Lab 4 : Scheduling to maximize profit

Course: CS20009.04 Data Structure
Name: Yijia Chen
Student Number: 24300240127
Date: November 25, 2025

Implementation

The aim is to maximize the profit of a set of jobs carried out in limited time. To obtain profit, each job selected should be completed before its deadline.

The problem can be solved by dynamic programming. It exhibits the property of optimal substructure. Consider job j with processing time t_j, a profit p_j, and a deadline d_j. If the job is included in the schedule that yield the most profit, then the schedule also contains the optimal schedule before the starting time of job j, $t - t_j$, where t is the time when job j is done. Otherwise, we can copy-and-paste a better subschedule to fit in the time before $t - t_j$ and obtain a greater profit by time t.

Therefore, we use the vector `dp` to memoize the maximum profit of a schedule that can be completed before time t for all $0 \leq t \leq \max\{d_j\}$. The scale of the problem correspond to the time t. The solution to a larger problem can be obtained by considering the subproblem, whose answer has been recorded in the vector `dp`.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <sstream>

using namespace std;

struct Job {
    int id;
    int processing_time;
    int profit;
    int deadline;
};

pair<int, vector<Job>> findMaxProfitSchedule(vector<Job>& jobs) {
    if (jobs.empty()) return {0, {}};

    // Sort jobs by deadline
    sort(jobs.begin(), jobs.end(),
         [](const Job& a, const Job& b) { return a.deadline < b.deadline; });

    int n = jobs.size();
    int max_deadline = 0;
    for (const auto& job : jobs) {
        max_deadline = max(max_deadline, job.deadline);
    }

    // DP arrays
    vector<int> dp(max_deadline + 1, 0);
    vector<vector<pair<int, int>>> decision(n, vector<pair<int, int>>(max_deadline + 1, {-1, -1}));

    // Dynamic programming
    for (int i = 0; i < n; i++) {
        int t_j = jobs[i].processing_time;
        int p_j = jobs[i].profit;
        int d_j = jobs[i].deadline;

        for (int t = d_j; t >= t_j; t--) {
            if (dp[t] < dp[t - t_j] + p_j) {
                dp[t] = dp[t - t_j] + p_j;
                decision[i][t] = {t - t_j, i};
            }
        }
    }

    // Find maximum profit
    int max_profit = 0;
    int max_time = 0;
    for (int t = 0; t <= max_deadline; t++) {
        if (dp[t] > max_profit) {
            max_profit = dp[t];
```

```
            max_time = t;
        }
    }

    // Reconstruct schedule
    vector<Job> schedule;
    int current_time = max_time;
    for (int i = n - 1; i >= 0; i--) {
        if (decision[i][current_time].first != -1) {
            schedule.push_back(jobs[i]);
            current_time = decision[i][current_time].first;
        }
    }
    reverse(schedule.begin(), schedule.end());

    return {max_profit, schedule};
}

// Parse input string into jobs
vector<Job> parseInput(const string& input) {
    vector<Job> jobs;
    stringstream ss(input);
    string line;
    int job_id = 1;

    while (getline(ss, line)) {
        if (line.empty()) continue;

        stringstream line_ss(line);
        int t, p, d;
        if (line_ss >> t >> p >> d) {
            jobs.push_back({job_id++, t, p, d});
        }
    }

    return jobs;
}

// Print the schedule
void printSchedule(int profit, const vector<Job>& schedule) {
    cout << "\n=== RESULTS ===" << endl;
    cout << "Maximum Profit: " << profit << endl;
    cout << "Jobs to Execute: " << schedule.size() << endl;
    cout << "Schedule:" << endl;

    int current_time = 0;
    int total_profit = 0;

    for (const auto& job : schedule) {
        int start_time = current_time;
        int end_time = current_time + job.processing_time;
        bool on_time = (end_time <= job.deadline);
        int job_profit = on_time ? job.profit : 0;
```

```
        total_profit += job_profit;

        cout << "  Job" << job.id << " (t=" << job.processing_time
             << ", p=" << job.profit << ", d=" << job.deadline << ")";
        cout << " -> Time " << start_time << "-" << end_time;
        cout << " | Profit: " << job_profit;
        cout << " | " << (on_time ? "ON TIME" : "LATE") << endl;

        current_time = end_time;
    }
    cout << "Total Profit: " << total_profit << endl;
    cout << "===============\n" << endl;
}

int main() {
    cout << "=== Job Scheduling Algorithm ===" << endl;
    cout << "Input format: Each line = processing_time profit deadline" << endl;
    cout << "Example: " << endl;
    cout << "2 100 2" << endl;
    cout << "1 50 1" << endl;
    cout << "2 150 3" << endl;
    cout << "Enter 'end' on a new line to process, 'quit' to exit\n" << endl;

    while (true) {
        cout << "Enter jobs (one per line):" << endl;

        vector<string> input_lines;
        string line;
        int job_count = 0;

        // Read input until "end" or EOF
        while (getline(cin, line)) {
            if (line == "quit") {
                cout << "Goodbye!" << endl;
                return 0;
            }
            if (line == "end") {
                break;
            }
            if (!line.empty()) {
                input_lines.push_back(line);
                job_count++;
            }
        }

        if (input_lines.empty()) {
            cout << "No jobs entered. Try again.\n" << endl;
            continue;
        }

        // Combine all input lines
        string combined_input;
        for (const auto& input_line : input_lines) {
```

```
            combined_input += input_line + "\n";
        }

        // Parse and process
        vector<Job> jobs = parseInput(combined_input);

        cout << "\nProcessing " << jobs.size() << " jobs..." << endl;
        cout << "Jobs entered:" << endl;
        for (const auto& job : jobs) {
            cout << "  Job" << job.id << ": t=" << job.processing_time
                 << ", p=" << job.profit << ", d=" << job.deadline << endl;
        }

        auto [profit, schedule] = findMaxProfitSchedule(jobs);
        printSchedule(profit, schedule);

        cout << "Ready for next input...\n" << endl;
    }
}
```

Note that greedy algorithm does not apply to this problem. The local optimality cannot lead to a global optimal solution. The following is a counter example:

```
j t  p  d
1 5 900 5
2 2 300 2
3 2 300 3
4 2 300 4
5 2 300 5
```

The optimal schedule by time 4 is, comoplete task 2 during time 1-2, and then complete task 5 during time 3-4. However, when we consider the schedule by time 5, completing task 1 alone leads to the most profit, because its profit is high enough to make up for the huge consumption of time.

Analysis

The algorithm consists of the following steps:

- Sort the jobs by their deadlines. It takes $O(n \lg n)$ time.
- Find the max deadline of all the jobs. It takes $O(n)$ time.
- Initialize the vector `dp`. It takes $O(1)$ time.
- Dynamic programming takes a bottom-up approach. The nested loop takes $O(n \times \max\{d_j\})$ time in the worst case. According to the project specification, the processing time is smaller than the number of jobs, so the time complexity is $O(n^2)$.
- Find the maximum profit by traversing all the elements in `dp`. It takes $O(n)$ time.
- Reconstruct the schedule. We start from the latest task and then backtrack to find the schedule. It takes $O(n)$ time. Reversing the `schedule` vector also takes $O(n)$ time.

It can be concluded that the total time complexity is $O(n^2)$.

Project Specification

Suppose you have one machine and a set of n jobs $a_1, a_2, \ldots a_n$, to process on that machine. Each job a_j has a processing time t_j, a profit p_j, and a deadline d_j. The machine can process only one job at a time, and job a_j must run uninterruptedly for t_j consecutive time units. If job a_j is completed by its deadline d_j, you receive a profit p_j, but if it is completed after its deadline, you receive a profit of 0. Give an algorithm to find the schedule that obtains the maximum amount of profit, assuming that all processing times are integers between 1 and n. what is the running time of you algorithm?

Grading.
(1)Algorithm and implemented code (including three use cases)(60%).
(2)Efficiency of the algorithm (20%).
(3)Document (20%)