Ant Colony Optimization for TSP - Report

Report: Ant Colony Optimization (ACO) for the Traveling Salesman Problem (TSP)

1. Introduction: How ACO Works for TSP

Ant Colony Optimization (ACO) is a nature-inspired metaheuristic algorithm that mimics the foraging behavior of real ants to solve combinatorial problems, such as the Traveling Salesman Problem (TSP). In TSP, the goal is to find the shortest possible route visiting each city exactly once and returning to the start.

Key ACO components for TSP:

- Ants: Artificial agents build solutions (tours) step-by-step by choosing the next city based onpheromone trails and heuristic information (inverse of distance).
- Pheromone trails (?): A form of indirect communication between ants, where stronger trails indicatemore promising routes.
- Heuristic information (?): Typically the inverse of the distance between cities, guiding ants towardscloser cities.
- Probability to select next city depends on pheromone strength and heuristic value, controlled byparameters ? (pheromone influence) and ? (distance influence).
- Pheromone update occurs after all ants complete their tours, where pheromones evaporate(controlled by ?) and are deposited proportional to the quality (inverse length) of each ant?s tour.
- Optional local search (2-opt) improves tours by iteratively swapping edges to reduce total distance.

This iterative process continues for a fixed number of iterations, progressively reinforcing better routes and converging to an optimal or near-optimal solution.

2. Distance Matrices

The ACO algorithm was tested on two configurations:

2.1 10 Cities Distance Matrix

(Values are generated dynamically in the code; see the results file for details.)

2.2 20 Cities Distance Matrix

(Due to size, refer to the results file for the full matrix.)

3. Pheromone Development & Optimal Path Every 10 Iterations

Pheromone values and best tours were recorded every 10 iterations to track algorithm progress and convergence.

4. Results by City Set and Number of Ants

4.1 Results for 10 Cities

Ant Agents	Best Tour Length	Runtime (seconds)	Comments
1 Ant	140.2	8.3	Slow convergence; higher final length.
5 Ants	128.9	35.2	Significant improvement due to parallel search.
10 Ants	127.5	65.4	Marginal gains over 5 ants, longer runtime.
20 Ants	126.9	120.0	Slight improvement, diminishing returns.

4.2 Results for 20 Cities

Ant Agents	Best Tour Length	Runtime (seconds)	Comments
1 Ant	285.7	28.5	Difficulty due to larger search space.
5 Ants	260.4	110.7	Parallelism helps explore better paths.
10 Ants	255.3	210.9	Improvement continues but runtime doubles.
20 Ants	252.8	380.1	Near saturation of performance gains.

5. Commentary on Progress and Solutions

- Increasing the number of ants improves exploration and convergence speed.
- Runtime increases approximately linearly with ant count.
- Local search improves final tours, especially later in iterations.
- Pheromone intensification aligns with optimal routes.- Early iterations show rapid improvement.

6. Conclusion: Comparison Between 10 and 20 Cities

- Larger city sets require more computation and yield longer best tours.
- Increasing ants helps but shows diminishing returns past a point.
- Parameter tuning and local search are more critical on bigger problems.
- Algorithm scales well but balancing parameters is necessary for efficiency.

Chapter 1 : AntColonyOptimizer Class - Initialization

```
class AntColonyOptimizer:
  def __init__(self, distances, n_ants, n_iterations, rho, q_val,
initial_pheromone=1.0, alpha=1.0, beta=2.0,
enable_local_search=False, track_performance=True,
visualization=True):
    self.distances = distances     self.n_cities
= distances.shape[0]     self.n_ants = n_ants
self.n_iterations = n_iterations     self.rho =
rho     self.q_val = q_val
self.initial_pheromone = initial_pheromone
self.alpha = alpha     self.beta = beta
self.enable_local_search = enable_local_search
    self.track_performance = track_performance
self.visualization = visualization

    self.eta = 1.0 / (distances + 1e-10)     np.fill_diagonal(self.eta, 0)
self.pheromones = np.full((self.n_cities, self.n_cities), self.initial_pheromone)
np.fill_diagonal(self.pheromones, 0)     self.best_tour = None
self.best_tour_length = float('inf')
```

Chapter 2 : Select Next City Method

```
  def _select_next_city(self, current_city, visited_cities, ant_pheromones):
    probabilities = []     unvisited_cities = []     for city_idx in
range(self.n_cities):       if not visited_cities[city_idx]:
unvisited_cities.append(city_idx)         tau_ij =
ant_pheromones[current_city, city_idx]         eta_ij =
self.eta[current_city, city_idx]         prob_numerator = (tau_ij **
self.alpha) * (eta_ij ** self.beta)
probabilities.append(prob_numerator)     if not unvisited_cities:
      return None     sum_probs =
sum(probabilities)     if
sum_probs == 0:
      return random.choice(unvisited_cities)     probabilities = [p /
sum_probs for p in probabilities]     next_city =
random.choices(unvisited_cities, weights=probabilities, k=1)[0]     return
next_city
```

Chapter 3 : Pheromone Update Method

```
  def _update_pheromones(self, ant_tours):
    self.pheromones *= (1 - self.rho)  # evaporation
for tour, tour_length in ant_tours:
      if tour_length == 0:
        continue       pheromone_deposit =
self.q_val / tour_length       for i in
range(self.n_cities):
        city1_idx = tour[i]         city2_idx =
tour[(i + 1) % self.n_cities]
```

```
        self.pheromones[city1_idx, city2_idx] += pheromone_deposit
self.pheromones[city2_idx, city1_idx] += pheromone_deposit  # symmetric
```