Finding Patterns in Visualizations of Programs

Cătălin F. Perțicaș
Romanian Institute of
Science and Technology
perticas@rist.ro

Bipin Indurkhya
Romanian Institute of
Science and Technology
and Jagiellonian University
indurkhya@rist.ro

Răzvan V. Florian
Romanian Institute of
Science and Technology
florian@rist.ro

Lehel Csató
Romanian Institute of
Science and Technology
and Babeș-Bolyai University
csato@rist.ro

Abstract

We present an approach to classify computer programs according to their semantics, by using adaptive and neural network-based algorithms. We first develop a visualization method that reflects program dynamics. In order to validate the existence of patterns in visualizations, we build five features that characterize sorting methods, and apply self-organizing maps to cluster them. We then use neural networks to classify five sorting algorithms: *InsertionSort*, *BubbleSort*, *HeapSort*, *QuickSort*, and *RandomSort*. Our experiments show above 90% accuracy on the validation set, thereby showing that the specific sorting algorithm can be inferred from a simple visualization.

1. Introduction and motivation

In recent years, there is a renewed interest in modeling the process of software creation. Some aspects of this research are:

1. Being able to model cognitive processes underlying algorithm construction and program generation, we can develop more effective methods of teaching problem solving, and create educational and debugging tools for learning to program. One related research area is to elicit the user's model of program execution. For example, Piaget's model of cognition has been applied to study code completion by professional software developers (Mărășoiu et al., 2015), as well as to study preconceptions of novice programmers about programming (Corney et al., 2012; da Rosa, 2015).

2. Being able to generate patterns to represent the dynamics of program execution, whether through visualization or other means, we can experiment with various clustering techniques to see if we can group these patterns in a way that captures the semantics of the programs. There have been a number of studies on visualizing the structure of a computer program. For example, Alhammad et al. (2016) evaluated three different tools for visualizing the following three aspects of programs: loops, object-oriented programming and parameter passing by value or reference. The evaluation, however is carried out with respect to learning to program.

3. Being able to describe the behavior of a program at a conceptual level from observing the pattern of its execution.

4. A long-term, and somewhat ambitious, goal of this research is to be able to apply machine learning techniques to create new programs that exhibit a given behavior. When trying

to automate the process of software creation, one idea is to create models that learn from code by applying machine learning techniques. Previous attempts to achieve this with a domain-specific language (DSL) include Balog et al. (2017) and Devlin et al. (2017), where the number of possible operations on the input is limited and well-defined, and the data for learning consists of input/output pairs. This encouraged us to approach simple toy-like problems, where some DSL could be identified, such as the sorting problem where the main operations are swaps.

This paper presents our initial results in following the second aspect of the research direction presented above. In particular, we are exploring ways to generate visualizations of program execution dynamics and applying clustering techniques on them to see if we can capture their semantics. Our choice of studying inline sorting algorithms provided us with two advantages: only one operation to track: swapping elements; only one array to analyze.

2. Related research

2.1. The Role of Sensorimotor Integration and Abstraction in Learning

From a cognitive perspective, it has been suggested (Corney et al., 2012) that learning to program occurs in several stages, though this view is not unique to the field of programming. The first of these stages is the sensorimotor stage. We combined tracking simple operations in programs with the idea of sensorimotor understanding and reached a novel idea, the image of algorithm, which is a representation that integrates the states on which the program operates and the actions taken by the program. This visualization is well-suited for learning models with an architecture designed for visual recognition, such as Convolutional Neural Networks (CNNs), introduced by Le Cun et al. (1990). To our knowledge, CNNs have not been used so far for program classification.

The next stages in the piagetian development for programming are increasingly more abstract, better tracing skills, understanding abstraction and program diagrams in familiar situations, and applying creativity and analogies to create programs for solving new problems. Similarly, deep neural networks (Schmidhuber, 2015) have layers which learn concepts of increasing level of abstraction. We tested this model, and in the future we plan to study whether deep neural networks could be constrained to reflect similar learning stages and abstractions.

2.2. Maps and Visualizations inspired from Dual-Coding

The sensorimotor approach to learning has support in embodied cognition approaches, where it is argued that the brain could not exist without its body. In a way, the body is a map of the mind and the mind is itself a map of the body. Hundhausen et al. (2002) mentions dual-coding, the interaction between two different symbolic systems generating cognition, as one of the key ingredients when it comes to effectiveness of visualizations. Along this idea, we investigated visualizations of spatial and temporal features of program executions. We used (C. Shaffer et al., 2007; C. A. Shaffer et al., 2010) as reference for the types of visualizations in the field of algorithms. Our choice of clustering technique, the self-organizing map, introduced by Kohonen (1990) can be used as an alternative visualization tool, although the reason we chose it was due to its superior performance.

3. Image of Algorithm (IoA)

3.1. Introduction

There are many aspects to representing run-time dynamics of programs. For example, one can look at the pattern of function/subroutine calls, the pattern of values being passed between different modules or the pattern of data objects that are being affected by the program. For our research, we focused on tracking parts of the computer memory that are being affected as the program executes. Our goal was to explore if visualizing these patterns as images, and then applying clustering algorithms to these images will result in clusters that are semantically meaningful: they reflect a conceptual structure.

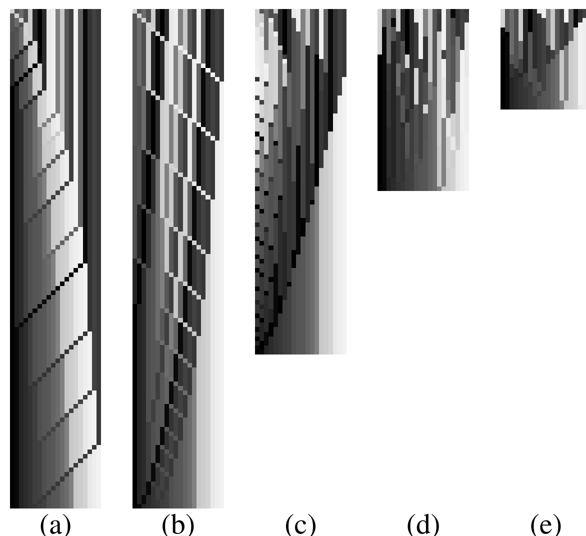


Figure 1 – IoA of various methods for sorting in increasing order. (a) InsertionSort, (b) BubbleSort, (c) HeapSort, (d) RandomSort, (e) QuickSort.

The evolution of the memory state of an algorithm is difficult to infer from its source code: it requires interpretation and knowledge of algorithms, logic and programming languages. A visual representation of the memory dynamics, although larger in size, is an alternative requiring less background information and is better anchored in the reality of program execution. Moreover, recent advancements in image processing motivated us to explore this way of representing program dynamics so that existing techniques of image clustering can be applied for classifying programs based on their execution dynamics.

3.2. Initial Observations

To test the practicality of our program representations, we started with sorting algorithms since they are well studied and have available implementations. We determined that if we assign gray-scale colors to the data stored in memory, and plot the changes through time, we get an image corresponding to how the program is affecting the data objects. We aimed to see if this image has a pattern that recurs over multiple executions of the algorithm given different inputs, so that it can be considered as the signature of the algorithm.

We selected five sorting algorithms — namely, *InsertionSort*, *BubbleSort*, *HeapSort*, *QuickSort*, and *RandomSort* — to study patterns in the visualizations of their effect on the data objects. Figure 1 shows images of algorithms. This visualization associates images to sampled executions of the five sorting algorithms mentioned above. Each row shows the state of the array being sorted at a given point in time. The array values were chosen between 0.0 and 1.0. We represented small-valued elements (closer to 0.0) as dark pixels and large-valued elements (closer to 1.0) as bright pixels. Whenever a swap occurred, we took a snapshot of the array and appended it to the image.

Let $\boldsymbol{A}_0 = [A_{0,0}, A_{0,1} \dots A_{0,N-1}]$ be the initial array, let $\boldsymbol{A}_i = [A_{i,0}, A_{i,1} \dots A_{i,N-1}]$ be the array at the i-th step. The image of the algorithm representation (IoA) of our program execution is simply the collection $IoA = \{\boldsymbol{A}_0, \boldsymbol{A}_1 \dots \boldsymbol{A}_T\}$, where T is the largest number of iterations among the program executions which we take into account. For simplicity, we used collections of equal size by appending the last row $T-k$ times, where k is the number of swaps in the current execution.

Thus, we get two-dimensional images with the horizontal axis representing the space (memory) of the program and the vertical axis representing the time (number of steps). These types of images are not unique to the program and not necessarily deterministic, but do show reoccurring patterns over several samples. We considered one swap operation to be one step along the time axis, although other operations can also be included. This visualization gave us some interesting

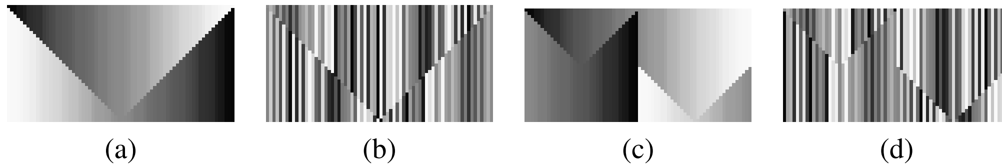


Figure 2 – IoA for reversing methods. Entire array reversal: (a) sorted, (b) random input. Reversal of first and second halves: (c) sorted, (d) random input.

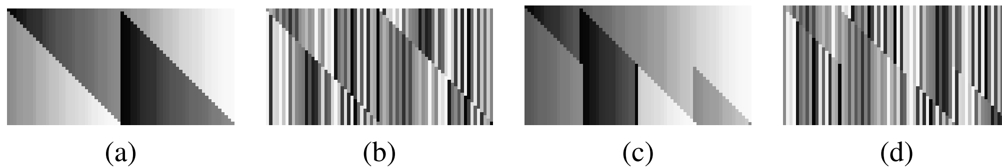


Figure 3 – IoA for interval swap methods. First with second half: (a) sorted, (b) random input. First with second quarter, then third with last quarter: (c) sorted, (d) random input.

insights into the execution patterns of different sorting algorithms.

For instance, it is visually apparent that the *HeapSort* (Figure 1c) works in two stages because its IoA changes its patterns after about one quarter of its execution length. Its first stage is counter-intuitive: it appears to do the opposite of sorting by placing many large elements at the beginning of the array. In fact, it creates the structure of its max-heap, which only makes sense for us in a tree-like visualization. The pattern in the second stage reveals the algorithm's approach to start with processing elements at the end of the array.

The visualization of *InsertionSort* (Figure 1a) reveals the opposite approach: it starts at the beginning of the array and shifts the array to the right as smaller elements get inserted at their right position. For the *BubbleSort* (Figure 1b), we can observe the 'bubble' phenomena as large elements gradually get pushed towards the end of the array. The *QuickSort* (Figure 1e) displays patterns that are not so obvious because of the pivot choice, which depends on the actual sampled values for the execution on the algorithms. From the perspective of the IoA, it is more similar to *RandomSort* (Figure 1d).

In order to improve our study of the patterns to be found in IoAs and to explore other patterns in problems using swaps of elements on arrays, we investigated a few non-sorting algorithms, such as plain array reversal and interval swaps (see Figures 2 and 3).

The reversal program takes chunks from the array and reverses them by iteratively swapping elements from the ends until it reaches the middle of the chunk. This creates a V-shape in the associated IoA. This pattern is visible irrespective of the initial order of the array (see sorted vs. random input in Figures 2 and 3). However, this is not the case for our sorting algorithms, which would simply exit if the array were sorted, with the exception of *HeapSort*.

The interval swap program switches the order of two intervals (chunks) in the array. This is done by iteratively swapping elements from the first interval with elements situated at the same position in the second interval. This program produces an upside-down N-shape.

4. Feature Extraction and Analysis of Patterns in IoAs

Our goal is to be able to extract relevant features from the IoAs automatically (see Section 6). As a preliminary step, we defined five features based on our knowledge of sorting methods: Distance between Current position and Final position (DCF), Minimum Number of Swaps until Sorted (MNS), Number of Inversions (NI), First Time in Place (FTP), and First time Displaced (FTD). The first three of these are spatial features that are extracted along the space axis of the program

execution representation. The last two are temporal features extracted along the time axis.

While the spatial features are only applicable to sorting algorithms and other problems involving permutations of objects, the temporal ones can be extracted from any program provided that the traced operations and data collections are available. We now describe in more detail the features and some of the patterns they reveal in the studied programs.

4.1. Spatial Features

Spatial features include the functions one might apply on a row or a subset of rows in the IoA. For this research, we focus on the features that are applicable to problems with a solution space represented by permutations of an array with distinct elements.

For any array at iteration t, we can assign a permutation P_t that represents the set of order numbers for A_t and if applied to A_t will sort the array. In other words, $P_{t,j} = k$ provided that $A_{t,j} = S_k$, with S being the sorted array in increasing order. Algorithms that sort the array in increasing order have $S = A_T$. For all the spatial features extracted, we will only need to work with these permutations. We use the following three features in this experiment:

Distance between Current and Final position (DCF). We sum the distances between the current position and the final/expected position for each element of the array (for a sorted array, this sum is equal to 0):

$$DCF(t) = \sum_{j=0}^{N-1} |P_{t,j} - j|$$

Minimum Number of Swaps (MNS). Another way to measure distance to the expected solution is to compute the minimum number of swaps required to sort it. This problem reduces to finding the cycles in the permutation. As the array gets closer to being sorted, this measure decreases:

$$MNS(t) = \sum_{j}^{numberOfCycles} cycleSize_j - 1$$

Number of Inversions (NI). A similar feature to measure proximity to the exit state of the sorting program is the number of inversions. The number of inversions for a sorted array is equal to 0:

$$NI(t) = \sum_{j}^{N-1} cardinal\{P_{t,k} | P_{t,j} > P_{t,k}, k > j\}$$

Figure 4a shows the behavior of *InsertionSort*. We can see that the NI value decreases linearly over the number of swaps until the program terminates. This visualization provides a nice hint regarding the time complexity of this algorithm because NI is at most N^2 and the problem does indeed belong to the $O(N^2)$ class. At the same time, we can also infer from the slow decrease in the MNS value that the algorithm is not very efficient.

Figure 4b shows the behavior of *BubbleSort*. From the point of view of the spatial features, it is very similar to *InsertionSort*. This algorithm is also $O(N^2)$ and is known to be inefficient with its swapping strategy.

Figure 4c shows the behavior of *HeapSort*. Due to the small array size, its time complexity advantage cannot be observed. Although the algorithm is known to be fast for large arrays, its high time-complexity constant makes it inefficient for small arrays. In the figure, we can distinguish two stages of the algorithm. First, it creates a heap structure. During this stage, the memory state gets further from the desired state, and we can see an increase in the spatial features values. In the second stage, there is an abrupt descent, which is not very smooth, and hints at the use of a tree-like data structure. Its time complexity is $O(n \log(n))$.

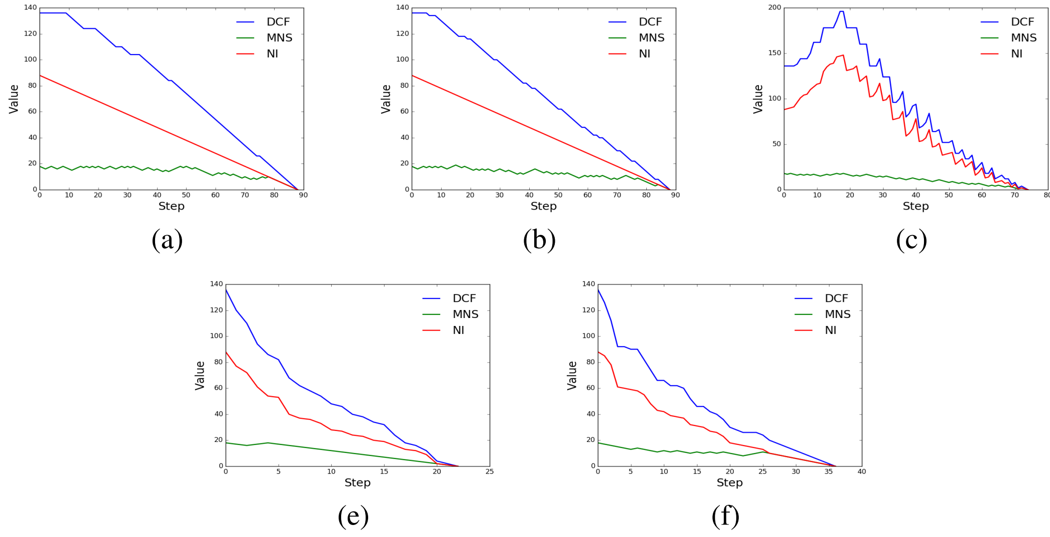


Figure 4 – Spatial features. (a) InsertionSort, (b) BubbleSort, (c) HeapSort, (d) QuickSort, (e) RandomSort. Blue: DCF, Red: NI, Green: MNS.

Figure 4d shows the behavior of *QuickSort*, a very efficient algorithm. We notice a very fast decrease in the feature values. Moreover, its MNS values are the only ones which show a few temporary increases. This suggests it often chooses optimal swaps for sorting the array.

Figure 4e shows the behavior of *RandomSort*. Interestingly enough, choosing random swaps generates a behavior that is similar up to a point to *QuickSort*. However, this can be misleading because this algorithm generates many more possible swaps than shown in the image, but only executes the valid ones, which appear in the image.

4.2. Temporal Features

Temporal features are the category of features we can extract along the time axis of our program execution representation, the IoA. Any function that we apply on a column or a subset of columns in the IoA is considered to be a temporal feature.

First Time In Place (FTP). For each memory cell tracked during the execution of the program, we can extract the necessary number of iterations for it to arrive to its final state>

$$FTP_i = argmin(A_{j,i} | A_{k,i} = A_{j,i}, \forall k > j)$$

This feature compresses the IoA in a way which made it possible to cluster the different algorithms. Dimensionality reduction is necessary in this case because the IoA feature set has a large cardinality and therefore confuses the learning algorithm in the absence of a pattern extraction interface. The FTP features reduce the dimensionality to the number of elements in the array, while keeping traces of the dynamics of the algorithm. We found that the FTP is invariant to the sorting order. Therefore, it can capture the sorting dynamics well, being able to put executions of the same sorting algorithm, but with different sorting order, under the same cluster. This was done using the self-organizing map model. However, the order invariance comes as a disadvantage when we try to discriminate by order.

First Time Displaced (FTD). For each memory cell which is tracked during the execution of the program, we can extract the necessary number of iterations for it to change its initial state:

$$FTD_i = argmax(A_{j,i} | A_{k,i} = A_{j,i}, \forall k < j)$$

An optimal sorting algorithm, which would perform a minimal number of swaps would have

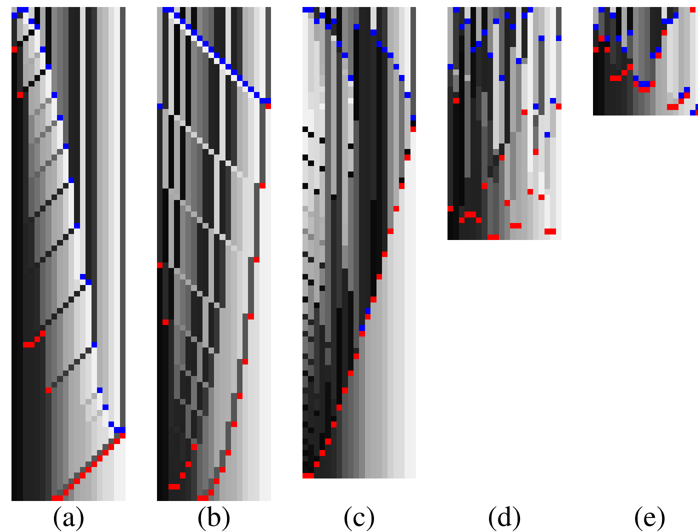


Figure 5 – Temporal Features. Blue: FTD, Red: FTP. (a) InsertionSort, (b) BubbleSort, (c) HeapSort, (d) RandomSort, (e) QuickSort.

the FTP coincide with FTD. The sorting methods which we investigated do not exhibit this behavior, so the two features follow different patterns. This can be seen in Figure 5.

5. Feature-based Clustering

5.1. Overview

The following experiments indicate that given the right set of features and clustering techniques, simple programs such as sorting an array can be naturally grouped into categories which reflect their semantics.

5.2. Self-Organizing Map (SOM)

Self-Organizing Maps are artificial neural networks where neurons are represented as cells in a grid. Each input that we feed in the network will activate the neuron with the most similar weights to the input. Each activation changes the surrounding neurons by adjusting their weights to be closer to the activated neuron. The closer the surrounding neuron is to the activated one, the stronger the adjustment is.

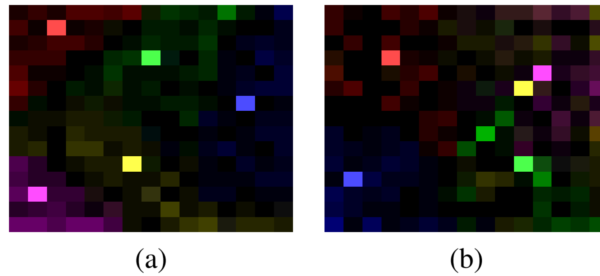
The grid consists of 15 x 15 or 20 x 20 neuron cells. The learning algorithm performs 30 activation iterations and the number of input samples is 100 for each of the 5 sorting methods. The arrays used for learning had 10 elements.

In Figures 6 and 7 there are 5 strongly-colored cells, each representing the centroid obtained by averaging the locations of the sample executions for each algorithm. For the remaining cells, the lighter they are, the more input samples activate the neuron represented by the cell. The clustering method has no prior knowledge of which execution sample belongs to which algorithm.

Out of the clustering methods for sorting algorithms, Self-Organizing Map with FTP worked the best in creating well-separated clusters, as shown in Figure 6a. A more thorough analysis of their performance will require assigning an accuracy metric to the clustering techniques.

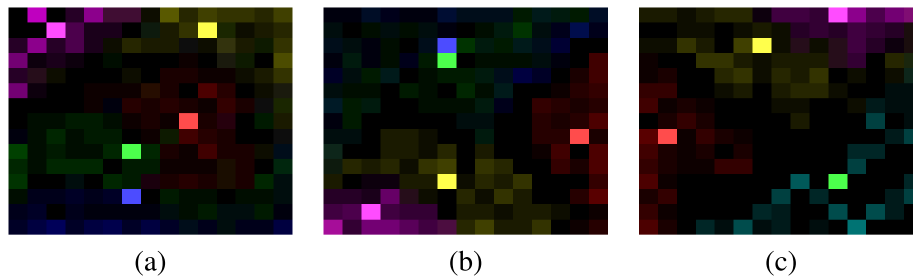
This SOM clustering algorithm also worked well on the MNS feature (Figure 7a), which ranked the second in our analysis.

On the other hand, the DCF (Figure 7b) and NI (Figure 7c) features cannot distinguish between *InsertionSort* and *BubbleSort*. This is to be somehow expected if we take a look at the DCF and NI curves in the two sorting algorithms: they are very similar visually. The FTD feature fails to distinguish between *QuickSort* and *RandomSort*, as shown in (Figure 6b).



(a) (b)

Figure 6 – Clustering on (a) FTP and (b) FTD for 5 sorting methods. Blue: InsertionSort, Green: BubbleSort, Red: HeapSort, Magenta: QuickSort, Yellow: RandomSort.



(a) (b) (c)

Figure 7 – Clustering on (a) MNS, (b) DCF and (c) NI for 5 sorting methods. Blue: InsertionSort, Green: BubbleSort, Red: HeapSort, Magenta: QuickSort, Yellow: RandomSort.

5.3. Control Experiments

In addition to our initial experiments, we tried clustering the five sorting programs as well as two non-sorting programs mentioned earlier: array reversal and interval swaps. We came up with two variants of these non-sorting programs, one in which the input is sorted and the execution patterns are very clear, and another one where the input is random (not sorted).

Our goals behind this experiment were to test the robustness of our technique — how well our approach would generalize to new programs — and to verify that we get some real cluster difference between sorting and non-sorting programs. We obtained interesting results as shown in Figure 8.

First observation: non-sorting clusters are well separated from the sorting clusters for both chosen features. This confirms the robustness of our features and technique for clustering.

Second observation: for the FTP feature, we get the same one-cell clusters for the two types of problems when controlled by sorted input versus non-sorted input. This reveals two facts: the problems of our choice (reverse and interval swap) do not depend on the array values, as opposed to sorting algorithms (inference on FTP clusters - overlap of sorted input vs. non-sorted input clusters); the two programs have a predetermined dynamics perfectly encoded by the FTP feature, so that the clustering algorithm assigns exactly one neural cell to all samples of these algorithms.

Third observation: clustering on MNS behaves similarly to the one on FTP, but only when the input is sorted. This is because different initial orderings result in different values for the spatial features (including MNS). For the non-sorted input, both programs get a well-defined cluster.

6. IoA Classification

6.1. Overview

The IoA classification is very much like the task of recognizing images. Using neural networks for IoA classification would confirm, if accurate enough, that a model with several layers would be able to capture the patterns inside these images.

For the classification experiment, we used ten types of classes, each corresponding to a pair of

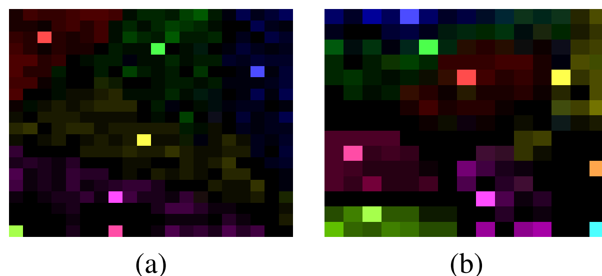


Figure 8 – Clustering on (a) FTP and (b) MNS for 9 programs. Blue: InsertionSort, Green: BubbleSort, Red: HeapSort, Magenta: QuickSort, Yellow: RandomSort, Cyan: Reverse, Lime-green: Reverse sorted, Orange: Interval swap sorted, Fuchsia: Interval swap.

one sorting algorithm and a sorting order (increasing or decreasing). For each class, we used 1000 samples, i.e. different input vectors, resulting in a dataset of 10.000 samples in total. Each input vector was generated randomly, with values from the vector being generated with a uniform distribution between 0 and 1.

We tried two approaches: multilayer perceptron and convolutional neural network, both with four layers on top of the input layer. We used 80% of the data for training and the remaining 20% for validation.

6.2. Multilayer Perceptron (MLP)

We normalized the IoAs, which have a 2D visual representation, by appending the last row until all the images had the same size. Then the images were represented in a vectorial form. Using this format, we trained the MLP on the collected data.

The architecture consisted of 4 layers, 3 hidden ones and one readout. For the hidden layers we used 500 learning units. The input had 420 units (10 array cells x 42 time steps) and the readout had 10 units (5 sorting algorithms x 2 sorting orders). The training time was noticeably smaller than in the CNN case, at the cost of less accuracy — 90% accuracy on the validation set.

6.3. Convolutional Neural Network (CNN)

We used the normalization applied for MLPs, then trained a convolutional neural network with four layers: convolutional (32 features), pooling (2x2 reduction), hidden (1024 units) and readout (10 units). The input had 420 units. This model achieved the highest accuracy - 92.5% on the validation set. This proves that deep learning models can generalize well the patterns extracted from IoA representations of programs, even from relatively small datasets.

Afterwards, we inspected the convolutional layer in order to understand the relevant patterns that help distinguish between different sorting algorithms. Our results showed that the feature activations do not differ too much between different sorting methods, which would roughly indicate that the same set of features could be used to discriminate between them. However, the feature activation maps of program execution for different sorting orders differ considerably. An analysis of the neuron activations from the first convolutional layer indicates that our classification algorithm relies on temporal patterns and the time complexity of the algorithm.

7. Conclusions and Future Work

We have developed the IoA method, which displays the patterns inside programs in a human readable way (Section 3.2). We have shown that automated learning models were able to use the visualizations, as well as features/patterns extracted from them (Section 4) in order to cluster (Section 5.2) and classify (Section 6) with high accuracy five different sorting algorithms. Moreover, we have shown that sorting algorithms generate patterns which can be distinguished from other algorithms (Section 5.3).

We estimate that the models explored so far to group and classify sorting programs based on the exhibited patterns can be extended to more complex algorithms. A large number of computer science primitives could be clustered and automatically identified. There is an abundance of operations besides swap, such as push, pop, append, replace, as well as other data structures, such as stacks, queues and trees, which can be used in future work.

8. Acknowledgement

This work was supported by the European Regional Development Fund and the Romanian Government through the Competitiveness Operational Programme 2014-2020, project ID P_37_679, MySMIS code 103319, contract no. 157/16.12.2016.

References

Alhammad, S., Atkinson, S., & Stuart, L. (2016). The role of visualisation in the study of computer programming. *Proceedings of the PPIG*.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2017). DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations (ICLR).*

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions. *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*, 77-86.

da Rosa, S. (2015). The construction of knowledge of basic algorithms and data structures by novice learners. *Proceedings of the PPIG.*

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., & Kohli, P. (2017). RobustFill: Neural Program Learning under Noisy I/O. *arXiv preprint arXiv:1703.07469.*

Hundhausen, C., Douglas, S., & Stasko, J. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*(13), 259-290.

Kohonen, T. (1990). The Self-Organizing Map. *Proceedings of the IEEE*, *78*(9), 1464-1480.

Le Cun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1990). Handwritten Digit Recognition with a Back-Propagation Network. *Advances in neural information processing systems*, 396-404.

Mărășoiu, M., Church, L., & Blackwell, A. (2015). An empirical investigation of code completion usage by professional software developers. *Proceedings of the PPIG.*

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*(C).

Shaffer, C., Cooper, M., & Edwards, S. (2007). Algorithm Visualization: A Report on the State of the Field. *ACM Special Interest Group on Computer Science Education*.

Shaffer, C. A., Cooper, M. L., Alon, A. J. D., Akbar, M., Stewart, M., Ponce, S., & Edwards, S. H. (2010). Algorithm visualization: The state of the field. *ACM Transactions on Computing Education*, *10*(3), 9:1–9:22.