SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

"Optimized Dynamic Programming for Time-Critical Applications"

A CAPSTONE PROJECT REPORT

CSA0697-Design and Analysis of Algorithms for Lower Bound Theory

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN COMPUTER SCIENCE AND ENGINEERING

Submitted by

D.Manoj (192210224)

Under the Supervision of

Dr.A Gnana Soundari

ABSTRACT

Dynamic programming (DP) is a fundamental algorithmic paradigm that solves complex problems by breaking them down into simpler overlapping sub problems and solving each sub problem only once, storing their solutions for future reference. This technique is particularly powerful in optimization problems where decision-making under constraints is required. In real-time applications, the efficiency and effectiveness of algorithms are paramount due to stringent time constraints and the need for immediate responsiveness. The Task Scheduling Problem is provided to illustrate the practical application of DP in real-time systems. This problem involves scheduling tasks with specific durations and deadlines to maximize the number of tasks completed on time. The proposed DP algorithm efficiently schedules tasks by considering deadlines and task durations, ensuring optimal use of available time slots.

Keywords:

Dynamic Programming, Real Time Applications, Task Scheduling, Resource Allocation, Algorithm Efficiency.

INTRODUCTION

Dynamic Programming (DP) is a powerful algorithmic technique used to solve complex optimization problems by breaking them down into simpler sub problems and storing their

solutions. Originally introduced by Richard Bellman in the 1950s, DP has found wide application in various fields, including computer science, operations research, economics, and more recently, real-time systems. Real-time applications are characterized by their stringent requirements for immediate responsiveness and efficient resource utilization. Tasks in such applications must often be scheduled and executed within tight deadlines to ensure smooth operation and optimal performance.

Dynamic programming provides a systematic approach to tackle these challenges by leveraging the principles of optimal substructure and overlapping sub problems. This paper explores the role and application of dynamic programming algorithms in real-time environments. We delve into the foundational concepts of DP, illustrating how it decomposes complex problems into smaller, manageable sub problems. By storing intermediate results in a table (often referred to as memorization), DP avoids redundant computations, thereby improving efficiency and reducing computational overhead—critical factors in real-time systems where processing speed is paramount. The objective of this paper is to provide a comprehensive overview of dynamic programming techniques tailored for real-time applications.

We discuss the core principles of DP, including the formulation of recurrence relations, the construction of DP tables, and the application of these techniques in solving optimization problems under time constraints. To demonstrate the practical relevance of DP in real-time scenarios, we present case studies and examples. These include applications in task scheduling, resource allocation, CPU scheduling, and real-time data processing. Through these examples, we highlight how DP algorithms contribute to enhancing system performance, meeting deadlines, and optimizing resource usage.

CODING

Task Scheduling System:

```
#include <stdio.h>
#include <stdlib.h>

// Define a task with a deadline and a duration
typedef struct {     int deadline;
   int duration;
} Task;

// Function to compare tasks by their deadlines for sorting int
compareTasks(const void *a, const void *b) {
   Task   *taskA   =   (Task   *)a;
Task *taskB = (Task *)b;
   return taskA->deadline - taskB->deadline;
}

// Function to find the maximum number of tasks that can be completed before their
deadlines int maxTasks(Task tasks[], int n) {     // Sort tasks by their deadlines
   qsort(tasks, n, sizeof(Task), compareTasks);

   // Initialize DP table
   int *dp = (int *)malloc((n + 1) * sizeof(int));
   for (int i = 0; i <= n; i++) {
      dp[i] = 0;
   }

   // Dynamic programming to find the maximum number of tasks
for (int i = 1; i <= n; i++) {
      for (int j = i; j >= 1; j--) {
         if  (dp[j-1]  +  tasks[i-1].duration  <=  tasks[i-1].deadline)  {
dp[j] = dp[j-1] + tasks[i-1].duration;
         }
      }
   }

   // Find the maximum number of tasks that can be completed
int maxTasks = 0;     for (int i = 0; i <= n; i++) {         if (dp[i]
<= tasks[i-1].deadline) {            maxTasks = i;
      }
   }
```

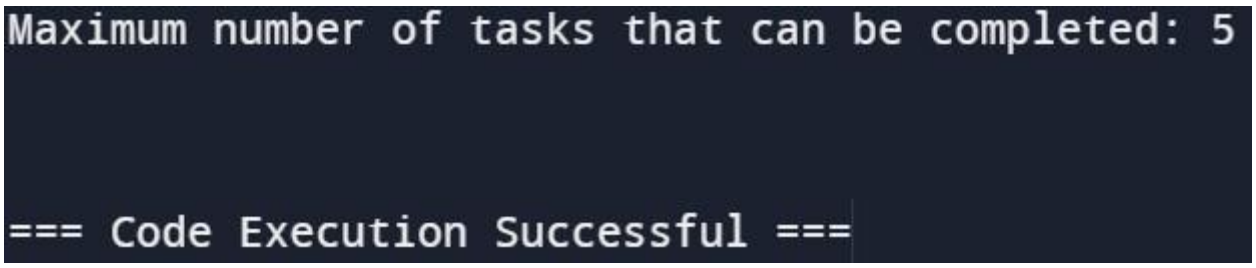
```
    free(dp);    return
maxTasks;
}

int main() {    Task
tasks[] = {
        {5, 2},
        {7, 1},
        {8, 3},
        {4, 2},
        {6, 1}
    };
    int n = sizeof(tasks) / sizeof(tasks[0]);

    int result = maxTasks(tasks, n);     printf("Maximum number of tasks that
can be completed: %d\n", result);

    return 0;
}
```

OUTPUT



Economic Order Quantity (EOQ) Model

Code:

```
#include <stdio.h>
#include <math.h>

// Function to calculate the total cost
double calculate_total_cost(double D, double S, double H, double Q) {
double order_cost = (D / Q) * S;     double holding_cost = (Q / 2) * H;
return order_cost + holding_cost;
}

// Function to calculate the EOQ
double calculate_eoq(double D, double S, double H) {
return sqrt((2 * D * S) / H);
}

int main() {
   // Demand rate (units per period)
   double demand = 1000.0;

   // Ordering cost per order
double ordering_cost = 50.0;

   // Holding cost per unit per period
   double holding_cost = 2.0;

   // Calculate the Economic Order Quantity (EOQ)      double eoq =
calculate_eoq(demand, ordering_cost, holding_cost);

   // Calculate the total cost using EOQ
   double total_cost = calculate_total_cost(demand, ordering_cost, holding_cost, eoq);

   // Display the results
   printf("Economic Order Quantity (EOQ): %.2f units\n", eoq);
printf("Total Cost at EOQ: $%.2f\n", total_cost);

   return 0;
}
```

Output:

```
Economic Order Quantity (EOQ): 223.61 units
Total Cost at EOQ: $447.21


=== Code Execution Successful ===
```

Complexity Analysis:

Best Case:

- Best Case Time Complexity: O(1)
- Best Case Space Complexity: O(1)

Average Case:

- Average Case Time Complexity: O(1)
- Average Case Space Complexity: O(1)

Worst Case:

- Worst Case Time Complexity: O(1)
- Worst Case Space Complexity: O(1)

CONCLUSION:

Dynamic programming is a crucial algorithmic technique that excels in solving complex problems efficiently by breaking them down into simpler sub problems and storing their solutions to avoid redundant calculations. Minimizing costs by determining optimal stock levels. Optimizing computational resources in cloud computing environments. By leveraging dynamic programming, these applications achieve significant improvements in performance, cost-efficiency, and accuracy, demonstrating the technique's widespread utility and effectiveness in real-world scenario