Unsupervised functional analysis of graphical programs for physical computing

1st Tom Neutens
Ghent University
IDLab, Department of Information Technology, Ghent University
Ghent, Belgium
Tom.Neutens@UGent.be

2nd Francis wyffels
Ghent University
IDLab, Department of Information Technology, Ghent University
Ghent, Belgium
Francis.wyffels@UGent.be

***Abstract*—Assessing students' code is a challenging and time intensive task for teachers. Facilitating this process is essential to stimulate teachers to teach programming in their classroom. In this paper we describe a new technique to autonomously assess program functionality for different programs in a physical computing context. Using a qualitative analysis on different scenarios, we show that our method can be used to group programs with similar functionality and distinguish programs with different functionality. Our approach will facilitate code assessment for teachers and is a first step toward a more intelligent way to evaluate how children learn programming.**

***Index Terms*—Automated assessment, programming courses, program embedding**

I. Introduction

Programming is becoming an increasingly important part of the curriculum in primary and secondary education. Consequently, teachers are challenged to find appropriate exercises for their group of learners. Many of these exercises can be found online on websites like code.org or Blockly maze. The learners can work through these exercises independently with limited interaction from the teacher. Teachers like these types of exercises because, even with little or no programming experience, they can guide the students through the learning path. In contrast, teachers often avoid more open programming contexts and environments where students are free to experiment because they lack confidence in their ability to guide and assess the exercises. In this paper, we give the first steps towards an automated assessment method that provides teachers with more powerful tools to guide and evaluate the programming process.

An assessment tool should be about more than checking if the solution to a specific exercise is correct; it should provide the teacher with valuable insights into the taught process of each student. It should enable a teacher to answer questions like: Which programming constructs are/aren't well understood? Through which intermediate solutions do different students get to a solution? Which misconceptions do learners have when solving specific programming questions? Which students write more compact/more verbose code? To autonomously answer these questions based on the students' code, we need a way to analyze the functional and structural information embedded in the programs. In this paper, we present a new method for extracting functional information from programs within a physical computing context. Our approach is based on unsupervised learning techniques, which make it applicable to all programs within our programming context without requiring the system to be retrained. To show the value of our technique, we perform a qualitative assessment of different scenarios.

II. Related work

Others have tried to create automated methods for assessing programs. One of the methods used is test-driven learning in which unit tests are defined for the different exercises the learners have to complete. As demonstrated by existing assessment systems like JavaBrat, WebCat, and Marmoset [1]–[3], well designed test-driven learning can be a great help for teachers when assessing programming assignments. However, creating these tests is often labor intensive and consequently practically infeasible to set up if the group of learners is small [4]. Additionally, it only provides high-level information about the performance of learners on specific criteria but gives limited insight into how students solve certain problems or the misconceptions they have. Another shortcoming of test-driven learning is that it is only applicable within a context where the assignments are well defined, and the outcome is fixed. However, many programming activities are open-ended and allow for learners to experiment while creating their application. To deal with the shortcomings of test-driven learning, others have proposed methods for identifying program functionality without the use of unit tests. Most of these methods use machine learning techniques to embed the program functionality into a vector space. In this vector space nearby points represent similar programs. In [5], they identify program function and structure by creating a list of Hoare triples for each program. These Hoare triples represent the state before a part of the program is executed, the part of the program that is executed and the state after the code has been executed. They use a technique inspired by non-linear auto-encoders to represent a program as a linear transformation between states and use the embedding for feedback propagation. Additionally, they show composability, which means the predicted end-state of a composed program is similar to the predicted end-state after first predicting the state after component one and then using this as input for component two. Which means the embedded programs combine in the same way as real programs do. The main drawback of this technique is that it only works within the context of a specific exercise and requires a significant amount of data to determine the embedding. Additionally, since the method combines functional and structural information, it makes it less intuitive to reason about how programs relate in the embedded space. Others have extracted functional information from programs using control flow and data flow features [6]. Control flow features identify the path through the code during execution while data flow features identify how the state of the program changes. They show promising results for declarative programs in identifying differences between different implementations of the same algorithm. However, the algorithm is limited to declarative programs without physical output. Moreover, they do not account for time behaviour which is essential in other programming contexts like physical computing. Another method for extracting functional information from programs was proposed in [7]. In this paper, the authors use visual program output to identify its functionality. This functional dataset is then labeled by assigning

it to one of the predefined milestones and knowledge stages. This dataset of images and labels is then used to train a deep convolutional neural network which can then be used to classify programs into milestones and knowledge stages. The method reports accuracies of 0.562 and 0.649 for the identification of milestones and knowledge steps, respectively. Other techniques for embedding program code have been proposed [8], [9] however, these techniques focus on embedding the program structure based on the code itself or its abstract syntax tree. This can be used to identify programs with similar structure or even predict function names. However, it provides limited information about the functionality of the program.

III. Context

We developed our code analysis technique within a physical computing context. Physical computing is often used to teach novices about programming since it adds a tangible component to the learning process. Several learning tools exist within this context [10]–[12]. We have developed our own set of physical computing tools which we use during multiple courses and workshops. These tools include a graphical programming environment based on Arduino[1] and Google Blockly[2]. Additionally, we use a custom micro-controller board based on the Arduino Leonardo, which is specifically designed to be used in a classroom allowing children to create a simple robot while having limited knowledge about electronics.

Analyzing program functionality within this context poses some challenges which are less relevant in other contexts: (1) Time based behavior. The functionality of these systems heavily relies on time. Imagine a system for a traffic light with the following pattern: 20 seconds red, 3 seconds orange, and 17 seconds green. This system functions correctly; however, if we change the timings to for example: 0.5 seconds red, 50 seconds orange and 9 seconds green, this system is flawed. Consequently, including this time behavior in our analysis is essential. (2) Open assignments. Physical systems are often used in more open learning contexts in which learners can create their own system. Consequently, our technique should be able to capture program functionality for many different programs. (3) Slow data collection. Since learners are working with physical systems a lot of time is spent building and debugging the hardware; consequently, the collection of programming data takes a lot more time. This limitation requires us to design a system which can perform an analysis based on a limited amount of data.

IV. Data

The programs we analyze are written for our microcontroller platform using our graphical programming interface based on Google Blockly. This means that all the programs use the Arduino based *setup-loop* structure, which is represented by the setup-loop block in our environment. The setup code is mainly used to initialize the environment and is usually short. The loop code contains the main body of the program, which interacts with the world by reading inputs from the real world and processing them to specific outputs that act upon this real world. The analysis in this paper limits itself to only a subset of possible inputs and outputs of the board since during most of our workshops, only these inputs and outputs were used. Currently we limited the inputs to the distance read from a sonar sensor (value between -400 and 400) and the state of five buttons (1 or 0). Note that a negative sonar distance can only occur in the real world when the sensor cannot read the distance; in that case the value -1 is returned. However, in our simulated environment we can simulate negative distances which is useful since it provides more information about how the program reacts to different inputs. Table I show an overview of the different outputs of our system. The inputs and outputs can be combined into infinitely many programs using different programming constructs which include: conditions, loops, variables, and time delays.

TABLE I: Possible outputs to our system with the corresponding value ranges.

Input	Value range
LED 1 to 9 state	$[0; 1]$
LCD-screen text	All combinations of 26 characters with length 16
DC-motor 1 and 2 speed	$[-255; 255]$
Servo motor angle	$[0; 180]$
Buzzer frequency	$[20; 20000]$

We collected a dataset of these programs by logging the interactions learners had with our programming environment during ten workshops. Each workshop had three sessions in which learners completed different programming tasks: Programming different poems on the LCD screen, making a two-wheeled robot ride different patterns on the floor and using distance sensor input to make the robot interact with its environment. To map out the coding path, we logged the code each time the learner changed an aspect of the code.

V. Method

To transform our programs into a functional vector representation, we drew our inspiration from Fourier analysis. Fourier analysis is generally used to convert a time-based signal (like an audio signal) into a linear combination of different base frequencies (individual notes). Since we are working with microcontroller programs, time is an essential factor determining functionality. Additionally, we observe that all microcontroller programs contain a certain fixed functionality defined by the code in the *setup* (DC component) and a periodically varying component determined by the code in the *loop* function (AC components). The *setup* function initializes the state of the microcontroller to a fixed value. The *loop* function will change the state of the board with some periodic repetition. This observation leads us to identify program functionality by transforming the variation of state in time to a set of base state frequencies defining the programs' function.

Transforming a program to functional vector representation requires multiple steps. Figure 1 gives an overview of the different steps needed to go from program to vector. First, the code is executed by a simulator. This simulator executes the code in a fictional environment in which all inputs are generated using a periodic sinusoidal function. For example, the sonar sensor measurement changes periodically between -400 and 400 over 2 seconds. These periodic changes make sure all inputs of the program get triggered regularly. The program is executed for $p = 30000$ code steps, which guarantees that all programs in our dataset traverse the loop code at least once. All instructions except the wait instruction take one code step. The wait instruction takes as many code steps as the number of milliseconds it pauses the program. Every $q = 33$ steps, the state of the board is saved. Table I shows an overview of the state which has to be saved. The different elements are saved in a different way depending on their value range. LEDs are saved by a 1 or a 0 if they are on or off respectively. The values for the dc-motors, servo-motor and buzzer are each grouped, and normalized. The grouping makes sure similar speeds, angles, or frequencies get the same value while

[1]https://www.arduino.cc/

[2]https://developers.google.com/blockly/

the normalization brings their weight to the same level as that of the LEDs. Normalisation makes sure different state components equally contribute to the total signal. For the LCD-screen we make a histogram with the occurrences of each letter on the screen. These numbers are then normalized according to the maximum number of occurrences. This results in a total state vector of $r = 39$ elements. Each time this vector is saved, it is added as a new row to a state matrix resulting in a $\lfloor p/q \rfloor \times r$ matrix. This matrix is then vectorised according to column-first order resulting in a vector s of length $n = \lfloor p/q \rfloor r$.

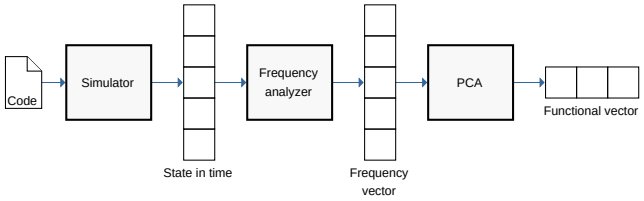

Fig. 1: The vectorisation pipeline which transforms the code into a functional vector.

$$\mathrm{H}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{1}$$

To convert our state-time vector into base frequencies, we use a Hadamard transform [13]. The Hadamard transform is a linear transformation based on the 2-D Fourier transform. We opted for the Hadamard transform because our time state vector is not a continuous function but a combination of different square waves. Since the Hadamard transform has square basis functions, it is better suited for our application than the Fourier transform. Additionally, the Hadamard transform is easier to compute since its values are not complex numbers. An example of the second-order Hadamard matrix is shown in equation 1. We use the H_9 matrix (512×512) and apply it sequentially to our state-time vector s resulting in a frequency distribution vector f with the same length N.

To reduce the dimension of the frequency vector, we apply principal component analysis (PCA) which results in a vector in a 15-dimensional vector. Note that applying PCA directly to the time series since the Independence condition is not satisfied for time series data. The resulting vector can now be used as a functional identifier of our program. In the following sections, we visualize these vectors and provide a qualitative analysis of the accuracy of the representation.

VI. Evaluation

Since we are working in an open problem context labeling our dataset based on program functionality is very labor intensive and practically infeasible. As a result, we evaluate our method by defining a set of scenarios designed to test the requirements that our system should fulfill. These requirements are: (1) Programs with similar functionality should have similar functional vectors. (2) Programs that react on the same input (ex. the sonar distance) but with different actions should be distinguishable. (3) Functional composability, a linear combination of the functional vectors of two programs should be similar to the functional vector of the combined programs. For example, a combination of the functional vector of a program which blinks an LED and the functional vector of a program which shows

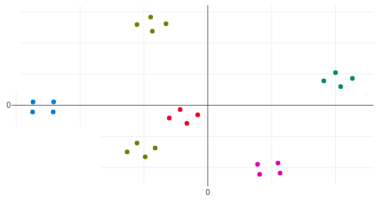

Fig. 2: This plot validates the vectorization of the DC component of a program. The clusters map to a program with a specific dc component, for example turning on an led at the start of the program and leaving in on the entire time like in Figure 3. A different dc component is put in a different cluster.

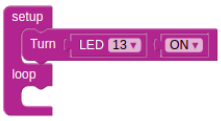

Fig. 3: An example program with a single DC component.

some text on the screen should be similar to the functional vector of the program which blinks an LED while showing the text on the screen. After validating these requirements, we analyze a subset of our dataset by selecting programs from the *drive in a square* assignment.

To evaluate the requirements, we defined five different scenarios. These scenarios include only a few programs and demonstrate the properties of our technique. To visualize the scenarios, we clustered them into two-dimensional space using t-sne [14]. In the t-sne plots, one color represents one program. Note that the same programs are not visualized on exactly the same point in the plot often resulting in small clusters containing the same program.

To verify requirement one, we used two scenarios. Figures 2 and 4 show the t-sne plots for each of the scenarios. The plot in Figure 2 shows five clusters. Each of these clusters contains a program with the same dc component. An example of one of these programs is shown in Figure 3. This clearly demonstrates that programs without cyclical components are separable. Figure 4 shows the visualisation of different programs which are time shifted versions of the the program shown in Figure 5. The plot demonstrates that smaller time shifts result in clusters that are nearby.

To verify requirement two, we used the scenario shown in Figure 6. Figure 6 demonstrates that our technique can separate programs that react to the sonar sensor input. Figure 7 shows one of the programs which are clustered. The other programs are variations on this program constructed by flipping the greater than sing or flipping the do and else statements. We also validated this requirement using button inputs. However, we were not able to include the results due to space constraints.

To assess requirement 3, we calculated the correlation between the sum of two programs in the embedded space and the embedding of the combined programs. In Figure 8 the calculation of the correlation is visualized. The correlation of 0.92 shows a strong relationship between the two terms in the embedded space indicating program composability. Note that the correlation of the first program in the sum with the combined program is only -0.53 and the correlation

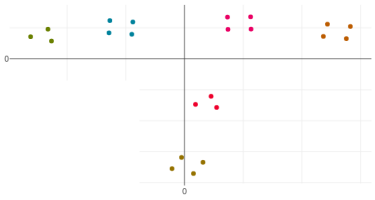

Fig. 4: This plot validates the similarity between similar programs. The program in Figure 5 shows the base program, the other programs used for clustering are created by time shifting this program by: 300 forward (orange cluster), 100 forward (pink cluster), 100 backwards (blue cluster), 300 backwards (green cluster). The bottom cluster represents the original program and the central cluster the empty program. This demonstrates that smaller time shifts in the program result in smaller differences in correlation between vectors in the embedded space.

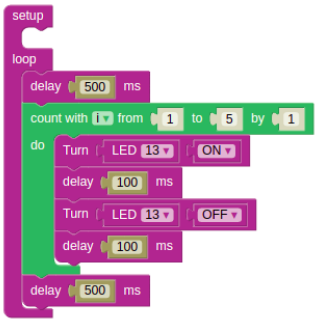

Fig. 5: The base program which is time shifted and clustered in Figure 4

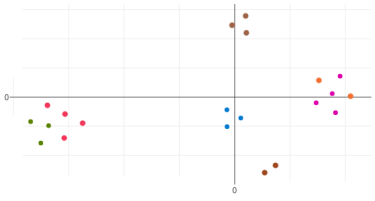

Fig. 6: The clustering plot shows similar programs which react differently to the same input. The clusters on the far left and far right contain variations on the program shown in Figure 7. The right cluster (orange and pink) contains two correct solutions to the *stop at wall* problem. The left most program (green and red) contains the incorrect programs. The center program contains the empty program and the brown clusters contain time delayed variations of the correct solution.

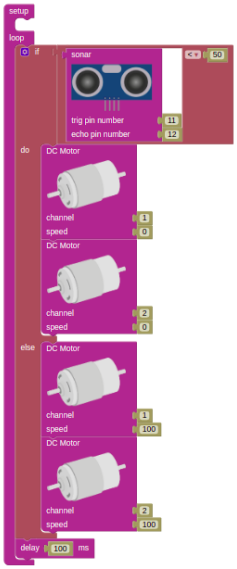

Fig. 7: The *stop at a wall program*: this program makes a riding robot stop at a wall if the distance is small. We vary this program by changing the greater than sign to less than and swapping the content of the *do* and *else* statements. This results in two correct solutions and two incorrect solutions.

between the second program in the sum and the combined program is only 0.37. The lower correlations between the terms and the final program strengthens the composability argument.

Finally, to assess the performance of the clustering on a realistic exercise, we generated a small dataset based on the solutions we found in the data we collected. The goal of the exercise is first to put your name on the LCD-screen and then let a riding robot ride in a square pattern on the floor. The dataset contains the following programs: 1) Putting different names on the LCD-screen. 2) Making one motor turn. 3) The correct motor commands but without delay. 4) The correct motor commands with the first delay but without the second. 5) The correct solution making the robot take a short turn (one motor on positive speed, the other on negative speed) 6) The correct solution making the robot take a long turn (one motor on 0, the other on positive speed). Figure 9 shows the clustering results and the programs in the clusters. These clusters are grouped in the image depending on the functionality of the programs in the clusters. Group 1 (c1) contains programs which only put text on the LCD-screen. All the other clustered programs in the dataset also print one of these pieces of text to the screen but combine it with other functions. Group 2 (c2) only contains null programs without functionality. Group 3 (c3) contains all correct programs in the dataset, which make the robot take a short turn. Group 4 (c4) contains the correct program without delays, which is a common misconception when learners first solve this assignment. The group has programs for the two different names which are printed on the LCD-screen. Group 5 and 6 (c5 and c6) contains both the correct program without the final delay (again a common misconception) and the program which only lets the robot drive straight. It might seem strange that these two types of programs are clustered together, however if the delay at the end of the program is omitted, the two last motor commands are only executed for a short period before returning to the initial motor settings. This is functionally the same as just setting the two motors. The programs are split over two clusters because they print a different

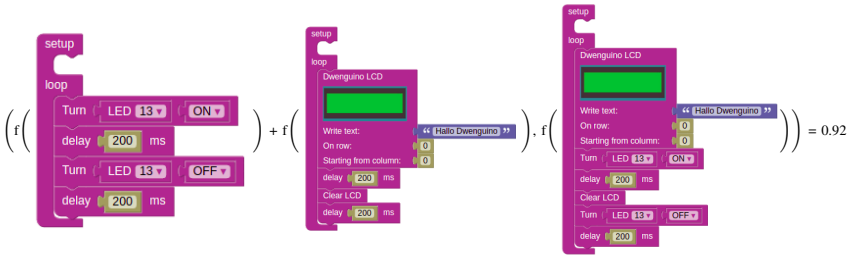

Fig. 8: This equation shows the correlation between the sum of the embedding of two programs and the embedding of the combined program. The correlation between program 1 and program 3 equals -0.53 and the correlation between program 2 and 3 equals 0.37.

name on the LCD-screen. The other programs in the plot which are not in a group are correct solutions that let the riding robot take a large turn with different names printed on the screen. We can see that most of these programs are nearby indicating they have similar functionality. However, since there are some outliers, we decided not to group them explicitly. A video of the clustering results is available at http://bit.ly/SIGCSE2020TN.

VII. Educational use

Our technique provides a new method for identifying and visualizing microcontroller program functionality. Additionally, we believe that our technique can lead to valuable applications in education. Literature has shown that teachers find it challenging to teach programming and physical computing [15]. The first step to help them to overcome these challenges is teacher training. However, teacher training programs are often time constrained resulting in strongly scaffoled learning [16]. This empowers teachers to teach a specific assignment but limits their ability to adapt their teaching to specific student needs and makes them shy away from open problem contexts. Our technique can help teachers to better understand the way their students learn while also giving them a novel way to assess programming exercises. Unsupervised functional analysis can be used for many different application. One application is identifying misconceptions in programming exercises. Common misconceptions like Group 4 in Figure 9 will usually lead to functionally similar programs. These programs will be visually clustered together. When teachers identify clusters which contain similar incorrect solutions of different students this can be identified as a misconception. Our technique can also be used to analyze functional correctness. Currently, functional correctness is usually assessed either visually or by writing functional tests, however, both are labour intensive. Visual inspection requires the teacher to execute every program of every student which is feasible for small groups but becomes impossible for larger ones. Writing functional tests for multiple exercises takes a significant amount of time. For small groups writing tests is not worth the effort. However, for large groups, writing tests can be beneficial. Our technique provides a middle ground, it facilitates the individual assessment by clustering correct solutions close to each other making it easier to identify the learners who correctly solved a problem. Additionally, our technique can be applied in open problem contexts where writing predefined functional tests is difficult or impossible. Finally, the technique does not require the initial time investment to write functional tests. Longer term, our technique can be used to create a tool which independently assesses students' performance enabling methods for automated feedback.

VIII. Future work

We explained a technique for unsupervised functional analysis of graphical microcontroller programs and shown its value in education using a qualitative assessment. These first results are useful; however, many possible improvements remain. A first step would be to do a quantitative analysis of our technique by comparing our clustering results to another evaluation technique. Next, testing within a real classroom setting is essential to establish its value for teachers and students. Finally, the technique can be combined with other information like code structure to autonomously generate a deeper understanding of the learning process. Once this understanding can be established, the final step is to use it to autonomously generate personalized feedback to each student.

References

[1] S. H. Edwards and M. A. Perez-Quinones, "Web-cat: automatically grading programming assignments," in *ACM SIGCSE Bulletin*, vol. 40, no. 3. ACM, 2008, pp. 328–328.

[2] J. C. Caiza and J. M. d. Álamo Ramiro, "Programming assignments automatic grading: review of tools and implementations," 2013.

[3] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, "Experiences with marmoset: designing and using an advanced submission and testing system for programming courses," in *ACM Sigcse Bulletin*, vol. 38, no. 3. ACM, 2006, pp. 13–17.

[4] V. Pieterse, "Automated assessment of programming assignments," in *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*. Open Universiteit, Heerlen, 2013, pp. 45–56.

[5] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, "Learning program embeddings to propagate feedback on student code," *arXiv preprint arXiv:1505.05969*, 2015.

[6] D. M. Perry, D. Kim, R. Samanta, and X. Zhang, "Semcluster: clustering of imperative programming assignments based on quantitative semantic features," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 860–873.

[7] L. Yan, N. McKeown, and C. Piech, "The pyramidsnapshot challenge: Understanding student process from visual output of programs," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 2019, pp. 119–125.

[8] D. Azcona, P. Arora, I.-H. Hsiao, and A. Smeaton, "user2code2vec: Embeddings for profiling students based on distributional representations of source code," in *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*. ACM, 2019, pp. 86–95.

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[10] S. Hutamarn, S. Chookaew, C. Wongwatkit, S. Howimanporn, T. Tonggeod, and S. PANJAN, "A stem robotics workshop to promote computational thinking process of pre-engineering students in thailand: Stemrobot," in *The 25th International Conference on Computers in Education*, 2017, pp. 514–522.

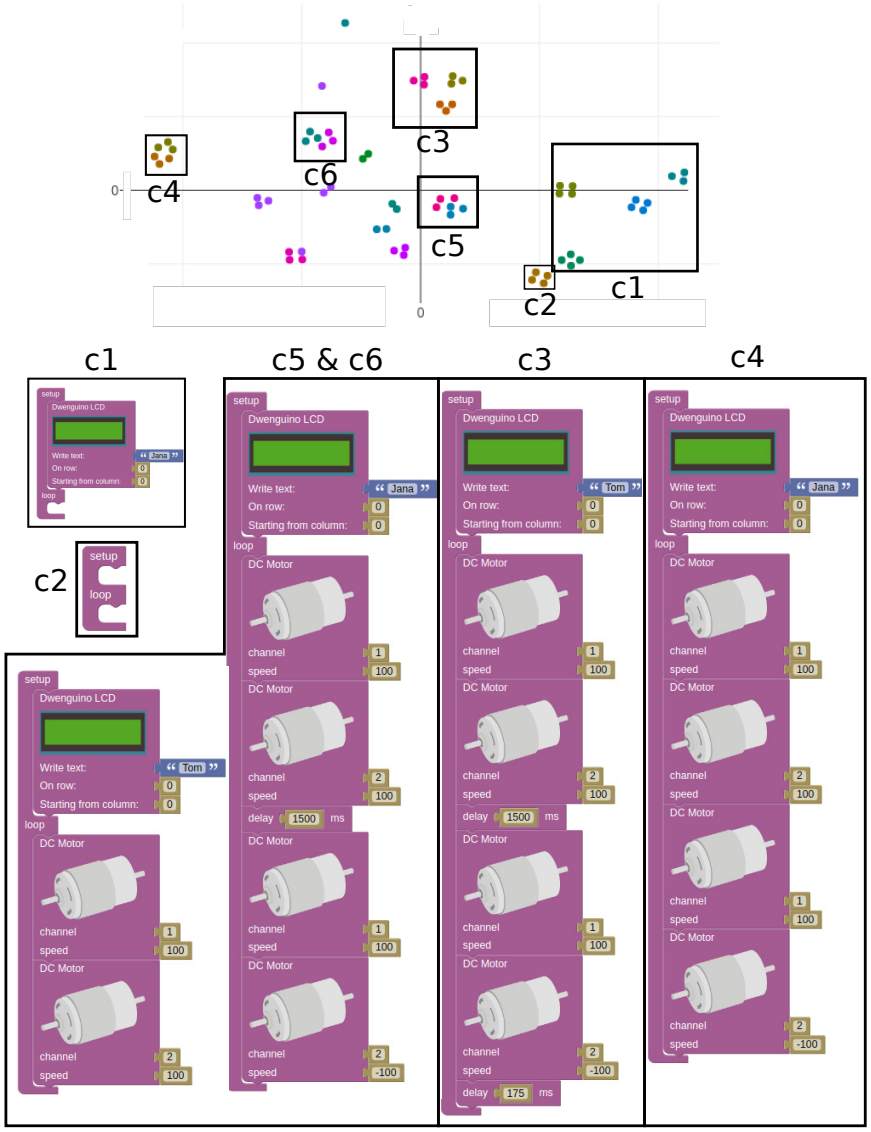

Fig. 9: Clustering result for a dataset sampled from an exercise where learners program a two wheeled riding robot has to drive a square pattern on the floor.

[11] M. E. Karim, S. Lemaignan, and F. Mondada, "A review: Can robots reshape k-12 stem education?" in *2015 IEEE International Workshop on Advanced Robotics and its Social Impacts (ARSO)*. IEEE, 2015, pp. 1–8.

[12] P. P. Merino, E. S. Ruiz, G. C. Fernandez, and M. C. Gil, "Robotic educational tool to engage students on engineering," in *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2016, pp. 1–4.

[13] W. K. Pratt, J. Kane, and H. C. Andrews, "Hadamard transform image coding," *Proceedings of the IEEE*, vol. 57, no. 1, pp. 58–68, 1969.

[14] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[15] F. B. V. Benitti, "Exploring the educational potential of robotics in schools: A systematic review," *Computers & Education*, vol. 58, no. 3, pp. 978–988, 2012.

[16] K. Jaipal-Jamani and C. Angeli, "Developing teacher self-efficacy to teach science and computational thinking with educational robotics: Using scaffolded programming scripts," in *Self-Efficacy in Instructional Technology Contexts*. Springer, 2018, pp. 183–203.