Report of Laboratories and Project Work for Computational Intelligence

Lorenzo Lombardi (s330654)

In collaboration with:
Giorgio Rondinone (s332156)
Simone Scalora (s329444)

10 febbraio 2025

Indice

1 Laboratory 1: Set Cover problem

1.1 Problem explanation

The problem we want to solve for this lab is the Set Cover problem. This is NP-hard problem that basically involves finding the minimum number of sets that cover all elements in a given universe. To tackle this problem, here's the approach used in our case.

1.2 Our solution: Custom Steepest Step

We found a modified and improved version of the steepest step to work best for this scenario. Please note that we are always randomizing the starting state. This algorithm uses a fitness function that exploits rarity of each item in order to prioritize rarer items. We also implement a steepest step with restarts based on the size of the parameters. This allows us to get a better solution by trying multiple times for smaller problems but in case of large problems (100000 universe size) we do not implement the restart mechanism as the size of this problem makes it a very time consuming to execute. There's also a cap for the number of steps without any improvement, so that we can compute the iteration it took to find a (supposedly) optimal solution. In our case, these parameters adapt themselves based on the universe size (our threshold is 50000). If the universe is bigger than the threshold, we remove the restart feature (NUM_RESTARTS=1) to avoid very long execution times. In case the universe is smaller than our threshold, we have a dynamic stagnation threshold defined as MAX_NO_IMPROVEMENT = (UNIVERSE_SIZE / 10) + 20 that tells us basically when we have to stop when we don't see any improvement. Our tweak function is a multiple mutation that randomly mutates values.

1.3 Main Algorithm Explained

- **Initialization:** Define variales for the best cost found, number of steps, history of fitness, best solution and the tweak function.
- **Restart loop:** For each restart we generate a random initial solution and we evaluate its fitness (and store it), then initialize variables to track last improvement and to track how many steps have passed without improvement.
- **Steepest Ascent Hill Climbing:**
    1. Generate candidate solutions (by applying the tweak function and storing candidates' scores)
    2. Select the best candidate and update new_solution and new_fitness
    3. Check for improvement, if we have improvement update solution and reset no_improvement_count. If we don't have improvement, increment no_improvement_count. If no_improvement_count exceeds the no improvement threshold, terminate early.

4. **Best Solution Selection:** After all restarts compute the cost of the current solution, if it is better than the previous best, update best$_solution$

The results are as follows for the given instances

Instance	**Cost**	**Calls**
Instance 1	681	281
Instance 2	5753	6.253
Instance 3	56.361	122.119
Instance 4	111.576	1.942.873
Instance 5	120.064	2.243.775
Instance 6	122.647	2.338.492

Tabella 1: Cost and Calls for Each Instance

1.4 Reviews i gave to other colleagues

Here's the review i gave to **UtkuKepir**'s code: Good solution, the multiple mutation performs well on this problem, and if you want i think you can try to do some small adjustments to make it perform even better. For example, you can try different starting solutions (instead of starting with all the sets selected) or fine tuning the multiple mutation function (i'm talking about the parameter 0.01) depending on the instance. In my case i also adopted a way to count how many items are covered (this helps if you select fewer sets as a starting solution) and i found that to be quite beneficial to the solution

Here's the review i gave to **Roberto34**'s code: Very good implementation, i really like how you start just from the mandatory sets and then build from there. I think that maybe you could also try to experiment other techniques like simulated annealing and see how that works out, but in my case the best solution i found was something similar as yours (i used the rarity of each item in order to weigh the sets). Maybe also randomizing the initial solution (keeping all the mandatory sets of course) could also be beneficial. I think you did a great job with this solution!

1.5 Reviews that my work received

mickp18 said: Personally, I found this solution very clever and well thought. I wasn't able to come up with a solution that used this approach, so it was very useful for me to see it and try to understand what I could have done better. Maybe something you could add to make it even more efficient is the use of a simulated annealing approach to avoid getting trapped in local minima. To be honest, it's just an idea, and I could be wrong, but since it's the technique I used, maybe it could help. Let me know if you try it, and congrats on the great work!

AlesoGio submitted this issue: You found a great solution, well done. I really liked how you combined different techniques in order to optimize the algorithm. A personal suggestion for large universe sizes, since restart takes too much time, maybe some sort of adaptive restart could be worth a try. Instead of waiting for all steps to complete, the algorithm restarts if there is no significant improvement after a certain threshold. But obviously you would need to identify a good threshold, that would make the algorithm more complex. Overall i really liked you solution, good job!

2 Laboratory 2: the Traveling Salesman Problem problem

2.1 Problem explanation

This laboratory was about trying to find solutions for the Traveling Salesman Problem. This is a classic optimization problem, asking: "Given a set of cities and the distances between them, what is the shortest possible route that visits each city exactly once and returns to the starting point?" Our input is indeed a set of cities and we are trying to output the shortest path possible visiting all the cities exactly once and returning to the start. This problem is considered NP-hard, meaning no known polynomial-time algorithm can solve all instances efficiently.

2.2 Our approaches and solutions

For this lab we provided two different solutions: one providing a faster solution that gave an answer in a short amount of time, but for larger datasets might not always return the optimal solution, and a slower but more accurate one that is able to provide a good solution given enough time.

2.2.1 Fast Solution: Greedy Algorithm

Our fast solution uses a greedy heuristic approach. We are iterating the algorithm 10 times, each time with different cities (given the greedy approach starting point matters!) logging each iteration the tour's cost and path length. Once the iterations are over we simply choose the best tour that we have.

2.2.2 Slow but more accurate solution: Optimized Inner-Over

Our slower but more accurate solution implements an Optimized Inner-Over algorithm using a genetic apporach with local improvements. First, we generate an initial population of candidate tours. We generate half the population randomly, and the other half with a greedy heuristic for better starting solutions. We then apply fast local improvement to refine these initial solutions, and finally we sort the population based on tour cost (distance) and we select the best ones. Then, we introduce the main evolution step. Each parent tour undergoes an Inner-Over mutation. The mutation selects a random city in the tour, uses another tour to determine a new segment to invert, occasionally applies smart mutation to shuffle small segments and applies fast local improvement with a 1/10 chanche. The algorithm then keeps the best tours from both the parents and offspring. We also have a parameter MAX_NO_IMPROVE which is a threshold that tells us how many generations without improvement we can allow before stopping the algorithm. If no improvement is seen in MAX_NO_IMPROVE generations, the algorithm stops early. Our final output is going to be the best tour, the total distance, the number of generations used and the total steps taken. Our results are as follows for the various datasets:

Country	Greedy Distance	Greedy Steps	Optimized Distance	Optimized Steps
Italy	4436.03	46	4174.94	58450
China	62480.50	726	61644.75	578200
Russia	41771.58	167	36066.52	122850
US	47873.51	326	42656.83	224700
Vanuatu	1475.53	8	1345.54	52500

2.3 Reviews i gave to other colleagues

Here's my review for **luca-bergamini**'s work: Really like the solution you came up with! Simulated Annealing seems to work great for this problem. I aappreciate that you have explained your solutions and results in the readme, makes everything easier to understand. I think you can try to improve the efficiency of your solution by dynamically adjusting the cooling based on the algorithm's progress. Also maybe you can try limiting the number of iterations that don't provide an improvement, so that after n number of iterations you don't "waste" any more steps since you have (supposedly) found an acceptable solution. Kudos to you for the great job!
Here's my take on **graicidem**'s code: Very good solution! I also used inver-over for my solution, so i also think that this method performs well on this problem. Really like how the code is well thought out, with a function for each algorithm, and the comments are really useful in understanding your code. I think you can benefit by tweaking the hyperparams, such as the population size (for exploration) and the number of generations. Sure this will require more steps but it could also lead to a better result in bigger datasets such as china. Dynamically changing the crossover as the algorithm runs can also help achieving a better solution. Kudos for the great code!

2.4 Reviews that my work received

SamueleVanini said this about my code: The proposed algorithm shows some improvements over the greedy solution presented in class. The use of "smart" individuals in the initial population is well thought out and should provide ample genetic diversity, especially given the large population size. Here are a few suggestions that could further enhance the quality of both the code and the repository: Dependency Management: To make it easier for others to run your code, consider using a tool like Poetry or a simpler alternative, such as a requirements.txt file, for managing dependencies. This would streamline the setup process and ensure the correct versions of libraries are used. Break Down the Code: Currently, the algorithm is contained in a single large block with several nested functions. Although inner functions are sometimes used to limit scope, here the nested structure reduces readability due to indentation depth and length. Consider breaking down the code into separate functions for improved clarity. Avoid Duplicate Parent Selection: Sampling two parents with replacement can result in selecting the same individual twice, which creates clones rather than introducing new genetic material. This may be counterproductive, especially given the generational population management approach used. It might be worth adjusting the sampling to avoid this. Include the Dataset: Since the dataset is relatively small, it could be convenient to include it directly in the repository. This would simplify testing for other users. Trait Retention in Optimization: Applying pseudo-deterministic optimization on offspring may disrupt the inherited traits from parents, potentially reducing genetic diversity. Consider reviewing this approach, as it could impact the effectiveness of the algorithm over generations.

AnjaliVaghjiani submitted this review: I really liked that you used a lot of comments to explain the steps with the code. It makes the logic much easier to follow and understand. While reading your code only error I found was that there is a duplication of tsp_cost(). Try to avoid using duplicate names to avoid confusion and redundancy. The choice of the inver-over algorithm is a strong design decision for solving TSP problems, as it offers a unique way of exploring potential solutions. You've implemented the key steps well, but one suggestion I have is to consider adding a dynamic adjustment to the mutation probability. This could enhance adaptability as the algorithm progresses. For instance, you might gradually decrease the mutation probability over time, allowing for more exploration in the early stages and finer adjustments as the algorithm narrows in on optimal solutions. Overall, I think you did an excellent job with the code! It's well-structured and clearly written, making it easy to understand. Keep up the great work!

3 Laboratory 3: N-puzzle problem

3.1 Problem explanation

The problem that we want to solve is simply to find the best solution to solve an NxN puzzle in as little moves as possible. We start from a random position and the puzzles may vary in dimension. The algortithm that we chose to use is the A* algorithm to find a solution. First, we initialize the problem for a given dimension by creating a random state.

3.2 Functions used

We have various functions to help us work with the puzzle. To mention a few:

- **available_actions(state):** Identifies possible moves based on the empty (0) position and returns a list of possible moves.
- **do_action(state, action):** Swaps two "tiles" to create a new state. Every time we move a tile, we see it as a new state.
- **get_goal_position(value):** This is used to check the goal position for a given number.
- **manhattan_distance(state):** We use the Manhattan distance heuristic as a way to measure distance in our puzzle. Computes the sum of the tile distances from their goal positions (using get_goal_position).
- **count_inversions(state):** We check how many tile pairs are out of order in our puzzle. This is used for the solvability check.
- **is_solvable(state):** Using count_inversions, we determine if the puzzle has a solution. For odd-sized grids, it must have an even number of inversions. For even-sized grids, $(inversions + blank\_row\_from\_bottom)2$ must match $blank\_row\_from\_bottom2$.

Note: to improve the heuristic we use two functions that calculate the linear conflicts and a way to give penalties for conflicts.

3.3 How the algorithm works

We then apply the algorithm as follows:

- Use a priority queue to store states sorted by cost
- Keep track of visited states to avoid loops
- We expand the nodes (possibile moves) and push them into the queue
- We guide the search by using linear_conflict_distance heuristic
- Stop when the goal state is reached.

To randomize the puzzle we start from a solved puzzle and we make 10,000 valid moves, and we ensure that it is solvable. The output of our code has

- **Quality:** Number of moves in the solution
- **Cost:** Number of nodes expanded
- **Efficiency:** Quality / Cost

Here are our results:

Grid Size	Quality	Cost
3x3	16	128
4x4	46	470300

Tabella 2: Quality and Cost for Different Grid Sizes

3.4 Reviews i gave to other colleagues

Here's the review i gave to **LucaIannello**'s work: The solution is very smart, i really like how you combined the heuristics in order to achieve a good formula for the priority. The only thing that i can suggest is to dynamically adapt BEAM$_SIZEbasedonthecomplexityofaproblem, insteadofhavingithardcoded.Otherthanthatithinkthatthisis$

Here's my take on **parmigggiana**'s code: This is outstanding work! I really appreciate the detailed readme which gives an overview of the code and also a mini tutorial on how to test its performance. The benchmark comes very handy for, well, benchmarking the performance, and makes the job of evaluating the solution much easier. I think also the improved$_manhattanisbrilliantasanheuristic, butithinkthemostimpressiveparth$

3.5 Reviews that my work received

This is **Andrea-1704**'s take on my code: I appreciated your implementation of the is_solvable function to verify the feasibility of the problem starting from the initial state. This is a well-thought-out addition that strengthens the robustness of your solution. I do have a stylistic suggestion: consider adding more detailed comments within the code and providing additional information in the README file. This would enhance the clarity and usability of your project for others. Your algorithm performs admirably and successfully determines the optimal solution for the problem. However, it struggles to solve puzzles with larger dimensions within a reasonable timeframe. I experimented with your solution by incorporating a sum of different heuristic metrics (including simply combining the ones you defined). With this adjustment, the algorithm was able to solve a 5x5 puzzle in a limited amount of time. You could also explore balancing execution time and solution optimality by adjusting the degree of overestimation in your heuristic. Overall, great work!

Gabriele-Raffaele said this about my work: This implementation of the A* algorithm for the N-Puzzle effectively combines Manhattan Distance with Linear Conflict to refine heuristic accuracy, while also checking puzzle solvability to handle random initial states. The code is clear, well-organized, and provides meaningful metrics like cost and efficiency. However, it would have been useful to test the code with smaller puzzle sizes, such as N=3, to better evaluate performance scalability and heuristic behavior on simpler configurations. Additionally, including results in the README along with a more detailed explanation of the code would significantly improve usability and clarity for other developers. Overall, a commendable effort!

4 Project work: Symbolic regression

This work was done by Lorenzo Lombardi (s330654), Giorgio Rondinone (s332156) and Simone Scalora (s329444)

4.1 Project Overview

The main focus of this project is to find the mathematical expressions that better predicts the given datasets. The provided code has been developed in order to perform symbolic regression, searching for mathematical expressions (in form of trees) to model a given dataset in order to minimize the Mean Squared Error, and thus achieving a better result. In order to do that we chose to implement Genetic Programming, that takes a population (of random binary trees in our case) and tries to iteratively improve it.

4.1.1 Expression Trees

The core data structures that we are going to use are binary trees. Each node represents either a mathematical operator (+,-,*...), a constant, or a variable (e.g. x[0]).
Keep in mind that we are using a "safe" version for many operators that clips values in order to mitigate warnings. We are recursively evaluating the tree to compute an output.

4.1.2 Population Initialization

Before we start generating the trees, we check the MSE of each operator alone, and we keep the values found for fitness evaluation (This is useful for problem 1 where the given dataset can be matched with a sin(x)).
We are generating a population of random trees, with each tree having a maximum depth to control complexity, and an early termination probability in order to have trees of different depths.

4.1.3 Fitness Evaluation

Each tree's fitness is calculated as the Mean Squared Error between its predictions and the target values. We are sorting the population by fitness, with lower MSE values indicating a better performing solution.

4.1.4 Genetic Operators

We are using a variety of genetic operators in order to implement GP:

- **Tournament Selection:** We use tournament selection in order to choose parents for reproduction. Trees with bettere fitness have a higher chance of being selected.
- **Dynamic Crossover:** We select two parents trees exchange subtrees in order to produce offspring. We chose to implement a dynamic crossover meaning that we will start with the more aggressive and explorative two-point crossover, based on the number of generations
- **Mutation:** We are introducing random changes to trees in order to encourage exploration.
- **Pruning:** We prune trees in order to keep complexity under control and limit redundancy.

4.1.5 Evolution

The randomly generated population evolves over multiple generations (MAX_GENERATIONS). We introduce elitism so that we keep the best performing trees in each generation in order to maintain the top solutions.
As we already said the algorithm adapts mutation rate and crossover strategy based on the number of generations that have passed until that point to achieve better overall performance

4.2 Project Solutions

Keep in mind we might change parameters based on the dataset in order to achieve better performance.

Config 1	**Config 2**
POPULATION SIZE = 100 MUTATION RATE = 0.3 MAX GENERATIONS = 100 TOURNAMENT SIZE = 10 MAX DEPTH = 8	POPULATION SIZE = 1000 MUTATION RATE = 0.3 MAX GENERATIONS = 10000 TOURNAMENT SIZE = 10 MAX DEPTH = 8

These two configurations are the ones that we found to work best considering time and performance. The first configuration is used for smaller datasets, while the second one is usually used for bigger datasets that are more difficult to approximate.

4.2.1 Problem 1

The dataset has shape **(1, 500)**.
Using **Config 1**
The formula that our algorithm managed to find is:
sin(x)
With an MSE of: **7.13e-34 (around 0)**

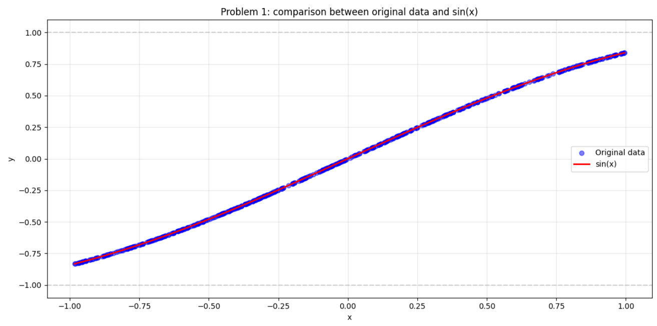

Figura 1: Problem 1

4.2.2 Problem 2

The dataset has shape **(3,5000)**.
Using **Config 2**
The formula that our algorithm managed to find is:
((((sqrt((((x[0] + x[2]) + (x[0] + (x[1] + x[0]))) + ((x[0] + (x[1] + x[0])) + x[2]))) + exp(((sqrt((x[0] + ((x[2] + log(x[2])) + x[2]))) +

x[2]) + (x[2] + log((x[2] + log(x[2]))))))) + sqrt((x[0] + (x[2] + (x[1] + x[0]))))) + sqrt(((x[0] + x[0]) + (x[1] + x[2])))) + sqrt(((x[0] + (x[2] + (x[1] + x[0]))) + sin(x[2]))))
With an MSE of: **1.93e+13**

4.2.3 Problem 3

The dataset has shape **(3,5000)**.
Using **Config 2**
The formula that our algorithm managed to find is:
(((((abs(exp(cos((abs(abs(abs(exp(sin(sin(sin(sin(sqrt(sqrt(x[1])))))))))) - x[1])))) - x[1]) - x[1]) + sin(sin(sqrt(x[1])))) + cos((abs(abs(abs(abs(exp(sin(sqrt(x[1]))))))) - x[1]))) + cos(cos(((abs(sqrt(sqrt(x[1]))) - cos((sqrt(x[1]) - x[1]))) - cos((sqrt(x[1]) - x[1])))))) + cos((abs(abs(exp(sin(sqrt(x[1]))))) - x[1]))) * 8.386974733167488)
With an MSE of: **330.27**

4.2.4 Problem 4

The dataset has shape **(2, 5000)**.
Using **Config 1**
The formula that our algorithm managed to find is:
(sin((cos(x[1]) + sin((cos(x[1]) + (cos(x[1]) + cos((cos(x[1]) + cos(sin((cos(x[1]) + (cos(x[1]) + cos(x[1]))))))))))))) + (cos(x[1]) + exp(cos(x[1])))) + (sin((cos(x[1]) + cos(sin((cos(x[1]) + (cos(x[1]) + cos((cos(x[1]) + exp(cos(x[1])))))))))) + ((cos(x[1]) + exp(cos(x[1]))) + cos(x[1]))))
With an MSE of: **0.0744**

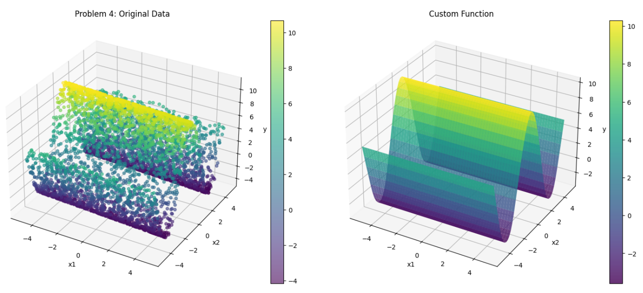

Figura 2: Problem 4

4.2.5 Problem 5

The dataset has shape **(2, 5000)**.
Using **Config 1**
The formula that our algorithm managed to find is:
sin((-6.802595075418944 / exp((abs(exp(4.165482260553965)) / log(3.7495617084686668)))))

With an MSE of: **5.57e-18**

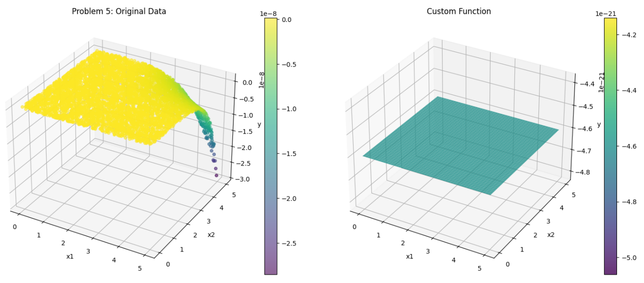

Figura 3: Problem 5

4.2.6 Problem 6

The dataset has shape **(2, 5000)**.
Using **Config 1**
The formula that our algorithm managed to find is:
(log(exp(x[1])) + (((log(exp(x[1])) + (((x[1] + x[1]) - ((x[1] + x[1]) - x[0])) - x[0])) + ((x[1] - ((x[1] + x[1]) - ((x[1] + x[1]) - x[0]))) / -3.121302705566653)) - x[0]))
With an MSE of: **0.0038**

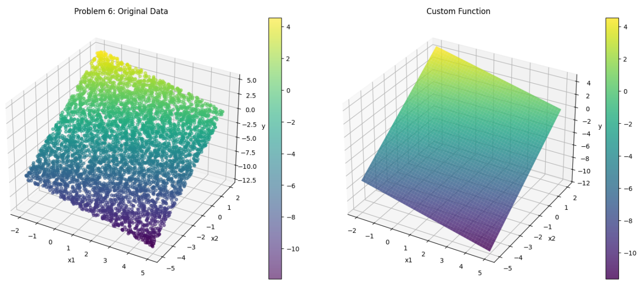

Figura 4: Problem 6

4.2.7 Problem 7

The dataset has shape **(2, 5000)**.
Using **Config 2**
The formula that our algorithm managed to find is:
(sqrt(exp(exp(cos(-6.09018062873057)))) - (x[0] * ((x[1] * cos((((x[0] - x[1]) * ((x[1] + x[1]) * cos((((x[0] - x[1]) * x[1]) * x[0])))) * x[0]))) * ((((x[0] + x[1]) * cos(((x[0] - x[1]) * x[0]))) * cos(cos(cos((cos(-6.09018062873057) - -6.09018062873057))))) * ((((x[0] + x[1]) * cos(((x[0] - x[1]) * x[1]))) * cos((((x[0] - x[1]) * x[1]) * x[0]))) * -7.541497887204642)))))

With an MSE of: **110.64**

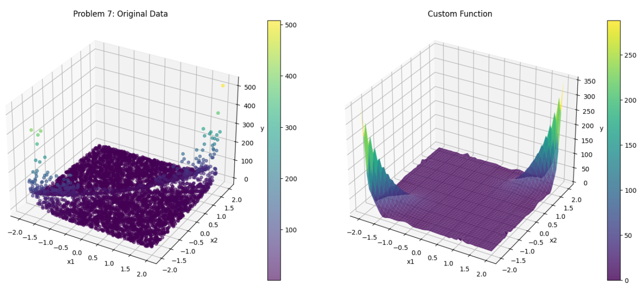

Figura 5: Problem 7

4.2.8 Problem 8

The dataset has shape **(6, 50000)**.
Using **Config 2**
The formula that our algorithm managed to find is:
(exp(abs(x[5])) * x[5]) * abs((4.6361949565668255 * 4.6361949565668255))
With an MSE of: **794137.02**