The Traveling Sales Person

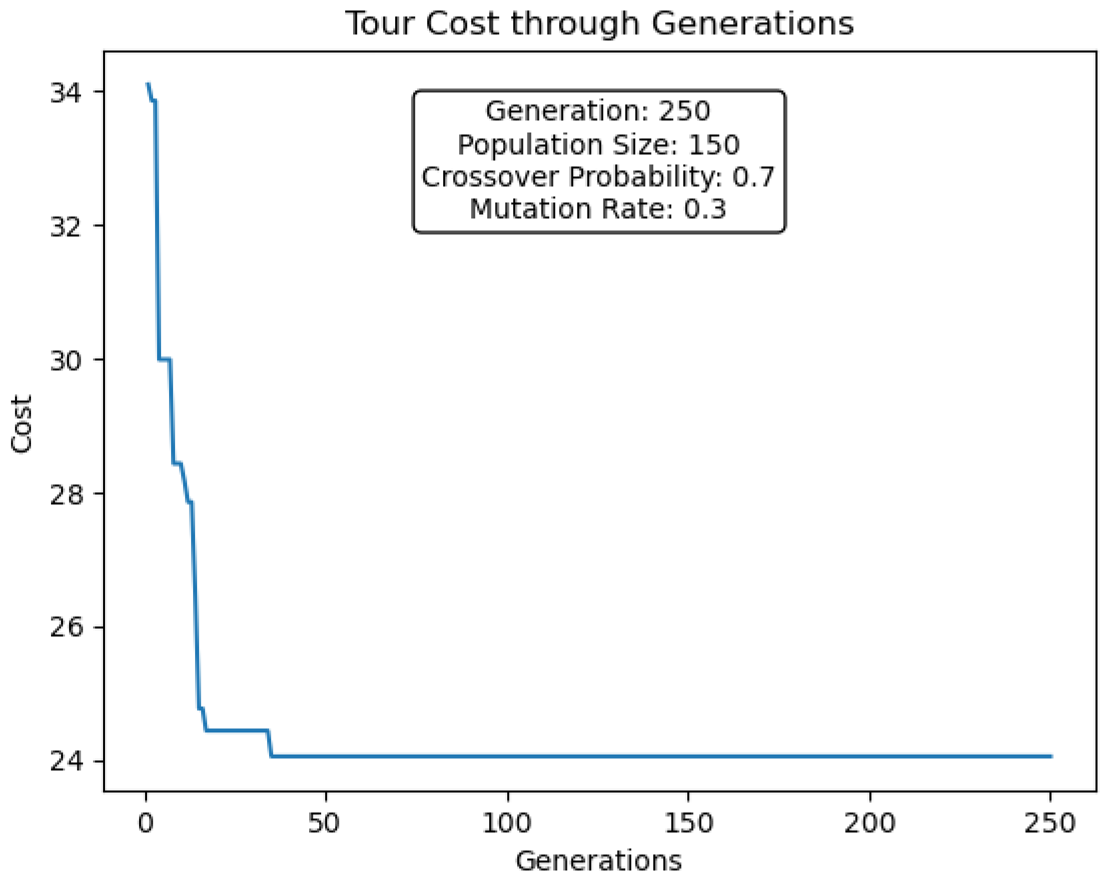
The problem is to minimize the distance travelled by a salesperson as they visit all the cities, visiting each city exactly once.

In [1]:
```
import random
import math
import re

#This class to define the stucture of node with city id and its 2D coordinates (x,y)
class Node:
    def __init__(self, id, x, y):
        self.id = int(id)
        self.x = float(x)
        self.y = float(y)

#Input city data and its co-ordinates (x, y) - This is various benchmark data
file_name = "burma14.tsp"
#file_name = "eil51.tsp"
#file_name = "berlin52.tsp"
#file_name = "eil76.tsp"
#file_name = "lin105.tsp"
#file_name = "lin318.tsp"


#Creating data set based on input city files
dataset = []
with open(file_name, "r") as f:
    for line in f:
        new_line = re.split(r'\s+', line.strip())
        if new_line[0].isdigit():
            id, x, y = new_line[0], float(new_line[1]), float(new_line[2])
            dataset.append(Node(id=id, x=x, y=y))

N = len(dataset) # Number of cities
print("Number of Cities:" + str(N))

#Creating distance matrix using Euclidean distance formula
def distance_matrix(node_list):
    matrix = [[0 for _ in range(N)] for _ in range(N)]

    for i in range(0, len(matrix)-1):
        for j in range(0, len(matrix[0])-1):
            matrix[node_list[i].id][node_list[j].id] = math.sqrt(
                pow((node_list[i].x - node_list[j].x), 2) + pow((node_list[i].y - node_l
            )
    return matrix

matrix = distance_matrix(dataset)

# The class is for Chromosome for maintaing Node list.
class Chromosome:
    def __init__(self, node_list):
        self.chromosome = node_list

        chr_representation = []
        for i in range(0, len(node_list)):
            chr_representation.append(self.chromosome[i].id)
        self.chr_representation = chr_representation

        distance = 0
        for j in range(1, len(self.chr_representation) - 1):  # get distances from the m
            distance += matrix[self.chr_representation[j]-1][self.chr_representation[j +
        self.cost = distance

        self.fitness_value = 1 / self.cost


# create a random chromosome and start and end points should be same
def create_random_list(n_list):

    start = n_list[0]
    temp = n_list[1:]
    temp = random.sample(temp, len(temp))
    temp.insert(0, start)
    temp.append(start)
    return temp


# initialize the population
def initialization(data, pop_size):
    initial_population = []
    for i in range(0, pop_size):
        temp = create_random_list(data)
        new_ch = Chromosome(temp)
        initial_population.append(new_ch)
    return initial_population


# Select parent chromosomes to create child chromosomes using tournament selection
def selection(population):
    ticket_1, ticket_2, ticket_3, ticket_4 = random.sample(range(0, 99), 4)

    # create candidate chromosomes based on ticket numbers
    candidate_1 = population[ticket_1]
    candidate_2 = population[ticket_2]
    candidate_3 = population[ticket_3]
    candidate_4 = population[ticket_4]

    # select the winner according to their costs
    if candidate_1.fitness_value > candidate_2.fitness_value:
        winner = candidate_1
    else:
        winner = candidate_2

    if candidate_3.fitness_value > winner.fitness_value:
        winner = candidate_3

    if candidate_4.fitness_value > winner.fitness_value:
        winner = candidate_4

    return winner


# Two points crossover
def crossover(p_1, p_2):
    point_1, point_2 = random.sample(range(1, len(p_1.chromosome)-1), 2)
    begin = min(point_1, point_2)
    end = max(point_1, point_2)

    child_1_1 = p_1.chromosome[:begin]
    child_1_2 = p_1.chromosome[end:]
    child_1 = child_1_1 + child_1_2
    child_2 = p_2.chromosome[begin:end+1]

    child_1_remain = [item for item in p_2.chromosome[1:-1] if item not in child_1]
    child_2_remain = [item for item in p_1.chromosome[1:-1] if item not in child_2]
    child_1 = child_1_1 + child_1_remain + child_1_2
    child_2 += child_2_remain

    child_2.insert(0, p_2.chromosome[0])
    child_2.append(p_2.chromosome[0])

    return child_1, child_2


# Mutation operation
def mutation(chromosome):  # swap two nodes of the chromosome
    mutation_index_1, mutation_index_2 = random.sample(range(1, 10), 2)
    chromosome[mutation_index_1], chromosome[mutation_index_2] = chromosome[mutation_ind
    return chromosome


# Find the best chromosome of the generation based on the cost
def find_best(generation):
    best = generation[0]
    for n in range(1, len(generation)):
        if generation[n].cost < best.cost:
            best = generation[n]
    return best

# Use elitism, crossover, mutation operators to create a new generation based on a previ

def create_new_generation(previous_generation, crossover_probability, mutation_rate):
    new_generation = [find_best(previous_generation)]  # This is for elitism. Keep the b

    # Use two chromosomes and create two chromosomes. So, iteration size will be half of
    for a in range(0, int(len(previous_generation)/2)):
        parent_1 = selection(previous_generation)
        parent_2 = selection(previous_generation)

        if random.random() < crossover_probability:
            child_1, child_2 = crossover(parent_1, parent_2)  # This will create node li
            child_1 = Chromosome(child_1)
            child_2 = Chromosome(child_2)
        else:
            child_1 = parent_1
            child_2 = parent_2

        if random.random() < mutation_rate:
            mutated = mutation(child_1.chromosome)
            child_1 = Chromosome(mutated)

        new_generation.append(child_1)
        new_generation.append(child_2)

    return new_generation


def genetic_algorithm(num_of_generations, pop_size, cross_prob, mutation_rate, data_list

    new_gen = initialization(data_list, pop_size)

    costs_for_plot = []
    for iteration in range(0, num_of_generations):
        new_gen = create_new_generation(new_gen, cross_prob, mutation_rate)
        costs_for_plot.append(find_best(new_gen).cost)

    return new_gen, costs_for_plot
```

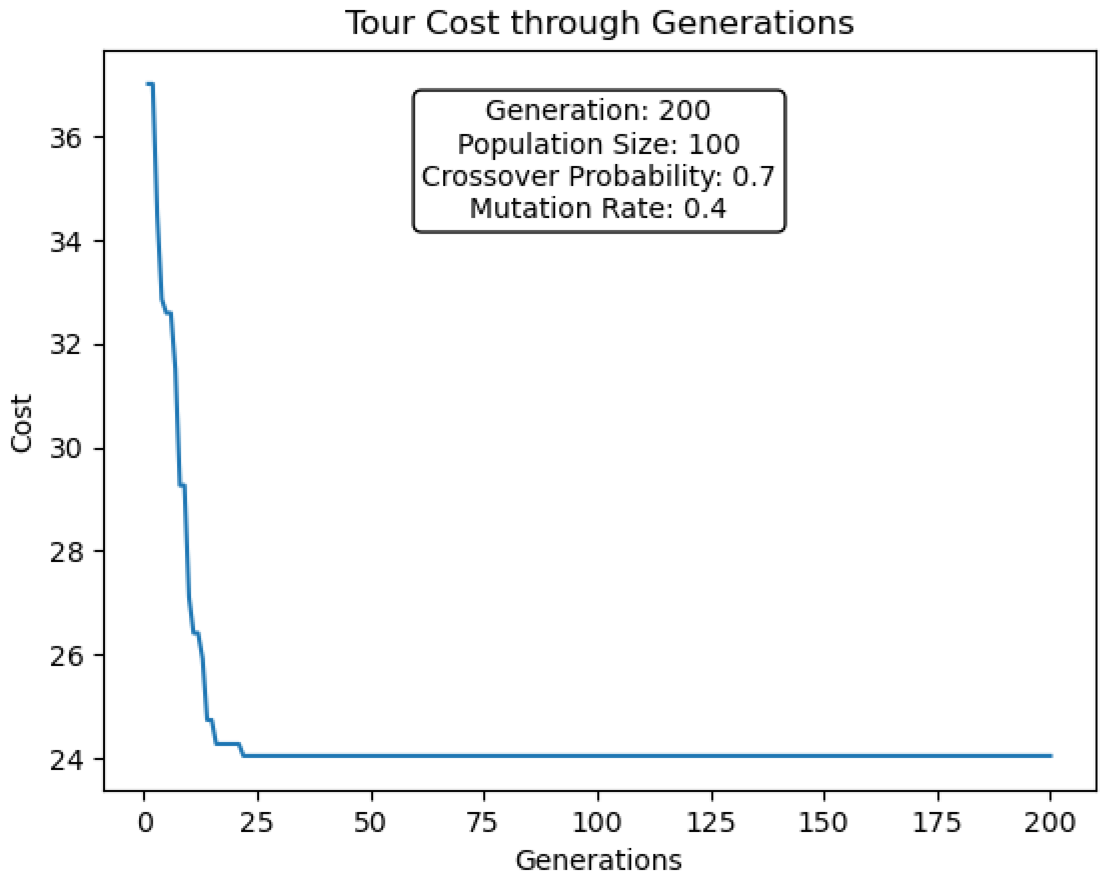
```
Number of Cities:14
```

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

def draw_path(solution):
    x_list = []
    y_list = []

    for m in range(0, len(solution.chromosome)):
        x_list.append(solution.chromosome[m].x)
        y_list.append(solution.chromosome[m].y)

    fig, ax = plt.subplots()
    plt.scatter(x_list, y_list)  # alpha=0.5

    ax.plot(x_list, y_list, '--', lw=2, color='black', ms=10)
    ax.set_xlim(0, 2000)
    ax.set_ylim(0, 1300)

    plt.show()

def draw_cost_generation(y_list, generation, pop_size, cross_prob, mutation_rate):
    x_list = np.arange(1, len(y_list) + 1)  # create a numpy list from 1 to the numbers

    plt.plot(x_list, y_list)

    plt.title("Tour Cost through Generations")
    plt.xlabel("Generations")
    plt.ylabel("Cost")

    parameter_label = f'Generation: {generation}\nPopulation Size: {pop_size}\nCrossover
    plt.annotate(parameter_label, xy=(0.5, 0.85), xycoords='axes fraction', fontsize=10,
                 bbox=dict(boxstyle="round,pad=0.3", edgecolor="black", facecolor="white

    plt.show()
```

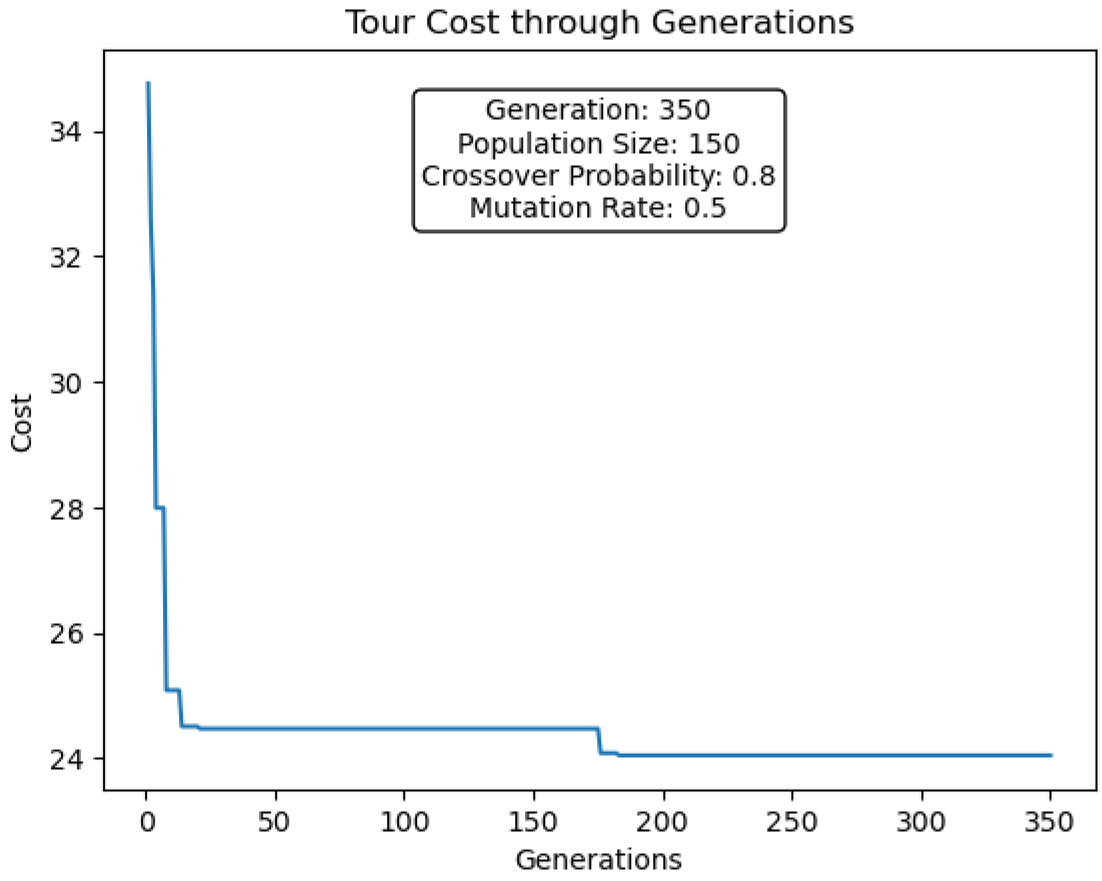
In [3]:
```
# Various Parameter combination to achieve best solution
parameter_combinations = [
    (250, 150, 0.7, 0.3),
    (200, 100, 0.7, 0.4),
    (350, 150, 0.8, 0.5),
    (300, 250, 0.8, 0.6),
    (400, 250, 0.8, 0.7),
    (700, 400, 0.9, 0.8),
]

for params in parameter_combinations:
    numbers_of_generations, population_size, crossover_probability, mutation_rate = para

    last_generation, y_axis = genetic_algorithm(
        num_of_generations=numbers_of_generations,
        pop_size=population_size,
        cross_prob=crossover_probability,
        mutation_rate=mutation_rate,
        data_list=dataset
    )

    best_solution = find_best(last_generation)

    best_cost_last_generation = last_generation[0].cost
    best_path_last_generation = last_generation[0].chr_representation
    print(f"The minimum tour length is: {best_cost_last_generation:.2f}")
    print(f"The best path is: {best_path_last_generation}")

    draw_cost_generation(y_axis, numbers_of_generations, population_size, crossover_prob
The minimum tour length is: 24.05
The best path is: [1, 11, 10, 12, 2, 3, 9, 14, 8, 7, 13, 4, 5, 6, 1]
```



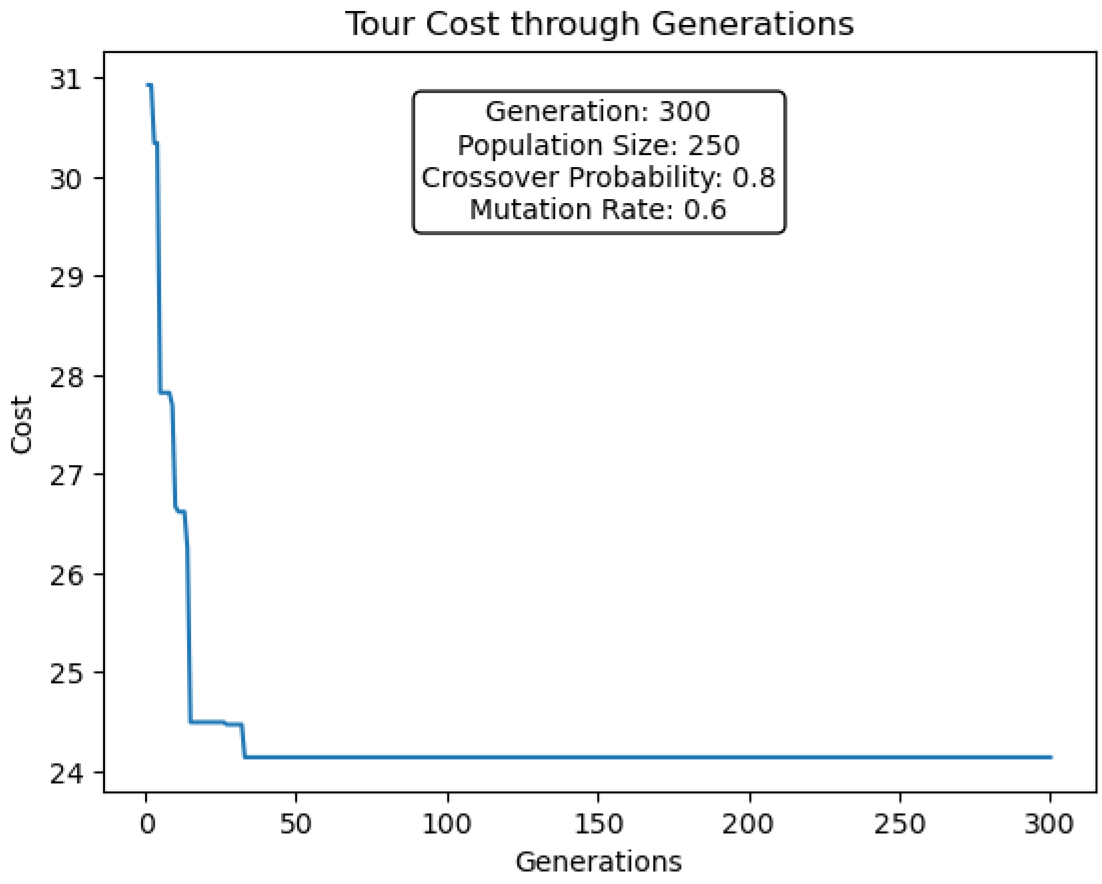
```
The minimum tour length is: 24.05
The best path is: [1, 11, 10, 12, 2, 3, 9, 14, 8, 7, 13, 4, 5, 6, 1]
```



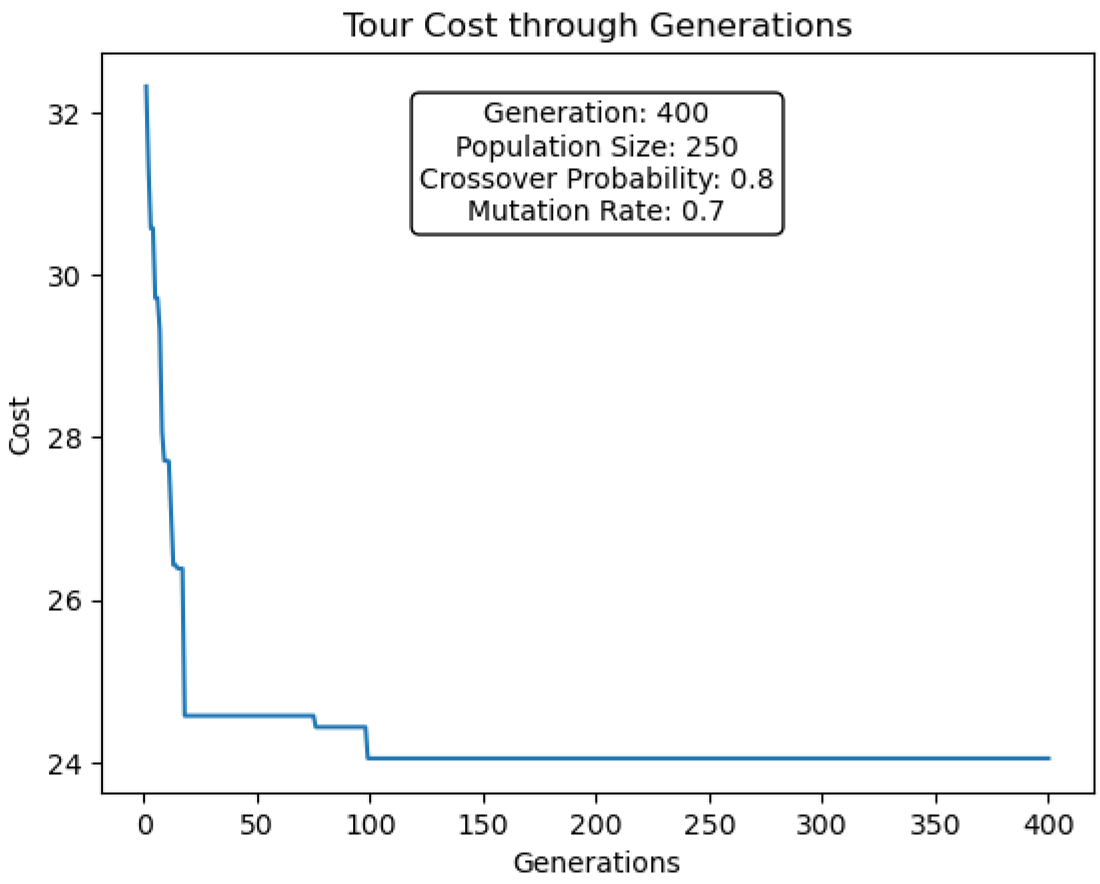
```
The minimum tour length is: 24.05
The best path is: [1, 6, 5, 4, 13, 7, 8, 14, 9, 3, 2, 12, 10, 11, 1]
```



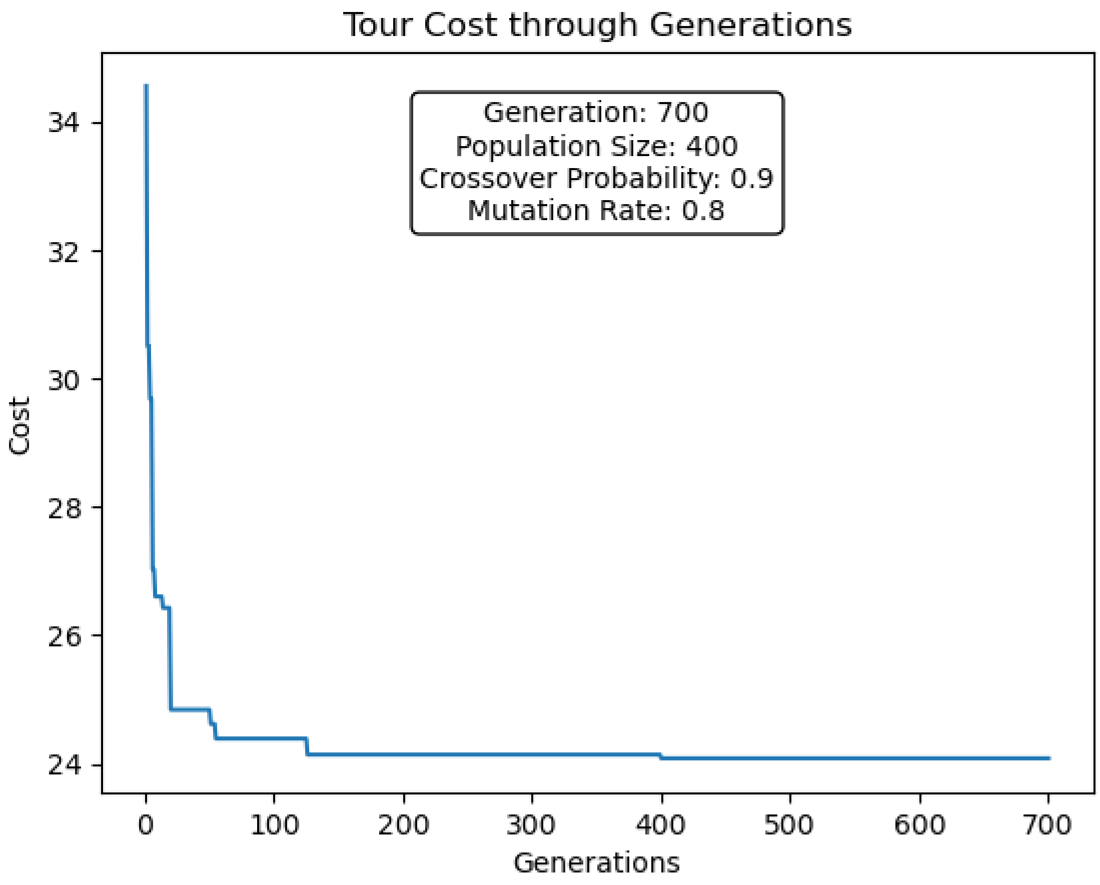
```
The minimum tour length is: 24.14
The best path is: [1, 11, 10, 12, 9, 2, 3, 4, 5, 13, 14, 8, 7, 6, 1]
```



```
The minimum tour length is: 24.05
The best path is: [1, 11, 10, 12, 2, 3, 9, 14, 8, 7, 13, 4, 5, 6, 1]
```



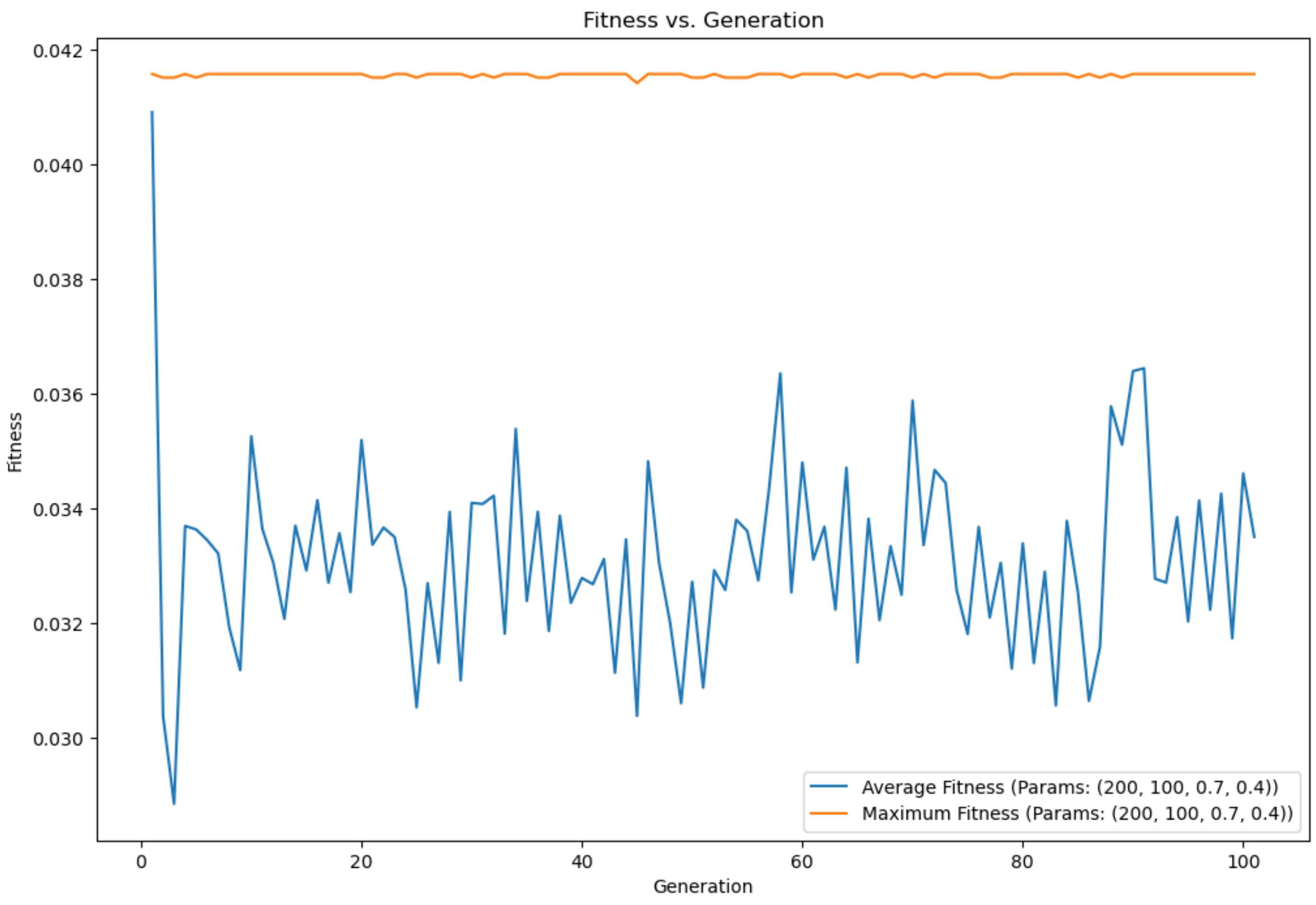
```
The minimum tour length is: 24.09
The best path is: [1, 11, 10, 12, 2, 3, 9, 14, 8, 13, 7, 6, 5, 4, 1]
```



In [4]:
```
import numpy as np
import matplotlib.pyplot as plt

# Define the parameter combinations
parameter_combinations = [(200, 100, 0.7, 0.4)] # for burma14.tsp
```

```
#parameter_combinations = [(350, 250, 0.8, 0.7)] # for eil51.tsp
#parameter_combinations = [(300, 250, 0.8, 0.6)] # for berlin52.tsp
#parameter_combinations = [(650, 400, 0.9, 0.6)] # for eil76.tsp
#parameter_combinations = [(700, 400, 0.9, 0.8)] # for lin105.tsp
#parameter_combinations = [(700, 400, 0.8, 0.8)] # for lin318.tsp

# Initialize arrays to store fitness values
num_generations = max([params[0] for params in parameter_combinations])
avg_fitness = np.zeros((num_generations, len(parameter_combinations)))
max_fitness = np.zeros((num_generations, len(parameter_combinations)))

num_runs = 30
for params_idx, params in enumerate(parameter_combinations):
    num_of_generations, pop_size, cross_prob, mutation_rate = params

    for _ in range(num_runs):
        np.random.seed()  # Generate a new random seed for each run
        last_generation, _ = genetic_algorithm(
            num_of_generations=num_of_generations,
            pop_size=pop_size,
            cross_prob=cross_prob,
            mutation_rate=mutation_rate,
            data_list=dataset
        )

        # Extract fitness values from the last generation
        fitness_values = [chromosome.fitness_value for chromosome in last_generation]

        # Update average and maximum fitness arrays
        avg_fitness[:len(fitness_values), params_idx] += np.array(fitness_values) / num_
        max_fitness[:len(fitness_values), params_idx] = np.maximum(max_fitness[:len(fitn

# Plotting with magnified figure size
plt.figure(figsize=(12, 8))

# Plotting
x_values = np.arange(1, num_generations+1)


for params_idx, params in enumerate(parameter_combinations):
    avg_fit = avg_fitness[:, params_idx]
    max_fit = max_fitness[:, params_idx]

    avg_mask = avg_fit != 0
    max_mask = max_fit != 0

    if np.any(avg_mask):
        plt.plot(x_values[avg_mask], avg_fit[avg_mask], label=f'Average Fitness (Params:

    if np.any(max_mask):
        plt.plot(x_values[max_mask], max_fit[max_mask], label=f'Maximum Fitness (Params:


# Customize plot appearance
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.title('Fitness vs. Generation')
plt.suptitle(file_name, color='red', weight='bold')
plt.legend()
plt.show()
```

burma14.tsp

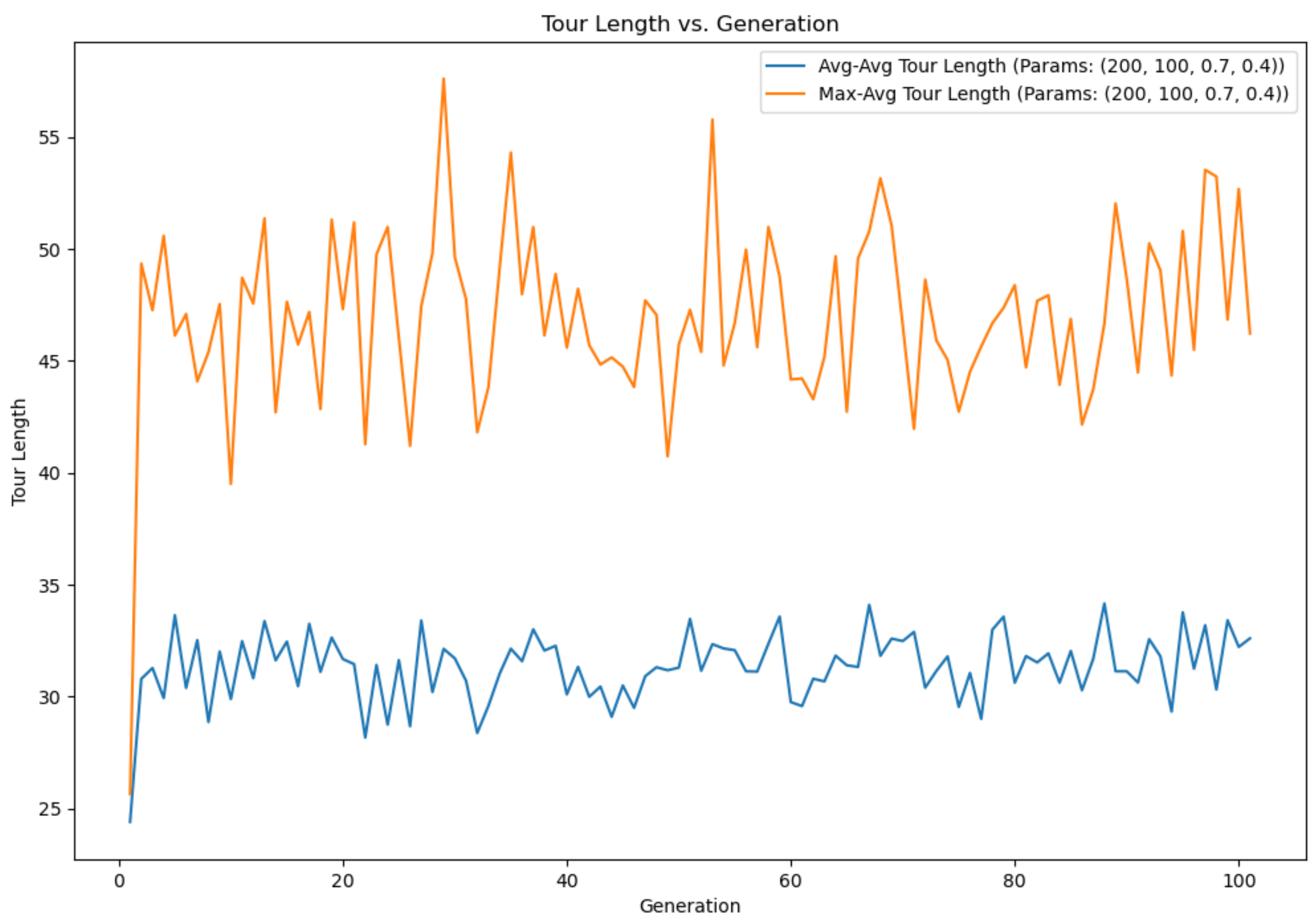


In [5]:

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

# Initialize arrays to store objective function values
num_generations = max([params[0] for params in parameter_combinations])
avg_avg_tour_length = np.zeros(num_generations)
max_avg_tour_length = np.zeros(num_generations)

num_runs = 30
for params_idx, params in enumerate(parameter_combinations):
    num_of_generations, pop_size, cross_prob, mutation_rate = params

    for _ in range(num_runs):
        np.random.seed()  # Generate a new random seed for each run
        last_generation, _ = genetic_algorithm(
            num_of_generations=num_of_generations,
            pop_size=pop_size,
            cross_prob=cross_prob,
            mutation_rate=mutation_rate,
            data_list=dataset
        )

        # Extract tour lengths from the last generation
        tour_lengths = [chromosome.cost for chromosome in last_generation]

        # Update average tour length arrays
        avg_avg_tour_length[:len(tour_lengths)] += np.array(tour_lengths) / (num_runs *
        max_avg_tour_length[:len(tour_lengths)] = np.maximum(max_avg_tour_length[:len(to

# Create masks to exclude zero values
avg_mask = avg_avg_tour_length != 0
max_mask = max_avg_tour_length != 0
```

```
# Plotting
x_values = np.arange(1, num_generations+1)

plt.figure(figsize=(12, 8))

if np.any(avg_mask):
    plt.plot(x_values[avg_mask], avg_avg_tour_length[avg_mask], label=f'Avg-Avg Tour Len

if np.any(max_mask):
    plt.plot(x_values[max_mask], max_avg_tour_length[max_mask], label=f'Max-Avg Tour Len

# Customize plot appearance
plt.xlabel('Generation')
plt.ylabel('Tour Length')
plt.title('Tour Length vs. Generation')
plt.suptitle(file_name, color='red', weight='bold')
plt.legend()
plt.show()
```

burma14.tsp



In [6]:
```
# Initialize a dictionary to store results
results = {}

optimal_cost = 24.05
quality_threshold = 1.1  # Set your desired quality threshold (e.g., 10% above optimum)

# Loop over each parameter combination
for params in parameter_combinations:
    num_generations, pop_size, cross_prob, mutation_rate = params

    # Perform evaluation for this parameter combination
    best_costs = []
    reliability_count = 0
    total_evaluations = 0

    for run in range(num_runs):
        final_generation, costs_for_plot = genetic_algorithm(
            num_of_generations=num_generations,
            pop_size=pop_size,
            cross_prob=cross_prob,
            mutation_rate=mutation_rate,
            data_list=dataset
        )

        best_cost = find_best(final_generation).cost
        best_costs.append(best_cost)

        if (best_cost / optimal_cost) <= quality_threshold:
            reliability_count += 1

        evaluations_needed = len(costs_for_plot)
        total_evaluations += evaluations_needed

    average_best_cost = sum(best_costs) / num_runs
    reliability_percentage = (reliability_count / num_runs) * 100
    average_evaluations = total_evaluations / num_runs

    # Store results in the dictionary
    results[params] = {
        "Average Best Cost": average_best_cost,
        "Reliability Percentage": reliability_percentage,
        "Average Evaluations": average_evaluations
    }

print(f"Dataset name: {file_name}")

# Print results for each parameter combination
for params, metrics in results.items():
    print(f"Parameters: {params}")
    print(f"Average Best Cost: {metrics['Average Best Cost']:.2f}")
    print(f"Percentage Distance from Optimum: {(metrics['Average Best Cost'] / optimal_c
    print(f"Reliability Percentage: {metrics['Reliability Percentage']}%")
    print(f"Average Evaluations: {metrics['Average Evaluations']}")
    print("="*30)
```

```
Dataset name: burma14.tsp
Parameters: (200, 100, 0.7, 0.4)
Average Best Cost: 24.40
Percentage Distance from Optimum: 101.44%
Reliability Percentage: 100.0%
Average Evaluations: 200.0
==============================
```

```
In [ ]:
```