Politecnico di Torino

COMPUTATIONAL INTELLIGENCE

"If you don't know where you are going, any road will get you there"

2024-2025

By Agnese Re
Student ID S325676
github.com/AgneseRe

My Exotic CI Report

Agnese Re

February 2025

Abstract

This report documents the activities carried out during the Computational Intelligence course, providing a detailed overview of the laboratory tasks, project work, computational techniques, and data processing methods employed. In particular, the implementation and resolution of three well-known problems — Set Cover Problem, Travelling Salesman Problem, and N-Puzzle Problem — with local search and path search algorithms are analysed, highlighting the results obtained and the challenges encountered. Additionally, the report includes a full development of a Symbolic Regression algorithm using Genetic Programming to find the mathematical formula that best fits a given set of data, along with a presentation of fuzzing illustrating its fundamental principles and practical applications.

Before delving into the laboratory and project work sections, some initial reflections from the first day of class are included. First impressions, opinions written in the moment and left unchanged since then, opinions you might revisit with nostalgia after months or years. Readers are free to skip this section and proceed directly with laboratory and project work.

Finally, a chronologically organized log of all activities is reported.

Contents

List of Figures

1 [2024-sep-23] INFO: An Unusual Kickoff

1.1 Hamsters and Intelligence: A Curious Start

First lesson done. I am a bit perplexed — in the best possible way, I promise. Don't worry, don't be alarmed. This lesson was **out of the ordinary**, different from what I am used to from every point of view. Maybe it's just me, but... I've never seen such vibrant and friendly slides before — fairly large font size, simple and stylized drawings (like the ones kids make). We're used to those serious slides, black text on a white background, they're professional, but they're also dull, they don't catch your attention. And then the slides we saw today were so intriguing. The images and the quizzes really grab your attention, not to mention the content, which I really appreciated. Maybe I should be embarrassed, but some concepts that were repeated, and perhaps seemed obvious to many students, were things I had never heard of before. I know, I'm hopeless.

Okok, beyond my thoughts, today we need to start writing a *PDF report detailing all our activities during the course*, a sort of log file. I want to create something that reflects my personality, something I enjoy, something that will be useful to me in the future, and something that stands as a testament to what I've learned in our Computational Intelligence course. I'll follow the given requirements, but I'll try to create something **exotic**, something unique, just as unique as this course. If you're wondering about the title of this subsection, the answer lies in the first slides of the lesson, right after the general course outlines. I have some quirky ideas in my life, but I never imagined running down the street carrying a handful of hamsters LOL. I believe that would definitely be a problem. But if I imagine someone I particularly dislike in this situation... well, it would be fun. Maybe I shouldn't have said that 😶‍🌫️.

1.2 Nice to meet you, Pandoc

A tip for our report: *write in Markdown (also with CryptPad), and generate PDF with Pandoc.* When I read this tip, I thought: "Fantastic. We're off to a great start. I don't even know what Pandoc is. Well done, Agnese!". So what did I do today? When you don't know something, today the computer is your best friend. I typed **Pandoc** on Google and found the magnificent official guide of this text converter. Yes, I understood only then that it was a **text converter** 🤣. Jokes aside, I read the *About* page and then, following the instructions in the *Installing* page, I download the latest installer. All the installers, for Windows, macOS and Linux, are available on a GitHub repo of John MacFarlane, philosophy professor at the University of California, Berkeley.

I wrote a very short markdown and tried to convert it to pdf, using pandoc. Ok, perfect. It works, although it is not as aesthetically pleasing as a document I can write in LaTex, using Overleaf. For now I prefer this, then maybe I change my mind. I don't know. In this moment, I'm writing my report in LaTex 😢.

1.3 Deeper in Turing Test

I've always heard of **Alan Turing**, but I have never delved into his figure, his contribution to the world of Computer Science and **Artificial Intelligence**. Today is the time to do so. I want to find out more about his namesake machine

and his famous test to determine whether a machine is capable of thinking like a human being. Then, today I heard of the **Coffee Test** as an example of *AGI*. It was new to me. I'll know how to spend this evening before going to bed.

2 [2024-sep-24] LAB0: Long live irony! ☕

I read about the Coffee Test and it intrigued me so much that I decided to base the first lab on it. I will play a trick on my colleagues, making them believe that an intelligent machine that prepares them **a cup of hot coffee** every morning really exists. They won't even have to get out of bed and a soothing aroma will fill every corner of their house. I will give them a full description of this machine — then tell them that it's all a joke and that they have to hurry up because they are late. Sorry colleagues.

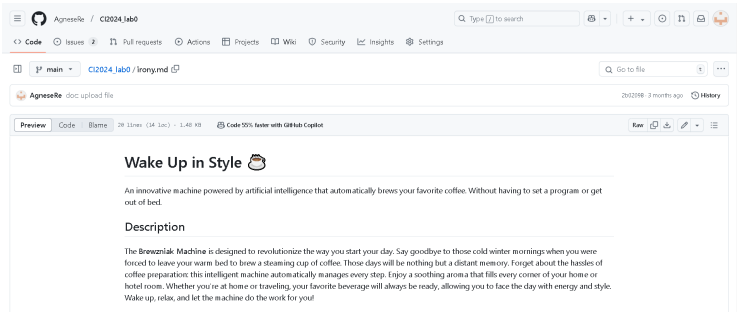

Figure 1: My irony: The Brewzniak Machine - initial part

I'm imagining the moment when they get to the end of the README (without knowing they've wasted precious time) and read *Just kidding* (*Figure 2*). If anyone actually believes that, I think I'll get a lot of curses. No please guys 🤣.

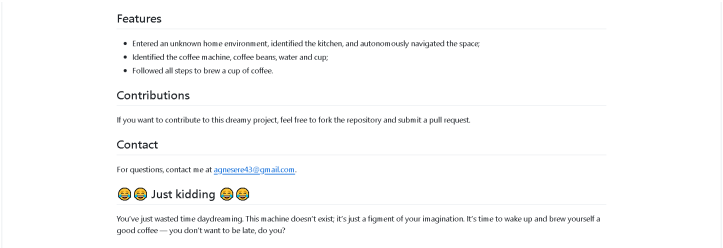

Figure 2: My irony: The Brewzniak Machine - final part

3 [2024-sep-27] REVIEW: Are you kidding me?

Even my colleagues make fun of me. I double-click on the **irony.md** file, thinking I'll find something funny, and instead... I forgive you, but only because I also played a prank on you. No, that's not true, I am not that touchy.

Today, I opened my first issue on GitHub, on *efemcy2245* lab0 repo. I had never opened an issue on GitHub in my university career. I'm not proud of it...

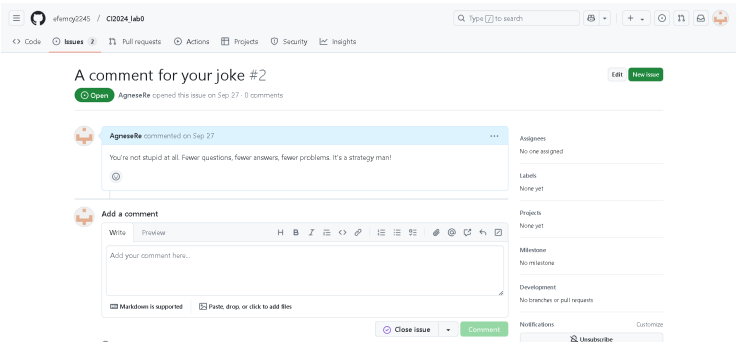

Figure 3: My review of lab0 - *efemcy2245* repo

On Lorenzo's repo double joke (*Figure 4*). He is very good with his irony, but... GitHub sent an email, informing me that my issue had been answered. I was convinced it was Lorenzo. But no. It was Antonio, *Asir98*, the other reviewer who, before opening a new issue, commented under mine 🤣.

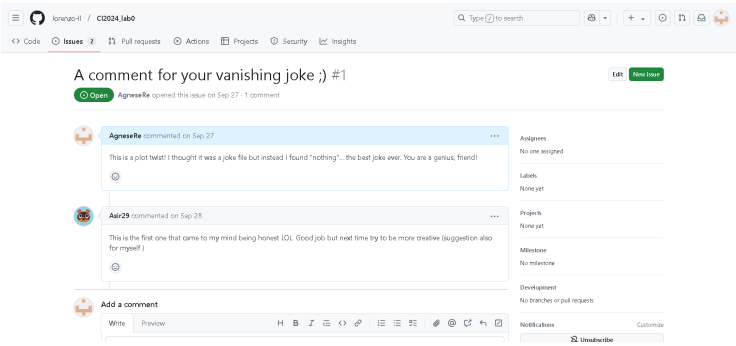

Figure 4: My review of lab0 - *lorenzo-II* repo

4 [2024-sep-28] ISSUE: A prophetic meeting

Someone read my joke about the innovative machine that makes coffee (The Brewzniak Machine). I'm already scared to read.

I can breathe a sigh of relief. The joke worked because I fooled them, but I think they appreciated the idea. Who wouldn't want a machine like that? Thank you Silvia, thank you Xhoana.

I am revising this subsection of the report on January 2, 2025 (which is my birthday and I celebrate like this, but ok — it is a personal choice) and I decided to change the title. **A prophetic meeting**. Yes because, spoiler, I would not have thought that I would find myself with Xhoana to do our group work on Symbolic Regression. It was truly a coincidence. But I am very happy about it. We both remembered each other's names, but we didn't immediately remember in what context we had "met". The two issues opened by Silvia and Xhoana on my lab0 repo are reported below, in *Figure 5* and *Figure 6*.

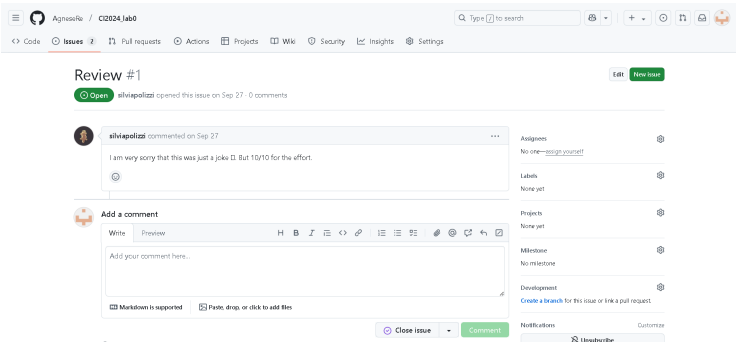

Figure 5: Issue of my lab0 repo - by *silviapolizzi*

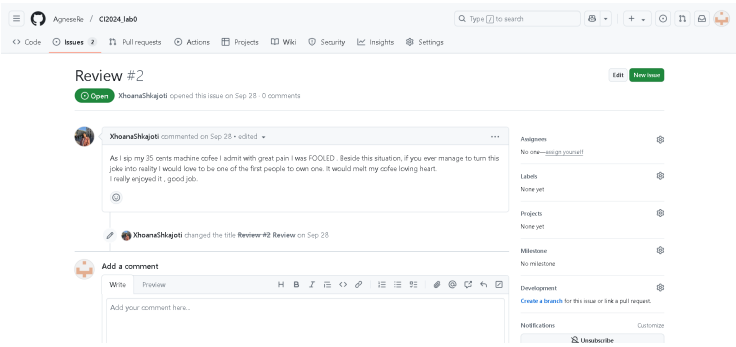

Figure 6: Issue of my lab0 repo - by *XhoanaShkajoti*

5 [2024-oct-09] LAB1: Set Cover Problem

5.1 Description of the Problem

The first laboratory focused on the Set Cover Problem, a classic NP-hard problem that involves identifying the minimum number of sets that cover all the elements in a given universe. Formally, given a universe of n elements $U = \{1, 2, ..., n\}$ and a collection of m subsets $S = \{S_1, S_2, ..., S_m\}$, the goal is to identify a smallest sub-collection of S whose union equals the universe U.

The analysis examined six different instances, each varying in the number of elements n (`UNIVERSE_SIZE`), the number of subsets m (`NUM_SETS`), and the density of the subsets (`DENSITY`). The instances are as follows:

- **Instance 1**: n = 100, m = 10, density = 0.2
- **Instance 2**: n = 1000, m = 100, density = 0.2
- **Instance 3**: n = 10000, m = 1000, density = 0.2
- **Instance 4**: n = 100000, m = 10000, density = 0.1
- **Instance 5**: n = 100000, m = 10000, density = 0.2
- **Instance 6**: n = 100000, m = 10000, density = 0.3

5.2 Implementation

Various optimization techniques are explored for solving the Set Cover Problem, with a particular focus on Random-Mutation Hill Climbing (RMHC) and Tabu Search. RMHC, a simple effective variant of hill climbing, introduces random mutations to move to the first better neighbor of the current solution, while Tabu Search enhances the search process by maintaining a short term memory structure, *i.e.* tabu list, to avoid revisiting previously explored states. The two local search algorithms are analyzed in terms of efficiency, solution quality and computational performance. It has been observed that Tabu Search generally outperforms Hill Climbing in solving the Set Cover Problem. This superiority is attributed to its ability to explore the solution space and avoid local optima.

Considering the fact that hill climbing techniques do not guarantee the achievement of the global optimum (quickly climb up to the first local optimum), different mutation strategies are implemented. As implemented in the corresponding function `single_mutation`, it is possible to randomly selected a single element of the solution array and flipped it, so select a random subset and add or remove it from the current solution, depending on whether it is already present or not. This results in a small, localized change, promoting **exploitation** of the current search region. A different approach is adopted in `multiple_mutation` function, where several elements of the solution array are mutated simultaneously. This introduces more significant changes, encouraging **exploration** of the solution space and helping the algorithm escape local optima. An adaptive strategy is implemented in the `multiple_mutation_strength` function, where the number of mutated elements varies based on a strength parameter. At the beginning of the optimization process, a higher strength value introduces broader exploration, allowing the algorithm to cover a wider search space. As the search progresses and better solutions are identified, the strength parameter is gradually reduced, leading to more exploitation and fine-tuning of the solution. This dynamic adjustment balances the trade-off between exploration and exploitation, improving the likelihood of finding high-quality solutions.

The implementation of all the functions mentioned above, along with the rest of the written code, is given in the following *Code* section.

5.3 Results

The following are the final fitness scores and the time taken to reach the optimal solution for all the six instances, using the Tabu Search algorithm. They are the same results reported in the *Cryptpad* shared sheet, slightly different from those reported in the GitHub readme as they belong to two different executions.

Instance	**Number of calls**	**Fitness**	**Computational Time**
1	105	290	$\approx$ 0.0 s
2	10386	6375	0.9 s
3	114617	118948	37.1 s
4	152959	1897840	36.38 min
5	143549	2139244	13.25 min
6	147714	2163626	21.59 min

Table 1: Calls, fitness and computational time for different instances

5.4 Code

Traditional Hill Climbing algorithm has the tendency to become stuck in sub-optimal regions (local optima), since it only accepts improvements. It works fine in convex search spaces or if it starts near to the optimal solution. In order to address this limitation, three different mutation functions are implemented.

```
def single_mutation(solution: np.ndarray) -> np.ndarray:
    new_solution = solution.copy()
    random_set = rng.integers(0, NUM_SETS)
    new_solution[random_set] = not new_solution[random_set]
    return new_solution

def multiple_mutation(solution: np.ndarray) -> np.ndarray:
    mask = rng.random(NUM_SETS) < 0.01
    new_solution = np.logical_xor(solution, mask)
    return new_solution

def multiple_mutation_strength(solution: np.ndarray, strength:
↪  float = 0.3) -> np.ndarray:
    mask = rng.random(NUM_SETS) < strength
    # Ensures that the function always mutates at least one bit
    if not np.any(mask):
        mask[np.random.randint(NUM_SETS)] = True
    new_solution = np.logical_xor(solution, mask)
    return new_solution
```

Starting from a random solution and choosing an adaptive mutation strategy, search for the *optimal* state in the following way. The `fitness` function is used to evaluate the quality of a solution, providing a measure of how much good the current solution is. The goal of the problem is to find a smallest sub-collection of subsets that cover the entire universe.

```
strength = 0.5  # High strength at the beginning

# Create a random solution of all True and compute fitness
solution = rng.random(NUM_SETS) < 1
solution_fitness = fitness(solution)
ic(solution_fitness)
history = [float(solution_fitness[1])]

for steps in range(10_000):
    if steps % 1000:
        strength /= 2   # decrease strength
    new_solution = multiple_mutation_strength(solution)
    history.append(float(fitness(new_solution)[1]))
    if fitness(new_solution) > solution_fitness:
        solution = new_solution
        solution_fitness = fitness(solution)

ic(fitness(solution))
```

The Tabu Search algorithm is reported here. Instead of starting from a random solution, a greedy approach is adopted. Iteratively select the subset that covers the largest number of uncovered elements, continuing until all elements of the universe are covered. At each step, choose the locally optimal solution (the subset with the greatest immediate coverage), without backtracking, in order to build a solution that is efficient but not necessarily globally optimal.

```
# Tabu parameters
ITERATION_NO_IMPROVEMENTS = 100
TABU_SIZE = 100    # int(UNIVERSE_SIZE * 0.10)

# Useful functions
def greedy_initial_solution():
    solution = []
    not_covered_elements = np.ones(UNIVERSE_SIZE, dtype = bool)

    while np.any(not_covered_elements):
        coverage = np.dot(SETS, not_covered_elements)
        selected_set = np.argmax(coverage)
        solution.append(selected_set)
        not_covered_elements =
        ↪  np.logical_and(not_covered_elements,
        ↪  ~SETS[selected_set])

    boolean_solution = [True if set_index in solution
    else False for set_index in range(NUM_SETS)]

    return boolean_solution

def get_neighbors(solution: np.ndarray) -> list[np.ndarray]:
    neighborhood = []

    if NUM_SETS > 1000:
        candidate_indices = rng.choice(NUM_SETS, 1_000,
        replace = False)
    else:
        candidate_indices = range(NUM_SETS) # all sets

    for set_index in candidate_indices:
        new_solution = solution.copy()
        new_solution[set_index] = not new_solution[set_index]
        if(valid(new_solution)):
            neighborhood.append(new_solution)

    return neighborhood

# --- Tabu Search Algorithm ---
initial_solution = greedy_initial_solution()

best_solution = initial_solution
```

```
best_neighbor = initial_solution
fitness_best_solution = fitness(best_solution)
print(f"Initial fitness: {fitness_best_solution[1]}")
no_improvements = 0
tabu_list = [initial_solution]
history = [float(fitness_best_solution[1])]

while no_improvements <= ITERATION_NO_IMPROVEMENTS:
    neighbors = get_neighbors(best_neighbor)
    improved = False

    # FIND THE BEST CANDIDATE FROM THE CANDIDATE LIST
    best_neighbor_fitness = (True, float('-inf'))
    for neighbor in neighbors:
        fitness_neighbor = fitness(neighbor)
        if(fitness_neighbor > best_neighbor_fitness and
            not any(np.array_equal(neighbor, tabu) for tabu in
            ↪  tabu_list)): # lexicographic comparison
            best_neighbor = neighbor
            best_neighbor_fitness = fitness_neighbor

    # NO NEIGHBORS, NOT IN THE TABU LIST
    if best_neighbor_fitness == float('-inf'):
        break

    # UPDATE SOLUTION AND TABU LIST
    history.append(float(best_neighbor_fitness[1]))
    if best_neighbor_fitness > fitness_best_solution:
        improved = True
        no_improvements = 0
        best_solution = best_neighbor
        fitness_best_solution = best_neighbor_fitness

    tabu_list.append(best_neighbor)
    if len(tabu_list) > TABU_SIZE:
        tabu_list.pop(0)

    if not improved:
        no_improvements += 1

print(f"Final fitness: {fitness_best_solution[1]}")
selected_sets = [i for i in range(NUM_SETS) if best_solution[i]
↪  == True]
print(f"Selected sets: {selected_sets}")
```

5.5 Obtained Reviews

Starting from laboratory 1, I began to appreciate more the issues opened by my colleagues. Helpful comments and suggestions for improving my code. I started replying to them, opening a new comment on the GitHub issue at the time.

Figure 7 shows the issue opened by Giorgia (*gghisolfo*) and my response. She suggested me a way to reduce the running time of my algorithm. Fantastic.

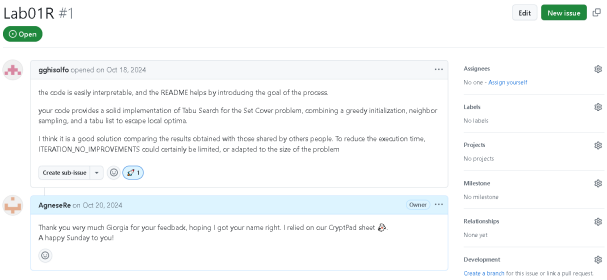

Figure 7: Issue of my lab1 repo - by *gghisolfo*

Instead, Giovanni (*GiovanniOrani*) analyzed the choices I made in implementing the code and suggested me how to improve it, too (*Figure 8*).

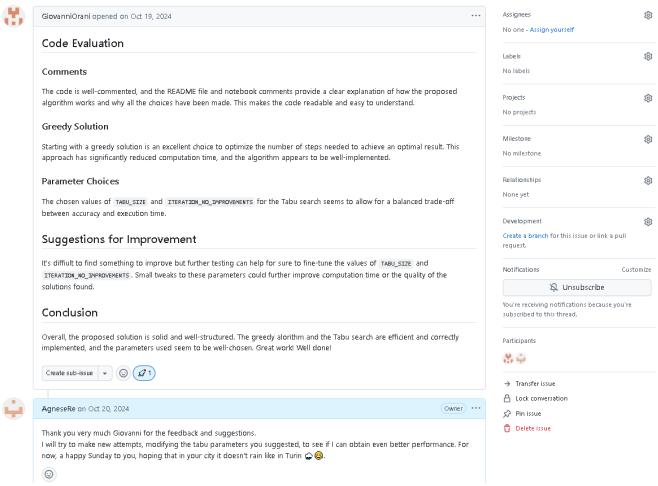

Figure 8: Issue of my lab1 repo - by *GiovanniOrani*

6 [2024-oct-19] REVIEW: Set Cover Problem

Today, I opened two issues: one on Marialuigia (*Mars1601*) repo and one on Luciana (*LucianaColella7*) repo. For an easier reading, I report directly the text here. Screenshots are available immediately afterwards (*Figure 9* and *Figure 10*).

- **On Mars1601 repo**

  Hi Marialuigia!

  Do I tell you the truth? When I opened your notebook on GitHub for the first time I was scared because I saw the side scroll bar that was really small due to the amount of content. I thought it was all code, only later

I realized they were printouts due to the `ic(iteration, best_fitness, temperature)` instruction. We already scared ourselves with the jokes of laboratory 0, not here too, come on 🤣.

I'm joking of course. Your work is great. I sincerely appreciated it and the effort you put into it is evident. You have implemented a solid version of **simulated annealing** and you have accompanied your project with a very nice README. The initial description and the subsequent graphs help the reader to immediately have a clear idea of the problem, how it was solved (with which strategies) and what performance was achieved by the algorithm with different instances.

If I were to criticize this work, I would be a liar. I can only make one observation. I noticed that, at each iteration of the algorithm, you swapped out a single set, in order to generate a new solution. You could experiment with a variable number of swaps to **increase the exploration** of the solution space, especially in the early stages with high temperature. What do you think about it? I'll leave you a very simple implementation of `multiple_mutation` function that you can use in your code in order to generate more aggressive new solutions. Feel free to edit it or criticize it if you find something wrong.

This is the only flaw. For the rest, congratulations for the commitment and care you put into the entire notebook.

Thank you very much for your code (for the quality and for the reflection it led me to do regarding the generation of new solutions). Thanks to you and to Carlo and Sergio who I saw mentioned as collaborators. Unity is strength and your work is proof of this 💪 🔥.

An happy weekend. See you soon.

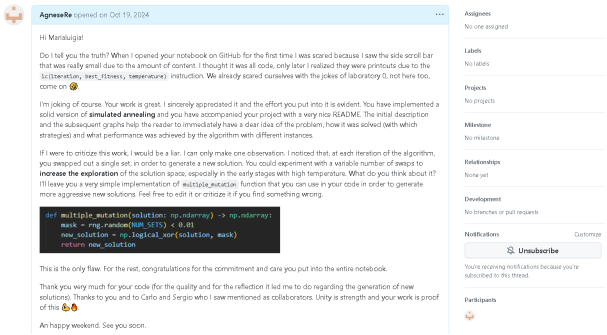

Figure 9: My review of lab1 - *Mars1601* repo

- **On LucianaColella7 repo**

  Hello Luciana 👋,

  I really appreciated your work. You started with a simple **RMHC algorithm** that any person, with a little effort, can understand its logic, and then you gradually moved on more advanced algorithms, like self-adaptive hill climbing and simulated annealing.

The idea of dynamically changing the strength of the mutation in the **self-adaptive version**, depending on the progress of the optimization process, is fantastic. You've made the hill climbing algorithm more adaptive and you've achieved a great balance between exploration and exploitation. If the algorithm finds a certain number of increasingly better solutions, you are right. I can explore new areas in the solution space, push myself further, increasing strength. Thank you very much for this point of reflection.

I also tried to do some experiments on your code, modifying the mutation strength and the buffer size and I think the choice you made is a good one. Well done! I'll leave you the link of an online resource that I found very interesting and which could help us improve the performance of the algorithm even further. It's an excerpt of a conference paper "On a Hill-Climbing Algorithm with Adaptive Step Size: Towards a Control Parameter-Less Black-Box Optimisation Algorithm": https://link.springer.com/chapter/10.1007/3-540-34783-6_56.

Just two notes. I'm very sorry that in the **simulated annealing** algorithm there is no check on whether the solution obtained is valid or not (a simple `if not valid(new_solution)` might have been enough) and that you didn't take the time to write the **README** because it would have been a great way to present your work and make it easier to understand. But these are the only flaws. For the rest, congratulations for the commitment and care you put into the entire notebook 🚀.

Thanks again for everything and an happy weekend.

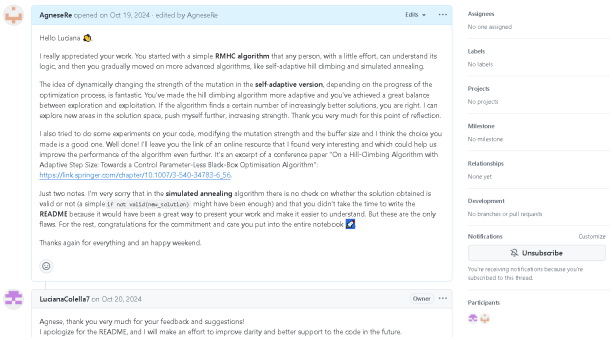

Figure 10: My review of lab1 - *LucianaColella7* repo

Luciana also responded to my issue. Perhaps we are all starting to better understand how it works as a tool and how useful it can be for improving and sharing code. Great.

7 [2024-nov-01] LAB2: TSP

7.1 Description of the Problem

The second laboratory focused on the Travelling Salesman Problem (TSP), an NP-hard problem that involves a salesman who needs to visit a set of cities and return to the starting point. The question that the TSP tries to answer

is: “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?”. The analysis of this problem was performed using data from five different countries: Vanuatu, Italy, Russia, US, and China.

7.2 Implementation

For addressing the Travelling Salesman Problem (TSP), two different approaches are used: a greedy approach that, from time to time, moves to the nearest city and an evolutionary algorithm based on mutation and cross-over operators. The performance of the algorithms are evaluated in terms of cost and number of steps across various instances. The process used is summarized as follows:

- **Initial population**: Initialize a population of `POPULATION_SIZE` random individuals, where each individual represents a unique route of the cities. In a first version of the algorithm, each individual was constructed by randomly permuting (shuffling) the order of cities. However, this technique provided initial solutions that were too far from the optimal, requiring an excessive number of generations to converge. In the second version of the algorithm (the one proposed in the notebook), the initial population is created using a more greedy approach. Instead of starting with purely random permutations, each individual begins from a randomly selected city and follows a “greedy neighbor” heuristic to build the path;
- **Cross-over**: An offspring is constructed crossing over two selected parents, chosen through tournament selection method. Cross-over occurs with a certain probability (`CROSSOVER_PROBABILITY`). The resulting offspring, obtained through the Inver-over crossover, is a new individual that asymmetrically inherits more traits from the first parent than the second parent. This asymmetry allows the offspring to mantain a strong similarity to the first parent, while incorporating some diversity from the second;
- **Mutation**: An offspring is constructed mutating a selected parent, chosen through tournament selection. Mutation occurs with a certain probability (`1-CROSSOVER_PROBABILITY`). The resulting offspring is obtained through the Displacement mutation, that cuts the genes between two randomly cut-points (start, end) from the chromosome of the parent and then inserts them, starting from a randomly selected location;
- **Hill Climbing**: A simple hill climbing technique is integrated into the genetic algorithm (GA) to enhance the optimization process. A constraint based on maximum number of attempts (`MAX_ATTEMPTS`) is imposed. This to prevent excessive computational effort.

The process of cross-over, mutation and hill climbing continues until the number of generated offsprings reaches the desidered size (`OFFSPRING_SIZE`). Then, the population is extended and only the fittest `POPULATION_SIZE` individuals are retained for the next generation.

7.3 Results

The greedy algorithm starts from a city and iteratively jumps to the nearest unvisited one. This process continues until all cities are visited. It’s relatively

fast, but it only provides a solution that approximates the shortest route. In the table below, the results obtained using the greedy algorithm and the optimal ones are compared. All costs are reported in km. China optimal route cost must be computed. Not available in the Wolfram Notebook provided during lesson.

Instance	No. Cities	Greedy Route Cost	Optimal Route Cost
Vanuatu	8	1475.53	1345.54
Italy	46	4436.03	4172.76
Russia	167	42334.16	32722.50
US	340	48050.03	39016.62
China	746	EA is used	to be computed

Table 2: Comparison of Greedy Route Costs and Optimal Route Costs

Better results are obtained using evolutionary algorithm, as discussed below.

Routes Plots and Conclusions EA

Vanuatu, Italy, Russia, US and China TSP plots are displayed below in this order. The first plot refers to Vanuatu, the second plot to Italy etc.

- While the optimal solution was achieved for the first two instances (Vanuatu and Italy), there is certainly room for improvement for the next three, despite numerous experiments being carried out. There have been various mutation and crossing over strategies tested and various combinations of parameters `CROSSOVER_PROBABILITY`, `OFFSPRING_SIZE` and `MAX_ATTEMPTS`, but the algorithm did not lead to better solutions than those reported below;

- 10_000 generations for Russia, US and China: since the result obtained with only 10_000 generations is not that far from optimal (34866.38 vs 32722.50 for Russia and 46719.28 vs 39016.62 for US), the algorithm should not be badly structured. Probably with a greater `POPULATION_SIZE` (e.g. 20_000 for US instance, 100_000 for China instance), it would reach the optimum. The process would become very expensive in computational terms and my calculator (a humble and now old laptop) isn't that good. It takes tens of minutes, also hours to calculate the result :(.

- Trying with a number of generations equal to 100_000 for China and waiting 200 minutes (process not yet finished, interrupted by keyboard, so a minor number of generations completed), the best path is the one shown in the sixth plot. If you waited exactly 100_000 generations (more than 200 minutes on my laptop), you would probably get a better result.

In each of the subsequent images the population size, the number of generations and the cross-over rate are reported, apart from the total distance travelled in km. Each city, identified by its coordinates, is represented by a red dot. The path for reaching a city from another one is represented by a blue dotted line.

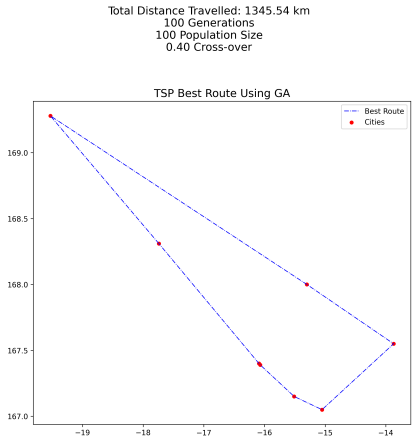

Figure 11: TSP Vanuatu Best Route Using GA

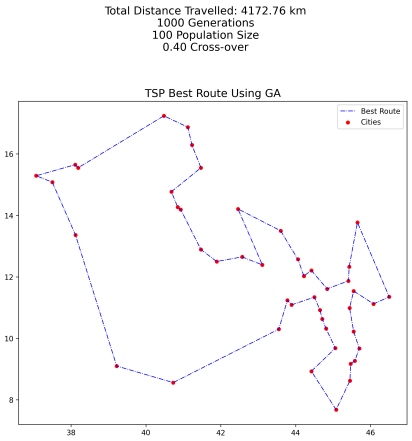

Figure 12: TSP Italy Best Route Using GA

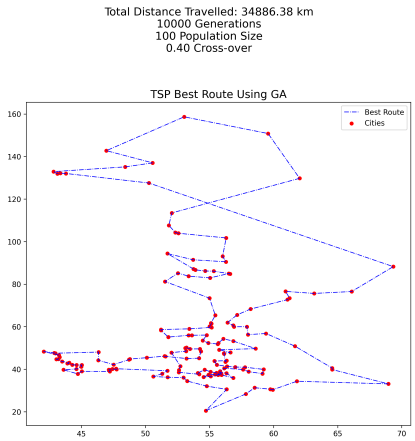

Figure 13: TSP Russia Best Route Using GA

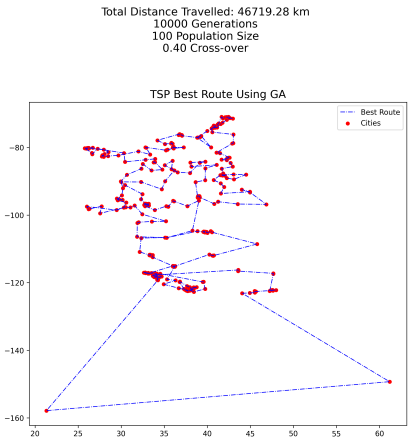

Figure 14: TSP US Best Route Using GA

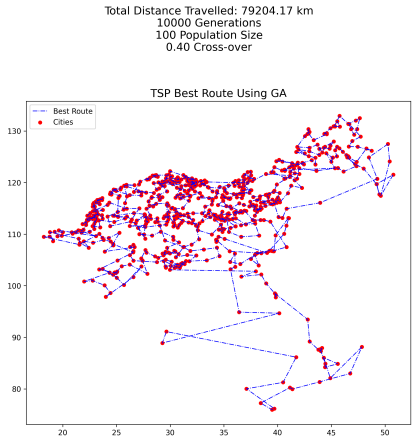

Figure 15: TSP China Best Route Using GA 10000 generations

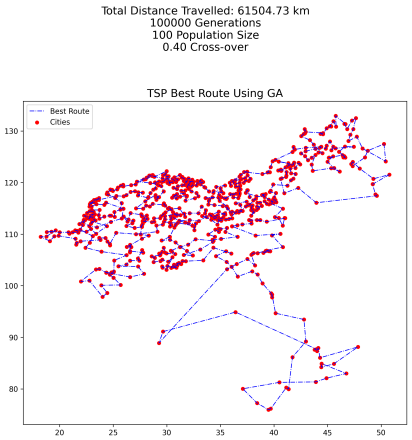

Figure 16: TSP China Best Route Using GA 100000 generations

7.4 Code

After reading comma-separated value (csv) file and computing the distance matrix between cities of a certain country, the data class `Individual` is defined. The genome attribute holds an individual's genome. The fitness attribute quantifies how well the individual performs with respect to the objective function. Initially, it is set to None. The fitness score will be calculate later, after the genome is evaluated.

```
@dataclass
class Individual:
    genome: np.ndarray
    fitness: float = None

def calculate_fitness(individual: Individual) -> float:
    tot_cost = 0.0
    for c1, c2 in zip(individual.genome,
    ↪  np.concatenate((individual.genome[1:],
    ↪  individual.genome[:1]))):
        tot_cost += DIST_MATRIX[c1, c2]
    return -tot_cost
```

For a Greedy approach (immediate cost gain, but approximate solution):

```
# At the beginning, no city is visited
visited = np.full(len(CITIES), False)

# Set the starting point and set it to visited
city = 0
visited[city] = True
dist = DIST_MATRIX.copy()

# List to store results
tsp = []
tsp.append(int(city))

while not np.all(visited):
    closest_cost = np.min([dist[city, adj] for adj in
    ↪  range(len(CITIES)) if not visited[adj] and adj != city])
    closest = np.where(dist[city] == closest_cost)[0][0]
    # Set the found city to visited and add it to the tsp list
    visited[closest] = True
    tsp.append(int(closest))
    # Update city
    city = closest

# Print result
best_path = Individual(tsp)
best_path.fitness = calculate_fitness(best_path)
print(f"{len(CITIES)} cities, shortest tour is
↪  {-best_path.fitness:.2f} km")
print([str(CITIES_NAMES[index]) for index in tsp])
```

Below are some important functions used in the Evolutionary Algorithm.

```
POPULATION_SIZE = 100
population = []
rng = np.random.default_rng()

def greedy_neighbor(start_city, k = 3):
    unvisited = set(CITY_INDEXES)
    unvisited.remove(start_city)
    current_city = start_city
    path = [int(current_city)]

    while unvisited:
        # Choose a random city from the closest k
        neighbors = sorted(unvisited, key=lambda city:
        ↪  DIST_MATRIX[current_city][city])[:k]
        next_city = rng.choice(neighbors)

        path.append(int(next_city))
        unvisited.remove(next_city)
        current_city = next_city

    new_individual = Individual(np.array(path))
    new_individual.fitness = calculate_fitness(new_individual)

    return new_individual

# Parents selection method
def parent_selection(population: np.ndarray) -> Individual:
    candidates = sorted(np.random.choice(population, 2), key =
    ↪  lambda i: i.fitness, reverse=True)
    return candidates[0]

# Two possible types of mutation
def swap_mutation(individual: Individual) -> Individual:
    new_individual = copy.deepcopy(individual)
    i1, i2 = np.random.choice(len(individual.genome), 2, replace
    ↪  = False)
    new_individual.genome[i1], new_individual.genome[i2] =
    ↪  new_individual.genome[i2], new_individual.genome[i1]
    new_individual.fitness = calculate_fitness(new_individual)
    return new_individual

def displacement_mutation(individual: Individual) -> Individual:
    genome = individual.genome
    start = np.random.randint(0, len(genome) - 2)
    end = np.random.randint(start + 1, len(genome) - 1)
    segment = genome[start: end+1]
    genome = np.concatenate((genome[:start], genome[end+1:]))
    random_location = np.random.randint(0, len(genome))
```

```
45      new_genome = np.concatenate((genome[:random_location],
        ↪  segment, genome[random_location:]))
46      new_individual = Individual(new_genome)
47      new_individual.fitness = calculate_fitness(new_individual)
48      return new_individual
49
50  # Two possible types of cross-over
51  def order_crossover(parent1: Individual, parent2: Individual):
52      parent1, parent2 = Individual(parent1.genome,
        ↪  parent1.fitness), Individual(parent2.genome,
        ↪  parent2.fitness)
53      size = len(parent1.genome)
54
55      child = np.full(size, -1)
56      start, end = sorted(np.random.choice(size, 2, replace=False))
57      child[start:end+1] = parent1.genome[start:end+1]
58      remaining_cities = [city for city in parent2.genome if city
        ↪  not in child]
59
60      j = 0
61      for i in range(size):
62          if child[i] == -1:
63              child[i] = remaining_cities[j]
64              j += 1
65
66      new_individual = Individual(child)
67      new_individual.fitness = calculate_fitness(new_individual)
68
69      return new_individual
70
71  def inver_over(parent1: Individual, parent2: Individual) ->
    ↪  Individual:
72      genome1, genome2 = parent1.genome, parent2.genome
73
74      index1 = np.random.randint(0, len(genome1))
75      gene1 = genome1[index1]
76
77      index2 = np.where(genome2 == gene1)[0][0]
78      gene2 = genome2[index2+1] if index2 != len(genome2)-1 else
        ↪  genome2[0]   # cyclic genotype
79
80      # generate offspring
81      flip_parent1 = np.concatenate((genome1[index1:],
        ↪  genome1[:index1]))
82      index_gene2_in_flip_parent1 = np.where(flip_parent1 ==
        ↪  gene2)[0][0]
83      sequence = flip_parent1[1:index_gene2_in_flip_parent1]
84      offspring_genome = np.concatenate((np.array([gene1]),
        ↪  np.array([gene2]), sequence[::-1],
        ↪  flip_parent1[index_gene2_in_flip_parent1+1:]))
```

```
85
86      offspring = Individual(offspring_genome)
87      offspring.fitness = calculate_fitness(offspring)
88
89      return offspring
```

Local search optimization techniques have been integrated into the genetic algorithm. The hill climbing technique was found to be useful for achieving optimal results.

```
1   MAX_ATTEMPTS = 20
2   INITIAL_TEMP = 100
3   COOLING_RATE = 0.95
4
5   def simulated_annealing(individual: Individual) -> Individual:
6       improved_individual = copy.deepcopy(individual)
7       temperature = INITIAL_TEMP
8       for _ in range(MAX_ATTEMPTS):
9           neighbor = displacement_mutation(improved_individual)
10          delta_fitness = neighbor.fitness -
            ↪  improved_individual.fitness
11          if delta_fitness > 0 or np.random.random() <
            ↪  np.exp(delta_fitness/temperature):
12              improved_individual = neighbor
13          temperature *= COOLING_RATE
14      return improved_individual
15
16  def hill_climbing(individual: Individual) -> Individual:
17      improved_individual = copy.deepcopy(individual)
18      for _ in range(MAX_ATTEMPTS):
19          neighbor = displacement_mutation(improved_individual)
20          delta_fitness = neighbor.fitness - individual.fitness
21          if delta_fitness > 0:
22              improved_individual = neighbor
23      return improved_individual
24
25  # Initial population
26  for _ in range(POPULATION_SIZE):
27          start_city = int(rng.choice(CITY_INDEXES))
28          population.append(greedy_neighbor(start_city))
29  # print(sorted([i.fitness for i in population], reverse=True))
```

```
1   # --- EVOLUTIONARY ALGORITHM PROCESS ---
2   CROSSOVER_PROBABILITY = .40
3   OFFSPRING_SIZE = POPULATION_SIZE//5
4   NUM_GENERATIONS = 100_000
5
6   for _ in range(NUM_GENERATIONS):
7       population.sort(key = lambda i: i.fitness, reverse=True)
8       offsprings = []
9
```

```
    while len(offsprings) < OFFSPRING_SIZE:
        xover_probability = np.random.random()
        if xover_probability < CROSSOVER_PROBABILITY:
            parent1 = parent_selection(population)
            parent2 = parent_selection(population)
            offspring = inver_over(parent1, parent2)
        else:   # no crossover
            parent1 = parent_selection(population)
            offspring = displacement_mutation(parent1)

        offsprings.append(hill_climbing(offspring))
    population.extend(offsprings)
    population.sort(key=lambda i: i.fitness, reverse=True)
    population = population[:POPULATION_SIZE]

population.sort(key = lambda i: i.fitness, reverse = True)
shortest_path = [str(CITIES_NAMES[index]) for index in
↪  population[0].genome]
print(f"{len(CITIES)} cities, shortest tour is
↪  {-population[0].fitness:.2f} km")
print(shortest_path)
```

7.5 Obtained Reviews

Unfortunately my lab2 repo was not assigned to any colleague and therefore none of them opened an issue. However, I received the following comment from GiuseppeEsposito98.

"This project provides a clear and detailed exploration of TSP solutions using both greedy and evolutionary approaches. A potential improvement could be to expand on the scalability of the evolutionary algorithm, particularly for larger datasets like the US and China.

For example, discussing how variations in parameters such as POPULATION_SIZE or NUM_GENERATIONS impact performance and solution quality could offer valuable insights. Adding this information would provide a more comprehensive view of the algorithm's adaptability and strengths."

8 [2024-nov-17] REVIEW: TSP

Today, I opened two issues: one on Giorgio (*GiorgioBongiovanni*) repo and one on Ali (*AliEdrisabadi*) repo. For an easier reading, I report directly the text here. Screenshots are available immediately afterwards (*Figure 17*, *Figure 18* and *Figure 19*).

- **On GiorgioBongiovanni repo**

  Hi Giorgio,

  I want to congratulate you from now on for the work you have done. Your code is clear and the results you have achieved are excellent. For the first instance (*Vanuatu*) you found the optimal solution, for the remaining ones your solutions are really very close to the optimal one. I really appreciated

that you carefully commented out your code and that you included a README file in your repo. This made it much easier to read the code, clear and smooth, absolutely not heavy. Well done! 👋 🎉

Suggestions

- Your algorithm already presents an excellent trade off between the quality of the result obtained and the computational cost, but in my opinion you could still improve it. On small instances such as the one relating to *Vanautu* and *Italy*, the algorithm may not need such a large **number of generations** to reach the optimal solution. Trying to run your code with only 1000 generations, instead of 100_000, for Italy, I got 4316.88 km as a result. Not great, but absolutely good. If you don't want to fix a number of generations, you could still assign the value 100_000 to `MAX_GENERATIONS`, but implement an **early stopping** mechanism that terminates execution if there are no significant improvements for a certain number of consecutive generations. What do you think? I'll give you just an idea of this.

```
no_improvement_counter = 0
for generation in range(MAX_GENERATIONS):
    # your code for creating new population
    if fitness_improved:
        no_improvement_counter = 0
    else:
        no_improvement_counter += 1

    if no_improvement_counter > 100:
        # stop the execution if more than 100
        ↪ generations with no improvement
        break
```

- You could try changing the value of the `POPULATION_SIZE` parameter. By starting with a larger population you could get even better solutions than what it has already achieved. Fine-tuning on the crossover probability `p` could also lead you to even better results. The crossover probability (`p=0.02`) may be too low.
- In the README file you could add first, second or third level titles to better structure it and make reading easier and more immediate. But this is just a comment to be meticulous. It is like looking for a needle in a haystack.

- **On AliEdrisabadi repo**

  Hello Ali,

  First of all, congratulations for the clarity and order of your code. It appears very clean and this helps the person reading a lot. I really appreciated the fact that you proposed two alternative solutions for the TSP problem, one greedy and one based on an evolutionary algorithm. I ran your code on my computer and I'll just give you some **food for thought** (food. I'm already craving pizza again - sorry, those are my thoughts at the moment). I know, I'm hopeless ...

Greedy algorithm 🤤

Your implementation is very good. I just wanted to analyze the two following points with you, nothing more.

- I noticed that an **infinite distance** is always returned. But I think this is due to a simple careless error. Instead of passing `dist_matrix` (here you set some distances to infinity to select the closest city among those not yet visited), try to pass `DIST_MATRIX`, the original distance matrix. This way your code works perfectly. Well done!
- Setting the distance from a certain city to infinity to prevent it from being chosen again once visited seemed like too brutal a solution. Let's say, too cruel. Here, I leave you a different solution that I found and which seems to work anyway. Less cruel let's say. I simply rewrote the first two lines inside the while loop. Nothing more. `adj` stands for adjacent city. Tell me what you think.

```
while not np.all(visited):
    # --- start modified lines of code (instead of setting np.inf) ---
    next_city_cost = np.min([dist_matrix[current_city, adj] for adj in range(len(CITIES)) if not visited[adj] and adj != current_city])
    next_city = np.where(dist_matrix[current_city] == next_city_cost)[0][0]
    # --- end ---
    logging.debug(f"Moving from {cities.at[current_city, 'City']} to {cities.at[next_city, 'City']} "
                  f"({dist_matrix[current_city, next_city]:.2f} km)")
    visited[next_city] = True
    current_city = next_city
    route.append(current_city)
```

In this case, in order not to further burden the report, only the screenshot of the second part of the issue is reported (after the screenshot of the review on *GiorgioBongiovanni* repo).

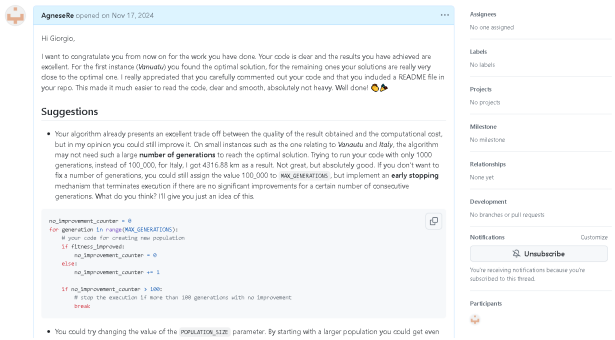

Figure 17: My review of lab2 - *GiorgioBongiovanni* repo

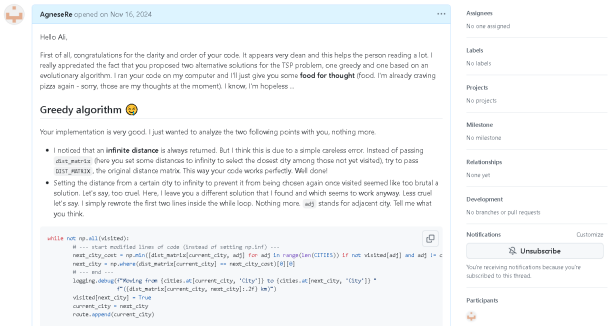

Figure 18: My review of lab2 - *AliEdrisabadi* repo first part

Figure 19: My review of lab2 - *AliEdrisabadi* repo second part

The third and last part of the issue on Ali repo is simply a conclusion.

9 [2024-nov-18] LAB3: N-Puzzle Problem

9.1 Description of the Problem

The third laboratory focused on the N-Puzzle problem, a well-known problem in artificial intelligence. The N-Puzzle is a sliding puzzle consisting of n numbered tiles and an empty space, that challenges the player to move pieces along certain routes in order to establish a specific end-configuration (usually in ascending numerical order). 8-puzzle consists of 8 tiles in a 3x3 frame, 15-puzzle consists of 15 tiles in a 4x4 frame, 24-puzzle consists of 24 tiles in a 5x5 frame etc. This program solves n-puzzle using A* algorithm with various heuristics and Problem Decomposition with Goal Tree.

9.2 Implementation

Heuristics

- Hamming Distance / Misplaced Tiles: the number of tiles that are not in the correct position. This heuristic is the simplest, but also the slowest. A huge amount of states will be explored in order to reach the goal state;

- Manhattan Distance / Taxicab Geometry: the Manhattan distance of a tile is the distance or the number of tiles away it is from its target position. It is calculated as the sum of the absolute differences between its current row and column and its target row and column. For a certain puzzle configuration, the Manhattan distance is the sum of the Manhattan distances of all the tiles, except the blank tile;
- Manhattan Distance / Taxicab Geometry: the Manhattan distance of a tile is the distance or the number of tiles away it is from its target position. It is calculated as the sum of the absolute differences between its current row and column and its target row and column. For a certain puzzle configuration, the Manhattan distance is the sum of the Manhattan distances of all the tiles, except the blank tile;
- Linear Conflict + Manhattan Distance / Taxicab Geometry: two tiles tile_a and tile_b are in linear conflict if they are both in their correct row or column, but they are reversed relative to their goal positions. They are in the same row or column, their target positions are in the same row or column and the target position of one of the tiles is occupied by the other tile in that row or column. The heuristic value is the Manhattan Distance plus two times the number of linear conflicts (manhattan_distance + 2 * n_linear_conflicts).

8-Puzzle Example

Starting from a given initial configuration, obtained by randomly moving the tiles for `RANDOMIZE_STEPS` steps, a path like the following is obtained. The implemented A* algorithm is particularly efficient. It obtains the solution in just 23 moves in a particularly short time, in the order of one second. The first state in the path is the initial_state, while the last state is the goal_state.

```
    # Path obtained with A*
[
    array([[7, 3, 4],
       [2, 0, 5],
       [1, 6, 8]]),
    array([[7, 3, 4],
       [2, 6, 5],
       [1, 0, 8]]),
    array([[7, 3, 4],
       [2, 6, 5],
       [1, 8, 0]]),
    ...
    array([[2, 7, 3],
       [1, 6, 4],
       [8, 0, 5]]),
    array([[2, 7, 3],
       [1, 0, 4],
       [8, 6, 5]]),
    array([[2, 0, 3],
       [1, 7, 4],
```

```
        [8, 6, 5]]),
    array([[0, 2, 3],
        [1, 7, 4],
        [8, 6, 5]]),
    array([[1, 2, 3],
        [0, 7, 4],
        [8, 6, 5]]),
    array([[1, 2, 3],
        [7, 0, 4],
        [8, 6, 5]]),
    array([[1, 2, 3],
        [7, 4, 0],
        [8, 6, 5]]),
    array([[1, 2, 3],
        [7, 4, 5],
        [8, 6, 0]]),
    array([[1, 2, 3],
        [7, 4, 5],
        [8, 0, 6]]),
    array([[1, 2, 3],
        [7, 4, 5],
        [0, 8, 6]]),
    array([[1, 2, 3],
        [0, 4, 5],
        [7, 8, 6]]),
    array([[1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]]),
    array([[1, 2, 3],
        [4, 5, 0],
        [7, 8, 6]]),
    array([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]])
]

# Printing results
Solve 8-puzzle in 23 moves
306 evaluated actions
```

9.3 Results

The following graph show the evolution of the heuristic function for the 8-Puzzle, in term of h_distance, used in the A* algorithm during the search process to solve the n-puzzle problem. The x-axis represents the number of steps taken by the algorithm, while the y-axis represents the value of the heuristic function at each step. It is possible to observe that the heuristic function fluctuates as the algorithm explores different states. Initially, the heuristic value tends to be higher as the algorithm is far from the goal. Then it tends to decrease, reflecting the reduced distance to the goal state. In the analyzed case, it is equal to 0

after 306 steps.

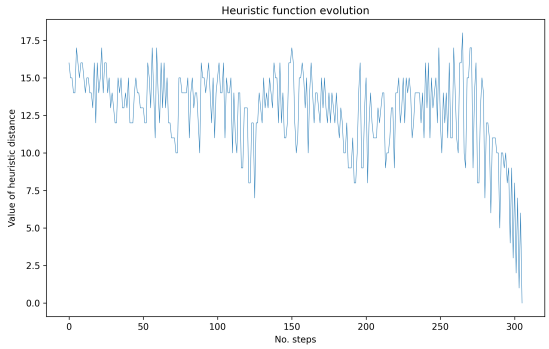

Figure 20: 8-Puzzle: Heuristic function evolution

9.4 Code

In order not to overly burden this document, it was decided to report only the core of the algorithm used, leaving out the initialization of the problem and the implementation of the various heuristics.

The A* (pronounced as "A star") algorithm is used to find the shortest path between the starting node (the initial configuration of the puzzle) and the goal node (the puzzle solved with all the tiles in the correct positions). It uses a heuristic function to help decide which path to follow next.

```
frontier = []
explored = set()
n_actions = 0
heapq.heappush(frontier, Node(initial_state, None, 0,
↪  h_distance))

heuristics_history = []

while frontier:
    n_actions += 1
    current_node = heapq.heappop(frontier)
    heuristics_history.append(current_node.h)

    # puzzle solved
    if np.array_equal(current_node.state, GOAL_STATE):
        path = []
        node = current_node
        while node.parent != None:
            path.append(node.state)
            node = node.parent
        path.append(initial_state)
        print(path[::-1])
```

```
        break

    # Explore neighbors of the current state if the puzzle is not solved yet
    for action in available_actions(current_node.state):
        new_state = do_action(current_node.state, action)
        if tuple(new_state.flatten()) not in explored:
            explored.add(tuple(new_state.flatten()))
            heuristic_distance = heuristic(new_state)
            heapq.heappush(frontier, Node(new_state, current_node, current_node.g + 1, heuristic_distance))

# Print results
print(f"\nSolve {PUZZLE_DIM*PUZZLE_DIM-1}-puzzle in {len(path)} moves")
print(f"{n_actions} evaluated actions")
```

Instead of solving one complex problem (go from an initial configuration to an end configuration), it may be convenient to decompose it in a certain number of smaller and easier sub-problems. The key idea is to move each tile in the correct position, one at a time. This step is repeated until all the tiles are in their correct positions. e.g. The first subgoal consists of placing tile number 1 in its correct position (0, 0). The second subgoal consists of placing tile number 2 in its correct position (0, 1).

```
n_actions_total = 0 # total number of actions evaluated (cost)

def solve_subgoal(initial_state: np.ndarray, target_number: int) -> tuple[np.ndarray, int]:
    target_position = CORRECT_POS[target_number]

    frontier = []
    explored = set()
    heapq.heappush(frontier, Node(initial_state, None, 0, heuristic(initial_state)))

    n_actions_dec = 0
    while frontier:
        n_actions_dec += 1
        current_node = heapq.heappop(frontier)

        # If the target number is already in the correct position
        if tuple(np.argwhere(current_node.state == target_number)[0]) == target_position:
            return (current_node.state, n_actions_dec)

        # Explore neighbors
        for action in available_actions(current_node.state):
            new_state = do_action(current_node.state, action)
            if tuple(new_state.flatten()) not in explored:
```

```
23                  explored.add(tuple(new_state.flatten()))
24                  heapq.heappush(frontier, Node(new_state,
                    ↪  current_node, current_node.g + 1,
                    ↪  heuristic(new_state)))
25
26      return (initial_state, n_actions_dec) # if impossible to
        ↪  solve
27
28  def solve_n_puzzle(initial_state: np.ndarray) ->
    ↪  tuple[list[np.ndarray], int]:
29      current_state = initial_state
30      n_actions_total = 0
31      steps = []
32
33      # Solve small sub-problem for each number
34      for target_number in range(1, PUZZLE_DIM**2):
35          current_state, n_actions_dec =
            ↪  solve_subgoal(current_state, target_number)
36          n_actions_total += n_actions_dec
37          steps.append(current_state)
38
39      return (steps, n_actions_total)
```

9.5 Obtained Reviews

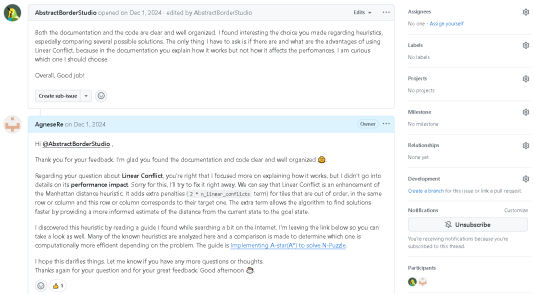

Figure 21: Issue of my lab3 repo - by *AbstractBorderStudio*

10 [2024-dec-01] REVIEW: N-Puzzle Problem

Today, I opened two issues: one on Nicolò (*CodeClimberNT*) repo and one on Simone (*simotmm*) repo. For an easier reading, I report directly the text here. Screenshots are available immediately afterwards (*Figure 22* and *Figure 23*).

- **On CodeClimberNT repo**

  Hello Nicolò,

  A big **thank you** to you for sharing this project with us and for making it public. To me, it's a GitHub repository on par with those often attached

to scientific papers written by researchers or university professors. Maybe it's just my impression, but I'm not joking. Your work is a source of inspiration for me to do better. Truly, congratulations 🦸.

Quality and Clearity

Your code looks clean and easy to understand, thanks in part to the markdown blocks in your Jupyter Notebook. Not to mention the README, simply neat and concise. I agree with you that visualizing the output (in this case with an animated GIF) is crucial for understanding problems like these, where there may be initial obstacles. I'm speechless, I really can't say anything more. It's clear the effort you've put into this. Well done!

Constructive Criticism (or Lack Thereof)

Probably for an issue to be useful and constructive, it should include some criticism. The problem is, I don't see anything to correct, maybe because I don't have the same expertise as you. Wait... maybe I can come up with one critique. Thanks for mentioning Professor *Keith Conrad*'s writing, The 15-Puzzle (and Rubik's Cube), but... I haven't read it and I believe it will make for a wonderful companion tonight before bed. A healthy and intriguing read. Because of you, I probably won't get any sleep tonight — I'll be too absorbed in reading it and thinking about puzzles! I'm joking of course.

Have a great Sunday Nicolò 🧩.

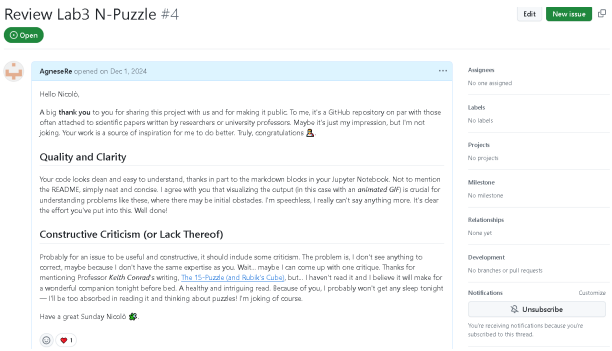

Figure 22: My review of lab3 - *CodeClimberNT* repo

- **On simotmm repo**

  Hello Simone 👋,

  Your work is well-organized and clearly structured. You've done an excellent job in setting up your Jupyter Notebook and providing a README file that contains all the relevant information about your program. This makes everything easy to follow and understand.

  Considerations

  The core of your algorithm is the `search` method, which accepts several parameters, including the `priority_function`. This design aspect is crucial because it provides great **flexibility** and **modularity** to the entire

algorithm. In fact, you call the search method twice, once for implementing $A*$ and once for Breadth-First Search. I really appreciate your choice because it allows you to adapt the behavior of the algorithm to solve a wide variety of problems, without modifying the core logic of the code. Great work! One hundred points for you 😎.

I noticed that for **A*** algorithm, you are using the number of misplaced tiles, the so-called Hamming Distance, as the heuristic. This is a good choice because it's very simple and easy to implement. However, it can become very slow for larger problems (even for 15-puzzle), as it may require exploring a large number of states to reach the goal. If I may offer you some advice, consider experimenting with other heuristics, such as the **Manhattan distance** or the **Linear Conflict** heuristic. These can often provide better performance by guiding the search more effectively and reducing the number of states explored. I'll leave you a simple guide (Implementing A-star(A*) to solve N-Puzzle) in which an analysis of some possible heuristics for solving the n-puzzle with A* is provided. It helped me a lot, and I want to share it with you. I'll also provide you a possible implementation of the Manhattan distance for your code. Tell me what you think about it and let me know if anything is unclear.

```
def manhattan_heuristic(state: State) -> int:
distance = 0

current_data = state._data
goal_data = GOAL._data

# loops over each tile of the puzzle
for i in range(current_data.shape[0]):
    for j in range(current_data.shape[1]):
        tile = current_data[i, j]

        if tile == 0:    # ignore empty tile
            continue

        goal_row, goal_col = np.where(goal_data == tile)
        goal_row, goal_col = goal_row[0], goal_col[0]

        distance += abs(i - goal_row) + abs(j - goal_col)

return distance
```

Conclusions

Beyond my suggestions, I would like to congratulate you on your work. Although there is a small typo in the final part of the README (you missed an 'a' in "It lways finds the shortest solution in terms of the number of actions" - I'm a bit picky, but it's really a shame to leave a typo in such a well-done piece of work like yours), your work is fantastic. You've provided me with some excellent points for reflection on how to solve this problem in an even better way, too. If you have any doubts or simply

want to discuss the **N-Puzzle** problem, feel free to contact me. Goodbye and happy Sunday! 👋.

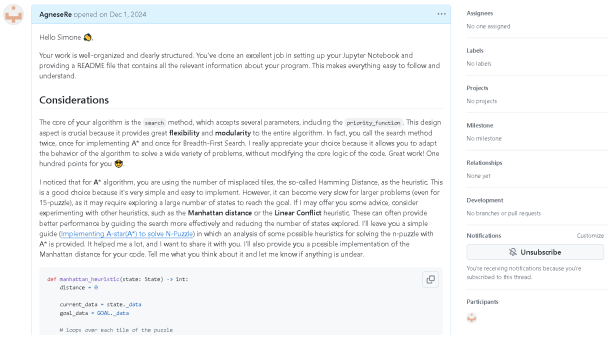

Figure 23: My review of lab3 - *simotmm* repo second part

11 [2024-jan-07] PRESENTATION: Fuzzing

During the Christmas break, I spent some time to better understand how **fuzz testing** works and how it can be used to detect bugs and vulnerabilities in software. A topic that was only mentioned during the course lessons, but that fascinated me right away. After having searched for a detailed definition of the term and having understood its origins[1], I tried to fully understand its functioning and investigate the different existing typologies (**fuzzers** classified on the basis of the inputs generation, on the knowledge of the program structure and on the knowledge of the input structure). Finally, I tried to implement fuzz testing for an authentication handler function in **Go** language. It was the first time I made a program to implement fuzzing and the first time I wrote code in Go language. The project repository is publicly available on my Github account, but I'll report a piece of code here anyway to give you an idea. Before doing this, I report here the first part of the README only to have a general idea.

> This project implements *fuzz testing* for an authentication handler function, `AuthHandler`. The main goal of the fuzz testing is to ensure that the handler functions work correctly under different input conditions, including valid, invalid and malicious inputs (*e.g.* SQL Injection).
>
> The authentication handler checks username and password, ensuring they are valid and meet specific criteria. This repository provides a testing suite using Go's fuzz testing capabilities to verify the handler's behavior under various scenarios.

`FuzzAuthHandler` function performs fuzz testing for the `AuthHandler` function. The first function leverages **Go's built-in fuzzing capabilities** to test the authentication handler against a variety of input scenarios, including valid, invalid, and malicious inputs. The goal is to ensure that `AuthHandler` behaves as expected under different conditions. The AuthHandler function accepts an HTTP POST request containing a JSON payload with username and password.

[1] Prof. Barton Miller said: "I wanted a name that would evoke the feeling of random, unstructured data. After trying out several ideas, I settled on the term fuzz."

```go
func FuzzAuthHandler(f *testing.F) {

	srv := httptest.NewServer(http.HandlerFunc(AuthHandler))
	defer srv.Close()

	// Fixed test cases (e.g. empty fields and SQL Injection)
	testCases := []AuthRequest{
		{"user1", ""},
		{"user1", "psw"},
		{"user1", "password1"},
		{"me", "password12345"},
		{"user!@#", "passw0rd!"},
		{"' OR '1'='1", "password123"},
		{"admin", "' OR '1'='1' --"},
		{"admin' OR '1'='1' --", "admin' OR '1'='1' --"},
		{"admin' UNION SELECT NULL, NULL --",
		↪  "password123"},
		{"admin' AND 1=1 --", "password123"},
		{"user2", "bozTS5tLf6mfVLFxn7SFxJxdfEBb9sX"},
	}

	// Add test cases to the fuzzer
	for _, testCase := range testCases {
		data, _ := json.Marshal(testCase)
		f.Add(data)
	}

	// Fuzzing execution
	f.Fuzz(func(t *testing.T, data []byte) {

		t.Logf("Generated input: %s", data)

		if !json.Valid(data) {
			t.Skip("Invalid json format")
		}

		authReq := AuthRequest{}
		err := json.Unmarshal(data, &authReq)
		if err != nil {
			t.Skip("Invalid json data: " +
			↪  err.Error())
		}

		// Send POST request to http test server
		resp, err := http.DefaultClient.Post(srv.URL,
		↪  "application/json", bytes.NewBuffer(data))

		if err != nil {
			t.Errorf("Error reaching HTTP API: %v",
			↪  err)
```

```
47                  }
48
49                  if resp.StatusCode == http.StatusBadRequest {
50                          t.Logf("Expected error for invalid
                            ↪  input")
51                  } else if resp.StatusCode ==
                    ↪  http.StatusUnauthorized {
52                          t.Logf("Authentication failed for invalid
                            ↪  credentials")
53                  } else if resp.StatusCode == http.StatusOK {
54                          var response string
55                          if err := json.NewDecoder(resp.Body).
56              Decode(&response); err != nil {
57                                  t.Errorf("Error decoding
                                    ↪  response: %v", err)
58                          }
59                          t.Logf("Response: %s", response)
60                  } else {
61                          t.Errorf("Unexpected status code %d",
                            ↪  resp.StatusCode)
62                  }
63
64                  defer resp.Body.Close()
65          })
66  }
```

Below are some slides from the presentation. The first slide (*Figure 24*), a slide regarding the advantages of fuzzing (*Figure 25*), and a slide about different types of fuzzers (*Figure 26*). Although I didn't have the opportunity to present fuzzing to my colleagues, this in-depth analysis allowed me to discover a topic of interest that I will certainly explore more in the future.

Figure 24: Fuzz Testing: Random Inputs for Breaking Things

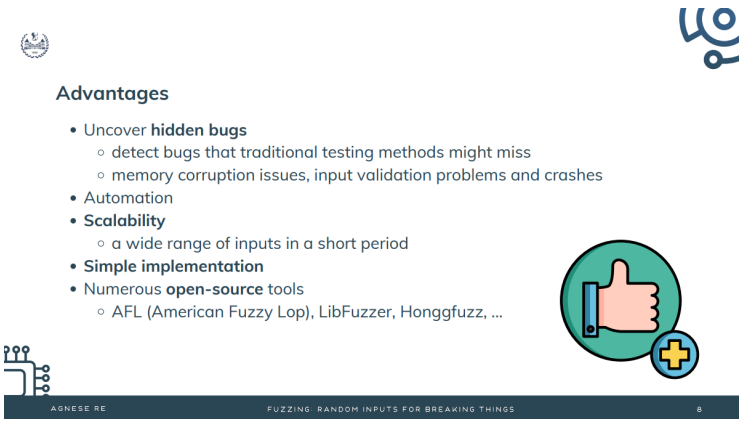

Figure 25: Fuzz Testing: Advantages

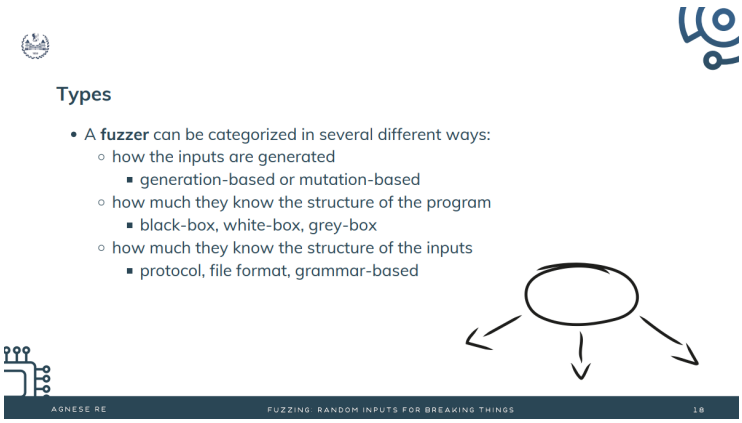

Figure 26: Fuzz Testing: Different Types of Fuzzers

12 PROJECT: Symbolic Regression with GP

This project was developed in collaboration with my colleagues Pasquale Ferraiuolo, Dennis Gobbi, and Xhoana Shkajoti. Their GitHub names are respectively *FerraiuoloP*, *GDennis01*, and *XhoanaShkajoti*. The logic implemented in the code is the same. Compared to the work developed together, only two small changes have been made:

- The `match-case` statement used in the `select_parents` function was replaced with a simple `if-else` statement, in order to ensure compatibility with Python 3.9 and earlier versions;
- In the context of the island model and migration events, I ensured that individuals always migrate to an island different from their current one. Previously, it was possible to migrate to the same island, resulting in no actual movement.

12.1 Description of the Problem

The project focused on the Symbolic Regression problem, a powerful machine learning technique that searches the space of mathematical expressions in order to find the model that best fits a given dataset, both in terms of accuracy and simplicity. More formally, given a dataset consisting of input-output pairs in the form (X, y), the task is to uncover a mathematical expression that closely approximates the relationship and the hidden patterns within the data.

This project involves working with eight distinct datasets, each containing a varying number of input variables and output values. The challenge is to identify the formula that most effectively fits the data, ensuring both high precision in the predictions and a compact, easily understandable final structure.

12.2 Implementation

The problem was tackled using an evolutionary algorithm. To efficiently represent mathematical formulas, a **Tree** structure was implemented with each formula constructed of smaller blocks organized within a **Node** structure. Each Node is characterized by its type (binary operator `B_OP`, unary operator `U_OP`, variable `VAR` or constant `CONST`), its value and its left and right children. A wide range of operators is adopted, covering both basic arithmetic and more advanced mathematical functions. Addition, subtraction, multiplication, division, maximum, minimum, module, absolute value, square root, sine, cosine, exponential, etc. Constants are randomly generated within a specified range of -10 to 10. A Tree, which can be either grow or full, is considered valid if it has a computable fitness, *i.e.* no division by zero, and no numerical overflow.

More details on the population initialization, mutation and crossover techniques used in the evolutionary process are provided in the following subsections.

12.2.1 Key Features

- **Island Model Genetic Algorithm**
  - The population is divided into a certain number of **subpopulations** (a.k.a. *islands*) which evolve separately and can occasionaly exchange individuals (a.k.a *migration*) in order to avoid local optima and rapid convergence;
  - The migration happens in a probablistic way. At every iteration a random individual can be selected to migrate from an island to another one;
  - At each migration event, according to a migration rate parameter, one or more individuals migrate from the source island to another random island, as a way to ensure equal chance of genetic mixing across islands.
- **Tree-Based Representation**
  - The evolutionary algorithm iteratively evolves a population of mathematical formulas, represented as full and grow trees;

  - Internal nodes are randomly chosen from function set (arithmetic, trigonometric, logarithmic and exponential operators), while leaves are randomly chosen from terminal set (constants and variables);
  - Each tree contains at least one instance of each variable of the problem dataset;
  - Each tree must be consistent with the core principles of the mathematical system (e.g. no divisions by 0).

- **Elitism**
  - To preserve high-quality solutions, the best individuals (a.k.a. elites) are directly inserted into the next generation, without being subjected to any change.

- **Parents Selection**
  - Different parents selection strategies are implemented. Fitness-proportional, rank based and tournament selection. If not specified, the default rank-based strategy is used.

- **Mutation and Crossover**
  - Various mutation mechanisms are implemented. Replaced a subtree with a new one (`mutate_subtree`) or modify a single node (`mutate_single_node`) in the selected parent tree;
  - Combine two different trees for generating new offsprings. This allows the algorithm to explore new regions in the search space, encouraging exploration instead of exploitation.

- **Takeover Detection**
  - It is possible that a particularly suitable individual, with an important competitive advantage, dominates the population. Its offspring spread rapidly in the population and genetic diversity decreases. To address this issue, the population is trimmed by keeping alive only unique individuals and re-generating the remaining ones;
  - In the proposed algorithm, takeover occurs when 50% of the population exhibits the same fitness.

12.2.2 How it works

- **Initialization**
  - A population of individuals (*trees*) is initialized on each island. Depending on the value assigned to the variable `grow_full_ratio`, each island's population is initialized with a number of full trees and grow trees. Although the literature often suggests a grow full ratio of 0.50, the proposed solution adopts a ratio of 0.95. This approach allows for more diverse and flexible initial solutions, leading to better exploration of the search space and avoiding a bloating phenomenon, tipycal of the generation of large trees;

  - In each of the `ISLAND_NUM` islands, there are `ISLAND_POPULATION` individuals.

- **Selection**
  - Parents are selected based on their fitness (in which measure the mathematical formula represented by the tree fits well the data provided as input), using various strategies (*e.g.* rank-based selection).

- **Reproduction**
  - Offsprings are generated through mutation and crossover genetic operators (0.65/0.35 ratio).

- **Evolution**
  - Over the course of generations, populations on the islands evolve and only the best performing trees survive.

- **Convergence**
  - The evolutionary process continues for a certain number of generations (`MAX_GENERATIONS`).

12.3 Results

The algorithm was tested on eight different input problems. For each problem, the final formula derived by the Symbolic Regression model, and the Mean Squared Error (MSE) are provided to assess the accuracy of the solution.

> Increasing the number of generations leads to progressively better formulas with lower MSE. However, since this process requires a large amount of time — sometimes exceeding two hours — it was decided to limit the maximum number of generations to 5000. This allows the process to be completed in acceptable times while still achieving reliable results.

12.3.1 Problem 1, Shape : (1,500)(500,)

The final formula found by the model for the first problem is:

```
def f1(x: np.ndarray) -> np.ndarray:
    return np.sin(x[0]) # final formula
```

The corresponding final Mean Squared Error (MSE) is: **7.125940794232773e-34**. An extremely low value that confirms the formula effectively captures the relationship between the input and output values. The algorithm takes approximately 5 minutes to complete all 5000 generations, but it is worth noting that a good result, on the order of 10^{-34} is already achieved within a few seconds, as early as the 100th iteration.

12.3.2 Problem 2, Shape: (3,5000)(5000,)

The final formula[2] found by the model for the second problem is:

```
def f2(x: np.ndarray) -> np.ndarray:
    return (((((x[1] * 1.0133379750665732) * -19.919416233677293)
    ↪  + ((-9.790571207340218 * x[2]) + (-9.735723057413209 *
    ↪  x[2]))) * ((1711.0001133355208 * np.cosh(x[0])) +
    ↪  -40620.92739861885)) + ((-140.73135021910855 - ((x[2] *
    ↪  x[1]) * -10.82318841194864)) * ((36.04153204685458 /
    ↪  (1.0133379750665732 / x[0])) * -216.83639344361913)))
```

The corresponding final MSE is: **6767157331999.958**, an high value that indicates the formula does not accurately represent the relationship between the input and output values. This suggests that the existing relationship between the variables is highly complex, and the algorithm struggles to uncover it.

12.3.3 Problem 3, Shape: (3,5000)(5000,)

The final formula found by the model for the third problem is:

```
def f3(x: np.ndarray) -> np.ndarray:
    return np.arctan(np.arctan(1.9423158330970622) +
    ↪  8.311556888397728) *
    ↪  np.cosh(np.cosh(np.arctan(1.9423158330970622))) -
    ↪  (np.square(x[0]) * np.rint(-2.065621497047198) +
    ↪  3.503326125821749 / np.reciprocal(x[2]) + x[1] *
    ↪  np.square(x[1]))
```

The corresponding final MSE is: **9.059643246177774e-05**.

12.3.4 Problem 4, Shape : (2,5000)(5000,)

The final formula found by the model for problem 4 is:

```
def f4(x: np.ndarray) -> np.ndarray:
    return 3.3325688447227532 + (np.cos(x[1]) /
    ↪  0.1428298184783164) - ((-0.5653322717250981 - x[0]) /
    ↪  -10.9872282724623779)
```

The corresponding final MSE is: **3.065305250369384e-06**.

12.3.5 Problem 5, Shape : (2,5000)(5000,)

The final formula found by the model for problem 5 is:

```
def f5(x: np.ndarray) -> np.ndarray:
    return ((2.467100421154991e-12 * (np.power(4.18958630424536,
    ↪  (3.017249747918303 - x[1])) - 4.084409304778444)) *
    ↪  np.power(x[0], x[1]))
```

The corresponding final MSE is: **1.182264470334063e-20**.

[2]The formula is reported in its collapsed version (use of the `to_np_formula` method with `use_std_operators` parameter equal to `True`). All possible calculations are performed (this justifies the presence of constants outside the range [-10, 10]) and the operators `np.add`, `np.subtract`, `np.multiply`, `np.divide` are replaced with +, -, *, / respectively.

12.3.6 Problem 6, Shape : (2,5000)(5000,)

The final formula found by the model for problem 6 is:

```
def f6(x: np.ndarray) -> np.ndarray:
    return (x[1] - (((-3.7176133491516232 * x[0]) +
    ↪  (3.7181064589095936 * x[1])) /
    ↪  (np.minimum(4.381559519733427, -3.803901432200341) -
    ↪  np.cbrt(3.7181064589095936))))
```

The corresponding final MSE is: **1.7729341707810 65e-08**.

12.3.7 Problem 7, Shape : (2,5000)(5000,)

The final formula found by the model for problem 7 is:

```
def f7(x: np.ndarray) -> np.ndarray:
    return ((np.sin((-4.287237004344295 - (x[0] - x[1]))) +
    ↪  np.sin(np.power(1556.6675533539744, (x[0] - x[1])))) *
    ↪  (np.exp((1.6053878748495762 + (x[0] * x[1]))) *
    ↪  np.sin((2232.504710869893 + (x[0] - x[1])))))
```

The corresponding final MSE is: **194.70615387010136**. The high MSE suggests that the formula does not accurately capture the relationship between the input and output values.

12.3.8 Problem 8, Shape : (6,50000)(50000,)

The final formula found by the model for problem 8 is:

```
def f8(x: np.ndarray) -> np.ndarray:
    return np.minimum(np.sinh(x[5]) *
    np.maximum(9.3067551639483366, x[0] / x[0] -
    ↪  8.1737401391616195) *
    np.maximum(np.maximum(np.minimum(np.cosh(x[5]),
    ↪  18.4423895407085805),
    np.minimum(np.square(x[5]), 15.037856692023084)),
    np.minimum(17.6972179719256010, np.square(x[5])) +
    np.minimum(np.square(x[5]), np.absolute(x[5]))) +
    np.minimum(np.sinh(x[4]) * np.maximum(37.0927078418996027,
    ↪  x[3]),
    np.maximum(np.sinh(x[5]),
    9.720508873329518) / 2.564537561212518 +
    np.sinh(x[4]) * -36.5491997365694977)
    ↪  -np.minimum(np.maximum(np.remainder(np.power(
    np.maximum(3.025150828938454, np.sin(x[1])),
    np.remainder(x[4], -4.190526875841163) + 2.5852944295947005),
    np.minimum(np.sinh(x[4]) + np.sinh(x[5]),
    np.minimum(6.530635069831985, x[2]) - 15.751711154922196)),
    np.maximum(-39.1515397544814214, 314.8402983836045133 /
    np.maximum(-4.933777461347297, x[3])) +
    (np.maximum(-9.174199129628121, x[3]) *
    ↪  np.exp(4.583029845151287)) /
```

```
    -1.5537384634524341),
    np.remainder(np.maximum(100.0199410769763602 /
(x[3] - 2.1332481808545545), -32.7772659233697216),
np.minimum(36.7506213572168948 * np.cosh(x[5]),
    -4.109412935321963 *
np.maximum(-9.62084356692828, x[3])) * 17.361335511767116)),
np.minimum(np.maximum(9.174199129628121, x[3]) +
    np.sinh(x[5]),
np.cosh(x[5]) + np.cos(x[5])) * np.minimum(9.720508873329518,
np.square(x[5])) + np.minimum(4.583029845151287,
    np.absolute(x[5])) +
np.maximum(44.8278193425921329 / np.absolute(x[5]),
    np.absolute(x[5]) +
6.695358044391064) * np.maximum(np.sinh(x[5]) + np.cos(x[5]),
-0.7294152703515632) * 8.680667755883558 -
np.minimum(np.square(np.maximum(-4.109412935321963 *
    np.sinh(x[5]),
np.minimum(17.6972179719256010, np.square(x[5])) +
np.minimum(np.square(x[5]), np.absolute(x[5])) )),
np.square(np.minimum(np.maximum(np.maximum(np.sinh(x[5]),
-7.8797548932089201), np.minimum(np.square(x[5]),
np.absolute(x[5]))), np.sinh(x[5]) +  0.6904692721820371 ))))
```

The corresponding final MSE is: **7752.954419575954**. A high value and an anything but compact formula that suggest the algorithm's inability to capture the underlying relationship between the input variables 😞. This may indicate that the chosen model is not suited to the problem, that the dataset contains significant noise, or that further tuning and feature selections are needed to improve performance. Perhaps a more compact formula with a slightly higher MSE would have been preferable, but it was decided to report the one with the lowest MSE, despite its complexity.

12.4 Code

Below is the implementation of the core method of the evolutionary algorithm. For the complete project, refer to the CI2024_project-work GitHub repo.

```python
def evolve(self, verbose=False, use_std_operators=False):
    best_tree_island = np.full(self.island_num, None,
    dtype=object)
    best_fitness_island = np.full(self.island_num, np.inf)
    global_best_fitness = np.inf
    global_best_tree = None
    take_over = np.full(self.island_num, False)
    for i in range(self.island_num):
        self.population[i].sort()

    # Main loop (for all generations)
    for generation in tqdm(range(self.max_generations)):
        # Takeover detection in island i
        for i in range(self.island_num):
```

```
14                  if take_over[i]:
15                      self.population[i] =
                        ↪  np.unique(self.population[i])
16                      new_trees = np.array([Tree("grow") for _ in
                        ↪  range(self.population_per_island-
17                      len(self.population[i]))])
18                      self.population[i] =
                        ↪  np.concatenate((self.population[i],
                        ↪  new_trees))
19                      self.population[i].sort()
20
21                  if np.random.rand() < self.migration_rate and
                    ↪  self.island_num > 1:
22                          # pick a random island to migrate to
23                          island_to_migrate = np.random.randint(0,
                            ↪  self.island_num)
24                          while i == island_to_migrate:
25                              island_to_migrate =
                                ↪  np.random.randint(0,self.island_num)
26                          # select a random tree in the island (index)
27                          random_index =
                            ↪  np.random.randint(0,len(self.population[i]))
28                          # move the tree from one island to the other
29                          self.population[island_to_migrate]=np.append(
30                          self.population[island_to_migrate],
31                          self.population[i][random_index])
32
33                          self.population[i]=np.delete(
34                          self.population[i], random_index)
35
36                      if(verbose):
37                          print(f"Migration at {generation} gen from
                            ↪  {i} to {island_to_migrate}")
38
39
40                  new_population = self.offspring_generation(island=i)
41
42                  self.population[i] =
                    ↪  np.concatenate((self.population[i],
                    ↪  new_population))
43                  self.population[i].sort()
44
45                  generation_best_fitness_island =
                    ↪  self.population[i][0].fitness
46
47                  if generation_best_fitness_island <
                    ↪  best_fitness_island[i]:
48                      best_fitness_island[i] =
                        ↪  generation_best_fitness_island
49                      best_tree_island[i] = self.population[i][0]
```

```
50
51
52                  if(best_fitness_island[i] < global_best_fitness):
53                      global_best_fitness = best_fitness_island[i]
54                      global_best_tree = best_tree_island[i]
55                      self.best_fitness_history.append((
56                          best_fitness_island[i], generation))
57
58
59              # trim the population to the best island_population
60              self.population[i] =
                ↪  self.population[i][:self.population_per_island]
61
62              n_best = [elem for elem in self.population[i] if
                ↪  elem.fitness == self.population[i][0].fitness]
63              take_over[i] = False
64              if len(n_best) > 0.5 * self.population_per_island:
65                  take_over[i] = True
66
67              if(generation%100 == 0 and verbose):
68                  print(f"Generation {generation + 1}, Island: {i},
                    ↪  Best Fitness: {best_fitness_island[i]}, Best
                    ↪  Formula: {best_tree_island[i].to_np_formula(
69                  use_std_operators=use_std_operators)}")
70
71              if global_best_fitness <= 1e-33:
72                  break
73
74          if(generation%100 == 0 and not verbose):
75              print(f"Generation {generation + 1}, Best Fitness:
                ↪  {global_best_fitness}, Best Formula:
                ↪  {global_best_tree.to_np_formula(
76              use_std_operators=use_std_operators)}")
77
78      return global_best_tree, global_best_fitness
```

13 Activity Log

- **[2024-sep-25] INFO: Tweak, word of the day**

  I have understood that great results do not come from drastic changes (exploration). It is through small, conscious steps that we explore the world, the possibilities that it holds for us and that we reach our goals (exploitation). We start from the current solution and we explore the neighborhood, **tweaking**, making only minor adjustments. It may not be the best solution (global optimum), but it's still a step forward (local optimum).

- **[2024-sep-26] INFO: 1-Max, first expriments**

  I'm starting to get more into the **Hill Climbing** mindset. Repeating

concepts a second time and adding new ones is always a good idea. This afternoon I rewrote my notes (on a piece of paper, I'm a future ancient computer engineer) and prepared some code for the **1-Max** problem to try. Tomorrow I'll turn the computer on and I'll try it. I'm very curious of the result.

- **[2024-sep-30] INFO: Fitness landscape**
- **[2024-oct-02] INFO: Beyond Hill Climbing: Annealing and Tabu**
- **[2024-oct-03] INFO: Set Cover Problem: Fitness and Mutation**
- **[2024-oct-07] INFO: Self Adaptation and Parameter Adjustment**
- **[2024-oct-10] INFO: The Power of Evolutionary Algorithms**
- **[2024-oct-14] INFO: Parents, Offsprings and Recombination**
- **[2024-oct-16] INFO: Understanding Parent Selection**
- **[2024-oct-17] INFO: First EA implementation in Python**
- **[2024-oct-21] INFO: Deep Dive into EA: Hypermodern GA**
- **[2024-oct-23] INFO: Travelling Salesman Problem Description**
- **[2024-oct-24] INFO: A Greedy Approach for TSP**
- **[2024-oct-28] INFO: Exploring Mutation and Crossover**
- **[2024-oct-31] INFO: A quick intro to PSO**
- **[2024-nov-04] INFO: EA and Lack of Diversity Problem**
- **[2024-nov-06] INFO: Lexicase Selection and Island Model**
- **[2024-nov-07] INFO: A Path like a Sequence of Decisions**
- **[2024-nov-11] INFO: Breadth vs. Depth First**
- **[2024-nov-14] INFO: A* Algorithm: What A Discovery**
- **[2024-nov-18] INFO: Searching For a Formula**
- **[2024-nov-21] INFO: An In-Depth Exploration of GP**
- **[2024-nov-27] INFO: Symbolic Regression using GP**
- **[2024-nov-28] INFO: Model-Based Reinforcement Learning**
- **[2024-dec-05] INFO: Random Policy vs. Greedy Policy**
- **[2024-dec-09] INFO: Model-Free Reinforcement Learning**
- **[2024-dec-12] INFO: Adversarial Search**
- **[2024-dec-16] INFO: Stochastic, Zero and Non-Zero Sum Games**
- **[2024-dec-19] INFO: John Holland's LCS**
- **[2025-jan-07] BONUS: The 8 Queens Puzzle: A Royal Challenge**
- **[2025-jan-09] BONUS: Six Friends and Q-Learning**

14 Conclusions

This course has provided me with valuable insight into a wide range of algorithms, as well as the knowledge necessary to design and implement intelligent solutions to problems where a direct solution may be impractical or unknown. The laboratories, the project work, and even more the reviews I received from my colleagues were particularly useful. They allowed me to analyze problems from different points of view, perhaps placing emphasis on aspects that I had overlooked and deepening my understanding. I'm excited to carry these lessons with me, applying the skills and insights I've gained to future projects and activities, and knowing that the journey of learning is one that never truly ends.

> "We can only see a short distance ahead, but we can see plenty there that needs to be done."
> — *Alan Turing*