CI2024_lab1

To solve the Set Cover optimization problem, I tried implementing several local search algorithms to be able to compare the solutions they found in different scenarios of the problem:

- Random Mutation Hill Climbing
- Hill Climbing with a more powerful tweak
- Hill Climbing with a more powerful tweak and 1 out of 5 rule
- Simulated Annealing
- Iterated Local Search
- Iterated Local Search with Simulated Annealing
- Tabu Search
- Iterated Local Search with Tabu Search
- Iterated Local Search with 1 out 5 rule

To solve the instances of problem 1-3, which are characterized by a smaller universe size and fewer sets, I decided to set the number of steps to 10,000. For the instances of problem 4-6, I chose to reduce the number of steps to 100 in order to limit the execution time for each algorithm to a few minutes. I made this decision because, with a higher number of iterations, I could have further improved the fitness obtained by each algorithm but I would have had to wait for hours. Ultimately, what matters in this analysis is the relative effectiveness of the algorithms in different scenarios rather than the precise search for the best possible fitness achieved by each algorithm.

1. As for the instance 1 of the problem, none of the implemented algorithms are able to improve the initial solution, which was generated by taking all the available subsets.
2. As for the instance 2 of the problem, the algorithms that achieved the highest fitness are:

- Iterated Local Search
- Iterated Local Search with Simulated Annealing
- Iterated Local Search with Tabu Search

3. As for the instance 3 of the problem, the algorithm that achieved the highest fitness is:

- Hill Climbing with a more powerful tweak and 1 out of 5 rule

4. As for the instance 4 of the problem, the algorithms that achieved the highest fitness are:

- Iterated Local Search
- Iterated Local Search with Simulated Annealing
- Iterated Local Search with Tabu Search

5. As for the instance 5 of the problem, the algorithms that achieved the highest fitness are:

- Iterated Local Search
- Iterated Local Search with Simulated Annealing
- Iterated Local Search with Tabu Search

6. As for the instance 6 of the problem, the algorithms that achieved the highest fitness are:

- Iterated Local Search
- Iterated Local Search with Simulated Annealing
- Iterated Local Search with Tabu Search
- Iterated Local Search with 1 out 5 rule

Overall, analyzing the behavior of the different algorithms in various scenarios, if I had to choose an algorithm to move forward with, I would opt for **Iterated Local Search with Simulated Annealing**.

Using **Iterated Local Search** where each local search phase is performed by **Simulated Annealing** offers several advantages, particularly in terms of balancing exploration and exploitation in optimization problems. Simulated Annealing is designed to escape local minima by accepting worse solutions with a certain probability, especially at higher temperatures. When combined with ILS, this allows the algorithm to explore a broader range of solutions, making it less likely to get stuck in suboptimal areas of the search space. ILS restarts the search process after each local search, potentially moving to entirely different regions of the search space. With SA guiding the local search, each restart begins from a diverse area, enhancing the overall exploration ability of the algorithm.

Simulated Annealing starts with a higher level of exploration (accepting worse solutions) and progressively focuses more on exploitation as the temperature decreases allowing the algorithm to fine-tune solutions as it converges. This provides a smooth transition from wide exploration to focused optimization. Within ILS, this feature allows each local search phase to refine the solutions more effectively as SA ensures that even the local searches explore sufficiently before focusing on local improvements.

The combination of ILS and SA creates a dynamic balance between local and global search. ILS helps to move across different regions of the solution space (global search) while Simulated Annealing allows for deep, focused exploration of each region (local search). This results in an algorithm that can avoid premature convergence on suboptimal solutions and increases the likelihood of finding global optima.

SA is known for its robustness in navigating complex fitness landscapes with many local minima. Its probabilistic acceptance criteria allow it to jump out of local minima, even in highly non-convex or noisy problem landscapes. ILS benefits from this ability because each local search is capable of making meaningful progress even in rugged fitness landscapes, improving the overall efficiency of the algorithm in solving challenging optimization problems.

By incorporating the temperature control mechanism of Simulated Annealing, ILS can dynamically adjust the search precision. Higher temperatures lead to larger steps and broader exploration while lower temperatures allow for more precise adjustments and exploitation. This feature is especially useful for multi-modal optimization problems, where diverse regions of the search space need to be explored first followed by a fine-tuning process.

The main strength of Simulated Annealing within the ILS framework is its ability to escape from local minima. In traditional local search methods like Hill Climbing, once a local minimum is reached, the search may stagnate. SA, however, continues to accept worse solutions with some probability, giving it the potential to move away from local minima and explore new regions. This escape mechanism increases the robustness of the ILS approach allowing it to deal with complex optimization problems that would otherwise trap simpler local search methods.

ILS operates by restarting the search process after each local search phase and Simulated Annealing provides a progressive refinement mechanism within each phase. After each iteration, ILS uses the results of the

previous local search (SA) to guide further exploration. This iterative process enables gradual improvements over time increasing the chances of finding better solutions as the algorithm progresses.

Suggestions from my colleagues

I received two reviews:

- https://github.com/GiorgioBongiovanni/CI2024_lab1/issues/1

TurcoRiccardo opened on Oct 16, 2024 ...

All the requested code instances were examined on different algorithms, verifying which of these is best suited to solving the problem (better fitness) with the appropriate considerations. These considerations could lead to achieving better results or show the solution better:

- In the Iterated Local Search algorithms I think it is more interesting to show in the graph all the points relating to the steps taken for each iteration and not just the result found in each iteration.
- In the Simulated Annealing algorithm you could add a condition that blocks the modification if the solution found through tweaking makes the solution invalid.
- In the Local Iterated Search with 1 out 5 rule algorithm we can try to use different value of strength for each iteration in order to have more different approaches to the problem.
- In the Tabu Search algorithms the second if condition is redundant, we can simplify the algorithm by removing it.

- https://github.com/GiorgioBongiovanni/CI2024_lab1/issues/2

mattiaottoborgo opened on Oct 22, 2024 ...

The code is well structured and divived into section, which helps the review to better examine the code.
I like the implementation of all the methods explained during lectures, which allowed me to understand better the pros and cons of each algorithm. The graphs are well done and help to understand the evolution of the algorithm.
Really good job!

Suggestions for my colleagues

I wrote one review:

- https://github.com/AleManera/CI2024_lab1/issues/2

The Random Mutation Hill Climbing local search algorithm is reasonably implemented because it uses the tweak function to flip a random set (either adding or removing it).
This is a common, simple mutation strategy in RMHC.
A potential issue here is that the mutation mechanism may be too local, leading to stagnation if the initial solution is close to a local minimum.
Exploring a strategy for escaping local minima, like adaptive mutation (something similar to 1 out of 5 rule) or combining with other techniques (for instance Simulated Annealing), could be worth discussing.
The algorithm uses a for loop over a large number of steps but there is no adaptive stopping condition.
This could be improved by implementing a stopping criterion based on the number of consecutive non-improving iterations or some convergence criterion.
The condition if (UNIVERSE_SIZE < 1000 and NUM_SETS < 100) adjusts the number of steps based on the problem's size, scaling it up for larger problems.

This adaptive scaling is a good feature though the scaling factor MAX_STEPS * 1000 might require further tuning to match the specific nature of different problems.
In order to increase exploration and thus avoid the algorithm getting trapped in the attraction basin of a local optimum, the tweak() function, which applies a modification/perturbation to the current solution, could be adjusted to apply a larger modification like in the following proposal:

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

This function generates an array of random numbers, one for each subset (NUM_SETS), using the function rng.random(NUM_SETS), which produces floating-point values between 0 and 1. By comparing these values with 0.01, a boolean mask (mask) is created, where each element has a 1% probability of being True (and 99% of being False). In other words, 1% of the elements in the mask will be True, while 99% will be False.
Keeping in mind the truth table of XOR, if I choose a low probability such as 0.01 = 1%, each element of the mask has a 1% probability of being True and a 99% probability of being False. An element of the current solution is modified only if the corresponding element of the mask is 1/True.
This means that the lower the threshold (which in this case is 0.01), the lower the number of True values in the mask and therefore the fewer modifications will be applied to the current solution.
Thus, if I want to increase the number of True values in the mask and consequently increase the number of modifications applied to the current solution, I need to choose a value higher than 0.01.
The multiple_mutation function introduces a strategy to improve the exploration of the algorithm.
Instead of making small, single or localized modifications to the current solution (as a classic tweak function would), this function applies multiple mutations spread across the solution.
In practice:

- Each element of the solution has a low probability (1%) of being flipped, introducing scattered and random modifications throughout the solution.
- This allows for the exploration of regions of the solution space further away from the current solution, potentially enabling the discovery of better solutions that would be hard to find with more conservative mutations.

The use of a mask with multiple mutations introduces an element of randomness distributed across the entire solution.
This allows for the exploration of regions of the solution space farther away than a single or localized mutation.
Since more elements of the solution are mutated simultaneously, the function can help avoid getting stuck in local minima.

The 1% probability (defined by the 0.01 threshold) can be easily modified to increase or decrease the level of mutation.
If you want to introduce more mutations you can increase the threshold value or reduce it for a less "perturbative" process.

CI2024_lab2

In developing the evolutionary algorithm needed to determine the closest approximate solution to the optimal one for the TSP problem, I was inspired by the research paper written by professors Tao and Michalewicz. They introduced a new type of crossover operator called "inver-over," which I chose to be the core of my algorithm.

In each generation, for each individual in the population, a city c is randomly selected.

Selection and inversion loop: If the if branch is entered (with probability p), the second city c' is selected from the remaining cities in the first individual. If the else branch is entered (with a probability complementary to p), another random individual is chosen and the second city c' is chosen as the city following c in this second individual.

Adjacency check: Before performing the inversion, it is checked whether c and c' are already adjacent in the first individual. If they are adjacent, the loop is exited. If they are not adjacent, the subsequence between c and c' is inverted in the first individual. After the inversion, c is updated with c', and the loop repeats, selecting a new c' starting from the just updated c.

Fitness Improvement Check: At the end of the loop, if there is an improvement in fitness, the individual in the population is replaced with its improved version.

In practice, the inver-over crossover is representative of the approach used in modern evolutionary algorithms because it doesn't strictly apply a single genetic operator (mutation or crossover), instead, it combines them by controlling their application with a certain probability p. It uses inversion mutation (which isn't too drastic, as it only reverses the order of a subsequence, thus keeping most cities in their original proximity in the sequence, which is crucial in TSP). This approach applies mutation to change the starting configuration while preserving enough connections between cities in a sequence that's good in terms of fitness because these connections are important. The crossover attempts to maintain the adjacency of cities in the inverted subsequence of the first individual, as well as the adjacency of cities c and c' in the second individual. The fact that the inversion boundaries in the first individual are chosen based on the adjacency of c and c' in the second individual ensures that the choice isn't entirely random but rather strategically aimed at being heuristically effective, since it tries to preserve the adjacency of c and c' from the second individual.

The inver-over crossover is designed to apply two types of variation—inversion mutation and crossover—without strictly adhering to just one.
By using a selection probability p, it dynamically chooses which method to apply, balancing exploration (through mutation) and exploitation (through crossover). This combination allows for small adjustments in solutions preserving connections between cities that contribute to a high fitness score. The inversion mutation flips the order of a subsequence, introducing variation without a complete overhaul. This is crucial in the TSP since it allows configuration changes while preserving geographical proximity between nearby cities. Maintaining many of the original connections helps avoid destroying beneficial structures, which is critical for optimization problems where good solutions are often similar to one another.

When crossover is applied by selecting inversion boundaries based on the position of c and c' in the second individual, inver-over avoids randomness by using heuristic information from another individual.
This ensures that the inverted subsequence retains neighborhood relationships that already exist in the

second individual enhancing the chances of improvement through structured rather than entirely random configurations.

The idea of leveraging the relative positions of c and c' in the second individual is based on the heuristic that these adjacent cities likely form a beneficial connection for fitness.
By avoiding drastic changes, the operator can preserve advantageous structures progressively optimizing the starting solution without breaking configurations that might already be close to optimal.

In addition to all this, to ensure that the initial population of 10 individuals from which the evolutionary algorithm starts is not entirely random, I chose to generate 9 individuals randomly through simple permutations and 1 individual as the solution of the greedy algorithm, whose execution is very fast and therefore does not introduce any noticeable delay before the start of the evolutionary algorithm's execution.

In summary, the inver-over crossover exemplifies a modern evolutionary approach: it doesn't strictly follow one genetic operator but modulates crossover and mutation adaptively introducing targeted changes and preserving beneficial connections. This ability to balance exploration with exploitation of information in the population makes inver-over particularly effective for TSP.

The results that I got are the following ones:

Country File	**Distance (km)**
vanuatu.csv	1345.544956473311 km
italy.csv	4207.504730128662 km
russia.csv	33907.84385417768 km
us.csv	41025.28007127058 km
china.csv	56414.62442982023 km

Suggestions from my colleagues

I received four reviews:

- https://github.com/GiorgioBongiovanni/CI2024_lab2/issues/1

First off I want to congratulate you for this solution its a well thought ,well structured and also very efficient. I ran all the instances myself and the results I got are even more optimal than the one you reported in the read me file . You've clearly put a lot of thought into not just solving the problem but making it accessible and understandable, which is especially clear in the README explanations. The use of the inver-over crossover was a really interesting choice, and I can see that it's based in research, which gives the lab a great academic foundation.

Strengths:

I believe the results speak for themselves when it comes to strengths. The algorithm on all the instances comes very close to the optimal value which is exactly what was required a slow but accurate solution.

Robustness: Since TSP problem in our case scaled from low number of cities to a high one the fact that the algorithm is able to perform well in all cases shows how solid and scalable the solution is

Detailed README: The README is a strong point here. It's clear and thorough giving the reasoning behind the algorithm choices. You explain the inver-over crossover operator well, even for those unfamiliar with the concept, and I appreciate that you included your inspiration from Tao and Michalewicz's work. It adds depth and context that makes it more than just code—it's a learning tool. It was very useful to my understanding of this topic.

The initialization: Apart from using inver-over I like that you use a random initialization to account for diversity with a greedy solution so you include a well performing one in the population to speed up the process of convergence.

Code clarity: The code is accompanied with comments which are very useful to understand and follow the logic of the code.

Enhancements:

Visualization: we are able to see the best value in all the generations printed out so we already have a good visualization but it would be great to have a graph(I love graphs :)) so we can visualize the whole process like how fast the the solution improves does it converge or is it ever stuck in an optima.

Early stop: The solution contains a high number of generation (100000) which is great for problems with a high number of cities but useless usage of resources for cases like Vanuatu. Maybe a stopping condition can help in this case.

The fast Solution: I noticed that you didn't implement further from the greedy algorithm we did in class for the fast solution which is great but maybe it would have been interesting to implement one of the methods we have learned prior which are faster than EA and how they compare with inaccuracy with it

I noticed that the code runs for 100 thousand generations which is a very high number but considering that you use a very small population of only 10 individuals which compensates in the computational power needed. Hey if it can run with no problem in my dead laptop it can run perfectly anywhere :).

Final Thoughts

Overall, this TSP lab is an impressive piece of work. It's well organized, well researched, and thoughtfully designed to teach the principles of optimization and algorithmic design. Even without the small enhancements suggested it's powerful resource for learning and experimentation.

Thank you for sharing this, it's clear that you've put a lot of effort into making this both effective and accessible. Great job!

- https://github.com/GiorgioBongiovanni/CI2024_lab2/issues/2

This is a really good job, achieving good results efficiently.

I appreciate the way in which you do the mutation/crossover with inver-over, as it brings novelty compared to traditional evolutionary algorithms that typically rely on a structured parent selection, mutation, and survival selection process.

However, your algorithm may risk slow convergence since it lacks strong selection methods, such as elitism or tournament selection

I propose you a different approach with elitism:

```
def evolutionary_algorithm_tsp():
    population = initialize_population()

    for generation in range(MAX_GENERATIONS):
        offsprings=[]
        for individual in population:
            # Create a new version of the individual using inver-over
            offsprings.append(inver_over(individual, population, p=0.02))
        population.extend(offsprings)
        # Print the best individual of the generation

        population = sorted(population, key=lambda ind: ind.fitness, reverse=True)[:POPULATION_SIZE]
        best_individual = max(population, key=lambda individual: individual.fitness)
        print(f"Generation {generation} - Best fitness: {-best_individual.fitness}")

    population = sorted(population, key=lambda ind: ind.fitness, reverse=True)[:POPULATION_SIZE]

    # Print the best individual of the generation
    best_individual = population[0]
    # Return the best solution found
    print("Best path found:", best_individual.genome)
    print("Distance of the best path:", -best_individual.fitness)
    return best_individual
```

It extends the population with the offsprings, and at each generation select the best POPULATION_SIZE individuals, It tends to find better solutions in fewer generations, though the sorting step slows down the algorithm

Obviously to use my proposal solution you should increase the POPULATION_SIZE and decrease the MAX_GENERATIONS.

Good values are:

POPULATION_SIZE = 1000

MAX_GENERATIONS = 10000
Additionally, implementing an early stopping strategy would be beneficial, as this algorithm typically reaches the best TSP solution in less than 60% of the total generations.

- https://github.com/GiorgioBongiovanni/CI2024_lab2/issues/3
The provided description of the structure of the algorithm is really complete and exhaustive, and also the coiches made in the implementation are very well coded! I have to congratulate you because I used the same techniques for crossover and mutation, but my results were way worse than yours. In general, the code is clearly structured and full of comments that make it perfectly readable and understandable. The only piece of advice I can give you is to try varying the hyperparameters like the population size and the number of generations (I don't know if you have already done it, probably yes) in order to look at how they influence the results and why. Overall, a really great job!!

- https://github.com/GiorgioBongiovanni/CI2024_lab2/issues/4
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.

Conclusions

Congratulations again for your work. The effort you put into it is evident and I believe that hard work always pays off. Thank you very much for sharing your code with us. See you soon. 👋

Suggestions for my colleagues

I wrote two reviews:

- https://github.com/nrbeo/CI2024_lab2/issues/1

The TSP problem focuses on finding the shortest Hamiltonian cycle in a complete graph, with the following fundamental characteristics:

- Cycle structure: The route must be cyclic, meaning the first and last nodes must coincide.
- Importance of adjacency: The quality of a solution depends on the proximity of cities in the route.
- Permutation constraint: Each city must be visited exactly once.

From my point of view, these constraints make some genetic operators more suitable than others. For example:

Inver-over crossover is designed to preserve favorable adjacencies and inversion mutation modifies the route in a localized manner maintaining the proximity of cities.

The developed genetic algorithm aims to find a more accurate solution to the TSP problem compared to the greedy algorithm. However, the best path found, for example, in the instance concerning Italy (6933.95 km) is significantly far from the known optimal solution (4172.76 km). This difference suggests potential improvements in the implemented approach.

Therefore, while this genetic algorithm provides a solid foundation for solving the TSP, it requires significant enhancements to approach the optimal solution.

The current approach uses:

- Order Crossover (OX): While suitable for permutation problems, it does not leverage semantic information related to adjacency. In TSP, the connections between cities are more critical than their relative positions.
- Multiple mutations (swap, scramble): Although flexible, these mutations may introduce excessive variation (scramble) or insufficient exploration (swap), reducing the efficiency of exploration.

Instead, the inver-over crossover combines inversion and crossover leveraging the structure of the TSP:

- Avoiding adjacency disruption: Inver-over preserves favorable connections between cities in two ways:
  It uses information from a second individual to guide the choice of the subsequence to invert.
  It selects cities based on their proximity in the graph.
- Targeted exploration: It generates variations of the route by inverting only a localized portion of the genome. This allows more "intelligent" exploration compared to random mutations.
  By using inver-over, the algorithm should better preserve local connections between cities while maintaining the cyclic nature of TSP solutions.

Here, I write an example pseudocode for integrating it into the algorithm:

```python
def inver_over_crossover(parent1, population, p):
    offspring = parent1[:]
    current_city = random.choice(offspring)  # Select a random city
    while True:
        # Decide whether to use another city from the same parent or another individual
        if random.random() < p:
            next_city = random.choice([c for c in offspring if c != current_city])
        else:
            random_individual = random.choice(population)
            next_city = random_individual[(random_individual.index(current_city) + 1) % len(random_individual)]

        # Terminate if current_city and next_city are already adjacent
        if abs(offspring.index(current_city) - offspring.index(next_city)) == 1:
            break

        # Reverse the subsequence
        i, j = offspring.index(current_city), offspring.index(next_city)
        if i > j:
            i, j = j, i
        offspring[i+1:j+1] = offspring[i+1:j+1][::-1]

        # Update the current city
        current_city = next_city
    return offspring
```

In addition, inversion mutation is a localized mutation that changes the order of cities between two points while maintaining the global structure of the route. It is particularly useful in TSP for the following reasons:

- Preserving geographical proximity: Reversing a subsequence keeps local connections between cities reducing the risk of creating suboptimal solutions.
- Balanced exploration and optimization: Unlike swap or scramble, inversion introduces significant changes without drastically altering the solution.

```python
def inversion_mutation(route):
    mutated_route = route[:]
    i, j = sorted(random.sample(range(1, len(mutated_route) - 1), 2))  # Avoid modifying the start/end city
    mutated_route[i:j+1] = reversed(mutated_route[i:j+1])
    return mutated_route
```

Other suggestions for improvement could be:
Retaining the top 10% of the best individuals (elitism) may reduce diversity.
A more balanced approach could combine elitism with random replacements to reintroduce variability.
Secondly, adding a 2-opt phase as a local search operator can significantly improve the quality of solutions.
Each individual can be refined locally before its fitness evaluation.
Thirdly, making the probabilities of crossover and mutation adaptive—for instance, based on the quality of the solutions found—can improve performance (more exploration in the early stages, more exploitation in the later stages).

In conclusion, the use of the Inver-Over crossover and the inversion mutation would better align the algorithm with the semantics of the TSP.
Although the current implementation provides a solid foundation, adopting these specific operators and achieving a better balance between exploration and exploitation could bring the solutions closer to the optimal ones.

- https://github.com/vitosilver99/CI2024_lab2/issues/1
First of all, I want to congratulate you on the excellent results you have achieved.
I haven't read a description from you but I believe you implemented the order crossover as the type of crossover operator and I think this was a key choice in achieving such high accuracy compared to the optimal results for the following reasons.
The TSP imposes specific constraints that any solution must adhere to:

- Valid Permutation: Each city must be visited exactly once.
- Adjacency: The order in which cities are visited determines the solution's quality since it affects the total distance.
- Cyclic Structure: The first and last cities must be connected to close the cycle.

The Order Crossover is designed to respect these constraints making it a natural choice for the TSP.
Order Crossover operates in two stages:

1. Preservation of Partial Order: A segment from the first parent is copied into the child at the same position.
2. Filling Missing Positions: The child is completed with the remaining cities from the second parent maintaining their relative order.
   This strategy ensures that the cities copied from the first parent maintain their order and local adjacencies and the cities from the second parent contribute diversity by introducing new connections.
   Order Crossover is particularly well-suited for the TSP for several reasons, explained in relation to the problem's constraints and goals:
   Firstly, the segment copied from the first parent remains intact in the child preserving local adjacencies.
   This is crucial in the TSP because good solutions often emerge from configurations that retain favorable connections between neighboring cities.
   Secondly, by filling the remaining positions with cities from the second parent, Order Crossover introduces new connections and increases diversity.
   This balance between exploitation (preserving good structures) and exploration (introducing new configurations) helps avoid local optima.
   Thirdly, Order Crossover always produces valid permutations, where each city appears exactly once and, while the TSP requires a cyclic structure, Order Crossover inherently manages the connection between the first and last cities, as the child is constructed to respect the parents' relative order.

A suggestion I would like to propose to improve the computation time of the genetic algorithm is the following:

Introducing an early stopping criterion based on the stagnation of the best individual's fitness is a practical strategy to improve the efficiency of the genetic algorithm.
If the fitness improvement stalls, the algorithm is likely close to a local minimum or the optimal solution thus continuing to generate new populations may not yield significant benefits.
For this reason, lack of fitness improvement over several generations is a sign that the population has reached a stable state.
This approach could avoid wasting computational resources on unnecessary iterations, especially for complex problems like the TSP.
The idea is to monitor the fitness improvement of the best individual. If the fitness does not improve for a predetermined number of consecutive generations (for example, 50 generations), the algorithm terminates:

```
no_improvement_count = 0
max_no_improvement = 50  # Maximum generations without fitness improvement

best_fitness = float('-inf')  # Initialize best fitness to the lowest possible value

for generation in range(MAX_GENERATIONS):
    # Execute the steps of the genetic algorithm
    # (e.g., selection, crossover, mutation, and fitness evaluation)
    ...

    # Find the fitness of the best individual in the current generation
    current_best_fitness = max(individual.fitness for individual in population)

    # Check if there is an improvement in the best fitness
    if current_best_fitness > best_fitness:  # Fitness must increase to improve
        best_fitness = current_best_fitness  # Update the best fitness
        no_improvement_count = 0  # Reset the no improvement counter
        print(f"Generation {generation}: Fitness improved to {best_fitness:.2f}")
    else:
        no_improvement_count += 1  # Increment the no improvement counter
        print(f"Generation {generation}: No improvement for {no_improvement_count} generations.")

    # Terminate if no improvement has been observed for the specified limit
    if no_improvement_count >= max_no_improvement:
        print(f"Terminating early at generation {generation} due to stagnation.")
        break

# Final output of the best solution
print(f"Algorithm terminated. Best fitness found: {best_fitness:.2f}")
```

From my point of view, instead of only tracking the best individual, you could also monitor the average fitness improvement of the entire population. This provides a more comprehensive picture of the algorithm's convergence.

In conclusion, it is likely that Order Crossover played a key role in preserving and combining the essential properties of the parents, leading to high-quality solutions. Good job!

CI2024_lab3

In this lab, I solved the `n^2 - 1` puzzle problem by interpreting it as a pathfinding problem. I can adopt this perspective because the puzzle can be seen as a problem requiring a sequence of decisions, where each decision corresponds to the move needed at each step to transition from the puzzle's initial random configuration to the desired final configuration.
The pathfinding algorithm I considered most effective to be implemented for solving this specific problem is the A* algorithm.
I decided to use a graph search approach and chose the Manhattan distance as the admissible and consistent heuristic.
The Manhattan distance is a heuristic commonly used in pathfinding and optimization problems, it measures the total distance a tile needs to travel to reach its goal position assuming movement is restricted to a grid and only allowed in horizontal or vertical directions.
It's admissible because it never overestimates the true cost to solve the puzzle as it assumes no obstructions or collisions and satisfies the triangle inequality ensuring that moving from one state to another does not violate heuristic consistency.

In pathfinding algorithm A*, the use of a heuristic function plays a crucial role in guiding the search toward the goal state more efficiently.
Unlike uninformed search algorithms (for example, Breadth-First Search or Uniform Cost Search), A* leverages heuristic information to prioritize exploration of the most promising states, thus reducing the number of expanded nodes and accelerating convergence.
A* evaluates each state n using the formula `f(n) = g(n) + h(n)` where:

- `g(n)` is the cost to reach the current state from the start state.
- `h(n)` is the heuristic estimate of the cost to reach the goal from the current state.

The heuristic function `h(n)` provides an estimate of how "close" the current state is to the goal.
By including `h(n)`, A* can focus on states that are likely part of the optimal path and reduce the exploration of irrelevant or less promising states.
Without a heuristic A* behaves like Uniform Cost Search expanding nodes based solely on `g(n)`, which represents the total cost from the start.
This leads to unnecessary exploration of states that are far from the goal but have a low accumulated cost.
With a heuristic like Manhattan distance, A* expands nodes by balancing the cost `g(n)` with the estimate `h(n)`: this means that states closer to the goal (based on `h(n)`) are prioritized, directing the search toward the solution.
An admissible heuristic ensures that `h(n) ≤ h(n)*` where `h(n)*` is the true cost to the goal.
This guarantees that A* never ignores the optimal solution but still avoids exploring unnecessarily.
Moreover, a consistent heuristic satisfies `h(n) ≤ c(n,n') + h(n')` where `c(n,n')` is the cost to transition from n to n': this ensures that the estimated cost f(n) along any path is non-decreasing, preventing redundant revisits of states and further improving efficiency.

Graph search

In graph search, it is essential to maintain a set of already expanded nodes to ensure that they are not re-added to the priority queue and then re-examined.

This is because, unlike tree search, graph search acknowledges that multiple paths can lead to the same state. Without a list of expanded nodes, the same state might be generated multiple times via different paths and added to the priority queue repeatedly.
This would lead to unnecessary expansions increasing computational overhead and wasting resources.
By tracking expanded nodes, the algorithm avoids revisiting states, reducing the total number of nodes expanded.
This is particularly critical in problems with cyclic or densely connected graphs: in a graph with cycles, failing to track expanded nodes could lead to infinite loops, where the algorithm keeps revisiting the same states.

Mutable objects can't be keys in a set

For this purpose, using `bytes` and `tobytes()` to convert a NumPy array into an immutable object is essential for efficiently tracking explored/expanded states in graph search.
Python's set and dict rely on hashing for fast lookups and require their keys to be immutable.
A NumPy array is mutable meaning its content can be changed after creation.
Because of this mutability, NumPy arrays cannot directly be used as keys in a set or dict.
`bytes` is an immutable sequence of bytes in Python: once created, its content cannot be changed.
By converting a NumPy array to bytes using `tobytes()`, I create an immutable representation of the array that retains the exact data and layout of the array in memory and can be hashed, allowing it to be used as a key in a set or dict.
Hashing is used to quickly check for membership in a set or as a key in a dict.
The hash function calculates a fixed-length value (hash) from the object which serves as its "identifier."
The object must not change after its hash is calculated because, if the object changes, its hash would change, leading to inconsistencies. By converting the array to `bytes`, I ensure the object remains immutable.
In addition, objects with different content should produce different hashes. The bytes representation from `tobytes()` ensures that arrays with different values produce distinct hashes.

Conditions for Solvability

The `n^2 - 1` puzzle has a fundamental mathematical property: not all initial configurations can lead to the goal state.
This limitation arises from the structure and rules of the puzzle, which restrict the ways tiles can be rearranged.
The puzzle can be represented as a permutation of tiles with one empty space (0) allowing movement. The solvability of a configuration depends on the number of inversions in the arrangement. An inversion occurs when a higher-numbered tile precedes a lower-numbered tile in the linearized array of the puzzle's tiles (ignoring the empty space).
By performing some mathematical analyses, I realized that the rules for determining whether a configuration is solvable differ depending on the dimension **n** of the puzzle:

- **Case 1: n odd**
  A configuration is solvable if the number of inversions is even.
- **Case 2: n even**
  A configuration is solvable if:
  1. The number of inversions is even and the blank tile (0) is on an odd row from the bottom.
  2. The number of inversions is odd and the blank tile (0) is on an even row from the bottom.

To ensure that my algorithm starts with a solvable configuration, I developed the function generate_random_solvable_puzzle().

It ensures solvability in the following way:

- It starts from the goal state since the goal state is guaranteed to be solvable.
- Apply valid moves: the function performs a series of random moves from the goal state using the puzzle's rules to generate new configurations. Since every move is valid and starts from a solvable state, the resulting configuration remains solvable.

Therefore, by design, I eliminate the need for computationally verifying the solvability of the resulting configuration, as it is guaranteed.

Time Complexity: `heapq` vs list

Finally, discussing the computational efficiency of the operations, I provide an in-depth explanation of why it is advantageous to use the heapq data structure to implement the priority queue instead of a list since using `heapq` to implement a priority queue offers significant computational advantages compared to a basic list. `heapq` is a Python module that implements a binary heap, a specialized tree-based data structure:

- It maintains the heap property: the smallest (or largest, for a max-heap) element is always at the root.
- Operations like insertion and extraction of the smallest element are highly efficient.

In the context of A*, the priority queue needs to quickly retrieve the node with the lowest f(n) and efficiently add new nodes as they are generated.
The key operations of a priority pueue are:

- Insert (Push): add a new element to the queue. Must maintain the order of priorities for efficient extraction of the minimum.
- Extract Minimum (Pop): remove and return the element with the smallest priority. The queue must always keep track of the smallest element efficiently.
- Peek: view the smallest element without removing it.

The comparison table shows the time complexities for these operations when using:

1. A **binary heap** (`heapq`)
2. A **list** (sorted or unsorted)

Operation	**heapq (Binary heap)**	**list (sorted or unsorted)**
Insert	O(log n)	O(1) (unsorted) or O(n) (sorted)
Extract Minimum	O(log n)	O(n) (unsorted) or O(1) (sorted)
Peek	O(1)	O(n) (unsorted) or O(1) (sorted)

Experimental results

In conclusion, I report the number of moves and the number of states explored by my implementation of the A* algorithm to solve 3 instances of different sizes of the problem, each with 3 randomly generated starting configurations of the puzzle.
Each of the 3 execution durations corresponds to the use of the Intel(R) Core(TM) i9-14900KF processor:

Puzzle Size	Starting State	Number of Moves	Number of Explored Nodes	Execution Time (hh:mm:ss.microsecond)
3	[[4, 1, 3],	9	17	00:00:00
	[7, 2, 0],			
	[5, 8, 6]]			
4	[[1, 2, 11, 7],	23	128	00:00:00.006000
	[0, 5, 12, 3],			
	[9, 6, 15, 4],			
	[13, 10, 8, 14]]			
5	[[1, 2, 3, 9, 4],	29	898	00:00:00.071051
	[6, 7, 8, 15, 5],			
	[16, 11, 12, 13, 10],			
	[17, 14, 23, 22, 24],			
	[21, 0, 18, 20, 19]]			

Suggestions from my colleagues

I received one review:

- https://github.com/GiorgioBongiovanni/CI2024_lab3/issues/1

This project effectively combines graph search principles with the A* algorithm, ensuring both optimality and efficiency. By tracking explored states and maintaining an accurate frontier, the implementation avoids redundant computations and unnecessary expansions. The choice to use heapq for the priority queue further strengthens the algorithm's performance.

Additionally, the use of the Manhattan distance heuristic it's admissible and consistent, guiding the search effectively while guaranteeing the optimal solution. Also the frequent use of comments and detailed metrics like number of moves, explored nodes and execution time make the lab project even more understandable.

One area for improvement would be to include a visualization of the puzzle's progression from the starting state to the goal state. This addition would make the solution path more intuitive and engaging, allowing readers to better appreciate the algorithm's step-by-step process.

Conclusion

This project is a strong and well-implemented solution, showcasing a clear understanding of the A* algorithm and heuristic design. With minor enhancements like visualization, the project could be even more impactful. Great job!

Suggestions for my colleagues

I wrote two reviews:

- https://github.com/mistru97/CI2024_lab3/issues/1

As far as I'm concerned, this implementation of the $n^2 - 1$ puzzle solver exhibits both strengths and weaknesses.
Firstly, a positive aspect is that the program is modular, separating concerns into distinct functions (for example, available_actions, do_action, goal_test, search): this makes the code easier to follow and maintain but some parts of the code lack comments making it harder to understand.
Secondly, multiple heuristics are provided (h_euclidean, h_manhattan, h_tile_not_in_place) allowing flexibility in the choice of the heuristic for guiding the search but, based on the tests that I performed, the Manhattan heuristic is the most effective for grid-based puzzles while you chose euclidean distance.
This code implements a graph search, as evidenced by the use of the state_cost dictionary to track the minimum cost for each visited state.
This strategy prevent re-exploration of the same states, which is a hallmark of graph search, unlike tree search which does not account for visited states.
According to:

```
if new_state not in state_cost and new_state not in [s[1] for s in frontier]:
```

a new state is considered only if it has not already been visited (not in state_cost) or is not already in the frontier (the check uses a list comprehension, which could be optimized with a set).
Therefore, the graph search requires the heuristic to be both admissible and consistent.
The Manhattan distance is more suitable for grid-based problems where movements are restricted to vertical and horizontal directions because it directly reflects the real constraints of such movements.
The Manhattan distance between two points measures the total number of steps required to move from one point to another along a grid where only vertical and horizontal moves are allowed and so reflects the actual movement cost in grid-based problems.
In a grid where movements are restricted to vertical and horizontal directions:

- A path from a point (x_1, y_1) to another point (x_2, y_2) must consist of $|x_2 - x_1|$ horizontal steps and $|y_2 - y_1|$ vertical steps.
- The Manhattan distance exactly captures this cost because it counts the required horizontal and vertical steps separately and sums them up.

The Manhattan distance is always admissible in grid-based problems because it represents the minimum possible cost to reach the goal, assuming no obstacles and it is always consistent because moving from state s to the next state s' in one vertical or horizontal step reduces the Manhattan distance by exactly 1, matching the transition cost c(s, s') = 1.
In addition, the Manhattan distance only requires subtraction and addition of coordinates, which are computationally inexpensive operations.
This makes it particularly suitable for problems with large grids or many states to evaluate.

On the other hand, the Euclidean distance between two points is
$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
The Euclidean distance assumes straight-line movement which includes diagonal moves.
In grid-based problems where only vertical and horizontal moves are allowed, this does not align with the actual cost of moving between states and thus underestimates too much the true cost.

The Euclidean distance is admissible because it never overestimates the true straight-line distance.
However, it is not always consistent in grid problems because it does not accurately account for the discrete steps required.
This can lead to situations where the heuristic suggests a lower cost than the actual transition requires violating the consistency condition.

For instance, consider a grid where movements are restricted to vertical and horizontal steps:
Start: (0, 0)
Goal: (3, 4)

1. **Manhattan Distance**:
   - Required steps: $|3-0| + |4-0| = 3+4 = 7$
2. **Euclidean Distance**:
   - Straight-line distance:
     $\sqrt{(3-0)^2+(4-0)^2} = \sqrt{9+16} = \sqrt{25} = 5$
   - The Euclidean distance underestimates the true cost because it assumes diagonal movement, which is not allowed.

As far as computational complexity is concerned, the Euclidean distance requires squaring and taking the square root of coordinates, which is computationally more expensive than the Manhattan distance.
In scenarios with many states, this additional cost can significantly slow down heuristic evaluations.

For the previous reasons, from my point of view, the Manhattan distance is the better choice for grid-based problems with vertical and horizontal movements because:

1. It aligns perfectly with the movement constraints.
2. It is admissible and consistent, ensuring the correctness and efficiency of search algorithms like A*.
3. It is computationally simpler and directly reflects the true cost of moving between states.

The Euclidean distance, while useful in problems with free diagonal movement, is not suitable for grids with orthogonal movement due to its tendency to underestimate the actual cost and its higher computational cost.
As proof that it would be necessary to reconsider the choice of heuristic, I tried running the pathfinding program to solve a 5x5 puzzle instance and, despite letting it run for 260 minutes, I was unable to obtain a solution.

In conclusion, with optimizations and better documentation, this algorithm could become a robust and reusable solution.

- https://github.com/GiorgiaModi/CI2024_lab3/issues/2

Hi Giorgia, great work on your implementation!
Your explanation of the project and the design choices you made are really impressive, it's detailed, well-structured and you've clearly put a lot of thought into explaining the algorithms and presenting the results.
I'd like to highlight some of the excellent aspects of your work and also discuss a few areas where there might be opportunities for optimization.
You introduced the n2−1 puzzle very well providing a clear description of the rules and the goal.
Including the example of the 4x4 goal configuration makes it easy for readers to visualize the problem.
This is especially helpful for anyone unfamiliar with the puzzle.

The choice of A* and Iterative Deepening Depth-First Search (DFS) shows a solid understanding of the trade-offs between informed and uninformed search algorithms.
Explaining the strengths and weaknesses of each algorithm—such as A* being optimal but

memory-intensive and DFS being memory-efficient but slower—is a strong point in your explanation.

Using the Manhattan Distance for A* was an excellent decision: this heuristic is admissible and consistent making it well-suited for the puzzle since movements are restricted to vertical and horizontal directions.
It's clear you considered the problem's constraints carefully when choosing the heuristic.

Implementing a second heuristic, linear conflict, demonstrates creativity and a willingness to experiment with different approaches. Even though it didn't improve performance, testing it shows your analytical approach to problem-solving.

The results table is very well-organized and informative because it provides valuable insights into how the algorithms perform with different puzzle sizes and configurations.
Mentioning how the performance depends on factors
like RANDOMIZED_STEPS and max_depth is a great touch.
It highlights the variability of the problem and gives readers context for interpreting the results.

In conclusion, you've shown a deep understanding of the algorithms by explaining how they handle larger puzzles and discussing why certain configurations might fail (for example, DFS not finding solutions within the depth limit).

Your explanation of DFS with bounds is clear but the results show that it struggles with larger puzzles due to the depth limit. It might be worth exploring alternative uninformed search algorithms, such as Breadth-First Search (BFS), for smaller puzzles, or implementing a more adaptive depth limit strategy.

While the results table is detailed, it could benefit from more analysis. For example:

- Why does DFS take significantly longer for larger puzzles compared to A*?
- Why does reducing RANDOMIZED_STEPS improve execution time for larger puzzles?
- How does the initial configuration impact the performance of each algorithm?

Adding this context would make the results even more insightful.

Although the linear conflict heuristic didn't outperform Manhattan Distance, it would be great to include a short explanation of why this might be the case. For instance:

- Linear conflict might add computational overhead without significantly improving the heuristic's informativeness.
- Highlighting such insights would strengthen your analysis and demonstrate your understanding of heuristic design.

Overall, Giorgia, your implementation is impressive and well-documented. You've clearly put a lot of effort into explaining the algorithms, their trade-offs and the results.
Great work overall—keep it up! 😊

Project Work:
Genetic Programming for Symbolic Regression

The algorithm I developed for this project is a Genetic Programming (GP) system specifically designed to solve symbolic regression tasks.
The main idea is to evolve mathematical expressions in the form of trees until they approximate a target dataset with high accuracy while keeping the formulas compact and interpretable.
Each individual in the population is represented as an **expression tree**, where the internal nodes correspond to mathematical operators and the leaves are either input variables or numeric constants.
The choice of the primitive set was deliberate: I included only relatively simple mathematical operators in order to avoid uncontrolled complexity and overfitting.
The operators consist of addition, subtraction, multiplication, and a **protected version of division** that returns 1 when the denominator is close to zero.
I also introduced a limited number of unary functions such as negation, logarithm, exponential, square root, absolute value, sine and square, as well as max and min for binary comparisons.
All these operators were **numerically guarded**: logarithms are applied only to positive values, square roots only to non-negative arguments, exponentials are clipped to prevent overflow, and division is stabilized with an epsilon.
This design ensures that the evaluation of individuals never crashes and avoids the generation of pathological expressions that would distort the fitness landscape.
At the same time, by keeping the operator set simple and dominated by algebraic transformations, the search space is constrained in a way that discourages overfitting and favors smooth, generalizable expressions.
Another distinctive element of my implementation is the use of **feature importance** during initialization.
Before generating the first population, the algorithm computes the mutual information between each input feature and the target output.
These scores are then normalized and used as **sampling weights** for terminals, meaning that more informative variables are more likely to appear in the individuals.
This strategy guides the search toward relevant parts of the feature space without removing the possibility of exploring less obvious relationships.
It improves convergence speed and helps the GP focus its resources effectively.
The **initial population** is created using a combination of the *full* and *grow* methods.
Full trees fill all levels up to a maximum depth, while grow trees randomly mix terminals and functions within depth limits.
By splitting the population between these two initialization strategies, the algorithm ensures structural diversity in terms of both tree size and shape.
This structural diversity at generation zero is critical for avoiding premature convergence and covering a broad region of the search space from the very beginning.

The **fitness function** is based on the mean squared error (MSE) between the predicted and actual outputs.
To discourage excessive growth of the trees, I added a **parsimony penalty** proportional to the number of nodes.
In this way, fitness balances two objectives: accuracy and simplicity.
This prevents bloat and promotes the discovery of interpretable formulas.
The penalty is not too strong, so accuracy remains the primary driver, but it is enough to keep tree sizes under control.
For **selection**, the algorithm uses **tournament selection** with a large tournament size (150).
This introduces strong selective pressure, ensuring that the best individuals are more likely to be chosen as parents.
Such strong exploitation, however, risks reducing diversity.
To counterbalance this, the system relies on variation operators and explicit anti-stagnation strategies, which restore exploration when the search risks becoming trapped.
Crossover and mutation are the two main sources of variation.
Crossover, applied in about 80% of cases, works by exchanging subtrees between two parents.
To keep offspring valid, the crossover operator includes a depth-aware trimming mechanism that ensures the resulting trees respect maximum depth constraints.
Mutation is applied in the remaining 20% of cases and here I adopted a **portfolio of mutation operators**.
The dominant one is **subtree mutation**, which replaces a random subtree with a newly generated one.
This operator accounts for 90% of all mutations and is the main driver of exploration, since it allows the algorithm to perform large structural jumps.
To fight bloat and keep models simple, I also included **hoist mutation**, which replaces a subtree with one of its children, thereby shrinking the overall tree.
Finally, **point mutation** introduces small local changes by replacing the value of a node with another of the same arity or by resampling a terminal.
The balance of probabilities across these mutation operators ensures both global exploration and fine-grained local adjustments.
In addition to variation, the algorithm maintains a degree of **elitism**: the top 5% of individuals are copied unchanged into the next generation.
This guarantees that progress is not lost and that the best solutions found so far are always preserved.
To avoid stagnation caused by too much elitism, elites are limited to this small fraction and the rest of the population is entirely regenerated.
The system also includes **explicit anti-stagnation strategies**.
Every 20 generations, it injects a large number of newly generated grow trees equal to half the population size.
This sudden injection of fresh genetic material helps maintain diversity and explore new regions of the search space.

Furthermore, if no improvement in the best fitness is observed for 50 consecutive generations, the algorithm replaces half of the population with new grow trees.
This acts as a strong restart mechanism, preventing the search from getting stuck in local minima.
These strategies, combined with subtree mutations and hoist mutations, maintain an active evolutionary dynamic and sustain exploration throughout the run.
To evaluate large populations efficiently, I implemented **parallel fitness evaluation** using Python's ProcessPoolExecutor across 24 cores.
The population is divided into balanced batches, each assigned to a core, and then all results are collected.
This parallelism makes the algorithm scalable and significantly reduces computation time, which is crucial given the size of the population and the number of generations.
Another important component is **post-evolution constant optimization**.
While GP is very good at discovering the correct symbolic structure of an equation, it is less effective at tuning numeric constants.
To overcome this limitation, once the best tree is found I extract all its numeric leaf constants and optimize them using the L-BFGS-B algorithm from scipy.optimize.minimize.
This optimization minimizes the mean squared error with the structure fixed, effectively combining the global structural search power of GP with the local precision of gradient-based optimization.
The result is a noticeable reduction in error without increasing model complexity.
Altogether, the algorithm is designed to balance exploration and exploitation.
Exploration is driven by subtree mutation, periodic injections of new individuals, stagnation resets, and diversity in initialization.
Exploitation is guaranteed by strong tournament selection, elitism, crossover and local coefficient optimization.
Overfitting and excessive complexity are kept under control by the restricted primitive set, parsimony pressure, hoist mutation and depth trimming.
Numerical guards ensure stability of the evaluation process even in the presence of risky operators and parallelism makes the algorithm efficient in practice.
The final outcome is a GP system that is robust, effective and interpretable.
It evolves formulas that approximate the data with high accuracy, avoids the pitfalls of bloat and stagnation and produces results that generalize well.
By combining symbolic evolution with post-hoc constant optimization, the system achieves both structural insight and numerical precision, offering a strong solution for symbolic regression within the framework of computational intelligence.