面向代码生成的大语言模型规划

Shun Zhang、Zhenfang Chen、Yikang ShenMIT-IBM Watson 人工智能实验室

Mingyu Ding香港大学

Joshua B. TenenbaumMIT 脑与认知科学系、分子与机器学习中心、计算机科学与人工智能实验室

Chuang Gan
马萨诸塞大学阿默斯特分校，MIT-IBM Watson 人工智能实验室

arXiv:2303.05510v1 [cs.LG] 9 Mar 2023

摘要

现有基于大语言模型的代码生成流程通常在解码过程中使用束搜索或采样算法。尽管生成的程序能够获得较高的基于令牌匹配的分数，但它们往往无法编译或产生错误的输出。主要原因在于，传统的 Transformer 解码算法可能并非代码生成的最佳选择。在这项工作中，我们提出了一种新颖的 Transformer 解码算法——规划引导的 Transformer 解码（PG-TD），该算法使用规划算法进行前瞻搜索，引导 Transformer 生成更优质的程序。具体来说，Transformer 不再只是优化生成序列的似然，而是借助规划器生成候选程序，并在公开测试用例上对其进行测试。因此，Transformer 能够做出更明智的决策，并生成最终有助于提升程序质量的词元。我们还设计了一种机制，在 Transformer 与规划器之间共享信息，从而提高算法的计算效率。我们在多个公开编程挑战基准上，使用若干大语言模型作为骨干模型对框架进行了实证评估。结果表明：1）与有竞争力的基线方法相比，该框架能够持续生成性能更高的程序；2）通过优化修改后的目标函数，该框架能够实现可控代码生成，例如生成简洁代码和注释丰富的代码[1]。

1 引言

Transformer 等大语言模型（Vaswani等人，2017a）已在自然语言处理、计算机视觉及其他诸多领域取得成功。得益于 Transformer 在序列建模方面的强大能力，人们将其用于代码生成（Wang 等人，2021年；Ahmad 等人，2021年），将程序视为文本序列进行处理。Transformer 在代码翻译（Roziere 等人，2022年）、代码补全（Chen 等人，2021a）和解决编程挑战题（Hendrycks 等人，2021年）等基准测试任务上取得了显著进展。近期，AlphaCode（Li 等人，2022年）借助在大型编程语料库上预训练的大型 Transformer 模型，甚至在编程竞赛中取得了具有竞争力的成绩。

像 AlphaCode 这样的基于 Transformer 的流程遵循自然语言处理的传统，并在Fan等人、2018年；Dabre 和 Fujita、2020年）的代过程中使用采样方法。具体而言，它们使用预训练代码生成 Transformer 采样大量完整程序，利用数据集提供的公开测试用例对这些程序进行评估，并输出通过测试用例数量最多的程序。与基于束搜索的方法相比，这些先采样再筛选的算法（我们称之为采样 + 筛选）能够利用测试用例，确实提升生成程序的质量。然而，在 Transformer 生成过程中，它们完全不考虑测试用例，而是在生成所有候选程序后才用测试用例评估程序。这会导致它们的算法采样效率低。与自然语言不同，程序可能会因一个生成错误的词元而

[1]项目主页：https://codeaimcts.github.io

彻底失效。因此，这些算法需要穷举采样大量程序，才能找到正确的解。

这些算法存在样本效率问题，主要是因为 Transformer 束搜索算法和采样算法（Vaswani等人，2017b）可能并非代码生成的最佳选择。理想的代码生成算法应当在生成过程中，一旦得知当前生成的程序必然会失败，就立即停止；同时，应引导生成过程，使其倾向于生成能够通过更多测试用例的成功程序。为实现这一目标，我们尝试在 Transformer 生成过程中应用规划算法。由于规划算法可以将生成程序的通过率作为优化目标，因此我们用它来评估生成代码的质量，并帮助 Transformer 模型做出更明智的决策。

本文探讨以下研究问题：能否将规划算法与预训练代码生成 *Transformer* 相结合，从而得到一种比传统 *Transformer* 生成算法和文献中广泛采用的采样 + 筛选方案生成更优程序的算法？为回答这一问题，我们提出了一种新算法：规划引导的 *Transformer* 解码（PG-TD）。在代码生成过程中，规划器执行前瞻搜索，找出能够生成高质量代码的词元。由于代码搜索空间庞大，规划器单独使用时可能无法高效地找到高质量代码；此时，预训练代码生成 Transformer 便发挥了作用。具体而言，规划器内部使用 Transformer 束搜索算法和下一个词元的概率来提供有用的启发式信息。我们发现，直接将规划器与 Transformer 相结合在计算上可能效率低下。因此，我们设计了一些机制，使 Transformer 与规划器能够共享信息，从而提高整体算法的效率。

我们强调，我们的算法是模型无关的，也就是说，任何标准的代码生成 Trans-former 模型都可以用作骨干 Transformer。重要的是，我们的算法无需获取更多示例解答，也无需微调 Transformer 模型来提升其性能。通过实证，我们发现，与其他基线方法相比，我们提出的算法在多种公认的指标下生成的程序质量更高。此外，我们还通过实证展示了该算法具有以下优势。1）通过改变规划器的奖励函数，我们的算法具有很强的通用性，能够优化不同的目标函数，而无需微调 Transformer 模型。2）我们的算法能够生成用于微调代码生成 Transformer 模型的解答，从而提升 Transformer 的性能。更具体地说，本文的贡献如下。

- 首先，我们提出了一种新颖的算法——规划引导的 Transformer 解码（PG-TD），它利用规划算法进行前瞻搜索，并引导 Transformer 生成更好的代码。我们的算法是模型无关的，能够与任何标准 Transformer 模型配合使用，而且不需要了解生成程序的语法。
- 其次，将规划算法直接整合到 Transformer 解码过程中，可能导致 Transformer 束搜索算法被冗余地使用。我们致力于设计能够显著提升该算法计算效率的机制。
- 第三，我们在竞赛编程基准测试上评估了该算法，并通过实验证明，与基线方法相比，该算法能够持续生成通过率和其他指标更优的程序。我们还表明，该算法用途广泛，能够针对可控代码生成优化通过率以外的目标，例如生成更简洁的代码和包含更多注释的代码。

2 相关工作

用于程序合成的 Transformer。我们的工作基于用于程序合成的 Transformer（Roziere 等人，2020年；Austin 等人，2021年）。受其在一系列自然语言任务中展现出的能力启发，现代基于 Transformer 的语言模型（Devlin 等人，2019年；Radford 等人，2019年；Raffel 等人，2020年）通过将编程语言视同自然语言，已被用于程序合成。研究人员开发了一系列基于 BERT 的 Transformer，用于处理代码语法（Kanade 等人，2020年；Feng 等人，2020年；Devlin 等人，2019年；Guo 等人，2020年）。随后，CodeX（Chen 等人，2021a）和 CodeT5（Wang 等人，2021年）分别采用 GPT2（Radford 等人，2019年）和 T5（Raffel 等人，2020年）作为代码理解与生成的基础模型。研究者探索了多种学习方法，包括从示例中学习（Ellis 等人，2021年）和神经符号方法（Nye 等人，2020年）。近期，AlphaCode（Li 等人，2022）结合了大规模

在海量程序数据上预训练的 Transformer 模型，并进行大规模采样，在编程竞赛中展现出具有竞争力的表现。所有这些工作主要专注于训练更强大的代码生成模型，在 Transformer 的生成过程中仍使用束搜索（Graves，2012）或采样（Fan等人，2018年）。

用于程序合成的测试用例。我们的工作也与使用单元测试（Myers，1979年）进行程序合成有关。Tufano等人（2020a；b）提出使用 Transformer 模型（Vaswani等人，2017a）生成测试用例及相应的准确断言语句。近期，Roziere 等人（2022年）利用自动生成的单元测试构建用于无监督代码翻译的并行训练数据。Chen 等人（2018年）；Gupta等人（2020年）；Chen 等人（2021年）直接根据输入输出对合成特定领域的程序，而无需问题描述。Ellis 等人（2019年）利用测试用例训练强化学习智能体，并采用序贯蒙特卡洛采样方法进行代码生成。与以往工作不同，我们将单元测试结果用作基于树搜索的规划算法的奖励信号，并进一步将该算法与基于 Transformer的大语言模型相结合，以生成更优质的代码。

面向代码生成的规划与强化学习。代码生成问题已被建模为序贯决策问题（Bunel等人，2018年；Chen 等人，2018年），这使得我们能够为代码生成设计并应用强化学习（RL）和规划算法。强化学习已用于 Transformer 代码生成的训练阶段和解码阶段。在训练阶段，Le等人(2022年)；Xu等人(2019b) 使用一种强化学习目标，优化生成程序的正确性，而不是优化它们与参考解答的相似度。在解码阶段，蒙特卡洛树搜索（MCTS）算法已被用于搜索高质量代码。然而，MCTS 本身无法扩展到更大的领域。它仅用于生成特定领域语言，或用于受限的程序合成任务，例如汇编代码生成（Xu等人，2019a）、Java字节码翻译（Lim和Yoo，2016年）以及机器人规划（Matulewicz，2022年）。这些任务的规模远小于我们工作中生成 Python 代码的规模。因此，它们的框架不需要大语言模型，也无需应对将规划算法与大语言模型相结合这一挑战。

结合深度学习或将默认策略作为先验知识后，蒙特卡洛树搜索在大型领域中更高效且适用性更强（Gelly &Silver，2011；Silver 等人，2016年；Simmons-Edler等人，2018年）。我们的工作采用了相同的总体方案。具体而言，我们将大语言模型与树搜索算法相结合，设计出一种能够解决竞赛编程问题的新算法，并设计了提升其效率的机制。

自然语言生成中的规划。规划算法（如蒙特卡洛树搜索）也被用于为不同的自然语言处理（NLP）任务寻找最优文本输出。例如，Scialom 等人 (2021年）；Leblond 等人(2021年）；Chaffin 等人 (2022年) 使用预训练判别器或预定义指标作为奖励函数。我们想强调，我们是首个将树搜索算法与大语言模型相结合，用于通用编程语言生成的团队。我们设计了这两个组件之间的接口，并解决了使该框架在计算上高效运行所面临的独特挑战。与此同时，其他搜索算法，如 A*算法（Hart 等人，1968）也应用于 Transformer 解码过程。Lu 等人(2021年) 研究了受约束文本生成问题，并将 A*算法与束搜索算法结合起来。然而，他们的约束条件以形式逻辑表达式的形式表示，这与我们的问题中最大化通过率的目标不同。

3 方法

3.1 概述

我们研究竞赛编程中的代码生成问题，如图所示。 1。智能体会获得一道编程题的自然语言描述。它需要理解问题描述，并生成能够解决该问题的程序。与 Li 等人(2022年)；Chen 等人 (2018年)；Ellis 等人 (2019年) 类似，我们假设智能体可以访问一组测试用例，每个测试用例由一对输入、输出字符串组成。给定输入字符串，智能体需要生成一个程序，使其输出与测试用例的输出字符串完全一致。我们的目标是生成一个能通过最多数量测试用例的程序。为了确定智能体能否生成可泛化到未见测试用例的程序，我们按照 *Li* 等人 的术语，将测试用例分为公开和私有测试用例，这一划分沿用 Li 等人(2022年)。在程序生成过程中，智能体只能

问题陈述
给定一个字符串 S。将 S 中的每个字符替换为 x 并打印结果。
约束条件
(1). S 是一个由小写英文字母组成的字符串。
(2). S 的长度介于1 和 100 之间（含两端）。
输入
输入按以下格式从标准输入提供： S
输出
将 S 中的每个字符替换为 x，并打印结果。
样例测试输入
$sardine$
样例测试输出
$xxxxxxx$

```
s=input()
s=list(s)
for i in range(len(s)):
    for j in range(len(s)):
        if s[i]=="x":
            s[i]=j
    print("".join(s))

```

束搜索（通过率：0.00）。

```
s=input()
s=list(s)
for i in range(len(s)):
    if s[i]=="x":
        s[i]="x"
    else:
        continue
print("".join(s))
```

采样 + 筛选（通过率：0.22）。

```
s=str(input())
for i in range(len(s)):
    if s[i]!="x":
        s=s[:i]+"x"+s[i+1:]

print(s)

```

PG-TD（通过率：1.00）。

图 1：竞赛编程中的代码生成示例，展示了问题描述（上方）以及由基线算法和我们的 PG-TD 算法生成的程序（下方）。

访问公开测试用例，而我们使用私有测试用例来评估它生成的程序。

Transformer 模型（Li 等人，2022年；Hendrycks 等人，2021年）由于具备序列到序列建模能力，已广泛应用于代码生成。在 Transformer 的生成过程中，采用束搜索（Graves，2012）和采样（Fan等人，2018年；Dabre 和 Fujita，2020年）来生成代码序列。然而，这些算法难以优化与训练目标不同的目标（通常是与参考解答的相似度）。因此，我们无法直接使用这些生成算法来生成旨在通过更多测试用例的程序。
另一方面，规划算法可以直接优化通过率或任何理想的编程目标。为使用规划算法，我们遵循 Bunel等人(2018年);Ellis 等人(2019年) 的方法，将代码生成问题形式化为马尔可夫决策过程（MDP）（Sutton & Barto，2018年）。在这一形式化中，**状态** s 是问题描述与部分或完整程序的拼接；完整程序以一个特殊的终止词元结尾。一个**动作** a 是 Transformer 词汇表中的一个词元。另有一个特殊的终止动作（终止词元），表示智能体认为程序已经完成。**状态转移函数**以确定性方式将状态 s 与一个词元 a拼接起来；当智能体执行终止动作时，一个回合结束。**奖励**是指状态 s 下程序在公开测试用例上的通过率，前提是 s 是完整程序（即，当 s 的最后一个词元是终止词元时）。部分程序的奖励始终为 0。

关于如何在马尔可夫决策过程中寻找最优策略，已有大量研究。本文考虑一种受蒙特卡洛树搜索（MCTS）启发的基于树搜索的规划算法，如图 2 所示。直观而言，树搜索算法会维护一棵树结构，其中节点对应状态，边对应动作。算法从根节点（初始状态）出发，在状态空间中搜索，以找到奖励较高的终止状态。它会维护：1）每个节点被访问的次数；2）一个价值函数，用于记录从节点（或状态） s 开始并执行动作 a 后所获得的最大奖励。算法会访问并扩展价值较高（因为这些节点会导向质量更高的程序）或访问次数较少（因为这些节点尚未得到充分探索）的节点。在本节接下来的部分中，我们将介绍如何将树搜索算法整合到 Transformer 的生成过程中。

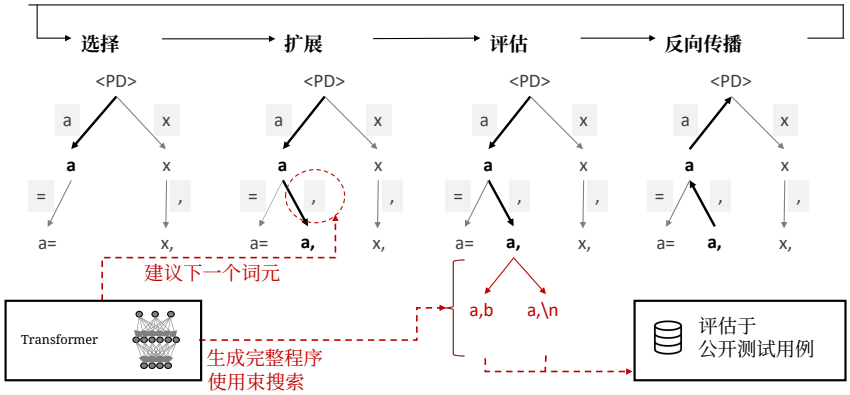


图 2：在代码生成的 Transformer 生成过程中使用蒙特卡洛树搜索算法的示意图。 <问题描述>表示问题描述。

3.2 规划引导的 Transformer 解码

现在，我们可以回答引言中提出的问题：能否将规划算法与预训练代码生成 Transformer 相结合，以生成更好的程序？我们设计了一种 Transformer 生成算法，其中使用树搜索算法进行前瞻规划。由于搜索空间庞大，单靠树搜索算法可能无法找到高质量的代码。因此，树搜索算法会利用传统的 Transformer 束搜索算法以及预训练 Transformer 提供的下一个词元的概率来引导搜索过程。

我们在算法1 中给出了规划引导的 Transformer 解码算法（PG-TD）的伪代码，并在图2中展示了整个过程。PG-TD 算法遵循与标准蒙特卡洛树搜索算法相同的框架，并基于Silver 等人 (2017年) 使用的实现。这里，我们重点介绍 Transformer 在树搜索步骤中的用法。算法的更多细节见第D.1节。

算法 1PG-TD 算法。

要求： $root$：当前状态； c：P-UCB 探索参数——参数； k：任意节点的子节点最大数量；
b：Transformer 束搜索的波束数量。

$program\ dict = $ 字典()
对于 $i \leftarrow 1, 2, \ldots, max\ rollouts$ **执行**
  $node \leftarrow root$
  # 选择
  当 $|node.children| > 0$ **时执行**
    $node \leftarrow$ P_UCB 选择$(node.children, c)$
  结束当
  # 扩展
  $next_tokens \leftarrow$ TOP_K$(node,\ k)$
  对 $next\ token \in next\ tokens$ **执行**
    $next_state \leftarrow$ 拼接$(node,\ next\ token)$
    为 $next\ state$ 创建一个节点 new_node
    将 $new\ node$ 添加到 $node$ 的子节点中
  结束循环
  # 评估
  $p \leftarrow$ 束搜索$(node, b)$
  $r \leftarrow$获取奖励(p)
  $program_dict[p] = r$
  # 反向传播
  更新 $node$ 及其祖先节点在树中的值为 r
结束循环
返回 奖励最高的程序 $program\ dict$

在**选择**步骤中，我们遵循Silver 等人(2017年) 的做法，使用 P-UCB 算法来选择要探索的树分支。在 P-UCB 中，我们根据 Transformer 确定的下一个词元的概率对探索项加权。因此，树搜索会更频繁地选择概率较高的词元。选择算法由探索参数 c 控制，c值越大，探索程度越高。P-UCB 的详细信息见第D.1节。

在**扩展**步骤中，选定树中的一个节点后，我们会选出可能的下一个词元，并将对应的下一个状态作为新节点添加到其子节点列表中（为简洁起见，节点也指其所表示的状态）。像标准蒙特卡洛树搜索那样随机采样一个词元，很可能会导致语法错误。因此，我们调用 TOP K 来获取最有可能的下一个词元，其中 TOP K$(s,\ k)$ 返回从 s开始的 k 个最有可能的下一个词元； k 是任一节点最多可拥有的子节点数量。对应的 k 下一个状态是将当前状态分别与每个

Transformer 建议的下一个词元拼接而成。这些下一个状态会添加到当前节点的子节点列表中。（第9-14行）

在**评估**步骤中，我们需要评估选定的 $node$。请注意，$node$ 可能仍是一个部分程序。我们无法直接评估部分程序的质量，因为我们不知道它将如何补全，也不知道它能通过多少测试用例。在这里，我们再次使用 Transformer，调用束搜索函数，从当前节点生成一个完整程序，其中束搜索$(s,\ b)$ 使用 Transformer 束搜索算法，以前缀 s 和束宽 b生成一个序列。我们在公开测试用例上运行生成的程序以获得其奖励，并将其设为 $node$ (第16-17行）的价值。此价值会沿树向上传播，从而更新其祖先节点的价值（第20行）。

Transformer 与树搜索之间的信息共享。 细心的读者可能会注意到，如果我们遵循上述算法，可能会产生大量重复计算。关键的观察是，Transformer 束搜索算法也隐式地构建了一个树结构，该结构可供后续树搜索迭代使用。本节接下来将介绍如何通过在 Transformer 束搜索算法与树搜索之间共享信息，来提高该算法的效率。

请看图 3中的示例，其中展示了 PG-TD 的两次迭代。在第 t 次迭代的评估步骤中，Transformer 束搜索算法会隐式地构建一棵树，以找出束中似然最高的序列（图 3 （左））。由于束中只保留 b 个部分程序，因此这棵树在每一层仅扩展似然最高的 b 个节点（图中为 $b = 2$）。其他节点则会被丢弃，不再由束搜索算法考虑。在第 $(t+1)$ 次迭代中，如果树搜索算法选择了“a, ”，那么这一状态已在第 t 次迭代的 Transformer 束搜索中访问过。当算法需要找到

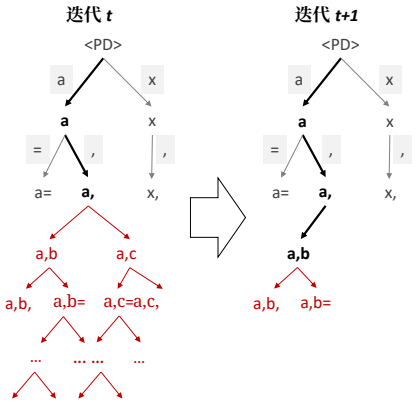


图 3：PG-TD 算法中缓存的示意图。树搜索部分以黑色表示，Transformer 束搜索部分以红色表示。

最可能的 top-k 个后续词元，这类信息已在第 t 次迭代中获取，可以直接复用，无需重新计算。在我们的实现中，我们会缓存 Transformer 束搜索算法生成的树结构。我们将这种实现称为**树结构缓存**。

我们还可以缓存 PG-TD 评估步骤中生成的完整程序。在第 t 次迭代的评估步骤中（图 3），假设贪心最优序列为“a,b=...”。在第（$t+1$）次迭代中，为了生成以“a,b”开头的序列，Transformer 束搜索算法会像之前一样生成相同的序列“a,b=...”。（当束宽大于 1 时，这不一定成立。我们将在第D.2节中对此进行说明。）为提高束搜索的效率，我们会缓存评估步骤中已生成的序列。在后续迭代的评估步骤中，PG-TD 会检查当前状态是否与此前生成的某个序列的前缀相匹配，并直接使用已生成的序列，而不调用 Transformer 束搜索函数。我们将这种实现称为**序列缓存**。下一节将通过实验证实这些缓存方法的有效性。

4 实证评估

在本节中，我们通过回答以下问题，实证考察 PG-TD 算法的有效性和效率。**Q1:** 与 Transformer 束搜索算法和其他有竞争力的基线方法相比，PG-TD 能否生成更好的程序？我们的算法是否具有模型无关性，在应用于不同的 Transformer 模型时都能取得提升？**Q2:** 树搜索算法是 PG-TD 中一种有效的规划算法吗？它是否优于其他规划算法或基于采样的算法？**Q3:** PG-TD 在运行 Transformer 束搜索算法的次数和所耗费的计算时间方面是否高效？**Q4:** 缓存方法能否有效节省计算时间？**Q5:** 我们能否利用 PG-TD 在搜索过程中生成的样本，对 Transformer 进行微调，从而生成更好的解答？

		通过率（%）				严格准确率（%）			
		APPS 入门	APPS 中级	APPS 竞赛	CodeContests	APPS 入门	APPS 中级	APPS 竞赛	CodeContests
APPS GPT-2	束搜索	11.95	9.55	5.04	5.10	5.50	2.10	1.00	0.00
	采样+筛选	25.19	24.13	11.92	20.40	**13.80**	5.70	2.30	3.64
	SMCG-TD	24.10	21.98	10.37	17.47	11.70	5.50	2.10	4.24
	PG-TD（$c = 4$）	**26.70**	**24.92**	**12.89**	**24.05**	13.10	**6.10**	**3.10**	**4.85**
APPS GPT-Neo	束搜索	14.32	9.80	6.39	5.73	6.70	2.00	2.10	0.00
	采样+筛选	27.71	24.85	12.55	25.26	**15.50**	5.80	3.00	4.24
	SMCG-TD	25.09	20.34	9.16	15.44	13.80	5.10	1.80	3.03
	PG-TD ($c = 4$)	**29.27**	**25.69**	**13.55**	**26.07**	**15.50**	**6.43**	**3.50**	**4.85**

表1：PG-TD 与其他算法的结果。所有算法的 Transformer 生成次数上限为 256。

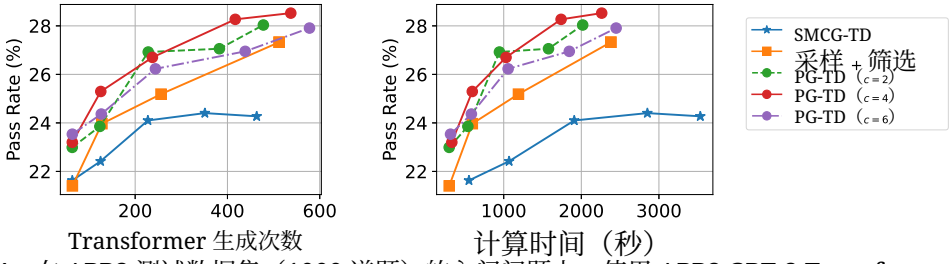


图4：在 APPS 测试数据集（1000 道题）的入门问题上，使用 APPS GPT-2 Transformer 模型时，PG-TD 与基线算法的通过率随 Transformer 生成次数（左）和计算时间（中）的变化。

数据集和模型。 近年来，已有多个竞赛编程数据集可用于代码生成算法的基准测试。对于每个编程问题，这些数据集都包含自然语言程序描述、示例解答和测试用例。我们在一些热门基准数据集上评估了 PG-TD 和基线算法：APPS（Hendrycks 等人，2021年）以及 AlphaCode 中的 CodeContests（Li 等人，2022年）。APPS 数据集未区分公开测试用例和私有测试用例。因此，我们将一个程序的所有测试用例均分为两组：第一组作为算法的公开测试用例，用于优化通过率；第二组作为私有测试用例，用于评估生成的程序。对于 CodeContests，我们将其公开测试用例和生成的测试用例作为公开测试用例，并将其私有测试用例作为私有测试用例。为了证明 PG-TD 具有模型无关性，可应用于不同的预训练 Transformer 并取得更好的性能，我们采用了文献中两种热门的预训练代码生成 Transformer：在 APPS训练数据集上微调的 GPT-2 和 GPT-Neo（Hendrycks 等人，2021年）。

算法。 我们将 PG-TD 与以下算法进行比较。 **束搜索** 仅使用 Trans- former 束搜索算法生成整个程序，不使用测试用例。这是Hendrycks 等人 (2021年)；Chen 等人 (2021a）。我们在 Transformer 生成函数中使用束宽为 5，这与Hendrycks 等人 (2021年）的选择相同。

我们还实现了两种基线方法，它们使用 Transformer 采样程序，并通过测试用例对其进行筛选。 **采样 + 筛选（S+F）**使用 Transformer采样算法生成一组程序。生成完所有程序后，它会计算所有程序在公开测试用例上的通过率，并返回通过率最高的程序。为避免生成可能导致程序完全无法运行的低概率词元，我们采用 top-3采样，即 Transformer 在每一步仅从最可能的三个词元中进行采样。采样温度为 1。此方法与 AlphaCode 中的算法相似（Li 等人，2022年），但由于我们生成的程序数量少得多，因此没有执行聚类。

为了确定树搜索算法的有效性，我们考虑了另一种基线算法，**序贯蒙特卡洛引导的 Transformer 解码（SMCG-TD）**，它用序贯蒙特卡洛算法（Ellis 等人，2019年）取代树搜索算法组件。这是一种迭代算法，维护着一个部分程序种群。它与 PG-TD 中的评估步骤相同，使用 Transformer 确定部分程序的适应度。适应度分数较高的部分程序更有可能在下一次迭代中被选中。关于基线算法的更多细节见第D.3节。

对于 PG-TD，我们默认将任意节点的最大子节点数 (k) 设为 3，并将束宽 (b) 设为 1。对于基线方法，我们在采样中最多采样 512 个程序

方法	时间（秒）
同时使用两种缓存方法	**1312.27**
仅使用序列缓存	1430.63
仅使用树结构缓存	1899.17
不使用缓存	2206.38

表 2：在 APPS 入门数据集的前 100 道题上，使用 APPS GPT-2 (1.5B) 时，缓存对 PG-TD 的影响。

方法	通过率(%)	严格准确率(%)
原始版本	14.32	6.70
使用 S+F 微调	15.24	**7.90**
使用 PG-TD 微调	**16.54**	**7.90**

表3：使用束搜索进行代码生成时，在 APPS 入门级数据集上微调后的 APPSGPT-2（2.7B）的性能。

+ 筛选，并在 SMCG-TD 的每次迭代中保留一组 200 个部分程序。除非另有说明，后续实验均采用这些参数。实验在 APPS 数据集的测试集（5000 道题）（Hendrycks 等人，2021年）和 CodeContests 的测试集（165 道题）（Li 等人，2022年）上进行。我们使用与Hendrycks 等人（2021年）相同的指标，即私有测试用例上的通过率和严格准确率。具体而言，通过率是指所有问题中，生成的程序通过的私有测试用例所占平均百分比。严格准确率是指生成的程序通过全部私有测试用例的问题所占百分比。

PG-TD 的有效性。 为了进行公平比较，我们评估各算法在使用相同*Transformer* 生成次数时找到的最佳程序。具体而言，Transformer 生成次数是指 PG-TD 和 SMCG-TD 中束搜索的函数调用次数（未找到并用于序列缓存的缓存序列时），以及 S+F 中采样函数的调用次数。

如表 1所示，在通过率指标下，PG-TD 在所有数据集上始终优于其他所有基线方法。随着我们对通过率进行优化，我们预期算法在严格准确率方面也会优于其他基线方法。事实几乎如此，唯一的例外是，在 APPS 入门数据集上，我们的算法与 S+F 不相上下，或表现不及它，这主要是因为该数据集中的程序难度较低。不同算法在此数据集上的差距也较小。总体而言，这些结果证实，我们的算法确实生成了能通过更多测试用例的程序，回答了**Q1**。具体来说，S+F 和 SMCG-TD 都使用测试用例来筛选 Transformer 生成的程序，但总体表现均不及 PG-TD。因此，基于树搜索的规划算法确实足够强大，并且能够有效地与 Transformer 集成，回答了**Q2**。我们还在第n@k 和 pass@k 指标下报告了 PG-TD 和 S+F 的表现，见第**A** 节（表**6**；

PG-TD 的效率。为了展示 PG-TD 的效率，我们报告了算法在相同计算时间下找到的最佳程序的通过率（图 4 （右））。实验在 APPS 测试集中的入门问题上进行。由于这些算法可以有不同的实现方式（可能会使用并行化），我们也报告了以 Transformer 生成次数为预算时的结果（图4 （左））。

对于 PG-TD，我们将 $k = 3$，$b = 1$ 并将 P-UCB 探索参数 c 分别设为 2、4 和 6。我们发现，采用 $c = 4$ 的 PG-TD 性能最佳，而设置 $c = 2$ 往往会导致探索不足，设置 $c = 6$则会导致过度探索。在相同的计算预算下，S+F 生成的程序通过率较低（回答 **Q3**）。这印证了我们的观察：S+F 与 AlphaCode 使用的算法（Li 等人，2022年）类似，只依据 Transformer 的生成概率来生成解答，直到所有程序生成完毕后才考虑它们的通过率。PG-TD 会在生成过程中主动考虑生成程序的通过率，因此效率更高。另一方面，SMCG-TD 也利用 Transformer 和公开测试用例来引导生成过程。然而，由于其采样特性，它无法像 PG-TD 那样进行多步前瞻搜索，因此性能较差。我们将 PG-TD 使用不同 k，b 以及 S+F 使用不同温度的更多结果留在第 A节。

缓存的有效性。针对树结构缓存和序列缓存的设计选择，我们进行了消融研究，以验证它们的效率。在表2中，我们比较了使用和不使用树结构缓存及序列缓存的 PG-TD 版本。正如预期的那样，不使用序列缓存时，算法需要重新生成完整序列，因此耗时大幅增加。不使用树结构缓存时，算法的速度稍慢，因为它需要调用 Trans-former 来获取最可能的下一个词元（回答 **Q4**）。

使用 PG-TD 生成的样本微调 Transformer。既然我们能够使用 PG-TD 算法生成通过率较高的解，能否利用这些生成的解对代码生成 Transformer 进行微调，进一步提升其性能？这或许能有效解决可用于训练代码生成 Transformer 的高质量人工编写程序稀缺的问题。具体而言，我们首先在 APPS 训练集上运行 PG-TD。随后，将通过率高于 80% 的生成解加入 APPS 样例解集合，并使用扩充后的解集合对 GPT-2（2.7B）模型进行微调。接着，我们使用微调后的 Transformer 模型运行束搜索，在测试集上生成解，以检验其性能是否有所提升。我们使用 APPS 测试集中面试级问题的前 500 道题进行验证，并使用 APPS 测试集中的入门级问题进行测试。结果见表3。使用 PG-TD 生成的解进行微调后，通过率和严格准确率均有所提升。更多详情见第D.4节。

优化其他代码生成目标。除了通过率之外，我们还可以通过使用不同的奖励函数让算法更加多样化。我们考虑了两个新的目标：代码长度惩罚和注释激励。如表4所示，代码长度惩罚奖励函数使模型生成更简洁的代码；注释激励奖励函数则鼓励模

方法	Code 长度 ↓	注释数量 ↑	Pass rate ↑
默认	248.42	0.68	**23.14**
长度惩罚	**190.73**	-	22.82
注释鼓励	-	**3.11**	21.65

表 4：可控代码生成的性能。代码长度表示生成的代码字符串的长度，注释数量表示包含注释的代码行数。

以生成包含更多注释的代码。两个目标在公开测试用例上仍取得了合理的通过率。我们在附录第C节提供了定性示例以及奖励函数的更多细节。

使用自动生成的测试用例。 本文使用的两个数据集都包含可用于计算 PG-TD 奖励的测试用例。然而，在现实中，人工指定的测试用例并非总是可用。最近，Chen 等人(2022年发现，在代码生成任务上预训练的 Transformer 也能通过在提示末尾添加`assert` 关键字来生成有用的测试用例。我们遵循Chen 等人(2022年) 的提示设计来自动生成测试用例，并使用自动生成的测试用例运行 PG-TD 算法。通过实验，我们确认，与束搜索相比，使用自动生成的测试用例验证生成的程序时，PG-TD 仍具有更高的严格准确率。方法和实验的详细信息见附录第B节。

5 讨论与结论

总之，我们提出了一种新颖的算法，将预训练 Transformer 的能力与受蒙特卡洛树搜索启发的树搜索算法相结合。我们对该算法进行了评估，并通过实验证明，在不同设置下，与其他基线算法相比，该算法生成的程序质量更高。我们还设计了针对模型结构的缓存机制，有助于节省计算开销。我们证明，该算法具有通用性，无需对 Transformer 进行微调，即可根据通过率以外的目标生成代码。我们希望这项工作能够启发更多方法，将规划算法融入 Transformer 的代码生成过程或其他问题中。

我们算法的一个局限在于依赖测试用例，不过我们发现，即使测试用例数量很少，也能帮助找到更好的解决方案（第 A 节，表7）。为了解决这一局限，我们提供了结果，表明算法可以利用自动生成的测试用例（第 B 节）。与纯 Transformer 束搜索算法相比，我们的算法计算开销也更大，因为它需要在树搜索算法中多次运行束搜索算法。未来，我们考虑使用价值函数来估计程序的通过率，以提升该框架的性能，这与 Silver 等人(2016年) 为精通围棋而采用的评估函数类似。我们可以像 Chen 等人(2018年) 那样，为部分程序学习神经状态表示。我们还考虑在评估步骤中使用并行树搜索算法（Chaslot 等人，2008），以并行计算不同状态。

致谢。本研究获得 MIT-IBM Watson 人工智能实验室、DARPA MCS 的支持，以及来自 MERL、思科和亚马逊的资助。我们还要感谢 AiMOS 提供计算支持；AiMOS 是 IBM Research 人工智能硬件中心的服务器集群。

参考文献

Wasi Ahmad、Saikat Chakraborty、Baishakhi Ray 和 Kai-Wei Chang。面向程序理解与代的统一预训练。在 *2021*年北美计算语言学协会分会会议论文集：人类语言技术，第 2655–2668 页，线上，2021年6月。计算语言学协会。 1

Jacob Austin、Augustus Odena、Maxwell Nye、Maarten Bosma、Henryk Michalewski、David Dohan、Ellen Jiang、Carrie Cai、Michael Terry、Quoc Le 和 Charles Sutton。使用大语言模型进行程序合成。*arXiv:2108*。*07732 [cs]*, 2021年8月。网址http://arxiv.org/abs/2108.07732。arXiv：2108.07732。 2

Rudy Bunel、Matthew Hausknecht、Jacob Devlin、Rishabh Singh 和 Pushmeet Kohli。利用语法和强化学习进行神经程序合成。 *ICLR*，2018年5月。网址http://arxiv.org/abs/1805.04276。arXiv：1805.04276。 3，4

Antoine Chaffin、Vincent Claveau 和 Ewa Kijak。Ppl-mcts：通过判别器引导的蒙特卡洛树搜索解码实现受约束文本生成。载于 *2022*年计算语言学协会北美分会年会论文集：人类语言技术，第 2953–2967 页，2022年。 3

Guillaume MJ-B Chaslot、Mark HM Winands 和 HJVD Herik。并行蒙特卡洛树搜索。载于国际计算机与游戏会议，第 60–71 页。Springer，2008年。 9

Bei Chen、Fengji Zhang、Anh Nguyen、Daoguang Zan、Zeqi Lin、Jian-Guang Lou 和 Weizhu Chen。CodeT：使用生成的测试进行代码生成，2022年7月。网址http://arxiv.org/abs/2207.10397。arXiv:2207.10397 [cs]。 9，17

Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman、Alex Ray、Raul Puri、Gretchen Krueger、Michael Petrov、Heidy Khlaaf、Girish Sastry、Pamela Mishkin、Brooke Chan、Scott Gray、Nick Ryder、Mikhail Pavlov、Alethea Power、Lukasz Kaiser、Mohammad Bavarian、Clemens Winter、Philippe Tillet、Felipe Petroski Such、Dave Cummings、Matthias Plappert、Fotios Chantzis、Elizabeth Barnes、Ariel Herbert-Voss、William Hebgen Guss、Alex Nichol、Alex Paino、Nikolas Tezak、Jie Tang、Igor Babuschkin、Suchir Balaji、Shantanu Jain、William Saunders、Christopher Hesse、Andrew N. Carr、Jan Leike、Josh Achiam、Vedant Misra、Evan Morikawa、Alec Radford、Matthew Knight、Miles Brundage、Mira Murati、Katie Mayer、Peter Welinder、Bob McGrew、Dario Amodei、Sam McCandlish、Ilya Sutskever 和 Wojciech Zaremba。《在代码上训练的大语言模型评估》。*arXiv:2107.03374 [cs]*, 2021年7月a。网址http://arxiv.org/abs/2107.03374。arXiv：2107.03374。1，2，7，15，17

Xinyun Chen、Chang Liu 和 Dawn Song。面向执行引导的神经程序合成。国际学习表征会议，2018年。3，9

Xinyun Chen、Dawn Song 和 Yuandong Tian。用于特定领域语言之外的神经程序合成的潜在执行。神经信息处理系统进展，2021年。3

Raj Dabre 和 Atsushi Fujita。用于训练神经机器翻译模型的 Softmax 温度调节。*arXiv*，2020年。1，4

Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。BERT：用于语言理解的深度双向变换器预训练。载于*2019*年计算语言学协会北美分会年会论文集：人类语言技术，第*1*卷（长篇与短篇论文），第4171–4186页，明尼阿波利斯，明尼苏达州，2019年6月。计算语言学协会。doi：10.18653/v1/N19-1423。网址：https://aclanthology.org/N19-1423。2

Kevin Ellis、Catherine Wong、Maxwell Nye、Mathias Sabl´e-Meyer、Lucas Morales、Luke Hewitt、Luc Cary、Armando Solar-Lezama 和 Joshua B Tenenbaum。Dreamcoder：通过唤醒-睡眠库学习引导归纳式程序合成。载于第*42*届 *ACM SIGPLAN* 编程语言设计与实现国际会议论文集，第835–850页，2021年。2

Kevin M. Ellis、Maxwell Nye、Yewen Pu、Felix Sosa、Joshua Tenenbaum 和 Armando Solar- Lezama。编写、执行、评估：使用交互式解释器进行程序合成。神经信息处理系统，2019年。3、4、7 Angela Fan、Mike Lewis 和 Yann Dauphin。分层神经故事生成。载于《计算语言学协会》第56届年会论文集（第1卷：长篇论文），第889–898页，2018年。1、3、4 Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang 等。CodeBERT：一种面向编程语言和自然语言的预训练模型。载于计算语言学协会论文集：EMNLP 2020，第1536–1547页，2020年。2 Sylvain Gelly 和 David Silver。蒙特卡洛树搜索与快速动作价值估计在计算机围棋中的应用。《人工智能》，175(11):1856–1875，2011年。3 Alex Graves。使用循环神经网络进行序列转导。载于国际机器学习大会：表征学习研讨会，2012年。3、4 Daya Guo、Shuo Ren、Shuai Lu、Zhangyin Feng、Duyu Tang、LIU Shujie、Long Zhou、Nan Duan、Alexey Svyatkovskiy、Shengyu Fu 等。GraphCodeBERT：利用数据流对代码表征进行预训练。载于国际学习表征会议，2020年。2 Kavi Gupta、Peter Ebert Christensen、Xinyun Chen 和 Dawn Song。合成、执行与调试：学习修复神经程序合成。神经信息处理系统进展，2020年。3 Peter E Hart、Nils J Nilsson 和 Bertram Raphael。最小成本路径启发式求解的形式化基础。IEEE系统科学与控制论汇刊，4(2):100–107，1968年。3 Dan Hendrycks、Steven Basart、Saurav Kadavath、Mantas Mazeika、Akul Arora、Ethan Guo、Collin Burns、Samir Puranik、Horace He、Dawn Song 和 Jacob Steinhardt。使用 APPS 衡量编程挑战能力。NeurIPS，2021年。1、4、7、8、14、17 Aditya Kanade、Petros Maniatis、Gogul Balakrishnan 和 Kensen Shi。学习并评估源代码的上下文嵌入。载于国际机器学习大会，第5110–5121页。PMLR，2020年。2 Levente Kocsis 和 Csaba Szepesv´ari。基于多臂老虎机的蒙特卡洛规划。载于欧洲机器学习会议，第282–293页。Springer，2006年。21 Hung Le、Yue Wang、Akhilesh Deepak Gotmare、Silvio Savarese 和 Steven C. H. Hoi。CodeRL：通过预训练模型和深度强化学习掌握代码生成。arXiv，2022年7月。doi：10.48550/arXiv.2207.01780。URL http://arxiv.org/abs/2207. 01780。arXiv:2207.01780 [cs]。3、24 R´emi Leblond、Jean-Baptiste Alayrac、Laurent Sifre、Miruna Pislar、Jean-Baptiste Lespiau、Ioannis Antonoglou、Karen Simonyan 和 Oriol Vinyals。超越束搜索的机器翻译解码。arXiv预印本 arXiv:2104.05336，2021年。3 Yujia Li、David Choi、Junyoung Chung、Nate Kushman、Julian Schrittwieser、R´emi Leblond、Tom Eccles、James Keeling、Felix Gimeno 和 Agustin Dal Lago。使用 AlphaCode 实现竞赛级代码生成。arXiv预印本 arXiv:2203.07814，2022年。1、2、3、4、7、8、14、17 Jinsuk Lim 和 Shin Yoo。实地报告：将蒙特卡洛树搜索应用于程序合成。载于国际搜索式软件工程研讨会，第304–310页。Springer，2016年。3

Ximing Lu、Sean Welleck、Peter West、Liwei Jiang、Jungo Kasai、Daniel Khashabi、Ronan Le Bras、Lianhui Qin、Youngjae Yu、Rowan Zellers、Noah A. Smith 和 Yejin Choi。NeuroLogic A*风格解码：结合前瞻启发式的受约束文本生成，2021年12月。URL http://arxiv.org/abs/2112.08726。arXiv:2112.08726 [cs]。3 Nadia Matulewicz。通过使用基于启发式的损失函数引导的蒙特卡洛树搜索进行归纳式程序合成。2022年。3 Glenford J Myers。《软件测试艺术》。1979年。3 Maxwell Nye、Armando Solar-Lezama、Josh Tenenbaum 和 Brenden M Lake。通过神经程序合成学习组合规则。神经信息处理系统进展，33:10832–10842，2020年。2 Alec Radford、Jeff Wu、Rewon Child、David Luan、Dario Amodei 和 Ilya Sutskever。语言模型是无监督多任务学习器。2019年。2、28 Colin Raffel、Noam Shazeer、Adam Roberts、Katherine Lee、Sharan Narang、Michael Matena、Yanqi Zhou、Wei Li 和 Peter J. Liu。探索统一文本到文本 Transformer 的迁移学习极限。《机器学习研究期刊》，2020年。URL http://jmlr.org/ papers/v21/20-074.html。2 Baptiste Roziere、Marie-Anne Lachaux、Lowik Chanussot 和 Guillaume Lample。编程语言的无监督翻译。神经信息处理系统进展，33，2020年。2 Baptiste Roziere、Jie M Zhang、Francois Charton、Mark Harman、Gabriel Synnaeve 和 Guillaume Lample。利用自动化单元测试实现无监督代码翻译。载于国际学习表征会议，2022年。1、3 Thomas Scialom、Paul-Alexis Dray、Jacopo Staiano、Sylvain Lamprier 和 Benjamin Piwowarski。要不要使用束搜索：这是一个关于语言生成对抗网络协作的问题。神经信息处理系统进展，34:26585–26597，2021年。3 David Silver、Aja Huang、Chris J. Maddison、Arthur Guez、Laurent Sifre、George Van Den Driess- che、Julian Schrittwieser、Ioannis Antonoglou、Veda Panneershelvam 和 Marc Lanctot。利用深度神经网络和树搜索掌握围棋。《自然》，529(7587):484，2016年。出版方：自然出版集团。3、9 David Silver、Thomas Hubert、Julian Schrittwieser、Ioannis Antonoglou、Matthew Lai、Arthur Guez、Marc Lanctot、Laurent Sifre、Dharshan Kumaran、Thore Graepel、Timothy Lillicrap、Karen Si- monyan 和 Demis Hassabis。通过通用强化学习算法自我对弈掌握国际象棋和将棋，2017年12月。URL http://arxiv.org/abs/1712.01815。arXiv:1712.01815 [cs]。5、21 Riley Simmons-Edler、Anders Miltner 和 Sebastian Seung。通过强化学习引导的树搜索进行程序合成。arXiv预印本 arXiv:1806.02932，2018年。3 Richard S. Sutton 和 Andrew G. Barto。《强化学习：导论》。麻省理工学院出版社，2018年10月。ISBN 978-0-262-35270-3。4 Michele Tufano、Dawn Drain、Alexey Svyatkovskiy、Shao Kun Deng 和 Neel Sundaresan。使用 Transformer 和焦点上下文生成单元测试用例。arXiv，2020a。3 Michele Tufano、Dawn Drain、Alexey Svyatkovskiy 和 Neel Sundaresan。使用预训练 Transformer 为单元测试用例生成准确的断言语句。arXiv，2020b。3 Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。《注意力就是你所需要的一切》。神经信息处理系统进展，2017a。1、3 Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、Lukasz Kaiser 和 Illia Polosukhin。《注意力就是你所需要的一切》。arXiv:1706.*03762 [*cs*]*, December 2017b。URL http://arxiv.org/abs/1706.03762。arXiv：1706.03762。2

Yue Wang、Weishi Wang、Shafiq Joty 和 Steven C.H. Hoi。CodeT5：面向代码理解与生成的标识符感知统一预训练编码器-解码器模型。载于*2021*年自然语言处理实证方法会议论文集，第 8696–8708 页，线上及多米尼加共和国蓬塔卡纳，2021年11月。计算语言学协会。doi：10.18653/v1/2021.emnlp-main.685。网址：https://aclanthology.org/2021.emnlp-main.685。1，2

Thomas Wolf、Lysandre Debut、Victor Sanh、Julien Chaumond、Clement Delangue、Anthony Moi、PierricCistac、TimRault、R´emiLouf 和MorganFuntowicz。《Hugging Face Transformers：最先进的自然语言处理》。*arXiv* 预印本 *arXiv:1910.03771*，2019年。17

Yifan Xu、Lu Dai、Udaikaran Singh、Kening Zhang 和 Zhuowen Tu。通过自学习进行神经程序合成。*arXiv* 预印本 *arXiv:1910.05865*，2019年a。3

Yifan Xu、Lu Dai、Udaikaran Singh、Kening Zhang 和 Zhuowen Tu。《通过自学习进行神经程序合成》。2019年10月b。doi：10.48550/arXiv.1910.05865。网址：https://arxiv.org/abs/1910.05865v1。3

附录

在本附录中，我们补充主论文的内容，提供更全面的实证评估来支持我们的主张，并更详细地描述算法，以帮助读者更好地理解我们的论文。

本附录的组织结构如下。

- 在第 A 节中，我们提供了算法和基线算法更全面的结果。我们还列出了所用数据集的许可证信息。
- 在第 B 节中，我们考虑未提供测试用例的情形，并使用自动生成的测试用例评估 PG-TD 算法。
- 在第 C 节中，我们提供实证证据，支持讨论部分中的论点：我们的算法用途广泛，除了通过率之外，还可用于优化其他代码生成目标。我们考察了生成简洁代码以及生成包含更多注释的代码这两个目标。
- 在第 D 节中，我们将进一步介绍 PG-TD 的各个组成部分以及基线算法的详细信息。
- 在第 E 节中，我们将展示更多由 PG-TD 算法和基线算法生成的代码示例。
- 在第 F 节中，我们将进一步讨论该算法的优势及其可能带来的负面社会影响。

A 实证评估

我们报告了算法及其他基线方法在整个 APPS 数据集（Hendrycks 等人，2021年）和 CodeContests（Li 等人，2022年）上的表现。APPS 测试数据集包含三个难度级别的编程题：入门级（1000 道题）、面试级（3000 道题）和竞赛级（1000 道题）。

除了主论文表 1 中的结果外，表 5 还展示了 Transformer 生成次数预算为 512 时的结果。正如预期，随着生成的程序数量增加，PG-TD 与其他方法之间的差距会缩小。采样 + 筛选在某些数据集子集上的严格准确率高于我们的算法。我们还测试了 PG-TD 的其他参数设置。在图 5 和 6中，我们改变束宽（$b = 1$，3，5），并设置 $c = 2$，4。在图 7中，我们改变任一节点的最大子节点数（$k = 2$，3，4）。
虽然人们可能会认为，扩大树的宽度（增加 k）有助于找到更好的程序，但考虑 Transformer 提出的概率较低的词元，未必有助于找到更好的程序。事实上，考虑低概率词元会浪费 Transformer 生成次数的数量预算和计算时间。另一方面，当模拟 rollout 的数量（对应于 Transformer 生成次数的数量）较少时，增加束宽（b）确实有助于提高通过率。然而，在模拟 rollout 数量较多时，增加束宽并不总能提高通过率（图 7），这可能是过拟合所致。即使使用了序列缓存，增加 b也会耗费更多计算时间。

在图 8中，我们在采样 + 筛选算法中采用了不同的温度（$t = 0.6$、0.8、1），这证实了我们在论文主体实验中选择的 $t = 1$确实取得了最佳性能。

n@k 和 pass@k 指标。我们还报告了 PG-TD 和 S+F 在 n@k 和 pass@k 指标下的表现（Li 等，2022）。对于给定的问题，每种算法都需要生成 k 个程序，并提交 n 个程序进行评估（提交在公开测试用例上奖励最高的 n 个程序）。n@k 是这样的问题所占的比例：提交的 n 个程序中至少有一个通过所有私有测试用例。pass@k 是这样的问题所占的比例：生成的 k 个程序中至少有一个通过所有私有测试用例（实际上即 k@k）。对于 PG-TD，k 个样本是前 k 次模拟 rollout 中找到的完整程序。结果列于表 6，评估对象为 APPS 入门问题。我们观察到，当 n 或 k 的值较小时，PG-TD 能找到的程序

		通过率 (%)				严格准确率 (%)			
		APPS 入门	APPS 中级	APPS 竞赛	CodeContests	APPS 入门	APPS 中级	APPS 竞赛	CodeContests
APPS GPT-2	束搜索	11.95	9.55	5.04	5.10	5.50	2.10	1.00	0.00
	采样+筛选	27.33	25.75	12.09	21.57	**14.80**	**6.97**	2.40	4.24
	SMCG-TD	24.40	22.18	10.60	17.72	11.80	5.87	2.50	4.24
	PG-TD（$c = 4$）	**28.27**	**26.13**	**13.74**	**25.91**	14.40	6.63	**4.00**	**4.85**
APPS GPT-Neo	束搜索	14.32	9.80	6.39	5.73	6.70	2.00	2.10	0.00
	采样+筛选	30.23	26.58	13.12	26.57	**16.70**	**7.40**	3.10	4.24
	SMCG-TD	24.77	20.52	9.19	15.42	14.00	5.20	1.70	2.42
	PG-TD ($c = 4$)	**30.38**	**26.89**	**13.28**	**27.10**	**16.70**	6.90	**3.20**	**5.45**

表5：PG-TD 与其他算法的结果。所有算法的 Transformer 生成次数上限均为 512。

	1@10	1@50	1@100
PG-TD ($c = 4$)	**8.1**	**10.2**	**11.6**
S+F	6.5	9.7	11.4
	5@10	5@50	5@100
PG-TD ($c = 4$)	**12.5**	**14.9**	16.2
S+F	11.9	14.6	**18.2**
	pass@10	pass@50	pass@100
PG-TD ($c = 4$)	**14.7**	**23.7**	27.3
S+F	**14.7**	23.2	**28.3**

表 6：PG-TD 和采样在 APPS 入门问题上的 **n@k** 与 **pass@k** 结果。

	通过率	严格准确率
0 个公开测试用例（束搜索基线）	13.61	4.2
1 个公开测试用例	23.27	7.4
3 个公开测试用例	33.02	13.4
5 个公开测试用例	32.99	14.2
一半的测试用例（平均有 10.44 个公开测试用例）	37.71	14.8

表7：在前 500 个 APPS 入门问题上评估使用不同数量公开测试用例的 PG-TD 性能。

更有可能通过所有私有测试用例。当生成更多程序（**k**）或提交更多程序进行评估（**n**）时，PG-TD 的性能与 S+F 相当或不及后者。

使用不同数量的公开测试用例。在 APPS 数据集上，我们将所有测试用例平均分为公开测试用例和私有测试用例。为了探究 PG-TD 达到满意性能所需的测试用例数量，我们分别使用 1、3 和 5 个公开测试用例，并将其余测试用例作为私有测试用例。结果如表7所示，评估对象为前 500 个 APPS 入门问题。我们发现，即使公开测试用例数量很少，PG-TD 也能找到通过率和严格准确率均高于束搜索的程序。这也表明 Transformer 可以在代码生成中起到正则化作用。即使公开测试用例数量较少，PG-TD 生成的程序仍与人工编写的程序相似，而不会对公开测试用例过拟合。

Codex 上的结果。为了评估我们的算法在更强大的代码生成 Transformer 上的效果，我们使用 Codex（Chen 等人，2021a）作为骨干模型，运行束搜索和 PG-TD 算法。我们遵循 Codex 指南中提供的“最佳实践”，在开头的块注释中写入问题描述，并让 Codex 补全其余部分。提示如下所示。

```
"""
Python 3 {problem
description} """
```

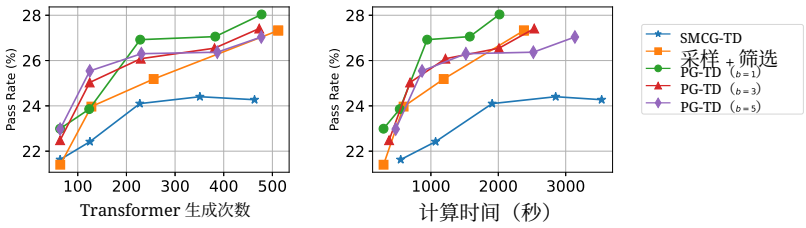


图 5：在 APPS 入门数据集上，使用 APPS GPT-2 Transformer 模型时 PG-TD ($c = 2$) 的结果。

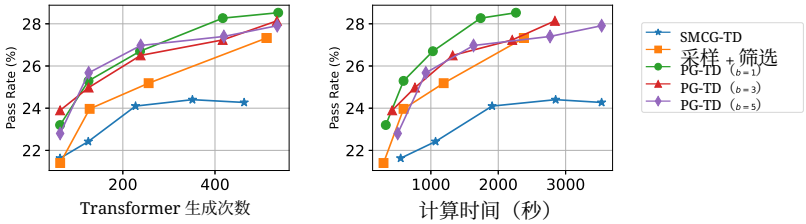


图 6：在 APPS 入门数据集上，使用 APPS GPT-2 Transformer 模型时 PG-TD ($c = 4$) 的结果。

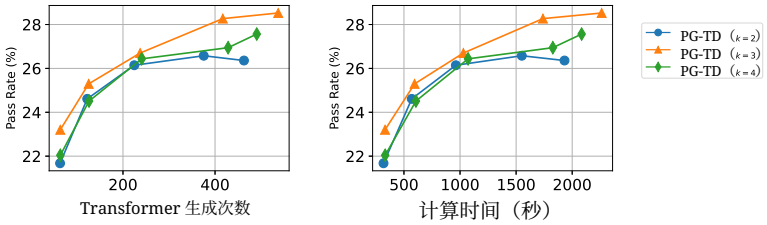


图 7：在 APPS 入门数据集上，使用 APPS GPT-2 Transformer 模型时 PG-TD ($c = 4$) 配合不同的 k 的结果。

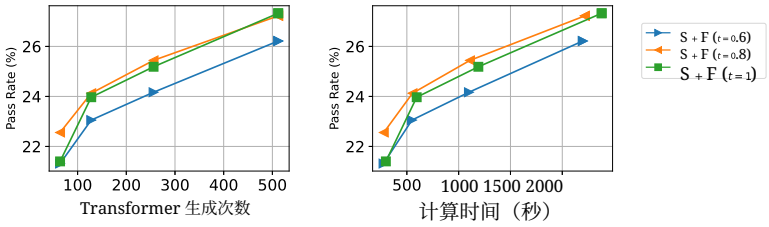


图 8：在 APPS 入门数据集上，使用 APPS GPT-2 Transformer 模型时，不同温度下的采样 + 筛选结果。

正如我们所预期的，PG-TD 能帮助 Codex 生成更好的代码（表8）。这进一步验证了我们的主张：我们的方法是模型无关的，能够帮助预训练代码生成 Transformer 生成更好的代码。受 OpenAI API 的限制，我们只能在部分数据上测试这些算法。在接下来的章节中，我们还将在图16中展示一个具体示例，说明 PG-TD 的表现优于束搜索。

使用基于采样的树搜索。 我们可以在 PG-TD 的评估步骤中使用采样方法，而不是束搜索，这样会更忠实地实现蒙特卡洛树搜索。我们对 PG-TD 的两个版本进行了实验，发现总体而言，使用束搜索的性能确实更好（表 9）。差距可能是因为我们只对每个节点采样一次来进行评估。通过多次采样，我们可以更准确地估计节点的价值。不过，这会增加算法的计算开销。

模型	解码算法	通过率（%）	严格准确率（%）
APPS GPT-2（1.5B）	束搜索	11.65	2
APPS GPT-2 (1.5B)	PG-TD	36.42	6
Codex (code-davinci-002)	束搜索	33.26	14
Codex (code-davinci-002)	PG-TD	59.14	31

表 8：比较使用不同 Transformer（APPS 和 Codex）作为代码生成骨干模型的效果，并在 APPS 的前 100 道入门问题上进行评估。

解码算法	通过率（%）			严格准确率（%）		
Transformer 生成次数	128	256	512	128	256	512
使用束搜索的 PG-TD	36.19	37.71	38.57	14.4	14.8	15.6
使用采样的 PG-TD	33.95	35.67	37.87	13.8	15.4	17.2

表 9：PG-TD 在评估步骤中使用束搜索与采样的比较，在 APPS 入门问题的前 500 道题上进行评估。

	编译错误（%）	运行时错误（%）
APPS GPT-2 (1.5B)，束搜索	5.58	32.95
APPS GPT-2 (1.5B)，采样 + 筛选	4.68	27.38
APPS GPT-2（15亿参数），PG-TD	1.93	19.5

Tab表10：失败案例分析。百分比是 APPS 入门数据集上的平均值 .

	严格准确率（%）
束搜索	26.82
PG-TD（使用自动生成的测试用例）	74.53

表11：使用 Codex 模型，在 HumanEval 数据集上评估束搜索和 TG-PD（使用自动生成的测试用例）的比较。

故障模式分析。在表10中，我们报告发生编译错误和运行时错误的测试用例的平均百分比。我们使用原始 APPS 代码库提供的错误分析工具。正如预期，PG-TD 会执行生成的程序，并筛选出通过率较高的程序，因此大幅降低了这两类错误的发生比例。

资源与许可证。我们的实验在配备两颗 Intel(R) Xeon(R) Gold 6258R CPU（@ 2.70GHz）和一块 V100-SXM2 GPU 的机器上运行。APPS 数据集（Hendrycks 等人，2021年）以 MIT 许可证发布 [2]。CodeContests 数据集（Li 等人，2022年）以 Apache 许可证 2.0 发布 [3]。Huggingface Transformer（Wolf 等人，2019年）以 Apache 许可证 2.0 发布 [4]。

B 使用自动生成的测试用例

当没有人工指定的测试用例时，我们的算法仍可依赖自动生成的测试用例。我们首先遵循 Chen 等人(2022年) 的做法，使用 Codex 模型，并在 HumanEval 数据集上评估代码生成算法（Chen 等人，2021a）。我们采用与 Chen 等人(2022年) 相同的流程来自动生成测试用例。具体而言，我们使用的提示中包含问题的自然语言描述，并在提示末尾附上一个 `assert` 语句。此外，我们移除了问题描述中的示例输入输出对，以避免

[2] https://github.com/hendrycks/apps/blob/main/LICENSE [3]
https://github.com/deepmind/code_contests/blob/main/LICENSE [4]
https://github.com/huggingface/transformers/blob/main/LICENSE

Transformer 直接复制问题描述中的测试用例。以下是一个用于生成测试用例的提示示例。

来自 typing 的 List

def has_close_elements(numbers: List[float], threshold: float) -> bool: """ 检查给定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。 """ pass

检查 has_close_elements 的正确性 def check(candidate): assert candidate

Codex 能生成正确的测试用例吗? 在 HumanEval 数据集中，生成的解答通过调用检查函数，由测试脚本进行评估。因此，我们使用严格准确率（pass@1）这一指标，统计生成的解答通过检查函数的问题所占的百分比。为了评估自动生成的测试用例的质量，我们计算示例解答（由人工程序员编写的正确解答）在这些测试用例上的严格准确率。显然，示例解答在真实 测试用例上的严格准确率应为 100%。我们在自动生成的测试用例上评估示例解答，发现相应的严格准确率为 72.56%，这证实自动生成的测试用例大多是正确的。

PG-TD 能利用自动生成的测试用例吗? 我们在表11中报告了生成的程序在 HumanEval 部分题目上的平均严格准确率。我们确认，PG-TD 的严格准确率更高，因为它使用高质量的自动生成测试用例来验证生成的程序。

C 面向其他代码生成目标的规划

除了优化通过率的默认奖励函数外，我们还通过设定两个新目标——代码长度惩罚和注释鼓励——展示了所提出算法的多功能性。

代码长度惩罚。我们通过使用以下奖励函数，让规划器生成更简洁的代码

$$\mathcal{R}_{\text{length}} = p + \lambda_1 \times e^{-l_c/t}, \tag{1}$$

其中 p 是公开测试用例集上的平均通过率， l_c 是代码字符串的长度。 λ和 t 是控制代码长度惩罚权重的超参数，分别设为0.1 和20。如图9所示，生成的解答更加简洁，代码字符串长度从 187 减少到 78，同时仍能通过所有测试用例。

注释鼓励。我们还可以生成包含更多注释的解决方案。我们通过使用以下奖励函数来实现这一目标

$$\mathcal{R}_{\text{comment}} = p + \lambda_1 \times e^{-l_c/t} + \lambda_2 \times \min(1, \frac{N_{cm}}{N_{max}}), \tag{2}$$

其中 N_{cm} 表示代码字符串中“#”的数量，而 λ_2 设为 0.2，用于控制注释行的权重。如果我们只将 $\lambda_2 \times N_{cm}$ 作为注释鼓励项，规划器就会生成包含重复且毫无意义的注释行的代码，以获得更多奖励。为了解决这个问题，我们加入代码长度惩罚项 $\lambda_1 \times e^{-l_c/t}$。此外，我们将注释行数量的奖励设定上限，令注释鼓励项为 $\lambda_2 \times \min(1, \frac{N_{cm}}{N_{max}})$，其中 $N_{max} = 5$对应图 10所示的示例。如图 10所示，使用这一设计的奖励函数，我们可以生成注释行更多的解。

问题陈述
Takahashi 喜欢回文。非回文字符串对他来说不可接受。每次拥抱一个字符串时，他都可以将其中一个字符改成自己选择的任意字符。给定一个字符串 S。求使 S 成为回文所需的最少拥抱次数。约束条件*(1).* S 是一个由小写英文字母组成的字符串。*(2).* S 的长度在1 到 100 之间（含端点）。输入输入以如下格式从标准输入给出：S。输出输出使 S 成为回文所需的最少拥抱次数。样例测试输入 ***1***$redcoder$样例测试输出 ***1***1例如，我们可以将第四个字符改成 o ，得到回文 *redooder*。

```
# 构造一个回文
s=输入()
n=长度(s)
c=0
如果n%
        对于 i在 范围(n//2):
                if s[i]! =s[n-i-1]:
                        c+=1
        打印(c)
否则:
        对于 i在 范围(n//2):
                if s[i]!=s[n-i-1]:
                        c+=1
        打印(c)
```

默认奖励函数（通过率：1.00，代码字符串长度：**187**）。

```
s=输入()
ans=0
对于i 在范围(长度(s)//2):
        if s[i]! =s[~i]: 答案+=1
打印(答案)
```

带代码长度惩罚的奖励函数（通过率：1.00，代码字符串长度：**78**）。

图9：一个带代码长度惩罚的代码生成示例。顶部显示问题描述。底部显示默认 PG-TD 和带代码长度惩罚的 PG-TD 生成的程序。通过代码长度惩罚奖励，我们可以生成更简洁的代码。

问题陈述$A + B$通常被用作最简单问题的示例，以展示某个竞赛平台。然而，一些科学家发现，有时这道题并不那么容易通过。想试试吗？输入输入包含两个整数 a 和 b $(0 \leq a, b \leq 10^3)$，二者之间以一个空格分隔。输出输出给定整数的和。样例输入 ***1***5 14样例输出 ***1***19样例输入 ***2***381 492样例输出 ***2***873

```
a, b = 映射(整数, 输入().split())
打印(a +b)

```

默认奖励函数。

```
#!/usrbin/env python3
# coding =utf-8 #py3 在 py2 上运行
a,b=映射(整数,输入().split())
打印(a+b)
```

带注释鼓励的奖励函数（带注释的代码）。

图 10：一个带有代码注释鼓励的代码生成示例。我们可以通过使用代码注释鼓励奖励来生成包含更多注释的代码。

D 算法详情

D.1 PG-TD

本节将更详细地介绍我们的 PG-TD 算法。我们在图 12中再次展示了我们的框架，以便于查阅。该算法维护一棵搜索树，其中节点对应状态，边对应动作。对于每个节点 s，算法都会记录其被访问的次数，记为 $s.visits$。对于一个状态-动作对 s， a （状态-动作对对应于树中的一条边），算法会维护一个函数 $Q(s,\ a)$，该函数表示从 s 出发并执行动作 a所能获得的最大奖励。通常，$Q(s,\ a)$ 是平均回报。由于我们的代码生成问题是确定性的（即不存在随机转移），我们发现，记录从某个节点生成的最佳程序的通过率能带来更好的性能。

算法首先**选择**一个节点进行扩展。它从根节点开始，递归地选择子树，直到到达一个此前尚未扩展过的节点（没有子节点）。一种常见的节点选择准则是上置信界（UCB）（Kocsis& Szepesv´ari，2006年），该准则在利用已知的优良状态与探索访问次数较少的状态之间进行权衡。我们改编了 Silver 等人(2017年中使用的 P-UCB 启发式（它是 UCB 的一种变体），如下所示：

$$\text{P-UCB}(s,a) = Q(s,a) + \beta(s) \cdot P_{\text{Transformer}}(a|s) \cdot \frac{\sqrt{\log(s.visits)}}{1+s'.visits}, \tag{3}$$

其中， s' 是通过在 s中采取动作 a 后确定性到达的状态； $P_{\text{Transformer}}(a|s)$是在给定部分程序 s的情况下，词元 a 作为下一个词元的概率，该概率由 Trans-former 模型提供。 β 是用于探索的权重，取决于对 s的访问次数，定义如下

$$\beta(s) = \log(\frac{s.visits + c_{base} + 1}{c_{base}}) + c. \tag{4}$$

P UCB选择 算法 1 中的函数返回 P-UCB 价值最高的动作，

$$\text{P_UCB_SELECT}(s,c) = \arg\max_a \text{P-UCB}(s,a). \tag{5}$$

注意， c 是公式 4中探索权重的参数。为使符号简洁，公式 5 中省略了它。

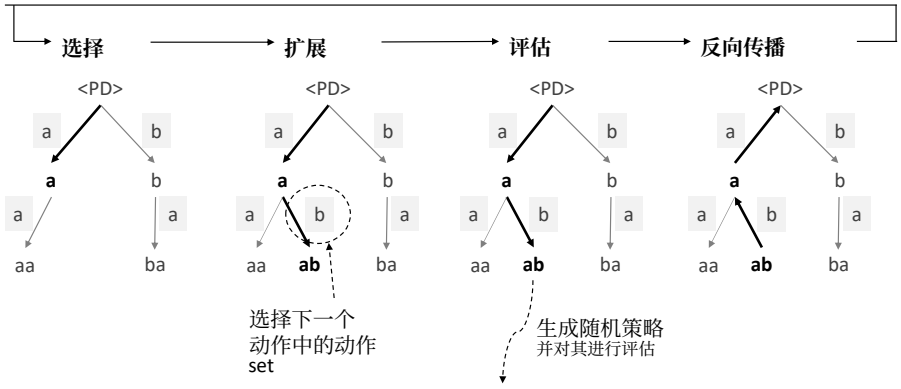
直观来说，如果以下情况成立，P UCB SELECT(s， c) 就会更频繁地返回动作 a： 1） $Q(s,\ a)$较大，这意味着它已从 s， a开始找到高质量的程序；或 2） $P_{\text{Transformer}}(a|s)$较大，这意味着 Transformer 认为 a 很可能是下一个词元；或 3） $s.visits$ 较大而 $s'.visits$ 较小，这意味着 s' 尚未得到充分探索。我们进行了超参数搜索，将$c_{base} = 10$设为相应值，并在实验中使用了 $c = 2$，4，6。

在接下来的步骤中，算法**通过将子节点添加到树中来扩展**该节点，并按照我们在正文中所述，使用束搜索**评估**该节点。最后，已完成程序 r 的奖励会**反向传播**到其父节点，递归进行，直至到达根节点，以更新它们的 Q 价值。对于树中从根节点到所选节点的轨迹上的所有状态-动作对， s'', a''，都会使用新的价值相应地更新 $Q(s'', a'')$: $Q(s'', a'') \leftarrow \max(Q(s'', a''),\ r)$。

与标准蒙特卡洛树搜索算法的比较。值得注意的是，我们在 PG-TD 中对蒙特卡洛树搜索的实现不同于标准蒙特卡洛树搜索算法。如果使用标准蒙特卡洛树搜索算法，在庞大的可能程序搜索空间中寻找高质量程序，将会在计算上难以实现。

为便于比较，我们在图中展示了标准的蒙特卡洛树搜索算法。11。在扩展步骤中，蒙特卡洛树搜索会选择动作集中的下一个可用动作，并将沿该动作执行后可到达的状态添加到树中。在示例中，采取了动作“b”，并将新状态“ab”添加到树中。在评估步骤中，蒙特卡洛树搜索从新状态开始执行随机策略，以评估该状态并计算该策略的价值。

将标准蒙特卡洛树搜索算法应用于状态空间或动作空间较大的领域并不现实，因为我们无法承担在扩展步骤中尝试所有可能动作的开销。在评估步骤中，使用随机策略估算新状态的价值也会导致较高的方差。在 PG-TD 中，我们使用预训练 Transformer 来克服标准蒙特卡洛树搜索算法的这些局限。



Fig图 11：标准蒙特卡洛树搜索算法，不使用预训练的 Trans
former. <问题描述> 表示问题描述。

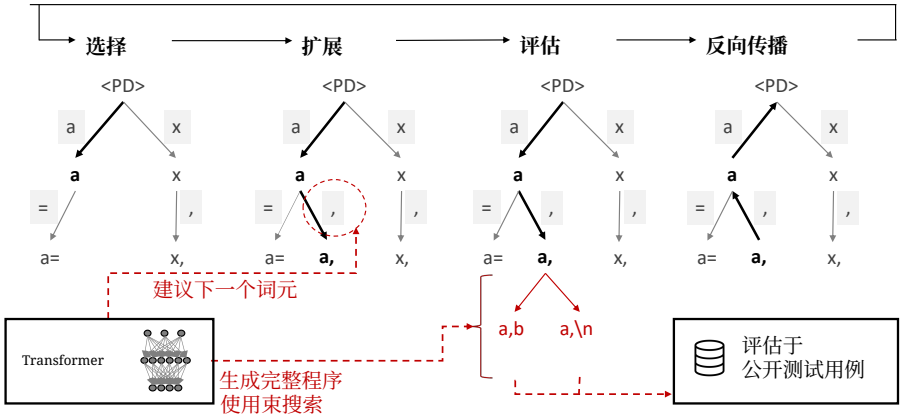


图12：为方便起见，此处再次展示我们提出的 PG-TD 框架。

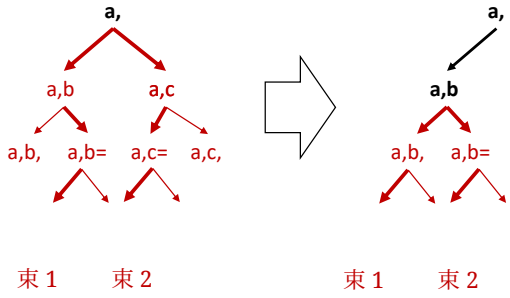


图13：示例说明：当 $b > 1$时，序列缓存可能无法返回正确的序列。

D.2 通过信息共享实现高效实现

正如正文中所讨论的，调用 Transformer 生成函数时可能会出现重复计算。我们已在正文中介绍树结构缓存。本节将进一步介绍序列缓存。

算法 2采样 + 筛选。

要求： $sample\ num$：使用 Transformer 生成的样本数量。

1: **对于** $i \leftarrow 1, 2, \ldots, sample\ nums$ **执行**
2: $p_i \leftarrow$ TRANSFORMER 示例($\langle PD \rangle$, $k = 5$)
3: $rewards[i] \leftarrow$ 获取奖励(p_i)
4: **结束循环**
5: **返回** p_i 最大的 $rewards[i]$

序列缓存。 序列缓存的目标是降低 PG-TD 评估步骤的计算成本。在评估步骤中，会调用 Transformer 束搜索算法，从当前节点生成一个完整程序。这些完整程序及其奖励都会被缓存，并在后续迭代中使用。

然而，当 $b > 1$，在之前的迭代中使用缓存序列可能并非最优。我们来看图 13中的例子，其中 $b = 2$。从“a,”开始（左图），假设束搜索算法分别通过搜索前缀为“a,b”和“a,c”的两个束，返回一个序列（粗体线条）。在下一次迭代中（右图），要使用束搜索算法评估“a,b”时，我们可能不会使用上一次迭代中找到的序列，因为该序列仅用一个束搜索“a,b”的子树。换句话说，从“a,b”开始，使用 $b = 2$重新运行束搜索算法，可能会找到似然更高的序列。

虽然在 PG-TD 的评估步骤中，使用 $b > 1$ 进行序列缓存可能会低估程序的价值，但出于节省计算量的考虑，我们仍对所有 b 选择使用序列缓存。

D.3 基线算法

采样 +筛选。我们已在正文中介绍采样算法。伪代码见算法2。

序贯蒙特卡洛引导的 Transformer 解码。序贯蒙特卡洛引导的 Transformer 解码（SMCG-TD）是另一种与 PG-TD 对比的算法，它同样使用 Transformer 评估部分程序。伪代码见算法3。可以从遗传算法的角度来理解它。SMCG-TD 维护一组部分程序，称为代，并对其进行迭代更新。对于该代中的每个部分程序，我们使用 Transformer 采样下一个词元，并将其追加到部分程序中，然后将其放入新一代。接着，我们以与 PG-TD 相同的方式评估新一代中的每个部分程序：使用 Transformer 根据部分程序生成完整程序，并计算其奖励（通过率）。然后，我们将这些奖励作为部分程序的适应度分数。每次迭代结束时，我们根据适应度对部分程序重新采样。因此，适应度分数较高的部分程序更有可能留在这一代中。最后，算法输出所找到的奖励最大的完整程序。

可以看出，从更高层面来看，SMCG-TD 与 PG-TD 相似。在代码生成过程中，它也会利用 Transformer 和公开测试用例来决定要生成的下一个词元。关键区别在于，SMCG-TD 采用基于采样的方法，而 PG-TD 采用树搜索算法。因此，在我们的实验中，SMCG-TD 的表现不如 PG-TD。

值得注意的是，SMCG-TD 调用束搜索时也实现了序列缓存。因此，即使一次生成过程中的多个部分程序可能相同，该算法也不会针对同一前缀多次调用 Transformer 来生成完整程序，这有效减少了调用 Transformer 的次数，并节省了计算时间。

D.4 微调

在我们的实验中，我们使用 PG-TD 生成的样本进一步微调 Transformer 模型。随后，我们通过运行束搜索，使用微调后的 Transformer 模型生成解答，

算法 3序贯蒙特卡洛引导的 Transformer 解码（SMCG-TD）。

要求： $pop\ size$：种群规模

$generation \leftarrow [\langle PD \rangle] * pop\ size$ # 使用问题描述初始化种群
$fitness \leftarrow [1.0/pop\ size] * pop\ size$ # 均匀分布
$t \leftarrow 0$
$complete\ programs \leftarrow$ 新建列表()
当 $|population| > 0$ 且 $t < max\ steps$ **时执行**
  $new_generation \leftarrow$ 新建列表()
  $new_fitness \leftarrow$ 新建列表()
  对于 $i \leftarrow 1, \ldots,$ LEN$(generation)$ **执行**
    $s \leftarrow$ 从 $generation$中采样，其中 $generation[j]$以.....的概率选中 $fitness[j]$
    $a \leftarrow$ 从 $P_{Transformer}(\cdot|s)$ 采样
    $s' \leftarrow$ 拼接$(s,\ a)$
    如果 $s'[-1] =$ <|文本结束|> **则**
      # 此样本结束生成，并保存到输出
      将 s' 添加到 $complete\ programs$
    else
      将 s' 添加到 $new\ generation$
      # 束搜索使用与 PG-TD 相同的序列缓存
      $p \leftarrow$ 束搜索$(s',\ b = 1)$
      $r \leftarrow$获取奖励(p)
      将 r 添加到 $new\ fitness$
    结束条件判断
  结束循环
  规范化 $new\ fitness$
  $generation \leftarrow new_generation$
  $fitness \leftarrow new_fitness$
  $t \leftarrow t + 1$
结束循环
返回 $complete\ programs$ 中奖励最大的程序

并观察到，与使用原始版本模型运行束搜索相比，通过率和严格准确率均有所提升。

我们设计中的一个挑战是，PG-TD 生成的解决方案具有不同的通过率。我们可以只使用通过率为 100% 的解决方案，但这会大幅限制可用于微调的样本数量。因此，我们收集所有通过率高于 80% 的解决方案。为了让 Transformer 了解训练样本各自不同的通过率，我们采用了类似于Le等人(2022年）的方法，

$$\mathcal{L}(\theta) = -\mathbb{E}_{W \sim p_\theta}[r(W) \nabla_\theta \log p_\theta(W|PD)], \tag{6}$$

其中，W表示一个完整的程序；r（W）是该程序的通过率；PD是程序描述。在未来的工作中，我们将研究不同的损失函数，并比较它们在提升 Transformer 模型性能方面的效果。

E 示例说明

在本节中，我们提供了更多由 PG-TD 和基线方法生成的定性示例，如图 14 和图15所示。PG-TD 算法生成的解具有比基线方法更高的通过率。我们还展示了一个具体示例，说明 PG-TD 如何在图 16 中优于束搜索，其中使用 Codex 作为骨干 Transformer。

如需了解 PG-TD 的逐步演示，请访问我们的项目网站：https://codeaimcts.github.io。

问题陈述

Takahashi 要和 *Aoki* 会面。他们计划在从现在起 *T* 分钟后，于距离 *Takahashi* 家 *D* 米的地方见面。*Takahashi* 现在会从家里出发，以每分钟 *S* 米的速度直线前往约定地点。他能及时赶到吗？约束条件*(1).* $1 \leq D \leq 10000$*(2).* $1 \leq T \leq 10000$*(3).* $1 \leq S \leq 10000$*(4).* 输入中的所有值均为整数。输入输入按以下格式从标准输入给出： $D\ T\ S$输出如果 *Takahashi* 能及时到达约定地点，输出 Yes；否则，输出 No。样例测试输入100015 80样例测试输出*Yes* 以每分钟 *80* 米的速度走 *1000* 米需要 *12.5* 分钟。他们计划在 *15* 分钟后见面，因此 *Takahashi* 能及时赶到。

```
# 在此烹饪你的菜肴
try:
    a=列表(映射(整数,
    ↪    输入().拆分(
    ↪    )))
    如果(a[2]==1):
        输出("Yes")
    否则:
        输出("No")
异常:
    pass
```

束搜索（通过率：0.125）。

```
# 烹饪你的菜肴
d,t,s=列表(映射(整数,
↪    输入().拆分
↪    ()))
如果(d==t):
    如果(s*(s+1)/s==t):
        打印(“是”)
    否则:
        打印("否")
否则:
    如果(s*(s+1)/s>t):
        打印("是")
    否则:
        打印("否")
```

采样 + 筛选（通过率：0.625）。

```
n,t,s=映射(整数,
↪    输入()。分割())
t1=n / s
如果t1 <= t:
    打印("是")
否则:
    打印("否")
```

PG-TD（通过率：1.00）。

图14：竞赛编程的代码生成示例。顶部显示问题描述。底部显示基线算法和我们的 PG-TD 算法生成的程序。

问题陈述

给定一个整数 r。半径为 r 的圆的面积是半径为 1 的圆的面积的多少倍？可以证明，在给定的约束条件下，答案始终是一个整数。

约束条件*(1).* $1 \leq r \leq 100$*. (2).* 输入中的所有价值都是整数。输入输入按照以下格式从标准输入给出： r*.*输出输出半径为 r的圆的面积除以半径为1的圆的面积所得的值，并将其作为整数输出*..*样例测试输入2样例测试输出4半径为2 的圆的面积是半径为1的圆的面积的4 倍。请注意，输出必须是整数——例如，4.0 将不被接受。

```
# 在这里编写你的菜肴
r=int(输入())
如果(r>=2*r):
        打印(r*r*2)
否则:
        打印(0)
```

束搜索（通过率：0.00）。

```
# 在此编写你的解题代码
r=int(输入())
如果(r>=2*r):
        打印(r*r*2)
否则:
        打印(0)
```

采样 + 筛选（通过率：0.00）。

```
# 在此编写你的菜品代码
r=int(输入())
打印(2*(r**2)-r**2)
```

PG-TD（通过率：1.00）。

图15：一个用于竞赛编程的代码生成示例。问题描述显示在上方。基线算法和我们的 PG-TD 算法生成的程序显示在下方。

问题陈述
给定一个由 n 个整数构成的数组 a 。
将数组的不稳定度定义为以下价值：^{n}最大值$_{i=1}$ $a_i -$ $\underset{\text{最小值}_{i=1}}{^{n}} a_i$.

你必须从该数组中恰好删除一个元素，使所得$(n-1)$个元素的数组的不稳定度最小。你的任务是计算可能的最小不稳定度。输入输入的第一行包含一个整数 n *(*$2 \leq n \leq 10^5$*)* — 数组 a 中的元素数量。输入的第二行包含 n 个整数 $a_1, a_2, \ldots, a_n$*(*$1 \leq a_i \leq 10^5$*)* — 数组 a 的元素。输出输出一个整数 — 如果必须从数组中恰好删除一个元素，数组可能达到的最小不稳定度 a。示例输入 *4 1 3 3 7* 输出 *2* 输入 *2 1 100000* 输出 *0*注释在第一个示例中，你可以删除 7，这样剩余数组的不稳定度将为 $3-1=2$。在第二个示例中，你可以删除 1 或 100000，这样剩余数组的不稳定度将分别为 $100000-100000=0$ 和 $1-1=0$ 。

```
n = 整数(输入())
a = 列表(映射(整数, 输入()。拆分()))

打印(最大值(a) - 最小值(a) - 1)
```

束搜索（通过率 0.00）。

```
定义 主函数():
    n = 整数(输入())
    一个 = 列表(映射(整数, 输入().拆分()))
    a.sort()
    打印(最小值(a[n-1] - a[1], a[n-2] - a[0]))

if __name__ == "__main__":
    main()
```

PG-TD（通过率 1.00）。

图16：使用 Codex（code-davinci-002）的代码生成示例。束搜索找到一个错误的解，而 PG-TD 找到了正确的解。该结果进一步证实，PG-TD 与模型无关，可用于不同的骨干 Transformer（GPT-2（1.5B）、GPT-Neo（2.7B）和 Codex）。

F 更多讨论

潜在优势。我们已经证明，与仅使用 Transformer 生成过程相比，我们的框架不仅有助于提高通过率，还能生成可优化不同目标的代码。这些成果无需对预训练 Transformer 模型进行微调即可实现。通过为规划算法设计不同的奖励函数，我们的框架让预训练 Transformer 更加灵活，并能适应不同任务。如果我们重新训练或微调本文中使用的 GPT-2 等 Transformer 模型，就需要训练数十亿个参数（Radford 等人，2019年）。我们的方法有望节省针对不同任务训练或微调 Transformer 所需的计算成本和能源消耗。

潜在的负面社会影响。 自动代码生成使任何人都更容易生成符合规范的程序。我们希望这一发展能够减轻软件工程师的负担，并实现 AI 辅助编程，甚至端到端的自动编程。然而，这也可能使开发恶意软件变得更容易。任何能够用自然语言描述恶意目标的人，都可以使用自动代码生成工具生成符合该目标的代码。当自动代码生成技术日趋成熟时，我们可能需要一个独立模块来筛查自然语言描述，并拒绝可能导致生成有害代码的请求。