

FINAL REPORT

1. Project Work

I did this project with two other student who are Anil Bayram Gogebakan whose number is s328470 and Oguzhan Akgun s328919. I contribute the project by creating Node class, writing random_tree, create_population, mutation_w_sa and cost functions. Also, I took part in writing some functions which are assign_population_fitness, simplify_constant, simplify_operators and migration.

We did this project in Anil Bayram Gogebakan's Github Repository. The link of the repository is https://github.com/anilbayramgogebakan/CI2024_project-work

1.1. Link

https://github.com/mericuluca/CI2024_project-work

1.2. Methodology

At first, the dataset is split into train and validation subsets. To avoid overfitting, we tried to calculate different costs and use them in different applications.

I created the node class. This class includes value, feature index, left, right and complexity. The Node can be an operator, constant or one of the features of x. The operators can be unary or binary. If the node is a unary operator, its value is the numpy function of that operator, its feature index is None, its left is another Node and its right is None. The complexity of this Node is calculated by multiplying the left node's complexity+1 by the complexity of the operator. If the node is a binary operator, its value is the numpy function of that operator, its feature index is None, its left is another Node, and its right is also another Node. The complexity of this Node is calculated by adding left node's complexity, right node's complexity and 1 and then the result of addition is multiplied the complexity of the operator. If the node is a constant, its value is a number, its feature index, left and right is None. The complexity of this Node is just 1. If the node is one of the features of x, its value is None, its feature index is the index of the feature, its left and right is None. The complexity of this Node is also 1.

There is also an Individual class, whose genome is Node. Individual also has fitness which is equal to the cost of that formula in genome calculated from train set, fitness_val which is equal to the cost of that formula in genome calculated from validation set, age which is the number of generations that individual exists and T which is the temperature value for simulated annealing used in mutation. Age is initialized as 0 and T is initialized as 1.

The cost(fitness) of the Individual is the mean squared error calculated by the train set and the fitness_val is the mean squared error calculated by the validation set.

Individuals are randomly created by giving the depth and the number of features in x.

4 different populations include 100 individuals are created. The depth of 50 of them is 0 and the depth of the other half is 7. However, some of the genomes of individuals include some part which are impossible to calculate. The calculation of costs of these individuals gives error. So, we deleted these individuals.

Each population generates new Individuals for 100 generations.

Generation Cycle

In each generation, at first, we increase all Individuals age by 1 and kill the ones whose age is more than 16 [1]. After that, we make a tournament selection to select which ones will be used to generate a new Individual [1].

The tournament selection selects n individuals randomly and compare their cost, the $n-1$ Individuals whose costs are low is eliminated and new n individuals selected, and this process continues as this until we have k number of winners to be used to generate new Individuals [1].

There are two ways of generating new Individuals which are crossover and mutation. The crossover chooses two random Individuals from the winners of tournament. Two children are generated by exchanging two randomly chosen nodes of two Individuals. The mutation is changing one randomly node of one randomly chosen Individual. However, we used simulated annealing. So, it checks the cost of the new Individual in every mutation. If its cost is smaller, we update the temperature value and add the child to the population. If its cost is larger, the probability is calculated and according to that probability we add the child to the population.

For 15 times which is BREED_NEW, we generate new Individuals from the winners of tournament. The new dataset includes some Individual genomes like (2+3). These genomes are simplified by `simplify_constant_population()` function. Then, we chose the best 3 Individuals. We called the elites. Elites are immortal in every generating we are finding new elites and update their age as 0. Finally, we check the population size, if its size is higher than 200, we took best 100 and continue.

Deduplicate Cycle

When the generation cycle repeated 15 times, we get into deduplicate cycle is done. In this cycle, the population is modified. For example, the genomes of Individuals can be the same. We delete the duplicates. Then, we simplify some of the operations like $\sin(1.3)$ and $\text{abs}(-1)$. Moreover, the model tends to create genomes which are just constants. We delete all these individuals. Additionally, to avoid overfitting, we want to limit the complexity of the genomes of Individuals and delete the complex ones. Finally, if the cost of best individual is less than 0.0001, we stop the evolution because we found a reasonable solution.

After 100 generation cycles, we have 4 evolved populations. From each population, we make tournament selection, and the winners migrate [1]. The populations exchange the winners in group of two. The new populations are evolved for 100 generation cycles. The migration process is repeated and the new populations are evolved for 100 generation cycles. The best Individuals from 4 population is compared and the best one is chosen as the result.

The results are too long and able to simplify. We wrote functions to simplify root Node. For example, if the node is $\sin(x[0]) + \sin(x[0])$ we wanted to make it $2*\sin(x[0])$. However, when we try that function the performance did not increase the performance. Moreover, not simplified versions can be useful for mutation and crossover [1].

1.3. Results

1.3.1. Problem_1

Found formula: $\text{np.sin}(x[0])$

MSE (train): 7.12594e-32

1.3.2. Problem_2

Found formula: $(\text{abs}((x[0] + x[0])) * ((\text{np.exp}(\text{abs}(x[0])) + \text{abs}(((\text{abs}(((x[0] - x[2]) + x[0]) / x[0])) + (x[0] + (((x[0] + ((x[0] + (x[0] + (x[0] - x[2])) * (\text{abs}(((x[0] + (x[0] + ((x[0] / x[0]) + (x[0] + ((x[0] - x[2]) + x[0])) + (\text{abs}(x[2]) + x[0])))) * x[0]) + x[0])) + \text{abs}(x[0])))) + x[0]) + (x[0] + x[0])) + (x[0] + x[0]) * \text{abs}((x[0] + x[0])))) + x[2])) * (x[0] + ((x[0] + ((x[0] + x[0]) + x[0])) + x[0])) + x[0]))$

MSE (train): 2.54801e+15

1.3.3. Problem_3

Found formula: $(((((\text{abs}(x[0]) - (x[2] - ((\text{abs}(x[0]) - (x[2] - (((\text{abs}(x[1]) - x[1]) - x[1]) - (((x[1] - ((\text{abs}(x[0] * \text{abs}(x[0])) - x[1]) + x[1])) - x[1]) - x[0]) - x[1]) - (((x[0] - ((x[1] - x[1]) - x[1])) + x[1]) - x[1]) - x[1])) - ((((((\text{abs}(x[1] + \text{abs}(x[1] * x[1])) - x[1]) - x[1]) - ((\text{abs}(x[0]) - (\text{abs}(x[1]) - ((\text{np.cos}(((x[0] - x[0]) - x[0]) - x[1]) - x[2])) - x[2])) - x[1]) - \text{abs}(x[0]) - x[1]) - ((x[1] - (\text{abs}(x[0]) - x[1]) - x[1])) - x[1]) - (((x[1] - x[2]) - x[1]) - x[1]) - ((x[1] - x[1]) + x[2])) - (((x[1] + \text{abs}(x[1] * x[1])) - x[1]) - x[1]) * x[1]))$

MSE (train): 423.184

1.3.4. Problem_4

Found formula: $((\text{np.cos}(x[1]) + \text{np.cos}(x[1])) + (((\text{np.cos}(x[1]) + (\text{np.cos}(x[1]) + \text{np.cos}(x[1])) + (\text{np.cos}(x[1]) + (\text{abs}(\text{abs}(x[1] / x[1])) + \text{abs}(x[1] / x[1])) + (\text{np.cos}(x[1]) / (\text{np.cos}(x[1]) + (\text{np.cos}(x[1]) + \text{np.cos}(x[1])))) + (x[1] / x[1])) + \text{np.cos}(x[1]))$

MSE (train): 7.14565

1.3.5. Problem_5

Found formula: 0

MSE (train): 5.57281e-16

1.3.6. Problem_6

Found formula: $(x[1] + (((x[1] - x[0]) * \text{np.sin}(1)) * \text{np.sin}(1)))$

MSE (train): 0.314885

1.3.7. Problem_7

Found formula: $(((((x[0] + ((x[0] + ((x[0] * x[0]) * x[1])) * \text{np.sin}((x[1] * x[1])))) * (x[1] + ((x[0] + ((x[1] + x[0]) + x[1])) * \text{np.sin}((x[0] * x[0])))) + (((x[1] + (\text{np.sin}((x[1] * (x[0] * x[0])) + \text{np.exp}(x[0] * x[1])) * x[1]) * x[0]) + (x[1] + (\text{abs}(x[1]) + \text{np.exp}(x[0] * x[1]))))$

MSE (train): 27404.7

1.3.8. Problem_8

Found formula: $(((((x[5] * \text{abs}(((x[5] + ((x[5] + x[5]) + x[5])) + x[5])) + x[5]) + x[5]) * \text{abs}(((x[5] + (x[5] + x[5])) + (x[5] + ((x[5] + ((x[5] + (x[5] + x[5]) + x[5])) / x[5]) + x[4])) + (((x[5] + ((x[5] + ((x[5] + (x[5] + ((x[2] + ((x[5] + x[5]) + x[5])) + x[5])) - (x[2] + x[5])) - ((x[5] + x[0]) + ((x[5] * \text{abs}(((x[5] + ((x[5] + x[5]) + x[5]) + x[5])) + x[5])) + x[5]) + (x[5] + x[5])) + x[5])) - (x[4] + (x[2] + x[5])) + x[5]))))$

MSE (train): 7.51788e+07

2. Labs

2.1. Lab 0

The lab was just creating a Github repository and writing a joke.

2.1.1. Link

https://github.com/mericuluca/CI2024_lab0

2.1.2. Reviews I did

2.1.2.1. Andreabioddo

https://github.com/andreabioddo/CI2024_lab0/issues/3

Such a good summary

2.1.2.2. Gabry323387


https://github.com/Gabry323387/CI2024_lab0/issues/2

Same here!!!

2.1.3. Reviews I received


2.1.3.1. UtkuKepir

https://github.com/mericuluca/CI2024_lab0/issues/2

Funny and smart 

2.1.3.2. Luca-bergamini

https://github.com/mericuluca/CI2024_lab0/issues/1

You made me laugh! 

2.2. Lab 1

The lab was solving set cover problem for different universe size, number of sets and density of the sets. I randomly produce a solution and changed the solution a little bit randomly if its exploring by the tweak function. I returned the new solution if its not in the tabu list and added to the tabu list to avoid having always the same solution. If the returned solutions cost is less than the best result, it becomes the best result and at the end the algorithm found the best solution.

2.2.1. Link

https://github.com/mericuluca/CI2024_lab1

2.2.2. Reviews I did

2.2.2.1. MelDashti

https://github.com/MelDashti/CI2024_lab1/issues/2

Nice solution, it is very good suitable for this problem. The tabu search buffer can be really computationally expensive. To improve the performance for the instances with many iterations, limiting the tabu size can be beneficial because it consumes time to search in large list. Also, limiting the tabu size will not decrease the performance a lot because it probably retry the recent solutions. Overall, it is a nice solution.

2.2.2.2. Ibrokhimzodab

https://github.com/Ibrokhimzodab/CI2024_lab1/issues/1

Code is missing. I can not give a review on it.

2.2.3. Reviews I received

2.2.3.1. LouiseT2

https://github.com/mericuluca/CI2024_lab1/issues/2

Hello, I am going to be reviewing your code.

First of all, I think you represented the problem well and choose a correct way to solve it. Hill climbing can get good results in a few iterations. But I do have some suggestions to improve your code:

1. In the tweak function, the new solution is really close to the one before which, combined with the hill climbing method can prevent you getting out of a local minimum. Especially when the set size is big. It may be good to increase the number of mutations between two proposed solutions.
2. Using Tabu is a good idea to avoid calculating two times the same score. However, checking the entire list each time can be time-consuming. It might be more efficient to limit the list to the most recently tried solutions, as they are likely closer to the current solution and more likely to match the new solutions after mutation.
3. It would have been a good idea to add legends to the plot to have a better understanding of the results.

4. Using a number of steps is a good idea in this case because you probably already tested your algorithm and observed at what time the cost function stopped going down. But to be more flexible to other situations (such as a change in the size of the universe, for example), it can be a good idea to implement an adaptive stopping criterion based on the behavior of the cost function, such as stopping when the relative improvement between iterations falls below a certain threshold. This way you would not have to change the max_steps variable for each instance.

2.2.3.2. Lorkenzo

https://github.com/mericuluca/CI2024_lab1/issues/1

Nice solution, it is very simple and can be suitable for optimizing easy problems, but could be a bit slow when solving problems with many data or many iterations. In fact keeping track of the already seen solutions with the tabu search buffer can be really computationally expensive. Maybe to improve the performance for the instances with many iterations you should try to insert a maximum dimension to the memory buffer instead of limiting the number of iterations. Another problem in the algorithm could be that a new solution is considered with an higher fitness than the actual solution only if it is both valid and with a lower cost. This strict condition can easily lead the algorithm to get stucked in a local maximum. Sometimes could be useful to allow the solution to go down in order to find a better solution, or allowing it to do bigger steps if the maximum is not improving for many iterations.

2.3. Lab 2

The lab was solving travelling salesman problem for different cities in different countries. I used the cycle crossover and inversion mutation. In the first method, the probability of mutation is 0.1 and the probability of crossover is 0.9. In the second method, the probability of mutation is 0.9 and the probability of crossover is 0.1.

2.3.1. Link

https://github.com/mericuluca/CI2024_lab2

2.3.2. Reviews I did

2.3.2.1. FabioGigante00

https://github.com/FabioGigante00/CI2024_lab2/issues/1

Hi, Your solution is great. However, I expect that genetic algorithm to perform better than greedy. So, maybe trying different mutation and crossover can improve the performance. Additionally, instead of trying 2 different probability for crossover and mutation, you can try different probabilities or adjusting it dynamically in response to lack of improvement might improve results over generations.

2.3.2.2. Blackhand01

https://github.com/Blackhand01/CI2024_lab2/issues/2

Hi, it is such a good solution and it was easy to understand what you did because you explained it very well. There are some things that came to my mind when I read your code. To enhance performance, you can consider using parallel processing and caching distance matrices to speed up calculations. Additionally, instead of sorting each tournament sample to find the best candidate, using a simple minimum function could enhance efficiency. Lastly, implementing an early stopping mechanism when the solution stabilizes over several generations can save computational time.

2.3.3. Reviews I received

2.3.3.1. TediBash

https://github.com/mericuluca/CI2024_lab2/issues/1

Hi Meric,

these are the thoughts that come to my mind when I look at your project:

- In the readme file you can include a lot of information that can help other people understand your project better and faster.
For example, it could include information about the problem, information about different strategies for solving the problem, and why you chose some and not others. It could include a table showing all the results obtained for different scenarios.
- If there is a dependency, it might be better to put it in the readme rather than in the notebook. es 'pip install icecream'
- You could add more comments, especially in the function definition.
Example. What are the input parameters, what does the function do and what does it return as output.
- It might be useful to create a separate block for the function definition, the global parameter and the code to be executed. This way, you can only re-run the "action" code.
- I can see that you are copying and pasting the same code for each country, instead of doing this it might be better to implement a loop in which each iteration compute different countries. In this way, if you need to change a part of your code, you only need to update one place.
- You could also have used the NetworkX library, thanks to the NetworkX library you could solve the TSP problem using Graphs.
- You could have implemented different types of crossovers, such as Inver-over, n-cut crossover, Partially_mapped_crossover, Cycle_crossover.

Keep it up!

2.3.3.2. Giorgio-Galasso

https://github.com/mericuluca/CI2024_lab2/issues/2

Hello Meric,

Let me start by saying that your code seems very straightforward and well-done.

At first, I was a bit intimidated by the length of the code, but after paying closer attention, I realized that you repeated the same code for all the countries in the dataset.

This is perfectly fine (I mean, it works), but if I may offer a suggestion, next time you could structure the code in a more streamlined way. For example, you could prepare lists to hold the various settings for each execution and then run the code in a single go.

From personal experience, it can be helpful to check out the Git repositories of others following the course; you can often find some very interesting tricks!

On that note, your code seems fairly standard and should work quite well. I know it might be a bit tedious to create (I skipped it too), but adding a README with a simple table would help interpret the results.

The HYPERMODERN approach with the crossover recommended by the professor seems like the best choice, so what can I say? Great work, and keep it up! 🙌

2.4. Lab 3

The lab was solving n_puzzle by using a search algorithm and I used A* search method. For heuristic function, I used Manhattan distance and wrote a function which is `calc_manhattan`. For path cost, the step number is used. Everything happens in a while loop. When the solution is equal to the goal, the loop ends. The algorithm starts with the initial state. The available actions for the initial state are found and the states after taking these actions are found. The costs are found for the next steps and added to the `states_list`. `states_list` is sorted by the costs and the next state is chosen by looking for the least cost. If the new state is not the goal, the available actions are found for this new state and the algorithm goes like this.

2.4.1. Link

https://github.com/mericuluca/CI2024_lab3

2.4.2. Reviews I did

2.4.2.1. YaelNaor11

https://github.com/YaelNaor11/CI2024_lab3/issues/2

Thanks for the code. Implementing A* algorithm to solve n-puzzle game task is a great choice because it is one of the most efficient searching algorithms. The solution is fine, however writing a readme file or writing some comments on your script could be a great idea to explain your code because it was hard for me to understand it. Using priority list, `heapq`, is such a great idea to make the code faster. I will use it in my next task. Well done!

2.4.2.2. fedspi00

https://github.com/fedspi00/CI2024_lab3/issues/2

Thanks for the code. You implemented A*, Breadth-First Search (BFS), and Iterative Deepening Depth-First Search (IDDFS) algorithms to solve n-puzzle game task which is a great way to compare the algorithms. The solutions seem fine, however 3x3 results are not enough to compare them. Firstly, readme file is a great idea. However, it would be much better if you also add some comments on your script because it was hard to understand for me. Well done!

2.4.3. Reviews I received

2.4.3.1. anilbayramgogebakan

https://github.com/mericuluca/CI2024_lab3/issues/2

Thanks for the code. It seems that you implemented an A* algorithm to solve n-puzzle game task which is a great choice because it is one of the most efficient searching algorithms. The solution seems fine, however there is still room for improvement because the code seems slower than it supposed to be.

Firstly, readme file is a great idea to explain your code. However, it would be much better if you also add some comments on your script. In this way, your work seems more organized and interpretable.

Secondly, sorting your list in every iteration is something very costly. Instead of this approach, you might consider using priority list such as `heapq` which I used in my implementation as well. I believe that it will make your code way more faster. Also, implementing your priority queue list object is another option.

Thirdly, I use hashing function in order to speed up the process. You can also try it.

Of course, they are small details to make your work better. However, it is already a great job. Well done!!!

2.4.3.2. Gdennis01

https://github.com/mericuluca/CI2024_lab3/issues/1

Your A* implementation seems fine however it's considerably slow. Here are some tips that may speed up significantly your code:

- Use a priority queue like heapq. I noticed that you relied on a simple list that is sorted at EVERY iteration. That overhead is not negligible at all and has an huge impact on the overall performance
- Use an hashed data structure like a set or a dictionary for a fast lookup. This should further speed up your code.
- For comparison, my code took about 21 seconds to run the provided example in your jupyter notebook, while yours took about 11 minutes.

These small changes can make a huge difference in performance!!

Other than that good job overall!

References

[1] M. Cranmer, "Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl," *arXiv.org*, May 02, 2023. <https://arxiv.org/abs/2305.01582>