Integrating Intelligent Feedback into Block Programming Environments

Thomas W. Price  {twprice@ncsu.edu}

Abstract
Block Programming Environments (BPEs) are effective tools for teaching novices to program. They are popular in informal settings, where instructors may not be available. Students who get stuck may give up in the absence of instructor assistance. BPEs generally lack contextualized feedback to help these struggling students. The Hint Factory allows students to make open-ended projects, such as apps and games (Fig. 2). They support “tinkering,” experimentation and various problem solving strategies. This leads to a very “sparse” interaction network, with little overlap among solutions. It is difficult to match a student requesting a hint to previous states in the network. Previous work has overcome some of these challenges for well-defined problems using state canonicalization and path construction. Further work is needed to generate hints for larger, creative and open-ended problems.

Challenges of BPEs
• BPEs allow students to make open-ended projects, such as apps and games (Fig. 2)
• They support “tinkering,” experimentation and various problem solving strategies
• This leads to a very “sparse” interaction network, with little overlap among solutions
• It is difficult to match a student requesting a hint to previous states in the network
• Once matched, there are fewer possible solution paths from which to generate hints
• Previous work has overcome some of these challenges for well-defined problems using state canonicalization and path construction
• Further work is needed to generate hints for larger, creative and open-ended problems

Current Progress
• Collected data from 31 middle school novices programming a simple game
• 17 used a BPE and 14 used textual coding
• Compared student performance; results suggest the BPE offers an advantage
• Analyzed the interaction network of students using the BPE
• Student solutions were too diverse to apply current hint generation techniques directly
• 95% of states were unique
• On average, only 17% of students’ states had a match among other students (Fig. 4)
• Using the subtree approach, on average 62% of students’ states had some subtree match
• This represents an upper bound for the usefulness of subtree matching

Future Work
• Collect a larger corpus of student programs
• Refine the subtree approach and explore additional strategies for reducing sparseness
• Generate a set of hints and integrate them into the programming environment
• Collect a second dataset from students who have on-demand hints available
• Collect pre- and post- surveys to measure self-efficacy
• Compare students performance and self-efficacy with and without data-driven hints

Introduction
• Block Programming Environments (BPEs) are effective tools for teaching novices to program
• They are popular in informal settings, where instructors may not be available
• Students who get stuck may give up in the absence of instructor assistance
• BPEs generally lack contextualized feedback to help these struggling students

Goal: to provide on-demand, data-driven hints to students working in BPEs:
• Focus specifically on supporting open-ended tasks, such as creating games and apps
• Generate hints automatically from previous students’ programs
• Support diverse solutions to a given programming problem

The Hint Factory
• A system for generating data-driven hints
• Input is an interaction network, comprised of previous student solution paths (Fig. 1)
• Vertices represent student “code states”
• Edges represent student actions (e.g. adding a block), weighted by their frequency
• A student requesting a hint is matched to an observed state and directed towards the goal
• Paths that historically led to successful completion of the problem are favored

Figure 1 An interaction network of students working on a simple programming task. Vertices represent code states and edges represent student actions.

Figure 2 An example of a typical open-ended project in a BPE, where students make a Whack-a-Mole game.

One Strategy: Subtrees
We can increase state overlap by breaking a problem down into smaller subtasks:
• Represent a student’s code as an abstract syntax tree (AST)
• Isolate the subtree which encompasses the student’s most recent actions
• Search for matching subtrees in previously observed states on known solution paths
• Advise the student to modify their subtree as the successful student did (Fig. 3)
• Advantage: greatly increases the probability of matching a previous student
• Limitation: ignores possibly relevant code outside the subtree, lowers hint quality

Figure 3 Having observed Student A (left), the subtree approach is able to generate a hint for Student B (right), despite the two students having very different programs. The hint subtree uses the [go to x y] block as its root.

Figure 4 A distribution of the proportion of student states with hints available using exact state matching and subtree matching. Does not account for hint quality.