Computational Intelligence Course Report

Table of Contents

Introduction

This **Computational Intelligence** course focused on solving real-world problems using computational models and optimization techniques. Throughout the semester, I explored various problem-solving strategies, algorithmic techniques, and heuristic approaches, applying them to different practical scenarios.

The report is structured as follows:

- **Labs:**
  I worked on three key labs, each tackling a different optimization problem:
    - **Lab 1: Set Cover Problem** – Implementing heuristic approaches for large-scale optimization.
    - **Lab 2: Traveling Salesman Problem (TSP)** – Using Evolutionary Algorithms to optimize route planning.
    - **Lab 3: N-Puzzle Path Search** – Comparing path search algorithms like A*, BFS, and DFS.

- **Presentation:**
  A planned presentation on **Deep Reinforcement Learning**, specifically focusing on training AI to play Snake using **Deep Q-Learning**. Although the presentation didn't proceed, it was a valuable learning experience.

- **Final Project:**
  A computational experiment involving **Genetic Programming for Formula Evolution**, where I explored optimization techniques related to Mean Squared Error (MSE) minimization.

- **Lecture Notes:**
  This section contains summaries of the notes I took during the semester for different parts of the course. These cover key concepts in Computational Intelligence, breaking down important ideas in a way that helped me understand and apply them better.

Lab 0: Getting Started

This was supposed to be the first lab, but I didn't quite make the deadline. In the midst of what felt like **World War 3**, my flight got canceled, which completely threw off my schedule. But hey, that's not an excuse—I fully understand the situation and take responsibility for not submitting the first lab on time.

The task itself was pretty straightforward: create a GitHub repo, upload a file named `irony.md`, and include a joke or witty comment about the first lecture. Even though I was late, I did end up completing it later (better late than never, right?).

Here's what I wrote in my file:

> "Since I was late to the first lab, looks like I've successfully optimized myself to be the epitome of what not to do."

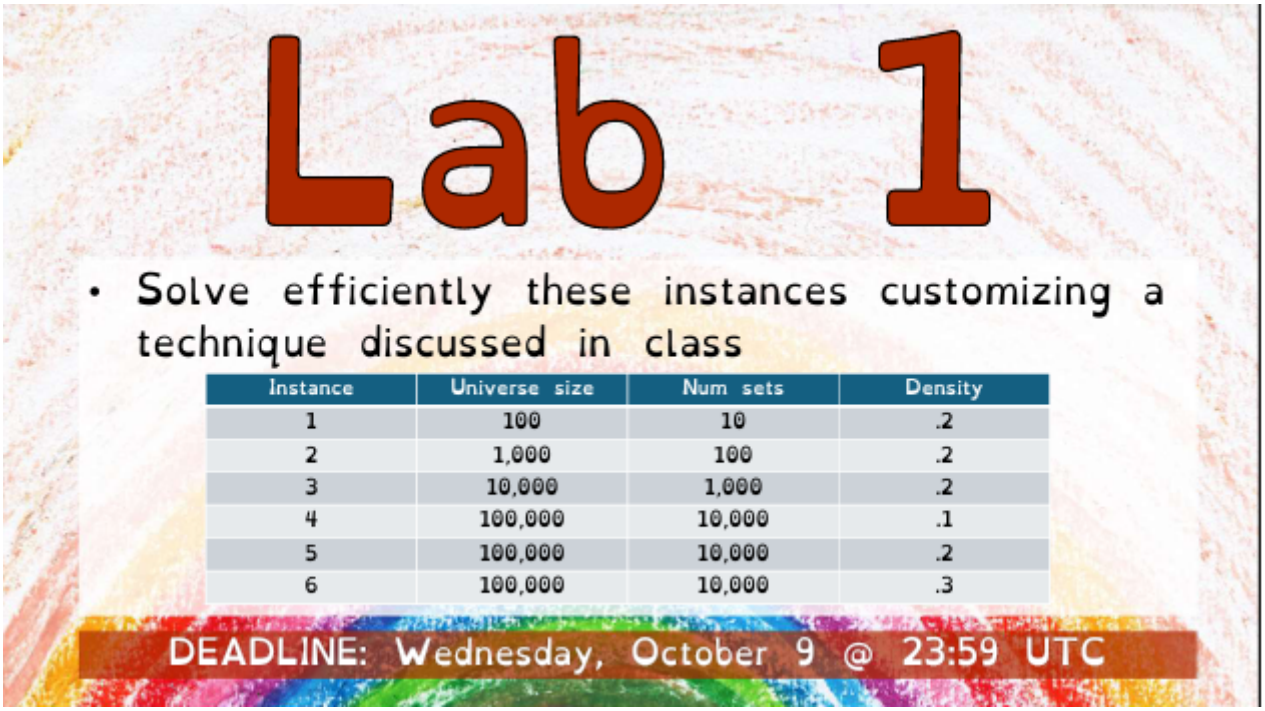
It was a small but fun task, and it gave me a chance to get a understanding of how we will do labs on github.

Set Cover Problem - Lab 1

🔗 **GitHub Repository:** CI2024 Lab 1 - Set Cover Problem

Problem Statement

This lab required solving instances of the Set Cover Problem with varying universe sizes, number of sets, and densities. The objective was to implement a solution that efficiently handles these variations and reports the initial and final fitness values.



The goal is to **find the minimum-cost subfamily of** S such that the union of these subsets covers U.

Approach

The implemented Tabu Search algorithm for the Set Cover Problem includes:

- **Dynamic Tabu List**: Starts with an initial size and grows as the search progresses, balancing between exploration and exploitation.
- **Aspiration Criteria**: Allows accepting moves in the tabu list if they lead to a solution better than the best found so far.
- **Multiple Mutation**: Generates diverse neighborhood solutions by potentially flipping multiple bits in the solution representation.
- **Fitness Tracking**: Monitors and stores the fitness of solutions throughout the search process.
- **Multi-Instance Testing**: Applies the algorithm to multiple problem instances with varying parameters, demonstrating its adaptability to different problem scales.
- **Results Visualization**: Provides graphical representation of the algorithm's performance for each instance, aiding in analysis and comparison.

The solution was tested across multiple instances, demonstrating its effectiveness in finding optimal or near-optimal solutions.

Results

Instance	Universe Size	Num Sets	Density	Initial Fitness	Final Fitness
1	100	10	0.2	-29.9648	0
2	1,000	100	0.2	-14241.1775	-6409.3919
3	10,000	1,000	0.2	-1243184.90	-571303.45
4	10,0000	10,000	0.1	-74629161.782	-74629161.78218833
5	100,000	10,000	0.2	-165060887.250	-165014923.3968753
6	100,000	10,000	0.3	-252231834.97	-252124227.919458

Performance Analysis

- The algorithm consistently improved the initial random solutions, the algorithm doesn't work well for 100,000. Haven't figured out why yet.
- Larger instances (100,000 universe size) required significantly more computation time.
- Density variations impacted the ease of finding valid solutions.

Challenges and Optimizations

- **Memory Management**: Implemented efficient data structures for large instances.
- **Runtime Optimization**: Used vectorized operations where possible.
- **Parameter Tuning**: Adjusted tabu list size and mutation rate for different instance sizes.
- **Biggest Challenge**: testing the algo on 100,000 universe size on my 6-year-old laptop was like using the free version of ChatGPT during peak hours.

Acknowledgments

For this Computational Intelligence lab submission, I explored various resources. While ChatGPT is a popular choice, I decided to give Claude AI a chance, and it proved to be a great help. As always, Google was my trusty companion throughout the process.

Feedback from Colleagues

Just like I was assigned to review my peers' repositories, my colleagues were also assigned to evaluate my implementation of the **Set Cover Problem**. This section includes the feedback I received from them, highlighting their observations, suggestions, and areas where they thought my approach could be improved.

Review from Mericuluca

- **Feedback**: Nice solution, it is very suitable for this problem. However, the tabu search buffer can be computationally expensive. Limiting the tabu size can be beneficial for instances with many iterations as it reduces search time in large lists. Limiting the tabu size will likely not decrease performance significantly because it prevents retrying recent solutions. Overall, it is a nice solution.

Review from Daviruss1969

- **Feedback**: The Tabu Search implementation is clear and yields good results. However, some recommendations:

    1. **Dynamic Constants**: Adjust parameters like `max_iterations`, `initial_tabu_size`, and others based on problem size (e.g., `UNIVERSE_SIZE * NUM_SETS`). This could involve defining thresholds to determine constant values dynamically.
    2. **Tweak Function Performance**: Update the `multiple_mutation` algorithm to use NumPy's vectorization for improved speed on large instances.

    Suggested Implementation:

```
# Dynamic constants example
def compute_constants():
    problem_size = UNIVERSE_SIZE * NUM_SETS

    if problem_size <= THRESHOLD1:
        return some_consts

    if problem_size > THRESHOLD1 and problem_size <= THRESHOLD2:
        return some_consts
    ...

# Vectorized tweak function
def multiple_mutation(solution, mutation_rate=0.01):
    mask = rng.random(NUM_SETS) < mutation_rate
    new_solution = np.logical_xor(solution, mask)
    return new_solution
```

My Response

- **Acknowledgment**: Thank you for your insightful review! Both suggestions are fantastic, and I agree they will improve performance, particularly for larger instances.
- **Next Steps**:

- I plan to implement dynamic constants based on problem size, which should help balance iterations and tabu list size effectively.
- The vectorized implementation of `multiple_mutation` is a much-needed optimization and will be included in my next iteration.

Your feedback has been incredibly valuable, and I appreciate the time and thought put into these suggestions!

Lab 1 Peer Reviews

As part of the lab assignment, I was assigned my peers' repositories to analyze, check, and provide feedback on their implementations of the **Set Cover Problem**. This section contains my reviews of their repositories, highlighting my observations, suggestions for improvement, and the feedback I provided.

My Review of Gabry323387's Implementation

Strengths

Gabry323387's implementation of the Hill Climbing algorithm for solving the Set Cover Problem is well-structured and functional. It effectively tackles the problem but has room for improvement to escape local optima and reduce high costs in certain scenarios.

Suggested Enhancements

1. **Simulated Annealing Addition**:
   - The current Hill Climbing algorithm accepts immediate improvements but risks getting stuck in local optima.
   - I modified the function to incorporate Simulated Annealing for better exploration of the solution space:

```
# Simulated Annealing modification for Hill Climbing
from tqdm import tqdm

def hill_climbing(initial_solution, temperature=1000, cooling_rate=0.99):
    current_solution = initial_solution
    history = [cost(current_solution)]
    current_temperature = temperature

    for _ in tqdm(range(NUM_STEPS)):
        solution = tweak(current_solution)
        if valid(solution):
            delta_cost = cost(solution) - cost(current_solution)
            if delta_cost < 0 or np.random.random() < np.exp(-delta_cost /
current_temperature):
                current_solution = solution
                history.append(cost(current_solution))
        current_temperature *= cooling_rate

    return current_solution, history
```

Results Comparison

Instance	Original Hill Climbing (Full)	Simulated Annealing (Full)
Instance 1	Cost: 7033.2 Calls: 569	Cost: 6656.04, Calls: 139
Instance 2	Cost: 1280177.08, Calls: 996	Cost: 1424093.9, Calls: 557
Instance 3	Cost: 112106316.01, Calls: 1000	Cost: 112028678.48, Calls: 477
Instance 6	Cost: 721125092.2, Calls: 1000	Cost: 713358286.04, Calls: 938

- Simulated Annealing improves performance in escaping local optima and finding better solutions earlier in the search process.

Suggestions for Future Improvements

1. **Improved Tweaking**:
    - Using vectorized operations in the mutation function for better efficiency.
2. **Dynamic Parameters**:
    - Adjust constants like `max_iterations` and `tabu_size` dynamically based on problem size.

Gabry323387's code is a solid starting point, and with these enhancements, it can achieve even greater efficiency and accuracy.

My Review of SamueleVanini's Implementation

Strengths

SamueleVanini’s implementation of Simulated Annealing for the Set Cover Problem is both efficient and elegant. The balance between exploration and exploitation is well-maintained, resulting in high-quality solutions.

Suggestions for Enhancements

1. **Instance Visualization**:
    - Including plots for all problem instances would make the analysis more comprehensive and visually informative. This addition would help reviewers and users better understand how the algorithm performs across different scenarios.

Feedback Summary

- The implemented Simulated Annealing algorithm is well-optimized and delivers consistent results.
- Despite attempts to improve fitness through parameter adjustments and code tweaks, the original implementation remains robust and balanced.

Suggested Improvements

To further enhance the understanding and utility of the results, providing detailed plots for all instances would be beneficial. Visualizing performance trends across instances adds value to the analysis and showcases the effectiveness of the algorithm.

Great job, SamueleVanini! The implementation is solid, and I'm looking forward to seeing how this progresses with future iterations.

Traveling Salesman Problem (TSP) Solver - Lab 2

GitHub Repository: CI2024 Lab 2 - Traveling Salesman Problem

Overview

As a baseline, a simple **greedy algorithm** was applied to the TSP dataset, yielding a best distance of **4436**. Trial and error revealed that using **elitism** significantly improved solution quality, so it has been incorporated in the final EA configuration to retain the best solutions across generations.

The basic greedy algorithm approach served as a comparison point for evaluating EA configurations. As a starting point, random permutations created from the initial dataset were used in the EA. However, a greedy solution as one of the starting possible solutions proved to be more time-efficient and logical, providing a better foundation for further optimization using EA algorithms.

Lab Requirements

The requirements for the lab are as follows:

- Solve the given TSP instances using both a fast but approximate algorithm and a slower, yet more accurate one.
- Report the final cost and the number of steps taken in each approach.
- The instances considered are provided in a CSV file containing the names and coordinates (latitude and longitude) of cities in Italy.

Dataset

The dataset used for this project is a CSV file containing a list of cities in Italy with their corresponding latitude and longitude coordinates.

Evolutionary Algorithm Experiments

Configuration Parameters

Base configuration for all experiments:

- Number of Cities: 46
- Population Size: 100
- Number of Parents: 20
- Number of Generations: 5000 & 10000
- Elite Size: 5

Experiment Results

Selection Method Experiments

Selection Method	Best Distance	Generations to Best	Observations
Basic Roulette Wheel	4591	3916	Slower convergence compared to the rest
Tournament (size=3)	4182.42	1632	Fastest convergence with the best distance
Rank-Based	4175	4365	Slow convergence but slightly better final result

Conclusion: Tournament Selection was used in subsequent experiments.

Mutation Experiments

Mutation Type	Rate	Best Distance	Generations to Best	Observations
Basic Swap	0.3	7054.69	1631	Worst performance compared to the others
Inversion	0.1	4560	2299	Significant improvement over Basic Swap
Inversion	0.3	4337	1135	Best results with faster convergence
Inversion	0.5	4688.8	2943	Slower and less effective than 0.3
Scramble	0.3	7340.92	2482	Poor performance

Conclusion: Inversion Mutation with a mutation rate of 0.3 was used in subsequent experiments.

Crossover Experiments

Crossover Type	Best Distance	Generations to Best	Observations
Basic Order	4257.6	3206	Balanced performance
PMX	4173.3	3131	Best performance in final distance and convergence
Edge Recombination	4272.74	2593	Fast convergence but slightly worse distance
Cycle Crossover	4654.5	3662	Worst performance in both distance and convergence

Conclusion: PMX Crossover was used in subsequent experiments.

Key Findings and Observations

1. **Selection Methods**:

- Tournament selection led to faster convergence and better fitness scores compared to basic Roulette Wheel and Rank-Based selection.

2. **Mutation Impact**:
   - Inversion mutation at a rate of 0.3 produced the best results, outperforming both higher and lower rates.
   - Scramble mutation showed consistently poor performance, possibly due to lack of directional improvement.

3. **Crossover Effects**:
   - PMX crossover produced the shortest routes, likely due to effective preservation of city relative positions.
   - Edge Recombination performed well in convergence speed but fell slightly short on final distance.

Best Configuration Found

```
Best Configuration:
- Selection Method: Tournament Selection (size=3)
- Mutation Type: Inversion
- Mutation Rate: 0.3
- Crossover Type: PMX
- Best Distance Achieved: 4173.3
```

Implementation Summary

Greedy Algorithm Baseline

The greedy algorithm was implemented as follows:

```
current_city = 0  # Starting city
visited = [False] * len(DIST_MATRIX)
visited[current_city] = True

total_distance = 0
while not all(visited):
    min_value = np.inf
    min_index = -1

    for j in range(len(DIST_MATRIX[current_city])):
        if DIST_MATRIX[current_city, j] < min_value and not visited[j] and
DIST_MATRIX[current_city, j] != 0:
            min_value = DIST_MATRIX[current_city, j]
            min_index = j

    if min_index != -1:
        visited[min_index] = True
```

```
        total_distance += min_value
        current_city = min_index

# Add the distance back to the starting city
if all(visited):
    total_distance += DIST_MATRIX[current_city, 0]
print(f'Total distance: {total_distance} km')
```

Evolutionary Algorithm (EA) Approach

Population Initialization

Random permutations of city indices were generated to create the initial population:

```
population = [np.random.permutation(num_cities) for _ in range(population_size)]
```

Selection Methods Tournament selection provided the best results:

```
def select_parents(population, fitness_scores, num_parents, tournament_size=3):
    selected_parents = []
    for _ in range(num_parents):
        tournament_indices = np.random.choice(len(population), tournament_size,
replace=False)
        winner_idx = tournament_indices[np.argmin([fitness_scores[i] for i in
tournament_indices])]
        selected_parents.append(population[winner_idx])
    return selected_parents
```

Crossover Implementation PMX Crossover:

```
def pmx_crossover(parent1, parent2):
    n = len(parent1)
    point1, point2 = sorted(np.random.choice(range(n), 2, replace=False))
    offspring = [-1] * n
    offspring[point1:point2] = parent1[point1:point2]

    mapping = dict(zip(parent1[point1:point2], parent2[point1:point2]))
    for i in range(n):
        if i < point1 or i >= point2:
            current = parent2[i]
            while current in mapping:
                current = mapping[current]
            offspring[i] = current
    return offspring
```

Mutation Implementation Inversion Mutation:

```
def inversion_mutate(path, mutation_rate):
    if np.random.rand() < mutation_rate:
        point1, point2 = sorted(np.random.choice(len(path), 2, replace=False))
        path[point1:point2+1] = path[point1:point2+1][::-1]
    return path
```

Results

The best configuration achieved a distance of **4173.3 km** after **3060 generations**.

Feedback from Colleagues

Just like I was assigned to review my peers' repositories, my colleagues were also assigned to evaluate my implementation of the **Traveling Salemen Problem**. This section includes the feedback I received from them, highlighting their observations, suggestions, and areas where they thought my approach could be improved.

Review by YounessB1

Comment: You did a great job, your code is well structured, and the README very helpful. You tried every variant of genetic algorithms, an impressive amount of effort. I'll study from your GitHub for the exam.

Only issue: You provided a solution only for Italy, but it makes sense; your CPU would have burned trying all these configurations for China.

Great job! Hope we never compete for the same job position because I am already losing.

Review by Daviruss1969 ⚙️

Comment: First of all, you have done such a great job, indeed. Implementing multiple genetic operators and comparing results was a great idea and must have been time-consuming! 💯

I also liked your README and notebook file; they are super clear and help a lot in understanding your point of view of the problem! 📃

Recommendations:

1. **Optimization of the fitness computation function** 💡
    - Your function `calculate_path_distance` is called many times to compute costs. So it needs to be as fast as possible and can be a bottleneck of your algorithm.
    - You can improve performance by using NumPy to vectorize it. Here's a template:

```
   def calculate_path_distance(path):
       distance = np.sum([DIST_MATRIX[path[i], path[i + 1]] for i in
range(len(path) - 1)])  # vectorization with np.sum
       distance += DIST_MATRIX[path[-1], path[0]]
       return distance
```

2. **Mixing genetic operators** 🔀
    - Since you have implemented many genetic operators, why not use all of their strengths?
    - By that, I mean selecting a specific genetic operator when you need more exploration/exploitation, more/less selective pressure, etc.

    Here's a template:

```
def generate_new_population(parents, mutation_rate=0.3, elite_size=5):
    new_population = []

    # Preserve the best paths (elitism)
    sorted_parents = sorted(parents, key=calculate_path_distance)
    new_population.extend(sorted_parents[:elite_size])

    # Generate the rest of the population using random selection
    while len(new_population) < len(parents):
        parent1 = random.choice(parents)
        parent2 = random.choice(parents)
        if SOME_CONDITION:
            child = pmx_crossover(parent1, parent2)
        elif SOME_OTHER_CONDITION:
            child = cycle_crossover(parent1, parent2)
        ...
        if SOME_CONDITION:
            child = inversion_mutate(child, mutation_rate)
        elif SOME_OTHER_CONDITION:
            child = scramble_mutation(child, mutation_rate)
        ...
        new_population.append(child)

    return new_population
```

Review by Lorkenzo

Comment: You've developed a solid algorithm with well-written and clearly commented code, and you've tested a variety of crossovers and mutations to reach good results. However, I'd like to give some suggestions to improve it even further:

1. **Country Selection:**
    - You provided results only for one country (Italy), making it difficult to assess if the algorithm performs well on other distributions. You've found hyperparameters that work well for this distribution, but it would be beneficial to make these hyperparameters adaptable, potentially based on the number of cities or specific distribution characteristics.

2. **Population Initialization:**
    - Currently, you start with `N` random permutations, which means your initial solutions may be far from a near-optimal solution. As noted in your code, Italy has 46 cities, resulting in 46! possible

permutations. In countries with hundreds of cities, like Russia or China, the algorithm could require a significant number of generations to approach an optimal solution. Using a fast greedy solution to help initialize the population could bring you closer to a high-quality starting point and reduce computational costs.

3. **Early Stopping Condition:**
    - To further reduce computational time, consider adding an early stopping condition that stops the algorithm if the fitness hasn't improved for several generations.

Great job overall! I hope these suggestions can help you make your algorithm even more robust and efficient!

My Reviews for Lab 2

As part of the lab assignment, I was assigned my peers' repositories to analyze, check, and provide feedback on their implementations of the **Traveling Salesman Problem**. This section contains my reviews of their repositories, highlighting my observations, suggestions for improvement, and the feedback I provided.

Review for simotmm (CI2024_lab2)

General Observations

I think this is a really good implementation of an evolutionary algorithm for solving the TSP! The use of elitism, order crossover, and inversion mutation is really well done and reminds me of my own implementation—I used similar techniques, so I can definitely appreciate the thought that went into this.

Suggestions and Observations

1. **Greedy Initialization**: One thing that could make this even better is starting with a greedy solution in the initial population. It might help speed things up and give better results. That said, I didn't use this in my implementation either since I wasn't sure if it was an option, so sticking to random initialization is still totally valid, and you got some great results with it. I can imagine how much faster the runtime would have been with a greedy starting solution though.
2. **README and Results**: Including results for all the countries (Italy, China, Russia, the US, and Vanuatu) is a big plus. I can only imagine running the larger instances like China must have taken a lot of time, so it's really impressive that all the values were calculated and noted down.
3. **Comments in the Code**: The code is clear and well-structured, but since the comments are in Italian, it made it a bit harder for me to follow as a non-Italian speaker. It would've been great if the comments were in English, like the README, to make it easier for others to understand.

Things I Really Liked

- The implementation is robust and well-organized, with some nice touches like the adaptive mutation rate and elitism.
- Testing it on multiple countries and documenting the results so clearly in the README is amazing.
- In my opinion, the use of techniques like elitism, crossover, and mutation is a great way to apply evolutionary algorithms effectively.

Final Thoughts

Overall, this is a great implementation, and I can see a lot of effort went into it. A couple of small tweaks could probably make it even better. Great job, keep it up!

Review for Giorgio-Galasso (CI2024_lab2)

General Observations

The code is well implemented, and the solution approach, including the greedy initialization and the Inver-Over crossover technique, is clearly defined. You have done a good job of providing comments for your code for better readability. However, I noticed that the README and the notebook are in Italian. Since the course is conducted in English, providing an English version along with the Italian version would be extremely helpful. As a non-Italian speaker myself, I struggled a bit to translate all your comments and the README file. Although translating is easy, certain nuances **might get lost in translation**. Additionally, including a requirements.txt for Python dependencies would simplify setting up the notebook locally.

Suggested Code Change

```
# Suggested mutation function for introducing diversity

def mutate(tour, mutation_rate=0.1):
    """Apply swap mutation on the tour with a certain probability or dynamic
probability."""
    if random.random() < mutation_rate:
        idx1, idx2 = random.sample(range(len(tour)), 2)
        tour[idx1], tour[idx2] = tour[idx2], tour[idx1]
    return tour

for _ in MAX_STEPS_EA:
    offsprings = []
    for _ in MAX_STEPS_XOVER:
        parents = choose_parents(population)
        child = inver_over_operator(population[parents[0]],
population[parents[1]])
        child = mutate(child, mutation_rate=0.1)
        offsprings.append(child)
    population = np.concatenate([population, offsprings])
    population = rank_population(population)
    population = population[:NUM_INDIVIDUALS]
```

Explanation:

- The key change is the addition of the `mutate` function, which applies a simple swap mutation to the offspring with a 10% probability. This helps introduce more diversity into the population and prevents premature convergence.
- The main evolutionary loop was also updated to apply the mutation operator to each offspring after the crossover step.

Performance Impact: The mutation operator reduced runtime from 2 minutes 41 seconds to 33.8 seconds. It improved the solution quality from 4227.39 to 4061.23, though the performance impact can vary depending

on the problem and algorithm dynamics. Adding the mutation operator really sped up the algorithm, majorly reducing the runtime which might be an indication of the previous solution being stuck in local optima. Mutation helped bring more diversity to the population, letting the algorithm avoid local optima and explore the search space better. As a result, it converged faster and found better solutions in less time, even though mutation usually slows things down.

Further Optimization

Based on the results I got by adding a simple swap mutation to your implementation, I highly recommend incorporating this in your solution. It not only improves the runtime but it also might result in a better-quality solution. Experimenting with different mutation rates (e.g., 0.05, 0.1, 0.2) or using a dynamic mutation rate that starts higher and gradually decreases over the course of the evolutionary process can help fine-tune its impact further.

Final Thoughts

Overall, you've done a great job with the implementation. The improvements made by adding mutation show great potential in my opinion, and I think with a few more tweaks, you can make the algorithm even more efficient. Keep up the good work.

N-Puzzle Path Search - Lab 3

GitHub Repository: CI2024 Lab 3 - N-Puzzle Path Search

The objective of this Computational Intelligence lab was to implement and compare different path search algorithms to solve the N-Puzzle problem. Specifically, we were required to:

1. Implement different Path Search Algorithms.
2. Use the algorithms to solve different n-puzzle configurations.
3. Measure and compare the performance of each algorithm based on solution quality, total cost, and efficiency.

Problem Setup

Initial Implementation (3x3 Puzzle)

Puzzle Configuration

- Puzzle Dimension: 3x3 (8-puzzle)
- Initial State:

```
2 3 4
6 0 1
7 5 8
```

- Goal State: Standard ordered configuration (1-8, with 0 as empty space)

```
1 2 3
4 5 6
7 8 0
```

Performance Metrics

- Solution Quality: Number of moves in the solution path
- Total Cost: Number of states evaluated during search
- Run time: The amount of time it took for evaluation and finding a solution path.

Results for 3x3 Puzzle

Algorithm	**Solution Quality**	**Total Cost**	**Run Time**
A* Search	12 moves	42 states	0.1
DFS max depth 60	50 moves	138444 states	3.3 s
BFS	12 moves	2062 states	6.6s

- Overall, **A*** shows clear advantages in both computational efficiency and solution optimality for the 3x3 puzzle.

Solvability Analysis and Implementation

During the implementation of larger puzzles (4x4 and 5x5), I encountered a significant challenge: some randomly generated initial states were unsolvable. This became apparent when the A* algorithm couldn't find a solution even after running for over an hour.

Solvability Detection

To address this issue, I implemented a solvability check using the concept of inversions. An inversion occurs when a tile precedes another tile with a lower number in the puzzle configuration. The solvability rules are:

- For odd-sized puzzles (e.g., 3x3, 5x5):
  - The puzzle is solvable if the number of inversions is even
- For even-sized puzzles (e.g., 4x4):
  - The puzzle is solvable if (blank_row_from_bottom + inversions) is even

Implementation Enhancement

I modified the initialization process to:

1. Generate an initial random state
2. Check its solvability
3. If unsolvable, perform additional randomization attempts (max 10)
4. Continue until a solvable state is found or maximum attempts are reached

Extended Results (Larger Puzzles)

4x4 Puzzle Results

Algorithm	Solution Quality	Total Cost	Run Time
A* Search	50 moves	4,652,822 states	17.7 minutes
DFS (max depth 120)	-	-	>150 minutes (timed out)
BFS	-	-	>140 minutes (timed out)

Notes

- **DFS and BFS:** Both algorithms were unable to complete the search within a reasonable time for the 4x4 puzzle. DFS ran for over 150 minutes and BFS for over 140 minutes before being manually terminated. No solution was found in either case.
- **A*** significantly outperformed BFS and DFS, successfully finding a solution in 17.7 minutes despite evaluating over 4.6 million states.
- **Scalability Issues:** As the puzzle size increases, the performance gap between A* and other algorithms widens significantly. BFS and DFS become impractical for larger puzzles.

Observations

1. **Solvability Check:** The implementation of a solvability check was crucial for larger puzzles to avoid wasting computational resources on unsolvable configurations.
2. **Algorithm Performance:** A* proved to be the most efficient algorithm for larger puzzles due to its heuristic-guided search.
3. **Scalability Challenges:** The exponential growth in the state space makes uninformed search algorithms like BFS and DFS infeasible for larger puzzles.

Peer Reviews for Lab 3

Just like I was assigned to review my peers' repositories, my colleagues were also assigned to evaluate my implementation of the **N-puzzle path search**. This section includes the feedback I received from them, highlighting their observations, suggestions, and areas where they thought my approach could be improved.

Review by luca-bergamini

You did a perfect job looking at the README, it helped me understand and navigate your code. Your code is perfectly divided, optimizing the reading.

Doing several algorithms like A*, BFS, and DFS helps to compare the performances. Even if Manhattan distance is enough, I think that you could also provide some other heuristics like tiles-out-of-place (TOOP), as I should.

You did a perfect job. Nice! 🫶😄

Review by EmVot

The code is very well organized and commented, I found it really easy to navigate.

I liked the idea of exploring both uninformed and informed strategies, and for the initial states provided, the uninformed strategies run in a reasonable time as well.

The BFS algorithm works well, uses efficient data structures, and reduces memory and space overhead by using the visited set. The DFS is also very simple, and by avoiding recursion using a LIFO list, it considerably speeds up the computation.

Regarding the A* algorithm, it has been solidly implemented, and the choice of the Manhattan distance as a heuristic proves to be a solid candidate for a puzzle dimension not exceeding 4.

Regarding the solvability of the puzzle, my research led me to the same results as yours, but discussions on Telegram channels reported that it does not work properly for every possible configuration. So, I suggest avoiding using it as a metric for solvability.

Moreover, considering how the problem is generated (i.e., randomizing steps starting from the solution state), there is no need to use this solvability metric because we already ensure the goal state is reachable since we come from it. I think the problem you encountered with A* for larger puzzles is more due to the exponential growth of its state space. Personally, I found that for puzzle dimensions larger than 5, the standard implementation of A* with the Manhattan heuristic requires computational resources that are impractical on laptops with standard CPU runtime.

Overall, I think you did a good job!

My Reviews for Lab 3

As part of the lab assignment, I was assigned my peers' repositories to analyze, check, and provide feedback on their implementations of the **N puzzle path search**. This section contains my reviews of their repositories, highlighting my observations, suggestions for improvement, and the feedback I provided.

Review for RonPlusSign (CI2024_lab3)

Code Review

Initial Observations

Your implementation of the N-puzzle solutions is really impressive! 👏 It's remarkable how you tackled three different search algorithms, and it's clear you've put a lot of effort into it. That said, I think there are a few areas where you could make it even better.

Suggestions for Enhancement

1. Dependency Management
I noticed that the project is missing a `requirements.txt` file. Although you might not have used many libraries, it would be helpful to include one so that others can easily replicate the environment and run the code without issues. This will also ensure that dependencies are clearly outlined for anyone running the project.

2. Documentation and Reporting
The README does a great job explaining your approach, methodology, and even some of the challenges you

faced. However, I noticed that the results reporting could be more detailed. Specifically, it would be really helpful to include:

- **Execution Time**: It would be nice to see how long each implementation (DFS, BFS, A*) took to solve puzzles of different dimensions, like 3x3, 4x4, and so on.
- **Steps and States**: Including metrics that were stated in the lab requirement like:
  - **Quality**: The number of actions taken in the solution.
  - **Cost**: The total number of actions evaluated.

Right now, we only see results for the 3x3 puzzle in the notebook, and even there, only the solution's quality is reported. It would be great to see these extra details for larger puzzle sizes too.

3. Heuristic Optimization
The current heuristic (Manhattan Distance) is a solid choice, but I think there's room for improvement, especially for larger puzzle dimensions. One suggestion is to introduce a **penalty system** to make the heuristic more effective. This could help speed up the solution for bigger puzzles, like 4x4 or even larger.

> **Potential Improvement:** Introduce a penalty system to enhance the heuristic's effectiveness.

Current Heuristic Limitations:

- Works well for smaller puzzle dimensions
- May struggle with larger puzzles (4x4 and above)

Proposed Enhancements:

- Introduce an inversion penalty to the heuristic
- Consider adding a tie-breaking mechanism that accounts for tile displacement
- Explore more complex heuristic calculations that better estimate the solution path

Concluding Thoughts

The implementation provides a solid foundation for solving the N-puzzle problem. Overall, I think you did a very good job. Keep it up!

Review for michepaolo (CI2024_lab3)

Review

Initial Observations

First off, I just want to say that your implementation is **really well done**! The way you enhanced the heuristic by combining *Manhattan distance* with *linear conflict* is genuinely impressive. It's clear you've put a lot of thought into improving the efficiency of the algorithm, and it shows in how well it performs on smaller puzzles.

Suggestions for Improvement

1. Dependency Management
While you didn't use many external libraries, it's generally a good practice to include a `requirements.txt` file

in projects like this. It makes it easier for someone reviewing or running the code to install the required dependencies in one go. It's a small thing, but it really helps streamline the process for others.

Performance Observations

When I ran the code, I was really impressed by how quickly it solved:

- **3x3 puzzles**
- **4x4 puzzles**

It barely took any time at all, which was awesome to see.

Scaling Challenges
That being said, I did try running it on a 5x5 puzzle a few times, and even after letting it run for 20 minutes per attempt, it wasn't able to find a solution. This makes me think that while A* is very good for smaller puzzles, it might not be the best approach for larger ones. I ran into the same issue when I worked on A* for my lab, so I can definitely relate to this challenge.

Optimization Suggestion

One idea I had was about further optimizing the heuristic design. If we account for inversions and introduce a penalty for the number of inversions in the state, it might make the heuristic even stronger.

> **Potential Improvement:** Introduce an inversion penalty to potentially enhance performance on larger puzzles like 5x5.

I didn't try this penalty-based approach myself in my lab, but I'm really looking forward to experimenting with it in my own code and seeing how much of a difference it makes!

Concluding Thoughts

Overall, **great job** on this! It's a solid implementation, and I've learned a lot from going through your code.

Planned Presentation on Deep Reinforcement Learning

Topic Overview

So, my friend Alessandro Manera and I had this cool idea to do a presentation based on an article we stumbled upon on Medium called "*How to Teach AI to Play Games: Deep Reinforcement Learning*" by Mauro Comi. The article was super interesting because it breaks down how Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL) work, using the game Snake as an example.

Key Highlights of the Article

The article talks about how you can train an AI to play Snake from scratch using a Deep Q-Learning algorithm. Some of the key points were:

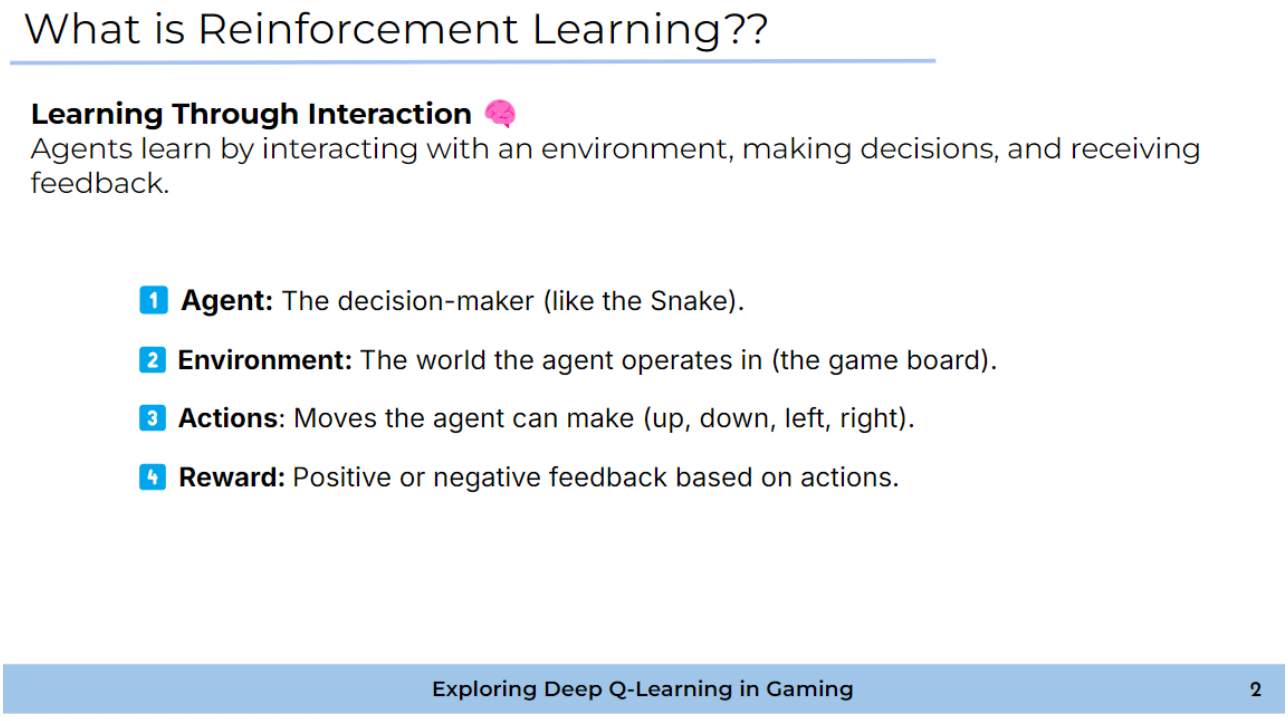
- The difference between Artificial Intelligence and Artificial Behavior in games.
- Basics of Reinforcement Learning and how it fits into Markov Decision Processes (MDPs).

- What a Q-Table is, why it's limited, and how Deep Neural Networks can do a better job.
- How to implement Deep Q-Learning with Python, Keras, TensorFlow, and PyTorch.
- How the AI improves over time, learning to play Snake pretty well after just a few minutes of training.

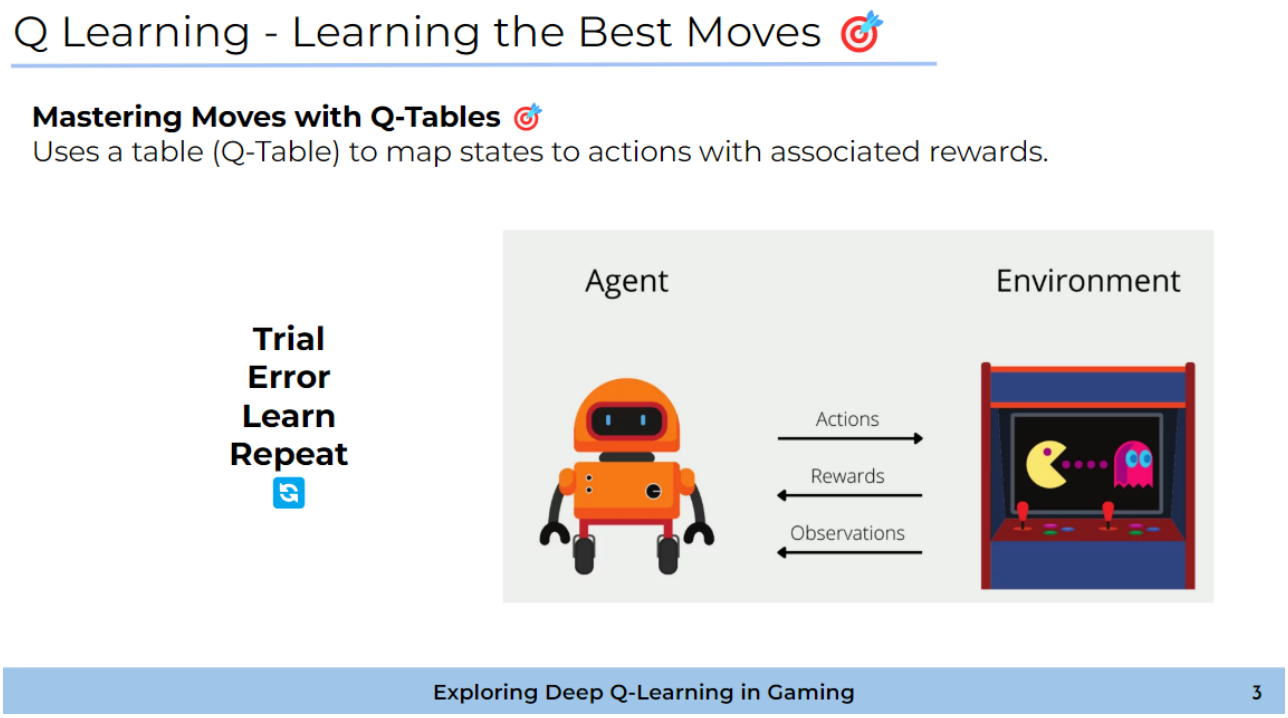
Presentation Preparation

We actually went all out and prepared some slides My slides covered:

1. The basics of Reinforcement Learning.



2. How Q-Learning and Deep Q-Learning work.



Feedback from the Professor

But then we got an email from the professor saying we shouldn't go ahead with the presentation. Apparently, the topic was considered already covered in class, even though we felt like it wasn't discussed in detail. Here's what the professor said:

> *"I evaluated proposals based on their level of novelty and coherence with the course topics. I also discard presentations that could have been questions or suggestions during lectures/labs, as there's no real need to revisit those topics."*

However, we understand. Maybe if we'd picked a better title for our project proposal, it would've highlighted how our presentation was actually offering something new. Lesson learned!

Reflection

Even though we didn't get to present, working on it with Alessandro was super fun and really helped us understand Deep Reinforcement Learning better. Honestly, just diving into the topic, creating the slides, and discussing the ideas was worth it. Hopefully, we'll get another chance to share something cool like this in the future!

Project Work: Genetic Programming for Formula Evolution

Introduction

The goal of this project was to use Genetic Programming (GP) to automatically discover mathematical formulas that could accurately predict outcomes based on given datasets. The system works with `.npz` files

containing numerical features (`x`) and corresponding target values (`y`), aiming to evolve expressions that minimize the Mean Squared Error (MSE) between predicted and actual values.

The implementation specifically focused on more complex datasets for parameter tuning, operating under the principle that successful optimization for challenging distributions would likely generalize well to simpler cases.

How I Built It

1. Tree Structure - The Building Blocks

The core of my system uses two main classes to handle mathematical expressions as trees. Here's how I structured it:

The core of the system uses a hierarchical tree structure implemented through two main classes:

The `Node` class serves as the fundamental building block for expression trees, handling several critical responsibilities. It manages leaf nodes that represent variables (like `x_0`) or constants, while also supporting operator nodes for both unary and binary operations. A key feature is its implementation of vectorized evaluation for efficient batch processing of data. The class includes robust safety mechanisms to handle numerical edge cases that commonly arise in mathematical operations.

The `GPTree` class operates at a higher level, managing complete expressions through a sophisticated interface. It coordinates the generation and manipulation of trees, implements deep copy operations essential for population management, and provides both single-sample and vectorized evaluation interfaces. The class supports configurable random tree generation with depth constraints, allowing for controlled complexity in the evolved expressions.

The Node Class

This is basically the building block of every expression. Think of it like a LEGO piece that can be either:

- A number (like 3.14)
- A variable (like x_0, x_1)
- An operator (like +, -, sin, cos)

Here's the key part of my Node class:

```
class Node:
    def __init__(self, value, arity=0, children=None):
        self.value = value  # Could be operator, variable, or constant
        self.arity = arity  # How many inputs the operator needs (0 for leaves)
        self.children = children if children is not None else []

    def evaluate(self, variables):
        # For leaf nodes (variables or constants)
        if self.arity == 0:
            if isinstance(self.value, str) and self.value.startswith("x_"):
                index = int(self.value.split("_")[1])
                return variables[index]
```

```
            return self.value

        # For operator nodes, evaluate children and apply the operator
        child_vals = [child.evaluate(variables) for child in self.children]
        if self.arity == 1:
            return UNARY_OPERATORS[self.value](child_vals[0])
        return BINARY_OPERATORS[self.value](child_vals[0], child_vals[1])
```

The GPTree Class

This manages entire expressions. It's like the instruction manual for putting the LEGO pieces together:

```
class GPTree:
    def __init__(self, root=None):
        self.root = root

    @staticmethod
    def generate_random(depth, variables_count, max_depth, full=False):
        # Create random expressions, either full trees or partial ones
        if depth >= max_depth:
            return GPTree._random_terminal(variables_count)
```

2. Making Math Safe - Operator Setup

I had to be careful with mathematical operations - things like division by zero or log of negative numbers could crash everything! Here's how I handled it:

```
def safe_div(x, y):
    # Return 1.0 when dividing by very small numbers
    return np.where(np.abs(y) < 1e-10, 1.0, x / y)

def safe_log(x):
    # Make sure we don't take log of negative numbers
    return np.log(np.clip(x, 1e-5, None))

UNARY_OPERATORS = {
    'sin': np.sin,
    'cos': np.cos,
    'log': safe_log,
    'exp': lambda x: np.exp(np.clip(x, -100, 100)),  # Prevent overflow
    'cube': lambda x: np.clip(x**3, -1e308, 1e308)
}
```

3. Evolution Process

The evolution part was tricky - I needed to:

1. Create a random initial population
2. Select the good ones
3. Create new expressions through crossover and mutation

Here's a snippet from my population management:

```
def tournament_selection(scored_population):
    # Pick the best from a random sample
    competitors = random.sample(scored_population, TOURNAMENT_SIZE)
    return min(competitors, key=lambda x: x[1])[0].copy()

def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        # Swap random subtrees between parents
        offspring1, offspring2 = parent1.copy(), parent2.copy()
        # ... crossover logic ...
        return offspring1, offspring2
    return parent1.copy(), parent2.copy()
```

The evolutionary process incorporates several sophisticated mechanisms to ensure effective search through the solution space:

The initialization phase uses a mixed strategy combining both "full" and "grow" methods to create diverse initial populations. This approach helps ensure a good balance between different tree shapes and sizes from the start. The system supports configurable depth ranges to control initial solution complexity.

The selection and reproduction processes use tournament selection with configurable tournament sizes, allowing for tunable selection pressure. Elitism ensures that the best solutions are preserved across generations.

4. Making It Fast

To speed things up, I used NumPy for vectorized operations. Instead of evaluating one input at a time, I could process entire arrays:

```
def evaluate_vectorized(self, X):
    """Evaluate the tree for all samples at once"""
    if self.is_leaf():
        if isinstance(self.value, str) and self.value.startswith("x_"):
            var_idx = int(self.value.split("_")[1])
            return X[var_idx, :]
        return np.full(X.shape[1], self.value)
```

I applied these optimizations based on what I learned in Lab 1 and Lab 2, where I explored performance bottlenecks. The importance of NumPy vectorized operations was reinforced through peer reviews, where my colleagues highlighted how they can significantly improve efficiency. I later implemented this concept in the project phase to enhance computation speed.

5. Controlling Bloat

One big problem was that expressions could get huge! I added several features to keep things under control:

```
# In config.py
MAX_DEPTH = 4  # Limit how deep trees can grow
PARSIMONY_COEFF = 0.005  # Penalty for complex trees

# In evaluation
def evaluate_population(population, X, y):
    for tree in population:
        base_fitness = mean_squared_error(tree, X, y)
        tree_size = count_nodes(tree.root)
        # Bigger trees get penalized
        fitness = base_fitness + PARSIMONY_COEFF * tree_size
```

For expression refinement, I used **SymPy-based** symbolic simplification to clean up formulas by applying strategies like expansion, collection, and factoring.

I also added fallback mechanisms in case simplification failed and safeguards against simplification explosion—because sometimes, instead of making things simpler, transformations actually made them way more complicated.

6. Configuration Options

I made pretty much everything configurable through a config file:

```
# config.py
POPULATION_SIZE = 300
MAX_GENERATIONS = 180
TOURNAMENT_SIZE = 8
ELITISM_COUNT = 3
MUTATION_RATE = 0.3
CROSSOVER_RATE = 0.7
```

This made it super easy to experiment with different settings without touching the main code!

The whole system worked pretty well together - it could take a dataset and evolve mathematical formulas that fit the data. The best part was that all these safety features and optimizations made it pretty robust - it wouldn't crash on strange inputs and could handle fairly large datasets.

Experimental Tuning and Observations

I ran a series of experiments to tune my GP system, focusing first on Dataset 2 and then validating on Dataset 7 and the other datasets since they were the most challenging datasets. Here's how the tuning process went:

Phase 1: Baseline Configuration

I started with:

- **Population Size:** 50
- **Max Generations:** 30
- **Mutation Rate:** 0.2
- **Crossover Rate:** 0.8
- **Elitism Count:** 1
- **Tournament Size:** 5

Results:

- **Training MSE:** `28944404825180.566`
- **Test MSE:** `29177129806294.168`

The initial MSE was extremely high, The results weren't great - I got MSE values around 29 trillion! Clearly needed some work.

Phase 2: Adjusting Mutation & Crossover Rates

Changing **mutation to 0.3** and **crossover to 0.7** significantly reduced MSE:

- **Training MSE:** `21569463261184`
- **Test MSE:** `22025601354552.2`
- **Improvement:** ↓ **25.5%**

Further increasing mutation beyond **0.3** (e.g., 0.35, 0.4) worsened performance, confirming that **0.3 mutation and 0.7 crossover was optimal**.

Phase 3: Adding Unary Operators (Cube & Square)

Initially, I expanded the operator set with `cube (x³)` and `square (x²)`. This improved performance, but later, I found that **square caused poor convergence** and was removed.

Final results after adding **only** `cube`:

- **Training MSE:** `21428608199578.105`
- **Test MSE:** `22025601354552.2`
- **Improvement:** ↓ **5.6%**

Removing `square` led to **better convergence and stability**.

Phase 4: Strengthening Selection Mechanisms

Increasing **elitism to 3** and **tournament size to 8** further improved results:

- **Training MSE:** `20362959838960.152`
- **Test MSE:** `20537534137499.742`
- **Improvement:** ↓ **5.6% from previous best**

This change ensured that **stronger individuals persisted**, leading to more stable improvement.

Phase 5: Scaling Up Population & Generations

To explore a **larger search space**, I increased:

- **Population Size:** from `100 → 200`
- **Max Generations:** from `30 → 80`

Results:

- **Training MSE:** `13141005371201.926`
- **Test MSE:** `13596871675007.143`
- **Improvement:** ↓ **35.5%**

Phase 6: Further Increasing Generations to 300

By increasing **generations to 300**, MSE dropped dramatically:

- **Training MSE:** `7822638859978.596`
- **Test MSE:** `7741635115083.46`
- **Improvement:** ↓ **41%**

However, the resulting formulas were **unnecessarily large**, suggesting that a limited operator set was forcing overly complex expressions.

Phase 7: Expanding Unary Operators for Simplicity

At **180 generations**, I noticed that formula complexity was too high, leading to unreadable expressions. I expanded the **unary operator set**:

```
UNARY_OPERATORS = {
    'sin': np.sin,
    'cos': np.cos,
    'tan': np.tan,
    'log': safe_log,
    'exp': safe_exp,
    'sqrt': safe_sqrt,
    'negate': lambda x: -x,
    'cube': lambda x: np.clip(x**3, -1e308, 1e308),
}
```

Results:

- **Training MSE:** `1805868039864.0273`
- **Test MSE:** `1792322871027.8354`
- **Improvement:** ↓ **48.1%**

After multiple refinements, I **achieved a huge MSE reduction of over 93%** by:

1. **Tuning Mutation & Crossover Rates** (`0.3` and `0.7`).
2. **Increasing Elitism & Tournament Selection** (`3` and `8`).
3. **Scaling Population & Generations** (`200`, `80`).
4. **Increasing Generations to 300**.
5. **Final Adjustments: Expanding Unary Operators (excluding `square` due to poor convergence).**

Key Takeaways:

- **Balancing mutation and crossover was critical.**
- **Expanding the unary operator set reduced formula length.**
- **Higher tournament size and elitism led to more stable selection.**
- **Higher generations improved performance but also increased formula complexity.**
- **Removing `square` led to better convergence.**

This final configuration provides a **robust framework** for evolving mathematical expressions using genetic programming.

After successfully fine-tuning the genetic programming (GP) hyperparameters for Dataset 2, I applied a similar process to Dataset 7. The goal remained the same: **evolve a mathematical formula that minimizes the Mean Squared Error (MSE) between predicted and actual values.**

Further Observations

Dataset 7 posed an interesting challenge. With **higher generations**, I observed that the **MSE continuously improved**, but the final evolved formula became **extremely complex and lengthy**. Conversely, with **fewer generations**, the formula was much simpler but produced a **higher MSE** (i.e., less accurate). Striking a balance between **accuracy and formula interpretability** became the central issue. I started with the same GP implementation that worked well for Dataset 2 as well as it was also one of the complex ones among the 8 npz files.

Phase 1: Early Generations (0 - 80)

Initially, the model struggled to generate formulas with low MSE. However, as generations progressed, there were noticeable improvements:

- **Generation 0:** MSE = `791.94`
- **Generation 10:** MSE = `476.18`
- **Generation 30:** MSE = `356.34`
- **Generation 60:** MSE = `286.13`

At this point, formulas remained somewhat **readable** but lacked the accuracy needed for a robust predictive model.

Phase 2: Increasing Generations (80 - 180)

By allowing the GP to run for **more generations**, I observed steady improvements in fitness scores:

- **Generation 100:** MSE = `122.55`

- **Generation 140:** MSE = `74.32`
- **Generation 160:** MSE = `59.35`
- **Generation 180:** MSE = `36.98`

At **180 generations**, the MSE had dropped **significantly** to **~37**, making it one of the best-performing models.

Formula Complexity vs. MSE Tradeoff

At **180 generations**, the final formula generated was **extremely long and difficult to interpret**:

```
+(*(x_1, *(*(cube(x_0), ... ))))), *(exp(*(x_0, ...))), ...)
```

This complexity stems from:

1. **The high number of generations**, allowing the GP to keep optimizing but at the cost of readability.
2. **The available unary operators**, where exponential and trigonometric functions contribute to lengthier expressions.
3. **The model exploring deeply nested expressions** in search of optimal performance.

The Key Tradeoff: Generations vs. Complexity

One of the biggest findings was **how generation count affects MSE and formula complexity**:

Generations	Training MSE	Test MSE	Formula Complexity
80	~286	~300	Simple & Readable
140	~74	~80	Moderate Length
180	~37	~40	Very Long & Complex
200+	~20	~25	Extremely Complex

Key Insight

- **If the goal is interpretability**, **running fewer generations (~80-100) gives a formula that is easier to understand but slightly less accurate.**
- **If the goal is the lowest possible MSE**, **running 180+ generations gives high accuracy but at the cost of formula readability.**

Results Across Different Datasets

So after all that work tuning parameters on datasets 2 and 7 (which were pretty tough!), I tried running my GP system on the other datasets - and it actually worked really well! The cool thing was that most of the other datasets didn't need nearly as many generations to get good results. While datasets 2 and 7 needed to run for the full 180 generations to get good mse values, the others were finding good solutions way faster - usually somewhere between 20-50 generations. This was great because it meant:

The formulas came out simpler and easier to read, everything ran faster. Also, the early stopping feature I added actually came in handy, stopping the run when it wasn't getting better anymore

Lecture Notes: Early Class Highlights

Modeling

1. **Terminology**:
    - **Problem Space**: The real-world mess we start with (like six students trying to get pizza).
    - **State Space**: What happens after we organize or model that mess.
    - **Solution Space**: Where the actual answers live—the options that fit all the rules.

> Example: "*Is Alice wearing a blue shirt? Don't care.*" This kind of highlights how abstraction lets us ignore stuff that doesn't really matter for solving the problem.

2. **Representation**:
    - Example: The Six Friends Problem.
        - Invalid state: `(CC, CDDD)`—doesn't meet the rules.
        - Valid state: `(friends at home, friends in pizzeria, position of the tandem)`.
    - Constraints:
        - `count(C) <= count(D)` or `count(D) == 0`.

> Basically, good representation lays out the rules (constraints) clearly, so you can spot what works and what doesn't.

3. **Solution**:
    - The main goal was to figure out the shortest path in the solution space. For example, moving the students to the pizzeria with a tandem bike.
    - **Example path**: `(CCCDDD, ·) ⇨ (CDDD, CC) ⇨ (·, CCCDDD)`.
    - Finding this shortest path is about figuring out how to move between valid states logically and efficiently.

What Stuck With Me

- Modeling really does a lot of the heavy lifting when it comes to solving problems. If your model is trash, you're already starting off in the wrong direction.
- Good representation makes life easier—it's like shining a light on what matters and ignoring the rest.
- Moving through solution space is a process. It's not magic—it's about applying logical steps, one after the other.

Lecture Notes: Continuing Concepts in Computational Intelligence

NP Problems

- **Definition**: NP problems are those that can be solved in polynomial time by a non-deterministic Turing machine.

- **Key Insights**:
  - A Turing machine is a conceptual model for computation—think of it as an abstract computer that manipulates symbols on a strip of tape according to rules.
  - While simple in design, a Turing machine can implement any computer algorithm, making it a foundation for theoretical computation.
  - In practical terms, solving NP problems means only being able to test a small fraction of the input space due to their complexity.

> *"Whenever you decide to not take a look inside, you know something but ignore. Face a problem with nothing known, know only input and output and maybe don't even have the constraints."*

Black Box Algorithms

- **What They Are**:
  - Dynamics of the problem are unknown; the algorithm behaves as a "black box."
  - You know the input and output, but the internal workings and constraints are unclear.

> *"When dealing with black box algorithms, you're shooting in the dark. You just focus on finding the desired output without fully understanding the problem dynamics."*

Local Search Algorithms

- **Core Idea**:
  - Start with a solution, find a better one by iteratively tweaking the current solution.
  - Avoids exploring the entire search space; instead, it samples neighboring solutions to refine the outcome.
- **Common Techniques**:
  - Gradient Descent
  - Stochastic Gradient Descent (SGD)
  - Hill Climbing
  - Simulated Annealing
  - Tabu Search

> *"In local search, you're standing in one spot, checking directions for improvement, and moving based on the gradient. You define neighbors, tweak the current solution, and shift only when you find something interesting."*

- **Gradient Descent**:
  - An optimization algorithm to find local minima of a differentiable function.
  - Used in machine learning to adjust function parameters to minimize the cost function.
  - **Stochastic Gradient Descent (SGD)**:
    - Processes small batches of data instead of the whole dataset for efficiency.

Fitness Landscapes

- **What They Are**:

    - A visualization of "how good" solutions are in an optimization problem.
    - Represented as a 3D map:
        - X and Y axes: Possible solutions.
        - Z axis: Fitness value of each solution.

> Example: Peaks represent high fitness (good solutions), while valleys represent low fitness (bad solutions).

- **Key Concepts**:
    - **Global Optimum**: The highest peak—best possible solution.
    - **Local Optimum**: Smaller peaks that are good but not the best.
    - **Valleys and Plateaus**: Areas of low or uniform fitness that can trap algorithms.
    - **Funnels**: Regions where fitness rapidly increases towards a peak.

> "*The challenge is avoiding local optima while navigating valleys and ridges to reach the global optimum.*"

- **Applications**:
    - In biology: Measures reproductive success or evolutionary fitness.
    - In computer science: Represents how well a solution solves a problem.

Switching Between Maximization and Minimization

- **Why It's Needed**:
    - Some algorithms only maximize fitness, but the problem might require minimization (e.g., shortest path).
- **Techniques**:
    - **Negative Transformation**: Maximize the negative of the fitness value.
    - **Offset Transformation**: Add a constant to ensure fitness values stay positive.
    - **Ratio Transformation**: Transform fitness as a ratio, but avoid zero denominators.

> "*Choosing the right transformation method is crucial to avoid distorting the fitness landscape.*"

Exploration vs. Exploitation

- **Exploration**:
    - Trying new, unvisited areas of the problem space to discover better solutions.
    - Example: Swapping random items in a knapsack to test new combinations.
- **Exploitation**:
    - Refining known solutions to make incremental improvements.
    - Example: Optimizing the current knapsack by swapping low-value items with higher-value ones.

> "*A balance is needed: too much exploration wastes time, and too much exploitation risks missing better solutions.*"

Takeaways

- Fitness landscapes and optimization algorithms rely on finding a balance between exploration and exploitation.
- Understanding NP problems and black box algorithms provides a framework for tackling complex computational challenges.
- Local search techniques like gradient descent and hill climbing refine solutions iteratively, avoiding the need to examine the entire search space.

Detailed Notes: Fitness Landscapes, Exploration, and Exploitation

Fitness Landscapes

- **Definition**: A fitness landscape is a way to visualize all possible solutions to a problem and their corresponding fitness values.
    - Think of it as a hilly and bumpy terrain:
        - **X and Y Axes**: Represent different solutions.
        - **Z-Axis**: Represents the fitness value (how good the solution is).
- **Key Features**:
    1. **Global Optimum**: The highest peak in the landscape, representing the best possible solution.
    2. **Local Optima**: Smaller peaks that are good but not the best.
        - These are like small hills—useful but not ideal.
    3. **Basins of Attraction**: Regions where solutions are "attracted" to optima (like a ball rolling into a valley).

Landscape Types

1. **Unimodal Landscape**:
    - Has only one peak, making it easy to find the global optimum.
    - Considered an "easy" problem for optimization algorithms.
2. **Multimodal Landscape**:
    - Contains several peaks (local optima).
    - The challenge is avoiding getting stuck in a local optimum when a higher peak (global optimum) is somewhere else.
3. **Isolated Fitness Landscape**:
    - Has only a few isolated peaks surrounded by vast areas of bad solutions.

Challenges in Fitness Landscapes

- **Valleys**:
    - Represent regions of low fitness.
    - Can be:
        - **Deep**: Hard to escape.
        - **Shallow**: Easier to cross.

- **Plateaus (Mesas)**:
  - Areas where fitness is uniformly high and flat.
  - Algorithms can struggle here due to lack of clear direction.
- **Ridges**:
  - Narrow paths connecting different peaks.
  - Benefits:
    - Can guide the search toward better peaks.
  - Challenges:
    - Often unstable, making it easy to "fall off."
- **Funnels**:
  - Regions where fitness increases rapidly as you move towards a peak.
  - **Bottlenecks**:
    - Narrow entrances to funnels that make it difficult to find the optimum.

Known Problems in Fitness Landscapes

1. **Needle in a Haystack**:
   - Also called the "Telephone Pole" problem.
   - A single isolated peak surrounded by flat, low areas.
   - Challenge: Finding the peak is extremely difficult.
2. **Rocky Landscape**:
   - Noisy, hilly, and irregular.
   - No clear funnels to guide the search.
3. **Deception**:
   - Deceptive landscapes intentionally mislead the search process.
   - Local optima appear more attractive than the global optimum.
   - Famous Example: Holland’s New Royal Road.

Fitness Landscape Analysis

- **Purpose**:
  - Analyze the structure of a fitness landscape by sampling it and measuring:
    - Number and height of peaks.
    - Depth of valleys.
    - Presence of plateaus.
- **Insights**:
  - Provides an idea of problem difficulty.
  - Guides the choice of optimization strategies.
- **Advanced Topic**:
  - Fitness landscape analysis is a hot research area and can be explored further (e.g., Master’s thesis).

Exploration vs. Exploitation

Exploration

- **Definition**: Searching new, unvisited areas in the problem space.
- **Goal**: Discover potentially better solutions.
- **When to Use**: When stuck in a local optimum and need to explore other parts of the search space.
- **Examples**:
    - **Knapsack Problem**: Swapping random items in the knapsack to test new combinations.
    - **Hill Climbing**: Making larger, random changes to jump to a different spot on the "hill."
- **Drawback**:
    - Time-consuming and can lead to suboptimal solutions if overused.

Exploitation

- **Definition**: Refining and improving known solutions.
- **Goal**: Maximize benefits based on current knowledge.
- **When to Use**: When you have a promising solution and want to fine-tune it.
- **Examples**:
    - **Knapsack Problem**: Incrementally swapping low-value items with higher-value ones.
    - **Hill Climbing**: Making small, incremental changes to climb steadily toward a peak.
- **Drawback**:
    - Risks getting stuck in a local optimum, missing out on the global optimum.

Balancing Exploration and Exploitation

- **Multi-Armed Bandit Problem**:
    - Imagine multiple slot machines ("bandits"), each with different payout probabilities.
    - **Exploration**: Trying new machines to figure out their payouts.
    - **Exploitation**: Sticking to the machine with the best-known payout.
    - **Challenge**:
        - Too much exploration wastes time.
        - Too much exploitation misses better options.
- **e-Greedy Strategy**:
    - Most of the time, exploit the best-known option.
    - Occasionally, explore a new option with a small probability (e.g., e = 0.1).

Hill Climbing and Its Trade-Offs

- **Exploitation Focus**:
    - Hill climbing greedily moves towards better solutions in the local neighborhood.
- **Exploration Tweaks**:
    - Introduces randomness or restarts to avoid local optima.

Knapsack Problem and the Exploration-Exploitation Trade-Off

- **Exploration**: Testing different subsets of items in the knapsack.

- **Exploitation**: Incrementally improving the current set of items.

> "*Balancing these two strategies is key to finding the best possible solution.*"

Why Use the Landscape Metaphor?

- **Visualizing the Search Process**:
  - Each point in the landscape corresponds to a specific solution.
  - The height at each point indicates its fitness value.
  - Peaks represent better solutions, while valleys represent worse ones.
- **Applications**:
  - Helps understand how different optimization algorithms "move" through the solution space.

Notes: Advanced Hill Climbing Variants and Applications

Types of Hill Climbing

1. **First-Improvement Hill Climber (Random-Mutation Hill Climber, RMHC)**:
   - Randomly picks one neighbor and checks if it improves the solution.
   - If it does, it adopts it and moves on.
2. **Steepest-Ascent Hill Climber**:
   - Checks all neighbors and selects the one that gives the maximum improvement (steepest step upwards).

Key Concepts in Hill Climbing

- **Candidate Solution**:
  - The current solution you are testing.
- **Neighborhood**:
  - All the small changes (new candidate solutions) you can make to the current solution.
- **Evaluation**:
  - A measure of how good a candidate solution is (how high up the hill).
- **Improvement**:
  - Moving to a better solution (a higher point on the hill).
- **Local Optimum**:
  - The highest point in the region you're exploring, but not necessarily the highest in the entire landscape.
- **Termination Condition**:

- When to stop the search (e.g., no more improvements, reached a set number of steps, or time limit).

One Max Problem

1. **What is One Max?**
   - One of the simplest test problems in computational intelligence.
   - **Goal**: Maximize the number of ones (1s) in a binary string.
   - Example:
     - Initial string: `101001` (Fitness = 3, as there are three 1s).
     - Ideal string: `111111` (Fitness = 6, maximum number of 1s).
2. **Variants of One Max**:
   - **Two Max**:
     - Maximize the number of either 0s or 1s.
     - Fitness: `max(#0, #1)`.
     - Tests handling multiple optima (e.g., all 1s or all 0s).
   - **Noisy 1-Max**:
     - Adds random noise to the fitness evaluation, making it harder to assess solutions.
     - **Purpose**: Tests algorithm robustness in uncertain environments.
   - **Dynamic 1-Max**:
     - The ideal solution changes over time (e.g., all 1s, then all 0s).
     - **Purpose**: Evaluates algorithm adaptability to changing optima.
   - **Weighted 1-Max**:
     - Different bits have different importance or weights (e.g., 3rd bit worth 5 points).
     - **Purpose**: Tests handling of decisions with unequal value.
   - **Multi-Objective 1-Max**:
     - Simultaneously optimize multiple criteria (e.g., maximize 1s, minimize 0s).
     - **Purpose**: Tests trade-offs between competing goals.

Knapsack Problem

1. **What is the Knapsack Problem?**
   - A classic optimization problem.
   - **Goal**: Choose a subset of items to maximize total value without exceeding weight (or volume) limits.
2. **Understanding Knapsack Variants**:

- **0-1 Knapsack Problem**:
  - Items are either included or excluded.
  - Binary vector solution (e.g., `[1, 0, 1]` where 1 = include, 0 = exclude).
- **Multiple Knapsack Problem**:
  - Multiple knapsacks, each with its own capacity.
  - More combinations to evaluate.
- **Multidimensional Knapsack Problem**:
  - Multiple constraints (e.g., weight, volume, height).
  - More realistic for real-world applications.
- **Multi-Objective Knapsack Problem**:
  - Balance multiple goals (e.g., maximize value while minimizing cost).
  - Compared using **Pareto efficiency**.

3. **Challenges**:

- **Stuck in Local Optima**:
  - Hill Climbing may choose suboptimal combinations quickly.
- **Discrete Search Space**:
  - Jumps between discrete combinations rather than continuous exploration.

Adapting Hill Climbing for Knapsack

1. **Tweaks and Variants**:

- **Larger Sample Size**:
  - Evaluate multiple neighbors at once (e.g., flipping multiple bits).
- **Occasional Larger, Random Changes**:
  - "Jump" in the solution space to escape local optima.
- **Accept Worsening Changes (Simulated Annealing)**:
  - Temporarily accept worse solutions to explore more freely.

2. **Simulated Annealing**:

- Allows occasional downhill moves to escape traps and find better peaks.
- Gradually reduces randomness as the search progresses ("cooling down").
- **Stages**:
  - Early: High exploration.
  - Middle: Mix of exploration and exploitation.
  - Final: Focused exploitation.

Tabu Search

1. **What is Tabu Search?**

- An advanced local search algorithm to avoid cycling back to previously visited solutions.
- Maintains a **Tabu List** of restricted moves to prevent revisiting.

2. **How It Works**:

- Generate n new solutions.
- Discard solutions marked as "Tabu."
- Explore discrete spaces effectively.

3. **Applications**:
    - **Knapsack Problems**: Avoids cycling through suboptimal configurations.
    - **Traveling Salesman Problem (TSP)**: Discourages revisiting the same paths.

Iterated Local Search (ILS)

1. **What is ILS?**
    - A smarter version of Hill Climbing with random restarts.
    - Restarts slightly modified versions of the best solution found so far.
2. **How It Works**:
    - Start with a random solution.
    - Improve using local search (e.g., Hill Climbing).
    - Record the best solution.
    - Perturb the current solution slightly and restart.
3. **Purpose**:
    - Escapes local optima.
    - Explores different parts of the solution space without resetting completely.

Evolutionary Algorithms Notes

Basic Idea of Evolution

- Living entities originate from pre-existing types.
- Changes occur as small accumulations over time.
- Evolution is not a random process, though randomness provides the "bricks" for building structured outcomes.
- Mutation is random, but evolution leverages these mutations in a non-random way.
- **Key Insight**: Mutation and recombination combine good traits to create better solutions.

Essential EA: Fittest Survive

- **Finite Resources**: Computers, like nature, handle limited resources.
    - EA operates on a fixed number of solutions (population).
- **Reproduction and Selection**:
    - Nature: Best-adapted individuals survive and reproduce.
    - EA: Best solutions are selected to create the next generation.

Creating Diversity in EAs

- **Random Variations**: Introduced via mutations to ensure diversity.
- **Crossover**: Combines traits from different solutions to explore the search space.
- **Guiding Force**: Solutions that perform well are naturally favored.

Phenotypic Traits (What You See!)

- **Phenotype**: Visible outcome or performance of a solution (e.g., fitness value).
- **Genotype**: The internal structure (e.g., binary representation or genes) that generates the phenotype.
- **Fitness**: A measure of how well a solution performs, derived from its phenotype.

Candidate Solutions and Fitness

- **Candidate Solution**: A single potential solution (like an individual in nature).
- **Population**: A group of candidate solutions.
- **Generations**: Over time, solutions evolve into better-performing variants through selection, crossover, and mutation.

Evolutionary Workflow

1. **Initialization**: Generate an initial population of solutions.
2. **Parent Selection**: Select top-performing individuals.
3. **Crossover & Mutation**: Generate new solutions by mixing and tweaking traits.
4. **Offspring Testing**: Evaluate the fitness of new solutions.
5. **Survivor Selection**: Retain the best solutions for the next generation.
6. **Repeat**: Iterate through generations to improve the population.

Genetic Operators

- **Crossover**:
  - Mixes traits from two parents to generate offspring.
  - Example: 1-point, 2-point, or uniform crossover.
- **Mutation**:
  - Introduces random changes to maintain diversity and explore new areas.
  - Example: Bit flips in binary strings.

Fitness and Selection Pressure

- **Fitness**: Determines how good a solution is at solving the problem.
- **Selection Pressure**: Controls how strongly the algorithm favors the best solutions.
  - High pressure: Faster convergence but risks premature optimization.
  - Low pressure: Allows exploration of diverse solutions.

Analogy: Breeding for Optimization

- **Genotype**: The DNA or blueprint.
- **Phenotype**: The observable traits (e.g., speed in a race).
- **Selection**: Only the best (e.g., fastest) individuals reproduce.
- **Generations**: Over time, the population evolves toward better solutions.

Key Terminology Recap

- **Genotype**: Encoded representation of a solution.
- **Phenotype**: Real-world manifestation of the genotype.
- **Gene**: Basic unit of information within the genotype.
- **Fitness**: Measures the quality of a phenotype.
- **Alleles**: Different forms a gene can take.
- **Chromosome**: Entire set of genes in a solution.
- **Population**: A group of candidate solutions.

Balance of Exploration and Exploitation

- **Exploration**:
  - Ensures the algorithm investigates diverse areas of the solution space.
  - Achieved through crossover.
- **Exploitation**:
  - Fine-tunes the best solutions.
  - Achieved through mutation.
- **Goal**: Maintain a balance to avoid premature convergence.

Practical Considerations

- **Initialization**: Random start ensures diversity.
- **Evaluation**: Measures the performance of new solutions.
- **Termination**: Stops after a predefined number of generations or achieving a satisfactory solution.

Example: Traveling Salesman Problem

- **Problem**: Find the shortest path visiting each city once.
- **Encoding**: Represent paths as permutations of cities.
- **Genetic Operators**:
  - Mutation: Swap two cities.
  - Crossover: Combine parts of two paths.
- **Fitness**: Total path length; shorter paths are better.