Problem Set 3

Hiba Mallick - 24015

CSE 317 Design and Analysis of Algorithms
Spring 2024

1 Valuable Knapsack

To find the maximum total value of items that can be placed in the knapsack without exceeding its capacity and obtain a running time $O(nV)$ where V is the value of the optimal solution, we will modify our original approach from class with running time $O(nS)$.

Originally, we added memoization using a table of n rows and S columns where each row is an item and S is the capacity of the knapsack. In our table for this approach, we use a table with n rows (items) and $V + 1$ columns (minimum value needed). Each cell $K[i][j]$ stores the minimum size of a knapsack needed to achieve a value of j using the first i items.

To construct the table, we will fill each cell.
For $V \leq 0$, the minimum size will be 0.
For a single item with the value V_i, where V_i is greater than the target value j, the minimum size for that V_i will be infinity.
For cells where the size exceeds the capacity of the knapsack, the minimum size remains the same as when the current item was not included.
Else, we use a recurrence relation to compare the minimum size with the first $i - 1$ items and value j and the minimum size with the first $i - 1$ items and value $(j - V_i)$, plus the size (S_i) of the current item (i).

The recurrence relationship is as follows:

$$\begin{aligned}
&\text{for } j = 0, K[i][j] = 0 \\
&\text{for } i = 1 \text{ and } V_i > j, K[i][j] = \infty \\
&\text{for } K[i-1, j - V_i] + S_i > S, \quad K[i][j] = K[i-1][j] \\
&\text{else } K[i][j] = \min(K[i-1][j], K[i-1][j - V_i] + S_i)
\end{aligned}$$

Since V, the optimal solution, is unknown initially, we will fill the table column-wise starting from $j = 0$. For each cell in a column, we use the recurrence to find the minimum size based on previous cells. We stop when a column filled entirely with infinity is encountered.

Thus the optimal solution value (V) is the last column with at least one non-infinity value so the minimum size for this value (V) is $K(n, k)$.

This solution is in $O(nV)$ time since filling the table of dimensions $n \times (V + 1)$ takes $O(n \times (V + 1))$ time, which is equal to $O(nV)$.

2 Forbidden Program

(a) To compute the minimum number of letters that must be deleted from S to form a new string S', which has no common subsequence with F, we will use a lookup table. With the requirement that F has no repeated letters, we know that each letter in S occurs only once in F, so creating this lookup table will only cost us $O(m)$ for adding each letter of F.

When we use our lookup table to create the new string S', we will need to perform only n lookups since S only contains n letters, and we need to check if each element of S is present in F.

We will create a 2D array T of size i by j where each entry holds the minimum number of letters needed to be deleted from $S[1]$ to $S[i]$ to avoid the subsequence in $F[1]$ to $F[j]$. We will calculate the elements of T by checking if each $S[i]$ is equal to $F[j]$.

After constructing the lookup table for F:
For each letter $S[i]$ in S, perform a lookup of $S[i]$ in the lookup table.
If $S[i]$ is not in the lookup table, i.e., not in F, go to $S[i+1]$.
If $S[i]$ is in the lookup table, we will get $F[j]$, the element where $S[i]$ is present in F.
To consider subsequences of both more than length 1 and of length, we can decide whether to keep $S[i]$ in S or delete it.
We will use our $T[i][j]$ table to avoid the subsequence of F ending at $F[j-1]$ plus one for deleting $S[i]$.

Since creating the lookup table costs $O(m)$ and looking up each element costs $O(n)$, our total cost is $O(n+m)$.

(b) Now we will remove the requirement that F has no repeated letters and find a solution in $O(nm)$. We will create a 2D array T of size i by j where each entry holds the minimum number of letters needed to be deleted from $S[1]$ to $S[i]$ to avoid the subsequence in $F[1]$ to $F[j]$. We will calculate the elements of T by checking if each $S[i]$ is equal to $F[j]$.

Initially, $T[i][0]$ will be ∞ since infinite deletions are needed to make sure S does not have an empty subsequence. If $S[i] \neq F[j]$, then $S[1]$ to $S[i]$ does not contain the subsequence in $F[1]$ to $F[j]$ and we move on to $S[i+1]$. If $S[i] = F[j]$, we will either:

- Keep $S[i]$ and make sure that the sequence $S[1]$ to $S[i-1]$ does not contain $F[1]$ to $F[j-1]$, hence the minimum number of deletions is

$T[i-1][j-1]$.

- Delete $S[i]$ and thus the minimum number of deletions is $T[i-1][j]+1$ (since we are deleting $S[i]$). This step can be written as:

$$T[i][j] = \min(T[i-1][j-1], T[i-1][j]+1)$$

Since each entry of T takes constant time when running a top-down approach, and there are nm elements, this algorithm works in $O(mn)$.

3 Minimize My Pain

To find the solution that incurs the least pain via dynamic programming, we can break down our problem into subproblems. Since we have s number of shops, we will have s sets of gates where s is the number of shops that we have currently placed.

To find the optimal sets of these gates, we need to minimize the average distance between the gates and the shops. Since average $= \frac{\text{sum}}{s}$, we can use the sum of the distances and minimize it. To achieve this, we can place a shop in the middle of a set of gates. We can calculate the median of the distances from the gates we have picked to ensure we incur the least pain. Thus, the optimal placement for a shop within a selected set of gates will be the median of the distances of the gates, which can be found in $O(1)$ complexity.

In the case that there are more shops than gates, the pain incurred is 0 since each shop will be placed with each gate along the line.

To find the least pain incurred, we will construct a 2D array K of each row where each entry stores the total distance from each gate when a shop is optimally placed between gates i to j. This is done in $O(n \cdot 1)$ since finding optimal placement takes $O(1)$, and optimal placement is found for n gates.

To find the least pain incurred, we will use a recurrence relation with memoization to create an array $P[i][j]$ when j shops are placed between i gates; we need to find the total sum to divide it by the number of gates. For this, we will use the following algorithm.

```
if j = 1, then P[i][j] = K(1,i)
if j i then P[i][j] = 0
else P[i][j] = $\min_{1 \leq k \leq i-2}(P[k][j-1], X[k+1][i])$
```

We will fill the array bottom up, and once we calculate $P[n][s]$, we will divide the value obtained by n to find the least incurred pain. The complexity for filling each cell is $O(n)$, and since there are n^2 cells in the matrix, filling the entire table takes $O(n^3)$.

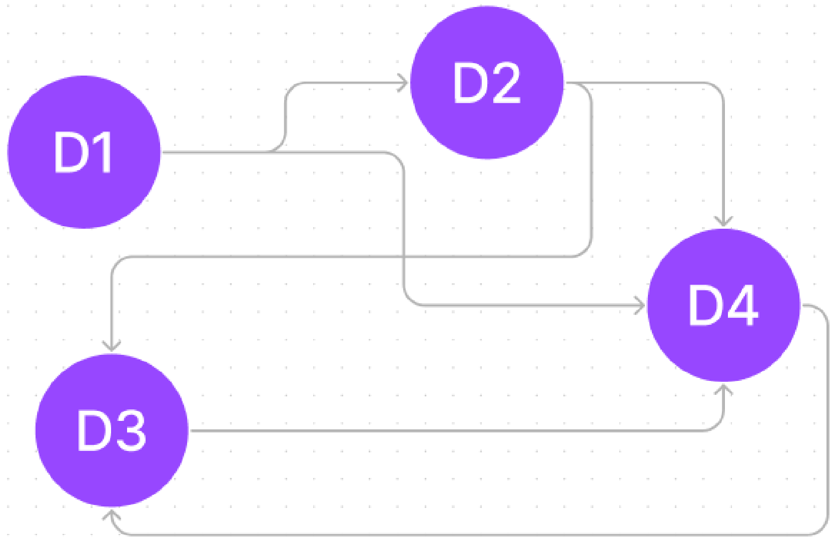

Figure 1: Q4 (a)

4 Graph Realization

(a) There is a directed graph with the specification which is shown in Figure 1 above. The graph is as follows:

- D1 goes to D2 and D4.
- D2 goes to D3 and D4.
- D3 goes to D4.
- D4 goes to D3

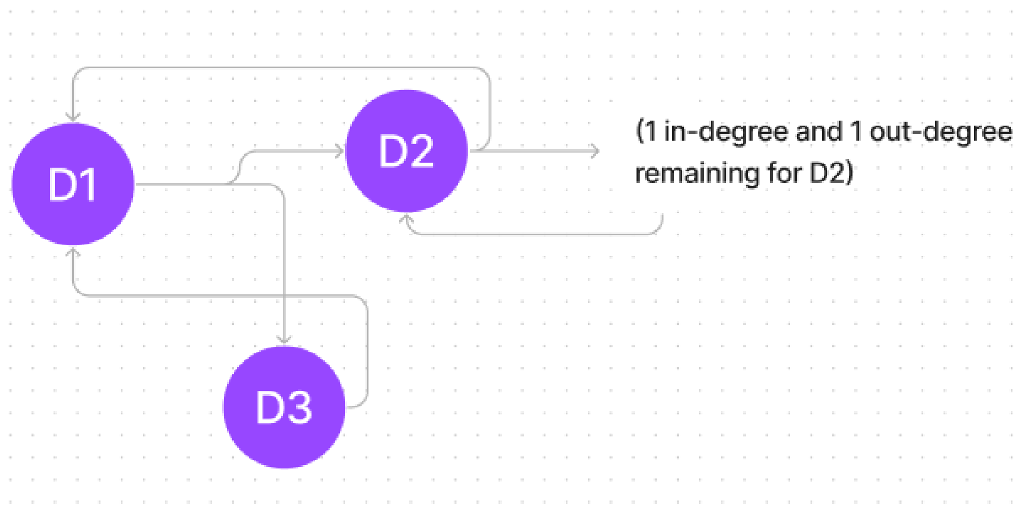

Figure 2: Q4 (b)

(b)
A directed graph cannot exist with the given specifications. As shown in the figure above, the 3 nodes cannot satisfy the given in and out-degrees specified so D2 is left with a remaining in-degree and an out-degree.

(c)
To find an efficient algorithm for the graph realization problem, we will construct a network flow graph as shown in Figure 3. This graph will have $2n$ internal nodes and a source node and sink node.

The nodes connected to the source are on level 1, and the paths from the source to each respective node have a capacity equal to the in-degree of that respective node.

Similarly, the nodes on level $d-1$, where d is the distance from the source to the sink, are connected directly to the sink node and have a capacity equal to the out-degree of that specific node.

The nodes on level 1 (i nodes) and $d-1$ (o nodes) are connected in a specific way. Each node $D[m]_i$ is connected to all $D[n]_o$ where $m \neq n$. This is to avoid self-loops in the graph realization. Each of the paths between the i nodes and o nodes has a capacity of 1.

After this construction, we can apply a specific optimization of the Ford-Fulkerson method, the Edmonds-Karp algorithm, to find the maximum flow

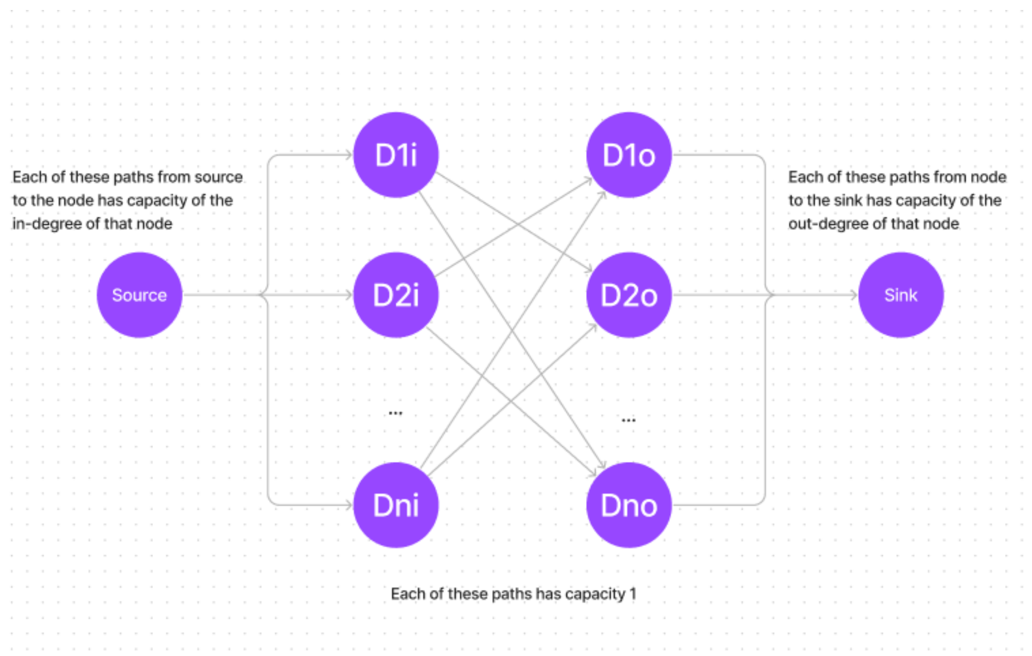
Each of these paths from source to the node has capacity of the in-degree of that node
Source
D1i
D2i
...
Dni
D1o
D2o
...
Dno
Each of these paths from node to the sink has capacity of the out-degree of that node
Sink
Each of these paths has capacity 1

Figure 3: Q4 (c)

over this network by iterating to completely saturate the graph.

In the algorithm, we will use this method to create an $m \times n$ adjacency matrix F for the graph we constructed. The matrix will have the i nodes in the rows and o nodes in the columns, and each entry will be the flow on that path.

We will traverse through all o nodes for each i node and
if $m = n$, then $F[m][n] = 0$ (to avoid self-loops)
else $F[m][n] = 1$ (if the edge is saturated).

For an additional check, we will sum up the non-zero elements in each row and each column of our adjacency matrix to ensure our graph is realizable. The sum of the non-zero elements on the ith row should be the in-degrees of the ith node. Similarly, the sum of the non-zero elements of the ith column should be the out-degrees of the ith node. If it is not, our graph is not realizable.