NHÓM 11: PHẠM NGUYỄN DUY ANH - PHẠM NGỌC PHÚ THỊNH

BÁO CÁO BÀI TẬP CS112

FUCS VÀ GÁNH NẶNG MƯU SINH

Ngày 15 tháng 12 năm 2025

Mục lục

1 Phương pháp thiết kế thuật toán

1.1 Tên phương pháp

Phương pháp được sử dụng để giải quyết bài toán này là **Beam Search** (Tìm kiếm chùm) kết hợp với tư duy **Dynamic Programming** (Quy hoạch động).

1.2 Mô tả nguyên lý và cách áp dụng

1.2.1 1. Mô hình hóa bài toán

Bài toán yêu cầu lựa chọn tập con các tác vụ để tối đa hóa lợi nhuận mà không vi phạm deadline. Đây là biến thể của bài toán *Knapsack (Cái túi)* nhưng có thêm yếu tố thời gian (deadline riêng biệt cho từng tác vụ).

Nếu giải bằng Quy hoạch động (DP) truyền thống:

- Gọi $DP[i][t]$ là lợi nhuận lớn nhất khi xét đến tác vụ i tại thời điểm t.
- Tuy nhiên, t có thể lên tới 10^{12} (theo Subtask 2), do đó không thể khởi tạo mảng hay bảng DP thông thường vì giới hạn bộ nhớ.

1.2.2 2. Chiến lược Beam Search

Để giải quyết vấn đề bùng nổ không gian trạng thái, ta sử dụng Beam Search. Thay vì lưu trữ toàn bộ các trạng thái có thể xảy ra ở mỗi bước (Breadth-First Search), ta chỉ giữ lại một số lượng cố định K (Beam Width) các trạng thái "tiềm năng nhất".

Các bước thực hiện chính:

1. **Sắp xếp (Sorting):** Sắp xếp các tác vụ theo thứ tự Deadline (t_i) tăng dần. Điều này giúp ta xử lý tuần tự theo trục thời gian.
2. **Sinh trạng thái (Expansion):** Tại mỗi bước xét tác vụ i, từ mỗi trạng thái cũ $(time, profit)$, ta có 2 lựa chọn:
   - Không làm tác vụ i: Trạng thái giữ nguyên.
   - Làm tác vụ i (nếu $time + r_i \leq t_i$): Trạng thái mới $(time + r_i, profit + p_i)$.
3. **Cắt tỉa (Pruning):** Đây là bước quan trọng nhất.
   - *Dominated States:* Loại bỏ các trạng thái kém hiệu quả. Nếu trạng thái A tốn ít thời gian hơn trạng thái B nhưng lại có lợi nhuận cao hơn, thì trạng thái B bị loại bỏ.
   - *Beam Width Selection:* Giới hạn số lượng trạng thái tối đa (trong bài chọn $BEAM\_WIDTH \approx 15,000$).

1.2.3 Cài đặt Python minh họa: Cấu trúc dữ liệu

```python
import sys
# Cau truc tac vu
class Task:
    def __init__(self, original_id, r, t, p):
        self.original_id = original_id
        self.r, self.t, self.p = r, t, p

# Cau truc trang thai
```

```
class State:
    def __init__(self, time, profit, task_index, prev_state_idx):
        self.time = time
        self.profit = profit
        self.task_index = task_index
        self.prev_state_idx = prev_state_idx
```

Listing 1: Khởi tạo và Sắp xếp tác vụ

2 Tính phù hợp của phương pháp

2.1 Đối với Subtask 2 ($N \leq 500, T \leq 10^{12}$)

Phương pháp Beam Search là lựa chọn tối ưu vì:

- **Quản lý bộ nhớ:** Không phụ thuộc vào giá trị thời gian T (tránh lỗi Memory Limit Exceeded của mảng DP truyền thống).
- **Cân bằng độ chính xác:** Với $BEAM\_WIDTH$ đủ lớn, thuật toán tìm được nghiệm rất sát với tối ưu (hoặc chính là tối ưu) trong thời gian cho phép.
- **Tính chất trội (Dominance):** Bài toán cho phép loại bỏ rất nhiều trạng thái tồi, giúp Beam Search hội tụ nhanh về các phương án tốt.

2.2 Đối với Subtask 1 ($N \leq 20$)

Mặc dù Beam Search là thuật toán Heuristic, nhưng với N nhỏ, tổng số trạng thái tối đa là $2^{20} \approx 10^6$. Với $BEAM\_WIDTH = 15,000$ và cơ chế cắt tỉa (Pruning), thuật toán sẽ duyệt qua hầu hết các trạng thái quan trọng. Do đó, nó hoạt động tương đương thuật toán vét cạn/chính xác, đảm bảo tìm ra **nghiệm tối ưu tuyệt đối** đúng như yêu cầu.

3 Phân tích độ phức tạp

3.1 Độ phức tạp thời gian

Gọi N là số lượng tác vụ và B là độ rộng chùm ($BEAM\_WIDTH$).

- Tại mỗi bước i (tổng N bước), ta sinh ra $2B$ trạng thái.
- Việc sắp xếp và lọc $2B$ trạng thái tốn $O(B \log B)$.
- Tổng thời gian: $O(N \cdot B \log B)$.
- Với $N = 500, B = 15000$, số phép tính $\approx 10^8$, khả thi trong giới hạn 5 giây.

3.2 Độ phức tạp không gian

- Cần lưu vết N lớp để truy hồi kết quả.
- Tổng không gian: $O(N \cdot B)$.
- Với $N = 500$, bộ nhớ sử dụng khoảng vài trăm MB (trong giới hạn 1024MB).

4 Phụ lục: Mã nguồn đầy đủ

Dưới đây là mã nguồn Python hoàn chỉnh để giải quyết bài toán:

```python
import sys

# Cau truc tac vu
class Task:
    def __init__(self, original_id, r, t, p):
        self.original_id = original_id
        self.r = r  # Thoi gian chay
        self.t = t  # Deadline
        self.p = p  # Loi nhuan

# Cau truc trang thai cho Beam Search
class State:
    def __init__(self, time, profit, task_index, prev_state_idx):
        self.time = time
        self.profit = profit
        self.task_index = task_index
        self.prev_state_idx = prev_state_idx

BEAM_WIDTH = 15000

def solve():
    # Doc input tu stdin
    input_data = sys.stdin.read().split()
    if not input_data: return

    iterator = iter(input_data)
    try:
        n = int(next(iterator))
        tasks = []
        for i in range(n):
            r = int(next(iterator))
            t = int(next(iterator))
            p = int(next(iterator))
            tasks.append(Task(i + 1, r, t, p))

        # 1. Sap xep tac vu theo deadline tang dan
        tasks.sort(key=lambda x: x.t)

        # 2. Khoi tao Beam Search
        layers = [[State(0, 0, -1, -1)]]

        for i in range(n):
            current_task = tasks[i]
            prev_layer = layers[-1]
            next_layer = []

            for j, s in enumerate(prev_layer):
                # Lua chon 1: Khong chon
                next_layer.append(State(s.time, s.profit, -1, j))
                # Lua chon 2: Chon (neu kip)
                if s.time + current_task.r <= current_task.t:
                    next_layer.append(State(
                        s.time + current_task.r,
                        s.profit + current_task.p,
                        i, j
                    ))

```

```
            # 3. Pruning & Beam Selection
            next_layer.sort(key=lambda x: (x.time, -x.profit))

            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)

            if len(pruned_layer) > BEAM_WIDTH:
                layers.append(pruned_layer[-BEAM_WIDTH:])
            else:
                layers.append(pruned_layer)

        # 4. Truy vet ket qua
        final_layer = layers[-1]
        best_state = max(final_layer, key=lambda x: x.profit)
        print(best_state.profit)

        selected_ids = []
        current_layer_idx = len(layers) - 1
        current_state_idx = final_layer.index(best_state)

        while current_layer_idx > 0:
            s = layers[current_layer_idx][current_state_idx]
            if s.task_index != -1:
                selected_ids.append(tasks[s.task_index].original_id)
            current_state_idx = s.prev_state_idx
            current_layer_idx -= 1

        print(*(selected_ids[::-1]))

    except StopIteration:
        pass

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

Listing 2: Full Source Code