Algorithmic Documentation: Pathfinding and Interval Coverage

xAI

May 2025

Mohammad Hassaan Ejaz
22I - 2434
SE - B
Design and Analysis of Algorithms - Assignment 3

Introduction

This document provides a detailed analysis of two C++ algorithms: the **Minimum Cost Grid Path Finder** and the **Minimum Red Interval Coverage**. We discuss the algorithmic approaches, their time and space complexities, the rationale for choosing specific strategies, and a performance analysis. The algorithms leverage dynamic programming and greedy strategies, respectively, to solve pathfinding and interval coverage problems efficiently.

Minimum Cost Grid Path Finder

Problem Description

The Minimum Cost Grid Path Finder computes the minimum-cost path from the top-left to the bottom-right cell in an $m \times n$ grid, allowing only `DOWN` and `RIGHT` moves. Each cell has a cost, and the goal is to minimize the total cost of the path.

Algorithm: Dynamic Programming

We use a **dynamic programming (DP)** approach to solve this problem. The algorithm constructs a DP table where each cell `dp[i][j]` represents the minimum cost to reach position (i, j) from $(0, 0)$. A path-tracking matrix records whether the minimum cost comes from a `DOWN` or `RIGHT` move.

```
Matrix dp(rows, List(cols));
Path from(rows, Directions(cols));
dp[0][0] = m[0][0];
// Initialize first row
for j = 1 to cols-1:
    dp[0][j] = dp[0][j-1] + m[0][j];
    from[0][j] = RIGHT;
// Initialize first column
```

```
for i = 1 to rows-1:
    dp[i][0] = dp[i-1][0] + m[i][0];
    from[i][0] = DOWN;
// Fill DP table
for i = 1 to rows-1:
    for j = 1 to cols-1:
        if dp[i-1][j] < dp[i][j-1]:
            dp[i][j] = dp[i-1][j] + m[i][j];
            from[i][j] = DOWN;
        else:
            dp[i][j] = dp[i][j-1] + m[i][j];
            from[i][j] = RIGHT;
// Reconstruct path
Directions directions;
i = rows-1, j = cols-1;
while i > 0 or j > 0:
    d = from[i][j];
    directions.push_back(d);
    if d == DOWN: i--;
    else: j--;
reverse(directions);
return directions;
```

Time Complexity

- **Initialization**: Filling the first row and column takes $O(n)$ and $O(m)$, respectively. - **DP Table**: Computing `dp[i][j]` for $i=1$ to $m-1$, $j=1$ to $n-1$ takes $O(m \cdot n)$. - **Path Reconstruction**: Backtracking from $(m-1, n-1)$ to $(0,0)$ takes $O(m+n)$. - **Total**: $O(m \cdot n)$, dominated by the DP table computation.

Space Complexity

- **DP Table**: $O(m \cdot n)$ for the `dp` matrix. - **Path Matrix**: $O(m \cdot n)$ for the `from` matrix. - **Output**: $O(m+n)$ for the directions vector. - **Total**: $O(m \cdot n)$.

Rationale for Algorithm Choice

Dynamic programming is ideal because: - The problem exhibits **optimal substructure**: The minimum cost to reach (i, j) depends on the minimum costs to reach $(i-1, j)$ or $(i, j-1)$. - It has **overlapping subproblems**: Computing `dp[i][j]` reuses previously computed values. - A greedy approach (e.g., always choosing the smaller adjacent cost) fails to guarantee optimality, as it may lead to suboptimal paths. - Alternatives like Dijkstra's algorithm are overkill, with a complexity of $O(mn \log(mn))$, since the grid structure and limited moves (only `DOWN` and `RIGHT`) allow a simpler DP solution.

Analysis

- **Efficiency**: The $O(m \cdot n)$ time complexity is optimal for computing the minimum cost across all cells, as each cell must be considered. - **Robustness**: The algorithm handles

arbitrary grid sizes and costs, including negative values, as long as the grid is well-formed. - **Limitations**: The space complexity of $O(m \cdot n)$ can be improved to $O(n)$ by storing only the previous row of the DP table, but this optimization was not implemented to prioritize path reconstruction clarity. - **Use Case**: Suitable for grid-based pathfinding in games, robotics, or network routing where only orthogonal moves are allowed.

Minimum Red Interval Coverage

Problem Description

Given two sets of intervals (`red` and `blue`), the goal is to find the minimum number of `red` intervals that fully cover each `blue` interval. An interval $[x, y)$ covers another interval $[a, b)$ if $x \leq a$ and $y \geq b$.

Algorithm: Greedy Strategy

We use a **greedy algorithm** that sorts both `red` and `blue` intervals by start time and iteratively selects the `red` interval with the furthest end time that covers each `blue` interval.

```
_Timeline red = sort(_red);
_Timeline blue = sort(_blue);
int i = 0, count = 0;
_Timeline resultant;
for each b in blue:
    start_b = b.x, end_b = b.y;
    best_end = -1;
    while i < red.size() and red[i].x <= start_b:
        if red[i].y >= end_b:
            best_end = max(best_end, red[i].y);
        i++;
    if best_end >= end_b:
        resultant.push_back(red[i-1]);
        count++;
return {count, resultant};
```

Time Complexity

- **Sorting**: Sorting `red` and `blue` intervals takes $O(n \log n)$ and $O(m \log m)$, where n and m are the sizes of `red` and `blue` timelines. - **Iteration**: Processing each `blue` interval involves scanning `red` intervals, with a total of $O(n)$ across all iterations (since `i` never decreases). - **Total**: $O(n \log n + m \log m)$, dominated by sorting.

Space Complexity

- **Input Storage**: $O(n+m)$ for sorted `red` and `blue` intervals. - **Output**: $O(m)$ for the `resultant` timeline in the worst case. - **Total**: $O(n+m)$.

Rationale for Algorithm Choice

The greedy strategy is optimal because: - The problem resembles the **interval scheduling** problem, where selecting the interval with the furthest end time ensures maximum coverage per selection. - Sorting by start time ensures we consider all `red` intervals that could cover a `blue` interval's start. - A dynamic programming approach would be unnecessarily complex, as the greedy choice property holds: For each `blue` interval, choosing the `red` interval with the maximum end time among valid candidates minimizes the number of intervals needed. - Brute-force enumeration of all `red` interval combinations is infeasible, with a complexity of $O(2^n)$.

Analysis

- **Efficiency**: The $O(n \log n + m \log m)$ complexity is near-optimal, as sorting is necessary to ensure correct interval ordering. - **Correctness**: The greedy choice guarantees the minimum number of intervals, as proven in interval coverage literature (e.g., similar to the interval scheduling maximization problem). - **Limitations**: The algorithm assumes intervals are well-formed (i.e., $x \leq y$). It may fail to cover all `blue` intervals if no valid `red` intervals exist, but this is handled by skipping uncovered intervals. - **Use Case**: Applicable in scheduling, resource allocation, or network coverage problems where one set of resources must cover another.

Conclusion

Both algorithms are well-suited to their respective problems: - The **Minimum Cost Grid Path Finder** uses dynamic programming to efficiently compute the optimal path in a grid, balancing time and space complexity. - The **Minimum Red Interval Coverage** employs a greedy strategy to minimize the number of intervals, leveraging sorting for efficiency. These implementations demonstrate the power of tailored algorithmic strategies in solving pathfinding and coverage problems, with clear applications in real-world scenarios.