Individual Report

Thanawat Anansuthivara

730054312 [ta541@exeter.ac.uk](mailto:ta541@exeter.ac.uk)

University of Exeter

ECMM409 Nature-Inspired Computation

CA2: Solving Complex problems with Nature-inspired Computation.

1. Introduction

The objective of this individual report is to present my contributions to the group project undertaken as part of the ECMM409 Nature-Inspired Computation course. The primary focus of our project is to optimize the Traveling Thief Problem (TTP), a complex and multifaceted challenge that combines the elements of the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP), with the nature-inspired computation. This problem is distinctive in its multi-objective nature, arising from the interdependencies between its various components, notably the time traveled, and the profit made by the thief.

In this report, I will elucidate the specific areas of the project to which I significantly contributed, explore the range of nature-inspired algorithms that we examined and utilized, and deliberate on the adaptation process of these algorithms to effectively address the TTP's objectives. My intention is to provide a thorough account of my involvement in this collaborative effort, highlighting the challenges encountered and the strategies employed to solve the problems. It aims not only to showcase my role in this complex optimization task but also to share the valuable insights and learnings acquired through this project and collaborative experience.

2. Personal Contribution

In this project, I have mainly developed the codebase for nature-inspired computation algorithms in python language. During the discussion at the early period of the coursework, we have decided to write 3 algorithms, including Genetic Algorithm, Ant Colony Optimization, and Particle swarm Optimization since they were introduced in the class lecture and suitable for solving TTP. I have assigned to develop GA and PSO to evaluate the performance of the models so we can discuss which models will be used in our group report. Furthermore, I also contribute to discussion on developing the algorithm and choosing the final model for the group project.

2.1 Nature-inspired Algorithm in Python

The algorithms are conducted in the Python, a simple with high quality programming language, from the scratch. To enhance efficiency, I created utility functions (Appendix: Utilization function) that could be used for both algorithms. This avoided redundancy and streamlined our development process. These functions included data reading capabilities and cost functions, essential for optimizing the algorithms based on the specific criteria of the TTP. Moreover, the algorithms were structured as class objects in Python. This design choice facilitated easier experimentation and analysis, allowing us to modify and test different aspects of the algorithms.

2.1.1 Genetic Algorithm

The algorithm is developed from the coursework 1 where GA was implemented to optimize the Traveling Salesman Problem. To improve the efficient of the code, the structures are recreated with more simplicity. The chromosome class is written as a representative of a solution in the population which it carries path, packing plan, and phenome (profit, time, and net profit). Regard of my coursework 1 where I studied and implemented GA for TSP, the combination of ordered crossover and inversion crossover performed the best solution with different numerical parameters so I decided to put them into a new code as a default operator for TTP. In case of the plan, the simple crossover and simple swap are implemented as a default. Finally, the GA class instance is constructed to used for experiment as it will be simple for the team and to store the result into its own instance so it can be tracked during the experiment.

2.1.2 Particle Swarm Optimization

In addition to GA, PSA is also developed to optimize TTP with expect that it will surpass the GA performance. This is a challenge task due to the problem that PSO purpose was to optimize the continuous value tasks which mean I must find the approach to define a particle movement and fitness function for TTP, ensuring that both route efficiency and knapsack optimization were addressed concurrently. My focus was on ensuring the correct function for updating the path and plan and achieving a balance between global and local search capabilities, enabling the algorithm to efficiently explore the solution space of the TTP. The velocity function is defined as a swap method for updating the particle follow the personal best and global best solution which means the algorithm will take a long time to execute since it must check every particle solution compared the personal best and global best. To introduce the diversity, the multiple flip item and reverse path are introduced as it will create the new solution as the mutation operation does for GA where the decision that it will go for more diversity or follow the personal/global best solution is dynamically changed with update probabilities function to introduce dynamically rate as it should have in the original format. Same goes for GA, PSO has written as class instance of Particle and PSO (Space) so it can be modified and tracked at ease.

2.2 Contribution in Group Discussions

As part of the team, I actively participated in group discussions, contributing ideas and insights that shaped our approach to solving the TTP. My contributions included:

* Algorithm Selection and Strategy: I played a pivotal role in selecting GA and PSO for our project, drawing on my understanding of their strengths and applicability to multi-objective optimization problems like TTP.
* Problem Decomposition: I assisted in breaking down the complex TTP into manageable sub-problems, ensuring that our solutions addressed both the TSP and KP aspects effectively.
* Algorithm Adaptation: I suggested modifications and adaptations to the algorithms to better suit the unique challenges posed by TTP, such as balancing exploration and exploitation in PSO.

3. Algorithm Survey

To identify potential algorithms, extensive research was conducted, encompassing academic literature, existing solutions for TSP and KP, and broader nature-inspired computation methods. The focus was on algorithms known for their effectiveness in combinatorial and multi-objective optimization.

Several algorithms emerged as strong candidates include Genetic Algorithm (GA): Known for its robustness in exploring large search spaces and its applicability to both TSP and KP, Ant Colony Optimization (ACO): Effective for path optimization problems like TSP, with potential adaptability for the packing aspect of TTP, And Particle Swarm Optimization (PSO): Typically used for continuous optimization, but its adaptability for discrete problems like TTP was intriguing.

Genetic Algorithm (GA):

The Genetic Algorithm was chosen for its proven success in complex problems like the Traveling Salesman Problem and Knapsack Problem, making it highly suitable for the Traveling Thief Problem (TTP). Its adaptability in handling dual objectives — route optimization and knapsack item selection — was a key factor in its selection.

Practicality and Team Expertise:

The team's familiarity with GA influenced its choice, ensuring an efficient development process. Theoretical understanding combined with practical considerations allowed us to effectively adapt GA to the unique challenges of TTP.

4. Experimentation

The experimentation phase began with baseline tests to understand the performance of our chosen algorithms - Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) - on the TTP. Initial runs were conducted using default parameters to establish a fundamental understanding of each algorithm's behavior in the context of TTP.

Subsequent experiments focused on tuning parameters such as population size, mutation rate, and crossover probability for GA, and swarm size, inertia weight, cognitive and social components for PSO. This fine-tuning was crucial to balance exploration and exploitation capabilities of the algorithms.

Based on early results, modifications were made to the algorithms. For GA, we introduced an adaptive mutation mechanism to maintain diversity in the population. In PSO, adjustments were made to the velocity update rule to better navigate the discrete solution space of TTP.

Parallel to these individual algorithm enhancements, comparative analysis was conducted to assess the relative strengths and weaknesses of GA and PSO in handling the intricacies of TTP. This involved evaluating both algorithms across a range of test instances, varying in city count, item number, and knapsack capacity.

Recognizing the need for problem-specific strategies, we incorporated heuristics into our algorithms. For instance, a greedy item selection heuristic was integrated into GA to improve the knapsack packing efficiency, while PSO was augmented with local search techniques to refine paths more effectively.

The effectiveness of our algorithms was measured using multiple metrics, including the total profit, travel time, and the convergence rate. We also monitored computational efficiency to ensure practical applicability.

The experimentation was iterative, with each round of testing providing insights that fueled further refinements. This iterative process was critical in gradually enhancing the algorithms' performance.

The culmination of our experimentation was a comprehensive evaluation against the benchmark problems set for the competition. This final assessment was pivotal in validating the effectiveness of our algorithms and strategies in solving the TTP.

The experimentation process was not only about optimizing algorithms but also about understanding the complex nature of TTP. It was a journey that provided valuable insights into algorithmic design and the practical aspects of solving multi-objective, nature-inspired computation problems.

5. Teamwork progress

At the outset, our team quickly aligned on the importance of effective collaboration and communication. We established regular meetings to discuss strategies, allocate tasks, and monitor progress. Each member brought unique strengths to the table, which we leveraged to enhance our approach to the TTP challenge.

Tasks were divided based on individual expertise and interests. Some team members focused on algorithm development, while others concentrated on data analysis and testing. This division allowed us to work efficiently and make significant progress in parallel streams.

Throughout the project, we faced several challenges, including integrating different components of the algorithm and optimizing its performance. These hurdles were met with a collaborative spirit, where brainstorming and collective problem-solving played a crucial role. The diverse skill set within our team enabled us to approach these issues from various angles, leading to innovative solutions.

Effective communication was key to our progress. We utilized various tools to stay connected, share updates, and make collective decisions. Open discussions and respectful consideration of each member's input helped us in refining our strategies and achieving consensus on critical aspects of the project.

The project was a learning curve for all team members. Working on the complex TTP, we not only enhanced our technical skills but also developed a deeper understanding of teamwork dynamics in a research-oriented environment.

6. Conclusion

Our project to tackle the Traveling Thief Problem (TTP) using Nature-inspired Computation Algorithms demonstrated notable success, combining iterative enhancements and strategic heuristics to address the complexities of TTP. The effective balance achieved by GA in travel and knapsack optimization, complemented by the innovative exploration strategies of PSO, was reflected in the significant improvements observed in benchmark problems.

The collaborative dynamics of the team played a crucial role in our success. Regular meetings, open communication, and efficient problem-solving and task distribution enhanced our individual efforts and collectively led to a robust and effective solution.

Looking forward, there are promising avenues for further exploration. Integrating machine learning for real-time adaptive decision-making and experimenting with hybrid algorithms or adaptive parameter tuning could potentially elevate our solution's effectiveness, especially for more complex TTP instances. This project has not only been a success in developing effective algorithms for a challenging problem but has also provided invaluable experience in teamwork and algorithm development, setting a solid foundation for future computational challenges.

7. Reference

8. Appendix: Codebase

8.1 Utilization function

8.1.1 Reading and Loading Dataset

In order to keep the format of the dataset used for algorithms, the Dataclass is introduced to store the data in specific format so it would reduce the confusion during the multiple algorithm development. The TTP is used to store the data from the text file which will be used for the main algorithm. The City and Item are defined to Store the city’s coordinate and item property.

dataclass

class City:

    index : int

    X : float

    Y : float

@dataclass

class Item:

    index : int

    Profit : int

    Weight : int

    Node : int

@dataclass

class TTP:

    Name :str = None

    DTYPE : str = None

    Dimension : int = 0

    ITEMS : int = 0

    CAPACITY : int = 0

    MIN\_SPEED : float = 0

    MAX\_SPEED : float = 0

    RENTING\_RATIO : float = 0

    EDGE\_W : str = None

    NODE : List[City] = None

    ITEM : List[Item] = None

Due to the special format of the dataset, the read\_problem function is defined to read and store the data into TTP object.

def read\_problem(file\_path:str):

    with open(file\_path,'r') as file:

        lines = file.readlines()

    data = TTP(NODE=[],ITEM=[])

    for i , line in enumerate(lines):

        if line.startswith("PROBLEM NAME"):

            data.Name = line.split(':')[-1].strip()

        elif line.startswith("KNAPSACK DATA TYPE"):

            data.DTYPE = line.split(':')[-1].strip()

        elif line.startswith("DIMENSION"):

            data.Dimension = int(line.split(':')[-1].strip())

        elif line.startswith("NUMBER OF ITEMS"):

            data.ITEMS = int(line.split(':')[-1].strip())

        elif line.startswith("CAPACITY OF KNAPSACK"):

            data.CAPACITY = int(line.split(':')[-1].strip())

        elif line.startswith("MIN SPEED"):

            data.MIN\_SPEED = float(line.split(':')[-1].strip())

        elif line.startswith("MAX SPEED"):

            data.MAX\_SPEED = float(line.split(':')[-1].strip())

        elif line.startswith("RENTING RATIO"):

            data.RENTING\_RATIO = float(line.split(':')[-1].strip())

        elif line.startswith("EDGE\_WEIGHT\_TYPE"):

            data.EDGE\_W = line.split(':')[-1].strip()

        elif line.startswith("NODE\_COORD\_SECTION"):

            for j in range(1,data.Dimension+1):

                node = lines[i+j].split()

                data.NODE.append(City(index=int(node[0]),X=float(node[1]),Y=float(node[2])))

        elif line.startswith("ITEMS SECTION"):

            for j in range(1,data.ITEMS+1):

                item = lines[i+j].split()

                data.ITEM.append(

                    Item(int(item[0]),int(item[1]),int(item[2]),int(item[3]))

                )

        else:

            pass

    return data

8.1.2 Multiple-objective cost function

Since the problem is defined as a multiple-objective optimisation, the independencies are evaluated through the iteration so the cost function must define to calculate both metric by calculate\_time\_and\_profit function which require metadata and set of coordinate and item property data. Moreover, to ensure that there are no rule-violated solutions. The condition is implemented.

def calculate\_time\_and\_profit(solution: List[int], plan: List[int], nodes: List[City], items: List[Item], min\_speed, max\_speed, max\_weight):

    total\_time = 0

    total\_profit = 0

    current\_weight = 0

    # Calculate the total travel time

    for i in range(len(solution)):

        current\_city\_index = solution[i]

        next\_city\_index = solution[0] if i == len(solution) - 1 else solution[i + 1]

        current\_city = nodes[current\_city\_index - 1]

        next\_city = nodes[next\_city\_index - 1]

        # Update current weight based on items picked at the current city

        for item, is\_picked in zip(items, plan):

            if is\_picked and item.Node == current\_city\_index:

                current\_weight += item.Weight

        # Calculate speed based on current weight

        speed = max\_speed - (current\_weight / max\_weight) \* (max\_speed - min\_speed)

        speed = max(speed, min\_speed)  # Ensure speed doesn't drop below minimum

        # Distance between current city and next city

        distance = euclidean\_distance((current\_city.X, current\_city.Y), (next\_city.X, next\_city.Y))

        # Update time with time to next city

        total\_time += distance / speed

# condition if weight violoated the capacity available

        if current\_weight > max\_weight:

            return np.Inf , 0.0

    # Calculate total profit from picked items

    for item, is\_picked in zip(items, plan):

        if is\_picked:

            total\_profit += item.Profit

    return total\_time, total\_profit

where Euclidean distance is defined separately

def euclidean\_distance(p1: Tuple[float, float], p2: Tuple[float, float]) -> float:

    return math.ceil(np.sqrt((p1[0] - p2[0])\*\*2 + (p1[1] - p2[1])\*\*2))

8.2 Genetic Algorithm

8.2.1 Dataclass

Dataclass is implemented to define the Chromosome object to make it easier to coding as oriental object programming.

@dataclass

class Phenome:

    time : float

    profit : float

    net\_profit : float

@dataclass

class Chromosome:

    path : List[int]

    plan : List[int]

    phenome : Phenome

8.2.2 Class instance

class GA:

    def \_\_init\_\_(self,problem\_data : TTP, population\_size, iterations, tour\_size, mut\_prob=1.0, cross\_prob=1.0) -> None:

        self.problem\_data = problem\_data

        self.iterations = iterations

        self.population\_size = population\_size

        self.tour\_size = tour\_size

        self.mut\_prob = mut\_prob

        self.cross\_prob = cross\_prob

        self.population = self.init\_population()

        # self.population = [Chromosome(\*generate\_ttp\_solution(

        #     problem\_data.Dimension,problem\_data.ITEM,problem\_data.CAPACITY

        # )) for \_ in range(population\_size)]

        self.best\_chromosome = sorted(self.population, key=lambda c: c.phenome.net\_profit)[-1]

        self.best\_history = [self.best\_chromosome.phenome.net\_profit]

        self.avg\_history = [sum(c.phenome.net\_profit for c in self.population) / len(self.population)]

        self.time\_execute = 0

    def init\_population(self):

        pop\_temp = []

        for \_ in range(self.population\_size):

            path , plan = generate\_ttp\_solution(self.problem\_data.Dimension,self.problem\_data.ITEM,self.problem\_data.CAPACITY)

            time , profit = calculate\_time\_and\_profit(path,plan,self.problem\_data.NODE,self.problem\_data.ITEM,self.problem\_data.MIN\_SPEED,self.problem\_data.MAX\_SPEED,self.problem\_data.CAPACITY)

            net\_profit = profit - (time\*self.problem\_data.RENTING\_RATIO)

            phenome = Phenome(time,profit,net\_profit)

            chorm = Chromosome(path,plan,phenome)

            pop\_temp.append(chorm)

        return pop\_temp

    def run(self):

        start = time.time()

        for \_ in range(self.iterations):

            p1 = self.tournament\_selection(self.population,self.tour\_size)

            p2 = self.tournament\_selection(self.population,self.tour\_size)

            if np.random.rand() < self.cross\_prob:

                p1, p2 = self.ordered\_crossover(p1,p2)

            if np.random.rand() < self.mut\_prob:

                p1 = self.inversion\_mutation(p1)

                p2 = self.inversion\_mutation(p2)

            self.population = self.replace\_weakest(self.population, p1)

            self.population = self.replace\_weakest(self.population, p2)

            current\_best = sorted(self.population, key=lambda c: c.phenome.net\_profit)[-1]

            if current\_best.phenome.net\_profit > self.best\_chromosome.phenome.net\_profit:

                self.best\_chromosome = current\_best

            self.best\_history.append(self.best\_chromosome.phenome.net\_profit)

            self.avg\_history.append(sum(c.phenome.net\_profit for c in self.population) / len(self.population))

        end = time.time()

        self.time\_execute = end- start

        return self.population , current\_best

    def ordered\_crossover(self,parent1: Chromosome, parent2: Chromosome) -> Tuple[Chromosome, Chromosome]:

        # Select crossover points for the path

        start, end = sorted(np.random.choice(range(len(parent1.path)), 2))

        # Create segments from parents

        parent1\_segment = parent1.path[start:end]

        parent2\_segment = parent2.path[start:end]

        # Create offspring paths excluding parent segments

        offspring1\_path = [city for city in parent2.path if city not in parent1\_segment]

        offspring2\_path = [city for city in parent1.path if city not in parent2\_segment]

        # Insert parent segments into offspring paths

        offspring1\_path[start:start] = parent1\_segment

        offspring2\_path[start:start] = parent2\_segment

        # For the plan, using a simple one-point crossover

        crossover\_point = np.random.randint(1, len(parent1.plan) - 1)

        offspring1\_plan = parent1.plan[:crossover\_point] + parent2.plan[crossover\_point:]

        offspring2\_plan = parent2.plan[:crossover\_point] + parent1.plan[crossover\_point:]

        time1 , profit1 = calculate\_time\_and\_profit(offspring1\_path,offspring1\_plan,self.problem\_data.NODE,self.problem\_data.ITEM,self.problem\_data.MIN\_SPEED,self.problem\_data.MAX\_SPEED,self.problem\_data.CAPACITY)

        net\_profit1 = profit1 - (time1\*self.problem\_data.RENTING\_RATIO)

        phenome1 = Phenome(time1,profit1,net\_profit1)

        time2 , profit2 = calculate\_time\_and\_profit(offspring2\_path,offspring2\_plan,self.problem\_data.NODE,self.problem\_data.ITEM,self.problem\_data.MIN\_SPEED,self.problem\_data.MAX\_SPEED,self.problem\_data.CAPACITY)

        net\_profit2 = profit2 - (time2\*self.problem\_data.RENTING\_RATIO)

        phenome2 = Phenome(time2,profit2,net\_profit2)

        offspring1 = Chromosome(offspring1\_path, offspring1\_plan, phenome1)

        offspring2 = Chromosome(offspring2\_path, offspring2\_plan, phenome2)

        # # Create new Chromosome instances for offspring

        # offspring1 = Chromosome(offspring1\_path, offspring1\_plan)

        # offspring2 = Chromosome(offspring2\_path, offspring2\_plan)

        return offspring1, offspring2

    def inversion\_mutation(self,chromosome: Chromosome):

        # Ensure there are at least two elements in the path

        if len(chromosome.path) < 2:

            return chromosome

        path = chromosome.path.copy()

        plan = chromosome.plan.copy()

        # Choose two distinct random positions in the path

        pos1, pos2 = sorted(np.random.choice(range(len(chromosome.path)), 2))

        # Invert the order of elements between pos1 and pos2

        path[pos1:pos2 + 1] = reversed(path[pos1:pos2 + 1])

        plan[pos1:pos2 + 1] = reversed(plan[pos1:pos2 + 1])

        new\_time , new\_profit = calculate\_time\_and\_profit(path,plan,self.problem\_data.NODE,self.problem\_data.ITEM,self.problem\_data.MIN\_SPEED,self.problem\_data.MAX\_SPEED,self.problem\_data.CAPACITY)

        new\_net\_profit = new\_profit - (new\_time\*self.problem\_data.RENTING\_RATIO)

        phenome = Phenome(new\_time,new\_profit,new\_net\_profit)

        return Chromosome(path,plan,phenome)

    def replace\_weakest(self,population : List[Chromosome], candidates:Chromosome):

        keys = [x.phenome.net\_profit for x in population]

        weakest\_index = np.argmin(keys)

        if candidates.phenome.net\_profit > population[weakest\_index].phenome.net\_profit:

            population[weakest\_index] = candidates

        return population

    def tournament\_selection(self,population: List[int], tournament\_size: int) -> Chromosome:

        """

        Selects a single Chromosome from the population using tournament selection.

        :param population: An instance of the Population class containing Chromosomes.

        :param tournament\_size: The number of Chromosomes to be selected for each tournament.

        :return: The winning Chromosome with the highest net profit.

        """

        # Ensure the tournament size is not larger than the population size

        tournament\_size = min(tournament\_size, len(population))

        # Randomly select 'tournament\_size' individuals from the population

        tournament\_contestants = np.random.choice(population, size=tournament\_size, replace=False)

        # Determine the winner based on the highest net profit

        winner = max(tournament\_contestants, key=lambda chromo: chromo.phenome.net\_profit)

        return winner

8.2.3 Extended function

def two\_opt\_swap(path, i, k):

    """ Perform a 2-opt swap by reversing the path segment between i and k """

    new\_path = path[:i] + path[i:k+1][::-1] + path[k+1:]

    return new\_path

def apply\_two\_opt\_local\_search(path, node\_data, min\_speed, max\_speed, capacity):

    """ Apply 2-opt local search to improve the path """

    improved = True

    while improved:

        improved = False

        for i in range(1, len(path) - 2):

            for k in range(i + 1, len(path)):

                new\_path = two\_opt\_swap(path, i, k)

                # Calculate new time and check if it's an improvement

                new\_time, \_ = calculate\_time\_and\_profit(new\_path, plan, node\_data, item\_data, min\_speed, max\_speed, capacity)

                current\_time, \_ = calculate\_time\_and\_profit(path, plan, node\_data, item\_data, min\_speed, max\_speed, capacity)

                if new\_time < current\_time:

                    path = new\_path

                    improved = True

                    break  # Improvement found, exit inner loop

            if improved:

                break  # Improvement found, exit outer loop

    return path

where they are replaced in the crossover and mutation function

class GA:

    …rest of the code…

    def ordered\_crossover(self,parent1: Chromosome, parent2: Chromosome) -> Tuple[Chromosome, Chromosome]:

        # Select crossover points for the path

        start, end = sorted(np.random.choice(range(len(parent1.path)), 2))

        # Create segments from parents

        parent1\_segment = parent1.path[start:end]

        parent2\_segment = parent2.path[start:end]

        # Create offspring paths excluding parent segments

        offspring1\_path = [city for city in parent2.path if city not in parent1\_segment]

        offspring2\_path = [city for city in parent1.path if city not in parent2\_segment]

        # Insert parent segments into offspring paths

        offspring1\_path[start:start] = parent1\_segment

        offspring2\_path[start:start] = parent2\_segment

        # For the plan, using a simple one-point crossover

        crossover\_point = np.random.randint(1, len(parent1.plan) - 1)

        offspring1\_plan = parent1.plan[:crossover\_point] + parent2.plan[crossover\_point:]

        offspring2\_plan = parent2.plan[:crossover\_point] + parent1.plan[crossover\_point:]

        offspring1\_path = apply\_two\_opt\_local\_search(offspring1\_path, self.problem\_data.NODE, self.problem\_data.MIN\_SPEED, self.problem\_data.MAX\_SPEED, self.problem\_data.CAPACITY)

        offspring2\_path = apply\_two\_opt\_local\_search(offspring2\_path, self.problem\_data.NODE, self.problem\_data.MIN\_SPEED, self.problem\_data.MAX\_SPEED, self.problem\_data.CAPACITY)

        time1 , profit1 = calculate\_time\_and\_profit(offspring1\_path,offspring1\_plan,self.problem\_data.NODE,self.problem\_data.ITEM,self.problem\_data.MIN\_SPEED,self.problem\_data.MAX\_SPEED,self.problem\_data.CAPACITY)

        net\_profit1 = profit1 - (time1\*self.problem\_data.RENTING\_RATIO)

        phenome1 = Phenome(time1,profit1,net\_profit1)

        time2 , profit2 = calculate\_time\_and\_profit(offspring2\_path,offspring2\_plan,self.problem\_data.NODE,self.problem\_data.ITEM,self.problem\_data.MIN\_SPEED,self.problem\_data.MAX\_SPEED,self.problem\_data.CAPACITY)

        net\_profit2 = profit2 - (time2\*self.problem\_data.RENTING\_RATIO)

        phenome2 = Phenome(time2,profit2,net\_profit2)

        offspring1 = Chromosome(offspring1\_path, offspring1\_plan, phenome1)

        offspring2 = Chromosome(offspring2\_path, offspring2\_plan, phenome2)

        # # Create new Chromosome instances for offspring

        # offspring1 = Chromosome(offspring1\_path, offspring1\_plan)

        # offspring2 = Chromosome(offspring2\_path, offspring2\_plan)

        return offspring1, offspring2

    def inversion\_mutation(self,chromosome: Chromosome):

        # Ensure there are at least two elements in the path

        if len(chromosome.path) < 2:

            return chromosome

        path = chromosome.path.copy()

        plan = chromosome.plan.copy()

        # Choose two distinct random positions in the path

        pos1, pos2 = sorted(np.random.choice(range(len(chromosome.path)), 2))

        # Invert the order of elements between pos1 and pos2

        path[pos1:pos2 + 1] = reversed(path[pos1:pos2 + 1])

        plan[pos1:pos2 + 1] = reversed(plan[pos1:pos2 + 1])

        mutated\_path = apply\_two\_opt\_local\_search(path, self.problem\_data.NODE, self.problem\_data.MIN\_SPEED, self.problem\_data.MAX\_SPEED, self.problem\_data.CAPACITY)

        new\_time , new\_profit = calculate\_time\_and\_profit(mutated\_path,plan,self.problem\_data.NODE,self.problem\_data.ITEM,self.problem\_data.MIN\_SPEED,self.problem\_data.MAX\_SPEED,self.problem\_data.CAPACITY)

        new\_net\_profit = new\_profit - (new\_time\*self.problem\_data.RENTING\_RATIO)

        phenome = Phenome(new\_time,new\_profit,new\_net\_profit)

        return Chromosome(mutated\_path,plan,phenome)

8.3 Particle Swarm Optimization

8.2.1 Dataclass

@dataclass

class Fitness:

    time: float

    profit: float

    net\_profit: float

8.2.2 Class instance

Implementation of PSO algorithm and particle behaviour.

8.3.2.1 Particle

@dataclass

class Particle:

    data: TTP

    path: List[int]

    plan: List[int]

    current\_score: Fitness = None

    velocity: List = None

    personal\_best\_path: List[int] = None

    personal\_best\_plan: List[int] = None

    personal\_best\_score: Fitness = None

    def \_\_post\_init\_\_(self):

        self.velocity = self.velocity if self.velocity is not None else []

        self.current\_score = self.calculate\_score()

        self.personal\_best\_path = self.path.copy()

        self.personal\_best\_plan = self.plan.copy()

        self.personal\_best\_score = self.current\_score

    def calculate\_score(self):

        time, profit = calculate\_time\_and\_profit(self.path, self.plan, self.data.NODE, self.data.ITEM, self.data.MIN\_SPEED, self.data.MAX\_SPEED, self.data.CAPACITY)

        net\_profit = profit - (self.data.RENTING\_RATIO \* time)

        return Fitness(time, profit, net\_profit)

    def update\_personal\_best(self):

        self.current\_score = self.calculate\_score()

        if self.current\_score.net\_profit > self.personal\_best\_score.net\_profit:

            self.personal\_best\_path = self.path.copy()

            self.personal\_best\_plan = self.plan.copy()

            self.personal\_best\_score = self.current\_score

    def update\_velocity(self, global\_best\_particle):

        # Simple velocity update logic (can be enhanced for better performance)

        for i in range(len(self.path)):

            if np.random.rand() < pbest\_prob and self.path[i] != self.personal\_best\_path[i]:

                self.velocity.append(('path', i, self.personal\_best\_path.index(self.path[i])))

            if np.random.rand() < gbest\_prob and self.path[i] != global\_best\_particle.path[i]:

                self.velocity.append(('path', i, global\_best\_particle.path.index(self.path[i])))

        for i in range(len(self.plan)):

            if np.random.rand() < pbest\_prob and self.plan[i] != self.personal\_best\_plan[i]:

                self.velocity.append(('plan', i, self.personal\_best\_plan.index(self.plan[i])))

            if np.random.rand() < gbest\_prob and self.plan[i] != global\_best\_particle.plan[i]:

                self.velocity.append(('plan', i, global\_best\_particle.plan.index(self.plan[i])))

    def apply\_velocity(self):

        for change in self.velocity:

            if change[0] == 'path':

                self.path[change[1]], self.path[change[2]] = self.path[change[2]], self.path[change[1]]

            elif change[0] == 'plan':

                self.plan[change[1]], self.plan[change[2]] = self.plan[change[2]], self.plan[change[1]]

    def clear\_velocity(self):

        self.velocity.clear()

    def flip\_multiple\_items(self, flip\_count):

        for \_ in range(flip\_count):

            if len(self.plan) > 0:

                item\_idx = np.random.choice(len(self.plan))

                self.plan[item\_idx] = 1 - self.plan[item\_idx]

8.3.2.2 PSO

class PSO:

    def \_\_init\_\_(self, data, num\_particles, iterations, gbest\_prob=1.0, pbest\_prob=1.0):

        self.data = data

        self.num\_particles = num\_particles

        self.iterations = iterations

        self.gbest\_prob = gbest\_prob

        self.pbest\_prob = pbest\_prob

        self.particles = [Particle(data, \*generate\_ttp\_solution(data.Dimension, data.ITEM, data.CAPACITY)) for \_ in range(num\_particles)]

        self.gbest\_particle = max(self.particles, key=lambda p: p.personal\_best\_score.net\_profit)

    def run(self):

        for \_ in range(self.iterations):

            for particle in self.particles:

                particle.clear\_velocity()

                particle.update\_velocity(self.gbest\_particle)

                particle.apply\_velocity()

                particle.update\_personal\_best()

                if particle.personal\_best\_score.net\_profit > self.gbest\_particle.personal\_best\_score.net\_profit:

                    self.gbest\_particle = particle

        return self.gbest\_particle.personal\_best\_path, self.gbest\_particle.personal\_best\_plan, self.gbest\_particle.personal\_best\_score

8.2.2 Extended function

Enhancing particle behaviour with unary operations and dynamic update logic.

In Particle class

def flip\_multiple\_items(self, flip\_count):

        for \_ in range(flip\_count):

            if len(self.plan) > 0:

                item\_idx = np.random.choice(len(self.plan))

                self.plan[item\_idx] = 1 - self.plan[item\_idx]

    def swap\_adjacent\_cities(self):

        if len(self.path) > 1:

            idx = np.random.randint(0, len(self.path) - 1)

            self.path[idx], self.path[idx + 1] = self.path[idx + 1], self.path[idx]

    def reverse\_subsection\_path(self):

        if len(self.path) > 2:

            start, end = sorted(np.random.choice(len(self.path), 2, replace=False))

            self.path[start:end+1] = reversed(self.path[start:end+1])

    def apply\_unary\_operator(self):

        operation = np.random.choice(['swap\_adjacent', 'flip\_item', 'reverse\_subsection'])

        if operation == 'swap\_adjacent':

            self.swap\_adjacent\_cities()

        elif operation == 'flip\_item':

            flip\_count = np.random.randint(1, len(self.plan)+1)  # Random number of items to flip

            self.flip\_multiple\_items(flip\_count)

        elif operation == 'reverse\_subsection':

            self.reverse\_subsection\_path()

def update\_velocity(self, global\_best\_particle, self\_p, pbest\_p, gbest\_p):

        rnd = np.random.rand()

        if rnd < self\_p:

            self.apply\_unary\_operator()

        elif rnd < self\_p + pbest\_p:

            # Velocity update logic towards personal best

            for i in range(len(self.path)):

                if np.random.rand() < pbest\_p and self.path[i] != self.personal\_best\_path[i]:

                    self.velocity.append(('path', i, self.personal\_best\_path.index(self.path[i])))

            for i in range(len(self.plan)):

                if np.random.rand() < pbest\_p and self.plan[i] != self.personal\_best\_plan[i]:

                    self.velocity.append(('plan', i, self.personal\_best\_plan.index(self.plan[i])))

        else:

            # Velocity update logic towards global best

            for i in range(len(self.path)):

                if np.random.rand() < gbest\_p and self.path[i] != global\_best\_particle.path[i]:

                    self.velocity.append(('path', i, global\_best\_particle.path.index(self.path[i])))

            for i in range(len(self.plan)):

                if np.random.rand() < gbest\_p and self.plan[i] != global\_best\_particle.plan[i]:

                    self.velocity.append(('plan', i, global\_best\_particle.plan.index(self.plan[i])))

In PSO class

   def update\_probabilities(self):

        # Update the probabilities as per the paper's recommendation

        self.self\_prob \*= 0.95

        self.pbest\_prob \*= 1.01

        self.gbest\_prob = 1 - (self.self\_prob + self.pbest\_prob)