REWARD-ORIENTED TASK OFFLOADING UNDER LIMITED EDGE SERVER POWER FOR MULTIACCESS EDGE COMPUTING

Minseok Song , Yeongju Lee and Kyungmin Kim

TEAM 13

Adarssh KG :CB.SC.U4CSE23658

Ravindran G :CB.SC.U4CSE23647

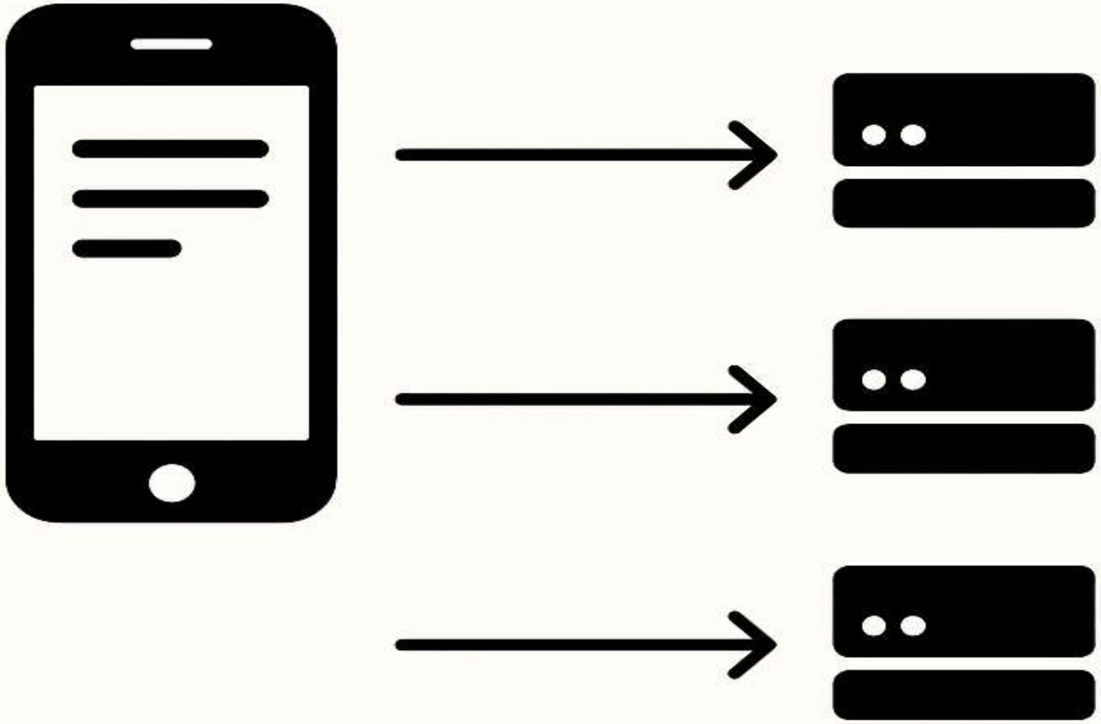
Pratyush Yadav :CB.SC.U4CSE23641

Paarthu Reddy :CB.SC.U4CSE23639

Sanjesh Nandha K :CB.SC.U4CSE23242

Deepak S :CB.SC.U4CSE23267

The Edge Problem



- Edge servers have limited CPU capacity and a strict global power budget
- Some servers get overloaded
- Power consumption exceeds the system budget
- Total reward (profit from completed tasks) drops

Problem: How to decide which tasks to offload and to which servers, so that total reward is maximized without exceeding power limits?

TWO-PHASE SOLUTION

PHASE-1

•Calculate how much workload (utilization cap) each server can handle under the global power budget.

•Uses reward-to-power efficiency to set these caps.

PHASE-2

•Assign tasks to servers within the given caps.
•Formulated as a Minimum Cost Maximum Flow (MCMF) problem.
•Two approaches: EAA-NTS (no task split) and EAA-TS (with task split).

WHY Edge Sim Py. ??

- Provides a realistic simulation of task arrivals, server workloads
- Ability to **track resource usage and power consumption**
- Let's us test and compare algorithms (MUD, EAA, CI) under controlled scenarios.

WORKFLOW

SYSTEM MODELLING — Defining edge servers' specifications and tasks

PHASE 1 — MUD algorithm implementation

PHASE 2 — EAP algorithm implementation

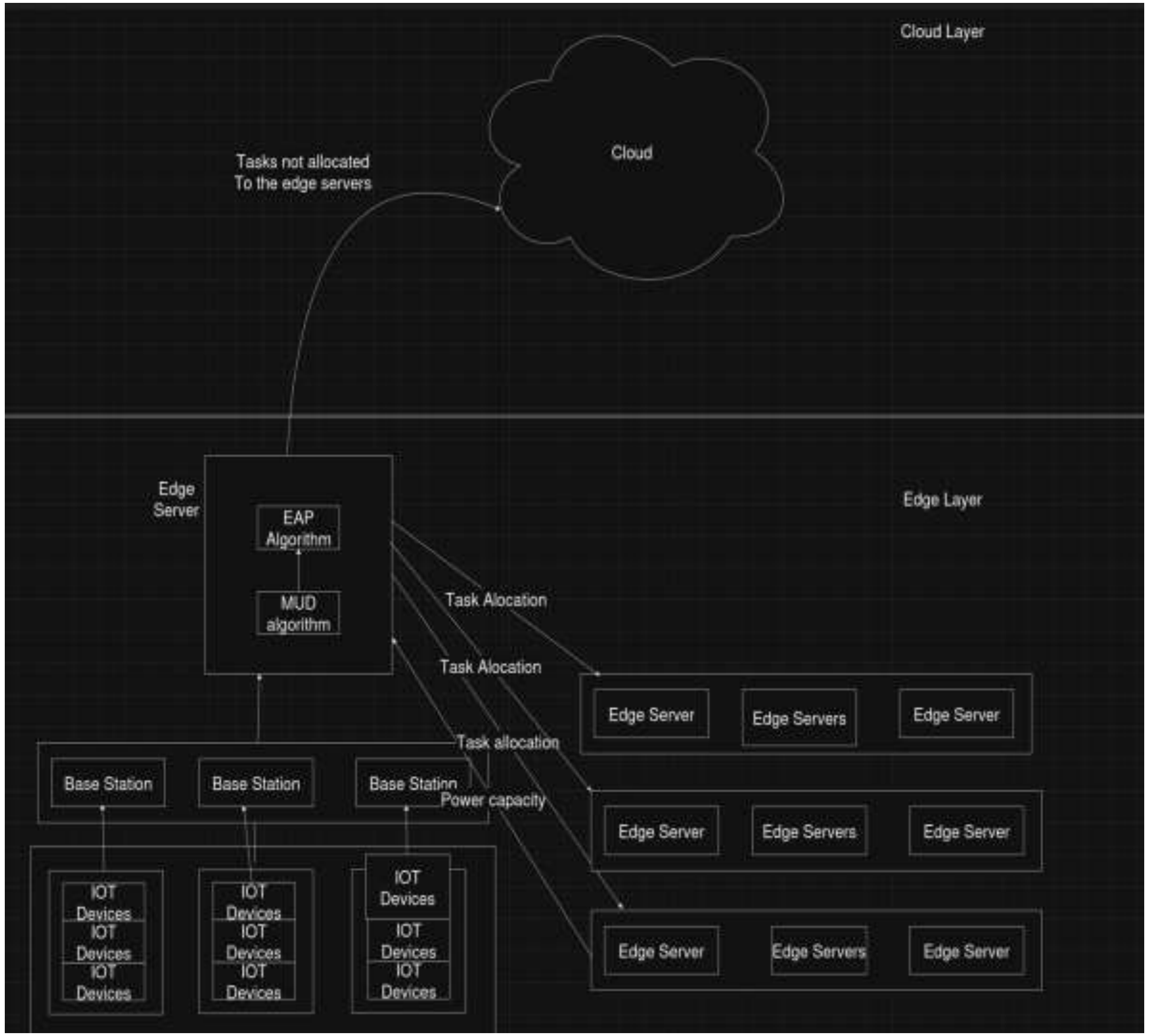
FINAL IMPLEMENTATION — Final result through simulation

IMPLEMENTATION STRATEGY:

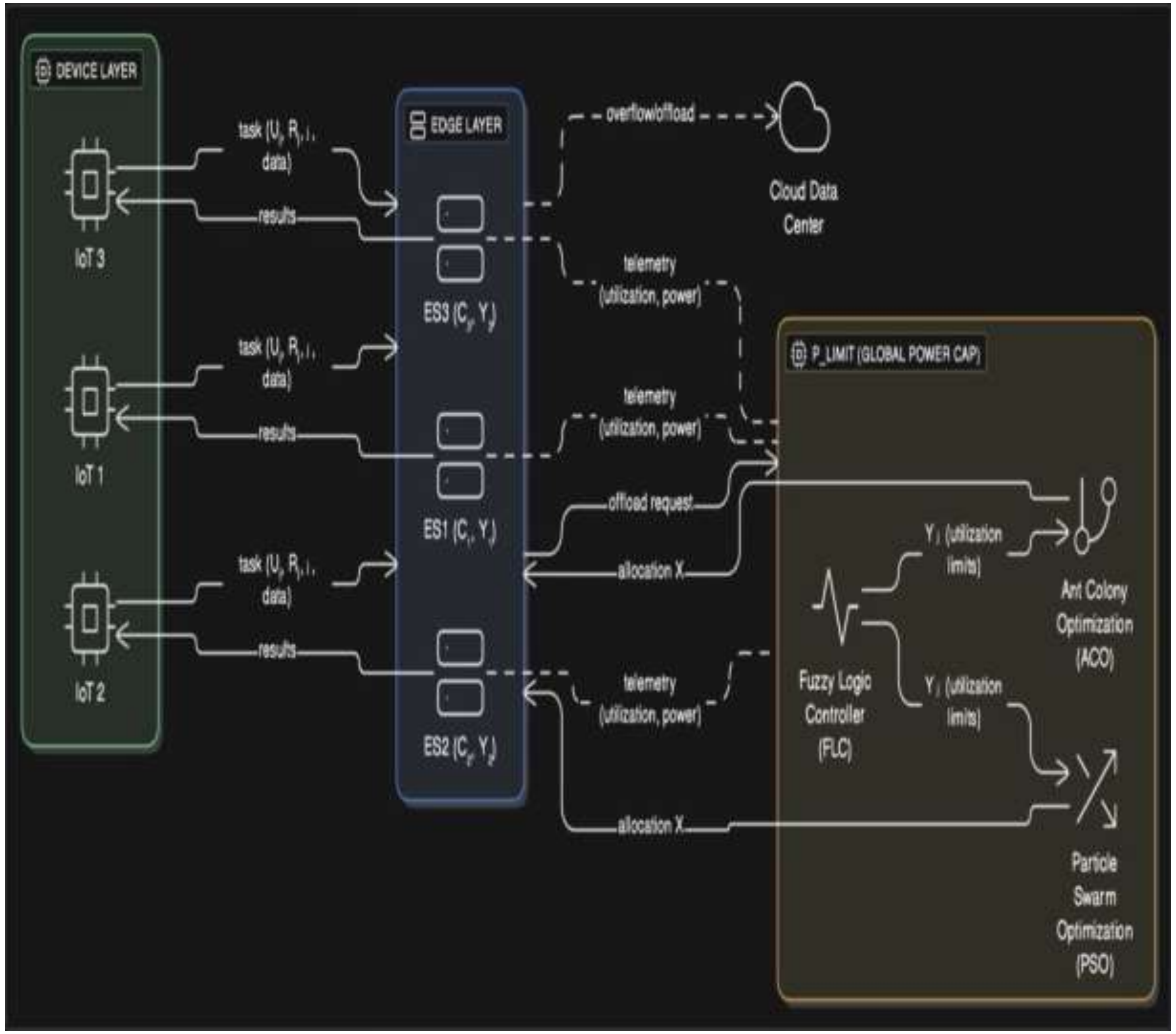
- **mud.py** → Phase 1 algorithm: finds max utilization caps for each server under the global power budget.
- **eaa_nts.py** → Task selection policy (which tasks should be offloaded).
- **eaa_ts.py** → Server selection policy (which edge server executes each task).
- **main.py** → Integrates everything with **EdgeSimPy**:
    - Defines the environment (tasks, servers, events).
    - Applies MUD for utilization caps.
    - Uses EAA-NTS and EAA-TS for final task offloading.
    - Runs the simulation and outputs performance metrics (reward, utilization, power).

ARCHITECTURE DIAGRAM

With edge implementation



With CI implementation



MUD ALGORITHM

```python
import random
class MUDAlgorithm:
    def __init__(self, power_limit, tasks, edge_servers, power_function):
        self.power_limit = power_limit
        self.tasks = tasks
        self.edge_servers = edge_servers
        self.power_function = power_function
        self.Y = {}

    def run(self):
        X_tmp = {task.id: 0 for task in self.tasks}
        P_used = 0
        S_task = set(task.id for task in self.tasks)
        self.Y = {es.id: 0 for es in self.edge_servers}

        while S_task:
            best_valgo = -float('inf')
            best_task_id = None
            best_es_id = None
            best_palgo = 0
            for task in self.tasks:
                if task.id not in S_task:
                    continue
                for es in self.edge_servers:
                    if task.coverage[es.id] == 0:
                        continue
                    current_usage = sum(
                        t.usage for t in self.tasks if X_tmp[t.id] == es.id)
                    current_util = current_usage / es.capacity if es.capacity > 0 else 0
                    before_power = self.power_function(es, current_util)
                    after_util = (current_usage + task.usage) / \
                        es.capacity if es.capacity > 0 else 0
                    after_power = self.power_function(es, after_util)
                    P_algo = after_power - before_power
```

Set all servers’ utilization to zero and calculate idle system power.
We search while tasks remain and power ≤ budget, search feasible assignments.
For each task–server pair, reward-to-power efficiency is calculate.

Limits each server’s CPU usage to stay within total power cap.
Power capping is the process of setting a maximum limit on the power consumption of a system to prevent overuse and ensure stable operation.

```python
                if P_algo <= 0:
                    continue
                V_algo = task.rewards[es.id] / P_algo
                if V_algo > best_valgo and (P_used + P_algo) <= self.power_limit:
                    best_valgo = V_algo
                    best_task_id = task.id
                    best_es_id = es.id
                    best_palgo = P_algo
        if best_task_id is not None:
            P_used += best_palgo
            X_tmp[best_task_id] = best_es_id
            S_task.remove(best_task_id)
        else:
            break
    for es in self.edge_servers:
        self.Y[es.id] = sum(
            t.usage for t in self.tasks if X_tmp[t.id] == es.id)
    return self.Y
```

CONT...

It calculates how much extra power the server ould use if the task is added, and then divides the task's reward by this power increase. This gives a reward-per-power ratio.

The algorithm always picks the t**ask–server pai**r with the **highest ratio.**

EAA with Task Split

- Objective : Maximize total reward from offloading by choosing where each task runs, under per-server utilization caps derived from power limits and radio coverage constraints

- Convert reward maximization into a minimum-cost flow by putting negative unit reward on task to server edges, so minimizing cost equals maximizing reward.

- Task splitting is enabled naturally because a task node can send up to its usage across multiple outgoing edges to different ESs if that increases total reward within capacities.

Input: A flow network G in Fig. 3;
Output: $\forall i, m, X_{i,m}$ for $\tau_{i,m}$;
Residual network G^{R} of G;
Temporary variables for all the edges, (a, b) in network G^{R}: $f(a, b)$, $f(b, a)$, $C(a, b)$ and $C(b, a)$;
Temporary variable: $I_i^{\mathrm{sub}} \leftarrow 1$, $(i = 1, ..., N^{\mathrm{task}})$;
Construct a residual graph G^{R} based on G;
for *all edges* (a, b) *in* G^R **do**
  $C(b, a) \leftarrow -C(u, z)$;
  $f(a, b) \leftarrow 0$;
  $f(b, a) \leftarrow 0$;
end
while *TRUE* **do**
  Run the SPFA algorithm to find an augmenting path p from s to t using the minimum cost from G^{R};
  if *there is no augmenting path* p *from* G^{R} **then**
    break;
  end
  Augment all the flows along p to network G^{R};
  for *all edges* $(a, b) \in p$ **do**
    Update the flow, $f(a, b)$ based on flow augmentation;
    Update the flow, $f(b, a)$ based on flow augmentation;
  end
end
for $j = 1$ to $j = N^{\mathrm{es}}$ **do**
  for *all edges from task to ES vertices,* (v_i, w_j) *in* G^R **do**
    while $f(v_i, w_j) > 0$ **do**
      $X_{i,I_i^{\mathrm{sub}}} \leftarrow j$;
      $I_i^{\mathrm{sub}} \leftarrow I_i^{\mathrm{sub}} + 1$;
      $f(v_i, w_j) \leftarrow f(v_i, w_j) - 1$;
    end
  end
end

What is this MCMF graph ? How is it built ?

1) The source vertex s is connected to all the task vertexes v1,..., vN task . Each edge (s, vi) has a capacity Ui and a cost of 0.

2) Each vertex vi, (i = 1,...,N task) is connected to the vertices wj for which Hi,j = 1. Each edge (vi,wj) has a capacity Ui and a cost, −Runit i,j .

3) Each vertex wj, (j = 1,...,N ES) is connected to the sink vertex t. Each edge (wj, t) has a capacity Yj and a cost of 0.

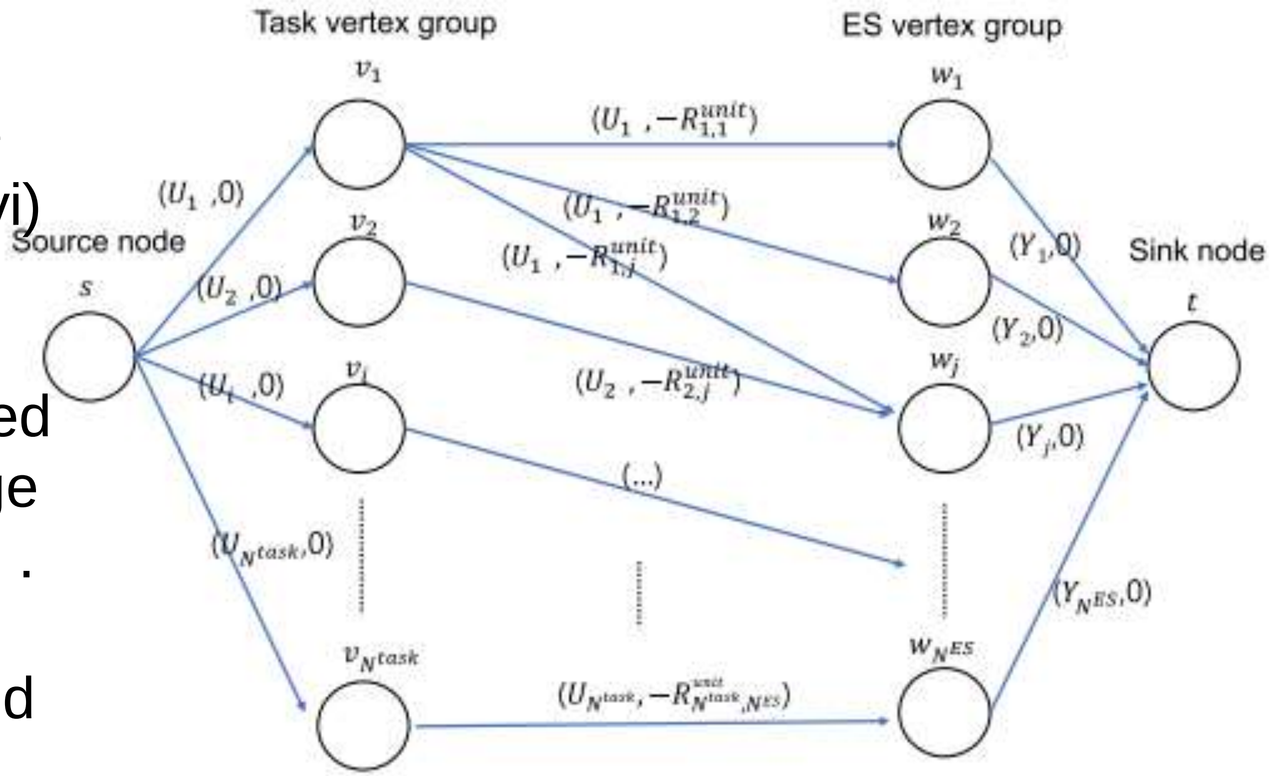


Fig. 3. Example of an MCMF graph representing an EAP.

EDGE ALLOCATION ALGORITHM(TS)

```
class EAA_TS:
    def __init__(self, tasks, edge_servers, Y):
        self.tasks = tasks
        self.edge_servers = edge_servers
        self.Y = Y
```

•Loads **tasks**, **edge servers**, and their **capacity limits (Y)**.

•Prepares for allocation with **task splitting allowed**.

```
def power_function(self, es, utilization):
    coeffs = es.power_active_coeffs
    utils = sorted(coeffs.keys())
    if utilization <= utils[0]:
        alpha = coeffs[utils[0]]
    elif utilization >= utils[-1]:
        alpha = coeffs[utils[-1]]
    else:
        for i in range(len(utils)-1):
            if utils[i] <= utilization <= utils[i+1]:
                u1, u2 = utils[i], utils[i+1]
                a1, a2 = coeffs[u1], coeffs[u2]
                alpha = a1 + (a2 - a1) * (utilization - u1) / (u2 - u1)
                break
    return alpha * utilization + es.power_idle * (1 - utilization)
```

•Estimates **server power consumption** at a given utilization.

•Uses interpolation between known power values.

•Ensures allocation respects **energy constraints**.

EDGE ALLOCATION ALGO (TS)

This Builds a Min-Cost Max-Flow graph

- **Nodes:**
  - *Source* → all tasks start here.
  - *Tasks* → represent workloads to be assigned.
  - *Edge Servers* → where tasks can be placed.
  - *Sink* → collects all assigned workloads.
- **Edges:**
  - Show **which task can go to which server**.
  - **Capacities** = how much work can flow (task usage & server limits Y).
  - **Weights = – reward per unit** (so the solver tries to maximize reward by minimizing cost).

What this function Does ??

- The MCMF algorithm sends workload “flow” through the graph.
- It determines how much of each task is assigned to each server.
- Tasks may be split across servers if required.
- The final allocation maximizes total reward while respecting server capacities and the overall power budget.


```python
def build_mcmf_graph(self):
    G = nx.DiGraph()
    total_task_usage = sum(task.usage for task in self.tasks)
    G.add_node('source', demand=-total_task_usage)
    G.add_node('sink', demand=total_task_usage)
    for task in self.tasks:
        task_node = f'task_{task.id}'
        G.add_node(task_node, demand=0)
        G.add_edge('source', task_node, capacity=task.usage, weight=0)
    for es in self.edge_servers:
        es_node = f'es_{es.id}'
        G.add_node(es_node, demand=0)
        G.add_edge(es_node, 'sink',
                   capacity=self.Y.get(es.id, 0), weight=0)
    for task in self.tasks:
        task_node = f'task_{task.id}'
        for es in self.edge_servers:
            if task.coverage[es.id] == 1:
                es_node = f'es_{es.id}'
                reward_per_unit = task.rewards[es.id] / \
                    task.usage if task.usage > 0 else 0
                G.add_edge(task_node, es_node,
                           capacity=task.usage, weight=-reward_per_unit)
    return G
```

```python
def run(self):
    G = self.build_mcmf_graph()
    try:
        flow_dict = nx.min_cost_flow(G)
    except Exception as e:
        return {task.id: [] for task in self.tasks}
    allocation = {task.id: [] for task in self.tasks}
    for task in self.tasks:
        task_node = f'task_{task.id}'
        for es in self.edge_servers:
            es_node = f'es_{es.id}'
            flow = flow_dict.get(task_node, {}).get(es_node, 0)
            if flow > 0:
                allocation[task.id].append((es.id, flow))
    return allocation
```

•Runs **min-cost flow solver** from NetworkX.
•If successful → gets the **optimal allocation** of tasks across servers.
•Each task may be split across multiple servers if needed.
•If solver fails → returns empty allocations

What's NetworkX?
•A **Python library** for creating and analyzing graphs and networks.
•Let's you easily build nodes & edges, then run algorithms on them.
•Provides ready-to-use solvers like **Min-Cost Max-Flow**

EDGE ALLOCATION ALGO (TS)

EDGE ALLOCATION ALGORITHM(NTS)

What the code does (EAA-NTS):

Step 1: Initializes tasks, edge servers, and utilization caps (Y).

Step 2: Sorts tasks by their **highest possible reward** (greedy priority).

Step 3: For each task:

- Checks candidate edge servers where it can run.
- Prefers the server giving **highest reward** (reward-oriented).
- Allocates only if server has **enough residual capacity** (respecting utilization cap Y).

Step 4: If no server can host it, the task is assigned to 0 (meaning dropped/not allocated).

Step 5: Returns the final **task-to-server allocation**.

```
class EAA_NTS:
    def __init__(self, tasks, edge_servers, Y):
        self.tasks = tasks
        self.edge_servers = edge_servers
        self.Y = Y

    def run(self):
        allocation = {}
        residual_capacity = {es.id: self.Y.get(
            es.id, 0) for es in self.edge_servers}
        sorted_tasks = sorted(self.tasks, key=lambda t: max(
            t.rewards.values()) if t.rewards else 0, reverse=True)
        for task in sorted_tasks:
            allocated = False
            sorted_es = sorted(
                self.edge_servers, key=lambda es: task.rewards.get(es.id, 0), reverse=True)
            for es in sorted_es:
                if task.coverage[es.id] == 1 and residual_capacity[es.id] >= task.usage:
                    allocation[task.id] = es.id
                    residual_capacity[es.id] -= task.usage
                    allocated = True
                    break
            if not allocated:
                allocation[task.id] = 0
        return allocation
```

MAIN.PY

Function by function explanation:

- **__init__** – Initializes power budget, prepares lists for edge servers and tasks, and creates empty utilization limits (Y).
- **setup_environment** – Builds the network:
- Creates random base stations with coordinates and coverage.
- Creates edge servers with random capacity & power models.
- Links each server to a base station.
- **power_function** – Estimates energy use of a server at different utilizations using interpolation of power coefficients.
- **generate_tasks** – Creates tasks with random demand (usage), assigns rewards per server, and sets coverage to all servers.
- **run_simulation** – Core execution:
- Runs **MUD Algorithm** → calculates safe utilization caps (Y).
- Runs **EAA-TS** → allocates tasks using threshold strategy.
- Runs **EAA-NTS** → allocates tasks without thresholds.
- Prints results of all strategies.
- **Main Block** – Starts simulation: sets environment, generates tasks, then runs all algorithms.

FUZZY ALGORITHM IN PHASE 1

```python
import numpy as np
#Triangular Membership
def triangular(x, a, b, c):
    return max(min((x - a) / (b - a + 1e-9), (c - x) / (c - b + 1e-9)), 0)
def reward_low(x): return triangular(x, 0, 0, 40)
def reward_med(x): return triangular(x, 30, 60, 90)
def reward_high(x): return triangular(x, 70, 100, 120)
def power_low(x): return triangular(x, 0, 0, 15)
def power_med(x): return triangular(x, 10, 25, 40)
def power_high(x): return triangular(x, 30, 50, 50)
def util_low(x): return triangular(x, 0, 0, 30)
def util_med(x): return triangular(x, 20, 50, 80)
def util_high(x): return triangular(x, 60, 100, 100)
priority_values = {'reject': 20, 'moderate': 50, 'strong': 90}

#Fuzzy Priority
def fuzzy_priority(reward, power, util):
    rL, rM, rH = reward_low(reward), reward_med(reward), reward_high(reward)
    pL, pM, pH = power_low(power), power_med(power), power_high(power)
    uL, uM, uH = util_low(util), util_med(util), util_high(util)
    rules = []
    rules.append(min(rH, pL) * priority_values['strong'])
    rules.append(min(rH, pH) * priority_values['moderate'])
    rules.append(min(rL, pH) * priority_values['reject'])
    rules.append(uH * priority_values['reject'])
    rules.append(min(rM, pM) * priority_values['moderate'])
    rules.append(min(rL, pL) * priority_values['moderate'])
    weights = [min(rH, pL), min(rH, pH), min(rL, pH), uH, min(rM, pM), min(rL, pL)]
    return 0 if sum(weights) == 0 else sum(rules) / sum(weights)
```

Membership Function Type:

- Uses the triangular membership function, which defines how much each input (reward, power, utilization) belongs to categories like "Low", "Medium", "High".
- Returns values between 0 and 1 for each fuzzy set, enabling continuous grading of inputs.

Fuzzification of Inputs:

- Inputs for each task and server (reward, power cost, utilization) are mapped into fuzzy categories using the membership functions.
- This transforms numeric values into linguistic levels (e.g., "reward is high", "power is low"), supporting soft decision-making.

```python
#Fuzzy Phase Algorithm
def fuzzy_phase(tasks, es_dict, power_budget):
    n_tasks = len(tasks)
    n_servers = len(es_dict)
    alloc_matrix = np.zeros((n_tasks, n_servers))
    unassigned_tasks = set(tasks)
    for es in es_dict.values():
        es.current_utilization = 0
    total_power = sum([es.power(0) for es in es_dict.values()])
    while unassigned_tasks and total_power <= power_budget:
        best, best_score = None, -float('inf')
        for task in unassigned_tasks:
            for es_name in task.candidates:
                es = es_dict[es_name]
                next_util = es.current_utilization + task.cpu
                if next_util > es.capacity:
                    continue

                p_before = es.power(es.current_utilization / es.capacity)
                p_after = es.power(next_util / es.capacity)
                delta_power = max(p_after - p_before, 1e-3)
                if total_power - p_before + p_after > power_budget:
                    continue

                score = fuzzy_priority(task.reward, delta_power, es.current_utilization)
                if score > best_score:
                    best = (task, es, es_name)
                    best_score = score
        if best is None:
            break
        task, es, es_name = best
        es.current_utilization += task.cpu
        alloc_matrix[task.id, es.id] = task.cpu
        total_power = sum([srv.power(srv.current_utilization / srv.capacity) for srv in es_dict.values()])
        unassigned_tasks.remove(task)
```

Priority Values for Decision-Making:

- Fuzzy rule outputs are mapped to crisp priority values ("strong", "moderate", "reject") for actionable scoring.
- Numerical priority scores enable aggregation, comparison, and selection among server-task options.

Defuzzification and Allocation:

- The algorithm computes a weighted average of all rule outputs to obtain a final score for each allocation candidate.
- The highest-scoring server-task assignment is chosen, subject to power and capacity constraints.

```python
# Outputs
Y = {es.id: es.current_utilization for es in es_dict.values()}
reward = sum(
    task.rewards[es.id] * alloc_matrix[task.id, es.id] / task.cpu
    for task in tasks for es in es_dict.values()
    if alloc_matrix[task.id, es.id] > 0
)
server_load = alloc_matrix.sum(axis=0)
frac_util = server_load / np.array([es.capacity for es in es_dict.values()])
power = np.array([
    es.power_idle + list(es.power_active_coeffs.values())[-1] * (u**2) * 100
    for es, u in zip(es_dict.values(), frac_util)
])
total_power = power.sum()
return Y, alloc_matrix, reward, server_load, total_power
```

The fuzzy output provides a weighted and adaptive decision by translating linguistic rules and uncertain input data into actionable numerical scores through defuzzification. This enables the allocation process to balance reward, power cost, and server utilization more flexibly than traditional threshold-based algorithms. As a result, resource assignment is optimized under constraints, leading to improved efficiency and utilization in complex systems

Priority Rules

- If Reward is High AND Power is Low → Priority = Strong
- If Reward is High AND Power is High → Priority = Moderate
- If Reward is Low AND Power is High → Priority = Reject
- If Utilization is High → Priority = Reject
- If Reward is Medium AND Power is Medium → Priority = Moderate
- If Reward is Low AND Power is Low → Priority = Moderate

```python
import numpy as np
# Projection
def project_row_to_simplex_leq1(row):
    r = np.clip(row, 0, 1)
    s = r.sum()
    if s <= 1: return r
    u = np.sort(r)[::-1]
    cssv = np.cumsum(u)
    rho = np.nonzero(u * np.arange(1, len(u)+1) > (cssv - 1))[0][-1]
    theta = (cssv[rho] - 1) / (rho + 1.0)
    return np.maximum(r - theta, 0)
def project_matrix(X):
    return np.array([project_row_to_simplex_leq1(row) for row in X])
# Fitness
def fitness(X, U, R, C, idle, slope, Plimit, penalty_w=1e6):
    reward = np.sum(R * (U.reshape(-1,1) * X))
    server_load = (U.reshape(-1,1) * X).sum(axis=0)
    frac_util = server_load / C
    power = idle + slope * (frac_util**2) * 100
    total_power = np.sum(power)
    tasks_per_server = (X > 1e-6).sum(axis=0)
    latency_penalty = 10.0 * np.sum(tasks_per_server)
    penalty = penalty_w * np.sum(np.maximum(server_load - C, 0))
    penalty += penalty_w * max(total_power - Plimit, 0)
    penalty += penalty_w * np.sum(np.maximum(X.sum(axis=1) - 1, 0))
    return -(reward - latency_penalty) + penalty
```

CI IN PHASE 2

Particle Swarm Optimization (PSO) for EAA-TS

- **Nature-inspired metaheuristic** based on bird flocking & fish schooling.
- Works with a **swarm of candidate solutions (particles)**.

Workflow:

Initialize Swarm

- Random allocations of tasks to servers.
- Each allocation projected to satisfy ∑alloc ≤ 1 (per task).

Evaluate Fitness

- Reward = Σ (R[i,j] × U[i] × X[i,j])
- Apply penalties for:
    - Capacity violations
    - Power budget breach
    - Multiple-server over-assignment

```python
# PSO Allocation
def pso_allocate(U, R, C, idle, slope, Plimit,
                 swarm_size=30, iters=200,
                 w=0.72, c1=1.4, c2=1.4, seed=42):
    rng = np.random.default_rng(seed)
    n_tasks, n_servers = R.shape
    # Initialize swarm
    X = rng.random((swarm_size, n_tasks, n_servers))
    for k in range(swarm_size):
        X[k] = project_matrix(X[k])
    V = rng.normal(0, 0.1, size=(swarm_size, n_tasks, n_servers))
    # Personal/global bests
    pbest = X.copy()
    pbest_val = np.array([fitness(X[k], U, R, C, idle, slope, Plimit) for k in range(swarm_size)])
    g_idx = np.argmin(pbest_val)
    gbest = pbest[g_idx].copy()
    gbest_val = pbest_val[g_idx]
    # Main loop
    for t in range(iters):
        r1 = rng.random((swarm_size, n_tasks, n_servers))
        r2 = rng.random((swarm_size, n_tasks, n_servers))
        V = w*V + c1*r1*(pbest - X) + c2*r2*(gbest - X)
        X = np.clip(X + V, 0, 1)
        for k in range(swarm_size):
            X[k] = project_matrix(X[k])
        vals = np.array([fitness(X[k], U, R, C, idle, slope, Plimit) for k in range(swarm_size)])
        improved = vals < pbest_val
        pbest[improved] = X[improved]
        pbest_val[improved] = vals[improved]
        if pbest_val.min() < gbest_val:
            g_idx = np.argmin(pbest_val)
            gbest = pbest[g_idx].copy()
            gbest_val = pbest_val[g_idx]
```

```python
# Metrics
reward = np.sum(R * (U.reshape(-1,1) * gbest))
server_load = (U.reshape(-1,1) * gbest).sum(axis=0)
frac_util = server_load / C
power = idle + slope * (frac_util**2) * 100
total_power = np.sum(power)
return gbest, reward, server_load, total_power
```

Update Bests

- Each particle remembers **personal best (pbest)**.
- Swarm tracks **global best (gbest)**.

Velocity & Position Update

Iterate until convergence.

Ant Colony Optimization (ACO) for EAA-NTS

- **Nature-inspired metaheuristic** based on ants finding shortest paths via pheromone trails.
- Uses a population of **artificial ants** to explore allocation possibilities.
- Each ant builds a solution by **assigning tasks to servers sequentially**.

Workflow

Initialize Pheromone Trails

- Equal pheromone on all task–server edges.
- Heuristic = reward / server capacity.

Construct Solutions

- Each ant assigns tasks to servers based on:
- Infeasible servers masked out.

Evaluate Fitness

- Reward – penalties (capacity violation, power violation, latency).

Update Pheromone

- Evaporation (ρ).
- Reinforcement from elite/best ants.

Iterate until convergence.

```python
import numpy as np
# Fitness
def fitness(X, U, R, C, idle, slope, Plimit, penalty_w=1e6):
    reward = np.sum(R * (U.reshape(-1,1) * X))
    server_load = (U.reshape(-1,1) * X).sum(axis=0)
    frac_util = server_load / C
    power = idle + slope * (frac_util**2) * 100
    total_power = np.sum(power)
    tasks_per_server = (X > 1e-6).sum(axis=0)
    latency_penalty = 10.0 * np.sum(tasks_per_server)
    penalty = penalty_w * np.sum(np.maximum(server_load - C, 0))
    penalty += penalty_w * max(total_power - Plimit, 0)
    penalty += penalty_w * np.sum(np.maximum(X.sum(axis=1) - 1, 0))
    return -(reward - latency_penalty) + penalty
```

```python
# ACO Allocation
def aco_allocate(U, R, C, idle, slope, Plimit,
                 n_ants=30, iters=100, alpha=1.0, beta=2.0,
                 rho=0.2, Q=10, seed=42):
    rng = np.random.default_rng(seed)
    n_tasks, n_servers = R.shape
    pheromone = np.ones((n_tasks, n_servers))
    heuristic = R / (np.maximum(C[np.newaxis, :], U[:, np.newaxis]))
    best_X = None
    best_fit = np.inf
    for epoch in range(iters):
        solutions, scores = [], []
        for ant in range(n_ants):
            X = np.zeros((n_tasks, n_servers))
            server_remaining = C.copy()

            for i in range(n_tasks):
                prob = (pheromone[i] ** alpha) * (heuristic[i] ** beta)
                mask = (server_remaining - U[i]) >= 0
                if not np.any(mask): mask[:] = True
                prob = prob * mask
                if prob.sum() == 0: prob = mask.astype(float)
                prob = prob / prob.sum()

                s = rng.choice(n_servers, p=prob)
                X[i, s] = 1.0
                server_remaining[s] -= U[i]

            fit = fitness(X, U, R, C, idle, slope, Plimit)
            solutions.append(X); scores.append(fit)
            if fit < best_fit:
                best_fit = fit; best_X = X.copy()
        pheromone *= (1 - rho)
        elite_idx = np.argsort(scores)[:max(1, n_ants // 5)]
        for idx in elite_idx:
            sc, X_elite = scores[idx], solutions[idx]
            for i in range(n_tasks):
                j = np.argmax(X_elite[i])
                pheromone[i, j] += Q / (1.0 + max(0, sc))
```

```python
best_reward = np.sum(R * (U.reshape(-1,1) * best_X))
server_load = (U.reshape(-1,1) * best_X).sum(axis=0)
frac_util = server_load / C
power = idle + slope * (frac_util**2) * 100
total_power = power.sum()
return best_X, best_reward, server_load, total_power
```

WHY COMPUTATIONAL INTELLIGENCE (CI) APPROACH IS GOOD

- **Global Search** – CI (PSO/ACO) explores entire solution space, not stuck in local optima.
- **Flexibility** – Can handle both splitting (PSO) and non-splitting (ACO) allocation naturally.
- **Multi-factor Optimization** – Considers reward, power, and utilization simultaneously.
- **Scalability** – Works well as number of tasks/servers grows (large combinatorial spaces).
- **Adaptability** – Can adjust to dynamic workloads or changing power budgets.
- **Closer to real-world decision-making** – mimics adaptive, heuristic human reasoning (e.g., fuzzy logic).

ADVANTAGES OF CI IN TASK OFFLOADING

- **Better Performance under Constraints**
  – Especially effective at low power budgets (tight Plimit).
- **Supports Complex Scenarios**
  – Nonlinear power models, heterogeneous rewards, large tasks.
- **Improved Resource Utilization**
  – PSO enables fractional allocation (splitting), maximizing server usage.
- **Resilience**
  – Handles uncertainty/noise in rewards or workloads.
- **Extensibility**
  – Can easily include extra objectives (latency, fairness, cost).

DATASET LINKS:

Server specs dataset : https://www.spec.org/power_ssj2008/results/res2025q1/

Tasks dataset : https://www.kaggle.com/datasets/ziya07/iiot-edge-computing

Contribution:
Individual contribution-
Pratyush(CB.SC.U4CSE23641) - MUD algorithm
Paarthu(CB.SC.U4CSE23639) - Function integrations and environment setup
Ravindran (CB.SC.U4CSE23647)- EAA_NTS algorithm
Adarssh (CB.SC.U4CSE23658) - EAA_TS algorithm , data parsing and formatting
Sanjesh(CB.SC.U4CSE23242) – Fuzzy, PSO, data parsing and formatting
Deepak (CB.SC.U4CSE23267) – ACO and Environment setup

THANK YOU