Analysis of Algorithms

BLG 335E

Project 3 Report

Abdullah Jafar Mansour Shamout
shamout21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 29.12.2023

1. Implementation

1.1. Complexities over Pseudo-codes

1.1.1. Weighted Interval Scheduling

Algorithm 1 Weighted Interval Scheduling

```
1: procedure WeightedIntervalScheduling(I)
2:     Sort intervals I by finish time                          ▷ O(n log n)
3:     Let dp[1...n] be an array                                ▷ O(n)
4:     dp[1] = p_1                                              ▷ O(1)
5:     for i = 2 to n do                                        ▷ O(n)
6:         Find the latest interval j such that f_j ≤ s_i       ▷ O(log n)
7:         dp[i] = max(dp[i − 1], p_i + dp[j])                  ▷ O(1)
8:     end for
9:     return dp[n]                                             ▷ O(1)
10: end procedure
```

The time complexity of the Weighted Interval Scheduling algorithm is $O(n \log n)$, where n is the number of intervals.

1.1.2. Knapsack Problem

Algorithm 2 Knapsack Problem

```
1: procedure Knapsack(items, budget)
2:     Let n be the number of items                             ▷ O(1)
3:     Let dp[0...n][0...budget] be a 2D array                  ▷ O(n × budget)
4:     for i = 0 to n do                                        ▷ O(n × budget)
5:         for w = 0 to budget do                               ▷ O(n × budget)
6:             if i = 0 or w = 0 then                           ▷ O(1)
7:                 dp[i][w] = 0                                 ▷ O(1)
8:             else if weight[i − 1] > w then                   ▷ O(1)
9:                 dp[i][w] = dp[i − 1][w]                      ▷ O(1)
10:            else
11:                dp[i][w] = max(dp[i − 1][w], value[i − 1] + dp[i − 1][w − weight[i − 1]]) ▷ O(1)
12:            end if
13:        end for
14:    end for
15:    return dp[n][budget]                                     ▷ O(1)
16: end procedure
```

The time complexity of the Knapsack Problem algorithm is $O(n \times \text{budget})$, where n is the number of items and budget is the maximum weight the knapsack can hold.

1.2. Explanation of the Code and Solution

The provided C++ code implements two algorithms: Weighted Interval Scheduling and the Knapsack Problem.

1.2.1. Weighted Interval Scheduling

The ‘WeightedIntervalScheduling‘ function in the code implements the Weighted Interval Scheduling algorithm. Here’s how it works:

1. **Sort Intervals:** First, the algorithm sorts the given intervals by their finish times. This sorting step takes $O(n \log n)$ time, where n is the number of intervals.

2. **Dynamic Programming:** Then, the algorithm iterates over the sorted intervals from the second interval to the last. For each interval, it finds the latest non-conflicting interval j and calculates the maximum priority gained by including the current interval. This step takes $O(n)$ time.

3. **Return Maximum Priority:** Finally, the algorithm returns the maximum priority that can be gained by scheduling the intervals optimally. This step takes $O(1)$ time.

The time complexity of the ‘WeightedIntervalScheduling‘ function is $O(n \log n)$, where n is the number of intervals.

1.2.2. Knapsack Problem

The ‘Knapsack‘ function in the code implements the Knapsack Problem algorithm. Here’s how it works:

1. **Dynamic Programming:** The algorithm uses a 2D array ‘dp‘ to store the maximum value that can be obtained with a given budget and a given number of items. It iterates over each item and each possible budget value, filling the ‘dp‘ array.

2. **Base Case:** If there are no items ($i = 0$) or the budget is zero ($w = 0$), the maximum value that can be obtained is zero.

3. **Main Calculation:** For each item, the algorithm checks if the current item can be included in the knapsack with the current budget. If it can, it considers whether it’s better to include the item or to exclude it. The maximum value is stored in the ‘dp‘ array.

4. **Return Maximum Value:** Finally, the algorithm returns the maximum value that can be obtained with the given budget and the given items.

The time complexity of the ‘Knapsack‘ function is $O(n \times \text{budget})$, where n is the number of items and budget is the maximum weight the knapsack can hold.

Overall, the provided code efficiently solves both the Weighted Interval Scheduling problem and the Knapsack Problem using dynamic programming.

1.3. Questions and Answers

1.3.1. What are the factors that affect the performance of the algorithm you developed using the dynamic programming approach?

1. **Number of Time Intervals**:
   The time complexity of the algorithm is influenced by the number of time intervals in the input data. A larger number of time intervals leads to longer processing times.

2. **Number of Rooms and Floors**:
   The algorithm's performance is influenced by the number of rooms and floors in the input. More rooms and floors result in larger data structures and potentially longer processing times.

3. **Priority Values**:
   The range and distribution of priority values in the input data significantly impact both the time and space complexity of the algorithm.

4. **Total Budget**:
   For the knapsack part of the algorithm, the total budget determines the size of the dynamic programming table. This directly affects both time and space complexity.

5. **Number of Items**:
   The performance of the algorithm is also affected by the number of items in the input data. A larger number of items leads to a larger dynamic programming table and potentially longer processing times.

1.3.2. What are the differences between Dynamic Programming and Greedy Approach? What are the advantages of dynamic programming?

Dynamic Programming

- **Definition**: Dynamic Programming is a method for solving complex problems by breaking them down into simpler subproblems and solving each subproblem only once. The results of the subproblems are stored, and the solution is built up by combining these results.

- **Optimality**: Dynamic programming guarantees an optimal solution since it considers all possible solutions to subproblems.

- **Example**: In the weighted interval scheduling problem, dynamic programming is used to find the optimal schedule by considering all possible combinations of intervals.

Greedy Approach

- **Definition**: Greedy approach, on the other hand, makes decisions based on the current best choice without considering the future consequences. It chooses the locally optimal solution at each step with the hope of finding a global optimum.
- **Optimality**: Greedy algorithms may not always find the optimal solution since they do not consider all possible solutions and may make choices that seem best at the moment but lead to suboptimal solutions in the end.
- **Example**: In the same weighted interval scheduling problem, a greedy approach might select intervals based solely on their immediate benefit without considering the overall optimal schedule.

Advantages of Dynamic Programming

- **Guaranteed Optimal Solution**: Dynamic programming guarantees an optimal solution since it systematically considers all possible solutions to subproblems.
- **Efficiency**: Despite considering all possible solutions, dynamic programming avoids redundant calculations by storing and reusing the results of subproblems.
- **Versatility**: Dynamic programming can be applied to a wide range of problems, from optimization to combinatorial problems, making it a versatile problem-solving technique.