In [1]:

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
```

In [2]:

```
class Graph:
    def __init__(self, n_cities):
        self.n_cities = n_cities
        self.routes = {}
        self.cities = list(range(n_cities))
        self.init_graph()

    def exists(self, src, dest):
        if (src,dest) in self.routes:
            return True
        return False

    def init_graph(self):
        for i in range(self.n_cities):
            for j in range(self.n_cities):
                if i!=j:
                    cost = random.randint(10, 100)
                    self.add_edge(i, j, cost)

    def add_edge(self, src, dest, cost):
        self.routes[(src,dest)] = cost

    def display_graph(self):
        for route in self.routes:
            print(f"{route[0]} -> {route[1]}: {self.routes[route]}")

    def get_cost(self, src, dest):
        if (src,dest) in self.routes:
            return self.routes[(src,dest)]

    def generate_random_path(self, population):
        while True:
            sample = random.sample(self.cities, self.n_cities)
            if sample not in population:
                return sample
```

In [3]:

```
class Particle:
    def __init__(self, position):
        self.position = position
        self.fitness = 0
        self.best_position = self.position
        self.best_fitness = self.fitness
        self.is_best = True

    # Evaluate fitness of the particle and set best fitness and position(local min)
    def evaluate_fitness(self, graph):
        cost = 0
        for i in range(0,len(self.position)-1):
            cost += graph.get_cost(self.position[i], self.position[i+1])
        self.fitness = math.inf if cost == 0 else 1/cost
        if self.fitness > self.best_fitness:
            self.best_fitness = self.fitness
            self.best_position = self.position
            self.is_best = True
        else:
            self.is_best = False
```

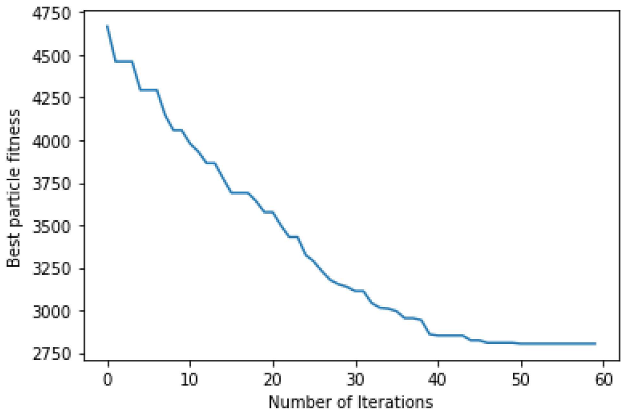
In [4]:

```
class Swarm:
    def __init__(self, options, graph, max_population):
        self.dimension = graph.n_cities
        self.n_particles = max_population
        self.population = []
        self.best_particle = None
        self.options = options
        self.graph = graph
        self.fitness_graph = []
        self.generate_population()

    # Generate initial swarm population
    def generate_population(self):
        self.population = []
        for i in range(self.n_particles):
            position = self.graph.generate_random_path(self.population)
            self.population.append(Particle(position))
        self.best_particle = random.choice(self.population)

    # Evaluate Fitness of the swarm and set best pasition of the swarm (global max)
    def evaluate_fitness(self):
        for particle in self.population:
            particle.evaluate_fitness(self.graph)
            if particle.is_best:
                if particle.fitness > self.best_particle.fitness:
                    self.best_particle = particle
        self.fitness_graph.append(1/self.best_particle.fitness)
        print(f"\nGlobal Best Particle: {self.best_particle.position}, Fitness: {1/self.best_particle.fitness}")

    # Function to implement position swaps
    def swap(self, best, current, probability):
        for i in range(len(best)):
            if best[i] != current[i]:
                if probability >= np.random.uniform(0,1):
                    swap_index = best.index(current[i])
                    current[i], current[swap_index] = current[swap_index], current[i]
        return current

    # Update position of the particles in the swarm
    def update_swarm(self):
        global_best_position = self.best_particle.position
        for particle in self.population:
            local_best_solution = particle.best_position
            if not particle.is_best:
                particle.position = self.swap(local_best_solution, particle.position, self.options['alpha'])
            if global_best_position != particle.position:
                particle.position = self.swap(global_best_position, particle.position, self.options['beta'])

    # Check for termination
    def terminate(self):
        if len(self.fitness_graph) > 10:
            if len(set(self.fitness_graph[-10:])) == 1:
                return True
        return False

    # Plotting graph to show fitness trend
    def plot_graph(self):
        plt.plot(self.fitness_graph)
        plt.xlabel("Number of Iterations")
        plt.ylabel("Best particle fitness")
```

In [5]:
```
def optimize(max_iter=100, max_population=1000):
    graph = Graph(100)
    options = { 'alpha': 0.8, 'beta': 0.2 }
    swarm = Swarm(options, graph, max_population)
    for i in range(max_iter):
        swarm.evaluate_fitness()
        if swarm.terminate():
            break
        swarm.update_swarm()
    swarm.plot_graph()
```

In [6]:
```
%%time
optimize()
```

```
5, 83, 66, 5, 55, 45, 9, 54, 34, 85, 64, 22, 82, 36, 72, 15, 92, 24, 32, 4, 63, 48, 14, 47, 16, 73, 76, 91, 0, 60, 65, 59, 74, 96, 17, 6, 25, 1, 51, 3
1, 70, 27, 2, 44, 8, 58, 77, 23, 56, 86, 69, 99, 33, 10, 29, 88, 38, 18, 40, 42, 68, 19, 28, 49, 30, 57, 78, 93], Fitness: 2804.0
Wall time: 12.7 s
```



In [ ]: