CS345 Theoretical Assignment 4 Submission

Naman Kumar Jaiswal
220687
namankj22@iitk.ac.in

Fall Semester 2024

Question 1: Ford-Fulkerson may fail to terminate on a network with 6 nodes and 9 edges with 1 irrational capacity edge

The following is an example of a network with 6 nodes and 9 edges with real capacities for the edges.

Example:

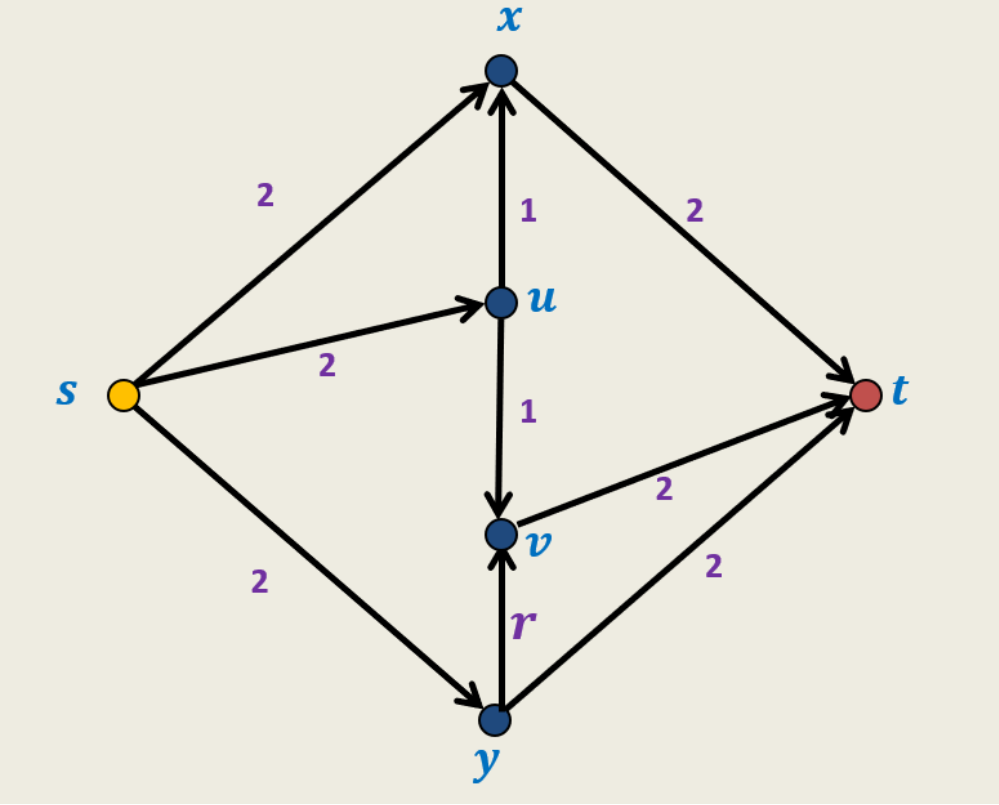

Figure 1: Network with 6 nodes and 9 edges with read weight capcities

We can have a network $G = (V, E)$ with capacity function c s.t. $c(x, y) \in R^+$ such that $V = \{s, x, u, v, y, t\}$ and edges are:

Start	End	Capacity
s	x	2
s	u	2
s	y	2
u	x	1
u	v	1
y	v	r
x	t	2
v	t	2
y	t	2

Table 1: Network edges with their capacity listed

Here, r is defined to be the solution of the equation

$$1 - r = r^2$$

$$r = \frac{\sqrt{5} - 1}{2}$$

$$r \approx 0.616$$

r, also called the golden ratio, is an irrational number. Also, by observation we can see that we can have the following flow to get a total network flow of 5. If we call $A = \{s, x, u\}$, we observe that the cut$(A, \overline{A})$ is the min-cut for the problem.There is no path from A to $\overline{A}$ in the residual network. Hence, we can claim 5 to be the max flow of the problem via min cut - max flow theorem. Ford-Fulkerson should terminate with an answer of 5 for this problem. Let us go for a dry run for this problem.

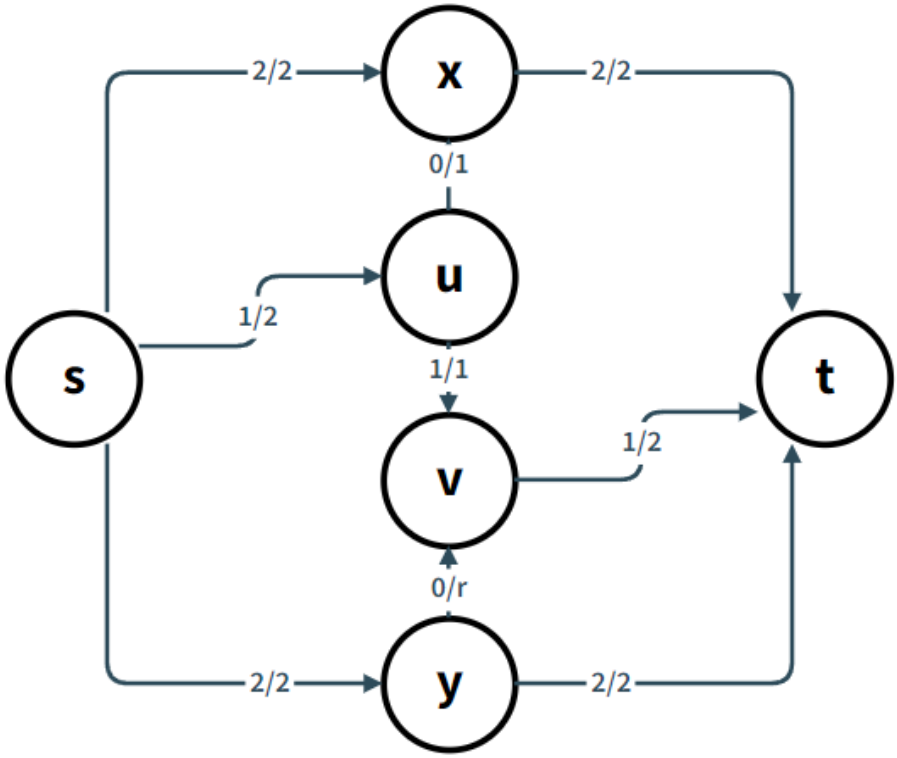

Figure 2: Network Flow with total flow of 5

Dry Run:

For our dry run let us begin by defining 3 augmented paths in the graph G, that will enable us to define a possible non-terminating sequence of augmented paths which the Ford-Fulkerson Algorithm might follow. We define:

$$p_1 = \{s, y, v, u, x, t\}$$

$$p_2 = \{s, u, v, y, t\}$$

$$p_3 = \{s, x, u, v, t\}$$

$$p_i = \{s, u, v, t\}$$

For our dry run, we are going to keep track of 4 parameters at each step - residual capacities for edges (u,x), (y,v) and (u,v) in the residual graph, Total Flow accumulated for the following sequence of augmenting paths - to conclude that Ford-Fulkerson shall never terminate here.

$$S = p_i p_1 p_2 p_1 p_3 p_1 p_2 p_1 p_3 p_1 p_2 p_1 p_3 \cdots (p_1 p_2 p_1 p_3)^n \cdots$$

To make our analysis cleaner and simpler, we are going to define the following functions:

$$f(n) = r^1 + r^2 + r^3 + r^4 + \cdots r^n = \sum_{i=1}^{n} r^i$$

$$g(n) = r^2 + r^4 + r^6 + \cdots r^n = \sum_{i=1}^{n/2} r^{2i}$$

Note: The inputs to the function g will be even for our analysis.

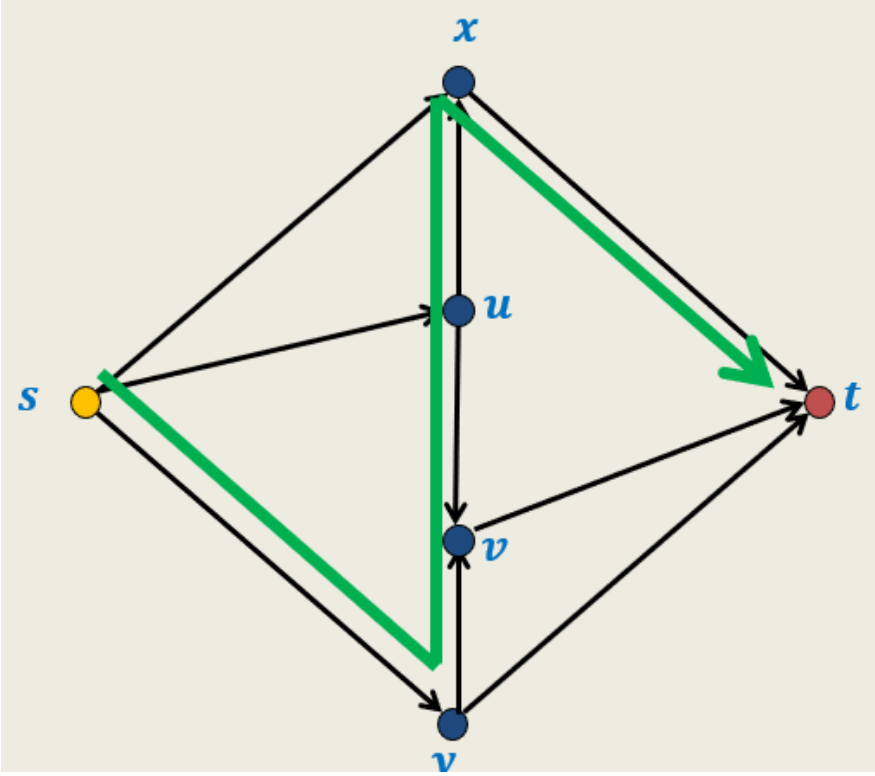

Figure 3: Path p_1

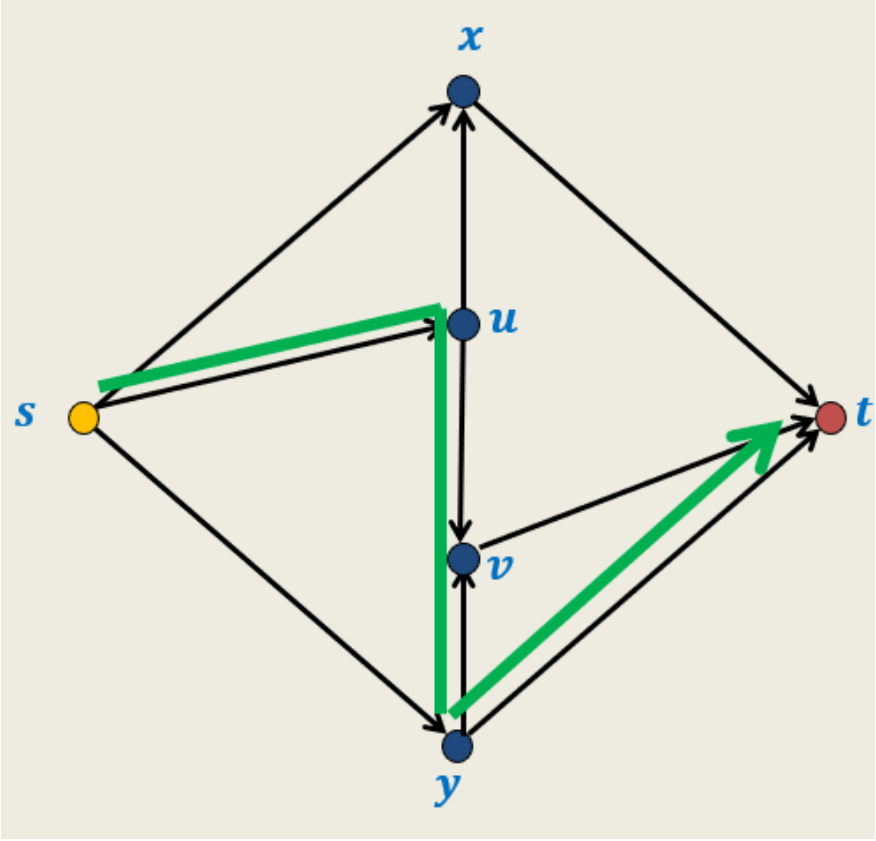

Figure 4: Path p_2

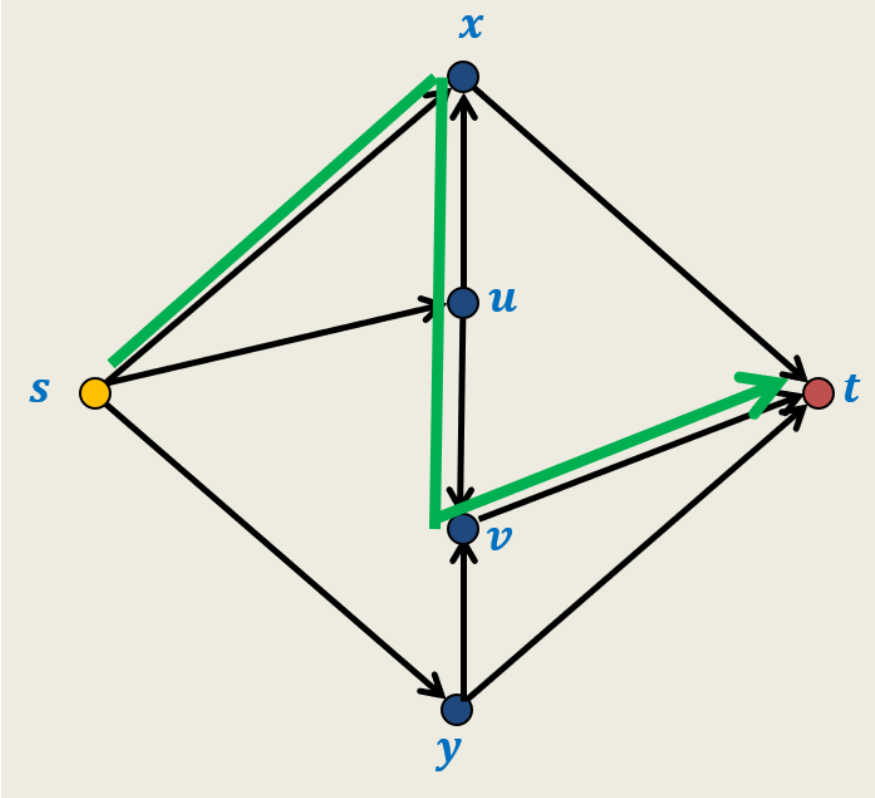

Figure 5: Path p_3

Step	**Augmenting Path**	**Bottleneck Capacity**	**Residual Capacities left in G_f**			**Total Flow Accumulated**
			(u,x)	(y,v)	(u,v)	
1	p_i	1	$r^0 = 1$	r	0	1
2	p_1	r^1	$r^0 - r^1 = r^2$	0	r^1	$1 + r^1$
3	p_2	r^1	r^2	r^1	0	$1 + 2r^1$
4	p_1	r^2	0	$r^1 - r^2 = r^3$	r^2	$1 + 2r^1 + r^2$
5	p_3	r^2	r^2	r^3	0	$1 + 2r^1 + 2r^2$
6	p_1	r^3	r^4	0	r^3	$1 + 2r^1 + 2r^2 + r^3$
7	p_2	r^3	r^4	r^3	0	$1 + 2r^1 + 2r^2 + 2r^3$
8	p_1	r^4	0	r^5	r^4	$1 + 2r^1 + 2r^2 + 2r^3 + r^4$
9	p_3	r^4	r^4	r^5	0	$1 + 2r^1 + 2r^2 + 2r^3 + 2r^4$
13	p_3	r^6	r^6	r^7	0	$1 + 2\sum_{k=1}^{6} r^k$
17	p_3	r^8	r^8	r^9	0	$1 + 2\sum_{k=1}^{8} r^k$

Table 2: Dry Run: Augmenting paths, Bottleneck Capacities, residual capacities and total flow at each step

Claim:

After $(1+4n)^{th}$ step of the algorithm, the total flow is $1 + 2\sum_{k=1}^{2n} r^k$ and the flow network looks like the following:

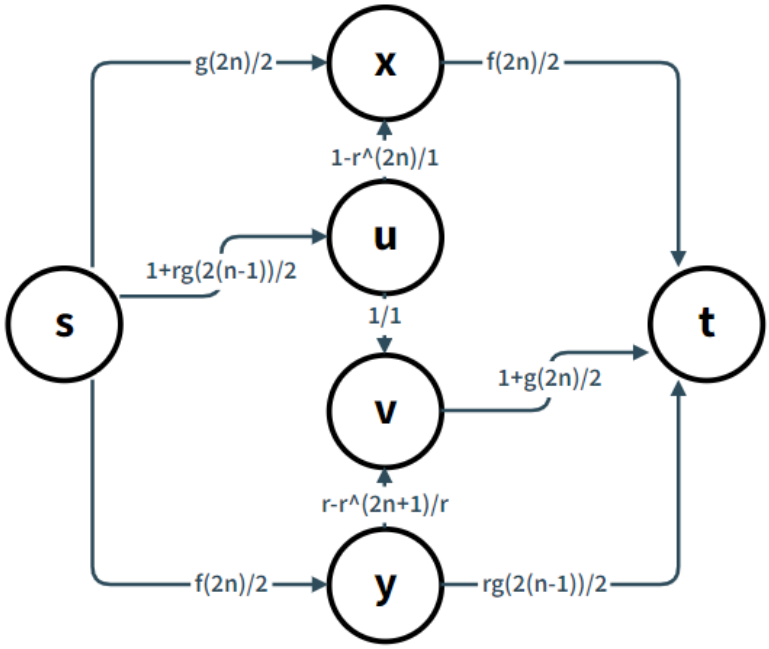

Figure 6: General flow after 1+4n steps

Proof:

Given the claim is true for the $(1+4n)^{th}$, prove it's correctness for the $(1+4(n+1))^{th}$ step:

1. Step 2 + 4n: we will choose the path p_1 and compute the bottle neck capacity as r^{2n+1} from the edge (y,v). We will send this flow along (s,y), (y,v), (v,u), (u,x), (x,t) and change the flow network.
   Residual capacities of (u,x): 1 - $(1 - r^{2n} + r^{2n+1}) = r^{2n+2}$
   (u,v): 1 - $(1-r^{2n+1}) = r^{2n+1}$
   (y,v): r - r = 0

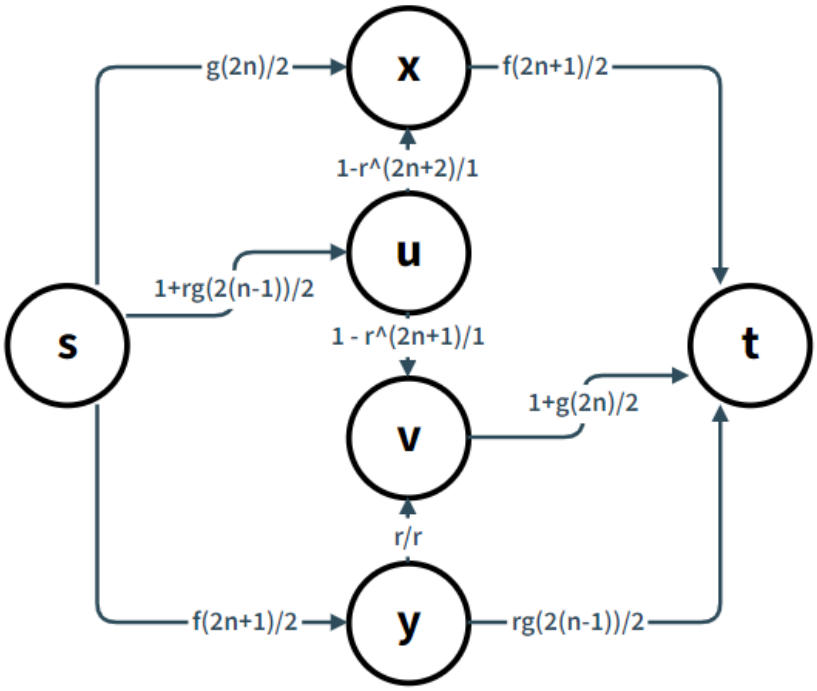

Figure 7: General flow after 2+4n steps

2. Step 3 + 4n: we will choose the path p_2 and compute the bottle neck capacity as r^{2n+1} from the edge (y,t). We will send this flow along (s,u), (u,v), (v,y), (y,t) and change the flow network.
   Residual capacities of (u,x): r^{2n+2}
   (u,v): 1 - (1) = 0
   (y,v): r -(r - r^{2n+1}) = r^{2n+1}

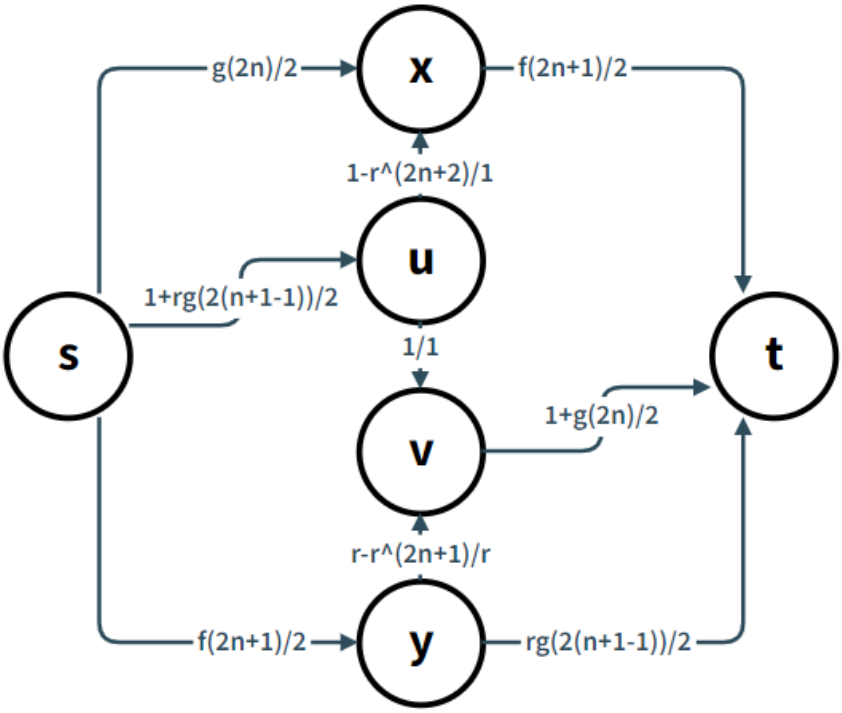

Figure 8: General flow after 3+4n steps

3. Step 4 + 4n: we will choose the path p_1 and compute the bottle neck capacity as r^{2n+2} from the edge (y,v). We will send this flow along (s,y), (y,v), (v,u), (u,x), (x,t) and change the flow network.
   Residual capacities of (u,x): 1 - 1 = 0
   (u,v): 1 - (1 - r^{2n+2}) = r^{2n+2}
   (y,v): r -(r - r^{2n+1} + r^{2n+2}) = r^{2n+3}

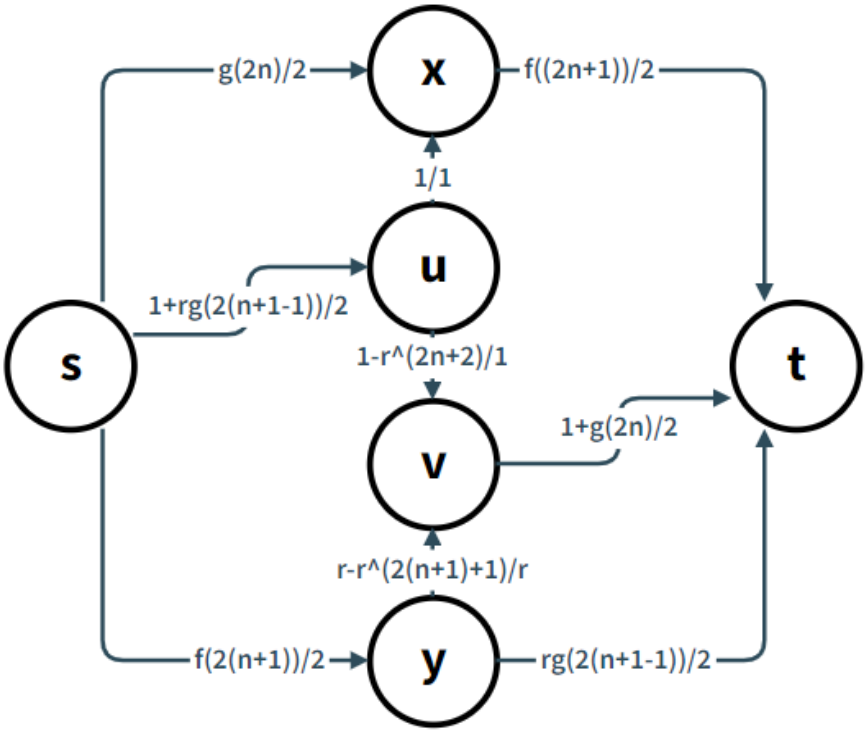

Figure 9: General flow after 4+4n steps

4. Step 1 + 4(n+1): we will choose the path p_3 and compute the bottle neck capacity as r^{2n+2} from the edge (y,v). We will send this flow along (s,x), (x,u), (u, v), (v,t) and change the flow network.
   Residual capacities of (u,x): 1 - (1 - $r^{2(n+1)}$) = r^{2n+2}
   (u,v): 1 - 1 = 0
   (y,v): r^{2n+3}

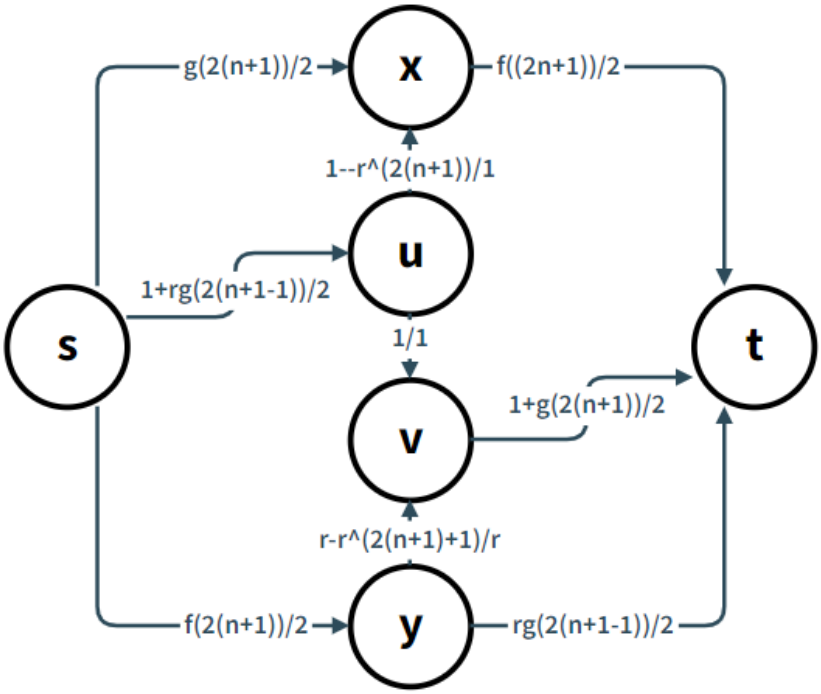

Figure 10: General flow after 1+4(n+1) steps

The Total flow is $1 + 2\sum_{k=1}^{2n} r^k + r^{2n+1} + r^{2n+1} + r^{2n+2} + r^{2n+2} = 1 + 2\sum_{k=1}^{2(n+1)} r^k$. If the flow network follows the generalisation at the n^{th} induction step, it will also follow it at the $(n+1)^{th}$ induction step.

Base Case (induction step 0 and 1):

This is trivial, we can directly compute and compare.

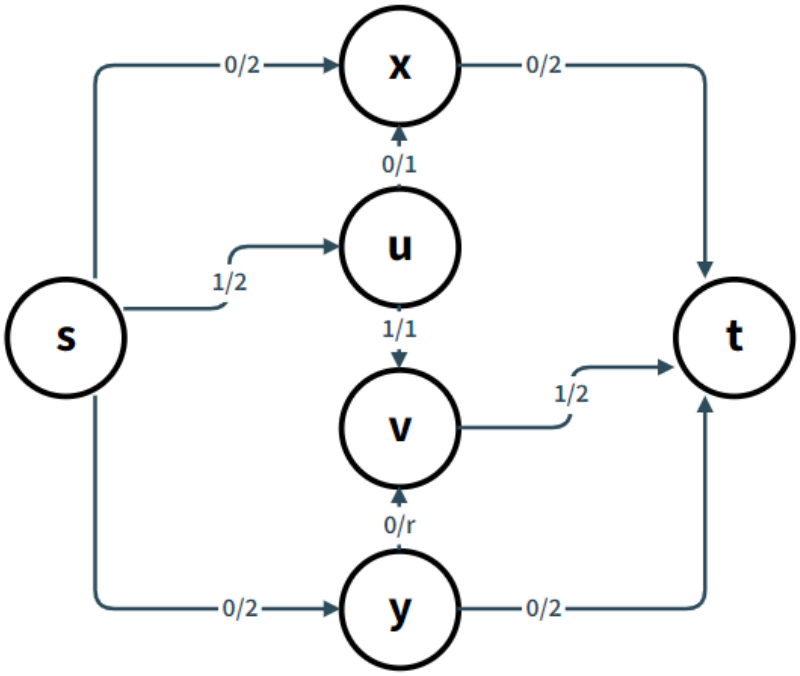

Figure 11: Flow after 1 step = 1

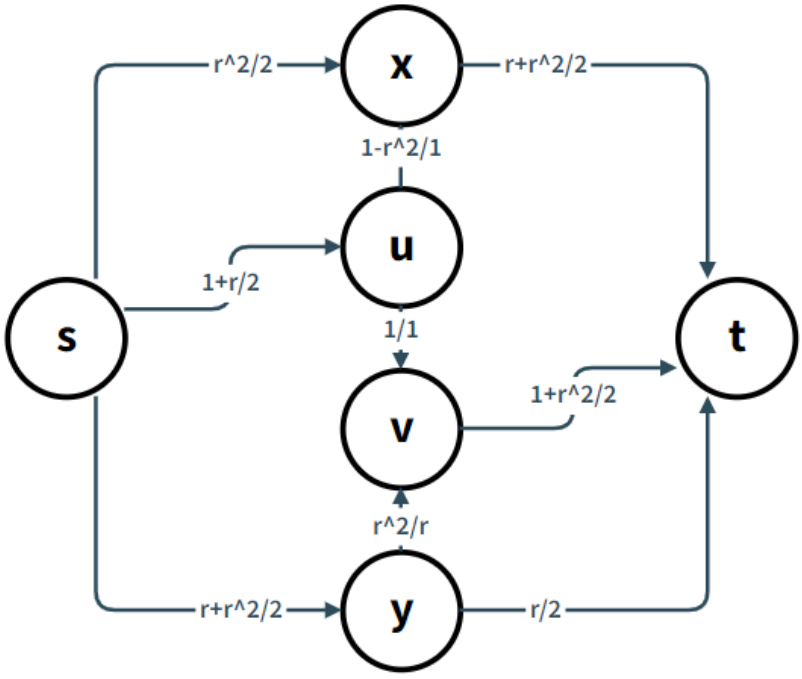

Figure 12: Flow after 5 steps = $1 + 2r^1 + 2r^2$

Conclusion:

Hence Proved, the total flow in the Ford Fulkerson algorithm after (1 + 4n) steps $= 1 + 2\sum_{k=1}^{2n} r^k \ \forall n \in N$

$$1 + 2\sum_{k=1}^{2n} r^k < 1 + 2\sum_{k=1}^{\infty} r^k = 1 + \frac{2r}{1-r} = 1 + \frac{2}{r} = 1 + 2(r+1) < 3 + 2r < 5$$

Ford-Fulkerson does not terminate for this problem if we pick the S sequence of paths, even after 1 + 4n steps, there exist augmenting paths $p_1p_2p_1p_3 \ \forall n \in N$. The maxflow will never reach near to 5.

Question 2: Dynamic Multi-Set (Amortized Analysis)

We need to design a data structure to support the following operations for a dynamic multiset S of integers, which allows duplicate values:

1. Insert(S, x): Inserts x into S
2. Delete-Larger-Half(S): Delete the largest $\lceil \frac{|S|}{2} \rceil$ elements from S
3. Report-Max(S): Report the largest element from the set S

The elements of the datastructure should be printable in $O(n)$ at all times, where n is the number of elements in the data structure.

Heuristic:

1. Since we need to implement $O(m)$ time complexity for a sequence of m operations, which is devoid of any log factor, we can safely conclude that the desired data structure is not a tree based data structure. We should try with a linear data structure.
2. We can find the median of an array in linear time, we can use this algorithm to potentially identify the largest $\frac{|S|}{2}$ elements of the array.

Data Structure:

We can use an unsorted array and a max variable for storing the elements. The array will store the elements in any arbitrary order (the ordering does not follow any pattern). The max variable stores the largest element of the array at all times. This variable can be updated on the go.
The used data structure is Linear with the size $O(n+1) \equiv O(n)$.

Operations:

We will use the following Operation definition:

- **Insert(S,x):**
  - Insert x into the the unsorted array directly.
  - Update the max-variable to store the maximum of the prior maximum and x.

  Note: If the max-variable stored the maximum element before this operation, it will do the same after insertion as well. If the size of unsorted array was n, it will now be n+1.
- **Delete-Larger-Half(S):**
  - Compute the median of the array using median in Linear time algorithm (Quick Select).
  - Partition the elements around the median.
  - Create a new empty array and insert the elements smaller than median into the new array. Make sure to include $|S|$ - $\lceil \frac{|S|}{2} \rceil$ into the new array. There may be duplicate values.
  - Compute the maximum of this new array and store it in the max-variable. Delete the original array and call the new array and new max-variable as the data structure.

  Note: If the max-variable stored the maximum element before this operation, it will do the same after insertion as well. If the size of unsorted array was n, it will now be $\lceil \frac{n}{2} \rceil$. A detailed analysis of median in linear time is given at the end as a form of appendix.
- **Report-Max(S):**
  - Just report the value stored in max-variable.

Note: We have made no changes in the values stored in the data structure. If the max-variable stored the maximum element before this operation, it will do the same after insertion as well. If the size of unsorted array was n, it will still be n.

- **Print(S):**
  - Just report the elements of the array.

  Note: This proves that the data structure is printable in $O(n)$ at all times.

Pseudocode:

1. Initialising the data-structure:

```
Input: None
Output: Empty Array and Empty Variable, together Empty Data Structure

InitializeDataStructure():
    // Initialize an empty array to store elements
    S ← []

    // Set max_variable to None as there are no elements
    max_variable = None

    return S, max_variable
```

Listing 1: Initialization of Data Structure

2. Inserting x into the datastructure:

```
Input: Array S, Max-variable max_variable, Element x
Output: Updated Array S with x inserted, Updated max_variable

Insert(S, max_variable, x):
    // Append x to the array S
    S.append(x)

    // Update max_variable if necessary
    if max_variable is None or x > max_variable then
        max_variable ← x

    return S, max_variable
```

Listing 2: Insert Element x into Data Structure

3. Reporting max-element of the datastructure:

```
Input: Array S, max_variable
Output: Maximum element in S

Report-Max(S, max_variable):
    // Check if max_variable is defined
    if max_variable is None then
        return "Error: Data structure is empty"

    // Return the maximum element
    return max_variable
```

Listing 3: Report Maximum Element from Data Structure

4. Deleting-Larger-half of the datastructure:

```
Input: Array S, max_variable
Output: Array S with the largest ⌈|S|/2⌉ elements removed, updated max_variable

Delete-Larger-Half(S):
    // Check if the array is empty
    if S is empty then
        return S, max_variable

    // Step 1: Find the median using a Linear Time Median Algorithm
    median = FindMedian(S)

    // Step 2: Partition S around the median
    left_half ← []

    // number of elements
    n ← |S|

    for element in S do
        if element <= median then
            left_half.append(element)

    // Step 3: Update S to only include the smaller half
    S ← left_half[1:⌊n/2⌋]

    // there will be ⌊n/2⌋ elements here

    // Step 4: Update max_variable to the maximum element in the updated S, if
     S is not empty
    if S is not empty then
        max_variable ← max(S)
    else
        max_variable ← None

    return S, max_variable
```

Listing 4: Delete Larger Half of Array

5. Printing the elements of the datastructure:

```
Input: Array S, max_variable
Output: Elements of S

Print(S, max_variable):
    // Check if the array is empty
    if S is empty then
        print "The data structure is empty"

    // Iterate over each element in S and print it
    for element in S do
        print element
```

Listing 5: Print Elements of Data Structure

Correctness:

- In each operation, the max-variable is correctly updated or recomputed if needed and stored as none for empty arrays. Thus the Report-Max function can be argued to be correct.
- In the delete operation, we only take the elements less than equal to the median and only take the first $\lfloor \frac{|S|}{2} \rfloor$ elements. This argues the correctness of Delete Operation.

- Arguing the correctness of Insert operation is trivial.

Time Complexity Analysis

1. Insert Operation: It only involves appending an element at the end of an array and then updating the max-variable with the max of previous value and the new element. Appending take $O(1)$ time, updating max-variable takes $O(1)$ time. In total $O(1)$ time.

2. Delete Operation: It involves by first computing the median of the array in Linear time. This is done in $O(n)$ (More info at the end). Then we partition the array around the median, this takes O(n) time as well. We will create a new array and copy the first $\lfloor \frac{n}{2} \rfloor$ elements into the new array and deleting the old one. This takes $O(n)$ time. Finally, we update the max-variable over the new array. This takes $O(n)$ time as well. In total, this operation also takes $O(n)$ time.

3. Report Max Operation: It only involves returning the max-variable value. This can be done in $O(1)$ time. In total this operation takes $O(1)$ time.

Amortized Cost Analysis

For amortized we will begin by defining the following:

1. let k be chosen such that, time taken by Insert Operation = $O(1) \leq$ k units, time taken by Report Max Operation = $O(1) \leq$ k units and time taken by Delete-Larger-half Operation = $O(n) \leq$ k*n units, where n is the number of elements in the data structure.

2. We will define our potential function Φ(i) = 2k*n, where n is the number of elements in the data structure and Φ(i) denotes the potential after i operations.

3. We assume that the sequence begins from an empty data structure. This is a necessary so that our potential function attains $\Phi(0) = 0$.

Now we can begin our analysis. We must see that the potential function chosen is a valid potential function because

1. $\Phi(0) = 0$ is satisfied for the base condition.

2. Φ(i) ≥ 0 $\forall i$ is satisfied as the number of elements in the data structure is always positive.

Let us define:

$$\text{Amortized cost of i}^{th} \text{ operation} = \text{Actual cost of i}^{th} \text{ operation} + \Phi\text{(i)} - \Phi\text{(i-1)}$$
$$\text{Amortized cost of m operations} = \sum_{i=1}^{m} [\text{Actual cost of i}^{th} \text{ operation} + \Phi\text{(i)} - \Phi\text{(i-1)}]$$
$$\text{Amortized cost of m operations} = \text{Actual cost of m operations} + \Phi\text{(n)} - \Phi(0)$$
$$\text{Amortized cost of m operations} = \text{Actual cost of m operations} + \Phi\text{(n)}$$
$$\text{Amortized cost of m operations} \geq \text{Actual cost of m operations}$$

Now we proceed with amortized cost computation:

1. Insert: Amortized cost < k + 2k(n+1 - n) < k + 2k
   The amortized of insert operation is less than 3k units.

2. Delete: Amortized cost < k*n + 2k($\lfloor \frac{|S|}{2} \rfloor$ - n) < k*n + 2k(-$\lceil \frac{|S|}{2} \rceil$) < 0 units. The amortized cost of delete operation is less than 3k units.

3. Report Max: Amortized cost < k + 2k(n-n) < k units. The amortized cost of Report Max is less than 3k units.

We can conclude that the

Amortized cost of any operation < 3k units
Amortized cost of m operations < 3k*m units
Actual cost of m operations < Amortized cost of m operations < 3k*m units

Hence, we conclude that the total cost is $O(m)$ bounded.

Median in Linear Time Algorithm

Referred from: ESO207 lectures and Online Source
To efficiently find the median of an unsorted list of integers, we can utilize the Quickselect algorithm. Quickselect operates similarly to Quicksort, selecting a pivot and partitioning the list into elements less than and greater than the pivot. The median can then be determined in linear time.

The Quickselect algorithm is defined as follows:

```
def quickselect(l, k, pivot_fn):
    if len(l) == 1:
        assert k == 0
        return l[0]

    pivot = pivot_fn(l)

    lows = [el for el in l if el < pivot]
    highs = [el for el in l if el > pivot]
    pivots = [el for el in l if el == pivot]

    if k < len(lows):
        return quickselect(lows, k, pivot_fn)
    elif k < len(lows) + len(pivots):
        return pivots[0]
    else:
        return quickselect(highs, k - len(lows) - len(pivots), pivot_fn)
```

Listing 6: Quickselect Algorithm

To find the median, we can use the following function, which utilizes the Quickselect algorithm. This function accounts for both odd and even lengths of the list.

```
def quickselect_median(l, pivot_fn=random.choice):
    if len(l) % 2 == 1:
        return quickselect(l, len(l) // 2, pivot_fn)
    else:
        return 0.5 * (quickselect(l, len(l) / 2 - 1, pivot_fn) +
                      quickselect(l, len(l) / 2, pivot_fn))
```

Listing 7: Finding the Median

Additionally, a better pivot selection strategy can be implemented to ensure linear time complexity in all cases. The `pick_pivot` function groups elements into chunks, finds the median of each chunk, and then recursively selects the median of the medians as a pivot.

```
def pick_pivot(l):
    if len(l) < 5:
        return nlogn_median(l)

    chunks = chunked(l, 5)
    full_chunks = [chunk for chunk in chunks if len(chunk) == 5]
    sorted_groups = [sorted(chunk) for chunk in full_chunks]
    medians = [chunk[2] for chunk in sorted_groups]

    return quickselect_median(medians, pick_pivot)
```

Listing 8: Pivot Selection Algorithm

Finally, we can implement a helper function to split the list into manageable chunks.

```
def chunked(l, chunk_size):
    return [l[i:i + chunk_size] for i in range(0, len(l), chunk_size)]
```

Listing 9: Chunking Helper Function