Synthesizing a Progression of Subtasks for Block-Based Visual Programming Tasks

Alperen Tercan* ATERCAN@MPI-SWS.ORG
Max Planck Institute for Software Systems

Ahana Ghosh* GAHANA@MPI-SWS.ORG
Max Planck Institute for Software Systems

Hasan Ferit Eniser HFENISER@MPI-SWS.ORG
Max Planck Institute for Software Systems

Maria Christakis MARIA.CHRISTAKIS@TUWIEN.AC.AT
TU Wien

Adish Singla ADISHS@MPI-SWS.ORG
Max Planck Institute for Software Systems

Abstract

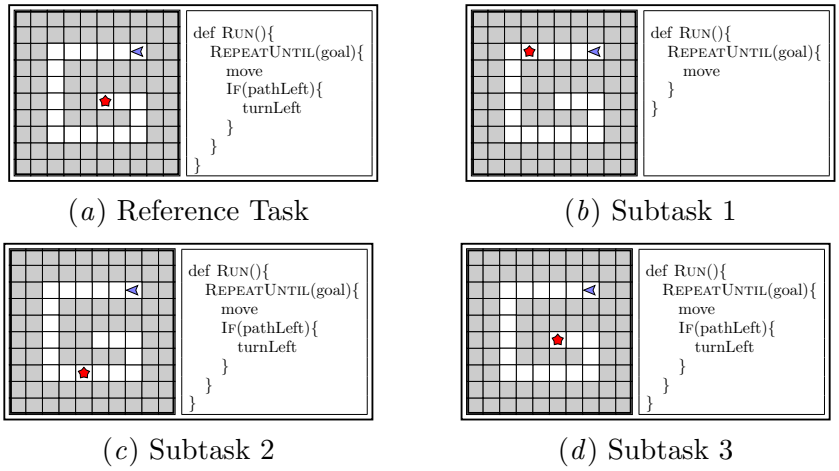
Block-based visual programming environments play an increasingly important role in introducing computing concepts to K-12 students. The open-ended and conceptual nature of these visual programming tasks make them challenging for novice programmers. A natural approach to providing assistance for problem-solving is breaking down a complex task into a progression of simpler subtasks. However, this is not trivial given that the solution codes are typically nested and have non-linear execution behavior. In this paper, we formalize the problem of synthesizing such a progression for a given reference task in a visual programming domain. We propose a novel synthesis algorithm that generates a progression of subtasks that are high-quality, well-spaced in terms of their complexity, and solving this progression leads to solving the reference task. We conduct a user study to demonstrate that our synthesized progression of subtasks can assist a novice programmer in solving tasks from the *Hour of Code: Maze Challenge* (Code.org, 2022c) by *Code.org* (Code.org, 2022a).

Keywords: block-based visual programming, task synthesis, subtasks, symbolic execution

1. Introduction

The emergence of block-based visual programming platforms has made coding more accessible and appealing to novice programmers, including K-12 students. Led by the success of programming environments like *Scratch* (Resnick et al., 2009), initiatives like *Hour of Code* (Code.org, 2022b) by Code.org (Code.org, 2022a), and online courses like *Intro to Programming with Karel the Dog* by CodeHS.com (CodeHS, 2022), block-based visual programming has become integral to introductory CS education. Importantly, in contrast to typical text-based programming, block-based visual programming reduces the burden of learning syntax and puts direct emphasis on fostering computational thinking and general problem-solving (Weintrop and Wilensky, 2015; Price and Barnes, 2017, 2015).

*Alperen Tercan did this work as part of an internship at the Max Planck Institute for Software Systems (MPI-SWS), Germany. Alperen Tercan led the implementation of the ProgreSSyn algorithm; Ahana Ghosh led the evaluation with novice programmers via a user study.



(*a*) Reference Task (*b*) Subtask 1

(*c*) Subtask 2 (*d*) Subtask 3

Figure 1: Illustration of our synthesis algorithm on Maze16 task from the *Hour of Code: Maze Challenge* (Code.org, 2022c) by *Code.org* (Code.org, 2022a). **(a)** shows visual grid of reference task $\mathtt{T}^{\text{ref}}$ and its solution code $\mathtt{C}^{\mathtt{T}^{\text{ref}},*}$ which are provided as input to our synthesis algorithm. The task requires writing code to guide the "avatar" (purple dart) to the "goal" (red star) on the grid using a maximum of 4 code blocks and block types {`RepeatUntil`,`If`, `move`,`turnLeft`,`turnRight`}. **(b), (c)**, and **(d)** show the progression of $K = 3$ subtasks for $\mathtt{T}^{\text{ref}}$ synthesized by our algorithm ProgreSyn.

Programming tasks on these platforms require multi-step deductive reasoning, which is often challenging for novices and leads to low success rates (Piech et al., 2015; Price et al., 2017; Wu et al., 2019; Efremov et al., 2020; Ghosh et al., 2022). Even seemingly simple tasks, like the one in Figure 1(*a*) from a popular porgramming platform, had less than a 50% success rate among novices, who commonly struggle with nested constructs in visual programming domains (Ghosh et al., 2024). To handle this task complexity, a natural strategy is to "break down a task" into subtasks which have been effective in various domains (Decker et al., 2019; Morrison et al., 2015), including geometry proof problems (McKendree, 1990), Parson's coding problems (Morrison et al., 2016), and robotics (Bakker and Schmidhuber, 2004). Inspired by this, we seek to develop algorithms that can synthesize a progression of subtasks for visual programming tasks.

Automatically generating such progressions for visual programming tasks is challenging due to nested code structures and their "non-linear" execution behavior on the visual grid. Existing subtasking techniques relying on "linear" behaviors in domains like path-navigation or robotics do not apply here (Bakker and Schmidhuber, 2004). Additionally, domains with numerous practice tasks, sometimes over 100, tailored to student misconceptions make manual subtask creation by experts or teachers labor-intensive (Hromkovic et al., 2017; Ahmed et al., 2020). Recent works have explored the use of generative AI and large language models (LLMs) such as ChatGPT to synthesize tasks in programming domains (Phung et al., 2023; Denny et al., 2024). However, state-of-the-art LLMs still struggle in visual programming domains as they are unable to combine spatial, logical, and programming skills (Padurean et al., 2024; Singla, 2023).

To this end, we formalize our objective of synthesizing a progression of programming subtasks and propose our synthesis algorithm, ProgresSyn. ProgresSyn overcomes the key challenges discussed above by reasoning about the execution behavior of the provided solution code on the reference task's visual grids. For example, consider the reference task in Figure 1(*a*) from the *Hour of Code: Maze Challenge* (Code.org, 2022c). Given such a reference task along with its solution code as input, we seek to synthesize a progression of subtasks with the following properties: (a) each subtask is a standalone high-quality programming task; (b) the complexity of solving a subtask in the progression increases gradually, i.e., the subtasks are well-spaced w.r.t. their complexity; (c) solving the progression would help in increasing success rate of solving the reference task. We demonstrate the effectiveness of ProgresSyn in aiding novice programmers through an online study (Section 4). We publicly share the web app used in the study[1] and the implementation of ProgresSyn[2] to facilitate future research (Sections 3 and 4).

2. Problem Setup

In this section, we introduce definitions and formalize our objective.

2.1. Preliminaries

Task space. We denote a task in a visual programming domain as the tuple $\mathtt{T} = (\mathtt{T}_{\mathrm{IO}}, \mathtt{T}_{\mathrm{CC}})$, where $\mathtt{T}_{\mathrm{IO}}$ denotes the n input-output pairs in the programming task and $\mathtt{T}_{\mathrm{CC}}$ denotes the code constraints that a solution code for the task must satisfy. For visual block-based programming domains, $\mathtt{T}_{\mathrm{IO}}$ consists of visual grids and $\mathtt{T}_{\mathrm{CC}}$ include a limit on the maximum number of blocks used along with the type of code blocks allowed. For example, the reference task shown in Figure 1(*a*) consists of a single input-output pair in the form of the visual task grid and its code constraints set a limit of at most 4 blocks where the allowed block types are {`RepeatUntil`,`If`, `move`,`turnLeft`,`turnRight`}. We denote the space of tasks as $\mathbb{T}$.

Code space and solution codes. The domain specific language (DSL) of the programming environment defines the space of codes $\mathbb{C}$ (Ahmed et al., 2020; Bunel et al., 2018). A code $\mathtt{C} \in \mathbb{C}$ is characterized by the following attributes: $\mathtt{C}_{\mathrm{depth}}$ measures the depth of the abstract syntax tree (AST) of $\mathtt{C}$, $\mathtt{C}_{\mathrm{size}}$ is the number of code blocks in $\mathtt{C}$, and $\mathtt{C}_{\mathrm{blocks}}$ is the types of code blocks in $\mathtt{C}$. We define $\mathtt{C} \in \mathbb{C}$ as a solution code for a task $\mathtt{T} \in \mathbb{T}$ if all of the following conditions hold: execution of $\mathtt{C}$ solves all the $\mathtt{T}_{\mathrm{IO}}$ visual grids of $\mathtt{T}$ and $\mathtt{C}$ satisfies all the code constraints $\mathtt{T}_{\mathrm{CC}}$. We denote a specific solution code of a task $\mathtt{T}$ as $\mathtt{C}^{\mathtt{T},*}$. The solution code for the reference task in Figure 1(*a*) is shown next to its visual grid.

Task and code complexity. We define the complexity of solving a task using a domain-specific function $\mathcal{F}^{\mathbb{T}}_{\mathrm{complex}} : \mathbb{T} \to \mathbb{R}$. We also define a code complexity measure using function $\mathcal{F}^{\mathbb{C}}_{\mathrm{complex}}$. Typically, in block-based programming environments, complexity of a code depends on the depth and size of its AST (Piech et al., 2015; Ghosh et al., 2022). Motivated by this, we define $\mathcal{F}^{\mathbb{C}}_{\mathrm{complex}} = \kappa * \mathtt{C}_{\mathrm{depth}} + \mathtt{C}_{\mathrm{size}}$, where $\kappa \in \mathbb{N}$. For instance, empty code {`Run`} has complexity $\kappa * 1 + 0$, and code {`Run`{`Repeat`(4){`move`}}} has complexity $\kappa * 2 + 2$. Using

[1] Link: https://www.teaching-blocks-subtasks.cc
[2] Link: https://github.com/machine-teaching-group/ProgresSyn

the notion of the solution code of a task, we define our complexity measure of a task in this domain as $\mathcal{F}^{\mathbb{T}}_{\text{complex}}(\mathtt{T}) = \mathcal{F}^{\mathbb{C}}_{\text{complex}}(\mathtt{C}^{\mathtt{T},*})$.

2.2. Objective

Our goal is to synthesize a progression of subtasks for a given reference task, such that solving this progression increases the success rate of solving the reference task. We use the increased success rate as a proxy for measuring the helpfulness of our synthesized progression. Next, we introduce the notion of progression of subtasks and its complexity, and then formalize our synthesis objective.

Progression of subtasks. For a reference task $\mathtt{T}^{\text{ref}}$, its solution code $\mathtt{C}^{\text{ref},*}$, and a fixed budget K, we denote a progression of subtasks for $\mathtt{T}^{\text{ref}}$ as a sequence $\omega(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\text{ref},*}, K) := ((\mathtt{T}^k, \mathtt{C}^{k,*}))_{k=1,2,\ldots,K}$ where the following holds $\forall k$: (a) $\mathtt{C}^{k,*}$ is the solution code of $\mathtt{T}^k$; (b) $|\mathtt{T}^k_{\text{IO}}| \leq |\mathtt{T}^{\text{ref}}_{\text{IO}}|$. We also have $\mathtt{T}^K = \mathtt{T}^{\text{ref}}$ and $\mathtt{C}^{K,*} = \mathtt{C}^{\text{ref},*}$. We denote the set of all such progressions of K-subtasks as $\Omega(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\text{ref},*}, K)$.

Complexity of a progression of subtasks. We capture the complexity of a progression of subtasks using the function $\mathcal{F}^{\Omega}_{\text{complex}}$. Specifically, $\mathcal{F}^{\Omega}_{\text{complex}}(\omega; \mathtt{T}^{\text{ref}}, \mathtt{C}^{\text{ref},*}, K)$ for a given reference task $\mathtt{T}^{\text{ref}}$ captures the worst case complexity jump in the solution codes of subtasks of ω. More formally:

$$\mathcal{F}^{\Omega}_{\text{complex}}(\omega; \mathtt{T}^{\text{ref}}, \mathtt{C}^{\text{ref},*}, K) = \max_{k \in \{1,\ldots,K\}} \left\{ \min_{k' \in \{0,\ldots,k-1\}} \{\mathcal{F}^{\mathbb{C}}_{\text{complex}}(\mathtt{C}^{k,*}) - \mathcal{F}^{\mathbb{C}}_{\text{complex}}(\mathtt{C}^{k',*})\} \right\} \quad (1)$$

where code $\mathtt{C}^{k,*}$ denotes solution code of subtask k in ω and $\mathtt{C}^{0,*}$ denotes empty code $\{\texttt{RUN}\}$.

Our synthesis objective. Our objective is to synthesize a progression of K subtasks for a given reference task $\mathtt{T}^{\text{ref}}$ with minimal complexity w.r.t. Equation 1. Our formalism is based on the intuition that lowering complexity reduces the cognitive load of solving the progression, while still assisting problem-solving of the reference task (McKendree, 1990). More formally, we seek to generate a progression of subtasks based on the following:

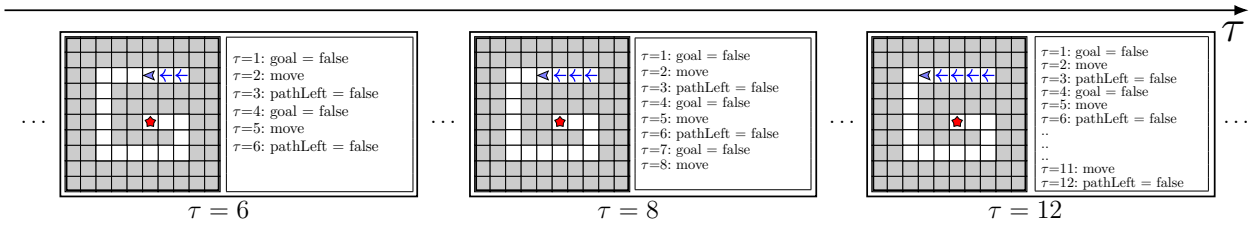
$$\text{Minimize}_{\omega \in \Omega(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\text{ref},*}, K)} \; \mathcal{F}^{\Omega}_{\text{complex}}(\omega; \mathtt{T}^{\text{ref}}, \mathtt{C}^{\text{ref},*}, K). \quad (2)$$

Furthermore, our synthesized progression of subtasks must have the following properties:

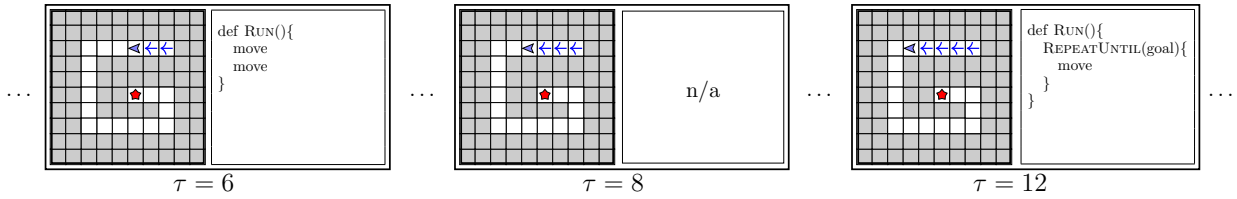
1. The subtasks are of high quality. We define the quality of a task using domain-specific function $\mathcal{F}^{\mathbb{T}}_{\text{qual}} : \mathbb{T} \to \mathbb{R}$ that measures the general quality of a task (Ahmed et al., 2020).
2. Visual grids of subtasks are minimal modifications of the reference task's visual grid(s).
3. Subtasks in the progression are diverse, i.e., as different from each other as possible.

3. Our Synthesis Algorithm

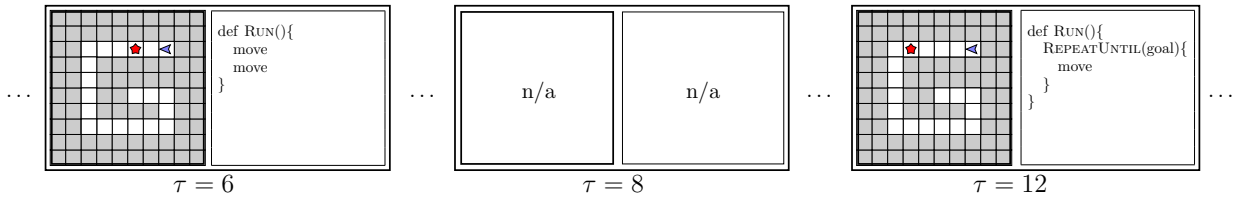
We present our algorithm, ProgresSyn, for synthesizing a progression of K subtasks for a reference task and its solution code $(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\mathtt{T}^{\text{ref}},*})$. We first present our algorithm for tasks with a single input-output pair (specifically, a single visual grid). Then, we discuss extensions of the algorithm for tasks with multiple input-output pairs. The algorithm has 4 stages which are detailed below. Figure 2 illustrates stages 1–3 applied to $\mathtt{T}^{\text{ref}}$ shown in Figure 1(*a*).



(*a*) Stage 1: Execution trace of solution code on visual grid



(*b*) Stage 2: Post processing of execution trace based on code quality and code validity



(*c*) Stage 3: Modifying the visual grids of the processed execution trace

Figure 2: Illustration of Stages 1–3 of our algorithm on reference task shown in Figure 1(*a*). "n/a" denotes invalid codes and tasks. In Stage 4, we select a progression of $K = 3$ subtasks shown in Figures 1(*b*), 1(*c*), and 1(*d*).

Stage 1: Execution trace on the single grid (Figure 2(*a*)). In this stage, we "linearize" the solution code $\mathtt{C}^{\mathtt{T}^{\text{ref}},*}$ by obtaining the full execution trace of this code on the visual grid $\mathtt{T}^{\text{ref}}_{\text{IO}}$. Specifically, the execution is represented as a series of the following pairs: the sequence of code commands executed and the state of the visual task grid after their execution. As an example, in Figure 2(*a*) we show three pairs in the execution trace for the reference task from Figure 1(*a*) at steps $\tau = 6, 8,$ and 12.

Stage 2: Post-processing the trace based on code validity/quality (Figure 2(*b*)). This stage filters the execution trace and generates potential solution codes of the subtasks. We begin by filtering those elements of the trace whose code commands lead to invalid codes. For example, in Figure 2(*b*), code at step $\tau = 8$ is filtered because the corresponding code commands in Stage 1 terminate on `move`, which is in the middle of the body of loop `RepeatUntil` of $\mathtt{C}^{\mathtt{T}^{\text{ref}},*}$. For the remaining elements, we generate codes from their code commands – these codes eventually serve as solution codes for the subtasks.

Stage 3: Modifying grids in the trace via symbolic execution (Figure 2(*c*)). In this stage, we generate task grids for each of the codes from the sequence obtained in Stage

2. We achieve this using symbolic execution techniques (King, 1976; Ahmed et al., 2020). Specifically, during symbolic execution, we make minimal modifications to the grids of the subtasks w.r.t. $\mathtt{T}^{\text{ref}}_{\text{IO}}$ and generate high-quality subtasks. For example, consider step $\tau = 6$ in Figures 2(b) and 2(c) . After symbolic execution on the code from Figure 2(b), we obtain the grid in Figure 2(c). Observe how the "goal" (red star) is set at the final location of "avatar" on the grid from Figure 2(b) to generate the grid in Figure 2(c).

Stage 4: Generating subtasks via subsequence selection. In the final stage, we generate a progression of K subtasks by picking a sequence of size K from the processed execution trace in Stage 3. Here, we seek to minimize the task complexity jump between consecutive subtasks, while still being of high quality and diverse. We obtain this subsequence using constrained optimization methods on the objective function defined based on the task complexity jump between consecutive subtasks. More specifically, from the processed execution trace we obtain the set of all subsequences of length K which is denoted as $\Omega(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\text{ref},*}, K)$. Using Ω, we optimize for Equation 2 to obtain our final progression. We use dynamic programming techniques to optimize our objective.

Extension to tasks with multiple input-output pairs. So far, we have presented our algorithm for a task with a single input-output (IO) pair, i.e., $n = 1$. Its extension to tasks with multiple IO pairs can be found in the longer version of our paper (Tercan et al., 2023) – next, we briefly outline the key ideas of this extension. For a task with $n > 1$ input-output (IO) pairs, our algorithm first progressively introduces these IO pairs in the form of a sequence of subtasks, i.e., the first subtask contains one IO pair, the second subtask contains two IO pairs, and the final subtask contains all the n IO pairs. After generating this sequence, the first subtask with one IO pair is further decomposed into $K - n + 1$ subtasks by applying Stages 1–4 described above. Together, they form the final progression of subtasks for the multi-IO task. This overall generation process is optimized to maximize the objective defined in Equation 2. We apply this extension to the *Karel* programming domain (Pattis, 1995) where tasks have multiple IO pairs; see an illustrative example in (Tercan et al., 2023).

4. Assisting Novice Human Programmers

We evaluate the utility of ProgreSSyn in assisting novice programmers in solving visual programming tasks on a popular platform. We first present the research questions around which our study is centered followed by the subtasking methods evaluated. Next, we present details of the online platform, our user study setup, and the participants of the user study. Finally, we present the results of the study and discuss some of its limitations.

Research questions. We center our user study around the following research questions *(RQs)* to measure the efficacy of ProgreSSyn: (i) *RQ1: Usefulness of subtasking.* Does solving a progression of subtasks increase the success rate on the reference task? (ii) *RQ2: Well-spaced code complexity.* Do progressions with subtasks that are well spaced w.r.t. their code complexity improve the success rate more in comparison to progressions that violate this property? (iii) *RQ3: Retaining visual context of the reference task.* Do the progression of subtasks that retain the visual context of the reference task in their grids improve the success rate more in comparison to progressions that violate this property?

Subtasking methods evaluated. To answer these research questions, we evaluate different subtasking methods and compare them with ProgreSSyn for $K = 3$. Methods Same-TC and Same-C (collectively called Same) generate a progression of $K = 3$ subtasks which are not well spaced w.r.t their code complexity. Specifically, Same-TC contains $K = 3$ subtasks all of which are the same as the reference task. Same-C minimally alters the visual grids of the subtasks while their solution codes remain the same as that of the reference task. Methods Crafted-v1 and Crafted-v2 (collectively called Crafted) consist of a human-generated progression of $K = 3$ subtasks that are well spaced w.r.t. their code complexity but have task grids not retaining the visual context of the reference task. As a default setting without any subtasking, Default captures the setting where a participant is presented with $\mathtt{T}^{\text{ref}}$ and given 10 tries to solve it. We note that a participant spends up to 40 problem-solving attempts in all the subtasking methods (Same, Crafted, ProgreSSyn) and up to 10 problem-solving attempts without subtasking (Default).

Online platform. We developed a web app for our subtasking framework (see link in Footnote 2) where we enabled the above-mentioned subtasking methods. Our app uses the publicly available toolkit of Blockly Games (Games, 2022) and provides an interface for a participant to solve a block-based visual programming task through a progression of subtasks. We used two reference tasks for our study from the *Hour of Code: Maze Challenge* by *Code.org* (Code.org, 2022c): Maze08 and Maze16 (illustrated in Figure 1(*a*)).

Study setup. We used the online platform described above for conducting the study. Before logging into the app, each participant is presented with a 4-minute instructional video about block-based visual programming to familiarize themselves with the platform. After logging into the app, a participant gets assigned a reference task $\mathtt{T}^{\text{ref}} \in \{\text{Maze08}, \text{Maze16}\}$ and one of the subtasking methods at random. These elements constituted a "session" for a participant. Specifically, a participation session comprised of the following steps:

1. Step 1: The participant is shown the reference task and given 10 attempts to solve it. If they are successful, they exit the platform; otherwise, they proceed to the next step.
2. Steps 2*a* and 2*b*: The participant is presented with the first two subtasks from the progression synthesized by the assigned subtasking method, and given 10 attempts to solve each subtask.
3. Step 3: The participant is presented with the third subtask (i.e., the reference task itself from Step 1 again), and given 10 attempts to solve it.

Participants. We recruited participants for the study from Amazon Mechanical Turk; an IRB approval had been obtained for the study. The participants were US-based adults, without expertise in block-based visual programming. The study took at most 30-35 minutes to complete per participant. Due to the costs involved (over 4 USD per task for a participant), we used only two reference tasks for the study.

Results. We present detailed results in Figure 3 and analyze them w.r.t. our RQs. In total, we had about 600 participation sessions across two tasks. To validate the *usefulness of subtasking (RQ1)*, we compare the success rate on reference tasks for methods Same-TC and ProgreSSyn. We find a 5.1% increase in the success rate for ProgreSSyn, suggesting the usefulness of subtasks in problem-solving. To investigate the effect of *well-spaced code complexity (RQ2)* in a progression, we compare the success rates for Same-C and

Method	Total participants			Fraction succeeded		
	All	Maze08	Maze16	All	Maze08	Maze16
Default	114	57	57	0.605	0.807	0.403
Same-TC	114	57	57	0.667	0.842	0.491
Same-C	116	59	57	0.672	0.847	0.491
Same	230	116	114	0.669	0.845	0.491
Crafted	235	117	118	0.647	0.838	0.458
ProgreSyn	117	58	59	0.701	0.862	0.542

Figure 3: Results for Maze08 and Maze16. "All" shows aggregated results over the two tasks.

ProgreSyn. We find that ProgreSyn outperforms Same-C by 4.3%. This suggests the importance of using progressions with well-spaced code complexity. To investigate the effect of *retaining the visual context of the reference task (RQ3)* in the progression, we compare Crafted and ProgreSyn. We find a 8.3% increase in success rate for ProgreSyn compared to Crafted, suggesting the importance of retaining the visual context. Furthermore, we find that Same also outperforms Crafted. We hypothesize that this is because Crafted synthesizes subtasks that are visually very different from the reference task, possibly making the progression more confusing and leading to lower success rates on the reference task.

Limitations and possible extensions. Next, we discuss a few limitations of our current study. Our study was limited to about 600 participants given the high costs involved and the reported results are not statistically significant. A larger scale user study, with substantially more number of participants, would be needed to further validate the statistical significance of the results. Furthermore, we conducted our study with adult novice programmers. In the future, it would be important to conduct longitudinal studies with students in classrooms to measure the pedagogical value of our algorithm.

5. Conclusions and Future Work

In this paper, we tackled the problem of synthesizing a progression of subtasks for a given block-based programming task. Our novel algorithm, ProgreSyn, automatically synthesizes this progression using execution traces and symbolic execution techniques. We showcased its effectiveness in aiding novice programmers with tasks from a popular programming platform. Looking ahead, several promising directions for future work emerge. First, we could customize the subtask progression based on the novice programmer's latest code attempt to address their misconceptions. Alternatively, we could use their attempt to direct them to a specific subtask within the progression generated for the reference task. Second, we could explore extensions of our approach to more complex block-based visual programming domains and text-based domains like Python. Third, it would be interesting to further explore learning-based strategies and generative AI models for automated subtask generation.

Acknowledgments

We would like to thank the reviewers for their feedback. Ahana Ghosh acknowledges support by Microsoft Research through its PhD Scholarship Programme. Funded/Co-funded by the European Union (ERC, TOPS, 101039090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

Umair Z. Ahmed, Maria Christakis, Aleksandr Efremov, Nigel Fernandez, Ahana Ghosh, Abhik Roychoudhury, and Adish Singla. Synthesizing Tasks for Block-based Programming. In *NeurIPS*, 2020.

Bram Bakker and Jürgen Schmidhuber. Hierarchical Reinforcement Learning Based on Subgoal Discovery and Subpolicy Specialization. In *Intelligent Autonomous Systems*, 2004.

Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *ICLR*, 2018.

CodeHS. CodeHS.com: Teaching Coding and Computer Science. https://codehs.com/, 2022.

Code.org. Code.org: Learn Computer Science. https://code.org/, 2022a.

Code.org. Hour of Code Initiative. https://hourofcode.com/, 2022b.

Code.org. Hour of Code: Classic Maze Challenge. https://studio.code.org/s/hourofcode, 2022c.

Adrienne Decker, Lauren E. Margulieux, and Briana B. Morrison. Using the SOLO Taxonomy to Understand Subgoal Labels Effect in CS1. In *ICER*, 2019.

Paul Denny, Sumit Gulwani, Neil T. Heffernan, Tanja Käser, Steven Moore, Anna N. Rafferty, and Adish Singla. Generative AI for Education (GAIED): Advances, Opportunities, and Challenges. *CoRR*, abs/2402.01580, 2024.

Aleksandr Efremov, Ahana Ghosh, and Adish Singla. Zero-shot Learning of Hint Policy via Reinforcement Learning and Program Synthesis. In *EDM*, 2020.

Blockly Games. Games for Tomorrow's Programmers. https://blockly.games/, 2022.

Ahana Ghosh, Sebastian Tschiatschek, Sam Devlin, and Adish Singla. Adaptive Scaffolding in Block-based Programming via Synthesizing New Tasks as Pop Quizzes. In *AIED*, 2022.

Ahana Ghosh, Liina Malva, and Adish Singla. Analyzing–Evaluating–Creating: Assessing Computational Thinking and Problem Solving in Visual Programming Domains. In *SIGCSE*, 2024.

Juraj Hromkovic, Giovanni Serafini, and Jacqueline Staub. XLogoOnline: A Single-Page, Browser-Based Programming Environment for Schools Aiming at Reducing Cognitive Load on Pupils. In *ISSEP*, 2017.

James C. King. Symbolic Execution and Program Testing. *Communications of ACM*, 1976.

Jean McKendree. Effective Feedback Content for Tutoring Complex Skills. *HCI*, 1990.

Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *ICER*, 2015.

Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. Subgoals Help Students Solve Parsons Problems. In *SIGCSE*, 2016.

Victor-Alexandru Padurean, Georgios Tzannetos, and Adish Singla. Neural Task Synthesis for Visual Programming. *Transations of Machine Learning Research*, 2024.

Richard E Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming.* John Wiley & Sons, Inc., 1995.

Tung Phung, Victor-Alexandru Padurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors. In *ICER V.2*, 2023.

Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas J. Guibas. Autonomously Generating Hints by Inferring Problem Solving Policies. In *L@S*, 2015.

Thomas W. Price and Tiffany Barnes. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *ICER*, 2015.

Thomas W. Price and Tiffany Barnes. Position Paper: Block-Based Programming Should Offer Intelligent Support for Learners. *IEEE Blocks and Beyond Workshop*, 2017.

Thomas W. Price, Rui Zhi, and Tiffany Barnes. Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior. In *AIED*, 2017.

Mitchel Resnick et al. Scratch: Programming for All. *Communications of the ACM*, 2009.

Adish Singla. Evaluating ChatGPT and GPT-4 for Visual Programming. In *ICER V.2*, 2023.

Alperen Tercan, Ahana Ghosh, Hasan Ferit Eniser, Maria Christakis, and Adish Singla. Synthesizing a Progression of Subtasks for Block-Based Visual Programming Tasks. *CoRR*, abs/2305.17518, 2023.

David Weintrop and Uri Wilensky. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-Based Programming. In *IDC*, 2015.

Mike Wu, Milan Mosse, Noah D. Goodman, and Chris Piech. Zero Shot Learning for Code Education: Rubric Sampling with Deep Learning Inference. In *AAAI*, 2019.