Analysis of Algorithms II

BLG 336E

Project 3 Report

KAAN KARATAŞ

karatask20@itu.edu.tr

Faculty of Computer and Informatics Engineering

Department of Computer Engineering

Date of submission: 16.05.2024

1. Implementation

1.1. Introduction

The project revolves around implementing a scheduling system to manage renovation works of the new computer department faculty building. The core objective is to optimize the allocation of resources across various floors, ensuring timely completion of renovation tasks while prioritizing critical areas. The system is structured to handle input data from text files. A weighted interval scheduling algorithm is designed and used to maximize priority gain by assigning tasks within available time intervals. The program serves to measure traversal time, serving as a benchmark for evaluating the efficiency of different scheduling strategies.

1.2. Struct Types

1.2.1. Time Interval Struct

The "TimeIntervals" struct encapsulates essential data representing intervals of availability within the faculty building, including attributes such as the floor name, room number, start time, and end time. It facilitates efficient scheduling and resource allocation for renovation tasks.

Algorithm 1 TimeIntervals Structure Definition

```
struct TimeIntervals {
        string floor_name, room_no, start_time, end_time;
};
```

1.2.2. Priority Struct

The "Priority" struct represents the priority level assigned to each room within the faculty building. It contains attributes such as the floor name, room number, and room priority, enabling the scheduling system to prioritize renovation tasks based on predefined criteria.

Algorithm 2 Priority Structure Definition

```
struct Priority {
        string floor_name, room_no;
        int room_priority;
};
```

1.2.3. Item Struct

The "Item" struct represents an item that can be considered for purchase or inclusion in a collection. It comprises attributes such as the item's name, price, and value, providing essential information for decision-making processes like budget allocation or optimization.

Algorithm 3 Item Structure Definition

```
struct Item {
    string name;
    int price;
    float value;
};
```

1.2.4. Schedule Struct

The Schedule struct combines time intervals and priority information, encapsulating the scheduling data necessary for renovation tasks within the faculty building. It includes attributes such as TimeIntervals and Priority, allowing the system to organize and prioritize renovation activities efficiently. Additionally, it overloads the < operator to facilitate sorting based on the end times of intervals, enabling chronological ordering for scheduling tasks.

Algorithm 4 Schedule Structure Definition

```
struct Schedule {
    TimeIntervals interval;
    Priority priority;
    bool operator<(const Schedule &other) const {
        return interval.end_time < other.interval.end_time;        ▷ $O(1)$
    }
};
```

1.3. Helper Functions

The "is_conflicting" function determines whether two schedules overlap, indicating a conflict in the renovation schedule. It compares the end time of the first schedule with the start time of the second schedule and returns true if they conflict.

Algorithm 5 is_conflicting Function

```
function is_conflicting(s1, s2)
    return stoi(s1.interval.end_time) > stoi(s2.interval.start_time);        ▷ $O(1)$
end function
```

The "latest_non_Conflicting" function searches for the latest non-conflicting schedule preceding the current schedule in a sorted list of schedules. It iterates backward through the list, checking for conflicts with earlier schedules. If a non-conflicting schedule is found, its index is returned; otherwise, it returns -1 if no such schedule exists. These functions are essential for ensuring that renovation tasks are scheduled without conflicts and in an optimal order.

Algorithm 6 latest_non_Conflicting Function

```
1: function latest_non_Conflicting(schedules, i)
2:     for $j = i - 1$ to $0$ do                                  ▷ $O(n)$
3:         if !is_conflicting(schedules[j], schedules[i]) then    ▷ $O(1)$
4:             return $j$;                                        ▷ $O(1)$
5:         end if
6:     end for
7:     return $-1$;                                               ▷ $O(1)$
8: end function
```

1.4. Weighted Interval Scheduling

The "weighted_interval_scheduling" function implements a dynamic programming approach to find the optimal schedule for renovation tasks based on their priorities and time intervals. It begins by sorting the schedules by their end times, ensuring chronological order. Then, it iterates through the sorted schedules, calculating the maximum priority achievable for each schedule while considering conflicts with earlier tasks. By dynamically updating the maximum priority for each schedule, it constructs a solution that maximizes the overall priority gain. Finally, it traverses the schedule list backward to identify the optimal solution, excluding conflicting schedules and ensuring the highest priority tasks are included.

Algorithm 7 weighted_interval_scheduling Function

```
 1: function weighted_interval_scheduling($schedules$)
 2:     $optimal\_solution \leftarrow \{\}$;                                          ▷ $O(1)$
 3:     if $schedules.size() == 0$ then
 4:         return $optimal\_solution$;
 5:     end if                                                                    ▷ $O(1)$
 6:     sort($schedules.begin(), schedules.end()$);                           ▷ $O(n \log n)$
 7:     $max\_priority \leftarrow$ vector of size $schedules.size()$ initialized with zeros;   ▷ $O(n)$
 8:     $max\_priority[0] \leftarrow schedules[0].priority.room\_priority$;          ▷ $O(1)$
 9:     for $i \leftarrow 1$ to $schedules.size() - 1$ do                                ▷ $O(n)$
10:         $latest\_non\_conflicting\_index \leftarrow$ latest_non_Conflicting($schedules, i$);   ▷ $O(n)$
11:         $priority \leftarrow schedules[i].priority.room\_priority$;                ▷ $O(1)$
12:         if $latest\_non\_conflicting\_index \neq -1$ then                         ▷ $O(1)$
13:             $priority += max\_priority[latest\_non\_conflicting\_index]$;         ▷ $O(1)$
14:         end if
15:         $max\_priority[i] \leftarrow$ max of $priority$ and $max\_priority[i-1]$;       ▷ $O(1)$
16:     end for
17:     $reverse\_index \leftarrow schedules.size() - 1$;                                ▷ $O(1)$
18:     while $reverse\_index \geq 0$ do                                              ▷ $O(n)$
19:         if $reverse\_index == 0$ or $max\_priority[reverse\_index] \neq max\_priority[reverse\_index - 1]$ then   ▷ $O(1)$
20:             push_back $schedules[reverse\_index]$ into $optimal\_solution$;       ▷ $O(1)$
21:             $reverse\_index \leftarrow$ latest_non_Conflicting($schedules, reverse\_index$);   ▷ $O(n)$
22:             continue;                                                         ▷ $O(1)$
23:         end if
24:         $reverse\_index -= 1$;                                                     ▷ $O(1)$
25:     end while
26:     reverse($optimal\_solution.begin(), optimal\_solution.end()$);                ▷ $O(n)$
27:     return $optimal\_solution$;                                                  ▷ $O(1)$
28: end function
```

The time complexity of the weighted_interval_scheduling function is dominated by the sorting operation at $O(n \log n)$, where n is the number of schedules. The subsequent loop iterating over the schedules has a linear time complexity of $O(n)$, as it performs operations such as finding the latest non-conflicting schedule and updating priorities. Therefore, the overall time complexity of the function is $O(n \log n)$ due to the sorting step, where n is the number of schedules provided as input.

1.5. Knapsack Algorithm

The "knapsack" function implements a dynamic programming approach to solve the knapsack problem, where the objective is to select a combination of items with the maximum value while respecting a given budget constraint. It initializes a 2D vector dp to store the maximum value achievable for different combinations of items and budgets.

It then iterates over each item and budget combination, calculating the maximum value that can be achieved considering whether to include the current item or not. Once the dynamic programming table is filled, it backtracks to determine which items were selected to achieve the maximum value.

Algorithm 8 knapsack Function

```
1: function knapsack(items, budget)
2:     $dp \leftarrow$ 2D vector of size $($items.size()$ + 1) \times ($budget$ + 1)$ initialized with zeros;    ▷ $O(n \cdot$ budget$)$
3:     for $i \leftarrow 1$ to items.size() do    ▷ $O(n)$
4:         for $j \leftarrow 1$ to budget do    ▷ $O($budget$)$
5:             if items$[i-1].price \leq j$ then    ▷ $O(1)$
6:                 $dp[i][j] \leftarrow$ max of $dp[i-1][j]$ and $dp[i-1][j -$ items$[i-1].price] +$ items$[i-1].value$;    ▷ $O(1)$
7:             else
8:                 $dp[i][j] \leftarrow dp[i-1][j]$;    ▷ $O(1)$
9:             end if
10:        end for
11:    end for
12:    $selectedItems \leftarrow$ empty vector;    ▷ $O(1)$
13:    $i \leftarrow$ items.size();    ▷ $O(1)$
14:    $j \leftarrow$ budget;    ▷ $O(1)$
15:    while $i > 0$ and $j > 0$ do    ▷ $O(n)$
16:        if $dp[i][j] \neq dp[i-1][j]$ then    ▷ $O(1)$
17:            push_back items$[i-1]$ into $selectedItems$;    ▷ $O(1)$
18:            $j \leftarrow j -$ items$[i-1].price$;    ▷ $O(1)$
19:        end if
20:        $i \leftarrow i - 1$;    ▷ $O(1)$
21:    end while
22:    return $selectedItems$;    ▷ $O(1)$
23: end function
```

The time complexity of the knapsack function is $O(n \cdot budget)$, where n is the number of items and budget is the maximum budget, which ultimately comes down to $O(n)$ This is because the function utilizes dynamic programming to compute the maximum value that can be obtained with a given budget, considering each item. The nested loops iterate through all items and budgets, resulting in a time complexity proportional to the product of these two quantities.

1.6. Fill Schedules

The "fill_schedules" function populates a map of schedules based on provided room intervals and priorities. It iterates over each room's intervals, then over each time interval within those rooms. For each time interval, it creates a Schedule object and assigns the time interval and priority obtained from the input maps. These schedules are then added to the corresponding room's schedule list in the output map.

Algorithm 9 fill_schedules Function

```
1:  function fill_schedules(roomIntervals, priorities, schedules)
2:      for interval in roomIntervals do                                          ▷ $O(n)$
3:          for timeInterval in interval.second do                                ▷ $O(m)$
4:              $schedule \leftarrow$ new $Schedule$ object;                      ▷ $O(1)$
5:              $schedule.interval \leftarrow$ timeInterval;                      ▷ $O(1)$
6:              $schedule.priority \leftarrow$ priorities[interval.first + timeInterval.room_no];   ▷ $O(1)$
7:              schedules[interval.first].push_back(schedule);                    ▷ $O(1)$
8:          end for
9:      end for
10: end function
```

1.7. Input Reading

These four functions are responsible for the extraction of data from text files, preparing it for further processing within the scheduling system.

- "read_room_time_intervals": This function reads time intervals for each room from a specified file and organizes them into a map structure. It iterates through each line of the file, parsing the floor name, room number, start time, and end time for each interval. These intervals are then grouped by floor name and stored in a map.

- "read_priorities": Similarly, this function reads priority data from a file and constructs a map of priorities for each room. It iterates through the lines of the file, extracting the floor name, room number, and priority level for each entry. These priorities are then stored in a map, indexed by a combination of floor name and room number.

- "read_items": This function is responsible for reading item data from a file, preparing it for use in the knapsack optimization process. It iterates through each line of the file, extracting the name, price, and value of each item. These items are then stored in a vector, ready to be processed by the knapsack algorithm to determine the optimal selection within budget constraints.

The pseudo code of these 3 functions can be observed below.

Algorithm 10 read_room_time_intervals Function

```
function read_room_time_intervals(filename)
    intervals ← empty map of string to vector of TimeIntervals;        ▷ $O(1)$
    ifstream file(filename);                                             ▷ $O(1)$
    if file.is_open() then                                               ▷ $O(1)$
        string line;                                                     ▷ $O(1)$
        getline(file, line);                                             ▷ $O(1)$
        while getline(file, line) do                                     ▷ $O(n)$
            stringstream ss(line);                                       ▷ $O(1)$
            TimeIntervals interval;                                      ▷ $O(1)$
            getline(ss, interval.floor_name, ' ');                       ▷ $O(1)$
            getline(ss, interval.room_no, ' ');                          ▷ $O(1)$
            getline(ss, interval.start_time, ' ');                       ▷ $O(1)$
            getline(ss, interval.end_time, ' ');                         ▷ $O(1)$
            intervals[interval.floor_name].push_back(interval);          ▷ $O(1)$
        end while
        file.close();                                                    ▷ $O(1)$
    end if
    return intervals;                                                    ▷ $O(1)$
end function
```

Algorithm 11 read_priorities Function

```
function read_priorities(filename)
    priorities ← empty map of string to Priority;                        ▷ $O(1)$
    ifstream file(filename);                                             ▷ $O(1)$
    if file.is_open() then                                               ▷ $O(1)$
        string line;                                                     ▷ $O(1)$
        getline(file, line);                                             ▷ $O(1)$
        while getline(file, line) do                                     ▷ $O(n)$
            stringstream ss(line);                                       ▷ $O(1)$
            Priority priority;                                           ▷ $O(1)$
            getline(ss, priority.floor_name, ' ');                       ▷ $O(1)$
            getline(ss, priority.room_no, ' ');                          ▷ $O(1)$
            ss » priority.room_priority;                                 ▷ $O(1)$
            priorities[priority.floor_name + priority.room_no] ← $priority$;   ▷ $O(1)$
        end while
        file.close();                                                    ▷ $O(1)$
    end if
    return priorities;                                                   ▷ $O(1)$
end function
```

Algorithm 12 read_priorities Function

```
function read_priorities(filename)
    priorities ← empty map of string to Priority;                              ▷ $O(1)$
    ifstream file(filename);                                                    ▷ $O(1)$
    if file.is_open() then                                                      ▷ $O(1)$
        string line;                                                            ▷ $O(1)$
        getline(file, line);                                                    ▷ $O(1)$
        while getline(file, line) do                                            ▷ $O(n)$
            stringstream ss(line);                                              ▷ $O(1)$
            Priority priority;                                                  ▷ $O(1)$
            getline(ss, priority.floor_name, ' ');                              ▷ $O(1)$
            getline(ss, priority.room_no, ' ');                                 ▷ $O(1)$
            ss » priority.room_priority;                                        ▷ $O(1)$
            priorities[priority.floor_name + priority.room_no] ← $priority$;    ▷ $O(1)$
        end while
        file.close();                                                           ▷ $O(1)$
    end if
    return priorities;                                                          ▷ $O(1)$
end function
```

1.8. Printing The Outputs

These functions handle the display of selected items and schedules, providing valuable insights into the scheduling and resource allocation process.

- "print_selected_items": This function prints the selected items along with their total value. It iterates through the vector of selected items, accumulating their values to calculate the total value. Afterward, it prints the total value followed by each item's name, allowing for easy visualization of the selected items and their overall value.
- "print_schedule": Similarly, this function prints the schedule for each floor, along with the total priority gain. It first calculates the total priority gain by summing up the priorities of all schedules. Then, it prints the floor name and priority gain. For each schedule, it prints the floor name, room number, start time, and end time.

The pseudo-code of these 2 functions can be seen below.

Algorithm 13 print_selected_items Function

```
procedure print_selected_items(selectedItems)
    totalValue ← $0.0$;                                  ▷ $O(1)$
    for each item in selectedItems do                    ▷ $O(n)$
        totalValue $+=$ item.value;                      ▷ $O(1)$
    end for
    print"Total Value –> "totalValue;                    ▷ $O(1)$
    for each item in selectedItems do                    ▷ $O(n)$
        printitem.name;                                  ▷ $O(1)$
    end for
end procedure
```

Algorithm 14 print_schedule Function

```
procedure print_schedule(schedules)
    totalValue ← $0$;                                    ▷ $O(1)$
    for each schedule in schedules do                    ▷ $O(n)$
        totalValue $+=$ schedule.priority.room_priority; ▷ $O(1)$
    end for
    printschedules[0].priority.floor_name" –> Priority Gain: ";   ▷ $O(1)$
    fixedsetprecision(1)round(totalValue);               ▷ $O(1)$
    for each schedule in schedules do                    ▷ $O(n)$
        printschedule.interval.floor_nameschedule.interval.room_no;   ▷ $O(1)$
        setw(2)setfill('0');                             ▷ $O(1)$
        schedule.interval.start_timesetw(2)setfill('0')schedule.interval.end_time;   ▷
$O(1)$
    end for
end procedure
```

2. Extra Comments

2.1. Performance Of The Algorithm

The performance of the dynamic programming algorithm developed for scheduling is influenced by several factors:

- Number of Schedules: The algorithm's performance may decrease as the number of schedules increases since iterating over all schedules is involved to calculate priorities and determine conflicts.
- Interval Length: Longer intervals may lead to more conflicts and subsequently impact performance due to the increased number of comparisons required.
- Resources of the System: The availability of computational resources, such as CPU and memory, can affect the algorithm's performance. Insufficient resources may lead to slower execution.
- Priority Assignment: The efficiency of the algorithm can be affected by how priorities are assigned to schedules. If priorities are unevenly for example, it may lead to longer processing times.

2.2. Dynamic Programming vs Greedy Approach

The DP and the Greedy Approach have unique ways of dealing with optimization problems: they approach it differently. DP is one of the key features of dynamic programming that separates problems into subproblems of smaller size, and each problem is only solved once and the results are stored in the table for re-use. By enumeration of all possible choices and recursively, it ensures the best solution for each sub-problem. Contrary to this, the Greedy Approach makes locally optimal decisions at every specific point, trying to get immediate rather than thinking of long-term outcomes. However, the advantage of the DP approach is that it is optimized and applicable across a wide range of problems, while the greedy method concentrates on simplicity and speed, usually giving up optimality for speed. While DP is computationally intensive it gives a guarantee of a best solution using which many optimization problems can be solved and hence is a power tool.