Oguzhan Akgun – s328919

Labs:

Lab 1: github.com/oguzhaaan/CI2024_lab01

This lab was not evaluated because of the wrong spelling in the repository name. In this lab, I used **Hill Climbing** to solve the Set Cover Problem by improving the solution step by step. A common problem with Hill Climbing is that it can get stuck, repeating the same moves and wasting time. To fix this, I added **Tabu Search**. It keeps track of solutions already tried and stops the algorithm from selecting the same units again. This helped the algorithm explore new options and avoid getting stuck. By combining Hill Climbing with Tabu Search, the method worked better, especially on larger and harder datasets. This shows how adding simple rules can improve the performance of a basic algorithm.

Lab 2: github.com/oguzhaaan/CI2024_lab2

In this lab, the Evolutionary Algorithm was applied to solve the Travelling Salesman Problem (TSP). The goal was to find the shortest possible route that visits all locations in a dataset, such as in countries like Vanuatu, Italy, US, Russia, and China, and returns to the starting point. To achieve this, **Cycle Crossover** and **Swap Mutation** were implemented as key genetic operators.

Cycle Crossover combined two parent routes by exchanging cycles of elements, ensuring valid offspring that preserved traits from both parents. This helped the algorithm explore new potential routes effectively. Swap Mutation introduced small changes by swapping two cities in a route, refining the solutions and helping the algorithm avoid getting stuck in local optima. Different probabilities for mutation and crossover were tested to analyze their effect on finding optimal routes. Adjusting these probabilities allowed the algorithm to balance exploring new routes and improving current ones, making it effective across various dataset sizes.

Oguzhan Akgun – s328919

The review I got from GiuseppeEsposito98 for Lab 2:
github.com/LouiseT2/CI2024_lab2/issues/2

The code is well written and well commented, however, there are a couple suggestions that could help you to improve the correctness and the performance of the code.

Suggestion 1 (minor)

In the implementation of Swap Mutation you write random2=random1 and then you change the random2 variable in an if clause. I am not sure that, by doing so, random1 is not changed as well and I believe that in the Swap Mutation, you don't want random1 to be changed according to random2. For this reason, I would suggest to create a deepcopy of random1 when you assign the object to the variable random2.
Specifically:
from copy import deepcopy

random2=deepcopy(random1)
this will create another variable that points to an independent instance of random1.

Suggestion 2 (major)

As main suggestion I would say that your version of the Evolutionary Algorithm can be significantly improved in scalability, generalization capabilities and accuracy of the solution by making it adaptive. I have seen that you assign a probability for exploitation (mutation) and for exploration (crossover) but an adaptive probability can be set based, for example, on the distance that you have already walked when you reach a new point of the graph and then you have to decide the next step.

The review I made for LouiseT2 for Lab 2:

github.com/LouiseT2/CI2024_lab2/issues/2

Great work! I appreciate the effort put into enhancing the greedy algorithm with lookahead, even though it didn't improve the results. It might be worth experimenting with other heuristics, like nearest neighbor combined with randomized restarts, to see if it leads to better outcomes.

The simulated annealing approach is impressive, especially with the exponential cooling schedule. Perhaps exploring alternative cooling functions or adjusting the neighbor generation method, such as swapping multiple cities instead of just one, could yield even more precise results.

Overall, excellent implementation! A few additional tweaks could make these algorithms even more robust. Keep it up! 👍

Oguzhan Akgun – s328919

The review I made for graicidem for Lab2:

github.com/graicidem/CI2024_lab2/issues/3

Great solution! I really like your choice of inver-over and inversion mutation; they fit the problem well. Your code is also well-structured and easy to follow with clear comments.

Here are a couple of suggestions:

1- Dynamic Probabilities: Try changing the crossover and mutation probabilities as the generations progress. Starting with higher crossover and later focusing more on mutation could improve your results.
2- Population Size: Increasing the population size might help explore more diverse solutions, especially for larger datasets.
Overall, awesome work! With some small tweaks, this could perform even better. 👍

Lab 3: https://github.com/oguzhaaan/CI2024_lab3

In this lab, the goal was to solve the N-Puzzle problem using search algorithms and compare their performance on different size of N-Puzzles. For the 3x3 puzzle, **Breadth-First Search (BFS)** was used. It found a solution in 18 steps after evaluating 24,375 states. However, BFS was too slow for larger puzzles because it explores all possible states.

To handle the 4x4 puzzle, the **A*** algorithm was used **with Manhattan distance** as a heuristic and a **priority queue** for better state selection. A* found a solution in 34 steps after evaluating 101,043 states with 150 randomized steps. However, for larger configurations like 100,000 randomized steps, A* took too much time. This lab showed how important it is to choose the right algorithm and heuristic for solving larger problems efficiently.

Oguzhan Akgun – s328919

The review I got from andreazenotto for Lab 3:

github.com/oguzhaaan/CI2024_lab3/issues/2

This project takes a solid shot at solving the sliding puzzle problem, using BFS for 3x3 puzzles and A* for 4x4 puzzles. The algorithm choices make sense: BFS, while heavy on computation, works fine for the smaller grid, and A* with Manhattan distance shows a good grasp of optimization for tackling the tougher 4x4 version. The priority queue in A* is a nice touch, keeping things efficient and in line with best practices.

The code is well-structured, with clear, modular functions that make it easy to follow. Comments are concise but helpful, explaining the logic without overloading the reader. The design feels flexible enough to try out new heuristics or even scale to bigger puzzles, which is a plus.

The README does the job but could use more depth. It'd be great to see more context about why these algorithms were chosen and how they compare. Visuals or examples—like step-by-step solutions or before-and-after states—would make the project way more engaging and easier to understand.

Overall, this is a strong project that shows a solid understanding of the problem and the algorithms involved. Great work!

The review I got from Daviruss1969 for Lab 3:

github.com/oguzhaaan/CI2024_lab3/issues/1

General ⚙️

First of all, I liked your implementation of both the BFS and A* algorithm, they were pretty clear, so easy to review 💯.

You also manage to get good result in terms of quality for both algorithms 👍.

But I still have some recommendations to do, feel free to go ahead and implement them if you like them 🚀.

Implementation of a greedy algorithm 🤑

As I say in the introduction part of my review, you are good in terms of quality but your cost isn't that good, and for bigger N-puzzle you don't have any results.

Oguzhan Akgun – s328919

Implementing a greedy algorithm can solve this issue, it will decrease a lot the quality (increase the value) but will also decrease the cost of your algorithm.

A simple algorithm to implement looking at your solution is Greedy Best Fit (GBF), since you can take your A* algorithm and just update the new_f computation. Indeed GBF doesn't take the current cost in the computation of f(n).

Optimisation of your Manhattan distance function 📈

The performance of the Manhattan distance function is crucial for the A* algorithm. Indeed, it will be called many many times.

This line goal_pos = np.argwhere(goal == state[r, c])[0] is called N2 times for each call and can be a bottleneck of your algorithm.

The review I made for Giorgio-Galasso for Lab 3:

github.com/Giorgio-Galasso/CI2024_lab3/issues/2

This code offers a solid solution to the N-Puzzle problem, employing the A* search algorithm for puzzle-solving. The heuristic used combines two key factors: the Manhattan distance and the number of misplaced tiles. This combination is then modified with an exponential factor to make the heuristic more aggressive, providing a more refined estimate of the cost to reach the goal. While the inclusion of both heuristics is a good strategy, further experimentation with different weighting or balance between them could potentially improve the algorithm's efficiency and performance.

The heapq module is utilized to implement a priority queue for managing the states during the search process. This choice ensures that the states with the lowest estimated cost are explored first, optimizing the search. However, the fixed MAX_STEP limit may be too restrictive, particularly for more complex puzzles, and a dynamic step-limiting mechanism could better handle a variety of scenarios, allowing the algorithm to run longer for more difficult puzzles without sacrificing performance.

The code is well-structured and modular, with each function serving a clear and specific purpose. For example, the find_zero function isolates the position of the empty tile, while get_neighbors generates all valid states from the current state by swapping the empty tile. The reconstruct_path function elegantly tracks the solution path by backtracking through the states. This modular approach makes the code easier to maintain and extend, as well as more understandable for anyone looking to learn from or modify the implementation. Great work! 😊 👍

The review I made for FruttoCheap for Lab 3:

github.com/FruttoCheap/CI2024_lab3/issues/2

The code is well-structured, implementing puzzle-solving algorithms such as BFS, DFS, IDDFS, UCS, and A* with clear separation of logic for each method. The manhattan_distance heuristic in A* is appropriately chosen for guiding the search. However, there are areas for improvement: more inline comments could make the implementations easier to follow, the depth limit in IDDFS should be configurable, and some redundancy exists, such as repeating the initial state in tests, which could be streamlined. The solution path output format should be better documented, and the test cases could cover a broader range of scenarios to ensure thorough validation. Additionally, BFS and DFS could be optimized for faster state exploration. Good job 👍

Oguzhan Akgun – s328919

Final Project

In this project, our primary objective is to develop and apply symbolic regression methods to solve a given dataset while achieving the lowest Mean Squared Error (MSE) score. By minimizing the MSE, we ensure that the discovered equations not only capture the relationships within the data but also provide reliable predictive performance. To enhance the search efficiency and maintain a balance between exploration and exploitation, we incorporate techniques such as **migration**, **aging**, and the **killing of eldest individuals**, inspired by the approaches detailed in Cranmer et al.'s work on symbolic regression algorithms (https://arxiv.org/pdf/2305.01582). These methods allow us to prevent premature convergence, maintain diversity in the population, and systematically refine the solutions by leveraging evolutionary principles. I was involved in all the simplification processes of the project, focusing on reducing complexity and discovering better formulas to improve the results.

To begin with, we defined node class to represent our tree structure. Each node can be constant, operator that is listed in op_list (+,-,*,sin,cos,exp,abs,/,log,tan) or one of the features of x. If constant, value will be equal to that number. If operator, value will come from np.{operator}. If feature of x, we know that it is a feature, value will be None, and the index of feature will be stored in feature_index. There are left and right values to represent child nodes. If there is only one child node, only left value exists. We also make complexity calculation for each node. Nodes create individuals. Individuals have also train and validation cost as fitness and fitness_val which is equal to cost of that formula, age value which is initially set to 0 and T value for simulated annealing to be used in mutation process.

In the evolve function, after randomly creating a population with the size of 100 individuals and calculating their fitness values, we used tournament selection to select the ones who will produce new individuals by organizing tournaments for 3 random individuals and eliminating individuals whose costs are the largest. In our code we defined elites' number as 3 not to eliminate them during killing eldest process. We also implemented aging methodology to avoid local minimum, increase diversity and exploration. In generation creating process loop we used mutation and crossover methods with probability of %80 for each individual and increased age by 1 for each cycle.

Oguzhan Akgun – s328919

We tried to simplify constants of genome in individuals for example instead of (2+3) directly will be shown as 5. Initially, we observed that the dataset had a low likelihood of being constant. However, we lost variation as the population started producing too many constants. To address this issue, we implemented a **killing constants** strategy, where individuals that are purely constants are directly removed from the population. This helped maintain the balance between exploration and exploitation within the algorithm and reducing the complexity. As part of the evolution process, we implemented a **simplification operation** to reduce unnecessary complexity in the mathematical expressions generated by individuals. Specifically, if numerical constants are used as arguments in mathematical functions like sin, cos, tan, or abs, we directly calculate their results. For example, instead of representing sin(π/2) or abs(-5), the algorithm simplifies them to their respective constant values 1 and 5. This optimization ensures that the individuals in the population maintain concise and computationally efficient expressions, preventing redundant calculations during fitness evaluation.

The results we generated were often quite lengthy, so we added functions to simplify the root expressions. For instance, an equation like log(x[0]) + log(x[0]) could be simplified to 2*log(x[0]). However, this simplification didn't improve performance. In some cases, keeping the expressions unsimplified actually worked better for mutation and crossover, as it introduced more diversity for the algorithm to explore.

To enhance diversity within the population and avoid redundant evaluations, we implemented a **deduplication mechanism**. This process identifies individuals with identical genomes and removes duplicates, ensuring that only one individual with a unique genome remains in the population. By eliminating identical solutions, we reduce computational overhead and maintain a more diverse set of candidates, allowing the algorithm to focus on exploring new areas of the solution space.

To ensure computational efficiency and convergence, we implemented several mechanisms in the evolutionary process. When the dataset size doubles, we continue with the best 100 individuals to balance computational cost and maintain a diverse yet manageable population. Additionally, the evolution process halts if the cost of any individual becomes extremely small, such as below a threshold of 0.0001. This stopping criterion ensures that the algorithm terminates when an optimal or near-optimal solution is found, avoiding unnecessary computation in subsequent generations. By focusing on the top-performing individuals and introducing this convergence condition, we streamline the search process while maintaining high accuracy.

Oguzhan Akgun – s328919

To enhance diversity and improve the optimization process, we utilized a multi-population strategy with 4 separate populations. Each population evolves independently using the methods described earlier, including selection, mutation, crossover, aging, and deduplication. At specific intervals, **migration** is performed within each population, where individuals are exchanged to share genetic information and prevent stagnation. By incorporating this multi-population approach with migration, we ensure a balance between exploration and exploitation, allowing the algorithm to avoid local minima and find the most accurate and interpretable solution for the dataset. After migration, each population continues to evolve further. This cycle of **evolve → migration → evolve** repeats until the stopping criteria are met. Once the evolution process concludes, the **best individual** from each of the 4 populations is selected. Finally, the overall best individual among these is chosen as the optimal solution for the problem.

Github Repository:

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

Collaborators:

Anil Bayram Gogebakan – s328470

Meric Ulucay – s328899