# *A Comparison in the Implementation of The Travelling Salesman Problem using Brute – Force and Ant Colony Optimisation*

*J Component Report for the course Data Structures and Algorithms (CSE2003)*

*Submitted By:*


| NAME | REG. NO. | SLOT |
|---|---|---|
| AMAN ANAND | 19BCE0521 | B1 |
| AKSHAT TRIPATHI | 19BCE0527 | B1 |
| AYUSH KHARE | 19BCE0498 | B1 |


*In partial fulfilment for the award of the degree of*

Bachelor of Technology
in

**COMPUTER SCIENCE ENGINEERING**

Under the guidance of

**Faculty: Prof. Raghavan R**


**School of Computer Science and Engineering**


**Academic Year: 2019 - 2020**

# *REVIEW*

# *TWO*

# ***Review Two***

## Drawbacks of Brute-Force Algorithm:

We as a team have mainly focused on overcoming three of the major drawbacks of the brute force algorithm used to implement the Travelling Salesman Problem. These drawbacks include:

### a) Combinatorial Explosion (or) the curse of dimensionality

Combinatorial Explosion (or commonly referred to as the curse of dimensionality) refers to the large number of combination which arise when one undertakes the Brute- Force method to solve a problem. Travelling Salesman Problem essentially find a route between the cities to visit. These cities are present as vertices in a graph and the “paths” traversed by the salesman refer to the edges in the graph. Now, we know that in a closed polygon the number of edges formed between the n vertices can be brought about by the formula ${}^{n}C_2$, as from the n vertices we choose any two vertices to join which forms an edge. When employing the brute force method, it judges each and every path and then finds the optimum route for the salesman. In essence, the brute force method has ${}^{n}C_2$ iterations in the program and additional computational time (although small) to find the optimal path. In small groups of vertices brute force can be applied as the value of n is small and the computation time is still small. But if we were extrapolate the above theory to a large set of vertices (as is often the case, when we consider a large “city”) the value of n increases significantly. This causes the value of ${}^{n}C_2$ to increase manifold and causes the computational time to also increase manifold. This drawback which results in compilation time of the program sky-rocketing, is called as the curse of dimensionality or combinatorial explosion.

**b) Time Complexity**

As discussed above, in large “cities” or graph when we employ the Travelling Salesman Algorithm, we find that the computational time has increased significantly. It is common knowledge that the time complexity of program varies directly with the number of iterations in a program. As explained in the previous mentioned point, the more the number of cities that we include in the problem statement, the greater is the number of iterations that we have to undergo. And as mentioned above, it is clear that more the number of iterations, the more is the time complexity of the program. When implementing this program for a huge city the value of ‘’n” increases severely and the time complexity becomes mountainous decreasing the efficiency of the program.

**c) Memory Storage Efficiency**

In addition to the above two problems it can easily be deduced that one startlingly clear problem becomes the storage of all the combination that the brute force algorithm produces, so that we can come back and analyse these combination so as to furnish the most optimum path. Again, with such a heavy load of data, the efficiency in storage reduces significantly, hence reduces the neatness of the program and its extent of efficiency.

**Proposal to overcome the above adversities.**

To reduce the impact of the above mentioned drawback and to some extent even overcome it, we have undertaken using a slightly different algorithmic approach to the travelling salesman problem. The algorithm that our team chose is the ACO (Ant Colony Optimization) Technique. ACO has been covered in detail in the previous review, but we would like to shed some light in the less highlighted areas. Like any other algorithm, ACO also has its set of drawbacks. To quickly summarize them, they include:

a) **Probability distribution can change for each iteration.**

After each iteration of the ACO the probability distribution for the next path changes according to the formula.

b) **Have a difficult theoretical analysis.**

c) **Have uncertain time to convergence.**

The time taken by the program to converge to the end point of the graph and produce an optimum path for the result is variable as the probability of the next path taken changes after each iteration.

d) **Have dependant sequences of random decisions.**

Navigating past the above mentioned difficulties, ACO still provides a more optimal result in cases where the brute force algorithm falters majorly. Whereas brute forces algorithm faces difficulties like, time complexity, and combinatorial explosion when the number of vertices or “cities” in the graph increases past a certain point, ACO algorithm takes the large number of cities in stride and this algorithm gives us a much more efficient time complexity **O(nlogn)** with respect to the brute force technique **O($n^2$)**.

The way this algorithm essentially works is:
At the beginning, l ants are placed to the n cities randomly. Then each ant decides the next city to be visited according to the probability $p_{ij}^{k}$. After n iterations of this process, every ant completes a tour. Obviously, the ants with shorter tours should leave more pheromone than those with longer tours. Therefore, the trail levels are updated as on a tour each ant leaves pheromone quantity given by Q/Lk, where Q is a constant and Lk the length of its tour,

respectively. On the other hand, the pheromone will evaporate as the time goes by. Where t is the iteration counter, $\rho \in [0, 1]$ the parameter to regulate the reduction of $\tau_{ij}$, $\Delta\tau_{ij}$ the total increase of trail level on edge (i, j) and $\Delta\tau_{ijk}$ the increase of trail level on edge (i, j) caused by ant k, respectively. After the pheromone trail updating process, the next iteration t + 1 will start.

## Our Coding Implementation (40%)

```
class ACO(object):
    def __init__(self, ant_count: int, generations: int, alpha: float,
beta: float, rho: float, q: int,
                 strategy: int):
        """
        :param ant_count:
        :param generations:
        :param alpha: relative importance of pheromone
        :param beta: relative importance of heuristic information
        :param rho: pheromone residual coefficient
        :param q: pheromone intensity
        :param strategy: pheromone update strategy. 0 - ant-cycle, 1 -
ant-quality, 2 - ant-density
        """
        self.Q = q
        self.rho = rho
        self.beta = beta
        self.alpha = alpha
        self.ant_count = ant_count
        self.generations = generations
        self.update_strategy = strategy

    def _update_pheromone(self, graph: Graph, ants: list):
        for i, row in enumerate(graph.pheromone):
```

```
            for j, col in enumerate(row):
                graph.pheromone[i][j] *= self.rho
                for ant in ants:
                    graph.pheromone[i][j] += ant.pheromone_delta[i][j]

    # noinspection PyProtectedMember
    def solve(self, graph: Graph):
        """
        :param graph:
        """
        best_cost = float('inf')
        best_solution = []
        for gen in range(self.generations):
            # noinspection PyUnusedLocal
            ants = [_Ant(self, graph) for i in range(self.ant_count)]
            for ant in ants:
                for i in range(graph.rank - 1):
                    ant._select_next()
                ant.total_cost += graph.matrix[ant.tabu[-1]][ant.tabu[0]]
                if ant.total_cost < best_cost:
                    best_cost = ant.total_cost
                    best_solution = [] + ant.tabu
                # update pheromone
                ant._update_pheromone_delta()
            self._update_pheromone(graph, ants)
            # print('generation #{}, best cost: {}, path: {}'.format(gen,
best_cost, best_solution))
        return best_solution, best_cost


class _Ant(object):
    def __init__(self, aco: ACO, graph: Graph):
        self.colony = aco
        self.graph = graph
        self.total_cost = 0.0
```

```
        self.tabu = []  # tabu list
        self.pheromone_delta = []  # the local increase of pheromone
        self.allowed = [i for i in range(graph.rank)]  # nodes which are
allowed for the next selection
        self.eta = [[0 if i == j else 1 / graph.matrix[i][j] for j in
range(graph.rank)] for i in
                     range(graph.rank)]  # heuristic information
        start = random.randint(0, graph.rank - 1)  # start from any node
        self.tabu.append(start)
        self.current = start
        self.allowed.remove(start)

    def _select_next(self):
        denominator = 0
        for i in self.allowed:
            denominator += self.graph.pheromone[self.current][i] **
self.colony.alpha * self.eta[self.current][
                                                                    i] **
self.colony.beta
        # noinspection PyUnusedLocal
        probabilities = [0 for i in range(self.graph.rank)]  # probabilities
for moving to a node in the next step
        for i in range(self.graph.rank):
            try:
                self.allowed.index(i)  # test if allowed list contains i
                probabilities[i] = self.graph.pheromone[self.current][i] **
self.colony.alpha * \
                    self.eta[self.current][i] ** self.colony.beta / denominator
            except ValueError:
                pass  # do nothing
        # select next node by probability roulette
        selected = 0
        rand = random.random()
        for i, probability in enumerate(probabilities):
            rand -= probability
```

```
        if rand <= 0:
            selected = i
            break
    self.allowed.remove(selected)
    self.tabu.append(selected)
    self.total_cost += self.graph.matrix[self.current][selected]
    self.current = selected

# noinspection PyUnusedLocal
def _update_pheromone_delta(self):
    self.pheromone_delta = [[0 for j in range(self.graph.rank)] for i in 
range(self.graph.rank)]
    for _ in range(1, len(self.tabu)):
        i = self.tabu[_ - 1]
        j = self.tabu[_]
        if self.colony.update_strategy == 1:  # ant-quality system
            self.pheromone_delta[i][j] = self.colony.Q
        elif self.colony.update_strategy == 2:  # ant-density system
            # noinspection PyTypeChecker
            self.pheromone_delta[i][j] = self.colony.Q / 
self.graph.matrix[i][j]
        else:  # ant-cycle system
            self.pheromone_delta[i][j] = self.colony.Q / self.total_cost
```

**X—X—X—X—X—X—X—X—X—X—X—X—X—X—X—X—X—X—X**