ShanghaiTech University

CS240 Algorithm Design and Analysis

Spring 2025

Problem Set 2

Due: 23:59, Apr. 1, 2025

1. Submit your solutions to the course Gradescope.
2. If you want to submit a handwritten version, scan it clearly.
3. Late submissions are not allowed.
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:

Imagine you are a stock market trader. Each day, the price of a particular stock is given by an array `prices` with length N , where `prices`$[i]$ represents the stock price on the i-th day ($i \in [0, N-1]$). You are allowed to perform at most $\texttt{k} < \frac{N}{2}$ transactions. Each transaction consists of buying the stock on one day and selling it on a later day. However, you cannot hold more than one stock at a time (i.e., you must sell the stock before buying again).

Your goal is to maximize your profit by strategically choosing when to buy and sell the stock, given the constraint on the number of transactions. Please design a dynamic programming algorithm to solve the problem. Please include the state transition function with base cases and analysis.

Example Scenario:

Suppose the stock prices over 6 days are given by:

$$\texttt{prices} = [3, 2, 6, 5, 0, 3],$$

and you are allowed to perform at most $\texttt{k} = 2$ transactions. The maximum profit in this case is 7, achieved by:

- Buying on day 1 (price 2) and selling on day 2 (price 6), for a profit of 4.
- Buying on day 4 (price 0) and selling on day 5 (price 3), for a profit of 3.

The maximum profit you can achieve is $4 + 3 = 7$.

Solution:

Let $\texttt{OPT}(i, j)$ represent the maximum profit achievable using at most i transactions up to day j. The recurrence relation for $\texttt{OPT}(i, j)$ is as follows:

$$\texttt{OPT}(i,j) = \max \begin{cases} \texttt{OPT}(i, j-1), & \text{Do not perform any transaction on day } j, \\ \max_{0 \le m < j} \left(\texttt{prices}[j] - \texttt{prices}[m] + \texttt{OPT}(i-1, m) \right), & \text{Perform a transaction on day } j. \end{cases}$$

- $\texttt{OPT}(i, j-1)$: The maximum profit up to day $j-1$ using at most i transactions (no transaction on day j).
- $\texttt{prices}[j] - \texttt{prices}[m] + \texttt{OPT}(i-1, m)$: The profit from buying on day m and selling on day j, combined with the maximum profit from $i-1$ transactions up to day m.

The maximum profit is given by max_k $\texttt{OPT}(k, n-1)$ (or just $\texttt{OPT}$(k, n-1) is also correct), where n is the number of days.

Problem 2:

Imagine a traveler arriving at a grand monument with n steps leading to a spectacular view. Eager to explore every possible path, the traveler can ascend the stairs by taking either one step or leaping two steps at a time. Each unique sequence of steps creates a different route to the top. How many ways can the traveler reach the summit? Please design a dynamic programming algorithm to solve the problem. Please include the state transition function with base cases and analysis.

Example Scenario:

Suppose there are 4 steps in total. we can achieved by:

- 1,1,1,1
- 1,1,2
- 1,2,1
- 2,1,1
- 2,2

There are 5 ways.

Solution:

Let $\mathrm{OPT}(j)$ denote the number of ways to reach the j^{th} stair. To reach stair j, the traveler can either:

- Take one step from stair $j-1$, or
- Take two steps from stair $j-2$.

This yields the recurrence:

$$\mathrm{OPT}(j) = \mathrm{OPT}(j-1) + \mathrm{OPT}(j-2) \quad \text{for } j \geq 2.$$

With the base cases defined as:

$$\mathrm{OPT}(0) = 1 \quad \text{and} \quad \mathrm{OPT}(1) = 1.$$

Thus, the number of ways to climb n stairs is given by $\mathrm{OPT}(n)$.

Problem 3:

Imagine you are a master craftsman with a unique rod of length n inches. Each segment of the rod has a different market value, as given by the array `price[]` with length n, where `price[i-1]` is the market value for rod of length i. Your challenge is to cut the rod into pieces such that you maximize your total profit. Which combination of cuts will fetch you the highest possible earnings? Please design a dynamic programming algorithm to solve the problem. Please include the state transition function with base cases and analysis.

Example Scenario 1:

Input: `price` = [2, 6, 10, 12, 15, 18, 21, 25] denotes the price for the rods whose length ranges from 1 inch to 8 inches.

Output: 26

Explanation: The maximum obtainable revenue is 26. This is achieved by cutting the rod into pieces of lengths 2, 3, and 3:

$$\texttt{price}[1] + \texttt{price}[2] + \texttt{price}[2] = 6 + 10 + 10 = 26.$$

Solution:

Let $\mathrm{OPT}(j)$ denote the maximum revenue obtainable for a rod of length j. For $j \geq 1$, the recurrence relation is given by:

$$\mathrm{OPT}(j) = \max_{1 \leq i \leq j} \Big(\texttt{price}[i-1] + \mathrm{OPT}(j-i) \Big). \tag{1}$$

With the base condition defined as:

$$\mathrm{OPT}(0) = 0. \tag{2}$$

Thus, the maximum revenue for a rod of length n is $\mathrm{OPT}(n)$.

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, select an augmenting path according to the following criteria:

1. **Lexicographical Order Criterion:** Among paths of equal length, select the lexicographically smallest one.
2. **Shortest Path Criterion:** Choose the augmenting path with the fewest number of edges.

When choosing augmenting paths, please follow the above criteria accordingly. For example:

- If the available augmenting paths are

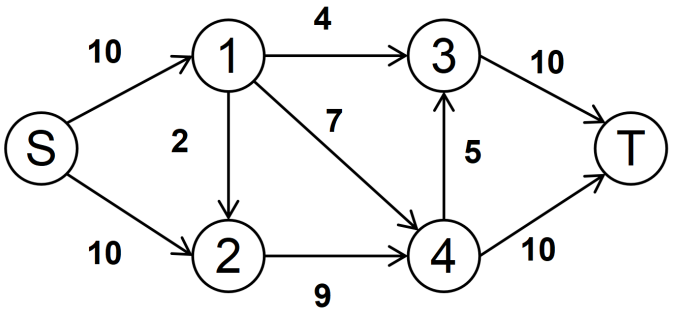
$$s \to 2 \to t \quad \text{and} \quad s \to 3 \to t,$$

  then select $s \to 2 \to t$ since 2 is lexicographically smaller than 3.

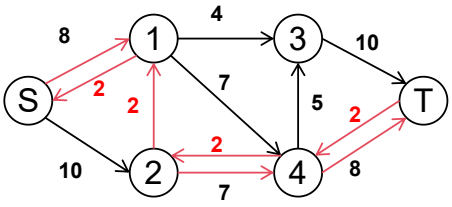
- If the available augmenting paths are

$$s \to 1 \to 2 \to 4 \to t \quad \text{and} \quad s \to 1 \to 2 \to 4 \to 3 \to t,$$

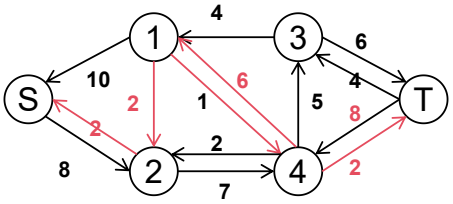
  then choose $s \to 1 \to 2 \to 4 \to t$ since it has fewer edges.



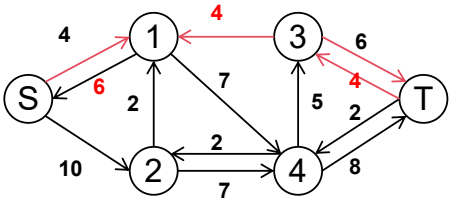
Solution:



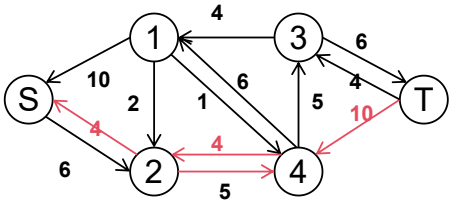
S→1→2→4→T maxflow=2



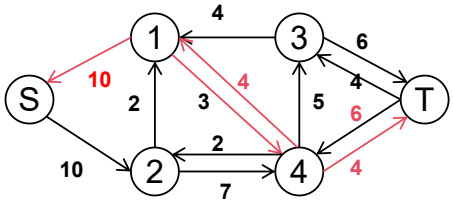
S→2→1→4→T maxflow=10+2=12



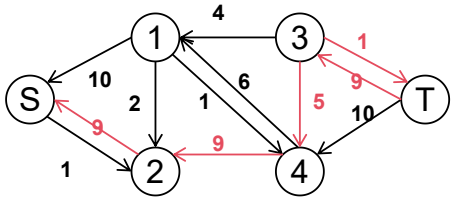
S→1→3→T maxflow=2+4=6



S→2→4→T maxflow=12+2=14



S→1→4→T maxflow=6+4=10



S→2→4→3→T maxflow=14+5=19

Problem 5:

Imagine you are an adventurer exploring a grid-shaped treasure map. The map is an $m \times n$ grid, where each cell contains a certain amount of treasure. Your goal is to collect as much treasure as possible, but there's a catch: you cannot collect treasure from adjacent cells. Two cells are considered adjacent if they share a common side (up, down, left, or right). In other words, two elements (i, j) and (k, l) are considered adjacent if they are directly next to each other horizontally or vertically, i.e., $|i - k| + |j - l| = 1$. Diagonal elements are not considered adjacent.

Design an efficient strategy to maximize the total treasure you can collect while respecting the adjacency constraint and describe your algorithm. (Hint: Network flow)

Solution:

Imagine a treasure map represented by a matrix, where each element denotes the amount of treasure hidden at that location. Our goal is to collect the maximum amount of treasure possible by selecting certain entries of the matrix, while following specific connectivity rules. We can solve this problem by constructing a network flow model as follows:

Network Construction

1. **Checkerboard Coloring:** Treat the matrix as a table and perform a black-white (checkerboard) dyeing of the cells.
2. **Node Representation:** Consider each matrix entry as a node in a network. Additionally, introduce two special nodes: a source s and a sink t.
3. **Source and Sink Connections:**
   - Create a directed edge from s to each **black** node, with the capacity of the edge set to the treasure amount at that entry.
   - Create a directed edge from each **white** node to t, with the capacity set to the treasure amount at that entry.
4. **Interconnecting Edges:** For every edge connecting a black node to an adjacent white node (according to the connectivity rules of the map), add a directed edge from the black node to the white node with ∞ (infinity) capacity.

Max Flow Computation and Treasure Extraction

Run the Ford-Fulkerson algorithm on the constructed network to compute the maximum flow f. Let v be the total sum of treasure amounts in the matrix. Then, the maximum collectible treasure is given by:

$$\text{Maximum Treasure} = v - f.$$

The selected treasure map entries correspond to those nodes for which the flow on the edge connecting the node with s or t is equal to its capacity, i.e., $f(e) = c(e)$.

Problem 6:

In a region, there are n factories producing goods of amount p_i and m villages that need these goods. The factories and villages are connected with roads (for any factory, there is a road to every village), each with a limited capacity c_i for transporting goods. Every village has a required amount of goods d_i. The goal is to determine if it is possible to distribute the goods in such a way that all villages receive their required amounts without exceeding the capacity of any road (demands for every villages are safisfied). Please describe your algorithm.

Solution:

To solve this problem, we can model it as a network flow problem. By adding a source node connected to all factories and a sink node connected to all villages, we can determine if a feasible circulation exists by checking if the maximum flow in the network equals the total demand of the villages.

To determine if a feasible circulation exists, we transform the problem into a maximum-flow problem and use a maximum-flow algorithm such as the Edmonds-Karp algorithm.

Algorithm Steps:

1. Construct the flow network $G = (V, E)$:
   - Add a source node s and connect it to each factory f_i with an edge of capacity p_i.
   - Add a sink node t and connect each village v_j to t with an edge of capacity d_j.
   - Retain the original roads between factories and villages with their respective capacities c_i.
2. Compute the maximum flow in G using a maximum-flow algorithm.
3. Check if the maximum flow value equals the total demand:

$$\text{Maximum flow value}(G) = \sum_{j \in V} d_j.$$

If true, a feasible circulation exists; otherwise, it does not.