Vikram Thirumaran

Prof. Thanh H. Nguyen

CS 315

March 2nd, 2025

HW7

Q1. Greedy (Easy) (25 points)

Problem Understanding

You have n jobs that need to be pre-processed on a single supercomputer before being moved to parallel PCs for further processing. The goal is to minimize the total completion time of all jobs.

Each job $J_i=(s_i,f_i)$:

- Requires s_i units of time on the supercomputer.
- Requires f_i units of time on the PC.

If we process jobs in order $J_1, J_2,\ldots,J_n$, the finish time of job J_i is:

$$T_i = (\sum_{k=1}^{i} s_k) + f_i$$

The total completion time we need to minimize is:

$$C_{max}=maxT_i$$

Step 1: Key Insight (Greedy Strategy)

Since the PC processing can be done in parallel, the goal is to minimize the time the last job finishes.

A key observation is that the supercomputer is the bottleneck. If a job with a large f_i (PC processing time) is left towards the end, it will dominate the final completion time.

Thus, to minimize C_{max} , we should schedule jobs in decreasing order of f_i.

Step 2: Algorithm Design

1. Sort jobs in decreasing order of f_i
    - That is, schedule jobs with longer PC times earlier.
2. Process jobs on the supercomputer in this order.
3. Compute the total completion time.

Step 3: Proof of Correctness

1. Optimal Substructure:
    - If we already have an optimal schedule for the first *i -1* jobs, adding the i^{th} job should maintain optimality.
2. Greedy Choice Property:
    - By scheduling longer PC jobs first, we prevent them from delaying the final job excessively.
    - If we swapped a job with larger *f* later in the sequence, it would only increase the total completion time.

Step 4: Runtime Analysis

- Sorting the jobs takes *O(nlogn)*.
- Processing jobs takes *O(n)*.
- Total complexity: *O(nlogn)*.

This meets the requirement of an O(*nlogn)* algorithm.

Thus, sorting by decreasing f_i ensures that the job with the longest PC time starts earlier, reducing the delay at the end. ■

Q2. Greedy (Medium)

Understanding the Problem

We have *n* workers, and each worker must complete:

- One morning task m_i (taking m_i hours).
- One afternoon task a_j (taking a_j hours).

Each worker's total working time is:

$$T_i = m_i + a_j$$

- If $T_i \leq W$, there is no overtime cost.

- If $T_i > W$, the overtime cost is:

$$s(T_i - W)$$

where:

- W is the maximum regular working hours.
- s is the overtime pay per extra hour.

Step 1: Key Observations

- A bad assignment could lead to some workers having very high overtime, which increases the overall cost.
- The best way to minimize overtime is to balance the workload across workers.

Thus, we should avoid pairing long tasks together.

Step 2: Greedy Strategy

To achieve a balanced workload, we use the following approach:

1. Sort morning tasks in increasing order: $m_1 \leq m_2 \leq \cdots \leq m_n$
2. Sort afternoon tasks in decreasing order: $a_1 \geq a_2 \geq \cdots \geq a_n$
3. Pair them in this order:
    - Assign the smallest morning task m_1 with the largest afternoon task a_1.
    - Assign the second smallest morning task $m_2$with the second largest afternoon task a_2
    - Continue this process for all workers.

Step 3: Why is this Optimal?

This greedy pairing ensures that:

- Large afternoon tasks do not get paired with large morning tasks, preventing extreme overtime costs.
- The overall workload is evenly spread among workers, reducing the maximum overtime cost.

If we instead paired large morning tasks with large afternoon tasks, some workers would end up working significantly more than W, leading to higher total overtime.

Step 4: Complexity Analysis

- Sorting morning tasks → *O(nlogn)*
- Sorting afternoon tasks → *O(nlogn)*
- Pairing and calculating overtime → *O(n)*

Thus, the total runtime complexity is *O(nlogn)*, which meets the problem's requirements.

Final Answer

To minimize overtime costs:

1. Sort morning tasks in increasing order.
2. Sort afternoon tasks in decreasing order.
3. Pair them in this order and calculate the total overtime cost.

This ensures an efficient and optimal schedule for minimizing overtime costs.

Q3: Max Flow - Min Cut (Basic)

Problem Breakdown

We are given a network graph *G* with:

- A source node *s*
- A sink node *t*
- Edges with integer capacities

Objective:

1. Determine the maximum flow from *s* to *t*.
2. Find the minimum cut in the network.

Step 1: Understanding Max Flow

The maximum flow in a flow network is the maximum amount of flow that can be sent from the source sss to the sink *t*, while respecting the capacity constraints on each edge.

We use Ford-Fulkerson Algorithm to determine max flow:

- Find an augmenting path using BFS (Edmonds-Karp) or DFS.
- Push as much flow as possible through this path.
- Repeat until no more augmenting paths exist.

Step 2: Understanding Min Cut

The minimum cut is the smallest capacity sum of edges that, when removed, would disconnect sss from *t*.

Key property:

- Min Cut = Max Flow (by the Max-Flow Min-Cut Theorem).
- The min cut consists of edges crossing from the reachable nodes in the residual graph (after max flow is achieved) to non-reachable nodes.

Step 3: Algorithm to Solve

1. Compute max flow using Ford-Fulkerson / Edmonds-Karp.
    - Start with zero flow.
    - Find augmenting paths using BFS.
    - Push flow along augmenting paths until no more exist.
    - The final flow value is the max flow.
2. Find min cut:
    - Run BFS from sss in the residual graph.
    - Mark all reachable nodes.
    - The min cut consists of edges from reachable nodes to non-reachable nodes.

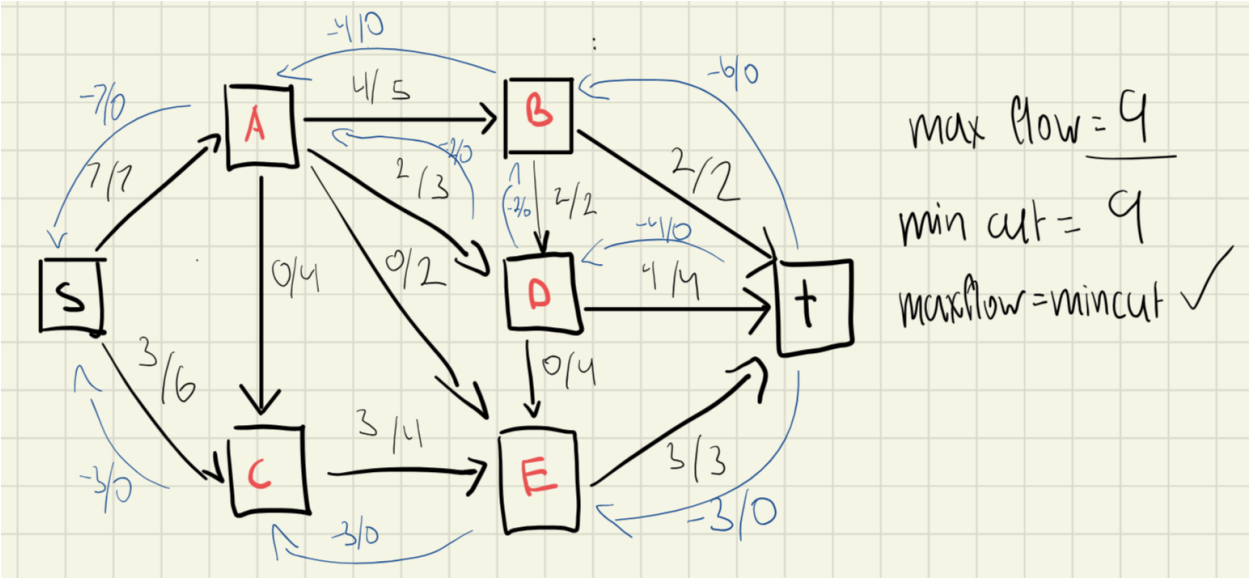
Step 4: Complexity Analysis

- Ford-Fulkerson (DFS-based): $O(E \cdot F)$, where F is the max flow value.
- Edmonds-Karp (BFS-based): $O(VE^2)$.
- Finding min cut: $O(V+E)$ (using BFS on the residual graph).

Step 5: Final Answer

1. Run the max-flow algorithm (Ford-Fulkerson or Edmonds-Karp) to compute the max flow.
2. Run BFS on the residual graph from *s*, marking all reachable nodes.
3. Identify the min-cut edges as those going from reachable nodes to non-reachable nodes.

Thus, Max Flow = Min Cut, and we find both optimally using this approach.



Q4: Max Flow - Min Cut (Easy)

The problem describes a task allocation scenario where:

- Each worker *i* can only work on a subset S_i of tasks.
- Each worker can handle at most *k* tasks.
- Each task must be assigned to at least *w* workers.

We need to determine whether there exists a **valid assignment** that satisfies all constraints.

Step 1: Model the Problem as a Max Flow Network

We can solve this using a flow network by constructing a bipartite graph:

Graph Construction

1. Source Node: *s*
2. Worker Nodes: $W_1, W_2, \ldots, W_n$
3. Task Nodes: $T_2, T_2, \ldots, T_m$
4. Sink Node: *t*

Edges & Capacities

- Source to Workers:
    - Connect *s* to each worker W_i capacity *k* (since each worker can handle at most *k* tasks).
- Workers to Tasks:

    - If a worker W_i can perform task T_j(i.e., $j \in S_i$), add an edge from W_i to T_j with infinite capacity (ensuring valid flow).
- Tasks to Sink:
    - Connect each task T_j to the sink t with capacity w (ensuring each task gets assigned at least w workers).

Step 2: Solve Using Max Flow

- Compute the Maximum Flow from s to t.
- Check if the flow equals $m \cdot w$ (i.e., each task receives at least w workers).
- If max flow = $m \cdot w$, a valid assignment exists.
- Otherwise, no valid assignment is possible.

Step 3: Algorithm Complexity

- Graph Construction: $O(n+m)$
- Max Flow Computation (Edmonds-Karp Algorithm): $O(VE^2)$, where $V=n+m+2$, and E is the number of edges.
- Since $E \approx n \cdot m$, the complexity simplifies to $O(n^2m^2)$, which is polynomial time.■

Final Answer

1. Construct the flow network as described.
2. Compute max flow from s to t.
3. Check if max flow = $m \cdot w$
    - If yes, a valid assignment exists.
    - If no, no valid assignment is possible.