SOLVING NP-HARD PROBLEMS USING AGENT-BASED TECHNIQUES AND OPTIMIZATION ALGORITHMS

Chandraveer Teeluck (Team Leader)	***2116730***
Yashvin Bhayraw	***2114853***
Esaie Moos	***2114762***
Aftab GoolamHossen	***2113401***

Table of Contents

	Esaie Moos	Yashvin Bhayraw	Aftab GoolamHossen	Abhi Teeluck
Selection of NP-Hard Problem				
Understanding the problem				
Agent-Based Techniques				
Optimization Algorithms				
Proposed Solution Approach & Implementation using Google Colab				
Experimental Evaluation				
Discussion and Conclusion				

1. Selection of NP-Hard Problem

Chosen Problem: Traveling Salesman Problem

The Traveling Salesman Problem(TSP) is a classic NP-Hard problem in combinatorial optimization. It is concerned with finding the shortest possible route that visits each city exactly once and returns to the original city. Despite its seemingly simple formulation, the TSP is known for its computational complexity as the number of possible solutions grows factorially with the number of cities.

2. Understanding the problem

The TSP can be formally and simply defined as: It is given a list of cities and the distances between each pair of cities, and the task is to find the shortest possible tour that visits each city exactly once and returns to the original city.

Constraints:

- Each city must be visited exactly once.
- The journey must start and end at the same city.
- The problem assumes a complete graph, which means there is a connection between every pair of cities.

Practical Applications

Manufacturing: To minimise downtime, maximise throughput, and cut costs, manufacturing businesses need to schedule production efficiently. A manufacturing line's task sequence can be optimised by using the TSP to make sure that each step is finished in the most effective order possible. Manufacturers can increase their operational efficiency, better use their resources, and become more competitive in the market by addressing the TSP.

Delivery and Logistics: Optimising delivery truck routes is a daily task for corporations in the delivery and logistics domain. The TSP comes into play in determining the best order for their trucks to stop in order to reduce fuel consumption, trip time, and total operating expenses. These businesses can increase the effectiveness of their distribution networks and deliver goods faster while also increasing customer satisfaction by solving the TSP.

Computer Chip Design: Computer chip design involves routing connections between different components on a chip, such as processors, memory modules, and input/output ports. The TSP can be utilized to find the most efficient layout for these connections, minimizing the distance between components and optimizing signal propagation times. By solving the TSP, chip designers can improve the performance and reliability of their designs, leading to faster and more energy-efficient electronic devices.

Urban Planning: Urban planners face the challenge of optimizing various aspects of city infrastructure, including transportation networks, waste management systems, and public service delivery. The TSP can be applied to optimize routes for garbage collection trucks, public transit vehicles, and emergency response services, ensuring efficient resource allocation and reducing congestion and pollution in urban areas. By solving the TSP, urban planners can create more sustainable and livable cities for residents and visitors alike.

3. Agent-Based Techniques

Agent-based techniques are all about breaking down complex systems into smaller, autonomous agents that interact with each other and their environment. These agents follow rules or behaviors and can make decisions on their own. When it comes to problem-solving, we can use agent-based techniques to simulate intelligent agents searching for solutions.

Agent-based techniques offer several impactful roles for solving NP-hard problems like the TSP:

- **Distributed Problem-Solving**: Agents can work independently or collaboratively to explore different parts of the solution space, allowing for parallel and distributed computation.
- **Scalability**: Agent-based approaches can scale well to large problem instances by decentralizing the problem-solving process and leveraging parallelism.
- **Adaptability**: Agents can adapt their behaviour based on local information and interact dynamically with the environment, potentially leading to more flexible and robust solutions.

Advantages

1. **Cost Savings**: Businesses can save a lot of money by putting AI bots into use. Organisations can attain increased operational efficiency through the optimisation of resource allocation and the reduction of labour expenses. Additionally, AI agents reduce the possibility of human error, which can result in expensive blunders. By integrating AI agents, industries including manufacturing, shipping, and customer service have experienced significant cost reductions.
2. **Flexibility**: ABM offers a naturally occurring framework for adjusting the agents' complexity in terms of behaviour, level of reason, capacity for learning and evolution, and interaction rules. The capacity to switch between multiple levels of aggregation and description is another aspect of flexibility. It is simple to experiment with aggregate agents,

subgroups of agents, and single agents, all of which live in a particular model at different levels of description.

Limitations

1. **Validation**: Validating and calibrating agent-based models can be challenging due to the inherent complexity and non-linearity of agent interactions. Ensuring that the behaviors and interactions of agents accurately reflect real-world phenomena requires extensive empirical data and experimentation.

2. **Computational Overhead**: Agent-based modeling can impose a significant computational overhead, especially when dealing with large numbers of agents or complex interactions. Each agent operates independently, requiring computational resources to process its perception, decision-making, and interactions with other agents and the environment. As the number of agents increases, the computational complexity of the model grows, potentially leading to scalability issues and increased computational requirements.

4. Optimization Algorithms

4.1 Genetic Algorithm (GA)

Genetic Algorithms are searching algorithms that use the concepts of natural selection and heredity. GA is a sub-component of evolutionary computation. GA can be used to find the optimal solution instantly and with a very high accuracy. The ability of genetic algorithm is to explore large spaces of solutions effectively, perform global search, maintains diversity, flexible to different problem characteristics makes them well-suited for solving NP-Hard problems.

4.1.2 Operations performed in GA

1. Selection (Encoding of a chromosome)

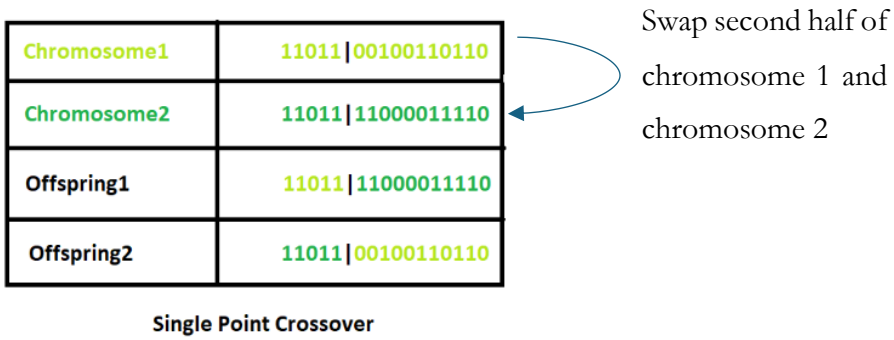
A chromosome will be in a format that will contain data concerning to the solution that it represents. The most used methodology is binary string format. After binary string format used, the chromosome will look like this:

Chromosome 1 1101100100110110
Chromosome 2 1101111000011110

Every one of the chromosomes can be plot by a binary string. Each bit contains some aspect of the solution. Individuals from the population are chosen to participate in the reproduction process, with the fittest individuals having a higher chance of being selected.

2. Crossover

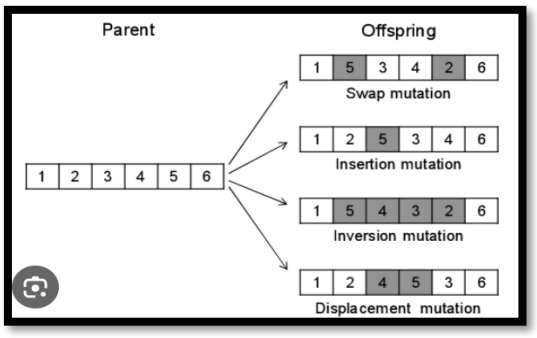
Once the individuals have been chosen, pairs of individuals are chosen to undergo crossover/recombination.



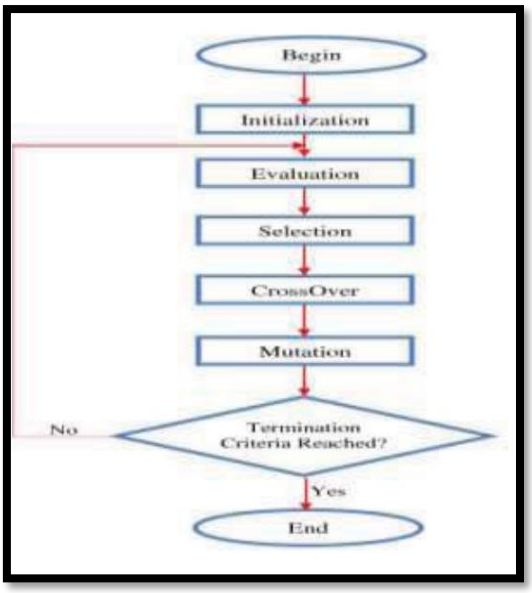
Single Point Crossover

3. Mutation

Mutation is performed to prevent the falling of all solution in the population into a local optimum of the solved problem. Offspring results from crossover randomly changes by mutation operation.



Below you will find a flowchart diagram that explains the mechanism of GA:



5. Proposed Solution Approach & Implementation using Google Colab

The proposed solution approach selected is using the Genetic Algorithm along with a model-based agent to solve the Travelling Salesman Problem. Below you will find the implementation, detailed explanation, and integration of the model-based agent in the GA optimization algorithm to solve the TSP in google colaboratory. The code is well documented to get a clear understanding of how the algorithm works.

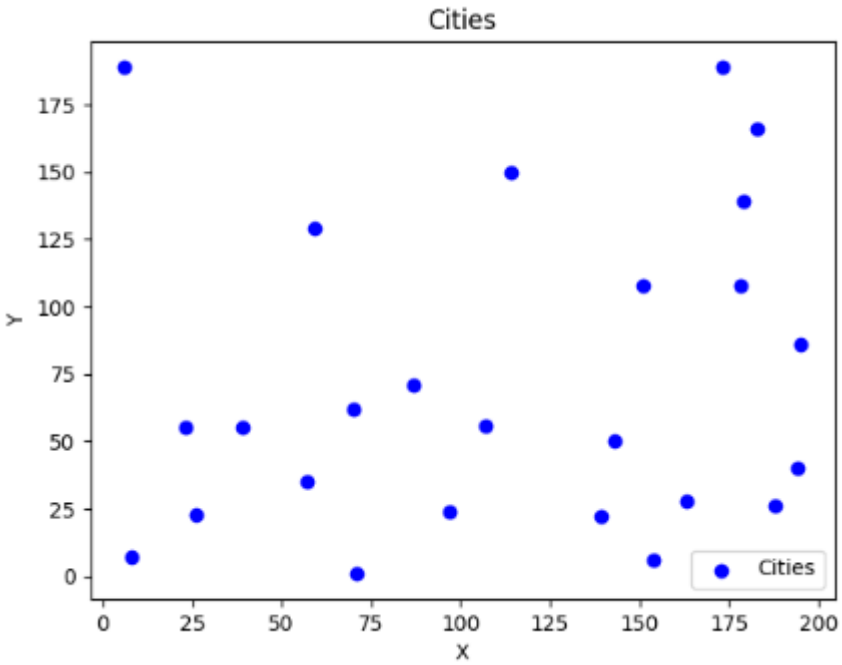
Link: https://colab.research.google.com/drive/1CRrLgrJ57AXL33M0vVqPcQrdWPAqzpbV

```python
from random import randint, random, sample, seed
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt

'''The class city represents individual cities. Each city
has an X and Y coordinates.'''
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

'''The agent class is used to represent potential solutions for the TSP.
Each agent has 2 attributes. The attribute route will store the path and the
order in which the agent visits each cities. The attribute fitness will store the quality of the solution'''
class Agent:
    def __init__(self, city_list):
        self.route = sample(city_list, len(city_list))
        self.fitness = self.calculate_fitness()

    '''This function will calculate the total distance travelled between cities when the
    agent visits from one city to another. This function will also calculate the distance when returning to
    the starting city from the end city and add it to the total distance travelled. The fitness of the solution will then be
    determined by the reciprocal of total_distance'''

    def calculate_fitness(self):
        total_distance = sum(self.calculate_distance(self.route[i], self.route[i + 1]) for i in range(len(self.route) - 1))
        total_distance += self.calculate_distance(self.route[-1], self.route[0])
        return 1 / total_distance

    def calculate_distance(self, city1, city2):
        return sqrt((city2.x - city1.x) ** 2 + (city2.y - city1.y) ** 2)
```

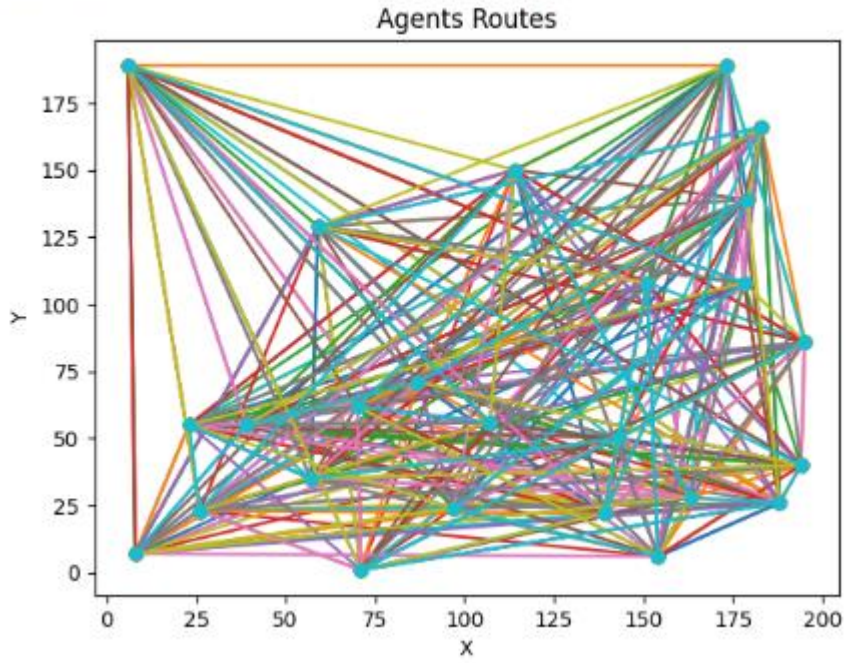
```
'''The class Mappy initializes the number of agents in the population
and initializes a list of cities representing the cities in the TSP '''
class Mappy:
    def __init__(self, population_num, city_list):
        self.city_list = city_list # Initializing the list named city_list.
        self.agents = [Agent(city_list) for _ in range(population_num)] # Creating multiple agent objects each assigned with a given city list.
        self.population_num = population_num # Initializing the population.
        self.best_agent = max(self.agents, key=lambda agent: agent.fitness) # Identifies the agent with the highest value of fitness

    def display_best_solution(self):
      print("Best Solution:")
      print("Path:", [city.__dict__ for city in self.best_agent.route])
      print("Fitness:", self.best_agent.fitness)

    def execute(self, generations):
        best_fitnesses = []

        # Plot initial cities
        self.plot_cities()

        for generation in range(generations):
            self.sort_agents() # Sorts the agents based on their fitness value in descending order.
            self.update_best_agent() # Updates the "best_agent" attribute to reflect the agent with the highest fitness after sorting.
            best_fitnesses.append(self.best_agent.fitness) # Adds the agent with the best fitness to the best_fitnesses list.

            # Prints the generation number and the fitness of the best agent for tracking progress.
            print(f"Generation {generation + 1} - Best Fitness: {self.best_agent.fitness}")

            # Perform agent interactions (local search)
            for agent in self.agents:
                self.local_search(agent)

            # Perform crossover to generate offspring
            self.nextGeneration()

            # Plot agents' routes
            self.plot_agents_routes()

        self.plot_fitness_progress(best_fitnesses)
        self.display_best_solution()

    def sort_agents(self):
        self.agents.sort(key=lambda agent: agent.fitness, reverse=True)

    def update_best_agent(self):
        self.best_agent = max(self.agents, key=lambda agent: agent.fitness)
```

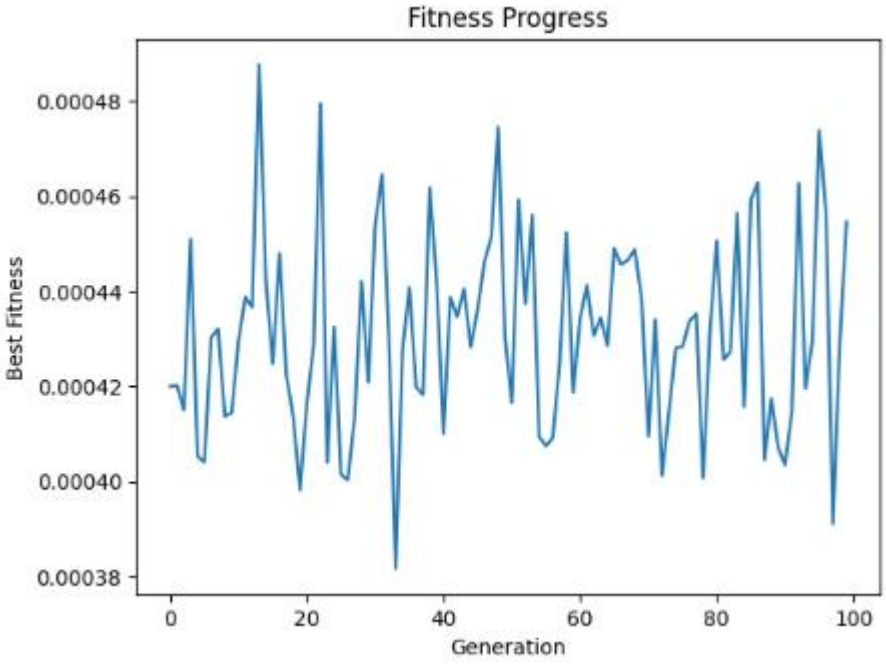
```
# This is the mutation function. Swap mutation occurs between two selected positions in agent's path and new fitness value is calculated.
def local_search(self, agent):
    pos1, pos2 = sorted(sample(range(len(agent.route)), 2))
    agent.route[pos1:pos2+1] = reversed(agent.route[pos1:pos2+1])
    agent.fitness = agent.calculate_fitness()

'''The roulette wheel selection method selects two parents based on their fitness value.'''
def rouletteWheelSelection(self):
    parents = [] # Two parents
    total = 0
    roulette = []
    for i in range(len(self.agents)):
        total_fit = sum(agent.fitness for agent in self.agents)
        portion = self.agents[i].fitness / total_fit
        roulette.append((i, total, total + portion))
        total += portion
    spin = random()
    picked = [i for i in roulette if i[1] <= spin < i[2]]
    spin2 = random()
    picked2 = [i for i in roulette if i[1] <= spin2 < i[2]]
    while picked[0][0] == picked2[0][0]:
        spin2 = random()
        picked2 = [i for i in roulette if i[1] <= spin2 < i[2]]
    parents.append(self.agents[picked[0][0]])
    parents.append(self.agents[picked2[0][0]])
    return parents

'''Once two parents have been selected by using the roulette wheel selection algorithm, the crossover method
performs one-point crossover operation with the two parents to produce two offsprings.'''
def crossover(self, parent1, parent2):
    crossover_point = randint(1, len(parent1.route) - 1)
    child1_route = parent1.route[:crossover_point] + [city for city in parent2.route if city not in parent1.route[:crossover_point]]
    child2_route = parent2.route[:crossover_point] + [city for city in parent1.route if city not in parent2.route[:crossover_point]]
    child1 = Agent(child1_route)
    child2 = Agent(child2_route)
    return child1, child2

'''After the crossover, the two offsprings are added in the next generation of solutions.'''
def nextGeneration(self):
    nextGen = []
    for _ in range(self.population_num // 2):
        parent1, parent2 = self.rouletteWheelSelection()
        child1, child2 = self.crossover(parent1, parent2)
        nextGen.append(child1)
        nextGen.append(child2)
    self.agents = nextGen

def plot_cities(self):
    plt.scatter([city.x for city in self.city_list], [city.y for city in self.city_list], color='blue', label='Cities')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Cities')
    plt.legend()
    plt.show()
```

```
    def plot_agents_routes(self):
        for agent in self.agents:
            agent_route_x = [city.x for city in agent.route]
            agent_route_y = [city.y for city in agent.route]
            plt.plot(agent_route_x, agent_route_y, marker='o')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('Agents Routes')
        plt.show()

    def plot_fitness_progress(self, best_fitnesses):
        plt.plot(range(len(best_fitnesses)), best_fitnesses)
        plt.xlabel('Generation')
        plt.ylabel('Best Fitness')
        plt.title('Fitness Progress')
        plt.show()

# Example usage
seed()
city_list = [City(randint(0, 200), randint(0, 200)) for _ in range(25)]
tsp_solver = Mappy(population_num=20, city_list=city_list)
tsp_solver.execute(generations=100)
```



```
Generation 1 - Best Fitness: 0.0004199341230214139
```





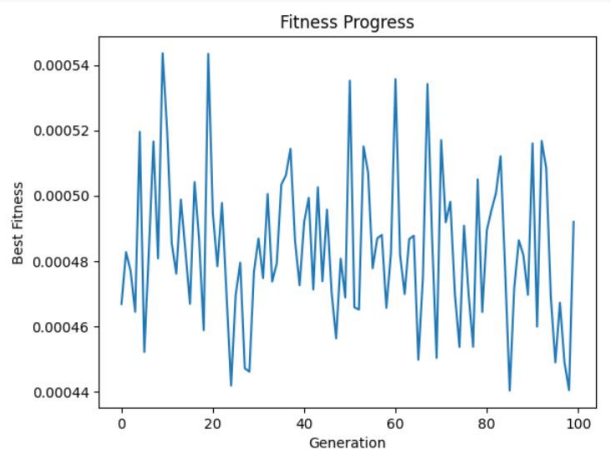

```
Best Solution:
Path: [{'x': 59, 'y': 129}, {'x': 26, 'y': 23},
Fitness: 0.00047586530833445954
```

7. Experimental Evaluation

7.1. Adjusting the population num parameter

Increasing the population parameter:

1) The chance to find high-quality fitness values increases.
2) More exploration space to find high-quality solutions (Fitness).
3) Longer converging and execution time to near-optimal solution.
4) More computer memory is required to store and evaluate a larger number of agents in each generation.
5) Reduces the risk of getting stuck in local optima.



When the population parameter has changed from 20 to 25.

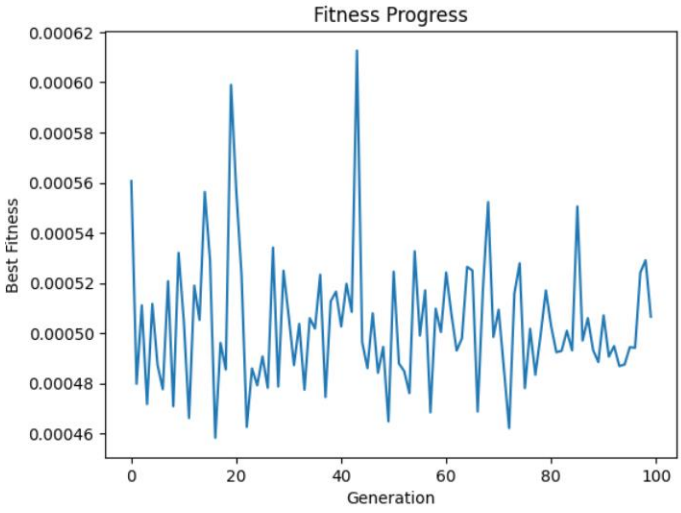
```
Best Solution:
Path: [{'x': 78, 'y': 170}, {'x
Fitness: 0.0005427638838872731
```

Execution time:

✓ 1m 6s completed at 09:50

Decreasing the population parameter:

1) Leads to faster convergence.
2) Decrease in computational time.
3) Reduce risk of getting stuck in local optima.



When the population parameter has changed from 25 to 10.

```
Best Solution:
Path: [{'x': 98, 'y': 118}, {':
Fitness: 0.0004954057791265466
```

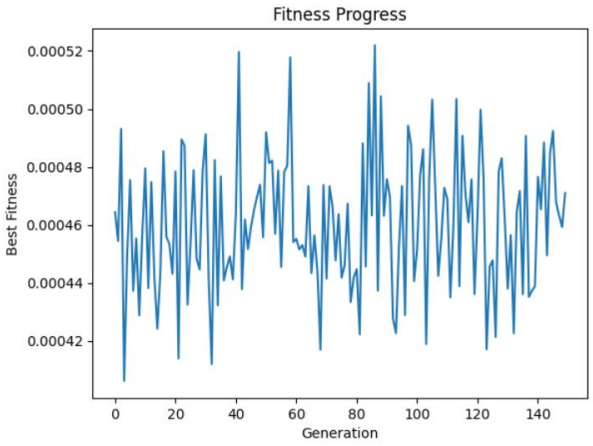
Execution time:

✓ 57s completed at 10:04

7.2. Adjusting the number of generations

Increasing the number of generations:

1) More exploration of the solution space performed by the agents.
2) Leads to better quality solutions (Fitness).
3) Larger execution and convergence time to near-optimal solution.
4) Reduce risk of getting stuck in local optima.
5) More space and memory needed to execute.



When the number of generations has changed from 100 to 150.

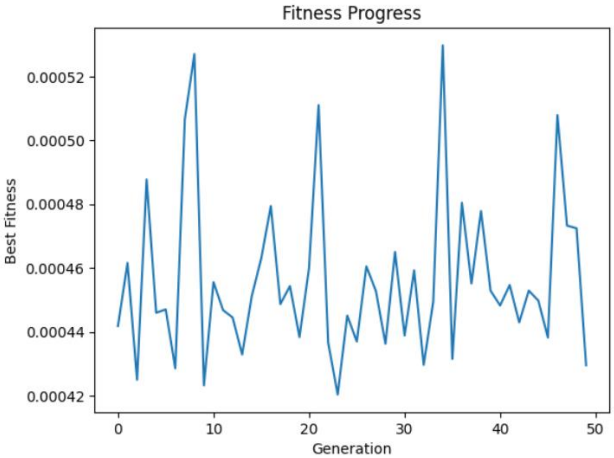
```
Best Solution:
Path: [{'x': 172, 'y': 22}, {'
Fitness: 0.0004679081612835522
```

Execution time:

✓ 1m 53s completed at 10:20

Decreasing the number of generations:

1) Reduces the algorithm ability to explore the solution space well. Faster convergence to suboptimal solutions.
2) Faster execution time.
3) Lower solution quality.
4) Low memory required to execute.
5) Increases the risk of getting stuck in a local optima.



When the number of generations has changed from 150 to 50.

```
Best Solution:
Path: [{'x': 101, 'y': 184}, {'
Fitness: 0.00043246041194336437
```

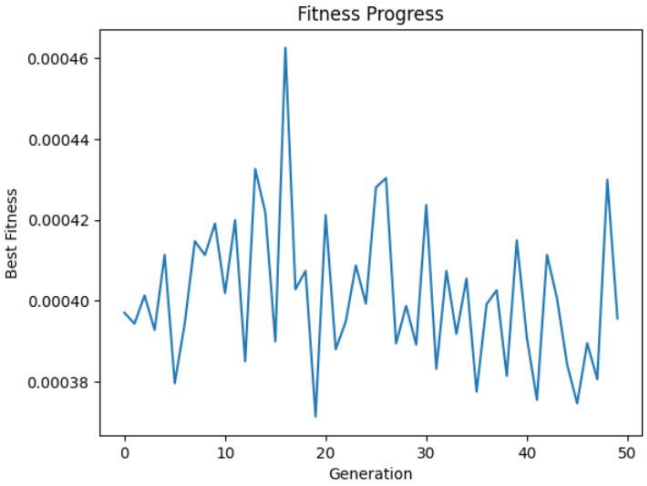
Execution time:

✓ 28s completed at 10:25

7.3. Adjusting the number of cities

Increasing the number of cities:

1) More cities to visit hence increasing the complexity of the algorithm.
2) Longer execution and convergence time to near-optimal solution.
3) More exploration space.
4) Better quality solutions (Fitness values).
5) More paths to visit. More memory is needed to evaluate more possible solutions.



When the number of cities has changed from 25 to 30.

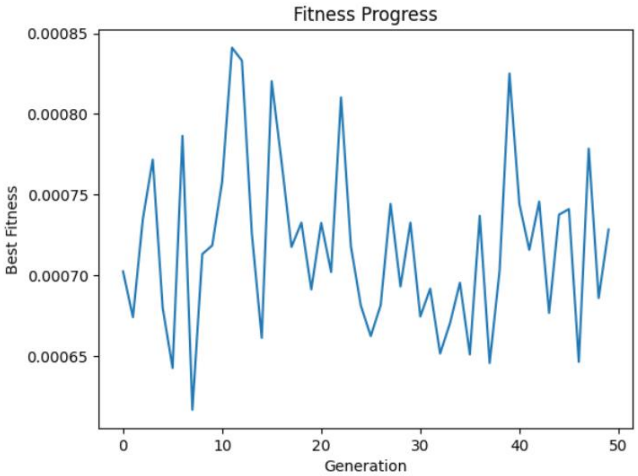
```
Best Solution:
Path: [{'x': 107, 'y': 145}, {'
Fitness: 0.0003993511890243511
```

Execution time:

✓ 25s completed at 10:35

Decreasing the number of cities:

1) Less cities to visit hence decreasing the complexity of the algorithm.
2) Less execution and convergence time to near-optimal solution.
3) Less exploration space.
4) Low quality solutions (Fitness values).
5) Less paths to visit. Less memory is needed to evaluate more possible solutions.



When number of cities changed from 30 to 15.

```
Best Solution:
Path: [{'x': 147, 'y': 46}, {
Fitness: 0.000613272780031008
```

Execution time:

✓ 23s completed at 10:44

8. Discussion and Conclusion

Our approach to solving one of the most well-known NP-hard issues in combinatorial optimisation, the Travelling Salesman Problem (TSP), is complex and involves the use of both optimization algorithms and agent-based strategies.

Promising outcomes have been demonstrated when optimisation methods such as Genetic methods (GAs) are combined with agent-based techniques that mimic autonomous agents interacting with their surroundings. Our solution strategy makes use of the advantages of both approaches to effectively traverse the TSP's large solution space.

Through our experimentation and analysis, several key observations have emerged:

1. **Performance Trade-offs**: By varying factors like the population size, the number of generations, and the number of cities, trade-offs between computational resources, convergence speed, and solution quality have been identified. For example, larger populations and more generations typically result in higher-quality solutions, but at the expense of larger memory and processing time require

2. **Scalability**: Our method has proven to be scalable, especially when it comes to the quantity of cities included in the TSP instance. Even though bigger problem instances tend to be more complex, our solution strategy demonstrates flexibility in managing these kinds of difficulties.

3. **Robustness**: Agent-based techniques offer diverse problem-solving skills and adaptability, which enhance the resilience of our solution approach. This resilience is necessary for successfully negotiating dynamic, complicated problem situations.

4. **Practical Implications**: This solution approach's effective implementation for the TSP has important real-world ramifications in a number of areas, such as computer chip design, manufacturing, logistics, and urban planning. Businesses and organisations can cut costs, increase overall productivity, and improve operational efficiency by optimising routing and resource allocation.
   para

While our approach has demonstrated promise, there are areas for further exploration and improvement:

1. **Parameter Tuning**: The convergence rate and solution quality may be further improved by fine-tuning GA parameters including mutation rate, crossover rate, and selection processes.

2. **Real-World Validation**: Conducting real-world validation of our solution approach by applying it to large-scale, practical TSP instances would provide valuable insights into its effectiveness and applicability in real-world scenarios.

Conclusion

In summary, the TSP serves as an example of how this investigation into the use of agent-based approaches and optimisation algorithms to solve NP-hard problems has demonstrated the value of combining various approaches to address challenging computational issues. We have set the foundation for future study and useful applications in solving a variety of NP-hard problems by utilising the advantages of agent-based modelling and optimisation techniques.