Lời giải - Nhóm 5

Trịnh Duy Hưng

December 15, 2025

1 Tổng quan bài toán

Bài toán yêu cầu lựa chọn một tập hợp con các tác vụ từ N tác vụ cho trước và sắp xếp thứ tự thực hiện sao cho tổng lợi nhuận thu được là lớn nhất. Điều kiện ràng buộc là mỗi tác vụ được chọn phải hoàn thành trước hoặc đúng thời hạn (Deadline).

Đây là một biến thể của bài toán **Knapsack 0/1 (Cái túi)** kết hợp với bài toán **Lập lịch (Scheduling)**, thuộc lớp bài toán NP-Hard.

2 Giải pháp cho Subtask 1 ($N \leq 20$)

2.1 Phương pháp thiết kế thuật toán

- **Phương pháp:** Quay lui (Backtracking) kết hợp Nhánh cận (Branch and Bound).
- **Tiền xử lý:** Sắp xếp các tác vụ theo thứ tự **Deadline (t_i) tăng dần**.
- **Nguyên lý:** Dựa trên nguyên tắc *Earliest Deadline First (EDF)*. Nếu tồn tại một tập hợp các tác vụ có thể hoàn thành đúng hạn, thì việc thực hiện chúng theo thứ tự deadline tăng dần luôn là một phương án hợp lệ.
- **Mô tả thực hiện:** Xây dựng hàm đệ quy `Try(i, currentTime, currentProfit)` để xét tác vụ thứ i:

  1. **Nhánh 1 (Không chọn tác vụ i):** Gọi đệ quy `Try(i+1, currentTime, currentProfit)`.
  2. **Nhánh 2 (Chọn tác vụ i):** Chỉ thực hiện nếu thỏa mãn điều kiện $currentTime + r_i \leq t_i$. Nếu thỏa, cập nhật thời gian, lợi nhuận và gọi `Try(i+1, currentTime +` $r_i, currentProfit + p_i$).

  Khi xét hết N tác vụ, cập nhật kết quả tối ưu toàn cục.

2.2 Tính phù hợp của phương pháp

- Với $N \leq 20$, không gian trạng thái tối đa là $2^{20} \approx 1,048,576$. Máy tính hiện đại xử lý lượng phép tính này trong vài mili-giây.
- Phương pháp này vét cạn mọi trường hợp khả dĩ, do đó đảm bảo tìm ra nghiệm **tối ưu tuyệt đối** (Exact Solution), đáp ứng yêu cầu tính điểm của Subtask 1.

2.3 Phân tích độ phức tạp

- **Độ phức tạp thời gian:** $\mathcal{O}(2^N)$. Trong trường hợp xấu nhất, thuật toán phải duyệt qua tất cả các tập con của tập tác vụ.
- **Độ phức tạp không gian:** $\mathcal{O}(N)$ dùng cho ngăn xếp đệ quy (recursion stack) và mảng lưu vết nghiệm.

3 Giải pháp cho Subtask 2 ($N \leq 500$)

3.1 Phương pháp thiết kế thuật toán

- **Phương pháp: Beam Search** (Tìm kiếm chùm) - Một dạng Heuristic của Quy hoạch động.
- **Nguyên lý hoạt động:** Thay vì duyệt toàn bộ (như Subtask 1) hay chỉ chọn 1 phương án tốt nhất cục bộ (như Greedy thuần túy), Beam Search duy trì danh sách K **trạng thái tốt nhất** tại mỗi bước.
- **Định nghĩa Trạng thái:** Một cặp $(Time, Profit)$ đại diện cho tổng thời gian đã dùng và tổng lợi nhuận đạt được.
- **Quy trình:**
  1. Sắp xếp N tác vụ theo Deadline tăng dần.
  2. Khởi tạo danh sách trạng thái (Beam) ban đầu chỉ chứa $\{0, 0\}$.
  3. Với mỗi tác vụ mới, mở rộng tất cả các trạng thái trong Beam thành 2 nhánh: *Chọn* hoặc *Không chọn*.
  4. **Sàng lọc (Pruning):**
     - Loại bỏ trạng thái vi phạm deadline.
     - Loại bỏ trạng thái bị áp đảo (Dominated): Trạng thái A bị loại nếu có trạng thái B sao cho $Time_B \leq Time_A$ và $Profit_B \geq Profit_A$.
     - **Cắt chùm (Beam Width):** Chỉ giữ lại tối đa W trạng thái có lợi nhuận cao nhất để chuyển sang bước tiếp theo.

3.2 Tính phù hợp của phương pháp

- Với $N = 500$, độ phức tạp 2^{500} là bất khả thi để tìm nghiệm chính xác.
- Phương pháp Quy hoạch động truyền thống (Knapsack) không khả thi vì deadline t_i lên tới 10^{12}, không thể khởi tạo mảng DP theo trọng lượng.
- **Lý do chọn Beam Search:**
  - Là giải pháp Heuristic cân bằng tốt giữa độ chính xác và tốc độ.
  - Phù hợp với cấu trúc bài toán có tính chất xây dựng dần (constructive) và thứ tự deadline cố định.
- **Cấu hình tham số:** Chọn Beam Width $W = 15,000$.
- **Lý do chọn tham số:** Với giới hạn thời gian 5 giây, máy tính có thể xử lý khoảng $5 \cdot 10^8$ phép tính. Tổng chi phí ước lượng là $500 \times W \log W \approx 10^8$ phép tính, hoàn toàn khả thi và an toàn, đồng thời đảm bảo độ chính xác rất cao.

3.3 Phân tích độ phức tạp

- **Độ phức tạp thời gian:** $\mathcal{O}(N \cdot W \cdot \log W)$. Tại mỗi bước trong N bước, danh sách trạng thái tăng lên $2W$, sau đó cần sắp xếp để chọn ra W phần tử tốt nhất, mất $\mathcal{O}(W \log W)$.
- **Độ phức tạp không gian:** $\mathcal{O}(N \cdot W)$. Cần lưu trữ vết của W trạng thái qua N bước để truy vết lại kết quả các chỉ số (ID) tác vụ.

4 Code minh hoạ

```
import sys

# Tăng giới hạn đệ quy nếu cần thiết (dù thuật toán này là iterative)
sys.setrecursionlimit(2000)

class Task:
    __slots__ = ('original_id', 'r', 't', 'p')
    def __init__(self, original_id, r, t, p):
        self.original_id = original_id
        self.r = r
        self.t = t
        self.p = p

class State:
    # __slots__ giúp giảm đáng kể bộ nhớ tiêu thụ khi tạo hàng ngàn object
    __slots__ = ('time', 'profit', 'task_index', 'prev_state')
    def __init__(self, time, profit, task_index, prev_state):
        self.time = time
        self.profit = profit
        self.task_index = task_index
        # Thay vì lưu index (int), trong Python ta lưu tham chiếu trực tiếp
        # đến object trạng thái trước đó để truy vết dễ hơn.
        self.prev_state = prev_state

def solve():
    # Đọc dữ liệu input nhanh
    input_data = sys.stdin.read().split()
    if not input_data:
        return

    iterator = iter(input_data)
    try:
        n = int(next(iterator))
    except StopIteration:
        return

    tasks = []
    for i in range(n):
        r = int(next(iterator))
        t = int(next(iterator))
        p = int(next(iterator))
        # original_id bắt đầu từ 1
        tasks.append(Task(i + 1, r, t, p))

    # BEAM_WIDTH: Số lượng trạng thái tối đa giữ lại ở mỗi bước.
    # Python chậm hơn C++ nên nếu bị Time Limit, bạn có thể giảm xuống 5000-10000.
    BEAM_WIDTH = 15000

    # 1. Sắp xếp tác vụ theo deadline tăng dần
```

```
tasks.sort(key=lambda x: x.t)

# Layer khởi tạo: Time=0, Profit=0, chưa chọn task nào (-1), không có cha (None)
# layers chỉ cần giữ layer hiện tại để tiết kiệm bộ nhớ,
# nhưng để giống logic C++ ta dùng list. Ở đây ta chỉ cần current_layer.
current_layer = [State(0, 0, -1, None)]

for i in range(n):
    current_task = tasks[i]
    next_layer = []

    # Duyệt qua tất cả các trạng thái của bước trước
    for s in current_layer:
        # Lựa chọn 1: KHÔNG chọn tác vụ hiện tại
        next_layer.append(State(s.time, s.profit, -1, s))

        # Lựa chọn 2: CHỌN tác vụ hiện tại (nếu kịp deadline)
        if s.time + current_task.r <= current_task.t:
            next_layer.append(State(
                s.time + current_task.r,
                s.profit + current_task.p,
                i, # Lưu index của task trong mảng đã sort
                s  # Lưu tham chiếu đến trạng thái cha
            ))

    # --- BƯỚC QUAN TRỌNG: LỌC VÀ TỐI ƯU (PRUNING) ---

    # 1. Sắp xếp theo Thời gian tăng dần, nếu trùng thời gian thì Profit giảm dần
    # (Để logic pruning bên dưới hoạt động hiệu quả)
    next_layer.sort(key=lambda x: (x.time, -x.profit))

    # 2. Lọc bỏ các trạng thái bị áp đảo (Dominated)
    pruned_layer = []
    max_profit_seen = -1

    for s in next_layer:
        if s.profit > max_profit_seen:
            max_profit_seen = s.profit
            pruned_layer.append(s)

    # 3. Beam Search: Cắt bớt nếu quá nhiều trạng thái
    # pruned_layer lúc này đã có tính chất: Time tăng dần VÀ Profit tăng dần.
    # Ta giữ lại các phần tử cuối mảng (có Profit cao nhất).
    if len(pruned_layer) > BEAM_WIDTH:
        current_layer = pruned_layer[-BEAM_WIDTH:]
    else:
        current_layer = pruned_layer

# --- TRUY VẾT KẾT QUẢ ---

# Tìm trạng thái có lợi nhuận lớn nhất ở layer cuối cùng
```

```
    best_state = None
    max_p = -1

    for s in current_layer:
        if s.profit > max_p:
            max_p = s.profit
            best_state = s

    print(max_p)

    # Truy vết ngược nhờ tham chiếu prev_state
    selected_ids = []
    curr = best_state
    while curr is not None:
        if curr.task_index != -1:
            selected_ids.append(tasks[curr.task_index].original_id)
        curr = curr.prev_state

    # In danh sách ID theo đúng thứ tự thời gian (đảo ngược lại danh sách truy vết)
    print(*(selected_ids[::-1]))

if __name__ == '__main__':
    solve()
```