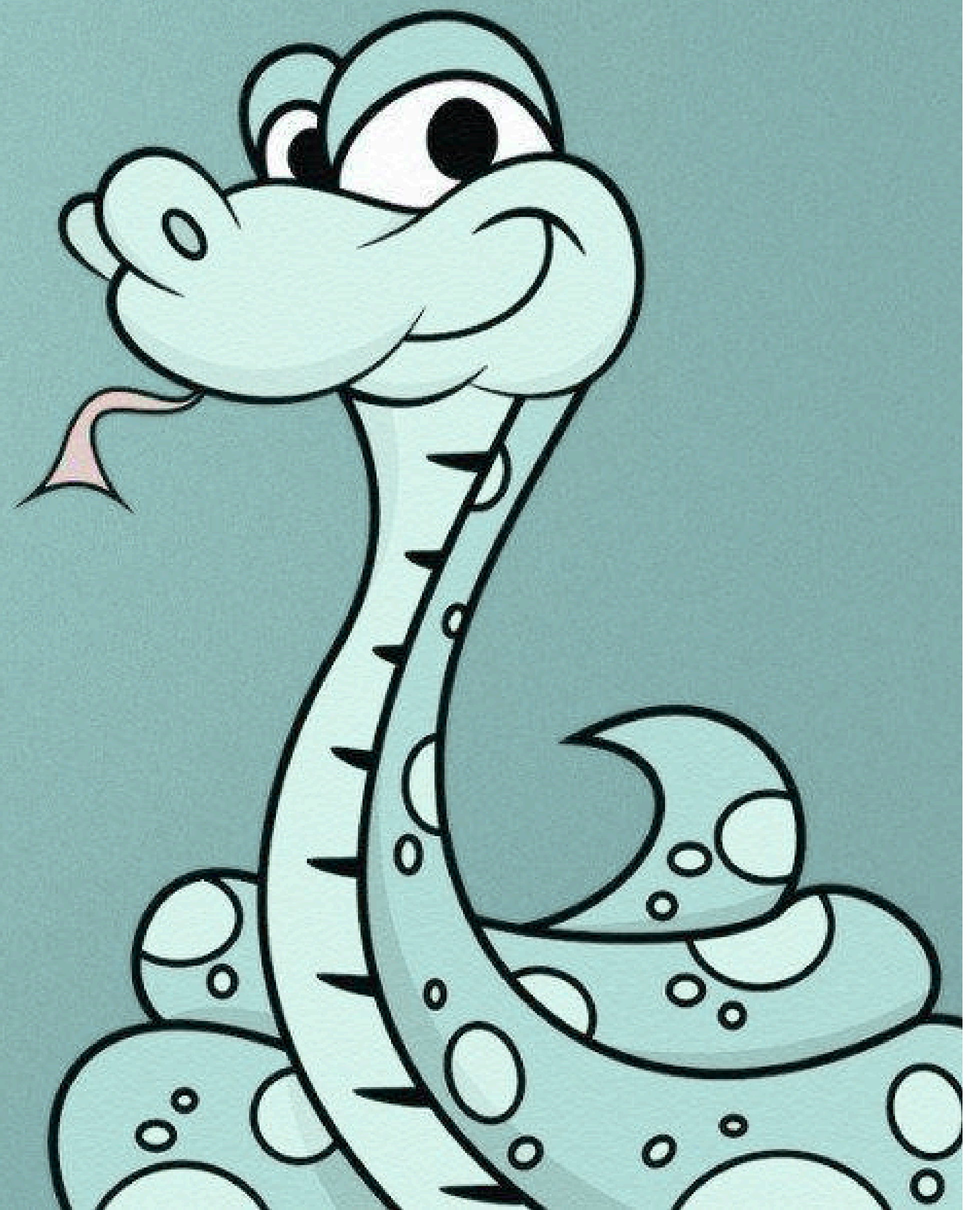
SNAKE GAME

PRESENTED BY SAAD THAPLAWALA AND SHARJEEL CHANDNA



Origin of the creative idea

Nostalgia meets optimization

Inspired by classic Snake game frustrations:

- Human players rarely reach theoretical score limits
- Inefficient manual trial-and-error learning

Goal: Create AI that outperforms human capabilities

Innovation:

- Sequential Algorithm Testing: EA and PSO for Strategy Optimization
- Headless simulation enables rapid training cycles

The AI Challenge

Teaching an AI to outperform humans in Snake requires solving three critical challenges:

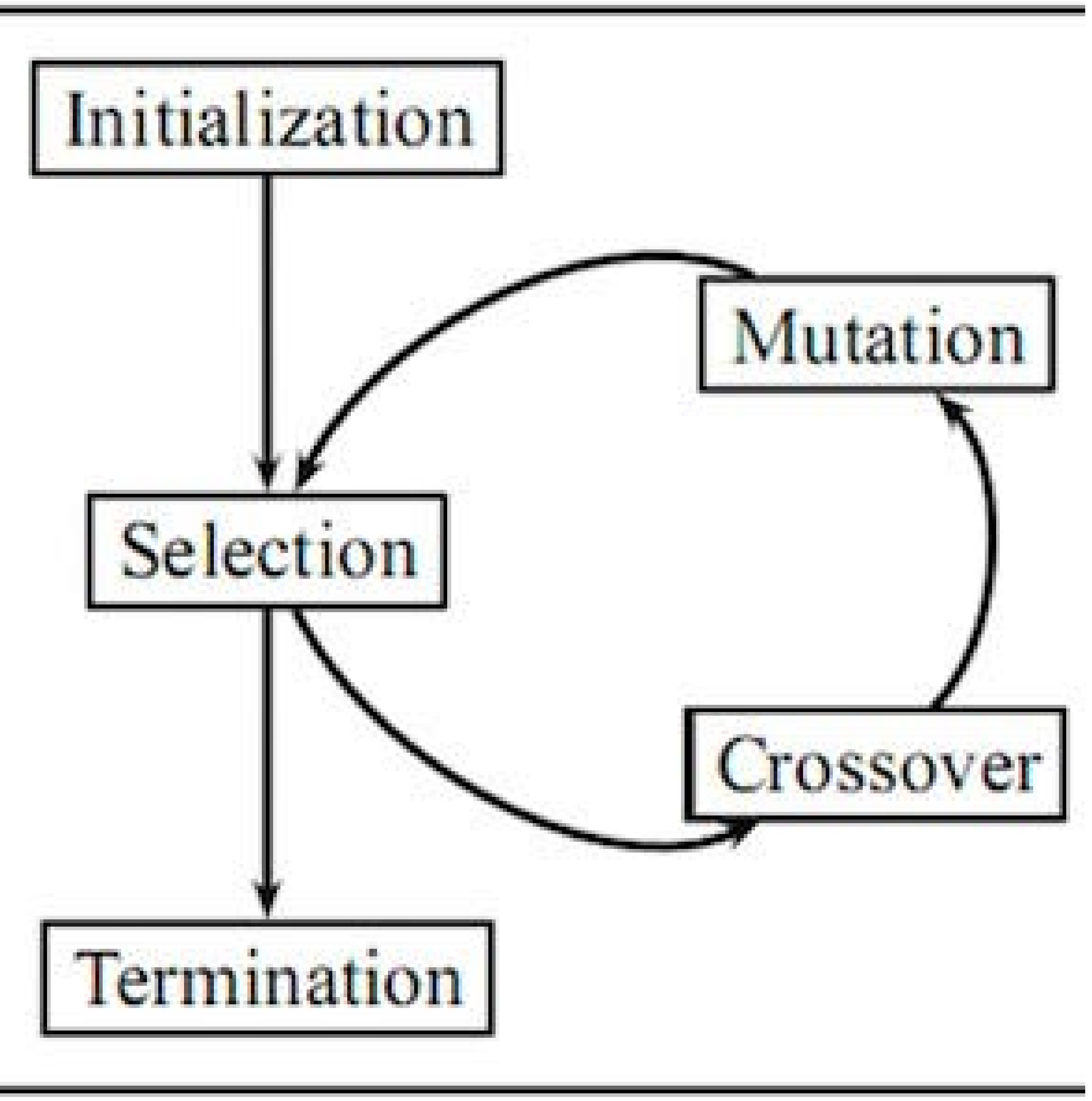
1. Collision Avoidance: Navigating walls and growing body with zero margin for error.
2. Apple Targeting: Balancing immediate reward vs long-term path efficiency.
3. Space Optimization: Maximizing coverage in finite grid (500×500 pixels).

Training autonomous snake agents

Comparing EA vs PSO

Goal: Optimal Path Finding

Evolutionary Algorithm



An Evolutionary Algorithm (EA) is a population-based optimization technique inspired by biological evolution. It mimics natural selection to solve complex problems by evolving solutions over generations.

Evolutionary Algorithms (EA) mimic Darwinian natural selection. Just as species evolve through reproduction, mutation, and survival pressures, EA iteratively refines solutions by:

- Selecting high-performing "parent" solutions,
- Combining their traits via crossover,
- Introducing diversity through random mutations.

EA Snake implementation

The Evolutionary Algorithm (EA) drives snake behavior by evolving neural network controllers through genetic operations. Each "chromosome" represents a complete set of weights for the neural network that maps game states to actions.

Neural Network:

Input: 8 normalized features (apple distance/angle, dangers, position)

Hidden Layers: 12 → 8 tanh-activated neurons (Xavier initialization)

Output: 3 actions (straight/left/right)

Genetic Parameters:

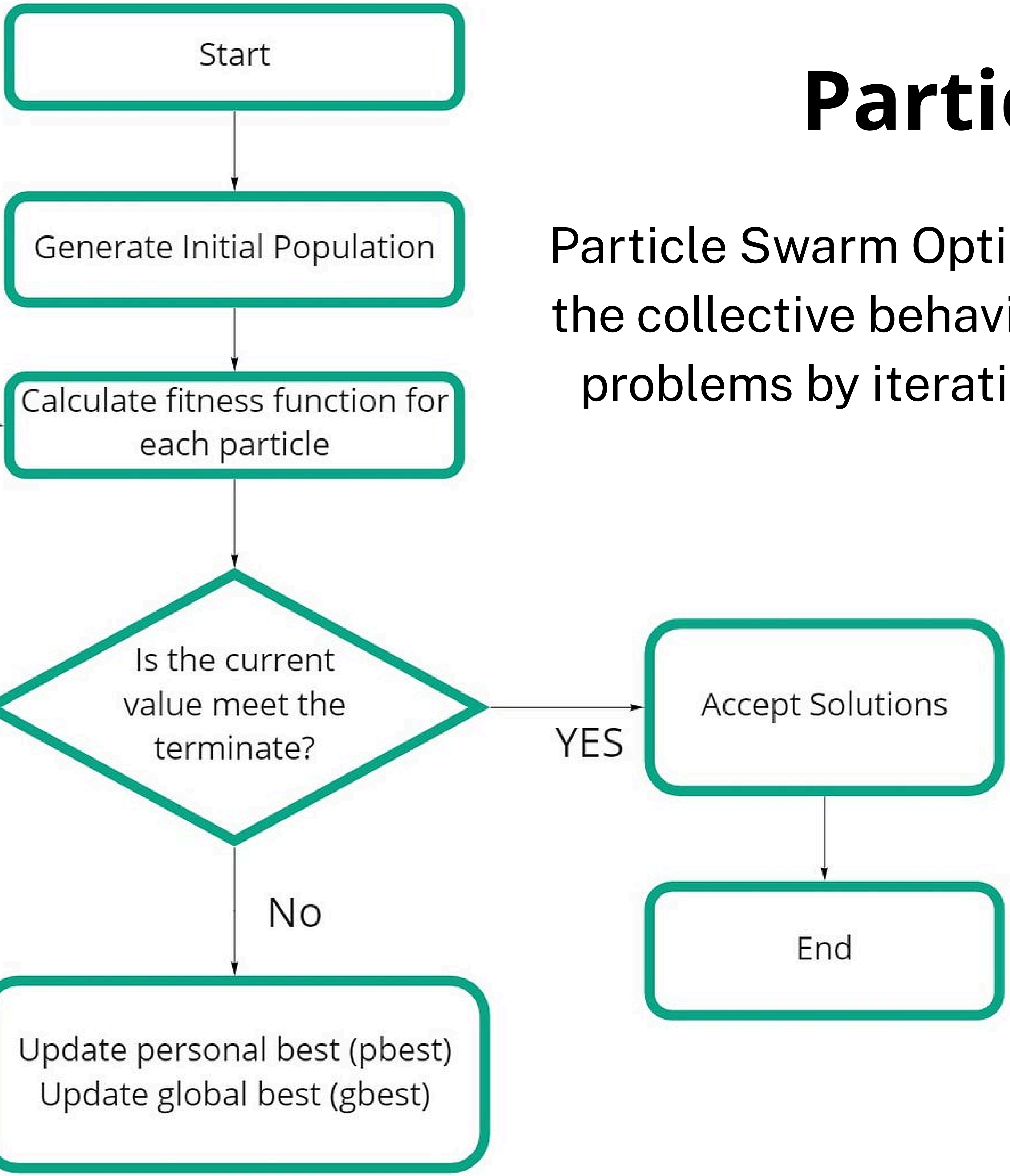
- Crossover rates: 0.2-1.0 (configurable)
- Mutation rates: 0.01-0.1 (adaptive Gaussian)
- Selection: Tournament (size=3) favors high-fitness snakes

Behavior Emergence:

Gen 1: Random movements (avg. 2 apples)

Gen 50: Basic apple-seeking (avg. 12 apples)

Gen 200: Collision avoidance + path optimization (peak 42 apples)



Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a computational method inspired by the collective behavior of bird flocks or fish schools. It solves optimization problems by iteratively improving candidate solutions (called particles) based on social collaboration.

Each particle adjusts its trajectory based on:

1. Its personal best discovery (pBest),
2. The swarm's global best solution (gBest),
3. Its own momentum (inertia).

Particles continuously update their positions using a velocity vector that balances individual experience with collective wisdom, causing the entire swarm to "surf" toward optimal regions.

PSO Snake implementation

Particle Swarm Optimization (PSO) treats each neural network as a "particle" that collaboratively explores weight space. Particles adjust trajectories based on personal best (pBest) and swarm best (gBest) solutions.

Optimization Process:

1. Iteration 1: Particles explore randomly (avg. 3 apples)
2. Iteration 15: Rapid convergence via gBest sharing (avg. 22 apples)
3. Iteration 60: Performance plateau (peak 38 apples)

Particle Representation:

Position: 216 neural weights (same as EA chromosome)

Velocity: Weight adjustment direction/magnitude

Hyperparameters:

Inertia (w): 0.2-1.0 (higher = more exploration)

Cognitive (c1): 1.0-5.0 (self-trust)

Social (c2): 1.0-5.0 (swarm-trust)

Boundary Handling: Weights constrained to [-2, 2]

EA vs PSO

Metric	EA	PSO
Optimization	Genetic operations	Velocity updates
Convergence	Slow, steady improvement	Rapid early gains
Diversity	High (mutation)	Guided by gBest
Best Config	Pop=130, crossover=0.1, mut=0.05, crossover type = 1 point	Pop=180, w=0.6, c1=1.0, c2=3.0

GUI

Fitness Function

```
 # Base Score (Quadratic Reward)
f score == 0:
   fitness = steps * 0.1  # Small survival bonus
lse:
   fitness = (score ** 2) * 100 + steps * 0.5  # Apples heavily re

Apple Proximity Tracking
urr_dist = distance_to_apple()
f prev_distance and curr_dist < prev_distance:
   fitness += 5  # Reward approaching apple
lse:
   fitness -= 2   # Penalize moving away

Wall Avoidance Penalty
f distance_to_wall() < 30:
   fitness -= 10

Efficiency Bonus (Post-Game)
f score > 0:
   fitness += (score ** 2) * 50  # Apple bonus
   fitness += max(0, 100 - direction_changes) * score  # Smooth pa

Early Termination Penalty
f steps < 100:
   fitness *= 0.5
```

Key Components

- Apple Acquisition:
  Quadratic reward ($score^2 \times 100$) + small bonus (+5/step) when moving toward apples.
- Survival:
  Rewards for longevity (steps × 0.5) and minimal survival bonus (steps × 0.1).
- Efficiency:
  Penalizes turns (-0.5/change), rewards straight paths (+2/step).
- Danger Avoidance:
  -10 near walls; fitness halved for short runs (<100 steps).
- Exploration:
  Distance-based rewards promote purposeful movement; input adapts in extended modes.