CS-GY 6033 : Design and Analysis of Algorithms , Section C : Assignment-6

Khwaab Thareja — N15911999 — kt3180

December 2024

Problem 6-1: Ups and Downs

Suppose we are given as input a directed graph $G = (V, E)$ with distinct real edge weights $w(u, v) \in \mathbb{R}$ (both positive and negative). We wish to find the shortest paths from a single source vertex $s \in V$.

Without any additional information, we would probably run Bellman-Ford. Suppose, however, that we are given one additional restriction: the edge weights along any shortest path starting from s first strictly increase, then strictly decrease, then strictly increase.

For example, the sequence of edge weights on a shortest path may be $\langle 3, 7, 15, 9, -3, 2, 8 \rangle$, but they could not be $\langle 3, 7, 4, 8, 1 \rangle$. (There may be other paths in the graph that do not obey this increasing-decreasing-increasing property. The restriction applies only to shortest paths.)

(a) Provide the best single-source shortest paths algorithm you can, assuming the graph observes the increasing-decreasing-increasing restriction on shortest paths.

Note: Your algorithm should be much faster than Bellman-Ford, or it will not receive any credit. **Hint:** First think about how you would solve the problem if edge weights only increase along shortest paths, then extend your algorithm to handle the actual problem statement. You will want to use sorting as a subroutine, so you should assume that you have an algorithm that sorts n elements in $O(n \log n)$ time.

(b) Analyze the running time of your algorithm.

(c) Explain how to test whether the shortest paths from s do in fact obey the increasing-decreasing-increasing property. How efficiently can you do that?

Answer (a)

The algorithm leverages a **stage-based relaxation process**:

1. **Stage 1**: Edges are relaxed in **increasing order** of weights to identify the first increasing phase of the shortest paths.
2. **Stage 2**: Edges are relaxed in **decreasing order** of weights to find paths that decrease after the first phase.
3. **Stage 3**: Edges are relaxed again in **increasing order** of weights to account for the final increasing phase.

This ensures that only paths adhering to the I-D-I property are considered, eliminating invalid paths and improving efficiency.

Algorithm

1. **Input Preparation**:
   - Represent the graph as $G = (V, E)$ with distinct edge weights $w(u, v) \in \mathbb{R}$.
   - Initialize a **dynamic programming table** $dp[v][k]$, where $dp[v][k]$ is the shortest path distance to vertex v in stage k ($k = 1, 2, 3$).
2. **Initialization**:
   - Set $dp[s][1] = 0$, as the distance from the source s to itself in Stage 1 is zero.
   - For all other vertices and stages, set $dp[v][k] = \infty$.
3. **Relaxation in Three Stages**:

   (a) **Stage 1 (Increasing)**:
   - Sort the edges in **ascending order** by weight.
   - For each edge (u, v), relax it only if its weight is larger than the previous edge in this stage:

   $$dp[v][1] = \min(dp[v][1], dp[u][1] + w(u, v)).$$

   (b) **Stage 2 (Decreasing)**:
   - Sort the edges in **descending order** by weight.
   - Relax edge (u, v) only if its weight is smaller than the weight of the last edge used in Stage 1:

   $$dp[v][2] = \min\left(d_2(v), d_1(u) + w(u, v), d_2(u) + w(u, v)\right)$$

   (c) **Stage 3 (Increasing Again)**:
   - Sort the edges in **ascending order** by weight again.
   - Relax edge (u, v) only if its weight is larger than the weight of the last edge used in Stage 2:

   $$dp[v][3] = \min\left(d_3(v), d_2(u) + w(u, v), d_3(u) + w(u, v)\right)$$

4. **Output the Result**:
    - After completing the three stages, the shortest path to any vertex v that satisfies the I-D-I property is stored in $dp[v][3]$.

Answer (b)

Time Complexity

- **Sorting**: Sorting the edges in ascending and descending order takes $O(E \log E)$, where E is the number of edges.
- **Relaxation**: Each stage processes all edges once, which takes $O(E)$ per stage.
- **Overall Complexity**: Sorting: $O(E \log E)$, Relaxation: $3 \cdot O(E) = O(E)$. Total complexity is:

  $$O(E \log E),$$

  which is much faster than the $O(VE)$ of Bellman-Ford.

Answer (c)

We need to verify whether the shortest paths originating from a source vertex s follow the **increasing-decreasing-increasing (I-D-I)** edge weight pattern. The edge weights along the shortest paths must satisfy the following:

1. First, the weights **strictly increase**.
2. Next, the weights **strictly decrease**.
3. Finally, the weights **strictly increase** again.

Steps for Verification

1. Compute Shortest Paths:

- Use the Algorithm mentioned in Part (a) to find the shortest path

2. Extract Edge Weights:

- For each shortest path, extract the sequence of edge weights, denoted as $W_v = \langle w_1, w_2, \dots, w_k \rangle$, where w_i represents the weight of the i-th edge in the path.

3. Verify the I-D-I Pattern:

- Start with the first phase (**increasing phase**).
- Traverse the sequence of edge weights and check if:
  - The weights **strictly increase** during the first phase.
  - The weights **strictly decrease** during the second phase.
  - The weights **strictly increase** again during the final phase.
- Transitions between phases are allowed under the following conditions:
  1. Transition from Phase 1 to Phase 2 when a decrease in weight is detected.
  2. Transition from Phase 2 to Phase 3 when an increase in weight is detected.
- Ensure that the path ends in the third phase (increasing again). If it does not, the path does not satisfy the I-D-I property.

Conditions for Each Phase

Phase 1 (Increasing):

- Remain in this phase if $w_{\text{current}} > w_{\text{last}}$.
- Transition to Phase 2 (Decreasing) if $w_{\text{current}} < w_{\text{last}}$.
- The path violates the I-D-I property if $w_{\text{current}} = w_{\text{last}}$.

Phase 2 (Decreasing):

- Remain in this phase if $w_{\text{current}} < w_{\text{last}}$.
- Transition to Phase 3 (Increasing Again) if $w_{\text{current}} > w_{\text{last}}$.
- The path violates the I-D-I property if $w_{\text{current}} = w_{\text{last}}$.

Phase 3 (Increasing Again):

- Remain in this phase if $w_{\text{current}} > w_{\text{last}}$.
- The path violates the I-D-I property if $w_{\text{current}} \leq w_{\text{last}}$.

Time Complexity

For a Single Path:

- Traversing the sequence of edge weights to verify the I-D-I property takes $O(k)$, where k is the number of edges in the path.

For All Paths:

- If there are p paths, and the total number of edges across all paths is E, the total time complexity is:
$$O(E)$$

6.2 Edge-Disjoint Paths Problem

In graph theory, the **Edge-Disjoint Paths Problem** is to determine the maximum number of edge-disjoint paths between two given nodes in a graph. Two paths are considered **edge-disjoint** if they do not share any edges.

Problem Statement

Given a directed graph $G = (V, E)$, and two nodes s (source) and t (sink), find the maximum number of edge-disjoint paths from s to t.

Example

Consider the following directed graph $G = (V, E)$ with source s and sink t:
In this example, there are 3 edge-disjoint paths:

1. $s \to b \to d \to t$
2. $s \to a \to c \to t$
3. $s \to c \to e \to t$

(a) Design an algorithm that computes the maximum edge-disjoint paths and returns the corresponding disjoint paths.

Answer (a)

The Edge-Disjoint Paths Problem can be solved using a maximum flow algorithm. The maximum number of edge-disjoint paths between a source s and a sink t is equal to the maximum flow in the graph when each edge has a capacity of 1.

Algorithm Steps

1. **Assign Capacities:** Assign a capacity of 1 to every edge in the graph $G = (V, E)$.
2. **Compute Maximum Flow:** Use a maximum flow algorithm like the **Ford-Fulkerson method**.
3. **Extract Edge-Disjoint Paths:** After computing the maximum flow, extract the edge-disjoint paths by tracing the flow in the residual graph. Each unit of flow corresponds to an edge-disjoint path.

Algorithm Pseudocode

```
from collections import defaultdict, deque

def bfs_find_augmenting_path(graph, capacity, flow, source,
    sink, parent):
    visited = set()
    queue = deque([source])
    visited.add(source)

    while queue:
        current = queue.popleft()
        for neighbor in graph[current]:
            if neighbor not in visited and capacity[current
                ][neighbor] - flow[current][neighbor] > 0:
                queue.append(neighbor)
                visited.add(neighbor)
                parent[neighbor] = current
                if neighbor == sink:
                    return True
    return False

def ford_fulkerson(graph, capacity, source, sink):
    flow = defaultdict(lambda: defaultdict(int))
    max_flow = 0
    parent = {}

    while bfs_find_augmenting_path(graph, capacity, flow,
        source, sink, parent):
        path_flow = float('inf')
        v = sink
        while v != source:
            u = parent[v]
            path_flow = min(path_flow, capacity[u][v] - flow
                [u][v])
            v = u

        v = sink
        while v != source:
            u = parent[v]
            flow[u][v] += path_flow
            flow[v][u] -= path_flow
            v = u

        max_flow += path_flow

    return max_flow, flow

def extract_edge_disjoint_paths(graph, flow, source, sink):
    paths = []
```

```python
    residual_graph = defaultdict(list)
    for u in graph:
        for v in graph[u]:
            if flow[u][v] > 0:
                residual_graph[u].append(v)

    while True:
        path = []
        visited = set()
        stack = [source]
        while stack:
            node = stack.pop()
            path.append(node)
            if node == sink:
                break
            visited.add(node)
            for neighbor in residual_graph[node]:
                if neighbor not in visited:
                    stack.append(neighbor)
                    residual_graph[node].remove(neighbor)
                    break
        else:
            break

        paths.append(path)

    return paths

def max_edge_disjoint_paths(graph, source, sink):
    capacity = defaultdict(lambda: defaultdict(int))
    for u in graph:
        for v in graph[u]:
            capacity[u][v] = 1

    max_flow, flow = ford_fulkerson(graph, capacity, source,
         sink)
    paths = extract_edge_disjoint_paths(graph, flow, source,
         sink)
    return max_flow, paths
```

Time Complexity

- **Maximum Flow:** The Ford-Fulkerson algorithm (with BFS) has a time complexity of $O(E \cdot V^2)$, where E is the number of edges and V is the number of vertices.
- **Path Extraction:** Extracting paths from the residual graph takes $O(E)$.
- **Total Time Complexity:** $O(E \cdot V^2)$.

Problem 6-3: Scheduling Classes

The registrar's office is facing challenges in optimizing class schedules for students. They have reached out to you, as an expert in optimization problems, to provide assistance.

Input

The input to the problem includes the following:

- An integer n specifying the number of students. For convenience, assume the students are identified by an ID $1, 2, 3, \ldots, n$.
- An integer m specifying the number of courses. For convenience, assume the courses are identified by a course number $1, 2, \ldots, m$.
- For each student i, a list S_i of 8 courses that the student wishes to take. For example:

  $$S_1 = [2, 4, 5, 7, 9, 10, 11, 13],$$

  which means that Student 1 is willing to take Course 2, Course 4, Course 5, Course 7, Course 9, Course 10, Course 11, or Course 13.
- For each course j, a number C_j indicating the maximum number of students that can enroll in the course.

You would like to assign students to classes to maximize the total enrollment across all classes, subject to the following constraints:

- Each student can only be enrolled in a subset of their 8 selected courses.
- Each student can only be enrolled in at most 5 courses total.
- No course enrollment can exceed its specified capacity.

Tasks

(a) Algorithm Design

Provide an algorithm that uses **maximum flow** as a black box to find an optimum solution for this problem. The algorithm should output which students are assigned to which courses, not just the number of seats filled.

Note that the problem is not explicitly stated as a graph problem. You must describe the following:

1. How to construct the corresponding flow network.
2. How to transform the maximum flow solution into a solution to the original problem.

(b) Running Time Analysis

Analyze the **worst-case running time** of your algorithm with respect to the number of students n and the number of courses m, assuming the Ford-Fulkerson algorithm is used as the underlying maximum flow algorithm.

Answer (a)

Solution: Assigning Students to Courses

(a) Algorithm

Step 1: Represent the Problem as a Flow Network

We can model the problem as a flow network structured as follows:

- **Nodes:**
  - **Source** (s): Represents the starting point of the flow.
  - **Student Nodes** ($S_1, S_2, \dots, S_n$): Each student is represented as a node.
  - **Course Nodes** ($C_1, C_2, \dots, C_m$): Each course is represented as a node.
  - **Sink** (t): Represents the endpoint of the flow.
- **Edges and Capacities:**
  - Add edges from the source s to each student S_i with a capacity of 5, ensuring that no student is assigned to more than 5 courses.
  - Add edges from each student S_i to the courses $C_j \in S_i$ (courses in the student's preference list) with a capacity of 1, indicating that a student can only enroll in a course once.
  - Add edges from each course C_j to the sink t with a capacity equal to the course's maximum enrollment capacity C_j.

Step 2: Compute Maximum Flow

Run a standard maximum flow algorithm, such as **Ford-Fulkerson**, to compute the maximum flow from the source (s) to the sink (t). This ensures the total number of students enrolled is maximized while adhering to the constraints.

Step 3: Extract Student-Course Assignments

Using the resulting flow network:

1. For each edge between a student node S_i and a course node C_j, check if the flow value on the edge is 1.

2. If the flow value is 1, assign the student S_i to the course C_j.
3. Compile and output the list of courses assigned to each student.

(b) Time Complexity

The algorithm's time complexity depends on:

- n: Number of students.
- m: Number of courses.

Nodes and Edges

- **Nodes:** The total number of nodes in the flow network is:

$$|V| = n + m + 2 \quad \text{(students, courses, source, sink)}.$$

- **Edges:** The total number of edges in the flow network is:

$$|E| = n + 8n + m = 9n + m.$$

Ford-Fulkerson Complexity

The time complexity of the Ford-Fulkerson algorithm is given by:

$$O(E \cdot F),$$

where:

- $E = 9n + m \quad$ (number of edges).
- $F \leq 5n \quad$ (maximum flow, since each student can take at most 5 courses).

  Substituting the values:

$$O(E \cdot F) = O((9n + m) \cdot 5n) = O(45n^2 + 5mn).$$

Final Complexity

The overall time complexity of the algorithm is:

$$O(45n^2 + 5mn) \quad \text{or simplified as} \quad O(n^2 + mn).$$

Problem 6-4: Minimum Path Cover

Given a directed acyclic graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the goal is to determine the **minimum number of robots** needed to serve all vertices of the graph. Each robot can start and end at any vertex in G and can move only along the edges of the graph. Once a robot visits a vertex v, it can serve v. Multiple robots are allowed to visit the same vertex.

Example

Consider the graph shown in Figure 2. In this example, 3 robots are required to serve all the vertices. The paths for the robots are as follows:

- $a \to c \to f \to g \to j$,
- $d \to f \to h$,
- $b \to e \to i$.

Figure 2: Using 3 robots to serve all the vertices in the graph.
—

(a) One Robot Per Vertex

Now assume that each vertex must be covered by exactly one robot. Provide an algorithm using **maximum flow** to solve this problem. Describe the construction of the flow graph. You do not need to prove the correctness of your algorithm.
—

(b) Multiple Robots Per Vertex

Solve the case where each vertex can be served by multiple robots, using the algorithm from the previous problem.

Answer (a)

Part A is similar to airline scheduling problem. To ensure that each vertex is served by exactly one robot, we use **vertex splitting** to enforce the constraint. This transforms the problem into a maximum flow problem.

Flow Graph Construction

1. **Vertex Splitting**: - For each vertex $v \in V$ in the original graph, create two nodes:

- v_{in}: Represents the entry into vertex v.

- v_{out}: Represents the exit from vertex v.

- Add an edge from v_{in} to v_{out} with capacity 1. This ensures that at most one robot can serve vertex v.

2. **Edges**: - For each edge $(u, v) \in E$ in the original graph, add an edge from u_{out} to v_{in} with capacity 1. This preserves the connectivity of the graph.

3. **Source and Sink**: - Add a source node s and connect it to all v_{in} nodes corresponding to vertices with no incoming edges in the original graph. - Add a sink node t and connect it to all v_{out} nodes corresponding to vertices with no outgoing edges in the original graph.

Flow Constraints

- Each vertex can only be visited once:

- The edge between v_{in} and v_{out} has a capacity of 1, limiting the flow through the vertex to 1.
- The capacity of edges between u_{out} and v_{in} is also 1, preventing multiple robots from entering the same vertex.
- The set of in and out vertices forms a bipartite matching.

Algorithm

1. Construct the flow graph as described above.
2. Use a standard maximum flow algorithm (e.g., Ford-Fulkerson or Edmonds-Karp) to compute the maximum flow from s to t.
3. The maximum flow value corresponds to the Maximum bipartite matching.
4. The size of the minimum path cover is:

$$\text{Minimum Path Cover} = |V| - \text{Maximum Bipartite Matching}.$$

5. Extract the paths corresponding to the flow to determine the routes for the robots.

Answer (b)

In this part, each vertex in the Directed Acyclic Graph (DAG) can be covered by multiple robots. The goal is to determine the **minimum number of robots** required to serve all vertices, where robots can start and stop arbitrarily, and vertex coverage may overlap.

We solve this problem using the **maximum flow approach** with a modified flow network.

If robots are allowed to overlap coverage of vertices, we do not enforce strict disjoint paths. Instead, we allow robots to start and stop arbitrarily, and the **maximum flow** in a modified graph gives the minimum number of robots required.

Algorithm

Step 1: Construct the Flow Graph To construct the flow network, we perform the following steps:

1. **Split Each Vertex:** For each vertex $v \in V$ in the original DAG:
   - Split v into two nodes: v_{in} (entry node) and v_{out} (exit node).
   - Add an edge $v_{\text{in}} \to v_{\text{out}}$ with capacity 1.
2. **Add Original Graph Edges:** For each directed edge $(u, v) \in E$ in the original DAG:
   - Add an edge $u_{\text{out}} \to v_{\text{in}}$ with capacity ∞.
3. **Connect Source and Sink:**
   - Add a source node s and a sink node t.
   - Connect $s \to v_{\text{in}}$ for all $v \in V$ with capacity ∞.
   - Connect $v_{\text{out}} \to t$ for all $v \in V$ with capacity ∞.

Step 2: Compute Maximum Flow Use a **maximum flow algorithm** (e.g., Ford-Fulkerson) to compute the flow from the source s to the sink t in the modified flow network. The value of the maximum flow represents the total number of robots required.

Step 3: Interpret the Flow The total incoming flow to the sink t gives the **minimum number of robots** needed. Each unit of flow corresponds to one robot serving the graph.

Working

- **Vertex Splitting:** Splitting each vertex into v_{in} and v_{out} ensures that each vertex can be ”served” individually by a robot.
- **Source and Sink Connections:** Connecting all v_{in} to the source s, and v_{out} to the sink t allows robots to start and stop at any vertex.
- **Flow Interpretation:** The maximum flow value represents the minimum number of robots needed to serve all vertices.

Final Answer

The minimum number of robots required is equal to the **maximum flow value** in the modified flow graph. This approach efficiently ensures all vertices are served, allowing robots to overlap coverage and start or stop at any vertex.