Task Scheduling in Edge Servers with Limited Solar Energy and Unlimited Battery

Mukka Koushik | Munagala Devi Naga Sai Srinivas | Pinneti Nitish Kumar

April 19, 2024

Problem Statement

In this problem, we are presented with a scenario involving multiple edge servers, each equipped with a solar panel of capacity S and an infinite-capacity battery. The day is divided into T time slots. For each time slot j and each edge server i, we are provided with the power generation $S[i][j]$ and the number of tasks arrived $D[i][j]$, where $S[i][j] \leq S$. These values are known at time 0.

Tasks must be executed within the same time slot they arrive. Upon execution, each task consumes a unit of utilization. The power consumption at an edge server in a given time slot is the cube of the number of tasks executed in that time slot at that server. Importantly, each edge server has infinite compute capacity.

Within a time slot, we have the flexibility to either store some of the power generated by the solar panel at the edge server, utilize some already stored power to execute additional tasks, or refrain from any action with the battery. However, if power has been stored in the battery in previous time slots at the edge server, it can only be used up to the amount stored in the current slot. Task migration between edge servers is possible at no cost. However, battery power cannot be transferred from one edge server to another.

The objective of the problem is to maximize the total number of task executions throughout the day.

1 Introduction

The task scheduling problem in edge servers with limited solar energy and unlimited battery capacity involves optimizing the execution of tasks across multiple edge servers over a given time period. In this report, we discuss various approaches we have tries to maximize the total number of task executions throughout the day.

2 Terminology

1. T**:** Number of time slots.
2. M**:** Number of servers.
3. $S_{\mathbf{max}}$**:** Maximum solar power that can be generated.
4. $S[i][j]$**:** Power generated by the ith server in the jth time slot.
5. $D[i][j]$**:** Tasks arrived at the ith server in the jth time slot.
6. $Task[j]$**:** Number of tasks arrived in the jth time slot. (Since all tasks in the same time slot can be migrated between the servers)

$$Task[j] = \sum_{i=1}^{M} D[i][j]$$

3 Different Approaches tried

3.1 Brute Force Approach

In the brute force approach, the problem is decomposed into states, each dependent on the server number, time slot number, energy levels of all servers (including their batteries), and the remaining number of tasks available for execution in each time slot.

The approach begins by starting at the first server in the first time slot and choosing to execute $0, 1, 2, \ldots$ tasks on it. The energy is then updated, and the next server is recursively called, performing the same function. If the last server is reached, the approach calls the first server of the next time step. This process continues until all possible combinations of task executions are explored.

Throughout this exploration, the number of tasks executed is stored, along with tasks executed by servers after it and all servers in latter time steps. Finally, the maximum number of tasks executed is chosen among all possibilities.

However, it's important to note that this approach may lead to exponential time complexity, especially if the subproblems do not repeat, which has been observed in some situations.

Time Complexity: $O((T \times S_{\max} + M)^T)$

3.2 Dynamic Programming Approach

In the dynamic programming approach, we first calculate the total energy generated in each server across all time slots and sort the servers based on this total generated energy. This sorting ensures that tasks are executed on machines with lower energy levels, as it is more beneficial to execute tasks on servers with less energy.

In each time step, we store all the tasks arrived in a single array, as these tasks can be distributed across different servers in each time step. Then, for each server in the sorted order, we distribute the tasks across all time steps according to the available number of tasks and energy in each time step. Since energy utilization/availability only depends on whether the server has executed tasks in previous time steps, consideration of other servers is not relevant. Dynamic programming is used for this task distribution to maximize the number of tasks executed in each server across all time steps.

Each state in the dynamic programming process depends on the time step and the energy available, which is bounded by $T \times S$. After completing dynamic programming for a single server, tasks in each time step can be updated, and the same process can be continued for other servers.

However, it's important to note that this approach does not consider the relation between servers in a time slot. Observations from test cases show that it may not always yield optimal results compared to the brute force approach.

Time Complexity: $O(T^2 \times M \times (T \times S_{\max})^{\frac{1}{3}})$

3.3 Greedy Approach: Sorting Servers Based on Energy in Each Time Slot

In this approach, in each time step, we greedily sort the servers based on their total available energy (i.e., energy generated and already available energy from the battery). The rationale behind this approach is to execute tasks with as little energy as possible to maximize the number of tasks executed. For example, if executing 2 tasks consumes 8 units of energy, while executing 3 tasks consumes 27 units of energy, it is beneficial to execute tasks with less energy.

For instance, consider 3 servers with energy levels of 8, 8, and 27, respectively, and a number of available tasks of 2. Instead of executing the tasks on the server with energy 27, we can execute 2 tasks on the servers with energy 8 each. This leaves more energy available in the next time step, thus maximizing the number of tasks that can be executed.

However, it's important to note that this heuristic may fail in certain cases where the energy production in servers follows a different order. For example, consider 2 servers with 3 time slots and energy levels of (8, 27), (0, 8), and (0, 0), respectively, and a number of tasks of 4, 3, and 1. In such scenarios, the above heuristic may not yield the maximum possible number of task executions.

3.4 Greedy Approach: Reverse Sorting Servers Based on Energy in Each Time Slot

In this approach, similar to the previous one, we sort the servers based on their available energies but in decreasing order. The rationale behind this approach is to allocate available tasks in such a manner that most tasks are executed in the initial stages.

However, this heuristic also fails in scenarios where saving energy instead of executing tasks might lead to more task executions in future time steps. For instance, consider 2 servers with 2 time slots and energy levels of (27, 8) and (0, 0), respectively, and a number of tasks of 5 and 2 in each time step. If we save energy in the first time step by executing only 4 tasks, we can execute 2 more tasks in the second time step. Otherwise, we can only execute 5 tasks.

4 Final Approach Reached

4.1 Initialization

1. Initialize vectors to store solar power generation $S[i][j]$ and task arrival $D[i][j]$ for all servers and time slots.
2. Calculate the total number of tasks arriving in each time slot j by summing $D[i][j]$ for all servers i.

4.2 Algorithmic approach

1. Implement a greedy algorithm to assign tasks to servers efficiently.
2. Maintain a cumulative solar power vector for each server and time slot to track available energy.
3. Iterate through each time slot:
   - For the first time slot, check if there are tasks arrived and there is enough energy to execute the task.
   - For subsequent time slots, consider the stored energy from previous slots and available solar power.
   - Execute tasks if there is enough energy and tasks are available.
   - Update cumulative solar power, task execution, and energy utilization accordingly.

4.3 Task Execution

1. Execute tasks in the same time slot, consuming a unit of utilization per task.
2. The power consumed in a time slot at an edge server is the cube of the number of tasks executed in that slot.
3. Utilize stored battery power if available, considering the capacity limit of each server.

4.4 Optimization

1. Continuously iterate through time slots until no more tasks can be executed, optimizing task assignment and energy utilization.
2. Track the total number of tasks executed throughout the day.

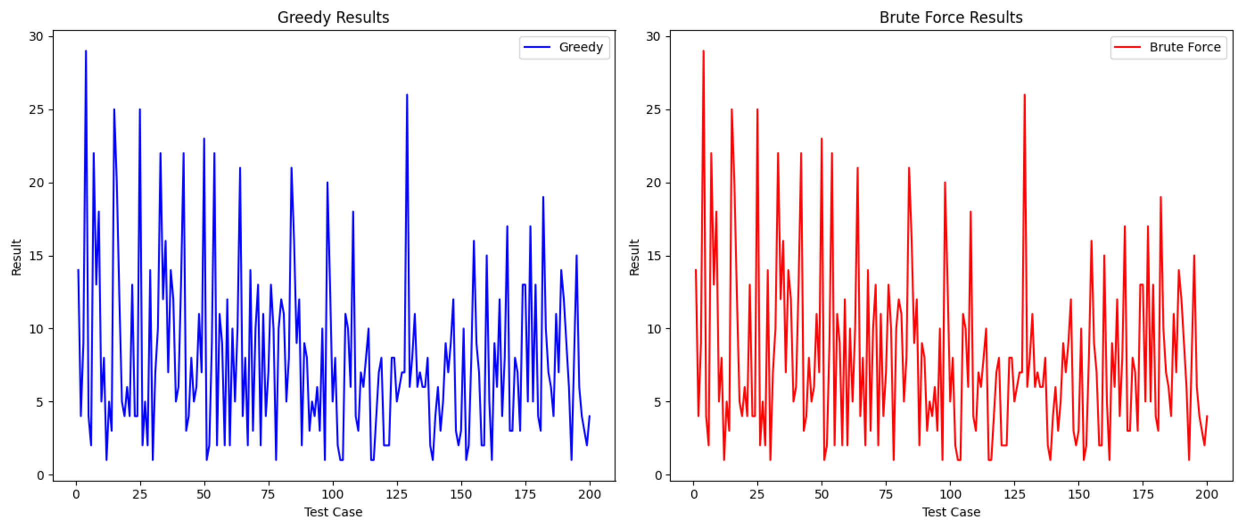

Figure 1: Comparision dot plot between Brute Force and Greedy Approaches

4.5 Greedy Task Assignment Algorithm

Algorithm 1: Greedy Task Assignment Algorithm

```
Input: Number of edge servers $M$, number of time slots $T$, solar power generation matrix $S$, task
       arrival matrix $D$
Output: Total number of tasks executed final_tasks
Function greedy($task\_arrived, S$):
    Initialize cumulative_S[$M$][$T$] and previousStages[$M$] with zeros;
    Initialize tasksExecuted[$M$][$T$] with zeros;
    isExecutionDone $\leftarrow$ true;
    while isExecutionDone do
        isExecutionDone $\leftarrow$ false;
        for $j \leftarrow 0$ to $T-1$ do
            for $i \leftarrow 0$ to $M-1$ do
                if $j = 0$ then
                    ExecuteTask($i, j$);
                end
                else
                    ExecuteTask($i, j$, cumulative_S[$i$][$j-1$]);
                    $p \leftarrow j$;
                    leftAfterUtilisation $\leftarrow$ cumulative_S[$i$][$j$] $-$ currentStageUtilisation;
                    while $p \geq 0$ and cumulative_S[$i$][$p$] $\geq$ leftAfterUtilisation do
                        cumulative_S[$i$][$p$] $\leftarrow$ leftAfterUtilisation;
                        $p \leftarrow p - 1$;
                    end
                end
            end
        end
    end
    Calculate total number of tasks executed final_tasks;
    return final_tasks;
```

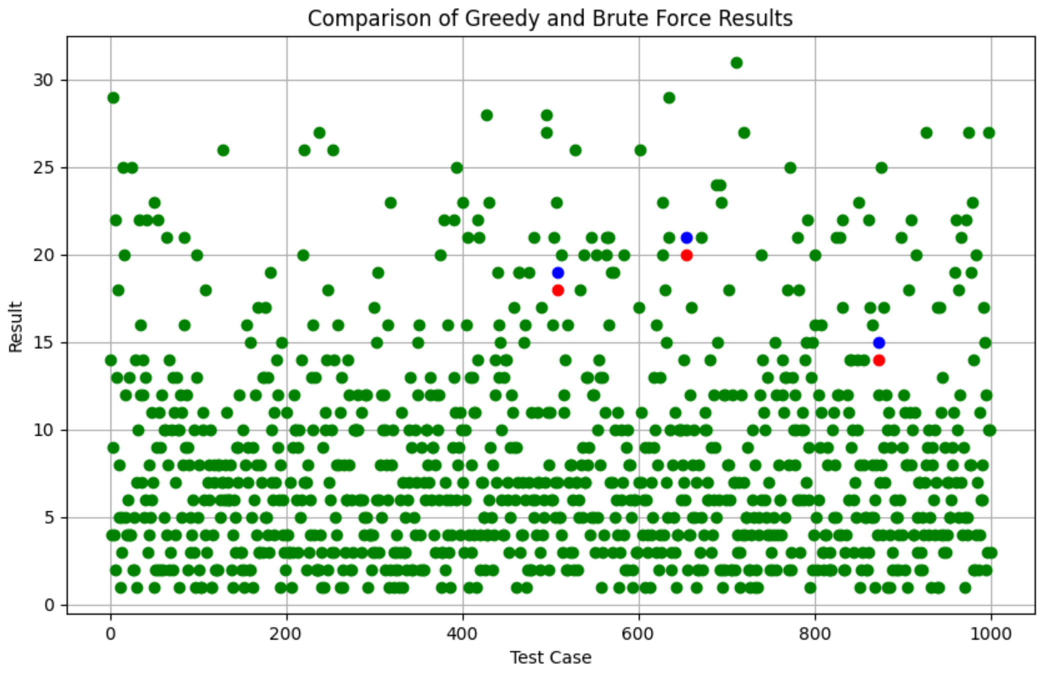

Figure 2: Comparision dot plot between Brute Force and Greedy Approaches

4.6 Time Complexity Analysis

To analyze the time complexity of the provided greedy task assignment algorithm, we can break down the major operations as follows:

1. **Main While Loop**: The Main while loop iterated till no energy left to do any task or no tasks remaining. resulting in a time complexity of $O((T \times S_{max})^{1/3})$
2. **Timeslots Loop**: This loop iterates through all time slots T and all edge servers M, resulting in a time complexity of $O(T \times M)$.
3. **Nested servers Loop**: Within each iteration of the main loop, there are nested loops that also iterate through all time slots, edge servers, and also updating the cumulative energy, resulting in an additional time complexity of $O(T \times M \times T)$.
4. **Final Calculation**: After the main loop, there is a final calculation to determine the total number of tasks executed, which also has a time complexity of $O(T \times M)$.

Combining these steps, the overall time complexity of the algorithm is dominated by the nested loops, resulting in,

$$\textbf{Time Complexity: } O((T^2 \times M)(T \times S_{max})^{1/3})$$

.

5 Conclusion

By efficiently managing power utilization and task execution across edge servers using the greedy algorithm, we maximized the total number of task executions for the entire day, considering the constraints and objectives outlined.In each iteration, we only execute same number of tasks across all time steps, so, it doesnot matter much that about the exact time step, we chose to execute the tasks.For example, in first iteration only 1 task should be executed, as it takes same amount of power, it doesnot much matter, the time step we execute in(as the number of tasks executed doesnot change wherever we execute it).This was the major criterion we used to develop the algorithm.