W04ITE-SM4017G

Optimization Methods: Theory and Applications

PROJECT –2

Evolutionary Design of IoT Edge Computing Architecture

Full Name: Ece Kobanç

Student ID: 283531

1- Introduction

The proliferation of Internet of Things (IoT) devices has revolutionized numerous industries, enabling unprecedented levels of automation, data collection, and real-time analytics. IoT devices, ranging from simple sensors to complex industrial machines, are increasingly integrated into our daily lives and business operations. As these devices become more ubiquitous, the need for optimizing their performance while minimizing energy consumption has emerged as a critical challenge.

1.1- Importance of Energy Consumption in IoT Devices

Energy consumption is a pivotal concern in the design and deployment of IoT devices. While many IoT devices operate on battery power, a significant number also depend on other forms of energy, such as electricity and water, especially in industrial and large-scale applications. Efficient energy consumption across these sources ensures the longevity and reliability of IoT systems, reducing maintenance costs and downtime.

In industrial automation, for instance, IoT devices often manage processes that consume substantial amounts of electricity. These devices must operate efficiently to minimize electricity usage, which can lead to significant cost savings. Similarly, in applications such as smart agriculture and water management, IoT devices control systems that use water resources. Efficient water consumption in these contexts is crucial for sustainability and cost-effectiveness.

1.2- Significance of High Performance in IoT Devices

High performance in IoT devices is essential to meet the demanding requirements of modern applications. Performance metrics for IoT devices often include processing speed, latency, memory usage, and the ability to handle multiple tasks simultaneously. High performance ensures that devices can process data quickly, respond to events in real-time, and perform complex computations without delay. This is particularly crucial in applications such as autonomous vehicles, healthcare monitoring, and industrial automation, where timely and accurate data processing can have significant implications for safety, efficiency, and operational effectiveness.

2- Description of Problem

2.1- Balancing Energy Consumption and Performance

The challenge of balancing energy consumption and high performance in IoT devices necessitates advanced optimization techniques. Traditional methods of optimizing single objectives are inadequate in this context, as they may lead to solutions that favor

energy efficiency at the cost of performance, or vice versa. Multi-Objective Genetic Algorithms (MOGAs) offer a robust solution to this problem by simultaneously optimizing multiple conflicting objectives. By employing MOGAs, we can identify a set of optimal solutions, known as the Pareto front, which represent the best possible trade-offs between energy consumption and performance.

2.2- Objective of the Report

This report aims to explore the application of MOGAs in optimizing the energy consumption and performance of IoT devices. The report's findings of this report are expected to provide valuable insights for the design and deployment of more efficient and high-performing IoT systems, ultimately contributing to the advancement of sustainable and effective IoT solutions.

3- Algorithm Explanation

The algorithm implemented in this report is a Multi-Objective Genetic Algorithm (MOGA) designed to optimize the performance and energy consumption of IoT devices. This algorithm leverages the DEAP (Distributed Evolutionary Algorithms in Python) framework to find a set of optimal solutions that balance multiple conflicting objectives.

3.1- Data Generation Process

To effectively evaluate and optimize the performance and energy consumption of edge devices, a synthetic dataset was generated representing the operational metrics of multiple edge devices over time.

3.1.1- Function Definition for Data Generation

A function named generate_edge_data() is defined to create synthetic data. This function takes the following parameters:

num_edge_devices: The number of edge devices to simulate.

start_time: The starting timestamp for the data records.

num_records: The number of records (data points) to generate for each device.

frequency: The frequency (in seconds) at which data records are generated.

3.1.2- Data Generation Logic

The function iterates through each edge device and generates the specified number of records. For each record, it simulates the following metrics:

CPU Usage (%): Randomly generated between 50% and 90%.

Memory Usage (MB): Randomly generated between 1000 MB and 2000 MB.

Energy Consumption (Wh): Randomly generated between 1.0 Wh and 5.0 Wh.

Latency (ms): Randomly generated between 10 ms and 50 ms.

Task Count: Randomly generated integer between 10 and 30.

Queue Length: Randomly generated integer between 1 and 10.

3.1.3- Data Storage

The generated data is stored in a pandas Data Frame and then saved as a CSV file named edge_device_data.csv.

3.2- Multi-Objective Genetic Algorithm (MOGA) Design

3.2.1- Defining the Objective Functions

The objective functions evaluate each solution (individual) based on its performance and energy consumption attributes:

CPU Usage (%): Represents the percentage of CPU utilization.

Memory Usage (MB): Represents the memory consumed.

Energy Consumption (Wh): Represents the energy used.

Latency (ms): Represents the response time.

Task Count: Represents the number of tasks processed (maximized).

Queue Length: Represents the length of the task queue.

The objectives are normalized for comparison and inverted for minimization (except for Task Count, which is maximized).

```python
# Define the objective functions for performance and energy consumption
def evaluate(individual):
    cpu_usage, memory_usage, energy_consumption, latency, task_count, queue_length = individual

    # Normalize and invert the objectives for minimization
    normalized_cpu_usage = cpu_usage / max(data['CPU Usage (%)'])
    normalized_memory_usage = memory_usage / max(data['Memory Usage (MB)'])
    normalized_energy_consumption = energy_consumption / max(data['Energy Consumption (Wh)'])
    normalized_latency = latency / max(data['Latency (ms)'])
    normalized_task_count = task_count / max(data['Task Count'])  # To maximize

    # Inverting task_count for minimization (negative for maximization)
    return (normalized_cpu_usage, normalized_memory_usage, normalized_energy_consumption, normalized_latency, -normalized_task_count)
```

3.2.2- Setting Up the DEAP Framework

The DEAP framework is used to define the genetic algorithm's structure:

Fitness Function: Specifies that we are minimizing CPU Usage, Memory Usage, Energy Consumption, and Latency, while maximizing Task Count.

Individual and Population Initialization: Defines how individuals (solutions) and the population are initialized.

```python
# Set up the DEAP framework
creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0, -1.0, -1.0, 1.0))  # Minimizing all except task count
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
toolbox.register("attr_float", random.uniform, 0, 100)  # Assuming the range for normalization
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_float, n=6)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

3.2.3- Genetic Operators

The algorithm uses genetic operators for crossover, mutation, and selection:

Crossover: Uses the cxBlend operator, which blends the attributes of two parents to produce offspring.

Mutation: Uses a polynomial bounded mutation to introduce variability, ensuring the values stay within specified bounds.

Selection: Uses tournament selection to choose the best individuals for reproduction.

```
toolbox.register("mate", tools.cxBlend, alpha=0.5)

# Adjusted mutation function to ensure it works with floats correctly
def mutPolynomialBoundedReal(individual, eta, lower_bound, upper_bound, indpb):
    size = len(individual)
    for i in range(size):
        if random.random() <= indpb:
            current_gene_value = individual[i]  # Current gene value
            gene_lower_bound, gene_upper_bound = lower_bound, upper_bound  # Lower and upper bounds
            delta1 = (current_gene_value - gene_lower_bound) / (gene_upper_bound - gene_lower_bound)  # Calculate change rate
            delta2 = (gene_upper_bound - current_gene_value) / (gene_upper_bound - gene_lower_bound)  # Calculate change rate
            rand = random.random()
            mutation_power = 1.0 / (eta + 1.0)  # Mutation power
            if rand < 0.5:
                gene_change_amount = 1.0 - delta1  # Change amount
                val = 2.0 * rand + (1.0 - 2.0 * rand) * (gene_change_amount ** (eta + 1.0))
                delta_q = val ** mutation_power - 1.0
            else:
                gene_change_amount = 1.0 - delta2  # Change amount
                val = 2.0 * (1.0 - rand) + 2.0 * (rand - 0.5) * (gene_change_amount ** (eta + 1.0))
                delta_q = 1.0 - val ** mutation_power
            current_gene_value = current_gene_value + delta_q * (gene_upper_bound - gene_lower_bound)  # Calculate new gene value
            # Ensure current_gene_value is a real number and within bounds
            current_gene_value = float(min(max(current_gene_value.real, gene_lower_bound), gene_upper_bound))
            individual[i] = current_gene_value  # Assign new gene value to individual
    return individual,

toolbox.register("mutate", mutPolynomialBoundedReal, lower_bound=0.0, upper_bound=100.0, eta=20.0, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("evaluate", evaluate)
```

3.2.4- Population Initialization

The initial population is created with 200 individuals, each representing a potential solution.

```
# Initialize the population
population = toolbox.population(n=200)
```

3.2.5- Algorithm Parameters

The genetic algorithm parameters define the evolutionary process:

CXPB (Crossover Probability): The proportion of offspring that will be created through crossover.

MUTPB (Mutation Probability): The proportion of offspring that will be created through mutation.

NGEN (Number of Generations): The number of generations the algorithm will run.

```
# Define the algorithm parameters
CXPB, MUTPB, NGEN = 0.8, 0.2, 100  # Adjusted crossover and mutation probabilities and number of generations
```

3.2.6- Evolutionary Process

The eaMuPlusLambda function runs the evolutionary process, where mu is the number of individuals to select for the next generation, and lambda is the number of offspring generated.

```
# Use the built-in DEAP algorithm for the evolutionary process
result_pop, log = algorithms.eaMuPlusLambda(
    population, toolbox, mu=200, lambda_=400, cxpb=CXPB, mutpb=MUTPB, ngen=NGEN, stats=None, halloffame=None, verbose=True
)
```

3.2.7- Extracting the Pareto Front

The Pareto front is extracted to identify the set of optimal solutions that represent the best trade-offs between the objectives.

```
# Extract the Pareto front
pareto_front = tools.sortNondominated(result_pop, len(result_pop), first_front_only=True)[0]
```

3.2.8- Post-Processing

The best solutions are sorted and printed, providing the top configurations for IoT devices based on the optimized objectives.

```
# Post-processing: Extracting the solutions
best_solutions = sorted(pareto_front, key=lambda ind: (ind.fitness.values[0], ind.fitness.values[1], ind.fitness.values[2], ind.fitness.values[3], -ind.fi

# Collect the best solutions in a list
solutions = [individual for individual in best_solutions[:10]]

# Print the solutions
for solution in solutions:
    print(f"Solution: {solution}, Fitness: {solution.fitness.values}")

# Convert the fitness values to a list for further processing
fitness_values = [ind.fitness.values for ind in best_solutions]
```

4- Explanation of Results

Generations (gen):

The "gen" column represents the generation number in the evolutionary algorithm. Each generation is a complete cycle of the evolutionary process, including selection, crossover, mutation, and evaluation of individuals. In this case, there are 100 generations, indicating that the algorithm has gone through 100 cycles of evolution.

Number of Evaluations (nevals):

The "nevals" column represents the number of evaluations performed in each generation. In this context, 400 evaluations were performed in each generation. This means that 400 individual solutions were evaluated for their fitness in each generation.

Solutions:

Each solution represents an individual in the population, defined by a set of parameters (genes). In the context of this algorithm, each individual is represented by six parameters (cpu_usage, mem_usage, energy_consumption, latency, task_count, queue_length). The fitness values for each solution are calculated based on these parameters.

The solutions listed are the top solutions from the Pareto front. Each solution is represented by its parameter values and the corresponding fitness values.

Example Solution Breakdown

Solution: [-190.63975639556037, 64.9206365188524, 12.38628238726206, 54.36924956399016, 85.291371812581, 3.575837795800771]

Fitness: (-2.1279726433342327, 0.03259341228693041, 2.477323759291053, 1.0926506379536944, -2.941081786640724)

Parameters:

cpu_usage: -190.63975639556037

mem_usage: 64.9206365188524

energy_consumption: 12.38628238726206

latency: 54.36924956399016

task_count: 85.291371812581

queue_length: 3.575837795800771

Fitness Values:

cpu_usage: -2.1279726433342327 (normalized and inverted for minimization)

mem_usage: 0.03259341228693041 (normalized)

energy_consumption: 2.477323759291053 (normalized)

latency: 1.0926506379536944 (normalized)

task_count: -2.941081786640724 (normalized and inverted for maximization)

5- Comparison of Optimization Metrics

Population Size (n):

Values: 50, 100, 200

Larger population sizes generally provide a more diverse gene pool, which can improve the algorithm's ability to explore the solution space but at the cost of increased computational time.

Crossover Probability (CXPB):

Values: 0.6, 0.8

Higher crossover probabilities increase the likelihood of combining different parents' genetic materials, which can enhance exploration but might disrupt good solutions if too high.

Mutation Probability (MUTPB):

Values: 0.1, 0.2

Higher mutation probabilities introduce more randomness into the population, promoting exploration and preventing premature convergence but potentially leading to instability if too high.

Number of Generations (NGEN):

Values: 50, 100

More generations allow the algorithm to refine solutions further, typically improving performance metrics but also increasing computational time.

5.1- Metrics Evaluated

Hypervolume

Represents the volume of the objective space dominated by the Pareto front. Higher values indicate a better diversity and potentially more optimal solutions.

Spacing

Indicates the distribution of solutions along the Pareto front. Lower spacing values suggest a more homogeneous distribution, which is desirable for consistent performance.

Pareto Front Spread

Indicates the extent of the Pareto front. A larger spread means that solutions are more spread out, providing a wider range of trade-offs between objectives.

Experiment	Population Size (n)	Crossover Probability (CXPB)	Mutation Probability (MUTPB)	Number of Generations (NGEN)	Hypervolume Mean	Hypervolume Std	Spacing Mean	Spacing Std	Pareto Front Spread Mean	Pareto Front Spread Std
1	50	0.6	0.1	50	2229844.501096473	4189640.0774827385	75.17364709448809	108.0445493909 6263	3443.429152338933	4384.638878368607
2	50	0.6	0.2	100	272389.8072923673	350409.0738894745	33.176261620151 9	29.2334217986 0586	1654.459456100 9847	1515.354552651 7826
3	50	0.8	0.1	50	286839962.88186926	7448106 75.6321168	137.3047332013869	164.4651098116269	7868.249369999908	8612.299346698572
4	50	0.8	0.2	100	7345947709 7745.78	2155218505347 09.25	5324.671577707 18	10239.32056454 4396	268123.494380064	473663.7449472366
5	100	0.6	0.1	50	6870.149022175441	15818.882484315247	3.179151238915 0873	5.712120559965 817	167.2273714873765	281.3657144157397
6	100	0.6	0.2	100	2514721216 8372.44	7408124287 0605.94	3883977 1259.52 391	1156659 18397.4 9393	1860999 850755. 975	5537590 966464. 087
7	100	0.8	0.1	50	5962107248 61.2233	1440594403088.2036	7652.87 3400571 398	8506.19 7740505 988	504606. 6874630 467	634682. 7771726 593
8	100	0.8	0.2	100	2941704800 226044.0	385459847328229.3	182643. 6565669 751	199545. 0030830 0037	1250661 6.200785322	1515689 9.414369231
9	200	0.6	0.1	50	23705.944 718448467	45945.11 17422896 5	3.47094 5121080 103	1.92002 4332553 6444	218.764 3300973 519	128.387 0847087 2356
10	200	0.6	0.2	100	11506.677 36350425	11818.16822044420 8	12.8926 8801960 3279	20.5478 0506283 8936	585.304 9079327 831	712.859 7294102 796
11	200	0.8	0.1	50	7243337517.495624	10524201693.949154	71136.3 1559407 739	66819.8 0804207 422	4587843 .702957 761	4666805 .824644 102
12	200	0.8	0.2	100	7.321091448742059e+24	2.19608232333329 2e+25	6158742 2.75973 717	1317304 02.5372 5803	4265083 989.063 0045	9366931 881.031 858

5.2.- General Analysis of the Table for Energy Consumption

When evaluating the experiments, all three metrics were taken into account: Hypervolume, Spacing, and Pareto Front Spread. Additionally, the standard deviations of each were taken into account to understand the consistency of the performance of each experiment.

Hypervolume:

High hypervolume values indicate superior overall performance and energy efficiency. Experiments with notable hypervolume values are:

Experiment 3: High hypervolume with moderate variability.

Experiment 4: Extremely high hypervolume but very high variability.

Experiment 7: Very high hypervolume but with high variability.

Experiment 8: Extremely high hypervolume with the highest variability.

Experiment 11: High hypervolume with considerable variability.

Spacing:

Lower spacing values are preferable as they indicate more evenly distributed solutions along the Pareto front. Experiments with notable spacing values are:

Experiment 2: Relatively low spacing and standard deviation.

Experiment 5: Low spacing and standard deviation, suggesting consistent distribution of solutions.

Experiment 9: Low spacing with moderate variability, indicating fairly consistent solution distribution.

Experiment 10: Low spacing with moderate standard deviation, suggesting consistent performance.

Pareto Front Spread:

Lower Pareto Front Spread values indicate a more compact Pareto front, which is generally desirable. Experiments with notable Pareto Front Spread values are:

Experiment 5: Low Pareto Front Spread with moderate standard deviation.

Experiment 9: Low Pareto Front Spread and standard deviation, indicating consistent and compact Pareto front.

Experiment 10: Low Pareto Front Spread with moderate standard deviation, suggesting a compact and consistent Pareto front.

5.2.1- Best Experiments Based on Energy Consumption and Performance

Experiment 5 and Experiment 9 emerge as the most balanced choices in terms of energy consumption and performance due to their low range and Pareto Front Spread, indicating better uniformity and compactness of the solutions.

Experiment 5: The parameters used in Experiment 5 offer good uniformity and compactness with low range and Pareto Front Spread, despite lower overall performance. Therefore, experiment 5 is recommended for consistent performance with a compact Pareto front.

Experiment 9: The parameters used in Experiment 9 provide balanced performance with low range and Pareto Front Propagation, making it a strong candidate for consistent, compact, and energy-efficient solutions. Experiment 9 is also a strong candidate.

These experiments are recommended to achieve a balance between energy efficiency and high performance while maintaining a consistent and compact solution set.

6- REFERENCES

- Multi - Objective Genetic Algorithm (MOGA). (n.d.). (C) Copyright 2021. https://2021.help.altair.com/2021/hwdesktop/hst/topics/design_exploration/method_multi_objective_genetirc_algorithm_r.htm
- Long, Q., Wu, C., Wang, X., Jiang, L., & Li, J. (2015). A multiobjective genetic algorithm based on a discrete selection procedure. Mathematical Problems in Engineering, 2015, 1–17. https://doi.org/10.1155/2015/349781
- J. -H. Huh and Y. -S. Seo, "Understanding Edge Computing: Engineering Evolution With Artificial Intelligence," in IEEE Access, vol. 7, pp. 164229-164245, 2019, doi: 10.1109/ACCESS.2019.2945338. keywords: {Cloud computing;Edge computing;Servers;Data centers;Artificial intelligence;Internet of Things;Performance evaluation;Edge computing;cloud computing;Internet of Things;artificial intelligence;engineering},