VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sudarshan Komar (1BM22CS291)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019

Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Sudarshan Komar (1BM22CS291),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Sheetal V A Assistant Professor Department of CSE, BMSCE	**Dr. Kavitha Sooda** Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/SudarshanKomar/bis_lab

Program 1: Genetic Algorithm

Date 3,10,24
Page

lab-1

* Genetic algorithms (GA)

GAs are a type of optimization and search technique inspired by the process of natural selection and genetics. They are part of broader class of algorithms known as evolutionary algorithms. GAs operate by simulating the process of evolution, using techniques such as selection, crossover (recombination) and mutation to evolve solutions to problems over successive generations.

→ Key concepts:

- Population: A set of candidate solutions to the problem
- Chromosome: A representation of a solution, often encoded as a string of bits or number
- Fitness function: A function that evaluates how good a solution is at solving the problem
- Selection: The process of choosing the fittest individuals to reproduce.
- Crossover: Combining parts of two parent solutions to create new offspring.
- Mutation: Randomly altering parts of a solution to maintain genetic diversity

→ Application of GAs

1 Optimization problems

GAs are widely used to find optimal solutions in complex spaces such as

scheduling

Resource allocation

Route planning

2 Machine learning

GAs can optimize hyperparameters, select features, improve model performance

3 Engineering design:

Used for optimize design parameters in fields like structural engineering, aerospace and automotive design.

4 Financial modeling:

GAs can help in optimizing investment portfolios and predicting marketing trends..

5 Game Development:

Used to evolve strategies for non-player characters (NPCs) and optimize game machines.

Date: / /
Page

★ Traveling salesmen problem (TSP)

Pseudocode for TSP

```
Function calculate_fitness(tour, distance_matrix):
    total_distance = 0
    for i from 0 to length(tour)-1:
        total_distance += distance_matrix[
            tour[i]][tour[(i+1) mod length(tour)]]
    Return total distance

Function tournament_selection(population,
        fitness_scores, tournament_size):
    selected = Random_sample(population,
        fitness_scores, tournament_size)
    selected = sort(selected by fitness score)
    Return selected[0] //Best individual

Function order_crossover(parent1, parent2):
    size = length(parent1)
    start, end = Randomly selected two indices
    child = Array of size size filled with -1
    copy segment from parent1[start]
        to parent1[end] into child[start]
        to child[end]
    current_pos = (end+1) mod size
    For each city in parent2:
        If city not in child:
            child[current_pos] = city
            current_pos = (current_pos + 1) mod size
    Return child.
```

Date __/__/__
Page____

function swap-mutation (tour, mutation-ra
  If Random() < mutation-ratio:
    idx1, idx2 = Randomly select two indices from tour
    swap tour[idx1] and tour[idx2]

output:
i/p parameters.
coords = np.random.rand(num-cities, 2) * 100
{ city coordinates
  city 0 : (23.45, 67.89)
  city 1 : (12.34, 45.67)
  city 2 : (78.90, 12.34)
  city 3 : (56.78, 90.12)
  city 4 : (34.56, 23.45) }

o/p values;

Best Tour : [0, 4, 1, 3, 2]
Best distance: 215.67

geneticalg

November 20, 2024

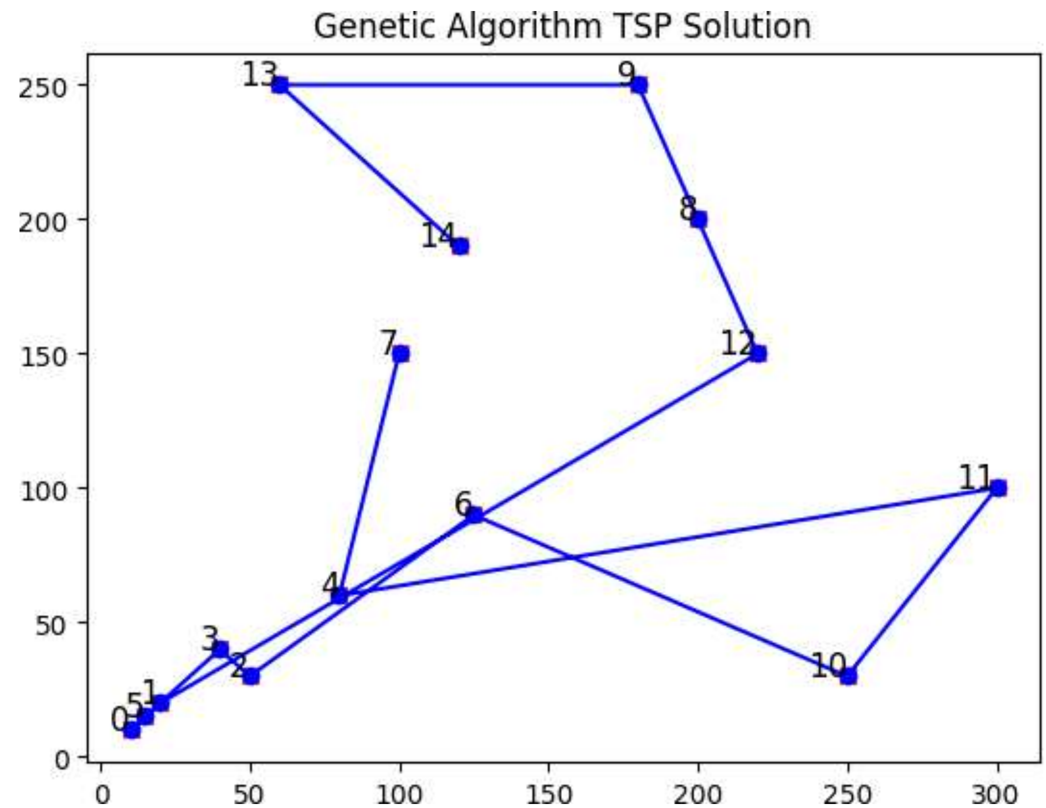
[3]:
```
#Genetic Algorithm to solve traveling salesman problem
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

import numpy as np import
random
import matplotlib.pyplot as plt

# Define cities as coordinates
def define_cities():
    return np.array([
        [10, 10], [20, 20], [50, 30], [40, 40], [80, 60],
        [15, 15], [125, 90], [100, 150], [200, 200], [180, 250],
        [250, 30], [300, 100], [220, 150], [60, 250], [120, 190]
    ])

# Compute the distance between two cities
def compute_distance(city1, city2):
    return np.linalg.norm(city1 - city2)

# Create the distance matrix for all cities
def compute_distance_matrix(cities): num_cities =
    len(cities)
    dist_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(num_cities):
            dist_matrix[i, j] = compute_distance(cities[i], cities[j])
    return dist_matrix

# Calculate the total length of a given tour
def calculate_tour_length(tour, dist_matrix): length = 0
    for i in range(len(tour) - 1):
        length += dist_matrix[tour[i], tour[i + 1]]
    length += dist_matrix[tour[-1], tour[0]]  # Return to starting city
    return length

# Generate the initial population of tours
```

```
def initialize_population(num_individuals, num_cities): population =
    [np.random.permutation(num_cities) for _ in
range(num_individuals)]
    return population

# Perform tournament selection to choose parents
def tournament_selection(population, fitness, tournament_size=3):
    selected = np.random.choice(len(population), tournament_size, replace=False) best_idx =
    selected[np.argmin([fitness[i] for i in selected])]
    return population[best_idx]

# Perform ordered crossover
def crossover(parent1, parent2): size =
    len(parent1)
    start, end = sorted(random.sample(range(size), 2)) child = [-1] *
    size
    child[start:end] = parent1[start:end]

    idx = end
    for gene in parent2:
        if gene not in child:
            if idx >= size: idx = 0
            child[idx] = gene idx +=
            1
    return child

# Perform mutation by swapping two cities in the tour
def mutate(tour, mutation_rate):
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(tour)), 2) tour[i], tour[j] =
        tour[j], tour[i]

# Genetic Algorithm
def genetic_algorithm(cities, num_individuals, num_generations, mutation_rate): dist_matrix =
    compute_distance_matrix(cities)
    num_cities = len(cities)
    population = initialize_population(num_individuals, num_cities)

    best_tour = None
    best_length = float('inf')

    for generation in range(num_generations):
        fitness = [calculate_tour_length(tour, dist_matrix) for tour in
population]
        new_population = []
```

```
            for _ in range(num_individuals // 2):
                  parent1 = tournament_selection(population, fitness) parent2 =
                  tournament_selection(population,      fitness)      child1      =
                  crossover(parent1, parent2)
                  child2 = crossover(parent2, parent1) mutate(child1,
                  mutation_rate) mutate(child2, mutation_rate)
                  new_population.extend([child1, child2])

            population = new_population current_best_idx

            =np.argmin(fitness)
            current_best_length = fitness[current_best_idx]

            if current_best_length < best_length: best_length =
                  current_best_length best_tour=
                  population[current_best_idx]

      return best_tour, best_length

# Visualize the best tour
def plot_tour(cities, best_tour): tour_cities=
      cities[best_tour]
      plt.plot(tour_cities[:, 0], tour_cities[:, 1], 'bo-', markersize=6)
      plt.scatter(cities[:, 0], cities[:, 1], color='red', marker='x')
      for i, city in enumerate(cities):
            plt.text(city[0], city[1], f'{i}', fontsize=12, ha='right') plt.title("Genetic Algorithm TSP Solution")
      plt.show()

# Main Execution
if __name____ == "__main__": cities
      =define_cities()
      num_individuals = 50
      num_generations = 200
      mutation_rate = 0.1

      best_tour, best_length = genetic_algorithm(cities, num_individuals,

  num_generations, mutation_rate)
      print("Tour Order:", best_tour)
      print(f"Best Tour Length: {best_length:.2f}")

      plot_tour(cities, best_tour)
```

Name:Sudarshan Komar
USN:1BM22CS291

Tour Order: [14, 13, 9, 8, 12, 1, 0, 5, 3, 2, 6, 10, 11, 4, 7]
Best Tour Length: 996.55



Program 2 : Particle Swarm Optimization Algorithm

Date: 24.10.2024
Page

Lab-2

* Particle Swarm Optimization (PSO)

→ Algorithm

Parameters of problem

* Number of dimensions (d)
* Lower bound (minx)
* Upper bound (maxx)

Hyperparameters of the algorithm:

* Number of particles (N)
* maxm no. of iterations (max-iter)
* Inertia (w)
* Cognition of particle (c1)
* Social influence of swarm (c2)

Step 1: Randomly initilize swarm population of N particles X_i (i=1 ... n)

Step 2: Select hyperparameter values w, c1 and c2

Step 3: For Iter in range(max-iter):
  For i in range(N):
    a. Compute new velocity of ith particle
       swarm[i].velocity =
         w * swarm[i].velocity +
         r1 * c1 * (swarm[i].bestPos -
           swarm[i].position) +
         r2 * c2 * (best-pos-swarm -
           swarm[i].position)
    b. compute new position of ith particle using its new velocity
       swarm[i].position += swarm[i].velocity
    c. If position is not in range [minx, maxx] then clip it

Date / /
Page

if swarm[i].position < minx:
 swarm[i].position = minx
elif swarm[i].position > maxx:
 swarm[i].position = maxx

d. Update new best of this particle and new best of swarm:

if swarm[i].fitness < swarm[i].bestFitness:
 swarm[i].bestFitness = swarm[i].fitness
 swarm[i].bestPos = swarm[i].position

if swarm[i].fitness < best_fitness_swarm
 best_fitness_swarm = swarm[i].fitness
 best_pos_swarm = swarm[i].position

End-for
End-for

Step4: Return best particle of swarm

Output:

i/p parameters
num_particles → 30
dimension → 2
max_iteration → 1000
c1 → 1.5
c2 → 1.5
objective function $f(x) = \sum_{i=1}^{d} x_i^2 \Rightarrow x_1^2 + x_2^2$

o/p
best position (−0.0023, 0.0015)
best value (0.0005) ≈ (0)

particleswarmoptm

November 20, 2024

[ ]:

```
#particle swarm optimization algorithm to minimize objective fn
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

import numpy as np

# Objective function (e.g., Sphere function)
def objective_function(x):
    return sum(xi**2 for xi in x)

class Particle:
    def __init__(self, dim):
        self.position = np.random.rand(dim) * 10 - 5  # Random position in
↪range [-5, 5]
        self.velocity = np.random.rand(dim) * 2 - 1                    # Random velocity
        self.best_position = self.position.copy() self.best_value =
        objective_function(self.position)

def pso(num_particles, dimensions, max_iterations): w = 0.5  #
    Inertia weight
    c1 = 1.5  # Cognitive coefficient
    c2 = 1.5  # Social coefficient

    # Initialize particles
    particles = [Particle(dimensions) for _ in range(num_particles)] global_best_position =
    particles[0].best_position.copy() global_best_value = particles[0].best_value

    for t in range(max_iterations):
        for particle in particles:
            # Update velocity
            r1, r2 = np.random.rand(dimensions), np.random.rand(dimensions) particle.velocity = (w *
            particle.velocity +
                                     c1 * r1 * (particle.best_position - particle.
↪position) +
                                     c2 * r2 * (global_best_position - particle.
↪position))
```

```
            # Update position
            particle.position += particle.velocity

            # Evaluate fitness
            value = objective_function(particle.position)

            # Update personal best
            if value < particle.best_value:
                particle.best_value = value
                particle.best_position = particle.position.copy()

            # Update global best
            if value < global_best_value:
                global_best_value = value
                global_best_position = particle.position.copy()

    # Print only the final best result
    print(f"Best Position: {global_best_position}, Best Value:
s{global_best_value}")
    return global_best_position, global_best_value

# Parameters
num_particles = 30
dimensions = 2
max_iterations = 1000

best_position, best_value = pso(num_particles, dimensions, max_iterations)
```

Name:Sudarshan Komar
USN:1BM22CS291
Best Position: [5.92457810e-110 3.05564784e-109], Best Value:
9.687989953612073e-218

Program 3: Ant Colony Optimization Algorithm

Lab-3

Date / /
Page

* Ant colony optimization.

Algorithm.

initilize pheromone values $\forall i, j \in [1, n] : \tau_{ij} \mapsto \tau_0$

repeat

for each ant $l \in \{1, \dots m\}$ do

initlize selection set $S \mapsto \{1, \dots n\}$

randomly choose starting city $i_0 \in S$ for ant l

move to starting city $i \mapsto i_0$

while $S \neq \phi$ do

remove current city from selection set $S \mapsto S \setminus \{i\}$

choose next city in the tour with probability $p_{ij} = \dfrac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{h \in S} \tau_{ih}^{\alpha} \cdot \eta_{ih}^{\beta}}$

update solution vector $\pi_l(i) \mapsto j$

move to new city $i \mapsto j$

end while

finalize solution vector $\pi_l(i) \mapsto i_0$

end for

for each solution π_l, $l \in \{1, \dots m\}$ do

calculate tour length $f(\pi_l) \mapsto \sum_{i=1} d_{i \pi_l(i)}$

end for

for all (i, j) ~~do~~

evaporate pheromone $\tau_{ij} \mapsto (1-\rho) \cdot \tau_{ij}$

end for

determine best soln for iteration $\pi^{+} = \arg\min_{l \in [1, m]} f(\pi_l)$

if π^+ better than current best π^* ie
$f(\pi^+) < f(\pi^*)$ then
set $\pi^* \rightarrow \pi^+$
end if

for all $(i,j) \in \pi^+$ do
reinforce $\tau_{ij} \rightarrow \tau_{ij} + \Delta/2$
end for

for all $(i,j) \in \pi^*$ do
reinforce $\tau_{ij} \mapsto \tau_{ij} + \Delta/2$
end for
Until condition for termination met

⇒ Output
i/p values
no. of ants = 50
Alpha (influence of pheromone) = 1
Beta (influence of distance) = 2
Rho (pheromone evaporation rate) = 0.1
Max iterations = 100
City coordinates

o/p:
Best tour length: ~~[illegible]~~ 985.25

antcoloptm

November 20, 2024

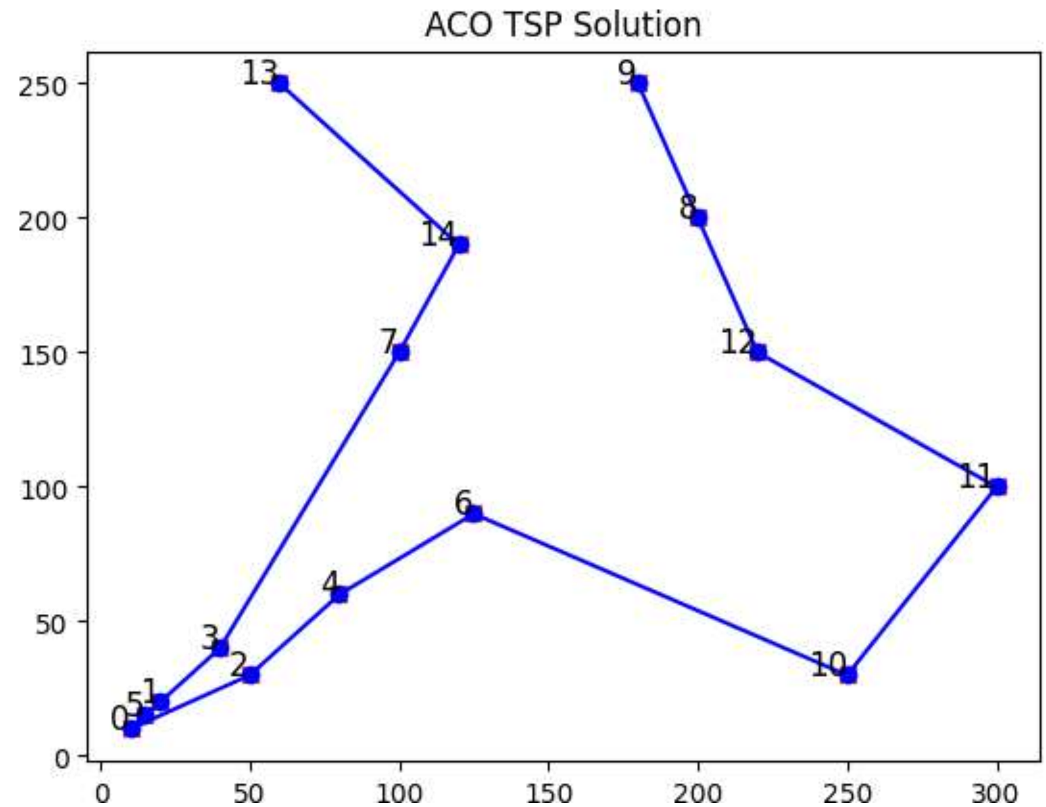
[ ]:
```
#Ant Colony Optimization algorithm to solve the Traveling Salesman Problem
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

import numpy as np
import random
import matplotlib.pyplot as plt

NUM_ANTS = 50
ALPHA = 1.0  # Influence of pheromone
BETA = 2.0        # Influence of distance
RHO = 0.1         # Pheromone evaporation rate
Q = 100             # Pheromone deposit constant
MAX_ITER = 100  # Maximum number of iterations


def define_cities():
      return np.array([ [10,
            10],
            [20, 20],
            [50, 30],
            [40, 40],
            [80, 60],
            [15, 15],
            [125, 90],
            [100, 150],
            [200, 200],
            [180, 250],
            [250, 30],
            [300, 100],
            [220, 150],
            [60, 250],
            [120, 190],
      ])

# Compute the distance matrix
def compute_distance_matrix(cities): num_cities =
      len(cities)
```

```
    distance_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            dist = np.linalg.norm(cities[i] - cities[j]) distance_matrix[i, j]
            = dist distance_matrix[j, i] = dist
    return distance_matrix

# Initialize pheromone matrix
def initialize_pheromone_matrix(num_cities):
    pheromone_matrix = np.ones((num_cities, num_cities))  # Pheromone starts as
1 for all edges
    np.fill_diagonal(pheromone_matrix, 0)  # No pheromone on the diagonal
(self-loops)
    return pheromone_matrix

# Calculate the total length of a tour
def calculate_tour_length(tour, dist_matrix): length = 0
    for i in range(len(tour) - 1):
        length += dist_matrix[tour[i], tour[i + 1]]
    length += dist_matrix[tour[-1], tour[0]]  # Returning to the start
    return length

# Ant solution construction (probabilistic decision on next city)
def construct_solution(num_cities, pheromone_matrix, dist_matrix):
    tour = [random.randint(0, num_cities - 1)]  # Start from a random city
    visited = set(tour)

    while len(tour) < num_cities: current_city =
        tour[-1] probabilities = []
        for next_city in range(num_cities):
            if next_city not in visited:
                pheromone = pheromone_matrix[current_city, next_city] ** ALPHA distance =
                (1.0 / dist_matrix[current_city, next_city]) ** BETA
                probabilities.append(pheromone * distance)
            else:
                probabilities.append(0)

        total_prob = sum(probabilities)
        probabilities = [p / total_prob for p in probabilities]

        # Choose the next city based on the probabilities
        next_city = np.random.choice(range(num_cities), p=probabilities) tour.append(next_city)
        visited.add(next_city)
```

```
    return tour

# Update the pheromone matrix based on the solutions found by ants
def update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour):
    # Evaporate pheromone
    pheromone_matrix *= (1 - RHO)

    # Add pheromone for all ants
    for tour in all_tours:
        tour_length = calculate_tour_length(tour, dist_matrix)
        for i in range(len(tour) - 1):
            pheromone_matrix[tour[i], tour[i + 1]] += Q / tour_length pheromone_matrix[tour[-
        1], tour[0]] += Q / calculate_tour_length(tour,

 dist_matrix)
    # Add pheromone for the best tour
    best_length = calculate_tour_length(best_tour, dist_matrix)
    for i in range(len(best_tour) - 1):
        pheromone_matrix[best_tour[i], best_tour[i + 1]] += Q / best_length
    pheromone_matrix[best_tour[-1], best_tour[0]] += Q / best_length

# Main ACO algorithm for solving TSP
def ant_colony_optimization(cities, dist_matrix, pheromone_matrix, max_iter): best_tour = None
    best_tour_length = float('inf')

    # Main loop
    for iteration in range(max_iter): all_tours = []

        # Step 1: All ants construct their solutions
        for _ in range(NUM_ANTS):
            tour = construct_solution(len(cities), pheromone_matrix,

 dist_matrix)
            all_tours.append(tour)
            tour_length = calculate_tour_length(tour, dist_matrix)

            # Step 2: Update the best tour if necessary
            if tour_length < best_tour_length: best_tour =
                tour best_tour_length = tour_length

        # Step 3: Update pheromone matrix
        update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour)

    return best_tour, best_tour_length
```

```
# Visualize the tour
def plot_tour(cities, best_tour): tour_cities =
        cities[best_tour]
        plt.plot(tour_cities[:, 0], tour_cities[:, 1], 'bo-', markersize=6)
        plt.scatter(cities[:, 0], cities[:, 1], color='red', marker='x')
        for i, city in enumerate(cities):
                plt.text(city[0], city[1], f'{i}', fontsize=12, ha='right') plt.title("ACO TSP Solution")
        plt.show()

# Main Execution
if __name__ == "__main__":

        cities = define_cities()
        dist_matrix = compute_distance_matrix(cities) pheromone_matrix =

        initialize_pheromone_matrix(len(cities))

        best_tour, best_tour_length = ant_colony_optimization(cities, dist_matrix,

    pheromone_matrix,  MAX_ITER)
        print(f"Best tour length: {best_tour_length:.2f}") plot_tour(cities,
        best_tour)
```

Name:Sudarshan Komar
USN:1BM22CS291
Best tour length: 985.25


ACO TSP Solution
13
9
14
8
7
12
11
6
4
3
2
10
1
5
0
250
200
150
100
50
0
0
50
100
150
200
250
300

Program 4: Cuckoo Search Algorithm

Date 14/11/2024
Page

Lab-4

* Cuckoo search Algorithm

algorithm CuckooSearch():
// INPUT
// n = initial population size
// Pa = fraction of worse nests to be abandoned and replaced
// Max iterations = the maxm no. of iterations
// f = the objective function to optimize()
// OUTPUT
// The best solution found.

Generate the initial populations of n host nests X_i (i = 1, 2 - - - n)
While t < Max iterations:
Get a cuckoo randomly by Levy flights
Evaluate its quality fitness F_i

j ← choose a nest among n randomly

if $F_i > F_j$:
Replace j by new solution

if A fraction (Pa) of worse nests are abandoned and new ones are built:
Keep the best solutions
Rank the solutions and find the current best

postprocess result and visualization

CSA for traffic signal optimization

Outputs.

i/p

dim (no. of intersections) → 3

bounds (Range of greenlight time) → [10,12]

num_nests → 20

max_iter → 100

Pa (Probability of abandonment) → 0.25

Lambda (levy flight parameter) → 1.5

o/p

best_soln (optimal greenlight time for each intersection

best_fitness (total wait time)

best_soln → [90.5, 110.3, 75.8]

best_fitness → 325.5

o/p seen (GH)

cuckoosearch

November 20, 2024

[10]:

```
#cuckoo Search algorithm to optimize green light timings at a traffic
↪intersection to minimize total waiting time
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

import numpy as np
from scipy.special import gamma

def fitness_function(x):
    waiting_times = np.array([10 + (x[i] ** 1.5) / 50 for i in range(len(x))]) total_waiting_time =
    np.sum(waiting_times)
    return total_waiting_time

def levy_flight(dim, beta=1.5):
    sigma_u = np.power((gamma(1 + beta) * np.sin(np.pi * beta / 2) /
                        gamma((1 + beta) / 2) * beta * (2 ** (beta - 1))), 1 /
↪beta)
    u = np.random.normal(0, sigma_u, dim)
    v = np.random.normal(0, 1, dim)
    step = u / np.power(np.abs(v), 1 / beta)
    return step

def cuckoo_search(dim, bounds, num_nests, max_iter, p_a=0.1, Lambda=1.5): nests =
    np.random.uniform(bounds[0], bounds[1], (num_nests, dim)) fitness =
    np.array([fitness_function(nest) for nest in nests])

    best_idx = np.argmin(fitness) best_nest =
    nests[best_idx] best_fitness = fitness[best_idx]

    for iter in range(max_iter): new_nests
        = np.copy(nests) for i in
        range(num_nests):
            step = levy_flight(dim, Lambda) * 0.1 new_nests[i] =
            nests[i] + step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])

        new_fitness = np.array([fitness_function(nest) for nest in new_nests])
```

```
        for i in range(num_nests):
            if new_fitness[i] < fitness[i]: nests[i] =
                new_nests[i] fitness[i] =
                new_fitness[i]

        if np.random.rand() < p_a:
            random_idx = np.random.randint(num_nests)
            nests[random_idx] = np.random.uniform(bounds[0], bounds[1], dim)
            fitness[random_idx] = fitness_function(nests[random_idx])

        current_best_idx = np.argmin(fitness) current_best_fitness =
        fitness[current_best_idx]

        if current_best_fitness < best_fitness: best_fitness
            =      current_best_fitness      best_nest      =
            nests[current_best_idx]

    return best_nest, best_fitness dim = 3
bounds = [10, 120]
num_nests = 20
max_iter = 100

best_solution, best_value = cuckoo_search(dim, bounds, num_nests, max_iter) print("Green Light

Timings (seconds):", best_solution)
print("Best Fitness Value (Total Waiting Time):", best_value)
```

Name:Sudarshan Komar
USN:1BM22CS291
Green Light Timings (seconds): [41.91091846 17.73463375 12.24440876] Best Fitness Value (Total Waiting Time): 37.77712475920917

Program 5: Grey Wolf Optimizer Algorithm

Date 21/11/2024
Page

I^st Lab-5

Gray wolf optimization

$\alpha \rightarrow$ best-fitness
$\beta \rightarrow$ second best-fitness
$\delta \rightarrow$ third best-fitness
$\omega \rightarrow$ Remaining fitness

$X_i = X_i + A \cdot D$

Initilize population of wolves with random position
Initilize alpha, beta, delta position and corresponding fitness value

for each iter from 1 to max
    For each wolf i in population
        if ~~fitness~~(i) > fitness(α)
                update α position
        else if fitness(i) > fitness(β)
                update β position
        else if fitness(i) > fitness(δ)
                update δ position

    For each wolf i in population
        calculate α, β, δ distances
        update position of word using
            $X_i = X_i + A \times D(\alpha), A \times D(\beta), A \times D(\delta)$
        if position is outside bounds
            Adjust position to be within bounds

return best solution found (α position & fitness)

Imp

Application: water distribution optimization

Input:

No of wolves: 30

max itr: 100

No of nodes: 5

demand at each node:

[10, 40, 30, 60, 70]

cost factor for pipes at each node

[1, 2, 1.5, 3, 2]

pump cost at each nodes

[0.1, 0.1, 0.1, 0.1, 0.1]

output

Best pipe size: [0.703, 0.734, 0.586, 0.482, 1.89]

Best flow rate: [0.884, 1.045, 1.117, 0.810, 1.385]

21/11/24

graywolfoptm-1

November 28, 2024

[6]:
```
#gray wolf optimization for water flow optimization
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

import numpy as np
import matplotlib.pyplot as plt


def objective_function(pipe_sizes, flow_rates, demand, node_pressure,
↪pipe_costs, pump_costs):
    # Pipe costs (cost proportional to diameter^2)
    pipe_cost = np.sum(pipe_sizes ** 2 * pipe_costs)  # cost related to the
↪pipe diameters

    # Pumping costs (assume pump power is proportional to flow rate)
    pump_cost = np.sum(flow_rates * pump_costs)  # cost related to pumping

    # Pressure constraints: penalize if pressure falls below threshold (say 20m)
    pressure_penalty = 0
    for i in range(len(node_pressure)):
        if node_pressure[i] < 20:
            pressure_penalty += (20 - node_pressure[i]) ** 2  # Penalize low
↪pressure

    # Demand satisfaction: penalize if flow at any node does not meet demand
    demand_penalty = 0
    for i in range(len(demand)):
        if flow_rates[i] < demand[i]:
            demand_penalty += (demand[i] - flow_rates[i]) ** 2  # Penalize
↪under-supply

    # Total objective: cost + penalties for pressure and demand violations
    total_cost = pipe_cost + pump_cost + pressure_penalty + demand_penalty
    return total_cost

# Define the Grey Wolf Optimization (GWO) class
class GreyWolfOptimization:
```

```
    def __init__(self, num_wolves,          max_iter, demand, pipe_costs,          pump_costs,
num_nodes):
        self.num_wolves = num_wolves
        self.max_iter = max_iter self.demand =
        demand self.pipe_costs = pipe_costs
        self.pump_costs = pump_costs
        self.num_nodes = num_nodes

        self.wolves = np.random.rand(self.num_wolves, 2 * self.num_nodes) self.alpha = None
        self.beta = None
        self.delta = None self.alpha_score =
        float('inf') self.beta_score = float('inf')
        self.delta_score = float('inf')

    def fitness(self, wolf):
        # Split wolf's position into pipe sizes and flow rates
        pipe_sizes = wolf[:self.num_nodes]  # First half of wolf is for pipe
sizes
        flow_rates = wolf[self.num_nodes:]  # Second half of wolf is for flow
rates
        # Initialize pressure array (just as an example, in practice you would
calculate this based on network model)
        node_pressure = np.random.rand(self.num_nodes) * 50  # Random pressure
values for each node (example)

        # Call the objective function to calculate cost
        return objective_function(pipe_sizes, flow_rates, self.demand,
node_pressure, self.pipe_costs, self.pump_costs)

    def update_positions(self):
        for i in range(self.num_wolves): A = 2 *
            np.random.rand(1) - 1 C = 2 *
            np.random.rand(1)
            D_alpha = np.abs(C * self.alpha - self.wolves[i]) X1 = self.alpha - A
            * D_alpha

            A = 2 * np.random.rand(1) - 1 C = 2 *
            np.random.rand(1)
            D_beta = np.abs(C * self.beta - self.wolves[i]) X2 = self.beta - A
            * D_beta
```

```
                A = 2 * np.random.rand(1) - 1 C = 2 *
                np.random.rand(1)
                D_delta = np.abs(C * self.delta - self.wolves[i])
                X3 = self.delta - A * D_delta

                # Update the wolf's position
                self.wolves[i] = (X1 + X2 + X3) / 3

    def optimize(self):
        for _ in range(self.max_iter):
            for i in range(self.num_wolves):
                fitness_value = self.fitness(self.wolves[i])

                # Update alpha, beta, and delta wolves based on fitness values
                if    fitness_value    <    self.alpha_score:
                    self.alpha_score    =    fitness_value
                    self.alpha = self.wolves[i]

                elif fitness_value < self.beta_score: self.beta_score =
                    fitness_value self.beta = self.wolves[i]

                elif fitness_value < self.delta_score: self.delta_score =
                    fitness_value self.delta = self.wolves[i]

            # Update positions of all wolves
            self.update_positions()

        return self.alpha  # Return the best solution

num_wolves = 30
max_iter = 1000
num_nodes = 6
demand = np.array([50, 100, 80, 150, 120, 180])
pipe_costs = np.array([1.0, 3.5, 1.3, 1.8, 2.0, 1.7])
pump_costs = np.array([0.2, 0.2, 0.18, 0.25, 0.2, 0.2])

# Initialize and run the Grey Wolf Optimization
gwo = GreyWolfOptimization(num_wolves, max_iter, demand, pipe_costs,
  pump_costs, num_nodes) best_solution =
gwo.optimize()

# Extract best solution: pipe sizes and flow rates best_pipe_sizes =
best_solution[:num_nodes] best_flow_rates = best_solution[num_nodes:]
```

```
print("Best Pipe Sizes:", best_pipe_sizes) print("Best
Flow Rates:", best_flow_rates)
```

Name:Sudarshan Komar
USN:1BM22CS291
Best Pipe Sizes: [0.00091585 0.00095113 0.00084839 0.00140677 0.00067415
0.00074783]
Best Flow Rates: [0.00055832 0.00113251 0.00048501 0.00093122 0.00049868
0.00107437]

Program 6: Parallel Cellular Algorithms and Programs

Date: / /
Page

Lab-6

Parallel cellular Algorithm (PCA)

Input:
f(x): The objective function to optimize
G: Grid dimensions
N: Neighborhood structure
T: Maximum number of iterations
other parameters: Population size, mutation rate

output:
best solution

Pseudocode:

```
Initialize grid G with random solution x[i][j]
compute fitness f(x[i][j]) for each cell

for t = 1 to T do:
    for each cell c[i][j] in G do
        neighbors = get_neighbors(c[i][j], N)
        new_state = update_state(c[i][j], neigh
                                         -bors)
        x[i][j] = new_state

    for each cell c[i][j] in G:
        f(x[i][j]) = compute_fitness(x[i][j])

    if convergence_criteria_met():
        break
x_best = find_best_solution(G)
return x_best
```

Application (visualizing fluid flows)

Input.

Grid size nx = 100 , ny = 50

tau = 0.6 (relaxation time)

steps = 500

obstacle-radius = 5

u0 = 0.1 (inflow velocity at left boundary)

output:

Graphs visualizing flowing fluid at each steps

parallelcellularalgo

November 28, 2024

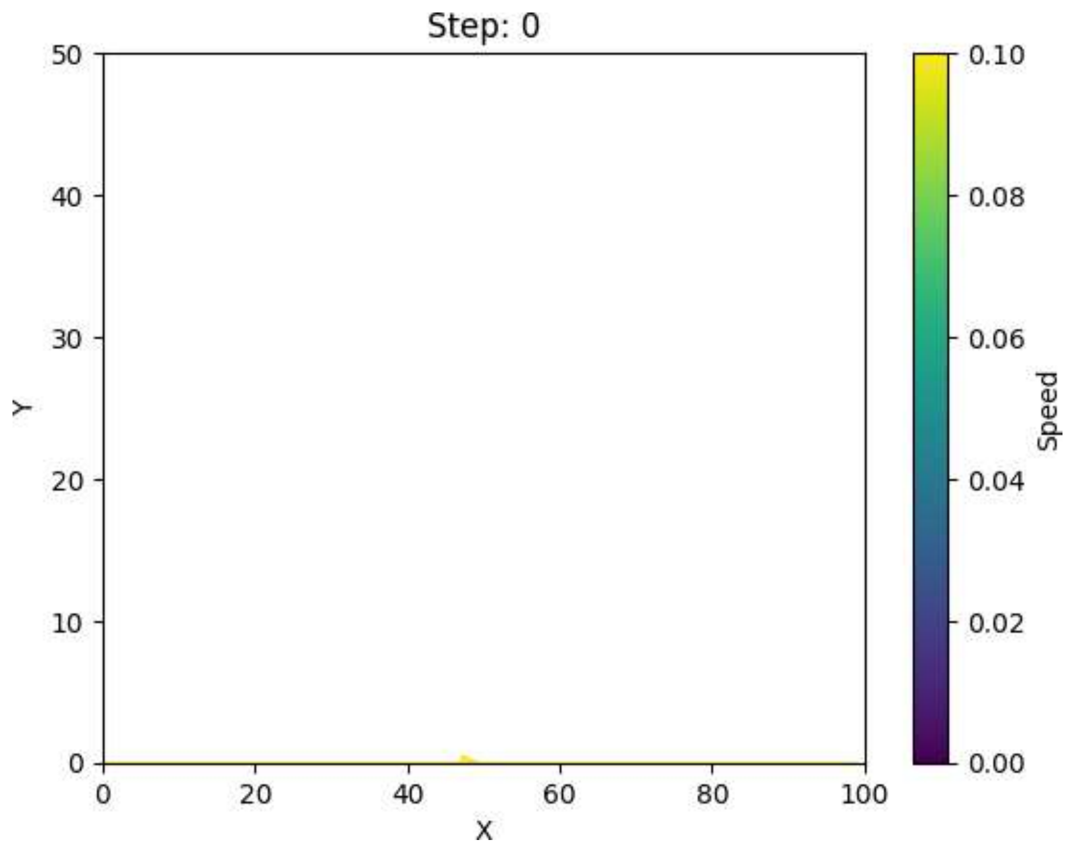
[7]:

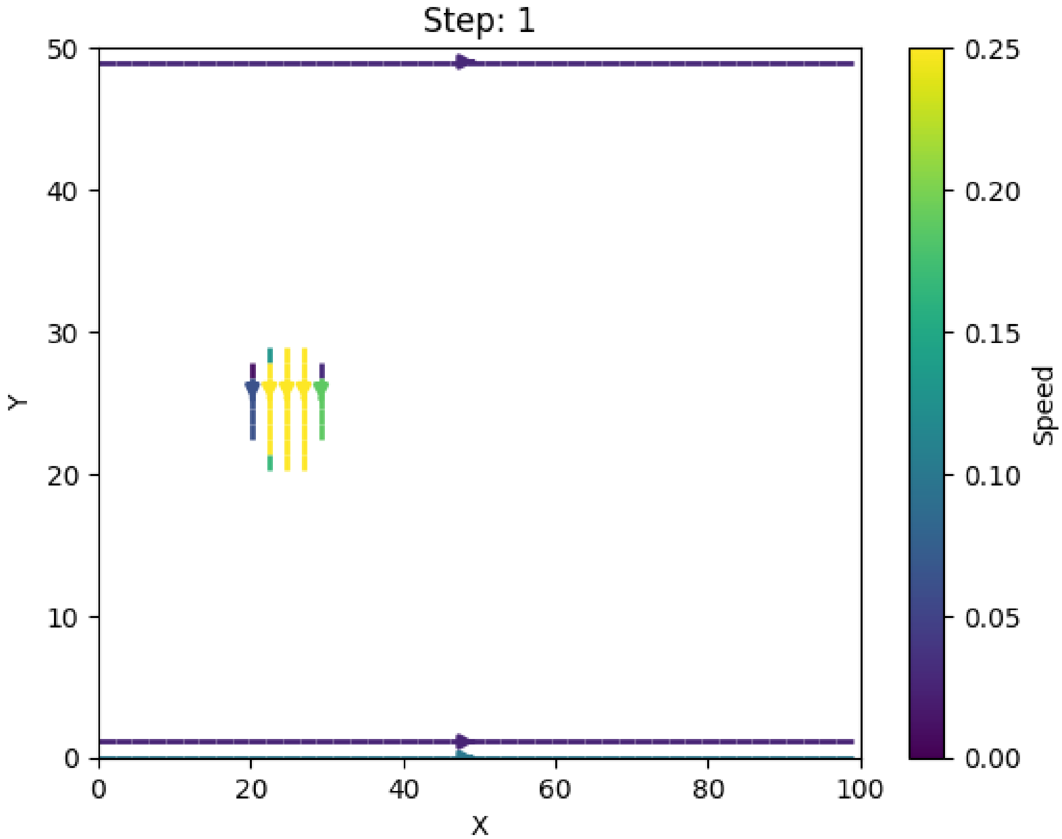
```
#parallel cellular algorithm for simulating motion of fluid
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

import numpy as np
import matplotlib.pyplot as plt


nx, ny = 100, 50  # Grid size
tau = 0.6                   # Relaxation time
steps = 50                  # Number of simulation steps
obstacle_radius = 5  # Radius of a circular obstacle u0 = 0.1
                            # Inflow velocity

# Lattice directions for D2Q9 model
directions = np.array([
    [0, 0],  [1, 0],  [0, 1],  [-1, 0], [0, -1],  # Rest and cardinal directions
    [1, 1], [-1, 1], [-1, -1], [1, -1]                      # Diagonal directions
])
weights = np.array([4/9] + [1/9]*4 + [1/36]*4)              # Weights for equilibrium

  distribution
# Initialize lattice distribution function
f = np.ones((9, nx, ny)) * weights[:, None, None] rho = np.ones((nx,
ny))  # Density
ux = np.zeros((nx, ny))  # x-velocity
uy = np.zeros((nx, ny))  # y-velocity

# Create an obstacle
obstacle = np.fromfunction(
    lambda x, y: (x - nx//4)**2 + (y - ny//2)**2 < obstacle_radius**2, (nx, ny)
)

# Bounce-back boundary condition
def bounce_back(f, obstacle):
    for i in range(9):
        f[i][obstacle] = f[8-i][obstacle]
```

```
# Collision step: Relaxation to equilibrium
def collision(f, rho, ux, uy):
        feq = equilibrium(rho, ux, uy) f += -
        (1/tau) * (f - feq)

# Streaming step: Shift populations to neighboring nodes
def streaming(f):
        for i, (dx, dy) in enumerate(directions): f[i] =
                np.roll(f[i], dx, axis=0)
                f[i] = np.roll(f[i], dy, axis=1)

# Compute equilibrium distribution function
def equilibrium(rho, ux, uy):
        cu = (directions[:, 0, None, None] * ux + directions[:, 1, None, None] * uy) usqr = ux**2 + uy**2
        return weights[:, None, None] * rho * (1 + 3*cu + 9/2*cu**2 - 3/2*usqr)

# Compute macroscopic variables
def macroscopic(f):
        rho = np.sum(f, axis=0)  # Sum over all directions
        ux = np.sum(f * directions[:, 0, None, None], axis=0) / rho uy = np.sum(f *
        directions[:, 1, None, None], axis=0) / rho return rho, ux, uy

# Main simulation loop
for step in range(steps):
        # Compute macroscopic variables rho, ux,
        uy = macroscopic(f) ux[:, 0] = u0
        uy[:, 0] = 0

        # Apply collision and streaming collision(f,
        rho, ux, uy) streaming(f)

        # Enforce bounce-back condition
        bounce_back(f, obstacle)

        # Visualization (every 50 steps)
        if True:
                # Create a grid for the streamplot
                Y, X = np.mgrid[0:ny, 0:nx]  # Grid for streamlines

                # Streamline plot with density control
                plt.streamplot(X, Y, ux.T, uy.T, color=np.sqrt(ux**2 + uy**2).T,

   linewidth=2, cmap='viridis', density=1.5)
```

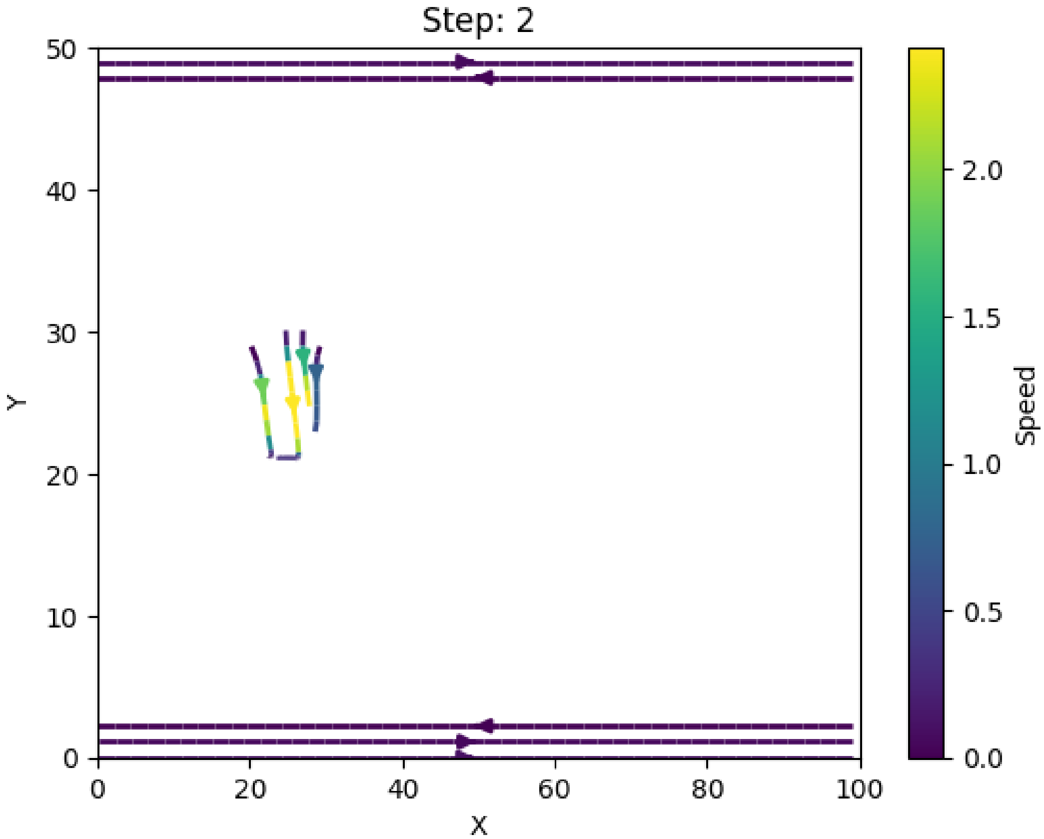
```
        # Add colorbar and title
        plt.colorbar(label='Speed') plt.title(f"Step:
        {step}") plt.xlabel("X")
        plt.ylabel("Y")
        plt.xlim(0,        nx)
        plt.ylim(0,        ny)
        plt.pause(0.1)

plt.show()
```

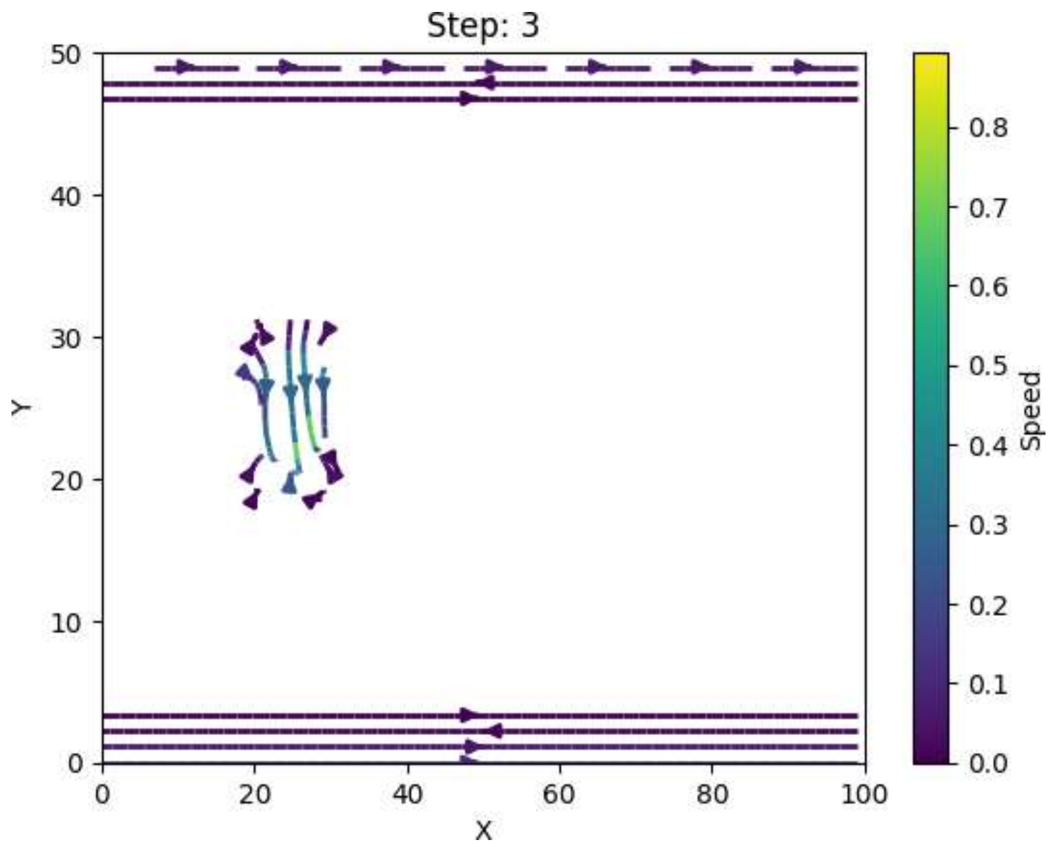
Name:Sudarshan Komar
USN:1BM22CS291



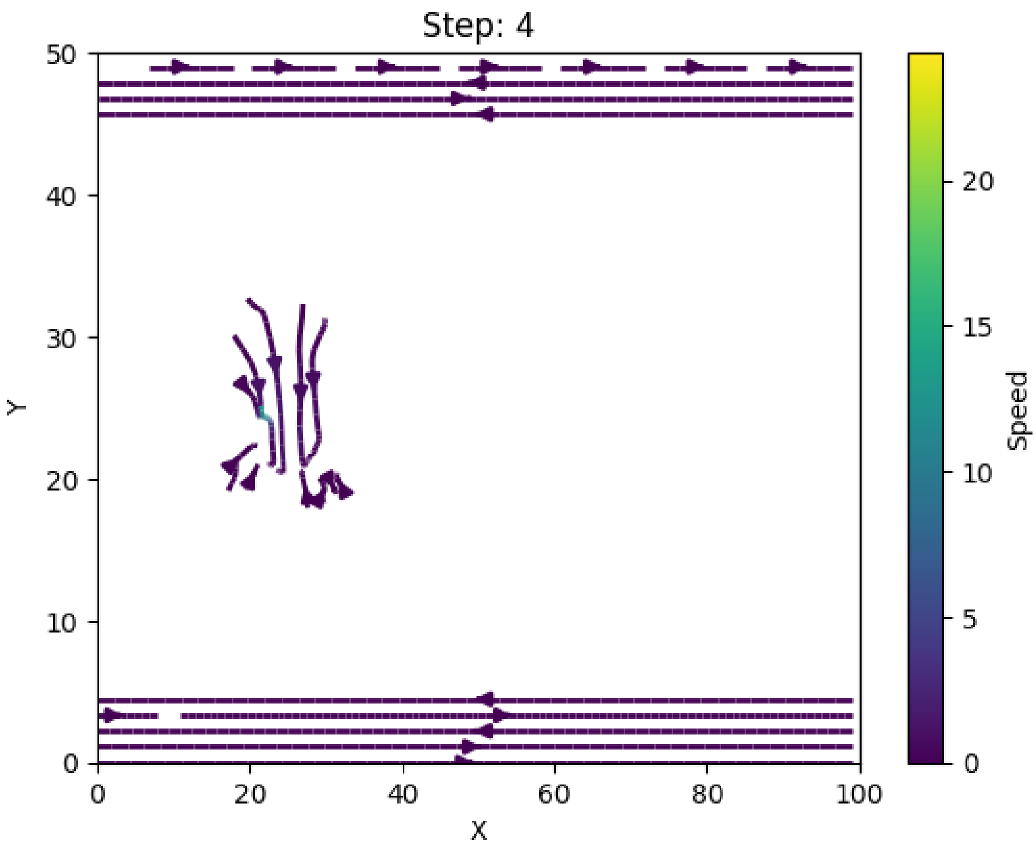


Step: 1
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
0.25
0.20
0.15
0.10
0.05
0.00
Speed

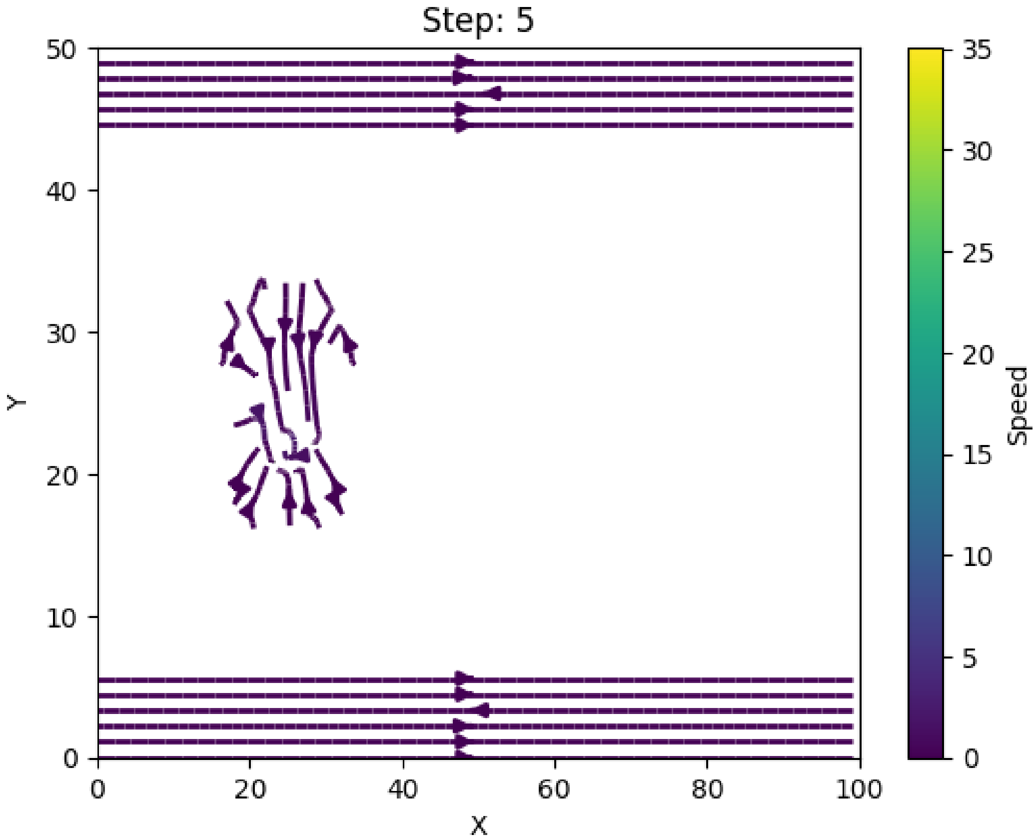

Step: 2
50
40
30
20
10
0
0
20
40
60
80
100
X
Y
2.0
1.5
1.0
0.5
0.0
Speed

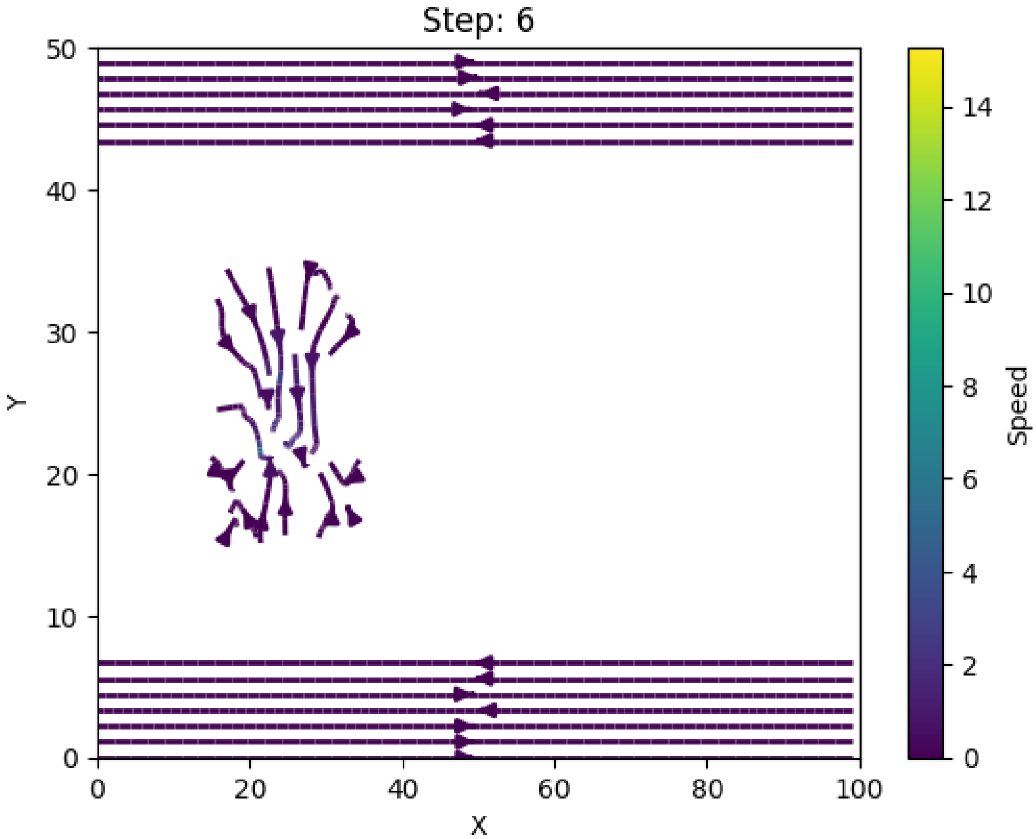


Step: 3
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

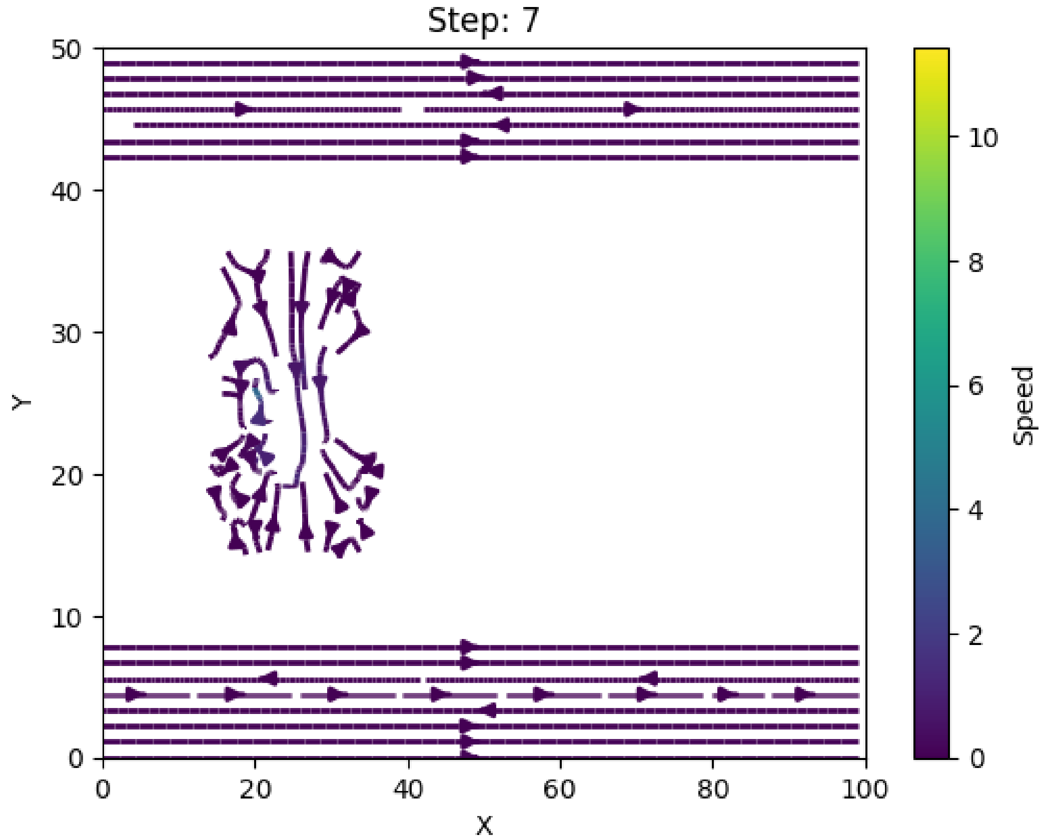


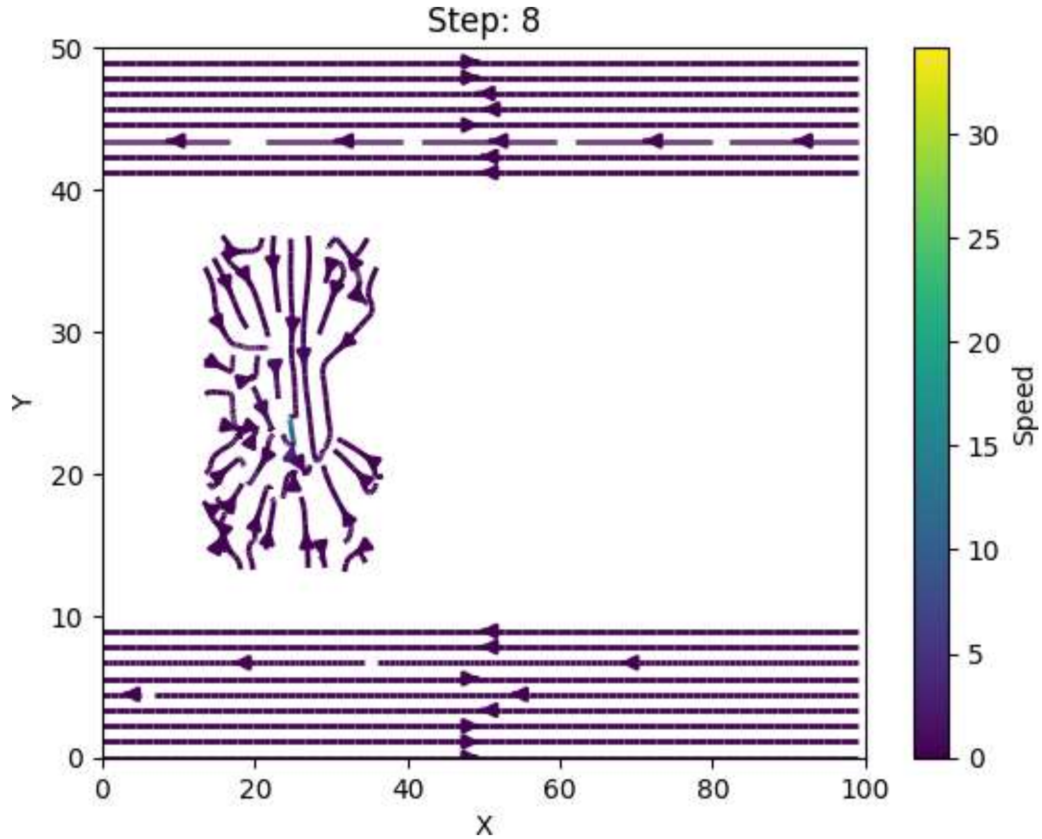
Step: 4
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
20
15
10
5
0
Speed


Step: 5
Y
X
Speed
50
40
30
20
10
0
0
20
40
60
80
100
35
30
25
20
15
10
5
0

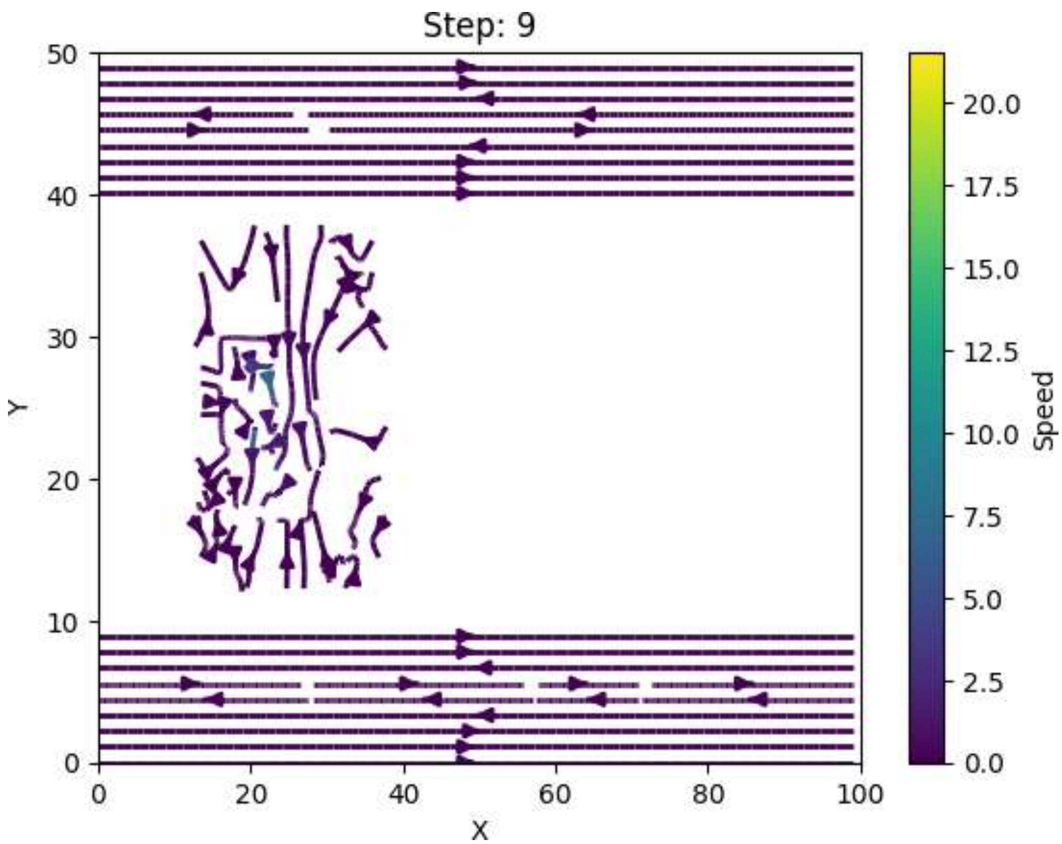


Step: 6
50
40
30
20
10
0
0
20
40
60
80
100
X
Y
0
2
4
6
8
10
12
14
Speed

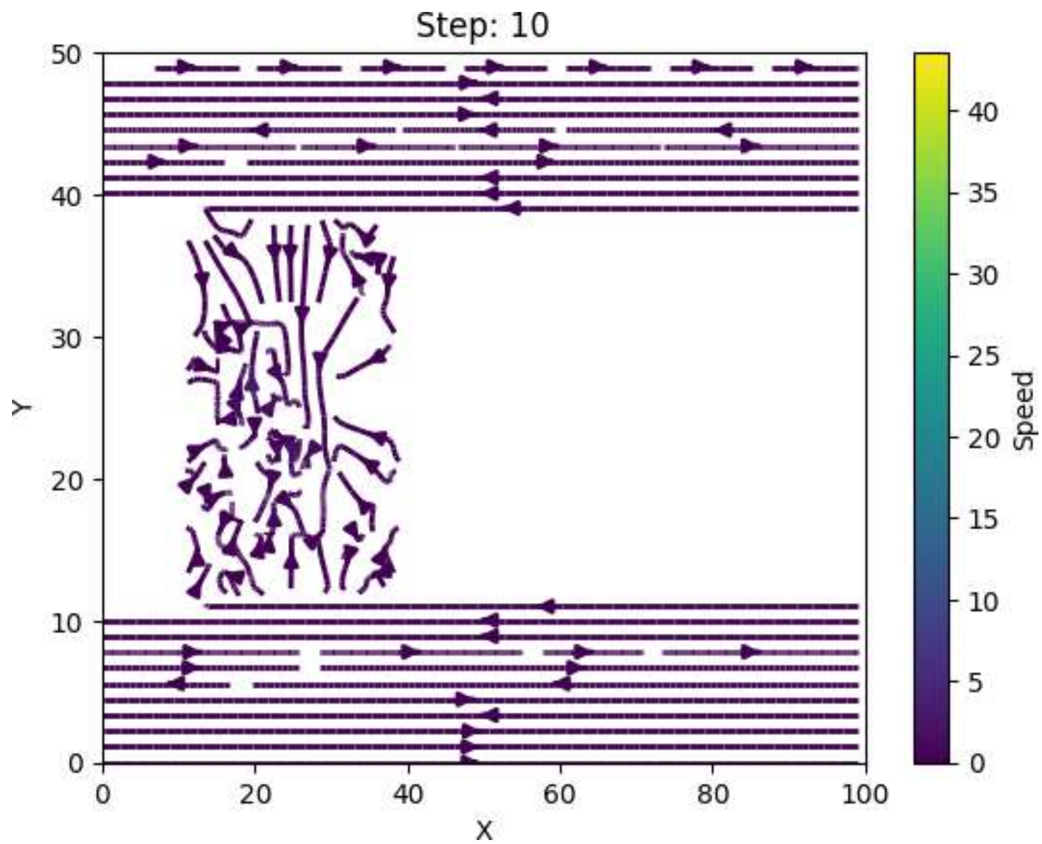

Step: 7
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
10
8
6
4
2
0
Speed



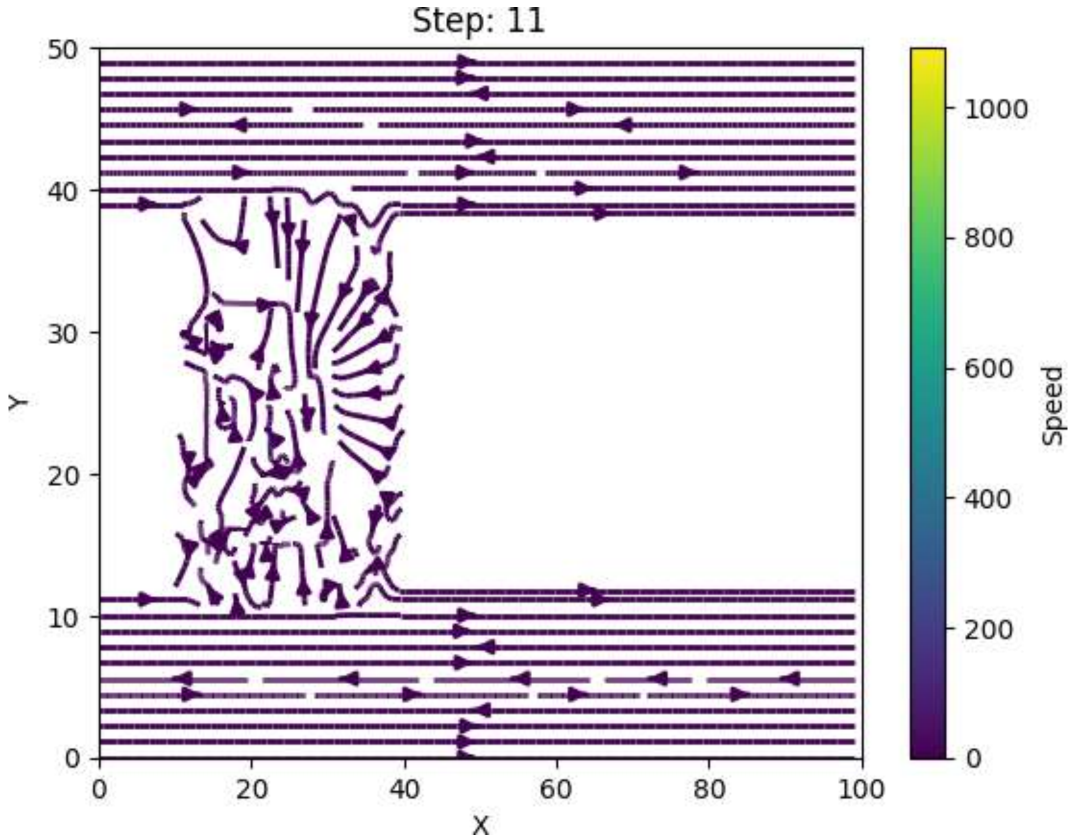
Step: 8
Speed
X
Y



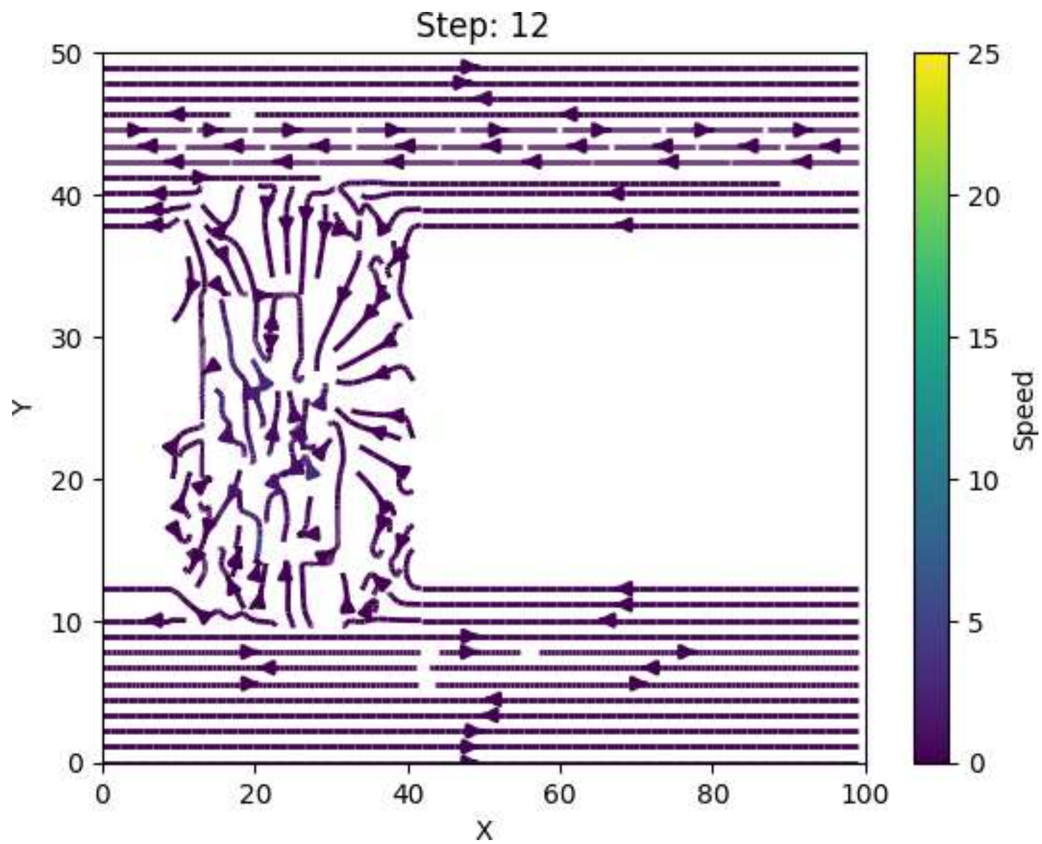
Step: 9
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Speed



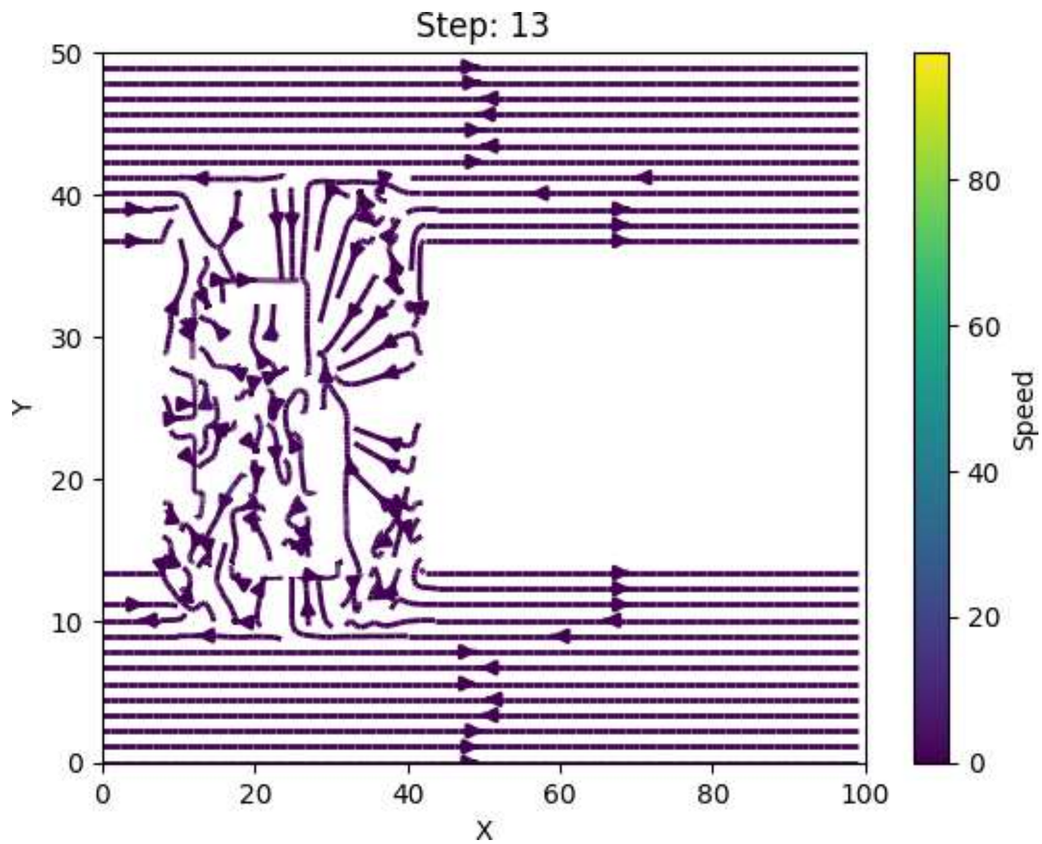
Step: 10
Y
X
Speed



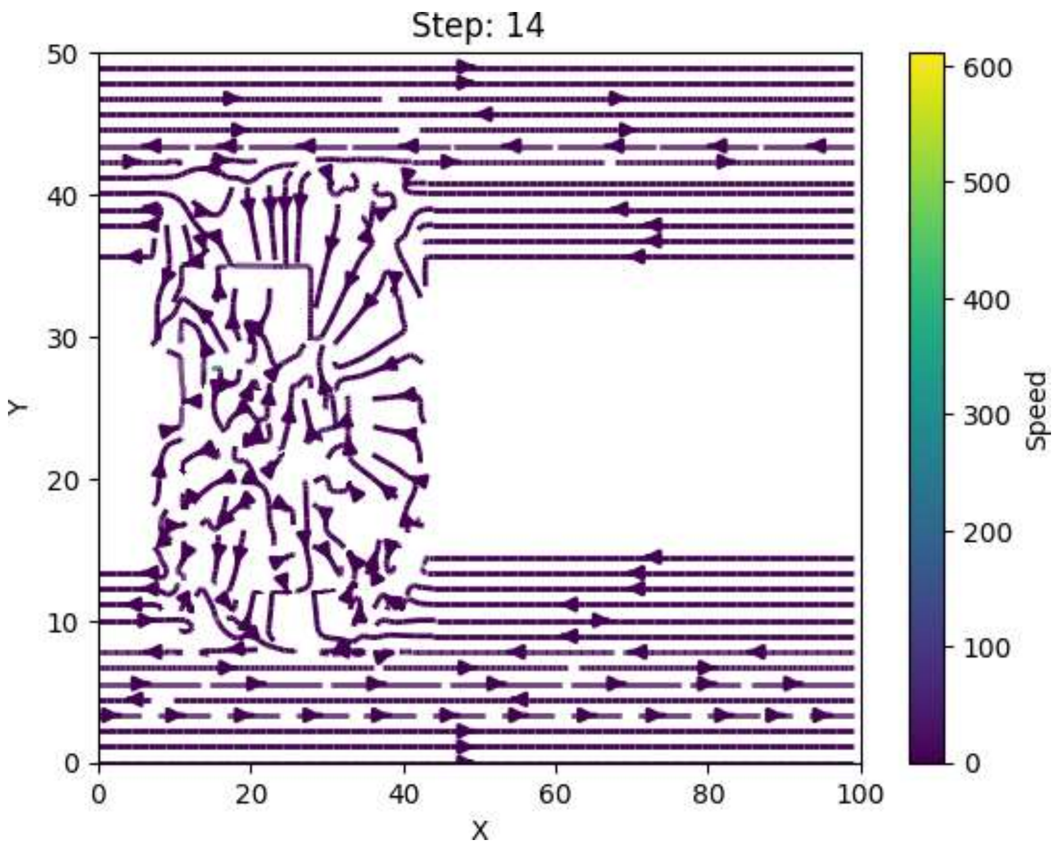
Step: 11
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
1000
800
600
400
200
0
Speed



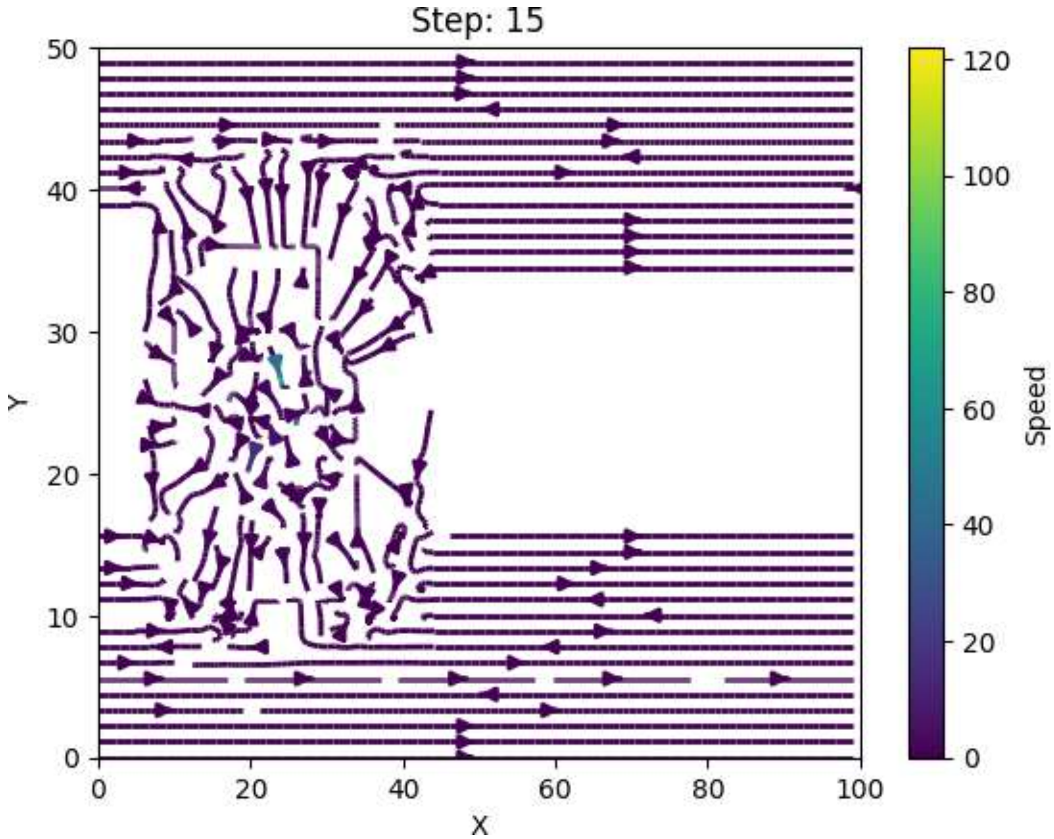
Step: 12
Y
X
Speed
0
20
40
60
80
100
0
10
20
30
40
50
5
15
25

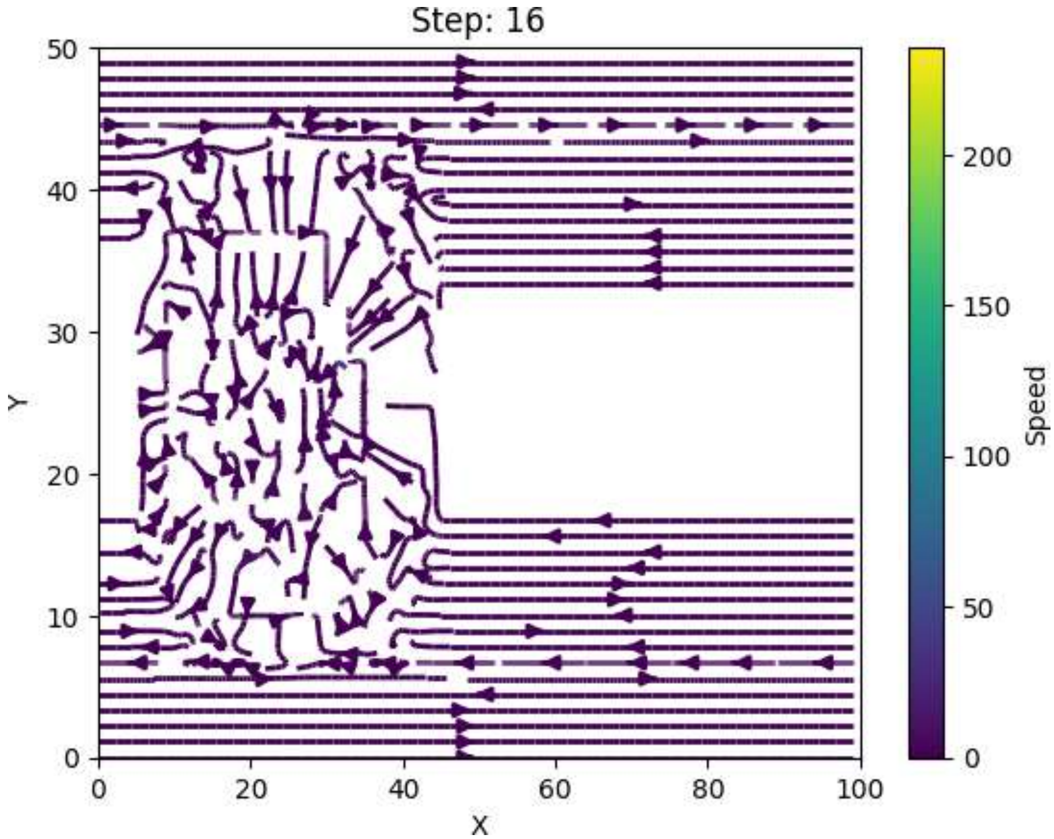


Step: 13
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
80
60
40
20
0
Speed

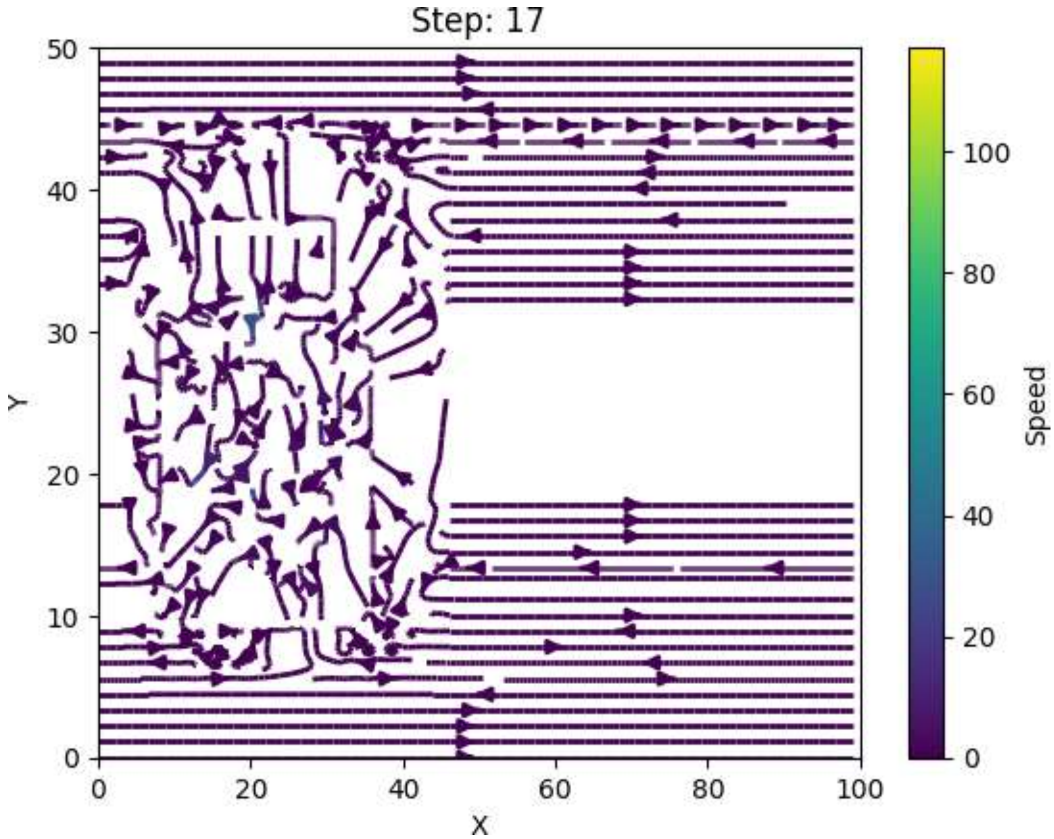


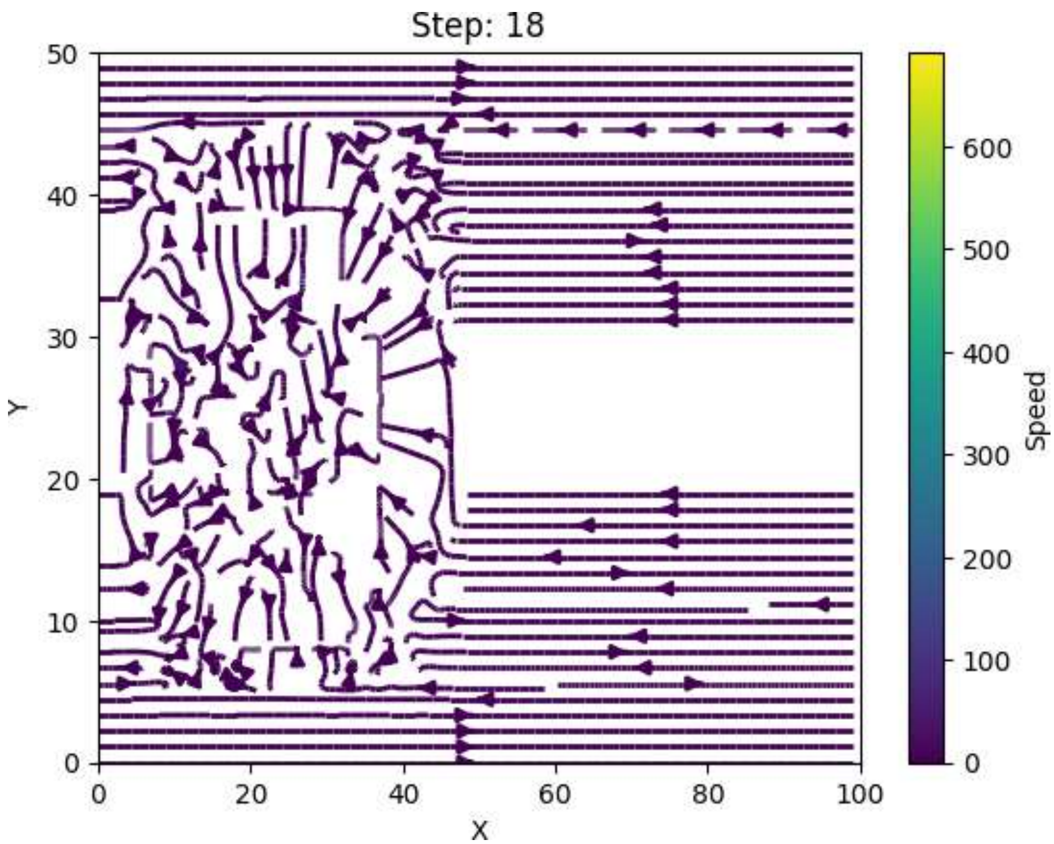
Step: 14
Y
X
Speed

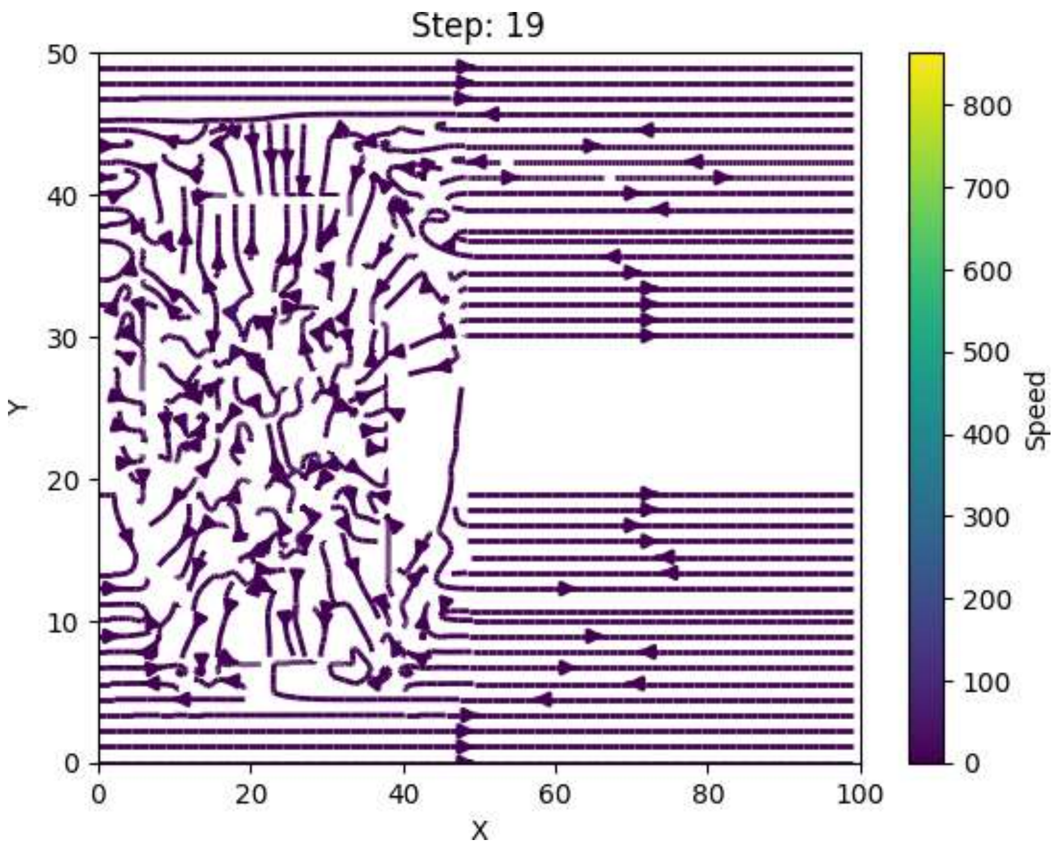

Step: 15
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
120
100
80
60
40
20
0
Speed

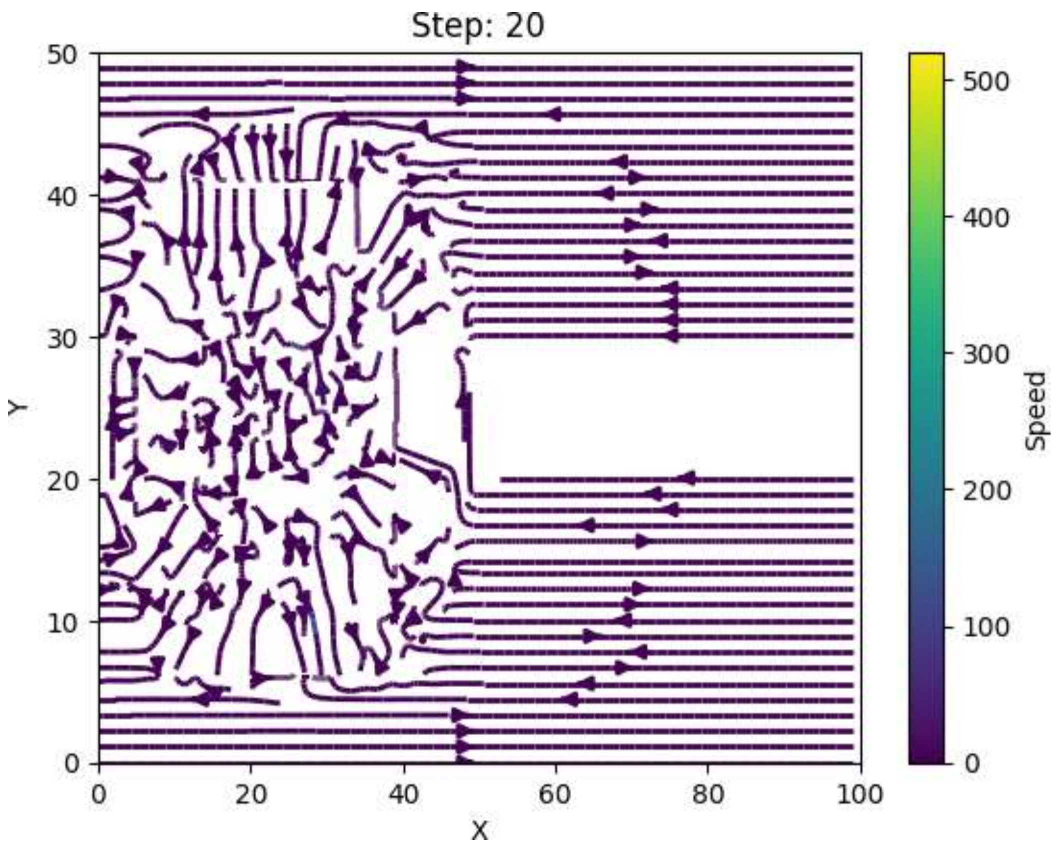


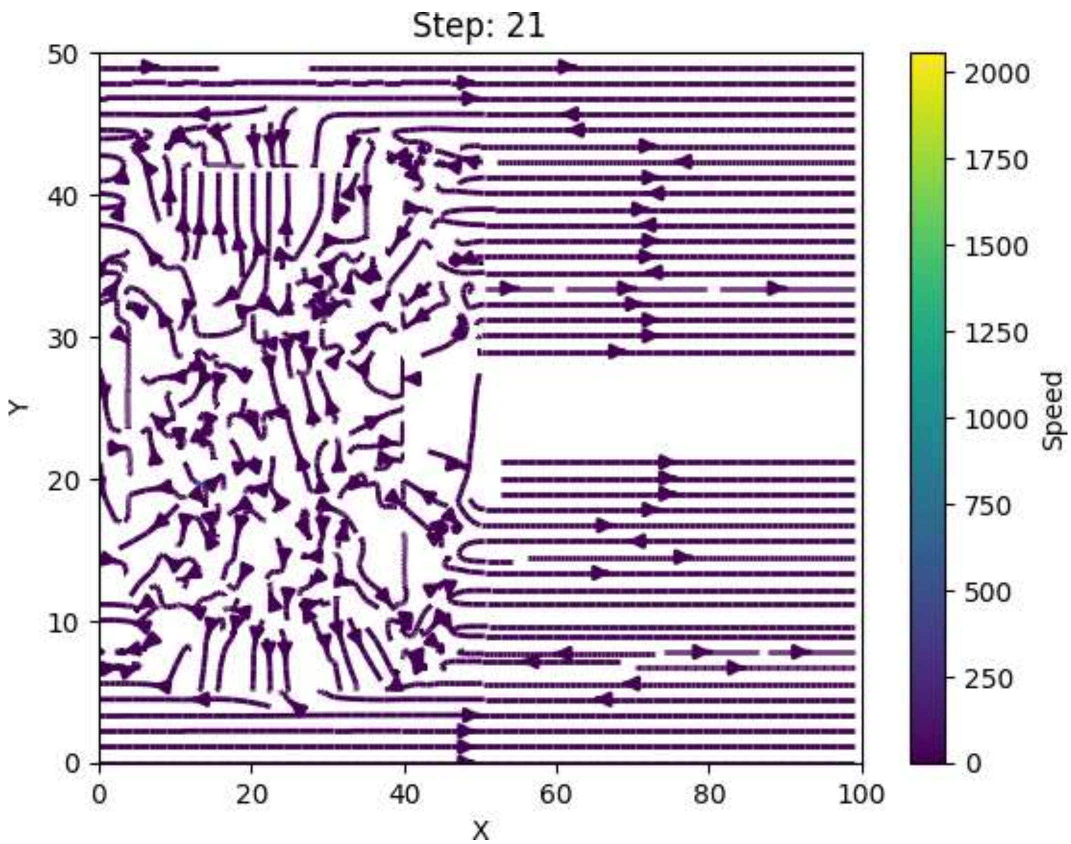
Step: 16
Y
X
Speed


Step: 17
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
Speed
100
80
60
40
20
0

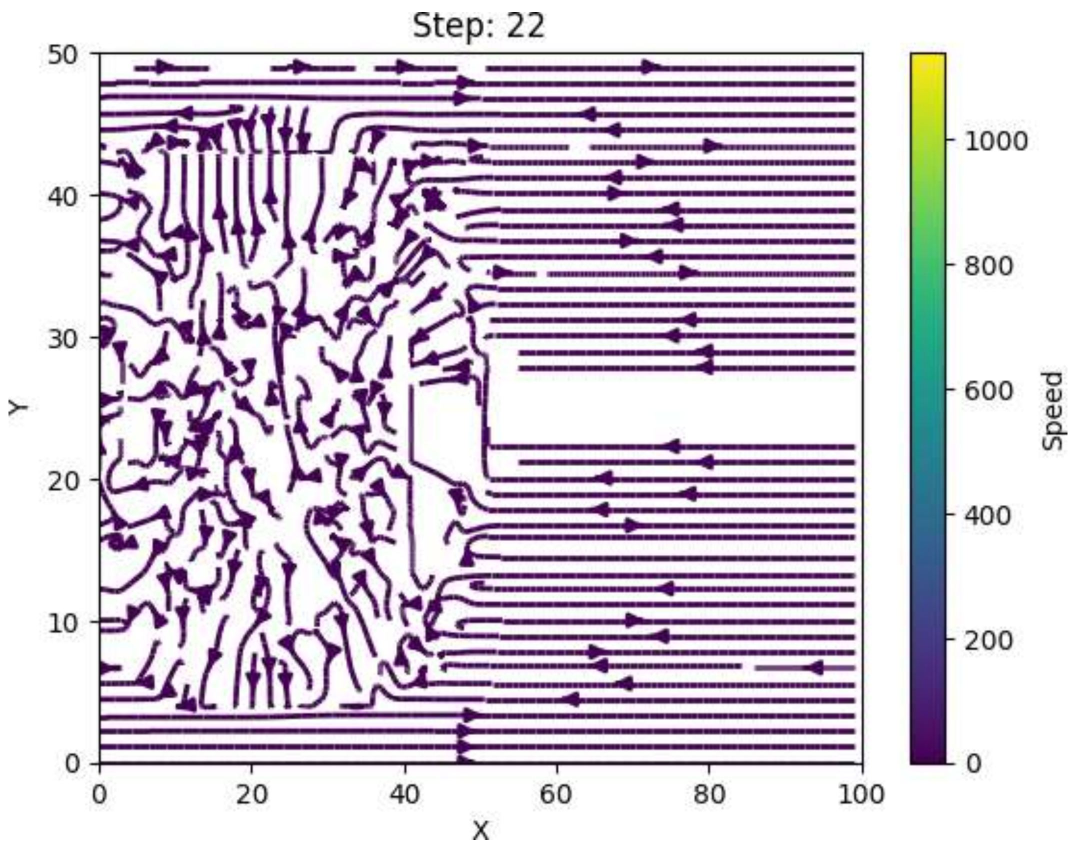

Step: 18
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
600
500
400
300
200
100
0
Speed

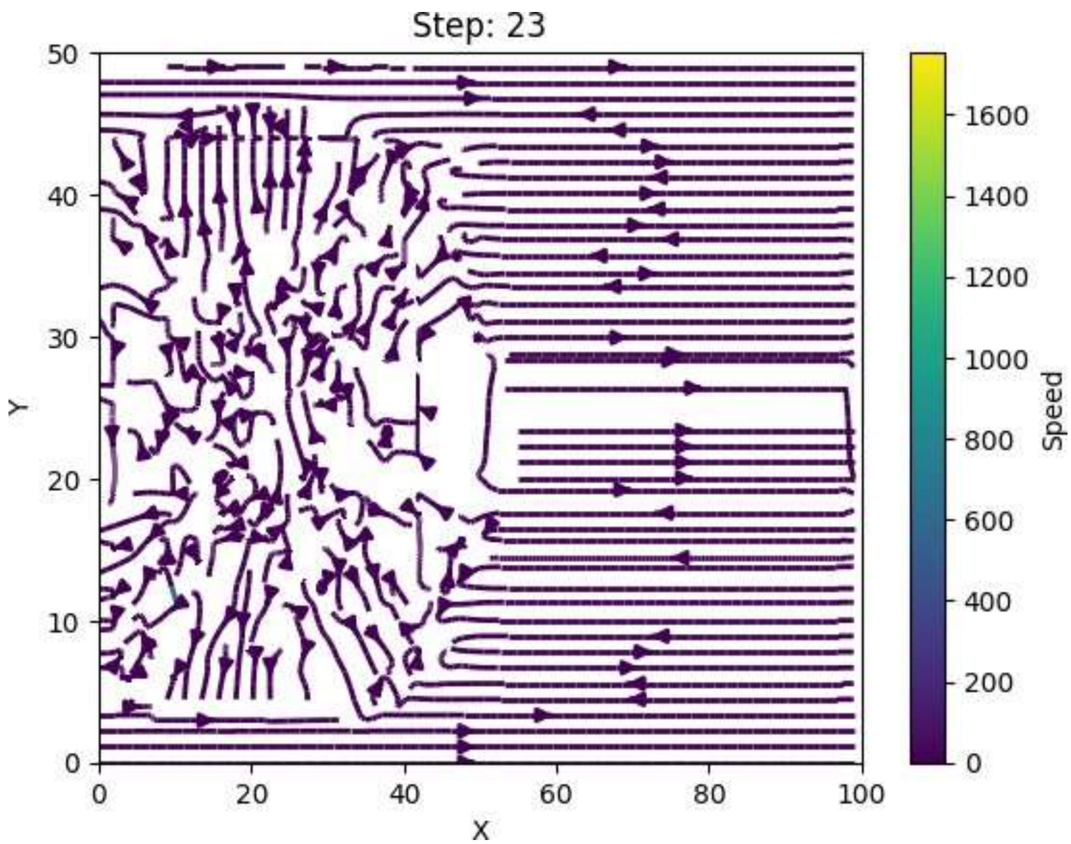

Step: 19
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
800
700
600
500
400
300
200
100
0
Speed

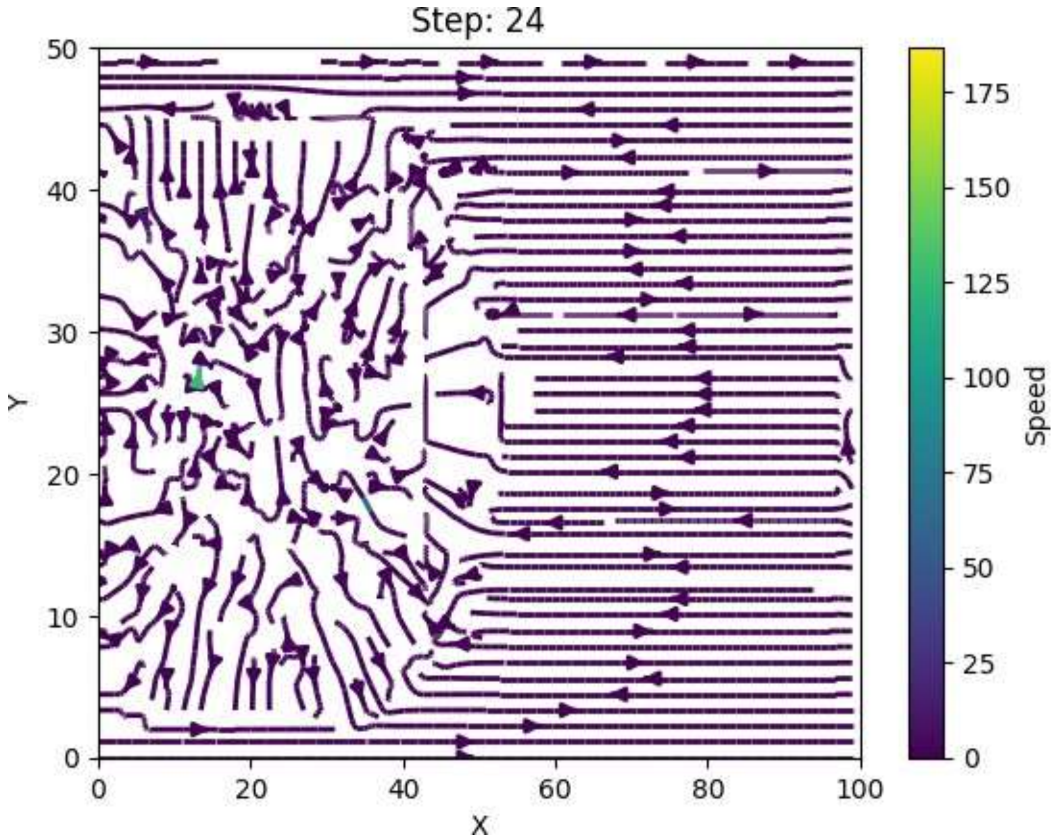

Step: 20
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
500
400
300
200
100
0
Speed



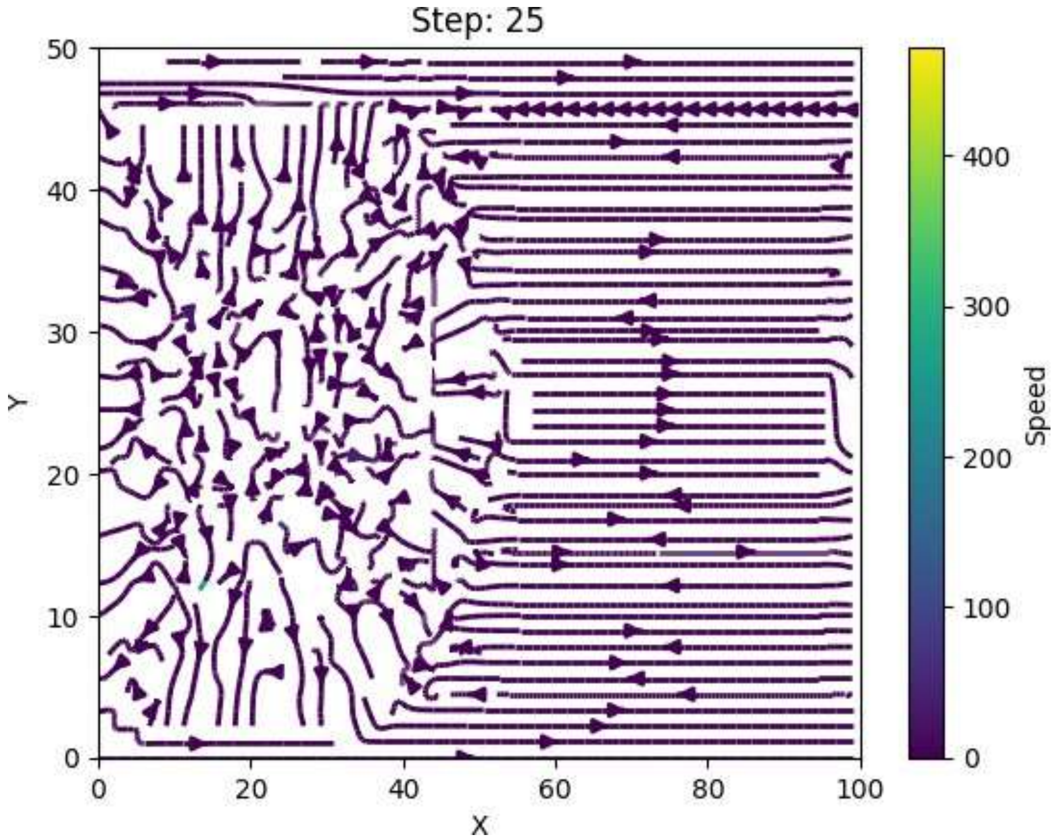
Step: 21
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
250
500
750
1000
1250
1500
1750
2000

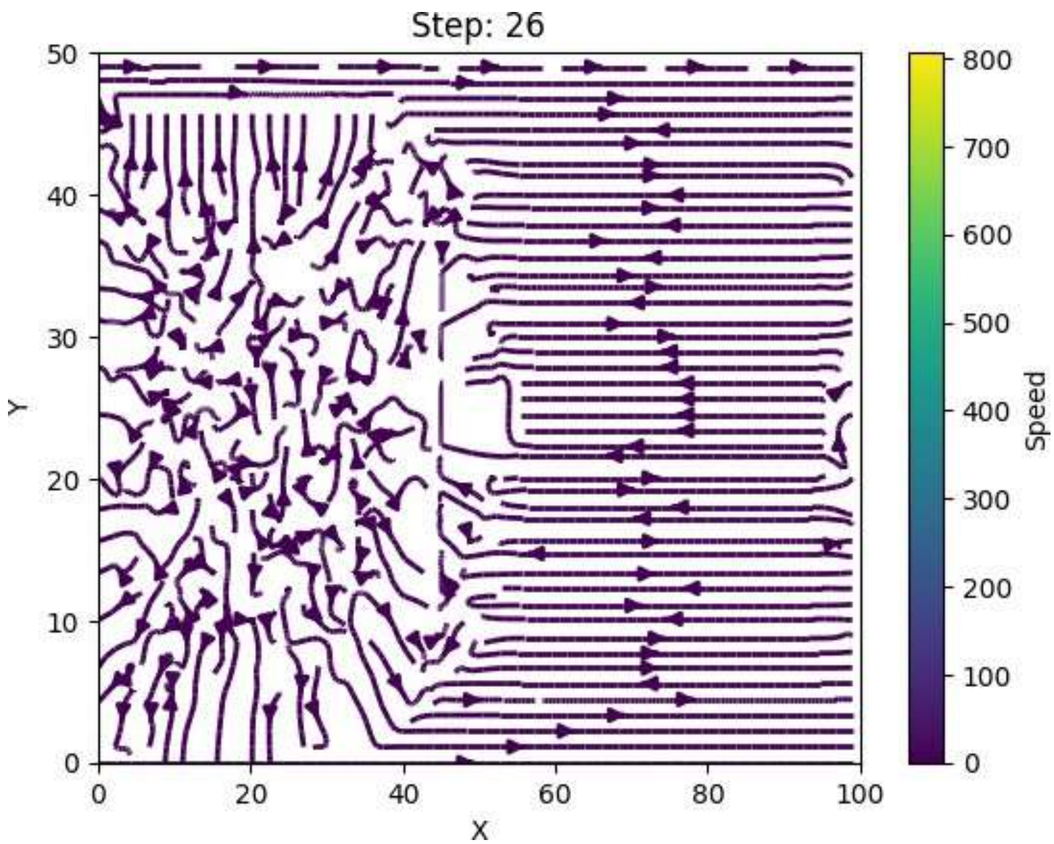

Step: 22
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
1000
800
600
400
200
0
Speed


Step: 23
50
40
30
20
10
0
0
20
40
60
80
100
X
Y
1600
1400
1200
1000
800
600
400
200
0
Speed

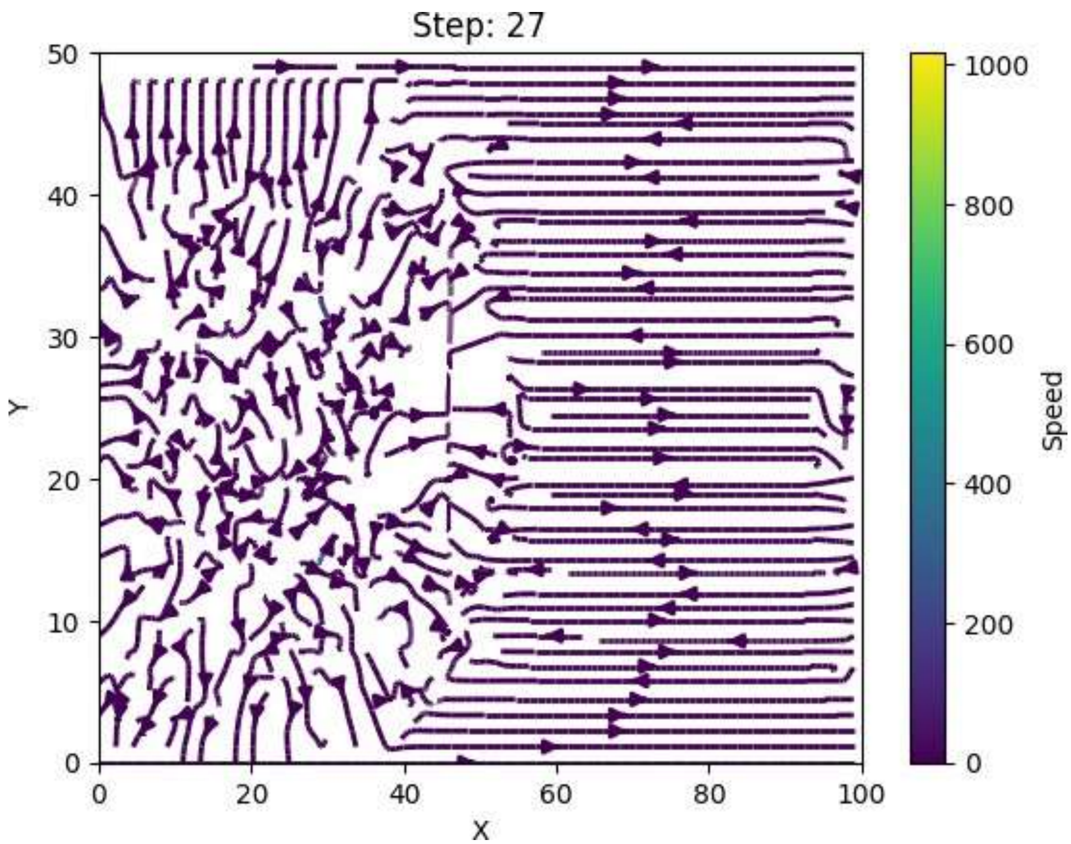


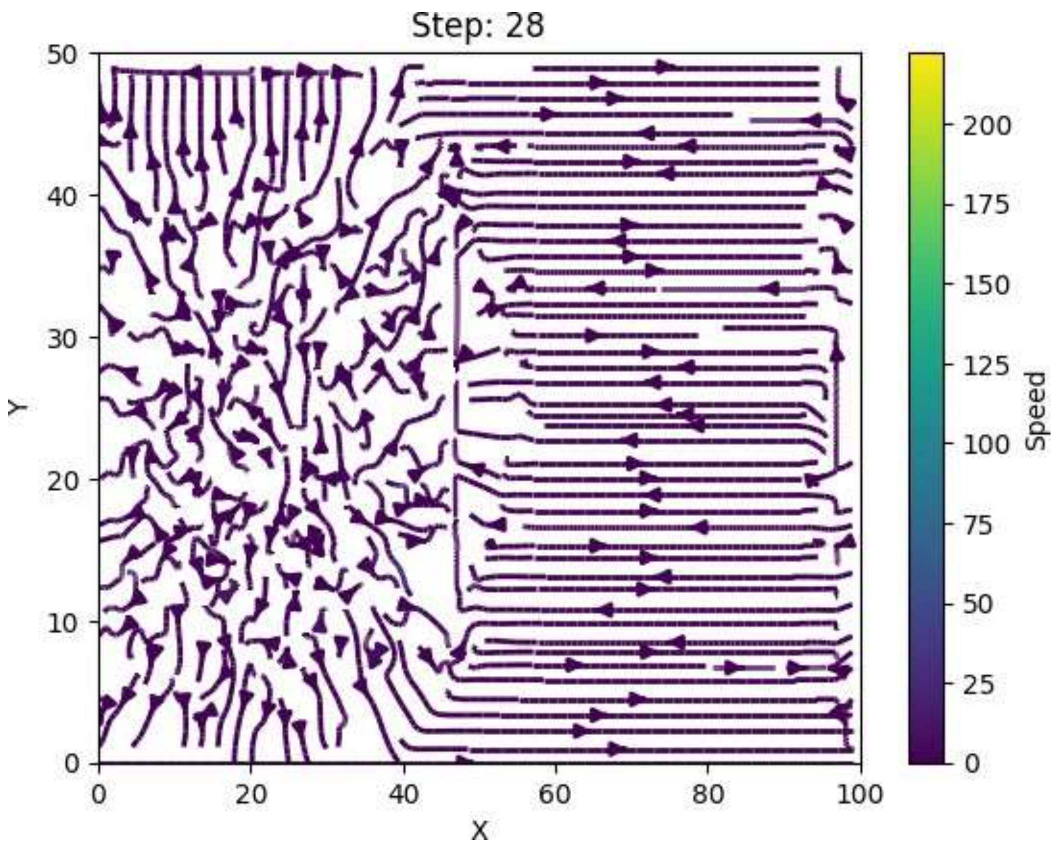
Step: 24
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
25
50
75
100
125
150
175

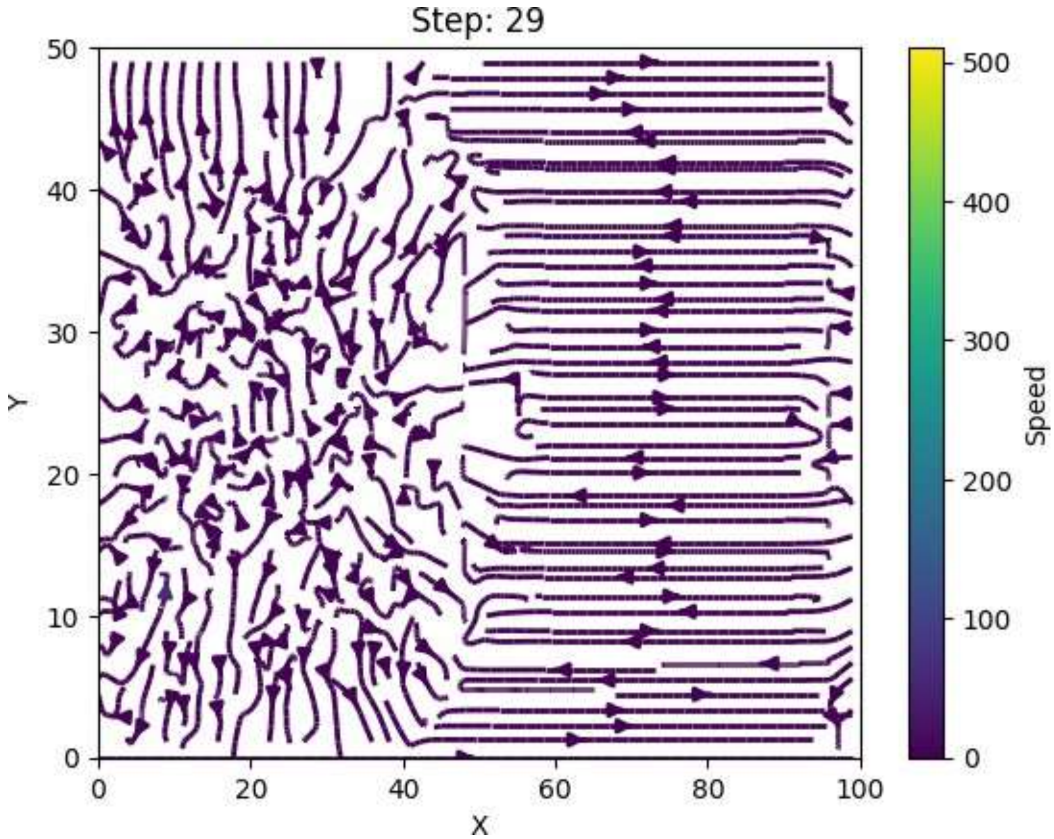

Step: 25
50
40
30
20
10
0
0
20
40
60
80
100
X
Y
400
300
200
100
0
Speed



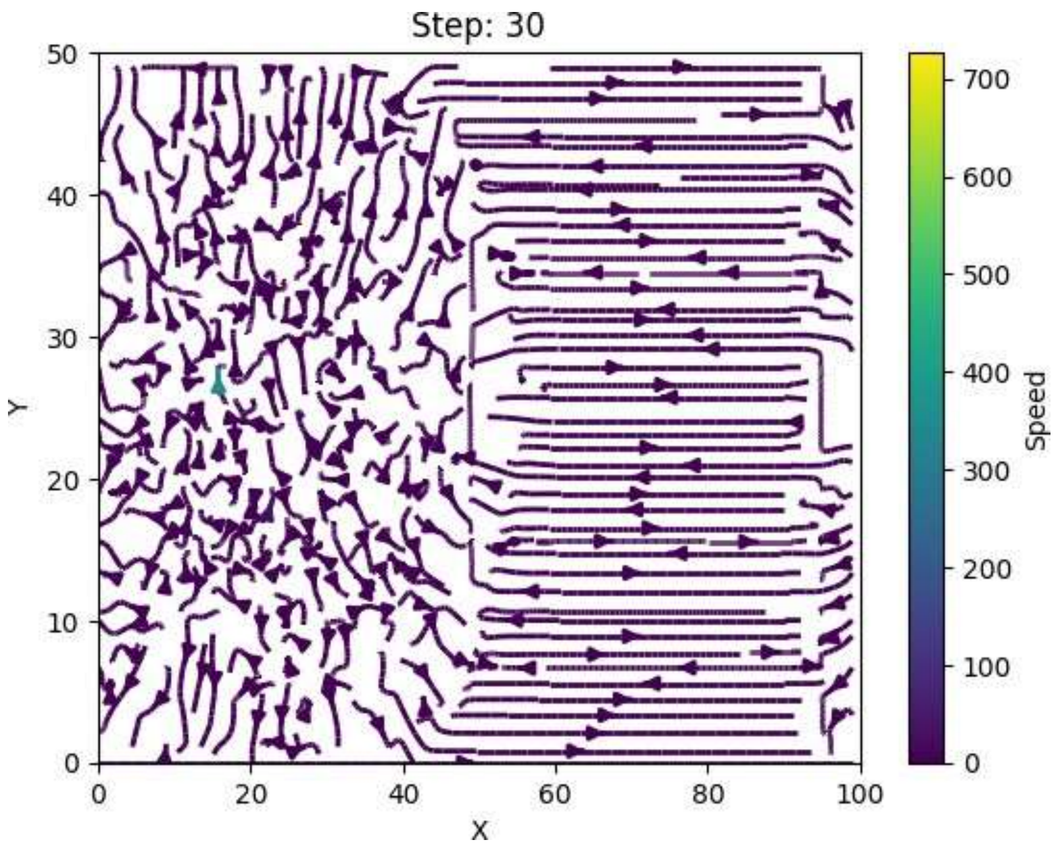
Step: 26
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
100
200
300
400
500
600
700
800

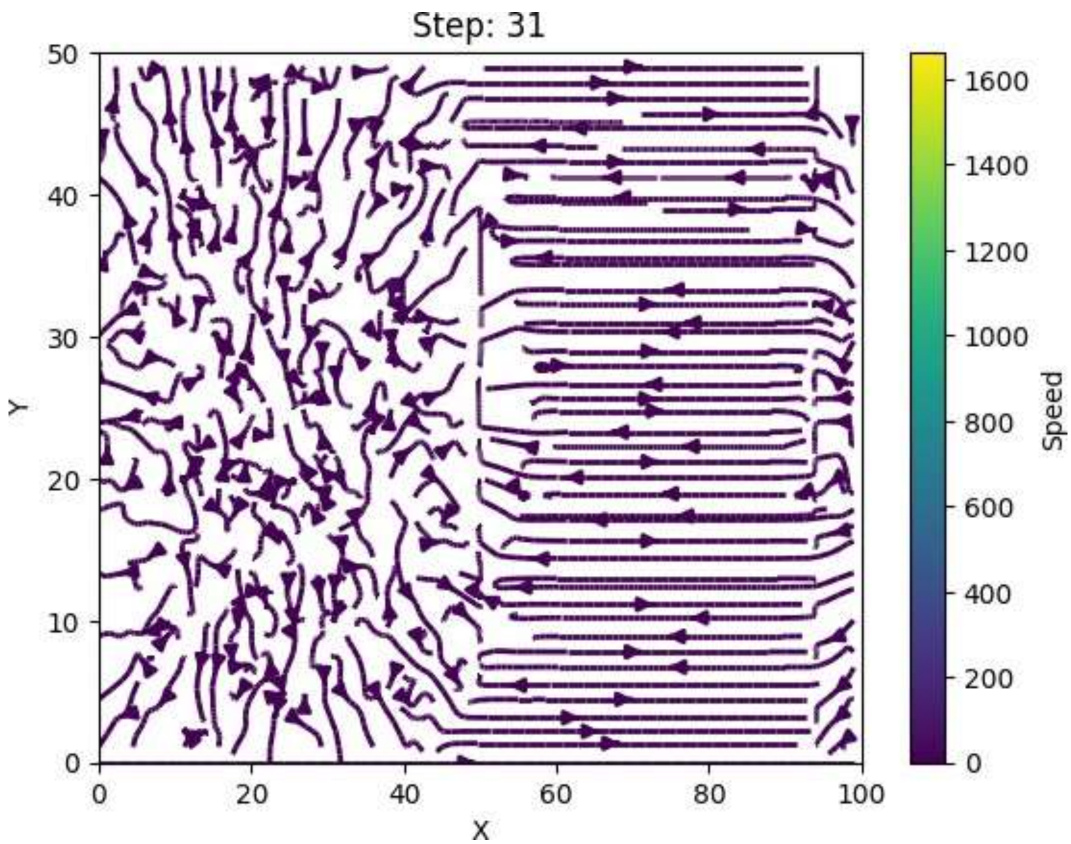

Step: 27
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
200
400
600
800
1000

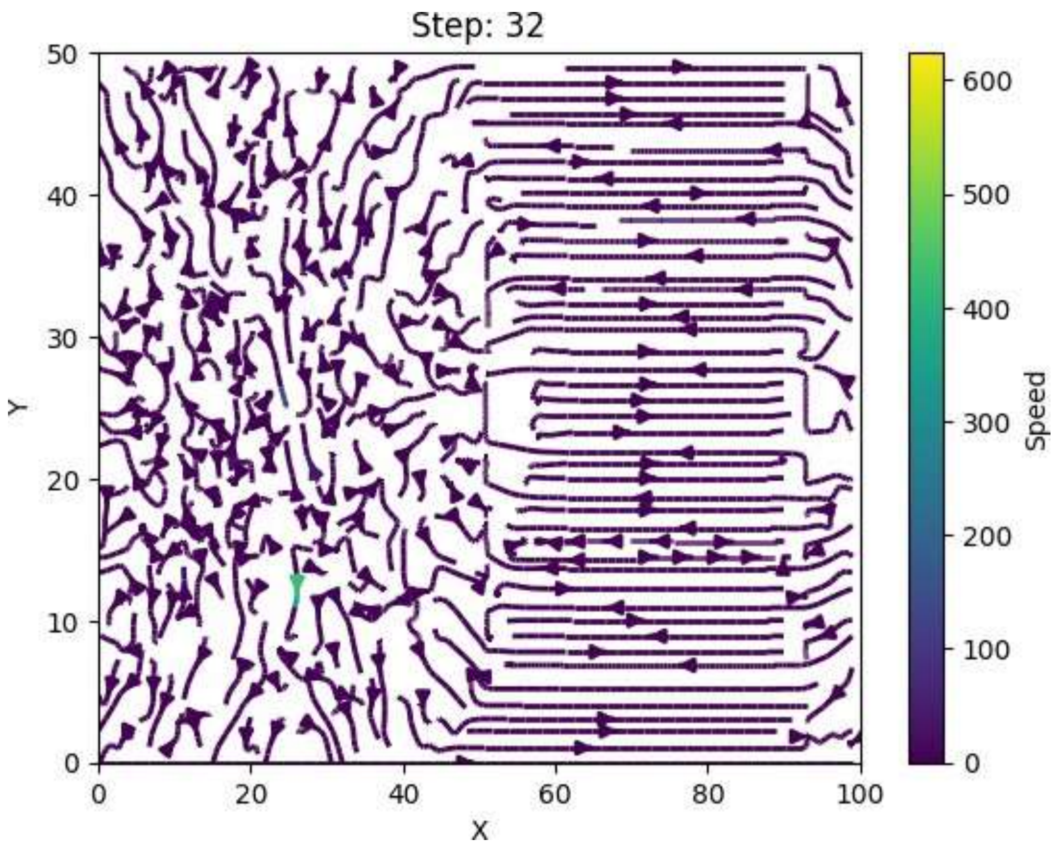

Step: 28
Speed
Y
X
0
10
20
30
40
50
0
20
40
60
80
100
0
25
50
75
100
125
150
175
200



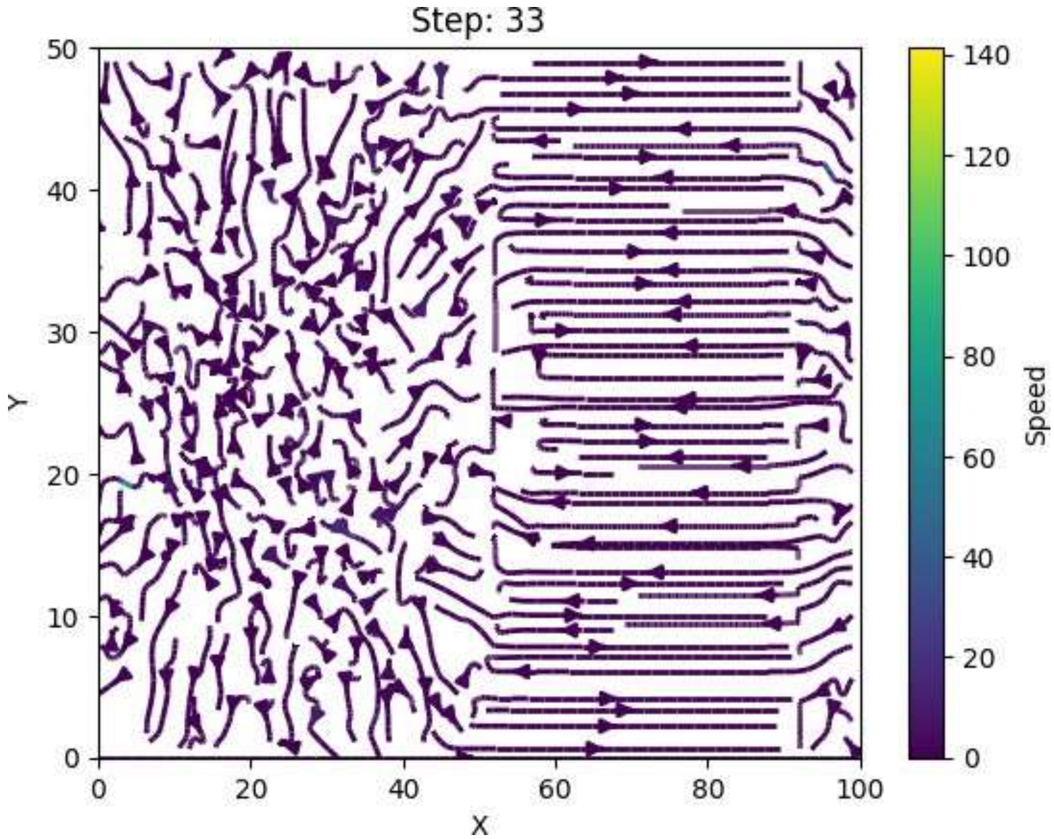
Step: 29
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
500
400
300
200
100
0
Speed

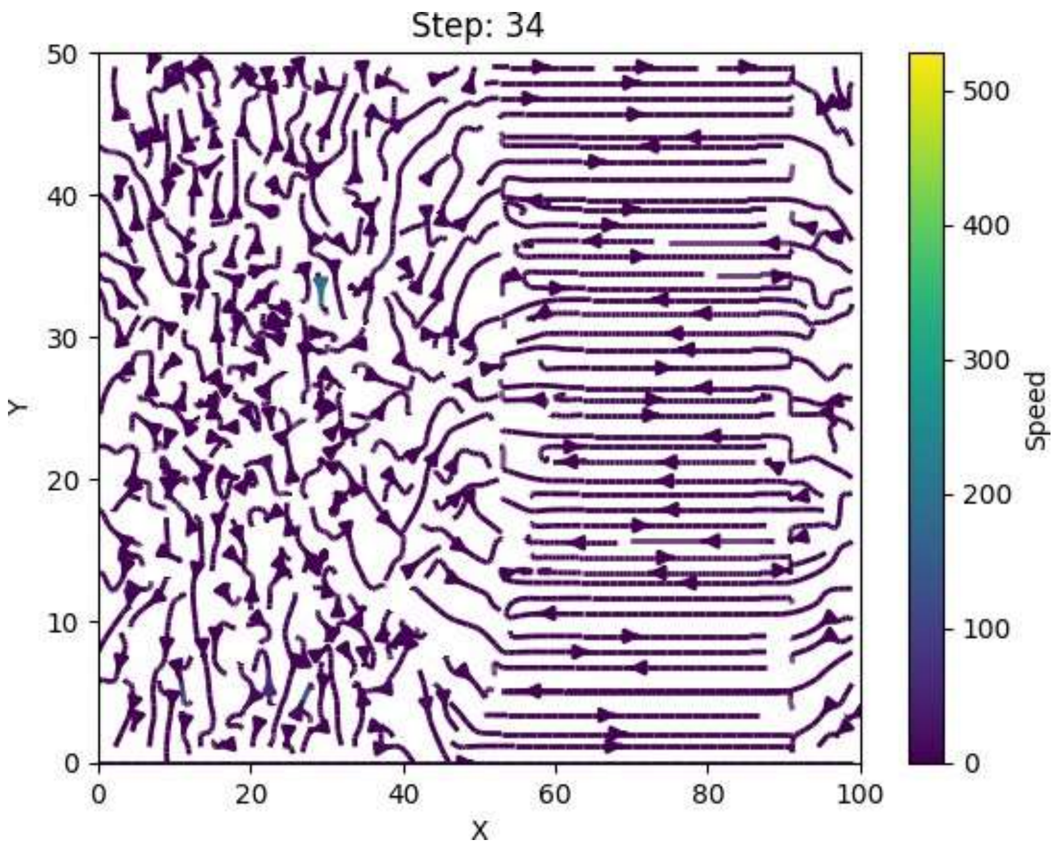

Step: 30
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
700
600
500
400
300
200
100
0
Speed

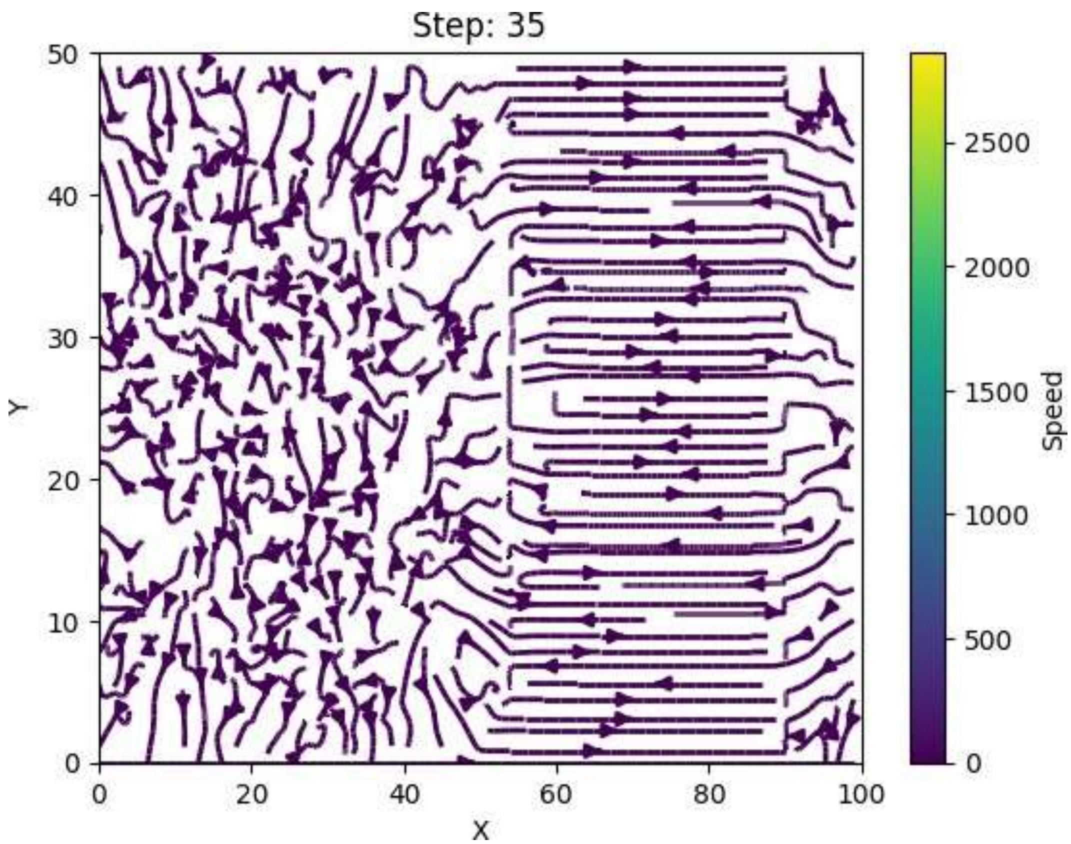

Step: 31
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
1600
1400
1200
1000
800
600
400
200
0
Speed

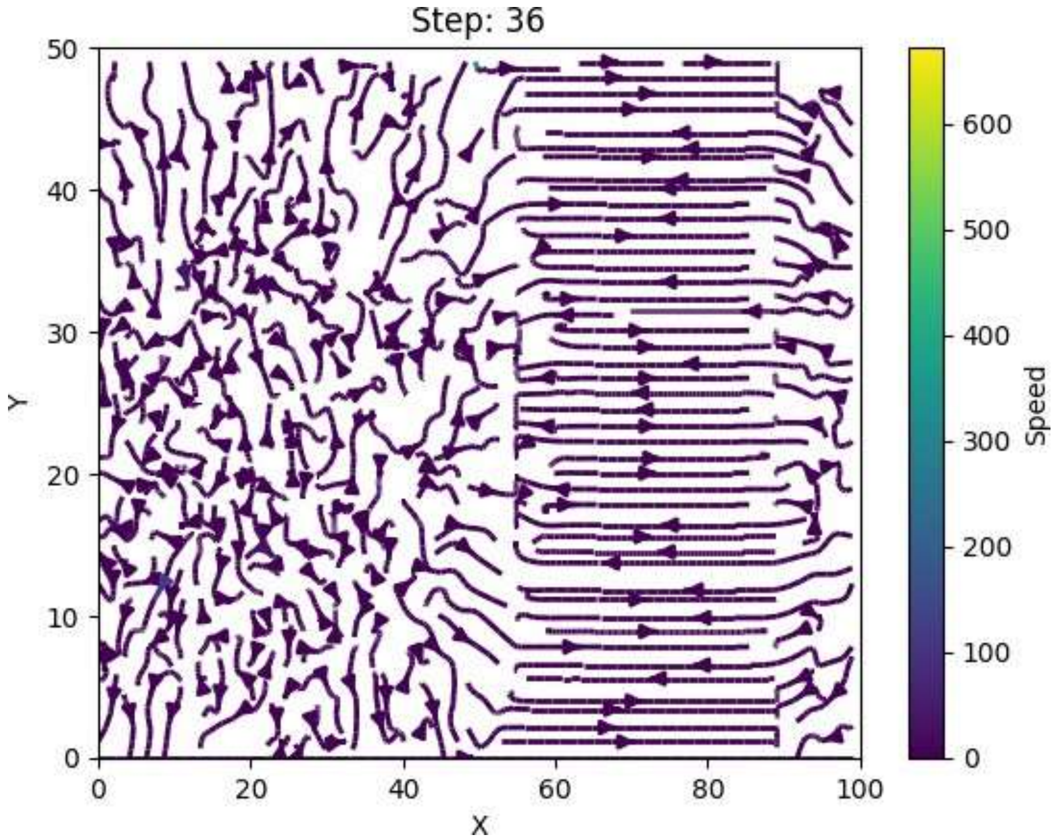


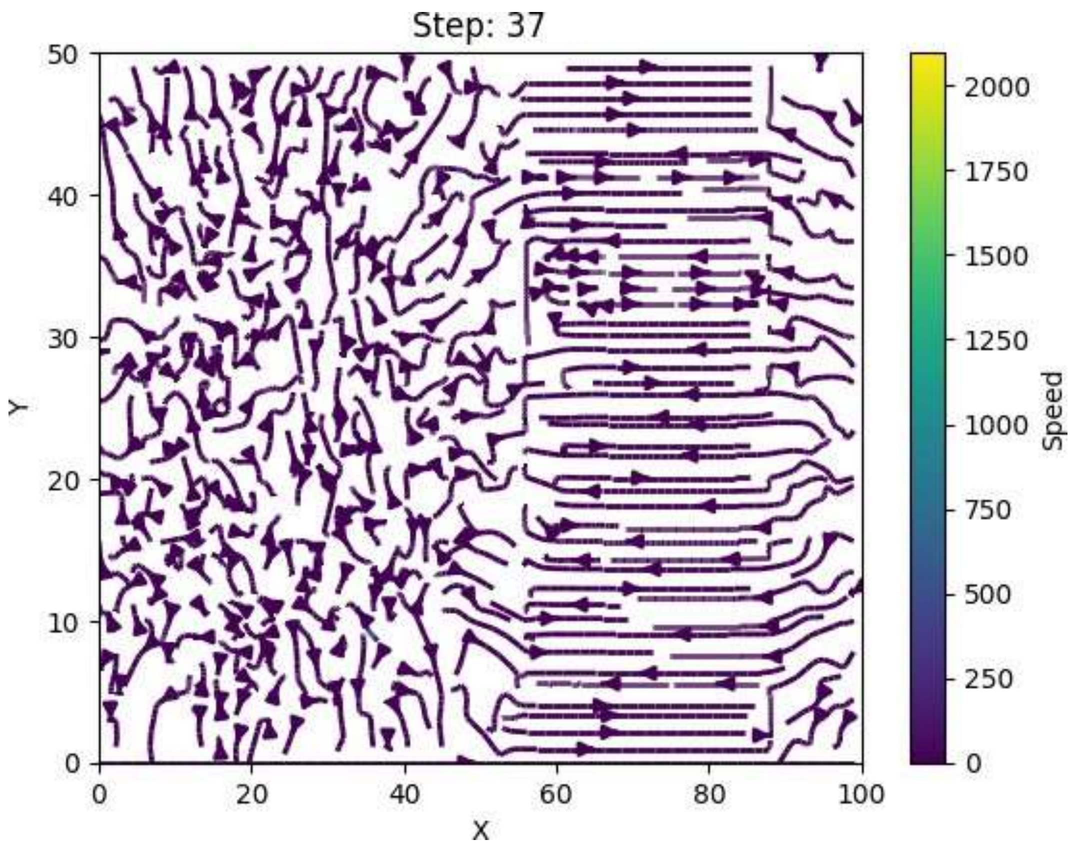
Step: 32
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
100
200
300
400
500
600


Step: 33
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
140
120
100
80
60
40
20
0
Speed

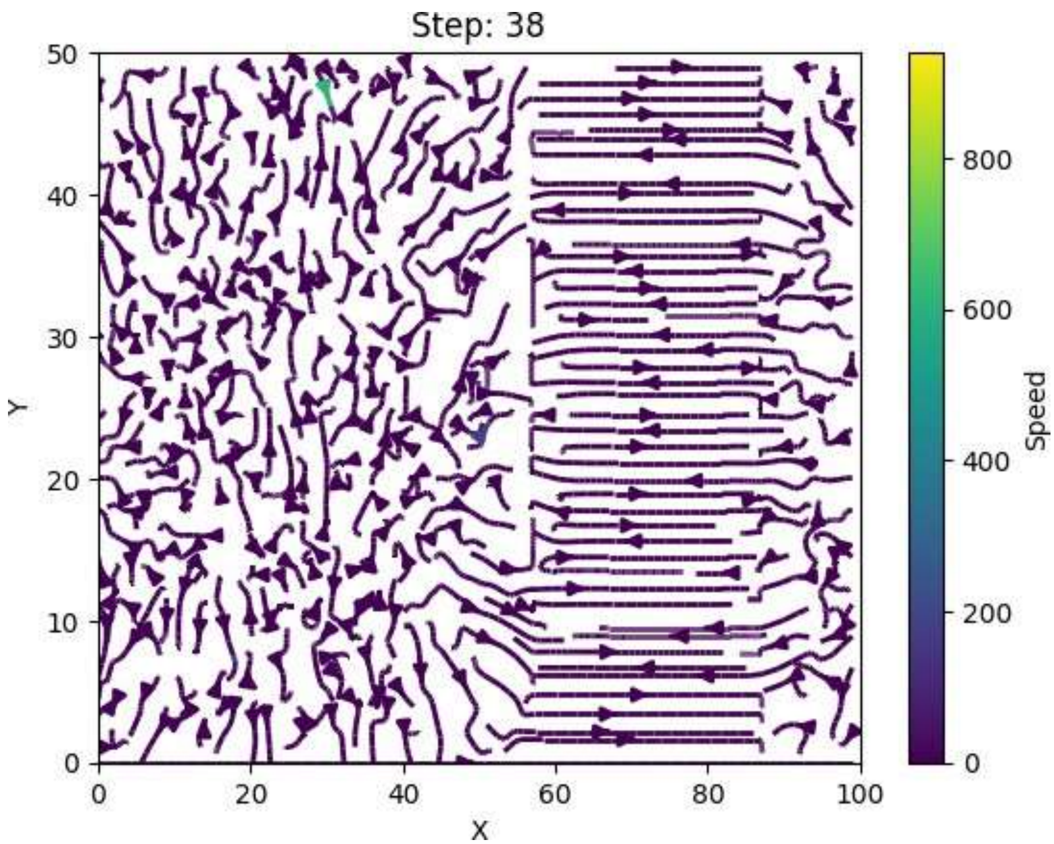

Step: 34
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
500
400
300
200
100
0
Speed

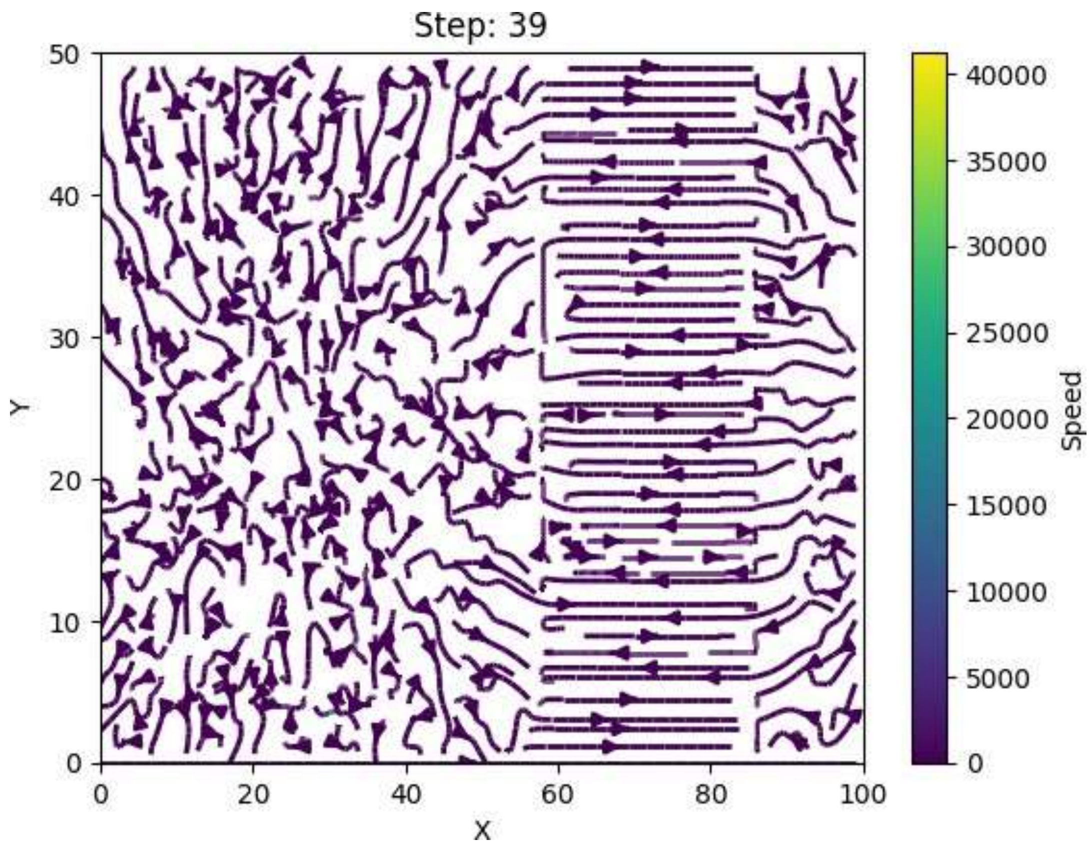

Step: 35
Y
X
Speed
0
20
40
60
80
100
0
10
20
30
40
50
2500
2000
1500
1000
500
0

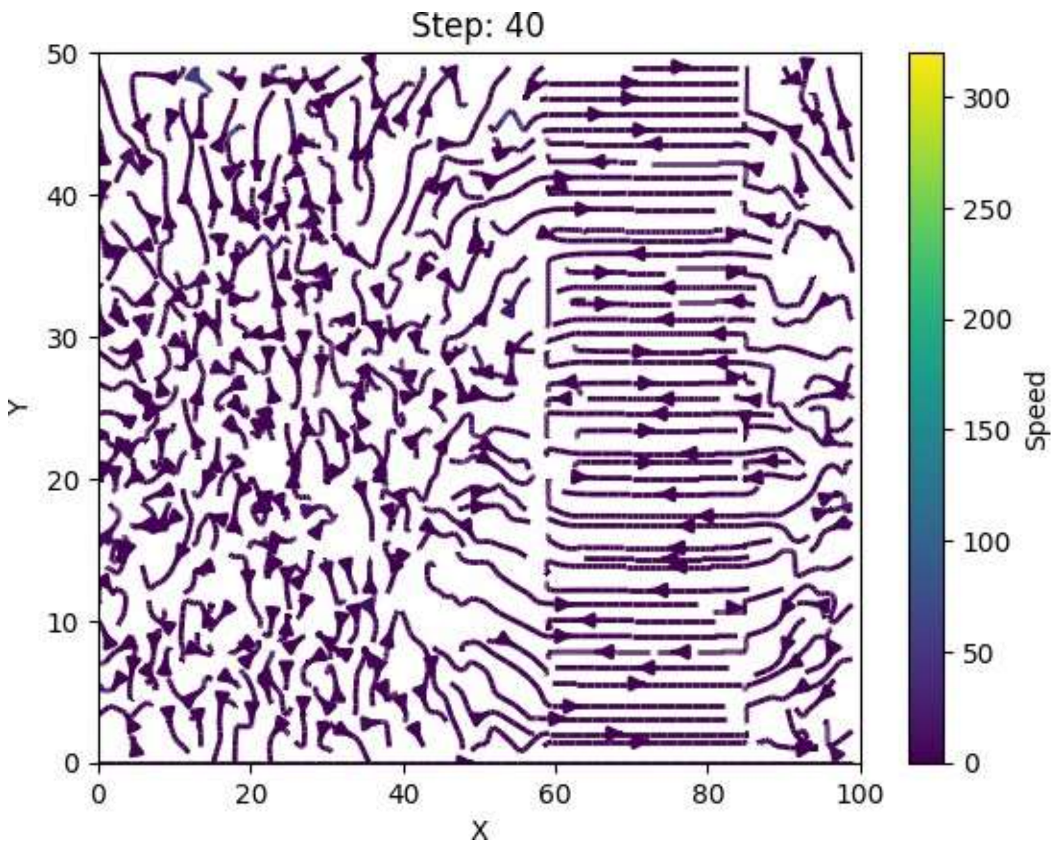

Step: 36
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
600
500
400
300
200
100
0
Speed



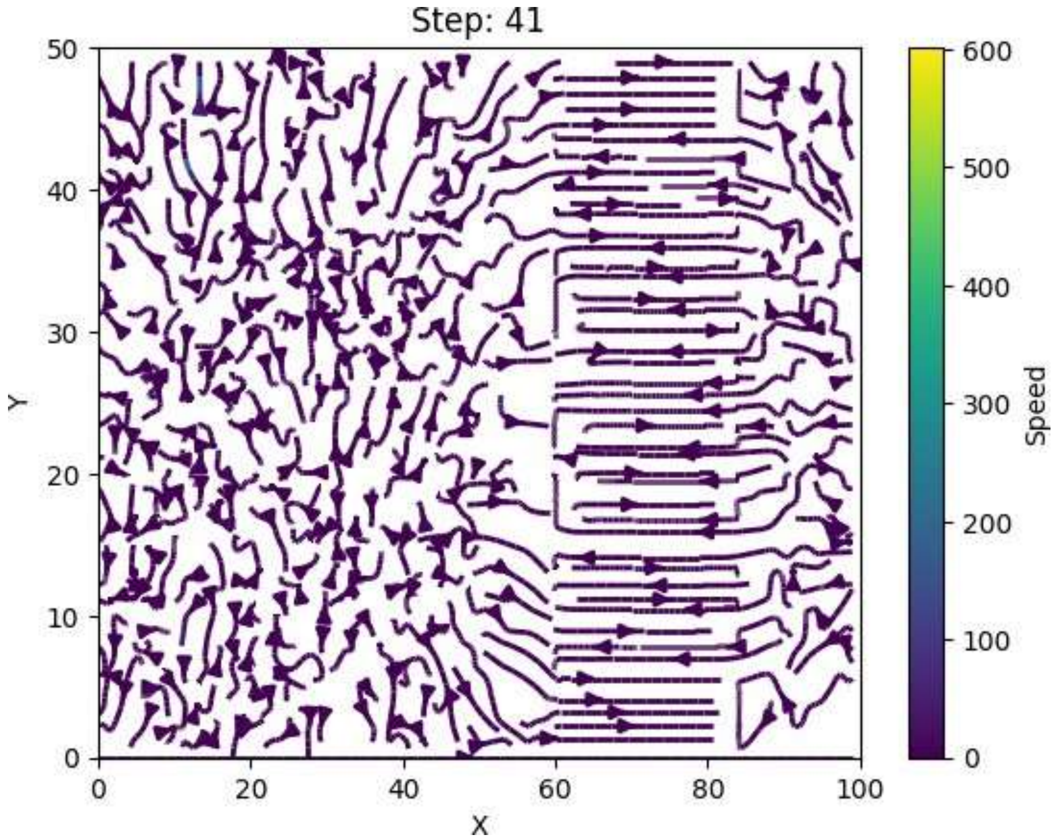
Step: 37
Y
X
Speed


Step: 38
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
200
400
600
800


Step: 39
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
40000
35000
30000
25000
20000
15000
10000
5000
0
Speed

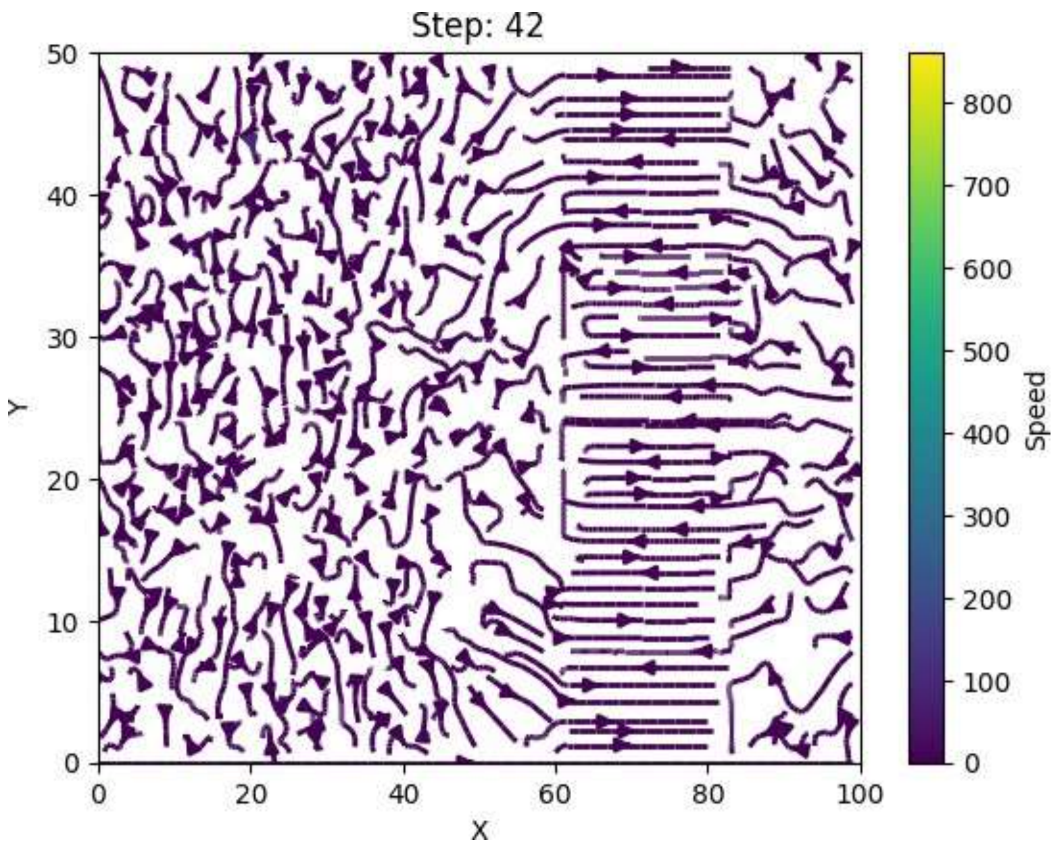


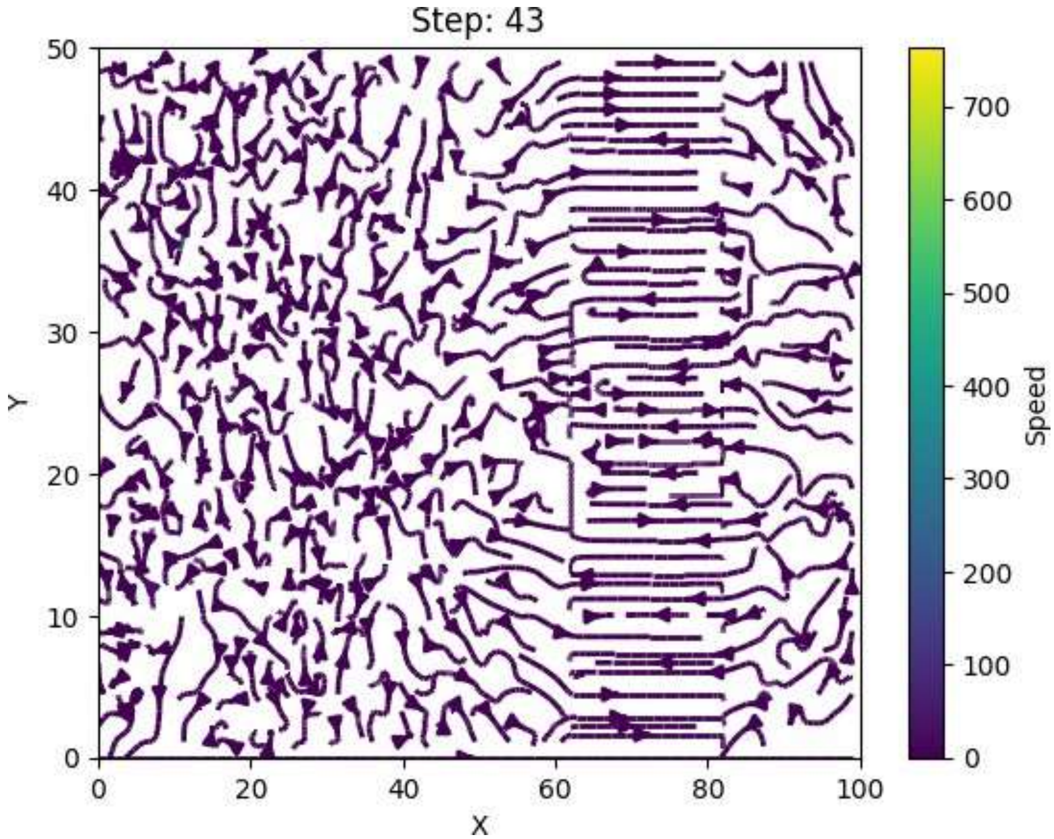
Step: 40
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
50
100
150
200
250
300

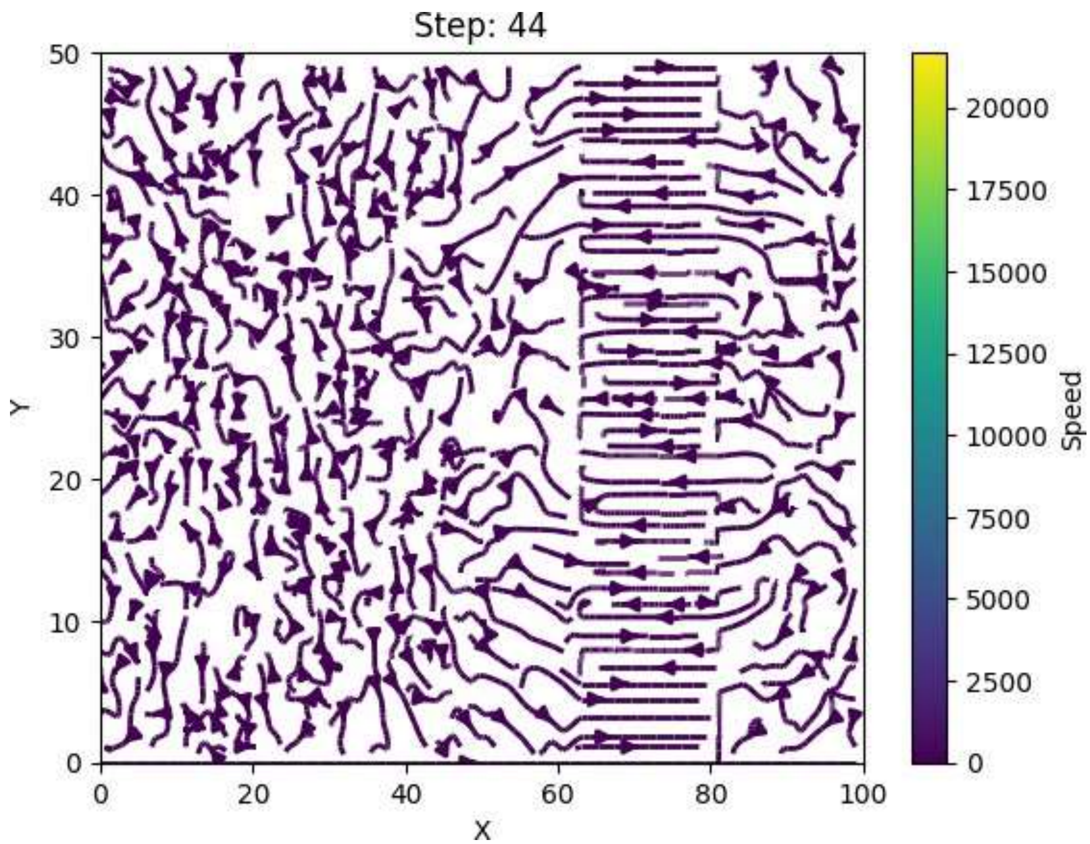


Step: 41
50
40
30
20
10
0
0
20
40
60
80
100
X
Y
600
500
400
300
200
100
0
Speed

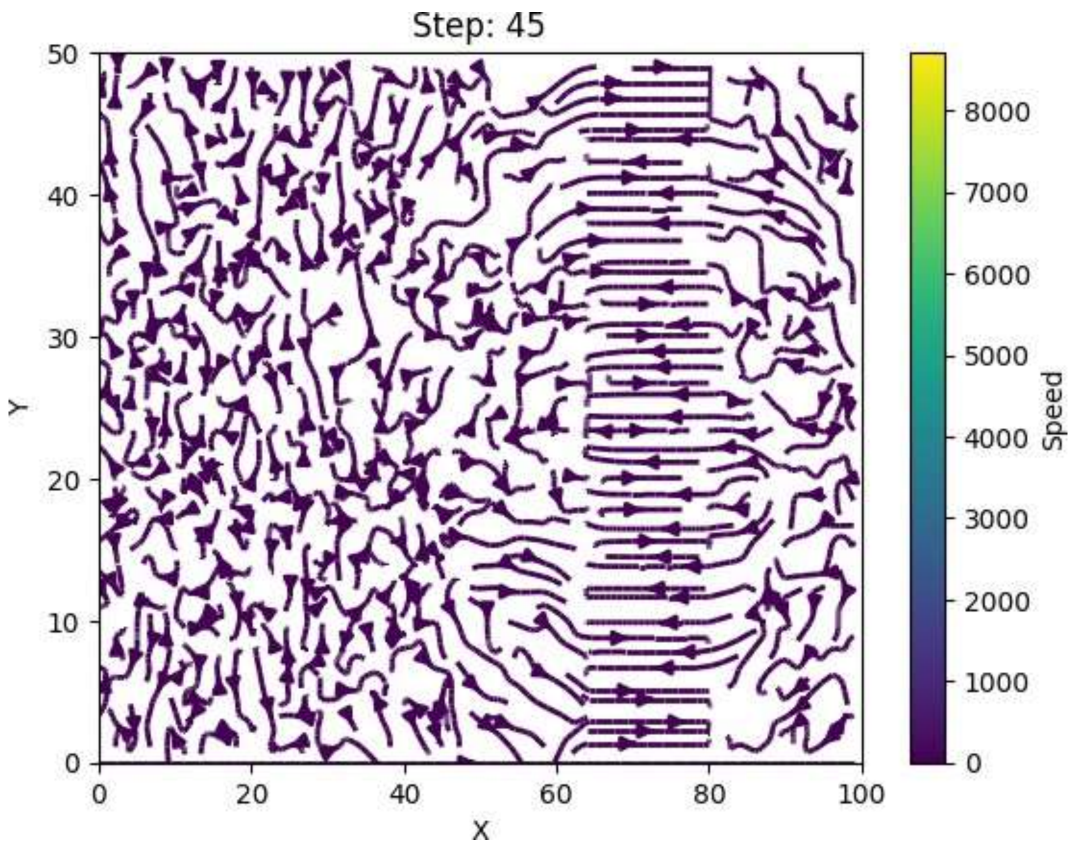
44

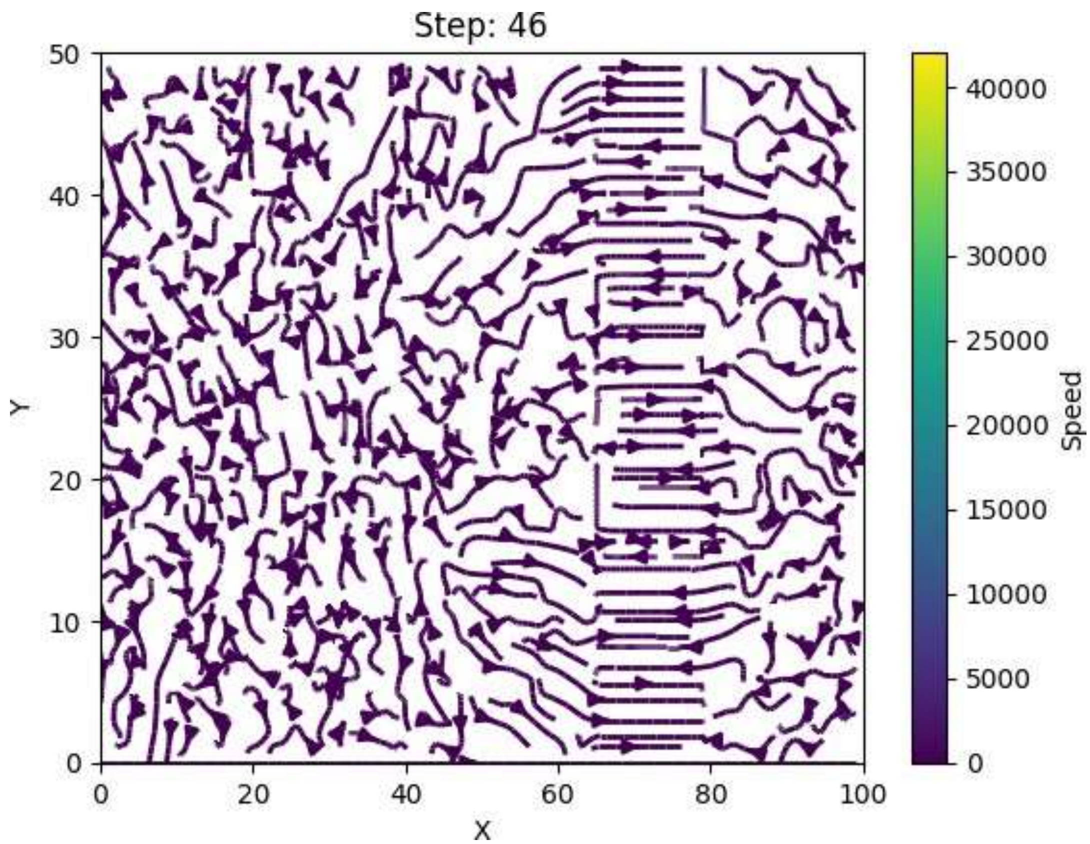

Step: 42
50
40
30
20
10
0
0
20
40
60
80
100
X
Y
800
700
600
500
400
300
200
100
0
Speed

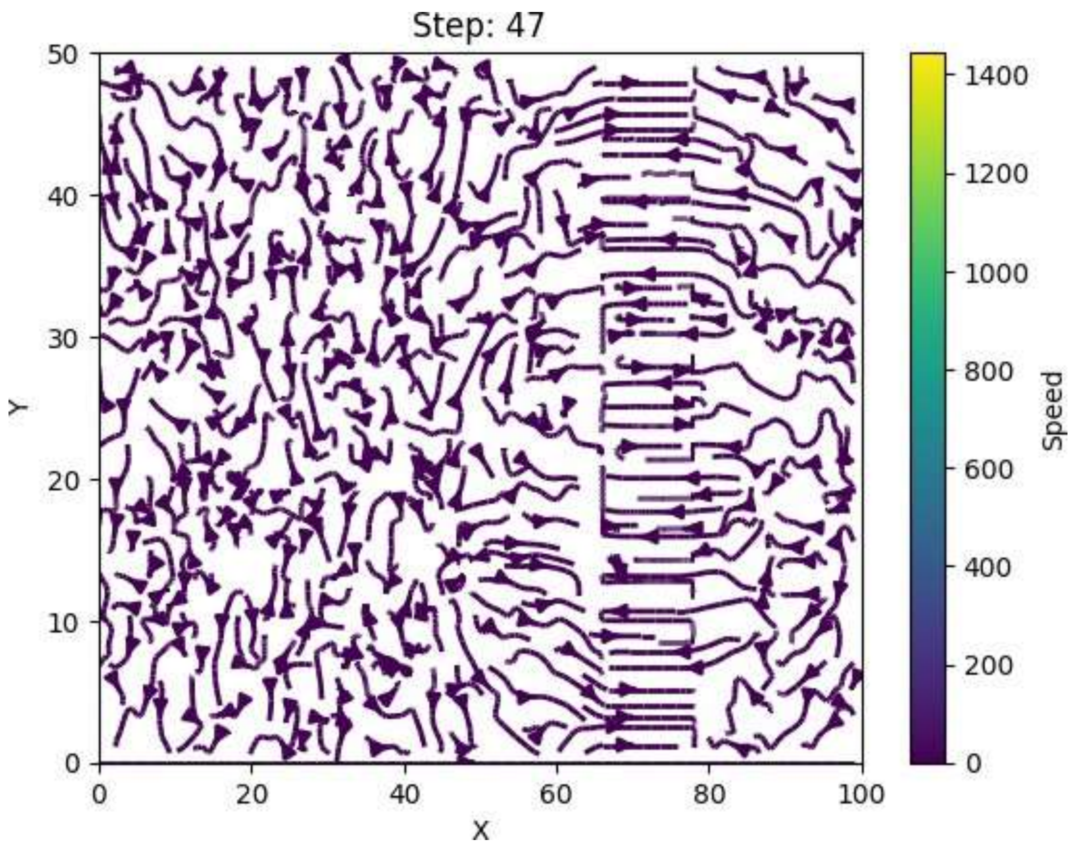

Step: 43
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
100
200
300
400
500
600
700

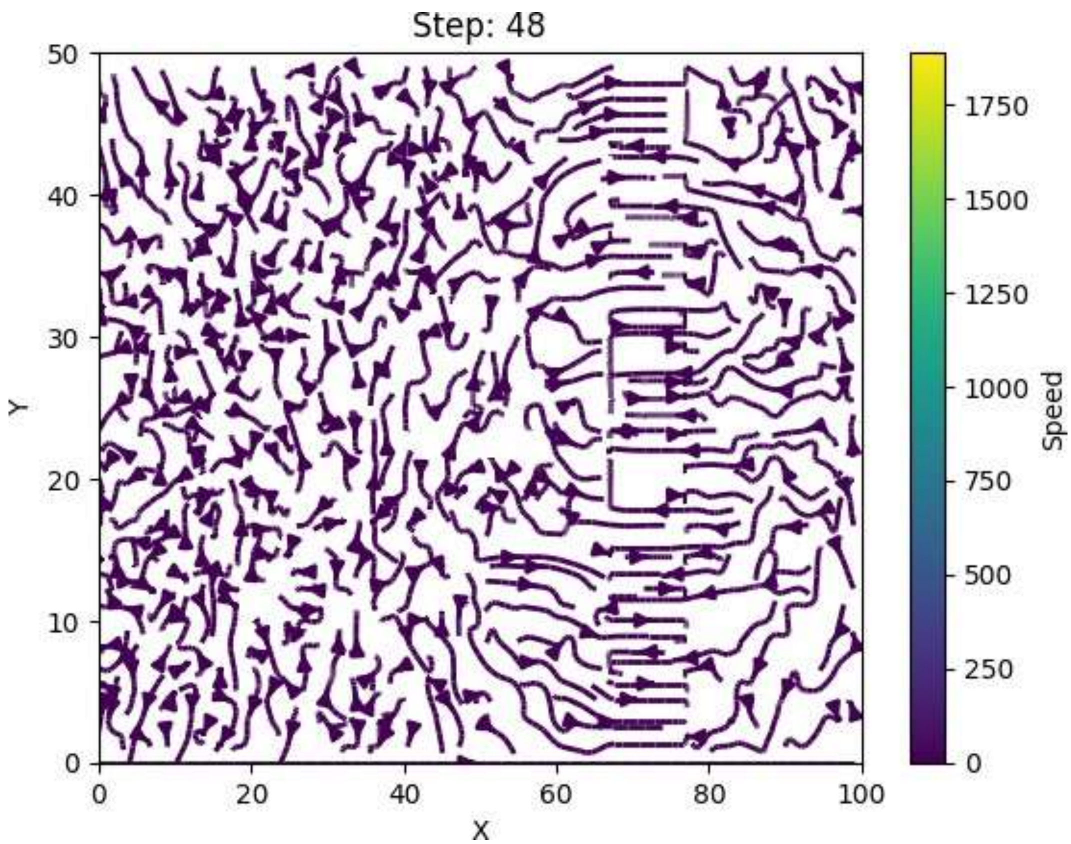


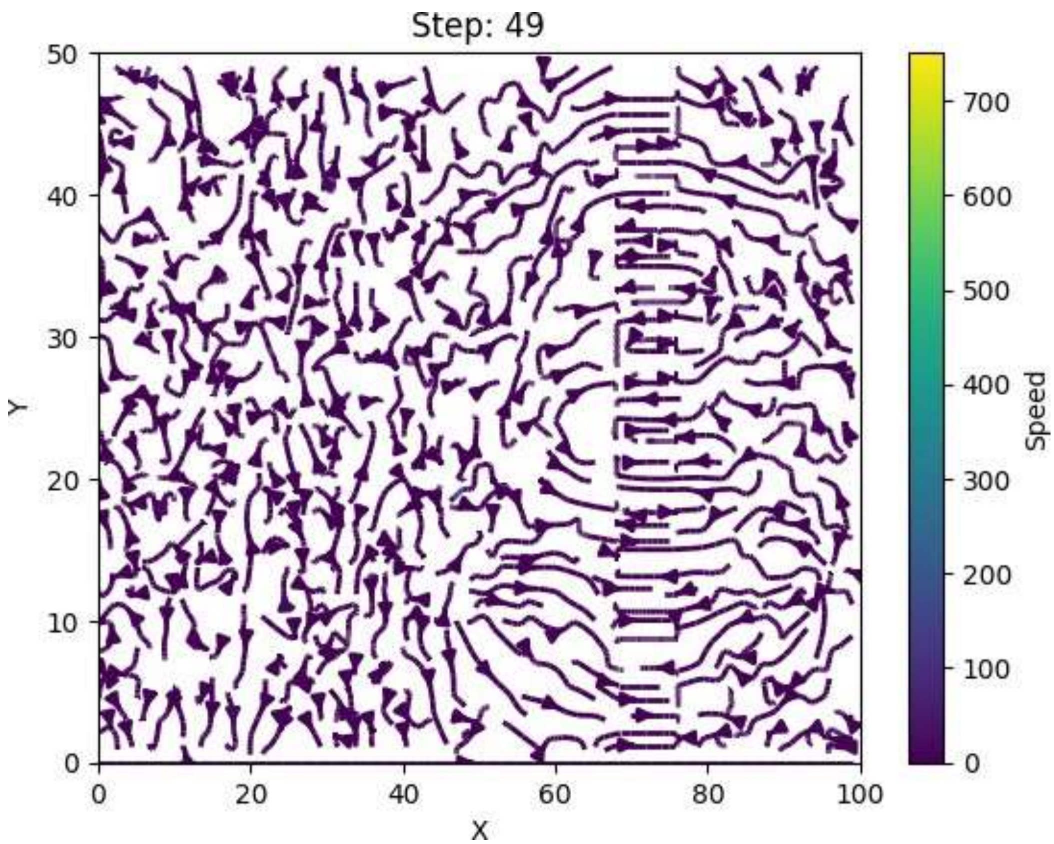
Step: 44
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
20000
17500
15000
12500
10000
7500
5000
2500
0
Speed


Step: 45
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
1000
2000
3000
4000
5000
6000
7000
8000


Step: 46
Y
X
Speed
0
5000
10000
15000
20000
25000
30000
35000
40000


Step: 47
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
1400
1200
1000
800
600
400
200
0
Speed


Step: 48
50
40
30
20
10
0
Y
0
20
40
60
80
100
X
1750
1500
1250
1000
750
500
250
0
Speed


Step: 49
Y
X
Speed
0
10
20
30
40
50
0
20
40
60
80
100
0
100
200
300
400
500
600
700

Program 7: Optimization via Gene Expression Algorithms

Date 17/12/2024
Page

Lab-7

⇒ Optimization via Gene Expression Algorithm

Pseudocode

Step 1 Define optimization problem and objective function

Step 2 Initilization of parameters
set population-size
set no-of-genes
set mutation rate
set crossover-rate
set max-generations

Step 3: Initilize population
For each individual i in population-size:
Generate random genetic sequence
store the genetic sequence in the population

Step 4 Evaluate fitness
For each individual in population
Translate gene into solution
compute fitness function
store the value of individuals

Step 5: Iterate
For generation in range(1, max-generation
a) select individual based on fitness
store it as mating-pool

b) Crossover
For each pair of individual in mating pool
whit propability rate

perform crossover to produce offspring
Add offspring to new-population

c) mutation
For each individual in new population
Randomly alter genes

d) Gene Expression
for each individual in new-population
Translate gene to solution
compute fitness

e) Replacement
Replace old population with new

f) Track best solution

step 6) output best solution.
→ Application: (optimization) / find min value
Input:
population_size = 20
no. of generations = 100
search range = [-10, 10]
crossover rate = 0.8
mutation rate = 0.1
mutation_step = 0.5

Output:
best solution i.e corresponding min value
for $f(x) = x^2 - 2x + 10$
output ⇒ $x = 1.00$ min value = 9.00

```
#minimizing quadratic equation(x^2 - 2x + 10) using Gene Expression Algorithm
print("Name:Sudarshan Komar","USN:1BM22CS291",sep=" ") import

random

# Objective function
def objective_function(x): return
      x**2 - 2*x + 10

# Initialize population
def initialize_population(pop_size, x_min, x_max):
      return [random.uniform(x_min, x_max) for _ in range(pop_size)]

# Evaluate fitness
def evaluate_fitness(population):
      return [objective_function(x) for x in population]

# Selection: Tournament Selection
def select_parents(population, fitness, tournament_size=3): selected
      = []
      for _ in range(len(population)):
            competitors = random.sample(list(zip(population, fitness)),
tournament_size)
            winner = min(competitors, key=lambda x: x[1])
            selected.append(winner[0])
      return selected

# Crossover: Arithmetic crossover
def crossover(parents, crossover_rate=0.8): offspring = []
      for i in range(0, len(parents), 2):
            p1, p2 = parents[i], parents[(i+1) % len(parents)] if
            random.random() < crossover_rate:
                  alpha = random.random()
                  child1 = alpha * p1 + (1 - alpha) * p2 child2 =
                  alpha * p2 + (1 - alpha) * p1
            else:
                  child1, child2 = p1, p2 offspring.extend([child1, child2])
      return offspring

# Mutation: Add small random noise
def mutate(offspring, mutation_rate=0.1, mutation_step=0.5): for i in
      range(len(offspring)):
            if random.random() < mutation_rate:
                  offspring[i] += random.uniform(-mutation_step, mutation_step)
      return offspring
```

```
# Gene Expression Algorithm
def gene_expression_algorithm(pop_size, generations, x_min, x_max):
      population = initialize_population(pop_size, x_min, x_max) best_solution =
      None
      best_fitness = float("inf")

      for gen in range(generations):
            fitness = evaluate_fitness(population) if
            min(fitness) < best_fitness:
                  best_fitness = min(fitness)
                  best_solution = population[fitness.index(best_fitness)]

            parents = select_parents(population, fitness) offspring =
            crossover(parents)
            population = mutate(offspring) return

      best_solution, best_fitness

pop_size = 20
generations = 100
x_min, x_max = -10, 10

best_x, best_fitness = gene_expression_algorithm(pop_size, generations, x_min,
x_max)
print(f"Best solution: x = {best_x:.5f}, Minimum value: f(x) =
{best_fitness:.5f}")
```

```
Name:Sudarshan Komar USN:1BM22CS291
Best solution: x = 1.00000, Minimum value: f(x) = 9.00000
```