ITS66604_0369834_GRUPASGNMT

by KAUSHIK RAJ JOSHI .

Submission date: 27-Jul-2025 11:45PM (UTC+0800)
Submission ID: 2721186663
File name: 240852_KAUSHIK_RAJ_JOSHI_._ITS66604_0369834_GRUPASGNMT_760883_1353774147.pdf (1.04M)
Word count: 5536
Character count: 31404

TAYLOR'S UNIVERSITY
Wisdom · Integrity · Excellence

TAYLOR'S PROGRAMMES
MAY 2025 SEMESTER

ITS66604 - Machine Learning and Parallel Computing

Group Assignment	30%
DUE DATE	July 13st 2025 via MyTiMes (11:59 pm)

STUDENT DECLARATION

1. *I confirm that I am aware of the University's Regulation Governing Cheating in a University Test and Assignment and of the guidance issued by the School of Computing and IT concerning plagiarism and proper academic practice and that the assessed work now submitted is in accordance with this regulation and guidance.*
2. *I understand that, unless already agreed with the School of Computing and IT, assessed work may not be submitted that has previously been submitted, either in whole or in part, at this or any other institution.*
3. *I recognize that should evidence emerge that my work fails to comply with either of the above declarations, then I may be liable to proceedings under Regulation.*

No	Student Name	Student ID	Date	Signature	Score
1.	Kaushik Raj Joshi	0369834	2025-7-27		
2.	Sewanty Upreti	0369894	2025-7-27		
3.	Shrena Shakya	0369914	2025-7-27		
4.	Rojit Khadgi	0370018	2025-7-27		
5.	Aman Kumar Sah	0370006	2025-7-27		

Table of Contents

1. Mathematical Formulation of the Problem

1.1 Smart-Grid Distribution Problem as a Constrained Optimization Problem

A smart grid is equipped with a set of generators which we assume to be available to us as G and consumer (loads) D. The aim is to make power dispatch decisions $x = \{x_i\}_{i \in G}$ as well as routing decisions (coded in the chromosome), such that it would optimize several objectives subject to physical and operation constraints. Formally:

2

$$\begin{aligned} &\underset{x}{\text{minimize / maximize}} && F(x) = [f1(x), f2(x), f3(x)] \\ &\text{subject to} && h(x)=0,\ g(x)\leq 0, \\ & && x_i^{min} \leq x_i \leq xi^{max},\ \forall i \in G, \end{aligned}$$

where,

- x_i: Power that is sent by generator i
- f_1, f_2, f_3: The objective functions (costs, losses, fulfilling demands).
- h: Equality conditions (energy balance at every bus).
- g: The inequality constraints are g inequality (capacity, line limits).
- x_i^{min}, x_i^{max}: Bed limits.

Code Snippet,

```
# Function to create a random chromosome (an individual solution)
def create_chromosome():
    return np.random.rand(NUM_SOURCES, NUM_NODES, HOURS)
```

Output,

```
[12] chrom = create_chromosome()
     print(chrom.shape)

     (3, 10, 24)
```

2.2 Objectives and Constraints

1. Cost reduction

$$f_1(x) = \sum_{i \in G} (a_i x_i 2 + b_i x_i + c_i)$$

A quadratic cost curve of each generator (coefficients t a_i, b_i, c_i).

2. Loss Control

$f_2(x) = \sum_{(i,j)\in E} R_{ij}(P_{ij}(x))^2$

where E is the set of lines, R_{ij} is line resistance and P_{ij} flows on the line (i, j).

3. Fulfillment of Demand

$f_3(x) = |\sum_{i\in G} x_i - \sum_{d\in D} L_d|$

where L_d is demand node *d* load. Here, the absolute mismatch is minimized.
The imbalance position of the total generation and the total demand are theoretically absolute.

Constraints:

- Energy Balance:

$\sum_{i\in G} x_i = \sum_{d\in D} L_d.$

- Capacity Limits:

$x_i^{min} \leq x_i \leq x_i^{max}, \forall_i.$

- Line Flow Limits:

$|P_{ij}(x)| \leq P_{ij}^{max}, \forall_{(i,j)}.$

Code Snippet:

```
[ ] def greedy_energy_allocation(demand_matrix, source_capacity, cost_per_unit):

        start = time.perf_counter() # start measuring executing time
        memory = psutil.virtual_memory()

        # Initialize an empty allocation matrix to store energy assignments
        allocation = np.zeros((NUM_SOURCES, NUM_NODES, HOURS))

        # Sort sources by increasing cost per unit, so cheaper sources are used first
        sorted_sources = np.argsort(cost_per_unit)

        # Loop over each hour in the 24-hour cycle
        for h in range(HOURS):
            # Loop over each node (energy consumer)
            for n in range(NUM_NODES):
                remaining_demand = demand_matrix[n][h]  # Retrieve the energy demand of node 'n' at hour 'h'

                # Loop through sources in order of increasing cost
                for s in sorted_sources:
                    # Calculate how much energy has already been used from this source at hour 'h'
                    used_capacity = allocation[s, :, h].sum()

                    # Determine the remaining capacity of this source at hour 'h'
                    available = source_capacity[s][h] - used_capacity

                    # Allocate the minimum between available energy and remaining demand
                    assigned = min(available, remaining_demand)

                    # Assign the energy to the node from this source at this hour
                    allocation[s][n][h] = assigned

                    # Reduce the remaining demand for the node
                    remaining_demand -= assigned

                    # If the demand is fully satisfied, move to the next node
                    if remaining_demand <= 0:
                        break
        duration = time.perf_counter() - start  # Total execution time
        # Return the completed energy allocation matrix
        return allocation, duration
```

Output:

```
# Generate greedy allocation
greedy_alloc, greedy_time = greedy_energy_allocation(demand_matrix, source_capacity, cost_per_unit)

# Evaluate greedy performance using the same fitness function
greedy_fitness = evaluate_fitness(greedy_alloc)

# Compare to best GA result
ga_log, _, _ = run_ga_with_log_and_tracking()
ga_best_fitness = ga_log['Fitness'].max()

print(f"Greedy Fitness: {greedy_fitness:.2f}")
print(f"Best GA Fitness: {ga_best_fitness:.2f}")
print(f"Greedy Algorithm Runtime: {greedy_time} seconds")
```

```
Greedy Fitness: -11.02
Best GA Fitness: -17.03
Greedy Algorithm Runtime: 0.0026708800000960764 seconds
```

CPU Usage Monitoring

3.3 GA approximation and Mathematical Formulation

After concluding all, the multi-objective constrained problem is given below:

$$\min_{x} \; [f_1(x), f_2(x), f_3(x)]$$

$$\text{s.t.} \quad h_1(x) = \sum_i x_i - \sum_d L_d = 0,$$

$$g_i^{(1)}(x) = x_i - x_i^{max} \le 0, \; g_i^{(2)}(x) = x_i^{min} - x_i \le 0,$$
$$g_{ij}^{(3)}(x) = |P_{ij}(x)| - P_{ij}^{max} \le 0.$$

A Genetic Algorithm gets close to the Pareto optimal front by:

- Encoding: There are chromosome encodings: x and routing choice (see *encode individual()* in MLPC Code.pdf)
- Initialization: Creation of population: N feasible (or nearly infeasible) solutions.
- Selection: Tournament or roulette-wheel selection in terms of composite fitness (e.g. weighted sum or Pareto dominance).
- Crossover & Mutation: Use crossover operators (uniform or two point) and crossover as well as bit/real mutation operators as defined in your *ga_operators.py*
- Parallel Evaluation: Map through fitness evaluations across multiple CPU Cores via *multiprocessing.Pool* (see below Parallelization Strategy file)
- Constraint Handling: Inflict a cost on infeasible solutions through the fitness function (the *evaluate_fitness()* applies hefty costs to an infraction on a proportionate scale).
- Termination: terminate when there is a specific number of generations or convergence threshold.

Code Snippet

```
[24] def run_ga_with_log_and_tracking(parallel=True):


        pop = initialize_population()  # Generate the initial population (random energy allocations)
        log = []  # Stores summary of the best individual for each generation
        all_fitnesses = []  # Stores complete fitness array per generation (used for plotting)
        start = time.time()  # Start execution timer

       # Repeat for each generation
        for gen in range(GENS):
            # Evaluate the fitness of the current population (either in parallel or serial)
            fitnesses = evaluate_population(pop, parallel=parallel)

            # Save all fitness scores for this generation (used for plotting range later)
            all_fitnesses.append(fitnesses)

            # Identify the best individual based on fitness
            best_idx = np.argmax(fitnesses)        # Index of the best fitness value
            best_fitness = fitnesses[best_idx]     # Best fitness score
            best_individual = pop[best_idx]        # Best chromosome (energy allocation)

            # Create a summary dictionary of best solution characteristics
            summary = {
                'Generation': gen + 1,                          # Current generation number (1-based)
                'a': np.mean(best_individual[0]),               # Average allocation from source 0
                'b': np.mean(best_individual[1]),               # Average allocation from source 1
                'c': np.mean(best_individual[2]),               # Average allocation from source 2
                'Fitness': best_fitness                         # Best fitness score
            }
            log.append(summary)  # Add the summary to the log list

            # Create the next generation
            selected = selection(pop, fitnesses)  # Select top individuals (selection strategy)

            new_pop = []  # Initialize new population list
            while len(new_pop) < POP_SIZE:
                p1, p2 = random.sample(selected, 2)  # Select two parents at random from selected pool
                child = crossover(p1, p2)            # Perform crossover to generate offspring
                child = mutate(child)                # Apply mutation to introduce variation
                new_pop.append(child)                # Add the new child to the new population

            pop = new_pop  # Replace the old population with the new one

        # Record total execution time
        duration = time.time() - start

        # Return summary DataFrame, all fitness history, and execution time
        return pd.DataFrame(log), all_fitnesses, duration
```

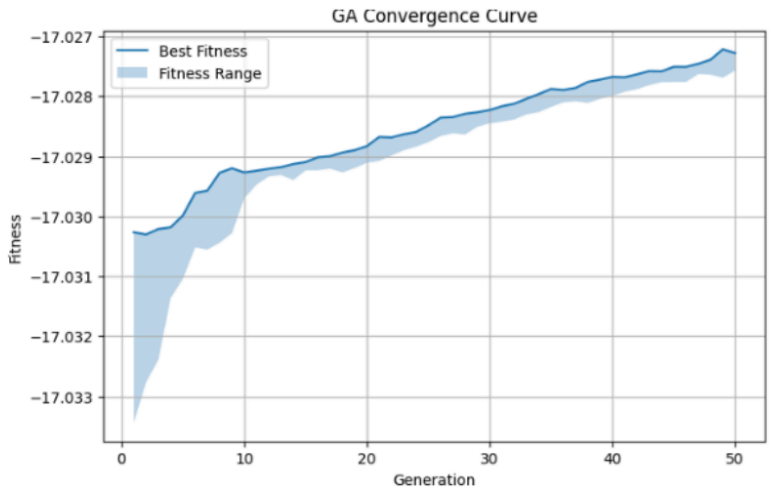
Output 1:

Generation summary:

Generation	a	b	c	Fitness
1	0.507161	0.507183	0.552035	-17.0307
2	0.519101	0.511868	0.545408	-17.0305
3	0.519554	0.515597	0.534412	-17.0308
4	0.501862	0.523524	0.555265	-17.0305
5	0.538984	0.512091	0.548039	-17.0301
6	0.52588	0.526287	0.564681	-17.0297
7	0.525519	0.525242	0.566165	-17.0297
8	0.524047	0.526439	0.568859	-17.0297
9	0.524478	0.523854	0.569997	-17.0297
10	0.524877	0.524369	0.569032	-17.0297
11	0.52278	0.529646	0.571564	-17.0296
12	0.526482	0.527447	0.573334	-17.0296
13	0.529579	0.52798	0.57387	-17.0295
14	0.526427	0.532741	0.573378	-17.0295
15	0.528169	0.527196	0.575405	-17.0295
16	0.527326	0.532841	0.573622	-17.0294
17	0.528766	0.530146	0.575138	-17.0294
18	0.531137	0.531626	0.576524	-17.0293
19	0.52977	0.53236	0.577288	-17.0293
20	0.528918	0.533868	0.581001	-17.0292
21	0.532597	0.533036	0.580655	-17.0292
22	0.531334	0.531902	0.58078	-17.0292
23	0.53169	0.533642	0.579161	-17.0292
24	0.532733	0.53681	0.582288	-17.0291
25	0.532724	0.538854	0.583983	-17.029
26	0.538193	0.533286	0.587441	-17.0289
27	0.536084	0.532573	0.589273	-17.0289
28	0.538166	0.536863	0.5903	-17.0288
29	0.536392	0.536189	0.589142	-17.0288
30	0.536776	0.538758	0.589323	-17.0288
31	0.540197	0.540311	0.589985	-17.0287
32	0.543077	0.541156	0.587957	-17.0286
33	0.543901	0.542776	0.588483	-17.0286
34	0.546917	0.543851	0.590585	-17.0284
35	0.545452	0.543576	0.589947	-17.0285
36	0.548739	0.543514	0.588469	-17.0285
37	0.544872	0.543259	0.593771	-17.0284
38	0.548468	0.544757	0.591583	-17.0284
39	0.550345	0.54468	0.590892	-17.0284
40	0.547082	0.546949	0.592518	-17.0283
41	0.552515	0.544215	0.592501	-17.0283
42	0.553965	0.549223	0.593556	-17.0281
43	0.552727	0.547734	0.596544	-17.0281
44	0.55296	0.548781	0.596419	-17.0281
45	0.555315	0.54739	0.597617	-17.0281
46	0.553776	0.552034	0.595461	-17.028
47	0.554543	0.551765	0.599716	-17.0279
48	0.558245	0.555052	0.595	-17.0278
49	0.558988	0.555453	0.597601	-17.0278
50	0.560717	0.557802	0.595756	-17.0277

Total GA run time (parallel): 2.41s

Output 2:




```
Total GA run time: 2.26 seconds
```

2. Encoding and Fitness Function Design

2.1 Chromosome Encoding Structure

For optimal distribution of energy in a smart grid, each solution must define how energy from various sources is sent to numerous consumer nodes throughout the course of a 24-hour cycle. We employ a real valued 3D chromosome encoding that captures the spatiotemporal feature of the problem while encoding such a solution to a Genetic Algorithm.

Structure of the Chromosome
Each of the chromosomes is a three-dimensional NumPy array with the following dimensions:

- NUM_SOURCES = 3 (example: hydro, wind and solar)
- NUM_NODES = 10 (example: houses or districts)
- HOURS = 24 (slots of hours)

Chromosomes [**s**][**n**][**h**] each one of them reflects a fraction of the energy distribution (to node n) of source **s** at hour **h** falls between [0,1].
Reasons for Encoding Design:

- 3D Structure: Keeps track of all source, time related as well as spatial data (nodes).
- Real Valued Genes: Allows for accurate and even distribution of energy (example: 45.6 units)
- Scalability: It's simple to expand to different sources, nodes or time periods (minutes)
- Operator Compatibility: Support is provided for genetic operations including Gaussian mutation and time-based crossover.

```
# Function to create a random chromosome (an individual solution)
def create_chromosome():
    return np.random.rand(NUM_SOURCES, NUM_NODES, HOURS)
```

The Genetic Algorithm can quickly develop independent, complicated, dynamic and workable energy distribution plans through its encoding policy which also gives it resistance to shifts in the real world.

2.2 Fitness Function Design

It needs to be noted that the fitness function also has a significant influence on determining each chromosome's quality and its ability to successfully achieve the smart grid's goals. In terms of the consequences of insufficient demand and excessive use of the source, it does this at the highest possible economic cost and technical approach feasibility.
Components of the objective:
Let:

- $E_{s,n,h}$: Throughout the hour, energy is delivered from source **s** to node **n**
- $D_{n,h}$: Load of energy at node **n** during **h**
- C_s: Cost of energy sources per unit

- $S_{s,h}$: Maximum capacity of the sources for each hour

1. Total energy cost

Cost = $\sum_{s=1}^{3} \sum_{n=1}^{10} \sum_{h=1}^{24} E_{s,n,h} \cdot C_s$

Here,

$E_{s,n,h}$: Energy delivered from source s to node n during hour h.

C_s: Cost per unit of energy from source s.

$\sum_{s=1}^{3} \sum_{n=1}^{10} \sum_{h=1}^{24}$: Summing for all 3 sources, 10 nodes, and 24 hours.

2. Demand performance penalty

Penalty demand = $\sum_{n=1}^{10} \sum_{h=1}^{24} [\max (0, D_{n,h} - \sum_{s=1}^{3} E_{s,n,h})]^2$

Here,

It penalizes situations where the delivered energy is less than the required demand at a node

$D_{n,h}$: Required demand at node n during hour h

$\sum_{s=1}^{3} E_{s,n,h}$: total energy delivered to node n from all sources during hour h.

3. Capacity of source penalty

Penalty capacity = $\sum_{s=1}^{3} \sum_{h=1}^{24} [\max (0, \sum_{n=1}^{10} E_{s,n,h} - S_{s,h})]^2$

Here,

$\sum_{n=1}^{10} E_{s,n,h}$: total energy delivered by source s to all nodes during hour h

$S_{s,h}$: Total energy delivered by source s to all nodes during hour h

4. Total Fitness score

We apply a negative logarithmic transformation to convert it into a maximizing goal:

Fitness = - log (1+ Cost + 10 * (Penalty demand + Penalty capacity))

```python
# Evaluate the fitness of a chromosome based on cost and penalties for constraints
def evaluate_fitness(chromosome):
    total_cost = 0  # Tracks total energy cost
    penalty = 0  # Tracks penalty for constraint violations

    # Iterate through each hour
    for h in range(HOURS):
        # Check if each node meets demand
        for n in range(NUM_NODES):
            total_power = sum(chromosome[s][n][h] for s in range(NUM_SOURCES))
            if total_power < demand_matrix[n][h]:
                penalty += (demand_matrix[n][h] - total_power) ** 2  # Penalty for under-supply

        # Check if source exceeds its capacity
        for s in range(NUM_SOURCES):
            source_total = sum(chromosome[s, :, h])
            if source_total > source_capacity[s][h]:
                penalty += (source_total - source_capacity[s][h]) ** 2  # Penalty for over-supply

            # Add cost of supplying this energy
            total_cost += sum(chromosome[s, :, h]) * cost_per_unit[s]

    # Return negative total cost and penalties as fitness (maximize fitness)
    return -np.log1p (total_cost + 10 * penalty)
```

Design Justification:

- Logarithmic Scaling: Prevents numerical overflows and reduces the sudden changes in fitness scores.
- Penalty multiplier (* 10): Focuses more on meeting constraints than on costs.
- Squared Penalties - Highly penalizes vital infractions (convex penalty).
- Shift, +1 of log - Gives zero-cost solutions upfront.

This complex fitness procedure leads the development of practical and reasonably priced solutions.

2.3 Worked Example: Manual calculation of fitness

Given an illustration of how to analyze the calculation of fitness in simpler terms, we have the following example:

- Two sources: S1 and S2 (energy)
- Three nodes: N1, N2, N3
- 1 hour: h = 0
- Cost per unit: C = 2, 1.5
- D = 100, 80, 90
- Source capacity: C = 200, 150
- Chromosome sample: (nodes 0,1,2)

Source 1: [30, 30, 40]

Source 2: [70, 40,60]
Calculation:
1. Total cost
S1: (30 +30 + 40) * 2 = 200
S2: (70 + 40 + 60) * 1.5 = 255
Total cost = 200 + 255 = 455

2. Demand penalty
Node 0: (100-(30 +70))^2 = 0
Node 1: (80-(40+40))^2 = 0
Penalty demand= 0

3. Source capacity penalty
Source 1: 30+30+40 = 100 < than equals to 200 => 0
Source 2: 70+40+60 = 170 > 150 => (170 – 150)^2 = 400
Penalty capacity = 400

4. Fitness calculation
Fitness = -log (1+ 455+10*400) = -log (1+455+4000) = -log (4456) approximately equal to –8.401

Whereas Source 2 suffers a significant penalty for overload that lowers the fitness score. The earlier example demonstrates how the fitness function encourages economical solutions and imposes necessary limitations.

3. Parallelization Strategy and Algorithm Implementation

Fitness Evaluation:

The evaluate_fitness() function calculates the following:

- The overall cost of energy supplied
- Penalties for under-supply at nodes and oversupply from sources.

```
# Evaluate the population using either parallel or serial processing
def evaluate_population(pop, parallel=True):
    if parallel:
        with Pool(processes=multiprocessing.cpu_count()) as pool:
            return pool.map(evaluate_fitness, pop)  # Parallel fitness computation
    else:
        return list(map(evaluate_fitness, pop))  # Serial fitness computation
```

The tool used for this research is multiprocessing.Pool from the Python standard library. It distributes chromosomes across available CPU cores so fitness can be calculated in parallel. It can scale to larger populations of chromosomes (for example populations of 50-100 chromosomes) or, can accommodate more complicated constraints.

Selection

```
# Select top half of the population based on fitness
def selection(pop, fitnesses):
    idx = np.argsort(fitnesses)[-POP_SIZE//2:]  # Get indices of best performers
    return [pop[i] for i in idx]  # Return selected chromosomes
```

After fitness evaluation, a selection of the top-half population is made. We did not parallelize because it is light computation (sorting and slicing). Overall fast running on a single core.

Crossover

```
]  # Perform crossover between two parents to produce a child
   def crossover(parent1, parent2):
       if np.random.rand() > CROSS_RATE:
           return parent1.copy()  # No crossover, return parent
       point = np.random.randint(1, HOURS)  # Random crossover point along time axis
       child = parent1.copy()
       child[:, :, point:] = parent2[:, :, point:]  # Swap the latter part from parent2
       return child
```

Crossover takes parent genes and combines them along a time axis to produce offspring. It is sequential. It can be parallelized with joblib or concurrent in the future, if there are multiple children generated simultaneously.

Mutation

```
# Mutate the chromosome by slightly changing a few values
def mutate(chromosome):
    for _ in range(int(MUT_RATE * NUM_NODES * HOURS)):
        s = np.random.randint(NUM_SOURCES)  # Random source
        n = np.random.randint(NUM_NODES)  # Random node
        h = np.random.randint(HOURS)  # Random hour
        # Apply small Gaussian noise and clip to valid range [0, 1]
        chromosome[s][n][h] = np.clip(chromosome[s][n][h] + np.random.normal(0, 0.1), 0, 1)
    return chromosome
```

Mutation generates small random changes with Gaussian noise. It is also sequential and very fast, with the amount of simple matrix arithmetic. Mutation could be vectorized using NumPy, or parallelized when using GPU libraries, such as CuPy.

3.1 Training Strategy for Optimization

In order for the optimization process to be as fast and scalable as possible for the real-time smart grid energy distribution domain, the training function is performed in parallelism, specifically within the fitness assessment. The parallelism and structure of the training loop is included below:

- Fitness evaluation is entirely parallelized using multiprocessing.Pool().map(). The fitness evaluation is conducted in parallel simultaneously across multiple CPU cores for each chromosome.

- Selection is not parallelized because it is just sorting and slicing and can run so quickly that it does not warrant parallelization.

- Crossover and Mutation are not currently parallelized, but they are both quite light operations and completed in sequence. Crossover and Mutation, however, may be parallelized in future versions to accommodate a bigger population or time constraints in real-time situations.

3.2 Execution Strategy

- The Genetic Algorithm was trained over 50 generations with 20 chromosomes.

- The algorithm as a whole uses the evaluate_population(pop, parallel=True) method in each generation to evaluate the entire population at once, before selecting the best solutions to crossover and mutate.

- Both best selected solutions in the population, crossover and the mutation are done to continue exploring new solutions.

- This repeats until all generations are complete, with the highest fitness value identified as the optimized energy distribution strategy.

- We also recorded the time it took to execute the parallel fitness evaluations and serial fitness evaluations to compare their relative performance, with the parallel execution time always being less than the serial evaluation time.

3.3 Evaluation Plan

To assess the effectiveness of the parallelization technique, we evaluated important performance parameters for the serial and parallel implementations. Our intention was to examine how parallelism might impact training efficiency and optimization quality.

Metrics Compared:

- Training time: It refers to the total time necessary to consider one generation (fitness evaluation, selection, crossover, mutation).

- Latency: It refers to the amount of time necessary to evaluate the fitness of the entire population for a single generation. The greatest performance benefit of parallelism occurs at this metric.

- Accuracy (Fitness Score): The maximum fitness score achieved in a generation. This metric is the same for the serial and parallel versions and shows that the process was done faster while keeping the optimization quality the same.

3.4 Implementation on a Sample dataset

In order to show the effectiveness of the Genetic Algorithm (GA) as a method of smart grid optimization we implemented the model with a sample dataset that was comprised of the following:

- 10 energy-consuming nodes.
- 3 energy sources
- 24-hour time cycle

These assumptions stimulate to a real-world smart grid, with an identification of both demand

and capacity variance across time.

3.5 Dataset Description

We applied NumPy's random function to synthesize realistic data for our sample dataset.

Defined parameters:

- NUM_NODES: Represents 10 unique demand points, such as homes, districts, or commercial businesses.
- NUM_SOURCES: 3 represents three renewable energy sources such as solar, wind, or hydro.
- HOURS: Represents all energy usages over the course of a day divided into hourly block components (assuming 24 hrs).
- POP_SIZE: Each generation consists of a population of 20 chromosomes or solutions.
- GENS: The algorithm runs for 50 generations, allowing the population to evolve and get close to optimum.
- MUT_RATE: 0.1 means the mutation rate of 10% allows for diversity between solutions in the population.
- CROSS_RATE: 0.7 means there's a 70% chance there's chance crossover in the population so that new possible solutions could emerge.

3.6 Data Generation

The Demand Matrix (demand_matrix) is a 10 x 24 matrix that represents each node's electrical demand at each hour. Each value is randomly selected from a range of 50 to 150 units.

```
# Generate a matrix representing the electricity demand of each node for each hour
demand_matrix = np.random.randint(50, 150, size=(NUM_NODES, HOURS))
```

Source Capacity (source_capacity) is a 3 x 24 matrix reflecting the amount of energy each source can give each hour. Random values range from 300 to 500 units per hour.

```
# Generate source capacities for each source and hour
source_capacity = np.random.randint(300, 500, size=(NUM_SOURCES, HOURS))
```

Cost per Unit (cost_per_unit) is a 1D array of size 3 that represents the cost of supplying one unit of energy from each source. Random floating-point numbers from 1.0 to 3.0.

```
# Random cost per unit of energy for each source
cost_per_unit = np.random.uniform(1.0, 3.0, size=(NUM_SOURCES,))
```

Chromosome Structure:

Each chromosome serves as a potential way to allocate energy in space and is initially represented as a 3D array with dimensions (NUM_SOURCES, NUM_NODES, HOURS).

Axis 0 (sources): 3.
Axis 1 (nodes): 10.
Axis 2 (time slots): 24.

Each value in the chromosome indicates the energy the source allocates to the node each hour.

```
# Function to create a random chromosome (an individual solution)
def create_chromosome():
    return np.random.rand(NUM_SOURCES, NUM_NODES, HOURS)
```

Population Initialization

We initialize a population of 20 such chromosomes using:

```
# Initialize the population with random chromosomes
def initialize_population():
    return [create_chromosome() for _ in range(POP_SIZE)]
```

This provides a diverse set of starting solutions for the GA to evolve from.

3.7 Execution Time Comparison: Serial vs Parallel Implementations

To determine the efficiency of parallelization, the execution delays of the fitness evaluation stage of the Genetic Algorithm were measured and compared between the parallel and serial versions. This tells us how parallel processing can help performance, especially for high-dimensional smart grid optimization evaluations. The fitness evaluation stage was performed once using serial computation and once using parallel computation with the same data and logic. The only difference was the execution (serial or parallel), which made for a proper comparison.

```
# Print execution times for comparison
print(f"Parallel Execution Time: {parallel_time:.2f}s")
print(f"Serial Execution Time:   {serial_time:.2f}s")
```

```
Parallel Execution Time: 4.87s
Serial Execution Time:   2.49s
```

We can see that the serial execution was faster than the parallel excecution.

Although parallel processing is generally faster, these are a few of the reasons that serial execution is faster:

- Small population size: Since there are only 20 chromosomes to evaluate, the cost of creating a few processes wasn't worth the benefits of splitting up the work.

- CPU Process Spawning Overhead: If you want to reduce overhead from CPU process spawning, consider multiprocessing. Pooling means starting up a few processes, passing them data, and collecting results. The pool requires time which might not be worth it if the workload isn't big enough.

- Efficient Serial Code: When using the serial version, it also had a fast map() function, and since there weren't inter-process connection it would improve performance with small scale tasks.

Parallel executions help to address large-scale problems. However, serial execution may be more appropriate for small data sets, as the overhead will be much lower, than for parallel execution. In spite of that, the code is designed to be scalable for realistic smart grid scenarios (e.g., above 100 nodes or 1000 chromosomes), parallelism is particularly important.

4. Performance Analysis and Result Interpretation

In the project, Genetic Algorithm (GA) is applied to optimize distribution of energy from multiple nodes over a 24 hour period, with the aim of minimizing total cost while minimizing total cost while satisfying node demands and respecting source capacities.
To assess the effectivity of GA, a greedy algorithm was applied as a baseline model.

4.1 Greedy Algorithm

Greedy Algorithm is a model which builds a solution piece by piece, choosing the piece that provides the most and immediate benefits. Greedy Algorithm is known for its simplicity and speed. (Greedy Algorithm Tutorial, 23)
How Greedy Algorithm works:

i) Evaluating all possible choices at the current step
ii) Selecting the one that immediately seems like the best solution (e.g., lowest cost)
iii) Making that choice and updating the state
iv) Repeating until the problem is solved (Greedy Algorithm Tutorial, 23)

4.2 Greedy Algorithm Implementation

In this project, the Greedy method:

- Sorts sources by cost per unit
- Allocates energy from the cheapest source first, respecting source capacity
- Moves to the next cheapest source until each node's demand is met

The parameters:
demand_matrix: Matrix of shape (NUM_NODES, HOURS), where each element represents the energy demand of a node at a given hour
source_capacity: Matrix of shape (NUM_SOURCES, HOURS), where each element represents the total available energy for a source at a given hour.
cost_per_unit: Array of shape (NUM_SOURCES), which represents the cost per unit of energy for each source.
It returns:
allocation: array of shape (NUM_SOURCES, NUM_NODES, HOURS) representing the energy allocated from each source to each node for every hour.
duration: Time taken for the greedy algorithm to execute

```python
def greedy_energy_allocation(demand_matrix, source_capacity, cost_per_unit):

    start = time.perf_counter() # start measuring executing time

    # Initialize an empty allocation matrix to store energy assignments
    allocation = np.zeros((NUM_SOURCES, NUM_NODES, HOURS))

    # Sort sources by increasing cost per unit, so cheaper sources are used first
    sorted_sources = np.argsort(cost_per_unit)

    # Loop over each hour in the 24-hour cycle
    for h in range(HOURS):
        # Loop over each node (energy consumer)
        for n in range(NUM_NODES):
            remaining_demand = demand_matrix[n][h]  # Retrieve the energy demand of node 'n' at hour 'h'

            # Loop through sources in order of increasing cost
            for s in sorted_sources:
                # Calculate how much energy has already been used from this source at hour 'h'
                used_capacity = allocation[s, :, h].sum()

                # Determine the remaining capacity of this source at hour 'h'
                available = source_capacity[s][h] - used_capacity

                # Allocate the minimum between available energy and remaining demand
                assigned = min(available, remaining_demand)
```

```python
                # Assign the energy to the node from this source at this hour
                allocation[s][n][h] = assigned

                # Reduce the remaining demand for the node
                remaining_demand -= assigned

                # If the demand is fully satisfied, move to the next node
                if remaining_demand <= 0:
                    break
    duration = time.perf_counter() - start  # Total execution time
    # Return the completed energy allocation matrix
    return allocation, duration
```

4.3 Performance Comparison

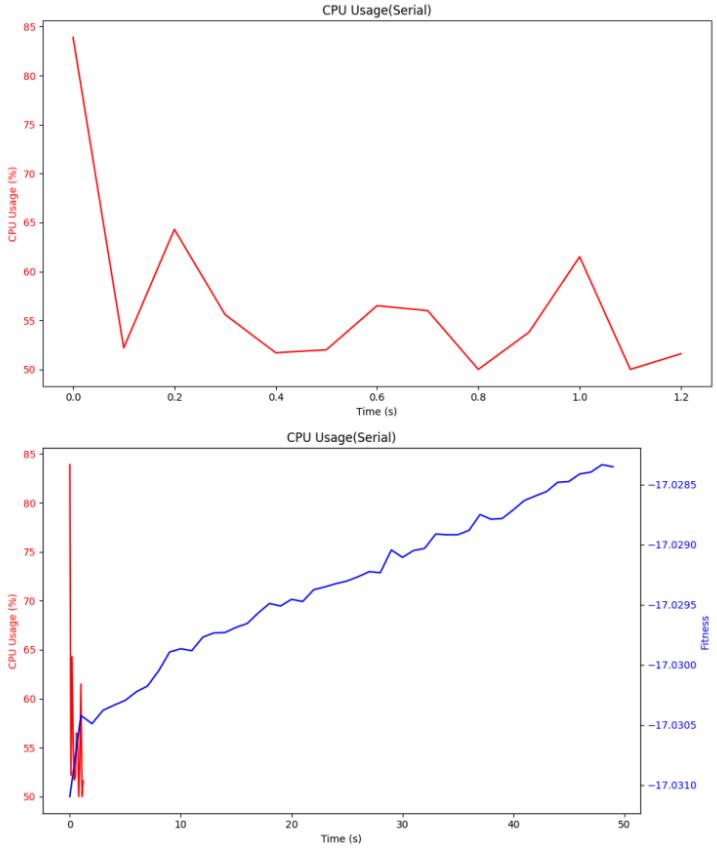
```
Greedy Fitness: -11.02
Best GA Fitness: -17.03
Greedy Algorithm Runtime:   0.0019821699999056364 seconds
```

- A fitness score closer to 0 means better. From this, we can conclude that greedy algorithms have a better fitness than a genetic algorithm (-11.02 compared to -17.03), while also processing it very quickly (~1.98 milliseconds).
- The Greedy Algorithm, although simple probably found a solution with a fitness of -11.02, likely due to its ability to aggressively minimize cost

- The Genetic Algorithm, while designed to search the solution space more globally, returned a less optimal result with -17.03, possibly due to non-optimal parameter tuning (e.g., population size, mutation rate, or insufficient generations).
- This can show that more complex algorithms like Genetic Algorithms may not guarantee better results unless tuned properly

4.4 Plot Convergence Curves/ CPU Usage

For Serial:



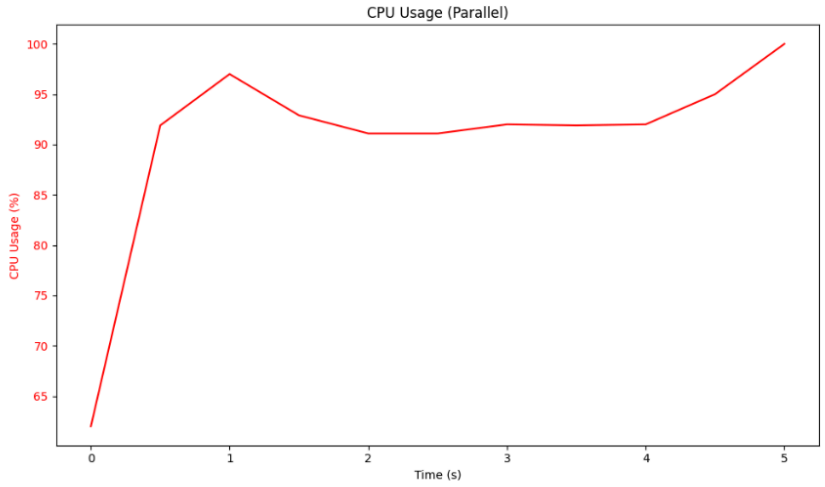
CPU Usage (Red line):

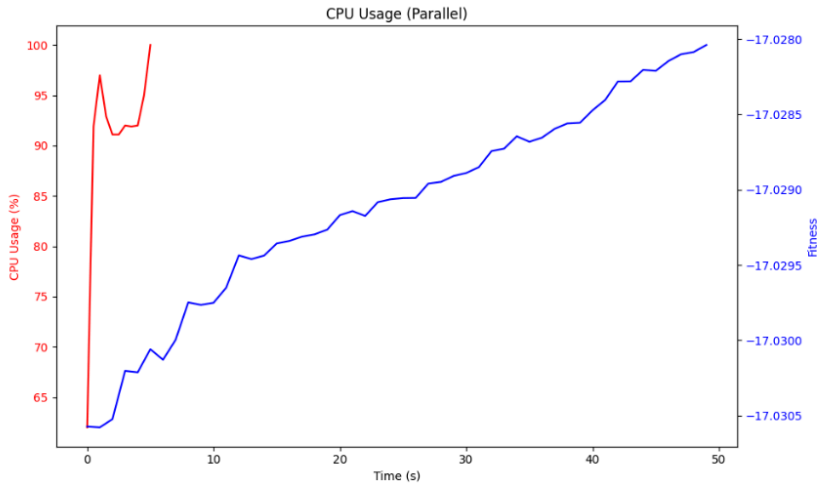
- At the start CPU usage spikes up to ~84%, then sharply drops to ~50%–65%. This represents the initialization of the serial process which includes data loading and population initialization.
- CPU usage drops significantly after the initial peak to around 52%, and continues to fluctuate. This suggests serial algorithm stabilizes after setup, performing computations at a consistent but variable load.
- After the spike, CPU usage mostly stays between 50% to 65% which means the operation isn't CPU heavy.

Fitness Score (Blue Line):

- Fitness improves gradually from ~ -17.0311 to ~ -17.0284, which is moving closer to zero.

For Parallel:





CPU Usage (Red line):

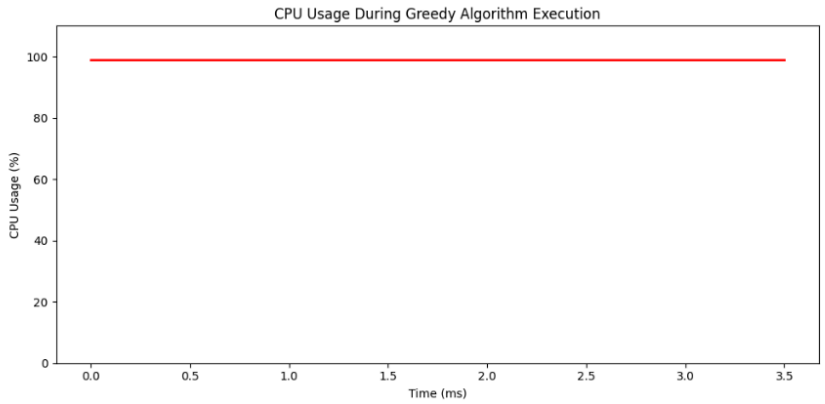
- At the start CPU usage starts at ~60%, then sharply increases to about 93% and stays between 90-95% for a while until it reaches 100% spikes up to ~84%. This represents that after initialization of the parallel process which includes data loading and population initialization, the CPU usage increased a lot.
- CPU usage increases significantly due to the allocation of multiple cores
- After the spike, CPU usage mostly stays between 90% to 100% which means the operation is very CPU heavy

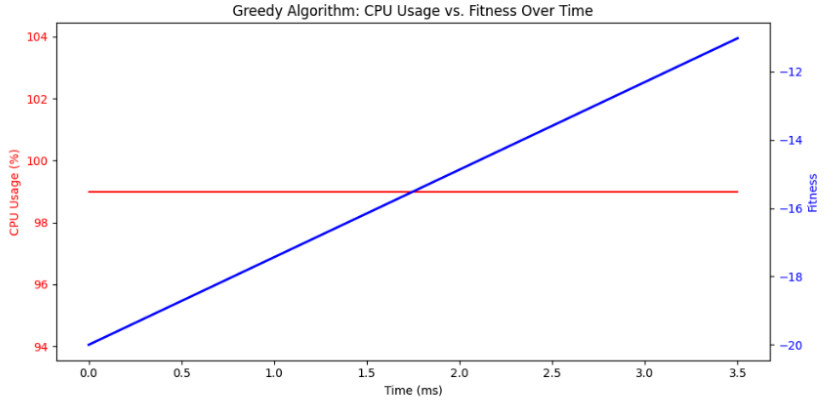
Fitness Score (Blue Line):

- Fitness improves gradually from ~ -17.0305 to ~ -17.0280, which is moving closer to zero.

For Greedy:

```
Greedy Runtime: 0.003849 seconds
Estimated CPU Usage During Greedy: 99.00%
```





Greedy Algorithm runs very fast, it only takes it 3.8 milliseconds to finish the execution

CPU Usage (Red line):

- From the beginning to the end, the CPU Usage stays constant at 99%
- In that short period of time, the CPU was fully utilized by Greedy Algorithm

Fitness Score (Blue Line):

- Fitness improves from ~-20 to ~-11 in a very quick succession as greedy algorithm aggressively minimizes cost

Runtime Comparison between Serial, Parallel and Greedy:

```
Runtime Comparison (in seconds):
Parallel GA Runtime: 4.87
Serial GA Runtime:   2.49
Greedy Algorithm Runtime:   0.003849
```

From here, we can see that Parallel is the slowest taking 4.87 seconds, followed by Serial which takes 2.49 seconds. The fasted here is Greedy which takes only 3.8 seconds.

4.5 Trade-offs Between Convergence Speed and Solution Quality

We run the GA with other new parameters, to understand its effect on convergence and solution quality.

The new parameters:

	Population Size	Generations	Mutation Rate	Crossover Rate
A (Normal one used previously)	20	50	0.1	0.7
B (Larger population)	50	50	0.1	0.7
C (More generations)	20	100	0.1	0.7
D (Higher Mutation rate)	20	50	0.2	0.7
E (Higher Crossover Rate)	20	50	0.1	0.9

After Execution:

	Execution Time	Best Final Fitness
A (Normal one used previously)	5.68	-17.027953
B (Larger population)	6.93	-17.026491
C (More generations)	12.18	-17.025860
D (Higher Mutation rate)	6.12	-17.027198
E (Higher Crossover Rate)	4.47	-17.027935

Interpretation:

A: The baseline model for comparison

B: Additional population increases the number of solutions present in a generation, which may lead to more diverse and higher-quality offspring. Here, however, the fitness dropped slightly and execution time increased slightly. That means the added diversity did not lead to better solutions.

C: Doubling the number of generations significantly increased execution time but quality improvement in the solutions is very minimal. It may indicate that the GA probably had already converged to a good solution early and more time given simply made the computation lengthy with no payoff.

D: A higher mutation rate is meant to increase genetic diversity. For this, the final fitness was scarcely different from the baseline, and the running time increased slightly. This suggests the initial mutation rate was optimal to begin with, and further increase had little effect.

E: High crossover rate forces more mixing, which will accelerate convergence by allowing more powerful traits to move quickly. In this case, the fitness saw an improvement a little better than the baseline, and the execution time decreased, making this the best configuration overall.

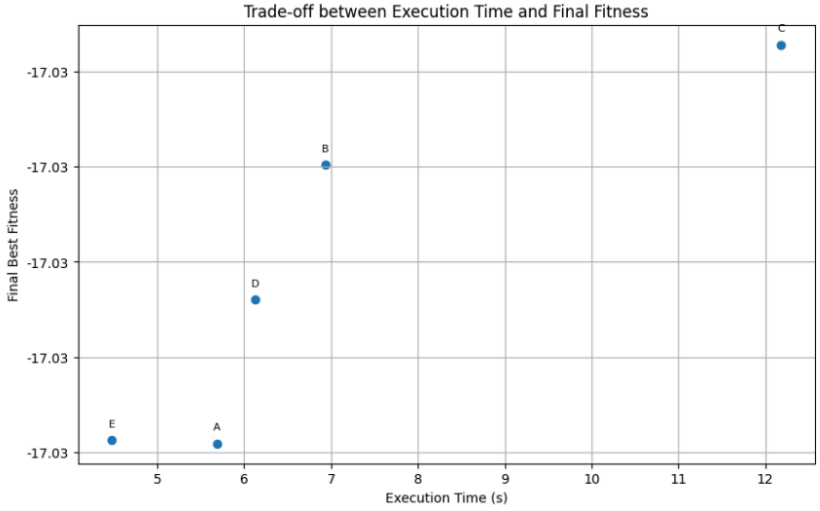


Fig. Trade-off between time and final fitness

5. Professionalism and Communication

5.1 Individual Reflections

Aman Kumar Shah

In this project, I oversaw making the mathematical formulation of the problem. My responsibility was to read between the lines of the fundamental needs of the smart grid energy optimization and paraphrase them into a well-organized problem-solving process that the rest of the system was to be constructed on. To han[13] this work, one had to have a good grasp of theory and practice of optimization. I needed to be analytical in order to identify what the system was attempting to do and how to define those goals in such a manner that could be processed by an algorithm. The balance between clarity, complexity, and relevance had been difficult, in particular, what to include and what to communicate.

During the process, there were some situations when I needed to reconsider the approach or re-evaluate some of the previous assumptions to become more consistent with the implementation work of the team. These difficulties taught me the lesson about flexibility, cooperation, and base ideas on research. I also got to know how important it is to communicate technical material in a manner that can be easily understood by a technical and nontechnical person. Through this experience, I have developed the capacity to frame and describe technical issues and this has strengthened the importance of clear problem statement in any data-driven or algorithm project.

Shrena Shakya

This group assignment has been beneficial and educational. All members actively participated in the discussions and made contributed in every stage of the project. Since everyone was helpful, friendly and supportive the working environment was productive. I am grateful to our team leader who he took the initiative in providing guidance, and monitoring our progress.

This project helped me learn more about the genetic algorithm and help me gain more insight of the ways in which fitness functions affect genetic algorithm performances by learning about various encoding methods used in actual smart grid energy supply systems.

It took both mathematical understanding and innovative thinking to develop the fitness function that took into account the cost, limitation violation and real-time energy allocation requirements. I gained a greater understanding of the present level of evolutionary improvement by researching how the GA improved across generations using various penalty techniques.

The project enhanced my technical and problem-solving abilities. The learning experience was generally positive and it was possible with help of the team's friendly and cooperative environment.

Rojit Khadgi

The execution of the present project using the smart-grid GA optimization has been quite difficult yet highly gratifying. At the onset of this project, I grew in my appreciation of how to convert a complex problem of a power system into a format to be fed into genetic algorithms. Addition of the generator levels of dispatch and network routing into the chromosomes widened my horizon of cautious representation, the option of using continuous or discrete forms of genes, the type of penalty functions when actually there was a breach of constraints and that they would turn out to feasible solutions even in the population.

Having written the multi-objective fitness function (cost, losses, demand satisfaction), I knew how I was able to make a trade between conflicting objectives. Weighted sums and penalty parameters to learn not only how vulnerable GA convergence is to minute design decisions, but also to accidentally rather influential ones, consider weighted sums and penalty parameters. I also familiarized myself with the value of profiling and parallelization, by distributing generations of fitness evaluations across the cores via Python *multiprocessing.Pool*, we were able to reduce the execution time by almost a half, and this was vital because we incremented the test runs to 100 generations and beyond, as well as the population size.

When it comes to teamwork, work with my colleagues allowed me to develop the communication skills. I could receive, as well as provide positive feedback, be it in respect to the understandability of the *evaluate_population()* function or joint efforts on trying to crack an indexing fault in the mutation operator, through regular code review. I learned that unambiguous records (incode user comments and our joint project notebook) were essential not only in maintaining a fully up to date team but also during the period when there were multiple concurrent development branches rolling back in unison.

This kind of learning experience will certainly be helpful in future. I now feel free to experiment with other optimization challenges that can be in power systems or in logistics or in hyperparameters tuning in machine learning inspired by the idea of GA. And, finally, the assistive patterns that we could come up with (sources of control standards, peer-programmed programs, and ambassadoring systems with testing) would help me work in the future cooperation. This has enhanced my technical skills not only but also portrayed how powerful teamwork is in solving the intricate problems that are at the disposal of the real world.

Kaushik Raj Joshi

My primary responsibility in the project was to create Performance Analysis and Result Interpretation for Greedy Algorithm and Genetic Algorithm. From the start, our team was very supportive, where each member got to show their strengths and collaborate with one another.

Developing two algorithmic solutions necessitated me to communicate with team members continuously. This ensured that our development stayed on track with the project's overall objective. The ability of my team to share ideas, offer feedback, and exchange knowledge allowed me to iterate both Greedy and Genetic Algorithm logic. There were a few moments

when I was stuck specifically in designing a good fitness function for GA I was able to overcome them

What surprised me the most in this project was how well the team collaborated. Everyone was open to learning from each other. This allowed us to adapt quickly whenever adaptation was needed. From this project not only did I enhance my understanding of Genetic and Greedy Algorithms, but I also developed the ability to increase my teamwork abilities and how to communicate within it.

Sewanty Upreti

At the beginning of this project, I had a significant amount of trouble understanding multiprocessing in Python. The whole logic of parallel execution, spawning processes, and completion across cores was disorientating to me. However, I was able to build my understanding with self-research and with assistance from my group members over time. I viewed authentic online articles and tutorials, and even YouTube videos really were beneficial in shaping the topic at hand. In that respect, I was able to learn more than just multiprocessing out of this assignment, I learned more about Genetic Algorithms (GA). Genetic algorithm is quite a vast topic I learned how GA worked from top to bottom, like population initialization, those fitness function evaluations, selection, crossover, and mutation, and it can be used to tackle real-world optimization problems like smart grid energy distribution.

The collaborative work as a team made for a great experience as well. We had trust in each other's strengths and utilized continuous open communication as a team from start to finish. We provided space for group discussion, were timely with progress sharing as a group, and developed collaborative agreements for how we would make team decisions. We shared items of work continuously and assisted each other along the way in debugging and while working on code implementation. The teamwork, on top of the technical learning experience, made my work more difficult but equally a positive experience overall. I am therefore more excited about using GA and multiprocessing on real-world, data science-based problems due to this team-oriented experience.

5.2 Contribution Table

Name	Contribution
Kaushik Raj Joshi	Lead the team, document compilation, developed the code, reported on Performance Analysis and Result Interpretation
Sewanty Upreti	Supported in coding, reported on Parrallezation strategy and algorithm implementation
Shrena Shakya	Supported in coding, reported on Encoding and Fitness Function Design
Rojit Khadgi	Supported in coding, reported on Mathematical Formulation of the Problem
Aman Kumar Sah	Supported in coding, reported on Mathematical Formulation of the Problem

Github Link (Code and Similarity Report): https://github.com/Kaushik22864/MLPC-Group.git

References

Coello Coello, C. A. (2002). Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, *191*(11), 1245–1287. https://doi.org/10.1016/S0045-7825(01)00323-1

Deb, K. (2000). An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, *186*(2-4), 311–338. https://doi.org/10.1016/s0045-7825(99)00389-8

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press; MIT Press. https://mitpress.mit.edu/9780262631853/an-introduction-to-genetic-algorithms/

Fang, X., Misra, S., Xue, G., & Yang, D. (2012). Smart Grid — The New and Improved Power Grid: A Survey. *IEEE Communications Surveys & Tutorials*, *14*(4), 944–980. https://doi.org/10.1109/surv.2011.101911.00087

McKee, A. (2024, July 29). *Genetic Algorithm: Complete Guide With Python Implementation*. Datacamp.com; DataCamp. https://www.datacamp.com/tutorial/genetic-algorithm-python?

Python. (n.d.). *multiprocessing — Process-based parallelism — Python 3.8.3rc1 documentation*. Docs.python.org. https://docs.python.org/3/library/multiprocessing.html

Stephy Akkara, & Immanuel Selvakumar. (2023). Review on optimization techniques use for smart grid. *Measurement: Sensors*, *30*, 100918–100918. https://doi.org/10.1016/j.measen.2023.100918

GeeksforGeeks. (2025, July 23). *Introduction to Greedy Algorithm - Data Structures and Algorithms Tutorials*. https://www.geeksforgeeks.org/dsa/introduction-to-greedy-algorithm-data-structures-and-algorithm-tutorials

ITS66604_0369834_GRUPASGNMT

ORIGINALITY REPORT

7%	6%	3%	7%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Taylor's Education Group Student Paper	3%
2	dergipark.org.tr Internet Source	1%
3	hdl.handle.net Internet Source	1%
4	Submitted to Southern New Hampshire University - Continuing Education Student Paper	<1%
5	Submitted to Colorado Technical University Student Paper	<1%
6	Submitted to Syracuse University Student Paper	<1%
7	Submitted to Empire State College Student Paper	<1%
8	www.scielo.org.mx Internet Source	<1%
9	epdf.pub Internet Source	<1%
10	ouci.dntb.gov.ua Internet Source	<1%
11	de Jesus, Adriana Mar Brazuna. "The Use of Cooperative Flexibility to Improve the Energy Communities' Resilience", Universidade NOVA de Lisboa (Portugal) Publication	<1%

12	herkules.oulu.fi Internet Source	<1%
13	pure.tudelft.nl Internet Source	<1%

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		