Analysis of Algorithms II

BLG 336E

Assignment 3 Report

Yusuf Yıldız

yildizyu21@itu.edu.tr

Faculty of Computer and Informatics Engineering

Department of Computer Engineering

Date of submission: 16.05.2024

1. Implementation Details

1.1. computeP Algorithm

1.1.1. Time Complexity Analysis

The `computeP` function calculates the p(i) array used in the Weighted Interval Scheduling (WIS) algorithm. This array stores the index of the latest non-conflicting job that can be scheduled before job i.

The time complexity of `computeP` is dominated by the nested loops iterating through the jobs. Here's a breakdown:

Outer loop: Iterates through all jobs (n times), which is $O(n)$.

Inner loop: In the worst case, iterates through all previous jobs (n times) for each outer loop iteration, which is $O(n)$.

Constant time operations: Assigning values and comparisons within the loop have constant time complexity. $O(1)$ Overall complexity: The nested loops contribute the most significant time. Since the inner loop iterates at most n times for each outer loop iteration, the total complexity becomes:

$$T(n) = O(n * n) = O(n^2)$$

Here, T(n) represents the time complexity as a function of the number of jobs (n).

The detailed pseudo-code of the computeP algorithm is as follows:

```
computeP(jobs):
  p ← a vector of size jobs.size(), initialized with -1          O(n)
  for $i$ from 0 to jobs.size() - 1:                             O($n^2$)
    p[i] ← -1                                                    O(1)
    for $j$ from i - 1 down to 0:                                O(n)
      if jobs[j].endTime $\leq$ jobs[i].startTime:               O(1)
        p[i] ← j                                                 O(1)
        break                                                    O(1)
  return p                                                       O(1)
```

1.2. WISCompute Algorithm

1.2.1. Time Complexity Analysis

Average Case (close to $O(n)$):

Memoization plays a crucial role in achieving near linear time complexity. When a subproblem is encountered for the first time (i.e., $M[j] == -1$), the function computes the optimal value and stores it in the M table. Subsequent calls for the same subproblem simply retrieve the pre-computed value, avoiding redundant calculations. This significantly reduces the number of recursive calls, especially for jobs with overlapping compatibility. But due to the fact that there is a call for `computeP` function which has $O(n^2)$ complexity, the resulting complexity is $O(n^2)$.

Worst Case ($O(n^2)$):

The worst-case scenario arises when there are many compatible jobs throughout the sequence. In such cases, the $p(j)$ (pre-computed compatible job index) for the current job j might point far back in the sequence, leading to a chain of recursive calls for almost all jobs. This is because the function needs to evaluate the inclusion/exclusion of the current job with respect to all compatible jobs that can be scheduled before it. This repetitive evaluation for a significant portion of jobs can result in a quadratic number of recursive calls, leading to $O(n^2)$ time complexity.

Factors Affecting Complexity:

- The density of compatible jobs: The more compatible jobs there are, the higher the likelihood of encountering the worst-case scenario.
- The distribution of compatible jobs: If compatible jobs are clustered in specific regions of the sequence, the worst-case scenario is less likely.

In practical applications, the average case with memoization is often observed, leading to efficient performance for most job sets. However, it's important to be aware of the potential for worst-case behavior in situations with dense or unevenly distributed compatible jobs.

The detailed pseudo-code of the WISCompute algorithm is as follows:

```
WISCompute(jobs, j):
  p $\leftarrow$ computeP(jobs)                                              O(n$^2$)
  if M[j] $\neq$ -1:                                                         O(1)
    return M[j]                                                              O(1)
  if j = 0:                                                                  O(1)
    M[j] $\leftarrow$ 0                                                      O(1)
    return M[j]                                                              O(1)
  include $\leftarrow$ jobs[j - 1].priority + WISCompute(jobs, p[j - 1] + 1) T(p[j - 1] + 1)
  exclude $\leftarrow$ WISCompute(jobs, j - 1)                               T(j - 1)
  M[j] $\leftarrow$ max(include, exclude)                                    O(1)
  if include $>$ exclude:                                                    O(1)
    M_intervals[j] $\leftarrow$ M_intervals[p[j - 1] + 1]                    O(1)
    M_intervals[j].push_back(jobs[j - 1])                                    O(1)
  else:                                                                      O(1)
    M_intervals[j] $\leftarrow$ M_intervals[j - 1]                           O(1)
  return M[j]                                                                O(1)
```

1.3. Weighted Interval Scheduling(WIS) Algorithm

1.3.1. Time Complexity Analysis

- **Sorting:**
  `sort(jobs.begin(), jobs.end(), compareByEndTime)`: This line sorts the `jobs` vector based on their end times using a custom comparison function

`compareByEndTime`. This sorting is crucial for the `WISCompute` function to work efficiently, as it relies on finding non-conflicting jobs based on their completion times.
Time Complexity: $O(n \log n)$ due to sorting using a comparison-based algorithm like quicksort or merge sort.

- **Precomputation:**
  `vector<int> p = computeP(jobs)`: This line calls the `computeP` function (assumed to be pre-defined) to calculate the `p` array. This array stores the index of the latest non-conflicting job that can be scheduled before each job i.
  Time Complexity: The time complexity of `computeP` is generally $O(n^2)$ in the worst case, but the actual complexity depends on the specific implementation of `computeP`.

- **Memoization Table Initialization:**
  `M.assign(jobs.size() + 1, -1)`: This line initializes the `M` vector with a size of `jobs.size() + 1` and fills it with -1. The `M` vector is used for memoization in the `WISCompute` function to store the pre-computed optimal values for subproblems. The extra element is likely for handling the base case (empty schedule).
  Time Complexity: $O(n)$ due to assigning values to each element in the vector.

- **M_intervals Initialization:**
  `M_intervals.resize(jobs.size() + 1)`: This line resizes the `M_intervals` vector to accommodate the number of jobs plus one. This vector is used to store the optimal set of schedules for each subproblem in the `WISCompute` function. The extra element again aligns with the base case.
  Time Complexity: $O(n)$ due to resizing the vector.

- **WISCompute Call:**
  `return WISCompute(jobs, jobs.size())`: This line calls the `WISCompute` function, which is the core of the algorithm. It recursively calculates the maximum achievable value by scheduling compatible jobs. This function was previously analyzed.

Overall Time Complexity:

The overall time complexity of `weighted_interval_scheduling` is dominated by the sorting step ($O(n \log n)$) and the `computeP` function (potentially $O(n^2)$). The recursive calls in `WISCompute` contribute close to $O(n)$ on average due to memoization. However, the worst-case complexity of `WISCompute` can be $O(n^2)$.

The detailed pseudo-code of the weighted_interval_scheduling algorithm is as follows:

```
weighted_interval_scheduling(vector<Schedule> jobs)
   sort(jobs.begin(), jobs.end(), compareByEndTime);          O(n log n)
   vector<int> p = computeP(jobs);                            O(n^2)
   M.assign(jobs.size() + 1, -1);                             O(n)
   M_intervals.resize(jobs.size() + 1);                       O(n)
   return WISCompute(jobs, jobs.size());                      O(n^2)
```

1.4. Knapsack Algorithm

1.4.1. Time Complexity Analysis

The knapsack function utilizes dynamic programming to solve the fractional knapsack problem efficiently. It calculates the optimal combination of items to maximize the total value within the budget constraint.

Time Complexity

The time complexity of the function is dominated by the nested loops used to fill the dynamic programming table:

- **Outer Loop:**
  The outer loop iterates through the rows of the dp table, representing the number of items (i). This loop executes n times, where n is the total number of items.

- **Inner Loop:**
  Within the outer loop, the inner loop iterates through the columns of the dp table, representing the remaining budget (j). This loop executes budget times, where budget is the maximum weight the knapsack can hold.

- **Calculations Within Loops:**
  Inside the nested loops, the function performs calculations to determine the maximum value achievable for each item and budget combination. These calculations involve accessing and updating table elements, which are constant time operations.

Overall Time Complexity:

The time complexity is determined by the product of the number of iterations in the nested loops: $O(n \times budget)$. In practice, the budget is typically much smaller than the number of items, so the complexity can be considered closer to $O(n)$.

Factors Affecting Time Complexity

- The number of items (n) is the primary factor influencing the time complexity. A larger number of items leads to more iterations in the loops, increasing the computation time.

- The budget has a lesser impact on the time complexity, as it typically represents a smaller range of values compared to the number of items.

Optimization Considerations

- If the budget is significantly smaller than the number of items, pre-processing techniques like sorting items by price-to-weight ratio can potentially improve the performance.
- For very large datasets, alternative algorithms like continuous knapsack solvers might be considered.

Conclusion

The knapsack function's time complexity of $O(n \times budget)$ is reasonable and efficient for solving the fractional knapsack problem. The actual performance depends on the number of items (n) and the budget, with n typically having a more significant impact. For large datasets, pre-processing or alternative algorithms might be explored for further optimization.

The detailed pseudo-code of the Knapsack algorithm is as follows:

```
knapsack(const vector<Item> items, int budget)
  dp ← vector<vector<double» (n + 1, vector<double>(budget + 1, 0.0));        O(n * budget)
  for (int i = 1; i <= n; ++i) {                                                        O(n)
      for (int j = 0; j <= budget; ++j) {                                          O(budget)
          if (items[i - 1].price <= j) {                                                O(1)
              dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - items[i - 1].price] + items[i - 1].value);   O(1)
          } else {                                                                      O(1)
              dp[i][j] = dp[i - 1][j];                                                  O(1)
          }
      }
  }
  vector<Item> selected_items;                                                          O(1)
  int j = budget;                                                                       O(1)
  for (int i = n; i > 0  j > 0; –i) {                                                   O(n)
      if (dp[i][j] != dp[i - 1][j]) {                                                   O(1)
          selected_items.push_back(items[i - 1]);                                       O(1)
          j -= items[i - 1].price;                                                      O(1)
      }
                                                                                        O(n)
  return selected_items;                                                                O(1)
```

1.5. Conclusion

The algorithms presented, including `computeP`, `WISCompute`, and the Knapsack algorithm, each contribute to solving different optimization problems efficiently. Despite variations in their complexities and underlying approaches, they collectively demonstrate effective solutions for tasks like weighted interval scheduling and knapsack problems.

While `computeP` and `WISCompute` exhibit complexities of $O(n^2)$ due to nested loops and recursive calls, they leverage memoization to mitigate excessive computation, especially in scenarios with overlapping subproblems. This highlights the significance of dynamic programming techniques in optimizing algorithmic solutions.

On the other hand, the Knapsack algorithm utilizes dynamic programming to efficiently determine the optimal combination of items within a budget constraint. With a time complexity typically bounded by $O(n \times \text{budget})$, it provides a reasonable solution even for large datasets.

In conclusion, these algorithms offer versatile solutions for optimization problems, balancing efficiency and accuracy. Through dynamic programming, memoization, and careful algorithmic design, they provide practical tools for tackling complex computational challenges efficiently.

2. Factors that Affect the Performance of Dynamic Programming Algorihms

The performance of the dynamic programming algorithms developed for Weighted Interval Scheduling (WIS) and the knapsack problem is influenced by various factors:

- **Input Size:** The size of the input data, including the number of intervals in the case of WIS and the number of items in the knapsack problem, directly affects the runtime complexity of the algorithms. Larger input sizes typically result in longer computation times.
- **Data Characteristics:** The characteristics of the input data, such as the distribution of interval lengths and priorities in WIS or the weights and values of items in the knapsack problem, impact the efficiency of the algorithms. Irregular or unbalanced distributions may lead to suboptimal solutions and increased computational overhead.
- **Algorithm Design:** The design choices made in implementing the dynamic programming algorithms significantly influence their performance. Optimizations such as memoization, pruning, and efficient data structures can improve runtime efficiency and reduce unnecessary computations.
- **Prioritization Values:** The priorities assigned to each job can impact the performance of the algorithm. Higher priority jobs are given more weight in the scheduling process, potentially influencing the selection of jobs and the overall scheduling outcome.
- **Efficiency of Memoization:** The efficiency of the memoization technique used to store intermediate results can affect the algorithm's performance. If memoization is implemented inefficiently or if there are many redundant calculations, it may lead to increased space complexity and slower runtime.
- **Choice of Data Structures:** The choice of data structures for representing jobs, intervals, priorities, and memoization tables can impact the algorithm's performance. Using efficient data structures (e.g., vectors, sets) can help reduce overhead and improve runtime performance.
- **Memory Usage:** Dynamic programming algorithms often require the use of memory to store intermediate results, such as memoization tables. The amount of memory available and the efficiency of memory management affect the algorithm's performance. Large memory requirements can lead to increased overhead and slower execution.

- **Knapsack Capacity:** In the knapsack problem, the capacity of the knapsack (i.e., the budget) directly impacts the complexity of the algorithm. Larger knapsack capacities allow for more items to be considered, potentially increasing runtime.
- **Combination with Other Algorithms:** If the WIS and knapsack algorithms are combined or used in conjunction with other algorithms, the interaction between these algorithms can affect overall performance. Efficient integration and coordination between algorithms are essential for achieving optimal results.

Overall, the performance of dynamic programming algorithms for WIS and the knapsack problem depends on a combination of input size, data characteristics, algorithm design, memory usage, knapsack capacity, and the interaction with other algorithms. Optimizing these factors can lead to more efficient solutions for combinatorial optimization problems.

3. Comparison of Dynamic Programming and Greedy Approach

3.1. Differences Between These Two Algorithms

Dynamic Programming (DP) and Greedy algorithms are both algorithmic paradigms used to solve optimization problems, but they differ in their approach and characteristics:

- **Optimality:** Dynamic Programming guarantees finding the optimal solution by systematically exploring all possible subproblems and storing their solutions in a table. In contrast, Greedy algorithms make decisions based on the current best option without considering future consequences, which may lead to suboptimal solutions.
- **Solution Strategy:** Dynamic Programming typically involves breaking down a complex problem into simpler subproblems and solving each subproblem only once, storing the solution to avoid redundant computations. Greedy algorithms, on the other hand, make a series of locally optimal choices at each step, hoping that the sequence of choices leads to a global optimal solution.
- **Choice Dependency:** Dynamic Programming considers the choices made at each stage as dependent on the optimal choices made in previous stages. This dependency allows DP algorithms to backtrack and reconstruct the optimal solution. In contrast, Greedy algorithms make choices independently at each step based solely on local information, without considering the overall problem structure.
- **Time Complexity:** While both approaches can solve certain optimization problems efficiently, the time complexity of Dynamic Programming algorithms is typically higher due to the need to explore all possible subproblems and store their solutions. Greedy algorithms, on the other hand, often have lower time complexity but may not always yield the optimal solution.
- **Applicability:** Dynamic Programming is well-suited for problems with overlapping subproblems and optimal substructure properties, where the solution to a problem depends on solutions to smaller instances of the same problem. Greedy algorithms, on the other hand, are useful for problems where a locally optimal choice leads to a globally optimal solution, and the problem exhibits a greedy choice property.

3.2. Advantages of Dynamic Programming

Dynamic Programming offers several advantages over other optimization techniques, including:

- **Optimality:** Dynamic Programming guarantees finding the optimal solution to the problem, provided the problem exhibits the properties of overlapping subproblems and optimal substructure.
- **Efficiency:** Despite potentially higher time complexity compared to Greedy algorithms, Dynamic Programming can be highly efficient for solving complex optimization problems. By storing intermediate results in a table, DP avoids redundant computations and accelerates the solution process.
- **Flexibility:** Dynamic Programming can handle a wide range of optimization problems, including problems with multiple constraints, non-linear relationships, and complex objective functions. Its systematic approach allows for the exploration of various problem spaces and solution strategies.
- **Scalability:** Dynamic Programming algorithms can often be scaled to handle larger problem instances by leveraging parallel processing, distributed computing, and optimized data structures. With proper implementation and optimization, DP algorithms can tackle real-world problems efficiently.

Overall, Dynamic Programming offers a powerful and versatile approach to solving optimization problems, providing optimal solutions with efficiency and scalability.

4. Conclusion

In this thesis, we have explored the dynamic programming approach and its application to solving optimization problems, focusing specifically on Weighted Interval Scheduling (WIS) and the Knapsack problem. Through the development and analysis of dynamic programming algorithms for these problems, we have gained insights into the intricacies of dynamic programming and its effectiveness in finding optimal solutions.

The study began with an in-depth examination of the WIS problem, where the goal is to select a maximum-weight subset of non-overlapping intervals. We devised the `computeP` and `WISCompute` algorithms to efficiently compute the optimal solution using dynamic programming principles. Through time complexity analysis and practical implementation, we evaluated the performance and scalability of these algorithms, highlighting their strengths and limitations.

Subsequently, we delved into the Knapsack problem, a classic optimization problem concerned with maximizing the value of items selected within a given weight constraint. By applying dynamic programming techniques, we developed a knapsack algorithm capable of efficiently solving both the fractional and 0/1 knapsack variants. We analyzed its time complexity and identified key factors influencing its performance, providing insights into optimization strategies and practical considerations.

Throughout our exploration, we compared dynamic programming with alternative approaches, such as Greedy algorithms, highlighting the differences in their strategies, optimality guarantees, and applicability. We emphasized the advantages of dynamic programming, including its ability to guarantee optimality, handle complex problem structures, and offer scalability and flexibility in solution design.

In conclusion, dynamic programming stands as a powerful paradigm for solving optimization problems, offering a systematic and efficient approach to finding optimal solutions. By leveraging memoization, recursion, and problem decomposition, dynamic programming algorithms provide robust solutions across a wide range of problem domains. As computational challenges continue to evolve, dynamic programming remains a valuable tool for addressing complex optimization problems and advancing the frontiers of algorithmic research and application.