RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	19CSE422A
Course Name	Computational Intelligence
Programme	B. Tech.
Department	Computer Science & Engineering
Faculty	Faculty of Engineering & Technology

Name of the Student	Subhendu Maji
Reg. No	18ETCS002121
Semester/Year	7th semester / 2018 batch
Course Leader/s	

Declaration Sheet				
Student Name	Subhendu Maji			
Reg. No	18ETCS002121			
Programme	B. Tech.		Semester/Year	7th sem /2018 batch
Course Code	19CSE422A			
Course Title	Computational Intelligence			
Course Date		to		
Course Leader				

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of university regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

Faculty of Engineering and Technology			
Ramaiah University of Applied Sciences			
Department	Computer Science and Engineering	Programme	B. Tech. in CSE- Summer
Semester/Batch	7/2018		
Course Code	19CSE422A	Course Title	Computational Intelligence
Course Leader	Dr. Vaishali R. Kulkarni and Dr. Monika Ravishankar		

Assignment-2

Reg. No.	18ETCS002121	Name of Student	Subhendu Maji

Sections		Marking Scheme	Mark		
			Max Marks	First Examiner Marks	Moderator
Part A	Part A				
	A1	Review of deterministic and metaheuristic algorithms	05		
	A2	Case studies	03		
	A3	Conclusion	02		
		Part-A Max Marks	10		
Part B					
	B.1	Design candidate solutions and fitness and algorithm	07		
	B.2	Discuss the graph model	03		
	B.3	Software Simulation with sample input and output	05		
		B.1 Max Marks	15		
		Total Assignment Marks	25		

Course Marks Tabulation				
Component-1 (B) Assignment	First Examiner	Remarks	Moderator	Remarks
A				
B				
Marks (out of 25)				

Question No. 1

Solution to Question No. 1:

1.1 A critical review of deterministic and computationally intelligent metaheuristic optimization algorithms

Mathematical optimization (alternatively spelled optimization) or mathematical programming is the selection of a best element, with regard to some criterion, from some set of available alternatives. Optimization problems of sorts arise in all quantitative disciplines from computer science and engineering to operations research and economics, and the development of solution methods has been of interest in mathematics for centuries.

In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function.

Deterministic Optimization techniques

- **Dynamic programming**

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution. Memorization is typically employed to enhance performance.
However, like deterministic dynamic programming also its stochastic variant suffers from the curse of dimensionality. For this reason, approximate solution methods are typically employed in practical applications

- **Linear programming**

Linear programming is a method to achieve the best outcome (such as maximum profit or lowest cost) in a mathematical model whose requirements are represented by linear relationships. Linear programming is a special case of mathematical programming (also known as mathematical optimization).

More formally, linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints. Its feasible region is a convex polytope, which is a set defined as the intersection of finitely many half spaces, each of which is defined by a linear inequality. Its objective function is a real-valued affine (linear) function defined on this polyhedron. A linear programming algorithm finds a point in the polytope where this function has the smallest (or largest) value if such a point exists.

- **Branch and bound**

Branch and bound) is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

The algorithm depends on efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search.

- **Integer Programming**

Integer programming expresses the optimization of a linear function subject to a set of linear constraints over integer variables. The statements presented in Linear programming: a production planning example are all linear programming models. However, linear programs with very large

numbers of variables and constraints can be solved efficiently. Unfortunately, this is no longer true when the variables are required to take integer values. Integer programming is the class of problems that can be expressed as the optimization of a linear function subject to a set of linear constraints over integer variables. It is in fact NP-hard. More important, perhaps, is the fact that the integer programs that can be solved to provable optimality in reasonable time are much smaller in size than their linear programming counterparts. There are exceptions, of course, and this documentation describes several important classes of integer programs that can be solved efficiently, but users of OPL should be warned that discrete problems are in general much harder to solve than linear programs.

Meta-Heuristic Optimization techniques

In computer science and mathematical optimization, 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. Metaheuristics sample a subset of solutions which is otherwise too large to be completely enumerated or otherwise explored. Metaheuristics may make relatively few assumptions about the optimization problem being solved and so may be usable for a variety of problems.

Compared to optimization algorithms and iterative methods, metaheuristics do not guarantee that a globally optimal solution can be found on some class of problems. Many metaheuristics implement some form of stochastic optimization, so that the solution found is dependent on the set of random variables generated. In combinatorial optimization, by searching over a large set of feasible solutions, metaheuristics can often find good solutions with less computational effort than optimization algorithms, iterative methods, or simple heuristics. As such, they are useful approaches for optimization problems.

Metaheuristics are used for combinatorial optimization in which an optimal solution is sought over a discrete search-space. An example problem is the travelling salesman problem where the search-space of candidate solutions grows faster than exponentially as the size of the problem increases, which makes an exhaustive search for the optimal solution infeasible. Additionally,

multidimensional combinatorial problems, including most design problems in engineering such as form-finding and behavior-finding, suffer from the curse of dimensionality, which also makes them infeasible for exhaustive search or analytical methods. Metaheuristics are also widely used for job shop scheduling and job selection problems. Popular metaheuristics for combinatorial problems include simulated annealing, genetic algorithms, scatter search, tabu search and Swarm Intelligence.

- **Swarm intelligence (SI**

Swarm intelligence (SI) is the collective behavior of decentralized, self-organized systems, natural or artificial. The concept is employed in work on artificial intelligence. The expression was introduced by Gerardo Beni and Jing Wang in 1989, in the context of cellular robotic systems.

SI systems consist typically of a population of simple agents or boids interacting locally with one another and with their environment. The inspiration often comes from nature, especially biological systems. The agents follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave, local, and to a certain degree random, interactions between such agents lead to the emergence of "intelligent" global behavior, unknown to the individual agents. Evolutionary algorithms (EA), particle swarm optimization (PSO), differential evolution (DE), ant colony optimization (ACO) and their variants dominate the field of nature-inspired metaheuristics. When appropriate parameters are determined, and when sufficient convergence stage is achieved, they often find a solution that is optimal, or near close to optimum – nevertheless, if one does not know optimal solution in advance, a quality of a solution is not known.[15] In spite of this obvious drawback it has been shown that these types of algorithms work well in practice, and have been extensively researched, and developed

- **Particle swarm optimization**

Particle swarm optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It solves a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formula over the particle's position and velocity. Each particle's movement is influenced by its local best-known position, but is also guided toward the best-known positions in the search-space, which are

updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.

PSO is a metaheuristic as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. Also, PSO does not use the gradient of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by classic optimization methods such as gradient descent and quasi-newton methods. However, metaheuristics such as PSO do not guarantee an optimal solution is ever found.

- **Ant Colony Optimization**

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.

As an example, ant colony optimizations a class of optimization algorithms modeled on the actions of an ant colony. Artificial 'ants' (e.g. simulation agents) locate optimal solutions by moving through a parameter space representing all possible solutions. Real ants lay down pheromones directing each other to resources while exploring their environment. The simulated 'ants' similarly record their positions and the quality of their solutions, so that in later simulation iterations more ants locate better solutions. One variation on this approach is the bee's algorithm, which is more analogous to the foraging patterns of the honey bee, another social insect.

This algorithm is a member of the ant colony algorithms family, in swarm intelligence methods, and it constitutes some metaheuristic optimizations.

- **Artificial bee colony**

Artificial bee colony (ABC) algorithm is an optimization technique that simulates the foraging behavior of honey bees, and has been successfully applied to various practical problems ABC belongs to the group of swarm intelligence algorithms and was proposed by Karaboga in 2005.

A set of honey bees, called swarm, can successfully accomplish tasks through social cooperation. In the ABC algorithm, there are three types of bees: employed bees, onlooker bees, and scout bees. The employed bees search food around the food source in their memory; meanwhile they share the information of these food sources to the onlooker bees. The onlooker bees tend to select good food sources from those found by the employed bees. The food source that has higher quality (fitness) will have a large chance to be selected by the onlooker bees than the one of lower quality. The scout bees are translated from a few employed bees, which abandon their food sources and search new ones.

In the ABC algorithm, the first half of the swarm consists of employed bees, and the second half constitutes the onlooker bees.

The number of employed bees or the onlooker bees is equal to the number of solutions in the swarm. The ABC generates a randomly distributed initial population of SN solutions (food sources), where SN denotes the swarm size.

- **Genetic algorithm**

a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection. Some examples of GA applications include optimizing decision trees for better performance, automatically solve sudoku puzzles, hyperparameter optimization, etc.

In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.

The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

1.2 Case studies using deterministic and metaheuristic algorithms

Case Study 1:

Operational zones for comparing metaheuristic and deterministic one-dimensional global optimization algorithms

Univariate continuous global optimization problems are considered in this case study. Several widely used multidimensional metaheuristic global optimization methods—genetic algorithm, differential evolution, particle swarm optimization, artificial bee colony algorithm, and firefly algorithm—are adapted to the univariate case and compared with three Lipschitz global optimization algorithms. For this purpose, it has been introduced a methodology allowing one to compare stochastic methods with deterministic ones by using operational characteristics originally proposed for working with deterministic algorithms only. As a result, a visual comparison of methods having different nature on classes of randomly generated test functions becomes possible. A detailed description of the new methodology for comparing, called “operational zones”, is given and results of wide numerical experiments with five metaheuristics and three Lipschitz algorithms are reported.

Let us consider the following univariate continuous global optimization problem

$$f^* := f(x^*) = \min f(x), \quad x \in [a, b],$$

where the function f(x) satisfies the Lipschitz condition over the search interval [a, b]:

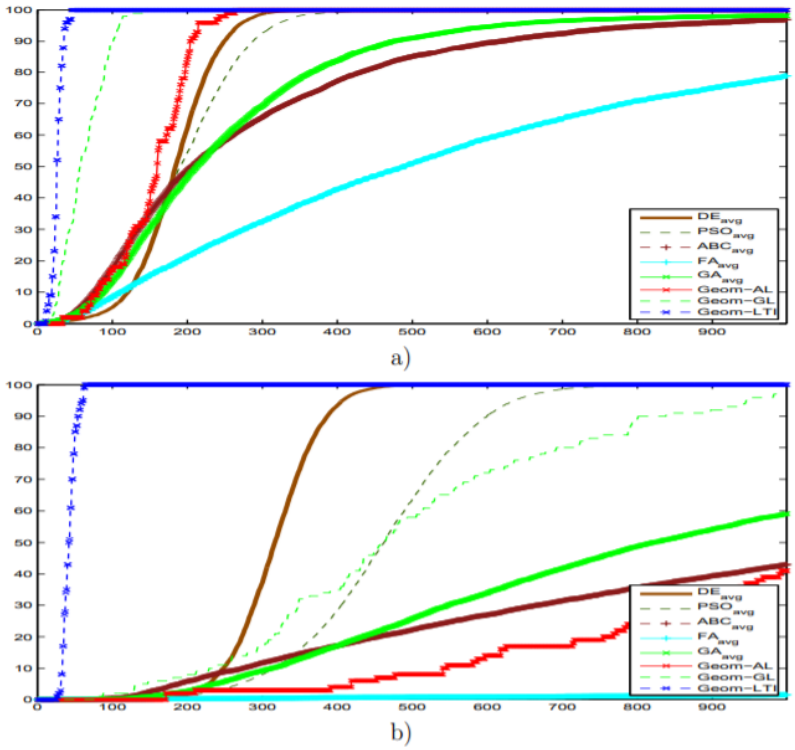
$$|f(x_1) - f(x_2)| \leq L|x_1 - x_2|, \quad x_1, x_2 \in [a, b],$$

where L is the Lipschitz constant, $0 < L < \infty$.

The traditional way to do this is to use a number of test functions and to show that a new method is better in some sense than its competitors. Then, a trade-off between the number of test functions, reliability of the comparison, and visibility of results arises. Clearly, a small number of test functions does not lead to a reliable comparison and a huge number of functions produces huge tables with a lot of data that sometimes are difficult for a fast visualization and an immediate comprehension

Let us consider among these methods the following two classes:

- Metaheuristic methods;
    - Genetic algorithm (GA),
    - Differential Evolution algorithm (DE),
    - Particle Swarm Optimization (PSO),
    - Artificial Bee Colony algorithm (ABC)
    - Firefly algorithm (FA)
- Lipschitz methods.
    - Geom-AL: Geometric method with an A priori given overestimate of the Lipschitz constant
    - Geom-GL: Geometric method with a Global estimate of the Lipschitz constant. It is the same method as Geom-AL but the Lipschitz constants are estimated adaptively during the search.
    - Geom-LTI: Geometric method with a maximum-additive Local Tuning and a pessimistic local Improvement



Average operational characteristics for metaheuristic methods and operational characteristics for Lipschitz methods with $\varepsilon = 10^{-4}$ (a) and $\varepsilon = 10^{-6}$ (b)

Case Study 2:

Deterministic vs Heuristic Algorithms for Eco-Driving Application in Metro Network

This case study investigates the performances of two different algorithms for calculating the metro vehicles speed profiles minimizing the energy consumption of a given path. The optimization problem, formulated as a Mixed-Integer Nonlinear Programming (MINLP) problem, take into account constraints related to the trip (time-table, distances, acceleration limits, etc.) and constraints related to the electric feeding system (line voltage, substations supplied power, etc.). We implement a deterministic algorithm based on Dynamic Programming Optimization (DPO) method, and a heuristic one based on Particle Swarm Optimization (PSO) method, and we evaluate the performances in terms of the time spent on search of a feasible solution and the goodness of the obtained solution. The resulting speed profiles are tested in a metro network simulation tool, based on the 'quasi static' backwards looking method, and taking into account the track topology, the characteristics of the metro system.

The problem we aim to solve now consists in finding the optimal/suboptimal speed profile obtained as sequence of basic control regimes that leads to the minimization of energy consumed for a given trip; the traction or braking force u is the control variable, and the state variables are the train position x and its speed v. We consider as objective function, the energy absorbed for a fixed amount of time T, thus the optimization problem can be written as:

$$\begin{cases} \min\limits_{u(t)} (J) \\ J = \int_0^T u(t)v(t) + \beta \left| \frac{du(t)}{dt} \right| dt \end{cases}$$

$$u_{\min} \le u(t) \le u_{\max}$$

$$0 \le v(t) \le V_{\max}$$

and the boundary conditions:

$$s(0) = 0, \qquad s(T) = L$$
$$v(0) = 0, \qquad v(T) = 0$$

The research of the eco-drive speed profiles can be formulated as a Mixed-Integer Nonlinear Programming problem (MINLP).

The high computational demand of deterministic algorithms in solving these problems, has suggested to find heuristic algorithms to explore more quickly the solution space, even finding a suboptimal solution.

In order to compare the performances of deterministic and heuristic algorithm, the optimization problem is addressed using both,

- DPO (Dynamic Programming Optimization Method)
- PSO (Particle Swarm Optimization Method)

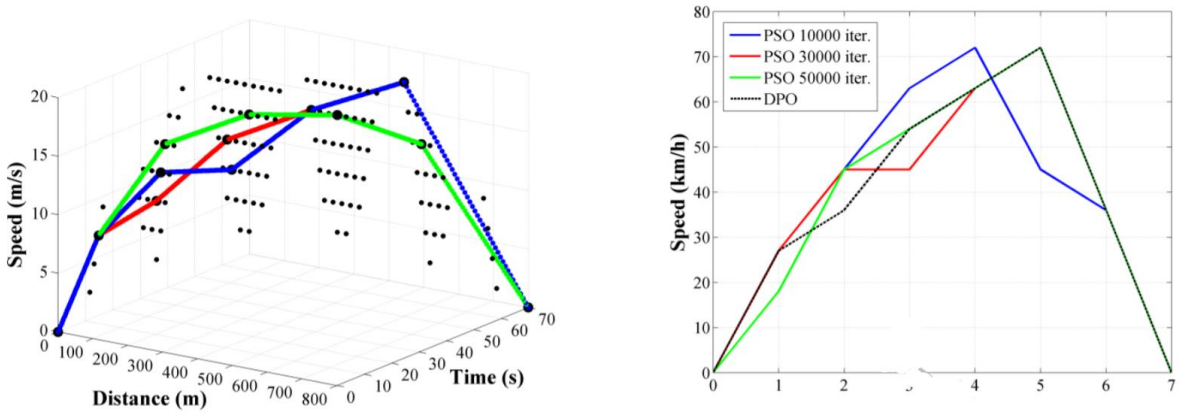
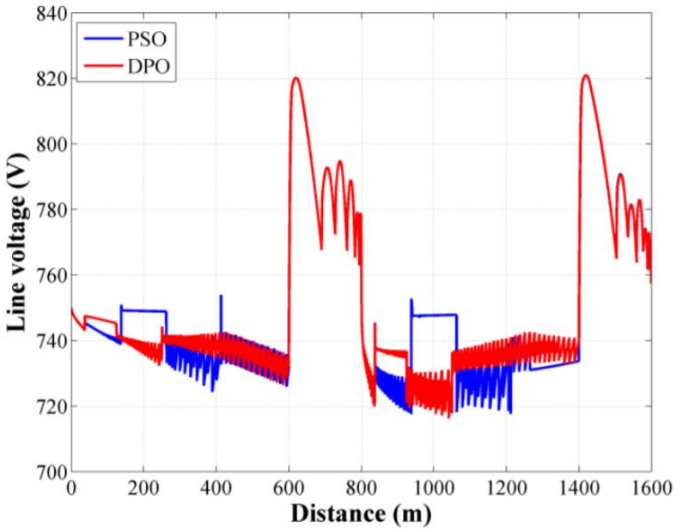


Figure (a) Comparison between DPO and PSO based eco-drive speed cycle

(b) Interactions effect on PSO eco-drive algorithm.

RESERVE TIME [S]	*DPO SUBSTATION ENERGY [kWh]*	*PSO SUBSTATION ENERGY [kWh]*
10	6.491	6.880
15	6.101	6.467
20	5.712	6.054



Comparison on line voltage: DPO and PSO algorithms.

1.3 Conclusion

The success of metaheuristic algorithms comes from the fact that they have seen to be applicable to almost all types of optimization problems, with continuous and/or discrete variables, with no requirements about differentiability of the functions considered, with all types of constraints, for linear or non-linear formulations. This is not the case for conventional deterministic methods, which are generally designed for some well-defined problems with strict limits and become of no use for other problems. To give an example, linear programming, for instance, is applicable only to linear problems with continuous variables with linear constraints and it cannot be applied to other problems is there is any non-linearity somewhere in the formulation. When the function to be optimized becomes quadratic, then one can apply the technique named quadratic programming which is designed only for those problems. If the variables are integers, then one has to apply a specially prepared algorithm. Thus, one can say that the biggest advantage of metaheuristic algorithms is their versatility, i.e., their applicability to almost all type of problems. As far as the speed of computations, one has to state that, in general, if a problem can be solved both by a conventional method and by a metaheuristic algorithm, the central processor time for the former will be much shorter. But this does not form an important disadvantage for metaheuristic algorithms considering that they are especially successful in cases where classical methods are inapplicable or insufficient.

Although deterministic approaches take advantage of analytical properties of the problem to generate a sequence of points that converge to a global solution, metaheuristic approaches have been found to be more flexible and efficient than deterministic approaches. For solving nonconvex or large-scale optimization problems, deterministic methods may not be easy to derive a globally optimal solution within reasonable time due to the high complexity of the problem. Metaheuristic approaches therefore are presented to reduce the computational time of solving an optimization problem, but the obtained solution is not guaranteed to be a feasible or globally optimal solution. These two types of optimization methods have different pros and cons. Therefore, integrating deterministic and metaheuristic approaches may be a good way of solving large-scale optimization problems for finding a global optimum.

Thus, to conclude, if highly precise answers are necessary, it is better to make use of deterministic algorithms, whereas, if imprecise answers are allowed to a certain extent, it may be beneficial to implement a metaheuristic algorithm. Moreover, depending on the problem and computation and space limitations, either a deterministic or metaheuristic algorithm might be more apt for the application.

Solution to Question No. 2:

2.1 Design solution candidates and fitness to meet the requirements of the problem

The traveling salesman problem (TSP) is a deceptively simple combinatorial problem.
It can be stated very simply: a salesman spends his time visiting n cities (or nodes) cyclically. In one tour he visits each city just once, and finishes up where he started. The question is: in what order should he visit the cities to minimize the distance traveled? TSP became a benchmark for many new approaches in combinatorial optimization (CO)

Due to the practical importance of CO problems, many algorithms to tackle them have been developed. These algorithms can be classified as either complete or approximate algorithms. Complete algorithms are guaranteed to find for every finite size instance of a CO problem an optimal solution in bounded time. Yet, for CO problems that are N P-hard, no polynomial time algorithm exists, assuming that $P \neq NP$. Therefore, complete methods might need exponential computation time in the worst-case. This often leads to computation times too high for practical purposes. Thus, the development of approximate methods—in which we sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a significantly reduced amount of time—has received more and more attention.

Ant colony optimization (ACO) is one of the most recent techniques for approximate optimization. The inspiring source of ACO algorithms are real ant colonies. More specifically, ACO is inspired by the ants' foraging behavior. At the core of this behavior is the indirect communication between the ants by means of chemical pheromone trails, which enables them to find short paths between their nest and food sources. This characteristic of real ant colonies is exploited in ACO algorithms in order to solve, for example, discrete optimization problems.
Marco Dorigo and colleagues introduced the first ACO algorithms in the early 1990's. The development of these algorithms was inspired by the observation of ant colonies. Ants are social insects. They live in colonies and their behavior is governed by the goal of colony survival rather than being focused on the survival of individuals. The behavior that provided the inspiration for

ACO is the ants' foraging behavior, and in particular, how ants can find shortest paths between food sources and their nest.

The functioning of an ant colony provides indirect communication with the help pheromones, which ants excrete. Pheromones are chemical substances which attract other ants searching for food. The attractiveness of a given path depends on the quantity of pheromones that the ant feels. Pheromone excretion is governed by some rules and has not always the same intensity. The quantity of pheromones depends on the attractiveness of the route. The use of more attractive route ensures that the ant exudes more pheromones on its way back and so that path is more also attractive for other ants. The important characteristic of pheromones is evaporation. This process depends on the time. When the way is no longer used, pheromones are more evaporated and the ants begin to use other paths. What is important for ACO algorithm the moving of ants. This motion is not deterministic, but it has stochastic character, so the ants can find the path, which is firstly unfavorable, but which is ultimately preferable for food search. The important characteristic is that a few individuals continuously use non-preferred path and look for another best way.

Design

Moving of virtual ant depends on the amount of pheromone on the graph edges. The probability pi k of transition of a virtual ant from the node i to the node k is given by formula below. We assume the existence of internal ant's memory.

$$p_i^k = \frac{\tau_i^\alpha + \eta_i^\beta}{\Sigma(\tau_{Ni}^\alpha + \eta_{Ni}^\beta)}$$

where

τ_i - indicates the attractiveness of transition in the past

η_i - adds to transition attractiveness for ants,

N^i - set of nodes connected to point i, without the last visited point before i,

a, β - parameters found by simulation.

Virtual ant is using the same reverse path as the path to the food resource based on his internal memory, but in opposite order and without cycles, which are eliminated. After elimination of the cycles, the ant puts the pheromone on the edges of reverse path according to formula:

$$\tau_{ij}^{t+1} = \tau_{ij}^{t} + \Delta\tau^{t}$$

where

τ_{ij}^{t}- value of pheromone in step t,

$\Delta\tau$- value by ants saved pheromones in step t.

Values $\Delta\tau$ can be constant or they can be changed depends of solution quality.

At last, the pheromones on the edges are evaporated. The evaporation helps to find the shortest path and provide that no other path will be assessed as the shortest. This evaporation of pheromones has an intensity ρ

This formula is applied on all graph edges with intensity ρ (interval (0, 1)).

On this knowledge we can compose an algorithm of ACO, which can be used for solving the travelling salesman problem.

1. Ant's scan graph G. The aim of this scan is to find an optimal solution
2. Every ant has its own memory, which is used for saving information about travelled path (for example about travelled nodes). This memory can also serve to ensure constraints or to evaluate of the solution.
3. The process begins in state x_s^k and has one or more ending constraints e^k. Let the actual state
4. of an ant be the state $x_r = (x_{(r-1)}, i)$ and no ending constraint is complied, so the ant moves to node j in neighborhood of the state $N^k(x_r)$ and the ant moves to the new state $(x_r, j) \in X$. In case that some ending constraint is complied with, the ant ends with process of scan. The transition to a state that represents unacceptable solution is usually banned by appropriate implementation of internal ant memory.
5. The next ant motion depends on the probability, which is calculated on the base of pheromone quantity on edges of graph, and it also takes into consideration its local memory and the acceptance of this step.

6. If the ant can to add new component of graph G_c, it can update the value of corresponding pheromone information (information is bound with corresponding edge, or aim node).
7. The ant can update pheromone values after reverse path construction by editing associate pheromone values.

2.2 Use any graph model to illustrate the results

```
import random
import operator
import matplotlib.pyplot as plt

class Graph(object):
    def __init__(self, cost_matrix: list, rank: int):
        """
        :param cost_matrix:
        :param rank: rank of the cost matrix
        """
        self.matrix = cost_matrix
        self.rank = rank
        self.pheromone = [[1 / (rank * rank)
                           for j in range(rank)] for i in range(rank)]


def create_cities(n_cities):
    cities = []
    points = []
    for i in range(n_cities):
        x = random.uniform(0, 1000)
        y = random.uniform(0, 1000)
        cities.append(dict(index=i+1, x=x, y=y))
        points.append((x, y))
    points.append(points[0])
    return cities, points


def plot(points, path: list, showPath=True):
    x = []
    y = []
    for point in points:
        x.append(point[0])
        y.append(point[1])
    y = list(map(operator.sub, [max(y) for i in range(len(points))], y))
    plt.plot(x, y, 'co')
    if showPath:
        for _ in range(1, len(path)):
            i = path[_ - 1]
            j = path[_]
            plt.arrow(x[i], y[i], x[j] - x[i], y[j] - y[i],
                      color='r', length_includes_head=True)

    plt.xlim(0, max(x) * 1.1)
    plt.ylim(0, max(y) * 1.1)
    plt.show()

def main():

    cities, points = create_cities(40)
    plot(points, [], False)


main()
```

Output:

```
PS D:\RUAS-sem-07\CI\assignment2\codes> python .\graph.py
PS D:\RUAS-sem-07\CI\assignment2\codes>
```

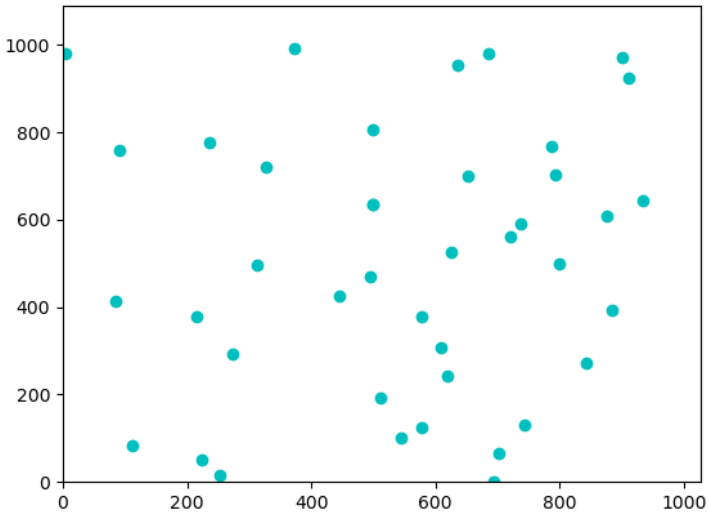


Figure 1 graph to visualize random cities

2.3 Software Simulation with sample input and output

ACO.py U main.py U

ACO.py > _Ant > __init__

```python
import random


class Graph(object):
    def __init__(self, cost_matrix: list, rank: int):
        """
        :param cost_matrix:
        :param rank: rank of the cost matrix
        """
        self.matrix = cost_matrix
        self.rank = rank
        self.pheromone = [[1 / (rank * rank)
                           for j in range(rank)] for i in range(rank)]

```

```python
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]

    def solve(self, graph: Graph):
        """
        :param graph:
        """
        best_cost = float('inf')
        best_solution = []
        bests = []

        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)
            bests.append(best_cost)
#             print('generation #{}, best cost: {}, path: {}'.format(gen, best_cost, best_solution))
        best_solution.append(best_solution[0])
        return best_solution, best_cost, bests

```

```
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
        # nodes which are allowed for the next selection
        self.allowed = [i for i in range(graph.rank)]
        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

        # probabilities for moving to a node in the next step
        probabilities = [0 for i in range(self.graph.rank)]
        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

    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
```

ACO.py U main.py U

main.py > create_cities

```python
import random
import operator
import math
import matplotlib.pyplot as plt

from ACO import ACO, Graph


def plot(points, path: list, showPath=True):
    x = []
    y = []
    for point in points:
        x.append(point[0])
        y.append(point[1])
    y = list(map(operator.sub, [max(y) for i in range(len(points))], y))
    plt.plot(x, y, 'co')
    if showPath:
        for _ in range(1, len(path)):
            i = path[_ - 1]
            j = path[_]
            plt.arrow(x[i], y[i], x[j] - x[i], y[j] - y[i],
                      color='m', length_includes_head=True)

    plt.xlim(0, max(x) * 1.1)
    plt.ylim(0, max(y) * 1.1)
    plt.show()


def distance(city1: dict, city2: dict):
    return math.sqrt((city1['x'] - city2['x']) ** 2 + (city1['y'] - city2['y']) ** 2)


def create_cities(n_cities):
    cities = []
    points = []
    for i in range(n_cities):
        x = random.uniform(0, 1000)
        y = random.uniform(0, 1000)
        cities.append(dict(index=i+1, x=x, y=y))
        points.append((x, y))
    points.append(points[0])
    return cities, points


def main(n_cities=10, n_ants=10, alpha=1, beta=5, rho=0.5, q=1, max_iter=100, strategy=0):

    cities, points = create_cities(n_cities)
    plot(points, [], False)
    cost_matrix = []
    rank = len(cities)

    for i in range(rank):
        row = []
        for j in range(rank):
            row.append(distance(cities[i], cities[j]))
        cost_matrix.append(row)

    aco = ACO(n_ants, max_iter, alpha, beta, rho, q, strategy)
    # aco = ACO(10, 100, 1.0, 10.0, 0.5, 10, 1)

    graph = Graph(cost_matrix, rank)
    path, cost, bests = aco.solve(graph)
    print('cost: {}, path: {}'.format(cost, path))
    plot(points, path)

    #  plotting best cost of each generation vs iteration
    plt.plot(bests)
    plt.show()
```

Output #1:

```
70
71    main(40, 10,  1.0, 10.0, 0.5, 10, 100, 0)
72
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS D:\RUAS-sem-07\CI\assignment2\codes> python .\main.py
cost: 5479.664684905925, path: [26, 30, 3, 13, 0, 12, 33, 20, 18, 11, 25, 22, 21, 34, 23, 17, 35, 8,
26]
PS D:\RUAS-sem-07\CI\assignment2\codes>
```

cost: 5479.664684905925, path: [26, 30, 3, 13, 0, 12, 33, 20, 18, 11, 25, 22, 21, 34, 23, 17, 35, 8, 9, 10, 2, 15, 6, 16, 27, 7, 28, 29, 19, 32, 5, 4, 24, 36, 38, 37, 14, 1, 31, 39, 26]

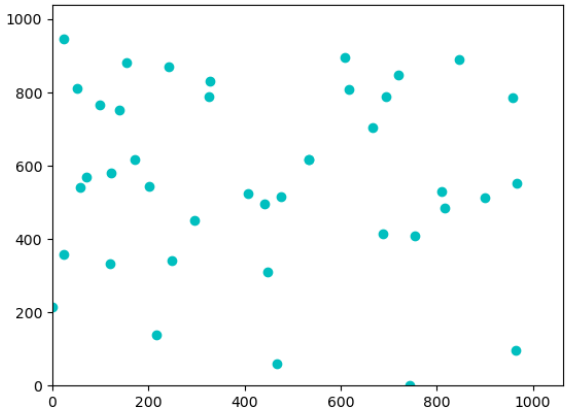


Figure 2 randomly generated cities

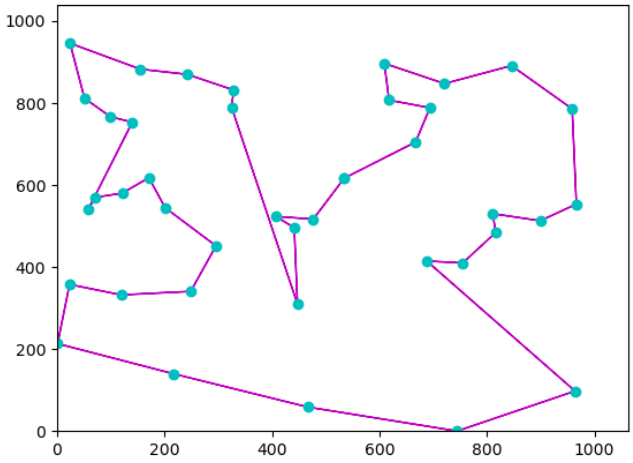


Figure 3 best path after 100 operations

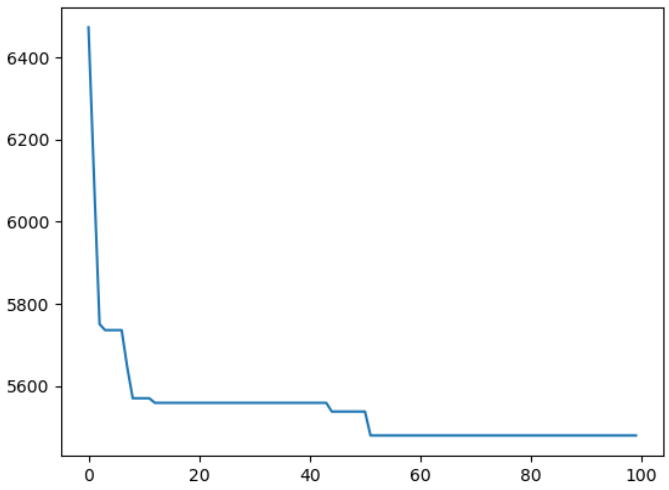


Figure 4 best of each generation vs iterations

Output #2:

```
70
71   main(50, 20,  1.0, 10.0, 0.5, 10, 150, 1)
72
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS D:\RUAS-sem-07\CI\assignment2\codes> python .\main.py
cost: 5732.667650038509, path: [42, 28, 11, 31, 30, 15, 29, 24, 12, 1, 18, 7, 14, 47, 2
 16, 39, 40, 36, 34, 44, 37, 8, 13, 46, 42]
PS D:\RUAS-sem-07\CI\assignment2\codes>
```

```
cost: 5732.667650038509, path: [42, 28, 11, 31, 30, 15, 29, 24, 12, 1, 18,
7, 14, 47, 27, 0, 38, 19, 17, 21, 22, 2, 48, 33, 4, 6, 23, 35, 43, 32, 45,
3, 20, 10, 5, 25, 41, 9, 49, 26, 16, 39, 40, 36, 34, 44, 37, 8, 13, 46, 42]
```

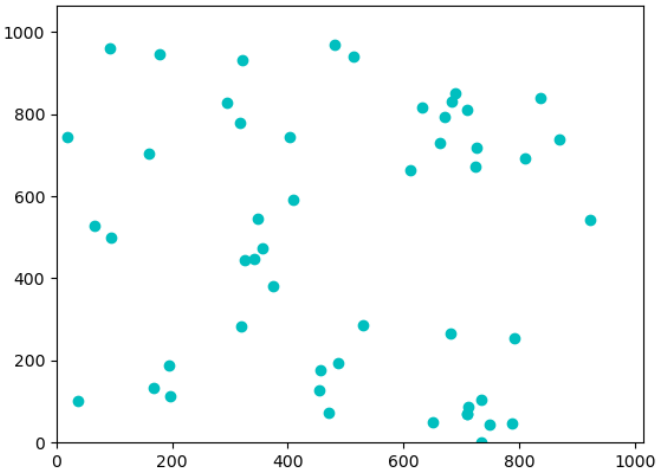


Figure 5 randomly generated cities

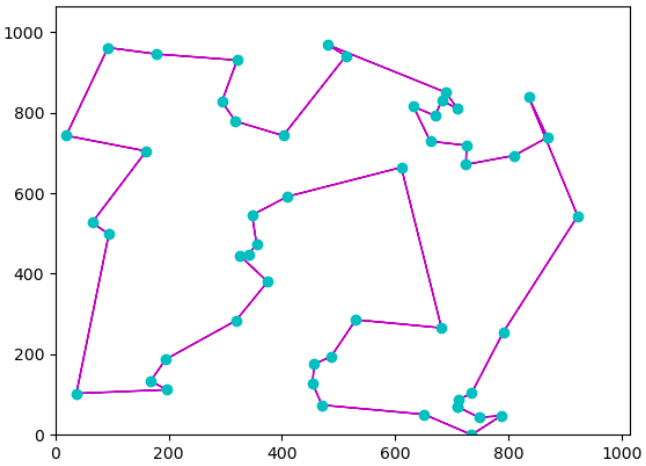


Figure 6 best path after 100 operations

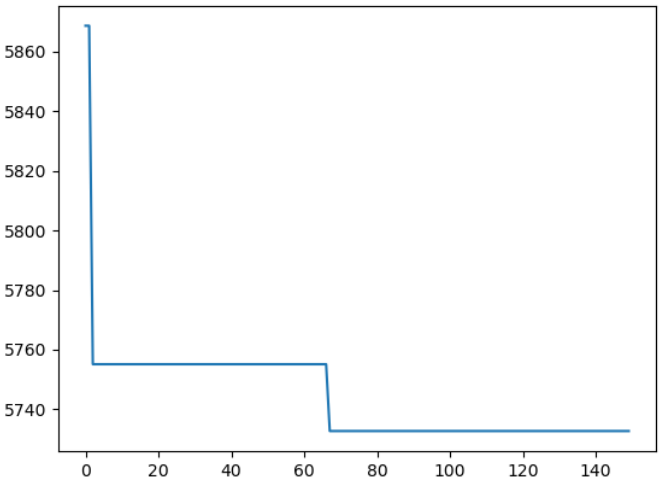


Figure 7 best of each generation vs iterations

1. https://www.researchgate.net/profile/Giuseppe-Graber-2/publication/280978633_Deterministic_vs_Heuristic_Algorithms_for_Eco-Driving_Application_in_Metro_Network/links/59808509aca272ebd41cc156/Deterministic-vs-Heuristic-Algorithms-for-Eco-Driving-Application-in-Metro-Network.pdf
2. http://ictactjournals.in/paper/IJSC_Vol_10_Iss_3_Paper_8_2117_2123.pdf
3. https://www.researchgate.net/profile/Yaroslav-Sergeyev/publication/303977061_Operational_zones_for_comparing_metaheuristic_and_deterministic_one-dimensional_global_optimization_algorithms/links/57e6699b08aedcd5d1aa970f/Operational-zones-for-comparing-metaheuristic-and-deterministic-one-dimensional-global-optimization-algorithms.pdf
4. https://www.hilarispublisher.com/open-access/solving-traveling-salesmen-problem-using-ant-colony-optimizationalgorithm-2168-9679-1000260.pdf
5. https://d1wqtxts1xzle7.cloudfront.net/53789161/j.pnsc.2008.03.02820170707-21843-brps3a.pdf?1499432373=&response-content-disposition=inline%3B+filename%3DAn_ant_colony_optimization_method_for_ge.pdf&Expires=1638540669&Signature=MGdvqeViD1QBpvcUTOSkwc2oI6R001BmASGKmIBisgdgt9TvYS~9oJab4rdK4ZckLBm0Vo8XQBf~SV-9LQjHqe7WpgmZ9ydSvPCHsHD-ke3ygXEzmTgKk1kir21nq070Pk2fuZOPf7FoClvgYQSkRbBrNg1PkmUHVO3psn2Ya5HXorVIq9fXSviaaCxuook2r0xbONrzcDArGL32d-4Joc4snbtFMSR4SrTUU380KI2NjG0ylYuvTzgFEDNEJ96~6yRjI3pILXLQdfhhiU6ITJS~I1ZbpMe2wi5Fpyp72fFgzNAxd8W495L9JlN7uf8tnihQVWM-m0JoRIrMq2fSqQ__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA
6. http://personal.cimat.mx:8181/~jortega/MaterialDidactico/modestoI10/Proyectos/Blum%20(2005)AntCol.pdf
7. https://en.wikipedia.org/wiki/Dynamic_programming
8. https://en.wikipedia.org/wiki/Branch_and_bound
9. https://en.wikipedia.org/wiki/Combinatorial_optimization
10. https://en.wikipedia.org/wiki/Metaheuristic