EAST WEST UNIVERSITY

Project Report

Efficient Task Scheduler With Algorithms

Submission Date: 20/05/2025

Group no: 08

Course Code: CSE246

Section - 04

Submitted To:

Dr. Tania Sultana
Assistant Professor
Department of Computer Science and Engineering

Group Members:

Name	ID	Contributions
Sihab Bin Sarwar	2023-1-60-043	25%
Ifteayj Ahmed	2022-1-60-002	25%
Shahriar Rahman	2022-3-60-036	25%
Abu Baker Siddik	2022-1-60-044	25%

Introduction

The Task Scheduling application is designed to assist users in optimizing task execution based on time constraints, priorities, and dependencies. It offers multiple algorithmic approaches—including 0/1 Knapsack (Dynamic Programming), Greedy Heuristics, and Topological Sorting—to help users select, prioritize, and order tasks efficiently. Suitable for academic, professional, and operational environments, the system empowers users to make informed decisions on resource allocation and task management. Emphasizing clarity, efficiency, and algorithmic rigor, this project addresses the growing need for intelligent task organization in complex scheduling scenarios.

Objectives

This project is built to enhance productivity of the user so that they can manage their task efficiently. This objectives in mind while building it are:

- To help users schedule tasks efficiently within a fixed time limit.
- To prioritize tasks based on their importance and duration.
- To handle task dependencies using topological sorting.
- To compare different scheduling algorithms for better decision-making.

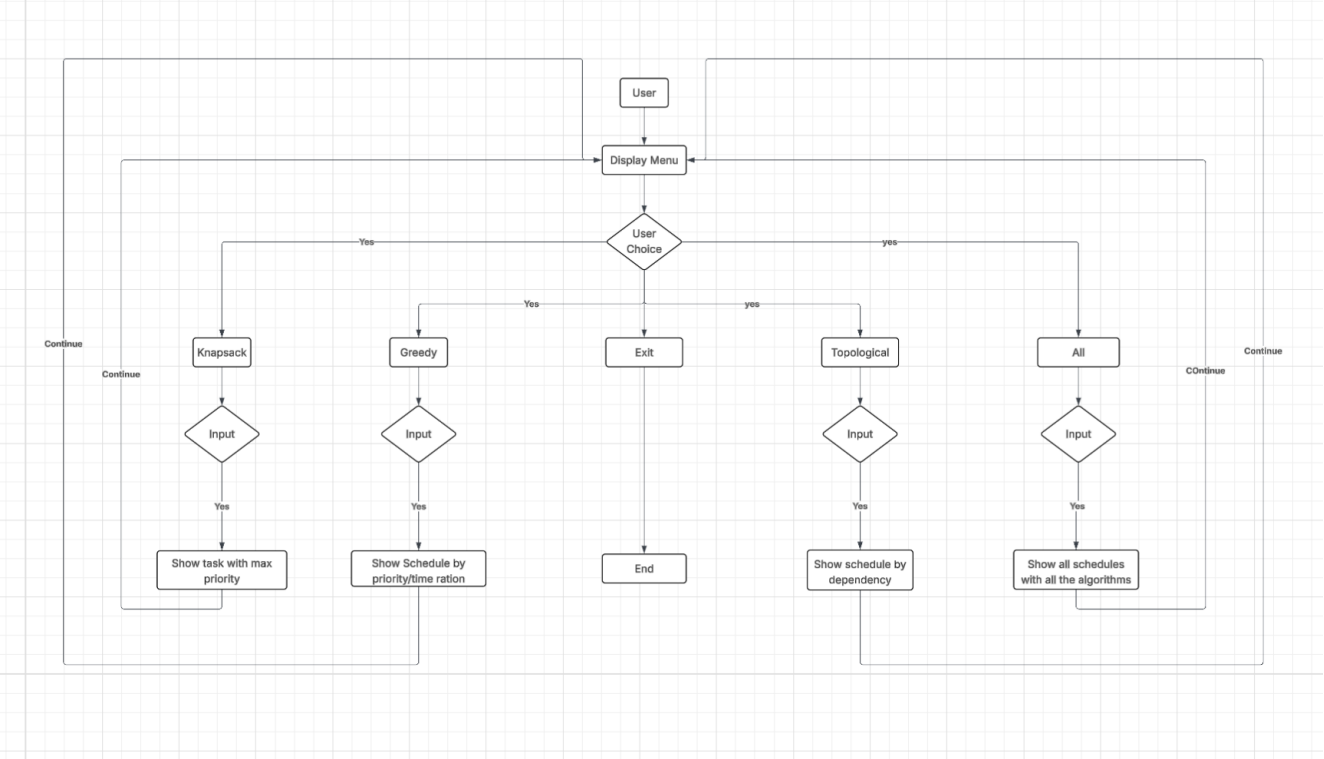
Technology Specification

- **Programming Language:** C++
- **Executing Environment:** GCC compiler, Terminal of OS
- **Key Libraries:**
    - #include <iostream>: For input & output operations.
    - #include <vector>: For dynamic arrays
    - #include <iomanip>: For formatted output (setw, setprecision)
    - #include <algorithm>: For sorting and max functions
    - #include <unordered_map>: For fast key-value lookup
    - #include <queue>: For implementing BFS or task queues
- **Key Algortithms:** Topological sort for dependency, Greedy for highest (priority/duration) & 0/1 Knapsack (DP) to maximize total priority within a time limit.

Data Flow Diagram

The flowchart shows the systemic process of - Efficient Task Scheduler program step – by – step. It opens up by showing the dashboard where user can select the algorithm for task scheduling or can select all the algorithm to compare all the schedule.

Efficient Task Scheduler



Features List:

- **Knapsack 0/1 (DP):** Allows user to make task schedule by maximizing the priority within the time limit.
- **Greedy:** User can make quick schedule by priority/time ratio.
- **Topological Sort:** If the tasks are dependent on each other user can choose this.
- **Run all:** User can compare the result of all the algorithms.

Implementation

0/1 Knapsack (DP):

```cpp
int knapsackSchedule(const vector<Task>& tasks, int maxTime, vector<Task>&
selectedTasks) {
    int n = tasks.size();
    vector<vector<int>> dp(n + 1, vector<int>(maxTime + 1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int t = 0; t <= maxTime; ++t) {
            if (tasks[i - 1].duration > t)
                dp[i][t] = dp[i - 1][t];
            else
                dp[i][t] = max(dp[i - 1][t], tasks[i - 1].priority + dp[i - 1]
                [t - tasks[i - 1].duration]);
        }
    }

    int t = maxTime;
    for (int i = n; i > 0 && t >= 0; --i) {
        if (dp[i][t] != dp[i - 1][t]) {
            selectedTasks.push_back(tasks[i - 1]);
            t -= tasks[i - 1].duration;
        }
    }

    return dp[n][maxTime];
}
```

Result Console:

```
================================================
0/1 Knapsack Result (Optimal)
================================================
Task ID    Duration     Priority  Dependencies
------------------------------------------------
1          2            3
3          4            3

Total Priority Achieved: 6
Time Used: 6 / 8 units
Efficiency (Priority per Time Unit): 1.00

Unselected Tasks (Missed Opportunities):
------------------------------------------------
Task ID: 2, Duration: 3, Priority: 2
Press any key to continue . . .
```

Greedy:

```cpp
int greedySchedule(vector<Task> tasks, int maxTime, vector<Task>&
selectedTasks) {
    sort(tasks.begin(), tasks.end(), [](const Task& a, const Task& b) {
        double r1 = (double)a.priority / a.duration;
        double r2 = (double)b.priority / b.duration;
        return r1 > r2;
    });

    int usedTime = 0, totalPriority = 0;
    for (const auto& task : tasks) {
        if (usedTime + task.duration <= maxTime) {
            selectedTasks.push_back(task);
            usedTime += task.duration;
            totalPriority += task.priority;
        }
    }
    return totalPriority;
}
```

Result Console:

```
================================================
Greedy Result (Heuristic)
================================================
Task ID   Duration    Priority  Dependencies
------------------------------------------------
1         2           3
2         3           4
Total Priority: 7

Press any key to continue . . .
```

Topological Sort:

```cpp
vector<int> topologicalSort(const vector<Task>& tasks) {
    unordered_map<int, vector<int>> adj;  // dependency -> dependent tasks
    unordered_map<int, int> inDegree;

    // Initialize inDegree for all tasks
    for (const auto& task : tasks) {
        inDegree[task.id] = 0;
    }

    // Build graph edges and compute in-degree
    for (const auto& task : tasks) {
        for (int dep : task.dependencies) {
            adj[dep].push_back(task.id);
            inDegree[task.id]++;
        }
    }

    queue<int> q;
    for (const auto& [taskId, deg] : inDegree) {
        if (deg == 0) {
            q.push(taskId);
        }
    }

    vector<int> order;
    while (!q.empty()) {
        int curr = q.front();
        q.pop();
        order.push_back(curr);

        for (int neighbor : adj[curr]) {
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }

    if ((int)order.size() != (int)tasks.size()) {
        cout << "\n!!! Cycle detected: No valid task order!!!\n";
        return {};
    }

    return order;
}
```

Result Console:

```
==================================================
Topological Sort (Dependency Order)
==================================================
Valid Execution Order: 3 1 2
Press any key to continue . . .
```

Result Console for Option 4:

```
================================================
Topological Sort (Dependency Order)
------------------------------------------------

!!! Cycle detected: No valid task order!!!
------------------------------------------------
0/1 Knapsack Result (Optimal)
------------------------------------------------
Task ID   Duration    Priority  Dependencies
------------------------------------------------
1         4           3         3
2         5           4         3 1
3         5           3         1
Total Priority: 10
Time Used: 14 / 20
Efficiency: 0.71
------------------------------------------------
Greedy Result (Heuristic)
------------------------------------------------
Task ID   Duration    Priority  Dependencies
------------------------------------------------
2         5           4         3 1
1         4           3         3
3         5           3         1
Total Priority: 10
Press any key to continue . . .
```

Note: As we know Topological doesn't work if there is a cycle. Here Task 1 depends on task 3 and task 3 depends on the task 1 thus creating a cycle and not giving any output.

Complexity:

- **Time Complexity:** O (n × T + n log n + e)
- **Space Complexity:** O (n × T + n + e)

Complexity Analysis

1. 0/1 Knapsack (Dynamic Programming)

- **Function:** knapsackSchedule(const vector<Task>&, int, vector<Task>&)
- **Time Complexity:**
  - O(n×T)
    - n = number of tasks
    - T = maxTime (total available time)
- **Space Complexity:**
  - O(n×T) for the DP table
  - O(n) additional for tracking the selected tasks

This approach guarantees an optimal solution but can become costly if both n and T are large.

2. Greedy Heuristic

- **Function:** greedySchedule(vector<Task>, int, vector<Task>&)
- **Time Complexity:**
  - O (n log n)
    - Sorting by priority/duration ratio dominates
- **Space Complexity:**
  - O(n) for the temporary task vector and the output list

Very efficient for large n, but does not always produce the optimal priority sum.

3. Topological Sort

- **Function:** topologicalSort(const vector<Task>&)
- **Time Complexity:**
  - O (n + e)
    - n = number of tasks
    - e = total number of dependency edges
- **Space Complexity:**
  - O(n+e) for the adjacency map and in-degree tracking
  - O(n) for the queue and result list

Ensures a valid execution order when tasks have dependencies, and detects cycles in O(n+e) time.

Results

The program successfully fulfills the following functionalities:

- Sorting or scheduling the task by priorities.
- If the tasks are dependent then schedule them considering dependencies.
- Compare all schedules given by those algorithms.
- Displaying the tasks in organized manner.

Limitations

- For large inputs memory usage becomes huge.
- Greedy algorithm doesn't provide optimal results.
- Topological Sort ignores the priorities.
- Task can't run concurrently or user can't add task while running the code.

Conclusion

The **Efficient Task Scheduler** efficiently manages tasks using algorithms like Knapsack DP , Greedy and Topological Sort. A file system can be introduced here to make a task planner for ahead. To make quick scheduling for emergencies or in a hurry this program works great. Though the efficiency depends on the system and the input size. This project highlights practical algorithm applications, enhancing productivity and time management.