 49]。然而，大多数实现依赖自回归（AR）生成，按从左到右的顺序逐个解码词元。对于语法和控制流这类本质上并非顺序结构的内容强加顺序生成，会使自回归模型产生结构失配，从而限制高质量代码所需的全局规划能力 [15]。此外，这种自回归过程也与人类的推理方式相悖，因为它忽略了管理大型代码库所必需的“由粗到细”策略 [18]。因此，大语言模型生成的代码常常表现出结构特异性和“代码异味”，给人工维护者带来认知摩擦 [1, 39, 44]。

这些摩擦的一个关键来源是，代码在本质上与自然语言序列截然不同 [8, 19]。从语法上看，圆括号、方括号和缩进必须在多行之间保持配对；从语义上看，

*贡献相同。

预印本。

变量必须处于正确的作用域中，执行流程也可能发生分支、循环，或以不同于函数定义的顺序执行 [15, 30]。自然语言通常能够容忍轻微的语法和措辞错误，但程序逻辑中的任何违规都可能轻易导致代码无法运行 [14, 23, 41]。这些差异促使我们探索顺序代码生成的替代方案，尤其是那些能够推理全局结构并自行纠正潜在错误的方案 [33]。

扩散语言模型（DLM）提供了这样一种替代方案，近期已开始被用于代码生成 [9, 16, 22, 25, 48]。通过将生成定义为从噪声词元序列出发的迭代去噪过程，DLM 首先勾勒高层结构和约束，再逐步通过低层实现细节加以完善。这种由粗到细生成不仅反映了人类程序员的认知策略 [18], ，还带来了更强的长度外推能力，使 DLM 能够在长序列上保持连贯性，而自回归模型在此类序列上的表现则会下降 [25]。

尽管如此，即使是经过代码训练的 DLM，也可能难以生成正确的程序。与早期版本的自回归代码模型类似，DLM 同样很容易生成会崩溃的程序 [25]。为缓解这些问题，越来越多的研究提出了用于提升生成质量的测试时适配方法，例如通过约束解码方法确保生成语法有效代码 [31, 46]。虽然这些方法有效且与我们的工作互为补充，但它们本质上都是事后处理：它们在生成过程中进行修正，而不是通过训练来提升模型对代码结构的理解。

我们的核心观察是，代码在其生命周期的每个阶段都呈现出固有的层次结构。在解析过程中，编译器首先将词元序列提升为抽象语法树（AST），以消解语法歧义 [3]。这种层次特性会在中间分析阶段再次显现：代码优化依赖于不同于语法结构的支配树，以推理控制流和循环嵌套 [12]。类似地，机器代码生成通常被建模为一个树覆盖问题，其中表达式子树会映射为指令序列 [24]。最后，层次结构也支配着执行本身：运行时调用栈组织函数调用，确保变量生命周期严格嵌套 [40]。代码中层次结构的普遍存在表明，成功的扩散语言建模不仅需要理解词元级分布，还需要跨多个抽象层次进行层次推理。

近期的一种方法提出，锚定 可作为一种有效框架，用于在扩散语言模型（DLM）的训练和生成中编码层次结构 [36, 37]。其核心思想是，扩散语言模型应首先揭示信息量较大的输出词元（例如名词、动词），然后由这些词元为周围细节的生成提供“锚定”。值得注意的是，与非锚定模型相比，锚定扩散语言模型在语言建模基准上取得了显著的性能提升，而且通常只需使用其一小部分参数。

然而，现有方法采用了无法捕捉代码特定层次结构的硬锚定方案。在硬锚定中，一个词元要么被锚定，要么未被锚定，且所有被锚定的词元都被赋予相同的权重。虽然这种方法建立了锚定词元与未锚定词元之间的层级关系，但并未进一步区分不同的锚点。因此，硬锚定无法区分顶层函数定义中的锚点与深度嵌套的 for循环中的锚点，也无法区分 if-else语句两个分支中的锚点。

为了解决硬锚定的局限性，我们提出了**AnchorTree**，一种针对代码特定层次结构设计的软锚定框架。在 AnchorTree 中，我们首先选择关键字（例如 if、for）和标识符（例如变量名）作为锚点词元，因为它们指定了句法结构和语义意图 [19]。然后，我们根据每个锚点在抽象语法树中的位置为其分配权重，越接近根节点的词元获得的权重越高。这种软锚定方案促使句法和语义上显著的词元按照其基于抽象语法树的句法层级顺序进行去噪，我们在图1中对此进行了突出说明。

我们的贡献总结如下：**(1)** 我们提出了 AnchorTree，这是一种软锚定框架，能够帮助扩散语言模型更好地学习代码的句法和语义层次结构。**(2)** 我们使用 AnchorTree 训练了一系列锚定扩散模型，并将其命名为 **AnCoder**。实验表明，它在 HumanEval 和 MBPP 等标准代码基准测试上的表现优于扩散语言模型基线。**(3)** 据我们所知，我们首次证明，软层次锚定方案能够显著提升扩散语言模型的代码生成质量。

2 相关工作

扩散语言模型 [26, 32] 通过学习逆转渐进加噪过程来生成词元序列，该过程通常包括随机词元掩码 [29, 35, 38, 43] 或翻转 [5, 29]。在实践中，扩散语言模型的生成过程从填充掩码的输入序列开始，并通过迭代解掩

（或翻转）词元，直到得到最终的连贯输出。这个过程展现了自我纠正生成错误 [47] 并提升整体连贯性的能力，尤其适用于结构化输出 [25]。然而，扩散语言模型的生成质量往往不如自回归模型 [32]。其中一个关键原因是，信息量丰富的词元（例如名词、动词）往往在去噪过程 [36], 中出现得太晚，使得生成的早期阶段缺少至关重要的条件信息。

锚定扩散模型。 为了解决信息性词元缺失的问题，Rout 等人 [36]提出了锚定扩散语言模型，明确优先对这类词元进行去噪。通过在去噪过程中尽早揭示这些“锚点词元”，模型便能依据信息更丰富的上下文来执行后续去噪步数。研究表明，锚定在多个领域均有效，包括自回归模型中的自然语言生成和数学推理 [36], ，并且也可以通过测试时适应 [37]以最小的架构改动实现。然而，现有方法将词元标记为硬锚点，这对代码生成未必有效。

用于代码生成的扩散模型。 越来越多的研究开始探索将扩散语言模型用于代码生成，包括 DiffuCoder [16], 、Dream [48],、CoDA [9],、Mercury [22],和 Gemini Diffusion [13]。这些方法主要是将扩散语言模型应用于代码数据集，重点关注架构选择、训练目标或解码策略。尽管这些方法证明了基于扩散的代码生成的可行性 [25], ，但它们在训练期间并未显式编码程序结构，因此仍容易出现语法有效性和语义错误 [31, 46]。

面向代码的结构感知扩散。 在标准掩码扩散中，干净序列的每个位置都会被独立地掩码，并训练去噪模型来重建原始词元。TreeDiff [50]对此进行了修改，引入了一个非位置独立且还取决于抽象语法树深度的前向加噪计划。然而，TreeDiff 保持去噪过程不变，仍使用标准的单阶段解码器。相比之下，我们明确修改了去噪过程，采用两阶段架构：首先使用锚点网络 按照句法层级的顺序，优先解掩具有信息量的词元；这些词元随后作为具有信息量的锚点，帮助去噪网络 解掩剩余词元。

3 预备知识

符号说明。 令 $\mathcal{V} = \{1, \ldots, K-1, K\}$ 表示词汇表中的词元集合，其中 K 是一个特殊的“掩码”词元。给定一个长度为 L 的词元序列 $x = (x^1, \ldots\ldots, x^L) \in \mathcal{V}^L$，我们可以将其等价地表示为独热向量序列 $\mathbf{x} = (\mathbf{x}^1, \ldots\ldots, \mathbf{x}^L)$，其中每个 $\mathbf{x}^l \in \{0,1\}^K$ 仅在位置 x^l 上为热。对于特殊掩码词元，令 $\mathbf{m} = (0, \ldots\ldots, 0, 1)^\top$ 表示其独热编码。我们假设训练数据 $\mathbf{x}$ 是从未知分布 $q(\cdot)$ 中采样得到的，该分布的支持集为 $\mathcal{V}^L$。

3.1 离散扩散语言模型

离散扩散语言模型 [5, 29, 38, 43, 45] 基于一个前向过程，该过程逐步对干净的词元序列 $\mathbf{x} = (\mathbf{x}^1, \ldots, \mathbf{x}^L)$ 加噪，最终将其转化为完全掩码的序列，如下所示：

$$(\mathbf{x}^1, \ldots, \mathbf{x}^L) \longrightarrow (\mathbf{z}_t^1, \ldots, \mathbf{z}_t^L) \longrightarrow (\mathbf{m}, \ldots, \mathbf{m}), \tag{1}$$

其中，时间从 $t = 0$ （全部干净）流逝至 $t = 1$ （完全掩码），且每个 $\mathbf{z}_t^l$ 都是一个潜在独热向量，随机取值为 $\mathbf{x}^l$ 或 $\mathbf{m}$ 。这一前向加噪过程在每个位置 l独立进行，其条件分布定义如下：

$$q(\mathbf{z}_t|\mathbf{x}) = \prod_{l=1}^{L} q(\mathbf{z}_t^l|\mathbf{x}), \quad q(\mathbf{z}_t^l|\mathbf{x}) = \mathrm{Cat}(\mathbf{z}_t^l; \alpha_t \mathbf{x}^l + (1-\alpha_t)\mathbf{m}), \tag{2}$$

且 $\alpha_t \in [0,1]$ 是一个单调递减的噪声调度，满足 $\alpha_0 = 1$ 和 $\alpha_1 = 0$。实践中，我们将前向过程离散化为 T个时间步，其中，对于索引 $i \in \{1, \ldots, T\}$， $t(i) = \frac{i}{T}$且 $s(i) = \frac{i-1}{T}$。当未指定 i时，令 s, t表示任意两个连续的时间步。

生成式扩散建模的目标是学习反向过程，也就是逆转时间流，将完全掩码的序列去噪为干净序列。为此，请注意，每个位置的前向条件分布都是一个具有如下转移概率的马尔可夫过程：

$$q(\mathbf{z}_t^l|\mathbf{z}_s^l) = \mathrm{Cat}\Big(\mathbf{z}_t^l; \tfrac{\alpha_t}{\alpha_s}\mathbf{z}_s^l + (1 - \tfrac{\alpha_t}{\alpha_s})\mathbf{m}\Big), \tag{3}$$

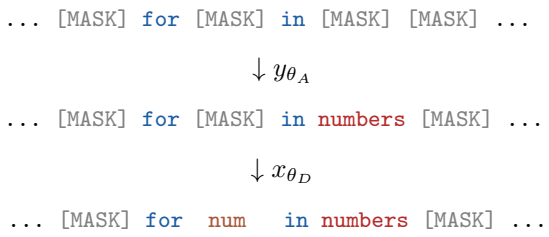

图 2： **ADLM 前向过程**。给定部分掩码的 z_t （顶部），锚点网络 y_{θ_A} 专注于解掩锚点词元 numbers （中部），这有助于去噪网络 x_{θ_D} 解掩其余词元，例如 num （底部）。锚点词元会在训练前标注（见第 5 节）。

其中 $\mathbf{z}_0^l = \mathbf{x}^l$。直观地说，这意味着某个位置一旦被掩码，在前向过程的剩余阶段就会一直保持掩码状态。我们的目标是逆转这一前向马尔可夫转移；假设 已知干净目标 $\mathbf{x} \sim q(\cdot)$，那么最优反向转移的形式为

$$q(\mathbf{z}_s^l|\mathbf{z}_t^l, \mathbf{x}^l) = \begin{cases} \mathrm{Cat}(\mathbf{z}_s^l; \mathbf{z}_t^l), & \mathbf{z}_t^l \neq \mathbf{m} \\ \mathrm{Cat}\left(\mathbf{z}_s^l; \frac{\alpha_s - \alpha_t}{1-\alpha_t}\mathbf{x}^l + \frac{1-\alpha_s}{1-\alpha_t}\mathbf{m}\right), & \mathbf{z}_t^l = \mathbf{m}. \end{cases} \tag{4}$$

也就是说，未掩码位置保持未掩码状态，而掩码位置则以一定概率翻转为$\mathbf{x}^l$。

然而，在生成过程中，干净目标序列$\mathbf{x}$ 是未知的，也就是说，我们没有样本$\mathbf{x} \sim q(\cdot)$，因此无法直接实现公式（4）。相反，我们的思路是使用一个由 θ 参数化的模型，根据当前迭代状态$\mathbf{z}_t$预测$\mathbf{x}$ 应当是什么。我们将此预测表示为

$$\mathbf{x}_\theta(\mathbf{z}_t) = (\mathbf{x}_\theta^1(\mathbf{z}_t), \ldots, \mathbf{x}_\theta^L(\mathbf{z}_t)), \tag{5}$$

其中，每个$\mathbf{x}_\theta^l(\mathbf{z}_t)$都是一个在 $\mathcal{V}$ 上定义的概率向量，由此可得恒等式 $p_\theta(\mathbf{x}^l|\mathbf{z}_t) = \langle \mathbf{x}_\theta^l(\mathbf{z}_t), \mathbf{x}^l \rangle$。我们还采用两项约束：零掩码，即每个$\mathbf{x}_\theta^l(\mathbf{z}_t)$在掩码词元上的概率质量为零；以及延续式解掩，即当$\mathbf{z}_t^l \neq \mathbf{m}$时，未掩码的$\mathbf{x}_\theta^l(\mathbf{z}_t) = \mathbf{z}_t^l$。据此，我们将由 θ参数化的反向去噪过程定义如下：

$$p_\theta(\mathbf{z}_s|\mathbf{z}_t) = \prod_{l=1}^{L} p_\theta(\mathbf{z}_s^l|\mathbf{z}_t), \quad p_\theta(\mathbf{z}_s^l|\mathbf{z}_t) = \begin{cases} \mathrm{Cat}(\mathbf{z}_s^l; \mathbf{z}_t^l), & \mathbf{z}_t^l \neq \mathbf{m} \\ \mathrm{Cat}\left(\mathbf{z}_s^l; \frac{\alpha_s - \alpha_t}{1-\alpha_t}\mathbf{x}_\theta^l(\mathbf{z}_t) + \frac{1-\alpha_s}{1-\alpha_t}\mathbf{m}\right), & \mathbf{z}_t^l = \mathbf{m}. \end{cases} \tag{6}$$

使用最大似然估计的标准负 ELBO（NELBO）框架，对以 θ 参数化的去噪过程进行训练。对于此处描述的扩散方案，损失等价于

$$\mathcal{L}_{\mathrm{NELBO}}(\mathbf{x}; \theta) = \sum_{i=1}^{T} \mathop{\mathbb{E}}_{\mathbf{z} \sim q(\cdot|\mathbf{x})} \left[\lambda_i \sum_{l=1}^{L} \log \langle \mathbf{x}_\theta^l(\mathbf{z}_{t(i)}), \mathbf{x}^l \rangle \right], \quad \lambda_i = \frac{\alpha_{t(i)} - \alpha_{s(i)}}{1 - \alpha_{t(i)}}, \tag{7}$$

其中， λ_i 是由噪声调度决定的时间索引 i 处的权重。

3.2 锚定扩散语言模型

标准扩散语言模型的一个局限是，生成所需的关键词元可能要到去噪过程后期才会解除掩码。当这类词元在大部分去噪轨迹中都处于掩码状态时，模型就必须在缺乏充分条件信息的情况下生成依赖于它们的内容，从而降低生成质量。为了解决这一问题，Rout 等人 [36] 提出了锚定扩散语言模型，通过显式地以一部分重要词元（称为锚定标记）作为生成条件。其核心直觉是，显著词元的存在会简化其余词元的生成，见图 2。

锚点化扩散语言模型（ADLM）。锚定扩散将单次去噪预测分解为两个阶段，其中标准预测网络$\mathbf{x}_\theta$（$\mathbf{z}_t$）被改写为以下组合

$$\mathbf{x}_\theta(\mathbf{z}_t) = \mathbf{x}_{\theta_D}\big(\mathbf{y}_{\theta_A}(\mathbf{z}_t)\big), \tag{8}$$

其中， $\theta = \{\theta_A, \theta_D\}$ 是锚点网络 $\mathbf{y}_{\theta_A}$ 和去噪网络 $\mathbf{x}_{\theta_D}$的参数划分。两个网络都在同一词汇表 $\mathcal{V}$上输出概率向量，我们将这种参数化形式称为锚定扩散语言模型。

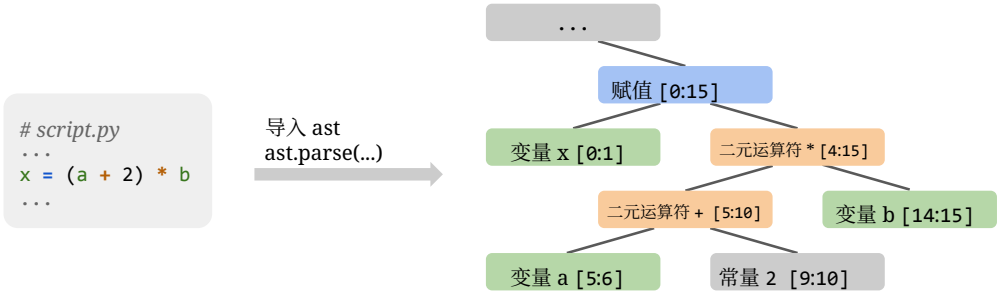

图3：**代码的句法层级**。一串源代码标记（左侧）被解析为抽象语法树（右侧）。此处，一条赋值语句构成一个子树，并嵌套于更广泛的程序上下文中（以 ... 表示）。抽象语法树节点标注了类型、数据和相对字符区间（例如 [5:10]），这些信息将高层语法构造映射到其顺序词元位置。

锚定训练目标。 训练过程中，会同时监督最终的去噪预测和中间锚点预测。由此得到锚定负 ELBO（ANELBO）目标：

$$\mathcal{L}_{\text{ANELBO}}(\mathbf{x};\theta_A,\theta_D)=\sum_{i=1}^{T}\mathop{\mathbb{E}}_{\mathbf{z}\sim q(\cdot|\mathbf{x})}\left[\lambda_i\sum_{l=1}^{L}\underbrace{\log\langle\mathbf{x}_\theta^l(\mathbf{z}_{t(i)}),\mathbf{x}^l\rangle}_{\text{NELBO term}}+\underbrace{\mu\log\langle\mathbf{y}_{\theta_A}^l(\mathbf{z}_{t(i)}),\mathbf{y}^l\rangle}_{\text{Anchor term}}\right],\tag{9}$$

其中 $\mu>0$ 控制锚点监督的强度，$\mathbf{y}^l$ 表示位置 l处的锚点标签。这会对中间预测施加辅助监督，以鼓励在去噪过程中更早恢复锚定标记。

锚点规范。 锚定扩散模型要求在训练之前通过预处理步骤指定锚点监督，该步骤决定哪些词元接收锚点标签。先前的工作 [36]使用依赖于任务的启发式方法定义锚点，例如标记频率、用于数学推理的数字提取，或用于自然语言任务的词性标注。这些方法会引入硬锚定，即每个词元要么被指定为锚点，要么被视为非锚点。
然而，锚定不必如此二元。一般而言，词元可能表现出不同程度的结构重要性，这表明可以采用软锚定 形式，使词元接受分级的锚点监督。在本工作中，我们利用代码固有的结构先验来定义锚点规范。接下来，我们将在第4节中介绍这一框架。

4 AnCoder：基于抽象语法树的锚定代码生成

我们的核心方法是利用代码固有的层次结构（尤其是抽象语法树（AST））来锚定扩散语言模型的生成。接下来，我们概述抽象语法树及其锚定潜力。然后，我们通过一项实验表明，抽象语法树所提供的层次信息实际上对去噪过程是有用的。我们展示如何将这些洞见转化为一种有效的软锚定策略**AnchorTree**，并利用它来训练**AnCoder** 扩散语言模型系列。

4.1 抽象语法树

抽象语法树是一种用于程序编译与分析的树状程序语法表示 [2] [3]。图3给出了一个示例，其中每个抽象语法树节点都标注了其语法类型（例如赋值、二元运算）、相关数据（例如变量名或字面值），以及一个相对字符区间，用于指示其关联代码区域的起止位置。具体而言，图3展示了嵌入在更广泛的Python脚本中的语句“x = (a + 2) * b”的抽象语法（子）树。此处，字符区间 [4:15] 对应于子表达式(a + 2) * b，而 [5:10]对应于内部二元运算a + 2，请注意其中存在空白字符。

抽象语法树具有两个适用于锚定扩散的特性。首先，抽象语法树节点按语法 y
语法 结构，例如变量赋值、函数调用和类定义，这为 a

[2]Python 抽象语法树规范： https://docs.python.org/3/library/ast .html

```
1 l4 def binary_search(arr, target):
2     left, right = 0, len(arr) - 1
3  l3 while left <= right:
4         mid = (left + right) // 2
5      l2 if arr[mid] == target:
6             return mid
7         elif arr[mid] < target:
8             left = mid + 1
9         else:
10            right = mid - 1
11    return -1
```

```
def binary_search(arr, target):
    left, right = 0, mid(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

```
def binary_search(arr, target):
    left, right = 0, mid(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

图 4：**基于抽象语法树的排序与 AnchorTree 权重** $\mu(l) = \omega(l) \cdot \eta(l)$。（左）从 mid 开始并按抽象语法树层次排序的升链示例 $l_0 \preceq l_1 \preceq \cdots \preceq l_4$。（中）我们将关键字（蓝色）和标识符（橙色）作为锚点，其中 let $\omega(l) = 1$。（右）每个位置均按抽象语法树深度加权，其中 let $\eta(l') \geq \eta(l)$ if $l' \succeq l$ 在基于抽象语法树的偏序中。

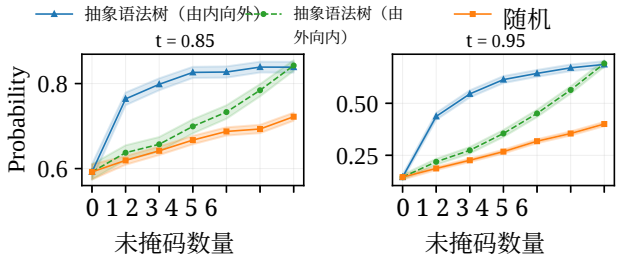

图 5：**依据抽象语法树祖先关系进行解掩可提升性能**。累积解掩目标位置 l_0 的抽象语法树祖先 l_1，l_2，.....比随机解掩其他词元更能提升性能。此外，优先揭示最接近目标的抽象语法树祖先（由内向外）时，性能提升最快。在中等噪声（$t = 0.85$）和高噪声（$t = 0.95$）设置下，这一趋势均成立。

锚点选择提供了丰富的设计空间。例如，可以选择函数签名或返回语句中的变量作为锚点，这通过标准的 AST 遍历算法即可轻松实现。其次，AST 为源代码定义了一种规范的句法层级：所有表达式和子表达式都可以根据区间包含关系自然地形成偏序。这种层级与扩散中的由粗到细去噪理念自然契合：粗略来说，我们希望先解除更靠近 AST 根节点的词元（例如，通常是函数与类定义）的掩码，再处理更靠近叶节点的词元（例如，深度嵌套的 if 语句）。

词元上的抽象语法树层次。 为形式化语法与去噪之间的联系，我们将抽象语法树层次映射到分词序列 $\mathbf{x} = (\mathbf{x}^1, \ldots, \mathbf{x}^L)$。首先，对于每个位置 l，我们将其与字符区间和 $\mathbf{x}^l$ 的字符区间相交的最深抽象语法树节点关联起来，并将该节点记为 node(l) [3]。接着，我们通过规定以下条件来定义词元位置上的偏序：$l \succeq l'$，当且仅当 node(l) 是抽象语法树中 node(l') 的严格祖先，或者 node(l) = node(l') 且 $l < l'$。例如，当分词器将 quicksort 这样的标识符拆分成多个词元时，就会出现后一种情况。在此偏序下，可以将 $l \succeq l'$ 理解为 l 在语法上比 l' 更粗粒度。换句话说，抽象语法树中位置较高的节点对应程序中语法上更粗粒度的元素，而树中更深的位置则对应更细粒度的细节。

4.2 抽象语法树对去噪有用吗?

假设给定一个部分掩码序列，其中位置 l 被掩码，并且允许我们将另一个位置解掩，作为 l 的提示。如果句法祖先关系有用，那么我们应该预期，解掩 l 的某个祖先位置，要优于解掩一个非祖先位置。

我们在一个预训练离散扩散模型和一个代码数据集 [4] 上检验这一假设。令 $\mathbf{x}$ 为代码的分词序列，$\mathbf{z}_t$ 为其加噪版本。我们额外在位置 $l_0, l_1, \ldots, l_k$ 处对 $\mathbf{z}_t$ 进行掩码，这些位置被特意选取为抽象语法树中的一条连续链。也就是说，$l_0 \preceq l_1 \preceq \cdots \preceq l_k$，且不存在另一个 l' 使得

[3]这解释了扩散语言模型分词器与抽象语法树解析器之间可能存在的字符级边界错位。[4]使用预训练的 8.09亿参数模型（阶段 1 和 2），并从监督微调数据集中选取 2000 个测试样本。参见第 5 节。

对任意 i 都有 $l_i \preceq l' \preceq l_{i+1}$。这使我们能够单独检验对 l_0 的祖先进行解掩如何可能改善 l_0 处的模型预测。例如，使用图 4（左）中的二分查找实现，其中令 l_0 对应于 return mid 行中的 mid 标记（图 4 第 6 行）。因此，从 mid（l_0）开始的一条可能的 AST 祖先标记链为 return（l_1）、if（l_2）、while（l_3）和 def（l_4）。

我们考虑两种对 $l_1, \ldots, l_k$ 进行解掩的策略，这些位置按其与 l_0 的基于 AST 的邻近程度排序。在由内向外顺序中，我们逐步优先解掩距离 l_0 最近的抽象语法树祖先；在 由外向内 顺序中，我们反转这一过程，优先解掩最远的祖先。在上述示例中：我们从 mid (l_0)、return (l_1)、if (l_2)、while (l_3) 和 def (l_4) 全部被掩码的状态开始。然后，对于由内向外顺序，我们先解掩 return (l_1)，再解掩 if (l_2)，依此类推，并在每次解掩迭代时查询模型在被掩码位置 l_0 处的预测。相反，对于由外向内顺序，我们先解掩 def (l_4)，再解掩 while (l_3)，依此类推，同样在每次迭代时查询模型在 l_0 处的预测。

我们将由内向外和由外向内的解掩顺序与一种基线进行比较，在该基线中，改为解掩这条链之外的若干随机位置。我们在图 5 中展示结果，从中观察到，解掩祖先位置能显著改善第 l_0 个位置上的预测。特别是，优先逐步解掩最近的祖先（由内向外）能最快地提高在 l_0 处预测正确标签的概率。

4.3 基于 AST 的分层软锚定

上述核心洞见是，揭示被遮蔽位置的祖先节点可以提升预测效果。此外，揭示的祖先节点越近，性能提升越大。这表明，可以采用一种锚定去噪方案，基于句法层级逐步取消对词元的遮蔽，首先取消对最靠近 AST 根节点的节点的遮蔽。

我们在**AnchorTree** 框架中，通过将 ANELBO 训练目标（公式 (9)）中的锚点权重 μ 参数化为依赖于位置且形式为

$$\mu(l) = \omega(l) \cdot \eta(l), \tag{10}$$

其中，$\omega(l) \in \{0, 1\}$ 是一个指示函数，用于指明哪些位置被锚定；$\eta(l) > 0$ 是一个随 AST 深度单调递减的调度函数，且当 $\eta(l') \geq \eta(l)$ 时有 $l' \preceq l$。这就形成了一种软锚定 方案：我们使用 ω 来选取锚定标记，然后通过 η为它们赋予重要性。

具体而言，对于关键字和标识符标记，我们令 $\omega(l) = 1$，并在图 4（中）中将其可视化。选择关键字是因为它们是程序表达句法结构的主要方式，例如，def 表示函数定义的开始，而 if 则表示 if 语句。另一方面，函数名和局部变量等标识符传达语义意图，尤其是在人工编写的代码 [19]中。因此，同时锚定关键字和标识符，使我们能够从 AST 中提取句法和语义信息；AST 是一种主要为句法分析而设计的数据结构。

接下来，我们通过调度函数 η 来捕捉层次顺序，该函数随节点(l)的抽象语法树深度单调递减，我们在图 4（右）中对其进行了可视化。作为实验中的一种具体实现，我们采用如下形式的指数深度衰减

$$\eta(l) = \gamma \cdot \exp(-\beta \cdot \max(\mathsf{depth}(l) - d_0, 0)), \tag{11}$$

其中，γ针对每种锚定策略进行调整，并固定为 $\beta = 0.7$；depth(l)表示节点(l)的抽象语法树深度；$d_0 = 2$表示开始衰减时的抽象语法树深度。请注意，当 $\beta = 0$时，我们得到硬锚定，其中$\mu = \gamma$。这种对 η 的参数化确保父节点中的词元比子节点中的词元获得更高权重，因此会被锚定模型优先处理。

将关键词和标识符锚定词元与基于抽象语法树深度的加权相结合，便定义了 **AnchorTree**。这一用于代码扩散的层次化软锚定框架构成了我们的 **AnCoder** 模型的基础，接下来我们将对此进行介绍。

5 实验

本节介绍如何将 AnchorTree 框架应用于训练参数量为 809M 的 AnCoder 模型，并在代码基准测试上将其与扩散语言模型基线进行评估。

架构。 我们采用第 3.2 节所述的两阶段锚定去噪器架构来初始化 AnCoder 模型。AnCoder 模型的总参数量为 809M：锚点网络 θ_A 包含 494M 个参数

表 1：**在标准代码基准测试上的评估**。我们报告了在 HumanEval 和 MBPP 上，经过 $T = 2048$ 扩散步数后的语法有效性（语法（%））和功能正确性（Pass@1（%））。AnchorTree 的表现显著优于硬锚定（关键词、标识符）和空锚定（$\gamma = 0$）基线。所有 809M 模型均采用第 3.2 节讨论的两阶段锚定去噪器架构。

			HumanEval		MBPP	
模型	Size	策略	语法（%）	Pass@1（%）	语法 (%)	Pass@1 (%)
自回归 [21]	494M	–	–	30.50	–	39.30
自回归 [21] (重新评估)	494M	–	93.90	26.83	95.59	36.55
MDLM [38]	494M	–	62.41	2.35	71.83	6.65
MDLM [38]	809M	–	65.58	3.29	74.46	6.34
AnCoder（我们的方法）	809M	Null	72.24	4.80	79.11	8.64
AnCoder（我们的方法）	809M	关键词	73.05	4.57	78.47	8.81
AnCoder（我们的方法）	809M	标识符	72.65	4.73	79.25	8.95
AnCoder（我们的方法）	809M	AnchorTree	**74.72**	**5.45**	**80.12**	**9.10**

（24 个 Transformer 层），使用 Qwen2.5-Coder-0.5B [21]的权重进行初始化；去噪网络θ_D 包含 315 M 个参数（12 层），使用 Qwen 的前 12 层进行初始化。参数量之所以如此，是因为 Qwen 的词嵌入与语言模型头共享 136M 个参数，且它的 24 个 Transformer 层每层包含 14.9M 个参数。为启用扩散语言建模，我们将 Qwen 的因果注意力转换为双向注意力。我们将所有输入截断或填充至上下文长度 $L = 1024$。

训练。我们使用 OpenCoder [20] 的数据集，并将训练分为三个阶段：自回归到扩散语言模型预训练适配（197K 步）、混合代码和数学推理任务的中期训练（171K 步），以及最后使用以 Python 为重点的数据集进行监督微调（50K 步）。我们使用 AdamW [28]，设置如下：学习率 10^{-4}、批次大小 512、余弦学习率调度器，以及 0.01 的预热比例。我们为扩散加噪过程使用余弦噪声调度 [51]，其中 $\alpha_t = \cos^2\left(\frac{t}{T} \cdot \frac{\pi}{2}\right)$；该设置也用于我们的 NELBO 和 ANELBO 损失函数（参见公式（7）、公式（9））。此设置用于训练 AnCoder 和基线方法。更多数据集与训练细节见附录 A。

推理。我们同样为反向过程使用余弦噪声调度。我们使用$T \in \{256, 512, 1024, 2048\}$ 反向去噪步数和 0.8 的采样温度对模型进行评估。对于 AnCoder 的实现，我们使用 0.1 的重新掩码率；我们发现这有助于提升性能。MDLM 基线使用 0.0 的重新掩码率。不过，我们不会将重新掩码率纳入扩散语言模型训练损失的时间索引权重 λ_i 中（参见公式（7））。

基准测试。我们在两个标准代码生成基准测试上进行评估：*HumanEval* [10], 包含 164 道 Python 编程题；以及*MBPP* [6], 包含 974 道题目。每个基准测试都提供一条描述任务的自然语言提示，以及一组用于评估的单元测试。我们报告*Pass@1*（定义为生成的解决方案通过全部单元测试的概率），以及语法有效性（用于衡量生成代码能否成功解析） [10]。对于 HumanEval 的每条提示，我们生成 20 个样本；对于 MBPP，则生成 10 个样本。附录 A 提供了示例提示和单元测试。

基线。我们的主要基线是 MDLM [38, 42], ，我们使用与 AnCoder 相同的骨干网络初始化方式和训练方案对其重新训练，以进行受控比较。所有 809M 模型，包括 809M MDLM基线，都采用第 3.2 节所述的两阶段锚定器-去噪器架构。此外，我们还与 Qwen2.5-Coder-0.5B [21],进行比较，后者是用于初始化 AnCoder 模型的 494M 自回归模型。

5.1 标准基准测试上的评估

我们在表 1 中展示了不同模型在 HumanEval 和 MBPP 基准测试上的性能。具体而言，AnCoder 始终优于同等规模的 MDLM 和锚定基线，其中 AnchorTree 锚定方案取得了最大增益。

在 HumanEval 上，与 809M 的 MDLM 相比，采用 AnchorTree 的 AnCoder 将语法有效性从 65.58% 提升至 74.72%（绝对提升 +9.14，相对提升 +13.9%），并将 Pass@1 从 3.29% 提升至 5.45%（绝对提升 +2.16，相对提升 +65.7%）。MBPP 上也观察到了类似趋势：AnCoder 将语法有效性从 74.46% 提升至 80.12%（绝对提升 +5.66，相对提升 +7.6%），并将 Pass@1 从 6.34% 提升至 9.10%（绝对提升 +2.76，相对提升 +43.5%）。值得注意的是，AnchorTree 的表现也优于其他基于关键词和标识符的锚定策略，这表明，与硬锚定相比，软锚定是更适用于代码的框架。

表 2：**在 HumanEval 上扩展测试时计算**。我们报告了语法有效性（语法（%））和功能正确性（Pass@1（%））随去噪步数 T 的变化情况。具体而言，增加去噪步数可以提升性能，其中锚定模型受益最大。

			$T=256$		$T=512$		$T=1024$		$T=2048$	
方法	Size	锚定	语法	Pass@1	语法	Pass@1	语法	Pass@1	语法	Pass@1
MDLM	494M	–	57.80	2.32	61.68	2.35	61.71	2.71	62.41	2.35
MDLM	809M	–	61.83	3.23	62.53	3.02	64.33	2.84	65.58	3.29
AnCoder	809M	Null	64.27	3.20	66.65	3.87	71.49	4.48	72.24	4.80
AnCoder	809M	关键词	62.07	2.96	66.25	3.17	69.57	**4.88**	73.05	4.57
AnCoder	809M	标识符	63.17	**3.38**	67.68	3.75	70.52	3.96	72.65	4.73
AnCoder	809M	AnchorTree	**64.48**	3.14	**68.75**	**4.15**	**71.83**	4.27	**74.72**	**5.45**

表3：**在MBPP上进行测试时计算扩展**。与表2类似，我们报告了不同模型在不同去噪步数T下的语法有效性（语法（%））和功能正确性（Pass@1（%））。我们同样观察到，随着 T 增加，性能有所提升，其中锚定模型受益最大。

			$T=256$		$T=512$		$T=1024$		$T=2048$	
方法	Size	锚定	语法	Pass@1	语法	Pass@1	语法	Pass@1	语法	Pass@1
MDLM	494M	–	67.65	6.29	69.73	6.45	71.27	6.74	71.83	6.65
MDLM	809M	–	69.76	5.95	71.99	6.61	72.73	6.62	74.46	6.34
AnCoder	809M	Null	71.04	**6.79**	75.42	7.00	78.52	7.81	79.11	8.64
AnCoder	809M	关键词	70.89	6.31	74.79	7.27	77.42	7.85	78.47	8.81
AnCoder	809M	标识符	72.08	6.60	75.54	7.10	77.78	8.30	79.25	8.95
AnCoder	809M	AnchorTree	**72.97**	6.74	**76.62**	**7.39**	**80.35**	**8.34**	**80.12**	**9.10**

然而，在两个基准测试中，所有扩散模型在语法有效性和 Pass@1 方面都不及自回归基线。尽管如此，采用 AnchorTree 的 AnCoder 相较于 809M MDLM 基线取得了最大幅度的提升，这表明锚定是弥合已有充分记载的自回归模型与扩散语言模型的性能差距的一种有前景的方法 [25, 32]。

5.2 测试时计算扩展的效果

扩散语言模型的一项关键优势是，可以通过增加去噪步数来提升生成质量。我们研究了不同基线在 HumanEval（表 2）和 MBPP（表3）上的表现，去噪步数为$T \in \{256, 512, 1024, 2048\}$。

对于在去噪过程中未实现重新掩码的 MDLM，随着 T 增加，我们观察到 494M 和 809M 模型的表现都只有小幅提升。事实上，尽管语法有效性分数不断上升，MDLM 的表现有时仍会趋于饱和，例如在 MBPP 上的 pass@1。相比之下，AnCoder（空值）随着 T的增加表现出更显著的提升，这表明去噪过程中的重新掩码率十分重要。启用锚定后（关键词、标识符、AnchorTree），提升幅度更大；在两个基准测试中，AnchorTree 在语法有效性和 pass@1 方面均取得了最佳表现，达到 $T = 2048$。总而言之，仅仅增加 T 对于高质量的扩散语言模型代码生成是不够的。相反，要有效地进行测试时扩展，就需要同时改进采样策略（例如重新掩码率）和结构引导，例如锚定。

5.3 消融研究

我们针对每种锚定策略消融了锚点监督权重 γ：在 OPC 退火数据集上训练两个训练轮次（12K 步），并在 HumanEval 上进行评测。图6展示了 Syntax（左）和 Pass@1（右）指标随锚点权重变化的结果。表格结果见表5（已移至附录 A.3）；这里我们总结主要趋势以及所选设置。根据消融研究，我们为 AnchorTree 选择 γ=0.03，为 Keyword 选择 γ=0.1，为 Identifier 锚点选择 γ=0.01。这些配置在表5中标出，并用于所有基准评测。

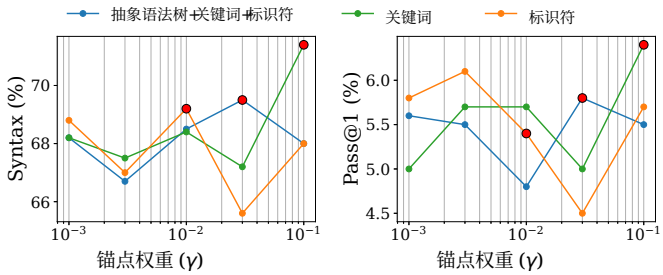

图6: **关于锚点权重的消融实验**。锚点权重为 γ on 时，语法（左）和 Pass@1（右）的变化 HumanE 三种锚定策略的验证集值。图表采用 $\gamma \in \{0.001, 0.003, 0.01, 0.03, 0.1\}$ on 对数-X-坐标轴。

6 讨论

锚定架构。 第 3.2 节所述的两阶段锚点-去噪器网络有若干设计选择，尤其是锚点网络向去噪器传递信息的方式。例如，可以让网络保留未掩码词元，也就是实际上将所有未掩码词元都视为去噪器的锚点。此外，还可以在锚点和去噪器之间实现残差连接；在这种情况下，锚点监督类似于线性探针，用于引导模型。对于哪些模型参数采用权重绑定，也存在多种设计选择，因为词嵌入和语言模型头往往规模不小（例如，AnCoder 中为 136M）。总而言之，锚定扩散语言模型的设计空间仍处于初期阶段，更多讨论请参见 Rout 等人 [36]。

锚点词元选择与加权。 虽然我们已证明关键字和标识符是代码生成的有效锚点，但其他选择也可能有效。例如，可以只锚定与控制流相关的关键字，包括 def、if 和 for 等，但不包括 True、None 和 import 等。此外，也可以只锚定特定标识符，例如函数参数中出现且长度超过 2 个字符的标识符。另外，虽然我们对 η 使用了简单的指数深度衰减方案（见公式 (11)），其他衰减计划也可能有效。

训练与优化。 众所周知，训练数据选择和优化器配置会显著影响扩散语言模型的性能 [9, 16, 22, 25, 48]。例如，可以筛选并整理质量更高的代码数据集，或在当前设置之外采用多个训练阶段。事实上，我们预计此类改动也会提升 AnCoder 的性能，尤其是在扩展至 10 亿参数规模以上时，例如使用更大的预训练自回归模型初始化扩散语言模型。不过，我们的工作侧重于隔离并评估锚定性能；为此，我们基于一套基础但足够的训练与优化流程构建了实现。

7 结论

我们提出了 AnchorTree，这是首个专为代码设计的层次化软锚定框架。AnchorTree 根据抽象语法树层次锚定关键词和标识符，既能捕捉句法结构和语义层次结构，又能实现高效的由粗到细的扩散去噪。我们通过一系列称为 AnCoder 的模型验证了该框架；实验结果表明，这些模型在标准代码基准测试中的表现优于扩散语言模型基线。我们的发现表明，利用代码固有的结构先验（如抽象语法树），是通过离散扩散语言模型实现参数高效、高质量代码生成的有效方法。

致谢

本研究得到美国国家科学基金会（NSF）项目 2019844 和 2112471、德克萨斯大学奥斯汀分校机器学习实验室的支持，并获得德克萨斯大学奥斯汀分校生成式人工智能中心（CGAI）和德克萨斯高级计算中心（TACC）提供的 Vista GPU 集群计算资源支持。

参考文献

[1] Altaf Allah Abbassi、Leuson Da Silva、Amin Nikanjam 和 Foutse Khomh。揭示大语言模型生成代码中的低效问题：迈向全面的分类体系。*arXiv* 预印本 *arXiv:2503.06327*，2025年。[2] Wasi Ahmad、Saikat Chakraborty、Baishakhi Ray 和 Kai-Wei Chang。面向程序理解与生成的统一预训练。载于计算语言学协会北美分会 *2021* 年会议论文集：人类语言技术，第 2655–2668 页，2021年。URL https://aclanthology.org/2021.naacl-main.211.pdf。[3] V Aho Alfred、S Lam Monica 和 D Ullman Jeffrey。编译器原理、技术与工具。培生教育，2007年。[4] AI Anthropic。Claude 3 模型系列：Opus、Sonnet、Haiku。*Claude-3* 模型卡，1:1，2024年。URL https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf。[5] Jacob Austin、Daniel D Johnson、Jonathan Ho、Daniel Tarlow 和 Rianne Van Den Berg。离散状态空间中的结构化去噪扩散模型。神经信息处理系统进展，34:17981–17993，2021年。[6] Jacob Austin、Augustus Odena、Maxwell Nye、Maarten Bosma、Henryk Michalewski、David Dohan、Ellen Jiang、Carrie Cai、Michael Terry、Quoc Le 等。利用大语言模型进行程序合成。*arXiv* 预印本 *arXiv:2108.07732*，2021年。[7] Mohammad Bavarian、Heewoo Jun、Nikolas Tezak、John Schulman、Christine McLeavey、Jerry Tworek 和 Mark Chen。高效训练语言模型以实现中间填充。*arXiv* 预印本 *arXiv:2207.14255*，2022年。URL https://arxiv.org/pdf/2207.14255。[8] Casey Casalnuovo、Kenji Sagae 和 Prem Devanbu。研究自然语言语料库与编程语言语料库之间的差异。经验软件工程，24(4):1823–1868，2019年。[9] Haolin Chen、Shiyu Wang、Can Qin、Bo Pang、Zuxin Liu、Jielin Qiu、Jianguo Zhang、Yingbo Zhou、Zeyuan Chen、Ran Xu 等。CoDA：通过扩散适配实现代码语言模型。*arXiv* 预印本 *arXiv:2510.03270*，2025年。[10] Mark Chen。评估基于代码训练的大语言模型。*arXiv* 预印本 *arXiv:2107.03374*，2021年。[11] Gheorghe Comanici、Eric Bieber、Mike Schaekermann、Ice Pasupat、Noveen Sachdeva、Inderjit Dhillon、Marcel Blistein、Ori Ram、Dan Zhang、Evan Rosen 等。Gemini 2.5：凭借高级推理、多模态、长上下文和新一代智能体能力，推动前沿发展。*arXiv* 预印本 *arXiv:2507.06261*，2025年。[12] Keith D Cooper 和 Linda Torczon。编译器工程。Morgan Kaufmann，2022年。[13] DeepMind。Gemini Diffusion。技术报告，DeepMind，2025年。URL https://deepmind.google/models/gemini-diffusion/。访问日期：2026-01-24。[14] Yihong Dong、Xue Jiang、Jiaru Qian、Tian Wang、Kechi Zhang、金芝和李戈。基于大语言模型的智能体代码生成综述。*arXiv* 预印本 *arXiv:2508.00083*，2025年。[15] Daniel Fried、Armen Aghajanyan、Jessy Lin、Sida Wang、Eric Wallace、Freda Shi、Ruiqi Zhong、Scott Yih、Luke Zettlemoyer 和 Mike Lewis。Incoder：用于代码填充与合成的生成模型。载于第十一届学习表征国际会议，2023年。URL https://openreview.net/forum?id=hQwb-lbM6EL。[16] Shansan Gong、Ruixiang Zhang、Huangjie Zheng、Jiatao Gu、Navdeep Jaitly、Lingpeng Kong 和 Yizhe Zhang。DiffuCoder：理解并改进用于代码生成的掩码扩散模型。*arXiv* 预印本 *arXiv:2506.20639*，2025年。[17] Daya Guo、Dejian Yang、Haowei Zhang、Junxiao Song、Ruoyu Zhang、Runxin Xu、Qihao Zhu、Shirong Ma、Peiyi Wang、Xiao Bi 等。DeepSeek-R1：通过强化学习激发大语言模型的推理能力。*arXiv* 预印本 *arXiv:2501.12948*，2025年。URL https://arxiv.org/pdf/ 2501.12948。

[18] Ava Heinonen、Bettina Lehtelä、Arto Hellas 和 Fabian Fagerholm。《综合关于程序员对程序、任务与概念的心智模型的研究——系统文献综述》。*Information and Software Technology*，164:107300，2023年。[19] Abram Hindle、Earl T Barr、Mark Gabel、Zhendong Su 和 Premkumar Devanbu。《软件的自然性》。*ACM*通讯，59(5):122–131，2016年。[20] Siming Huang、Tianhao Cheng、Jason Klein Liu、Weidi Xu、Jiaran Hao、Liuyihan Song、Yang Xu、Jian Yang、Jiaheng Liu、Chenchen Zhang 等。《OpenCoder：顶尖代码大语言模型的开放食谱》。载于计算语言学协会第*63*届年会论文集（第一卷：长篇论文），第33167–33193页，2025年。[21] Binyuan Hui、Jian Yang、Zeyu Cui、Jiaxi Yang、Dayiheng Liu、Lei Zhang、Tianyu Liu、Jiajun Zhang、Bowen Yu、Keming Lu 等。《Qwen2.5-Coder技术报告》。*arXiv*预印本 *arXiv:2409.12186*，2024年。[22] Samar Khanna、Siddhant Kharbanda、Shufan Li、Harshit Varma、Eric Wang、Sawyer Birnbaum、Ziyang Luo、Yanis Miraoui、Akash Palrecha、Stefano Ermon、Aditya Grover 和 Volodymyr Kuleshov。《Mercury：基于扩散的超快速语言模型》。*arXiv*预印本 *arXiv:2506.17298*，2025年。[23] Man Ho Lam、Chaozheng Wang、Jen-tse Huang 和 Michael R Lyu。《Codecrash：在结构与语义扰动下对大语言模型推理进行压力测试》。*arXiv*预印本 *arXiv:2504.14119*，2025年。[24] Chris Lattner 和 Vikram Adve。《LLVM：用于终身程序分析与转换的编译框架》。载于国际代码生成与优化研讨会，*2004*年。*CGO 2004*，第75–86页。IEEE，2004年。[25] Chengze Li、Yitong Zhang、Jia Li、Liyi Cai 和李戈。《超越自回归：扩散大语言模型用于代码生成的实证研究》。*arXiv*预印本 *arXiv:2509.11252*，2025年。[26] Xiang Li、John Thickstun、Ishaan Gulrajani、Percy S Liang 和 Tatsunori B Hashimoto。《Diffusion-LM改进了可控文本生成》。神经信息处理系统进展，35:4328–4343，2022年。[27] Yujia Li、David Choi、Junyoung Chung、Nate Kushman、Julian Schrittwieser、Rémi Leblond、Tom Eccles、James Keeling、Felix Gimeno、Agustin Dal Lago 等。《使用AlphaCode实现竞赛级代码生成》。《科学》，378(6624):1092–1097，2022年。URL https://www.science.org/doi/epdf/10.1126/science.abq1158。[28] Ilya Loshchilov 和 Frank Hutter。《解耦权重衰减正则化》。*arXiv*预印本 *arXiv:1711.05101*，2017年。[29] Aaron Lou、Chenlin Meng 和 Stefano Ermon。《通过估计数据分布的比率进行离散扩散建模》。*arXiv*预印本 *arXiv:2310.16834*，2023年。[30] Andreas C Müller 和 Sven Behnke。《Pystruct：使用Python进行结构化预测学习》。*J.Mach. Learn. Res.*，15(1):2055–2060，2014年。URL https://www.jmlr.org/papers/volume15/ mueller14a/mueller14a.pdf。[31] Niels Mündler、Jasper Dekoninck 和 Martin Vechev。《使用上下文无关文法对扩散大语言模型进行约束解码》。*arXiv*预印本 *arXiv:2508.10111*，2025年。URL https://arxiv.org/pdf/2508.10111。[32] Shen Nie、Fengqi Zhu、Zebin You、Xiaolu Zhang、Jingyang Ou、Jun Hu、Jun Zhou、Yankai Lin、Ji- Rong Wen 和 Chongxuan Li。《大语言扩散模型》。*arXiv*预印本 *arXiv:2502.09992*，2025年。[33] Theo X Olausson、Jeevana Priya Inala、Chenglong Wang、Jianfeng Gao 和 Armando Solar-Lezama。《自我修复是代码生成的灵丹妙药吗？》。*arXiv*预印本 *arXiv:2306.09896*，2023年。[34]OpenAI。《OpenAI o3和o4-mini系统卡》。系统卡，OpenAI，2025年。URL https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf。访问日期：2026-01-24。

[35] Jingyang Ou、Shen Nie、Kaiwen Xue、Fengqi Zhu、Jiacheng Sun、Zhenguo Li 和 Chongxuan Li。《你的吸收式离散扩散模型实际上在建模干净数据的条件分布》。*arXiv*预印本 *arXiv:2406.03736*，2024年。[36] Litu Rout、Constantine Caramanis 和 Sanjay Shakkottai。《锚定扩散语言模型》。载于第三十九届神经信息处理系统年会，2025年。URL https://openreview.net/forum?id=E8adS5srds。[37] Litu Rout、Andreas Lugmayr、Yasamin Jafarian、Srivatsan Varadharajan、Constantine Caramanis、Sanjay Shakkottai 和 Ira Kemelmacher-Shlizerman。《离散扩散后验采样的测试时锚定》。*arXiv* 预印本 *arXiv:2510.02291*，2025年。URL https://arxiv.org/pdf/2510. 02291。[38] Subham Sahoo、Marianne Arriola、Yair Schiff、Aaron Gokaslan、Edgar Marroquin、Justin Chiu、Alexander Rush 和 Volodymyr Kuleshov。《简单而有效的掩码扩散语言模型》。神经信息处理系统进展，37:130136–130184，2024年。[39] Alfred Santa Molison、Marcia Moraes、Glaucia Melo、Fabio Santos 和 Wesley KG Assuncao。《大语言模型生成的代码是否比人工编写的代码更易维护、更可靠？》。载于*2025*年*ACM/IEEE*国际实证软件工程与度量研讨会（*ESEM*），第151–162页。IEEE，2025年。[40] Michael Scott。《编程语言实践》。Morgan Kaufmann，2000年。[41] Amir Molzam Sharifloo、Maedeh Heydari、Parsa Kazerooni、Daniel Maninger 和 Mira Mezini。《大语言模型仍在哪些方面面临挑战？代码生成基准测试的深入分析》。载于*2025*年第二届*IEEE/ACM*人工智能赋能软件国际会议（*AIware*），第249–253页。IEEE，2025年。[42] Jiaxin Shi、Kehang Han、Zhe Wang、Arnaud Doucet 和 Michalis Titsias。《面向离散数据的简化通用掩码扩散》。神经信息处理系统进展，37:103131–103167，2024年。URL https://arxiv.org/pdf/2406.04329。[43] Jiaxin Shi、Kehang Han、Zhe Wang、Arnaud Doucet 和 Michalis Titsias。《面向离散数据的简化通用掩码扩散》。神经信息处理系统进展，37:103131–103167，2024年。[44] Mohammed Latif Siddiq、Lindsay Roney、Jiahao Zhang 和 Joanna Cecilia Da Silva Santos。《ChatGPT生成代码的质量评估及开发者对其的使用》。载于第*21*届国际软件仓库挖掘会议论文集，第152–156页，2024年。[45] Jascha Sohl-Dickstein、Eric Weiss、Niru Maheswaranathan 和 Surya Ganguli。《使用非平衡热力学进行深度无监督学习》。载于国际机器学习会议，第2256–2265页。PMLR，2015年。URL https://proceedings.mlr.press/v37/ sohl-dickstein15.pdf。[46] Tarun Suresh、Debangshu Banerjee、Shubham Ugare、Sasa Misailovic 和 Gagandeep Singh。《Dingo：扩散大语言模型的约束推断》。*arXiv*预印本 *arXiv:2505.23061*，2025年。[47] Guanghan Wang、Yair Schiff、Subham Sekhar Sahoo 和 Volodymyr Kuleshov。《通过推理时扩展对离散扩散模型进行重新掩码》。载于第三十九届神经信息处理系统年会，2025年。URL https://openreview.net/forum?id=IJryQAOy0p。[48] Zhihui Xie、Jiacheng Ye、Lin Zheng、Jiahui Gao、Jingwei Dong、Zirui Wu、Xueliang Zhao、Shansan Gong、Xin Jiang、Zhenguo Li 等。《Dream-Coder 7B：面向代码的开放扩散语言模型》。*arXiv*预印本 *arXiv:2509.01142*，2025年。[49] An Yang、Anfeng Li、Baosong Yang、Beichen Zhang、Binyuan Hui、Bo Zheng、Bowen Yu、Chang Gao、Chengen Huang、Chenxu Lv 等。《Qwen3技术报告》。*arXiv*预印本 *arXiv:2505.09388*，2025年。URL https://arxiv.org/pdf/2505.09388。[50] Yiming Zeng、Jinghan Cao、Zexin Li、Yiming Chen、Tao Ren、Dawei Xiang、Xidong Wu、Shangqian Gao 和 Tingting Yu。《TreeDiff：使用扩散大语言模型进行AST引导的代码生成》。*arXiv*预印本 *arXiv:2508.01473*，2025年。[51] Leo Zhang 和 Saifuddin Syed。《余弦调度对于掩码离散扩散模型而言是Fisher–Rao最优的》。*arXiv*预印本 *arXiv:2508.04884*，2025年。

A 附加实验

本附录提供支持论文主要结果的补充材料。我们补充了训练数据（附录 A.1）、基准测试说明（附录 A.2）、超参数消融（附录 A.3）以及生成代码的定性分析（附录 A.4）的更多细节。

A.1 训练与微调数据集

A.1.1 阶段 1：预训练

阶段 1 预训练用于将自回归代码模型适配到扩散语言建模范式。在此阶段，我们使用 OpenCoder 的 opc-fineweb-codecorpus [20], 进行训练，该语料库包含从网络规模编程数据中收集的 1.01 亿个代码示例，每个示例平均包含 490 个词元（总计约 494.9 亿个词元）。

该语料库广泛涵盖了现实世界中的编程模式、库和编码风格，非常适合用于初始化基于扩散的代码生成。在此阶段，模型学习对掩码代码序列进行去噪，同时保留从自回归预训练中继承的通用代码建模能力。我们训练一个周期（19.7 万个优化步数），发现这足以实现稳定的 AR到DLM适配。

A.1.2 阶段：中期训练

阶段 2 的中期训练旨在衔接大规模代码预训练与面向特定任务的监督微调，让模型接触结构化、合成及网络规模数据的多样化混合数据。对于基于扩散的模型而言，这一阶段尤为重要，因为它能促使模型在针对特定任务进行专门化之前，针对各种程序结构和语义模式形成稳健的去噪能力。

如表4所示，中期训练语料包含 2949 万个样本，取自 OpenCoder 退火数据集，总计约 103.3 亿个词元。数据划分为 99% 的训练集和 1% 的验证集，并在训练期间均匀混合。各组成部分各有不同用途:

算法代码。 此子集包含控制流和数据依赖定义明确的算法编程问题。它侧重于精确推理、递归和结构化程序逻辑，这些对于学习长程依赖和层次化程序结构至关重要。

合成代码片段。 合成代码子集由自动生成的代码片段组成，涵盖了广泛的编程构造。这些示例增加了结构多样性，并使模型接触到各种句法模式，有助于避免对狭窄的代码分布过拟合。

合成问答。 合成问答数据将自然语言问题与代码相关答案配对。这个子集增强了文本规格说明与代码生成之间的对齐，对于 HumanEval 和 MBPP 等基准测试风格的提示尤其重要。

FineWeb Math。 数学语料库提供结构化的数学推理问题，促使模型处理符号表达式、算术逻辑和逐步计算。这些能力与代码生成互为补充，并有助于对结构化序列进行精确去噪。

FineWeb 代码（采样）。 采样的 FineWeb 代码数据引入了真实世界、网络规模的编程内容，涵盖了纯合成数据或算法数据集中代表性不足的实用编码风格和惯用表达。

总体而言，这种混合数据旨在平衡结构严谨性、语义多样性和现实世界的多变性。通过在这一异构语料库上进行中期训练，AnCoder 学会了对广泛的程序结构进行去噪，为后续的监督微调阶段奠定了坚实基础。我们训练三个训练轮次，总计约 171K 个优化步数。

A.1.3 阶段 3：监督微调

阶段3监督微调（SFT）使扩散模型专门适用于下游 Python 代码生成任务。我们使用 opc-annealing-corpus 的 Python 子集，其中包括算法代码、合成代码片段和合成问答数据。该子集包含 335 万个样本，总计约 11 亿个词元。

表4：AnCoder 阶段 2 中期训练期间使用的数据集。我们报告了 OpenCoder 退火语料库中各组成部分的样本数量及每个样本的平均词元数（截断至 1024 个词元）。

数据集	样本	词元	领域
算法代码	5.32M	1.01B	算法程序
合成代码片段	3.08M	1.21B	生成代码
合成问答	10.60M	3.74B	代码相关问答
FineWeb Math	5.24M	2.44B	数学推理
FineWeb 代码（采样）	5.24M	1.93B	网络规模代码
总计	**29.49M**	**10.33B**	–

SFT 样本采用指令式提示，并配以目标代码解答，以使训练与 HumanEval 和 MBPP 等基准测试所使用的评估格式保持一致。我们使用 95%-5% 的训练-验证集划分进行 8 个训练轮次，约进行 5 万步优化。该阶段有助于自然语言规范与可执行代码之间实现精确对齐，补充预训练和中期训练阶段更侧重结构的目标。

A.2 评估基准测试

我们在两个标准代码生成基准测试上评估 AnCoder，这些基准测试既评估生成程序的语法正确性，也评估其功能行为。下面，我们简要介绍这些基准测试、评估协议以及数据集的使用情况。

HumanEval。 HumanEval [10] 是一个包含 164 道 Python 编程题的基准测试，旨在评估代码根据自然语言规范生成后的功能正确性。每道题都包含一个函数签名及描述预期行为的文档字符串，以及一组用于评测的隐藏单元测试。如果生成的解决方案通过所有单元测试，且未引发运行时错误或语法错误，则视为正确。HumanEval 强调算法推理、控制流以及对边界情况的正确处理。

MBPP。 MBPP（多数为基础 Python 程序） [6] 包含 974 道 Python 编程题，涵盖基本的算法与过程式概念。每道题都提供简短的自然语言描述和多个单元测试。与 HumanEval 相比，MBPP 更侧重于简单程序，但在问题类型上具有更大的多样性，因此是评估鲁棒性和泛化能力的有用补充基准测试。

评估指标。 我们报告两个主要指标：语法有效性，定义为能够通过 Python ast 模块的 ast.parse 成功解析且不含语法错误的生成程序所占的比例；以及*Pass@1*，定义为单个生成样本通过所有给定单元测试的概率。在 Pass@1 指标下，语法错误被视为失败，因为此类程序无法成功执行。

对于每个基准测试，我们针对每道题生成多个样本，并按照标准做法估算 Pass@1，即通过单元测试的生成结果的平均数量。除非另有说明，我们针对 HumanEval 的每道题生成 20 个样本，针对 MBPP 的每道题生成 10 个样本。

许可与使用。 本研究使用的所有数据集均公开可用，并获准用于研究。HumanEval 采用 MIT 许可证发布。MBPP 采用 Apache 2.0 许可证发布。我们仅根据各自的许可证，将这两个数据集用于非商业研究和评估。

A.3 锚点权重的消融实验

我们针对每种锚定策略，对锚点监督权重 γ 进行消融实验：在 OPC 退火数据集上训练 AnCoder 两个训练轮次（12K 步），并在 HumanEval 上进行评估。表5报告了所有测试的 γ 值对应的语法有效性和 Pass@1，而论文正文中的图6展示了相同的趋势。

对于 AnchorTree 策略，性能呈现出明显的单峰趋势。当 γ 从 0.001 增至 0.03 时，语法有效性和 Pass@1 均有所提升，之后性能略有下降。这

表5：在 OPC 退火数据集上训练两个训练轮次（12K 步数）后，HumanEval 上的锚点权重 γ 消融实验。我们报告语法有效性和通过率。每种锚点策略所选的配置以高亮显示。

锚点策略	锚点权重（γ）	语法（%）	Pass@1（%）
AnchorTree	0.001	68.2	5.6
	0.003	66.7	5.5
	0.01	68.5	4.8
	0.03	**69.5**	**5.8**
	0.1	68.0	5.5
关键词	0.001	68.2	5.0
	0.003	67.5	5.7
	0.01	68.4	5.7
	0.03	67.2	5.0
	0.1	**71.4**	**6.4**
标识符	0.001	68.8	5.8
	0.003	67.0	**6.1**
	0.01	**69.2**	5.4
	0.03	65.6	4.5
	0.1	68.0	5.7

表明，适度的锚点监督足以引导早期去噪，而过强的监督则可能对中间锚点预测造成过度正则化。

对于关键词锚定，较大的 γ 更有利。我们观察到，在 $\gamma = 0.1$ 时语法有效性和 Pass@1 均达到最高，这表明当锚点定义在词元级时，需要更强的监督才能持续影响去噪过程。

对于标识符锚定，两项指标在不同的 γ 取值下达到峰值。语法有效性在 $\gamma = 0.01$ 时最高，而 Pass@1 则在 $\gamma = 0.003$ 时达到最高。这凸显了强化标识符结构与保留下游功能正确性所需灵活性之间的权衡。

总体而言，这些结果证实，必须谨慎校准锚点监督。γ 的值过小会导致锚点信号利用不足，而值过大则可能妨碍后续去噪步数。基于这项消融实验，我们为 AnchorTree 选择 $\gamma = 0.03$，为关键词选择 $\gamma = 0.1$，为标识符锚定策略选择 $\gamma = 0.01$，并在所有后续实验中使用这些设置。

A.4 生成代码的定性分析

我们对标准扩散模型和 AnCoder 变体生成的代码进行定性分析，以更好地理解层次锚定对程序结构的影响。本分析中考察的所有模型参数量均相同（809M），并在相同的推理设置下进行评估。除非另有说明，所有示例均使用 $T = 2048$ 个去噪步数生成。

Task

函数： has_close_elements

```
定义 has_close_elements(numbers:List[浮点数],阈值: 浮点数)->
    布尔值:
    """ 检查给定的数字列表中，是否有任意两个数字之间的距离小于给定阈值。>>>
    has_close_elements([1.0, 2.0, 3.0], 0.5)

    错误
    >>>有_相近的_元素（[1.0，2.8，3.0，4.0，5.0，2.0]，0.3）True """
```

A.4.1 HumanEval 定性示例

我们考察 HumanEval 基准测试中的一个典型示例，以说明不同的锚定策略如何影响生成代码的结构和逻辑属性。任务如下：给定一个数字列表和一个阈值，判断列表中是否存在任意两个数字，其差值小于给定阈值。

MDLM 生成的代码语法有效，但未能实现预期逻辑。生成的代码引入了不必要的控制流，且数值推理不正确，尽管成功解析，行为在逻辑上仍无效。

生成代码

```
def has_close_elements ( numbers , 阈值 ): from collections import Counter epsilon = 浮点数 ( '无穷大 ') counter =Counter ( numbers ) while epsilon > 0: epsilon = 0. 0001 for num , count in counter . items () : if num +epsilon in counter and abs ( num - epsilon ) <阈值 : return True epsilon -= 1return False
```

AnCoder（空值）。采用两阶段架构但没有锚点监督时，AnCoder 生成的代码更简洁，但仍未能把握核心算法结构。该实现将基于索引的推理与阈值比较混杂在一起，与问题规格不符。

生成代码

```
定义 has_close_elements ( numbers , 阈值 ):对于 i 在 enumerate ( sorted ( numbers ) , 1) :如果 abs(i - numbers [i - 1]) ==abs(i - 阈值 ):返回 True返回 False
```

AnCoder（关键词）。关键词锚定鼓励模型引入恰当的控制流结构，但生成的逻辑仍然过于粗略。模型将问题简化为全局极值比较，因此无法检测彼此接近的相邻值。

生成代码

```
定义 有_接近_元素（数字，阈值）： 最小_数字 =最小（数字）最大_数字 =最大（数字）
返回 最大_数字 - 最小_数字 <= 阈值 或 最小_数字 - 最大_数字 <=阈值
```

AnCoder（标识符）。标识符锚定提升了变量使用一致性，但仍难以恢复元素之间正确的关系结构。该模型引入了紧凑表达式，但这些表达式并未忠实地实现所需的成对比较。

生成代码

```
def 有_接近_元素（数字，阈值）：返回最小值(num + 1-numbers.index(num)对于 num
在 numbers） <=阈值
```

AnCoder (AnchorTree)。 AnchorTree 生成了正确且简洁的实现。模型通过对输入进行排序并比较相邻元素，恢复出预期的算法结构，与人工编写的代码十分接近。

生成代码

```
定义 有_接近_元素(数字,阈值):数字 =排序(数字)对于 i 在范围内(长度(
数字)-1):如果绝对值(数字[i]-数字[i + 1]) <阈值:返回 真返回 假
```

总结。 这些示例表明，尽管标准扩散和弱锚定策略能够生成语法有效的代码，却往往无法恢复正确执行所需的层次结构。相比之下，AnchorTree 使 AnCoder 能够在去噪过程的早期识别并显现结构上重要的词元。因此，AnCoder 能够生成控制流和变量依赖关系正确的代码，且通常与标准的人工编写解决方案相匹配。

A.4.2 MBPP 定性示例

我们分析 MBPP基准测试中的一个典型示例，以进一步评估锚定如何影响结构化代码生成。任务如下：给定一个成本矩阵和两个位置，编写一个函数来确定这两个位置之间的最小成本路径。

Task

函数： 最小_成本

编写一个函数，找出从 (m, n) 到达自(0, 0)的 最小成本路径，给定成本矩阵 cost[][]和 cost 矩阵中的位置 (m, n)，位于 cost[][]。

```
断言 最小_成本([[1, 2, 3], [4, 8, 2], [1, 5, 3]], 2, 2) == 8断言 最
小_成本([[2, 3, 4], [5, 9, 3], [2, 6, 4]], 2, 2) == 12断言 最小_成
本([[3, 4, 5], [6, 10, 4], [3, 7, 5]], 2, 2) == 16‘‘‘python
```

MDLM 能够生成语法有效代码，但无法恢复连贯的算法结构。生成的实现包含不完整的控制流和未定义的状态转移，导致行为不正确。

生成代码

```
定义 最小_代价（代价，m，n）：q =列表
([[0, (), 0)])位置 = [(0, 0)]对于 (x, 位
置)，代价 在 q[-1]：如果 位置 == (m,
n)：返回 代价 q.追加((x +代价，位置))
```

AnCoder（空值）。在没有锚点监督的情况下，AnCoder 引入了一种基于优先队列的合理高层策略，但其实现存在索引不一致和终止条件错误的问题。

生成代码

```
import heapq def 最小_成本 ( 成本 , m , n): q = []已访问 = [[False] * m for _
in range (n)]heapq . heappush (q, ( 成本 [0][0] , 0 , 0) ) while q: c , x , y =
heapq . heappop (q) if x == m and y ==n: return c for dx , dy in [(1 ,0) , ( -1 ,
0) , (0 ,1) , (0 , -1)]: nx , ny =x +dx , y +dy if 0 <= nx < m and 0 <=
ny <n: heapq . heappush (q, (c +成本 [nx ][ny ] , nx , ny ))
```

AnCoder（关键词）。关键词锚定鼓励使用显式控制流结构，但模型仍无法正确实现状态转移和停止准则。

生成代码

```
from collections import deque
def min_path(cost, m, n):
    q = deque([(0, 0)])
    visited = set()
    while q:
        x, y = q.popleft()
        if (x, y) == (m, n):
            return 1
        for dx, dy in [(-1,0), (0,1), (1,0), (0,-1)]:
            q.append((x + dx, y + dy))
    return 0
```

AnCoder （标识符)。标识符锚定提高了变量一致性和复用性，但生成
d 代码仍然无法恢复正确的递推关系，并引入了不必要的复杂性。

生成代码

```
def find_min_cost ( cost , m , n): n =len ( cost ) costs = [[float ( ' inf ')] * n for
_ in range (n)]def dfs (i , j): if costs [i ][j]!=float ( 'inf '): return costs [i][j]c
osts [i ][j] =cost [i ][j] +min ( dfs (i -1 , j) , dfs (i ,j -1) ) return costs [i][j]
return dfs (m , n)
```

AnCoder（AnchorTree）。AnchorTree 使 AnCoder 能够生成正确且简洁的解决方案，具有清晰的递归结构和适当的基本情况。有趣的是，生成代码与用于寻找两个位置之间最小成本路径的标准动态规划表述高度吻合。

生成代码

```
def 最小_成本(成本 , m, n):
    如果m < 0 或n < 0:
        返回 浮点数('无穷大')
    如果m == 0 且n == 0:
        返回 代价[0][0]
    返回 代价[m][n] +最小值(
        最小值_代价(代价, m-1 , n),
        最小值_代价(代价, m, n-1) ,
        min_cost(成本 , m-1 , n-1) ,
    )
```

总结。这些示例反映了在 HumanEval 上观察到的趋势。标准掩码扩散模型和弱锚定策略往往生成语法有效但结构不正确的代码，而 AnchorTree 使 AnCoder 能够恢复层次化算法结构，从而得到正确且可解释的解决方案。

A.5 局限性

尽管 AnchorTree 能够持续提升扩散语言模型的语法有效性和功能正确性，但仍存在若干局限。首先，锚定扩散模型的发展尚处于起步阶段，在架构设计方面仍有很大的探索空间，以便更好地利用 AnchorTree 框架。此外，AnchorTree 专门针对从抽象语法树中提取的代码层次结构，这可能会限制其对动态执行行为进行推理的能力。值得庆幸的是，许多用于分析此类行为的经典算法都依赖于层次化数据结构的构建。因此，我们预计 AnchorTree 也能够扩展到这些场景中。