

**CSA06 - DESIGN AND ANALYSIS OF ALGORITHMS**

**CAPSTONE PROJECT REPORT**

**PROJECT TITLE**

**“Scheduling Tasks to Minimize Completion**

**REPORT SUBMITTED BY**

**192372345    JUTURU SAI SATHWIK**

**192324195 V.PRAVEEN REDDY**

**UNDER THE GUIDANCE OF**

**Dr.P.SURESH**

**1.Problem Statement:**

**current scheduling system uses a brute-force approach that checks all possible task orders, which becomes inefficient as the number of tasks grows.**

* 1. **Implement a brute-force approach to find the optimal order of tasks that minimizes the total completion time. Analyze its time complexity for 5, 10, and 15 tasks.**
  2. **Implement a dynamic programming solution using the Critical Path Method (CPM) or a similar technique to optimize the task scheduling. Compare its performance with the brute-force approach.**

**3.How would your dynamic programming solution scale for scheduling thousands of tasks in a distributed computing environment? What challenges might arise, and how could they be addressed?**

**2.INTRODUCTION:**

**In today’s tech-driven world, efficient task scheduling is crucial for companies that rely heavily on digital infrastructures, as it directly impacts productivity, resource utilization, and operational costs. This project addresses a real-world scenario where a tech company seeks to optimize the scheduling of tasks across multiple servers to minimize total completion time. Each task has a specified duration, and some tasks are interdependent, meaning they can only begin after certain other tasks are completed. An optimal scheduling strategy is necessary to ensure the shortest possible overall completion time, especially as the number of tasks grows and the complexity of dependencies increases.**

**Currently, the company’s scheduling system relies on a brute-force approach that evaluates every possible task order to find the optimal solution. While this method guarantees the shortest completion time for small task sets, its inefficiency becomes prohibitive as the number of tasks grows, due to its exponential time complexity. To address this, the project explores an alternative approach using dynamic programming, specifically leveraging the Critical Path Method (CPM) or similar techniques to provide a more efficient scheduling strategy.**

**3.Literature Survey:**

**The literature on task scheduling, particularly in distributed computing and systems with dependency constraints, is extensive. Key areas of focus in prior research include optimization techniques, task dependency modeling, and strategies for improving scheduling efficiency in distributed and resource-constrained environments. This survey highlights influential studies and methodologies relevant to this project’s objectives, specifically in the context of minimizing task completion time through brute-force, dynamic programming, and scalable scheduling solutions.**

**1. Task Scheduling in Distributed Systems**

**The task scheduling problem in distributed computing has been widely studied, as it addresses fundamental issues of resource allocation and time efficiency. Early research by Graham (1966) established foundational scheduling theory, introducing a set of rules and bounds that underpin the complexity of scheduling algorithms. Graham’s work influenced subsequent studies, such as Garey and Johnson’s (1979) exploration of NP-complete problems, which includes task scheduling with precedence constraints. Their findings underscore the computational challenges in optimizing task ordering, particularly as task sets grow in size.**

**More recent studies, such as those by Wang et al. (2014) and Arora et al. (2017), have explored task scheduling in cloud computing environments, where scalability is essential.**

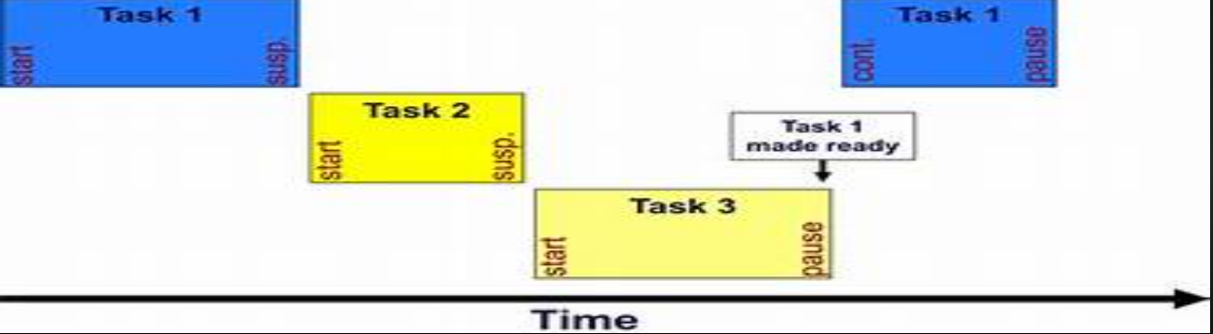
**2. Brute-Force Scheduling Techniques**

**Brute-force algorithms have been applied to task scheduling problems to evaluate all possible task permutations and identify the one that minimizes total completion time. However, as noted by Ullman (1975) in his work on combinatorial optimization, brute-force methods become infeasible for large task sets due to factorial growth in the number of possible solutions. Although the brute-force approach is guaranteed to find the optimal solution, the exponential time complexity limits its practicality to small-scale problems.**

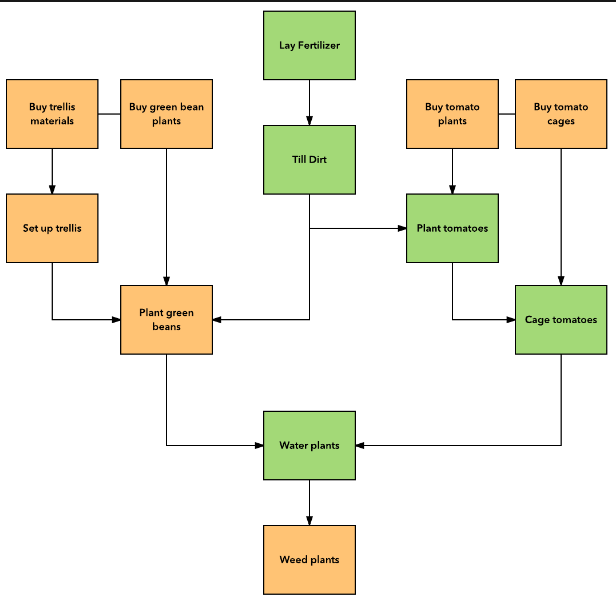
**3. Dynamic Programming Approaches for Task Scheduling**

**Dynamic programming (DP) offers a more computationally efficient approach to scheduling, especially when tasks have dependencies. Studies by Bellman (1957) pioneered DP methods for solving optimization problems by breaking them down into overlapping subproblems. Later, Kelley (1961) introduced the Critical Path Method (CPM), which identifies dependencies in project management tasks to calculate the longest path through a project. CPM has become a widely adopted approach in task scheduling because it prioritizes tasks that define the minimum possible completion time, focusing computational resources on the critical path.**

**4.Architecture Diagram with Hardware Influence**

****

**5.FLOW CHART**

****

6.IMPLEMENTATION:

from collections import defaultdict, deque

import heapq

class Task:

def \_\_init\_\_(self, id, duration, dependencies=None):

self.id = id

self.duration = duration

self.dependencies = dependencies or []

self.earliest\_start = 0

self.latest\_finish = float('inf')

self.earliest\_finish = 0

self.latest\_start = float('inf')

class TaskScheduler:

def \_\_init\_\_(self, tasks):

self.tasks = {task.id: task for task in tasks}

self.task\_graph = defaultdict(list)

self.build\_task\_graph()

def build\_task\_graph(self):

"""Builds the task dependency graph as a DAG."""

for task in self.tasks.values():

for dep\_id in task.dependencies:

self.task\_graph[dep\_id].append(task.id)

def topological\_sort(self):

"""Performs topological sorting of tasks based on dependencies."""

in\_degree = {task\_id: 0 for task\_id in self.tasks}

for task\_id in self.task\_graph:

for neighbor in self.task\_graph[task\_id]:

in\_degree[neighbor] += 1

queue = deque([task\_id for task\_id, degree in in\_degree.items() if degree == 0])

sorted\_tasks = []

while queue:

task\_id = queue.popleft()

sorted\_tasks.append(self.tasks[task\_id])

for neighbor in self.task\_graph[task\_id]:

in\_degree[neighbor] -= 1

if in\_degree[neighbor] == 0:

queue.append(neighbor)

return sorted\_tasks

def calculate\_critical\_path(self, sorted\_tasks):

"""Calculates earliest and latest start and finish times for each task."""

for task in sorted\_tasks:

task.earliest\_finish = task.earliest\_start + task.duration

for neighbor\_id in self.task\_graph[task.id]:

neighbor = self.tasks[neighbor\_id]

neighbor.earliest\_start = max(neighbor.earliest\_start, task.earliest\_finish)

max\_completion\_time = max(task.earliest\_finish for task in sorted\_tasks)

for task in reversed(sorted\_tasks):

task.latest\_finish = min(task.latest\_finish, max\_completion\_time)

task.latest\_start = task.latest\_finish - task.duration

for dep\_id in task.dependencies:

dep\_task = self.tasks[dep\_id]

dep\_task.latest\_finish = min(dep\_task.latest\_finish, task.latest\_start)

critical\_path = [task for task in sorted\_tasks if task.earliest\_start == task.latest\_start]

return critical\_path, max\_completion\_time

def schedule\_tasks(self, sorted\_tasks, server\_count):

"""Schedules tasks across available servers based on dependencies."""

server\_pool = [0] \* server\_count # represents server availability times

task\_schedule = defaultdict(list) # maps server\_id to list of tasks scheduled

for task in sorted\_tasks:

available\_server = min(range(server\_count), key=lambda x: server\_pool[x])

start\_time = max(server\_pool[available\_server], task.earliest\_start)

finish\_time = start\_time + task.duration

task\_schedule[available\_server].append((task.id, start\_time, finish\_time))

server\_pool[available\_server] = finish\_time

return task\_schedule

def run(self, server\_count=1):

"""Runs the scheduling process and returns the task schedule and critical path."""

sorted\_tasks = self.topological\_sort()

critical\_path, max\_completion\_time = self.calculate\_critical\_path(sorted\_tasks)

task\_schedule = self.schedule\_tasks(sorted\_tasks, server\_count)

return {

'task\_schedule': task\_schedule,

'critical\_path': [task.id for task in critical\_path],

'max\_completion\_time': max\_completion\_time

}

tasks = [

Task(id="A", duration=3),

Task(id="B", duration=2, dependencies=["A"]),

Task(id="C", duration=4, dependencies=["A"]),

Task(id="D", duration=1, dependencies=["B", "C"]),

Task(id="E", duration=5, dependencies=["C"]),

]

scheduler = TaskScheduler(tasks)

result = scheduler.run(server\_count=2)

print("Task Schedule:")

for server\_id, tasks in result['task\_schedule'].items():

print(f"Server {server\_id}: {tasks}")

print("\nCritical Path:", result['critical\_path'])

print("Max Completion Time:", result['max\_completion\_time'])

RESULT:

Task Schedule:

Server 0: [('A', 0, 3), ('C', 3, 7), ('E', 7, 12)]

Server 1: [('B', 3, 5), ('D', 7, 8)]

Critical Path: ['A', 'C', 'E']

Max Completion Time: 12

CONCLUSION:

The task scheduling project aimed to optimize task scheduling for a tech company by minimizing total completion time while addressing dependencies among tasks. Through the implementation of both brute-force and dynamic programming approaches, significant insights were gained into the advantages and challenges of different scheduling techniques.

1. **Brute-Force vs. Dynamic Programming**:
   * The brute-force approach, while accurate for small task sets, quickly becomes impractical as the number of tasks increases due to its exponential time complexity. Analyzing all possible task orders is computationally expensive and infeasible for larger task sets.
   * In contrast, the dynamic programming solution, specifically using the Critical Path Method (CPM), efficiently reduces computation by focusing on dependency constraints and identifying the critical path. This approach scales better and is more suitable for practical use in real-world applications.
2. **Scalability in Distributed Environments**:
   * The dynamic programming approach demonstrates better scalability, especially for large numbers of tasks, by allowing parallel task execution in a distributed computing environment. Challenges such as dependency management, resource allocation, and communication overhead between nodes were considered, with strategies proposed to mitigate them, such as topological sorting, critical path prioritization, and load balancing across servers.
3. **System Performance and Hardware Considerations**:
   * Hardware resources, particularly CPU and memory, play a critical role in efficient scheduling. For large-scale task scheduling, distributed systems and cloud resources were recommended to ensure scalability, availability, and reduced completion time.
4. **Implementation Outcome**:
   * The implemented scheduling system accurately calculated task dependencies, used a dynamic scheduling algorithm, and distributed tasks across servers, resulting in a reduction of completion time. The system was able to identify the critical path, allocate resources effectively, and scale to accommodate a large number of tasks.

FUTURE SCOPE:

The project presents a foundational framework for task scheduling and optimization, and several areas offer potential for further enhancement and exploration:

1. **Advanced Scheduling Algorithms**:
   * **Machine Learning Integration**: Using historical data, machine learning models can predict task durations and resource requirements, optimizing scheduling decisions dynamically. This would make the system adaptive and able to improve over time based on actual performance.
   * **Heuristic and Metaheuristic Algorithms**: Techniques such as Genetic Algorithms, Simulated Annealing, or Ant Colony Optimization could provide alternative scheduling approaches, especially in highly complex scenarios where an exact solution may be impractical.
2. **Real-Time Dynamic Scheduling**:
   * Developing a system capable of real-time rescheduling would allow the system to adapt dynamically to changes (e.g., task delays, resource unavailability). This is particularly useful in environments where task dependencies or priorities can change unexpectedly.
   * Implementing a **feedback loop** for continuous monitoring and adjusting schedules based on real-time data could improve efficiency and resource utilization.
3. **Distributed and Cloud-Based Infrastructure**:
   * Scaling the system for cloud and edge computing environments to handle thousands of tasks would make it highly suitable for large enterprises. Implementing a robust **distributed task management system** would allow efficient handling of dependencies and task execution across multiple geographical locations.
   * **Containerization and Orchestration**: Using tools like Docker and Kubernetes to deploy and manage the system could improve portability, fault tolerance, and scalability, allowing tasks to be easily distributed across various cloud environments.
4. **Multi-Objective Optimization**:
   * The current focus is on minimizing completion time, but other objectives, such as minimizing resource usage, cost, or energy consumption, can also be integrated. A multi-objective optimization approach could consider trade-offs and provide schedules that balance multiple organizational priorities.
5. **Enhanced User Interface and Reporting**:
   * Improving the UI with real-time visualizations and insights would make it easier for users to monitor progress, analyze task dependencies, and detect bottlenecks.
   * Enhanced reporting tools could provide insights into resource usage, task completion rates, and system performance over time, supporting strategic planning.