21AI601

Advanced Data Structures and Algorithms

2022-20223 Odd Semester

Lab Assignment Report

Name: Shivang Modi

Roll No.: CB.EN.P2AIE22007

Topic: Scheduling Problem using Dragonfly Algorithm

TABLE OF CONTENTS

1. Category of algorithm and Application

1.1 Dragonfly Algorithm (DA)

Dragonfly Algorithm (DA) is a novel swarm optimization technique, introduced by Mirjalili in 2015 to solve several optimization problems: alike discrete, single-objective, and multi-objective problems.

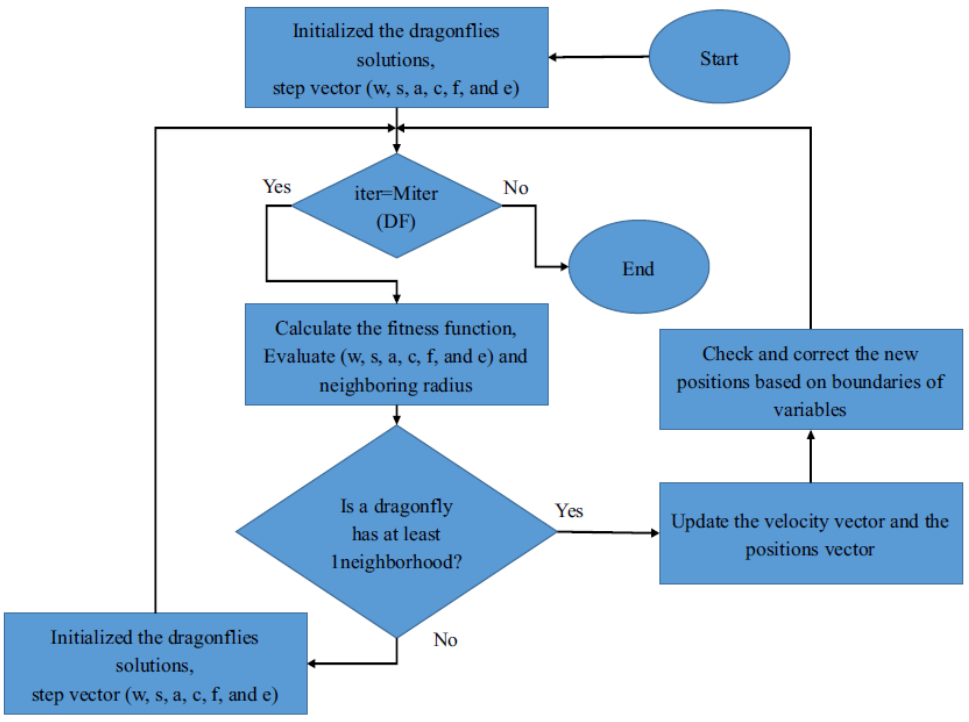

Initialized the dragonflies solutions, step vector (w, s, a, c, f, and e)
Start
Yes
iter=Miter (DF)
No
End
Calculate the fitness function, Evaluate (w, s, a, c, f, and e) and neighboring radius
Check and correct the new positions based on boundaries of variables
Is a dragonfly has at least 1neighborhood?
Yes
Update the velocity vector and the positions vector
Initialized the dragonflies solutions, step vector (w, s, a, c, f, and e)
No

Fig 1. The flowchart of the Dragonfly Algorithm

The principal inspiration for the dragonfly algorithm derived from the swarming behaviors of static dragonflies and dynamic dragonflies in nature life. Two fundamental aspects of optimization search (Exploration and Exploitation) are invented by simulating the social interplay of dragonflies in searching, exploring for foods, and avoiding foes when moving dynamically or moving statically. A conceptual illustration of swarms moving dynamically or swarms moving statistically is illustrated in Fig 2. The following subsections show the main procedure of the proposed multi-objective hybrid Dragonfly Algorithm with exploratory local search called **MHDA.** The flowchart of the proposed MHDA is provided in Fig 1.

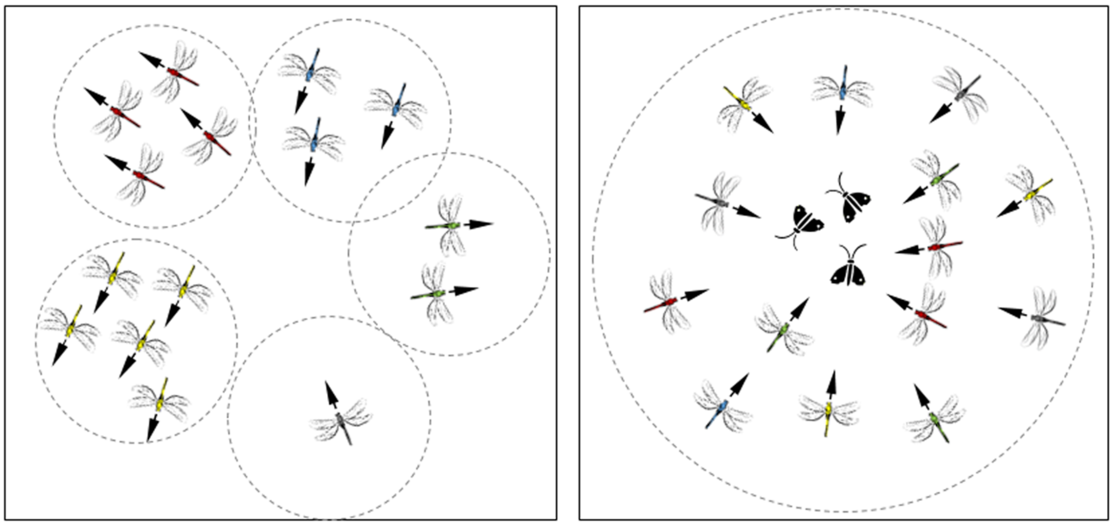


Fig 2. A conceptual illustration of swarms moving dynamically or statically

1.2 Exploration and exploitation search

The main aim of the swarm in nature-life is to survive, therefore the individuals should be interested in food sources and expel enemies outside, a dragonfly's life cycle consists of two milestones: adult and nymph. Regarding these operations, for the updating of the position (individuals), there are five main factors (i.e., separation, alignment, cohesion, attraction to food, and distraction from the enemy) in swarms as presented in Fig 3.

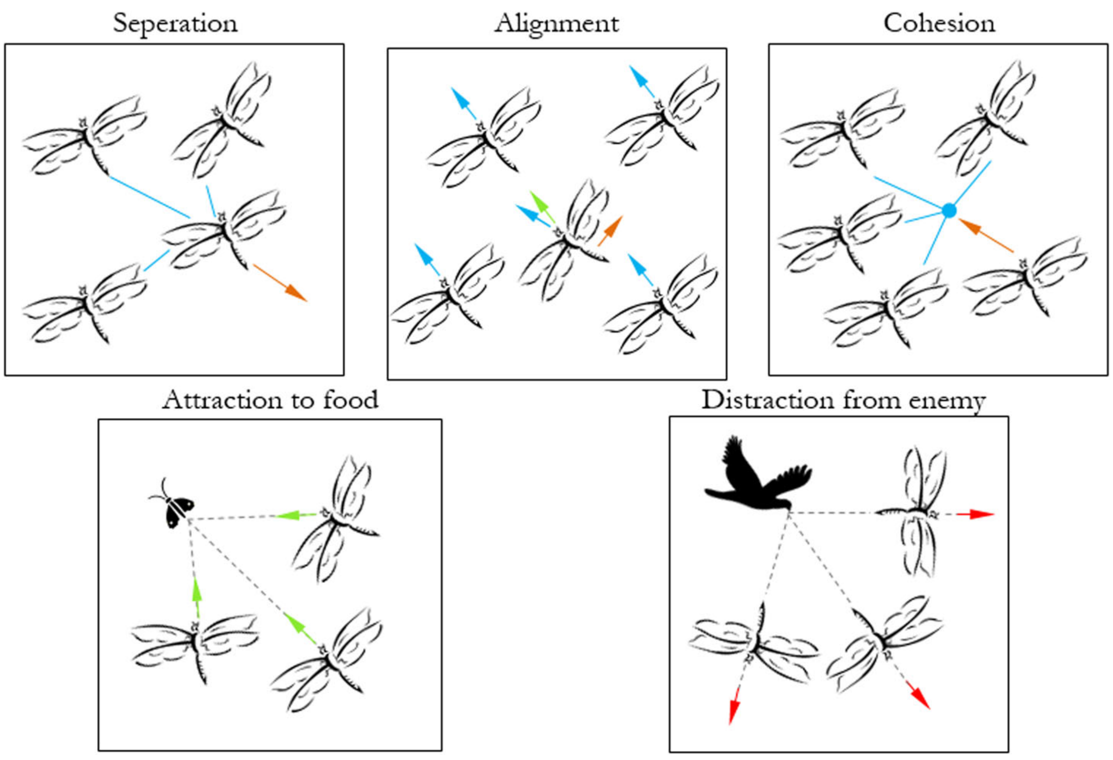


Fig 3. Five main factors in a swarm

These behaviors are mathematically modeled as follows:

1. **Separation:** $\boldsymbol{S_i = -\sum_{j=1}^{N} X - X_j}$ where X is the current position for an individual, X_j denoted to the neighboring individual of the j_{th} position, and N is denoted by the number of all current neighboring individuals.
2. **Alignment:** $\boldsymbol{A_i} = \frac{\sum_{j=1}^{N} Vj}{N}$ where V_j is denoted to the speed of j_{th} neighboring individual (velocity).
3. **Cohesion:** $\boldsymbol{C_i} = \frac{\sum_{j=1}^{N} Xj}{N} - X$ where X is denoted to the current position, N denoted to the number of all current neighborhoods, and X_j is denoted to a j_{th} neighboring individual of the current position
4. **Attraction to food:** $\boldsymbol{F_i = X^{+} - X}$, where X is the current position of the individual and X^{+}, is denoted by the position of the source food.
5. **Distraction from the enemy:** $\boldsymbol{E_i = X^{-} - X}$, where X is the current position of the individual and X^{-}, is denoted by the position of the current enemy.

1.3 Variants of the dragonfly algorithm

- Basic dragonfly algorithm
- Binary dragonfly algorithm
- Modifications of dragonfly algorithm
- Hybridizations of dragonfly algorithm
- Chaotic dragonfly algorithm
- Multi-objective dragonfly algorithm

1.4 Applications of dragonfly algorithm

- Benchmark functions
- Machine learning applications
- Optimal parameters applications
- Engineering applications
- Image processing
- Software engineering
- Network applications

2. Problem Statement

2.1 Introduction

The flexible job-shop scheduling problem (FJSP) is the classical NP-Hard problem first proposed by Brucker and Schile in 1990. FJSP evolves from JSP, which is more complex than JSP by the need to determine a routing policy. The problem of scheduling jobs in the FJSP could be decomposed into two sub-problems: the routing sub-problem that allocates a machine from alternative machines set to an operation, the scheduling subproblem that consists of sequencing the operations on all selected machines to obtain a feasible schedule to minimize the predefined objective function.
In this paper, two essential phases of dragonfly optimization - exploration and exploitation, are designed by modeling the social interaction of dragonflies in navigating, searching for food, and avoiding enemies when swarming dynamically or statistically.

2.2 Problem Description

The mathematical model of the flexible job-shop scheduling problem can be described as follows: n different jobs processing on m machines, each job has one or more operations, and each operation can be processed on one out of several machines. The determination of the scheduling scheme is the choice of the processing machine and the reasonable arrangement of the processing order of the workpiece. And optimize the performance of the evaluation index under the condition of satisfying the constraints as follows:

1. One machine can process only one operation at a time.
2. The processing process is not allowed to interrupt.
3. The processing sequence of the same job is determined.
4. Different jobs have the same priority level.

3. Algorithms to solve the problem

3.1 Description

3.1.1 Basic Dragonfly Algorithm

The basic dragonfly algorithm (BDA) is a meta-heuristic optimization algorithm inspired by the behavior of dragonflies. It is a simple yet effective algorithm that can be applied to a wide range of optimization problems, including flexible job-shop scheduling problems.

The BDA works by creating a population of solutions called dragonflies, which represent different possible schedules. Each dragonfly is randomly generated at the beginning of the algorithm, and its fitness (i.e., its quality as a solution) is evaluated by a fitness function. The algorithm then iteratively improves the dragonflies by applying local search methods to the best solutions.

In each iteration of the algorithm, a new dragonfly is generated by randomly selecting a solution from the population and making small changes to it. The new dragonfly's fitness is then evaluated, and if it is better than the previous solutions, it is added to the population. This process continues for a specified number of iterations, and at the end, the best dragonfly in the population is returned as the solution.

The BDA has some key features such as being simple, easy to implement, and doesn't require any gradient information. It also has some drawbacks such as it can get stuck in local optima, and its convergence rate may depend on the initialization and the shape of the search space.

In summary, The basic dragonfly algorithm (BDA) is a meta-heuristic optimization algorithm that uses randomness and local search to find an optimal solution, it can be applied to a wide range of optimization problems with simple implementation and easy to understand but it can also get stuck in local optima.

3.1.2 Multi-objective Dragonfly Algorithm

The multi-objective dragonfly algorithm (MODA) is a variant of the basic dragonfly algorithm (BDA) that is specifically designed to handle multi-objective optimization problems. Multi-objective optimization problems have more than one objective function to be minimized or maximized simultaneously. These problems often have multiple solutions that are considered optimal, as they provide a trade-off between the different objectives.

MODA works by creating a population of solutions, similar to the BDA, called dragonflies. Each dragonfly represents a possible schedule and is evaluated by multiple objective functions. The algorithm then iteratively improves the dragonflies by applying local search methods to the best solutions, similar to BDA. In addition, MODA uses an elite selection strategy for choosing the most promising solutions for generating new solutions, so it can escape from the local optima.

MODA is an improvement over BDA when the problem has multiple objectives because it can generate multiple solutions that tradeoff the different objectives in an acceptable way, whereas BDA will only generate one solution.

The key features of MODA include the ability to handle multi-objective optimization problems and its ability to find multiple solutions that trade-off different objectives. However, one drawback of MODA is that it may produce solutions that are dominated by other solutions, meaning they are worse in every objective than other solutions.

In summary, the multi-objective dragonfly algorithm (MODA) is a variant of the basic dragonfly algorithm (BDA) that is specifically designed for multi-objective optimization problems; it uses local search and elite selection strategy to generate multiple solutions that trade-off the different objectives. But the main drawback is that it may produce solutions that are dominated by other solutions.

3.2 Algorithm Pseudocode

3.2.1 Basic Dragonfly Algorithm

Pseudocode:-

```
def evaluate(self, job, machine):
    makespan = 0
    for j in job:
        completion_time = 0
        for task in j.tasks:
            m = machine[task[0]]
            processing_time = m.processing_times[task[1]]
            completion_time += processing_time
        makespan = max(makespan, completion_time)
    self.fitness = makespan

function BDA(population_size, num_iterations, job, machine)
    # Initialize the population of dragonflies
    population = list()
    for i in range(population_size):
        schedule = list()
        for j in job:
            schedule.append(random.sample(range(len(machine)), len(j.tasks)))
        dragonfly = Dragonfly(schedule)
        population.append(dragonfly)

    # Run the basic Dragonfly algorithm
    for i in range(num_iterations):
        for dragonfly in population:
            dragonfly.evaluate(job, machine)
        population.sort(key=lambda x: x.fitness)
        population = population[:50]
        new_dragonfly = random_dragonfly(job, machine)
        population.append(new_dragonfly)
    # Return the best schedule
    return population[0]
```

Explanation:-

- The BDA function takes three parameters: population_size, num_iterations, job, and machine, which represents the size of the population, the number of iterations the algorithm will run, the jobs to be scheduled, and the machines available for scheduling.
- The initial population of dragonflies is created by randomly generating schedules for each job, and then creating dragonflies with those schedules.
- The algorithm runs for a specified number of iterations, and in each iteration, the fitness of each dragonfly is evaluated by calling the **'evaluate()'** method.
- The population is then sorted based on the fitness of each dragonfly, and the population is truncated to keep only the best 50 dragonflies.
- A new dragonfly is then generated by randomly selecting a schedule for each job and creating a new dragonfly with that schedule.
- The new dragonfly is then added to the population, and the process repeats for the remaining number of iterations.
- Finally, the function returns the best schedule, which is the dragonfly with the highest fitness in the population.

3.2.2 Multi-objective Dragonfly Algorithm

Pseudocode:-

```
def evaluate(self, job, machine):
    makespan, tardiness = 0, 0
    for j in job:
        completion_time = 0
        for task in j.tasks:
            m = machine[task[0]]
            processing_time = m.processing_times[task[1]]
            completion_time += processing_time
        makespan = max(makespan, completion_time)
        tardiness += max(0, completion_time - j.due_date)
    self.fitness = makespan + tardiness

function MODA(population_size, num_iterations, job, machine)
    # Initialize population of dragonflies randomly
    population = list()
    for i in range(population_size):
        schedule = list()
        for j in job:
            schedule.append(random.sample(range(len(machine)), len(j.tasks)))
        dragonfly = Dragonfly(schedule)
        dragonfly.evaluate(job, machine)
        population.append(dragonfly)

    # Run MODA algorithm
    for i in range(num_iterations):
        # Sort dragonflies in non-ascending order based on fitness
        population.sort(key=lambda x: x.fitness, reverse=True)

        # Select top-ranked dragonflies as elite
        elite = population[:int(population_size / 2)]

        # Generate new dragonflies by crossover and mutation
        new_dragonflies = list()
        for j in range(int(population_size / 2)):
            parent1 = random.choice(elite)
            parent2 = random.choice(elite)
            child = list()
            for k in range(len(job)):
                if random.random() < 0.5:
                    child.append(parent1.schedule[k])
                else:
                    child.append(parent2.schedule[k])
            dragonfly = Dragonfly(child)
            new_dragonflies.append(dragonfly)

        # Evaluate fitness of new dragonflies
        for dragonfly in new_dragonflies:
            dragonfly.evaluate(job, machine)

        # Add new dragonflies to population
        population.extend(new_dragonflies)

    # Select the final scheduled
    population.sort(key=lambda x: x.fitness)
    return population[0]
```

Explanation:-

- The MODA function takes in three parameters: population_size, num_iterations, job, and machine, which represent the size of the population, the number of iterations the algorithm will run, the jobs to be scheduled, and the machines available for scheduling.
- The initial population of dragonflies is created by randomly generating schedules for each job, creating dragonflies with those schedules, and evaluating the fitness of each dragonfly using the **‘evaluate()’** method.
- The algorithm runs for a specified number of iterations. In each iteration, the dragonflies in the population are sorted based on their fitness in non-ascending order.
- Next, the top-ranked dragonflies, half the population size, are selected as elite. These elite dragonflies are the most promising solutions and will be used for generating new dragonflies.
- New dragonflies are generated by applying crossover and mutation operations on the elite dragonflies. The crossover operation is done by randomly selecting two elite dragonflies as parents and creating a child dragonfly by randomly selecting tasks from one of the two parents. The mutation operation is done by randomly selecting one task from

the child schedule and replacing it with a randomly selected task from the same job. The new schedule is used to create a new dragonfly.

- Fitness of new dragonflies is evaluated and new dragonflies are added to the population of solutions.
- After the specified number of iterations are completed the algorithm sorts the final population of dragonflies based on their fitness. The dragonfly with the best fitness is returned as the solution to the problem.

4. Complexity Analysis

4.1 Complexity analysis of BDA

The time complexity of the basic dragonfly algorithm (BDA) pseudocode depends on several factors, including the size of the population, the number of iterations, the number of jobs and the number of tasks, and the specific operations performed in the algorithm

- The outer loop of the algorithm iterates ‘num_iterations’ times, and this is the main factor determining the overall time complexity of the algorithm.
- The evaluation step is also important, in each iteration the evaluate function runs overall the jobs and tasks, this is an O(J*T) operation where J is the number of jobs and T is the number of tasks.
- The sorting operation has an O(n*log(n)) time complexity, where n is num_dragonflies.

The overall time complexity of the algorithm is then O(num_iterations * (J*T + n*log(n))).
The space complexity of the algorithm is O(num_dragonflies * (J*T)).

4.2 Complexity analysis of MODA

The time complexity of the multi-objective dragonfly algorithm (MODA) pseudocode depends on several factors, including the size of the population, the number of iterations, the number of jobs and the number of tasks, and the specific operations performed in the algorithm.

- The outer loop of the algorithm iterates ‘num_iterations’ times, and this is the main factor determining the overall time complexity of the algorithm.
- The evaluation step is also important, in each iteration the evaluate function runs overall the jobs and tasks, this is an O(J*T) operation where J is the number of jobs and T is the number of tasks.
- The sorting operation has an O(n*log(n)) time complexity, where n is num_dragonflies.

The overall time complexity of the algorithm is then O(num_iterations * (J*T + n*log(n))).
The space complexity of the algorithm is O(num_dragonflies * (J*T)).

5. Design Technique: Analysis

5.1 Design Technique analysis for the BDA

The basic dragonfly algorithm (BDA) is a type of evolutionary algorithm that is inspired by the behavior of dragonflies. The design technique of the BDA is based on the following techniques:

- **Population-based search:** BDA uses a population of solutions, represented by dragonflies, to explore the solution space. The population evolved over multiple iterations by applying genetic operators such as crossover.
- **Fitness evaluation:** The fitness of each dragonfly in the population is evaluated using a fitness function that measures how good a solution the dragonfly represents. The fitness function is problem-specific and is designed to minimize a single objective function, such as makespan or tardiness.
- **Selection:** The best dragonflies are selected for reproduction based on their fitness. The selection process can be based on various selection methods such as tournament selection, roulette wheel selection, etc…
- **Crossover:** The selected dragonflies are combined to create new dragonflies using the crossover. Crossover combines the genetic information of two-parent dragonflies to create one or more offspring dragonflies.
- **Elitism:** Some of the best solutions are preserved in the population across generations.

Overall, BDA is a type of optimization algorithm that uses a population-based search, fitness evaluation, selection, crossover, and elitism to find the best solution to a given scheduling problem.

It’s important to note that the performance of the BDA algorithm is highly dependent on specific implementation details, such as the choice of genetic operators, the size of the population, and the number of iterations. Additionally, the performance of the algorithm can also be affected by the characteristics of the problem being solved, such as the number of jobs and tasks, the complexity of the constraints, and the specific objective function.

5.2 Design technique analysis of the MODA

The multi-objective dragonfly algorithm (MODA) is an extension of the basic dragonfly algorithm (BDA) to handle multi-objective optimization problems. The design of MODA is based on the following techniques:

- **Multi-objective optimization:** MODA is designed to find a set of Pareto-optimal solutions that are optimal for multi-objectives. The algorithm uses the concept of Pareto-optimality to balance trade-offs between different objectives, rather than finding a single optimal solution.
- **Fitness evaluation:** The fitness of each dragonfly in the population is evaluated using a fitness function that measures how good a solution the

dragonfly represents. In contrast to BDA, the fitness function in MODA returns multiple objectives, such as makespan and tardiness.

- **Selection:** The best dragonflies are selected for reproduction based on their fitness. In MODA, the selection process is focused on selecting the elite solutions, which are those that are Pareto-optimal for multiple objectives.
- **Crossover:** The selected dragonflies are combined to create new dragonflies using the crossover. Crossover combines the genetic information of the two parent dragonflies to create one or more offspring dragonflies, similarly to BDA.
- **Elitism:** Some of the best solutions are preserved in the population across generations, similarly to BDA.

The multi-objective dragonfly algorithm (MODA) is an extension of the basic dragonfly algorithm (BDA) that is designed to handle multi-objective optimization problems. It uses a population-based search, fitness evaluation, selection, crossover, and elitism similar to BDA but with a few key differences.

It's important to note that the performance of the MODA algorithm is highly dependent on specific implementation details, such as the choice of genetic operators, the size of the population, and the number of iterations. Additionally, the performance of the algorithm can also be affected by the characteristics of the problem being solved, such as the number of jobs and tasks, the complexity of the constraints, and the specific objective function.

6. Data structures used for the implementation

I have used the following data structure for the implementation:

- The Job class uses a list to store the tasks, and a single variable to store the due date.
- The Machine class uses a list to store the processing times.
- The Dragonfly class uses a list to store the schedule, and a single variable to store the fitness.
- The Dragonfly Algorithm class uses a list to store the population of dragonflies and the elite dragonflies, and uses the sort method to sort the dragonflies based on their fitness values.
- Additionally, in the main function, the script uses lists to define the jobs and machines.

7. Implementation of algorithms

7.1 Code

```
import random

# Define the Job class
class Job:
  def __init__(self, tasks, due_date):
    self.tasks = tasks
    self.due_date = due_date

# Define the Machine class
class Machine:
  def __init__(self, processing_times):
    self.processing_times = processing_times

# Define the Dragonfly class
class Dragonfly:
  def __init__(self, schedule):
    self.schedule = schedule
    self.fitness = 0

  def evaluate(self, job, machine):
    makespan, tardiness = 0, 0
    for j in job:
      completion_time = 0
      for task in j.tasks:
        m = machine[task[0]]
        processing_time = m.processing_times[task[1]]
        completion_time += processing_time
      makespan = max(makespan, completion_time)
      tardiness += max(0, completion_time - j.due_date)
    self.fitness = (makespan, makespan + tardiness)

# Define the function to solve the flexible job-shop scheduling problem
class DragonFlyAlgorithm:
  @staticmethod
  def bda(job, machine, num_dragonflies, num_iterations):
    # Initialize the population of dragonflies
    population = list()
    for i in range(num_dragonflies):
      schedule = list()
      for j in job:
```

```python
            schedule.append(random.sample(range(len(machine)), len(j.tasks)))
        dragonfly = Dragonfly(schedule)
        population.append(dragonfly)

    # Run the basic Dragonfly algorithm
    for i in range(num_iterations):
        for dragonfly in population:
            dragonfly.evaluate(job, machine)
        population.sort(key=lambda x: x.fitness[0])
        population = population[:50]

    # Return the best schedule
    return population[0]

@staticmethod
def moda(job, machine, num_dragonflies, num_iterations):
    # Initialize population of dragonflies randomly
    population = list()
    for i in range(num_dragonflies):
        schedule = list()
        for j in job:
            schedule.append(random.sample(range(len(machine)), len(j.tasks)))
        dragonfly = Dragonfly(schedule)
        dragonfly.evaluate(job, machine)
        population.append(dragonfly)

    # Run MODA algorithm
    for _ in range(num_iterations):
        # Sort dragonflies in non-ascending order based on fitness
        population.sort(key=lambda x: x.fitness[1], reverse=True)

        # Select top-ranked dragonflies as elite
        elite = population[:int(num_dragonflies / 2)]

        # Generate new dragonflies by crossover and mutation
        new_dragonflies = list()
        for i in range(int(num_dragonflies / 2)):
            parent1 = random.choice(elite)
            parent2 = random.choice(elite)
            child = list()
            for j in range(len(job)):
                if random.random() < 0.5:
                    child.append(parent1.schedule[j])
                else:
                    child.append(parent2.schedule[j])
            dragonfly = Dragonfly(child)
            new_dragonflies.append(dragonfly)

        # Evaluate fitness of new dragonflies
        for dragonfly in new_dragonflies:
            dragonfly.evaluate(job, machine)

        # Add new dragonflies to population
        population.extend(new_dragonflies)

    # Select final schedule
    population.sort(key=lambda x: x.fitness[1])
    return population[0]
```

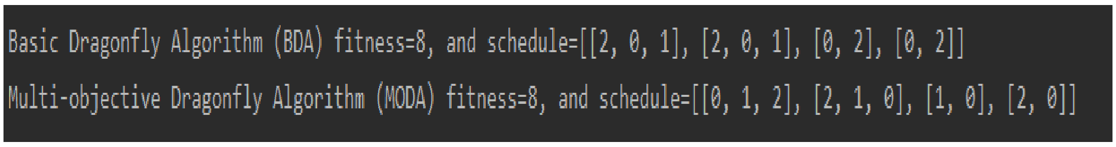
7.2 Sample Input

```
if __name__ == "__main__":
    # Define jobs and machines
    jobs = [Job([(0, 3), (1, 2), (2, 2)], 10),
            Job([(0, 1), (2, 1), (1, 3)], 8),
            Job([(1, 3), (2, 1)], 10),
            Job([(0, 2), (1, 1)], 5)]
    machines = [Machine([3, 2, 1, 3]),
                Machine([3, 1, 3, 2]),
                Machine([2, 3, 2, 1])]

    # Run BDA algorithm
    da = DragonFlyAlgorithm()
    bda = da.bda(jobs, machines, 10, 100)
    moda = da.moda(jobs, machines, 10, 100)

    # Print final schedule
        print(f'Basic Dragonfly Algorithm (BDA) fitness={bda.fitness[0]}, and
schedule={bda.schedule}')
print(f'Multi-objective Dragonfly Algorithm (MODA) fitness={moda.fitness[1]}, and
schedule={moda.schedule}')
```

7.3 Sample Output



```
Basic Dragonfly Algorithm (BDA) fitness=8, and schedule=[[2, 0, 1], [2, 0, 1], [0, 2], [0, 2]]
Multi-objective Dragonfly Algorithm (MODA) fitness=8, and schedule=[[0, 1, 2], [2, 1, 0], [1, 0], [2, 0]]
```

7.4 Running time measurement for the best case, the average case, and the worst case

The running time of the above program would depend on the specific input values and the parameters of the algorithm (such as the number of jobs, machines, dragonflies, and iterations). In general, the running time of the bda() method and the moda() method would be affected by the following factors:

- **The number of jobs and tasks:** The more jobs and tasks there are the more schedules the algorithm needs to generate, and evaluate, which would increase the running time.
- **The number of machines:** The more machines there are, the more possibilities there are for each task in the schedule, which would increase the running time.
- **The number of dragonflies:** The more dragonflies there are, the more schedules the algorithm needs to generate and evaluate, which would increase the running time.
- **The number of iterations:** The more iterations the algorithm runs, the more chances the algorithm has to find the optimal schedule, but it also increases the running time.

However, I have a rough idea of the running time of the algorithm:

- **Best case:** In the best case scenario, the algorithm finds the optimal schedule very quickly, and the running time would be determined by the time it takes to evaluate the first few schedules and the time it takes to sort the population of dragonflies.
- **Average case:** In the average case, the algorithm takes a moderate amount of time to find the optimal schedule. The running time would be determined by the number of iterations needed to find the optimal schedule, and the time it takes to evaluate and sort the population of dragonflies.
- **Worst case:** In the worst-case scenario, the algorithm takes a long time to find the optimal schedule. This could happen if the number of jobs, tasks, and machines is very large, the number of iterations is very large, or the algorithm gets stuck in a local optimum. In this case, the running time would be determined by the number of iterations and the time it takes to evaluate and sort the population of dragonflies.

It is worth nothing if the number of machines is much larger than the number of tasks, the dragonfly algorithm's running time would be much more efficient than other scheduling algorithms.

8. References

I have referred following research paper for the implementation:

- https://link.springer.com/article/10.1007/s00521-015-1920-1
- https://ieeexplore.ieee.org/document/9337684