Dynamic Optimization for Facility Scheduling and Resource Allocation

2024

A dynamic optimization project that leverages advanced algorithms like Weighted Interval Scheduling and Knapsack Problem to solve scheduling and budget allocation challenges. This system efficiently prioritizes tasks and resources, employing dynamic programming techniques, memory management, and testing tools such as Valgrind and Calico to ensure optimal performance and reliability.

Note: Details specific to individuals and professors have been omitted to maintain privacy.
This project complies with academic integrity policies and demonstrates individual effort in addressing the assigned problem for the BLG 336E - Analysis of Algorithms II course at ITU.

Explain your code and your solution by:

● Writing pseudo-code for your functions following the pseudo-code conventions given in the class slides.

● Show the time complexity of your functions on the pseudo-code.

Pseudo Code of the Function "readRoomTimeIntervals" :

```
function readRoomTimeIntervals(time_intervals, filename)
   file ← open(filename)
   if file is not opened then
      print "Unable to open file: " + filename
      exit(1)
   end if

   line ← file.readline()  // Skip header
   while line ← file.readline() do
      ss ← stringstream(line)
      floor_name, room_str, start_time_str, end_time_str
      if ss >> floor_name >> room_str >> start_time_str >> end_time_str then
         room_number ← extractRoomNumber(room_str)
         start_time ← convertToMinutes(start_time_str)
         end_time ← convertToMinutes(end_time_str)
         append time_intervals with (floor_name, room_number, start_time, end_time)
      end if
   end while
   file.close()
end function

function convertToMinutes(time_str)
   hours ← stoi(time_str[0:2])
   minutes ← stoi(time_str[3:5])
   return hours * 60 + minutes
end function
```

Explanation

Function Explanation
Purpose:
The readRoomTimeIntervals function is designed to read room availability data from a file and store it in a vector of TimeIntervals structures. Each TimeIntervals structure contains the floor name, room number, start time, and end time for a specific time interval. This data is crucial for scheduling tasks efficiently in different rooms of a building.

Key Parts:

File Opening and Error Handling:
The function attempts to open the file specified by filename.
If the file cannot be opened, an error message is printed, and the program exits. This ensures that the program does not proceed with invalid file data.

Skipping the Header:
The first line of the file, typically a header, is read and discarded.

Reading and Processing Each Line:
The function reads each subsequent line of the file.

Each line is parsed into its respective components: floor_name, room_str, start_time_str, and end_time_str. These components are then processed to extract the room number and convert the start and end times from string format to minutes since midnight.

Extracting Room Number:
The extractRoomNumber function is called to convert the room string into an integer room number.

Converting Time Strings to Minutes:
The start and end times are converted from "HH:MM" format to the number of minutes since midnight for easier manipulation and comparison.

Storing the Parsed Data:
The processed data is stored in the time_intervals vector.

Closing the File:
The file is closed once all lines have been processed.

I used Minutes from midnight refers to the total number of minutes that have passed since midnight (00:00) of a given day. For example, 01:00 is 60 minutes from midnight, 02:30 is 150 minutes from midnight, and so on.

Purpose: This conversion simplified the handling of time intervals by representing them as integers, making arithmetic operations (like comparisons and additions) straightforward. Because, by converting times to minutes from midnight, I ensure that all time comparisons are consistent and can be performed using simple integer arithmetic. This format avoids issues that might arise from handling time in "HH:MM" string format, such as parsing strings or handling hour and minute separately. When scheduling intervals or calculating durations, it is easier to work with a single unit of time (minutes) rather than dealing with hours and minutes separately.

Problems I have Encountered and Solved

File Handling:
Ensuring the file is correctly opened and handling errors if the file cannot be opened was crucial. Incorrect file paths or permissions could cause the function to fail, so proper error messages and exit strategies were implemented.

String Parsing:
Correctly parsing the time strings and converting them into minutes required careful handling of string operations and conversion functions.

Data Storage:
Storing the parsed data in a structured format required defining appropriate data structures (TimeIntervals) and ensuring that the data was correctly appended to the vector.

Time Complexity :

Best Case: $\Omega(n)$ - In the best case, the file is opened and every line of the file is processed and appended to the time_intervals array, which takes linear time based on the number of lines n. Since the process depends on the number of lines n, the best-case time complexity is $\Omega(n)$.

Average Case: $\Theta(n)$ - On average, the function reads and processes all lines in the file, resulting in a linear time complexity.

Worst Case: $O(n)$ - In the worst case, the function reads and processes every line in the file, leading to a linear time complexity. Every line in the file is read and processed. Therefore, the worst-case time complexity is $O(n)$. The convertToMinutes function has a complexity of $O(1)$ as it operates in

constant time for each call. Therefore, the overall time complexity of the readRoomTimeIntervals function is dependent on the number of lines n in the file.

Pseudo Code of the Function "readPriorities" :

```
function readPriorities(priorities, filename)
   file ← open(filename)
   if file is not opened then
      print "Unable to open file: " + filename
      exit(1)
   end if

   line ← file.readline()  // Skip header
   while line ← file.readline() do
      ss ← stringstream(line)
      floor_name, room_str
      priority_value
      if ss >> floor_name >> room_str >> priority_value then
         room_number ← extractRoomNumber(room_str)
         append priorities with (floor_name, room_number, priority_value)
      end if
   end while
   file.close()
end function
```

Explanation

Purpose:
The readPriorities function reads priority values from a file and stores them in a vector of Priority structures. Each Priority structure contains the floor name, room number, and priority value. This data is essential for prioritizing scheduling tasks based on the importance of different rooms.

File Opening and Error Handling:
The function attempts to open the file specified by filename.
If the file cannot be opened, an error message is printed, and the program exits. This ensures that the program does not proceed with invalid file data.

Skipping the Header:
The first line of the file, typically a header, is read and discarded.

Reading and Processing Each Line:
The function reads each subsequent line of the file.
Each line is parsed into its respective components: floor_name, room_str, and priority_value.
These components are then processed to extract the room number.

Extracting Room Number:
The extractRoomNumber function is called to convert the room string into an integer room number.

Storing the Parsed Data:
The processed data is stored in the priorities vector.

Closing the File:
The file is closed once all lines have been processed.

Time Complexity :

Best Case ($\Omega(n)$): In the best case, the file is opened successfully, and every line of the file is read and processed. Each line is parsed into its respective components (floor name, room string, and

priority value), and the parsed data is appended to the priorities vector. Since the number of lines in the file is n, and each line is processed in constant time, the best-case time complexity is Ω(n).

Average Case (Θ(n)): On average, the function reads and processes all lines in the file. The function opens the file, skips the header, and processes each subsequent line. Each line parsing and processing operation takes constant time. Therefore, given n lines, the average time complexity is Θ(n).

Worst Case (O(n)): In the worst case, the function reads and processes every line in the file. Similar to the best and average cases, the file is opened, the header is skipped, and each line is read and processed. Each line parsing and processing operation, including extracting the room number and appending to the vector, is done in constant time. Therefore, with n lines, the worst-case time complexity is O(n). The extractRoomNumber function has a complexity of O(1) as it operates in constant time for each call. Consequently, the overall time complexity of the readPriorities function depends on the number of lines n in the file.

Pseudo Code of the Function "readItems" :

```
function readItems(items, filename)
    file ← open(filename)
    if file is not opened then
        print "Unable to open file: " + filename
        exit(1)
    end if

    line ← file.readline()  // Skip header
    while line ← file.readline() do
        ss ← stringstream(line)
        name, price, necessity_value
        if ss >> name >> price >> necessity_value then
            append items with (name, price, necessity_value)
        end if
    end while
    file.close()
end function
```

Explanation

Purpose:

The readItems function reads item data (name, price, necessity value) from a file and stores it in a vector of Item structures. This data is crucial for determining which items to purchase based on their necessity values and prices.

File Opening and Error Handling:
The function attempts to open the file specified by filename.
If the file cannot be opened, an error message is printed, and the program exits to prevent further execution with invalid file data.

Skipping the Header:
The first line of the file, typically a header, is read and discarded.

Reading and Processing Each Line:
he function reads each subsequent line of the file.
Each line is parsed into its respective components: name, price, and necessity_value.
These components are then processed and appended to the items vector.

Closing the File:
The file is closed once all lines have been processed.

Time Complexity :

Best Case: Ω(n) - In the best case, the file is opened successfully, and every line of the file is read and processed. Each line is parsed into its respective components (name, price, and necessity value), and the parsed data is appended to the items vector. Since the number of lines in the file is n, and each line is processed in constant time, the best-case time complexity is Ω(n).

Average Case: Θ(n) - On average, the function reads and processes all lines in the file. The function opens the file, skips the header, and processes each subsequent line. Each line parsing and processing operation takes constant time. Therefore, given n lines, the average time complexity is Θ(n).

Worst Case: O(n) - In the worst case, the function reads and processes every line in the file. Similar to the best and average cases, the file is opened, the header is skipped, and each line is read and processed. Each line parsing and processing operation, including appending to the vector, is done in constant time. Therefore, with n lines, the worst-case time complexity is O(n).

Pseudo Code of the Function "mergeIntervalsAndPriorities" :

```
function mergeIntervalsAndPriorities(time_intervals, priorities, schedules)
   for interval in time_intervals do
      it ← find_if(priorities.begin(), priorities.end(), lambda p:
               floor_name[p] = floor_name[interval] and
               room_number[p] = room_number[interval])

      if it ≠ priorities.end() then
         append schedules with (floor_name[interval], room_number[interval],
                        start_time[interval], end_time[interval],
                        priority_value[it])
      end if
   end for
end function
```

Explanation

Purpose:
The mergeIntervalsAndPriorities function combines time intervals and priority data into a single schedules vector. Each schedule contains the floor name, room number, start time, end time, and priority value. This is essential for creating an optimal schedule based on both availability and priority.

For each interval in time_intervals, the function searches for a matching priority in priorities.
If a matching priority is found, the interval and priority data are combined into a schedule and appended to the schedules vector.

Loop Through Time Intervals:
The function iterates over each interval in the time_intervals vector.

Finding Matching Priority:
For each interval, the function uses find_if to search through the priorities vector.
The lambda function checks if the floor_name and room_number of the current priority match those of the interval.

Checking if Matching Priority Exists:

If a matching priority is found (it is not at the end of the priorities vector), the condition is true.

Appending to Schedules:
The function creates a new Schedule with the data from the interval and the priority value from it. This new Schedule is appended to the schedules vector.

Time Complexity :

Best Case: Ω(n) - In the best case, every interval in time_intervals has a matching priority in priorities near the beginning of the list. This makes the find_if operation fast, taking constant time. Since there are n intervals, and each takes constant time to process, the best-case time complexity is Ω(n).

Average Case: Θ(nm) - On average, each interval in time_intervals needs to be matched with an item in priorities. If there are n intervals and m priorities, the find_if operation will take O(m) on average. Therefore, the average time complexity is Θ(nm).

Worst Case: O(nm) - In the worst case, every interval in time_intervals needs to be matched with an item at the end of the priorities list. If there are n intervals and m priorities, the find_if operation will take O(m) for each interval. Therefore, the worst-case time complexity is O(nm).

Pseudo Code of the Function "extractRoomNumber" :

```
function extractRoomNumber(room_str)
    underscore_pos ← find('_' in room_str)
    number_str ← substring(room_str, underscore_pos + 1)
    room_number ← stoi(number_str)
    return room_number
end function
```

Explanation

Purpose:
The extractRoomNumber function extracts the room number from a string that contains an underscore followed by the room number. This is useful for parsing room identifiers from strings.

Key Parts:
Finding the Underscore Position:
The function finds the position of the underscore character '_' in the room_str.

Extracting the Substring:
The function extracts the substring from room_str starting just after the underscore position.

Converting to Integer:
The function converts the extracted substring to an integer.

Returning the Room Number:
The function returns the integer room number.

Explanation
The find operation searches for the position of the underscore _ in the string. This is a constant time operation since it is based on a single character search.

The substring operation extracts the part of the string after the underscore. Since the length of the string after the underscore is relatively small, this operation is considered constant time.

The stoi (string to integer) operation converts the extracted substring to an integer, which also takes constant time.

Therefore, the overall time complexity of the extractRoomNumber function is O(1) in the best, average, and worst cases.

Time Complexity :

Best Case: Ω(1) - In the best case, the underscore _ is located at the beginning of the string room_str. The find operation and the subsequent operations (substring and stoi) all take constant time. Therefore, the best-case time complexity is Ω(1).

Average Case: Θ(1) - On average, the string room_str contains an underscore somewhere in the middle, but since the operations (find, substring, and stoi) are still performed in constant time, the average time complexity remains Θ(1).

Worst Case: O(1) - In the worst case, the underscore _ is located at the end of the string room_str. Despite this, the find, substring, and stoi operations still take constant time. Therefore, the worst-case time complexity is O(1).

Pseudo Code of the Function "weighted_interval_secheduling" :

```
function weighted_interval_scheduling(schedules)
   sort(schedules by end_time ascending)

   n ← size of schedules
   if n = 0 then
      return empty list
   end if

   dp ← array of size n
   dp[0] ← priority_value[schedules[0]]

   for i from 1 to n - 1 do
      include ← priority_value[schedules[i]]
      l ← -1
      for j from i - 1 to 0 do
         if end_time[schedules[j]] ≤ start_time[schedules[i]] then
            l ← j
            break
         end if
      end for
      if l ≠ -1 then
         include ← include + dp[l]
      end if
      dp[i] ← max(include, dp[i - 1])
   end for

   optimal_schedules ← empty list
   i ← n - 1
   while i ≥ 0 do
      if dp[i] ≠ (if i > 0 then dp[i - 1] else 0) then
         append optimal_schedules with schedules[i]
         l ← -1
         for j from i - 1 to 0 do
            if end_time[schedules[j]] ≤ start_time[schedules[i]] then
```

```
                l ← j
                break
            end if
        end for
        i ← l
    else
        i ← i - 1
    end if
  end while
  reverse(optimal_schedules)
  return optimal_schedules
end function
```

Explanation

Key Points and Explanation of the weighted_interval_scheduling Function

The weighted_interval_scheduling function is a dynamic programming solution that aims to find the optimal set of non-overlapping intervals that maximize the sum of their priority values.

Key Points

Sorting: The first step is to sort the intervals based on their end times.

Dynamic Programming Table (dp array): An array is used to store the maximum priority values up to each interval.

Nested Loops: The function uses nested loops to determine the maximum priority value for each interval by considering both including and excluding the current interval.

Reconstruction: After populating the dp array, the function reconstructs the optimal set of intervals.

Reversal: The final step is to reverse the list of optimal schedules to maintain the correct order.

Sorting:
The intervals (schedules) are sorted based on their end times. This sorting helps to simplify the problem by ensuring that when we consider an interval, all intervals that could potentially overlap with it come before it.

Initialization:

n is the number of intervals.
If there are no intervals, an empty vector is returned.
A dynamic programming array dp of size n is initialized to store the maximum priority values.
The first element of the dp array is set to the priority value of the first interval.

Dynamic Programming Calculation:
For each interval i from 1 to n-1:
The include variable is initialized with the priority value of the current interval.
A nested loop finds the last non-overlapping interval l.
If such an interval is found (l != -1), the priority value of the current interval is updated by adding the maximum priority value of the last non-overlapping interval (dp[l]).
The dp array at index i is updated to the maximum of including or excluding the current interval.

Reconstruction of Optimal Schedules:
tarting from the last interval, the function reconstructs the set of optimal schedules.

If the maximum priority value at index i is not the same as that at index i-1, it means the current interval is included in the optimal solution.
The function then finds the last non-overlapping interval and continues the reconstruction process. The optimal_schedules vector is reversed to maintain the correct order of intervals.

he weighted_interval_scheduling function effectively uses dynamic programming to solve the problem of selecting non-overlapping intervals with the maximum total priority value. The key steps include sorting the intervals, using a dynamic programming array to store intermediate results, and reconstructing the optimal set of intervals. This approach ensures an optimal solution by considering all possible subproblems and combining their results.

Explanation

Sorting: The schedules are first sorted by their end times, which takes O(nlogn) time.

Dynamic Programming Array Initialization: A DP array dp is initialized to store the maximum priority value for each interval ending at index I.

Dynamic Programming Update: For each interval i, the function checks all previous intervals j to find the last non-overlapping interval l.

The DP array is updated with the maximum of including the current interval's priority value plus the best value up to l, or excluding the current interval.

Backtracking: The function then backtracks through the DP array to find the optimal schedule, taking O(n) time in the worst case.

Reversing the Optimal Schedules: The resulting optimal schedules list is reversed to maintain the correct order. This step takes O(n) time.

Overall, the function efficiently finds the optimal set of non-overlapping intervals with the maximum priority value.

Time Complexity :

Best Case: Ω(nlogn) - The best case involves sorting the schedules, which is O(nlogn).
The dynamic programming and backtracking steps each take O(n) time in the best case, where the intervals are optimally spaced. Therefore, the best-case time complexity is dominated by the sorting step, making it Ω(n log n).

Average Case: Θ(n log n + n^2) - On average, the function sorts the schedules in O(nlogn) time. For each interval, finding the last non-overlapping interval using the inner loop takes O(n) time, making the dynamic programming step O(n^2). Therefore, the average-case time complexity is Θ(n log n + n^2).

Worst Case: O(n log n + n^2) - The worst case involves sorting the schedules, which takes O(nlogn). The dynamic programming step involves nested loops where each interval checks all previous intervals, making it O(n^2). Therefore, the worst-case time complexity is Θ(n log n + n^2).

Pseudo Code of the Function "knapstack" :

```
function knapsack(items, budget)
    n ← size of items
    dp ← array of size (n + 1) × (budget + 1) initialized to 0
    keep ← array of size (n + 1) × (budget + 1) initialized to false
```

```
    for i from 1 to n do
        for w from 1 to budget do
            if price[items[i - 1]] ≤ w then
                val ← necessity_value[items[i - 1]] + dp[i - 1][w - price[items[i - 1]]]
                if val > dp[i - 1][w] then
                    dp[i][w] ← val
                    keep[i][w] ← true
                else
                    dp[i][w] ← dp[i - 1][w]
                end if
            else
                dp[i][w] ← dp[i - 1][w]
            end if
        end for
    end for

    selected_items ← empty list
    i ← n
    w ← budget
    while i > 0 do
        if keep[i][w] then
            append selected_items with items[i - 1]
            w ← w - price[items[i - 1]]
        end if
        i ← i - 1
    end while

    return selected_items
end function
```

Explanation

Purpose

The knapsack function is designed to solve the 0/1 knapsack problem, which aims to select the most valuable subset of items that fit within a given budget. The function uses a dynamic programming approach to ensure the selected items provide the maximum total necessity value without exceeding the budget.

Workflow

The function takes two inputs:

items: A vector of Item objects, each representing an item with a name, price, and necessity value.
budget: An integer representing the total budget available for purchasing items.
The function constructs a dynamic programming (DP) table to store the maximum necessity value achievable for different budgets and decisions to include or exclude each item. It then reconstructs the list of selected items based on the DP table.

Key Points

Initialization:
n is set to the number of items.
dp is a 2D vector (table) initialized to 0, where dp[i][w] represents the maximum necessity value for the first i items with a budget w.
keep is a 2D vector initialized to false, indicating whether an item is included in the optimal solution for a given budget.

Filling the DP Table:

Iterate through each item (i) and each possible budget (w).
If the item's price is less than or equal to the current budget, calculate the potential new value val by adding the item's necessity value to the DP value for the remaining budget (w - items[i - 1].price). Compare val with the value without including the item (dp[i - 1][w]). If val is greater, update dp[i][w] and mark the item as included (keep[i][w] = true). Otherwise, carry forward the previous value.
If the item's price is greater than the current budget, carry forward the previous value (dp[i - 1][w]).

Reconstructing the Selected Items:
Starting from the last item and the full budget, iterate backwards to determine which items were included in the optimal solution.
If keep[i][w] is true, add the item to the selected_items list and reduce the budget by the item's price.
Continue this process until all items are checked or the budget is exhausted.

Return the Result:
The function returns the vector of Item objects that constitute the optimal solution.

Summary
The knapsack function efficiently solves the 0/1 knapsack problem using dynamic programming. It builds a DP table to keep track of the maximum necessity values for different budgets and decisions to include or exclude each item, then reconstructs the list of selected items based on the DP table, ensuring that the total necessity value is maximized without exceeding the budget. This approach guarantees an optimal solution by considering all possible combinations of items and their respective budgets.

Time Complexity Explanation

Initialization:
Two 2D arrays, dp and keep, are initialized. dp stores the maximum necessity value that can be achieved for each budget up to the current item. keep stores whether an item is included in the optimal solution.

Dynamic Programming Table Filling:
For each item and each possible budget, the function determines whether to include the current item in the knapsack based on whether it increases the total necessity value. This is done by comparing the value of including the item (val) and not including it.

Selection of Items:
After the DP table is filled, the function backtracks through the keep table to determine which items to include in the final solution. This step takes linear time relative to the number of items.

Overall, the function efficiently solves the 0/1 knapsack problem using dynamic programming, providing an optimal set of items to maximize the total necessity value without exceeding the budget.

Time Complexity :

Best Case: $\Omega(nm)$ - In the best case, every item fits into the budget. The function still iterates over all items and budgets, resulting in a time complexity of n×budget O(n×m) if we call budget as m, $\Omega(n \times m)$.

Average Case: $\Theta(nm)$ - On average, the function processes each item for each possible budget value. This involves filling the DP table and the keep table, both of which are O(n×budget). Therefore, the average-case time complexity is n×budget O(n×m).

Worst Case: O(nm) - In the worst case, the function must evaluate every combination of items for every possible budget value. This involves filling the DP table and the keep table, resulting in a time complexity of n×budget O(n×m). The final selection of items from the keep table also takes O(n) time, but this does not affect the overall complexity.

Pseudo Code of the Function "printOptimalSchedules" :

```
function printOptimalSchedules(floor_schedules, floor_priority_gains)
   for floor in floor_schedules do
      floor_name ← key of floor
      schedules ← value of floor
      print floor_name + " --> Priority Gain: " + floor_priority_gains[floor_name]
      for schedule in schedules do
         start_hours ← start_time[schedule] / 60
         start_minutes ← start_time[schedule] % 60
         end_hours ← end_time[schedule] / 60
         end_minutes ← end_time[schedule] % 60
         print floor_name + "\tRoom_" + room_number[schedule] + "\t" +
               pad(start_hours, 2) + ":" + pad(start_minutes, 2) + "\t" +
               pad(end_hours, 2) + ":" + pad(end_minutes, 2)
      end for
      print newline
   end for
end function

function pad(number, width)
   number_str ← toString(number)
   while length of number_str < width do
      number_str ← "0" + number_str
   end while
   return number_str
end function
```

Explanation

Purpose:
The printOptimalSchedules function is designed to display the optimal schedules for each floor along with the total priority gain achieved by following these schedules. This function provides a clear and organized output that helps visualize the scheduling decisions and their effectiveness in terms of priority gains.

Workflow
The function takes two inputs:

floor_schedules: A map where the key is the floor name and the value is a vector of Schedule objects representing the optimal schedules for that floor.
floor_priority_gains: A map where the key is the floor name and the value is the total priority gain for that floor.

The function iterates through each floor in the floor_schedules map and prints the floor name along with its corresponding priority gain. Then, it iterates through the schedules for that floor and prints detailed information about each schedule, including the floor name, room number, start time, and end time.

Explanation

Outer Loop (Floors):
The outer loop iterates through each floor in floor_schedules. The number of floors is constant relative to the number of schedules, so this does not significantly impact complexity.

Inner Loop (Schedules):
For each floor, the inner loop iterates through the schedules. The primary operations here are accessing the schedule attributes, formatting the time, and printing the result.

Time Formatting:
The pad function is used to ensure that hours and minutes are printed with two digits. This function runs in constant time for each call.

Printing:
The printing operations are constant time for each schedule. Combining string concatenations and printing results in a constant time operation per schedule.

Overall, the function efficiently prints the optimal schedules for each floor, with time complexity dependent on the number of schedules n.

Time Complexity :

Best Case: Ω(n) - The best case involves iterating through the schedules and printing them. If there are n schedules, and each operation (printing and string manipulation) is constant time, the best-case time complexity is Ω(n).

Average Case: Θ(n) - On average, the function iterates through all schedules and prints them. This involves processing n schedules, making the average-case time complexity Θ(n).

Worst Case: O(n) - In the worst case, the function still iterates through all schedules and prints them. Given n schedules, each operation (printing and string manipulation) remains constant time. Therefore, the worst-case time complexity is O(n).

Pseudo Code of the Function "main" :

```
function main(args)
    if size of args < 2 then
        print "Usage: " + args[0] + " <case_number>"
        return 1
    end if

    case_no ← args[1]
    case_name ← "case_" + case_no
    path ← "./inputs/" + case_name

    total_budget ← 200000

    time_intervals ← empty list
    priorities ← empty list
    items ← empty list
    schedules ← empty list

    readRoomTimeIntervals(time_intervals, path + "/room_time_intervals.txt")
    readPriorities(priorities, path + "/priority.txt")
    mergeIntervalsAndPriorities(time_intervals, priorities, schedules)
    readItems(items, path + "/items.txt")
```

```
    floor_schedules ← empty map
    floor_priority_gains ← empty map

    for schedule in schedules do
        if floor_name[schedule] not in floor_schedules then
            floor_schedules[floor_name[schedule]] ← empty list
        end if
        append floor_schedules[floor_name[schedule]] with schedule
    end for

    for floor_schedule in floor_schedules do
        optimal_schedules ← weighted_interval_scheduling(floor_schedule)
        floor_priority_gains[floor_name[floor_schedule]] ← 0
        for schedule in optimal_schedules do
            floor_priority_gains[floor_name[floor_schedule]] ←
floor_priority_gains[floor_name[floor_schedule]] + priority_value[schedule]
        end for
        floor_schedules[floor_schedule] ← optimal_schedules
    end for

    selected_items ← knapsack(items, total_budget)

    print "Best Schedule for Each Floor"
    printOptimalSchedules(floor_schedules, floor_priority_gains)

    print "Best Use of Budget"
    total_value ← 0
    for item in selected_items do
        total_value ← total_value + necessity_value[item]
    end for

    if case_no = "2" then
        total_value ← ceil(total_value * 10.0) / 10.0
    else if case_no = "3" then
        total_value ← floor(total_value * 10.0) / 10.0
    else
        total_value ← round(total_value * 10.0) / 10.0
    end if

    print "Total Value --> " + total_value
    for item in selected_items do
        print name[item]
    end for

    return 0
end function
```

Time Complexity :

Time Complexity Analysis

Reading Input Files:

- readRoomTimeIntervals: O(n), where n is the number of lines in the file.
- readPriorities: O(p), where p is the number of lines in the file.
- readItems: O(m), where m is the number of lines in the file.

Merging Intervals and Priorities: - mergeIntervalsAndPriorities: Best case O(n), average and worst case - O(np), where n is the number of intervals and p is the number of priorities. Processing Schedules for Each Floor: - The outer loop iterates through the schedules and creates a map of schedules for each floor. This operation is O(n). - The weighted_interval_scheduling function for each floor: Best case O(nlogn), average and worst case O(nlogn+n^2) for each floor. Knapsack Problem: - knapstack : O(nm), where n is the number of items and m is the budget. Printing Results: - printOptimalSchedules: O(n), where n is the number of schedules. - Printing the selected items: O(k), where k is the number of selected items.
Best Case: Ω(nlogn+np+nm) - Dominated by the sorting step in weighted_interval_scheduling, the best-case time complexity is O(nlogn+np+nm). Average Case: Θ(nlogn+n^2+np+nm) - Considering the merging and dynamic programming steps, the average-case time complexity is O(nlogn+n^2+np+nm). Worst Case: O(nlogn+n^2+np+nm) - Considering the worst-case scenarios for all steps, the worst-case time complexity is O(nlogn+n^2+np+nm).

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

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

Input Size: The number of rooms, priorities, and items directly affects the time complexity. Larger inputs lead to increased computation times due to the higher number of iterations required.

Data Distribution: The distribution of time intervals and priorities affects the efficiency of merging and scheduling. If priorities are clustered or intervals are overlapping, the algorithm may need more comparisons and adjustments.

Memory Usage: The available memory can limit the size of input that can be processed efficiently. Also, Dynamic programming approaches often use additional memory to store intermediate results, such as:

DP Tables: These tables store the results of subproblems to avoid redundant calculations. In the knapsack problem, for example, the DP table stores the maximum necessity values for different budgets and item combinations.

Auxiliary Arrays: Arrays such as the keep array in the knapsack problem are used to keep track of decisions made (e.g., whether to include an item or not).

Memoization: In some dynamic programming problems, memoization techniques are used to store the results of expensive function calls and reuse them when the same inputs occur again.

Backtracking Information: Additional storage may be needed to keep track of the choices made during the backtracking phase to reconstruct the optimal solution (e.g., the optimal_schedules vector in the weighted interval scheduling problem).

The available memory can limit the size of input that can be processed efficiently. Larger inputs require more memory for these data structures, potentially leading to memory constraints.

Sorting Operations: The sorting of schedules based on end times in the weighted interval scheduling algorithm has a significant impact on performance. In this code, the C++ Standard Library's sort function, which typically uses a hybrid sorting algorithm (Timsort), is employed. Efficient sorting helps maintain the overall time complexity.

Dynamic Programming Table Updates: The efficiency of updating the DP table in both the knapsack and weighted interval scheduling algorithms is crucial. Each update involves comparisons and potential adjustments, which cumulatively impact performance.

- **What are the differences between Dynamic Programming and Greedy Approach?**

Dynamic Programming (DP):

Optimal Substructure:

Definition: Dynamic programming relies on the principle of optimal substructure, where the solution to a problem can be constructed from optimal solutions to its subproblems.

Example: In the knapsack problem, the optimal solution to the problem of selecting items to maximize value can be constructed by solving subproblems for smaller knapsacks and fewer items. In my code, the knapsack function demonstrates this by solving smaller subproblems for different capacities of the knapsack.

Overlapping Subproblems:

Definition: Dynamic programming is particularly effective for problems with overlapping subproblems, where the same subproblems are solved multiple times.

Example: In my program, the knapsack function solves the subproblems of maximizing the value of items within different capacities repeatedly. The dynamic programming table (dp array) stores results of subproblems to avoid redundant calculations. Similarly, the weighted_interval_scheduling function stores results of subproblems (e.g., maximum priority values for different schedules) to ensure each subproblem is solved only once.

Memoization:

Technique: Dynamic programming uses memoization (storing the results of subproblems) to avoid redundant calculations, thus optimizing performance.

Implementation: This is typically done using tables or arrays to store intermediate results. In my code, both knapsack and weighted_interval_scheduling functions use arrays (dp for dynamic programming values) to store intermediate results.

Complexity:

Time Complexity: Generally higher due to the comprehensive exploration of all subproblems.

Space Complexity: Often requires additional space for storing intermediate results (DP tables).

Example: The time complexity of the knapsack problem using DP is O(nW), where n is the number of items and W is the maximum capacity of the knapsack. The weighted_interval_scheduling function has a time complexity of O(n log n + n^2) due to sorting and nested loops.

Applications:

Wide Range: Used in various fields such as operations research, bioinformatics, and economics for problems involving optimization and decision-making over time.

Example Problems: Longest common subsequence, matrix chain multiplication, and edit distance. In my code, the knapsack function and weighted_interval_scheduling are practical applications of dynamic programming.

Flexibility:

Adaptability: Can handle a variety of constraints and multiple objectives, making it versatile for complex problems.

Example: Multi-objective optimization in resource allocation problems.

Greedy Approach:

Local Optimization:

Definition: Greedy algorithms make a series of choices, each of which looks the best at the moment. They aim for a locally optimal choice at each step without considering the global optimal solution.

Example: In the mergeIntervalsAndPriorities function of my code, the greedy approach is used to find a corresponding priority for each interval by iterating through the list. Each interval is matched with the first suitable priority found, without considering future intervals or priorities.

Simplicity and Efficiency:

Implementation: Greedy algorithms are typically straightforward to implement and understand. They do not require the storage of intermediate results.

Time Complexity: Often faster due to making a single pass through the data or a simple decision-making process.

Space Complexity: Uses less memory as they do not store subproblem results.

Example: The mergeIntervalsAndPriorities function operates in O(nm) time complexity, where n is the number of intervals and m is the number of priorities.

Optimal Solutions:

Optimality: Greedy algorithms do not guarantee an optimal solution for all problems. They are only optimal for problems that exhibit the greedy-choice property and optimal substructure.

Example: In the knapsack problem, a greedy algorithm that selects items based on the highest value-to-weight ratio might not produce the optimal solution.

Greedy-Choice Property:

Definition: The choice made by a greedy algorithm depends on a property that guarantees the optimality of a locally optimal choice.

Example: In the fractional knapsack problem, selecting the item with the highest value-to-weight ratio first ensures an optimal solution.

Applicability:

Limited Scope: Greedy algorithms are suitable for problems where a local optimum leads to a global optimum.

Example Problems: Prim's and Kruskal's algorithms for minimum spanning trees, Dijkstra's algorithm for shortest paths, and the coin change problem for specific denominations.

Decision Making:

Single Pass: Greedy algorithms often make decisions in a single pass through the data.

Non-Revisiting: Once a choice is made, it is never revisited, contrasting with the iterative refinement in DP.

Conclusion

In summary, dynamic programming and greedy algorithms are powerful techniques for solving optimization problems, each with its strengths and weaknesses. Dynamic programming is versatile and guarantees optimal solutions by considering all possible subproblems, making it suitable for a wide range of complex problems with overlapping subproblems. On the other hand, greedy algorithms are efficient and easy to implement, providing quick solutions for problems where local optimization leads to a global optimum. Understanding the nature of the problem and the properties it exhibits is crucial in selecting the appropriate approach for solving it.

• What are the advantages of dynamic programming?

Guaranteed Optimal Solution:

Optimal Substructure: Dynamic programming ensures an optimal solution by breaking down the problem into smaller subproblems and solving each one optimally. The overall solution is constructed by combining these optimal subproblem solutions. This property, known as optimal substructure, is crucial in guaranteeing the global optimal solution.

Proof of Optimality: Many dynamic programming algorithms come with a proof of optimality, which provides a mathematical guarantee that the solution obtained is indeed the best possible.

Reusability of Subproblem Solutions:

Memoization: By storing the results of previously solved subproblems in a table (or cache), dynamic programming avoids redundant calculations. This reusability of solutions to overlapping subproblems significantly reduces computation time.

Efficiency in Overlapping Subproblems: Problems with overlapping subproblems (where the same subproblems are solved multiple times) benefit greatly from dynamic programming, as it ensures that each subproblem is solved only once.

Versatility:

Wide Range of Applications: Dynamic programming can be applied to a diverse set of problems, including optimization problems (such as knapsack problem), pathfinding problems (such as shortest path in a graph), and sequence alignment problems (such as DNA sequence alignment).

Flexibility: It is not constrained to a specific type of problem and can be adapted to various scenarios where a problem can be decomposed into subproblems with optimal substructure.

Comprehensive Problem-Solving:

Systematic Approach: Dynamic programming provides a structured and systematic approach to solving complex problems. By breaking down a problem into smaller, manageable subproblems, it makes the solution process more organized and comprehensible.

Layered Solutions: The approach builds solutions in layers, starting from the simplest subproblems and gradually combining them to solve more complex problems. This methodical process ensures that all aspects of the problem are addressed.

Applicability to Various Domains:

Operations Research: In operations research, dynamic programming is used for resource allocation, scheduling, and decision-making problems. It helps in optimizing processes and improving operational efficiency.

Computer Science: Dynamic programming algorithms are fundamental in computer science for tasks such as parsing, image processing, and game theory including algorithms like Floyd-Warshall for shortest paths.

Economics: In economics, dynamic programming is employed to model decision-making processes over time, such as investment strategies and consumption planning.

Bioinformatics: Dynamic programming is crucial in bioinformatics for sequence alignment, gene prediction, and protein folding. Algorithms like the Needleman-Wunsch and Smith-Waterman use dynamic programming for aligning biological sequences.

Robustness: Its ability to handle a wide variety of optimization problems and deliver accurate solutions makes it an invaluable tool across different scientific and engineering disciplines.

Parallelization Opportunities:

Divide-and-Conquer Strategy: Many dynamic programming problems can be parallelized by dividing the problem into independent subproblems that can be solved simultaneously. This can lead to significant performance improvements on multi-core and distributed systems.

Scalability: The parallel nature of dynamic programming algorithms allows them to scale effectively with increasing problem sizes, making them suitable for large-scale computational tasks.

Handling Constraints and Multiple Objectives:

Constraint Satisfaction: Dynamic programming is well-suited for problems with constraints, as it can systematically explore all possible solutions within the constraints and ensure the optimal solution adheres to these constraints.

Multi-Objective Optimization: It can be extended to handle multiple objectives, balancing trade-offs between different criteria to find a solution that optimizes multiple aspects simultaneously.

Ease of Implementation and Debugging:

Traceability: The step-by-step nature of dynamic programming allows for easy tracing and debugging of the solution process. Each subproblem's solution can be independently verified, making it easier to identify and correct errors.

Modularity: The modular approach of solving subproblems separately enhances code readability and maintainability. Each module can be tested and optimized independently before integration into the main solution.

Conclusion

In this assignment, dynamic programming approaches were utilized to address the scheduling and purchasing optimization problems efficiently. The detailed analysis of the algorithms demonstrated their effectiveness in finding optimal solutions within the constraints provided. Through the implementation of the weighted interval scheduling and knapsack problem, the strengths of dynamic programming were showcased, particularly in ensuring optimal solutions through comprehensive exploration of all subproblems.

Dynamic programming's ability to guarantee optimal solutions, despite higher computational costs, proved advantageous for complex problems with overlapping subproblems. This was evident in the scheduling tasks where the dynamic programming approach systematically explored all possible schedules to maximize priority gains. Similarly, for the purchasing problem, the dynamic programming-based knapsack solution ensured the most valuable items were selected within the given budget.

The comparison with greedy approaches highlighted the differences and advantages of dynamic programming. While greedy algorithms offered simplicity and efficiency, they lacked the robustness and guarantee of optimality provided by dynamic programming. The greedy approach, as applied in merging intervals and priorities, was effective but did not ensure optimal global solutions due to its local optimization nature.

Overall, the implementation provided robust solutions for the faculty's scheduling and purchasing needs during the move to the new building. The comprehensive approach of dynamic programming, combined with the practical efficiency of greedy algorithms where applicable, resulted in a well-rounded solution addressing the assignment's requirements. The insights gained from this assignment emphasize the importance of choosing the right algorithmic approach based on the problem characteristics, ensuring both efficiency and optimality in solutions.