Synthesizing Tasks for Block-based Programming

Umair Z. Ahmed 1  Maria Christakis 2  Aleksandr Efremov 2  Nigel Fernandez 2  Ahana Ghosh 2  Abhik Roychoudhury 1  Adish Singla 2
1 National University of Singapore, {umair, abhik}@comp.nus.edu.sg,
2 MPI-SWS, {maria, aefremov, nfernand, gahana, adisha}@mpi-sws.org

Abstract

Block-based visual programming environments play a critical role in introducing computing concepts to K-12 students. One of the key pedagogical challenges in these environments is in designing new practice tasks for a student that match a desired level of difficulty and exercise specific programming concepts. In this paper, we formalize the problem of synthesizing visual programming tasks. In particular, given a reference visual task $T^{in}$ and its solution code $C^{in}$, we propose a novel methodology to automatically generate a set $\{ (T^{out}, C^{out}) \}$ of new tasks along with solution codes such that tasks $T^{in}$ and $T^{out}$ are conceptually similar but visually dissimilar. Our methodology is based on the realization that the mapping from the space of visual tasks to their solution codes is highly discontinuous; hence, directly mutating reference task $T^{in}$ to generate new tasks is futile. Our task synthesis algorithm operates by first mutating code $C^{in}$ to obtain a set of codes $\{ C^{out} \}$. Then, the algorithm performs symbolic execution over a code $C^{out}$ to obtain a visual task $T^{out}$; this step uses the Monte Carlo Tree Search (MCTS) procedure to guide the search in the symbolic tree. We demonstrate the effectiveness of our algorithm through an extensive empirical evaluation and user study on reference tasks taken from the Hour of Code: Classic Maze challenge by Code.org and the Intro to Programming with Karel course by CodeHS.com.

1 Introduction

Block-based visual programming environments are increasingly used nowadays to introduce computing concepts to novice programmers including children and K-12 students. Led by the success of environments like Scratch [29], initiatives like Hour of Code by Code.org [24] (HOC) and online platforms like CodeHS.com [21], block-based programming has become an integral part of introductory computer science education. Considering HOC alone, over one billion hours of block-based programming activity has been performed so far by over 50 million unique students worldwide [24, 55].

The societal need for enhancing K-12 computing education has led to a surge of interest in developing AI-driven systems for pedagogy of block-based programming [33, 26, 27, 34, 16]. Existing works have studied various aspects of intelligent support, including providing real-time next-step hints when a student is stuck solving a task [20, 56, 18, 17, 9], giving data-driven feedback about a student’s misconceptions [31, 19, 28, 30, 35], and demonstrating a worked-out solution for a task when a student lacks the required programming concepts [57]. An underlying assumption when providing such intelligent support is that afterwards the student can practice new similar tasks to finally learn the missing concepts. However, this assumption is far from reality in existing systems—the programming tasks are typically hand-curated by experts/tutors, and the available set of tasks is limited. Consider HOC’s Classic Maze challenge [23], which provides a progression of 20 tasks: Millions of students have attempted these tasks, yet when students fail to solve a task and receive assistance, they cannot practice similar tasks, hindering their ability to master the desired concepts. We seek to tackle this pedagogical challenge by developing techniques for synthesizing new programming tasks.

*Authors listed alphabetically; Correspondence to: Ahana Ghosh <gahana@mpi-sws.org>.
We formalize the problem of synthesizing visual programming tasks of the kind found in popular learning platforms like Code.org (see Fig. 1) and CodeHS.com (see Fig. 2). As input, we are given a reference task $T_{\text{in}}$, specified as a visual puzzle, and its solution code $C_{\text{in}}$. Our goal is to synthesize a set $\{(T_{\text{out}}, C_{\text{out}})\}$ of new tasks along with their solution codes that are conceptually similar but visually dissimilar to the input. This is motivated by the need for practice tasks that on one hand exercise the same concepts, while looking fresh in order to maintain student engagement.

When tackling the problem of synthesizing new tasks with the above desirable properties, three key challenges emerge. First, we are generating problems in a conceptual domain with no well-defined procedure that students follow to solve a task—consequently, existing work on educational problem generation in procedural domains does not apply in our setting [3, 11]. Second, the mapping from the space of visual tasks to their solution codes is highly discontinuous; hence, template-based problem generation techniques [32, 25] that rely on directly mutating the input to generate new tasks is ineffective (see Section 5 where we use this approach as a baseline). Furthermore, such a direct task-mutation approach would require access to an automated solution synthesizer; however, state-of-the-art program synthesis techniques are not yet on par with experts and their minimal solutions [5, 8, 6]. Third, the space of possible tasks and their solutions is potentially unbounded, and thus, any problem generation technique that relies on exhaustive enumeration is intractable [32, 1, 2].

To overcome these challenges, we propose a novel methodology that operates by first mutating the solution code $C_{\text{in}}$ to obtain a set of codes $\{C_{\text{out}}\}$, and then performing symbolic execution over a code $C_{\text{out}}$ to obtain a visual puzzle $T_{\text{out}}$. Mutation is efficient by creating an abstract representation of $C_{\text{in}}$ along with appropriate constraints and querying an SMT solver [4]; any solution to this query is a mutated code $C_{\text{out}}$. During symbolic execution, we use Monte Carlo Tree Search (MCTS) to guide the search over the (unbounded) symbolic execution tree. We demonstrate the effectiveness of our methodology by performing an extensive empirical evaluation and user study on a set of reference tasks from the Hour of Code challenge by Code.org and the Intro to Programming with Karel course by CodeHS.com. In summary, our main contributions are:

- We formalize the problem of synthesizing block-based visual programming tasks (Section 2).
- We present a novel approach for generating new visual tasks along with solution codes such that they are conceptually similar but visually dissimilar to a given reference task (Section 3).
- We demonstrate the effectiveness of our approach through an extensive empirical evaluation and user study on reference tasks from real-world programming platforms (Section 4 and Section 5).

2 Problem Formulation

The space of tasks. We define a task as a tuple $T := (T_{\text{vis}}, T_{\text{store}}, T_{\text{size}})$, where $T_{\text{vis}}$ denotes the visual puzzle, $T_{\text{store}}$ the available block types, and $T_{\text{size}}$ the maximum number of blocks allowed in the
solution code. For instance, considering the task $T := T^m$ in Fig. 1a, $T_{vis}$ is illustrated in Fig. 1a. $T_{store} = \{ \text{move, turnL, turnR, \textit{repeatUntil, if}\}$, and $T_{size} = 4$.

The space of codes. The programming environment has a domain-specific language (DSL), which defines the set of valid codes $C$ and is shown in Fig. 1a. A code $C \in C$ is characterized by several properties, such as the set $C_{\text{blocks}}$ of block types in $C$, the number of blocks $C_{\text{size}}$, the depth $C_{\text{depth}}$ of the corresponding Abstract Syntax Tree (AST), and the nesting structure $C_{\text{struct}}$ representing program concepts exercised by $C$. For instance, considering the code $C := C^m$ in Fig. 1b, $C_{\text{blocks}} = \{ \text{move, turnL, \textit{repeatUntil, if}\}$, $C_{\text{size}} = 4$, $C_{\text{depth}} = 3$, and $C_{\text{struct}} = \{ \text{Run\{RepeatUntil\{if\}\}}\}$.

Below, we introduce two useful definitions relating the task and code space.

**Definition 1** (Solution code). $C$ is a solution code for $T$ if the following holds: $C$ successfully solves the visual puzzle $T_{vis}$, $C_{\text{blocks}} \subseteq T_{\text{store}}$, and $C_{\text{size}} \leq T_{\text{size}}$. $C_T$ denotes the set of all solution codes for $T$.

**Definition 2** (Minimality of a task). Given a solvable task $T$ with $|C_T| \geq 1$ and a threshold $\delta \in \mathbb{N}$, the task is minimal if $\exists C \in C_T$ such that $C_{\text{size}} < T_{\text{size}} - \delta$.

Next, we introduce two definitions formalizing the notion of conceptual similarity. Definition 3 formalizes conceptual similarity of a task $T$ along with one solution code $C$. Since a task can have multiple solution codes, Definition 4 provides a stricter notion of conceptual similarity of a task $T$ for all its solution codes. These definitions are used in our objective of task synthesis in conditions (I) and (V) below.

**Definition 3** (Conceptual similarity of $(T, C)$). Given a reference $(T^n, C^n)$ and a threshold $\delta \in \mathbb{N}$, a task $T$ along with a solution code $C$ is conceptually similar to $(T^n, C^n)$ if the following holds: $T_{\text{store}} = T^n_{\text{store}}$, $T_{\text{size}} - T^n_{\text{size}} \leq \delta$, and $C_{\text{struct}} = C^n_{\text{struct}}$.

**Definition 4** (Conceptual similarity of $(T, C)$). Given a reference $(T^n, C^n)$ and a threshold $\delta \in \mathbb{N}$, a task $T$ is conceptually similar to $(T^n, C^n)$ if the following holds: $T_{\text{store}} = T^n_{\text{store}}$, $T_{\text{size}} - T^n_{\text{size}} \leq \delta$, and $\forall C \in C_T$, $C_{\text{struct}} = C^n_{\text{struct}}$.

Environment domain knowledge. We now formalize our domain knowledge about the block-based environment to measure visual dissimilarity of two tasks, and capture some notion of interestingness and quality of a task. Given tasks $T$ and $T^\prime$, we measure their visual dissimilarity by an environment-specific function $F_{\text{diss}}(T_{\text{vis}}, T^{\prime}_{\text{vis}}) \in [0, 1]$. Moreover, we measure generic quality of a task with function $F_{\text{qual}}(T_{\text{vis}}, C) \in [0, 1]$. We provide specific instantiations of $F_{\text{diss}}$ and $F_{\text{qual}}$ in our evaluation.

**Objective of task synthesis.** Given a reference task $T^n$ and a solution code $C^n \in C_T^n$, as input, we seek to generate a set $\{(T^{\text{out}}_n, C^{\text{out}}_n)\}$ of new tasks along with solution codes that are conceptually similar but visually dissimilar to the input. Formally, given parameters ($\delta_{\text{size}}, \delta_{\text{diss}}, \delta_{\text{qual}}$), our objective is to synthesize new tasks meeting the following conditions:

(I) $(T^{\text{out}}_n, C^{\text{out}}_n)$ is conceptually similar to $(T^n, C^n)$ with threshold $\delta_{\text{size}}$ in Definition 3.

(II) $T^{\text{out}}_n$ is visually dissimilar to $T^n$ with margin $\delta_{\text{diss}}$, i.e., $F_{\text{diss}}(T^{\prime}_{\text{vis}}, T^{\text{out}}_{\text{vis}}) \geq \delta_{\text{diss}}$.

(III) $T^{\text{out}}_n$ has a quality score above threshold $\delta_{\text{qual}}$, i.e., $F_{\text{qual}}(T^{\text{out}}_{\text{vis}}, C^{\text{out}}) \geq \delta_{\text{qual}}$.

In addition, depending on the use case, it is desirable that the new tasks satisfy the following criteria:

(IV) $C^{\text{out}}_n$ is different from the input solution code, i.e., $C^{\text{out}}_n \neq C^n$.

(V) $T^{\text{out}}_n$ is conceptually similar to $(T^n, C^n)$ with threshold $\delta_{\text{size}}$ in Definition 3.

(VI) $T^{\text{out}}_n$ is minimal as per Definition 2 for a desired value of $\delta_{\text{mini}}$ (e.g., $\delta_{\text{mini}} = 0$ or $\delta_{\text{mini}} = 1$).

3 **Our Task Synthesis Algorithm**

We now present the pipeline of our algorithm (see Fig. 3), which takes as input a reference task $T^n$ and its solution code $C^n$, and generates a set $\{(T^{\text{out}}_n, C^{\text{out}}_n)\}$ of new tasks with their solution codes. The goal is to design a task $T^{\text{out}}_n$ that is conceptually similar to $T^n$, but for new tasks $T^{\text{out}}_n$ to be visually dissimilar to $T^n$. This is achieved by two main stages: (1) mutation of $C^n$ to obtain a set $\{C^{\text{out}}_n\}$, and (2) symbolic execution of each $C^{\text{out}}_n$ to create a task $T^{\text{out}}_n$. The first stage, presented in Section 3.1, converts $C^n$ into an abstract representation restricted by a set of constraints (Fig. 3(a)), which must be satisfied by any generated $C^{\text{out}}_n$ (Fig. 3(b)). The second stage, described in Section 3.2, applies symbolic execution on each code $C^{\text{out}}_n$ to create a corresponding visual task $T^{\text{out}}_n$ (Fig. 3(c)) while using Monte Carlo Tree Search (MCTS) to guide the search in the symbolic execution tree.

![Figure 3: Stages in our task synthesis algorithm.](image-url)
we also use mapping $\omega$ to constraints imposed on action sequences nested within conditionals. As an example, consider

If optimal, i.e., action subsequences before and after this construct are not nested in it.

Repeat

possible values of the $\omega$ could result in converting all $\text{turnLeft}$ actions of a sequence to $\text{turnRight}$. $\Delta_2$ restricts the possible values of the $\text{Repeat}$ counter within threshold $\delta_{\text{iter}}$. $\Delta_3$ ensures that the $\text{Repeat}$ counter is optimal, i.e., action subsequences before and after this construct are not nested in it. $\Delta_4$ specifies the possible values of the $\text{IF}$ condition based on its value in the code, given by $\omega(\cdot \mid C)$. $\Delta_5$ refers to constraints imposed on action sequences nested within conditionals. As an example, consider

3.1 Code Mutation

This stage in our pipeline mutates code $C^m$ of task $T^m$ such that its conceptual elements are preserved. Our procedure consists of three main steps. First, we generate an abstract representation of $C^m$, called sketch. Second, we restrict the sketch with constraints that describe the space of its concrete instantiations. Although this formulation is inspired from work on generating algebra problems [32], we use it in the entirely different context of generating conceptually similar mutations of $C^m$. This is achieved in the last step, where we use the sketch and its constraints to query an SMT solver [4]; the query solutions are mutated codes $\{C^\text{out}\}$ such that $C^\text{out} = C^m$ (see Definition [3]).

Step 1: Sketch. The sketch of code $C$, denoted by $Q$, is an abstraction of $C$ capturing its skeleton and generalizing $C$ to the space of conceptually similar codes. $Q$, expressed in the language of Fig. [4d], is generated from $C$ with mapping $\Omega$. In particular, the map exploits the AST structure of the code: the AST is traversed in a depth-first manner, and all values are replaced with their corresponding sketch instantiations. Although this formulation is inspired from work on generating algebra problems [32], we use it in the entirely different context of generating conceptually similar mutations of $C^m$. This is achieved in the last step, where we use the sketch and its constraints to query an SMT solver [4]; the query solutions are mutated codes $\{C^\text{out}\}$ such that $C^\text{out} = C^m$ (see Definition [3]).

Step 2: Sketch constraints. Sketch constraints restrict the possible concrete instantiations of a sketch by encoding the required semantics of the mutated codes. All constraint types are in Fig. [4c].

In particular, $\Delta_0$ restricts the size of the mutated code within $\delta_{\text{size}}$. $\Delta_1$ specifies the allowed mutations to an action sequence based on its value in the code, given by $\omega(\cdot \mid C)$. For instance, this constraint could result in converting all $\text{turnLeft}$ actions of a sequence to $\text{turnRight}$. $\Delta_2$ restricts the possible values of the $\text{Repeat}$ counter within threshold $\delta_{\text{iter}}$. $\Delta_3$ ensures that the $\text{Repeat}$ counter is optimal, i.e., action subsequences before and after this construct are not nested in it. $\Delta_4$ specifies the possible values of the $\text{IF}$ condition based on its value in the code, given by $\omega(\cdot \mid C)$. $\Delta_5$ refers to constraints imposed on action sequences nested within conditionals. As an example, consider

Figure 4: Illustration of key steps in Code Mutation. Fig. [4d] shows code $C^m$ from Fig. [1b]. The code mutation stage, when applied to $C^m$, generates many output codes, including $C^\text{out}$ in Fig. [1d].
While this approach generates codes that are devoid of most semantic irregularities, it has its limitations. Certain irregularities continue to exist in some generated codes: An example of such a code included the action sequence `move, turnLeft, move, turnLeft, move, turnLeft, move, turnLeft`, whose output could be achieved by a single `turnRight`. All employed elimination sequences can be found in the supplementary material. The entire list of constraints applied on the solution code in Fig. 4d is shown in Fig. 4f.

**Step 3: SMT query.** For a sketch $Q$ generated from code $C$ and its constraints, we pose the following query to an SMT solver: (sketch $Q$, $Q$-constraints). As a result, the solver generates a set of instantiations, which are conceptually similar to $C$. In our implementation, we used the Z3 solver [7]. For the code in Fig. 4d, Z3 generated 66 mutated codes in 0.8s from an exhaustive space of 2,997 possible codes with $\delta_{\text{size}} = 2$. One such mutation is shown in Fig. 1d.

While this approach generates codes that are devoid of most semantic irregularities, it has its limitations. Certain irregularities continue to exist in some generated codes: An example of such a code included the action sequence `move, turnLeft, move, turnLeft, move, turnLeft, move, turnLeft`, whose output could be achieved by a single `turnRight`. All employed elimination sequences can be found in the supplementary material. The entire list of constraints applied on the solution code in Fig. 4d is shown in Fig. 4f.

**Step 3: SMT query.** For a sketch $Q$ generated from code $C$ and its constraints, we pose the following query to an SMT solver: (sketch $Q$, $Q$-constraints). As a result, the solver generates a set of instantiations, which are conceptually similar to $C$. In our implementation, we used the Z3 solver [7]. For the code in Fig. 4d, Z3 generated 66 mutated codes in 0.8s from an exhaustive space of 2,997 possible codes with $\delta_{\text{size}} = 2$. One such mutation is shown in Fig. 1d.

While this approach generates codes that are devoid of most semantic irregularities, it has its limitations. Certain irregularities continue to exist in some generated codes: An example of such a code included the action sequence `move, turnLeft, move, turnLeft, move, turnLeft, move, turnLeft`, whose output could be achieved by a single `turnRight`. All employed elimination sequences can be found in the supplementary material. The entire list of constraints applied on the solution code in Fig. 4d is shown in Fig. 4f.

**3.2 Symbolic Execution**

Symbolic execution [13] is an automated test-generation technique that symbolically explores execution paths in a program. During exploration of a path, it gathers symbolic constraints over program inputs from statements along the path. These constraints are then mutated (according to a search strategy), and an SMT solver is queried to generate new inputs that explore another path.

**Obtaining visual tasks with symbolic execution.** This stage in our pipeline applies symbolic execution on each generated code $C_{\text{out}}$ to obtain a suitable visual task $T_{\text{out}}$. The program inputs of $C_{\text{out}}$ are the agent’s initial location/orientation and the status of the grid cells (`unknown, free, blocked, goal`), which is initially `unknown`. Symbolic execution collects constraints over these from code statements. As in Fig. 5, for one path, symbolic execution generates a visual task for each path in $C_{\text{out}}$.

However, not all of these tasks are suitable. For instance, if the goal is reached after the first `move` in Fig. 1d, all other statements in $C_{\text{out}}$ are not covered, rendering the task less suitable for this code. Naively, symbolic execution could first enumerate all paths in $C_{\text{out}}$ and their corresponding tasks, and then rank them in terms of suitability. However, solution codes may have an unbounded number of paths, which leads to path explosion, that is, the inability to cover all paths with tractable resources.

**Guiding symbolic execution using Monte Carlo Tree Search (MCTS).** To address this issue, we use MCTS [14] as a search strategy in symbolic execution with the goal of generating more suitable tasks with fewer resources—we define task suitability next. Symbolic execution has been previously combined with MCTS in order to direct the exploration towards costly paths [13]. In the supplementary material, we provide an example demonstrating how MCTS could guide the symbolic execution in generating more suitable tasks.

As previously observed [12], a critical component of effectively applying MCTS is to define an evaluation function that describes the desired properties of the output, i.e., the visual tasks. Tailoring the evaluation function to our unique setting is exactly what differentiates our approach from existing work. In particular, our evaluation function, $F_{\text{score}}$, distinguishes suitable tasks by assigning a score ($\in [0, 1]$) to them, which guides the MCTS search. A higher $F_{\text{score}}$ indicates a more suitable task.
It's constituent components are: (i) \( F_{\text{cov}}(C_{\text{out}}) \in \{0, 1\} \), which evaluates to 1 in the event of complete coverage of code \( C_{\text{out}} \) by task \( T_{\text{vis}} \) and 0 otherwise; (ii) \( F_{\text{dis}}(T_{\text{vis}}, T_{\text{vis}}) \in [0, 1] \), which evaluates the dissimilarity of \( T_{\text{vis}} \) to \( T_{\text{vis}} \) (see Section 2); (iii) \( F_{\text{qual}}(T_{\text{vis}}, C_{\text{out}}) \in [0, 1] \), which evaluates the quality and validity of \( T_{\text{vis}} \); (iv) \( F_{\text{nocrash}}(C_{\text{out}}) \in \{0, 1\} \), which evaluates to 0 in case the agent crashes into a wall and 1 otherwise; and (v) \( F_{\text{nocrash}}(T_{\text{vis}}, C_{\text{out}}) \in \{0, 1\} \), which evaluates to 0 if there is a shortcut sequence of actions (a in Fig. 4a) smaller than \( C_{\text{out}} \) that solves \( T_{\text{vis}} \) and 1 otherwise. \( F_{\text{qual}} \) and \( F_{\text{nocrash}} \) also resolve the limitations of our mutation stage by eliminating codes and tasks that lead to undesirable agent behavior. We instantiate \( F_{\text{score}} \) in the next section.

4 Experimental Evaluation

In this section, we evaluate our task synthesis algorithm on HOC and Karel tasks. Our implementation is publicly available. While we give an overview of key results here, a detailed description of our setup and additional experiments can be found in the supplementary material.

4.1 Reference Tasks and Specifications

Reference tasks. We use a set of ten reference tasks from HOC and Karel, shown in Fig. 6. The HOC tasks were selected from the Hour of Code: Classic Maze challenge by Code.org [22] and the Karel tasks from the Intro to Programming with Karel course by CodeHS.com [23]. The DSL of Fig. 6 is generic in that it includes both HOC and Karel codes, with the following differences: (i) construct \( \text{while} \), marker-related actions \( \text{putM} \), \( \text{pickM} \), and conditions \( \text{noPathA} \), \( \text{noPathL} \), \( \text{noPathR} \), \( \text{marker} \), \( \text{noMarker} \) specific to Karel only; (ii) construct \( \text{repeatUntil} \) and \( \text{goal} \) are specific to HOC only. Furthermore, the puzzles for HOC and Karel are of different styles (see Fig. 1 and Fig. 2). For all tasks, the grid size of the puzzles is fixed to \( 10 \times 10 \) cells (grid-size parameter \( n = 10 \)).

Specification of scoring functions. \( F_{\text{qual}}(T_{\text{vis}}, C_{\text{out}}) \in [0, 1] \) was approximated as the sum of the normalized counts of ‘moves’, ‘turns’, ‘segments’, and ‘long-segments’ in the grid; segments and long-segments are sequences of \( \geq 3 \) and \( \geq 5 \) move actions respectively. More precisely, for HOC tasks, we used the following function where features are computed by executing \( C_{\text{out}} \) on \( T_{\text{vis}} \):

\[
F_{\text{HOC}}(T_{\text{vis}}, C_{\text{out}}) = \frac{1}{4} \left( \frac{\#\text{moves}}{2n} + \frac{\#\text{turns}}{n} + \frac{\#\text{segments}}{n/2} + \frac{\#\text{long-segments}}{n/3} \right).
\]

Furthermore, in our implementation, \( F_{\text{qual}}(\cdot) \) value was set to 0 when \( F_{\text{nocrash}}(\cdot) = 0 \). For Karel tasks, \( F_{\text{qual}} \) additionally included the normalized counts of \( \text{putM} \) and \( \text{pickM} \), and is provided in the supplementary material. \( F_{\text{dis}}(T_{\text{vis}}, T_{\text{vis}}) \in [0, 1] \) was computed based on the dissimilarity of the agent’s initial location/orientation w.r.t. \( T_{\text{vis}} \) and the grid-cell level dissimilarity based on the Hamming distance between \( T_{\text{vis}} \) and \( T_{\text{vis}} \). More precisely, we used the following function:

\[
F_{\text{dis}}(T_{\text{vis}}, T_{\text{vis}}) = \frac{1}{3} \left( \text{diss}(\text{loc} \mid T_{\text{vis}}, T_{\text{vis}}) + \text{diss}(\text{dir} \mid T_{\text{vis}}, T_{\text{vis}}) + \text{diss}(\text{grid-cells} \mid T_{\text{vis}}, T_{\text{vis}}) \right)
\]

where \( \text{diss}(\text{loc} \mid T_{\text{vis}}, T_{\text{vis}}) \in \{0, 1\} \), \( \text{diss}(\text{dir} \mid T_{\text{vis}}, T_{\text{vis}}) \in \{0, 1\} \), and \( \text{diss}(\text{grid-cells} \mid T_{\text{vis}}, T_{\text{vis}}) \in [0, 1] \) (after the Hamming distance is normalized with a factor of \( \frac{1}{n^2} \)).
Analyzing output tasks. We further analyze the generated tasks based on the objectives of Section 2. For H6, the fraction of tasks that satisfy these objectives is listed in the last three columns of Fig. 7. We observe that the vast majority of tasks meet the objectives, even if not by design. For H6, the fraction of tasks meeting objectives (V) and (VI) is met. The generated tasks for which C_{out} is picked only from the pool of generated tasks which have C_{out} ≥ 0.33, and (ii) δ_{qual} = 0.2 for codes with while or repeat until and 0.05 otherwise. We run MCTS 10 times per code, with each run generating one task. We set the maximum iterations of a run to 2 million (M) and the exploration constant to 2 [14]. Even when considering a tree depth of 2n (= 20), there are millions of leaves for difficult tasks H5 and H6, reflecting the complexity of task generation. For each code C_{out}, we generated 10 different visual tasks. To ensure sufficient diversity among the tasks generated for the same code, we introduced a measure $\mathcal{F}_{diversity}$. This measure, not only ensures visual task dissimilarity, but also ensures sufficient diversity in entire symbolic paths during generation (for details, see supplementary material).

<table>
<thead>
<tr>
<th>Task</th>
<th>Code Mutation</th>
<th>Symbolic Execution</th>
<th>Fraction of $T^{out}$ with criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T^m$</td>
<td>$C_{out}$</td>
<td>$T^{out}$</td>
<td>$8$:Time</td>
</tr>
<tr>
<td>H1</td>
<td>3,159</td>
<td>112</td>
<td>64</td>
</tr>
<tr>
<td>H2</td>
<td>8,991</td>
<td>594</td>
<td>138</td>
</tr>
<tr>
<td>H3</td>
<td>798,255</td>
<td>13,122</td>
<td>720</td>
</tr>
<tr>
<td>H4</td>
<td>5,913</td>
<td>152</td>
<td>108</td>
</tr>
<tr>
<td>H5</td>
<td>2,997</td>
<td>294</td>
<td>66</td>
</tr>
<tr>
<td>H6</td>
<td>1,728</td>
<td>294</td>
<td>54</td>
</tr>
<tr>
<td>K7</td>
<td>96,875</td>
<td>150</td>
<td>132</td>
</tr>
<tr>
<td>K8</td>
<td>484,875</td>
<td>4,506</td>
<td>990</td>
</tr>
<tr>
<td>K9</td>
<td>8,590 x 10^6</td>
<td>60,758</td>
<td>888</td>
</tr>
<tr>
<td>K10</td>
<td>32,025 x 10^6</td>
<td>19,328</td>
<td>1,404</td>
</tr>
</tbody>
</table>

Next, we define the evaluation function $\mathcal{F}_{score}(T^{out}, C_{out}, T^m, C^m) \in [0, 1]$ used by MCTS:

$$\mathcal{F}_{score}(T^{out}, C_{out}, T^m, C^m) = \begin{cases} \mathcal{F}_{qual}(T^{out}, C_{out}) \geq \delta_{qual}. \mathcal{F}_{nocut}(T^{out}, C_{out}) = 1, \mathcal{F}_{nocut}(T^{out}, C_{out}) = 1 \end{cases}.$$ (i)

$$\alpha_1 \mathcal{F}_{cov}(T^{out}, C_{out}) + \alpha_2 \mathcal{F}_{qual}(T^{out}, C_{out}) + \alpha_3 \mathcal{F}_{diss}(T^{out}, T^{out}).$$ (ii)

where $\mathcal{I}$ is an indicator function and each constant $\alpha = 1/3$. Component (ii) in the above function supplies the gradients for guiding the search in MCTS; Component (i) is applied at the end of the MCTS run to pick the output. More precisely, the best task (i.e, the one with the highest $\mathcal{F}_{score}$ value) is picked only from the pool of generated tasks which have $\mathcal{F}_{score}(\cdot) > 0$ and satisfy $\mathcal{F}_{cov}(\cdot) = 1$.

Specification of task synthesis and MCTS. As per Section 2 we set the following thresholds for our algorithm: (i) $\delta_{cov} = 2$, (ii) $\delta_{vis} = 0.33$, and (iii) $\delta_{qual} = 0.2$ for codes with while or repeat until, and 0.05 otherwise. We run MCTS 10 times per code, with each run generating one task. We set the maximum iterations of a run to 2 million (M) and the exploration constant to 2 [14]. Even when considering a tree depth of 2n (= 20), there are millions of leaves for difficult tasks H5 and H6, reflecting the complexity of task generation. For each code $C_{out}$, we generated 10 different visual tasks. To ensure sufficient diversity among the tasks generated for the same code, we introduced a measure $\mathcal{F}_{diversity}$. This measure, not only ensures visual task dissimilarity, but also ensures sufficient diversity in entire symbolic paths during generation (for details, see supplementary material).

4.2 Results

Performance of task synthesis algorithm. Fig. 7 shows the results of our algorithm. The second column illustrates the enormity of the unconstrained space of mutated codes; we only impose size constraint $\Delta_0$ from Fig. 4C. We then additionally impose constraint $\Delta_1$ resulting in a partially constrained space of mutated codes (column 3), and finally apply all constraints from Fig. 4C to obtain the final set of generated codes (column 4). This reflects the systematic reduction in the space of mutated codes by our constraints. Column 5 shows the total running time for generating the final codes, which denotes the time taken by Z3 to compute solutions to our mutation query. As discussed in Section 3.1, few codes with semantic irregularities still remain after the mutation stage. The symbolic execution stage eliminates these to obtain the reduced set of valid codes (column 6). Column 7 shows the final number of generated tasks and column 8 is the average time per output task (i.e., one MCTS run).

Analyzing output tasks. We further analyze the generated tasks based on the objectives of Section 2. All tasks satisfy properties (I)–(III) by design. Objective (IV) is easily achieved by excluding generated tasks for which $C_{out} = C^m$. For a random sample of 100 of the generated tasks per reference task, we performed manual validation to determine whether objectives (V) and (VI) are met. The fraction of tasks that satisfy these objectives is listed in the last three columns of Fig. 7. We observe that the vast majority of tasks meet the objectives, even if not by design. For H6, the fraction of tasks satisfying (VI) is low because the corresponding codes are generic enough to solve several puzzles.

Deep dive into an MCTS run. To offer more insight into the task generation process, we take a closer look at an MCTS run for task H5, shown in Fig. 8. Fig. 8a illustrates the improvement in various components of $\mathcal{F}_{score}$ as the number of MCTS iterations increases. Best tasks at different iterations are shown in Fig. 8b, 8c, 8d. As expected, the more the iterations, the better the tasks are.
We observed that $T^{\text{out}}$ can be easily extended to multiple I/O pairs by adapting techniques designed for generating diverse tasks.

### Remarks

We also ran the mutation stage by enumerating the programs within size constraints and then post-checking other constraints without Z3. This implementation leads to a run-time increase by a factor of 10 to 100 for different tasks. So, Z3 seems to be very effective by jointly considering all the constraints. As a search method, although MCTS seems computationally expensive, the actual run-time and memory footprint of an MCTS run depend on the unique traces explored (i.e., unique symbolic executions done)—this number is typically much lower than the number of iterations, also see discussion in the supplementary material. Considering the MCTS output in Figs. 8c–8d to obtain a comparable evaluation score through a random search, the corresponding number of unique symbolic executions required is at least 10 times more than executed by MCTS. We note that while we considered one I/O pair for Karel tasks, our methodology can be easily extended to multiple I/O pairs by adapting techniques designed for generating diverse tasks.

## 5 User Study and Comparison with Alternate Methods

In this section, we evaluate our task synthesis algorithm with a user study focusing on tasks H2, H4, H5, and H6. We developed an online app\footnote{https://www.teaching-blocks.cc/} which uses the publicly available toolkit of Blockly Games\cite{Games} and provides an interface for a participant to practice block-based programming tasks for HOC. Each “practice session” of the study involves three steps: (i) a reference task $T^{\text{ref}} \in \{H2, H4, H5, H6\}$ is shown to the participant along with its solution code $C^{\text{ref}}$, (ii) a new task $T^{\text{new}}$ is generated for which the participant has to provide a solution code $C^{\text{new}}$, differing from the reference solution code by a factor of 10 to 100 for different tasks. So, Z3 seems to be very effective by jointly considering all the constraints. As a search method, although MCTS seems computationally expensive, the actual run-time and memory footprint of an MCTS run depend on the unique traces explored (i.e., unique symbolic executions done)—this number is typically much lower than the number of iterations, also see discussion in the supplementary material. Considering the MCTS output in Figs. 8c–8d to obtain a comparable evaluation score through a random search, the corresponding number of unique symbolic executions required is at least 10 times more than executed by MCTS. We note that while we considered one I/O pair for Karel tasks, our methodology can be easily extended to multiple I/O pairs by adapting techniques designed for generating diverse tasks.

### Baselines and methods evaluated

We evaluated four different methods, including three baselines (SAME, TUTOR, MUTTASK) and our algorithm (SYNTASK). SAME generates tasks such that $T^{\text{in}} = T^{\text{out}}$. TUTOR produces tasks that are similar to $T^{\text{in}}$ and designed by an expert. We picked similar problems from the set of 20 Classic Maze challenge\cite{Classic Maze challenge} tasks exercising the same programming concepts: Maze 6, 9 for H2, Maze 11, 13 for H4, Maze 15, 17 for H5, and Maze 19 for H6.

MUTTASK generated tasks by directly mutating the grid-world of the original task, i.e., by moving the agent or goal by up to two cells and potentially changing the agent’s orientation. A total of 18, 20, 15, and 17 tasks were generated for H2, H4, H5, and H6, respectively. Fig. 10 shows two output tasks for H4 and illustrates the challenge in directly mutating the input task, given the high discontinuity in mapping from the space of tasks to their codes. For H4, a total of 14 out of 20 new tasks were structurally very different from the input.

SYNTASK uses our algorithm to generate tasks. We picked the generated tasks from three groups based on the size of the code mutations from which they were produced, differing from the reference solution code by $+\delta_{\text{size}}$ for $\delta_{\text{size}} \in \{0, 1, 2\}$. For H2 and H4, we randomly selected 5 tasks from each group, for a total of 15 new tasks per reference task. For H5 and H6, we selected 10 tasks from the first group ($\delta_{\text{size}} = 0$) only, due to their complexity stemming from nested constructs in their codes. We observed that TUTOR tasks for H5, H6 were also of $\delta_{\text{size}} = 0$, i.e., $C^{\text{new}} = C^{\text{size}}$. All the generated tasks picked for SYNTASK adhere to properties (I)–(VI) in Section 2.
We believe our techniques have the potential to drastically improve the success of pedagogy in block-based programming: Automatically generating new tasks that exercise specific programming concepts, while looking visually similar to input. We demonstrated the effectiveness of our methodology through an extensive empirical evaluation and user study on reference tasks from popular programming platforms. We believe our techniques have the potential to drastically improve the success of pedagogy in block-based visual programming environments by providing tutors and students with a substantial pool of new tasks. Beyond the application domain of programming education, our methodology can be used for generating large-scale datasets consisting of tasks and solution codes with desirable characteristics—this can be potentially useful for training neural program synthesis methods.

There are several promising directions for future work, including but not limited to: Learning a policy to guide the MCTS procedure (instead of running vanilla MCTS); automatically learning the constraints and cost function from a human-generated pool of problems; and applying our methodology to other programming environments (e.g., Python problems).
Broader Impact

This paper develops new techniques for improving pedagogy in block-based visual programming environments. Such programming environments are increasingly used nowadays to introduce computing concepts to novice programmers, and our work is motivated by the clear societal need of enhancing K-12 computing education. In existing systems, the programming tasks are hand-curated by tutors, and the available set of tasks is typically very limited. This severely limits the utility of existing systems for long-term learning as students do not have access to practice tasks for mastering the programming concepts.

We take a step towards tackling this challenge by developing a methodology to generate new practice tasks for a student that match a desired level of difficulty and exercise specific programming concepts. Our task synthesis algorithm is able to generate 1000’s of new similar tasks for reference tasks taken from the Hour of Code: Classic Maze challenge by Code.org and the Intro to Programming with Karel course by CodeHS.com. Our extensive experiments and user study further validate the quality of the generated tasks. Our task synthesis algorithm could be useful in many different ways in practical systems. For instance, tutors can assign new practice tasks as homework or quizzes to students to check their knowledge, students can automatically obtain new similar tasks after they failed to solve a given task and received assistance, and intelligent tutoring systems could automatically generate a personalized curriculum of problems for a student for long-term learning.

Acknowledgments and Disclosure of Funding

We would like to thank the anonymous reviewers for their helpful comments. Ahana Ghosh was supported by Microsoft Research through its PhD Scholarship Programme. Umair Z. Ahmed and Abhik Roychoudhury were supported by the National Research Foundation, Singapore and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001.

References


