Analysis of Algorithms II

BLG 336E

Project 3 Report

Mustafa Can Çalışkan
caliskanmu20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering

1. Introduction

In this project, I aim to solve the Weighted Interval Scheduling Problem (WISP) and the Knapsack Problem using dynamic programming techniques. These problems are fundamental in the field of computer science and operations research, often arising in various practical applications such as resource allocation, scheduling, and budgeting.

2. Implementation

2.1. Pseudo-Codes and Time Complexities

2.1.1. SortByFinishTime

Algorithm 1 SortByFinishTime

```
function SortByFinishTime(a, b)
    return a.fTime < b.fTime
end function
```

- Time Complexity: $O(1)$
- The function only compares two values, hence the constant time complexity.

2.1.2. FindLastNonConflictingSchedule

Algorithm 2 FindLastNonConflictingSchedule

```
function FindLastNonConflicting(sch, curIdx)
    low ← 0
    high ← curIdx − 1
    while low ≤ high do
        mid ← ⌊(low + high)/2⌋
        if sch[mid].fTime ≤ sch[curIdx].sTime then
            if mid + 1 ≤ high and sch[mid + 1].fTime ≤ sch[curIdx].sTime then
                low ← mid + 1
            else
                return mid
            end if
        else
            high ← mid − 1
        end if
    end while
    return −1
end function
```

- Time Complexity: $O(\log n)$
- This function performs a binary search, thus the time complexity is logarithmic with respect to the number of schedules.

2.1.3. WeightedIntervalScheduling

Algorithm 3 WeightedIntervalScheduling

```
function WIS(sch)
    Sort sch by finish time
    n ← size of sch
    maxP ← array of size n initialized to 0
    lastIdx ← array of size n initialized to -1
    maxP[0] ← sch[0].priority
    lastIdx[0] ← 0
    for i ← 1 to n − 1 do
        curProfit ← sch[i].priority
        lastNC ← FindLastNonConflicting(sch, i)
        if lastNC ≠ −1 then
            curProfit ← curProfit + maxP[lastNC]
        end if
        if curProfit > maxP[i − 1] then
            maxP[i] ← curProfit
            lastIdx[i] ← i
        else
            maxP[i] ← maxP[i − 1]
            lastIdx[i] ← lastIdx[i − 1]
        end if
    end for
    optSch ← empty array
    idx ← lastIdx[n − 1]
    while idx ≠ −1 do
        Append sch[idx] to optSch
        idx ← FindLastNonConflicting(sch, idx)
        if idx ≠ −1 then
            idx ← lastIdx[idx]
        end if
    end while
    Reverse optSch
    return optSch
end function
```

- Time Complexity: $O(n \log n)$
- Sorting the schedules takes $O(n \log n)$, and filling the DP array takes $O(n \log n)$ due to the binary search in each iteration. Reconstructing the solution takes $O(n)$.

2.1.4. Knapsack

Algorithm 4 Knapsack

```
function Knapsack(items, budget)
    n ← size of items
    maxV ← 2D array of size (n + 1) × (budget + 1)  initialized to 0
    include ← 2D array of size (n + 1) × (budget + 1)  initialized to false
    for i ← 1 to n do
        for b ← 0 to budget do
            if items[i − 1].price ≤ b then
                potV ← items[i − 1].priority + maxV[i − 1][b − int(items[i − 1].price)]
                if potV > maxV[i − 1][b] then
                    maxV[i][b] ← potV
                    include[i][b] ← true
                else
                    maxV[i][b] ← maxV[i − 1][b]
                end if
            else
                maxV[i][b] ← maxV[i − 1][b]
            end if
        end for
    end for
    result ← empty array
    remB ← budget
    for i ← n downto 1 and remB > 0 do
        if include[i][remB] then
            Append items[i − 1] to result
            remB ← remB − int(items[i − 1].price)
        end if
    end for
    return result
end function
```

- Time Complexity: $O(n \cdot W)$
- The function iterates over each item and each possible budget value, resulting in a time complexity of $O(n \cdot W)$, where n is the number of items and W is the budget.

2.1.5. CustomRound

Algorithm 5 CustomRound

function CustomRound(num)
  $decPart \leftarrow num - \lfloor num \rfloor$
  $roundDec \leftarrow$ round$(decPart \times 10)/10$
  if $roundDec \geq 0.5$ **then**
    return $\lceil num \times 10 \rceil / 10$
  else
    return $\lfloor num \times 10 \rfloor / 10$
  end if
end function

- Time Complexity: $O(1)$
- The function performs a fixed number of arithmetic operations, resulting in a constant time complexity.

2.1.6. ReadItems

Algorithm 6 ReadItems

function ReadItems($path$)
  $itemF \leftarrow$ open file at $path + "/items.txt"$
  if $itemF$ is not open **then**
    Throw error: Failed to open items file.
  end if
  $items \leftarrow$ empty array
  $line \leftarrow$ read line from $itemF$ (assuming first line is header)
  while line from $itemF$ is not empty **do**
    $ss \leftarrow$ stringstream from $line$
    $name, price, value \leftarrow$ values from ss
    Append $\{\{value\}, name, price\}$ to $items$
  end while
  return $items$
end function

- Time Complexity: $O(n)$
- The function reads each line of the file once and processes it, resulting in a linear time complexity relative to the number of lines in the file.

2.1.7. ReadPriorityMap

Algorithm 7 ReadPriorityMap

function ReadPriorityMap($path$)
  $priF \leftarrow$ open file at $path + "/priority.txt"$
  if $priF$ is not open **then**
    Throw error: Failed to open priority file.
  end if
  $priMap \leftarrow$ empty unordered map
  $line \leftarrow$ read line from $priF$ (assuming first line is header)
  while line from $priF$ is not empty **do**
    $ss \leftarrow$ stringstream from $line$
    $floor, room, pri \leftarrow$ values from ss
    $priMap[floor + "" + room] \leftarrow pri$
  end while
  return $priMap$
end function

- Time Complexity: $O(n)$
- The function reads each line of the file once and processes it, resulting in a linear time complexity relative to the number of lines in the file.

2.1.8. ReadFloorSchedules

Algorithm 8 ReadFloorSchedules

```
function ReadFloorSchedules(path, priMap)
    roomF ← open file at path + "/room_time_intervals.txt"
    if roomF is not open then
        Throw error: Failed to open room time intervals file.
    end if
    floorSch ← empty map
    line ← read line from roomF  (assuming first line is header)
    while line from roomF is not empty do
        ss ← stringstream from line
        floor, room, start, end ← values from ss
        sch ← new Schedule struct
        sch.floor ← floor
        sch.room ← room
        sch.sTime ← convert start to minutes
        sch.fTime ← convert end to minutes
        sch.priority ← priMap.at(floor + "" + room)
        Append sch to floorSch[floor]
    end while
    return floorSch
end function
```

- Time Complexity: $O(n)$
- The function reads each line of the file once and processes it, resulting in a linear time complexity relative to the number of lines in the file.

2.1.9. main

Algorithm 9 Main Function

```
function main(argc, argv)
    total_budget ← 200000
    case_no ← argv[1]
    case_name ← "case_" + case_no
    path ← "./inputs/" + case_name
    items ← readItems(path)
    priorityMap ← readPriorityMap(path)
    floorSchedules ← readFloorSchedules(path, priorityMap)
    for all floor in floorSchedules do
        schedules ← floorSchedules[floor]
        optimalSchedules ← weighted_interval_scheduling(schedules)
        totalPriority ← 0
        for all sched in optimalSchedules do
            totalPriority ← totalPriority + sched.roomPriority.priority
        end for
        print(floor, totalPriority)
        for all sched in optimalSchedules do
            Print Schedules
        end for
    end for
    selectedItems ← knapsack(items, total_budget)
    totalValue ← 0.0
    for all item in selectedItems do
        totalValue ← totalValue + item.value.priority
    end for
    print(totalValue)
    for all item in selectedItems do
        print(item.itemName)
    end for
end function
```

The main function of the program involves several key operations whose time complexity we need to analyze individually:

1. **Reading Items:**
    - Function: `readItems`
    - Complexity: $O(n)$, where n is the number of items. This is because each item is read from the file and inserted into a vector.
2. **Reading Priority Map:**
    - Function: `readPriorityMap`

- Complexity: $O(p)$, where p is the number of priority entries. Each entry is read from the file and inserted into an unordered map.

3. **Reading Floor Schedules:**

   - Function: `readFloorSchedules`
   - Complexity: $O(s \log p)$, where s is the number of schedules and p is the number of priority entries. Each schedule is read, its priority is fetched from the unordered map (which is $O(1)$ on average), and it is inserted into a map.

4. **Finding Optimal Schedules:**

   - Function: `weighted_interval_scheduling`
   - Complexity: $O(s \log s)$ for sorting the schedules by finish time and $O(s \log s)$ for the dynamic programming approach (each schedule involves a binary search).

5. **Solving the Knapsack Problem:**

   - Function: `knapsack`
   - Complexity: $O(nb)$, where n is the number of items and b is the budget. This is due to the dynamic programming table being filled with n items and b budget constraints.

6. **Printing Results:**

   - Complexity: $O(s+n)$, where s is the number of schedules and n is the number of items. This is because each optimal schedule and each selected item is printed.

Combining these complexities, the overall time complexity of the main function is dominated by the most expensive operations:

$$
\begin{aligned}
\text{Total Time Complexity} &= O(n) + O(p) + O(s \log p) + O(s \log s) + O(nb) + O(s+n) \\
&= O(n) + O(p) + O(s \log p) + O(s \log s) + O(nb) \\
&= O(s \log s + nb)
\end{aligned}
$$

Here, s is the number of schedules, n is the number of items, and b is the budget.

3. Discussions

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

   The performance of the algorithm developed using the dynamic programming approach is primarily affected by the size of the input data, including the number of schedules (s) and items (n), as well as the budget (b). For the interval scheduling problem, the time complexity $O(s \log s)$ is influenced by the need to sort schedules and perform binary searches. For the knapsack problem, the time complexity $O(nb)$ is affected by the dynamic programming table, where n is the number of items and b is the budget. Additionally, the efficiency of file reading operations and the structure of data (e.g., distribution of start and finish times for schedules) can impact performance. External factors such as the system's memory capacity and CPU speed also play a role in the overall efficiency of the algorithm.

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

   Dynamic Programming (DP) and Greedy approaches differ primarily in their problem-solving strategies. DP solves problems by breaking them into overlapping subproblems, solving each just once, and storing their solutions, making it suitable for problems with overlapping subproblems and optimal substructure, such as the knapsack problem. In contrast, the Greedy approach makes a series of locally optimal choices, aiming for a global optimum without considering future consequences, which works well for problems with the greedy-choice property and optimal substructure, like the fractional knapsack problem. The advantage of DP is that it guarantees finding an optimal solution by exploring all possible subproblem combinations and storing results to avoid redundant calculations, making it highly efficient for complex problems that cannot be effectively solved by greedy algorithms.

4. Conclusion

In this project, I have successfully implemented algorithms to solve the Weighted Interval Scheduling Problem (WISP) and the Knapsack Problem using dynamic programming approaches. Through detailed pseudo-code and analysis, I have demonstrated the effectiveness and efficiency of these algorithms.

The dynamic programming solutions for both problems exhibit significant advantages, such as ensuring optimal solutions and efficiently handling overlapping subproblems. The WISP algorithm, with its $O(n \log n)$ time complexity, effectively maximizes the sum of priorities for non-overlapping intervals. Similarly, the Knapsack algorithm, with its $O(n \cdot W)$ time complexity, optimally selects items to maximize the total value within the given budget constraints.

By addressing these problems, I have gained a deeper understanding of dynamic programming techniques and their applications in solving complex optimization problems. This project underscores the importance of algorithmic strategies in computer science and their practical implications in various fields.

Through this endeavor, I have not only enhanced my problem-solving skills but also contributed to the broader understanding of how dynamic programming can be leveraged to tackle real-world challenges efficiently and effectively.