Intro

Author: Andrea Fontana, s324581@studenti.polito.it

This report presents the laboratory performed, with the reviews provided and received, and the final project which aims to solve a symbolic regression problem for 8 provided datasets.

I gave the reviews to those assigned to me as reported in the following link: http://tiny.cc/ci24_github

Lab0 – irony.md

It is just about creating a irony.md file in a repository called CI2024_lab0.

You can reach my implementation at: **https://github.com/andrefo27/CI2024_lab0**

Implementation

Content of the file **irony.md**:

Why did the AI go to therapy? Because it had too many neural issues!

Received Reviews

From **CappePolito:**

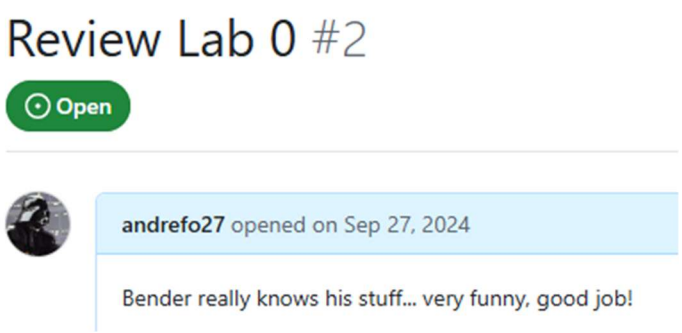
- 

From **sergiolampidecchia**:

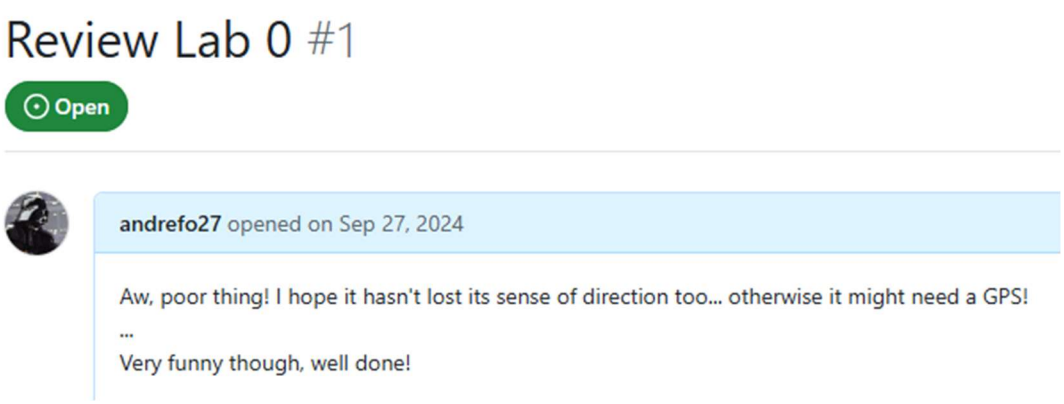
- 

Given Reviews

To **SamueleVanini**

- 

To **XhoanaShkajoti**

- 

Lab1 – Set Cover

You can find my implementation at: https://github.com/andrefo27/CI2024_lab1

Main focus is to solve the **set cover problem**: select the smallest number of subsets from a collection that cover all elements of a given universe with minimum cost.

There are six instances of the problem to solve. Each one specifies a "**density**", that represent the probability an element of the universe belongs to a specific subset.

Instance	Universe size	Num sets	Density
1	100	10	0.2
2	1,000	100	0.2
3	10,000	1,000	0.2
4	100,000	10,000	0.1
5	100,000	10,000	0.2
6	100,000	10,000	0.3

I tried using **simulated annealing technique** to solve the problem. Basically is **Hill Climbing** with a probability p != 0 of accepting a worsening solution s', where f(s) > f(s'):

$$p = e^{-\frac{f(s)-f(s')}{t}}$$

The problem was resolved with two different implementation:

- **set-cover.ipynb**
- **set-cover-threads.ipynb**

Implementation in "set-cover.ipynb"

A boolean matrix **SETS** is created to represent the sets and the elements they contain. It ensures that every element in the universe is covered by at least one set and calculates a cost for each set. The cost is a function of the cardinality of each set, which is the number of elements it contains.

Several functions are defined to help evaluate solutions:

- **valid(solution):** checks if a solution (a selection of sets) covers all elements of the universe.
- **cost(solution):** calculates the total cost of a solution.
- **tweak(solution)**: introduces a small variation in an existing solution by randomly "tweaking" the selection of sets.
- **fitness(solution):** returns a tuple that evaluates the quality of a solution, prioritizing validity and minimizing cost. The function returns the cost as a negative value to facilitate fitness maximization.

This is the reason why cost is returned negative:

- `(0, -100) > (0, -500)`
- `(1, -100) > (1, -500)`

The program uses the **Simulated Annealing** algorithm, which is a variant of the **Hill Climbing** algorithm. Unlike Hill Climbing, Simulated Annealing has a probability of accepting worse solutions to avoid getting stuck in a local minimum. The probability of accepting a worse solution depends on the cost difference and a "**temperature**" that gradually decreases over time.

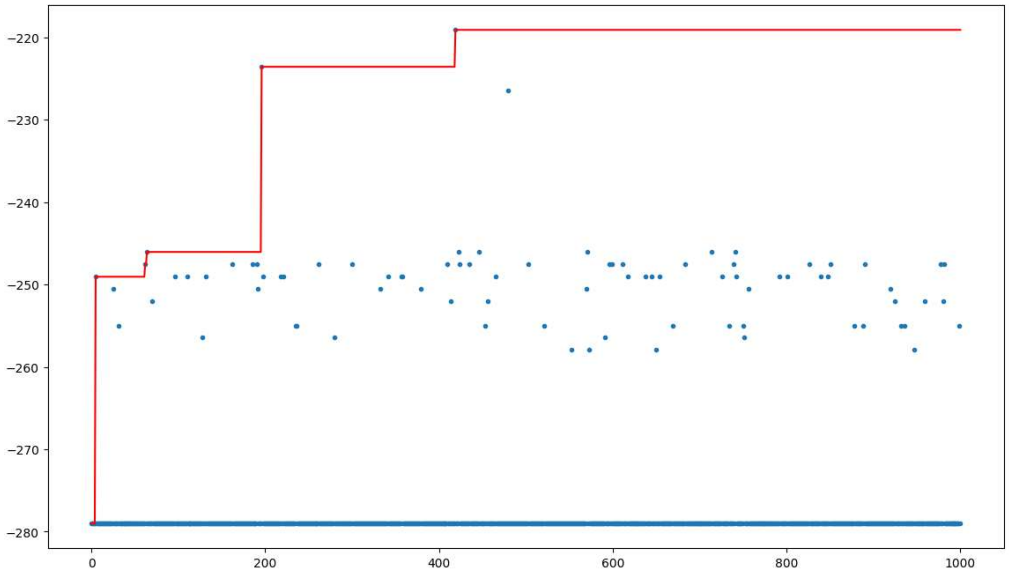
The algorithm starts with an initial solution and iterates for a predefined number of times, exploring the solution space. At each iteration, it generates a new solution "**neighboring**" the current one, calculates its fitness, and decides whether to accept it based on the Simulated Annealing rules. The algorithm keeps track of the best solution found so far and its cost. At the end of the iterations, it returns the best valid solution found and its cost.

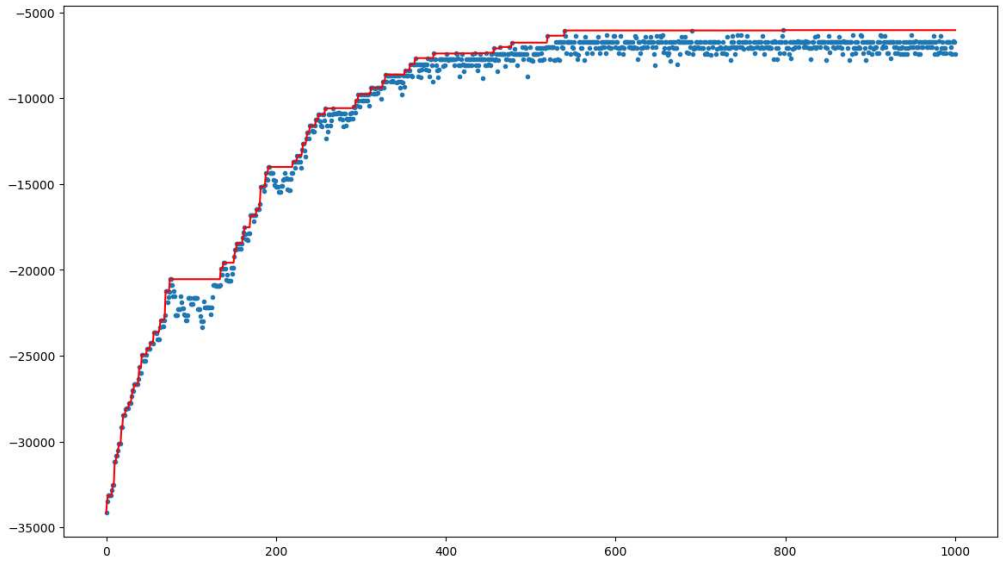
The solution is considered efficient if it can find a valid cover with a minimal total cost.

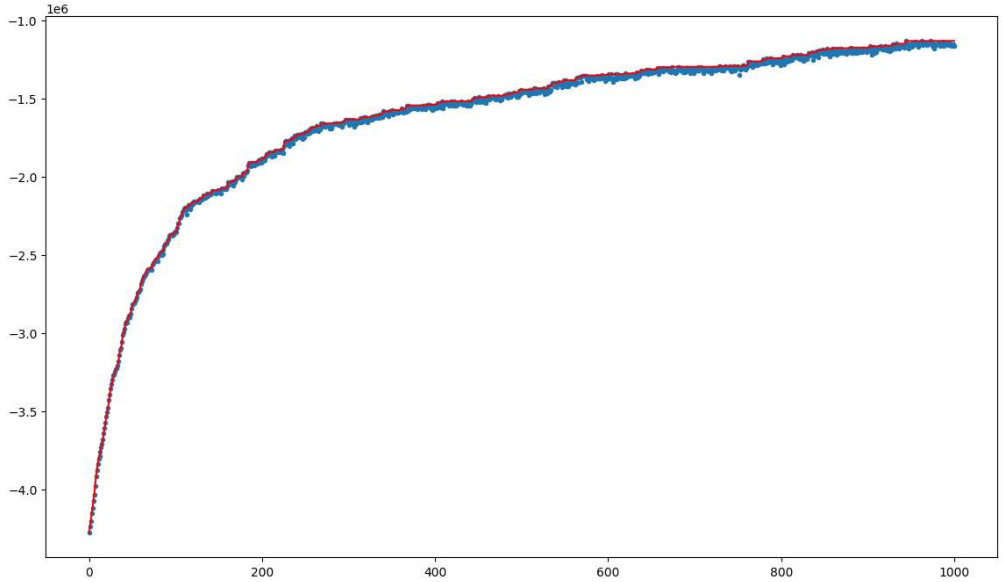
The following table shows the results obtained in the "set-cover" file with a fixed initial temperature of 7000:

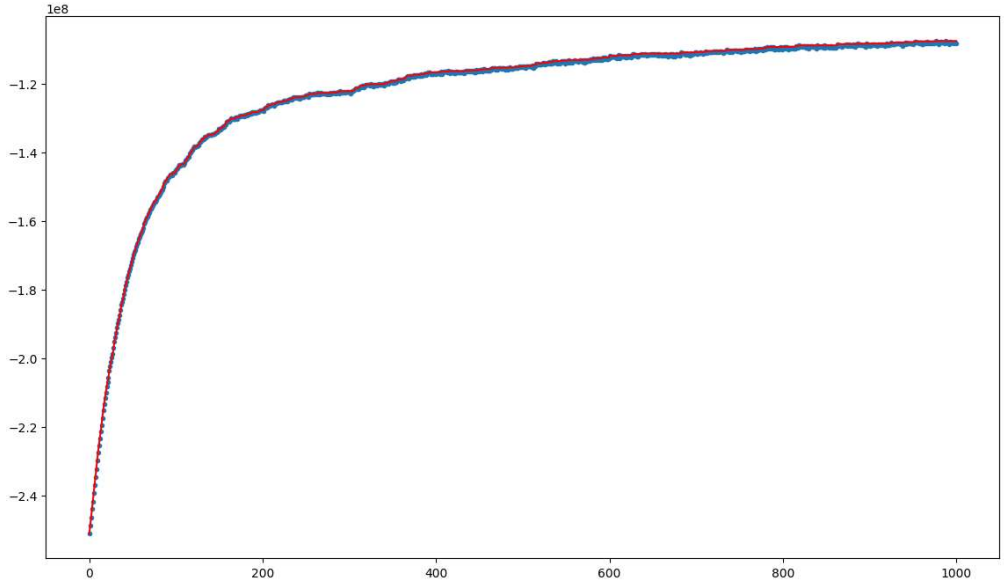
Instance	Best Cost	Num Iterations	Initial Temperature
1	279.0280561742059	1000	7000
2	6708.367353432016	1000	7000
3	1131628.7381450636	1000	7000
4	107483212.70203757	1000	7000
5	227458975.22152615	1000	7000
6	359460786.89962685	1000	7000

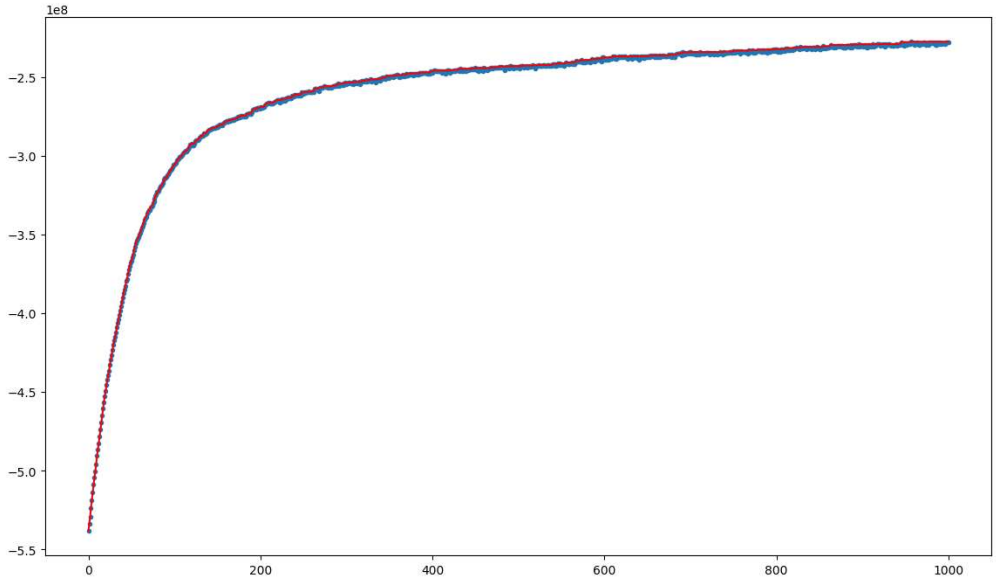
The following images represent the cost of the solution found at every iteration:

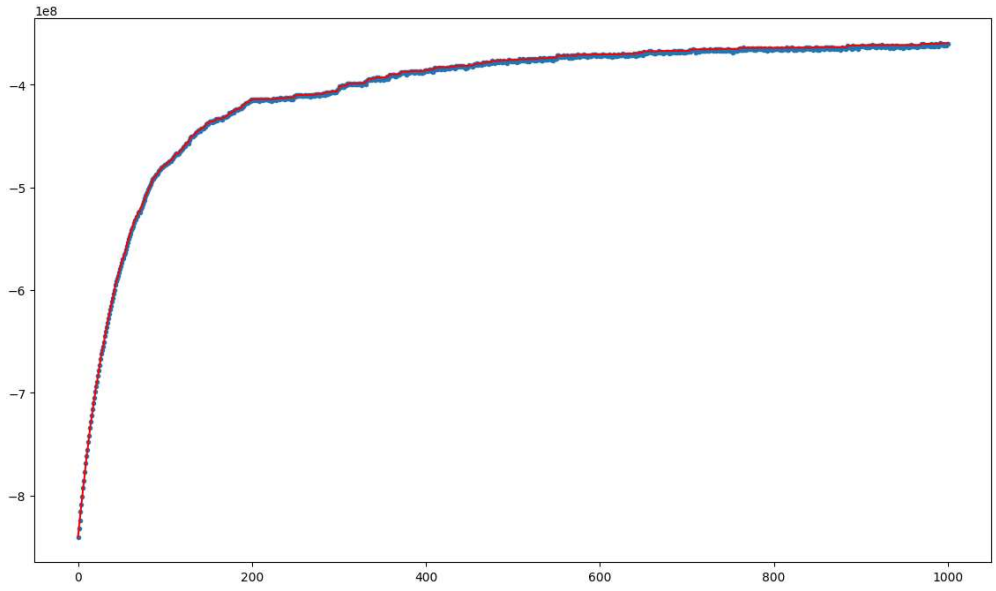
- **Instance 1**
  - 

- **Instance 2:**
  - 

- **Instance 3:**
  - 

- **Instance 4:**
  - 

- **Instance 5:**
  - 

- **Instance 6:**
  - 

Implementation in “set-cover-threads.ipynb”

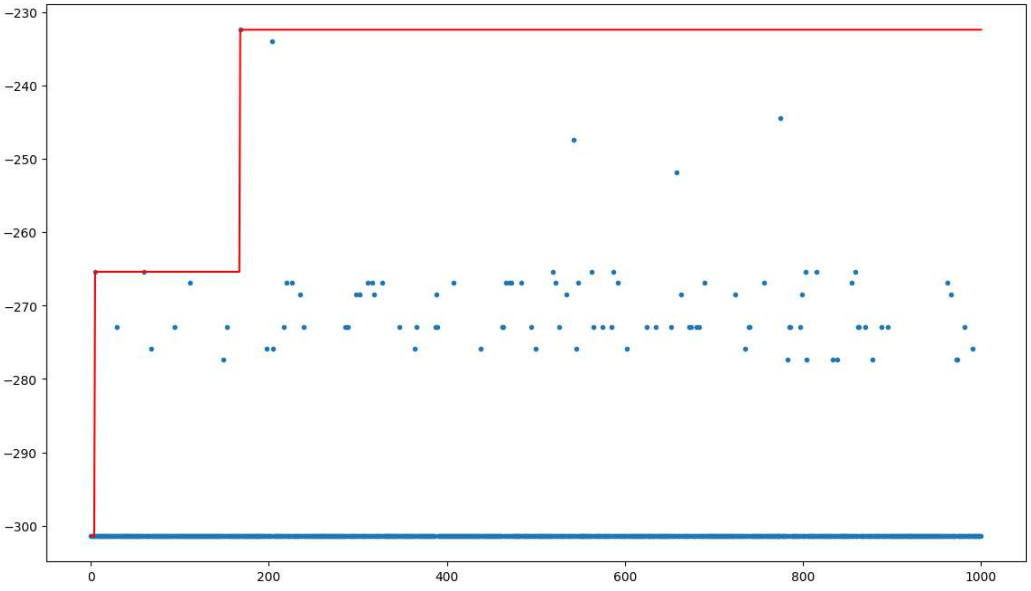
It exploits a **threadpoo**l of 4 threads for each istance examined.

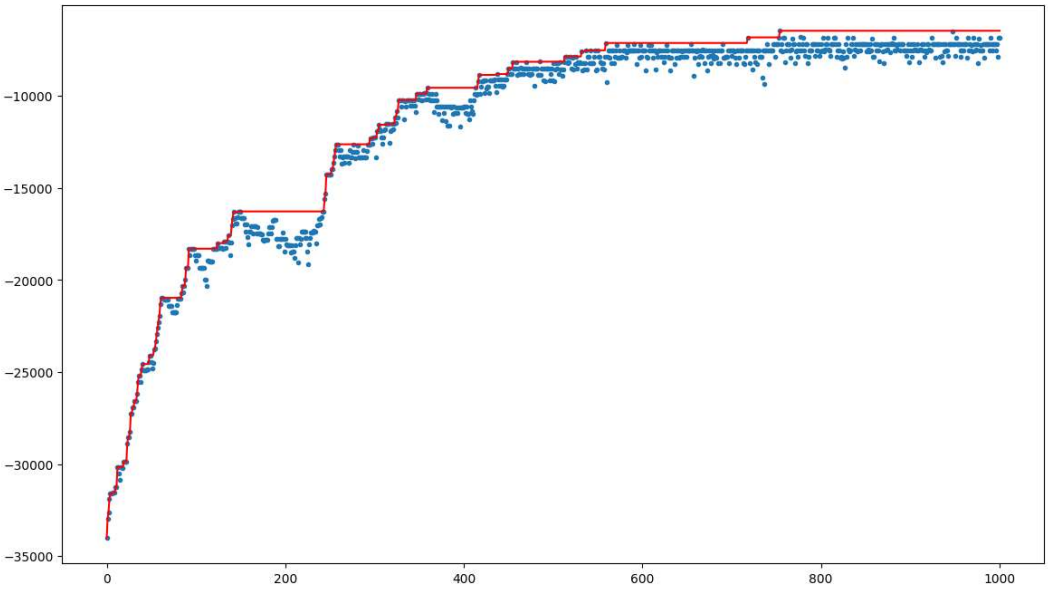
Since the algorithm used to solve the set cover problem employs simulated annealing with temperature and cooling rate, i don't know which initial temperature is best to start with. Therefore, for each instance of the problem, I am using 4 different initial temperatures ([7000, 12500, 17000, 25000]). When all the threads (for each instance) have finished, the best solution is displayed (the one with the minimum cost), indicating the initial temperature used.

The following table displays the best results obtained after executing the thread pool for each instance, with particular reference to the initial temperature used:

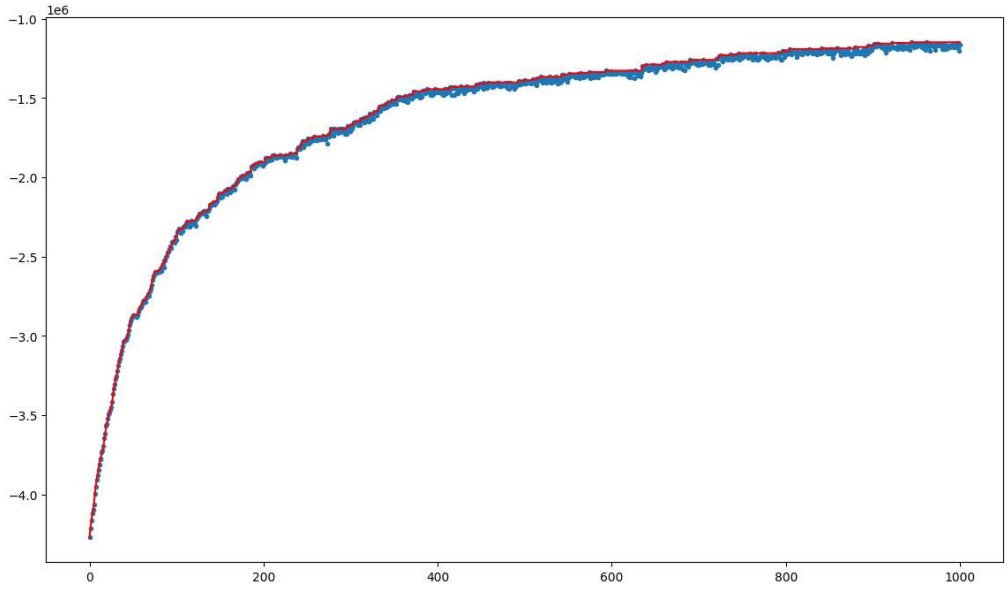
Instance	Best Cost	Num Iterations	Initial Temperature
1	301.41089319790115	1000	17000
2	6842.383862045828	1000	25000
3	1147787.8210480367	1000	7000
4	105649574.6579897	1000	7000
5	227435351.39098784	1000	7000
6	354687657.1795525	1000	25000

The following images represent the cost of the solution found at every iteration (it shows the evolution of the best solution among the 4 examined for each istance):

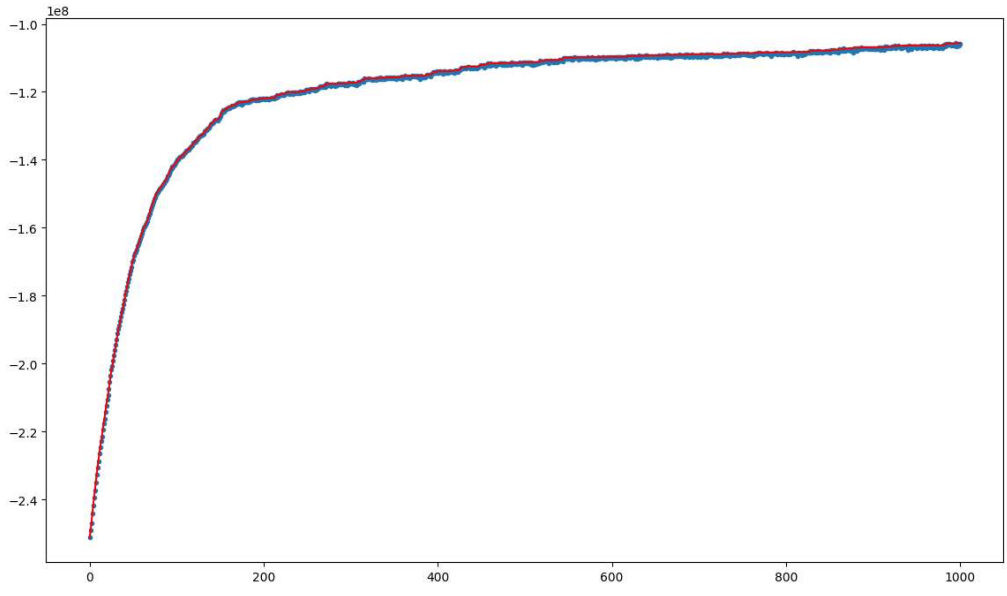
- **Istance 1:**
  - 

- **Istance 2:**
  - 

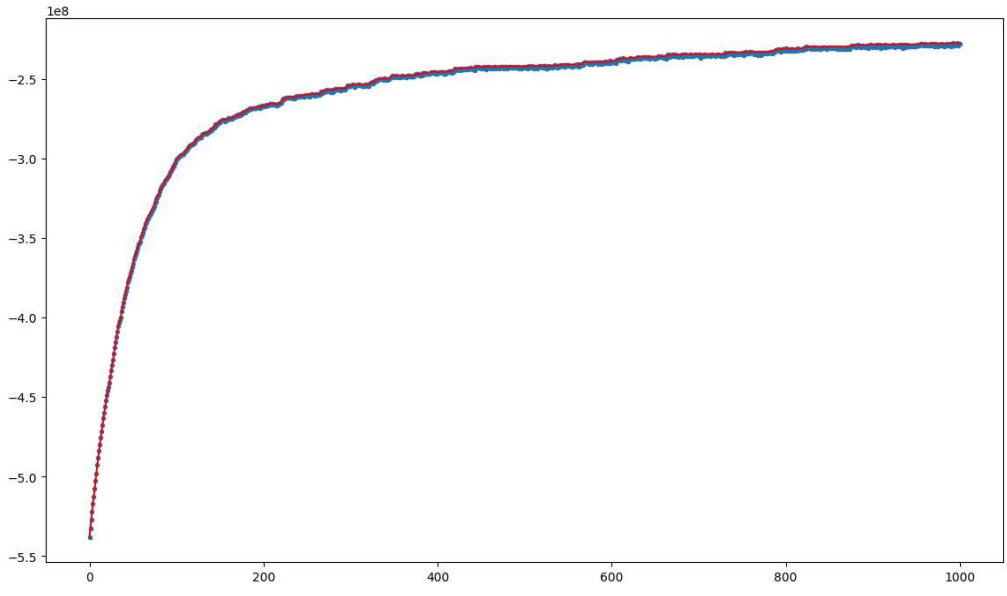
- **Istance 3:**

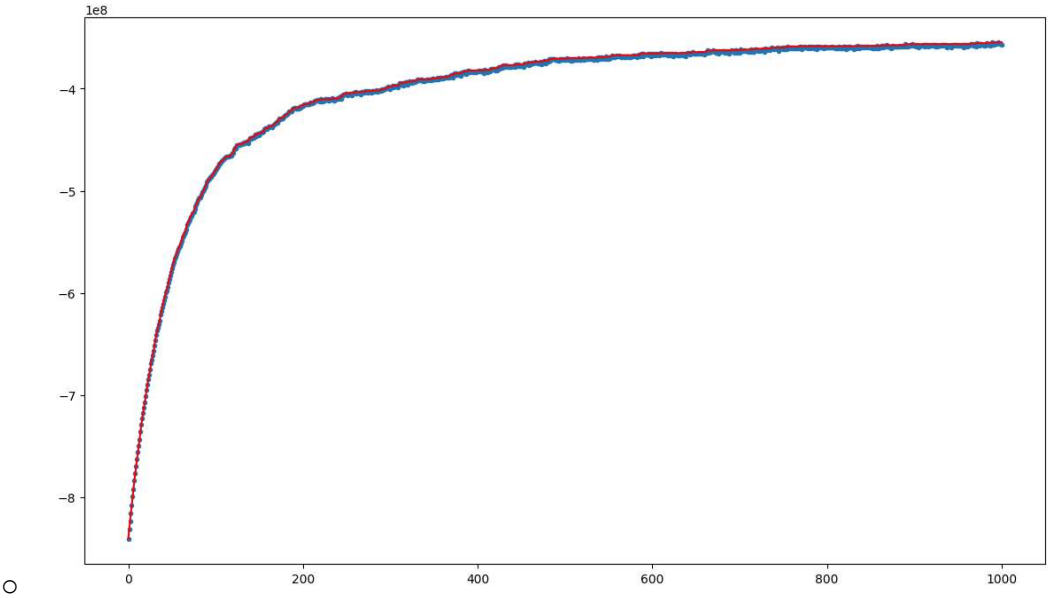


- **Istance 4:**



- **Istance 5:**



- **Istance 6:**
  - 

Given Reviews:

To **Luozirui597**

- **andrefo27** opened on Oct 18, 2024

  As regards **instance 1**:
  I tried some executions and i noticed that the initial solution is not valid till the end of the experiment, so there is never an optimization in terms of costs.
  I changed the initial solution to something always True and i noticed that some little tweak are accepted, rarely. Maybe is as you said, the change each time is not big enough.

  As regards **instance 2**, nothing to say, good job.

  As regards **instance 3**:
  After 20 restarts, seems that the optimal solution does not improve, this is because, as far i understood, you are resetting the current solution (that could lead to the global optimum).
  It would have been interesting to take the candidate solution for restart 20 and perform further steps to verify whether it would have undergone improvements.
  It's ok by the way.

  As regards **instance 4**:
  It would have been interesting to take the candidate solution for restart 3 and perform further steps to verify whether it would have undergone improvements.

  As regards **instance 5**:
  The tweak function is interesting, but there are few steps for each restart.

  As regards **instance 6**:
  From the graph results that is more and more difficult to reach a better solution after a lot of restarts as the straight lines suggest for the restarts 719 (circa) and 1769 (circa) respectively.

  In conclusion I can say that you have done a good job exploring various alternatives and the graphs you have plotted offer a valid assistance in the understanding of the problem.

- Response received from **Luozirui597**
  - **Luozirui597** on Oct 18, 2024 — Owner

    Thank you for your thoughtful suggestion! I'm glad to receive recognition from Darth Vader. :D
    Overall, thank you for your encouraging words and constructive feedback!
    It's great to know that the exploration of different alternatives and the visualizations have been helpful in understanding the problem.
    I'll keep these insights in mind and try do better!

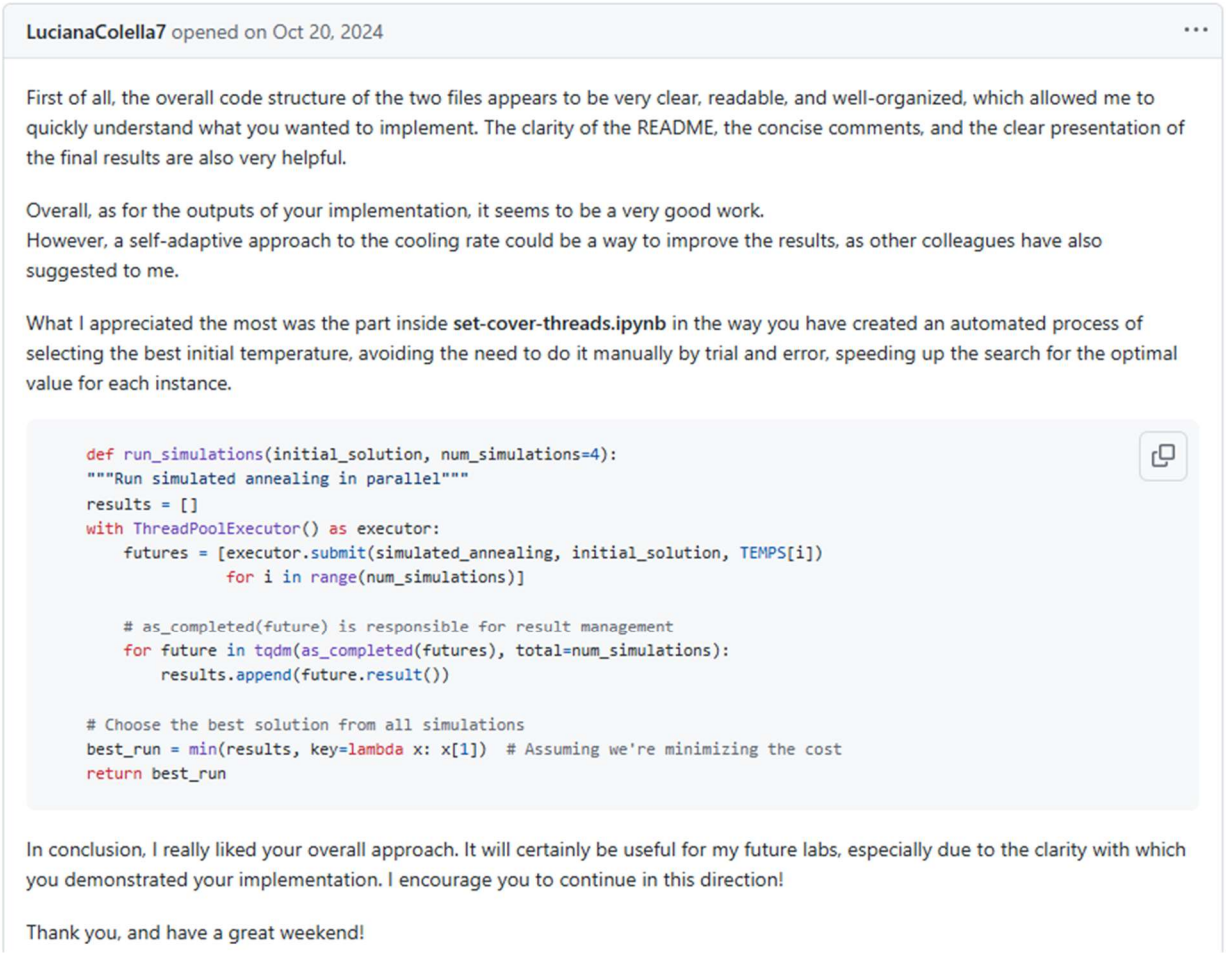
To **nomannrafiq**

- **andrefo27** opened on Oct 18, 2024

  There seems to be no solution associated with the lab1 resolution of the set cover problem in your repo.
  File **lab01.ipynb** has seven code cells empty, meanwhile file **lab1.ipynb** contains knapsack problem, and i was able to read it opening the file in a text editor since the file has some problem with notebook notation (**ipynb**).

Received Reviews

From **LucianaColella7**:



LucianaColella7 opened on Oct 20, 2024

First of all, the overall code structure of the two files appears to be very clear, readable, and well-organized, which allowed me to quickly understand what you wanted to implement. The clarity of the README, the concise comments, and the clear presentation of the final results are also very helpful.

Overall, as for the outputs of your implementation, it seems to be a very good work.
However, a self-adaptive approach to the cooling rate could be a way to improve the results, as other colleagues have also suggested to me.

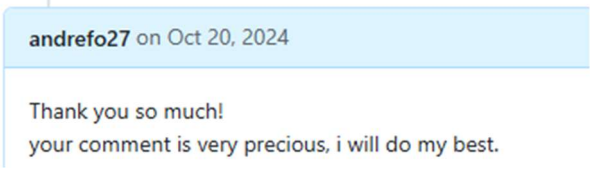
What I appreciated the most was the part inside **set-cover-threads.ipynb** in the way you have created an automated process of selecting the best initial temperature, avoiding the need to do it manually by trial and error, speeding up the search for the optimal value for each instance.

```
def run_simulations(initial_solution, num_simulations=4):
"""Run simulated annealing in parallel"""
results = []
with ThreadPoolExecutor() as executor:
    futures = [executor.submit(simulated_annealing, initial_solution, TEMPS[i])
               for i in range(num_simulations)]

    # as_completed(future) is responsible for result management
    for future in tqdm(as_completed(futures), total=num_simulations):
        results.append(future.result())

# Choose the best solution from all simulations
best_run = min(results, key=lambda x: x[1])  # Assuming we're minimizing the cost
return best_run
```

In conclusion, I really liked your overall approach. It will certainly be useful for my future labs, especially due to the clarity with which you demonstrated your implementation. I encourage you to continue in this direction!

Thank you, and have a great weekend!

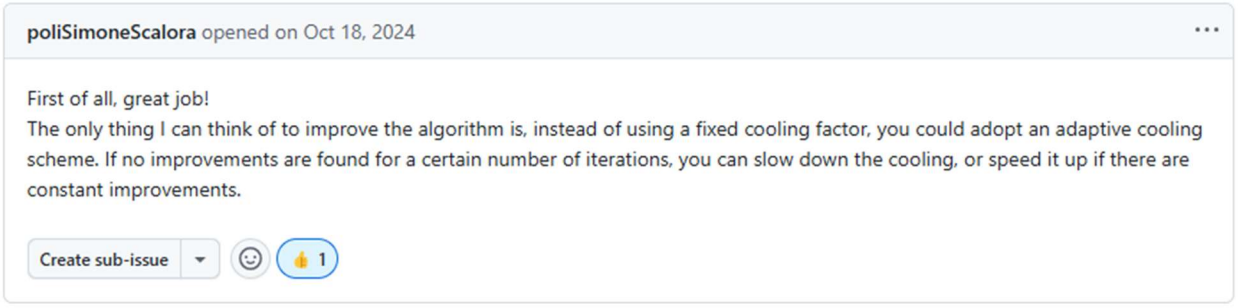
- My response:



andrefo27 on Oct 20, 2024

Thank you so much!
your comment is very precious, i will do my best.

From **poliSimoneScalora**



poliSimoneScalora opened on Oct 18, 2024

First of all, great job!
The only thing I can think of to improve the algorithm is, instead of using a fixed cooling factor, you could adopt an adaptive cooling scheme. If no improvements are found for a certain number of iterations, you can slow down the cooling, or speed it up if there are constant improvements.

Create sub-issue 1

LAB2 – Traveling Salesman Problem

You can find my implementation at: **https://github.com/andrefo27/CI2024_lab2**

The Traveling Salesman Problem (TSP) is the challenge of finding the shortest path or shortest possible route for a salesperson to take, given a starting point, a number of cities (nodes), and optionally an ending point.

A matrix stores the geographical distances ("**geodesic distance**") using the latitudes and longitudes of the respective cities. The cities under study were located in CSV files. Each row is represented by the tuple: "city_name", latitude, longitude. For this lab, we chose the first city in the CSV file as the starting and ending point of the circular route, which is where the list of study cities is defined.

The algorithm calculates a "cost" for each path, which is the total distance of the route. The cost is expressed in **kilometers**.

I do not know which approach is the fastest and most approximate or which is the slowest and most accurate.

Therefore, I implemented two solutions:

- **lab2-fast.ipynb**
- **lab2-slow.ipynb**

What I noticed, doing some experiments, is that as the generations increase, the possibility of obtaining an approximate global optimum increases, but this entails a significant increase in the search time. If the problem is large enough, the algorithm does not scale. If we try to increase the number of traits available, and therefore in this case the number of permutations, and therefore the number of individuals, this entails a significant increase in processing time.

Implementation in "lab2-fast.ipynb" – First approach

A population of N individuals is defined. An individual represents a possible sequence of cities. Each individual always starts and ends with the first city (circular route). These two "genes" can never be modified by the algorithms applied.

The **genetic algorithm** works as follows:

1. A population of N individuals and a set number of generations are defined.
2. For each generation, the following steps are performed:
    i. For each individual, its cost is calculated.
    ii. Parent selection occurs, sampled from 1/5 of the population (**tournament selection**).
    iii. The best 10 individuals (**shortest routes**), or a maximum of 10, are selected.
3. Until a number of offspring equivalent to half of the population is generated, the following steps are performed:
    i. Two parents are chosen from the set returned by the tournament selection.
    ii. A **cycle crossover** operation occurs between the parents, inheriting the consecutiveness of some cities: takes two parent paths and creates a child path by taking a segment from the first parent and filling the remaining cities from the second parent, ensuring no cities are duplicated.

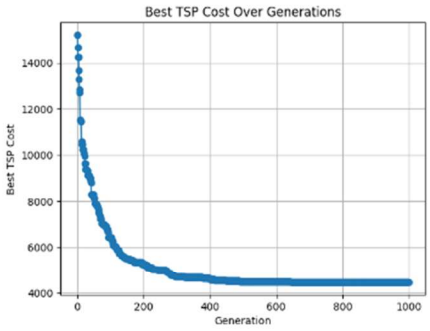
iii. A mutation algorithm is applied to the genes only if below a certain probability, which is not adaptive (**mutation_rate**). The algorithm applies one of four mutation types randomly:
    1. **Swap Mutation:** Swaps the positions of two randomly selected cities in the path.
    2. **Scramble Mutation:** Selects a random number of cities and shuffles their positions within a segment of the path.
    3. **Insert Mutation:** Removes a city and re-inserts it at a different random position.
    4. **Inversion Mutation:** Reverses the order of a random segment of the path
iv. The newly generated child is added to the offspring.

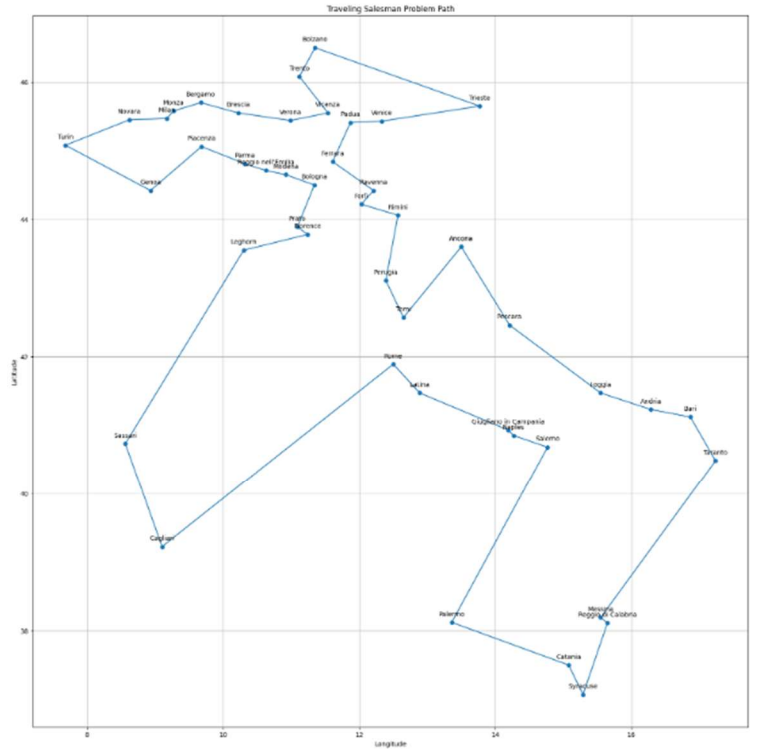
4. The offspring is added to the current population.
5. A **steady state** approach is adopted, where only the fittest (those with the lowest TSP costs) from the combined population survive; the others die.
6. Return to step 2.

For each dataset (the cities), an image is displyed, showing The best **TSP Cost** over generations and a graph that shows the best path:

Italy

Generations: 1000	Population: 150	Total Cost: 4459 (km)	Total Steps: 1000 (generations) within 150 individuals each

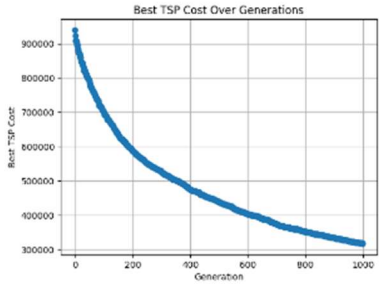




-

China:

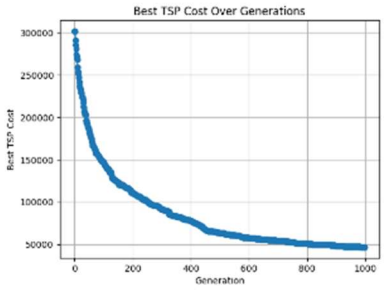
Generations: 1000	Population: 300	Total Cost: 317264 (km)	Total Steps: 1000 (generations) within 300 individuals each

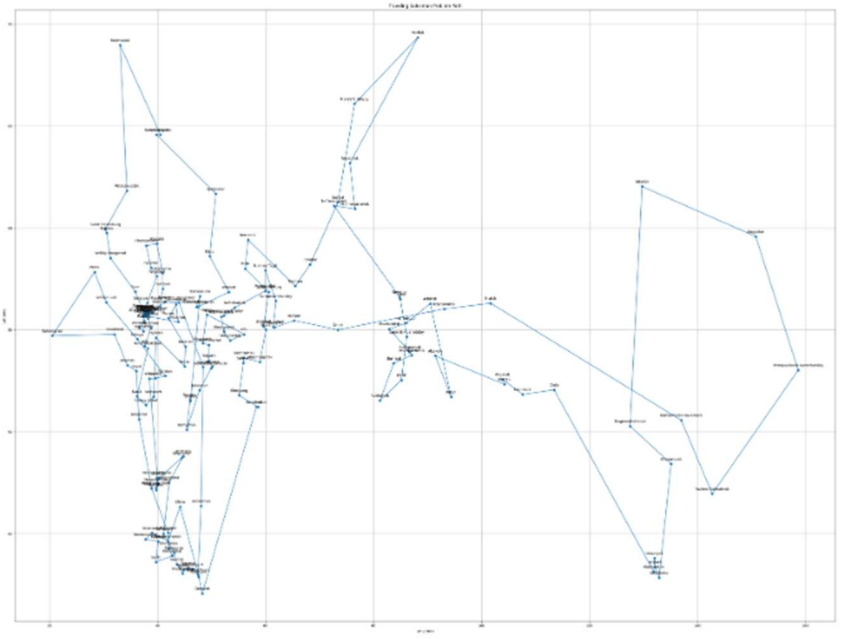


-

Russia

Generations: 1000	Population: 200	Total Cost: 46827 (km)	Total Steps: 1000 (generations) within 200 individuals each

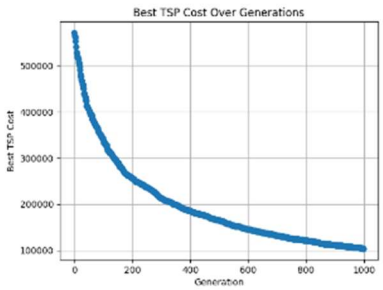


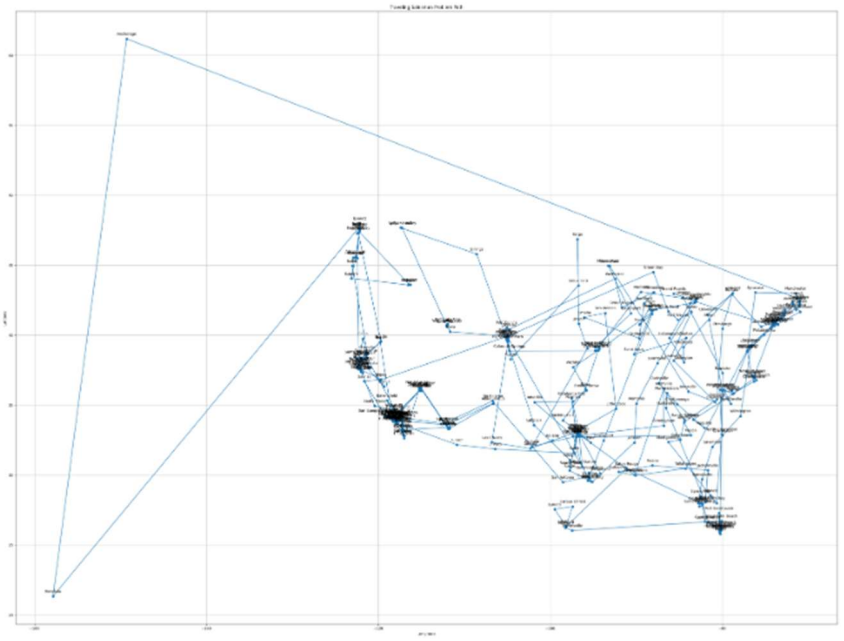


-

US:

Generations: 1000	Population: 300	Total Cost: 103613 (km)	Total Steps: 1000 (generations) within 300 individuals each

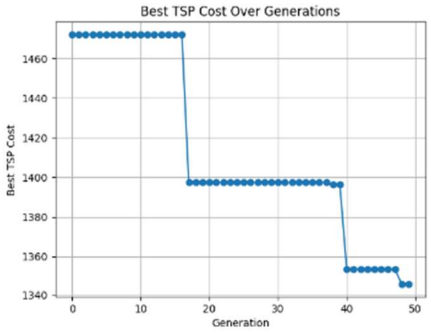


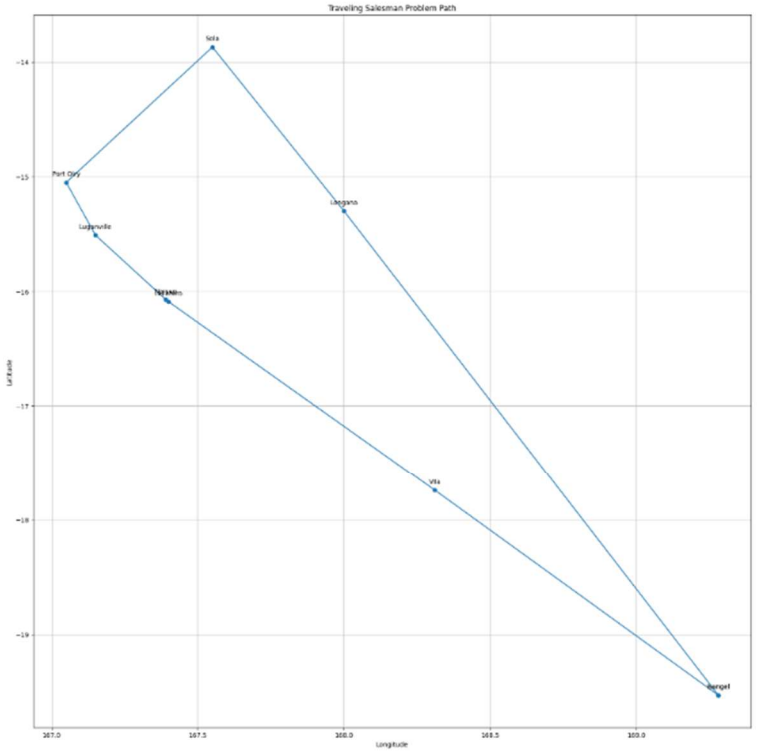


-

Vanuatu

Generations: 50	Population: 10	Total Cost: 1345 (km)	Total Steps: 50 (generations) within 10 individuals each





-

Implementation in "lab2-slow.ipynb" – Second approach

The model of the algorithm remains similar to the previous one, with some differences and the number of offsprings generated during each generation is augmented.

Generational model applied + Elitist strategy: The offspring corresponds to 150% of the population. The best 10% of individuals in the population pass to the next generation, while the rest are discarded (**Elitist strategy**). The top 60% of the offspring survive after competing for survival. The two sets are then combined to obtain the next generation. Number of individuals in each generation remains the same because, for example, if you have 100 individuals:

- 10% of 100 → 10
- Offspring = 100 * 1.5 = 150
- 60% of 150 → 90
- 10 + 90 = 100

Inver crossover has been added to the genetic algorithm. It constructs a child path by combining segments from the parents, sometimes inverting the order of the genes, which adds more diversity to the population.

A **simulated annealing** process has been implemented, with a **self-adaptive approach** to the **cooling rate**. A self-adaptive cooling rate dynamically adjusts the cooling rate based on the improvements observed in the solutions. This allows the algorithm to adapt more quickly if it detects stagnation (e.g., few improvements over several generations), while slowing down if it continues to find better solutions. This can prevent the temperature from cooling too fast, which might hinder further exploration. **Cooling rate** is always preserved between 0.8 and 1.1.

```
if current_best_cost < previous_best_cost:
    cooling_rate *= (1 - adapt_factor)  # Slow down cooling (less aggressive)
else:
    cooling_rate *= (1 + adapt_factor)  # Speed up cooling (more aggressive)
```

The mutation rate has been removed.

Instead of only accepting better solutions, the algorithm can accept a worse (higher-cost) mutation with a certain probability. This probability is based on a **temperature parameter** that decreases with each generation, making it less likely to accept worse solutions as the algorithm progresses. This approach helps the algorithm escape local minima and potentially find a better overall solution.

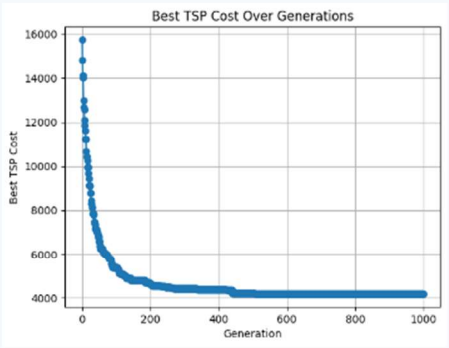
```
# Decide whether to accept the mutation
if mutated_cost < initial_cost:
    # Accept mutation if the cost is lower
    child = mutated_child
else:
    # Accept the worse solution with a probability based on temperature
    acceptance_probability = math.exp(-(mutated_cost - initial_cost) / temperature)
    if random.random() < acceptance_probability:
        child = mutated_child
```

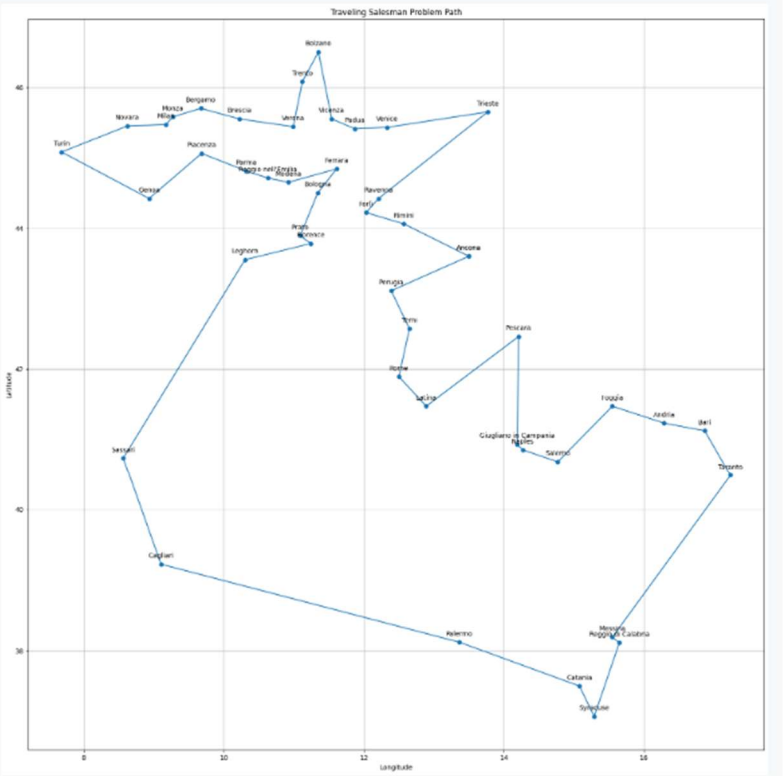
For each dataset (the cities), an image is displyed, showing The best TSP Cost over generations and a graph that shows the best path:

Italy

Generations: 1000	Population: 150	Total Cost: 4175 (km)	Total Steps: 1000 (generations) within 150 individuals each

Respect to first approach: -6.37%



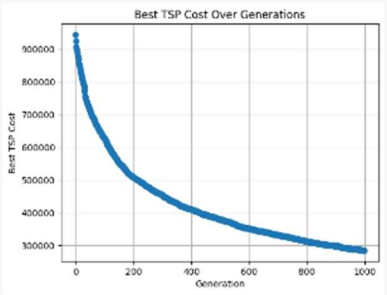


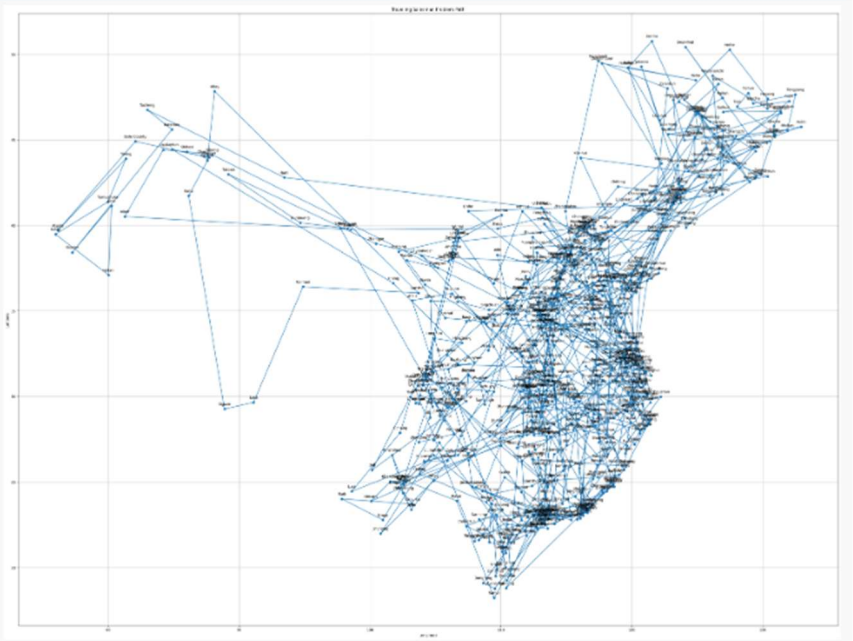
-

China

Generations: 1000	Population: 300	Total Cost: 285340 (km)	Total Steps: 1000 (generations) within 300 individuals each

Respect to first approach: -10.06% (but far slower)



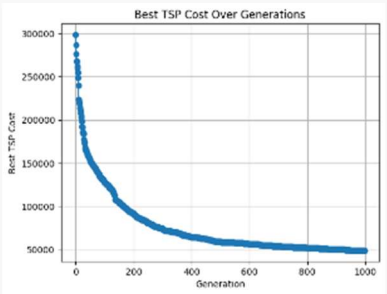


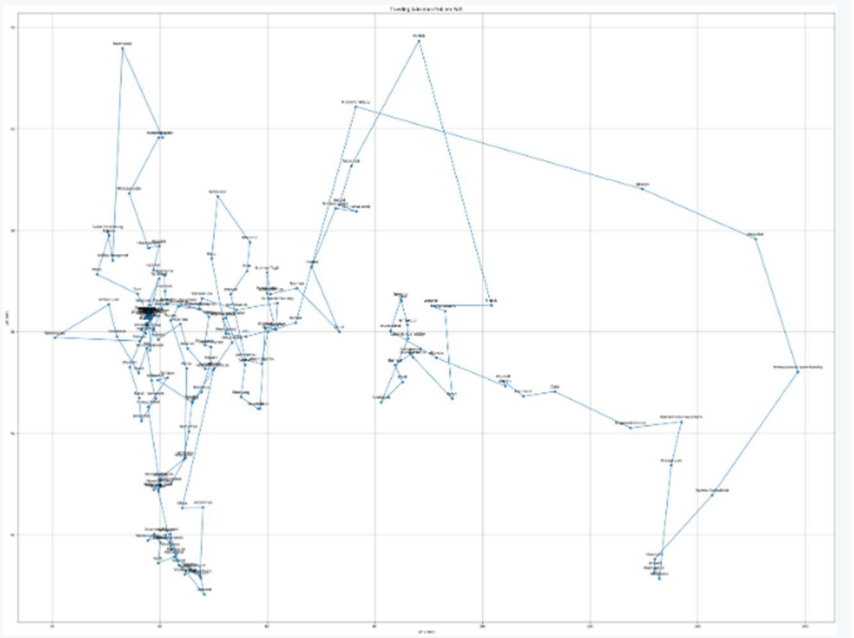
-

Russia

Generations: 1000	Population: 200	Total Cost: 49103 (km)	Total Steps: 1000 (generations) within 200 individuals each

Respect to first approach: +4.86% (worse solution here!)



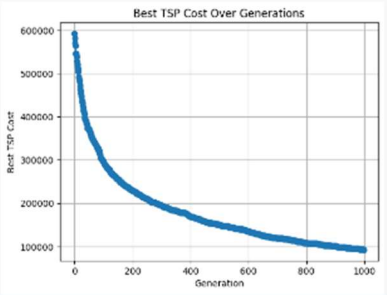


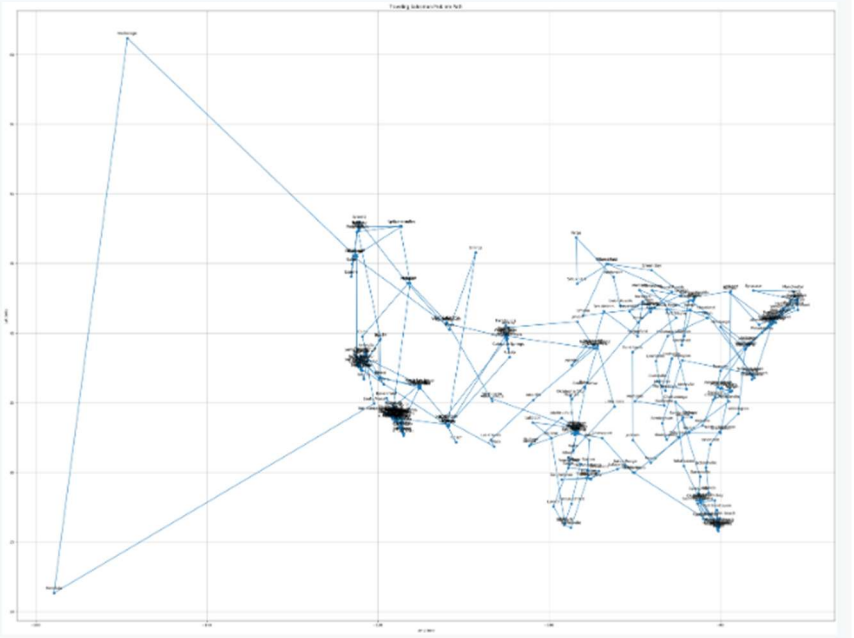
-

US

Generations: 1000	Population: 300	Total Cost: 92396 (km)	Total Steps: 1000 (generations) within 300 individuals each

Respect to first approach: -10.82% (but far slower)



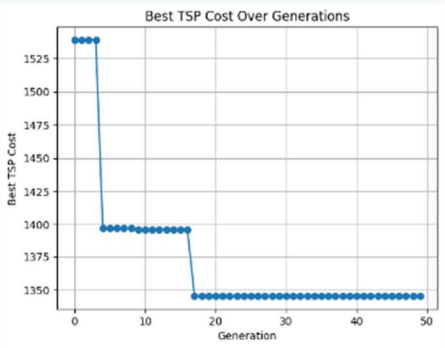


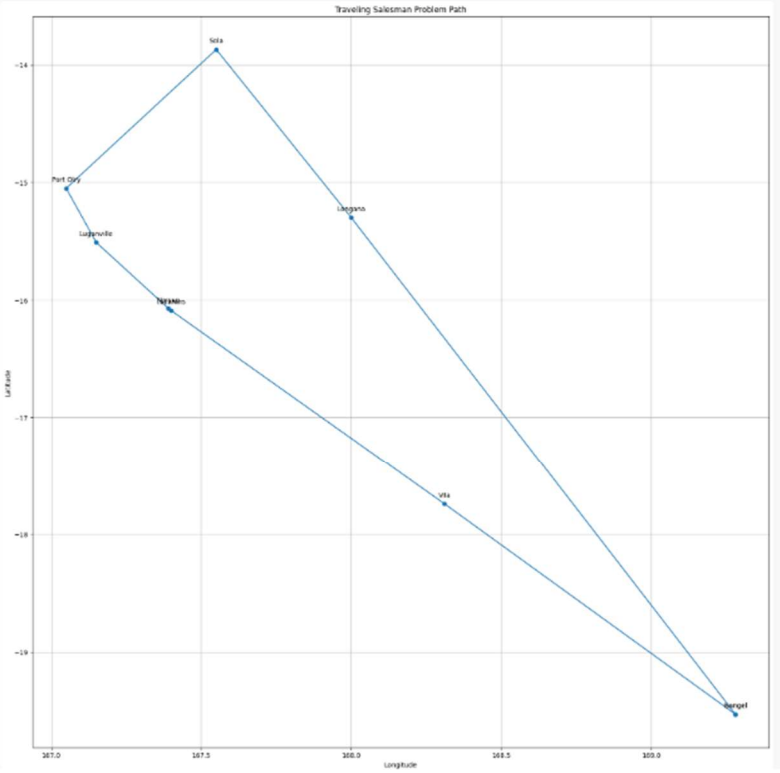
-

Vanuatu

Generations: 50	Population: 10	Total Cost: 1345 (km)	Total Steps: 50 (generations) within 10 individuals each

Respect to first approach: no changes





-

Given Reviews:

To **lorenzo709:**

andrefo27 opened on Nov 12, 2024

Initial Notes

Before starting, I'd like to clarify that `the code is simple and clear, and it accomplishes the lab task`.

Brute Force TSP Approach

This code performs a `nearest neighbor heuristic` approach to approximate a solution for the `Traveling Salesman Problem (TSP)` over a set of cities, using coordinates to calculate distances. Since it is a `greedy algorithm`, an approximate solution is acceptable, as it does not guarantee reaching the global optimum.

There is a function that is never used: `compute_centroid`.

Simulated Annealing Approach

I'm not entirely certain, but I think you should display the `current value` obtained after `N iterations`, rather than the `best value`, as the latter remains as historical data and does not necessarily match the actual value reached by the algorithm at the end of the iterations.

The `simulated annealing` implementation within the code is correct, as you apply the acceptance criterion:

```
np.random.random() < np.exp(-delta_cost / temperature)
```

when evaluating whether to accept a worse solution. Additionally, you update the `temperature` using the `cooling rate`.

However, when using temperature within the function `insert_mutation_temp`, it `affects how far a city can move`, simulating "controlled randomness." Higher temperatures allow greater flexibility, while lower temperatures restrict movement, which occurs progressively as temperature is meant to decrease over time.

Suggestions for Improving Simulated Annealing

To improve `simulated annealing`, you could consider using a `self-adaptive cooling rate` based on the improvement or deterioration in the `fitness` function. This would encourage `exploration` rather than `exploitation` as typically happens when temperature always decreases.

Alternatively, instead of having a single individual, you could work with a `population` of N individuals and apply an `evolutionary algorithm`, using either a `generational` or `steady-state` approach. In this way, you can apply `crossover functions` to increase genetic diversity. There's nothing stopping you from integrating `simulated annealing` with this solution.

To **LorenzoFormentin**

andrefo27 opened on Nov 12, 2024

General Observations

In general, the **approach** taken is **clear and simple**, and the **work has been well done**. Below are some of my observations:

Faster Approach: Tournament Selection

I approached it differently by creating a **population of permutations** that always start and end with the same city because I interpreted the problem as optimizing a **particular path** rather than finding the best route across multiple starting points after N generations.

Good **semantic integrity** has been maintained since you ensure each individual in the population (tour) always completes at the starting node. There is also good semantic integrity in the **swap mutation function** as you exclude the starting and ending "genes," ensuring the start city is not modified.

As far as I understand, by doing it this way:

```
if tsp_cost(current_best) < best_fitness:
    best_tour = current_best
    best_fitness = tsp_cost(current_best)

return best_tour, best_fitness
```

you are returning, at the end of N generations, the **best individual across all generations**, rather than the best from only the **current (last) generation**, which contains the currently "living" individuals. You could also build a record of the **current best** to show how the **evolutionary algorithm** performs generation by generation.

Slower Approach: Roulette Wheel Selection Based on Fitness

It's a good choice to ensure that each initial individual (permutation) within the population is **unique**.

It could be interesting to try multiple mutations on different tours, such as **swap mutation**, **insert mutation**, or **scramble mutation**.

Again, in this case, you are returning from the `evolutionary_algorithm` function the **best individual across all generations**, rather than the **best in the last generation**—i.e., among those individuals that remain "alive."

Received Reviews:

From **LorenzoUgo** and my response**:**

LorenzoUgo opened on Nov 16, 2024

Hi @andrefo27,
While checking your code i found nothing wrong, in fact it seemed very clean and understandable.
Good to integrate simulated annealing to maintain some degree of diversity within the population, for the slow solution.

I'm just wondering if the population size from one era to the next is maintained according to your rule: 10% original parent and 60% of the offsping.

Keep working

Create sub-issue

andrefo27 on Nov 17, 2024 Owner

The math provided should always keep the same population in each generation

let's do some simple math:
population size: 10
offspring: 10 * 1.5 = 15
elite population: 10% of 10 --> 1
survivors: 60% of 15 --> 9
new population: 1 + 9 = 10

LAB3 – N-Puzzle Problem

You can find my implementation at: https://github.com/andrefo27/CI2024_lab3

The goal is to solve efficiently a generic N^2-1 puzzle (also known as Mystic Square, Gem Puzzle, Boss Puzzle).

The main goal of the program is to find an optimal (or near-optimal) solution path from a given scrambled state of the puzzle to its solved state. The program determines this path by intelligently exploring possible moves, aiming to minimize the number of steps required using the **A* search algorithm**

The **evaluation metrics** used are:

- **Quality**: Number of actions in the solution (sequence of moves to solve the puzzle from the start).
- **Cost:** Total number of actions evaluated. (number of nodes expanded from the frontier)
- **Efficiency**: Quality vs. Cost ratio.

The **blank tile** in the puzzle is represented as **-1**.

Execution can start from:

- A **random matrix (**you can choose a fixed seed for the random generation**)**
- A **matrix defined by the user**.

Two methods for generating a **random matrix** are available:

- **Totally Random**: Uses a seed to ensure deterministic computation across different executions.
- **Controlled Randomness**: The matrix is generated from the goal state with a known number of permutations (called steps), fixed a priori. This allows evaluation of how the algorithm behaves as the complexity of the matrix increases.

Without a **solvable** initial state, the goal state **is unreachable**.

- **Solvability** is an invariable property:
    - Every legal move (swapping the empty tile with an adjacent tile) keeps the puzzle in a solvable state.
    - There is no need to recheck solvability during the execution of A*, as every new configuration generated will automatically be solvable.
- Check if a N*N - 1 puzzle is solvable or not by following below simple rules:
    - If N is **odd**, then puzzle instance is solvable if number of inversions is even in the input state.
    - If N is **even**, puzzle instance is solvable if
        - the blank is on an **even row counting from the bottom** (second-last, fourth-last, etc.) and number of inversions is odd.
        - the blank is on an **odd row counting from the bottom** (last, third-last, fifth-last, etc.) and number of inversions is even.
    - For all other cases, the puzzle instance is not solvable.
- If we assume the tiles written out in a single row (1D Array) instead of being spread in N-rows (2D Array), a pair of tiles (a, b) form an i**nversion** if “a” appears before “b” but **a > b**.

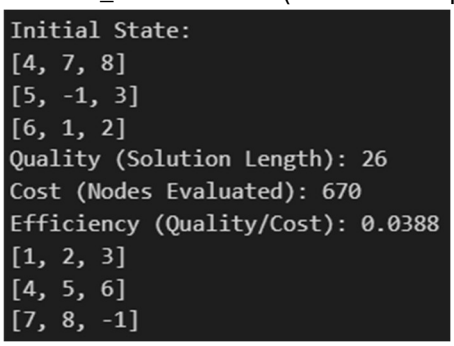
The problem has been resolved in two ways:

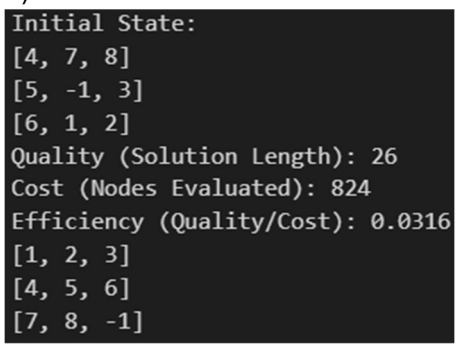
- **gem-puzzle_basic.ipynb**
- **gem-puzzle_adaptive.ipynb**

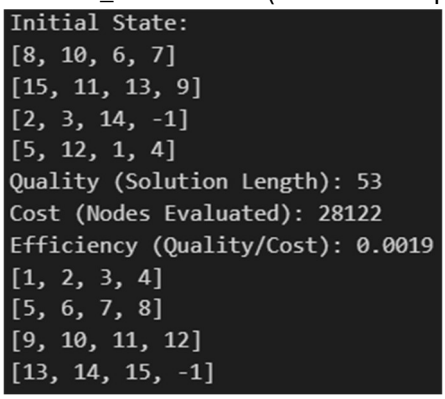
Considerations:

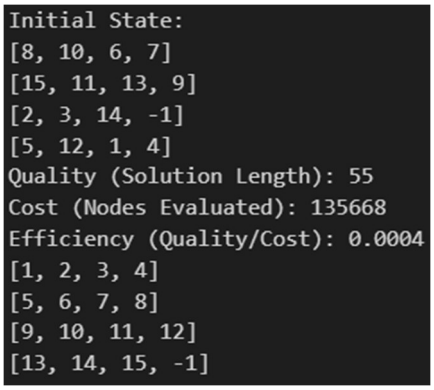
- I am aware that the implemented algorithm does not perform efficiently as the size of N increases. However, it achieves good results, as far as i tried, with **4x4 matrices** generated with maximum 1000 steps. With **5x5 matrices**, results can be obtained in reasonable times (less than a minute), but that is not guaranteed. To achieve that i suggest to not go over 200 random steps. Excessive steps may cause the algorithm to take unreasonable amounts of time and consume excessive memory, slowing down or even halting execution. The most important thing i noticed is that the result is some kind related to the **initial configuration of the matrix**. So it really matters how you build it at the starting point of the algorithm and affect the times the algorthm takes to reach the goal state.
- Be cautious of **RAM consumption**, as all explored nodes are stored in a set called "**visited**". This can lead to high memory usage.

Some examples compares the two methods:

- **N=3, RANDOM_STEPS=1000** (Basic vs Adaptive)
  - 

vs


- **N=4, RANDOM_STEPS=500** (Basic vs Adaptive)
  - 

vs


Implementation in "gem-puzzle_basic.ipynb"

The program is structured into:

- **Heuristics**: A* is an informed search algorithm, meaning it uses a **heuristic function** to guide its search. The program defines two heuristics:
  - **Manhattan Distance**: This heuristic calculates the sum of the horizontal and vertical distances of each tile from its goal position.

    - **Linear Conflict**: This heuristic identifies pairs of tiles that are in their correct row or column but are in the wrong order relative to each other, then at least one extra move is needed. It adds a penalty to the Manhattan distance for each such conflict, making the heuristic more accurate.
- **A* Search Algorithm**: The **a_star_optimized** function implements the core A* logic. It uses a **priority queue** (implemented with **heapq**) to manage **the frontier** of states to visit. The priority of each state is determined by **f(n) = g(n) + h(n)**, where:
    - **g(n)** is the **cost** to reach the current state from the start state (the number of moves).
    - **h(n**) is the **heuristic** value, estimating the cost from the current state to the goal state.
- **frontier**: priority queue (**min-heap**) of states to explore. Each **element = (f, state, g, parent)**:
    - **f = g + h** (cost-so-far + heuristic).
    - **state**: current board
    - **g**: depth (number of moves from start).
    - **parent**: reference to the parent state (for reconstructing the solution).
- **Solvability Check**: The **is_solvable** function determines if a given puzzle configuration can be solved. It does this by calculating the number of inversions and considering the position of the blank tile. This is a crucial step because some initial configurations of the N-Puzzle are impossible to solve.
- **State Generation**: The program includes functions (**generate_random_seed** and **generate_random_steps**) to create solvable, scrambled puzzle states to test the algorithm. This ensures that the A* algorithm is always given a problem with a **valid solution**.

Algorithm

- **Initialization**: The program sets up the size of the puzzle (N), defines the goal state, and generates a random, solvable start state.
- **A* Execution**:
    - It starts with the initial state and adds it to the **priority queue** with its calculated f(n) value.
    - It repeatedly extracts the state with the **lowest f(n)** from the queue (**min-heap**).
    - If the extracted state is the goal state, the algorithm has found a solution. It then reconstructs the path from the start to the goal by backtracking through parent pointers.
    - If the state isn't the goal, it generates all possible next states (by moving the blank tile up, down, left, or right).
    - For each new state, it calculates the new g(n) and h(n) and adds the state to the priority queue if it hasn't been visited before.
- **Path Reconstruction**: Once the goal is reached, the program traces back from the goal node to the start node to reconstruct the path, which represents the sequence of moves to solve the puzzle. It then prints the length of this path (**quality**), the number of nodes evaluated (**cost**), and the efficiency of the search.

```python
# Goal test
if current == goal:
    path = []
    state = current_tuple
    while state is not None:
        path.append([list(row) for row in state])
        state = state_to_parent[state]
    path.reverse()

    quality = len(path) - 1  # Number of actions to solve
    cost = total_actions_evaluated  # Total nodes expanded
    efficiency = quality / cost if cost != 0 else float('inf')

    print(f"Quality (Solution Length): {quality}")
    print(f"Cost (Nodes Evaluated): {cost}")
    print(f"Efficiency (Quality/Cost): {efficiency:.4f}")
    return path
```

Implementation in “gem-puzzle_adaptive.ipynb”

This adaptive version of the N-Puzzle solver differs from the basic version in its **A* search algorithm**, specifically in how it calculates the heuristic value. The basic version uses a single, fixed heuristic (Manhattan Distance plus Linear Conflict), while the adaptive version uses a combination of heuristics to balance **exploration** and **exploitation.**

The gem-puzzle_adaptive program introduces a new heuristic called **combined_heuristic**. This function randomly chooses between two different heuristic functions based on a "**magic number**" probability:

- **misplaced_tiles**: leverage **exploration**
- **linear_conflict_manhattan**: leverage **exploitation (**it combines manhattan distance + linear conflict**)**

The **magic_number** determines the probability of switching **heuristics** between **exploration** and **exploitation**:

- **Exploration**: Uses the **misplaced_tiles** heuristic, favoring new states even if they might not seem immediately optimal.
- **Exploitation**: Uses the **linear_conflict_manhattan** heuristic, which is more informed and typically leads toward the goal.

Adaptive Behavior: The **magic_number** starts at a high value (1.125 * n**(n-1)), favoring **exploration** initially. On each iteration, it decays by a factor (**adaptive_factor** = 0.985) until it reaches the **threshold** (0.0785). This gradual decay shifts the algorithm's behavior:

- **Early search**: Encourages broader exploration to escape local minima or find diverse paths.
- **Late search**: Focuses more on exploitation to efficiently converge to the goal.

Given Reviews:

To **AliEdrisabadi**

andrefo27 opened on Nov 29, 2024

Overview

The puzzle is solved by applying a sequence of valid moves starting from a randomized initial state. The sequence of updates after each move is displayed to help understand the problem-solving process.

The heuristic used is **admissible** and guarantees the optimality of the solution. This means the A* algorithm will find the shortest possible path to the solution.

The puzzle is randomized using a sequence of valid moves applied to the initial goal state, ensuring the puzzle is solvable. The number of random steps taken (`RANDOMIZE_STEPS`) can be adjusted to generate a sufficiently scrambled puzzle.

Optimizing Manhattan Distance Calculation

The Manhattan Distance is recalculated from scratch for every tile in every state. To optimize this, you can **precompute the goal positions** for each tile as follows:

```
goal_positions = {value: divmod(value - 1, PUZZLE_DIM) for value in range(1, PUZZLE_DIM**2)}
```

Then, for each state, you calculate the Manhattan Distance using the precomputed goal positions:

```
total_distance = 0
for i in range(PUZZLE_DIM):
    for j in range(PUZZLE_DIM):
        value = state[i, j]
        if value != 0:
            goal_x, goal_y = goal_positions[value]
            total_distance += abs(i - goal_x) + abs(j - goal_y)
```

By precomputing the goal positions, you avoid recalculating the goal coordinates for each tile every time.

Puzzle Scaling and Performance

As the puzzle size increases (e.g., from a 4x4 to a 7x7 grid), the number of moves required to randomize the puzzle (`RANDOMIZE_STEPS`) should scale linearly. However, as the grid size increases, the computational effort to find a solution increases as well. Even for a **5x5** puzzle with over **200 randomized steps**, the problem becomes increasingly difficult and time-consuming. This scaling behavior is typical, as the search space grows exponentially with the size of the puzzle, and, as far i understood, it really depends from the initial configuration of the matrix.

For matrices larger than **4x4**, combining multiple heuristics can significantly improve performance. One potential heuristic is **Linear Conflict**. This method identifies pairs of tiles that are in the correct row or column but are reversed in their order, thus improving the heuristic's accuracy. Try to combine it with the heuristic you have already defined.

Another alternative heuristic is the **Misplaced Tiles** heuristic, which simply counts how many tiles are not in their correct position. This heuristic is less efficient than Manhattan Distance but can still be useful in some cases.

Conclusion

The solution is clear. The modular structure and correct use of the Manhattan Distance heuristic ensure that the A* algorithm finds the optimal solution. Additionally, the implementation displays each step of the solution for better visualization, which helps in understanding the algorithm's decision-making process.

To **SaraaFrancee**

andrefo27 opened on Nov 29, 2024

Overview

BFS is faster in exploring all possible states systematically, though it might lead to more state exploration. UCS introduces heuristics to reduce exploration by prioritizing more favorable moves.

The `GOAL` state is generated randomly, rather than being a standard solved state (e.g., `1, 2, ..., n^2 - 1`). This introduces variability in solvability, as not all randomized matrices are guaranteed to be solvable. A function to validate solvability should be included.

Uniform-Cost Search (UCS):

- This algorithm explores fewer states than BFS, making it more efficient for larger puzzles. However, it sacrifices accuracy, as it may lead to solutions with more steps.
- The inclusion of "heuristics" (the three of priorities you have defined) makes UCS faster but less exhaustive compared to BFS.

Heuristic Refinement:
- In the UCS algorithm, consider swapping the order of `state_moves` and the tuple of the new state in the priority queue insertion. This might improve performance by prioritizing states with fewer moves.

Unused Function:
- The `min_move` function is defined but not used in the current implementation. If it is not required, it should be removed to avoid confusion.

Performance Tuning:
- In UCS, the computation of `common_in_matrices(state, GOAL)` is redundant within the loop, as `state` does not change. Moving it outside the loop would reduce unnecessary calculations.

Observations on Behavior

- The comparison of BFS and UCS reveals that while UCS often finds solutions with fewer total trials, the solutions themselves may include a larger number of moves. This suggests that UCS prioritizes efficient exploration but may overlook optimal solutions in terms of steps.

Conclusion

The structure of the code is generally well-organized. Some results with higher-order matrices (4x4, 5x5) would have been helpful. An approach commonly used to solve the N-Puzzle problem is implementing the A* algorithm with different heuristics to explore the solution space and determine which solution is the most acceptable.

Received Reviews:

From **CappePolito**

CappePolito opened on Nov 30, 2024

Hi, i really appreciated your code, it's very detailed and shows you tried a lot of algorithms and adaptive techniques. I focused mainly on the adaptive one as it's more advanced in this review.

I see you used combined heuristics, with Manhattan distance and linear conflict, and it's very interesting the use of the adaptive use of the magic_number to occasionally favor the less efficient tiles to encourage exploration and reduce the likelihood of getting stuck in a local minimal.

You also used A*, and the use of the decaying magic_number to prioritize exploration early in the search and shift later toward exploitation. I think this is what makes your code more efficient and i think it's a really good touch.

I see you also tried with heapq and states to try to minimize the ram used (i found that storing the puzzles as a number and computing an hash used even less ram, but it's a good way to improve the ram usage, but this wasnt the point of the lab XD)

Overall it's a really strong implementation and i think you did a really good job!

Symbolic Regression

The course project aims to find a mathematical expression that best fits a given dataset.

You can find the entire project structure at: https://github.com/andrefo27/CI2024_project-work

Structure of the **GitHub repo**:

- **data**: Contains the eight datasets used for the mathematical approximation.
- **src**: contains the **algorithm.ipynb** where the entire logic to resolve symbolic regression has been defined
- **s324581.py**: File that collects the selected formulas found for each problem. These are the functions obtained executing the GP algorithm with 100% of the entire dataset.
- **test**:
    - It contains all the evaluation formulas and their results for each dataset. You can experiment with the evaluations using the `**evaluation.py**` file.
    - It contains the `**s324581_datasplit.py**` with the functions obtained executing the GP algorithm with 60% of the entire dataset. For the final evaluation, you should use also this file to verifiy how the generalization works on the broader dataset
- **report_CI2024_project-work_s324581.pdf**: The file you are reading right now.

Main workflow of the project:

- **[1] The Experimental Objective**
    - The primary goal is to find a compact and accurate mathematical expression (or "model") that describes the relationship between a set of input variables (x) and an output variable (y). This is achieved by evolving a population of candidate expressions over multiple generations, treating each expression as a "chromosome" in a genetic algorithm. The experiment uses a **split-dataset approach**, where the expressions are evolved on a **training set** to prevent overfitting, and their final performance is evaluated on a separate, unseen **validation set**.
- **[2] Representing and Evolving the "Solutions"**
    - The program's core experiment lies in how it represents and manipulates potential solutions.
        - **Expression Trees:** Expressions are represented as **tree data structures** using a **Node** class. Internal nodes are mathematical functions (e.g., +, sin), while leaf nodes are variables (x0, x1, ...) or constants (from -1 to +1).
        - **Genetic Operators:** The algorithm experiments with various **genetic operators** to drive the evolutionary process. This is where a significant part of the experimentation takes place:
            - **Subtree Crossover:** Where sub-expressions (subtrees) are swapped between two parent expressions. This is the primary mechanism for combining desirable **traits** from different parents.
            - **Mutation:** The program experiments with five distinct mutation types to introduce diversity and prevent the population from stagnating:
                - **Subtree Mutation:** Replaces a random subtree with a new, randomly generated one.
                - **Point Mutation:** Swaps one node with another of the same type (e.g., + with -).
                - **Hoist Mutation:** Replaces a tree with one of its own subtrees, a powerful way to simplify solutions.

    - **Expansion Mutation:** A terminal node is replaced by a new function node and its required children. This adds complexity and grows the expression.
    - **Collapse Mutation:** A function node is replaced with one of its children (a terminal or another subtree). This reduces complexity.

- **[3] The Fitness Function: Defining "Success"**
    - The fitness function is the most critical experimental design choice. It determines which solutions are considered "good" and thus have a higher chance of reproducing. The **calculate_fitness** function uses a composite score to evaluate a solution, reflecting a **multi-objective optimization** approach. The components are:
        - **Mean Squared Error (MSE):** The primary metric, measuring prediction accuracy on the training data. A lower MSE is better.
        - **Size Penalty:** A penalty for overly large expressions, discouraging overly complex solutions and favoring **parsimony** (simpler, more elegant solutions).
        - **Unbalanced Variable Penalty:** Penalizes expressions that don't make sufficient use of all input variables.
        - **Ratio Penalty:** A penalty for a high **constant-to-variable ratio**, encouraging solutions that learn from the data structure itself rather than relying on hard-coded constants.
- **[4] The Main Experimental Loop**
    - The **genetic_programming** function coordinates the entire experiment.
        - **Initial Population Generation:** The experiment starts with a diverse population of random expressions.
        - **Iterative Evolution:** The program enters a loop where it iteratively evaluates, selects, and reproduces expressions for a fixed number of generations or until a sufficient level of fitness is reached.
        - **Elitism and Immigration:** To ensure progress, the experiment uses **elitism**, carrying over the best individuals to the next generation without modification. It also includes an "i**mmigration**" strategy to inject new, random expressions into the population if the fitness score plateaus, experimentally preventing the algorithm from getting stuck in a local optimum.
        - **Final Evaluation:** After the main loop, the final "best" expression is evaluated on the validation set. This final check is crucial for confirming that the solution generalizes well and is not simply overfitted to the training data.

Initial Configuration and Data Handling

Data Loading: It loads a NumPy archive file (problem_”x”.npz) which contains the input features (x) and the target values (y). All the eight dataset are under the folder “data” in the Github repo.

Data Split: The code uses a variable TRAIN_SIZE to split the data into a **training set** and a **validation set**. The training set is used for the evolutionary process, while the validation set is reserved to test the final solution's ability to generalize to unseen data, preventing **overfitting**.

Expression Representation: The Node Class

Node Class: The Node class is the building block of the expression tree. Each node can be a **function** (e.g., +, sin) or a **terminal** (a variable or a constant).

- **Function Nodes:** Have children that are themselves Node objects. The number of children is determined by the function's **arity** (number of arguments). For example, + is a binary function (arity 2), while sin is a unary function (arity 1).
- **Terminal Nodes:** Represent the input variables or numerical constants. These are the leaves of the tree and have no children.

evaluate Node's method: This recursive method traverses the tree to calculate the final numerical value of the expression. It applies the function at each node to the evaluated results of its children, working from the bottom up.

FUNCTION_METADATA: This dictionary defines the available functions, their arity, and the corresponding Python function.

```
FUNCTION_METADATA = {
    '+': {'arity': 2, 'func': lambda x, y: x + y},
    '-': {'arity': 2, 'func': lambda x, y: x - y},
    '*': {'arity': 2, 'func': lambda x, y: x * y},
    '/': {'arity': 2, 'func': safe_div},

    'sin': {'arity': 1, 'func': lambda x: np.sin(x)},
    'cos': {'arity': 1, 'func': lambda x: np.cos(x)},
    'tan': {'arity': 1, 'func': safe_tan},
    'cot': {'arity': 1, 'func': safe_cot},

    'sinh': {'arity': 1, 'func': lambda x: np.sinh(np.clip(x, -200, 200))},
    'cosh': {'arity': 1, 'func': lambda x: np.cosh(np.clip(x, -200, 200))},
    'tanh': {'arity': 1, 'func': lambda x: np.tanh(x)},

    'sqrt': {'arity': 1, 'func': safe_sqrt},
    'sqr': {'arity': 1, 'func': lambda x: np.power(np.clip(x, -1e+50, 1e+50), 2)},
    'cube': {'arity': 1, 'func': lambda x: np.power(np.clip(x, -1e+20, 1e+20), 3)},

    'ln': {'arity': 1, 'func': safe_ln},
    'exp': {'arity': 1, 'func': lambda x: np.exp(np.clip(x, -100, 350))},

    'arctan': {'arity': 1, 'func': lambda x: np.arctan(x)},
    'arccot': {'arity': 1, 'func': safe_arccot}
}
```

- Functions like: division, tan, cot, sqrt, ln, arccot; are wrapped in special modules to better express the evaluation of the tree. For example: the division is defined if the denominator is different from zero, otherwise there is ambiguity. To resolve this, we don't accept the evaluation, infact the division returns a NaN value, indicating that the actual prediction will give a NaN result and so a NaN fitness, indicating that the tree is not valid, and not able to fit in the population. → An important measure taken is to ensure that every tree in the population has always a valid fitness and so a valid evaluation (not a NaN).
- Special wrapper modules defined in the code:

```python
def safe_div(x, y):
    if np.isclose(y, 0):
        return np.nan
    return x / y

def safe_tan(x):
    if np.isclose(np.cos(x), 0):
        return np.nan
    return np.tan(x)

def safe_cot(x):
    if np.isclose(np.sin(x), 0):
        return np.nan
    return np.cos(x) / np.sin(x)

def safe_sqrt(x):
    if x < 0:
        return np.nan
    return np.sqrt(x)

def safe_ln(x):
    if x <= 0:
        return np.nan
    return np.log(x)

def safe_arccot(x):
    if np.isclose(x, 0):
        return np.pi / 2
    return np.arctan(1 / x)
```

Node class has also the following methods:

- **is_leaf** → determines if the "self" Node is a leaf
- **get_height** → return the depth of the tree
- **get_size** → return the total number of the nodes of the tree
- **get_all_subtrees** → return a List of all the Nodes (class Node) of the tree
- **find_parent_of** → returns the node whose child is defined in the input parameter
- **copy** → return a tree identical to the self one called

Fitness Evaluation: The calculate_fitness Function

Before talking about fitness evaluation, there is an useful method called "**count_constants_and_variables**", that does the following:

- it walks through the tree and counts how many **constants** and **variables** are in it, including how often each variable appears.
- It returns a dictionary like this:

```
{
  'constants': <number of constants>,
  'variables': <number of variables>,
  'variable_freq': {'x0': count, 'x1': count, ...}
}
```

  - It will be used by the "**calculate_fitness**" function

The fitness function "**calculate_fitness**" determines how "good" an expression is. A lower score indicates a better-performing individual. The function returns a tuple: (**total_fitness, mse**).

- **Primary Metric (MSE):** The base of the fitness score is the **Mean Squared Error (MSE)** on the training data. This is a standard metric for regression that penalizes large errors.
  - $$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$
    - n → number of samples
    - y_i → true target value for sample i
    - $\hat{y}_i$ → predicted value for sample i
- **Regularization Penalties:** To find not just accurate but also robust and simple solutions, the function adds several penalties:
  - **Size Penalty:** A penalty for expressions that exceed a certain complexity threshold. This discourages large, "**bloated**" expressions and promotes **parsimony**.
  - **Unbalanced Variable Penalty:** Penalizes solutions that don't make balanced use of all available input variables (x0, x1, etc.). This encourages the algorithm to find solutions that utilize all the provided information.
  - **Ratio Penalty:** A penalty for expressions with a high ratio of constant terminals to variable terminals. This is an experimental choice to favor solutions that rely more on the structure and interaction of input variables than on specific hard-coded numerical values.

The "**total_fitness**" returned by the function is made by the Primary Metric MSE + Regularization Penalties.

Population Generation

Before talking about population generation methods, there are two useful method to mention:

- **prune_tree**
  - it **prunes (limits) the depth of a symbolic expression tree** so it doesn't grow deeper than a given max_depth
  - If the current node is **deeper than the allowed depth**, the node is turned into a **leaf** (either from the terminal set or a random costant between -1 and +1)
  - If the depth limit is **not yet reached**, it **recursively prunes** the left and right children.
- **simplify_tree**
  - it **reduces the expression tree to a simpler equivalent form** by computing constant parts and applying basic algebraic identities.
  - **Unary operator simplification** – if the node is a unary operator (e.g., sin, cos, exp) applied to a constant, it directly computes the result and replaces the subtree with that constant.
  - **Binary operator simplification** – if both children are constants, it evaluates the operation (+, -, *, /) and replaces the subtree with the numeric result.
  - **Algebraic rules** (common simplifications):
    - A + A → 2 * A
    - A * 1 → A or 1 * A → A
    - A / 1 → A
    - A - A → 0
    - A + 0 → A or 0 + A → A
    - A * 0 → 0 or A * 0 → 0

The tree population is created by mixing **grow** and **full** methods for the tree generation.

The "**generate_population**" function create a subsets of the trees from the "**grow"** algorithm and the other one from the "**full**" algorithm.

- **Grow method**
  - Builds a tree **with variable depth ≤ max_depth**.
    - At the last level, it creates a **terminal node** (variable or constant).
    - Otherwise, it randomly chooses a function from the function set, a costant or a variable
  - Functions expand recursively into subtrees, while terminals end the branch.
- **Full method**
  - Builds a tree **with exact depth = max_depth**.
  - At the last level → only terminal nodes.
  - Otherwise → always chooses a function node from the function set, forcing full depth expansion.

The fitness of the tree being generated is evaluated on training data. The tree is not accept until a valid fitness is found. So if the fitness is not valid, the current tree is discarded and a new one will be generated.

```
if not np.isnan(fitness) and not np.isinf(fitness):
    return simplified_tree
```

The process of tree generation until its fitness is valid is encapsulated in wrapper methods: "**grow_and_simplify**" and "**full_and_simplify**". This is the logic followed:

- Generates a tree using **grow/full** method.
- Simplifies it (**simplify_tree**). There is no need to prune the tree as it is always with required depth.
- Evaluates fitness on training data.
- Repeats until a valid tree (finite fitness) is found.

Genetic Operators

These are the mechanisms that generate new candidate solutions from the current population.

The method that applies the crossover is called "**subtree_crossover**" and takes two trees and returns two new trees. It is important to apply a "**prune_tree"** first, after the crossover, to each of the children, because they might have exceeded the max depth required. Then a "**simplify_tree**" operation is applied to each children pruned.

The "**mutate**" function decide randomly one of the five possible mutation algorithms:

- "**subtree_mutation**" → the new subtree is generated by the function "**grow_and_simplify**", then the result individual (combined with the original tree) is first **pruned** and then **simplified**
- "**point_mutation**" → the resulting individual is then **simplified**
- "**hoist_mutation**" → the resulting individual is then **simplified**
- "**expansion_mutation**"→ the result individual is first **pruned** and then **simplified**
- "**collapse_mutation**" → the resulting individual is then **simplified**

Main Algorithm: The genetic_programming Loop

The **genetic_programming** loop orchestrates the entire evolutionary process.

Creates the initial population (mix of grow/full trees, simplified, and fitness-validated).

Keeps track of the best tree found so far (**best_individual_ever**) and its fitness. Also counts how many generations pass without improvement (**no_improvement_count**).

Each tree's fitness is calculated: the first fitness (**MSE** + **Regularization Penalties**) is used for selection process, the second fitness is the actual **MSE**.

Population is sorted by fitness (best → worst). Then it updates global best individual if current gen has an improvement.

Stops early if a very good solution is found.

For the **parent selection** an **overselection strategy** has been used: two groups are created. the first group is composed of the best individuals, that is, those with the lowest fitness. The second group is composed of the remaining individuals of the current population.

```
overselection_group1 = sorted_population[:elite_size]    # Top elite individuals
overselection_group2 = sorted_population[elite_size:]    # Rest of population
```

Then:

- 80% chance → select parent from elites.
- 20% chance → select parent from the rest.

If stuck too long without improvement:

- Keep the N best individuals (N=15, **5%** of the **overselection_group1**).
- Replace the rest with newly generated random individuals → **restart-like mechanism**

Survivor Selection (Elitism): Keep a small portion of the best trees directly in the next generation. I chose to keep the 8% of the best individuals in group 1, which in fact constitute the true elite of the population.

```
retained_size = int(len(overselection_group1) * 0.08)
elite_group = overselection_group1[:retained_size]
new_population = [ind.copy() for ind in elite_group]
```

The rest of the new population is created by applying **crossover** and **mutation**, selecting parents using the **overselection approach**. The following steps are done sequentially in a loop until the new population size has been reached (excluding new i**mmigrant** populations).

- Choose two parents with the **parent selection**
- Perform **crossover** if below a certain probability (**xover_rate**). The resulting children are **pruned**. Otherwise, if crossover has not been applied, the new children become the selected parents.
- Then, perform **mutation** if below a certain probability (**mut_rate**), first on the first child and then on the second one.

```
if random.random() < mut_rate:
    child1 = mutate(child1, max_depth)
elif random.random() < mut_rate:
    child2 = mutate(child2, max_depth)
```

- whether we used crossover or mutation we perform a **simplification** on the children

- Perform **fitness evaluation** for each child. Add it to the new population only if its fitness is valid

Then add a new random population (**immigrants**) to the current one being generated to mantain diversity.

Forced Mutation: Every 25 generations, mutate elite individuals even if they are good → avoid stagnation in local optima.

Replace old population with the new one and repeat until:

- Max generations reached, or
- Early stop condition met.

The best solution discovered throughout the entire process is then returned and evaluated on the validation set.

During the final evaluation, on the training and validation, $\mathbf{R^2}$ has been used.

- The **R-squared** measures the proportion of the variance of the dependent variable explained by the model:

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

-
  - $SS_{\text{res}} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ → sum of squared residues
  - $SS_{\text{tot}} = \sum_{i=1}^{n}(y_i - \bar{y})^2$ → total sum of squares
  - y_i → true value
  - $\hat{y}_i$ → predicted value
  - $\bar{y}$ → average of real values
- $\mathbf{R^2 = 1}$ → The model perfectly explains 100% of the variance in the dependent variable. Perfect fit.
- $\mathbf{0 < R^2 < 1}$ → The model explains a fraction of the variance. The closer to 1, the better the fit.
- $\mathbf{R^2 = 0}$ → The model does not explain any variance; predictions are no better than using the mean of the data.
- $\mathbf{R^2 < 0}$ → The model performs worse than simply predicting the mean; the predictions increase the error. (This is the case when predictions are always costants and true values are not.)

Results and Considerations

Explanation of the choice of the datasplit separation:

- When training a genetic programming model (or any machine learning model), it is important to **separate the dataset into training and validation sets** instead of training only on the entire dataset.
  - **Training set** → Used to evolve the trees and optimize their parameters (fitness is computed here).
  - **Validation set** → Used to evaluate how well the evolved expressions generalize to unseen data.

- If we were to train only on the full dataset, the algorithm might simply "memorize" the data points, producing expressions that fit the training data extremely well but fail to capture the underlying functional relationship. This problem is known as **overfitting**.

In short: splitting into **training + validation** ensures the GP model learns **general rules** instead of memorizing the given dataset.

The following are the different results obtained by my algorithm on the data under examination using the datasplit approach (**train + validation**).

Problem 1 - DataSplit

Parameter	**Value**
Tree depth	3
Best expression	sin(x0)
MSE (training set)	0.00000
R^2 (training set)	1.00000
MSE (validation set)	0.00000
R^2 (validation set)	1.00000

Problem 2 - Datasplit

Evaluation 1:

Metric	**Value**
Tree depth	7
Best expression	((x0 * (1166569.2224039633 - ((407211.57237809076 * (x1 + x0)) * ((x2 + x0) * 0.1751262493989698)))) + (((407211.57237809076 * (2 * x1)) + (407211.57237809076 * (x2 + (2 * x0)))) + (407211.57237809076 * x2)))
MSE (Training)	4,622,619,768,871.53516
R^2 (Training)	0.84239
MSE (Validation)	4,772,222,266,850.90918
R^2 (Validation)	0.84111

Evaluation 2:

Metric	Value
Tree depth	7
Best expression	(arctan(((2 * x0) + (0.98622902973299 + ((-0.9774299250715413 + x2) + x1)))) * (3634475.3477245686 + (((x1 + x0) * 0.01830621799485299) * ((-0.9989511938091487 * (x0 + x2)) * 3537659.0406569946))))
MSE (Training)	4174477565534.58887
R^2 (Training)	0.85767
MSE (Validation)	4570609576769.90430
R^2 (Validation)	0.84782

Evaluation 3:

Metric	Value
Tree depth	7
Best expression	((x0 + ((-0.019111713360799722 * ((x0 + 0.9840486856017361) * 3519078.1007138137)) * ((x2 + x1) * (x0 + -1.3856915093946087)))) + (arctan(((1.3787422579085915 * x0) + (x1 + x2))) * 3519078.1007138137))
MSE (Training)	3972573310403.7026367
R^2 (Training)	0.8645552
MSE (Validation)	4094539755307.2978516
R^2 (Validation)	0.8636713

Problem 3 - Datasplit

Evaluation 1

Metric	Value
Tree depth	7
Best expression	(((sqr(x0) + 3.993555241630826) - (2.468855564282463 * x2)) + ((sqr(x0) - (sqr(x1) * x1)) - x2))
MSE (Training)	0.0000431
R^2 (Training)	0.9999968
MSE (Validation)	0.0080377
R^2 (Validation)	0.9999970

Evaluation 2

Metric	Value
Tree depth	7
Best expression	((2.002683911863346 * sqr(x0)) + ((3.9850503589375528 - (x2 * 3.5009790345132275)) + cube((-0.00023539136658632374 - x1))))
MSE (Training)	0.00042
R^2 (Training)	1.00000
MSE (Validation)	0.00045
R^2 (Validation)	1.00000

Problem 4 - Datasplit

Evaluation 1

Metric	Value
Tree depth	5
Best expression	((6.995299191648694 * cos(x1)) + (3.2803889063120284 - (0.09093106209030315 * x0)))
MSE (Training)	0.10001
R^2 (Training)	1.00000
MSE (Validation)	0.10001
R^2 (Validation)	1.00000

Evaluation 2

Metric	Value
Tree depth	5
Best expression	((6.9600502069780825 * cos(x1)) + (3.2794524213733283 + (-0.09122965339457964 * x0)))
MSE (Training)	0.00069
R^2 (Training)	0.99996
MSE (Validation)	0.00076
R^2 (Validation)	0.99997

Problem 5 - Datasplit

Evaluation 1

Metric	Value
Tree depth	5
Best expression	((x1 + x0) * 0.00010064840791317857)
MSE (Training)	0.00000
R^2 (Training)	**-52418640901.59256**
MSE (Validation)	0.00000
R^2 (Validation)	**-63479809411.50989**

Evaluation 2

Metric	Value
Tree depth	5
Best expression	exp((-10.593151254153158 + (sin(x0) * (x1 * -0.30136023861483663))))
MSE (Training)	0.00000
R^2 (Training)	**-204993204.92060**
MSE (Validation)	0.00000
R^2 (Validation)	**-244383841.26212**

Problem 6 - Datasplit

Evaluation 1

Metric	Value
Tree depth	5
Best expression	((x1 * 1.6953937438370514) - (x0 * 0.6936705811025641))
MSE (Training)	0.00001
R^2 (Training)	1.00000
MSE (Validation)	0.00001
R^2 (Validation)	1.00000

Evaluation 2

Metric	Value
Tree depth	5
Best expression	((1.6947538279052852 * x1) - (0.6945673410208579 * x0))
MSE (Training)	0.0000000
R^2 (Training)	1.0000000
MSE (Validation)	0.0000004
R^2 (Validation)	1.0000000

Problem 7 - Datasplit

Evaluation 1

Metric	Value
Tree depth	5
Best expression	exp((1.561295175294946 + (x1 * (x0 + -0.02613681253974187))))
MSE (Training)	376.0732867
R^2 (Training)	0.5140949
MSE (Validation)	267.3120089
R^2 (Validation)	0.5626167

Evaluation 2

Metric	Value
Tree depth	5
Best expression	exp((1.5497230604686836 + (x0 * x1)))
MSE (Training)	374.76424
R^2 (Training)	0.51284
MSE (Validation)	258.69441
R^2 (Validation)	0.57672

Problem 8 - Datasplit

Evaluation 1

Metric	Value
Tree depth	10
Best expression	((x0 + (4.675760748480646 * (x3 - ((cos(x2) + x5) + (cosh(x4) + (cosh(x4) * 4.571646229301208)))))) + (cube(x5) + (((cube(x5) - x1) + (4.675760748480646 * ((sinh(x3) - (4.675760748480646 * (-0.8676042152839172 + x5))) - (2.1691631403452503 * cosh(x4))))) + ((cube(x5) + (sinh(x3) + (4.675760748480646 * cube(x5)))) + (4.675760748480646 * (sqr(x5) * cube(x5)))))))
MSE (Training)	4222.87850
R^2 (Training)	0.99981
MSE (Validation)	4279.13926
R^2 (Validation)	0.99981

Evaluation 2

Metric	Value
Tree depth	10
Best expression	((((cube(x3) + (cube((x5 - 6.206303451447836)) + cube(x3))) + cube(((x4 + sin((-1.3420634973170156 * x5))) - 6.206303451447836))) + (cosh(x5) * ((cube((x5 - 0.002825691031909529)) - (x4 + ((x5 - 6.206303451447836) + ((2 * x5) - x4)))) + cube((x5 + sin((-1.3420634973170156 * x5))))))) - ((((cube(((x4 + 0.25332376515564414) + x4)) + -305.55726001262485) + exp(x4)) + ((((-0.8000949351095057 + (x4 * x4)) * x4) + cube(((x4 + 1.5485941972169874) - 6.206303451447836))) * ((0.36758101663471887 + (x4 + 0.943379279335353)) + 2))) + (((x1 + (cosh(x4) + (6.206303451447836 - cube(x3)))) + (x0 + (0.6927580940814835 + ((0.33963595881532704 + x4) + cube((-0.8174479516729702 - x5)))))) + (x5 + x2))))
MSE (Training)	20773.6818010
R^2 (Training)	0.9990838
MSE (Validation)	20571.9936794
R^2 (Validation)	0.9990933

Evaluation 3:

Metric	Value
Tree depth	10
Best expression	(((((((x4 * x4) * 0.4991427146821015) + cos(x4)) + 0.21807710951788753) * -191.34488188898908) + (2 * (((cube(x3) + (x4 * x4)) - cosh(x4)) + (x4 * x4)))) - (-191.07040326540172 + cos(x2))) + ((((((-1.1397882197335478 + cos((-0.8652716641979434 * x4))) - -1.8176821373732027e-05) * -77.01579866243283) + (((0.9452511652132713 - sqr(x1)) + (cube(x3) + x1)) + (x4 * x4))) - cosh(x4)) + ((cube(x5) * sqr((2 * x5))) + (((11.636265720405797 + x0) - cosh(x4)) + (0.5358978385659909 + (sqr((x5 * x5)) * x5))))))
MSE (Training)	2171.2550465
R^2 (Training)	0.9999035
MSE (Validation)	2116.7229381
R^2 (Validation)	0.9999067

MSE Evaluation

For each of the previous evaluations, the total MSE (train + validation) was calculated taking into account the functions (for each dataset)

LEGENDA: f{problem_id}_{evaluation_number}

Problem	Function	Train size 60%	Entire dataset
1	f1	0.000000	0.000000
2	f2_ev1	4622619768863.040	4682460768054.790
2	f2_ev2	4174477462263.185	4332930266757.311
2	f2_ev3	3972573310403.703	4021359888365.141
3	f3_ev1	0.007901	0.007956
3	f3_ev2	0.000422	0.000434
4	f4_ev1	0.000013	0.000013
4	f4_ev2	0.000754	0.000758
5	f5_ev1	0.000000	0.000000
5	f5_ev2	0.000000	0.000000
6	f6_ev1	0.000006	0.000006
6	f6_ev2	0.000000	0.000000
7	f7_ev1	377.536340	333.446608
7	f7_ev2	378.513329	330.585762
8	f8_ev1	4253.713062	4263.883542
8	f8_ev2	20814.148136	20717.286354
8	f8_ev3	2191.779112	2161.756642

The functions, for each dataset, that have obtained the best results in the total evaluation have been selected to compose the file **s324581_datasplit.py** (As you may notice the functions are those highlighted in the above table).

As you can see the GP algorithm generalizes very well, so to conclude the project I evaluated the GP algorithm on the **entire dataset**.

The following are the results obtained:

Problem	**Expression**	**MSE**	**R^2**
1	sin(x0)	0.0000000	1.0000000
2	((2 * arctan(((2 * x0) + (x2 + x1)))) * (((x2 + x1) * (((x0 + x2) * -5509.0006723736315) * ((x0 * 0.5747860657142498) * x0))) + 1785299.4561500459))	2796718779356.910645	0.9055603
3	(((sqr(x0) - x2) - ((x2 * cosh(tanh(x1))) - 1.3302391133518534)) + ((x1 * ((x0 * 3.723724201610524e-12) - sqr(x1))) - (((x2 - 0.6392235377956723) - sqr(x0)) + -2.039590593909345)))	0.262319	0.9998976
4	((cos(x1) + 0.4675685895554704) * (cos(tanh(x0)) + 6.3984015985878955))	0.078527	0.9963682
5*****	exp((-10.593151254153158 + (sin(x0) * (x1 * -0.30136023861483663))))	0.000000	-204993204.92060
6	((1.6942484016981965 * x1) - (0.694681289433442 * x0))	0.0000003	1.0000000
7	(exp(((x0 * x1) * 1.0797632614418142)) * 3.665180185745142)	327.512789	0.5394640
8	((sqr((2.0 * x5)) * (cube(x5) + 0.004274202671157051)) + ((sqr(x5) * cube(x5)) + ((sqr((x4 * (2.0 * x4))) * -0.9626766210100366) + (((((cosh(x1) * -0.828060103226649) + (x2 - -0.7769462579782747)) + (cube(x3) + ((0.02837821037606747 * x0) + x0))) - (x2 - -1.8241925696267818)) + ((sqr((2.0 * x4)) * -0.7506875922133385) + (cube(x3) + (cube(x3) + 2.028632985742853)))))))	76.3244082	0.9999966

(*****) Only for the **problem_5** i have decided to include the solution already found "**f5_ev2**", because it generalizes well on the validation data.