Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

GUIDING PROGRAM SYNTHESIS WITH MONTE CARLO TREE SEARCH

Gongbo Zhang[1,2,3], and Yijie Peng[1,2,3]

[1]Guanghua School of Management, Peking University, Beijing, CHINA
[2]PKU-Wuhan Institute for Artificial Intelligence, Peking University, Wuhan, CHINA
[3]Xiangjiang Laboratory, Changsha, CHINA

ABSTRACT

Genetic Programming (GP) is a powerful evolutionary computation technique for automatic program synthesis, but it suffers from inefficient search and solution bloat. This paper proposes using Monte Carlo Tree Search (MCTS) to directly construct programs by formulating synthesis as a sequential decision problem modeled as a Markov Decision Process. We tailor the MCTS algorithm to the GP domain and demonstrate its theoretical consistency. Empirical evaluation on symbolic regression benchmarks shows that our approach consistently outperforms traditional GP, discovering higher-quality solutions with more compact structural complexity and suggesting that MCTS is a promising alternative for automatic program generation.

1 INTRODUCTION

Genetic Programming (GP) stands as a powerful evolutionary computation technique designed to automatically synthesize computer programs to solve complex problems (Koza 1982; Ahvanooey et al. 2019; O'Neill et al. 2010). Unlike traditional genetic algorithms, which manipulate fixed-length strings, GP evolves a population of hierarchical, variable-sized tree structures that represent executable programs or mathematical expressions. Its evolutionary cycle mimics biological evolution: promising programs are selected based on their fitness and then recombined and mutated to generate new variants. GP has achieved notable success in tasks such as symbolic regression, classification, feature selection, automated design, and other machine learning applications. The primary appeal of GP lies in its ability to discover novel program structures without requiring explicit instructions for solving the target problem. Nevertheless, GP faces several challenges in complex real-world tasks, including inefficient exploration of large search spaces, premature convergence on suboptimal solutions, and excessive computational costs due to code bloat. To improve the efficiency and effectiveness of program synthesis, we propose an alternative to the traditional evolutionary framework of GP by formulating program synthesis as a sequential decision-making process navigated through Monte Carlo Tree Search (MCTS), enabling effective exploration of the large and complex program landscape.

The fundamental mechanism of GP follows a structured evolutionary process, beginning with a randomly initialized population of program trees. Each individual is evaluated using a fitness function that measures the solution quality for the target problem. The algorithm then applies selection pressure to favor promising programs, followed by genetic operators—crossover, which exchanges subtrees between parent programs; mutation, which randomly alters nodes or subtrees; and reproduction, which directly copies high-fitness individuals—to generate offspring for the next generation. This iterative process continues until a termination criterion is met, typically based on solution quality or computational budget. GP has been successfully applied across numerous domains. For example, in machine learning, GP has demonstrated competitive performance in classification (Espejo et al. 2009), particularly for imbalanced datasets (Bhowan et al. 2012), as well as in feature construction (Muñoz et al. 2015) and ensemble learning (Folino, Pisani, and Sabatino 2016). In optimization, GP has contributed to scheduling problems, including job shop scheduling (Nguyen

et al. 2012), tower crane scheduling (Yin et al. 2024), project scheduling (Li et al. 2024), and cloud workflow scheduling (Zaki et al. 2024). GP has even enabled scientific discovery; for instance, Schmidt and Lipson (2009) use it to automatically identify physical laws and invariant equations from experimental data. Despite these successes, GP faces significant challenges that limit its broader applicability. The exponential growth of the search space with increasing program complexity makes exhaustive search computationally infeasible. This large search space, combined with deceptive fitness landscapes—where beneficial building blocks may initially appear unpromising—makes efficient exploration difficult. Luke and Panait (2006) identify bloat–the uncontrolled growth of program size without corresponding fitness improvements–as a factor that reduces both computational efficiency and solution interpretability. Moreover, Vanneschi and Poli (2012) demonstrate that genetic operators can effectively recombine and modify existing solutions but provide minimal guidance toward promising, unexplored regions, especially when beneficial program structures require coordinated changes across multiple components. These exploration–exploitation challenges have motivated research into alternative search paradigms that can more effectively navigate complex program spaces.

Monte Carlo Tree Search (MCTS) offers a compelling alternative for systematically exploring large decision spaces through strategic sampling and guided exploration. Originally formalized by Kocsis and Szepesvári (2006) through the Upper Confidence Bounds applied to Trees (UCT) algorithm, MCTS adaptively balances the exploration of unknown states with the exploitation of promising actions by building an asymmetric search tree that focuses computational resources on high-potential regions while still allocating some effort to less-explored areas. This balance is achieved through four key phases: selection of nodes based on exploration-exploitation criteria, expansion of the search tree, simulation to evaluate potential outcomes, and backpropagation of results to update node statistics. Browne et al. (2012) provide a comprehensive survey of MCTS methods, highlighting its applicability beyond games to various optimization problems. MCTS achieves landmark success when integrated into AlphaGo, where it is combined with deep neural networks to demonstrate superhuman performance in the complex game of Go (Silver et al. 2016). More recently, MCTS has played an important role in Chain-of-Thought reasoning for large language models (LLMs). In the simulation and optimization area, sampling procedures developed for the ranking and selection (R&S) problem have been shown to improve the efficiency of MCTS by enabling more effective selection strategies within the algorithm (Li et al. 2021; Zhang et al. 2022; Liu et al. 2024). The strength of MCTS lies precisely in its ability to address the exploration–exploitation dilemma that challenges traditional GP. By systematically tracking the potential of different decision paths and allocating computational resources accordingly, MCTS can efficiently guide search through complex spaces where the value of intermediate states is difficult to assess–exactly the challenge GP faces when evolving programs through rugged fitness landscapes. A notable related work has successfully applied MCTS to design improved mutation operators for GP, demonstrating the potential benefits of incorporating MCTS principles into the GP framework (Islam et al. 2020). However, that work focuses on a specific genetic operator, whereas our research aims to investigate the feasibility and effectiveness of using MCTS to guide the entire program evolution process, essentially casting the search for a program as a tree search problem within the MCTS framework. This fundamental difference in scope and application distinguishes our work from existing literature, positioning MCTS as a potential alternative or complementary approach to the traditional genetic operators for exploring the GP search space.

Building on the promising potential of MCTS for program synthesis, this work employs MCTS as the primary mechanism for solving GP problems. Instead of evolving a population over generations, we directly construct program trees through systematic exploration of the program space. We formulate the GP problem as a sequential decision-making process, where each decision involves extending or modifying a program structure, guided by the outcomes of MCTS simulations. The sequential nature of our approach allows for more directed program construction: each node in the MCTS tree represents a partial program, each action corresponds to adding a component (function or terminal), and each path from root to leaf represents a complete program construction sequence. By leveraging the inherent strengths

of MCTS in balancing exploration and exploitation, our approach systematically searches the program space, concentrating computational effort on promising regions while still exploring potentially valuable alternatives. We evaluate our method on two simple benchmark problems, and the results provide empirical evidence of the effectiveness of MCTS for program synthesis, demonstrating its strong potential and warranting further investigation into improving efficiency by incorporating deep neural networks. The key contributions of this work include: a framework that recasts program synthesis as a tree search problem; practical implementations of MCTS in program spaces; and experimental findings that advance an alternative search paradigm for automatic program generation.

The remainder of this paper is organized as follows. Section 2 formulates the problem of program synthesis as a sequential decision-making process, introducing the necessary mathematical framework and notation. Section 3 provides an overview of our proposed MCTS-based algorithm for program synthesis, detailing the selection, expansion, simulation, and backpropagation phases as adapted to the GP domain. Section 4 presents the experimental setup for the empirical validation of the proposed algorithm and discusses the results. The paper concludes with a summary of the findings and directions for future research.

2 PROBLEM FORMULATION

In this section, we first provide a general formulation of GP problems, then discuss heuristics for guiding the search process, especially those that can be integrated into the MCTS framework, and finally formulate the GP problem as a Markov Decision Process (MDP) to enable the application of MCTS.

2.1 Genetic Programming Fundamentals

GP evolves computer programs to solve problems automatically by mimicking the process of natural evolution. Formally, GP operates on a space defined by two essential sets: a function set $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$, where each function f_i has an associated arity $a_i \in \mathbb{N}$ representing the number of arguments it requires (e.g., arithmetic operators such as $+, -, \times, /$; mathematical functions such as $\sin, \cos, \log$; conditional operators like if-then-else; or domain-specific functions), and a terminal set $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ consisting of variables and constants that serve as leaf nodes in the program trees. Together, $\mathcal{F}$ and $\mathcal{T}$ form the primitive set $\mathcal{P} = \mathcal{F} \cup \mathcal{T}$ from which programs are constructed. A GP individual p is represented as a syntax tree, with internal nodes drawn from $\mathcal{F}$ and leaf nodes from $\mathcal{T}$, where each function node has exactly as many child nodes as its arity requires. The search space Ω comprises all possible trees constructible from $\mathcal{P}$ up to a specified maximum depth $D_{\max}$, which is used to control program complexity. This search space grows exponentially with increasing depth and with the cardinalities of $\mathcal{F}$ and $\mathcal{T}$, making exhaustive search computationally infeasible for most problems of interest.

For a specific problem domain, each program $p \in \Omega$ is evaluated using a fitness function $\phi : \Omega \to \mathbb{R}$ that quantifies how well the program solves the target problem. For example, in symbolic regression, the fitness is typically measured as the mean squared error between the outputs of the program and the target values across a training dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$, i.e., $f(p) = \frac{1}{N}\sum_{i=1}^N (\phi(\mathbf{x}_i) - y_i)^2$.

The GP optimization problem is to find a program $p^* \in \Omega$ such that $p^* = \arg\min_{p\in\Omega} \phi(p)$ (or $\arg\max$ for maximization problems). The traditional GP algorithm addresses this optimization problem through an iterative evolutionary process operating on a population $P = \{p_1, p_2, \ldots, p_\mu\}$ of candidate programs. The process typically begins with the random initialization of programs, denoted as $\mathcal{P}_0$. After evaluating $\phi(p)$ for each $p \in \mathcal{P}_t$ at generation t, selection mechanisms probabilistically favor higher-fitness programs to serve as parents for creating the next generation $\mathcal{P}_{t+1}$. Variation operators—crossover (which exchanges subtrees between parent programs), mutation (which randomly alters subtrees within programs), and reproduction (which directly copies high-fitness individuals)—generate offspring for the subsequent population. The process continues until a termination criterion is met, such as reaching a maximum number of generations or discovering a solution with satisfactory fitness. Figure 1 illustrates the tree structure of a GP individual.

Distinguishing properties of GP include its variable-length representation, which allows the structure and size of solutions to evolve; its hierarchical nature, which enables complex functional compositions; the closure property, which ensures that any subtree of a valid program is itself a valid program; and the sufficiency property, which requires that the primitive set be capable of expressing solutions to the target problem. These properties make GP powerful for problems where the solution structure is unknown a priori, but they also contribute to challenges such as bloat, premature convergence, and difficulty navigating rugged fitness landscapes.

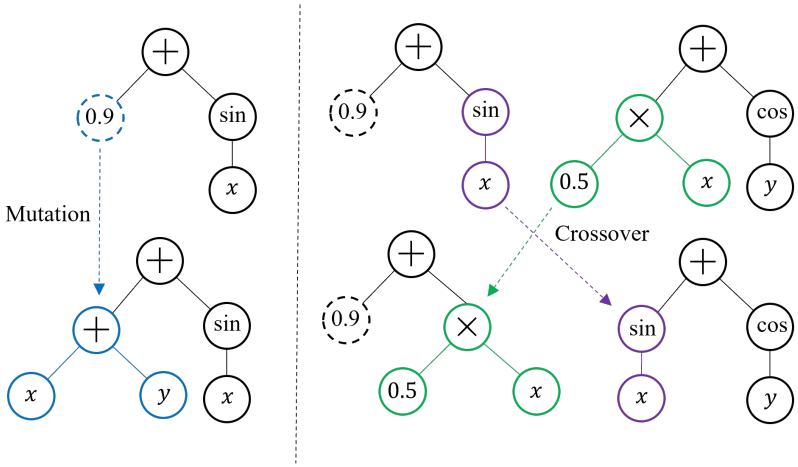


Figure 1: An example of a GP function represented as a tree structure.

2.2 Markov Decision Process Formulation

To apply MCTS to GP, we formulate program synthesis as a Markov Decision Process (MDP), enabling sequential decision-making over the program space. An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathbb{P}, \mathcal{R})$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathbb{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ is the transition function, and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function. Below, we define these components for general GP problems.

The state space $\mathcal{S}$ comprises all possible partial programs that can be constructed from the primitive set $\mathcal{P} = \mathcal{F} \cup \mathcal{T}$. Formally, a state $s \in \mathcal{S}$ represents a partially constructed program tree, denoted as $s = (T_s, L_s)$, where T_s is the current tree with nodes from $\mathcal{P} \cup \{\square\}$ (where $\square$ denotes an unfilled node or placeholder), and L_s is an ordered list of unfilled nodes that still need to be expanded. The initial state is defined as $s_0 = (\{\square\}, [\square])$, consisting of a single unfilled root node. Terminal states are those for which $L_s = \emptyset$, indicating that all nodes have been filled and the program is complete. This state representation satisfies the Markov property, as future expansions depend solely on the current partial tree structure and the remaining unfilled nodes, not on the sequence of decisions that led to the current state.

For a state $s = (T_s, L_s)$, the action space $\mathcal{A}(s)$ comprises all possible ways to expand the first unfilled node in L_s. An action $a \in \mathcal{A}(s)$ selects a function or terminal from $\mathcal{P}$ to replace this unfilled node. The available actions are constrained by the maximum depth D_{max} and any type constraints. For example, if expanding a node that requires an argument for a function $f \in F$, the selected action must insert a subtree rooted at a node whose return type matches the expected argument type for f. The action space is defined as:

$$\mathcal{A}(s) = \begin{cases} \{e \in \mathcal{F} \cup \mathcal{T} \mid T(e) \text{ is compatible with } RT(l_1)\} & \text{if } d(l_1) < D_{max} \\ \{e \in \mathcal{T} \mid T(e) \text{ is compatible with } RT(l_1)\} & \text{if } d(l_1) = D_{max} \end{cases},$$

where l_1 denotes the first unfilled node in L_s, $T(\cdot)$ represents the return type of an element, and $RT(\cdot)$ denotes the required return type at a given position, based on the parent function. This formulation naturally incorporates syntax constraints, ensuring that only valid programs are constructed.

The transition function $\mathbb{P}(s'|s,a)$ in our MDP is deterministic; that is, taking action a in state s leads to exactly one next state s' with probability 1 (and probability 0 for all other states). Given a state $s = (T_s, L_s)$ and an action $a \in \mathcal{A}(s)$ selecting an element $e \in \mathcal{F} \cup \mathcal{T}$, the next state $s' = (T_{s'}, L_{s'})$ is determined as follows:

$$L_{s'} = [l_2, l_3, \ldots, l_{|L_s|}] \cup [\square_1, \square_2, \ldots, \square_{\text{arity}(e)}]$$
$$T_{s'} = T_s \text{ with } l_1 \text{ replaced by } e \text{ and children } \{\square_1, \square_2, \ldots, \square_{\text{arity}(e)}\} \text{ added if } e \in \mathcal{F} \ ,$$

where, if $e \in \mathcal{F}$ with $\text{arity}(e) > 0$, the action creates $\text{arity}(e)$ new unfilled nodes as children of the newly filled node. If $e \in \mathcal{T}$, no new unfilled nodes are created. For example, consider a state s with tree $T_s = (+, x, \square)$ and unfilled node list $L_s = [\square]$, where $\square$ is the second argument of the $+$ operator. If action a selects $e = y \in \mathcal{T}$, the next state becomes $T_{s'} = (+, x, y)$ and $L_{s'} = []$ (an empty list, indicating a complete program). If instead action a selects $e = \times \in \mathcal{F}$, then the next state becomes $T_{s'} = (+, x, (\times, \square, \square))$ and $L_{s'} = [\square, \square]$ (two new unfilled nodes). This deterministic transition reflects the constructive nature of program building, where each action incrementally extends the program in a well-defined manner.

The reward function $\mathcal{R}(s,a,s')$ provides feedback on the quality of actions. Unlike traditional GP, where fitness is evaluated for complete individuals in each generation, our MDP employs a sparse reward structure that concentrates feedback at the completion of program construction:

$$\mathcal{R}(s,a,s') = \begin{cases} g(\phi(s')) & \text{if } s' \text{ is a terminal state (complete program)} \\ 0 & \text{otherwise} \end{cases} \ ,$$

where $\phi(s')$ is the fitness of the complete program represented by the terminal state s', and $g(\cdot)$ transforms the fitness into a reward signal suitable for maximization in the MCTS framework. For example, in symbolic regression tasks involving error minimization, if $s' = (+, x, 1)$ is a terminal state, then the reward is computed as: $\mathcal{R}(s,a,s') = -\frac{1}{n}\sum_{i=1}^{n}((x_i+1) - y_i)^2$. This sparse reward structure aligns with GP's focus on evaluating complete programs rather than partial solutions, addressing the challenge of meaningfully assessing incomplete programs.

The objective of an MDP is to find an optimal policy $\pi^* : \mathcal{S} \to \mathcal{A}$ that maximizes the expected cumulative reward: $\pi^* = \arg\max_\pi \mathbb{E}_\pi \left[\sum_{t=0}^{T} \gamma^t \mathcal{R}(s_t, \pi(s_t), s_{t+1})\right]$, where T is the time step at which a terminal state is reached. In our sparse reward setting with $\gamma = 1$, this simplifies to finding a policy that maximizes the expected terminal reward, $\mathbb{E}_\pi[g(\phi(s_T))]$, which is equivalent to finding the program with optimal fitness. The proposed MDP formulation builds a bridge between evolutionary program synthesis and sequential decision-making, enabling the application of MCTS to program exploration. By recasting GP as a sequential construction process rather than a parallel population-based evolution, we leverage the strengths of MCTS in handling large state spaces with delayed rewards–precisely the challenges that traditional GP faces in complex program synthesis tasks.

3 ADAPTING MCTS FOR PROGRAM SYNTHESIS

In this section, we present our approach for adapting MCTS to the domain of program synthesis. We provide an overview of how the canonical MCTS algorithm can be tailored to navigate the program space, as defined by our MDP formulation.

MCTS is a best-first search algorithm that builds an asymmetric search tree through repeated simulations, gradually focusing computational effort on the most promising regions of the search space. The algorithm treats program construction as a sequential decision process, where each node in the tree corresponds to a state representing a partial program, and each edge represents an action that adds a function or terminal to the evolving program tree. It iteratively refines this tree to identify a complete program that minimizes the fitness function. By integrating the program-tree representation of GP into the MCTS framework, we adapt each of the four canonical MCTS phases–selection, expansion, simulation, and backpropagation–to navigate the program space defined by our MDP formulation.

Selection Phase: The selection phase traverses the current search tree from the root node (representing the current decision-making state) to a leaf node using a selection policy that balances exploration and exploitation. To address the specific challenges of program synthesis, particularly the tendency toward bloat, we modify the original UCT formula as follows:

$$a^* = \arg\max_{a \in \mathcal{A}(s)} \left\{ \overline{Q}(s,a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}} - \lambda \cdot \text{complexity}(s,a) \right\} , \tag{1}$$

where $\overline{Q}(s,a)$ is the estimated value of taking action a from state s, $N(s)$ is the visit count for state s, $N(s,a)$ is the visit count for the state-action pair (s,a), and $c > 0$ is an exploration constant (typically set to $\sqrt{2}$) that controls the trade-off between exploration and exploitation. The term complexity(s,a) quantifies the structural complexity introduced by action a, and $\lambda \geq 0$ is a parameter controlling the strength of this complexity penalty. In particular, the complexity of an action a that selects an element $e \in \mathcal{F} \cup \mathcal{T}$ is assessed by:

$$\text{complexity}(s,a) = \alpha \cdot \text{node\_count}(s,a) + \beta \cdot \text{depth\_impact}(s,a) ,$$

where α and β are weighting parameters. The term node_count(s,a) measures the immediate increase in program size:

$$\text{node\_count}(s,a) = \begin{cases} \text{arity}(e) + 1 & \text{if } e \in \mathcal{F} \\ 1 & \text{if } e \in \mathcal{T} \end{cases} ,$$

and depth_impact$(s,a) = \frac{d(l_1)}{D_{\max}}$ measures how close the action brings the program to the maximum allowed depth. The selection process proceeds recursively from the root node, selecting actions according to the modified UCT criterion until it reaches a leaf node s_L, which is either a terminal state (i.e., a complete program) or a non-terminal state with unexplored actions.

Expansion Phase: Upon reaching a leaf node s_L that is not a terminal state, the expansion phase extends the search tree by adding one or more child nodes. Typically, an untried action $a \in \mathcal{A}(s_L)$ is selected, and the resulting state s' is added to the tree as a child of s_L. The new node s' is initialized with statistics $N(s',a) = 0$ and $\overline{Q}(s',a) = 0$ for all valid actions $a \in \mathcal{A}(s')$.

Simulation Phase: The simulation (rollout) phase estimates the value of the newly expanded node s' by completing the partial program to a terminal state and evaluating its fitness. Starting from s', a rollout policy π_{rollout} selects actions until a terminal state s_T (i.e., a complete program) is reached. The default rollout policy selects actions uniformly at random from the set of valid actions at each state, while respecting type constraints and depth limitations. To prevent excessive growth and improve the quality of simulated programs, we adjust the probability of selecting terminals versus functions based on the current depth: for $d \leq D_{\max}$,

$$p_{\text{terminal}} = \frac{|\mathcal{T}|}{|\mathcal{T}| + |\mathcal{F}|} + (1 - \frac{|\mathcal{T}|}{|\mathcal{T}| + |\mathcal{F}|}) \cdot \frac{d}{D_{\max}} , \tag{2}$$

where $\frac{|\mathcal{T}|}{|\mathcal{T}|+|\mathcal{F}|}$ serves as a baseline probability based on the relative sizes of the terminal and function sets, and we have $p_{\text{terminal}} = 1$ when $d = D_{\max}$. This depth-dependent adjustment promotes the generation of balanced trees without abrupt truncation. Once a terminal state s_T is reached, we evaluate the fitness $\phi(s_T)$ of the complete program using the appropriate objective function for the target problem and calculate the corresponding reward $\mathcal{R}(s_T)$.

Backpropagation Phase: In the backpropagation phase, the reward $\mathcal{R}(s_T)$ obtained from the simulation is propagated back up the search tree to update the statistics of all nodes visited during the selection and expansion phases. For each state-action pair (s,a) on the path from the root to the expanded node, the statistics are updated as follows:

$$N(s,a) \leftarrow N(s,a) + 1 ,$$
$$N(s) \leftarrow N(s) + 1 .$$

In addition, for program synthesis tasks where fitness exhibits significant variance due to the diverse structures of generated programs, we implement a bounded update rule to mitigate the impact of outlier rewards:

$$\overline{Q}(s,a) \leftarrow \overline{Q}(s,a) + \frac{\min(\max(\mathcal{R}(s_T), \overline{Q}(s,a) - \delta), \overline{Q}(s,a) + \delta) - \overline{Q}(s,a)}{N(s,a)} \ , \tag{3}$$

where $\delta > 0$ controls the maximum change in the estimated value per iteration. This prevents large oscillations caused by high variance in program fitness and ensures more stable and robust exploration of the program space.

This iterative process of selection, expansion, simulation, and backpropagation continues until the computational budget is exhausted. The algorithm then returns a complete program, typically the one corresponding to the path with the highest average reward. By systematically tracking the potential of different decision paths and allocating computational resources accordingly, MCTS efficiently explores the program space–focusing on promising regions while still exploring alternatives–and addresses the exploration–exploitation dilemma that traditional GP approaches often face.

Algorithm 1 presents the pseudocode for our MCTS-based program synthesis approach. The algorithm takes as input the function set $\mathcal{F}$, terminal set $\mathcal{T}$, maximum tree depth $D_{\max}$, and a computational budget (either maximum iterations or time limit). It outputs the best program discovered during the entire search. Moreover, the proposed MCTS framework inherently captures solution diversity within its search tree, allowing for the extraction of a portfolio of high-quality, structurally diverse programs from its most promising branches. This provides an alternative means of obtaining multiple high-quality candidates compared to the population diversity mechanism of traditional GP.

We analyze the computational complexity of the algorithm in Proposition 1 below.

Proposition 1. *The computational complexity of Algorithm 1 over the total budget B is* $O\big(B \cdot (D_{max} \cdot (|\mathcal{F}| + |\mathcal{T}|) + \phi_{eval})\big)$.

Proof. We analyze the complexity of each phase in the proposed MCTS-based program synthesis algorithm. In the selection phase, computing equation (1) for all $a \in \mathcal{A}(s)$ at each node s requires $O(|\mathcal{A}(s)|)$ operations, where $|\mathcal{A}(s)| \leq |\mathcal{F}| + |\mathcal{T}|$. The penalty term complexity(s,a) is computed in $O(1)$ time. Thus, the selection phase takes $O(D_{\max} \cdot (|\mathcal{F}| + |\mathcal{T}|))$ time. The expansion phase selects an untried action and updates the tree, requiring $O(1)$ for action selection (assuming a precomputed $\mathcal{A}(s)$) and $O(1)$ for tree updates, as node additions involve constant-time pointer operations.

In the simulation phase, the worst-case scenario involves filling all unfilled nodes up to depth $D_{\max}$. For each node, computing p_{terminal} using equation (2) and selecting an action both take $O(1)$ time, but evaluating $\mathcal{A}(s)$ may require $O(|\mathcal{F}| + |\mathcal{T}|)$ time to check type constraints. The number of actions is bounded by the maximum number of nodes in a tree of depth $D_{\max}$, which is $O(|\mathcal{F}|^{D_{\max}})$ for a function set with maximum arity. However, in practice, simulations typically require fewer steps due to frequent terminal selections. We approximate the simulation complexity as $O(D_{\max} \cdot (|\mathcal{F}| + |\mathcal{T}|))$, assuming balanced trees. In addition, fitness evaluation $\phi(s_T)$ depends on the problem (e.g., $O(N)$ for symbolic regression with N data points). In the backpropagation phase, at most $D_{\max}$ nodes are updated, where each update following equation (3) takes $O(1)$ time, yielding a total of $O(D_{\max})$. Therefore, the per-iteration complexity is dominated by the selection and simulation phases, resulting in $O(D_{\max} \cdot (|\mathcal{F}| + |\mathcal{T}|) + \phi_{\text{eval}})$. Summarizing the above, the proposition is proved. □

The complexity of Algorithm 1 is comparable to that of traditional GP, which evaluates $O(B \cdot \mu)$ programs, where μ is the population size. However, MCTS focuses computational effort on promising regions of the search space, potentially reducing the effective number of evaluations needed. Although MCTS introduces additional overhead due to tree management and statistical updates, this cost constitutes an investment in a more guided and intelligent search process.

We next establish the theoretical properties of our MCTS-based program synthesis algorithm. Let $Q^(s,a)$ denote the true maximum expected reward achievable from state s by taking action a and subsequently

Algorithm 1 MCTS for Program Synthesis

Input: Function set $\mathcal{F}$, Terminal set $\mathcal{T}$, Maximum depth $D_{\max}$, Computational budget B, Exploration constant c, Complexity penalty weight λ

Output: Best program p^*

Initialize search tree $\mathcal{T}_{search}$ with root node $s_0 = (\{\square\}, [\square])$
Initialize best program $p^* \leftarrow$ null, best fitness $\phi^* \leftarrow \infty$ (for minimization)
while computational budget B not exhausted **do**
  $s \leftarrow s_0$ ▷ Start from root node
  path $\leftarrow []$ ▷ Path of (state, action) pairs traversed
  while s is not a leaf node in $\mathcal{T}_{search}$ **do**
    $a^* \leftarrow \arg\max_{a \in \mathcal{A}(s)} \left\{ \overline{Q}(s,a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}} - \lambda \cdot \text{complexity}(s,a) \right\}$ ▷ Selection using equation (1)
    path.append$((s,a^*))$
    $s \leftarrow \text{TRANSITION}(s,a^*)$ ▷ Apply action to state
  end while
  if s is not terminal & s has unexplored actions in $\mathcal{A}(s)$ **then**
    Choose unexplored $a \in \mathcal{A}(s)$ ▷ Expansion
    path.append$((s,a))$
    $s' \leftarrow \text{TRANSITION}(s,a)$
    Add s' to $\mathcal{T}_{search}$ as child of s via action a
    $s \leftarrow s'$
  end if
  if s is terminal (complete program) **then**
    $\mathcal{R} \leftarrow g(\phi(s))$ ▷ Evaluate fitness and transform to reward
  else
    $\mathcal{R} \leftarrow \text{SIMULATE}(s, \mathcal{F}, \mathcal{T}, D_{\max})$ ▷ Simulation
  end if
  for each (s_i, a_i) in path **do**
    $N(s_i,a_i) \leftarrow N(s_i,a_i) + 1$ ▷ Backpropagation
    $N(s_i) \leftarrow N(s_i) + 1$
    $\overline{Q}(s_i,a_i) \leftarrow \overline{Q}(s_i,a_i) + \frac{\min(\max(R, \overline{Q}(s_i,a_i)-\delta), \overline{Q}(s_i,a_i)+\delta) - \overline{Q}(s_i,a_i)}{N(s_i,a_i)}$ ▷ Bounded update from equation (3)
  end for
  if s is terminal & $\phi(s) < \phi^*$ **then**
    $p^* \leftarrow$ program corresponding to s
    $\phi^* \leftarrow \phi(s)$
  end if
end while
  return p^*

following the optimal policy. An action $a^* \in \mathcal{A}(s)$ is optimal if $Q^*(s,a^*) = \max_{a' \in \mathcal{A}(s)} Q^*(s,a')$. Let $\overline{Q}_n(s,a)$ and $N_n(s,a)$ denote the estimated action value and visit count, respectively, for action a at state s after n total MCTS iterations. We assume the GP problem is formulated as a finite-horizon MDP $(\mathcal{S}, \mathcal{A}, \mathbb{P}, \mathcal{R})$ with deterministic transitions, as defined in Section 2, and under the following assumptions:

- Bounded Reward: There exists $R_{\max} > 0$ such that $|\mathcal{R}(s,a,s')| \leq R_{\max}$ for all terminal states.
- Finite Branching: For any state s, the action space $\mathcal{A}(s)$ is finite.
- Finite Horizon: All episodes terminate in at most H steps (bounded by $D_{\max}$).

Algorithm 2 SIMULATE: Rollout for Program Synthesis

Input: Current state $s = (T_s, L_s)$, Function set $\mathcal{F}$, Terminal set $\mathcal{T}$, Maximum depth $D_{\max}$
Output: Reward $\mathcal{R}$

while $L_s \neq \emptyset$ (program is incomplete) **do**
  $l_1 \leftarrow$ first unfilled node in L_s
  $d \leftarrow \text{depth}(l_1)$
  $p_{\text{terminal}} \leftarrow \frac{|\mathcal{T}|}{|\mathcal{T}|+|\mathcal{F}|} + (1 - \frac{|\mathcal{T}|}{|\mathcal{T}|+|\mathcal{F}|}) \cdot \frac{d}{D_{\max}}$ ▷ Using equation (2)
  $r \leftarrow \text{random}(0,1)$ ▷ Generate random number
  if $r < p_{\text{terminal}}$ & $\mathcal{T} \cap \mathcal{A}(s) \neq \emptyset$ **then**
    Select $t \in \mathcal{T} \cap \mathcal{A}(s)$ uniformly at random
    $a \leftarrow t$
  else
    Select $f \in \mathcal{F} \cap \mathcal{A}(s)$ uniformly at random
    $a \leftarrow f$
  end if
  $s \leftarrow \text{TRANSITION}(s, a)$ ▷ Apply action to update state
  Update T_s and L_s based on the action
end while
Evaluate the fitness $\phi(s)$ of the complete program
$\mathcal{R} \leftarrow g(\phi(s))$ ▷ Transform fitness to reward
return $\mathcal{R}$

- Bounded Complexity Penalty: The complexity penalty term $\lambda \cdot \text{complexity}(s,a)$ in equation (1) is bounded for all reachable (s,a); that is, $\exists M < \infty$ such that $\lambda \cdot \text{complexity}(s,a) \leq M$.

In the following, we show that the algorithm is consistent–that is, as the computational budget increases, the probability of selecting the optimal program converges to one.

Theorem 1 (Consistency) Let $a_n^*(s_0)$ be the action selected at the root node s_0 after n total iterations, based on maximizing the estimated value. Under the stated assumptions, with Algorithm 1, the action selected at the root node converges to an optimal action with probability 1 as $n \to \infty$:

$$\lim_{n\to\infty} \Pr\left(a_n^*(s_0) \in \arg\max_{a' \in \mathcal{A}(s_0)} Q^*(s_0, a')\right) = 1 \ .$$

Consequently, the best program returned by MCTS converges to the optimal program p^* with probability 1:

$$\lim_{n\to\infty} \Pr(p_n^* = p^*) = 1 \ .$$

Proof. The result follows from standard MCTS theory for finite MDPs with bounded rewards and deterministic transitions. A complete proof is relegated to future work. □

4 NUMERICAL EXPERIMENTS

In this section, we conduct numerical experiments to evaluate the empirical performance of the proposed algorithm (MCTS-GP). We compare it against two baseline methods: a standard Genetic Programming (SGP) algorithm, which follows the traditional generational GP framework using tournament selection, subtree crossover, and subtree mutation; and a Random Search (RS) approach, which generates a fixed number of random programs (matching the target computational budget) and returns the best one found. The methods are tested on two symbolic regression benchmark problems, where the goal is to find a

mathematical function $f(\mathbf{x})$ that best approximates a target function $f_{\text{target}}(\mathbf{x})$ based on a set of sample data points. Fitness is measured using the Mean Squared Error (MSE) on a training dataset.

Problem 1: Quartic Polynomial (QP). The first target function involves finding a mathematical expression that approximates the following quartic polynomial:

$$f_{\text{QP}}(x) = x^4 + x^3 + x^2 + x \ .$$

Problem 2: Trigonometric Function (TF). The second target function is defined as:

$$f_{\text{TF}}(x) = \sin(x^2)\cos(x) - 1 \ .$$

For both problems, we generate training data by sampling 100 points uniformly from the interval $[-1,1]$ and computing the corresponding function values. The fitness $\phi(p)$ of a candidate program p is measured using the Mean Squared Error (MSE), where lower values indicate better performance: $\phi(p) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} (p(x_i) - y_i)^2$.

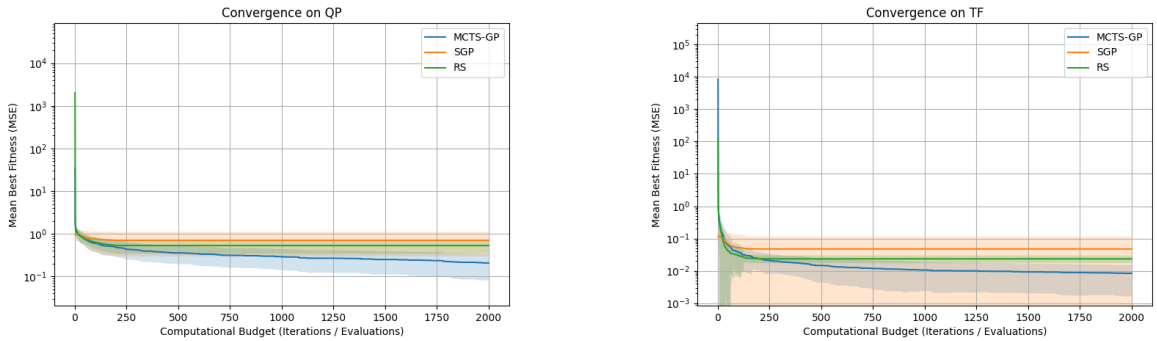
The experiments employed a common primitive set comprising the terminals $\mathcal{T} = \{x, [-1,1]\}$ and the function symbols $\mathcal{F} = \{+, -, \times, \%, \sin, \cos\}$, where protected division (%) returns 1 when the denominator is near zero. All algorithms limited the initial tree depth to $D_{\max} = 6$. For MCTS-GP, we use 2,000 iterations with parameters $C = 2.0$, $\lambda = 0.01$, $\alpha = \beta = 0.5$, and $\delta = 0.1$. SGP runs with a population size of $\mu = 50$ for 40 generations, totaling 2,000 fitness evaluations, using a crossover probability of 0.9, mutation probability of 0.1, tournament size of 7, and elitism of one individual. RS simply evaluates 2,000 randomly generated programs created via ramped half-and-half initialization. Performance is assessed by the best mean squared error and the node count of the best-of-run program. Each algorithm–problem pair was repeated for 100 independent runs to obtain reliable statistics. Table 1 presents the performance comparison, including the mean and standard deviation (in parentheses) over 100 independent runs for both the best fitness (MSE) and the average tree size.

Table 1: Comparison of MCTS-GP, SGP, and RS on Symbolic Regression Benchmarks (mean and standard deviation over 100 runs; $B = 2000$).

Algorithm	Problem 1: Quartic Polynomial (QP)		Problem 2: Trigonometric (TF)	
	Best Fitness (MSE)	**Avg. Tree Size**	**Best Fitness (MSE)**	**Avg. Tree Size**
MCTS-GP	0.207 (0.126)	9.8 (2.9)	0.008 (0.007)	7.3 (2.7)
SGP	0.701 (0.406)	6.9 (6.5)	0.048 (0.067)	5.1 (5.8)
RS	0.530 (0.189)	12.2 (9.8)	0.024 (0.006)	5.2 (6.8)

From Table 1, we can see that for the QP problem, MCTS-GP achieves the best average fitness, significantly outperforming both SGP and RS. This suggests that the guided search of MCTS-GP is more effective at finding accurate solutions compared to the population-based approach of SGP and the unguided sampling of RS, even within a limited budget. In terms of solution complexity, SGP finds the smallest trees on average, while MCTS-GP produces moderately sized trees, and RS generates the largest. The high standard deviation in the fitness of SGP indicates considerable variability in its performance across runs for this problem. For the TF problem, MCTS-GP again achieves the best average fitness, clearly surpassing both RS and SGP. Regarding tree size, SGP finds the most compact solutions on average, closely followed by RS, while MCTS-GP produces slightly larger trees. This may indicate that for the TF problem, very simple structures found quickly by SGP or RS can provide reasonable initial fitness, whereas MCTS-GP explores slightly more complex structures to achieve its superior accuracy. The relatively low standard deviation in fitness for both MCTS-GP and RS on the TF problem suggests more consistent performance compared to SGP. The convergence plots (Figure 2) visually illustrate these trends. For the QP problem, the curve for MCTS-GP (blue) converges rapidly to the lowest MSE level, while for the TF problem, MCTS-GP also shows consistent improvement. The plots suggest that with more iterations, MCTS-GP

could further improve its solutions, particularly in comparison to SGP, which sometimes stagnates or exhibits high variance. In conclusion, MCTS-GP demonstrates the best overall performance in terms of solution accuracy (achieving the lowest MSE on both problems). While SGP occasionally finds smaller trees, MCTS-GP maintains competitive compactness. The results support the potential of MCTS as an effective method for program synthesis, although further experiments on larger problem sizes are needed to fully assess its scalability and convergence properties relative to standard GP.



(a) Quartic Polynomial (QP)

(b) Trigonometric Function (TF)

Figure 2: Convergence plots showing the mean best fitness (MSE) over 100 runs as a function of computational budget ($B = 2000$) for MCTS-GP, SGP, and RS. Lower MSE indicates better performance.

5 CONCLUSIONS

This paper introduces a novel approach to program synthesis by formulating GP problems as a sequential decision-making process navigated through MCTS. By casting program construction as a Markov Decision Process (MDP), we leverage the inherent strengths of MCTS in balancing exploration and exploitation to systematically search the vast program space. The modified UCT selection strategy, which incorporates complexity penalties, effectively guides the search toward simpler and more efficient program structures, while the depth-dependent terminal selection probability during simulation encourages the generation of balanced program trees. Experimental results demonstrate that MCTS-based program synthesis consistently outperforms traditional GP on benchmark problems, discovering higher-quality solutions with comparable structural complexity.

Future work will focus on scaling the approach to larger and more complex problems to assess its generalizability and efficiency in real-world applications. In addition, integrating neural networks to guide MCTS can enhance its performance on large search spaces by predicting promising program structures or refining the simulation policy. As program synthesis continues to evolve, the framework presented in this paper offers a promising foundation for more efficient and effective automated program generation, with potential applications extending beyond traditional GP domains to areas such as automated machine learning, algorithm discovery, and software engineering. While recent LLM-based code generation approaches achieve impressive performance, they often lack type guarantees and structural validity. In contrast, our MCTS-GP framework ensures type-safe and syntactically correct program construction by design. Combining the flexibility of LLMs with the formal rigor of MCTS-GP offers a promising direction for future research.

ACKNOWLEDGMENTS

The work was supported by the National Natural Science Foundation of China (NSFC) under Grants 72501012, 72325007, and 72250065, and the China Postdoctoral Science Foundation under Grants 2025T180219 and 2025M770795.

REFERENCES

Ahvanooey, M. T., Q. Li, M. Wu, and S. Wang. 2019. "A Survey of Genetic Programming and Its Applications". *KSII Transactions on Internet and Information Systems (TIIS)* 13(4):1765–1794.

Bhowan, U., M. Johnston, M. Zhang, and X. Yao. 2012. "Evolving Diverse Ensembles Using Genetic Programming for Classification with Unbalanced Data". *IEEE Transactions on Evolutionary Computation* 17(3):368–386.

Browne, C. B., E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, *et al*. 2012. "A Survey of Monte Carlo Tree Search Methods". *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Espejo, P. G., S. Ventura, and F. Herrera. 2009. "A Survey on the Application of Genetic Programming to Classification". *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 40(2):121–144.

Folino, G., F. S. Pisani, and P. Sabatino. 2016. "An Incremental Ensemble Evolved by Using Genetic Programming to Efficiently Detect Drifts in Cyber Security Datasets". In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion,* July 20th–24th, Denver, USA, 1103–1110.

Islam, M., N. Kharma, and P. Grogono. 2020. "Mutation Operators for Genetic Programming Using Monte Carlo Tree Search". *Applied Soft Computing* 97:106717.

Kocsis, L., and C. Szepesvári. 2006. "Bandit-Based Monte-Carlo Planning". In *Proceedings of the 17th European Conference on Machine Learning,* September 18th–22th, Berlin, Germany, 282–293.

Koza, J. R. 1982. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge: MIT Press.

Li, L., H. Zhang, and S. Bai. 2024. "A Multi-Surrogate Genetic Programming Hyper-heuristic Algorithm for the Manufacturing Project Scheduling Problem with Setup Times under Dynamic and Interference Environments". *Expert Systems with Applications* 250:123854.

Li, Y., M. C. Fu, and J. Xu. 2021. "An Optimal Computing Budget Allocation Tree Policy for Monte Carlo Tree Search". *IEEE Transactions on Automatic Control* 67(6):2685–2699.

Liu, X., Y. Peng, G. Zhang, and R. Zhou. 2024. "An Efficient Node Selection Policy for Monte Carlo Tree Search with Neural Networks". *INFORMS Journal on Computing* 37(4):785–807.

Luke, S., and L. Panait. 2006. "A Comparison of Bloat Control Methods for Genetic Programming". *Evolutionary Computation* 14(3):309–344.

Muñoz, L., S. Silva, and L. Trujillo. 2015. "M3GP–Multiclass Classification with GP". In *Proceedings of the 18th European Conference on Genetic Programming,* April 8th–10th, Copenhagen, Denmark, 78–91.

Nguyen, S., M. Zhang, M. Johnston, and K. C. Tan. 2012. "A Computational Study of Representations in Genetic Programming to Evolve Dispatching Rules for the Job Shop Scheduling Problem". *IEEE Transactions on Evolutionary Computation* 17(5):621–639.

O'Neill, M., L. Vanneschi, S. Gustafson, and W. Banzhaf. 2010. "Open Issues In Genetic Programming". *Genetic Programming and Evolvable Machines* 11:339–363.

Schmidt, M., and H. Lipson. 2009. "Distilling Free-Form Natural Laws from Experimental Data". *Science* 324(5923):81–85.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, *et al*. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search". *Nature* 529(7587):484–489.

Vanneschi, L., and R. Poli. 2012. "Genetic Programming-Introduction, Applications, Theory and Open Issues". *Handbook of Natural Computing* 2:709–739.

Yin, J., H. Wang, J. Li, Z. Zhang, S. Cai, and W. Liu. 2024. "Dispatching Rule Design for Tower Crane Scheduling in Prefabricated Construction Via Genetic Programming". *Automation in Construction* 165:105588.

Zaki, T., Y. Zeiträg, R. Neves, and J. R. Figueira. 2024. "A Cooperative Coevolutionary Genetic Programming Hyper-Heuristic for Multi-Objective Makespan and Cost Optimization in Cloud Workflow Scheduling". *Computers & Operations Research* 172:106805.

Zhang, G., Y. Peng, and Y. Xu. 2022. "An Efficient Dynamic Sampling Policy for Monte Carlo Tree Search". In *2022 Winter Simulation Conference (WSC)*, 2760–2771 https://doi.org/10.1109/WSC57314.2022.10015374.

AUTHOR BIOGRAPHIES

GONGBO ZHANG is a Postdoctoral Fellow at the Guanghua School of Management, Peking University. His research interests include simulation optimization and artificial intelligence. His email address is gongbozhang@pku.edu.cn.

YIJIE PENG is an Associate Professor in Guanghua School of Management at Peking University. His research interests include stochastic modeling and analysis, simulation optimization, machine learning, data analytics, and healthcare. His email address is pengyijie@pku.edu.cn.