Final Report

Computational Intelligence

Author:
Tagliaferri Giovanni
s324286

Date:
February 10, 2025

GitHub:
`https://github.com/GiovanniTagliaferri/CI2024_project-work.git`

Contents

0 Intro

In this paper, I present all the laboratories and the final project conducted during the semester. For each laboratory I provide an introduction, its purpose, the solution I developed, the reviews I received and the ones I submitted for colleagues who were assigned to me.

As an example, I describe here the first assignment, Laboratory 0, which served as a preliminary exercise to familiarize ourselves with the workflow we would follow throughout the semester. The first task was to create a GitHub repository named `CI2024\_lab0`, containing a single file called `irony.md`, where we had to write a short pun. The second task involved reviewing the repositories of two colleagues by submitting an issue in which we rated their puns.

My solution was:

```
I have nothing fun to say.
```

Here are the reviews I received on my solution:

1. "HAHAHA !! HILARIOUS !"

Below are the reviews I submitted for my colleagues' puns:

1. "A simple but funny joke built on a pun that makes you laugh."
2. "I'm not sure if I got it, but if the meaning is that if you need to know something, you can literally 'hashtag' it out (like on social media), then I agree with it!"

After describing all the laboratories, the last section 4 is dedicated to the final project, which aimed to develop an algorithm for solving the symbolic regression task. In this chapter, the algorithm is explained in detail (the code can be found at the linked GitHub repository), followed by the results of eight benchmark problems.

1 Laboratory 1: Set Cover

Laboratory 1 was about solving the Set Cover problem that consists in finding a low-cost selection of sets that covers all elements in a given universe. We were provided with different problem instances and, for each one, we had to solve the problem by writing an algorithm, where each instance contained the `UNIVERSE_SIZE` (total number of elements), `NUM_SETS` (number of available sets that can be selected), and `DENSITY` (probability that an element belongs to a given set).

1.1 Solution

My approach was to use the Tabu search algorithm, implemented by the function `run_tabu_search(instance)` (1), that, before running the algorithm, initializes:

- A random boolean matrix `SETS`, where each row represents a set and each column an element. If an element is missing from all sets, it is randomly assigned to one.
- Set costs `COSTS`, computed as the sum of elements in each set, raised to the power of 1.1 to penalize larger sets.
- A tabu list `deque(maxlen=50)`, which stores recently visited solutions to prevent cycling.
- A fully selected initial solution, meaning all sets are initially included.

The proper algorithm then iteratively refines the solution over `1000` steps, generating and evaluating neighboring solutions by flipping random bits. If a neighbor is not in the tabu list, it is considered for selection. The best valid neighbor is chosen, and if it improves upon the current best solution, it updates the optimal solution found.

To handle larger instances efficiently, I also implemented a variation of the previous tabu search algorithm in `run_tabu_search(instance)` (2), which introduces a counter to stop early if no improvement is found after 100 steps, reducing computation time.

Finally, the algorithm outputs the best solution found, the number of steps taken, then plot of the cost history to visualize the optimization process.

Code 1: tabu_search

```
def run_tabu_search(instance):
    UNIVERSE_SIZE, NUM_SETS, DENSITY = instance
    print(f"UNIVERSE_SIZE: {UNIVERSE_SIZE}, NUM_SETS: {NUM_SETS}, DENSITY
        ↪ : {DENSITY}")

    # total number of steps to run the algorithm
    totSteps = 1000

    # define the sets
    SETS = np.random.binomial(1, DENSITY, size=(NUM_SETS, UNIVERSE_SIZE))
        ↪ .astype(bool)

    for s in range(UNIVERSE_SIZE):
        if not np.any(SETS[:, s]):
```

```
        SETS[np.random.randint(NUM_SETS), s] = True

# compute the costs by summing the number of elements in each set
    ↪ and raising to the power of 1.1
COSTS = np.power(SETS.sum(axis=1), 1.1) # or np.power

# initialize the random number generator
rng = np.random.Generator(np.random.PCG64([UNIVERSE_SIZE, NUM_SETS,
    ↪ int(10_000 * DENSITY)]))

### TABU SEARCH ###
# define the tabu list as a double-ended queue with a maximum size
    ↪ in order to keep automatically only
# the last tabu_size solutions
tabu_list = deque(maxlen=50)

# initialize a random solution given a percentage of being true
initial_solution = np.full(NUM_SETS, True)

# compute the fitness of the initial solution
current_solution = initial_solution
current_fitness = fitness(current_solution, SETS, COSTS)

# initialize the best solution and fitness
best_solution = np.copy(current_solution)
best_fitness = current_fitness

# number of iterations to find the best solution
best_steps = 0

# initialize the history of the costs
history = [current_fitness[1]]

print(f"Initial solution fitness: {current_fitness[1]}")

# run the algorithm totSteps times
for steps in range(totSteps):
    neighborhood = [] # list of neighbor solutions

    for _ in range(20): # generate 20 neighbors
        neighbor = tweak(current_solution, rng, NUM_SETS) # generate a
            ↪  neighbor solution by tweaking the current one
        if not any(np.array_equal(neighbor, sol) for sol in tabu_list)
            ↪ : # check if the neighbor is NOT in the tabu list
            neighborhood.append(neighbor)

    # if there is no neighbor
    if not neighborhood:
```

```
            continue

        # find the best neighbor
        best_neighbor = max(neighborhood, key=lambda x: fitness(x, SETS,
            ↪ COSTS))

        best_neighbor_fitness = fitness(best_neighbor, SETS, COSTS)

        # update the tabu list
        tabu_list.append(np.copy(best_neighbor))

        # update the current solution and fitness
        current_solution = best_neighbor
        current_fitness = best_neighbor_fitness

        if current_fitness > best_fitness:
            best_solution = np.copy(current_solution)
            best_fitness = current_fitness
            best_steps = steps

        history.append(current_fitness[1])

    # print the best solution and fitness and plot the history of the
        ↪ costs
    print(f"Best solution fitness: {best_fitness[1]}")
    print(f"Best solution found after {best_steps} steps")

    plt.figure(figsize = (7, 4))
    plt.plot(
        range(len(history)),
        list(accumulate(history, max)),
        color = "red"
    )

    plt.scatter(range(len(history)), history, marker = ".")
    plt.show()
```

Code 2: tabu_search_counter

```
def run_tabu_search_counter(instance):
UNIVERSE_SIZE, NUM_SETS, DENSITY = instance
print(f"UNIVERSE_SIZE: {UNIVERSE_SIZE}, NUM_SETS: {NUM_SETS}, DENSITY
    ↪ : {DENSITY}")

# total number of steps to run the algorithm
totSteps = 1000

# Counter that starts to count after I find a solution better than
    ↪ the previous one: if
```

```
# the counter reaches the value 50 without finding a better solution
    ↪ , the loop breaks.
# I used it to avoid computational problems with the instances with
    ↪ 100_000 elements.
counter = 0

# define the sets
SETS = np.random.binomial(1, DENSITY, size=(NUM_SETS, UNIVERSE_SIZE))
    ↪ .astype(bool)

for s in range(UNIVERSE_SIZE):
    if not np.any(SETS[:, s]):
        SETS[np.random.randint(NUM_SETS), s] = True

# compute the costs by summing the number of elements in each set
    ↪ and raising to the power of 1.1
COSTS = np.power(SETS.sum(axis=1), 1.1) # or np.power

# initialize the random number generator
rng = np.random.Generator(np.random.PCG64([UNIVERSE_SIZE, NUM_SETS,
    ↪ int(10_000 * DENSITY)]))

### TABU SEARCH ###
# define the tabu list as a double-ended queue with a maximum size
    ↪ in order to keep automatically only
# the last tabu_size solutions
tabu_list = deque(maxlen=50)

# initialize a random solution given a percentage of being true
initial_solution = np.full(NUM_SETS, True)

# compute the fitness of the initial solution
current_solution = initial_solution
current_fitness = fitness(current_solution, SETS, COSTS)

# initialize the best solution and fitness
best_solution = np.copy(current_solution)
best_fitness = current_fitness

# number of iterations to find the best solution
best_steps = 0

# initialize the history of the costs
history = [current_fitness[1]]

print(f"Initial solution fitness: {current_fitness[1]}")

# run the algorithm totSteps times
```

```
for steps in range(totSteps):
    neighborhood = [] # list of neighbor solutions

    for _ in range(20): # generate 20 neighbors
        neighbor = tweak(current_solution, rng, NUM_SETS) # generate a
            ↪ neighbor solution by tweaking the current one
        if not any(np.array_equal(neighbor, sol) for sol in tabu_list)
            ↪ : # check if the neighbor is NOT in the tabu list
            neighborhood.append(neighbor)

    # if there is no neighbor
    if not neighborhood:
        continue

    # find the best neighbor
    best_neighbor = max(neighborhood, key=lambda x: fitness(x, SETS,
        ↪ COSTS))

    best_neighbor_fitness = fitness(best_neighbor, SETS, COSTS)

    # update the tabu list
    tabu_list.append(np.copy(best_neighbor))

    # update the current solution and fitness
    current_solution = best_neighbor
    current_fitness = best_neighbor_fitness

    if current_fitness > best_fitness:
        best_solution = np.copy(current_solution)
        best_fitness = current_fitness
        best_steps = steps
        if(counter == 0): counter += 1 # if it's the first time I
            ↪ find a better solution, increase counter by 1

    # if counter is > 0 it means that we already found a best
        ↪ solution, so it's
    # increased at every iteration
    if(counter > 0): counter += 1

    history.append(current_fitness[1])

    # break the loop if the counter reaches 100
    if counter == 100:
        break

# print the best solution and fitness and plot the history of the
    ↪ costs
print(f"Best solution fitness: {best_fitness[1]}")
```

```
print(f"Best solution found after {best_steps} steps")

plt.figure(figsize = (7, 4))
plt.plot(
    range(len(history)),
    list(accumulate(history, max)),
    color = "red"
)

plt.scatter(range(len(history)), history, marker = ".")
plt.show()
```

1.2 Results

Plots for each instance:

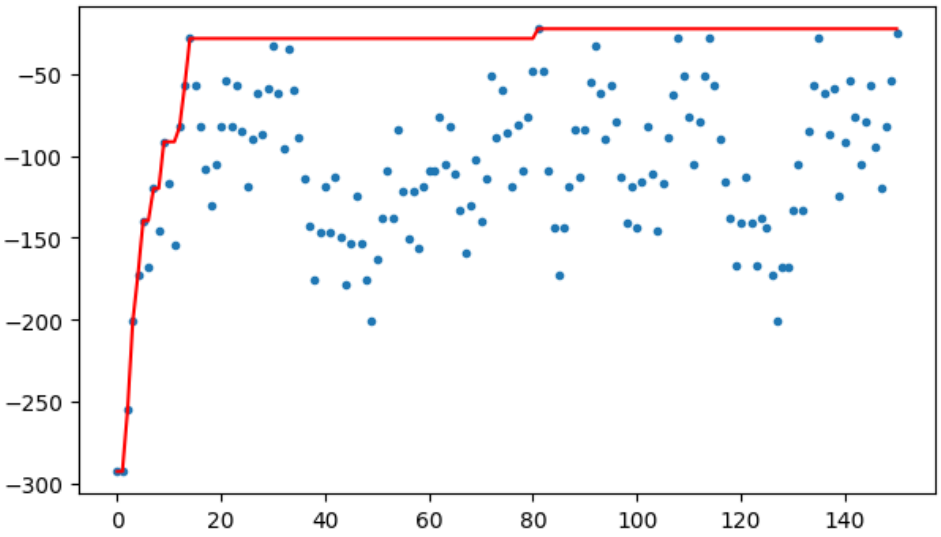

Figure 1: For instance (100, 10, 0.2).

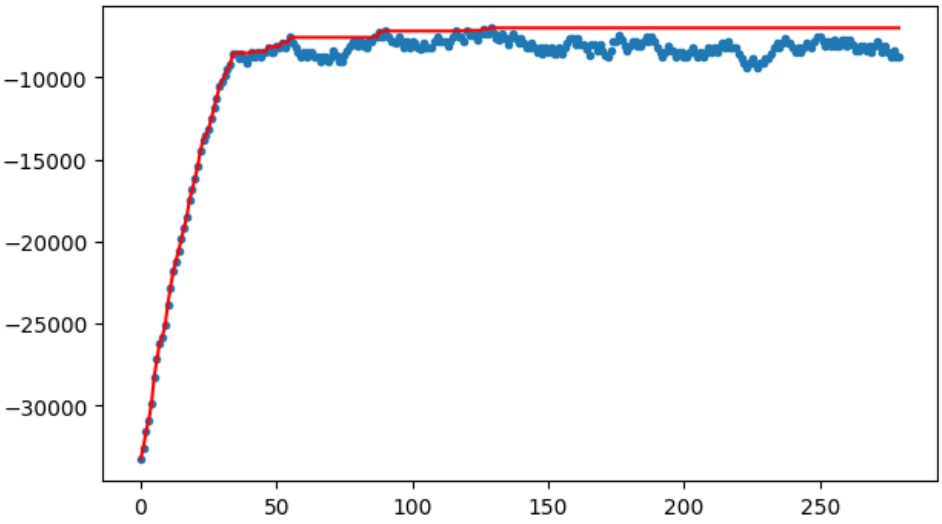

Figure 2: For instance (1000, 100, 0.2).

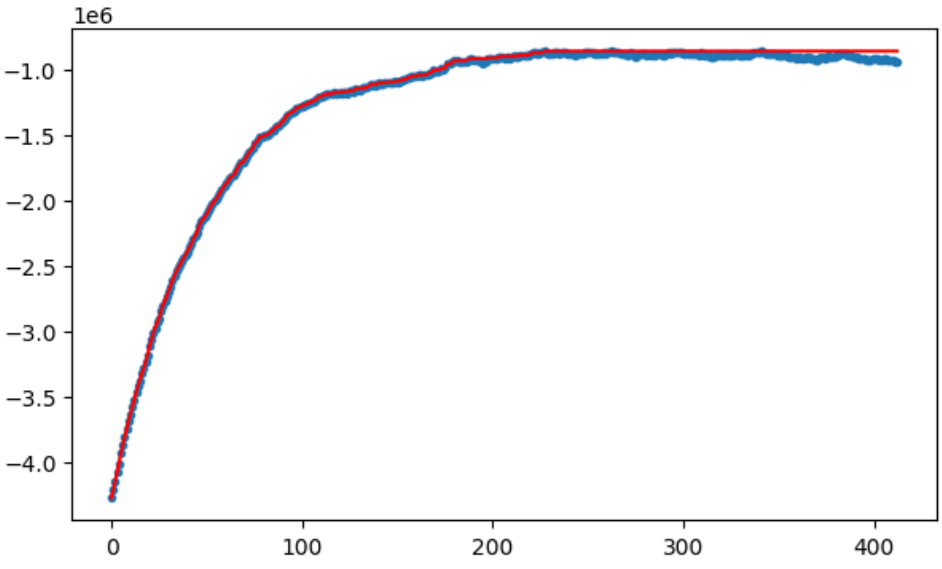

Figure 3: For instance (10000, 1000, 0.2).

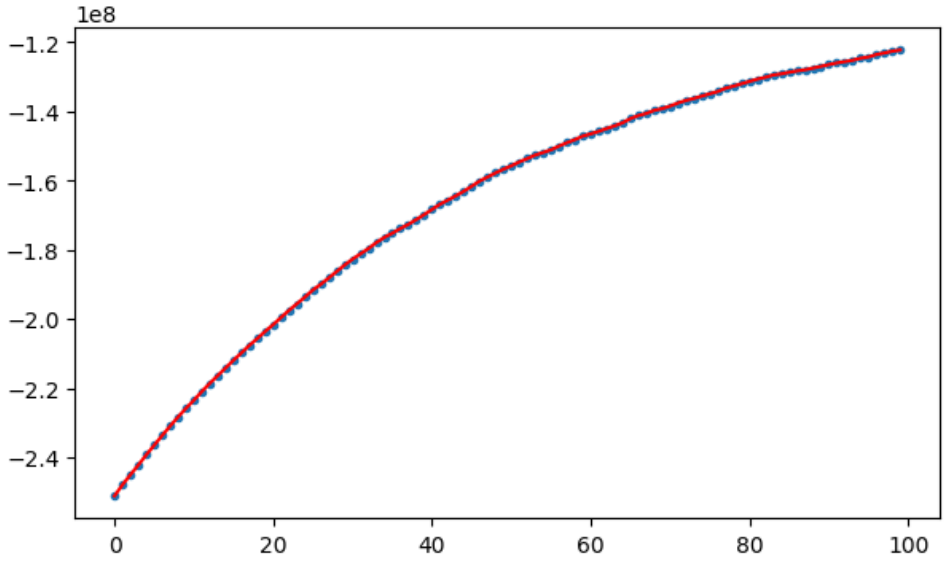

Figure 4: For instance (100000, 10000, 0.1).

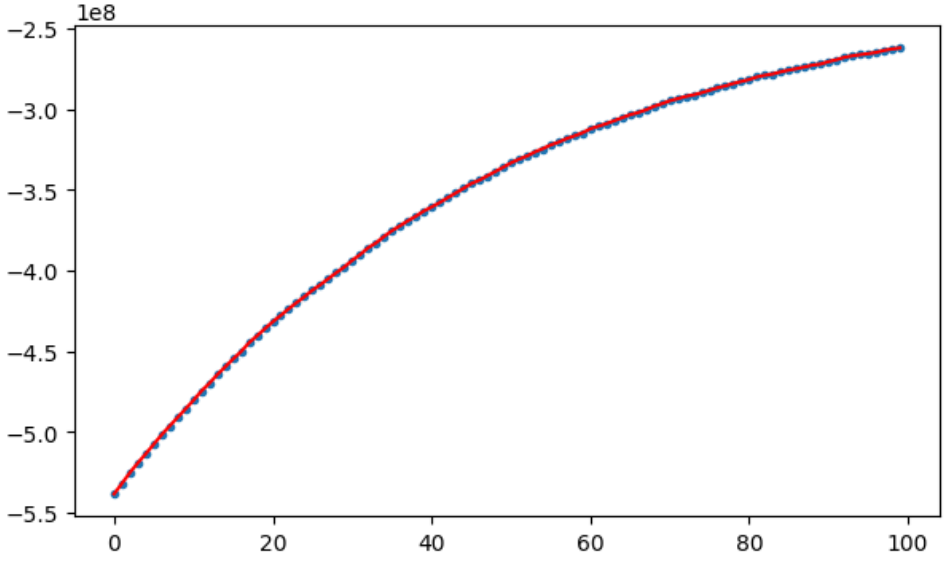

Figure 5: For instance (100000, 10000, 0.2).

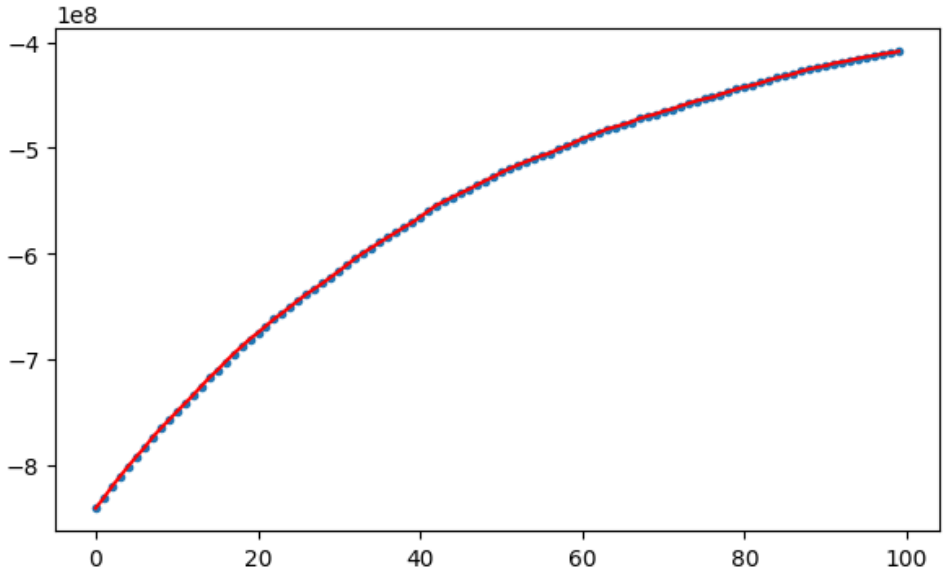

Figure 6: For instance (100000, 10000, 0.3).

Final results:

```
UNIVERSE_SIZE: 100, NUM_SETS: 10, DENSITY: 0.2

Initial solution fitness: -292.4849227546072
Best solution fitness: -292.4849227546072
Best solution found after 0 steps

---

UNIVERSE_SIZE: 1000, NUM_SETS: 100, DENSITY: 0.2

Initial solution fitness: -33267.94417889743
Best solution fitness: -6985.920549725916
Best solution found after 128 steps

---

UNIVERSE_SIZE: 10000, NUM_SETS: 1000, DENSITY: 0.2

Initial solution fitness: -4275456.167098381
Best solution fitness: -826980.8848266075
Best solution found after 518 steps

---

UNIVERSE_SIZE: 100000, NUM_SETS: 10000, DENSITY: 0.1

Initial solution fitness: -251188403.18458587
Best solution fitness: -122338481.08896929
Best solution found after 98 steps

---

UNIVERSE_SIZE: 100000, NUM_SETS: 10000, DENSITY: 0.2

Initial solution fitness: -538456653.703006
Best solution fitness: -262233908.03730047
Best solution found after 98 steps

---

UNIVERSE_SIZE: 100000, NUM_SETS: 10000, DENSITY: 0.3

Initial solution fitness: -841063092.403898
Best solution fitness: -408542986.4929375
Best solution found after 98 steps
```

1.3 Received Reviews

1. "The code is clear and well-explained, making it easy to follow and understand.

   The idea behind the implementation is to avoid considering the same solution twice using a sort of tabu-search approach.

   The solution creates a deque to store already explored solutions. It uses a steepest step approach, generating 20 neighbors of the current solution and selecting the best one based on fitness. Before proceeding, it checks that the chosen neighbor has not been previously explored. This process is repeated for a fixed number of iterations (`NumSteps`). If a better solution is found, it is saved as the "best solution" to avoid losing it.

   An intuitive solution is also provided to handle the computational overhead that arises from executing complex instances. The idea is to limit the number of iterations after a new best solution is reached.

   I really like the approach used in this solution, and it seems to be very well designed and explained.

   One possible improvement could be modifying the definition of the fitness function to move away from a simple "valid/invalid" binary approach. Instead, consider using a more incremental fitness function that allows measuring how invalid a solution is. For example, counting the number of violated constraints could provide a more nuanced evaluation of each solution.

   Overall, good job!"

2. "Tabu search was implemented to solve the problem of covering sets. Starting from a solution that includes all sets, a new solution is generated at each step by performing an XOR operation between the current solution and a random mask. Additionally, a tabu list was implemented to avoid retrying already tested solutions, which is a good approach!

   Suggestions for improvement:

   - The mutation probability is fixed at 1%. An adaptive mutation rate could be implemented, where the mutation probability increases if no improvement is found for several iterations. This would allow for more exploration when the algorithm is stuck in a local optimum.
   - Instead of a fixed number of steps (such as 1000), the algorithm could stop when the fitness improvement of the best solution becomes less than a certain threshold.
   - Dynamically adjust the size of the tabu list: If repetitive patterns or local optima are found, increasing the size of the tabu list can help to avoid these areas.
   - A more dynamic approach could be used to combine the stopping conditions, improving overall efficiency.

   Overall, great work! Keep it up!"

1.4 Given Reviews

1. "This code implements a variant of the Steepest Descent algorithm with a restart mechanism to solve an optimisation problem. The Steepest Descent method has been implemented as follows: for each iteration, the code generates a number of candidates equal to `STEEPEST_STEP_CANDIDATES = 5`, then selects the candidate with the lowest cost as the `"best_candidate."` It compares the cost of the `"best_candidate"` with the best solution found so far. If the `best_candidate_cost` is lower than the current best solution, it replaces it, and the step number along with the improvement in cost is displayed on the screen. The restart mechanism is introduced to prevent getting stuck in a single local minimum solution, allowing for further exploration. The algorithm is restarted `NUM_RESTARTS` times, initialising a new, distinct starting solution each time to find the optimal solution. The `"history"` list keeps track of the costs of each improvement for all steps and across all restarts. This approach, which combines Steepest Descent with a restart mechanism, has the advantage of finding high-quality solutions by exploring diverse regions of the solution space while minimising the risk of getting trapped in a local minimum. With the provided starting parameters (`UNIVERSE_SIZE = 10_000`, `NUM_SETS = 1000`, `DENSITY = 0.2`), the best solution found has a cost of approximately `1564258.82`. This solution could potentially be improved further by increasing the number of iterations of the algorithm. "

2. "I can't find the code to review."

2 Laboratory 2: Traveling Salesman Problem

This laboratory focuses on solving the Traveling Salesman Problem (TSP). Specifically, given different sets of cities from various countries, the goal is to find the shortest path that visits each city exactly once and returns to the starting point.

2.1 Solution

I solve the problem by combining two different approaches. The first approach uses a simple Greedy algorithm implemented in the function `tsp_greedy` (3). For each city considered as the starting point, the algorithm moves towards the city with the shortest Euclidean distance from the current one. At the end, it moves from the last city back to the starting one. The cost of each path is stored, and the algorithm returns both the shortest path found and a set of paths generated from each starting city (`paths_greedy`).

The second approach is based on a genetic algorithm implemented in the function `genetic_algorithm` (4). To accelerate the search for the optimal path, some individuals in the initial population are generated by applying mutations to the set of paths produced by the first algorithm (now stored as `BEST_SOLUTION_GREEDY`).

The fitness function of the genetic algorithm is defined as the inverse of the total computed distance. The algorithm consists of the following operations:

- `mutation(individual)` performs scramble mutation by shuffling a subset of cities in a path;
- `crossover(parent1, parent2)` performs partially mapped crossover between two parents;
- `parent_selection(population, fitness_values, tournament_size=10)` applies tournament selection to extract a number of top-performing parents equal to `NUM_PARENTS`.

Code 3: tsp_greedy

```
def distance(city1, city2):
    return ((city1.x - city2.x)**2 + (city1.y - city2.y)**2)**0.5

def tsp_greedy(cities):
    # for each city, run the greedy algorithm and keep track of the best
    ↪  solution in a dictionary with the city as key,
    # the path of the visited cities and the total distance as value
    dict_city_solution = {}

    for starting_city in cities.itertuples():
        current_city = starting_city

        # define a list of visited cities and add the starting city
        path = [starting_city.city]

        # define a list of cities to visit and remove the starting city
        cities_to_visit = cities[cities.city != starting_city.city]
```

```
    # initialize the total distance
    total_distance = 0

    # for each city to visit, find the closest city and add it to
       ↪ the path,
    # then move to that city and remove it from the cities to visit
    while not cities_to_visit.empty:
        cities_to_visit = cities_to_visit[cities_to_visit.city !=
           ↪ current_city.city]

        # find the closest city
        closest_city = None
        min_distance = float('inf')

        for city in cities_to_visit.itertuples():
            dist = distance(current_city, city)
            if dist < min_distance:
                closest_city = city
                min_distance = dist

        # check if a closest city was found
        if closest_city is not None:
            total_distance += min_distance
            current_city = closest_city
            path.append(current_city.city)

    # add the distance from the last city to the starting one
    total_distance += distance(current_city, starting_city)

    # save the city and the solution in the dictionary
    dict_city_solution[starting_city.city] = (path, total_distance)

# find the best solution
best_solution = None
best_distance = float('inf')

# for each city, check if the total distance is better than the best
   ↪  distance
for city, (path, total_distance) in dict_city_solution.items():
    if total_distance < best_distance:
        best_distance = total_distance
        best_solution = path

# return just the paths ranked by descending distance (used later
   ↪ for the genetic algorithm)
path_distance = list(dict_city_solution.values())
path_distance = [x[0] for x in path_distance]
```

```
    return best_solution, best_distance, path_distance

best_solution_greedy, best_distance_greedy, paths_greedy = tsp_greedy(
    ↪ cities)
print("Best solution: ", best_solution_greedy)
print(f"Best distance: {best_distance_greedy*100:.2f}")
print("Paths greedy: ", paths_greedy)
```

Code 4: genetic_algorithm

```
BEST_SOLUTION_GREEDY = paths_greedy
LIST_CITY = cities['city'].tolist()
LIST_CITY_SIZE = len(LIST_CITY)
POPULATION_SIZE = int(LIST_CITY_SIZE * 2.5)
NUM_ELITE = int(POPULATION_SIZE * 0.1)
MUTATION_RATE = 0.2
NUM_GENERATIONS = 100 # change it manually according to the instance
NUM_PARENTS = int(POPULATION_SIZE * 0.3)

# compute the distance between two cities
def distance(city1, city2):
    distance = ((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)**0.5
    return distance

# compute fitness of an individual
def fitness(individual):
    total_distance = 0
    for i in range(len(individual)):
        city1 = cities[cities['city'] == individual[i]]
        city2 = cities[cities['city'] == individual[(i+1) % len(individual
            ↪ )]] # circular list in order to go back to the starting
            ↪ city

        city1_x, city1_y = float(city1.iloc[0]['x']), float(city1.iloc
            ↪ [0]['y'])
        city2_x, city2_y = float(city2.iloc[0]['x']), float(city2.iloc
            ↪ [0]['y'])

        total_distance += distance((city1_x, city1_y), (city2_x, city2_y))

    return float(1 / total_distance)


# compute fitness of the population
def compute_fitness(population):
    fitness_values = []
    for individual in population:
        fitness_values.append(fitness(individual))
```

```
    return fitness_values

# check if the solution is valid
def valid_solution(individual):
    # check if all cities are explored and there are no duplicates
    if len(set(individual)) != len(individual):
        return False
    return True

# implement scramble mutation
def mutation(individual):
    r = random.random()
    if r < MUTATION_RATE:
        start, end = sorted(random.sample(range(len(individual)), 2))

        subset = individual[start:end + 1]
        random.shuffle(subset)

        individual[start:end + 1] = subset

    return individual

# generate the initial population: initialize the population with the
    ↪ individual obtained with the greedy algorithm,
# then generate the remaining individuals randomly
def generate_population():
    population = []
    for i in range(len(BEST_SOLUTION_GREEDY)):
        population.append(mutation(BEST_SOLUTION_GREEDY[i]))

    for _ in range(POPULATION_SIZE - len(BEST_SOLUTION_GREEDY)):
        individual = random.sample(LIST_CITY, len(LIST_CITY))
        while not valid_solution(individual):
            individual = random.sample(LIST_CITY, len(LIST_CITY))
        population.append(individual)
    return population

# implement the partially mapped crossover
def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))

    child = [None] * size

    for i in range(start, end + 1):
        child[i] = parent1[i]

    index = (end + 1) % size
```

```
    for gene in parent2:
        if gene not in child:
            while child[index] is not None:
                index = (index + 1) % size
            child[index] = gene
            index = (index + 1) % size

    return child

# implement the tournament selection
def parent_selection(population, fitness_values, tournament_size=10):
    selected_parents = []
    for _ in range(NUM_PARENTS):
        tournament = random.sample(range(len(population)),
            ↪ tournament_size)
        best = max(tournament, key=lambda idx: fitness_values[idx])
        selected_parents.append(population[best])

    return selected_parents

# create the next generation of the population
def evolve_population(population, fitness_values):
    sorted_population_with_fitness = sorted(zip(fitness_values,
        ↪ population), reverse=True)

    # keep the individuals sorted by fitness
    sorted_population = [individual for fitness, individual in
        ↪ sorted_population_with_fitness]

    new_population = [None] * NUM_ELITE

    # elitism but apply mutation to the elite
    for i in range(NUM_ELITE):
        new_population[i] = mutation(sorted_population[i])

    parents = parent_selection(population, fitness_values)

    while len(new_population) < POPULATION_SIZE:
        parent1, parent2 = random.sample(parents, 2)
        child = crossover(parent1, parent2)
        child = mutation(child)
        new_population.append(child)
    return new_population[:POPULATION_SIZE]

# implement the genetic algorithm
def genetic_algorithm():
    population = generate_population()
    best_solution = None
```

```
    best_distance = float('inf')

    for generation in range(NUM_GENERATIONS):
        print(f"Generation {generation + 1}/{NUM_GENERATIONS}...")

        fitness_values = compute_fitness(population)
        best_temp_index = fitness_values.index(max(fitness_values))
        best_temp_distance = 1 / fitness_values[best_temp_index]
        print(f"Best temp solution: {population[best_temp_index]}")
        print(f"Best temp distance: {best_temp_distance*100}")

        if best_temp_distance < best_distance:
            best_solution = population[best_temp_index]
            best_distance = best_temp_distance

        # evolve the population (not at the last generation)
        if generation != NUM_GENERATIONS - 1:
            population = evolve_population(population, fitness_values)

    return best_solution, best_distance

if __name__ == "__main__":
    best_solution, best_distance = genetic_algorithm()
    print("Best solution: ", best_solution)
    print(f"Best distance: {best_distance*100:.2f}")
```

2.2 Results

```
Vanuatu
- Greedy Algorithm: 1342.26
- Genetic Algorithm: 1222.27

Italy
- Greedy Algorithm: 4607.54
- Genetic Algorithm: 4530.79

Russia
- Greedy Algorithm: 52201.65
- Genetic Algorithm: 52201.65

United States
- Greedy Algorithm: 46774.72
- Genetic Algorithm: 46227.93

China
- Greedy Algorithm: 59964.51
- Genetic Algorithm: 59910.79
```

2.3 Received Reviews

1. "Congrats! You implemented a working evolutionary algorithm! Your code is easily readable and well-explained. I have only a few minor comments about it. Applying elitism and mutating the elite can lead to losing the best individuals in your population. I would suggest not mutating the elite . We are mostly interested in the vicinity trait between cities, scramble mutation isn't ideal for preserving that trait. You could try to implement also reverse mutation to better maintain that trait. Using an adaptive mutation rate that decreases with each generation could also be beneficial, making your algorithm more focused on exploitation rather than exploration toward the end of the evolution cycle."

2. "The code is well-structured, organized into logical sections: distance calculation, population generation, parent selection, crossover, mutation, and evolution. Each function is clearly defined, making the code easy to read and follow.

   It's interesting how the initial population is built by combining a greedy solution with randomly generated individuals. This approach ensures a good starting point while maintaining diversity across solutions.

   The scramble mutation is a valid technique for mixing the cities in a random sequence, introducing good variability into the solutions without completely destroying the parent information.

   Partially mapped crossover (PMX) is a solid technique for the TSP, as it preserves key features of the parental solutions during the creation of offspring, maintaining the validity of the city sequence as much as possible.

   Suggestions for Improvement:

   Introduce a country variable to simplify execution across different files.

   You might consider introducing additional mutation and crossover techniques (such as two-point crossover or inversion mutation) to make the algorithm more versatile, depending on the specific problem instance being solved.

   There is no convergence-based stopping condition, such as halting the algorithm when no significant improvements are observed in recent generations. Adding logic to stop the algorithm before reaching the maximum number of generations when improvements stabilize would help optimize runtime."

3. "Your work on the traveling salesman problem (TSP) with an approach combining the Greedy algorithm and a Genetic Algorithm is well done and shows a good understanding of both the theoretical foundations and their practical application. One aspect I noticed concerns the calculation of distances. You chose to calculate them manually using the Euclidean formula, and the method works well for small sets of cities. However, it would be useful to explore the use of a library such as Geopy, especially for larger datasets: these libraries are optimised to calculate distances between geographical coordinates, saving computational resources and improving the efficiency of the algorithm. Regarding the mutation, you have opted for a scramble mutation in both the implementation of the evolutionary algorithm and the elite solutions. This approach certainly introduces variety, but it can also lead to the loss of good gene segments, making it more difficult to maintain valid solutions between generations. I would suggest experimenting with an inversion mutation: instead of

randomly rearranging genes, this mutation reverses a section of genes, maintaining blocks that may contain good solutions. The scramble mutation, on the other hand, is good for avoiding the risk of stagnation and lack of diversity. Furthermore, as far as crossover is concerned, the partially mapped crossover (PMX) method you implemented is a good starting point. However, it can mix genes too much and break up sections of solutions that could be useful. You could try an inver-over crossover, which allows you to keep larger sections of good genes and preserve significant sequences, improving the maintenance of promising substructures between generations. Overall, this is solid work. Some minor adjustments such as the use of a distance library and slightly different mutation and crossover strategies could make your algorithm more efficient and further improve your results."

4. "Looking at the code, I do not see any particular problem that needs to be addressed.

   However, when analyzing the genetic algorithm, I have a couple of suggestions:

   - Consider using a form of mutation that minimizes changes in the edges, such as inversion mutation, instead of scramble mutation.
   - You probably do not need to apply mutation to the elitist individuals, as they are likely very similar in terms of fitness. At worst, applying mutation to them could result in the loss of some highly positive traits.
   - Setting the tournament selection size to a static 10 seems too restrictive for large sets like China. Instead, consider dynamically adjusting the size during computation—starting with a large value and gradually decreasing it to increase selective pressure."

2.4 Given Reviews

1. "To solve the Traveling Salesman Problem (TSP), this program implements a genetic algorithm that utilizes two different greedy algorithms to initialize the population.

   The first greedy algorithm is the simplest and aims to find a route that visits each city exactly once and returns to the starting city. This algorithm follows a *nearest neighbor* strategy, meaning it always selects the closest unvisited city from the current location. To achieve this, it maintains a list of visited cities and a matrix `dist` containing the distances between each pair of cities. To find the nearest neighbor, the algorithm loops until all cities are visited, and for each current city (`city`): it sets the distances to all previously visited cities to *infinity* to ignore them; the closest unvisited city (`closest`) is identified and added to the TSP route (the final path); the `city` variable is updated to this closest city, and the process repeats. After visiting all cities, the algorithm adds a final step to return to the starting city to complete the loop, then computes the fitness.

   The second greedy algorithm follows a more structured approach to solving the TSP by incrementally adding edges while avoiding cycles until the route is complete. This method starts with a sorted list of edges (segments) based on distance, attempting to create a Hamiltonian cycle (visiting each city once and returning to the starting city) by selecting the shortest available edge that does not form an invalid cycle or

exceed two edges per city. Initially, the algorithm initializes the following variables: `visited` (a set of visited cities), `edges` (a set of visited edges), and `counts` (a dictionary to track how many times each city is connected by an edge). The first edge (`start_segment`) is added to start the route, marking both cities in the edge as visited and increasing their edge count.

The genetic algorithm then iterates through the sorted edges until the tour is completed, executing the following steps for each edge: it checks if adding the edge would form a cycle using the `cyclic` function; it ensures that neither city in the edge already has two connections, as each city in a Hamiltonian cycle can only have two edges; if the edge passes these checks, it is added to `edges`, and the `visited` and `counts` sets are updated. After adding edges, the algorithm checks for two cities that are *lonely* (connected by only one edge) to attempt to close the loop: if exactly two such cities are found, a final edge is added between them to complete the cycle; if there are not exactly two lonely cities, it means a valid cycle cannot be formed, and an error message is printed. Using the edges, the algorithm constructs a valid TSP route (`tour`) by starting from one of the lonely cities, then following the edges until it completes the loop by returning to the starting city. Finally, `tour` is returned, and the fitness value is printed.

Regarding the genetic algorithm, for each generation up to `MAX_GENERATION`, a new population is created. For each population, up to `OFFSPRING_SIZE`, the following steps are executed: if the random mutation number is lower than 70%, execute `parent_selection` to select one parent from the population by applying tournament selection; if the number of the current generation is less than `MAX_GENERATION/2`, apply `inversion_mutation`, otherwise apply `inver_mutation`. If the random mutation number is higher than 70%, execute `edge_recombination_crossover` between two selected parents to generate a new child. The child is then added to the offspring population. The offspring are added to the current population, which is then sorted based on fitness (from best to worst), after which it is reduced back to its original size `POPULATION_SIZE`. At the end of each generation, the fitness of the best individual is stored in `best_fitness_per_generation`.

To further enhance the code, the following improvements are suggested: adding **elitism**, which preserves a subset of the best individuals from each generation to ensure good solutions are not lost; and implementing **dynamic crossover and mutation**, which adapts crossover and mutation rates dynamically as generations progress. Overall, the implementation effectively combines greedy algorithms with a genetic algorithm to optimize TSP solutions".

2. "The code implements two algorithms for solving the Traveling Salesman Problem (TSP), one following a greedy approach while the other follows an evolutionary approach.

   The greedy algorithm's goal is to find a route that visits each city exactly once and returns to the starting city. This algorithm follows a *nearest neighbor* strategy, selecting the closest unvisited city from the current location. To achieve this, it uses a matrix `dist` containing the distances between each pair of cities and a boolean array `visited`, of length equal to the number of cities, initialized to `False`, where if `visited[i] == True`, it means that the `i`-th city has been visited and vice versa.

Until `visited` is not entirely `True`, the algorithm executes the following steps: in the matrix `dist`, it sets the distances between the current city (`city`) and the others to infinity to prevent revisiting it; it finds the index in the current row of `dist` with the smallest distance to determine the closest city; it marks `closest` as visited and adds it to the final tour (`tsp`). Once all cities have been visited, the algorithm adds the starting city to `tsp`, then displays the total number of steps in the tour and the total distance.

The genetic algorithm offers a more complex way to find a solution, gradually improving the tours through selection, crossover, and mutation over many generations. There are two different implementations for the choice of parents: the first uses a uniform parent selection, while the second applies tournament selection. The algorithm initializes an empty `population` list of length `POPULATION_SIZE`, populated by individuals with the cities shuffled randomly (except for the `0`-th index, which is always in the first position).

Then, for `MAX_GENERATIONS` iterations, the algorithm performs the following steps to create a `new_population`: it finds two parents using the `parent_sel_uniform` function, which selects a random individual from the population (or `parent_sel_tournament` in the second implementation, applying tournament selection); it performs crossover to generate a child using `xover`, which computes order crossover on two parents; it applies inversion mutation on the new child using the `mutation` function; it computes the child's fitness and appends the child to `new_population`. At the end, the algorithm returns `best_individual` as the one with the highest fitness.

Although a great job has been done, I would like to suggest a few ideas to further improve performance: using **elitism** to retain the best individuals from one generation to the next, accelerating convergence and improving the final solution; and adapting the crossover and mutation rate as the algorithm progresses, to better balance exploration and exploitation. Overall, excellent work!".

3 Laboratory 3: Sliding-Puzzle Problem

The goal of this laboratory was to find an efficient way to solve a generic instance of the Sliding-Puzzle game with a random size.

3.1 Solution

I solved this problem by implementing the A* algorithm, a heuristic search that finds the optimal (shortest) path to the goal by considering both the actual cost to reach a state and an estimated cost to the goal. The algorithm can be found in the `algorithm(start, goal)` function (5), which takes as input the starting state of the puzzle and the final configuration of the solved puzzle. First, it checks if the puzzle is solvable using the `is_solvable(state)` function; if not, the algorithm terminates early. Otherwise, it initializes a priority queue to store the states to be evaluated, where each state is sorted based on the estimated total cost `f = g + h`, with `g` representing the number of moves taken so far and `h` computed using the heuristic function `compute_distance(state, goal)`. The algorithm explores valid moves using `get_neighbors(state, parent)`, ensuring efficient traversal while preventing redundant backtracking. When the goal state is reached, the function `final_path(dict_path, current)` reconstructs the optimal sequence of moves leading to the solution.

Code 5: A* algorithm

```
# get the neighbors (valid states) of the current state: also the
   ↪ parent is passed that represents the previous state
def get_neighbors(state, parent=None):
    neighbors = []
    size = len(state)
    zero_x, zero_y = zero_position(state) # find the position of the 0
       ↪ tile
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)] # define the possible
       ↪ moves of the 0 tile

    # for each possible move, compute the new position of the 0 tile and
       ↪  if it is valid, create a new state
    for row, col in moves:
        new_row, new_col = zero_x + row, zero_y + col
        if 0 <= new_row < size and 0 <= new_col < size:
            new_state = deepcopy(state)
            new_state[zero_x][zero_y], new_state[new_row][new_col] =
               ↪ new_state[new_row][new_col], new_state[zero_x][zero_y]

            # exclude the parent state from the neighbors
            if parent is None or new_state != parent:
                neighbors.append(new_state)
    return neighbors
```

```
# initialize the start state by making a number of random moves from
    ↪ the goal state
def init_start_state(goal_state, num_moves):
    state = deepcopy(goal_state)

    # for each move, get the neighbors of the current state and choose
        ↪ one randomly
    for _ in range(num_moves):
        neighbors = get_neighbors(state)
        state = random.choice(neighbors)
    return state

# initialize the goal state given its size
def init_goal_state(n):
    goal_state = [[0] * n for _ in range(n)]
    count = 1
    for i in range(n):
        for j in range(n):
            goal_state[i][j] = count
            count += 1
    goal_state[n - 1][n - 1] = 0
    return goal_state

# compute the distance function combining two heuristics: the Manhattan
    ↪  distance and the number of misplaced tiles
def compute_distance(state, goal):
    distance = 0
    size = len(state)
    misplaced_tiles = 0

    for i in range(size):
        for j in range(size):
            value = state[i][j]
            if value != 0:
                goal_x, goal_y = divmod(value - 1, size)
                distance += abs(i - goal_x) + abs(j - goal_y) # compute
                    ↪ the Manhattan distance
                if state[i][j] != goal[i][j]: # count the number of
                    ↪ misplaced tiles
                    misplaced_tiles += 1

    return distance + misplaced_tiles

# return the position of the 0 in the current state
def zero_position(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
```

```
                return i, j

# check if the current state is the goal state
def is_goal(state, goal):
    return state == goal

# check if the puzzle is solvable for the following cases:
# - if the size of the puzzle is odd, the puzzle instance is solvable
    ↪ if the number of inversions is even
# - if the size of the puzzle is even, the puzzle instance is solvable
    ↪ if the number of inversions plus the row of the blank tile from
    ↪ the bottom is odd
def is_solvable(state):
    size = len(state)
    list_states = [x for row in state for x in row if x != 0]

    inversions = 0
    for i in range(len(list_states)):
        for j in range(i + 1, len(list_states)):
            if list_states[i] > list_states[j]:
                inversions += 1

    if size % 2 == 1:
        return inversions % 2 == 0

    zero_x, _ = zero_position(state)
    row_from_bottom = size - zero_x

    if row_from_bottom % 2 == 0:
        return inversions % 2 == 1
    else:
        return inversions % 2 == 0

# algorithm that solves the puzzle using an implementation of the A*
    ↪ algorithm
def algorithm(start, goal):
    # check if the puzzle is solvable
    if not is_solvable(start):
        print("Puzzle is not solvable.")
        return None, None

    # initialize the list of states that need to be evaluated: every
        ↪ element is a tuple with
    # (f, g, current state, previous state)
    list_states = []
    heapq.heappush(list_states, (0, 0, start, None)) # use heap to keep
        ↪ the states sorted by f
```

```
    # a dictionary that contains the previous state for each state, used
       ↪ at the end to reconstruct the path
    dict_path = {}

    state_g = {str(start): 0} # the cost of getting from the start state
       ↪ to the current state
    state_f = {str(start): compute_distance(start, goal)} # the total
       ↪ cost of the current state

    cost = 0 # the number of states that have been evaluated

    # while there are states to evaluate, pop the state with the lowest
       ↪ f
    while list_states:
        _, g, current, previous = heapq.heappop(list_states)
        if is_goal(current, goal):
            return cost, final_path(dict_path, current)

        cost += 1

        # for each neighbor of the current state, compute the new
           ↪ state_g and state_f, then add it to the list of states
        # if it is not already present or if the new state_g is better
        for neighbor in get_neighbors(current, previous):
            next_state_g = g + 1 # the cost of getting from the start
               ↪ state to the neighbor

            if str(neighbor) not in state_g or next_state_g < state_g[str(
               ↪ neighbor)]:
                state_g[str(neighbor)] = next_state_g
                state_f[str(neighbor)] = next_state_g + compute_distance(
                   ↪ neighbor, goal)
                heapq.heappush(list_states, (state_f[str(neighbor)],
                   ↪ next_state_g, neighbor, current))
                dict_path[str(neighbor)] = current

    return None, None

# construct the path from the start state to the goal state using the
   ↪ dictionary that contains the previous state for each state
def final_path(dict_path, current):
    path = [current]
    while str(current) in dict_path:
        current = dict_path[str(current)]
        path.append(current)
    return path[::-1]
```

3.2 Results

The following results present the evaluation metrics for different puzzle sizes n, including the initial configuration, quality score, cost, and efficiency.

For $n = 3$: **Quality:** 25, **Cost:** 1858, **Efficiency:** 0.013455328310010764

$$\begin{bmatrix} 0 & 4 & 8 \\ 1 & 5 & 2 \\ 3 & 7 & 6 \end{bmatrix}$$

For $n = 4$: **Quality:** 49, **Cost:** 12615, **Efficiency:** 0.003884264764169639

$$\begin{bmatrix} 8 & 6 & 9 & 3 \\ 1 & 13 & 5 & 15 \\ 2 & 14 & 0 & 4 \\ 10 & 12 & 11 & 7 \end{bmatrix}$$

For $n = 5$: **Quality:** 61, **Cost:** 86423, **Efficiency:** 0.0007058306237922776

$$\begin{bmatrix} 1 & 13 & 9 & 10 & 3 \\ 4 & 0 & 16 & 12 & 5 \\ 7 & 8 & 2 & 14 & 15 \\ 22 & 18 & 17 & 23 & 20 \\ 6 & 11 & 21 & 19 & 24 \end{bmatrix}$$

For $n = 6$: **Quality:** 67, **Cost:** 67411, **Efficiency:** 0.0009939030721988994

$$\begin{bmatrix} 1 & 3 & 9 & 4 & 0 & 6 \\ 8 & 27 & 2 & 16 & 5 & 12 \\ 7 & 19 & 14 & 15 & 10 & 24 \\ 13 & 28 & 32 & 17 & 11 & 20 \\ 25 & 26 & 21 & 23 & 18 & 29 \\ 31 & 33 & 34 & 35 & 22 & 30 \end{bmatrix}$$

3.3 Received Reviews

1. "It seems that you implemented A* algorithm in order to solve this task which is a great choice because A* is one of the most efficient search algorithms. Thanks for providing results and explanations in your readme file. Also, your code seems very organized with a lot of explanatory comments.

   Firstly, using priority list is a very good choice. It make your implementation very fast because other approach might be sorting states in every iteration which is very inefficient compare to your approach.

   To be honest, your results seem very strong. Only improvement that I can suggest is you might consider using something different than string while storing the state list. Apart from that, everything seems fine.

   Good job, keep working!!"

2. "This code implements an N-Puzzle solver using A* with a combination of Manhattan distance and misplaced tiles as heuristics. It effectively handles state generation, solvability checks, and path reconstruction. However, a few improvements can be made:

- **Memory Efficiency:** Using `str(state)` for state comparisons can be memory-heavy. A more compact representation (e.g., tuples or bit-strings) would save memory, especially for larger puzzles. Additionally, `deepcopy` could be replaced with in-place swaps to optimize memory usage.
- **Path Reconstruction:** Instead of reversing the path after reconstruction, it could be built from start to goal directly, reducing overhead.
- **Performance Optimization:** The algorithm could benefit from pruning techniques or iterative deepening to reduce unnecessary state evaluations, particularly for larger puzzles.
- **Heuristic Improvement:** While combining Manhattan distance and misplaced tiles is effective, experimenting with other heuristics (e.g., Linear Conflict) might improve performance for certain configurations.
- **Edge Case Handling:** More robust handling for small and large puzzles could help prevent potential issues with very simple or highly complex configurations.

Overall, the implementation is solid, but optimizing memory usage, performance, and heuristic strategies could make it even more efficient. ;)"

3.4 Given Reviews

- "This code provides a solution to the N-Puzzle problem by implementing three different search algorithms: BFS, DFS, and A*.

  The Breadth-First Search (BFS) algorithm explores all possible states at a given depth (number of moves) before moving to deeper levels, ensuring the shortest solution path is found if one exists. It uses a queue to store states to be explored and a visited set to avoid redundancy. For each dequeued state, it computes all valid next states and adds unvisited ones to the queue while marking them as visited. The algorithm terminates when the target state is found, returning the solution path, or when the queue is empty, indicating no solution exists.

  The Depth-First Search (DFS) algorithm explores possible states by following one path as deeply as possible before backtracking, which may result in suboptimal solutions. It uses a stack to store states and a visited set to avoid revisiting. For each popped state, it computes all valid next states, adding unvisited ones to the stack. The algorithm continues until the target state is found, returning the solution path, or until the stack is empty, indicating no solution exists.

  The A* Search algorithm uses a priority queue `open_set` to explore states, ranking them based on the sum of the path cost (steps taken) and an estimated cost to the target (heuristic). For each dequeued state, it computes valid next states, calculates their priority, and adds unvisited ones to the queue. The algorithm terminates when the target state is found, returning the optimal solution path.

  I appreciated that you compared different algorithms and included a GIF to display the moves required to reach the solution. Additionally, regarding the A* algorithm, I liked the implemented heuristic, given by the Manhattan distance elevated to the term:

$$(0.05 \cdot \log(0.01 \cdot \text{FACTOR} \cdot \text{step} + 1) + 1)$$

where FACTOR $= \frac{\text{PUZZLE\_DIM}^2}{4^2}$, which is linearly dependent on the puzzle's size. This makes the heuristic more aggressive by progressively increasing its weight as the algorithm advances, allowing for faster convergence on larger puzzles; however, it may overestimate the cost. Good job!"

- "This code provides a solution to the N-Puzzle problem by implementing three different search algorithms: BFS, DFS, and A*.

  The Breadth-First Search (BFS) is implemented by exploring states in the order of their depth (distance from the initial state). The frontier is managed as a priority queue (`frontier`), where states are prioritized by their depth using a `priority_function` that increments with the number of explored states (`len(state_cost)`). Each valid action generates a new state, which is added to the frontier if it has not been visited yet. The algorithm terminates when the goal state is reached, and the path to the solution is reconstructed by backtracking through the `parent_state` dictionary.

  The Depth-First Search (DFS) implementation is similar to BFS, with the main difference being the priority function used to manage the frontier. While BFS prioritizes states based on their depth in increasing order (using `len(state_cost)`), DFS uses a negative depth (`-len(state_cost)`) to prioritize deeper states, effectively exploring one branch of the tree as far as possible before backtracking.

  The A* Search algorithm has a similar structure to the previous ones, with the key difference being the priority function. In A*, the priority function combines the actual cost to reach a state (`state_cost[s]`) with an estimated cost to the goal (calculated using a heuristic, such as Euclidean distance, Manhattan distance, or the number of misplaced tiles). This allows A* to balance the exploration of new states with proximity to the goal, making it both complete and optimal.

  I liked how you provided implementations of different algorithms to solve the puzzle. I would like to suggest trying a new heuristic that combines the Manhattan and `tile_not_in_place` heuristics in a weighted way, as they are the most suitable for this kind of problem.

  Overall, nice work!"

4 Project: Symbolic Regression

As briefly introduced in Chapter 0, the goal of the final project was to design an algorithm for solving the symbolic regression task as efficiently and precisely as possible. The algorithm is explained in detail in Section 4.1, while Section 4.2 presents its performance on eight predefined benchmark problems, each accompanied by a plot showing the distribution of the original data alongside the predictions made by the algorithm, as well as the fitness value, which quantifies the difference between the two distributions.

4.1 Algorithm

In this paragraph, the data structures used are first described along with their methods, followed by an explanation of the algorithm and its functions.

4.1.1 Data structures

The class `Node` represents the individual components of a formula, such as operations and numbers, and has the following attributes:

- `value`: can store an operation (unary or binary), a constant value or a variable.
- `left` and `right` (optional) attributes: both are `None` if the node represents a constant or a variable, only `left` contains another `Node` for unary operations, and both `left` and `right` contain a `Node` for binary operations.

It provides the following methods:

- `is_leaf()`: returns `True` if the node is a leaf, meaning it represents a variable or a constant (i.e., has no children).
- `is_unary()`: checks whether the node is a unary operator, ensuring it has a valid left child and no right child.
- `is_binary()`: checks whether the node is a binary operator, requiring both left and right children.
- `evaluate(variables)`: recursively evaluates the expression tree by computing the numerical value of the formula, retrieving values also when the computation fails (division by 0, square root of negative values, etc.). In these cases, the results will be `nan` or `inf` and it will be handled in the `compute_fitness` method of the class `Formula`.
- `to_formula()`: converts the expression tree into a human-readable mathematical formula as a string.
- `count_nodes()`: recursively counts the total number of nodes in the expression tree.

The complete formulas are represented by the class `Formula`, which consists of a sequence of `Node` elements representing a full mathematical expression. The attributes are:

- `root`: contains the first `Node` of the formula;
- `fitness`: the value of `100*MSE`, measuring how different the formula is compared to the original one;
- `num_nodes`: the number of nodes in the formula.

The class provides the following methods:

- `evaluate(X_values)`: computes the numerical value of the formula given the input variables.
- `to_formula()`: converts the expression tree into a human-readable mathematical formula.
- `compute_nodes()`: updates the total number of nodes in the tree.
- `compute_fitness(X_dicts, y)`: first, it calls the `evaluate` functions to compute the output values for the given formula and then, if there is any invalid value (`nan` or `inf`), puts the fitness to `inf`, otherwise calculates the fitness value using Mean Squared Error (MSE) multiplied by 100.
- `update(X_dicts, y)`: updates the fitness value and the current number of nodes of the formula.

4.1.2 Functions

The first part of the algorithm consists of generating the initial population using the `generate_population` function, which repeatedly calls `generate_random_tree` to generate and return a random formula. The newly generated formula is then validated to ensure that its fitness is neither `None` nor an infinite value (`inf`), indicating that the function evaluation has succeeded.

The genetic algorithm functions are embedded in `offspring_generation`, which generates new formulas by applying tournament selection for parent selection, two mutation operations chosen randomly, and one crossover operation:

- `tournament_selection`: creates a tournament pool composed of 80% elite individuals and 20% non-elite ones, then selects either a random individual from this pool or, with lower probability, the individual with the smallest number of nodes;
- `subtree_mutation`: selects a random node from the formula and replaces it with a newly generated random subtree;
- `point_mutation`: selects a random node from the formula and substitutes it with a new node of a similar type;
- `crossover`: produces a new child formula by combining two parent formulas selected via the selection process.

The algorithm is executed by calling the `run_evolution` function, which first generates a new population and then, for `num_epochs` iterations:

- updates the population by generating 60% of new individuals using `offspring_generation`, while the remaining 40% are created randomly to balance exploration and exploitation;
- extracts the formula with the best fitness so far and updates the global best;
- at the end, plots the distributions of the true output data and the predicted values using the best formula.

4.1.3 Solution

The full code can be found at the GitHub repository:
`https://github.com/GiovanniTagliaferri/CI2024_project-work.git`.

4.2 Benchmark

Here are the best formulas found for each problem, along with the optimal parameters used to initialize the process and the corresponding final distribution plots.

4.2.1 Problem 1

Parameter	Value
Depth	3
Population Size	50
Number of Epochs	50

Table 1: Best parameters used for the problem.

Best Formula: $sin(x_0)$
Best Fitness: 7.125941×10^{-32}

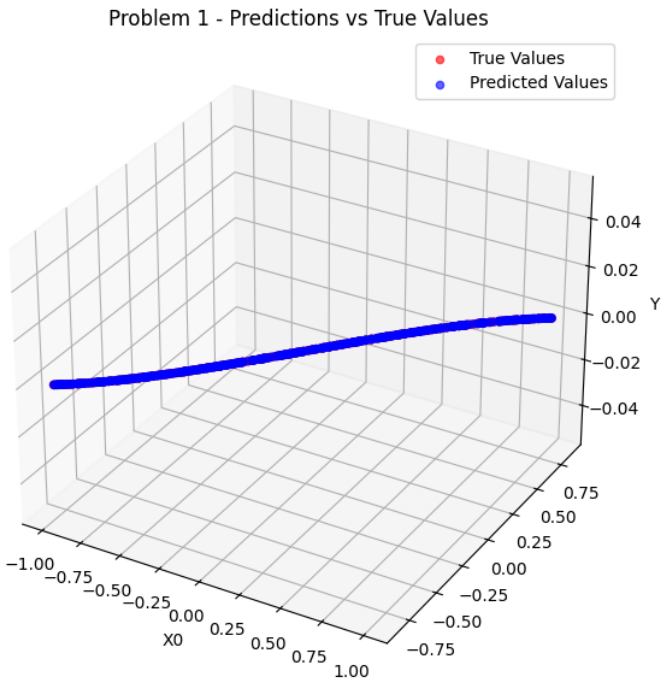

Figure 7: Problem 1.

4.2.2 Problem 2

Parameter	Value
Depth	6
Population Size	500
Number of Epochs	200

Table 2: Best parameters used for Problem 2.

Best Formula:

$$\arctan\left(\arctan\left((x_0 \cdot \arccos(\arctan(\sin(-4.99)))) - \arctan((x_0 \cdot 2.06) - x_1)\right) \cdot \exp\left(\frac{x_1}{-4.99}\right)\right.$$
$$-\arctan\left(\arctan\left(\arctan\left((x_0 \cdot \arccos(\arctan(\sin(-4.99)))) - \arctan((x_0 \cdot 2.06) - x_1)\right) \cdot \exp\left(\frac{x_1}{-4.99}\right)\right.\right.$$
$$-\arctan\left(\arctan\left(\arctan\left((x_0 \cdot \arccos(\arctan(\sin(x_1)))) - \arctan((x_0 \cdot 3.94) - x_1)\right) \cdot \arccos(\arctan(\sin(0.08)))\right)\right.$$
$$-\arctan\left(\arctan\left(\arctan\left((x_0 \cdot \arccos(\arctan(\sin(x_1)))) - \arctan((x_0 \cdot 3.94) - x_1)\right) \cdot \arccos(\arctan(\sin(x_1)))\right)\right.$$
$$\left.\left.\left.\left.\left.-\arctan((x_0 \cdot (-4.88)) - x_2)\right) \cdot (-4.88)\right) \cdot (-4.88)\right) \cdot (-4.88)\right) \cdot (-4.88) - x_2\right)\right)\right) \cdot \exp\left((4.99 - (-4.99)) + 4.99\right)\right)$$

Best Fitness: 1.265454×10^{15}

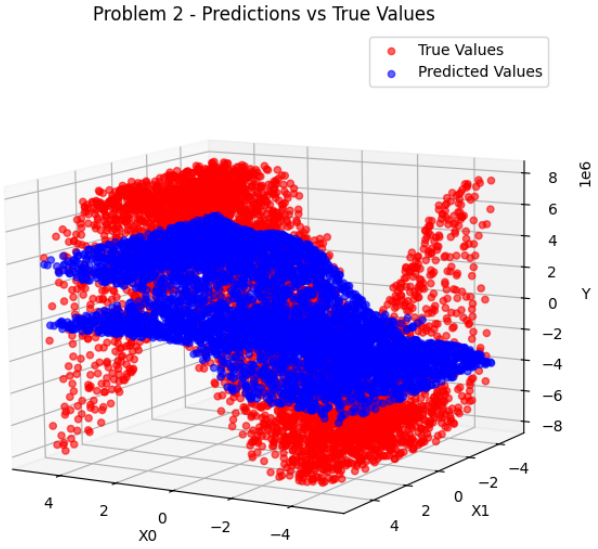

Figure 8: Problem 2.

4.2.3 Problem 3

Parameter	Value
Depth	3
Population Size	100
Number of Epochs	100

Table 3: Best parameters used for Problem 3.

Best Formula:

$$((x_1 \cdot \tanh(-4.9996) \cdot x_1) - ((-4.9996) \cdot \tanh(x_1))) \cdot x_1$$

Best Fitness: 4.4356×10^4

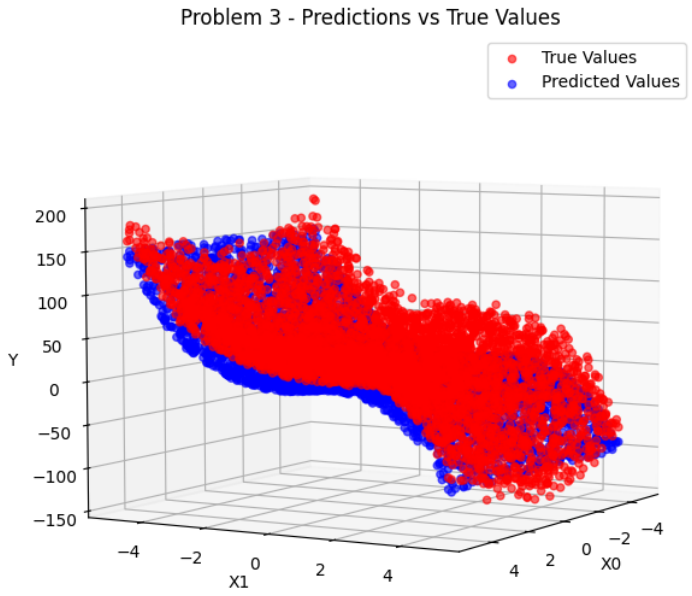

Figure 9: Problem 3.

4.2.4 Problem 4

Parameter	Value
Depth	3
Population Size	100
Number of Epochs	200

Table 4: Best parameters used for Problem 4.

Best Formula:

$$\frac{\cosh(4.15) - \cosh(\cosh(x_1))}{\exp(\cosh(x_1))}$$

Best Fitness: 4.4012×10^2

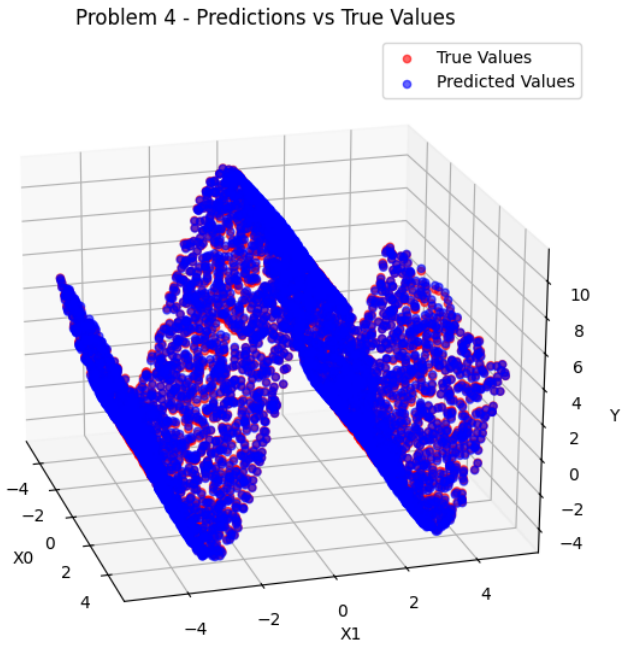

Figure 10: Problem 4.

4.2.5 Problem 5

Parameter	Value
Depth	6
Population Size	100
Number of Epochs	200

Table 5: Best parameters used for Problem 5.

Best Formula:

$$\sinh\left(\log\left(\sqrt{\sqrt{\sinh\left(\sqrt{\sinh(\sqrt{x_0})}\right)}}\right)\cdot\arcsin\left(\log\left(\sinh\left(\sqrt{\sinh(\sqrt{x_1})}\right)\right)\cdot\right.\right.$$
$$\tan\left(\log\left(\sinh(\sqrt{x_0})\right)\cdot\tan\left(\log\left(\sinh(\sqrt{x_1})\right)\cdot\tan\left(\log\left(\sinh(\sqrt{x_0})\right)\cdot\tanh\left(\sqrt{\sinh(\sqrt{x_0})}\cdot\right.\right.\right.\right.$$
$$\tanh\left(\arcsin\left(\tanh(1.0901^{x_1})\right)\cdot\tanh\left(\arcsin\left(\tanh(1.0901^{x_1})\right)\cdot\right.\right.\right.$$
$$\left.\left.\left.\left.\left.\left.\left.\left.\left.\arcsin\left(\log(\tanh(3.6245\times 3.0200))\right)\right)\right)\right)\right)\right)\right)\right)\cdot\sinh(\sqrt{x_1})$$

Best Fitness: 7.8718×10^{-17}

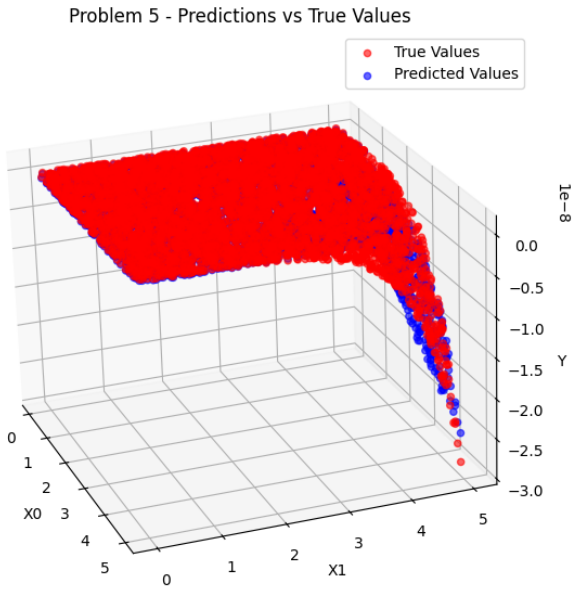

Figure 11: Problem 5.

4.2.6 Problem 6

Parameter	Value
Depth	3
Population Size	200
Number of Epochs	200

Table 6: Best parameters used for Problem 6.

Best Formula:

$$\left(\frac{x_1 + 0.8481}{0.8481^{3.1698}}\right) + (\arctan(\arctan(x_0)) - (x_0 - \arctan(-4.9989)))$$

Best Fitness: 8.0238×10^{0}

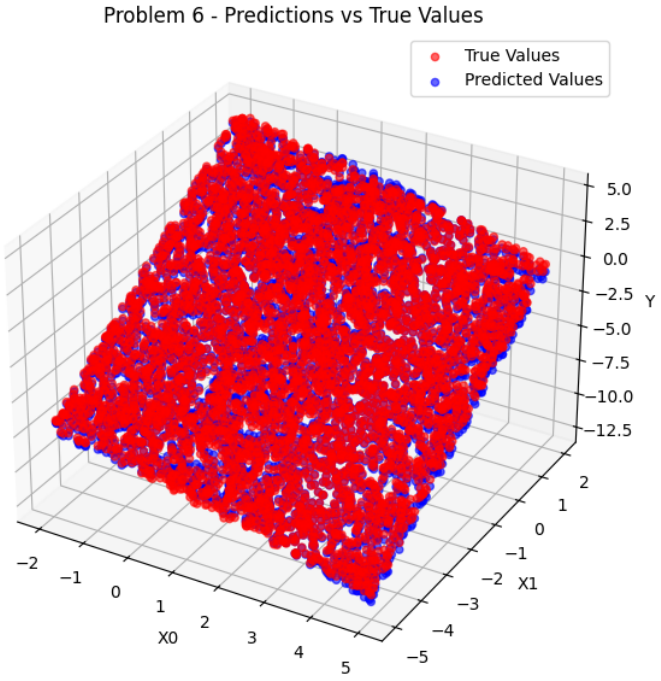

Figure 12: Problem 6.

4.2.7 Problem 7

Parameter	Value
Depth	3
Population Size	100
Number of Epochs	200

Table 7: Best parameters used for Problem 7.

Best Formula:

$$\left(\exp\left(\frac{x_1}{0.6750}\right)\right)^{x_0}$$

Best Fitness: 3.4916×10^4

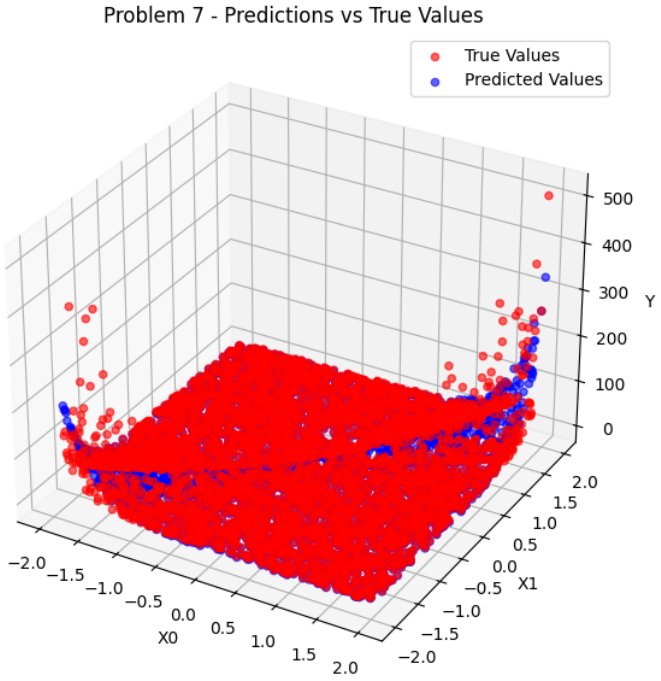

Figure 13: Problem 7.

4.2.8 Problem 8

Parameter	Value
Depth	6
Population Size	100
Number of Epochs	200

Table 8: Best parameters used for Problem 8.

Best Formula:

$$\sinh(x_5 + x_5) + \sinh(x_5 + x_5)$$

Best Fitness: 2.4131×10^8

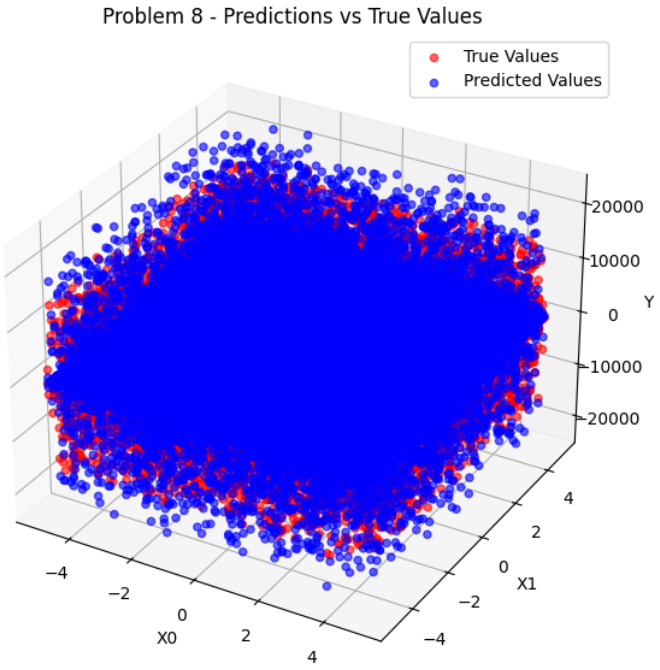

Figure 14: Problem 8.