Evolving Bagging Ensembles with Genetic Programming

Korbinian Federholzner

Ludwig Maximilian University of Munich, Munich 80539, Germany
K.Federholzner@campus.lmu.de

Abstract. *Bagging* is a common method in Machine Learning to get more accuracy for predictors. Applied to *Genetic Programming (GP)* this can have a big advantage, in getting better results. There are multiple implementations of such algorithms that focus on getting better performance on *GP* with *Bagging*. Most of those algorithms tend to be complex in implementation and fast or simple and slow. The algorithm presented in this paper called *Simple Simultaneous Ensemble Genetic Programming (2SEGP)* tries to extend classic *GP* with *bagging* only by a few features to be simple as possible but also provides a similar speed to the more complex implementations.

Keywords: Genetic programming · ensemble learning · machine learning · bagging · evolutionary algorithms

1 Introduction

GP algorithms are typically used to approximate solutions to NP-Hard Problems as well as other optimization problems that normally would be too computationally expensive. Even though those solutions are often good enough they can never be viewed as optimal due to the random nature of *GP*. In Machine Learning (ML) a technique called *bagging* is used to get a better approximation of the resulting predictors. Some *GP* algorithms try to use *bagging* because of that reason. Using *bagging* without modifying the *GP* algorithm will increase the run time by a lot and thus a lot of complex *GP* algorithms have emerged that try to tackle this issue. One algorithm *2SEGP*, the algorithm discussed in this paper, does this by also trying to not have the complexity of implementation that would normally come from those modifications. This paper starts in section 2 by explaining the basics on which most *GP* algorithms are built. Followed by section 3 in which the basics of *bagging* are shown. Section 4 than shows how *2SEGP* unites the basics explained in the previous sections to a new algorithm. Lastly, the paper is finished off with a Conclusion in section 5.

2 Genetic Programming

The Darwinian theory by Charles Darwin is the theory, that individuals of a species of organisms survive natural selection and develop with the help of re-

production and inheritance [5]. A class of algorithms, in *computational intelligence*, called the *Evolutionary Algorithms*, try to model this theory. One of those algorithms, *Genetic Programming (GP)* does this with the help of syntax trees. The nodes of a syntax tree are represented by arithmetic operations (+, -, *, /, min, ...) called functions, while the leaves are variables and constants (x, a, 1, 2, ...) called terminals. Traversing the tree is done bottom-up from a leaf to the root node. Figure 1 shows a syntax tree that is constructed from the formula: $(\frac{y}{2.33}) * \log(x) + \max((x+2), x)$

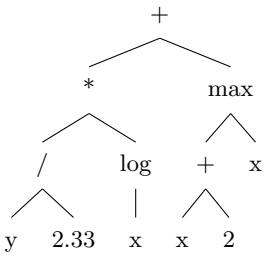

Fig. 1. Syntax tree of a *GP* algorithm with the formula: $(\frac{y}{2.33}) * \log(x) + \max((x+2), x)$

GP algorithms generally have the same structure, as shown in figure 2. They start with an initial population of randomly created subtrees. Those individuals get their fitness value calculated. Followed by a selection, weighted by the fitness values, in which the parents are chosen. In the crossover step, the parents create new offspring until the new population has the same size of individuals as the old population. The newly created children each have a small chance to be mutated. This procedure gets repeated until the termination criteria is met. This can be for example either a fixed maximum of iterations or iteration until the newly created children do not change anymore.

2.1 Initial Population

As a starting point for a given number of individuals in the population, there are N random trees generated. From a given set of functions and terminals, each node of a tree gets assigned a random function and each leaf a random terminal. There are multiple ways of filling up the tree, such as the *grow*, *full* and *ramped half-and-half* methods. Essentially a tree gets filled until a specified depth. In the *full* method, all leaves have the same depth. The *grow* method stops when it hits the depth limit and then fills its leaves [13]. This results in a tree, that has terminals on different depth levels.*Ramped half-and-half* initializes half the population with the *full* method and the other half with the *grow* method [9]. Figure 3 shows two trees generated from *full* and *grow*.

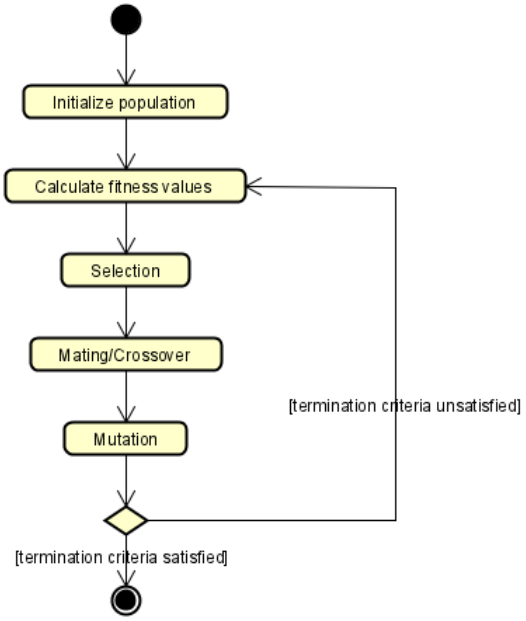

Fig. 2. An overview of the general procedure of a *GP* algorithm

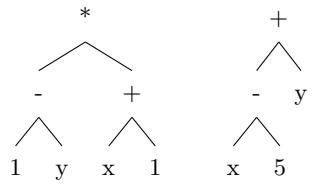

Fig. 3. The left syntax tree shows a tree generated with the *full* method and the right tree shows a tree generated with the *grow* method

2.2 Fitness

Similar to the phrase "survival of the fittest" from Darwin's evolutionary theory, the best individuals in the population of a *GP* algorithm should have the highest probability to survive and create new offspring. The performance of an individual is measured with a fitness function. The closer the value of a fitness function is to the desired value the better the individual. The fittest individuals have the highest chance to mate and create new children for the next generation. Fitness functions often get measured with *mean squared error (MSE)* or the *mean absolute error (MAE)*.

2.3 Selection

For the creation of a new generation, a selection process is needed. In this process, the parents are getting chosen to favor the individuals with the best fitness values. Choosing the $\frac{1}{p}$ fittest individuals is called *truncation selection*. The advantage of *truncation selection* is that it is very easy to implement and very efficient. But choosing only the fittest individuals is not always the best idea, since it might come with a loss of diversity. This is because the children of the fittest individuals will most likely be the fittest individuals in the next generation. Since the fittest will also have most children, this will lead to a population where almost all individuals have some sort of heritage from the same parents and thus converge to a single individual. In *tournament selection*, n random individuals from the population are chosen, which then get compared to each other. The fittest of them are chosen to be parents. Two or more tournament-winning individuals are then used for the generation of a new child. This procedure is repeated until there have been enough parents chosen to create a new generation has the same size as the old population [12]. *Tournament selection* on the other hand guarantees more diversity because individuals are chosen at random. There could be a scenario where the best individual never gets chosen for a tournament and thus never gets to create a child.

2.4 Crossover

In Crossover two or more individuals that have survived the selection produce offspring. The goal of this step is to create a new population of children of the same size as the previous one. In *GP* the most common variant is *subtree crossover*. First, a node, called the crossover point is selected for each parent, then everything above, up to the root node, is removed in the first parent. For the second parent, the crossover point and everything below it is removed. Both then merge at crossover points, as illustrated in Figure 4. All of those operations are done on a copy of the parents to not lose the parents because they could be needed for the creation of more offspring. The crossover operation can be set to a certain probability (commonly 90%), which means it is not applied every time. In that case, either *mutation* or *reproduction* is used. *Reproduction* simply copies the fitter parent to the next generation.

2.5 Mutation

Since the initial population is generated randomly, some possible subtree combinations might be missing. To combat this, *mutation* is used. The most common mutation variant called is *headless chicken crossover* [11]. It selects a random node of the tree and substitutes it with a new randomly created subtree. Another form of mutation is *point mutation* [7]. In this form, a random node gets selected, but instead of replacing the whole subtree, only the node itself gets changed to a new primitive. Generally, the *mutation rate* is very small, most of the time around 1%. Figure 5 takes the child produced in figure 4 and shows an example of what applying those mutations could look like.

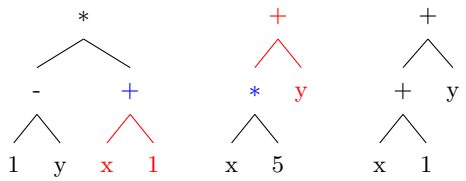

Fig. 4. The blue functions mark the crossover points and the red marked subtrees are the subtrees used for the offspring. The third tree shows the resulting child for the next generation

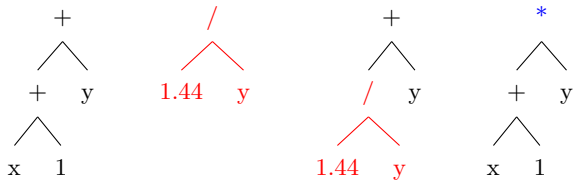

Fig. 5. The first tree is the child from figure 4, the second red tree is the random subtree which was produced using *headless chicken crossover*, which then gets applied to result in tree number three. On the last tree, *point* mutation was applied on the root node.

3 Bagging

Bagging, also known as ***b**ootstrap **agg**regat**ing***, is a method for gaining more accuracy through the aggregation of multiple predictors. To get β predictors, the training set T has to be sampled with replacement, to get multiple realizations $T_1, T_2, ..., T_\beta$, also called bootstrap samples, of the original training set T. The predictors trained by those training sets, have to be aggregated, traditionally with a majority voting for classification or averaging for regression [4].

4 Simple Simultaneous Ensemble Genetic Programming (2SEGP)

There are many algorithms that combine *Bagging* and *GP*, those can be categorized into *Simple Independent Ensemble Learning Approaches(SIEL)* and *Complex Simultaneous Ensemble Learning Algorithms (CSEL)*. *SIEL*-applications, are very simple, they execute *GP* on each of the bootstrapped training sets and then aggregate all estimators to a single estimation. This makes them very simple to implement, but inefficient. *CSEL*-algorithms, on the other hand, are very complex and efficient. They try to evaluate an ensemble in one go. *2SEGP* is an algorithm that tries to have the simplicity of a *SIEL*-application, while being comparable in performance to a *CSEL*-Algorithm. To accomplish this the *2SEGP* changes two classical aspects of a generic *SIEL*-application, the individual selection and the fitness function. Other than that, the initialization uses

Ramped half-and-half, but it increments the depth of the trees, to get even more variation. The calculation of the fitness value uses the *MSE*, on which *linear scaling* is applied [8]. Mating is done with *subtree crossover* and mutation with *headless chicken crossover* [16].

4.1 Individual Selection

The *2SEGP* algorithm modifies the selection of *bagging* slightly. Each individual gets evaluated uniformly across all bootstrap samples, which means it doesn't have a single fitness value, it has β of them. Each individual, disregarding fitness, creates a child with either the *mutation* or *subtree crossover* operations applied. In the case of *subtree crossover*, another random individual is selected. The reason for not taking fitness into account when producing children is, that most of the time the top-ranked individuals are very similar to each other. This leads to a drop-off in the variation of individuals. Instead, each child gets calculated a new fitness value. Together with all parents and children, the top $\frac{n_{pop}}{\beta}$ individuals, ranked by fitness, for each bootstrap sample $T_1, T_2, ..., T_\beta$, get chosen. This has a similar effect as just using the best parents for reproduction since bad individuals are not selected for the next generation.

4.2 Changes to Fitness-Evaluation

Since each individual gets evaluated over all bootstrap samples $T_1, T_2, ..., T_\beta$ and the size of each sample is the same as the original sample T, this would lead to a high cost of computation. *2SEGP* utilizes the property that each sample is derived from the original sample T. Instead of computing a value for each $T_1, T_2, ..., T_\beta$, the value of an individual gets just computed once for T. A separate collection of indices is held for each bootstrap sample $S_1, S_2, ..., S_\beta$. Each S_j contains n indices that correspond with an index from the original sample T. Rather than recomputing each sample, the value computed for the original T corresponding to the index gets used. This results in the computational complexity of $O(ln) + O(n\beta) + O(n\beta) = O(n(l + \beta))$. where l is the operation cost for traversing the whole tree and n is the number of samples of the training set. In the case of an *SIEL*-algorithm, this complexity would be $O(nl\beta)$, since every individual would have to be trained n times on every bootstrap sample $T_1, T_2, ..., T_\beta$.

4.3 Linear Scaling of fitness

Scaling the fitness function can have huge benefits since it helps not run into dead ends, where the most dominant individuals of a population all converge into one individual, thus losing diversity. If done right Scaling will increase the performance of a *GP* algorithm. There are a lot of different ways to scale the fitness function, *2SEGP* uses Linear Scaling. This reduces the *MSE* of a syntax tree, by applying an optimal linear transformation to the outputs. It's computed by applying the slope and the intercept and then scaling the *MSE* with it:

$$a = \overline{y} - b\overline{o} \tag{1}$$

$$b = \frac{\sum_{i=1}^{n}(y_i - \overline{y})(o_i - \overline{o})}{\sum_{i=1}^{n}(o_i - \overline{o})^2} \tag{2}$$

$$MSE^{a,b}(y, o) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (a + bo_i))^2 \tag{3}$$

Where y contains the labels, o the outputs of the syntax trees after the evaluation, n the size of the data set and a, b being both of the coefficients. $\overline{o}$ and $\overline{y}$ represent the mean of the respective variables. This doesn't increase the complexity by a lot, but increases the performance, for a lot of problems.

4.4 Example run

To show a very simple example run of *2SEGP* the code from [1] is used. For the parameters, a population size of $n_{pop} = 4$ and an ensemble size of $\beta = 2$ are chosen. As data set T the Boston housing prices are used, which has $|T| = 500$ samples and $d = 13$ features. This results in a function set of $\{+, -, *, /\}$ and a terminal set of $\{erc, x_1, x_2, ..., x_{13}\}$. erc is Ephemeral Random Constant (ERC) as proposed by Koza [9] and describes a random numerical constant that is inserted in case it gets chosen. Each of the 13 features from the data set, gets a variable $(x_1, x_2, ..., x_{13})$. Since trees get chosen by *Ramped half-and-half* and depth starts at 2, with increments until a threshold, the random trees generated will look like the trees in Figure 6.

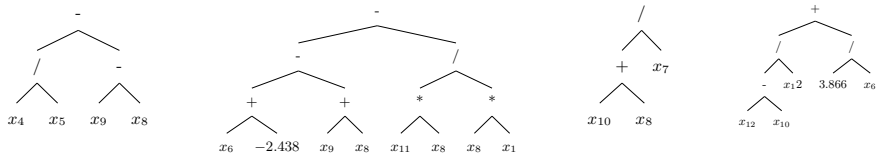

Fig. 6. In the initial population, the top two trees are created by *full* and the bottom two by *grow*

For reproduction, all of the created trees will produce the children with the help of *subtree crossover* and *mutation*. For this example, both the probability of *mutation* and *subtree crossover* is set to 50%. Note that this is an example and a lower *mutation* rate could be better. Because of the selection process, all parents get a chance to be included in the next generation anyways, *reproduction* is not included at all. Figure 7 shows all children that got generated.

Each child and parent get their fitness value calculated, one for every value in the samples of T_1 and T_2. The top $\frac{n_{pop}}{\beta}$ individuals for each sample are chosen to be in the next population. Figure 8 shows the individual selection.

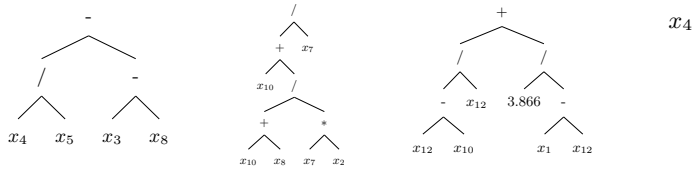

Fig. 7. For the children created from the initial population, the tree with only root node x_4 got selected for *mutation* and got substituted by only one terminal

	T1	T2		
parents				new Population:
(-), (/,-)	58.77	50.59		(-), (/,-)
(-), (-,/)	57.67	70.9		(-), (-,/)
(/)	55.23	65.35		(/)
(+)	88.2	75.1		(x_4)
children				
(-)	64.06	57.2		
(/)	65.35	61.23		
(+)	75.36	73.3		
(x_4)	64.85	52.67		

Fig. 8. The fitness evaluation of the population. Green individuals are included in the next population, the other red ones are getting discarded

After the new population is formed, the *2SEGP* algorithm, repeats from calculating fitness again. This gets repeated until a certain time is elapsed, or a specified number of populations are created. The resulting individuals are the β elites, the best of each sample $T_1, T_2, ...T_\beta$. A majority vote between them will determine the final result for given inputs. After 30 generations, in the example case, the resulting elites both have the same structure. Figure 9, shows the resulting elites, with both having a fitness value of 30.315 for T_1 and 29.998 for T_2.

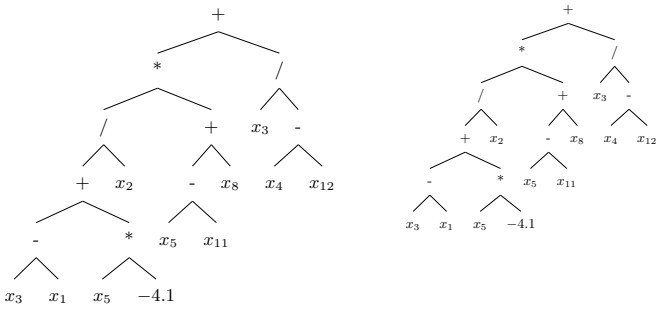

Fig. 9. The resulting elites, in this example the last two individuals are exactly the same

4.5 Comparision with state-of-the-art algorithms

The paper [16] compares *2SEGP* to other state-of-the-art algorithms, by comparing them on accuracy. On both the tasks of classification and regression, the represented algorithms that are considered some of the leading ones for those tasks are chosen. The algorithms for classification are:

- Diverse Niching Genetic Programming (DNGP) [19]
- Ensemble GP with weighted voting (eGPw) [14]
- Multidimensional Multiclass GP with Multidimensional Populations (M3GP) [14]
- Classic Genetic Programming (cGP)

And for the regression task:

- Diverse Niching Genetic Programming (DNGP) [19]
- Spatial Structure with Bootstrapped Elitism (SS+BE) [6]
- Gene-pool Optimal Mixing Evolutionary Algorithm GP-GOMEA [18]
- Random Desired Operator RDO [17]
- Genetic Programming Toolbox for The Identification of Physica Systems (GPTIPS) [15]
- Evolutionary Feature Synthesis (EFS) [3]
- Geometric Semantic Genetic Programming with Reduced Trees GSGP-Red [10]
- Classic Genetic Programming (cGP)

All of those algorithms get compared by some selected real-world datasets from the UCI repository [2]. The Figure 10 shows the resulting benchmarks from testing *2SEGP* in the classification task. Figure 11 on the other hand displays the results of the regression task. Images 10 shows the accuracy ± interquartile range on how accurate the prediction of each algorithm was on the respective dataset. The other image, 11 measures the performance with the Median root-mean-square deviation (RMSE) ± interquartile range The results show that

	Training					Test				
Algorithm	BCW	Heart	Iono	Parks	Sonar	BCW	Heart	Iono	Parks	Sonar
2SEGP (ours)	0.995±0.005	0.944±0.022	0.976±0.017	0.948±0.011	0.966±0.034	0.965±0.018	0.815±0.062	0.896±0.047	0.936±0.012	0.738±0.067
w/oLS (ours)	0.995±0.006	0.947±0.021	0.978±0.012	0.892±0.021	0.959±0.036	0.965±0.013	0.809±0.049	0.896±0.047	0.885±0.031	0.754±0.067
DNGP	0.979±0.010	0.915±0.021	0.955±0.015	0.931±0.057	0.924±0.043	0.959±0.019	0.815±0.049	0.901±0.026	0.917±0.055	0.730±0.063
eGPw	0.983±0.008	0.907±0.025	0.884±0.032	0.923±0.042	0.924±0.034	0.956±0.018	0.790±0.034	0.830±0.057	0.822±0.064	0.762±0.060
M3GP	0.971±0.002	0.970±0.017	0.932±0.042	0.981±0.024	1.000±0.012	0.957±0.014	0.778±0.069	0.871±0.057	0.897±0.051	0.810±0.071
cGP	0.964±0.016	0.825±0.033	0.773±0.060	0.842±0.077	0.769±0.055	0.961±0.018	0.784±0.049	0.745±0.057	0.797±0.102	0.714±0.044

Fig. 10. Benchmark results on the classification task, w/oLS is *2SEGP* without linear scaling

bagging helps with increasing the accuracy of *2SEGP* so much, that is just as good as most of the state-of-art algorithms, even outperforming them on a lot of tasks.

	Training				Test			
Algorithm	ASN	CCS	ENC	ENH	ASN	CCS	ENC	ENH
2SEGP (ours)	2.899±0.290	5.822±0.353	1.606±0.200	0.886±0.556	3.082★ ±0.438	6.565±0.439	1.801±0.263	0.961±0.553
DivNichGP	3.360±0.343	6.615±0.454	1.809±0.190	1.079±0.415	3.458 ±0.487	7.031±0.370	1.930±0.156	1.158±0.398
SS+BE	3.271±0.316	6.517±0.412	1.882±0.363	1.190±0.291	3.416 ±0.333	6.986±0.744	1.946±0.380	1.204±0.366
GP-GOMEA	3.264±0.172	6.286±0.300	1.589±0.079	0.739±0.138	3.346 ±0.238	6.777±0.313	1.702±0.200	0.804±0.184
RDO^{xLS}_{+LS}	3.482±0.172	6.476±0.249	1.703±0.125	0.819±0.186	3.579 ±0.245	6.800±0.423	1.791±0.180	0.881±0.309
cGP	3.160±0.295	6.279±0.305	1.851±0.441	1.196±0.431	3.359 ±0.379	6.759±0.623	2.041±0.383	1.267±0.556
GPTIPS	-	-	-	-	4.138	8.762	2.907	2.538
mGPTIPS	-	-	-	-	4.003	7.178	2.278	1.717
EFS	-	-	-	-	3.623	6.429★	1.640★	0.546★
GSGP-Red	-	-	-	-	12.140	8.798	3.172	2.726

Fig. 11. Benchmark results on the regression task

5 Conclusion

Combining *bagging* with *GP* does have major benefits in accuracy, which makes it better than pure *GP*. The only downside of this combination is the increase in computational complexity since each sample created by *bagging* will increase the time needed. The algorithm *2SEGP* presented in this paper shows that only a few minor modifications are enough to reduce this increase in runtime, while still keeping the major benefits of the accuracy gained from *bagging*.

References

1. Simple simultaneous ensemble genetic programming (2017), https://github.com/marcovirgolin/2SEGP
2. Uci machine learning repository. (2017), http://archive.ics.uci.edu/ml
3. Arnaldo, I., O'Reilly, U.M., Veeramachaneni, K.: Building predictive models via feature synthesis. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. p. 983–990. GECCO '15, Association for Computing Machinery (2015)
4. Breiman, L.: Bagging predictors. Machine Learning **24**(2), 123–140 (1996)
5. Darwin, C.: On the Origin of Species by Means of Natural Selection. Murray, London (1859)
6. Dick, G., Owen, C.A., Whigham, P.A.: Evolving bagging ensembles using a spatially-structured niching method. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 418–425. GECCO '18, Association for Computing Machinery, New York, NY, USA (2018)
7. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, New York (1989)
8. Keijzer, M.: Scaled symbolic regression. Genetic Programming and Evolvable Machines **5**, 259–269 (09 2004)
9. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection. Complex adaptive systems, MIT Press, 1 edn. (1992)
10. Martins, J.F.B.S., Oliveira, L.O.V.B., Miranda, L.F., Casadei, F., Pappa, G.L.: Solving the exponential growth of symbolic regression trees in geometric semantic

genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 1151–1158. GECCO '18, Association for Computing Machinery, New York, NY, USA (2018)
11. McPhee, N.F., Ohs, B., Hutchison, T.: Semantic building blocks in genetic programming. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) Genetic Programming. pp. 134–145. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
12. Noraini Mohd Razali, J.G.: Genetic algorithm performance with different selection strategies in solving tsp. Proceedings of the World Congress on Engineering 2011 Vol II (2011)
13. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd (2008)
14. Rodrigues, N.M., Batista, J.E., Silva, S.: Ensemble genetic programming. CoRR (2020)
15. Searson, D.P.: GPTIPS 2: an open-source software platform for symbolic data mining. CoRR (2014)
16. Virgolin, M.: Genetic programming is naturally suited to evolve bagging ensembles. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 830–839. GECCO '21, Association for Computing Machinery, New York, NY, USA (2021)
17. Virgolin, M., Alderliesten, T., Bosman, P.A.N.: Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 1084–1092. GECCO '19, Association for Computing Machinery, New York, NY, USA (2019)
18. Virgolin, M., Alderliesten, T., Witteveen, C., Bosman, P.A.N.: Scalable genetic programming by gene-pool optimal mixing and input-space entropy-based building-block learning. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 1041–1048. GECCO '17, Association for Computing Machinery, New York, NY, USA (2017)
19. Wang, S., Mei, Y., Zhang, M.: Novel ensemble genetic programming hyper-heuristics for uncertain capacitated arc routing problem. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 1093–1101. GECCO '19, Association for Computing Machinery (2019)