Team - 5
Charchit Gupta
Aryan Agarwal
Utkarsh Mishra
Lokesh Gautham

Algorithmic Methods to Approach the Travelling Salesman Problem

Travelling Salesman Problem

The Travelling Salesman Problem deals with finding an efficient solution for a person to take, given a list of specific cities to cover exactly once. It is an “NP-hard” problem. NP-hard problems are at least as hard as any other NP-problem (non-deterministic in polynomial time). In this project, we have arrived at approaches to tackle the travelling salesman problem and compare them.

Motivation

We wanted to explore a common problem and existing ways to tackle it. We thought TSP is a perfect problem to work on and hence we chose this topic.

Algorithms

Particle Swarm Optimization

Introduction

Swarm Intelligence (SI) refers to a type of AI system based on collective behaviour of decentralised and self-organised systems. SI systems are typically made of a population of simple agents interacting within themselves and with the surroundings. SI systems are found in nature, such as ant colonies, bird flocking, etc. NASA uses SI based systems for planetary mapping.

Particle Swarm Optimisation (PSO) was first described in 1995 by James Kennedy and Russell C. Eberhart. It is an optimisation technique that focuses on finding values/solutions that minimise or maximise the objective function while satisfying the constraints. In PSO, the problem is plotted in a space and seeded with an initial velocity and communication channels between particles.

PSO for Travelling Salesman Problem

STEP 1: PSO is initialised with multiple route configurations. Some of them are random and some are greedy based on the starting index. We define pbest and pbest_cost, that is, the best route possible for each configuration that we started with and the cost associated with it. We also define gbest and gbest_cost, that is, the best route among all routes and cost associated with it. Here each configuration is similar to a particle in PSO and multiple configurations correspond to a swarm.

STEP 2: In every iteration, we try to change each route/configuration according to pbest with pbest_probability and according to gbest with gbest_probability by swapping cities in the route. The probability determines the speed of convergence. The probability cannot be too large or too small. Then we compare pbest_cost with cost of the current/new route and then change pbest and pbest_cost accordingly. Finally we change gbest by comparing pbest for all particles.

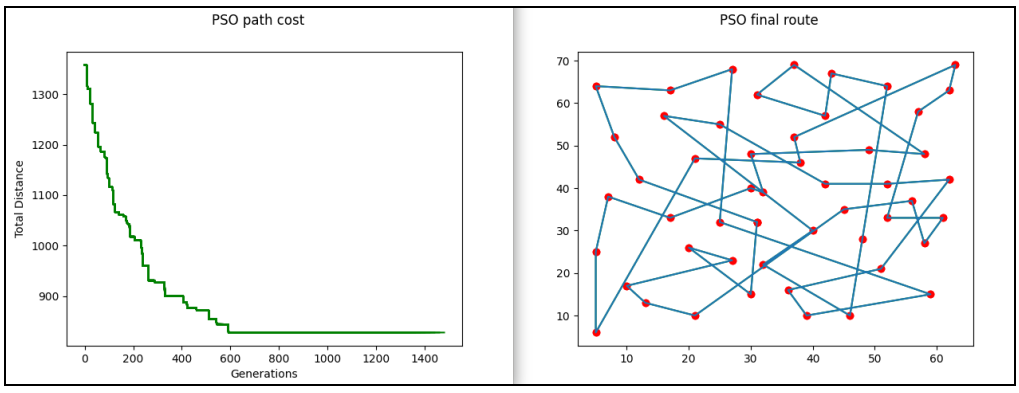
STEP 3: We keep repeating STEP 2 until there is some convergence or according to the number of iterations.

The parameters to tune in this algorithm are the starting index in the initial greedy configuration, the number of initial random configurations, pbest_probability, gbest_probability and iterations.

Results

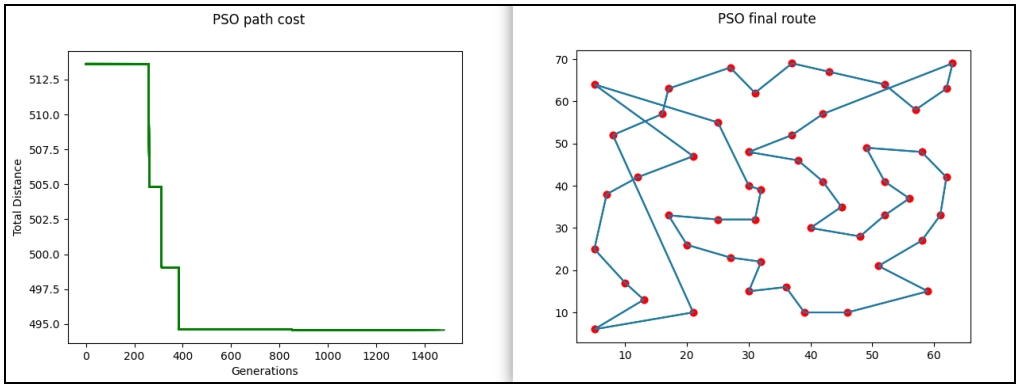
We have used the datasets available in the TSP online library.

#Cities = 51, #Initial greedy configurations = 0, #Initial random configurations = 255, pbest_probabilty = 0.85, gbest_probabiltiy = 0.02, #Iterations = 1500



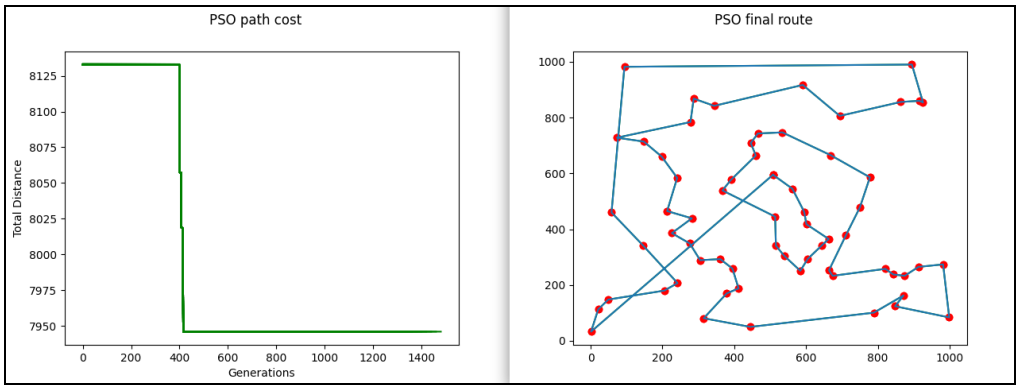
Initial path cost = 1357.89, Final path cost = 828.07

#Cities = 51, #Initial greedy configurations = 5, #Initial random configurations = 255, pbest_probabilty = 0.85, gbest_probabiltiy = 0.02, #Iterations = 1500



Initial path cost = 513.6, Final path cost = 494.57

#Cities = 64, #Initial greedy configurations = 6, #Initial random configurations = 320, pbest_probabilty = 0.85, gbest_probabiltiy = 0.02, #Iterations = 1500



Initial path cost = 8132.9, Final path cost = 7945.91

Time Complexity

The time complexity for this algorithm is O(#iterations x #cities x #initial configurations).

References

https://www.slideshare.net/khashe62/practical-swarm-optimization-pso

Simulated Annealing

Introduction

Simulated annealing (**SA**) is a probabilistic technique for approximating the global optimum of a given function. It is a random-search technique which exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system. It forms the basis of an optimisation technique for combinatorial and other problems.
Specifically, it is a metaheuristic (a **metaheuristic** is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity) to approximate global optimization in a large search space for an optimization problem. It is often used when the search space is discrete (for example the traveling salesman problem).
For problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time, simulated annealing may be preferable to exact algorithms such as gradient descent or branch and bound.

Simulated annealing is based on metallurgical practices by which a material is heated to a high temperature and cooled. At high temperatures, atoms may shift unpredictably, often eliminating impurities as the material cools into a pure crystal. This is replicated via the simulated annealing optimization algorithm, with the energy state corresponding to the current solution.

Simulated annealing can be used for very hard computational optimization problems where exact algorithms fail; even though it usually achieves an approximate solution to the global minimum, it could be enough for many practical problems.

Logic

In our case, we don’t necessarily need to find a strictly optimal value — finding a near-optimal value would satisfy our goal. In simulated annealing we introduce a degree

of stochasticity, potentially shifting from a better solution to a worse one, in an attempt to escape local minima and converge to a value closer to the global optimum.

In this algorithm, we define an initial temperature, often set as 1, and a minimum temperature, on the order of 10^-4. The current temperature is multiplied by some fraction alpha and thus decreased until it reaches the minimum temperature. For each distinct temperature value, we run the core optimization routine a fixed number of times. The optimization routine consists of finding a neighboring solution and accepting it with probability $e^{f(c) - f(n)}$ where c is the current solution and n is the neighboring solution. A neighboring solution is found by applying a slight perturbation to the current solution. This randomness is useful to escape the common pitfall of optimization heuristics — getting trapped in local minima. By potentially accepting a less optimal solution than we currently have, and accepting it with probability inverse to the increase in cost, the algorithm is more likely to converge near the global optimum. Designing a neighbor function is quite tricky and must be done on a case by case basis.

Step 1: Initialize - Start with the initial route and cost obtained from the greedy algorithm..

Step 2: Move – Choose two random cities and reverse the sequence in which they are being approached.

Step 3: Calculate score - calculate the change in the cost due to the move made.

Step 4: Choose - Depending on the change in cost, accept or reject the move. The prob of acceptance depends on the current “temperature”.

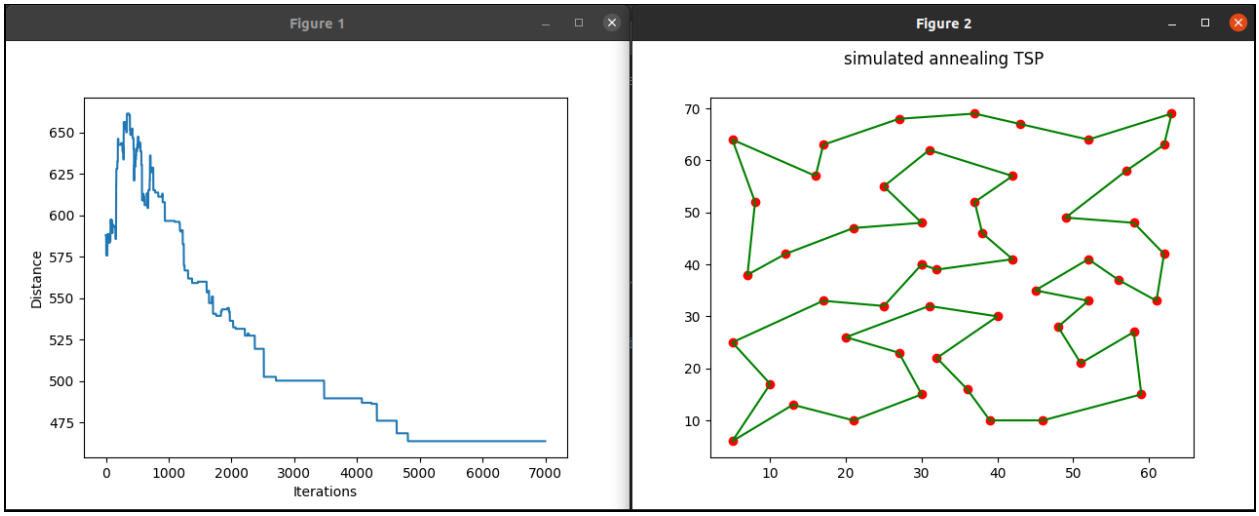
Step 5: Update and repeat-Update the temperature value by lowering the temperature. Go back to Step 2. Repeat for a fixed number of iterations.

The process is done until “Freezing Point” is reached.

“Temperature” : Travelling Cost, “Freezing Point” : Global Minimum

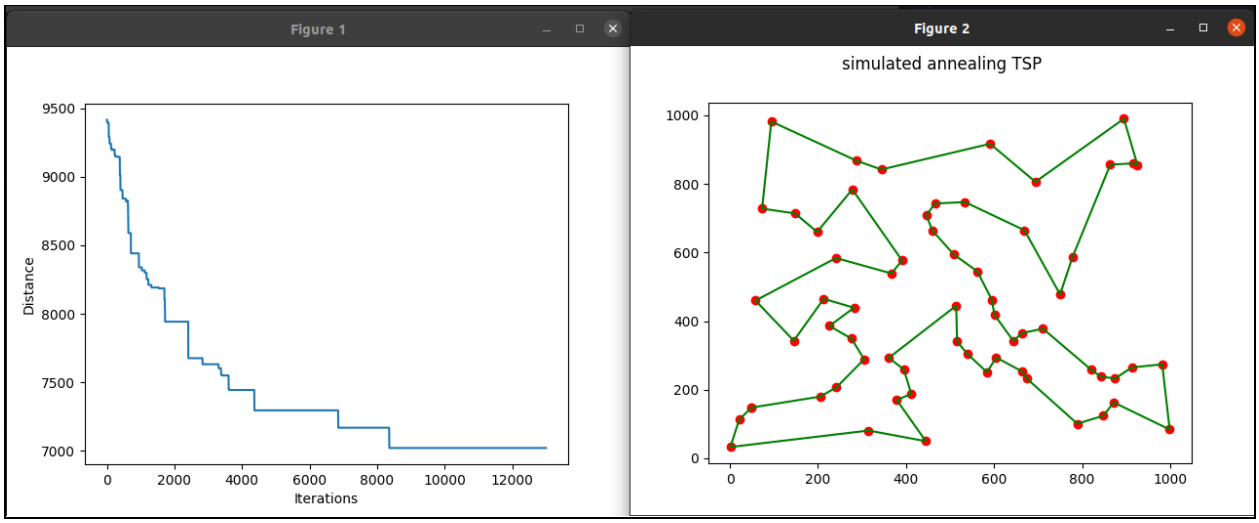
Results

#Cities = 51, #iterations = 7000



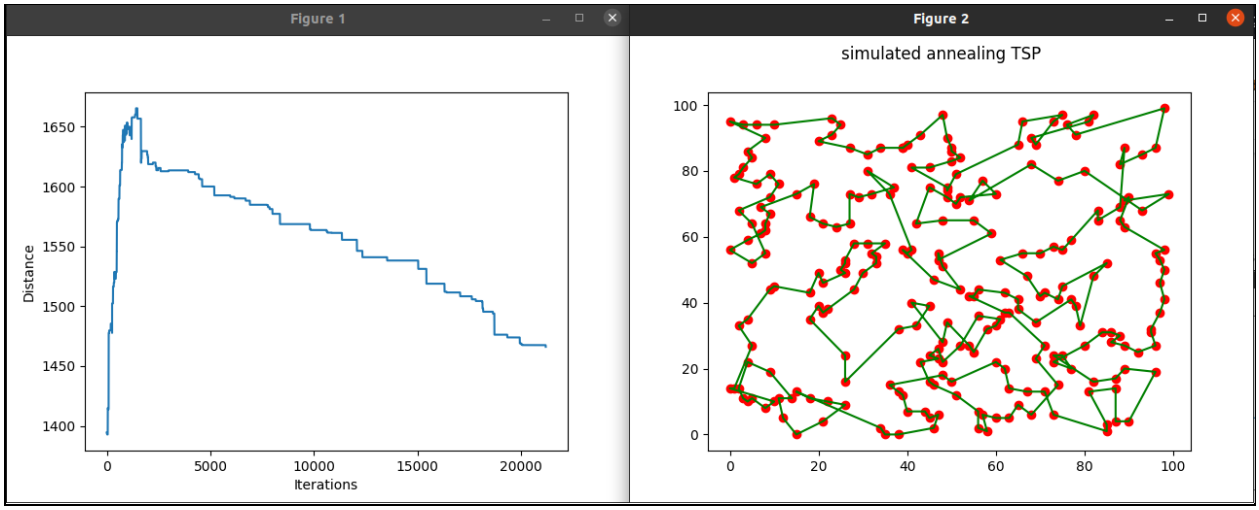
Greedy fitness = 588.074 Best fitness = 463.634

#Cities = 64, #iterations = 13000



Greedy fitness = 9415.388 Best fitness = 7021.015

#Cities = 256, #iterations = 21183



Greedy fitness = 1394.401 Best fitness = 1392.777

References

https://www.geeksforgeeks.org/simulated-annealing/

https://www.slideshare.net/udaywankar/optimization-simulated-annealing-80891498

Genetic Algorithm

Introduction

Genetic algorithms are a class of algorithms to solve optimization problems in the “nature’s way”. It is inspired by Darwin’s theory of Natural Evolution. Just like how only the fittest organisms survive and contribute to the next generation, the fittest (good) solutions (for a problem) survive and take part in evolution and mutation, leading to better-optimized solutions.

The Flow of the algorithm

The Travelling Salesman Problem looks at optimizing the best route to cover all the cities. We can make some important analogies of elements of this problem with the natural world.

- **A “route” can be seen as a chromosome**
- **A set of possible routes can be seen as a “population” of chromosomes**
- **A chromosome constitutes of genes, therefore the “cities” which are making a route can be seen as “genes”**

- **Two chromosomes (parents) creating a new offspring (child) is called reproduction**
- **Fitness is a metric deciding how good a route is. In our implementation, it is 1/(total edge cost from that route).**
- **mating_pool is the set of "fit" chromosomes that are ready for creating the next generation of chromosomes.**
- **Mutation, is the process of small and random re-ordering of the genes in the chromosome, that is, re-ordering of cities in the routes.**
- **Elitism is a way in which the ideal fitters in a generation are directly taken to the next generation. (The merit being fitness)**

Steps

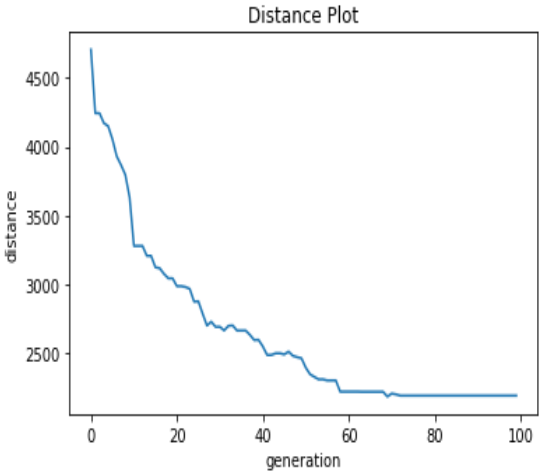
1. **Initializing a population of routes. We have initialized a population of size 100.**
2. **The next step is to rank the chromosomes in the populations based on their fitness.**
3. **Passing the population to a selection phase, where the best chromosomes get a chance to take part in the breeding.**
4. **Breeding by considering elitism.**
5. **Once the children have been created, they pass through a random mutation stage.**
6. **Now we have successfully created a new generation of chromosomes from the old generation. We repeat the entire process to simulate multiple generations.**

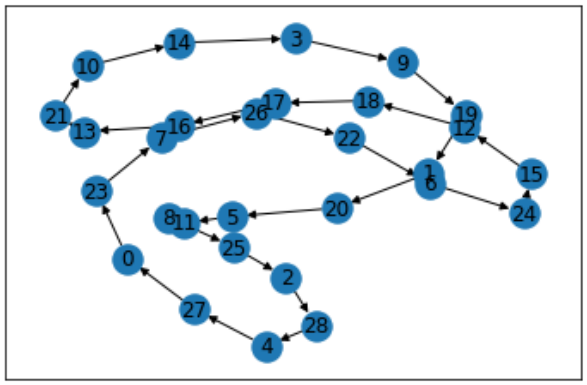
Results

We have performed genetic simulations on the bayes29 dataset which is a collection of 29 cities. We are given the distance matrix between the 29 cities.

The initial population had the best distance (the complete distance of the route) as **4705** units

After **100** generations of simulation, with a mutation rate of **0.01** and 20 elites every





generation, we finally arrive at a population with the best distance as **2195 units.**

The best distance curve wrt generations and the best order obtained.

Time Complexity

Time complexity is O(g(nm + nm + n)), where g is the number of generations, n is the population size and m is the size of individual chromosomes.

Ant Colony Optimization Algorithm

Introduction

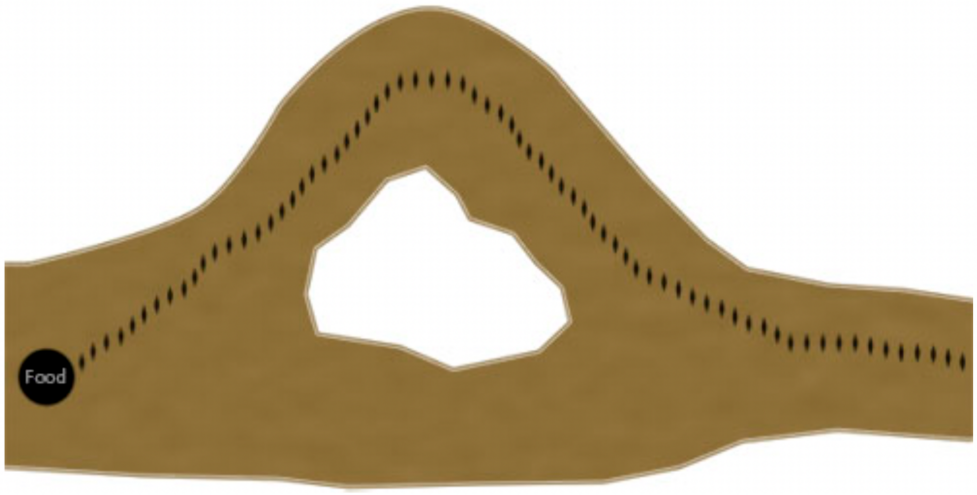
In computer science and operations research, the **ant colony optimization** algorithm (**ACO**) is a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs. Artificial ants stand for multi-agent methods inspired by the behavior of real ants. The pheromone-based communication of biological ants is often the predominant paradigm used. Combinations of artificial ants and local search algorithms have become a method of choice for numerous optimization tasks involving some sort of graph, e.g., vehicle routing and internet routing.

This algorithm is a member of the ant colony algorithms family, in swarm intelligence methods, and it constitutes some metaheuristic optimizations. Initially proposed by Marco Dorigo in 1992, the first algorithm was aiming to search for an optimal path in a graph, based on the behavior of ants seeking a path between their colony and a source of food. The original idea has since diversified to solve a wider class of numerical

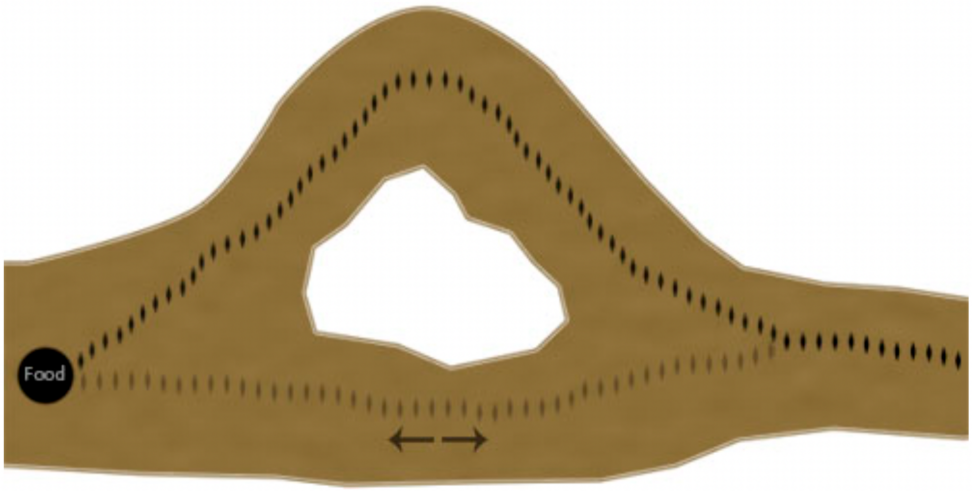
problems, and as a result, several problems have emerged, drawing on various aspects of the behavior of ants. Typical applications of ant colony optimization are combinatorial optimization problems such as the traveling salesman problem, however it can also be used to solve various scheduling and routing problems.

Understanding Ants

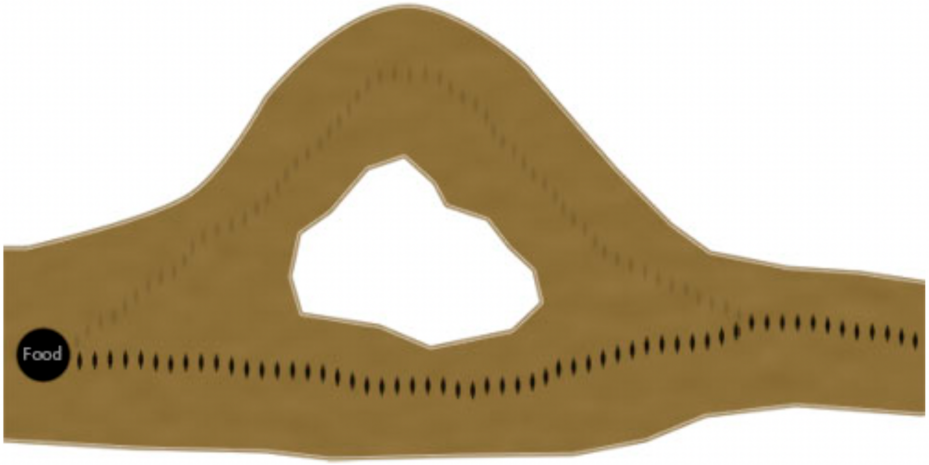
Ants tend to have blurry eyesight, particularly in smaller species, and a few subterranean taxa are completely blind, so how do they follow to get to the food using the shortest path? This essentially is the ant colony optimization problem, for which it's first important to understand the natural behavior of ants. One of the main characteristics of ants is their ability to leave pheromone trails, this helps them to communicate with other ants and follow the same paths the previous ones did, so when they come across pheromone trails they tend to expect food if they decide to follow it. But they can't follow two paths if they ever come across two pheromone trails, therefore, **they have a higher probability of following the path with higher pheromone strength**. Also, if the paths have no pheromone at all, then they pick their path randomly. Now let's consider the following scenario and assume that the first ant chose the longer path (due to no pheromone).



Now since it's all probabilistic, most of the ants would pick the original (longer) path, but some ants (maybe the second one) would pick the shorter path.



Currently, the longer one has a stronger pheromone trail, but the shorter one also keeps getting picked randomly (with lower frequency). But the important thing to notice is that in the second path, the ants have an advantage of getting to the food quicker and coming back. For example, it might take 15 minutes in the longer path, while it only takes 9 minutes in the shorter path. Now, we know that the pheromone essence evaporates with time. This implies that the pheromone levels deposited on the shorter path by the time the ants return would be higher than that deposited on the longer one. Hence due to this property of the ants, the shorter paths would gradually acquire more pheromone then the longer ones over time. Therefore, the longer paths are less travelled upon, this reduces into the travelling salesman problem.



The Algorithm and converting ACO into TSP

We have N iterations, at each iteration we drop M ants at random cities. Each ant follows a path and returns to its initial city. While going to the next city, it chooses the one with higher pheromone content with a higher probability. Here it is important to note that this would work only when all the cities are interconnected. One more thing is to be noted, for the ant to consider all possible scenarios other than the current best, some amount of randomness is required to be added. To allow this, a heuristic value is calculated which helps us in the searching process. While incorporating the ACO algorithm into the TSP, this heuristic will be proportional to the length of the edge connecting the two cities. The larger the edge, the less likely it is to be picked.

Probability of choosing the next city

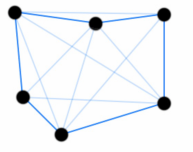
The probability for going from city i to city j for an ant k can be given as:

$$p_{ij}^{k} = \frac{[t_{ij}]^{\alpha} \cdot [n_{ij}]^{\beta}}{\sum_{l \in N_l^k} [t_{il}]^{\alpha} \cdot [n_{il}]^{\beta}}$$

t_{ij} represents the amount of pheromone present on the $edge_{ij}$ and n_{ij} represents the heuristic value(i.e. distance). The parameters alpha and beta are used to control the importance of the pheromone and the distances. Here the summation is for all the possible next cities for the ant k.

The Pheromone Update (Local)

Once the ant has travelled all the cities and returns back to the initial city, the pheromone that the ant left on the path has to be mathematically incorporated into the code. Therefore we add a small quantity (same for all the edges) to all the edges according to the following formula.The amount Q/C^k is added to every edge of the path that the ant travelled, where C^k is the total distance travelled by the ant. Q decides how much weightage we want to give to the pheromone update.

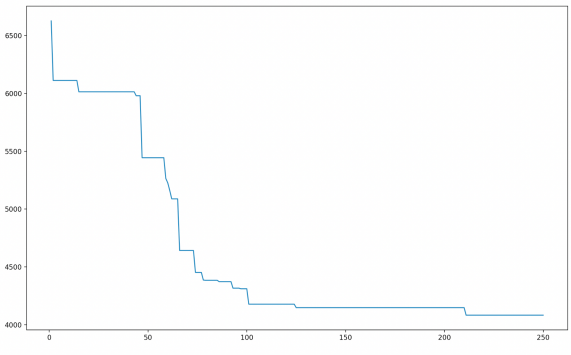


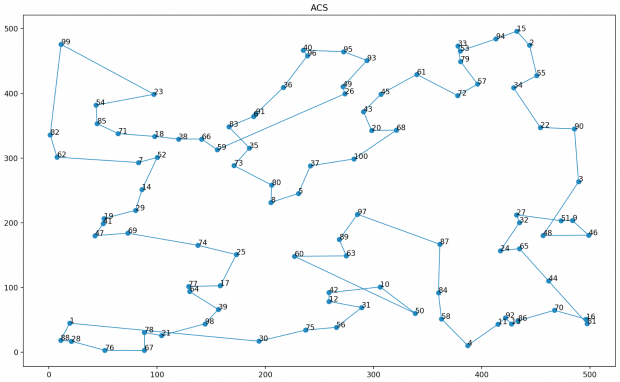
The Pheromone Update (Global)

It is natural for the pheromones to evaporate with time, therefore, we incorporate this by scaling the pheromone levels by (1 - rho) at the end of every iteration.

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij}$$

Where tau_{ij} is the pheromone level between city i and city j. And this is done for all the edges in the graph.





Results - ACO

The trip sequence that the salesman should take found using the ACS approach looks like:

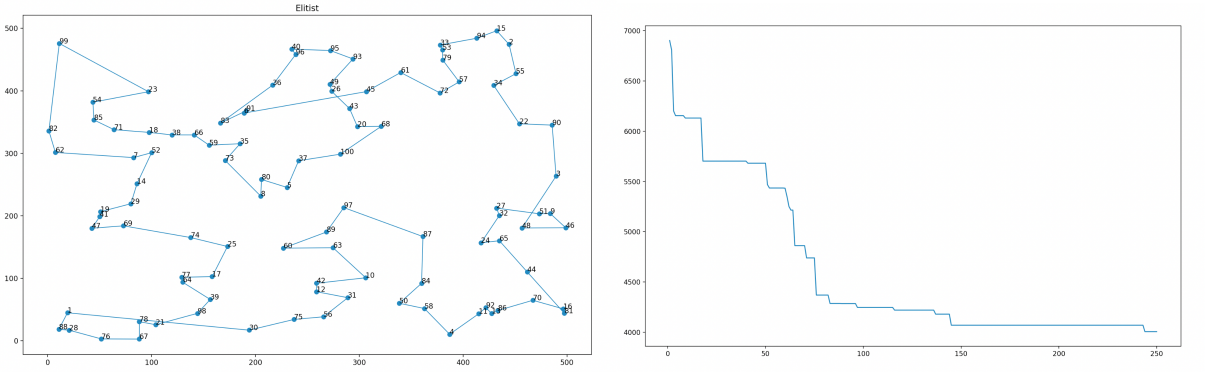
86 -> 83 -> 57 -> 3 -> 10 -> 91 -> 12 -> 85 -> 69 -> 15 -> 80 -> 43 -> 64 -> 23 -> 31 -> 26 -> 50 -> 8 -> 45 -> 47 -> 2 -> 89 -> 21 -> 33 -> 54 -> 1 -> 14 -> 93 -> 52 -> 32 -> 78 -> 56 -> 71 -> 60 -> 44 -> 42 -> 19 -> 67 -> 99 -> 36 -> 4 -> 7 -> 79 -> 72 -> 34 -> 82 -> 5 -> 90 -> 35 -> 95 -> 39 -> 94 -> 92 -> 48 -> 25 -> 58 -> 65 -> 37 -> 17 -> 70 -> 84 -> 53 -> 22 -> 98 -> 81 -> 61 -> 6 -> 51 -> 13 -> 28 -> 18 -> 40 -> 46 -> 68 -> 73 -> 24 -> 16 -> 76 -> 63 -> 38 -> 97 -> 20 -> 77 -> 66 -> 75 -> 27 -> 87 -> 0 -> 29 -> 74 -> 55 -> 30 -> 11 -> 41 -> 9 -> 49 -> 59 -> 62 -> 88 -> 96 -> 86

Total distance = 4082.8143632357737

The Elitist ACO

This is a slight modification over the original ACO, here while performing the pheromone global update, we also add an extra pheromone amount before scaling all the

pheromone levels. This extra amount is added along the global best route, hence the name, “elitist” approach.



Results - Elitist

The trip sequence that the salesman should take found using the Elitist approach looks like:

35 -> 95 -> 39 -> 94 -> 92 -> 48 -> 25 -> 42 -> 19 -> 67 -> 99 -> 36 -> 4 -> 79 -> 7 -> 72 -> 34 -> 58 -> 65 -> 37 -> 17 -> 70 -> 84 -> 53 -> 22 -> 98 -> 81 -> 61 -> 6 -> 51 -> 13 -> 28 -> 18 -> 40 -> 46 -> 68 -> 73 -> 24 -> 16 -> 76 -> 63 -> 38 -> 97 -> 20 -> 77 -> 66 -> 75 -> 27 -> 87 -> 0 -> 29 -> 74 -> 55 -> 30 -> 11 -> 41 -> 9 -> 62 -> 59 -> 88 -> 96 -> 86 -> 83 -> 49 -> 57 -> 3 -> 10 -> 91 -> 12 -> 85 -> 69 -> 15 -> 80 -> 43 -> 64 -> 23 -> 31 -> 26 -> 50 -> 8 -> 45 -> 47 -> 2 -> 89 -> 21 -> 33 -> 54 -> 1 -> 14 -> 93 -> 32 -> 52 -> 78 -> 56 -> 71 -> 60 -> 44 -> 5 -> 90 -> 82 -> 35

Total distance = 4005.820865987321

Time Complexity: O(I * A * N)

I = number of iterations

A = Number of Ants

N = Number of cities

The Greedy approach

This approach is pretty straightforward, here we randomly pick a starting city, and at each step, we pick the nearest city as the next city. For each iteration, it takes O(n) time to search for the next city, and there are n iterations.

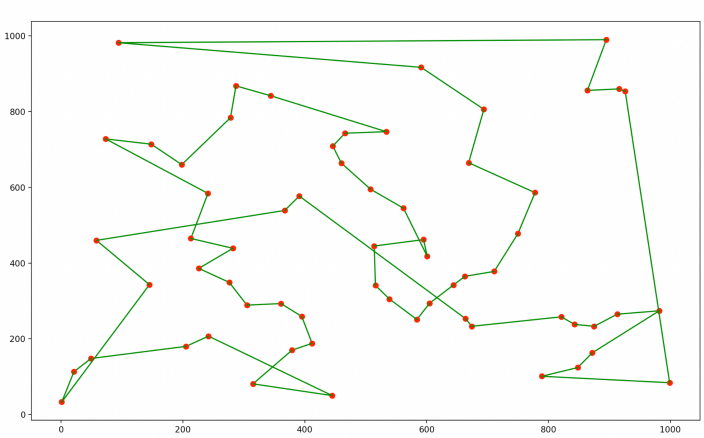
Time Complexity: $O(n^2)$

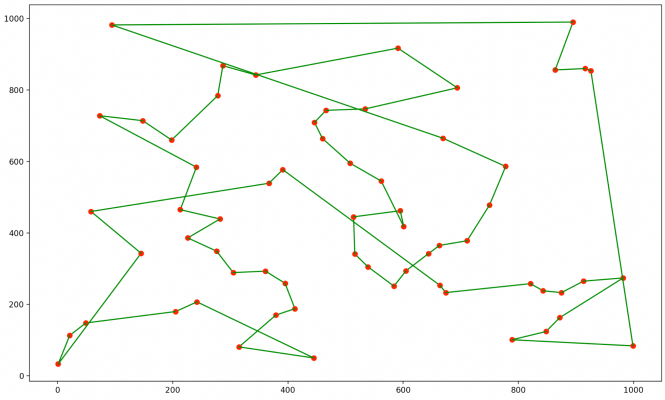
Results

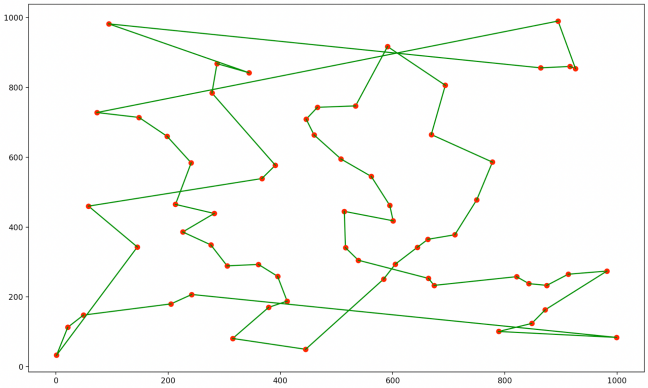
8335.26492773339 - Starting from city 1

8473.08785127396 - Starting from city 5

8563.666692185201 - Starting from city 10







Conclusion

While ACO provides better solutions with fewer iterations, it takes more time per iteration as compared to GA, which ends up providing better solutions in the same timespan with more iterations. ACO in general provides better performance across the board with datasets containing more cities. ACO has similar performance with GA with the same iteration count i and value of m (ant/chromosome count), however it is infeasible to run ACO with the same values as GA for i and m, as ACO is simply slower to compute. GA also has fewer dataset-dependent parameters, making it easier and faster to run as compared to ACO, which you initially have to find the optimal parameters for that dataset in order to ensure best performance that can compete with GA. Furthermore, in terms of shortest distance between the cities, PSO is better than GA and SA. However, PSO appeared in the last order in term of execution time.

Furthermore, in term of shortest distance between the cities, PSO stated

better than GA and SA. However, PSO appeared in the last

order in term of execution time.