ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2024

Problem Set 2

Due: 23:59, Nov. 10, 2024

1. Submit your solutions to the course Gradescope.
2. If you want to submit a handwritten version, scan it clearly.
3. Late homeworks submitted within 24 hours of the due date will be marked down 25%. Homeworks submitted more than 24 hours after the due date will not be accepted unless there is a valid reason, such as a medical or family emergency.
4. You are required to follow ShanghaiTech's academic honesty policies. You are allowed to discuss problems with other students, but you must write up your solutions by yourselves. You are not allowed to copy materials from other students or from online or published resources. Violating academic honesty can result in serious penalties.

Problem 1:

In a small-town convenience store, the owner, Mr. Wang, enjoys arranging bags of candy by the counter for customers to pick from. Each candy bag contains a different number of candies; for instance, a small bag has 1 candies, a medium bag has 2, and the largest bag holds 5 candies. One day, a customer named Tony comes to the store hoping to buy an exact quantity of candies, such as 12 pieces, to satisfy his sweet cravings.

Mr. Wang tells Tony he can select any number and type of candy bags, as long as they add up precisely to the desired number of candies. You aim to find a way to help Tony obtain his exact candy count using the minimum number of bags. If it's impossible to achieve the exact number of candies with the available bags, you should inform Tony that his request cannot be fulfilled.

Determine the minimum number of candy bags Tony needs to reach his exact desired quantity. If it's not possible, return -1. Provide the state transition equation and the corresponding pseudocode.

Exapmle 1: the available candy bag types in the store are bags = [1, 2, 5], and Tony's desired quantity is pieces = 12. The output is 3, as the minimum solution requires taking two 5-candy bags and one 2-candy bag.

Example 2: bags = [2], pieces = 3, the output is -1.

Solution:

To solve this problem using dynamic programming (DP), we can use the following approach:

Define the DP Array: Let dp[i] represent the minimum number of bags required to get exactly i candies.

Initialize the Array: Set dp[0] = 0 because 0 candies require 0 bags. For all other values, initialize dp[i] = inf (a large number), representing an unreachable state initially.

Transition Equation: For each bag size in bags, and for each target quantity from bags[j] up to pieces, update dp[i] as follows:

dp[i]=min(dp[i],dp[i−bags[j]]+1)

This formula checks if we can reach i candies by using one more bag of size bags[j] on top of the minimum bags needed for (i - bags[j]).

Result: If dp[pieces] is still inf after filling the array, it means it's impossible to reach the target pieces with the given bags, so we return -1. Otherwise, dp[pieces] contains the minimum number of bags required.

```
def minBags(bags, pieces):
    # Initialize DP array
    dp = [float('inf')] * (pieces + 1)
    dp[0] = 0  # Base case: 0 candies require 0 bags

    # Fill the DP array
    for bag in bags:
        for i in range(bag, pieces + 1):
            dp[i] = min(dp[i], dp[i - bag] + 1)

    # Result
    return dp[pieces] if dp[pieces] != float('inf') else -1
```

Problem 2:

You are provided with three sequences A, B, and C. The lengths of these sequences are m, n, and $m+n$, respectively. In other words, sequence C has a length that equals the combined lengths of sequences A and B. Develop an algorithm to determine if the sequences A and B can be interleaved to form sequence C, while maintaining the relative order of characters in A and B.

Example 1: Given $A = aabb$, $B = cba$, and $C = acabbab$, the algorithm should return **true**.

Example 2: Given $A = aabb$, $B = cba$, and $C = aaabbbc$, the algorithm should return **false**.

Solution:

Let $M[i, j]$ denote whether the first i letters in A and the first j letters in B can be merged to form the first $i + j$ letters in C.

$M[i, j]$ = true if and only if ($M[i-1, j]$ = true and $A[i] = C[i+j]$) or ($M[i, j-1]$ = true and $B[j] = C[i+j]$).

Run DP.

Problem 3:

On a wall composed of a line of grid cells, there is a painting robot. The robot follows these two painting rules:

1. The robot can only paint using **one color** at a time, and it can only paint a continuous area of the same color.
2. The robot can start painting from any grid cell and can **overwrite** areas that have been previously painted.

Given a string s, where each character represents the color of a grid cell (e.g., "r" for red, "b" for blue, "g" for green), your task is to calculate the minimum number of painting operations needed for the robot to cover all the grid cells. Provide the state transition equation and the corresponding pseudocode.

Example 1:
Input: $s =$ "rrrbbb"
Output: 2
Explanation: First, paint the red area "rrr", then paint the blue area "bbb".
Example 2:
Input: $s =$ "rbr"
Output: 2
Explanation: First, paint the entire row with red "rrr", then overwrite the middle cell with blue "b".

Solution:

We can solve this problem using dynamic programming. Let $f[i][j]$ represent the minimum number of painting operations required to complete the section from index i to j.

When calculating $f[i][j]$, we consider the following cases:

1. **Boundary Condition**: If $i = j$, meaning the section has only one grid cell, then only one painting operation is required:

$$f[i][i] = 1.$$

2. **Same Color**: If $s[i] = s[j]$, meaning the colors at both ends are the same, the robot can cover $s[j]$ while painting $s[i]$. Therefore, we only need to consider how to complete the section from i to $j - 1$, giving:

$$f[i][j] = f[i][j - 1].$$

We don't need to worry about the exact painting plan for section $[i, j-1]$ because we can always complete $s[i]$ first and cover $s[j]$ simultaneously.

3. **Different Colors**: If $s[i] \neq s[j]$, meaning the colors at both ends are different, the robot needs to separately complete the left and right segments. We denote these segments as $[i, k]$ and $[k+1, j]$, where $i \leq k < j$. In this case:

$$f[i][j] = \min_{k=i}^{j-1}(f[i][k] + f[k+1][j]).$$

The complete state transition equations can be summarized as:

$$f[i][j] = \begin{cases} 1, & \text{if } i = j \\ f[i][j-1], & \text{if } s[i] = s[j] \\ \min_{k=i}^{j-1}(f[i][j], f[i][k] + f[k+1][j]), & \text{if } s[i] \neq s[j] \end{cases}$$

The final answer will be $f[0][n-1]$, where n is the length of the string s.

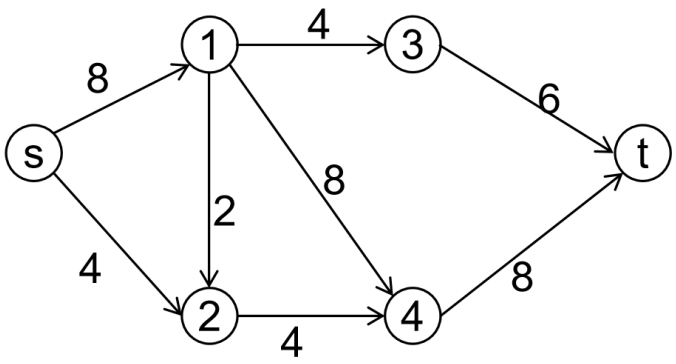
Algorithm 1 Robert Painter

```
n ← length of s
f ← 2D array of size n × n, initialized with 0
for i ← n − 1 to 0 do                          ▷ Fill the DP table from bottom to top
    f[i][i] ← 1                                 ▷ Base case: single character
    for j ← i + 1 to n − 1 do
        if s[i] = s[j] then
            f[i][j] ← f[i][j − 1]
        else
            min_value ← ∞
            for k ← i to j − 1 do               ▷ Find minimum value for split positions
                min_value ← min(min_value, f[i][k] + f[k + 1][j])
            end for
            f[i][j] ← min_value
        end if
    end for
end for
return f[0][n − 1]                              ▷ The result is stored in f[0][n − 1]
```

Time Complexity: $O(n^3)$, where n is the length of the string s.
Space Complexity: $O(n^2)$, where n is the length of the string s.

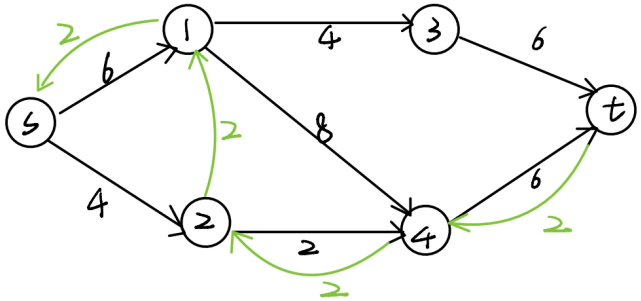
Problem 4:

Execute the Ford-Fulkerson algorithm on the flow network shown in the figure below. Display the residual network after each flow update and determine the maximum flow at the end. During each iteration, choose the lexicographically smallest available augmenting path. For example, if two augmenting paths are $s \to 2 \to t$ and $s \to 3 \to t$, select $s \to 2 \to t$ since it is lexicographically smaller. Likewise, if the paths $s \to 3 \to t$ and $s \to 2 \to 4 \to t$ are available, choose $s \to 2 \to 4 \to t$.

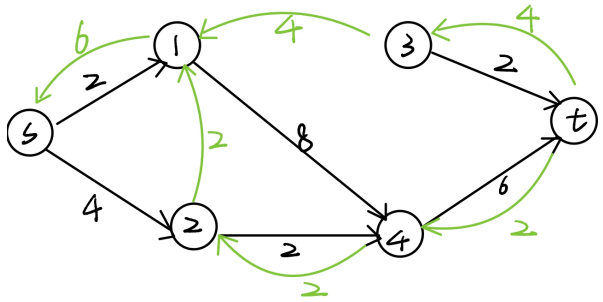


Solution:

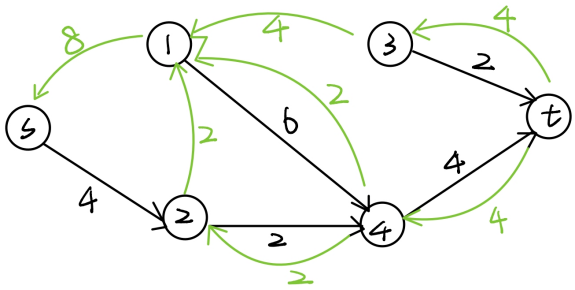
Step1: the augmenting path is s $\to$ 1 $\to$ 2 $\to$ 4 $\to$ t. The capacity of this path is min(8,2,4,8)=2.



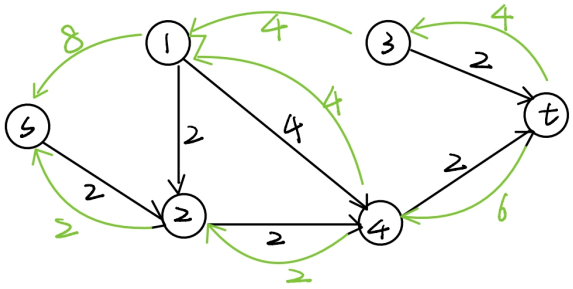
Step2: the augmenting path is s $\to$ 1 $\to$ 3 $\to$ t. The capacity of this path is min(6,4,6)=4.



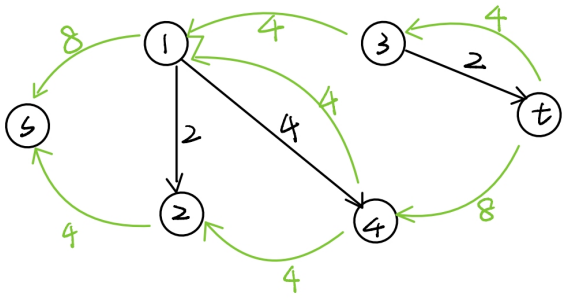
Step3: the augmenting path is s → 1 → 4 → t. The capacity of this path is min(2,6,8)=2.



Step4: the augmenting path is s → 2 → 1 → 4 → t. The capacity of this path is min(2,4,6,4)=2.



Step5: the augmenting path is s → 2 → 4 → t. The capacity of this path is min(2,2,2)=2.



The final maximum flow = 12.

Problem 5:

Yansen, a student at ShanghaiTech, recently started playing *Animal Crossing*. The game follows real-time (a week has 7 days).

There are n events, each requiring different materials, and each event is available on a few days of the week. Yansen can complete up to e events per day due to academic and research commitments.

Yansen wants to build a special Gundam and has calculated the materials needed. Starting on Monday, what is the minimum number of days (including rest days) to collect all the materials? (Hint: Network Flow)

Input Description:

The first line contains two integers, n, e ($1 \leq n \leq 1000$, $1 \leq e \leq 100$), separated by a space, representing the number of different events and the maximum number of events Yansen can complete in a day.

The next n lines, each containing $m_i + 2$ integers $c_i, m_i, a_{i1}, a_{i2}, \ldots, a_{im_i}$ ($1 \leq c_i \leq 10^5$, $1 \leq m_i \leq 7$, $1 \leq a_{ij} \leq 7$, for any $j \neq k, a_{ij} \neq a_{ik}$), represent the required number of completions c_i, and the days of the week when the event is available.

Output Description:

Output a single integer representing the minimum number of days Yansen needs to gather all the required materials.

Input:

5 2
2 4 1 3 5 7
2 3 2 4 6
2 2 1 2
2 2 3 4
2 2 5 6

Output:5

Explanation:

The solution can proceed as follows:
Day 1 (Monday): Complete events 1 and 3 once each.
Day 2 (Tuesday): Complete events 2 and 3 once each.
Day 3 (Wednesday): Complete events 1 and 4 once each.
Day 4 (Thursday): Complete events 2 and 4 once each.
Day 5 (Friday): Complete event 5 twice.
Exactly 5 days are needed to finish all requirements.

Solution:

Using Binary Search:

We treat the answer (minimum days) as a range and initialize it from $[0, 10^9/e]$, meaning the minimum is possibly 0 days, and the maximum is $10^9/e$ days.

We use binary search to repeatedly try different values of days, *mid*, to check whether all event requirements can be met within *mid* days.

Using Network Flow for Feasibility Checking

For each *mid* obtained from the binary search, use a network flow algorithm to determine if all events can be satisfied within *mid* days.

Construct a flow network graph that consists of the source, sink, day nodes, and event nodes, and solve the maximum flow problem.

1. Node Setup

Source node S and sink node T.

Day nodes: representing each day from 1 to 7. These are labeled from 3 to 9.

Event nodes: representing the n events, labeled from $9 + 1$ to $9 + n$.

2. Edge Setup

Connect the source node S to each day node. The capacity represents the maximum number of events that can be completed on that day. The capacity is calculated as:

$$\left(\left\lfloor \frac{\text{mid}}{7} \right\rfloor + [\text{mid mod } 7 \geq i] \right) \times e$$

This means there are $\lfloor \frac{\text{mid}}{7} \rfloor$ complete cycles of a week, and if mid mod $7 \geq i$, there will be one extra opportunity on day i.

Connect day nodes to event nodes, representing the days on which each event can be completed. The capacity is set to infinity, indicating that an event can be completed multiple times on a valid day.

Connect each event node to the sink node T. The capacity for these edges is set to the number of times the event needs to be completed, c_i.

3. Checking if the Conditions are Met

Calculate the maximum flow from the source S to the sink T.

If the maximum flow is greater than or equal to the total requirement $\sum c_i$, then it is possible to satisfy all the event requirements within mid days.

If the conditions are met, try reducing the number of days; otherwise, increase the number of days.

Problem 6:

Consider an $m * n$ matrix where each entry is a positive integer. The task is to choose a subset of the matrix's elements such that no two selected elements are adjacent. In this context, an element at position (i, j) is adjacent to the elements at $(i, j \pm 1)$ and $(i \pm 1, j)$, while it is not adjacent to the elements at $(i \pm 1, j \pm 1)$. Develop an efficient algorithm to maximize the total sum of the selected elements. Using the method of **network flow**, **design an efficient algorithm** to maximize the total sum of the selected elements and **prove its correctness**.

Solution:

Consider the matrix as a graph. Each element is a vertex and connects to its adjacent elements. Our problem now is converted to finding the maximum weighted independent vertex set in the graph. In any graph, the complement of an independent set is a vertex cover and vice versa, so our problem is equivalent to finding the minimum weight vertex cover in the graph. The latter can be solved using maximum flow techniques.

Define two sets X and Y. Put all elements in the matrix into one of these two sets so that there are no adjacent elements in the same set. Build a bipartite graph, and introduce a source s and a sink t:

1. Create a directed edge from s to each element in X. Every edge from s to the element in X has a capacity that is the same as the value of the current element.

2. Create a directed edge from each element in Y to t. Every edge from an element in Y to t has a capacity that is the same as the value of the current element.

3. Create directed edges from each element in X to its adjacent elements in Y. The capacity of these edges is infinity.

Now find the minimum $s-t$ cut in the constructed network. The value of the cut is the weight of the minimum vertex cover. To see why this is true, think about the cut edges: They can't be the edges connecting adjacent elements, because those have infinite capacity. If an edge that connects s and an element in X is cut, this corresponds to taking the corresponding element in X into the vertex cover. If we do not cut this edge, we need to cut all the edges from adjacent elements on the right side.

Thus, to actually reconstruct the vertex cover, just collect all the vertices that are adjacent to cut edges, and the required result is the sum of the elements in the matrix minus the maximum flow.