BLG 336E Analysis of Algorithms II Homework 3

Due: Thursday, May 16, 2024

Lecturer: Prof. Mehmet Baysan CRN: 21349

Fatih Baskın
150210710

May 14, 2024

Contents

1 Pseudocodes **3**
1.1 Knapsack . . . 3
1.2 Weighted Interval Scheduling . . . 4

2 Questions & Answers **6**
2.1 Factors Affecting the Performance . . . 6
2.2 Differences Between Greedy & Dynamic Programming . . . 7

3 Implementation Details **7**
3.1 Extra Libraries Used . . . 7
3.2 Extra Structs . . . 7
3.3 Extra Functions . . . 7
3.4 Predefined Structs . . . 8

4 Problems Encountered **8**
4.1 Problem of Multiple Solutions . . . 8
4.2 Floating Point Estimation Errors . . . 8

1 Pseudocodes

1.1 Knapsack

For selecting the items with the most value for the computer engineering department, the knapsack method is used. Our particular problem was a 0-1 knapsack problem, every item had only one instance and you could decide on taking that item or not.

To make it understandable with regular knapsack terms, items had item names, price,s and value attributes. Normally in knapsack problems, there are item numbers, weight, and price. In normal knapsack, the limiting factor is the capacity of the knapsack. On here however, the limiting factor is the budget and the weights of the items are the prices of those items. We are trying to maximize the total value of the items for the department. The time complexity of this algorithm would be $O(N \times C)$ where N is the number of items and C is the capacity (in our terms the budget). Space complexity is the same, $O(N \times C)$ as well. Here is the pseudocode:

```
FUNCTION KNAPSACK (Items, budget):
  n <- LENGTH(Items)
  initialize DP matrix with sizes n, budget
  i <- 1
  b <- 1
  // Dynammic Programming to pick items
  FOR i in range 1 to n: // N iterations
    FOR b in range 1 to budget: // B iterations
      current_item <- Items[i]
      previous_budget <- b - current_item.price
      IF previous_budget < 0: // We cannot take the current item due to current budget
        IF i = 1: // If there is no item before current item
          DP[i][b] <- 0
        ELSE: // There is an item before current item
          DP[i][b] <- DP[i-1][b] // Take the solution before current item
        ENDIF
      ELSE: // There is enough budget to take current item
        IF i = 1: // If there is no item before current item
          DP[i][b] <- current_item.value // Take the current item
        ELSE: // There are previous items
          // Then we can either don't take the item or take the item
          value_without_taking_item <- DP[i-1][b]
          value_With_taking_item <- DP[i-1][b-1] + current_item.value
          DP[i][b] <- max(value_without_taking_item, value_With_taking_item)
        ENDIF
      ENDIF
    ENDFOR
  ENDFOR
  // Backtracking to pick selected items
  initialize selected_items as an empty array
  i <- n
  b <- budget
  // At most one decrements for n and b, O(n + b) complexity here
  while i > 0 and b > 0: // We need to backtrack using DP matrix
    current_item = Items[i]
    IF i = 1: // Corner case, only one item remaining
      IF b >= current_item.value: // There is still enough budget to take the item
        selected_items.append(current_item) // Take the current item
      ELSE: // Then there is not enough budget to take the item
        i <- i-1
      ENDIF
    ELSE: // Then it is not a corner case, there are plenty of items left
      IF DP[i][b] = DP[i-1][b]: // If current solution is the same with subproblem
        i <- i-1 // without current item, then don't take the current item
      ELSE: // If current problem did cause some changes in the values, then take
```

```
        selected_items.append(current_item) // the current item
        i <- i - 1 // go to the subproblem without current item
        b <- b - current_item.vaulue // go to the subproblem with less budget
      ENDIF
    ENDIF
  ENDWHILE
  RETURN selected_items
```

Note that the provided pseudocode is similar but not as the same as my code. I have used zero padding to my advantage since it is possible to reach zero indexes and consequently, I had less if-else branching in my code. But, the logic implied above is the same as my code. You can take the current item with current budget or don't. You need to check the subproblem with the current budget and with previous items if you don't include the current item or the subproblem with the previous budget and previous items if you include the current item.

DP[current item][current budget] = max(DP[previous item][current budget], DP[previous item][previous budget] + current item value)

With the same logic, I have backtracked and saved the selected items. This operation has the time complexity of $O(N + C)$ since I can decrement item count N times and current budget by C times. I would go out of bounds if I reached a value smaller than 1. Since I can decrement budget or item index by 1 in the worst case, then it has a time complexity of $O(N + C)$. The space complexity is $O(N)$ since it is possible to select every item.

Overall, the time complexity is $O(N \times C)$ and space complexity is $O(N \times C)$.

1.2 Weighted Interval Scheduling

Weighted interval scheduling is used for selecting schedules by picking the best compatible bunch of schedules (compatible meaning non-intersecting) that maximizes their weights (their values).

Each schedule would have its start-time, end-time, and weight. For our problem each schedule had a floor, a room, a start-time, and an end-time. Each room on each floor had their weights. Therefore converting the given time intervals and priorities into a compatible schedule was required. Because of this, I have used the map data structure from C++ STL (red-black tree with key-value pairs) and mapped each floor with their rooms and rooms with their priorities using maps inside maps. With the floor key, in the first map, the user would get a map containing rooms and priorities. With the room key, the user would get the priority of that room. Constructing this double map data structure would take $O(Nlog^2N)$ assuming N is the number of rooms. Accessing a room's priority would take $O(log^2N)$ time. The space complexity of this map is $O(N)$ since there are N entries.

After mapping the priorities, proper data structure for storing the start-time, end-time, room name, and room weight (schedule weight) was necessary. Assuming there are M room time intervals, I have used the previously constructed map to construct a new map, mapping newly created data structure `Schedule` containing start-time, end-time, room name and room priority with the corresponding floor. The time complexity of this operation is $O(M \times log^2N)$ with space complexity of $O(M)$ since there are M entries in the tree. Pseudocode for the preprocessing is given below

```
FUNCTION preprocess_priorities(priorities, time_intervals):
  // Initialize a map to store priority of room x in floor y
  initialize floor_room_priority as an empty map of string - map of string - int
  // Populate the floor_room_priority map
  FOR each priority IN priorities: // N log^2 N complexity
    floor_room_priority[priority.floor][priority.room] <- priority.priority
  ENDFOR
  // Initialize a map to store schedules for each floor
  initialize schedules as an empty map of string - array of Schedule
  // Populate the schedules map using the floor_room_priority map
  FOR each time_interval IN time_intervals: // M log^2 N complexity
    floor <- time_interval.floor
    room <- time_interval.room
```

```
    start <- time_interval.start
    end <- time_interval.end
    priority <- floor_room_priority[floor][room]
    // Add the schedule to the schedules of the floor
    schedules[floor].append(Schedule(room, start, end, priority))
  ENDFOR
  RETURN schedules
```

After this, the weighted interval scheduling is called. The time complexity of this function is $O(M^2)$ and the space complexity is $O(M)$. Pseudocode is given below:

```
FUNCTION weighted_interval_scheduling(schedules):
  // Sort the schedules by end time O(M log M) time complexity
  sort schedules BY end_time
  num_schedules <- LENGTH(schedules)
  initialize compatible_schedule array with capacity num_schedules
  // Vector to hold previously compatible schedule's index for each schedule
  i <- 1
  j <- 1
  // This nested loop has O(M^2) time complexity but closer to O(M) in reality
  // Since the schedules are sorted by their end times
  FOR i FROM 1 TO num_schedules: // N iterations
    compatible_schedule[i] <- -1 // Initially, no compatible schedule
    current_start_time <- to_minutes(schedules[i].start)
    FOR j FROM i DOWNTO 1: // 1 2 ... N iterations
      previous_end_time <- to_minutes(schedules[j].end)
      IF previous_end_time <= current_start_time:
        compatible_schedule[i] <- j
        BREAK
      ENDIF
    ENDFOR
  ENDFOR
  // Dynamic programming table to store the solution until schedule i
  initialize dynammic_programming array of size num_schedules initialized to 0
  // Base case, the first schedule has the highest priority by itself
  dynammic_programming[0] <- schedules[0].priority
  // Calculate the optimal schedule for schedules until i using the
  // previous compatible schedule
  i <- 1
  FOR i FROM 2 TO num_schedules:
    // Calculate the value of the current schedule
    current_priority <- schedules[i].priority
    // If there is a compatible schedule, add value of the optimal solution
    // until the compatible schedule to the current schedule
    priority_with_current <- current_priority
    IF compatible_schedule[i] != -1 THEN
        priority_with_current <- priority_with_current +
                                 dynammic_programming[compatible_schedule[i]]
    ENDIF
    // Calculate the value without the current schedule
    priority_without_current <- dynammic_programming[i - 1]
    // Select the schedule if it is more valuable
    dynammic_programming[i] <- MAX(priority_with_current,
                                   priority_without_current)
  ENDFOR
  // array to store the optimal schedules
  initialize optimal_schedules as an empty array
  // Backtrack to find the selected schedules
  i <- num_schedules
  // Backtracking has O(M) time complexity since we are checking M schedules at most
  WHILE i > 0:
```

```
    // If the current schedule is selected, add it to the optimal schedules
    // Current schedule is selected if the value of the current schedule
    // is different than the value of the previous schedule
    // Also add the current schedule to the list if the current schedule is the only
    // schedule left (i = 1)
    IF i = 1 OR dynammic_programming[i] != dynammic_programming[i - 1] THEN
      append schedules[i] TO optimal_schedules
      // Move to the previous compatible schedule
      i <- compatible_schedule[i]
    ELSE
      i <- i - 1
    ENDIF
  ENDWHILE
  // Optimal schedules are in the reverse order, sort them to give true output.
  // This has O(M log M) time complexity.
  SORT optimal_schedules BY end_time
  RETURN optimal_schedules
```

The function for finding the weighted interval schedule has the time complexity of $O(M^2)$ due to nested loop for finding the previously compatible interval but if schedules have small intervals then the complexity will come closer to linear time but due to sorting in the best case with short intervals, time complexity would be $O(MlogM)$. Space complexity would be $O(M)$.

By combining the mapping preprocessing and weighted interval scheduling, the overall time complexity would be $O(M^2 + (M + N) \times log^2 N)$ and overall space complexity would be $O(N + M)$, where N standing for the number of rooms and M standing for the number of intervals. Overall code looks like this:

```
FUNCTION find_optimal_schedules_for_each_floor(schedules):
  // Initialize a map to store optimal schedules for each floor
  initialize optimal_schedules as an empty map
  // Iterate through each floor and its schedules
  FOR each floor_schedule IN schedules: // In total, M schedules in schedules
    // O(M^2) time complexity of this operation, M schedules in total among all floors
    floor <- floor_schedule.floor
    floor_schedules <- floor_schedule.schedules
    // Find optimal schedules for the current floor
    optimal_floor_schedules <- weighted_interval_scheduling(floor_schedules)
    // Store the optimal schedules for the current floor
    optimal_schedules[floor] <- optimal_floor_schedules
  ENDFOR
  RETURN optimal_schedules
```

2 Questions & Answers

2.1 Factors Affecting the Performance

For the knapsack problem, if we notate the number of items by N and budget by C then both N and C have a direct effect on runtime since the algorithm has time and space complexities of $O(N \times C)$.

In addition to this, there are other factors affecting the complexity of the knapsack problem, such as the length of item names. Longer names mean more time copying those items, and more time used in the runtime. And also more memory space would be used to store those items in the memory.

And the last factor would affect the runtime is the prices of these items. If item prices are low, then while backtracking, less of a value would be subtracted from the budget, therefore the loop would take longer.

For the time complexity of weighted interval scheduling, if there are M intervals the weighted interval scheduling alone has $O(M^2)$ time complexity but since we need to prepare some preprocessing, constructing the map has $O(Nlog^2 N)$

time complexity if there are N rooms and mapping time intervals with room priorities would have $O(M \times N^2)$ time complexity. then the overall complexity would be $O(M^2 + (M + N) \times log^2 N)$.
In addition to this, if the time intervals are short the nested loop used in the weighted interval scheduling will last less than $O(M^2)$ and sorting those time intervals would start to dominate and time complexity of weighted interval scheduling will reduce to $O(MlogM)$. Overall time complexity would be $O(M \times logM + (M + N) \times log^2 N)$

Weighted interval scheduling uses $O(M + N)$ space complexity. Changing the number of rooms or number of intervals would directly affect the space complexity, as well as it does effects time complexity.

2.2 Differences Between Greedy & Dynamic Programming

Greedy algorithms would consider the answers for the short term, they make choices myopically and expect to get optimal results globally. This approach works for scheduling without weights, but with weights, the previous choices affect future results therefore saving previous choices would be necessary.

The naive approach is to try every possible combination of schedules to get the optimal result, which has $O(2^N)$ time complexity. It is possible to optimize this by remembering previous results. One key observation is that if a time interval ends before another one starts, then they are not compatible. Then we can use this notion to build our solution by looking up the optimal solution ending at time t and adding the current interval's value. With this logic, the dynamic programming method is used.

Greedy solutions would be easier to implement but it is not guaranteed that they will always yield the optimal result. This can also be said for the knapsack problem. The naive approach is to take the items with best value-price raito but there might be some cases where taking a good ratio item with a very large price that does not let us take items anymore would be a corner case. There would be another combination of items that would take more items and utilize all of our budget and the ending total value would be much more.

Because of this, dynamic programming is used for knapsack problems, problems are solved for the subproblems, where there are fewer items and fewer budget. With the dynamic programming approach, it is possible to remember the previously solved subproblems, compare them, and add to the current problem's solution, in this case, whether taking the item or leaving the item as is and using the previous solution with fewer items with the same budget.

3 Implementation Details

3.1 Extra Libraries Used

I have used `map` library to assign priorities of rooms to time intervals. The `map` is a red-black tree file structure, it stores key-value pairs and orders keys using the red-black tree.

In addition, I have used `cmath` library to round floating point numbers.

3.2 Extra Structs

I have defined `Time` struct, which keeps hours and minutes and can convert time into minutes, for ex: 17:50 would yield $17 \times 60 + 50 = 1070$, it is used for comparing two different time intervals or schedules. It also has read into input stream operator `>>` overloaded and write into output stream operator `<<` overloaded to handle times easier with input files and output to the console.

3.3 Extra Functions

I have implemented `print_double` function which returns the single decimal after the point floating point representation of a double as a string. Here, I had to add 0,01 to the double value, to fix the possible floating point errors.

I have also implemented the `read_file` function, which is a template function and reads the appropriate type of data from the input files. It skips the first line and reads the rest of the values by using the `template T`'s overloaded read from input stream operator `>>`. It returns a vector of `template T`s.

3.4 Predefined Structs

`TimeIntervals`, `Item`, `Priority` structs are used to store the information from the input files. I have overloaded read from the input stream operator `>>` of them to use the template function `read_file` function.

Another given predefined struct was `Schedule`, which contained room information, start time, end time, and priority of the room directly. I have used this struct in the weighted interval scheduling function. I have used the map of maps to store the priority of a given room on a given floor and I have mapped the priorities using this map of maps to create `Schedule`s from `TimeIntervals`. It contained its own printing operator `<<` to print the schedules in the specified format.

4 Problems Encountered

4.1 Problem of Multiple Solutions

It is possible to get multiple true solutions that yield the same priority gain or the same total value with a different set of schedules or items. It would depend on the implementation of those algorithms. In my solution, my results were the same as the calico. I didn't have such a problem but it does not mean that I would not get such a problem in the future.

4.2 Floating Point Estimation Errors

One problem that I had was with printing floating point values. When using the `round()` function in C++, sometimes it would round the value to up, and sometimes it would round the value to down. The problem was, in the second test case in Calico, my program would output 36.6 but Calico was expecting 36.7 but the list of items was the same. I found the solution by adding 0.01 to the floating point number to fix this issue. I also implemented the `print_double` function to write the result in the specified format.