Computational Intelligence

Assignment 2: Swarm Intelligence

By

Nabila Zahra & Muhammad Youshay

Date: March 13, 2024

1 Coloring Graphs using ACO

The vertex coloring problem is a well-known problem in graph theory, where the objective is to assign colors to the vertices of a graph such that no two adjacent vertices have the same color, using the minimum number of colors possible.

In this question, we present the implementation of the Ant Colony Optimization (ACO) technique to solve the vertex coloring problem. ACO is a Meta-heuristic algorithm inspired by the foraging behavior of ants, where ants deposit pheromones on the paths they traverse, and subsequent ants are more likely to follow paths with higher pheromone concentrations.

1.1 Problem Formulation

The vertex coloring problem is defined as follows:

Given an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the goal is to find a coloring function $c : V \to N$ such that for every edge $(u, v) \in E$, $c(u) \neq c(v)$, and the number of distinct colors used is minimized.

1.2 Algorithm Description

The algorithm starts by initializing the pheromone matrix with a small constant value for all edges of the given undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. It also takes the number of ants (*numAnts*), maximum number of iterations (*maxIterations*), pheromone influence (α), heuristic influence (β), and pheromone evaporation rate (ρ) as inputs. The algorithm aims to find a vertex coloring solution for the graph G with the minimum number of colors used.

The main loop runs for the specified number of *maxIterations*. In each iteration, a set of *numAnts* ants is created. Each ant starts its tour by selecting a random starting node. While the tour is not complete, the ant calculates the probability of selecting each unvisited node based on the pheromone trail and a heuristic function that considers the degree of the node and the colors already assigned to its neighbors. The next node to visit is selected based on the calculated probabilities. The selected node is then visited and assigned the smallest available color that is not used by any of its neighbors.

After an ant completes its tour, the pheromone matrix is updated based on the quality of the ant's solution. The algorithm then finds the best solution among all ants in the current iteration. If the current best solution is better than the global best solution, the global best solution is updated. A local search is performed on the best solution to further improve it. Finally, the pheromone matrix is updated based on the global best solution.

After the main loop completes, the global best solution is visualized by coloring the vertices of the graph. The fitness history, including the best fitness and average fitness, is plotted over the iterations. The algorithm returns the global best solution as the output.

In our implementation, we used the following parameters initially:

- Number of ants: 20
- Maximum iterations: 50
- α (pheromone influence): 0.8
- β (heuristic influence): 0.8
- ρ (pheromone evaporation rate): 0.8

We have also incorporated a local search technique to improve the quality of the solutions found by the ants.

1.3 Code Implementation

The implementation of the Ant Colony Optimization algorithm for the vertex coloring problem is provided in Python. The main components of the implementation are:

1.3.1 AntColonyOptimization Class

This class encapsulates the ACO algorithm and its various components, such as loading the graph, calculating fitness, selecting the next node, performing local search, and running the algorithm.

```
class AntColonyOptimization:
    def __init__(self, numAnts=20, maxIterations=50, alpha
    =0.8, beta=0.8, rho=0.8):
        # Initialize parameters

    def loadGraph(self, filename):
        # Load the graph from the file

    def calculateFitness(self, ant):
        # Calculate the fitness of an ant's solution

    def selectNextNode(self, ant, unvisited):
        # Select the next node for an ant based on pheromone
     and heuristic

    def runAlgorithm(self, filename):
        # Run the ACO algorithm on the given file
        # Initialize ants, iterate, update pheromones
        # Perform local search, visualize and plot the best
    solution

    def performLocalSearch(self):
        # Perform local search to improve the best solution
```

```
21
22      def visualizeSolution(self):
23          # Visualize the best solution using NetworkX
24
25      def plotFitnessHistory(self):
26          # Plot the fitness history over iterations
```

1.3.2 Ant Class

This class represents an individual ant and its solution construction process.

```
class Ant:
    def __init__(self, graph):
        # Initialize the ant with the graph

    def startTour(self):
        # Start the tour by selecting the initial node

    def visitNode(self, node):
        # Visit a node and color it

    def updatePheromone(self, pheromoneMatrix, rho):
        # Update the pheromone matrix based on the ant's
    solution
```

The main function in the code loads the problem instance file and runs the ACO algorithm on it.

```
def main():
    filename = "queen11_11.col"
    antColonyOptimization = AntColonyOptimization()
    antColonyOptimization.runAlgorithm(filename)
```

1.4 Parameter Tuning

We conducted experiments to fine-tune the parameters of the ACO algorithm, including the number of ants, pheromone evaporation rate, and local search probability. We varied these parameters and evaluated the performance of the algorithm using different problem instances.

Table 1: Instances and Parameter Settings

Instance	Iterations	No. of Ants	Alpha	Beta	Rho
1	50	20	0.8	0.8	0.8
2	100	20	0.8	0.8	0.8
3	100	20	1	2	0.8
4	100	30	1	2	0.8
5	200	30	1	2	0.8

Reasons for choosing the parameter values:

1. **Instance 1:** This instance uses the default parameter values which were given to us. These values can serve as a baseline for comparison with other instances.

2. **Instance 2:** This instance increases the number of iterations to 100 while keeping the other parameters the same as Instance 1. This is to observe the impact of increasing the number of iterations on the solution quality.

3. **Instance 3:** In this instance, the values of α (pheromone influence) and β (heuristic influence) are changed to 1 and 2, respectively. These values give more weight to the heuristic function compared to the pheromone trail when selecting the next node. This can help in exploring more diverse solutions.

4. **Instance 4:** This instance increases the Number Of Ants to 30. This gives even more emphasis on the heuristic function compared to the pheromone trail. This is to observe the impact of increasing the number of Ants on the solution.

5. **Instance 5:** This instance increases the number of iterations to 200 while keeping the α, β, and ρ values the same as Instance 4. This is to observe the impact of increasing the number of iterations with increased no of ants.

By running the ACO algorithm with these different parameter settings, we observed the behavior of the algorithm and the quality of the solutions obtained for the vertex coloring problem.

1.5 Results

We tested our implementation on two problem instances: "le450-15b.col" and "queen11_11.col". The algorithm was run on all the instances mentioned above. These were the results we got.

1.5.1 Instance 1 - queen11_11.col

Iteration 0: Best fitness 16, Average fitness 16.95
Iteration 10: Best fitness 15, Average fitness 16.5
Iteration 20: Best fitness 15, Average fitness 16.45
Iteration 30: Best fitness 15, Average fitness 16.35
Iteration 40: Best fitness 15, Average fitness 16.6
Best fitness: 15 AND Average fitness: 16.65 after 50 iterations.

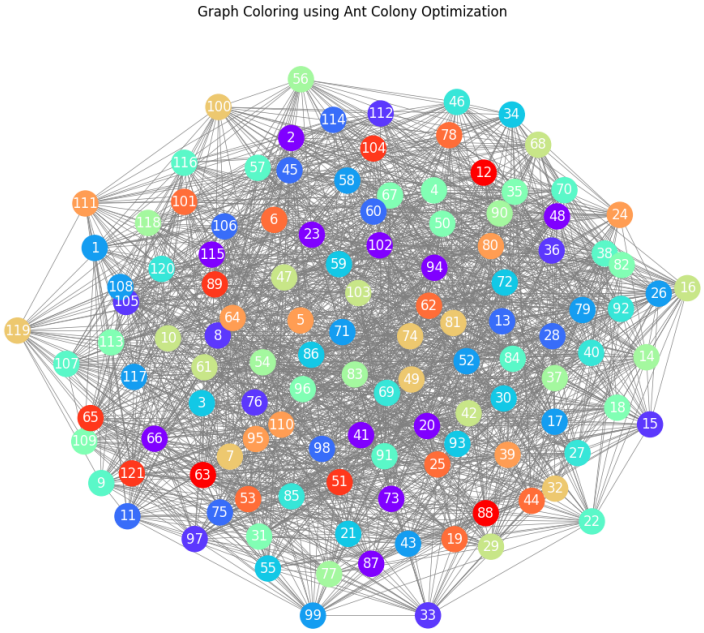

Figure 1: Instance 1 Graph for "queen11_11.col"

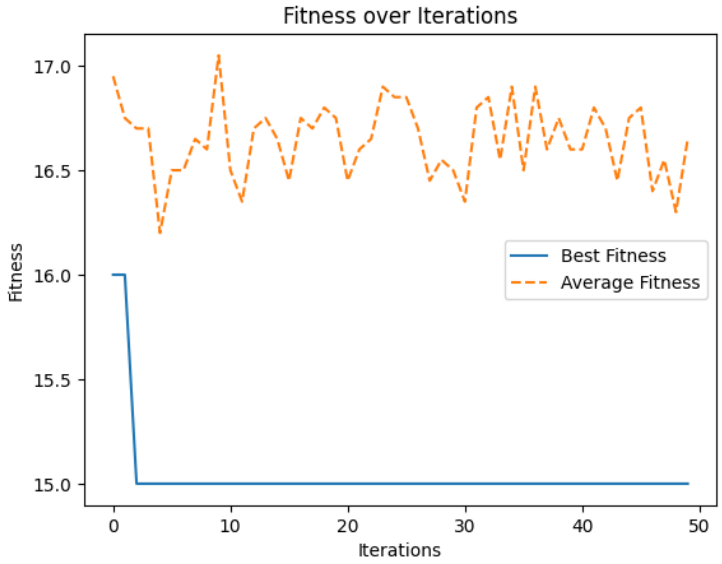

Figure 2: Instance 1 Result for "queen11_11.col"

1.5.2 Instance 2 - queen11_11.col

Iteration 0: Best fitness 16, Average fitness 16.75
Iteration 10: Best fitness 15, Average fitness 16.45
Iteration 20: Best fitness 15, Average fitness 16.5
Iteration 30: Best fitness 15, Average fitness 16.65
Iteration 40: Best fitness 15, Average fitness 16.7
Iteration 50: Best fitness 15, Average fitness 16.65
Iteration 60: Best fitness 15, Average fitness 16.6
Iteration 70: Best fitness 15, Average fitness 16.55
Iteration 80: Best fitness 15, Average fitness 16.8
Iteration 90: Best fitness 15, Average fitness 16.75
Best fitness: 15 AND Average fitness: 16.75 after 100 iterations.

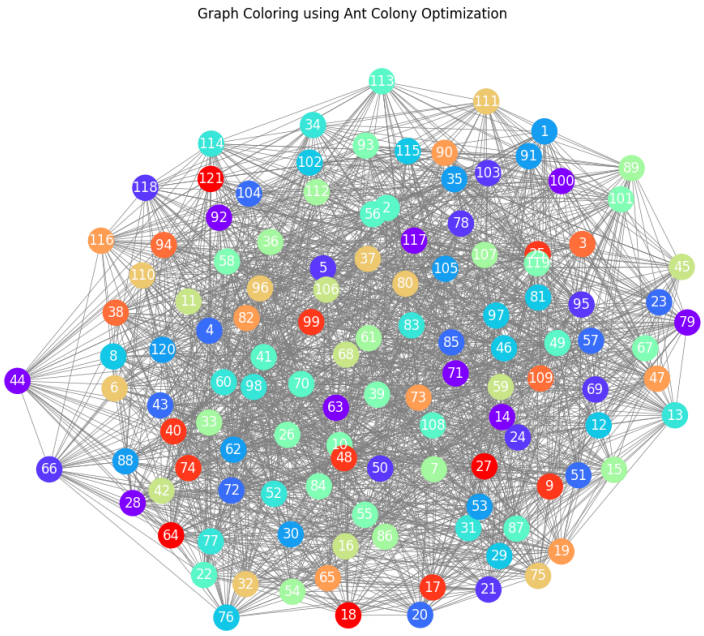

Figure 3: Instance 2 Graph for "queen11_11.col"

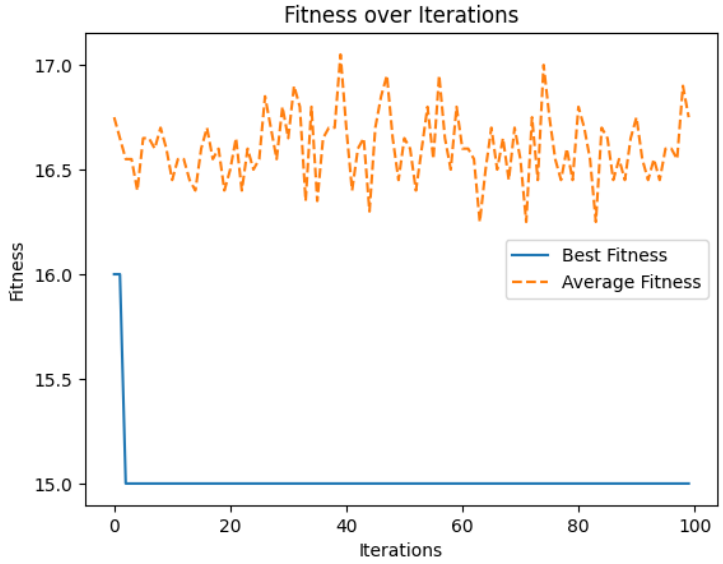

Figure 4: Instance 2 Result for "queen11_11.col"

1.5.3 Instance 3 - queen11_11.col

Iteration 0: Best fitness 16, Average fitness 16.75
Iteration 10: Best fitness 15, Average fitness 16.35
Iteration 20: Best fitness 15, Average fitness 16.6
Iteration 30: Best fitness 15, Average fitness 16.6
Iteration 40: Best fitness 15, Average fitness 16.45
Iteration 50: Best fitness 15, Average fitness 16.6
Iteration 60: Best fitness 15, Average fitness 16.7
Iteration 70: Best fitness 15, Average fitness 16.85
Iteration 80: Best fitness 15, Average fitness 16.35
Iteration 90: Best fitness 15, Average fitness 16.55
Best fitness: 15 AND Average fitness: 16.6 after 100 iterations.

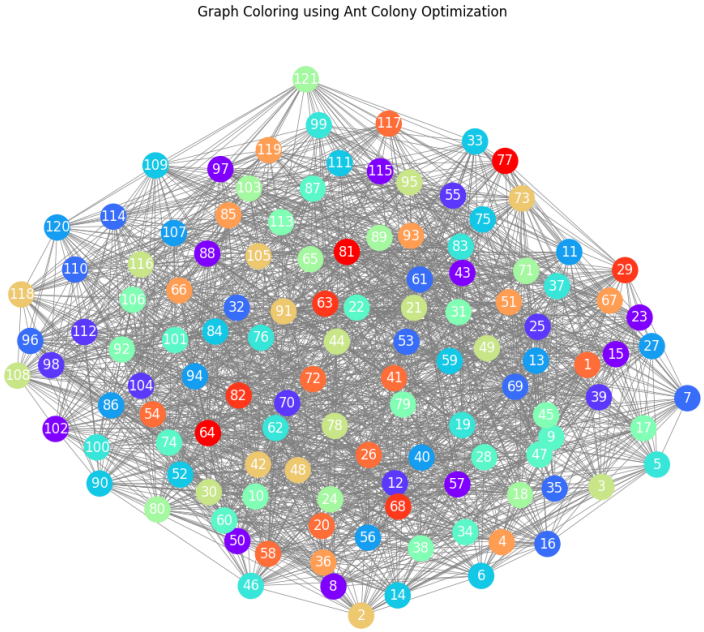

Figure 5: Instance 3 Graph for "queen11_11.col"

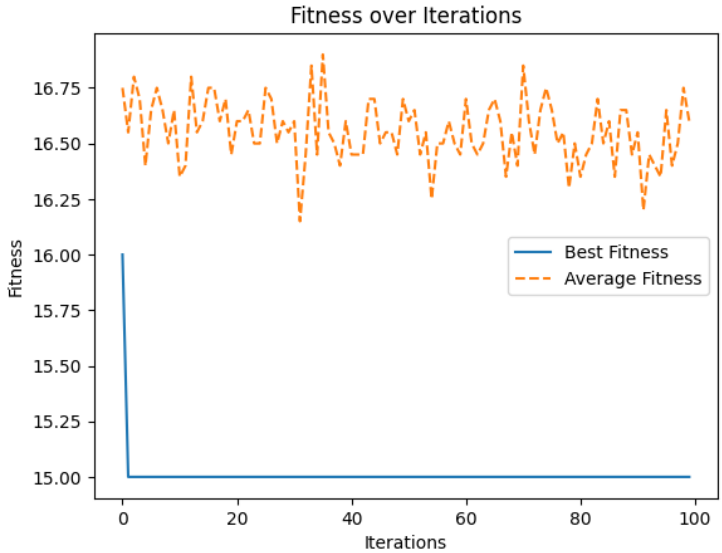

Figure 6: Instance 3 Result for "queen11_11.col"

1.5.4 Instance 4 - queen11_11.col

Iteration 0: Best fitness 15, Average fitness 16.866666666666667
Iteration 10: Best fitness 15, Average fitness 16.666666666666668
Iteration 20: Best fitness 15, Average fitness 16.633333333333333
Iteration 30: Best fitness 15, Average fitness 16.5
Iteration 40: Best fitness 15, Average fitness 16.3
Iteration 50: Best fitness 15, Average fitness 16.3
Iteration 60: Best fitness 15, Average fitness 16.533333333333335
Iteration 70: Best fitness 15, Average fitness 16.433333333333334
Iteration 80: Best fitness 15, Average fitness 16.433333333333334
Iteration 90: Best fitness 14, Average fitness 16.666666666666668
Best fitness: 14 AND Average fitness: 16.466666666666665 after 100 iterations.

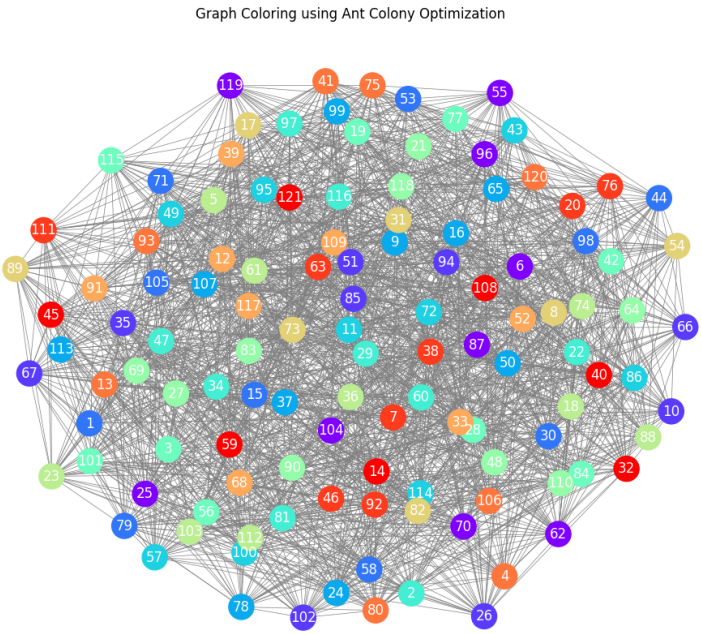

Figure 7: Instance 4 Graph for "queen11_11.col"

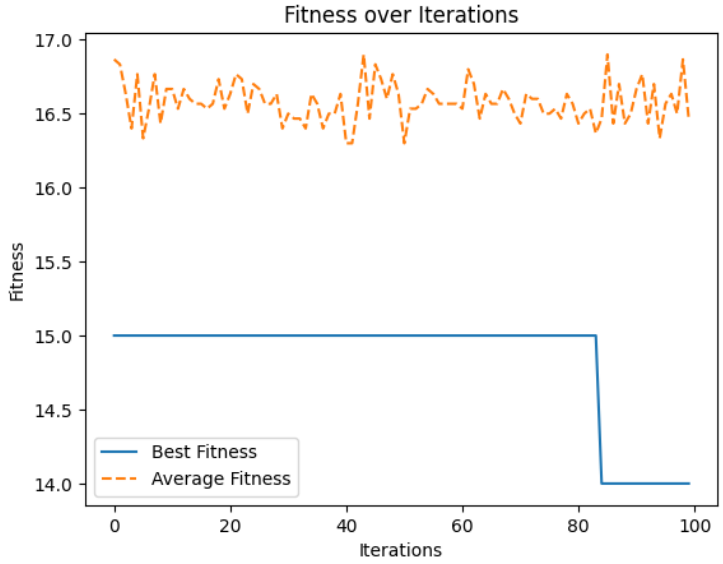

Figure 8: Instance 4 Result for "queen11_11.col"

1.5.5 Instance 5 - queen11_11.col

Iteration 0: Best fitness 16, Average fitness 16.5
Iteration 10: Best fitness 15, Average fitness 16.6
Iteration 20: Best fitness 14, Average fitness 16.7
Iteration 30: Best fitness 14, Average fitness 16.633333333333333
Iteration 40: Best fitness 14, Average fitness 16.6
Iteration 50: Best fitness 14, Average fitness 16.6
Iteration 60: Best fitness 14, Average fitness 16.666666666666668
Iteration 70: Best fitness 14, Average fitness 16.433333333333334
Iteration 80: Best fitness 14, Average fitness 16.9
Iteration 90: Best fitness 14, Average fitness 16.6
Iteration 100: Best fitness 14, Average fitness 16.566666666666666
Iteration 110: Best fitness 14, Average fitness 16.533333333333335
Iteration 120: Best fitness 14, Average fitness 16.533333333333335
Iteration 130: Best fitness 14, Average fitness 16.566666666666666
Iteration 140: Best fitness 14, Average fitness 16.566666666666666
Iteration 150: Best fitness 14, Average fitness 16.466666666666665
Iteration 160: Best fitness 14, Average fitness 16.666666666666668
Iteration 170: Best fitness 14, Average fitness 16.666666666666668
Iteration 180: Best fitness 14, Average fitness 16.4
Iteration 190: Best fitness 14, Average fitness 16.333333333333332
Best fitness: 14 AND Average fitness: 16.5 after 200 iterations.

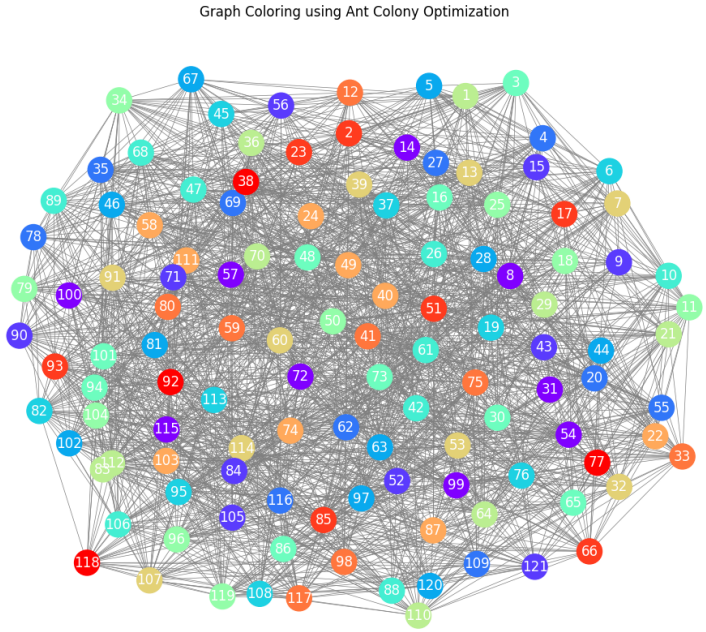

Figure 9: Instance 5 Graph for "queen11_11.col"

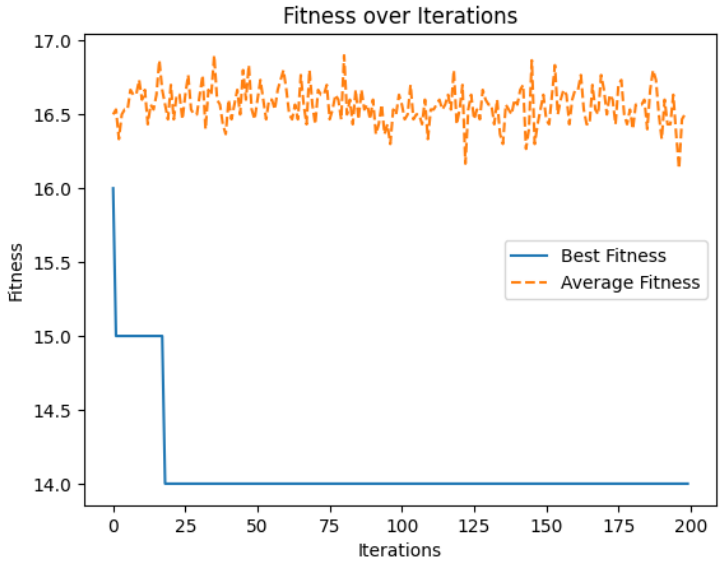

Figure 10: Instance 5 Result for "queen11_11.col"

1.5.6 Instance 1 - le450-15b.col

Iteration 0: Best fitness 20, Average fitness 21.0
Iteration 10: Best fitness 19, Average fitness 20.75
Iteration 20: Best fitness 19, Average fitness 20.4
Iteration 30: Best fitness 19, Average fitness 20.55
Iteration 40: Best fitness 19, Average fitness 20.7
Best fitness: 19 AND Average fitness: 20.55 after 50 iterations.

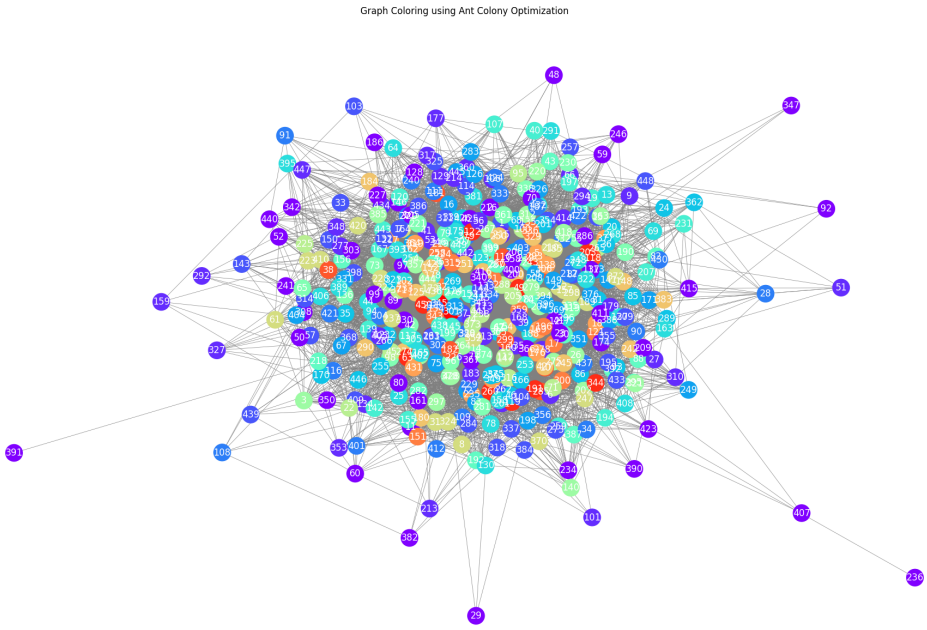

Figure 11: Instance 1 Graph for "le450_15b.col"

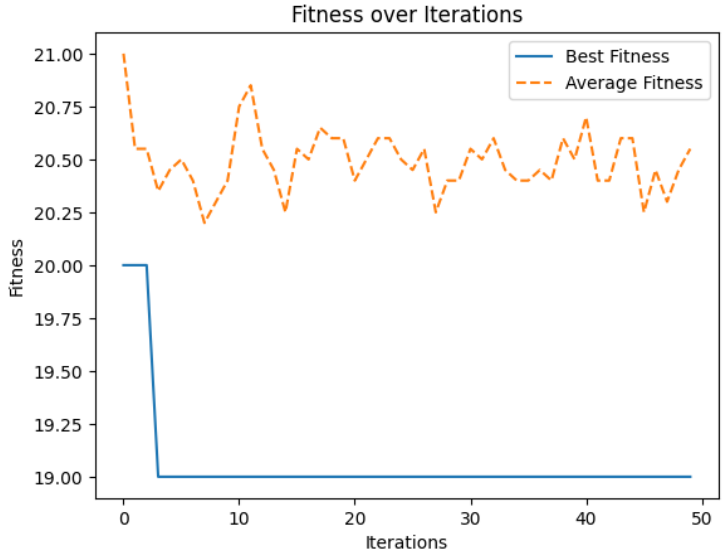

Figure 12: Instance 1 Result for "le450_15b.col"

1.5.7 Instance 2 - le450-15b.col

Iteration 0: Best fitness 20, Average fitness 21.3
Iteration 10: Best fitness 19, Average fitness 20.55
Iteration 20: Best fitness 19, Average fitness 20.45
Iteration 30: Best fitness 19, Average fitness 20.7
Iteration 40: Best fitness 19, Average fitness 20.4
Iteration 50: Best fitness 19, Average fitness 20.5
Iteration 60: Best fitness 19, Average fitness 20.5
Iteration 70: Best fitness 19, Average fitness 20.6
Iteration 80: Best fitness 19, Average fitness 20.45
Iteration 90: Best fitness 19, Average fitness 20.45
Best fitness: 19 AND Average fitness: 20.3 after 100 iterations.

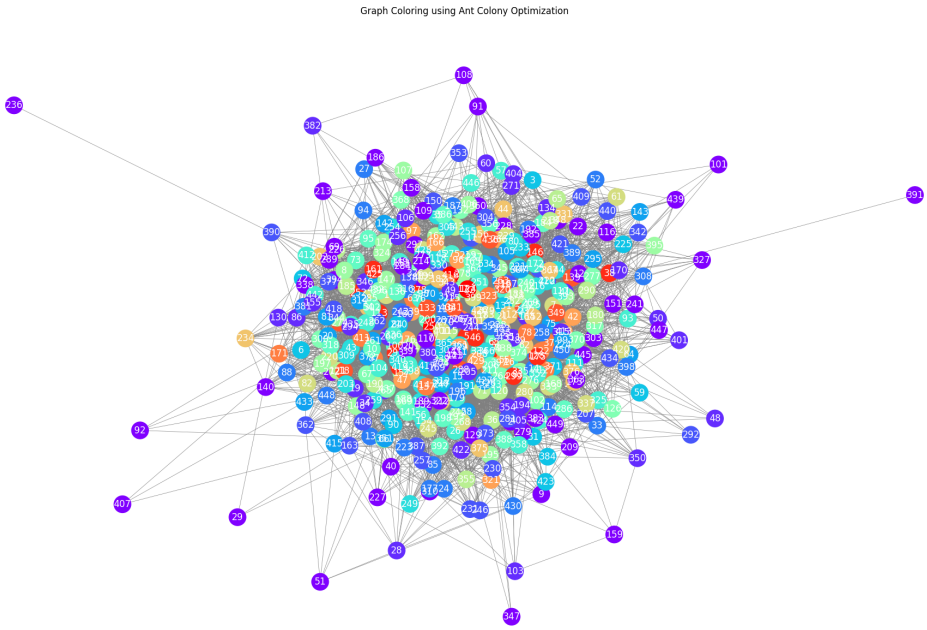

Figure 13: Instance 2 Graph for "le450_15b.col"

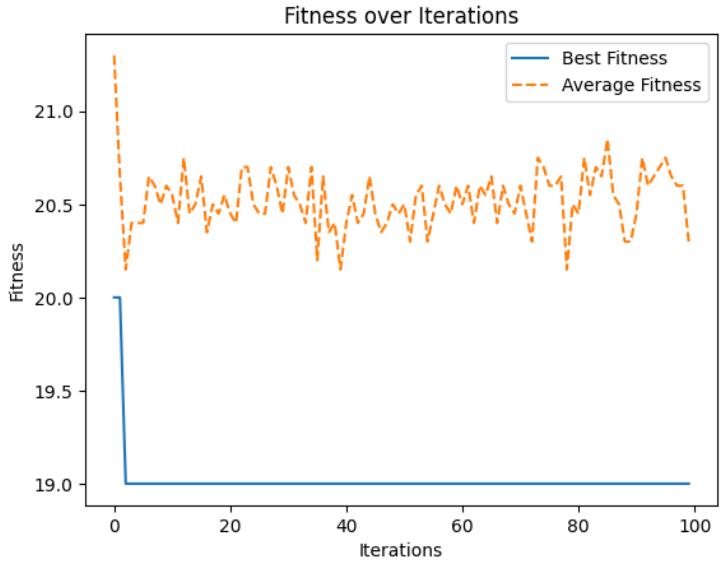

Figure 14: Instance 2 Result for "le450_15b.col"

1.5.8 Instance 3 - le450-15b.col

Iteration 0: Best fitness 20, Average fitness 20.75
Iteration 10: Best fitness 18, Average fitness 20.0
Iteration 20: Best fitness 18, Average fitness 19.7
Iteration 30: Best fitness 18, Average fitness 20.0
Iteration 40: Best fitness 18, Average fitness 20.1
Iteration 50: Best fitness 18, Average fitness 20.15
Iteration 60: Best fitness 18, Average fitness 20.15
Iteration 70: Best fitness 18, Average fitness 20.05
Iteration 80: Best fitness 18, Average fitness 20.1
Iteration 90: Best fitness 18, Average fitness 19.95
Best fitness: 18 AND Average fitness: 20.05 after 100 iterations.

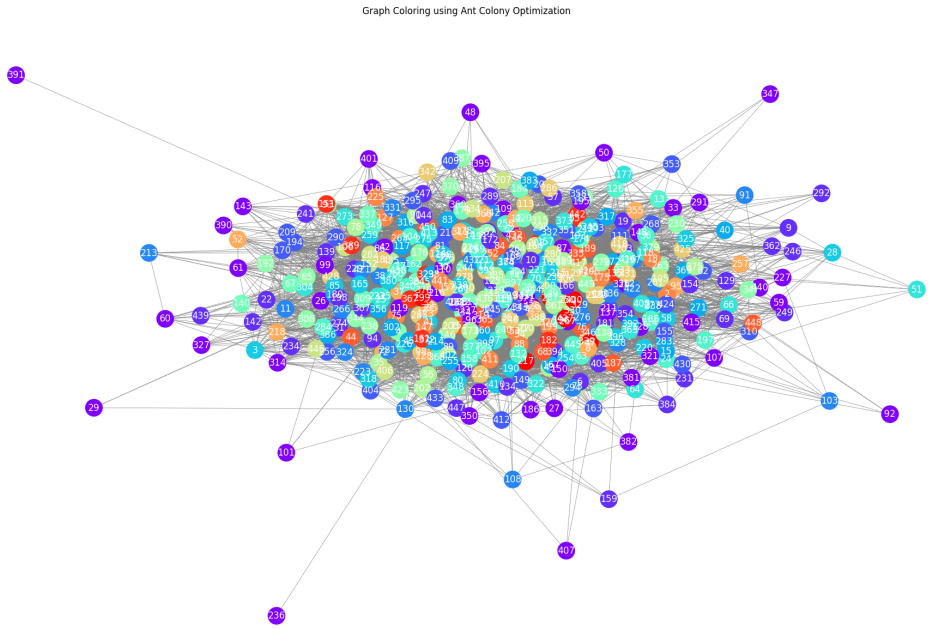

Figure 15: Instance 3 Graph for "le450_15b.col"

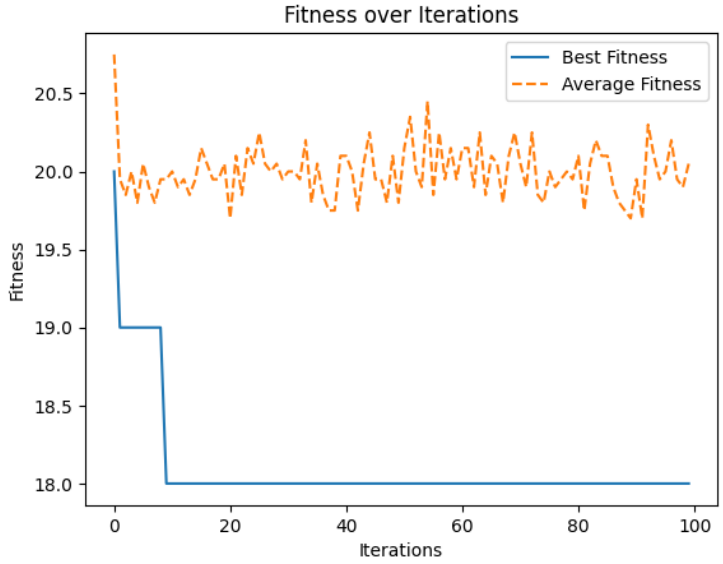

Figure 16: Instance 3 Result for "le450_15b.col"

1.5.9 Instance 4 - le450-15b.col

Iteration 0: Best fitness 20, Average fitness 20.933333333333334
Iteration 10: Best fitness 19, Average fitness 19.766666666666666
Iteration 20: Best fitness 18, Average fitness 19.966666666666665
Iteration 30: Best fitness 18, Average fitness 20.2
Iteration 40: Best fitness 18, Average fitness 19.8
Iteration 50: Best fitness 18, Average fitness 20.1
Iteration 60: Best fitness 18, Average fitness 20.133333333333333
Iteration 70: Best fitness 18, Average fitness 20.133333333333333
Iteration 80: Best fitness 18, Average fitness 19.966666666666665
Iteration 90: Best fitness 18, Average fitness 19.833333333333332
Best fitness: 18 AND Average fitness: 19.833333333333332 after 100 iterations.

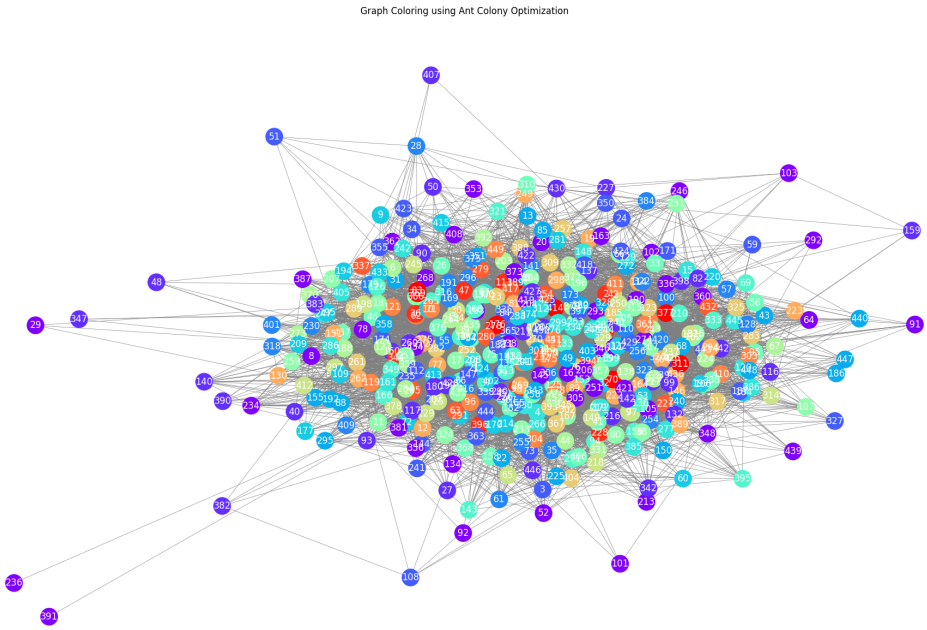

Figure 17: Instance 4 Graph for "le450_15b.col"

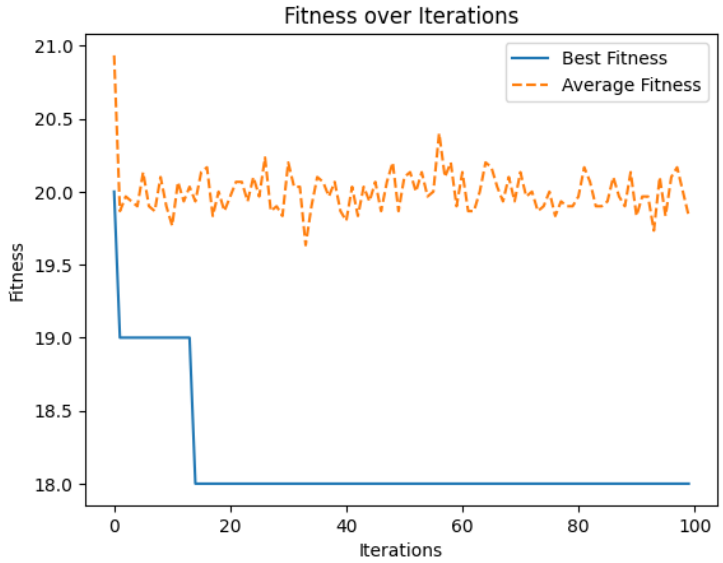

Figure 18: Instance 4 Result for "le450_15b.col"

1.5.10 Instance 5 - le450-15b.col

Iteration 0: Best fitness 20, Average fitness 20.833333333333332
Iteration 10: Best fitness 18, Average fitness 20.0
Iteration 20: Best fitness 18, Average fitness 19.8
Iteration 30: Best fitness 18, Average fitness 19.966666666666665
Iteration 40: Best fitness 18, Average fitness 20.2
Iteration 50: Best fitness 18, Average fitness 20.066666666666666
Iteration 60: Best fitness 18, Average fitness 20.266666666666666
Iteration 70: Best fitness 18, Average fitness 20.1
Iteration 80: Best fitness 18, Average fitness 19.933333333333334
Iteration 90: Best fitness 18, Average fitness 19.866666666666667
Iteration 100: Best fitness 18, Average fitness 20.166666666666668
Iteration 110: Best fitness 18, Average fitness 20.0
Iteration 120: Best fitness 18, Average fitness 19.833333333333332
Iteration 130: Best fitness 18, Average fitness 20.066666666666666
Iteration 140: Best fitness 18, Average fitness 19.9
Iteration 150: Best fitness 18, Average fitness 19.966666666666665
Iteration 160: Best fitness 18, Average fitness 19.966666666666665
Iteration 170: Best fitness 18, Average fitness 20.133333333333333
Iteration 180: Best fitness 18, Average fitness 19.933333333333334
Iteration 190: Best fitness 18, Average fitness 19.866666666666667

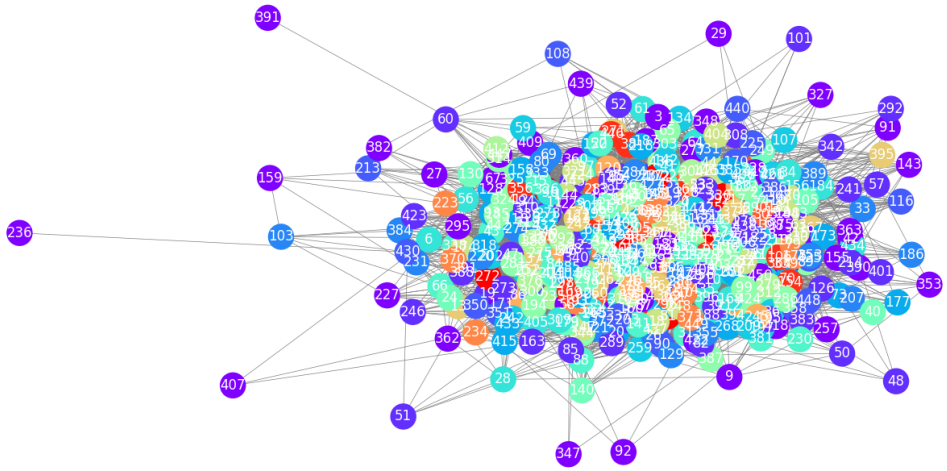

Figure 19: Instance 5 Graph for "le450_15b.col"

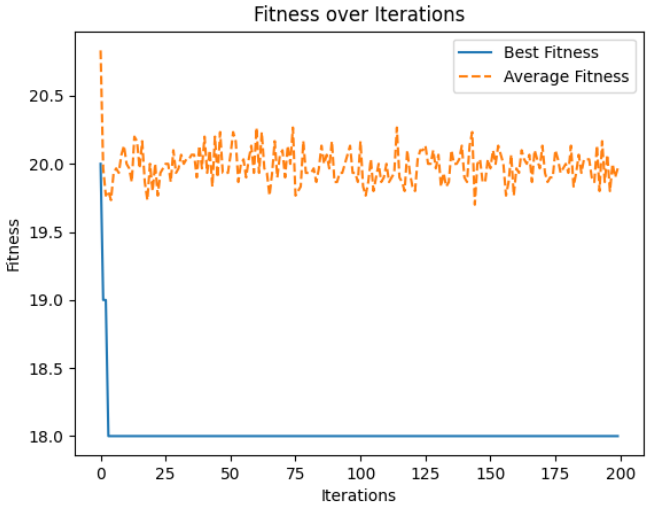

Figure 20: Instance 5 Result for "le450_15b.col"

1.6 Performance Evaluation And Conclusion

From the results obtained, it is evident that adjusting the parameters of the Ant Colony Optimization algorithm impacts the quality of the solutions for the vertex coloring problem. The initial parameter values of $\alpha = 0.8$ and $\beta = 0.8$ did not yield optimal results for both the "queen11_11.col" and "le450-15b.col" datasets. However, when we increased the heuristic influence (β) to 2, giving more weight to the heuristic function compared to the pheromone trail, the algorithm's performance improved considerably.

For the "queen11_11.col" dataset, the best solution found by our implementation utilized 14 colors, which is the optimal solution. This was achieved by increasing the number of iterations. The local search technique played a crucial role in refining the solutions found by the ants, leading to better quality results.

Furthermore, we observed that increasing the number of ants from 20 to 30 also contributed to finding better solutions for the "queen11_11.col" instance. This can be attributed to the fact that a larger number of ants explored a more diverse set of solutions, allowing the algorithm to effectively navigate the search space and converge to the optimal solution.

For the larger "le450-15b.col" dataset, our implementation achieved a solution using 18 colors, which is close to the optimal solution of 15 colors reported in the literature. The algorithm's performance on this instance was satisfactory with similar parameter settings as the "queen11_11.col" instance, demonstrating the robustness of the implementation across different problem sizes.

In addition to parameter tuning and local search techniques, it's crucial to acknowledge the role of stochasticity in the Ant Colony Optimization (ACO)

algorithm. Despite using the same parameters, the results may vary across different runs due to the inherent stochastic nature of the algorithm. This variability is influenced by factors such as the order in which solutions are constructed by ants, and the exploration of the search space. Thus, while certain parameter configurations may lead to improved performance on average, this may not always be the case on every run.

Overall, the results highlight the importance of parameter tuning and the incorporation of local search techniques in the Ant Colony Optimization algorithm for the vertex coloring problem. By carefully adjusting the balance between pheromone influence and heuristic influence, and leveraging local search strategies, our implementation was able to find high-quality solutions for both small and large problem instances.

2 Visualizing Swarms (Particle Systems)

We have developed a stimulation of a particle system which shows the fluid dynamics, integrating environmental effects such as wind, temperature, and obstacles. The simulation is visualized in a processing environment, offering interactive controls to adjust parameters and visualize the effects in real-time.

Link to the stimulation video: Click here to watch the video

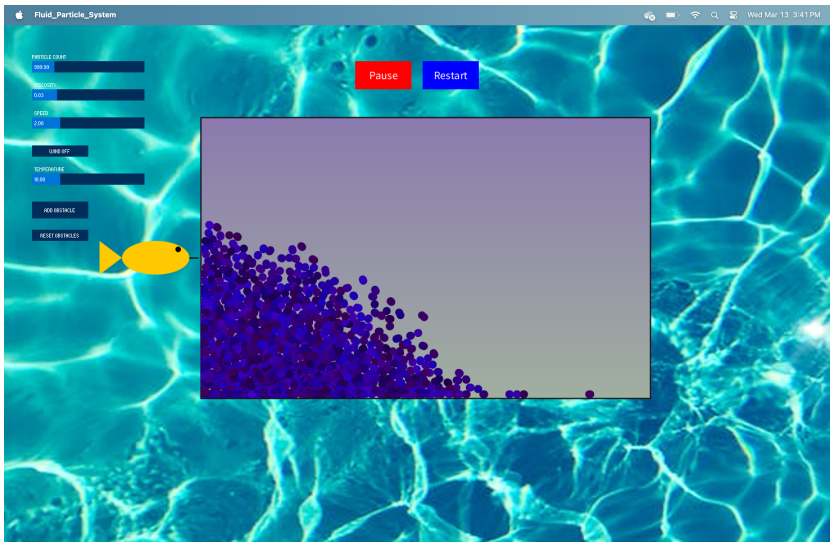

Figure 21: Fluid Particle System

2.1 Concept Behind the Simulation

Particle System Dynamics

Each particle represents a small element of the fluid, having properties like

position, velocity, color and lifespan. Particles move within a defined space, influenced by forces of gravity, wind, speed, and interactions with obstacles. Viscosity is simulated through damping the particle velocity, representing the energy loss due to internal friction and resistance encountered in real fluids. Temperature affects the color of the particles, visually representing thermal effects on the fluid. It could also affect the particle's lifespan, simulating how temperature impacts fluid properties.

Environmental and Interactive Effects: Wind applies a directional force to the particles, simulating external influences on fluid movement. Temperature changes not only affect the visual representation (color) but could also alter dynamics like particle speed, viscosity, or interactions. Obstacles introduce complexity to fluid flow, forcing particles to navigate around them, simulating real-world interactions between fluids and solid objects.

User Interaction:
Users can interactively adjust particle count, viscosity, speed, and temperature to observe how each parameter influences the fluid's behavior. The ability to toggle wind and add obstacles allows users to experiment with external influences on the fluid simulation.

2.2 Problem Formulation

Objective: To create a dynamic particle system that simulates fluid-like behavior, incorporating environmental factors and user interactions to manipulate the fluid flow and observe changes in real-time.

Inputs:

- Particle count: Number of particles in the system.
- Viscosity: A damping factor that simulates the resistance encountered by particles. The lower the viscosity, the more easily the fluid flows.
- Speed: A multiplier affecting particle velocity.
- Wind: A vector force that simulates wind acting on the particles. If applied, it works for 5 seconds only.
- Temperature: Influences particle color and affects the lifespan of particle so the higher the temperature the lower the lifespan of the particle.
- Obstacles: Barriers that particles interact with, altering their trajectory. Fixed sized obstacles can be added by the user.

Controls: The stimulation has sliders to adjust particle count, fluidity, speed, and temperature. There are buttons to toggle wind, add obstacles, and reset the simulation.

Visualization: The stimulation shows fluid particles flowing out of a fish mouth into a container. The particles are contained in a container where they can move and interact according to the environmental factors. The particles change color based on the temperature, offering a visual cue of the system's state and the temperature has an effect on the particle's lifespan so they begin to evaporate quickly when temperature is higher. There are obstacles that particles can collide with and bounce off as well. The wind flows only in the left to right direction and it's affect on the particles is for 5 seconds only.