MID TERM
Colorado CSCI 5454

Rithik Kumar Athiganur Senthil

October 13, 2022

1 Problem 1 (10 points)

The *Ultimate Subsequence Problem* is this:

- **Input**: a sequence of positive integers given as an array A of length n.
- **Output**: yes" if there exists any subsequence of A that add up to exactly 42. Otherwise, “no”. If the answer is “yes”, also output the subsequence.

Give a dynamic programming solution to the Ultimate Subsequence Problem. You should specify all parts of a DP solution and prove correctness.

Note: recall that a subsequence is the result of deleting some entries from the original sequence.

Solution:
In order to solve the problem using dynamic programming approach we will use a 2-D boolean array where $array[i][j]$ will return if there exists a sum j considering a combination of the first i elements from the input array.

Input: Take input of an integer array A of length n.

Output: By considering the elements of the input array, form a subsequence such that the sum of the subsequence add upto 42.

We will use an 2-D boolean array named $dp[i][j]$ to store the values during the dynamic programming algorithm.

Subproblem: The boolean array $dp[i][j]$ returns 'yes' at any time if there exists a sum j using a combination of the first i elements in the input array, so $dp[i][j]$ will be true if there is such subsequence of the input array and false if it is not and output 'no'.

Computing final answer: Using the input array A, we can compute the values in the boolean array for every values starting from $0th$ row and column and finally compute the desired result $dp[n][42]$.

Recurrence:

- **Basecase:**

  $dp[i][0] = true \; \forall \; i \in$ 0,1,2,....,n
  $dp[0][j] = false \; \forall \; j \in$ 1,2,3,...,42

- **Inductive Case:**

  if $A[i-1] > j$,

  $$dp[i][j] = dp[i-1][j]$$

  else,

  $$dp[i][j] = dp[i-1][j] \; || \; dp[i-1][j - A[i-1]]$$

  Here, for every value $dp[i][j]$ we have two possibilities, either include the current element at i,j or exclude that elements and just take in account only till the previous case.

Algorithm 1 Ultimate Subsequence Problem

```
take input of a non-empty array 'A' of n integers
initialize an array for dynamic programming as dp[n+1][43]
for i in 0,1,2,....,n do
  set $dp[i][0] = true$
end for
for j in 1,2,....,42 do
  set $dp[0][j] = false$
end for
for i in 1,2,....n do
  for j in 1,2,.....42 do
    if $A[i-1] > j$ then
      set $dp[i][j] = dp[i-1][j]$
    else
      set $dp[i][j] = dp[i-1][j] \; || \; dp[i-1][j - A[i-1]]$
    end if
  end for
end for
if $dp[n][42]$ is true then
  call reconstruction algorithm
end if
return $dp[n][42]$ which is whether we can form a subsequence from array A of $n$ elements
to add upto 42, output 'yes' if true and 'no' is not.
```

Algorithm 2 Reconstruct Ultimate Subsequence Problem

```
initialize an empty array $prev[]$
Set $i = n$ and $j = 42$
while i > 0 $and$ j > 0 do
  if $A[i-1] \leq i$ then
    if dp[i][j] != dp[i-1][j] then
      Set $j = j - A[i-1]$
      Set $i = i - 1$
      Add $A[i-1]$ to $prev[]array$
    end if
    Set $i = i - 1$
  end if
end while
Return prev array which contains the elements taken.
```

Reconstruction: We will try to reconstruct by taking the elements from the input array, we will start from the last element in the $dp[][]$ array and store the elements in another

array named $prev[]$.
Starting from $dp[n][42]$ we traverse upward and look for the element at the top if $dp[i][j]$ is same as $dp[i-1][j]$ we will update i as $i-1$ until they are not same.
Then if $A[i-1] \leq j$ we will set j as $j - A[i-1]$ ans $i = i-1$.
We will perform this iteration until both i and j become 0 and keep updating the $prev[]$ array.

Correctness: By using the base case we already have the values of the 0th row and 0th column of $dp[][]$ array computed:

$$dp[i][0] = true \ \forall \ i \in \text{0,1,2,....,n}$$
$$dp[0][j] = false \ \forall \ j \in \text{1,2,3,...,42}$$

And we can find the value of $dp[i][j]$ using induction:

$$dp[i][j] = dp[i-1][j]$$

else,

$$dp[i][j] = dp[i-1][j] \ || \ dp[i-1][j - A[i-1]]$$

Like this by solving the subproblems we would have computed the correct value for whether the input array contains the subsequence that add upto 42 or not and return $dp[n][42]$.

2 Problem 2 (10 points)

2.1 Part a (6 points)

The *Ants Who Can't Share Problem* is this:

- We are given an unweighted, undirected graph G. The vertices represent key locations in the yard and edges represent being able to move between the locations.
- We are given two vertices s and t. The start vertex s is the ant nest and the end vertex t is a piece of cake.
- We must decide how to send out ants from the nest to reach the cake. Each ant must be given a path from the nest to the cake.
- Unfortunately, these ants *can't share*. Each ant must have the edges in its path all to itself. The other ants can't use any of this ants' edges. The paths are allowed to share vertices, since the ants are only there for a moment.
- Output: **What is the maximum number of ants we can send to the cake without making them share edges?**

Give a *reduction* from Ants Who Can't Share to a problem you already know how to solve from class. Briefly justify your solution (a full proof is not necessary).
Note: recall that a subsequence is the result of deleting some entries from the original sequence.

Solution:
Let's take an unweighted and undirected graph $G = (V, E)$:
Now, we will convert this as a weighted and directed graph which is bidirectional because:

We will initialize the capacities of all the edges as 1 because when one ant goes from source to the sink, another ant cannot share any edges in that path. The edges are bidirectional because we initially have an undirected graph, we don't know if the ant can travel from an edge u to v or from v to u. Now, we will run the Ford-Fulkerson algorithm until all the paths from source to sink are considered without a common edge among them.

The above reduction can be justified by:

For all the augmented paths there cannot be any common edges which means that no two ants will have the possibility to intersect except at the sink, so every augmented path can only be used by one ant.
The capacities of all the edges would be 1 which implies that the max flow for any augmented path would be 1.
Once the Ford-Fulkerson algorithm is completed, the max-flow will imply the maximum

number of ants that can reach the cake.

Let's take the below given graph $G = (V, E)$ for this case:

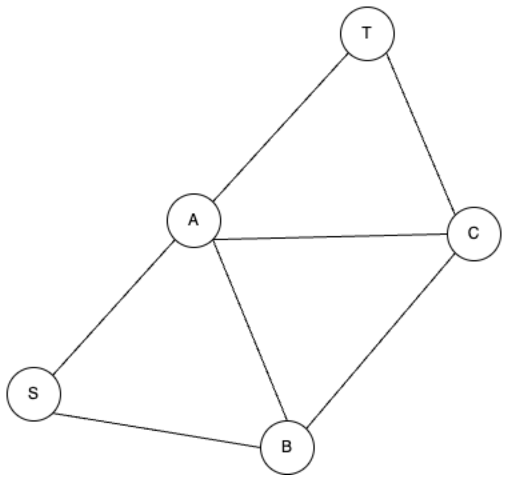

Figure 1: Graph

The above graph is converted to directed and weighted which is bidirectional:

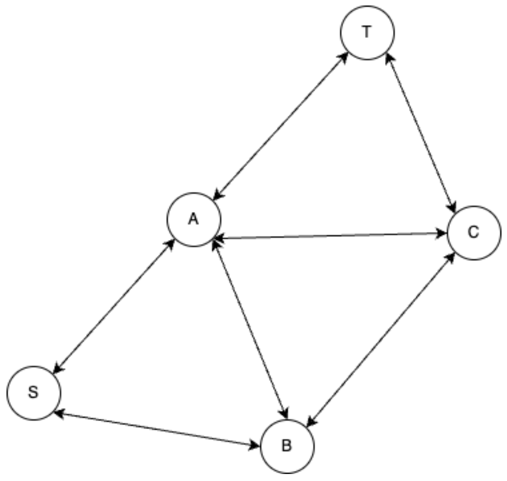

Figure 2: Graph

Initial stage:

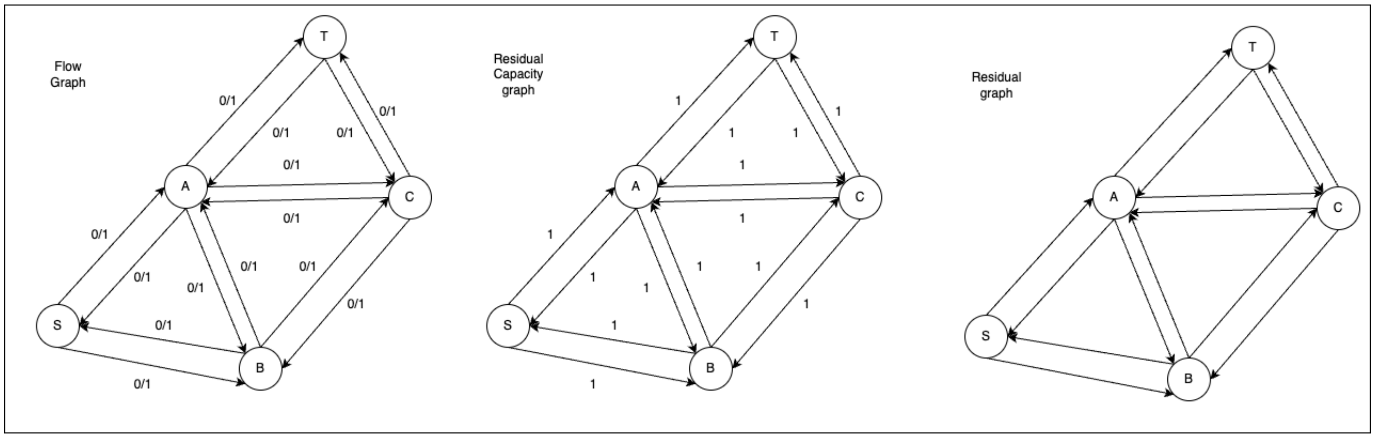

Figure 3: Graph

Cycle 1:
For the augmented path we will take the path S → A → T.

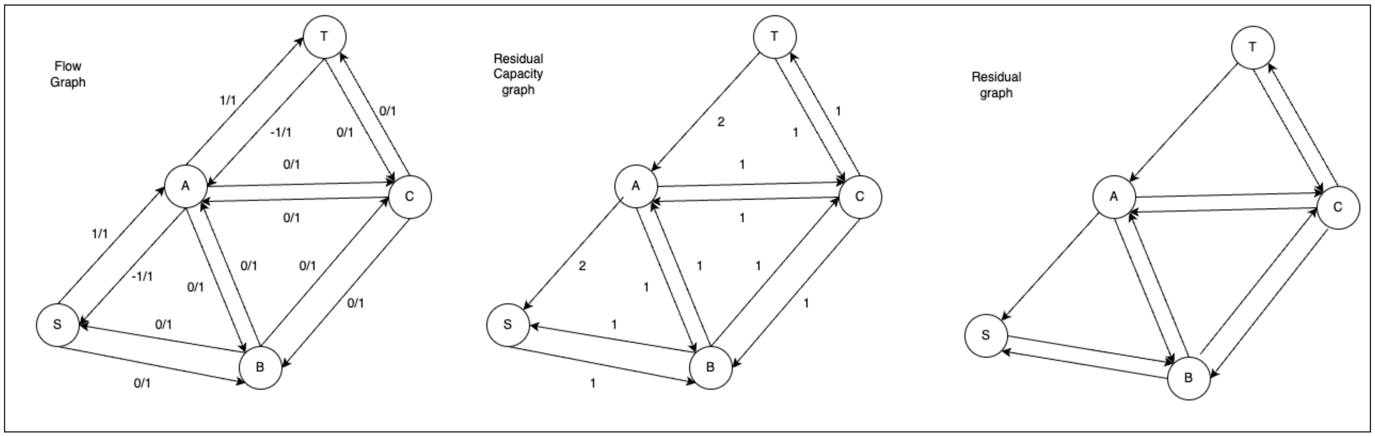

Figure 4: Graph

Cycle 2:
For the augmented path we will take the path S → B → C → T.

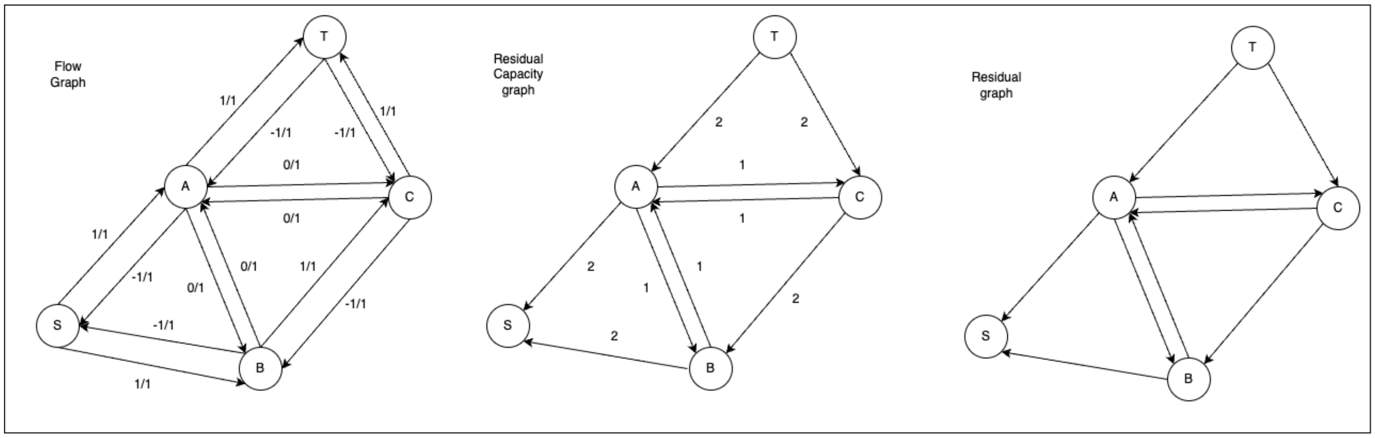

Figure 5: Graph

Once the Ford-Fulkerson algorithm has completed these 2 cycles there are no more augmented paths that does not have a common edge, so the algorithm stops here.
After these two cycles we have the max flow value as 2, hence at most two ants can reach to the cake.

2.2 Part b (1 bonus point)

Also explain how to output which paths that these ants should take (not just the number of ants).

Solution:

Here we can use DFS alogorithm, on the flow instance obtained after the Ford-Fulkerson algorithm to find the paths the ant will take to reach from source to sink.
For the example graph taken above we have the max flow as 2, Hence after using the DFS algorithm, we get these two paths:

S → A → T
S → B → C → T.

2.3 Part c (4 bonus point)

The *Blocking Ants Problem* is this:

- The input is an instance of the Ants Who Can't Share Problem, i.e. a yard graph G, nest location s, and cake location t.

- We don't want to let the ants eat the piece of cake, so we have decided to build barriers along some of the edges in the yard.
- Output: **What is the fewest number of edges we need to barricade to ensure no ant can reach the cake?**

Give an algorithm for the Blocking Ants Problem. Briefly justify your answer (a full proof is not necessary). You may heavily rely on your answer to Part (a), Ants Who Can't Share.

Solution:
In order to not let any ants taste the cake we have to put barricades cutting the edges, we need to partition the vertices into two sets N and M which will disconnect source and the sink.
In order to put barricades we will cut the edges such that the cut made cuts minimum number of edges.

The max flow min cut theorem states that the max flow is equal to min cut, in the below graph the min cut value is nothing but the number of edges in min cut.
Let's assume that we block less than number of edges in the min cut, then one of the min cut edge remains unblocked, each of the min cut will be there in one of the path of ant, as 2 paths cant share a same min cut edge or one ant cant take 2 min cut edges. Since atleast one will be unblocked, atleast one ant will reach the end. So our assumption is wrong and we conclude the min cut value or the max flow value is the minimum number of edges that needs to be blocked.

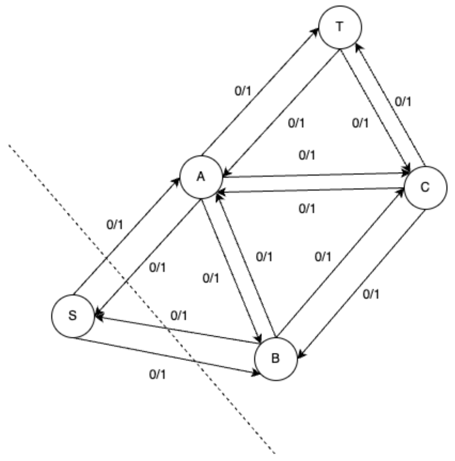

Figure 6: Graph

2.4 Part d (1 bonus point)

Also explain how to output which edges should be barricaded (not just the number of edges).

Solution:

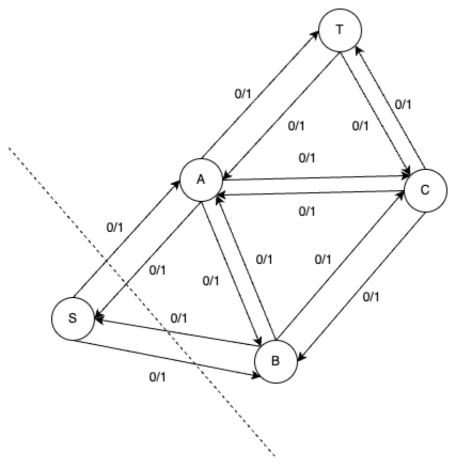

Figure 7: Graph

As per the example taken we have the min cut as given above:
So the sets N and M are:

N = {S}
M ={A,B,C,T}

In order to output which edges should be barricaded, we can leverage DFS here again: for all the vertices in the set N, we will check if there is an edge present crossing the cut from N to M, so if there is any edge present we store the edge for output.

Here the min cut value is 2 cutting 2 edges, so the vertex in set N, S is sharing two edges with A and B which is being cut from N to M.
Hence we output the edges as:

$$S \to A$$
$$S \to B$$