Directed Acyclic Program Graph Applied to Supervised Classification

Thibaut Bellanger
LDR, ESIEA
Laval, France
thibaut.bellanger@esiea.fr

Matthieu Le Berre
LDR, ESIEA
Laval, France
matthieu.leberre@esiea.fr

Manuel Clergue
LDR, ESIEA
Laval, France
manuel.clergue@esiea.fr

Jin-Kao Hao
LERIA, Université d'Angers
Angers, France
jin-kao.hao@univ-angers.fr

ABSTRACT

In the realm of Machine Learning, the pursuit of simpler yet effective models has led to increased interest in decision trees due to their interpretability and efficiency. However, their inherent simplicity often limits their ability to handle intricate patterns in data. This paper introduces a novel approach termed Directed Acyclic Graphs of Programs, inspired by evolutionary strategies, to address this challenge. By iteratively constructing program graphs from binary decision makers, our method offers a balance of simplicity and performance for classification tasks. Notably, we emphasize the preservation of model interpretability and expressiveness, avoiding the use of ensemble techniques like voting. Experimental evaluations demonstrate the superiority of our approach over existing methods in terms of both effectiveness and interpretability.

KEYWORDS

Evolutionary Algorithm, Genetic Programming, Local Search, Supervised Classification

ACM Reference Format:
Thibaut Bellanger, Matthieu Le Berre, Manuel Clergue, and Jin-Kao Hao. 2024. Directed Acyclic Program Graph Applied to Supervised Classification. In *Genetic and Evolutionary Computation Conference (GECCO '24 Companion), July 14–18, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3638530.3664115

1 INTRODUCTION

In Machine Learning, there is a growing interest in searching for models that might not perform as well as neural networks, but are easier to understand, more compact, and more energy efficient. Decision trees are one such method getting more attention because they meet these requirements [4]. However, decision trees can't always handle complicated problems because they use simple rules to split data. This makes it hard for them to deal with complex patterns [15]. This is where evolutionary computation can help, since it can offer new ways to create these rules, making decision trees better at handling different and complex situations. Additionally it can help to create more complex models, like graphs of programs, which can be more efficient than decision trees for some problems [9].

In this paper, we propose a new approach, inspired by the Tangled Program Graph (TPG) [9] and Tree in Tree (TnT) [17] methods, to build Directed Acyclic Program Graphs (DAPGs). The idea is to build a graph of programs which make local decisions at each node, until a leaf is reached. The leaf is then the class predicted.

The base component of our method is a program that acts as a binary decision maker. These binary programs are inspired by binary decomposition techniques, which offer a way to enhance classification performance on multiclass problems [1, 5–8]. One way to aggregate results from binary classifiers is to use a Decision Directed Acyclic Graph (DDAG) [11]. The issue with DDAG obtained by binary decomposition is that they have static structure, which means that the structure cannot adapt to the problem.

Our approach is a constructive evolutionary approach to produce program graphs. The idea is to start from the leaves, which represent classes or labels, and iteratively add on top, nodes containing binary programs until a node is able to manage the entire problem. Using this approach, the output is printable, compact and tends to be more explainable.

The method combines several techniques: the use of local search to generate binary programs [12, 13, 16] and crossover between program graphs in order to build more complex structures able to manage multi-class classification.

Some research has been done combining ensemble learning and genetic programming [12]. While performance may increase due to the ensemble approach, we claim that this gain is not worth the cost in compactness, explicability and frugality. To ensure explainability, our approach does not make use of ensemble techniques which avoids aggregating strategies such as voting.

With this work, we aim to demonstrate the value of DAPGs in classification tasks. The rest of the paper is organized as follows. In Section 2 we describe binary program generation and show results obtained with a binary problem. Then, in Section 3, we present our evolutionary algorithm, building DAPGs with binary programs on multi-class problems. We also present results obtained with our

GECCO '24 Companion, July 14–18, 2024, Melbourne, VIC, Australia
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0495-6/24/07...$15.00
https://doi.org/10.1145/3638530.3664115

method against similar methods like TPG and TnT in Section 3.3 Finally, we conclude in Section 4.

2 BINARY PROGRAM

First we present how binary programs are built, using operators and features, and optimized through local search with neutral adaptive walk. Then, we present our binary program's performance on a difficult binary classification dataset as well as the output given by our method.

2.1 Program's construction

The base component of our method is a program that acts as a binary classifier. The program takes, as inputs, an array containing the features, all between 0 and 1 due to min-max normalization of numerical features or one-hot encoding of categorical ones, of the instance to be classified and returns a real value. This value is compared to a threshold[1] to determine the class of the instance. If the output value is above or equal to the threshold, the class is 0; otherwise, it is 1. To accelerate the program search process, when a program's accuracy score on the training set falls below 50%, we reverse its decision. In other words, if the program originally returns 0, we make it return 1, and vice versa. This allows us to automatically generate programs with better training performances than random.

A program is simply a fixed length sequence of I instructions, where each instruction consists of an operator and two operands. The form of these programs is similar to those handled by linear genetic programming [10], except that there is only one internal register, modified as the execution progresses. The available operators are shown in table 1[2]. It is worth noting that none of them use constant values, and each instruction is always dealing with feature values.

Function	Description
ADD(A,B)	return A+B
MINUS(A,B)	return A-B
MUL(A,B)	return A×B
MIN(A,B)	return A if A<B else B
MAX(A,B)	return A if A>B else B
GREATER(A,B)	return A if A>B else -A

Table 1: List of instructions available for program.

The first instruction in a program has its two operands drawn from the features of the instance to be classified. Any subsequent instruction takes one operand from the instance's features and the result of the previous instruction. The program returns the result of the final instruction as shown in figure 1, which is the python transcription of one program obtained. To export programs in python, and for reasons of implementation simplicity, inputs are a register of size of number of features plus one. The first cell of the register, $R[0]$ is used as an intermediate result variable, and the rest of the register is filled with the normalized feature values. Due to the accumulative nature of our program, $R[0]$ is returned as the final value.

We generate programs with operators and operands randomly picked from the operator list and feature list for a fixed number of instructions and then apply local search through neutral adaptive walk to search for better programs. The first randomly generated program becomes the current solution then, adaptive walk iteratively applies the mutation operator, and updates the current solution only if the new solution is *at least* as good as the current solution. It is worth noticing that contrary to classical adaptive walk, we allow the current solution to be updated even if the new solution is not better than the current one. This seems to perform better when the search space is neutral [3], as it is the case in classification problems optimized on accuracy.

At the end of the walk, when a maximum number of steps is reached or when a program obtains 100% accuracy in the training set, we return the current solution, which is the best program found through the walk. Mutation operator is a hyper-parameter and is chosen from a list of different mutation functions as the most suitable for the problem we are dealing with.

2.2 Results

We tested our binary program construction method on the GAMETES[3] dataset, which is, regarding [12], the most challenging one for all algorithms presented as none of them seem to be able to achieve more than 55 percent accuracy in the testing set within this dataset. GAMETES dataset contains 1600 instances, 1000 numerical features, and two classes. This dataset is balanced, which is crucial as we use accuracy to measure performance. Mutation used for this dataset works as follows: for each instruction, we change either the operator or every operand other than $R[0]$, based on a fifty percent chance. The length of the programs is experimentally set to 7 with a systematic search, and the maximum number of steps of the walk is set to 10 million.

Experiments were performed on a Gigabyte AMD EPYC 7003 DP server system, with 2 AMD ROME 7662 processors and Linux operating system. Our method is implemented in the C language, and expriments have been made over 30 independant seeds, including the train (70%) / test (30%) split.

In order to compare ourself with state-of-the-art algorithms, we extracted performance of Standard Genetic Programming (GP), Multidimensional Multiclass GP with Multidimensional Populations (M3GP) and Ensemble Genetic Programming (eGP) from [12]. Since performance was depicted via boxplots, we filled our results table with values that appeared to approximate the data distribution. Performance of our binary program is presented in table 2.

Although the values read may not be exactly those due to the challenging interpretation of exact values in a boxplot, we are confident that this will not skew the comparison, as our results are well above the margin of error in interpretation.

Our method is better on testing accuracy and seems to outperform other Genetic Programming algorithms. Furthermore, this approach does not suffer from overfitting on this dataset. As the

[1]In the following, the threshold is set to 0.5, which makes sense as features are normalized between 0 and 1 and there are no constants in the programs.
[2]We followed the usage of GP to refer to an *expression* as a *program*

[3]https://www.openml.org/search?type=data&sort=runs&id=40645&status=active

	Train	Test	Size
GP*	55±1	50±1	400±100
M3GP*	75±1.5	49±1.5	200±10
eGP-W5*	62±0.2	50±1	225±30
Binary Program	74.78±4.22	73.32±6.19	7±0

Table 2: Performance with training and testing set on GAMETES (%). Size is in number of instructions.
***Based on scores from [12].**

program obtained contains only 7 instructions, it is very energy-efficient and explainable. The entire program achieving 73% of accuracy on testing set is shown in figure 1.

```
def prog0(R):
    R[0] = R[374] + R[286]
    R[0] = R[0] * R[266]
    R[0] = min(R[0], R[338])
    R[0] = R[0] * R[703]
    R[0] = R[0] + R[1000]
    R[0] = R[0] - R[999]
    R[0] = R[0] if R[0]>R[289] else -R[0]
    return R[0]
```

Figure 1: Python transcript of the program obtained on GAMETES dataset.

In conclusion, we saw that our method for generating binary program seems to be efficient on binary classification. We can now use it as basic element in buidling DAPG.

3 DIRECTED ACYCLIC PROGRAM GRAPH (DAPG)

3.1 Handling multi-class problems

To handle multiclass problems, we utilize a directed acyclic graph comprising of two kinds of nodes:

(1) Leaves, which lack successors and represent the labels of the problem.
(2) Program nodes which possess two successors and a variable number of predecessors. If a program node has no predecessors, it is termed a root.

Considering a root and an instance to be classified, the prediction of the label is done as follows : First, the current node is set as the root and the program associated is executed on the instance. The result of the program determines which successor to move to. If the result is less (resp. more) than the threshold (0.5), the left (resp. right) successor is chosen as the current node. This process continues until a leaf node is reached, at which point the corresponding label is returned. This approach enables the utilization of binary programs to optimize multiclass classification problems.

3.2 Evolutionary algorithm to optimize DAPG

The graph construction is achieved through an evolutionary process, inspired by the TPG construction procedure. This process is broken down into initialization, a construction loop, and a post-optimization procedure aimed at eliminating redundant nodes.

Initially, leaves are generated, with one corresponding to each label of the problem and containing no program. Additionally, a population of potential successors, denoted as S, is initialized, which initially includes each leaf. Finally, an archive A is created specifically to store useful programs.

During each iteration, the population of successors includes the most promising nodes of the graph based on their training dataset performances. These nodes in the population are candidates for mating, facilitating the creation of new nodes. Subsequently, for each iteration, we generate a pool of N (set to 5 in the following) nodes, formed through crossover between two nodes from the population of potential successors.

Let n_1 and n_2 be two potential successors; the score of the pair is determined by the number of instances correctly classified by n_1 as a root or by n_2 as a root. This score represents the maximum achievable accuracy through a crossover between n_1 and n_2. To select the two successors, we generate all possible pairs and compute their respective scores. Next, we randomly choose 30% of the possible pairs and finally select the most promising ones based on the computed scores.

Once the successors are chosen, the crossover proceeds as follows:

(1) A subproblem is formulated, considering only instances misclassified by exactly one of the successors. If an instance is misclassified by the first successor, it is labeled as one; otherwise, it is labeled as zero. This created subproblem becomes a binary classification problem.
(2) A starting program is selected in the archive A based on its performance on the generated subproblem.
(3) Subsequently, we utilize the adaptive walk described in Section 2 of this paper to optimize this program considering the subproblem.

Once the generation of the pool is completed, we compare each new node n_{new} with each item n_{old} contained in the population of successors, using the following comparison function applied to the entire training dataset:

$$OP(n_{new}, n_{old}) = \begin{cases} 1 & \text{if } n_{new} \text{ well classifies all instances} \\ & \text{well-classified by } n_{old} \\ 0 & \text{otherwise} \end{cases}$$

A new node n_{new} is inserted in the population S only if no existing successors n_{old} in the list outperforms n_{new}. Conversely, any existing successors n_{old} are removed from the population S if at least one new node n_{new} ouperforms them.

Once the maximum number of iterations is reached, we designate the node with the highest accuracy score as the root r, and eliminate all nodes that are not part of the spanning tree of r.

3.3 Results

We chose two multi-class benchmark problems, PENDIGITS and MNIST. Both are digit image classification with 10 classes. PENDIGITS contains 16 features and 11k instances while MNIST contains 784 features and 70k instances. Experiments have been made in the same environment as GAMETES except that the train - test

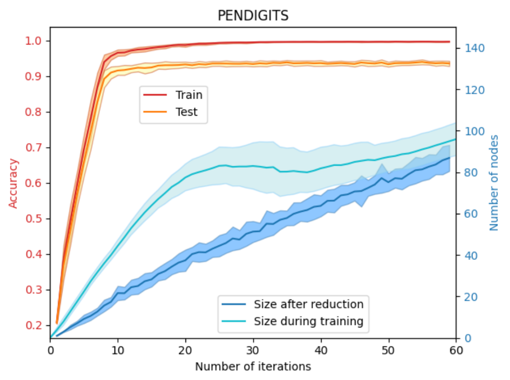

Figure 2: Performances* on training and testing set on PENDIGITS with respect to the number of iterations.
***Accuracy and size are averaged over 30 independant seeds.**

split was the same for the 30 seeds. For both datasets, the mutation operator functions in the following manner: it randomly selects one instruction and completely regenerates it, including both the operator and features operands. Maximum number of steps of walk is set to 50k for both datasets.

	Train	Test	Size
TPG*	66.34±1.66	63.32±2.33	49±9
TNT*	-	92.61±0.53	125±0
DAPG	99.35±0.19	93.54±0.66	44.43±5.12

Table 3: Performances on training and testing set on PENDIGITS (%). Size is in number of nodes.
***Based on scores from [2] and [17].**

On PENDIGITS in table 3, DAPG is better than TPG and TnT while preserving compactness. Based on Figure 2, we chose to stop convergence after 25 iterations, since it allows convergence on the training set with a minimal length. Other tradeoffs could be chosen, for example lower performing but smaller programs. According to parameters, the theoretical growth is 5 nodes per iteration. However, when considering raw size (before final reduction) expansion, the algorithm demonstrates a nuanced behavior. Initially, during iterations in the range of 0 to 20, it generates 2.5 nodes per iteration, indicating a rejection of half of the nodes produced. This is followed by a period of stagnation or decline from iterations 20 to 40, where the algorithm discards all generated nodes or maintains a one-to-one ratio of generation to destruction. Beyond iteration 40, the rate drops to less than one node per iteration. Conversely, in the reduced size, which is the length of the DAPG free from unreachable nodes, we find a stable growth slightly exceeding one node per iteration. It is worth noticing that TPG, which has a similar model structure, does not converge on multi-class problems, as shown in [2], the results are reported in table 3.

On MNIST in table 4,results are obtained after 30 iterations and DAPG is also better than TPG and still compact. It is worth noting that the "Size" in table 4 is the reduced size. However, DAPG does not reach TnT performance on testing accuracy. As depicted in Figure 3,

	Train	Test	Size
TPG*	46.76±1.3	47.35±1.42	55±14
TNT*	-	90.87±0.31	600±0
DAPG	85.80±1.56	84.89±1.34	41.03±7.63

Table 4: Performances on training and testing set on MNIST (%). Size is in number of nodes.
***Based on scores from [2] and [17].**

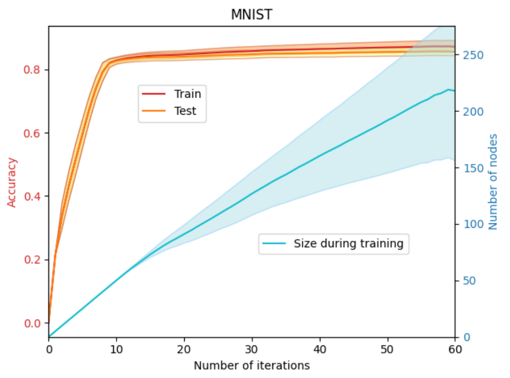

Figure 3: Performances* on training and testing set on MNIST with respect to the number of iterations.
***Accuracy and size are averaged over 30 independant seeds.**

convergence appears to be restricted. This limitation likely arises from the persisting challenges in the task, which could explain the asymptotic performance curve observed, as illustrated in Figure 2, even in PENDIGITS.

4 CONCLUSION

In conclusion, our binary programs seem to be effective compared to the literature. Maybe a stack representation of programs, as [14], could lead to better results due to new crossover and mutation mechanisms, at the expense of introducing introns. The fixed length of programs could be seen as a limitation, as it could be interesting to have a variable length program that adapts to problem difficulty. On the other hand, this will surely introduce bloat, which is a problem in genetic programming.

A first approach of Directed Acyclic Program Graph has been made, inspired by TPG, which strongly improves performance on supervised classification tasks. However, TnT, which uses a top-down construction, like decision trees, seems to be better with some classification problems. Future work will be to try to combine top-down, like TnT, and bottom-up, like TPG and DAPG, construction in the same approach to create graphs of programs even more efficient but still compact and explainable.

ACKNOWLEDGMENTS

We are grateful to the reviewers for their valuable comments and suggestions which helped us to improve the paper. Support for the first author (PhD scholarship) from the department council of La Mayenne (France) is acknowledged.

REFERENCES

[1] Erin L. Allwein, Robert E. Schapire, and Yoram Singer. 2001. Reducing multiclass to binary: a unifying approach for margin classifiers. *The Journal of Machine Learning Research* 1 (Sept. 2001), 113–141.

[2] Thibaut Bellanger, Matthieu Berre, Manuel Clergue, and Jin-Kao Hao. 2023. A One-Vs-One Approach to Improve Tangled Program Graph Performance on Classification Tasks:. In *Proceedings of the 15th International Joint Conference on Computational Intelligence*. SCITEPRESS - Science and Technology Publications, Rome, Italy, 53–63.

[3] Manuel Clergue, Sébastien Verel, and Enrico Formenti. 2018. An Iterated Local Search to find many solutions of the 6-states Firing Squad Synchronization Problem. *Applied Soft Computing* 66 (May 2018), 449–461.

[4] Vinícius G. Costa and Carlos E. Pedreira. 2023. Recent advances in decision trees: an updated survey. *Artificial Intelligence Review* 56 (Oct. 2023), 4765–4800.

[5] Alberto Fernández, Victoria López, Mikel Galar, María José del Jesus, and Francisco Herrera. 2013. Analysing the classification of imbalanced data-sets with multiple classes: Binarization techniques and ad-hoc approaches. *Knowledge-Based Systems* 42 (April 2013), 97–110.

[6] Mikel Galar, Alberto Fernández, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. 2011. An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes. *Pattern Recognition* 44, 8 (Aug. 2011), 1761–1776.

[7] Trevor Hastie and Robert Tibshirani. 1998. Classification by pairwise coupling. *The Annals of Statistics* 26, 2 (April 1998), 451–471. Publisher: Institute of Mathematical Statistics.

[8] Seokho Kang, Sungzoon Cho, and Pilsung Kang. 2015. Constructing a multi-class classifier using one-against-one approach with different binary classifiers. *Neurocomputing* 149 (Feb. 2015), 677–682.

[9] Stephen Kelly and Malcolm I. Heywood. 2017. Emergent Tangled Graph Representations for Atari Game Playing Agents. In *Genetic Programming*, James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.). Vol. 10196. Springer International Publishing, Cham, 64–79. Series Title: Lecture Notes in Computer Science.

[10] Wolfgang Banzhaf Markus F. Brameier. 2007. Basic Concepts of Linear Genetic Programming. In *Linear Genetic Programming*. Springer US, Boston, MA, 13–34.

[11] John Platt, Nello Cristianini, and John Shawe-Taylor. 1999. Large Margin DAGs for Multiclass Classification. In *Advances in Neural Information Processing Systems*, Vol. 12. MIT Press.

[12] Nuno M. Rodrigues, João E. Batista, and Sara Silva. 2020. Ensemble Genetic Programming. In *Genetic Programming (Lecture Notes in Computer Science)*, Ting Hu, Nuno Lourenço, Eric Medvet, and Federico Divina (Eds.). Springer International Publishing, Cham, 151–166.

[13] Leo Willyanto Santoso, Bhopendra Singh, S. Suman Rajest, R. Regin, and Karrar Hameed Kadhim. 2021. A Genetic Programming Approach to Binary Classification Problem. *EAI Endorsed Transactions on Energy Web* 8, 31 (2021), e11–e11. Number: 31.

[14] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3 (2002), 7–40.

[15] Akbar Telikani, Amirhessam Tahmassebi, Wolfgang Banzhaf, and Amir H. Gandomi. 2021. Evolutionary Machine Learning: A Survey. *Comput. Surveys* 54, 8 (Oct. 2021), 161:1–161:35. QID: Q114071181.

[16] Emigdio Z-Flores, Leonardo Trujillo, Oliver Schütze, and Pierrick Legrand. 2015. A Local Search Approach to Genetic Programming for Binary Classification. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, Madrid Spain, 1151–1158.

[17] Bingzhao Zhu and Mahsa Shoaran. 2021. Tree in Tree: from Decision Trees to Decision Graphs. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 13707–13718.