Computational Intelligence Report

Michele Russo s326522

Abstract

This report provides an overview of all the laboratories and the project undertaken during the Computational Intelligence course. Specifically, the laboratories focus on the implementation and resolution of three key problems: the Set Cover Problem, the Traveling Salesman Problem, and the N-puzzle Problem. Additionally, the project addresses the Symbolic Regression Problem, involving the resolution of eight distinct given scenarios. The report outlines the methodologies employed to tackle these problems, presents the results achieved, and includes a discussion of these findings. Each section is dedicated to a specific problem, culminating in a conclusion that summarizes the work carried out and the results obtained.

Problem 1: Set Cover Problem

1.1 Description of the problem

The first laboratory focused on the Set Cover Problem, a classical and NP-hard problem in computer science. The problem is defined as follows: given a set of elements U={1,2,...,n}U={1,2,...,n} and a collection of subsets S={S1,S2,...,Sm}S={S1,S2 ,...,Sm}, the objective is to identify the smallest subset of SS that covers all elements in UU.

The problem was analyzed across six distinct instances, varying in the number of elements nn, the number of subsets mm, and the density of the subsets. The instances considered are as follows:

- **Instance 1**: n=100, m=10, density = 0.2
- **Instance 2**: n=1,000, m=100, density = 0.2
- **Instance 3**: n=10,000, m=1,000, density = 0.2
- **Instance 4**: n=100,000, m=10,000, density = 0.1
- **Instance 5**: n=100,000, m=10,000, density = 0.2
- **Instance 6**: n=100,000, m=10,000, density = 0.3

1.2 Implementation Details

1. Problem Initialization
The universe size (nn), number of subsets (mm), and density were used to define the problem. Each subset covers a proportion of the universe, dictated by the density parameter. A random number generator ensured reproducibility, and additional checks ensured every element in the universe was included in at least one subset.

Code Snippet:

```
UNIVERSE_SIZE = 10000
NUM_SETS = 1000
DENSITY = 0.2

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

2. Cost Calculation

The cost of each subset was determined by the number of elements it covers, raised to the power of 1.1. This approach penalized large subsets, favoring smaller, efficient ones.

Code Snippet:

```
COSTS = np.power(SETS.sum(axis=1), 1.1)
```

3. Solution Initialization and Validation

a. Solution Representation:

Solutions were represented as boolean arrays of size mm, where each entry indicates whether a subset is included in the solution.

b. Initial Random Solution:

An initial solution was generated, with each subset included with a 50% probability. This solution formed the starting point for further optimization.

Code Snippet:

```
solution = rng.random(NUM_SETS) < 0.5
```

c. Validity Check:

The valid function verified whether a solution covered the entire universe.

```
def valid(solution):
    phenotype = np.logical_or.reduce(SETS[solution])
    return np.all(phenotype)
```

d. Cost Evaluation:

The cost function calculated the total cost of a solution.

```
def cost(solution):
    return COSTS[solution].sum()
```

e. Fitness Function:

The fitness function combined validity and cost to guide the optimization process.

```
def fitness(solution: np.ndarray):
    return (valid(solution), -cost(solution))
```

4. Self-Adaptive Evolutionary Algorithm

The main optimization approach employed a self-adaptive evolutionary algorithm. Mutation strength was dynamically adjusted based on the performance of recent iterations.

a. Mutation Mechanism:
The mutation function flipped the inclusion of subsets in the solution with a probability defined by the strength parameter.

```
def multiple_mutation_strength(solution: np.ndarray, strength: float = 0.3):
    mask = rng.random(NUM_SETS) < strength
    if not np.any(mask):
        mask[np.random.randint(NUM_SETS)] = True
    new_solution = np.logical_xor(solution, mask)
    return new_solution
```

b. Strength Adaptation:
Mutation strength was adjusted based on a buffer that tracked recent improvements.

```
BUFFER_SIZE = 13
buffer = list()
for steps in range(10_000):
    new_solution = multiple_mutation_strength(solution, strength)
    f = fitness(new_solution)
    buffer.append(f > solution_fitness)
    buffer = buffer[-BUFFER_SIZE:]
    if sum(buffer) > BUFFER_SIZE / 2:
        strength *= 1.2
    elif sum(buffer) < BUFFER_SIZE / 2:
        strength /= 1.2
```

Results

- **Final Solution:**
  The final solution covered all elements of the universe, as verified by the `valid` function. Its cost was evaluated using the `cost` function.
- **Fitness Trend:**
  The fitness values demonstrated steady improvement over iterations, highlighting the effectiveness of the self-adaptive mechanism.

Visualization:

```
plt.plot(range(len(history)), list(accumulate(history, max)), color="red")
plt.scatter(range(len(history)), history, marker=".")
```

This implementation efficiently tackled the Set Cover Problem using a hybrid approach of random initialization, dynamic mutation, and self-adaptive optimization. The evolutionary algorithm successfully balanced exploration and exploitation, leading to a valid and cost-effective solution.

1.3 Results

The table provides a performance summary of the **"Set Busters: Adaptive Greedy Edition"** algorithm, including the number of calls and the resulting costs for each problem instance. Below is a breakdown of the results:

Instance	#Calls	Cost
1	10,015	270
2	10,051	7,407
3	10,456	137,111
4	13,119	44,188,974
5	13,033	92,784,643
6	13,080	145,822,753

Key Observations

1. **Instance Complexity:**
   - Instances with higher universe sizes (nn) and larger densities require more function calls and yield higher costs.
   - For instance:
     - Instance 1 (n=100n=100, m=10m=10, density = 0.2) has a significantly lower cost (270) compared to Instance 6 (n=100,000n=100,000, m=10,000m=10,000, density = 0.3), where the cost rises to 145,822,753.
2. **Adaptive Greedy Performance:**
   - The algorithm consistently maintains a small number of calls (~10,000 to 13,100), showing its computational efficiency.
   - The increase in cost correlates with the instance's complexity rather than inefficiencies in the algorithm itself.
3. **Scalability:**
   - While costs scale exponentially with problem size and density, the number of function calls scales modestly, demonstrating the robustness of the adaptive greedy approach.

1.4 Obtained reviews

The code done was reviewed by two collegues of the course. In the following section I will report the obtained revies:

Review 1 from fedspi00: “I want to start by saying that the code is very clear, thanks to the exhaustive comments that you provided, they made it easier to understand exactly how your implementation works. I see that you used a self-adaptive approach in order to find better solutions. I liked how you buffered the fitness solutions by increasing/decreasing its strength depending on if more than half the elements in the buffer were true/false for the next mutation, in order to take full advantage of the exploration-exploitation principle. All these

techniques combined show a great understanding of how to tackle such problems. Although I have two minor recommendations:

some comments were written in english and italian, which could be a little confusing for someone to understand;
explaining why certain parameters values were chosen, like the length of the buffer (13) or the strength increase/decrease (20%).
These are really small fixes, overall I think this is a very strong and clean solution, congrats!!"

Review 2 from michepaolo: "I really like the use of the strength parameter to reach a solution. (Indeed I used it too!)

Some suggestions I can give you are:

Try to use the '1 out of 5' success rule, it can be a good way to decide whether to increase or decrease the strength.

You can try setting the number of steps required by looking at the initial data dimensions of the problem. For the first ones, you can reach a good solution with fewer cycles.

I discovered that the initial value of the strength parameter is also important. You can experiment with it to see if it affects the results in terms of performance."

2 Problem 2: Traveling Salesman Problem

2.1 Problem Description

The second laboratory focused on solving the Traveling Salesman Problem (TSP). In this problem, a salesman needs to visit a set of cities and return to the starting city, with the objective of minimizing the total travel distance. The problem can be formally defined as follows: given a list of cities and the distances between each pair, find the shortest possible route that visits every city exactly once and then returns to the starting point.

For this analysis, the problem was explored using five different countries and the distances between their cities. The countries considered were:

- Vanuatu
- Italy
- China
- Russia
- United States of America

2.2 Implementation for the Traveling Salesman Problem (TSP)

The approach I implemented for solving the Traveling Salesman Problem (TSP) combines a **Greedy Algorithm** for generating initial solutions and a **Genetic Algorithm (GA)** for further optimization. Below is a detailed breakdown of the key components of the implementation.

1. Problem Setup and Input

The problem is defined by a set of cities, where each city has a geographical location (latitude and longitude). The goal is to find the shortest possible route that starts from a city, visits all other cities exactly once, and returns to the starting city.

- **City Data**: The cities' information (city name, latitude, and longitude) is loaded from a CSV file for a specific country (e.g., "Italy"). The CSV file contains the cities' names and their respective coordinates.
- **Distance Matrix**: Using the geographical coordinates, I calculate the **distance matrix** that stores the pairwise distances between every two cities using **geodesic distance** (the shortest path between two points on the Earth's surface). This matrix is later used to compute the total travel distance for any given route.

2. Greedy Algorithm for Initial Solution

The initial population of solutions is generated using a **Greedy Algorithm**, which starts from a randomly selected city and repeatedly chooses the nearest unvisited city until all cities are visited. The procedure is as follows:

- **Starting Point**: A random city is chosen as the starting point for the route.
- **Iterative Selection**: At each step, the nearest unvisited city is selected and added to the route.
- **Completion**: Once all cities are visited, the route loops back to the starting city, ensuring the route forms a cycle.

This approach ensures that a valid route is created quickly, though it does not necessarily produce the optimal solution. The greedy approach can get stuck in **local optima**, particularly when the number of cities increases.

3. Initial Population of Routes

After generating the initial greedy route, I create an initial population of potential solutions by applying the greedy algorithm multiple times (for a specified number of routes). These routes will then be refined through further optimization via the genetic algorithm.

4. Route Evaluation

Each route is evaluated based on its **total distance**. The function route_distance calculates the distance of a given route by summing the pairwise distances between consecutive cities in the

route, including the return to the starting city at the end. This total distance serves as the fitness of the solution.

5. Genetic Algorithm for Optimization

The genetic algorithm (GA) refines the initial solutions by applying two main genetic operators: **mutation** and **crossover**. The GA works by evolving the population of routes through multiple generations, selecting the fittest individuals (shortest routes), and introducing variations to explore new solutions.

5.1 Mutation: Inversion Mutation

I used **inversion mutation** to introduce diversity into the population. Inversion mutation works as follows:

- Two random cities (indices) are selected from the route.
- The segment of cities between these two indices is reversed.
- The resulting mutated route is considered for evaluation.

This mutation ensures that the population evolves by introducing small variations that could lead to better routes.

5.2 Crossover: Inver Over Crossover

The **inver-over crossover** method combines two parent routes to produce a child route. This process attempts to preserve the best segments of each parent route while introducing new combinations:

- A starting point is chosen within the child route.
- Segments of the parent routes are swapped, using the next city in the second parent to guide the crossover.
- This method strives to maintain continuity in the path while allowing the child route to incorporate beneficial segments from both parents.

6. Selection and Replacement

After generating offspring (children) via crossover and mutation, the population is updated by replacing one or more parents with the child if the child's route has a shorter distance (better fitness). The population evolves over multiple generations, improving the routes in each iteration.

7. Algorithm Performance

- **Greedy Algorithm**: The greedy algorithm provides a **fast initial solution**, but it may not always yield the optimal solution. It is prone to local optima, especially when the number of cities increases, because it does not explore the entire solution space. However, it serves as a good starting point in combination with the genetic algorithm.
- **Genetic Algorithm (GA)**: The GA explores a broader solution space by simulating the process of natural selection. It is more computationally expensive than the greedy

approach because it involves evaluating multiple solutions (routes) over several generations. However, it is more likely to find a near-optimal solution by combining good features from different routes through crossover and mutation.

2.3 Results

An output's example from generation 1 to generation 1000, for country 'Italy', is:

--- Generation 1 ---

Route 0: ['Piacenza', 'Parma', "Reggio nell'Emilia", 'Modena', 'Bologna', 'Ferrara', 'Padua', 'Vicenza', 'Verona', 'Brescia', 'Bergamo', 'Monza', 'Milan', 'Novara', 'Turin', 'Genoa', 'Leghorn', 'Prato', 'Florence', 'Forlì', 'Ravenna', 'Rimini', 'Ancona', 'Perugia', 'Terni', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Venice', 'Trento', 'Bolzano', 'Piacenza'] | Distance: 4576.195155974608

Route 1: ['Turin', 'Novara', 'Milan', 'Monza', 'Bergamo', 'Brescia', 'Verona', 'Vicenza', 'Padua', 'Venice', 'Ferrara', 'Bologna', 'Modena', "Reggio nell'Emilia", 'Parma', 'Piacenza', 'Genoa', 'Leghorn', 'Prato', 'Florence', 'Forlì', 'Ravenna', 'Rimini', 'Ancona', 'Perugia', 'Terni', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Bolzano', 'Trento', 'Turin'] | Distance: 4662.176125656765

Mutated Route: ['Piacenza', 'Parma', "Reggio nell'Emilia", 'Modena', 'Bologna', 'Ferrara', 'Padua', 'Vicenza', 'Verona', 'Leghorn', 'Genoa', 'Turin', 'Novara', 'Milan', 'Monza', 'Bergamo', 'Brescia', 'Prato', 'Florence', 'Forlì', 'Ravenna', 'Rimini', 'Ancona', 'Perugia', 'Terni', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Venice', 'Trento', 'Bolzano', 'Piacenza']

Mutated Route Distance: 4854.7488619549285

Mutated Route: ['Turin', 'Novara', 'Milan', 'Monza', 'Bergamo', 'Brescia', 'Verona', 'Vicenza', 'Padua', 'Venice', 'Ferrara', 'Bologna', 'Modena', "Reggio nell'Emilia", 'Parma', 'Salerno', 'Naples', 'Giugliano in Campania', 'Latina', 'Rome', 'Terni', 'Perugia', 'Ancona', 'Rimini', 'Ravenna', 'Forlì', 'Florence', 'Prato', 'Leghorn', 'Genoa', 'Piacenza', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Bolzano', 'Trento', 'Turin']

Mutated Route Distance: 5701.835495327372

Child Route: ['Novara', 'Milan', 'Monza', 'Bergamo', 'Brescia', 'Verona', 'Vicenza', 'Padua', 'Ferrara', 'Bologna', 'Modena', "Reggio nell'Emilia", 'Parma', 'Piacenza', 'Venice', 'Genoa', 'Leghorn', 'Prato', 'Florence', 'Forlì', 'Ravenna', 'Rimini', 'Ancona', 'Perugia', 'Terni', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Bolzano', 'Trento', 'Turin', 'Novara']

Child Route Distance: 5016.350405709363

--- Generation 2 ---

Route 0: ['Piacenza', 'Parma', "Reggio nell'Emilia", 'Modena', 'Bologna', 'Ferrara', 'Padua', 'Vicenza', 'Verona', 'Brescia', 'Bergamo', 'Monza', 'Milan', 'Novara', 'Turin', 'Genoa', 'Leghorn', 'Prato', 'Florence', 'Forlì', 'Ravenna', 'Rimini', 'Ancona', 'Perugia', 'Terni', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Venice', 'Trento', 'Bolzano', 'Piacenza'] | Distance: 4576.195155974608

Route 1: ['Turin', 'Novara', 'Milan', 'Monza', 'Bergamo', 'Brescia', 'Verona', 'Vicenza', 'Padua', 'Venice', 'Ferrara', 'Bologna', 'Modena', "Reggio nell'Emilia", 'Parma', 'Piacenza', 'Genoa', 'Leghorn', 'Prato', 'Florence', 'Forlì', 'Ravenna', 'Rimini', 'Ancona', 'Perugia', 'Terni', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Bolzano', 'Trento', 'Turin'] | Distance: 4662.176125656765

Mutated Route: ['Piacenza', 'Parma', "Reggio nell'Emilia", 'Modena', 'Bologna', 'Ferrara', 'Padua', 'Vicenza', 'Verona', 'Brescia', 'Bergamo', 'Monza', 'Milan', 'Novara', 'Turin', 'Genoa', 'Leghorn', 'Prato', 'Florence', 'Forlì', 'Ravenna', 'Rimini', 'Ancona', 'Perugia', 'Terni', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Venice', 'Trieste', 'Pescara', 'Sassari', 'Cagliari', 'Palermo', 'Trento', 'Bolzano', 'Piacenza']

Mutated Route Distance: 6115.1589941211

Mutated Route: ['Turin', 'Novara', 'Milan', 'Monza', 'Bergamo', 'Brescia', 'Verona', 'Reggio di Calabria', 'Messina', 'Taranto', 'Bari', 'Andria', 'Foggia', 'Salerno', 'Naples', 'Giugliano in Campania', 'Latina', 'Rome', 'Terni', 'Perugia', 'Ancona', 'Rimini', 'Ravenna', 'Forlì', 'Florence', 'Prato', 'Leghorn', 'Genoa', 'Piacenza', 'Parma', "Reggio nell'Emilia", 'Modena', 'Bologna', 'Ferrara', 'Venice', 'Padua', 'Vicenza', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Bolzano', 'Trento', 'Turin']

Mutated Route Distance: 6375.777535283195

Child Route: ['Novara', 'Milan', 'Monza', 'Bergamo', 'Brescia', 'Verona', 'Vicenza', 'Padua', 'Ferrara', 'Bologna', 'Modena', "Reggio nell'Emilia", 'Parma', 'Piacenza', 'Venice', 'Genoa', 'Leghorn', 'Prato', 'Florence', 'Forlì', 'Ravenna', 'Rimini', 'Ancona', 'Perugia', 'Terni', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Catania', 'Syracuse', 'Palermo', 'Cagliari', 'Sassari', 'Pescara', 'Trieste', 'Bolzano', 'Trento', 'Turin', 'Novara']

Child Route Distance: 5016.350405709363

...

--- Best Solution Found ---

Best Route: ["Reggio nell'Emilia", 'Modena', 'Bologna', 'Ferrara', 'Venice', 'Trieste', 'Pescara', 'Rome', 'Latina', 'Giugliano in Campania', 'Naples', 'Salerno', 'Foggia', 'Andria', 'Bari', 'Taranto', 'Messina', 'Reggio di Calabria', 'Syracuse', 'Catania', 'Palermo', 'Cagliari', 'Sassari', 'Terni', 'Perugia', 'Ancona', 'Rimini', 'Ravenna', 'Forlì', 'Florence', 'Prato', 'Leghorn', 'Genoa', 'Turin', 'Novara', 'Milan', 'Monza', 'Bergamo', 'Brescia', 'Trento', 'Bolzano', 'Vicenza', 'Padua', 'Verona', 'Piacenza', 'Parma', "Reggio nell'Emilia"]

Best Route Distance: 4505.8017130027165

2.4 Obtained Reviews

Review 1 from andreabioddo: “I appreciate the way you commented almost each line of your code. That makes the code more understable and clean. I've seen also that you used inner over crossover and inversion_mutation. From my point of view, they are well implemented and use them is a good idea in order to implement a good approch and efficient to solve TSP. What I suggest is just add a README to put the information clearer and more understable using tables etc..”

Review 2 from Nick18-beep: “Issue 1: Improvements to the Greedy Algorithm

Description:
The greedy algorithm used for route generation constructs a solution by selecting the nearest unvisited point, starting from a randomly chosen city. This approach is fast but has some limitations, as it can easily get stuck in local minima. Below are some proposed improvements:

1. Introduce Diversification: Currently, the algorithm always starts from a random city and selects the closest point. To enhance diversity, consider strategies like:
    - Starting from different cities.
    - Introducing randomness in the selection of the next point.
    - Modifying the nearest neighbor approach to consider second or third closest points.
2. Integration of Local Optimization Strategies: After generating the greedy route, apply local optimization techniques, such as the 2-opt algorithm, to refine the final route.

Issue 2: Enhancements to the Genetic Algorithm for Route Optimization

Description:
Now, for the genetic algorithm you’re using to solve the TSP, it benefits from an initial population and standard genetic operators, but it also has some limitations. These can reduce its ability to explore the solution space effectively and converge towards optimal solutions. Here’s what you could do to improve it:

1. Diversify Initial Population: At the moment, the initial population is generated using the greedy approach, which could limit diversity and cause premature convergence. It might help to create some random routes or use mixed techniques, like a modified nearest neighbor, for more variety.

2. Optimize the Fitness Function: The fitness function route_distance could be optimized by directly using city indices, reducing computational complexity and speeding up calculations.
3. Dynamic Stopping Criterion: Currently, the maximum number of generations (max_generations) is fixed. The algorithm could benefit from a dynamic stopping criterion, such as:
    - Terminating if there are no improvements in the best solution over a specified number of generations.
    - Using a threshold for variation in average fitness to decide when to stop.
4. Parameter Optimization: Experiment with different combinations of parameters (pop_size, mutation_rate, elitism_size, etc.) to find the most effective settings. Implementing an automatic tuning procedure may also be beneficial.
5. Crossover Selection: Based on my results, the inver_over_crossover was not the most effective choice. It may be beneficial to experiment with pmx_crossover(Partially Mapped Crossover) instead.

And by the way, Inversion Mutation is a solid choice for TSP optimization. It helps keep sub-paths continuous, making the local search more effective. Good job on including that one! ”

3 Problem 3: n-puzzle problem

3.1 Description of the problem

The third laboratory focused on solving the **n-puzzle problem**, a well-known challenge in artificial intelligence. The n-puzzle consists of a square board with **$n^2 - 1$ tiles**, each numbered from 1 to **$n^2 - 1$**, and one empty space. The objective is to rearrange the tiles from a given initial configuration to a goal configuration by sliding the tiles into the empty space.

In this problem, you are given an initial state of the n-puzzle board and a goal state. The task is to determine the sequence of moves that transforms the board from the initial state into the goal state.

3.2 N-Puzzle Problem Implementation

*Approach: A Search Algorithm**

The *A search algorithm** was chosen to solve the puzzle because it combines both **cost** and **heuristic** information to efficiently explore the solution space. The algorithm uses a priority queue to expand states in the order of their **f(n)** value, where:

- **f(n) = g(n) + h(n)**
    - **g(n)** is the cost to reach the current state.
    - **h(n)** is the heuristic estimate of the cost to reach the goal from the current state.

In this implementation, the **Manhattan Distance** is used as the heuristic function to estimate the number of moves required to reach the goal state. The algorithm explores states by sliding tiles into the empty space and continues until it reaches the goal state.

Implementation Details

Puzzle Configuration and Goal State

The puzzle is represented as a 2D **NumPy array**. For a 3x3 grid (8-puzzle), the goal state is a configuration where tiles are arranged in increasing order from 1 to 8, and the blank space (0) is in the last position. The goal state is initialized as:

```
GOAL_STATE = np.array([i for i in range(1, PUZZLE_DIM**2)] + [0]).reshape((PUZZLE_DIM, PUZZLE_DIM))
```

Utility Functions

Several utility functions were implemented to handle specific operations in the puzzle:

- **find_blank(state)**: Finds the position of the blank tile (0) in the current puzzle state.
- **is_goal(state)**: Checks if the current state matches the goal state.
- **get_possible_moves(state)**: Returns the list of valid moves (up, down, left, right) based on the current position of the blank tile.
- **move_blank(state, new_blank_pos)**: Returns a new state generated by moving the blank tile to a new position.
- **manhattan_distance(state)**: Calculates the Manhattan Distance heuristic, which sums the absolute differences in the row and column positions of each tile compared to the goal state.

A Search Algorithm*

The core of the implementation is the *A search algorithm**, which works as follows:

1. **Initialization**: The initial puzzle state is pushed onto a priority queue (frontier) with an initial cost of 0. The algorithm also maintains dictionaries to track the cost of reaching each state, the actual states, and the parent states for backtracking once the solution is found.
2. **State Expansion**: The algorithm explores all valid moves from the current state by sliding the tiles into the empty space. For each new state, the cost to reach it is calculated, and the heuristic is applied to estimate the remaining distance to the goal.
3. **Goal Check**: At each step, the algorithm checks if the current state matches the goal state. If the goal is reached, the path is reconstructed by backtracking through the parent states.
4. **Priority Queue**: The states are stored in a priority queue, with the priority determined by their **f(n) = g(n) + h(n)**value. The state with the lowest f(n) is expanded first, ensuring that the most promising states are explored.
5. **Termination**: The algorithm terminates either when the goal state is found or when all possible states have been explored without finding a solution.

```
def a_star_search(initial_state):
    frontier = []
    heapq.heappush(frontier, (0, tuple(map(tuple, initial_state)), 0, None))
    cost_so_far = {tuple(map(tuple, initial_state)): 0}
    state_map = {tuple(map(tuple, initial_state)): initial_state.copy()}
    parents = {tuple(map(tuple, initial_state)): None}

    while frontier:
```

```
    _, current_key, g_n, parent_key = heapq.heappop(frontier)
    current_state = state_map[current_key]
    if is_goal(current_state):
        path = []
        while parent_key:
            path.append(current_state)
            current_state = state_map[parent_key]
            parent_key = parents[parent_key]
        path.reverse()
        return path
```

Puzzle Initialization

The initial state of the puzzle is generated by shuffling the numbers from 1 to 8 and placing the blank space (0) at a random position. The shuffled state is then reshaped into a 3x3 grid.

```
initial_state = np.array([i for i in range(1, PUZZLE_DIM**2)] + [0])
np.random.shuffle(initial_state)
initial_state = initial_state.reshape((PUZZLE_DIM, PUZZLE_DIM))
```

Solution Path

Once the solution is found, the algorithm reconstructs the sequence of states leading from the initial state to the goal state by backtracking through the parent states. If no solution is found, the algorithm returns None.

```
solution_path = a_star_search(initial_state)
if solution_path:
    for step in solution_path:
        print(step, "\n")
else:
    print("No solution found.")
```

3.3 Results and Performance

The algorithm was tested on a randomly shuffled 3x3 puzzle. Upon running the implementation, the program successfully found the solution, providing the sequence of moves needed to solve the puzzle. The algorithm's efficiency is significantly improved by using the A* search with the Manhattan Distance heuristic, ensuring that the shortest path to the goal is found while minimizing unnecessary state explorations.

For larger puzzles (e.g., 4x4 or 5x5 grids), the same approach can be applied, although the computational complexity increases as the size of the puzzle grows. The A* search is optimal and guarantees the shortest solution, but the search space expands rapidly as the puzzle size increases.

3.4 Obtained Reviews

For this lab I have received only 1 review.

Review from efemcy2245: “The implementation of A* is good, in general is the best algorithm to find a better solution than Breadth First and Depth First.
There are some think that are not clear from by point:

- if i am using the line: new_cost = g_n + 1 for points that are not in the frontier, i am computing the cost as using a unit cost, and in theory we should use the function h() also for points that are not yet discovered.

-if state_key not in cost_so_far or new_cost < cost_so_far[state_key] this line is not clear because we are updating the cost function if the state is already seen, so we need to update the state the should be higher then the previous state, like in the slides of the professor, so should be new_state in frontier and state_cost[new_state] > state_cost[state] + cost

- maybe you could use the structure of the professor for the example of min_max optimization in which you merge in the while the condition that state is None or it is not the goal, the state None means for invalid states (fallen in a loop), for higher matrix the algorithm cann't find the solution.

finally i can say that the algorithm as is written is simple to read and very clean (all line commented).
Good job :)
”

4 Project: Symbolic Regression

4.1 Description of the problem

This project focuses on the **Symbolic Regression** problem, which is a machine learning task aimed at discovering mathematical formulas or expressions that best represent a given dataset. The goal is to identify a symbolic expression that effectively captures the relationship between input variables and corresponding output values, enabling accurate predictions and providing insights into the underlying data patterns. The problem is defined as follows: given a dataset consisting of input-output pairs, the objective is to find a symbolic expression that closely approximates the relationship between the inputs and outputs. In this particular project, we work with 8 distinct datasets, each containing a varying number of input variables and output values. The challenge is to find the formula that provides the best fit for each dataset.

4.2 Implementation

The Symbolic Regression (SR) implementation is an evolutionary algorithm that aims to discover symbolic expressions that model a given dataset. The goal is to evolve a population of mathematical expressions, represented as trees, using genetic operators like crossover and mutation, and iteratively improving the models through fitness evaluation. The implementation follows a modular approach, breaking down the problem into smaller tasks such as tree creation, selection, mutation, and evolution. Here is a detailed explanation of the entire implementation:

1. Initialization and Data Loading

The SymbolicRegression class begins by initializing important parameters such as population size, number of generations, mutation and crossover rates, and the path to a .npz file containing the dataset. The dataset is loaded using numpy, extracting x (input features) and y (target output) arrays. The number of variables in the dataset is determined by the shape of x, and variable names are dynamically generated to match the number of features.

2. Tree Creation and Representation

One of the key components of symbolic regression is the creation of symbolic trees that represent mathematical expressions. Each individual in the population is represented as a tree, where nodes are either operators (such as addition, multiplication, or sine) or leaf nodes that hold variables (like x0, x1) or constants.

The challenge in this implementation is building these trees correctly, especially ensuring the proper usage of binary and unary operators. For binary operators, which require two child nodes (left and right), the tree creation must ensure balanced subtrees. Unary operators, like sin or sqrt, require only one child node. Therefore, the tree-building function must handle both types of operators properly, ensuring that no illegal tree structures (e.g., unary operators with more than one child) are created.

The process starts with recursive calls that build the tree from the root, and at each step, the algorithm decides whether to create a binary operator node or a unary operator node. If a leaf node is needed, it will choose either a constant or a variable. A crucial part of this function is managing the node count to avoid overly deep trees, which can lead to overfitting or computational inefficiency.

3. Fitness Evaluation

Fitness evaluation is an essential step to guide the evolution of the population. The fitness of each individual (symbolic tree) is calculated by measuring how well its corresponding expression fits the data. The fitness function is typically the Mean Squared Error (MSE) between the predicted and actual output values, and the goal is to minimize this error.

Each tree in the population is evaluated by applying the symbolic expression it represents to the input data and calculating the MSE. The fitness value is then assigned to the individual, which will be used in subsequent selection, crossover, and mutation operations. The implementation also includes error handling to ensure that individuals with invalid expressions (e.g., division by zero or mathematical errors) are assigned high fitness values (poor solutions) to avoid their selection in future generations.

4. Selection Mechanism

Selection is a critical aspect of evolutionary algorithms, ensuring that individuals with better fitness values have a higher chance of being selected for reproduction. In this implementation, **tournament selection** is used. During tournament selection, a random subset of the population is chosen, and the best individual (the one with the lowest fitness value) is selected from this group.

This method promotes the survival of the fittest individuals while maintaining diversity in the population. The size of the tournament is dynamically adjusted based on the fitness of the best individual, favoring exploration when the model's error is high and exploitation when the model is performing well.

5. Crossover (Recombination)

Crossover is the genetic operator that combines two parent individuals to produce offspring. In symbolic regression, this involves selecting random subtrees from two parent trees and swapping them to create new offspring. This process is known as **subtree crossover**.

The crossover function ensures that the resulting child trees are valid, i.e., they must respect the rules for binary and unary operators. The depth of the trees is also controlled to ensure that the offspring do not grow too deep, which could lead to overfitting or computational inefficiency. If a child tree violates the depth constraints, the algorithm reverts to the parent tree.

The challenge here is maintaining the structural integrity of the trees after crossover while ensuring that the resulting offspring can still represent valid mathematical expressions.

6. Mutation

Mutation introduces small, random changes to an individual to explore the search space further. In this implementation, a **point mutation** is applied to randomly selected nodes in a tree. If the selected node is a leaf node (constant or variable), it is replaced with a new randomly selected leaf. If the node is an operator, it is replaced with a randomly chosen operator of the same type (either unary or binary).

Mutation is typically applied with a lower probability compared to crossover, as the goal is to introduce small variations rather than large changes. The mutation rate is adaptive, changing during the evolution process based on the error (MSE) of the best individual. When the error is high, the mutation rate is increased to promote exploration; when the error is low, the mutation rate is decreased to allow for more exploitation of the current best solutions.

7. Elitism

Elitism is a mechanism that ensures the best individuals are preserved in each generation. The top elitism_sizeindividuals, based on their fitness values, are carried over to the next generation without any changes. This guarantees that the population does not lose the best solutions and that they can continue to evolve and improve.

The elitism size is dynamically determined, with a default of 10% of the population, but it can be adjusted based on the size of the population.

8. Evolutionary Process

The main loop of the evolutionary process involves iterating through multiple generations, each of which consists of the following steps:

1. **Evaluate Fitness:** Evaluate the fitness of each individual in the population.
2. **Selection:** Select individuals based on their fitness using tournament selection.
3. **Crossover and Mutation:** Apply crossover and mutation to create offspring.
4. **Elitism:** Carry over the best individuals from the current generation.
5. **Replace Population:** Replace the old population with the new generation.

The process continues until a stopping condition is met, such as reaching the maximum number of generations or achieving a satisfactory error threshold. During evolution, the algorithm dynamically adjusts parameters like the mutation rate and tournament size based on the MSE percentage of the best individual, helping to guide the exploration-exploitation tradeoff.

9. Stopping Criteria

The algorithm includes stopping conditions based on both the number of generations and the MSE of the best individual. If the best individual reaches a satisfactory MSE (e.g., below 10%), the algorithm will terminate early, as it has found a good solution. If the MSE is still high after the maximum number of generations, the algorithm can extend its runtime to continue evolving and refining the solution.

Additionally, there are safeguards to ensure that the population remains valid throughout the process, with checks for invalid individuals and tree structures.

Overview

The overall approach to symbolic regression combines genetic algorithms with the symbolic representation of mathematical expressions. One of the most challenging aspects of the implementation is **correctly building and managing the tree structures**, especially handling binary and unary operators properly. The algorithm's effectiveness depends on balancing exploration (mutation and crossover) with exploitation (elitism and fitness-based selection). The adaptive nature of the algorithm, including dynamic mutation rates and tournament sizes, helps it find optimal or near-optimal solutions for symbolic regression tasks.

4.3 Results and Discussion

I obtained relatively good results for most datasets.

Dataset 1

MSE value: 7.125940794232773e-34

Best individual: sin(x0)

Dataset 2

MSE value: 1.90651e+15

Best individual: ((((x[0] + (0.75024 + -0.12047)) * np.square((330.89 - ((-0.86566 / x[0]) - (x[0] - (np.square(x[1]) * x[1] * x[2])))))) / 0.086551)

Dataset 3

MSE value: 14406.4

Best individual: (np.exp(np.sin((np.square(6.8302 - x[1])) + 3.9603))) + \

((11.053 - x[2]) + ((np.exp(0.00047697 - x[1])) - np.exp(x[1])) + np.square(x[0]))

Dataset 4

MSE value: 682.051

Best individual: np.exp(np.exp(np.cos(x[1]))) - np.sqrt(5)

Dataset 5

MSE value: 1.08094e-16

Best individual*:* np.sin(np.sin((np.exp((x[1] + x[0])) * -6.4885e-13)))

Dataset 6

MSE value: 254.818

Best individual: x[1] - np.abs(x[0] * np.cos(np.sqrt(np.abs(np.log(5) + np.pi) * 2)))

Dataset 7

MSE value: 54177.8

Best individual: np.exp((((np.cos((x[1] * (x[1] - 0.13746))) + np.cos((x[0] * x[0]))) * -0.941) + (((x[1] * 0.78139) * x[0]) - 0.65089))) + 4.0888

Dataset 8

MSE value: 8.07006e+08

Best individual: x[5] * (np.exp(7) * np.log(np.pi))