Codepourri: Creating Visual Coding Tutorials Using A Volunteer Crowd Of Learners

Mitchell Gordon and Philip J. Guo
Department of Computer Science
University of Rochester
Rochester, NY 14627
mgord12@u.rochester.edu, pg@cs.rochester.edu

***Abstract*—A common way to learn is by studying written step-by-step tutorials such as worked examples. However, tutorials for computer programming can be tedious to create since a static text-based format cannot convey what happens as code executes. We created a system called Codepourri that enables people to easily create visual coding tutorials by annotating steps in an automatically-generated program visualization. Using Codepourri, we developed a novel crowdsourcing workflow where learners who are visiting an educational website (www.pythontutor.com) collectively create a tutorial by annotating execution steps in a piece of code and then voting on the best annotations. Since there are far more learners than experts, using learners as a crowd is a potentially more scalable way of creating tutorials. Our experiments with 4 expert judges and 101 learners adding 145 raw annotations to two pieces of textbook Python code show the learner crowd's annotations to be accurate, informative, and containing some insights that even experts missed.**

Keywords*—*program visualization, worked examples, crowdsourcing, tutorial creation, CS education

I. Introduction

Decades of computing education research has shown that programming is hard to learn since it is a cognitively complex task requiring one's mind to manipulate abstract and dynamic state [1], [2]. Novices often struggle to develop robust mental models of code execution [3] and are susceptible to hundreds of common misconceptions about how their code works [2].

One effective way to learn programming is by studying tutorials of how people (e.g., peers or instructors) approach and solve coding problems. These tutorials – sometimes called worked examples or worked-out examples [4] – capture a person's thoughts as they walk through an example piece of code one step at a time. By studying these tutorials, a novice is able to learn vicariously from the insights of others and avoid forming misconceptions. This phenomenon, called the worked-example effect [4], [5], [6], [7], is well-studied in educational psychology; it states that studying worked examples reduces cognitive load on novices so that they can devote more of their attention to understanding concepts rather than, say, grappling with syntax errors as they try to write their own code.

Worked examples are pervasive in subjects such as math and physics, but surprisingly, are not as common in programming. One possible reason for this is that it is tedious to create worked examples for code. A tutorial creator can either:

- Annotate each line of source code with explanatory comments. This approach is convenient but limited in power since source code is static and cannot capture runtime semantics such as stack frames, data structure values, and pointer aliasing.
- Manually draw diagrams of what happens step-by-step during code execution and then annotate those diagrams. This approach is powerful but tedious.

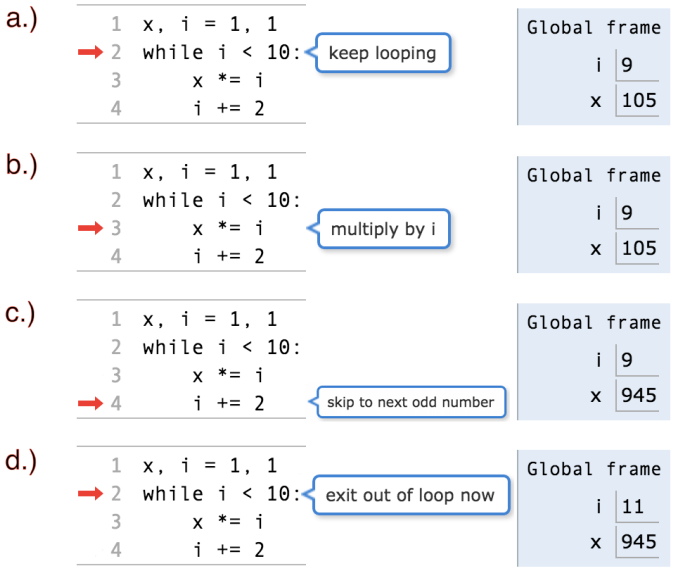

Fig. 1. Codepourri is a system that allows a tutorial creator to write code, automatically visualize its execution state, and annotate each step of execution (blue bubbles) to provide a step-by-step walkthrough. This simplified screenshot shows four annotated steps in a Python program with a `while` loop and two global variables, `i` and `x`, with their values shown to the right.

To make it easier to create these sorts of tutorials, we created a tool called Codepourri, which extends an automated program visualization tool [8] with inline annotations. Codepourri enables a tutorial creator to write code, execute it to automatically produce a step-by-step visualization of the stack frames, variables, data structure values, and pointers present during execution, and then annotate any line of code at any step with notes in pop-up bubbles. Codepourri tutorials are web pages that can be embedded within other online resources.

The *visual coding tutorials* created using Codepourri capture both static and dynamic properties of code, and does not require the tutorial creator to tediously draw diagrams. The learner can play back these tutorials step-by-step like a video

and study the annotations at each step. For instance, Figure 1 shows four steps in a simple Python code tutorial involving a `while` loop and two global variables, `i` and `x`, whose values change throughout execution. Note that the annotations in Figure 1a. and Figure 1d. are both next to line 2, but the context is different since those represent two different steps in execution (the loop condition in the latter is false, so it should exit out of the loop). This kind of subtlety is hard to capture by simply commenting a piece of source code.

We envision instructors using Codepourri to create visual coding tutorials for their own electronic lecture notes and online digital textbooks. However, a more unusual but promising use-case that we explore in this paper is to instead *use a crowd of volunteer learners to create visual coding tutorials*.

Why try to use learners to create tutorials? The primary reason is scalability: there are far more learners than experts in any domain, so leveraging learners drastically expands the pool of tutorial creators and means that more tutorials can get made. Novices in particular benefit from studying multiple examples [9] to get different perspectives on the same piece of code. A secondary reason is that learners can sometimes provide insights that experts miss, because they are currently in the process of learning the material themselves. In contrast, experts suffer from the expert blind spot [10] because they forgot what it was like to be a novice, so they can have a hard time explaining concepts at a basic level suitable for novices.

We created a crowdsourcing workflow where learners on a popular Python education website, Online Python Tutor [8], can use Codepourri to annotate pieces of example code and vote on the best annotations to create a tutorial out of them. To our knowledge, we are the first to leverage a crowd of anonymous volunteer learners to create code tutorials. Our unique crowd is unlike using learners in a formal course because courses are not anonymous, and it is also unlike using a paid worker crowd on Amazon Mechanical Turk.

To evaluate the efficacy of Codepourri and learner-based crowdsourcing, we uploaded two pieces of Python code from an introductory programming textbook [11] to the Codepourri interface on the Online Python Tutor website. In one week, 101 learners contributed 145 raw annotations. Four expert judges (three CS professors and one teaching assistant) rated 64.8% of raw annotations as being accurate, and 17% of those contained surprising insights that even the experts did not think to provide (i.e., due to the expert blind spot [10]). When these raw annotations were aggregated together using crowdsourced voting mechanisms to create a single tutorial with the best annotation shown at each execution step, experts judged that tutorial to be comparable in quality to the one that they created.

Our results show that this workflow is promising for basic code tutorials of the sort used in introductory programming courses. In future work, we plan to extend it to handle more sophisticated code such as those in advanced courses, perhaps blending together annotations from experts and novices to combine the best insights from both populations.

This paper makes the following contributions:

- A system called Codepourri that enables people to easily create visual coding tutorials that capture both static and dynamic properties of code execution.
- A novel crowdsourcing workflow where volunteer learners use Codepourri to collectively create tutorials.
- Experimental results showing that crowd-created tutorials are comparable to or better than expert-created ones, for code taken from an introductory textbook.

II. Background and Related Work

The educational inspiration behind Codepourri is *worked examples* – annotated step-by-step tutorials of how to solve a problem such as a math or physics derivation. According to cognitive load theory, asking novice learners to solve problems on their own too early strains their working memory; instead, they can learn better by first studying a step-by-step tutorial of how an expert solves those problems [4], [5]. Worked examples are pervasive in subjects such as math and physics but have not been used much in teaching programming. Pirolli used worked examples to teach recursion in Lisp [7], and more recently, Margulieux et al. used worked examples to teach the Android App Inventor visual language [6]. One reason why worked examples are not more common in teaching programming is that it is tedious to manually draw the data structures at each step of code execution and then label all of the components with explanatory text. Codepourri makes it easy to create worked examples for computer programming since it automatically visualizes execution state [8] and enables the creator to annotate every line of code and step. However, note that Codepourri works only on code that has already been written (e.g., as part of a textbook or lecture notes) and cannot be used to build up worked examples from a blank slate.

One phenomenon that limits the effectiveness of worked examples is the *split-attention effect* [12], which occurs when explanatory text is separated from diagrams, thereby forcing students to split their attention by glancing back and forth. Codepourri reduces this effect by allowing people to directly create and view annotations beside the relevant line of code.

Tutorials such as worked examples are usually created by expert instructors, but in this paper we explore the idea of instead using a crowd of learners to create tutorials. The most relevant project in this area is Crowdy [13], which provides an interface for learners to annotate educational videos with *subgoal labels* [14] – higher-level summaries of each subsection within the video. Crowdy introduced the concept of *learnersourcing*, which is a variant of crowdsourcing that recruits a crowd of unpaid learners rather than, say, paid workers on Amazon Mechanical Turk or unpaid volunteers in citizen science [15] or CAPTCHAs. The main design challenge in such a system is to motivate learners to work for free. Aside from general altruism, a more direct motivation is educational research that shows how the act of annotating encourages *self-explanation* (i.e., explaining a concept to yourself to reinforce your own knowledge), which can lead to comparable learning gains as some forms of direct problem solving [14], [16], [17].

VidWiki [18] is a tool similar to Crowdy that allows a crowd to directly write text and draw diagrams as live layers on top of existing online videos. And NB [19] allows a crowd to select text within a PDF document and start embedded discussions in the margin besides that text. Although these systems are not confined to use only by novices, the case studies presented in those papers involved using novices to annotate educational videos and lecture notes, respectively.

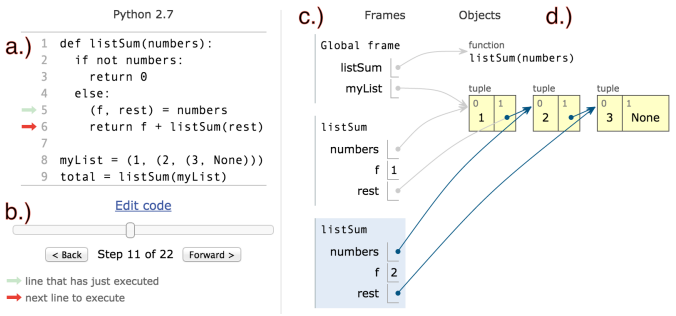

Fig. 2. We built Codepourri upon Online Python Tutor, a Web-based program visualization tool that allows the user to: a.) write code, b.) single-step through all execution steps, and see a visualization of c.) stack frames and d.) heap objects. Here it is visualizing recursive function calls to traverse a linked list.

To our knowledge, no prior work has applied learnersourcing to computing education, so we are the first to use a learner crowd to create step-by-step tutorials for computer programs. Perhaps the reason why nobody has explored this domain yet is that there is no convenient mechanism to draw and annotate each step of program execution. Simply annotating a piece of source code does not capture runtime semantics that are important for worked examples. Our Codepourri tool provides this necessary mechanism, which expands a static piece of code into a video-like animation that can be stepped through and annotated, similar to an educational video.

Finally, an alternative way to annotate execution steps is to have the computer automatically generate annotations by, say, analyzing the current execution context and producing a natural-language description of it. This technique is used in pedagogical programming environments such as Gidget [20] and several of those described in surveys by Kelleher and Pausch [21] and by Sorva et al. [22]. Automatically-generated annotations are more scalable than manually-generated ones but are only as insightful as the rules and vocabulary that the creators originally programmed into them.

III. Prior Work: Online Python Tutor

We built Codepourri upon a Web-based automated program visualization tool called Online Python Tutor [8]. To use this tool, the user first visits www.pythontutor.com and writes code directly in their browser (Figure 2a.). Despite its legacy name, Online Python Tutor supports coding in five popular languages – Python (v2 and v3), Java, JavaScript, TypeScript, and Ruby – although the majority of users write Python. When the user presses the "Visualize Execution" button on the page, their code is sent to the Online Python Tutor server to execute in a sandbox. The server sends a complete execution trace back to the user's browser, usually in less than two seconds. The user can then step forward and backward through all execution steps using a navigation slider and buttons (Figure 2b.). At each step, the user sees a detailed visualization of their code's runtime state, which includes stack frames, variables, data structures, and pointers (Figure 2c. and d.).

Online Python Tutor can compactly visualize arbitrary heap graphs consisting of custom nested and linked data structures (e.g., Figure 2d). These automatically-generated visualizations mimic what people manually draw on the board or on paper when explaining code execution state.

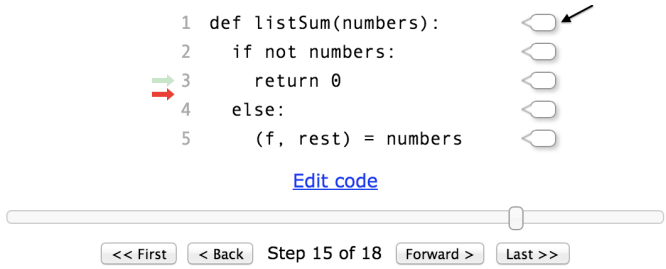

Fig. 3. Codepourri augments Online Python Tutor by placing a bubble next to each line of code (see black arrow). When the user clicks on a bubble, it expands into an editable text field. Each step gets its own independent set of bubbles. The code in this figure has 18 sets of 5 bubbles (18 steps x 5 lines).

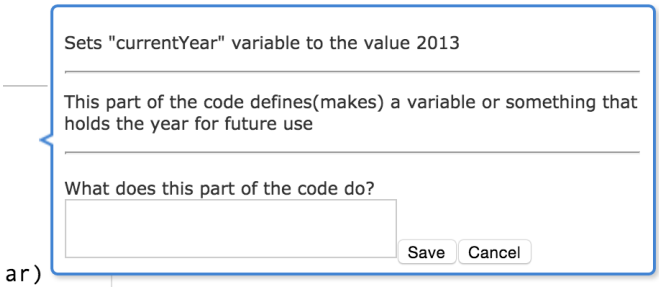

Fig. 4. Once a user has clicked on a bubble, they are able to view all other users' annotations and also to add their own.

IV. The Codepourri System

To enable people to easily create programming tutorials, we created a system called Codepourri (Figure 5) atop Online Python Tutor. Codepourri displays a piece of code and allows the user to step through its execution, add annotations, view annotations left by other users, and vote on what they think are the best ones (see the next section on crowdsourcing for more details). The ideas behind Codepourri are programming language agnostic, and the current system works on the five languages supported by Online Python Tutor.

Traditional code annotation involves leaving comments inside the code itself and does not take into account what happens during execution. In contrast, Codepourri allows users to add annotations to the visualizations at each step of execution. This is done by using a model that creates pop-up bubbles for annotations, which depend both on the current line of code and the current execution step. Figure 3 shows a piece of code with 5 lines and 18 execution steps; there are $5 \times 18 = 90$ possible bubbles. Pop-up bubbles have additional advantages over inline comments: They do not change the spacing or appearance of the original code and can be selectively hidden.

Code annotations that depend upon the state of execution are problematic for the traditional inline method of commenting code, because a given line of code often executes more than once (see Figure 1a. and Figure 1d.), but an ordinary text comment cannot easily reference such execution context. Therefore, rather than displaying comments inline in the code itself, Codepourri displays clickable expanding bubbles to the right of each line of code (Figure 3). To add a new annotation for a given line, the user clicks on the bubble corresponding to that line to bring a up list of past user annotations, along with a text input field to enter their own. The user can step forward

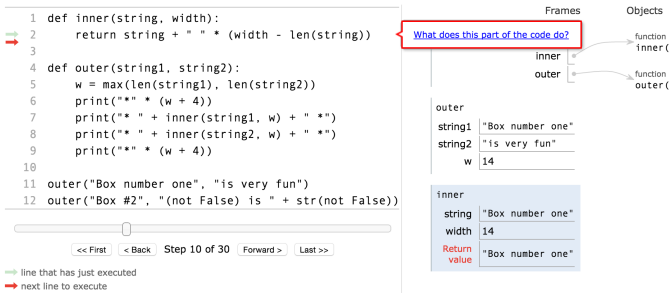

Fig. 5. The crowdsourcing task prompt for having a crowd of learners annotate execution steps using Codepourri. To add a new annotation for a given line of code, click on the bubble (outlined in red) at the right of that line to bring up a list of past annotations along with an editable text box.

and backward to view execution state and add annotations to any bubble that they see. To quickly preview the annotations for a given bubble, the user can mouse over the bubble to expand it into a scrollable list (Figure 4).

We implemented Codepourri using standard Web technologies. Once a tutorial has been created, it can be published as an ordinary dynamic web page that can be embedded in online resources such as electronic lecture notes or textbooks. A learner can step through execution, see visualizations of data structures, and study the annotations made by tutorial creators.

V. Crowdsourced Tutorial Generation

The Codepourri system allows anyone to create a tutorial, and then allows learners to view the tutorial online by stepping through the code and reading the annotations at each step. Thus, an instructor can create a tutorial using their expertise and post a URL for their students to study. Although that is a reasonable use case, there are far more learners than instructors in any educational setting, so can we leverage learners themselves as a crowd to create tutorials rather than using up the instructor's scarce time? To find out, we experimented with a novel crowdsourcing approach to coding tutorial generation.

A. Why Use Learners as a Crowd?

The primary reason is scalability: Since there are far more learners than experts in any domain, drawing from a crowd of learners drastically expands the pool of potential tutorial creators. We experimented on the population of learners who visit the Online Python Tutor website. Since the purpose of Online Python Tutor is to help its users learn to code via visualizations, the majority of its users are learners. Many of its users are referred there from introductory computer programming MOOCs and digital textbooks [8]. We did not pay workers but instead appealed to their altruism in helping other learners such as themselves.

A secondary reason for using learners is that instructors often, without realizing it, assume that their audience has more background knowledge than they actually do. This well-documented phenomenon is known as the *expert blind spot* [10] – experts forget what it was like to be a novice, so they might skip over rudimentary explanations that novices need to see. Having a crowd of learners create tutorials could fill in some conceptual gaps that experts miss when making tutorials.

To our knowledge, we are the first to leverage a crowd of anonymous volunteer learners to create coding tutorials. Our unique crowd is unlike using learners in a formal course because courses are not anonymous, and it is also unlike using a paid worker crowd on Amazon Mechanical Turk. Also, we are not employing any gamification tactics; workers are contributing purely motivated by their desire to help others.

B. Codepourri Crowdsourcing Workflow

To generate a visual coding tutorial using Codepourri:

1) **Setup**. Create the task by writing code, visualizing it using Online Python Tutor, and generating a unique Codepourri page (Figure 5). Then put a banner ad at the top of the Online Python Tutor site (Figure 6) that links to that task's URL.

2) **Annotate**. When any visitor on that website clicks that banner, it opens Codepourri in a new window and loads a page like Figure 5. We gave the workers the following instructions: *"You have arrived at a random step in this code. Your goal is to help create a tutorial that describes what this code does. Each step of execution has a bubble on the line that is executing. Click on the bubble to add annotations that describe what the code does at the current step. Feel free to add an annotation for any step."* Codepourri decides which step of execution the worker should first be routed to. While workers are free to annotate any step of the code that they wish, in our experience, the step that they first arrive at is the step for which a worker is most likely to add an annotation. This initial routing decision is based upon where the code most needs additional annotations, which is calculated using density metrics (see Section VII for details).

3) **Aggregate**. The final phase in our workflow is creating a tutorial by combining together the annotations gathered from the crowd. To do this, it picks the best annotation for each step using input from a comparative voting procedure in which each worker compares the annotation they just added to other annotations previously added by other workers. In Section VIII, we describe the enhancements we made to Codepourri's user interface to support this aggregation phase.

VI. Experimental Setup

To refine the design of Codepourri and to assess the quality of tutorials generated by a crowd of learners, we performed several formative tests followed by a final evaluation. We evaluated Codepourri using pieces of Python code taken from *Computer Science Circles* [11], a popular digital textbook for learning basic programming. We took two pieces of code at different levels of difficulty – easy and medium – based on the textbook sections in which they appeared. Using pieces of code at multiple levels of difficulty allowed us to determine whether code complexity affects annotation quantity and quality. The easy code consisted just of simple variable assignments, an if-statement comparing two numbers, and a simple print statement. The medium code was more challenging: it included multiple functions that called each other and several complex print statements with variables as parameters. Figure 10 shows the easy code, and Figure 5 shows the medium code.

We recruited volunteer workers to annotate these two pieces of code using an advertisement on the Online Python Tutor website. We first tried using a simple text link saying

Fig. 6. A banner advertisement for our crowdsourcing experiment appears at the top of the Online Python Tutor web page (code editor shown here). When the user clicks that banner, the Codepourri interface opens in a new window.

"Want to help our research? Click here!" However, we found that this advertisement received almost zero clicks, despite thousands of daily visitors on that site. Thus, we tried a more informative and visually enticing banner ad (Figure 6), which consisted of a mildly blurred-out preview of the code that we wanted the user to annotate as well as the link asking for participation. An important benefit of this banner ad is that by showing a blurred preview of the real code that the user will be annotating, it informs repeat Online Python Tutor users when there is new code that they have not yet annotated. This final version of the banner ad resulted in a click-rate of 8.64% of all Online Python Tutor visitors over the course of the banner's 10-week deployment throughout all of our experiments.

For all experiments, we report participation numbers in terms of *active users*. To be considered an active user, once the user has arrived at Codepourri, they must take at least one step through the visualization. Only considering active users more accurately reflects usage patterns of Codepourri because, as a strictly volunteer task, many potential workers click on the advertisement out of curiosity, arrive at Codepourri, determine that this task will require non-trivial effort, and immediately close the page. When determining the efficacy of Codepourri, we want to count only workers that are at least interested in learning more about our task and *considering* participation; we use visualization engagement as a proxy for this intent.

VII. Formative Tests and System Refinement

We ran three formative tests to optimize Codepourri's user interface so that it could gather the largest number of high-quality annotations in the shortest amount of time.

Version 1: Pilot Study. Our initial version allowed for annotations on any line of code at any step of execution and always started users at the first step of execution. We ran a pilot deployment for one week with the medium difficulty code. Very few workers chose to participate, and of those who did, their combined annotations could not easily be joined to create a tutorial that explains what the code is trying to accomplish. For instance, one worker left an annotation on a blank line saying *"Line break for clarity - this is good programming practice."* While this may be a valid annotation and perhaps a useful style tip, it does not advance learners' understanding of code execution semantics. The main lessons we learned from this pilot were that having too many bubbles shown at each step might have overwhelmed workers and led some workers to annotate less interesting parts such as blank lines.

Version 2: One Bubble Per Line. To try to raise the participation rate and annotation quality, we tested a new version that allowed annotations only on one line per step of code execution: the line that is executing at that step. While allowing workers to annotate lines of code that are not currently executing might still provide useful annotations, by giving workers a more manageable, less intimidating, and more specific task (they see only one bubble at a time), we hoped that workers would be more willing to complete it.

We tested this version using both the easy and medium difficulty code, deploying each for one week. For the easy code, 15 out of 154 unique workers chose to add an annotation; collectively they added 50 total annotations (an average of 3.33 annotations per worker). For the medium code, 15 out of 239 unique workers chose to add an annotation, and 35 total annotations were added (2.33 annotations per worker).

Version 3: Weighted Routing and Question Prompts. During the Version 2 test, we discovered two new limitations. First, we found that workers heavily annotated the first few steps of code execution, but the density of annotations dropped drastically for later steps. This likely occurred because Codepourri always started each worker at the first step of execution, so they probably annotated only the first few steps and then exited. To alleviate this problem, we implemented a routing system that sends workers to a step randomly chosen between all steps with the smallest number of annotations (possibly zero). They can either annotate that step or navigate to any other step.

Second, users interpreted the task differently since the annotation bubble contained no instruction prompt. To get more precise annotations, we prototyped a feature that statically analyzes each line of code and asks a specific question related to that line. For example, if a line of code was a function definition, it would ask "Why is this function being called, and how is it used in this step?" If a variable `result` was being set, it would ask "Why is the variable 'result' set here? What is it used for?" Questions were displayed prominently inside of the bubbles prompting users to add annotations.

When deploying this version for a week on each piece of code, we found that for the easy code, 17 out of 149 unique workers chose to add an annotation. On average, each worker made 2.1 annotations, for a total of 35. For the medium code, 15 out of 162 unique workers chose to add an annotation. On average, each worker made 2 annotations, for a total of 30.

The routing feature successfully evened out the distribution of annotations, though there was still a slight bias towards the first few steps of code execution. We believe this is because some workers choose to simply step back to the first step of execution in order to gain overall context.

The question-asking feature succeeded in that the content of the annotations was more consistent, but we actually lost out on some interesting information and insight that workers had shared in previous annotations because workers ignored lower-level observations. For example, a typical annotation for a step in this version sounded like *"this variable is used to find out how old any person is currently."* This described what the purpose of the variable assignment is, but forgoes lower-level descriptions of what the code is accomplishing. Thus, we decided not to move further with asking specific questions to workers. Instead, we changed our prompt to be a generic one, simply asking: "What does this part of the code do?" (Figure 5). Note that in our pilot, Codepourri did not even have a prompt; now it has a generic prompt that provides

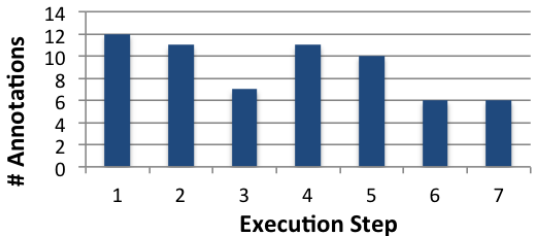

Fig. 7. Number of learner annotations on each step of the easy code.

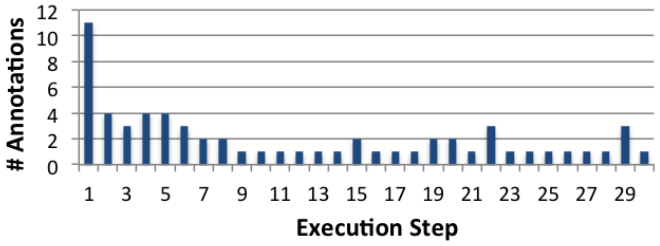

Fig. 8. Number of annotations on each step of the medium code. Note that the number drops off in later steps because the same lines of code re-execute.

some guidance. These changes led to a fourth iteration of Codepourri, which we used for our final evaluation.

VIII. Evaluation of Learner Annotation Quality

Using the fourth and final iteration of Codepourri, we used banner advertisements (Figure 6) to solicit learner annotations for the easy code for one week, and then for the medium code for one week. For the easy code, 51 out of 446 unique workers who interacted with the page chose to add an annotation (11.4% participation). Each worker contributed an average of 1.27 annotations, for a total of 66 annotations. For the medium code, 50 out of 581 unique workers chose to add an annotation (8.6% participation). Each worker contributed an average of 1.58 annotations, for a total of 79. Figures 7 and 8 show how many annotations were made on each execution step.

To gauge the quality of the learner crowd's raw annotations and resulting tutorials, we ran a study with four experts. Three of them are professors who teach introductory programming courses in Python, and the fourth is an advanced computer science undergraduate with experience as a teaching assistant. Each session lasted for one hour, and we asked the expert to:

1) Create tutorials for both the easy and medium code using Codepourri.
2) Rate the quality of all raw annotations left by learners during the one-week online deployment of the easy and medium code.
3) Compare the final learner-created tutorials generated by two aggregation methods.

A. Raw Annotation Quality

We asked the experts to rate each of the 145 total raw annotations made by learners on the easy and medium code. They could choose from six possible ratings: BS for junk (e.g., random text, spam), W (Wrong) for an annotation that provides incorrect information, R (Right) for one that provides correct information, and G (Great) for an exceptional annotation. They could also mark R or G annotations with an extra S for

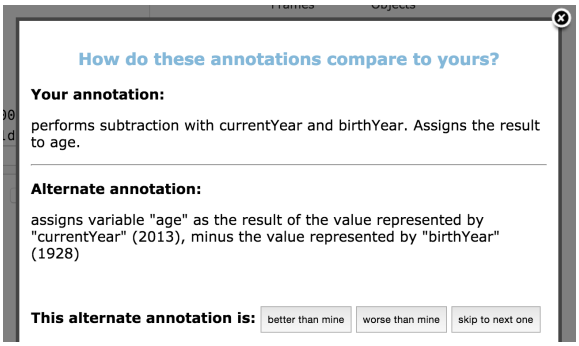

Fig. 9. The comparison voting UI for workers judge the quality of an annotation that they just made by comparing it to annotations that prior workers made for the same step, seeing one competing annotation at a time.

Surprising if they saw something that they would not have thought to mention themselves.

For example, all four experts agreed that this annotation was GS (Great and Surprising): *"We create a first accessible integer value containing the number 1928 and labeled birthYear. It could have been named however we wanted but programmers usually try to name their variables in a way so they can remember what it contains at any time. If we had written aa = 1928, code would have been exactly the same (because computers don't care what the name of the variable is) but it would have been more difficult to make a bigger code."* In contrast, this one for the same step was simply an R (Right): *"Initialize variable 'birthyear' to integer 1928."*

Here are the percentages of total annotations that each expert assigned to each quality category:

	BS	W	R	RS	G	GS
Expert 1	18.4	14.7	53.7	1.5	5.1	6.6
Expert 2	18	15.1	17.3	2.2	36.7	10.8
Expert 3	20.7	15.2	52.8	1.4	7.6	1.4
Expert 4	17.3	20.7	38.7	16.7	3.3	3.3
Average	18.6	16.4	40.6	5.5	13.2	5.5

All annotations rated an R or higher contain correct information. While there is some disagreement between how experts differentiated between R vs. G, and how they assigned S, experts have a very high agreement as to how many annotations are BS, W, or generally correct. On average, 64.8% of annotations were rated at least an R, which is high considering that they were all made by anonymous volunteer learners. 17% of correct annotations also had an S label, which means that they provided insights that even our experts did not think to provide (i.e., manifestation of the expert blind spot).

B. Creating a Tutorial by Choosing the Best Annotations

After gathering raw annotations, Codepourri uses the crowd to aggregate them together to create a single tutorial where each step has at most one annotation (see Figure 1).

Given that our crowd consists solely of anonymous learner volunteers, we cannot rely on all annotations to be good. Also, simply displaying a large number of possibly redundant annotations would not make for a good tutorial because it could confuse or overwhelm the tutorial reader. To solve this problem, we developed a way to use our same learner crowd to choose the best annotation for each execution step. We

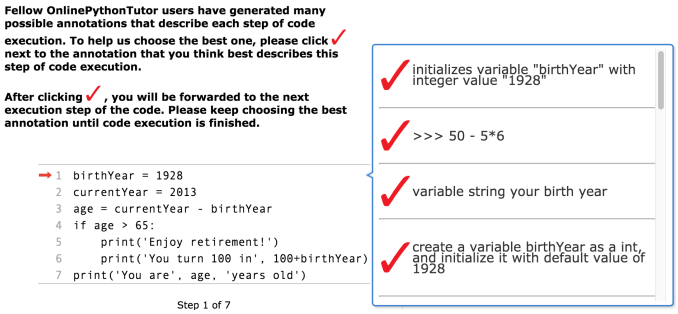

Fig. 10. The 'Choose the Best' interface where a new set of workers judge the quality of annotations contributed by a prior set. All annotations for a given execution step are shown, and the worker chooses the best one.

experimented with two aggregation methods. We used only the easy code for these experiments since it had more annotations per step than the medium code (see Figure 7 vs. Figure 8), making it better for testing aggregation techniques.

Method 1: Comparison Voting. The idea for this method is that workers directly compare the annotation that they just added against all other annotations that already exist for a given execution step. To implement this, Codepourri automatically pops up a voting interface right after a worker adds an annotation (Figure 9). It displays the worker's annotation at the top, followed by a single annotation left by a previous worker. They can choose whether they thought the competing annotation was better or worse than their own annotation, or skip if they were not sure. Casting a vote automatically shows the next competing annotation, and this process repeats until the worker has voted against all annotations or chooses to quit.

To aggregate these votes and choose the single best annotation for each execution step, we performed the following calculation: take the number of times that each annotation was voted as better than a competing annotation and divide by the total number of votes that annotation took part in. The annotation for which this ratio is the highest wins.

The upside of this method is that, rather than recruiting separate crowd workers to vote, we can simply use workers who are already adding annotations. One potential downside is that, since it is done immediately after adding an annotation, workers can vote only for/against prior annotations, meaning that only the final worker is able to vote for/against all annotations. Another potential downside is that, since this approach requires workers to compare their own annotations to others, some may be biased towards voting for their own annotations. Of the 234 total votes contributed by 47 workers, 151 votes were for a worker's own annotation, 38 were in favor of the competing annotation, and 45 votes chose to skip.

Method 2: Choose the Best. The idea here is that, rather than integrating quality judgement into the annotation process, run a second crowdsourcing task where a new set of workers are recruited to rate annotations after the first set of workers has finished contributing all of them. This method has the downside of needing a separate task, but the upside is that all annotations are considered by an unbiased set of new workers.

To implement this, an interface displays all annotations for a given execution step as a list in a bubble, with a red check mark next to each (Figure 10). We asked each worker to choose "the annotation that you think best describes this step of code execution." Each worker starts at the first execution step (not a random one), and once they make a choice, they continue to the next step. This process continues until the worker has cast a vote for all steps. Workers may leave the session at any time, and their prior votes are still counted. We received a total of 516 total votes contributed by 87 workers. The annotation at each step with the most votes is included in the final tutorial.

Best Aggregation Method: We asked our four experts to compare the tutorials produced by both methods. Half felt that method 1 was better. Expert 1 said that method 1's tutorial was more precise; expert 3 said that several of method 1's annotations were significantly more detailed. The other half cited that method 2's annotations for the last few steps sounded more like they belonged in a tutorial. All four experts agreed that choosing between the two tutorials was difficult because they were both high quality. Given that that the experts arrived at a tie, we conclude that they are of comparable quality. Given that method 1 (comparison voting) is more efficient since it does not require a separate task with another set of workers, we choose it as Codepourri's aggregation method.

C. Overall Quality of Learner-Created Codepourri Tutorials

All four experts agreed that the raw annotations and resulting tutorials created by the crowd are comparable to those created by experts such as themselves. They agreed that while some annotations were not quite as precise as those provided by experts, other annotations were more detailed and better explained concepts than they would have thought to do. Experts 1 and 3 pointed out that for complex code with many execution steps, recruiting a learner crowd has a significant advantage over an expert because the expert may suffer from fatigue that results in decreased quality or detail when annotating dozens or more steps; in contrast, an individual crowd worker needs to annotate only a small number of steps, and the final tutorial is created via automatic aggregation. Expert 2 said that the downside of crowdsourcing is that the inconsistent tone and writing style of different workers could make annotations jarring to read. Expert 4 was pleasantly surprised that many learners evaluated the concrete values of variables in their annotations (e.g., *"On the next line, prints 'You turn 100 in 2028' (without the single quotes), since birthYear=1928 (from line 1) and 100+birthyear=2028."*), rather than simply describing them in abstract terms.

We asked our experts to create their own tutorial using Codepourri for both the easy and medium code. Experts added a total of 51 annotations. We now compare the expert-created tutorials to the easy code tutorial created by the learner crowd (aggregated using Method 1). The learner-created tutorial was more detailed and complete. For instance, here are the annotations for step 4:

- **Expert 1**: *"print a retirement message if you're older than 65 years old"*
- **Expert 2**: *"checks if the difference was large enough"*
- **Expert 3**: *"check if ready for retirement"*
- **Expert 4**: *"We check the value of 'age' to see if the person's age is 65 years or older."*
- **Learner-created**: *"This checks whether age is greater than 65. If so, it will execute the print statements located in the indented block below it."*

The learner-created annotation mentions everything that the best expert-created one (from Expert 1) does, and also provides more details that might not be obvious to novices. Other steps had even more differences between learner-crowd-generated and expert-generated tutorials. Many of the expert annotations lack specific information on what is happening in the code, while the learner-crowd annotations included these details. Experts chose to not annotate some steps at all, perhaps due to the expert blind spot; i.e., feeling that those steps are too trivial to explain. Our four experts left annotations on only 4, 6, 4, and 7 of the 7 total steps, respectively, while Figure 7 shows that learners left multiple annotations on all steps.

D. Codepourri Versus Traditional Tutorial Creation Methods

At the end of our study, we asked our experts about how using Codepourri compared to making tutorials by simply commenting code or writing text. All four agreed that Codepourri is preferable. Specifically, Expert 2 appreciated being able to describe dynamic execution state, saying that it was "super helpful when working through code with loops, complicated function calls, and recursion." Expert 3 said that inline comments can break up the spacing in code and external text is too far removed from code, but Codepourri annotation bubbles show up right next to the code. Expert 3 also said that Codepourri helps users not get distracted by annotations that are not relevant to the current execution step.

E. The Potential for Using Codepourri in the Classroom

We also asked our experts about their thoughts on incorporating Codepourri into courses they teach. Expert 2 said they think that a Codepourri tutorial created by a professor or teaching assistant would be a helpful tool while teaching. Expert 2 also thinks that, while expecting students to contribute annotations on a voluntary basis is not realistic, they can see great pedagogical value in making students add annotations as a homework assignment. They envision the use of comparison voting as a peer grading technique. Expert 4 said that they would "love to use the annotation feature in my course." They envision using the voting feature as an in-class activity, and using the results to both gauge students' understanding of topics and address their misconceptions in real time.

F. Study Limitations

While we asked experts to evaluate the quality of learner-created tutorials, we did not formally test a group of learners to assess how well they actually learned using these tutorials. Also, we did not optimize for total elapsed time. In this study, we gave the crowd one week to generate each tutorial. In the future, we could use a real-time crowdsourcing such as LegionTools [23] and other techniques [24] to speed up this process if the learner population is large enough. Finally, we found that the difficulty of the code that we asked our volunteer workers to annotate had a direct effect on participation rates. During our formative tests, we also tried adding a third piece of 'hard' code, which was actively tricky and included nested functions (closures) and misleading calls. Almost no workers annotated this code, though. Thus, it is an open question as to how this technique will scale to complex code taught in advanced courses. Despite this limitation, we have shown that this technique has potential for creating tutorials for code at the level of introductory-level courses, which can help a large population of novices who are starting to learn programming.

IX. Future Work

Motivating workers. Giving better motivation to workers could help even out the distribution of annotations across steps, increase the quality of annotations, and increase the total number of workers willing to annotate. Help sites such as Stack Overflow use an account system with reputation points attached to each user to encourage participation [25]. Codepourri could incorporate a similar system, rewarding points based on the quality of annotations, which it can do by scaling reward amounts based on upvotes. We could also give a higher reward amount when workers annotate steps of code that have the most need for new annotations; these are often indicated by a lack of agreement amongst raters as to which is the best annotation at that step (i.e., the least consensus).

Targeted recruitment ads and intelligent routing. Since we recruit workers from Online Python Tutor, we are able to link usage habits from that website with Codepourri usage habits. One possible use of this data is to create targeted banner advertisement for recruitment. By analyzing the code that a worker naturally writes on that site, we can determine what a worker's interests are. For example, if they spend a lot of time stepping through loops, we know that this worker has an interest in learning about how loops work. We can then create a targeted ad that asks the worker to help us annotate execution steps of a loop. Since we are asking the worker to complete a task about a subject that we know they are interested in, the worker might be more interested in annotating this type of code. This technique will help alleviate the earlier-discussed issue of having an uneven distribution of annotations. The system can be even more intelligent by considering how well it thinks a worker understands certain topics (based on both their own written code and quality of their annotations) and how difficult it thinks a certain step of code is to annotate. We can use this data both to increase the quality of annotations and also to increase the learning value that workers may receive by annotating code, such as sending them to annotate areas of code that are just slightly above their expertise level.

X. Conclusion

We have presented Codepourri, a system that enables people to create visual coding tutorials by annotating execution steps within an automatically-generated program visualization. We have also developed a novel crowdsourcing workflow where learners volunteer to make annotations within Codepourri and then vote on the best ones to use in a tutorial. In an experiment involving easy and medium difficulty code examples from an introductory programming textbook, experts judged the crowd-created tutorials to be comparable in quality to their own and were pleasantly surprised that the crowd provided some insights that even they did not think to provide. Thus, this technique has potential to scale up the creation of tutorials for introductory-level programming content by leveraging the collective time of a large crowd of online learners rather than using up the scarce time of experts.

Acknowledgments

Thanks to Walter Lasecki for his feedback on this project.

REFERENCES

[1] M. Guzdial, "Limitations of MOOCs for Computing Education - addressing our needs: MOOCs and technology to advance learning and learning research (ubiquity symposium)," *Ubiquity*, 2014. [Online]. Available: http://doi.acm.org/10.1145/2591683

[2] J. Sorva, "Visual program simulation in introductory programming education," Ph.D. Dissertation, Aalto University, 2012.

[3] B. Du Boulay, "Some difficulties of learning to program," *Jour. Educational Computing Research*, vol. 2, no. 1, 1986. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1207/S15327809JLS0904_3

[4] J. Sweller and G. A. Cooper, "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction*, vol. 2, pp. 58–89, 1985.

[5] M. C. Linn and M. J. Clancy, "The case for case studies of programming problems," *Communications of the ACM*, vol. 35, no. 3, pp. 121–132, Mar. 1992. [Online]. Available: http://doi.acm.org/10.1145/131295.131301

[6] L. E. Margulieux, M. Guzdial, and R. Catrambone, "Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications," in *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ser. ICER '12. New York, NY, USA: ACM, 2012, pp. 71–78. [Online]. Available: http://doi.acm.org/10.1145/2361276.2361291

[7] P. Pirolli, "Effects of examples and their explanations in a lesson on recursion: A production system analysis," *Cognition and Instruction*, vol. 8, no. 3, pp. 207–259, 1991.

[8] P. J. Guo, "Online Python Tutor: Embeddable Web-based Program Visualization for CS Education," ser. SIGCSE '13. ACM, 2013, pp. 579–584.

[9] J.-P. Guo, M. F. Pang, L.-Y. Yang, and Y. Ding, "Learning from comparing multiple examples: On the dilemma of similar or different," *Educational Psychology Review*, vol. 24, no. 2, pp. 251–269, 2012.

[10] M. J. Nathan, K. R. Koedinger, and M. W. Alibali, "Expert blind spot: When content knowledge eclipses pedagogical content knowledge," in *Proceedings of the Third International Conference on Cognitive Science*. Citeseer, 2001, pp. 644–648.

[11] D. Pritchard and T. Vasiga, "CS Circles: An in-browser Python course for beginners," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 591–596. [Online]. Available: http://doi.acm.org/10.1145/2445196.2445370

[12] P. Chandler and J. Sweller, "The split-attention effect as a factor in the design of instruction," *British Journal of Educational Psychology*, vol. 62, no. 2, pp. 233–246, 1992. [Online]. Available: http://dx.doi.org/10.1111/j.2044-8279.1992.tb01017.x

[13] S. Weir, J. Kim, K. Z. Gajos, and R. C. Miller, "Learnersourcing subgoal labels for how-to videos," in *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, ser. CSCW '15. New York, NY, USA: ACM, 2015, pp. 405–416.

[14] R. Catrambone, "The subgoal learning model: Creating better examples so that students can solve novel problems," *Journal of Experimental Psychology; General*, vol. 127, no. 4, pp. 355–376, 1998.

[15] A. Wiggins and K. Crowston, "From conservation to crowdsourcing: A typology of citizen science," in *System Sciences (HICSS), 2011 44th Hawaii International Conference on*. IEEE, 2011, pp. 1–10.

[16] K. Bielaczyc, P. Pirolli, and A. Brown, "Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving," *Cognition and instruction*, vol. 13, no. 2, pp. 221–252, 1995.

[17] W. A. Sandoval, J. G. Trafton, and B. J. Reiser, "The effects of self-explanation on studying examples and solving problems," in *In Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc*, 1995, pp. 511–532.

[18] A. Cross, M. Bayyapunedi, D. Ravindran, E. Cutrell, and W. Thies, "Vidwiki: Enabling the crowd to improve the legibility of online educational videos," in *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, ser. CSCW '14. New York, NY, USA: ACM, 2014, pp. 1167–1175.

[19] S. Zyto, D. Karger, M. Ackerman, and S. Mahajan, "Successful classroom deployment of a social document annotation system," ser. CHI '12. ACM, 2012, pp. 1883–1892. [Online]. Available: http://doi.acm.org/10.1145/2207676.2208326

[20] M. J. Lee and A. J. Ko, "Personifying programming tool feedback improves novice programmers' learning," in *Proceedings of the Seventh International Workshop on Computing Education Research*, ser. ICER '11. New York, NY, USA: ACM, 2011, pp. 109–116. [Online]. Available: http://doi.acm.org/10.1145/2016911.2016934

[21] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, no. 2, pp. 83–137, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1089733.1089734

[22] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 15:1–15:64, Nov. 2013. [Online]. Available: http://doi.acm.org/10.1145/2490822

[23] M. Gordon and W. Lasecki, "LegionTools," http://rochci.github.io/LegionTools/, 2014.

[24] M. S. Bernstein, J. Brandt, R. C. Miller, and D. R. Karger, "Crowds in two seconds: enabling realtime crowd-powered interfaces," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, Santa Barbara, CA, USA, October 16-19, 2011*, 2011, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/2047196.2047201

[25] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest q&a site in the west," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '11. New York, NY, USA: ACM, 2011, pp. 2857–2866.