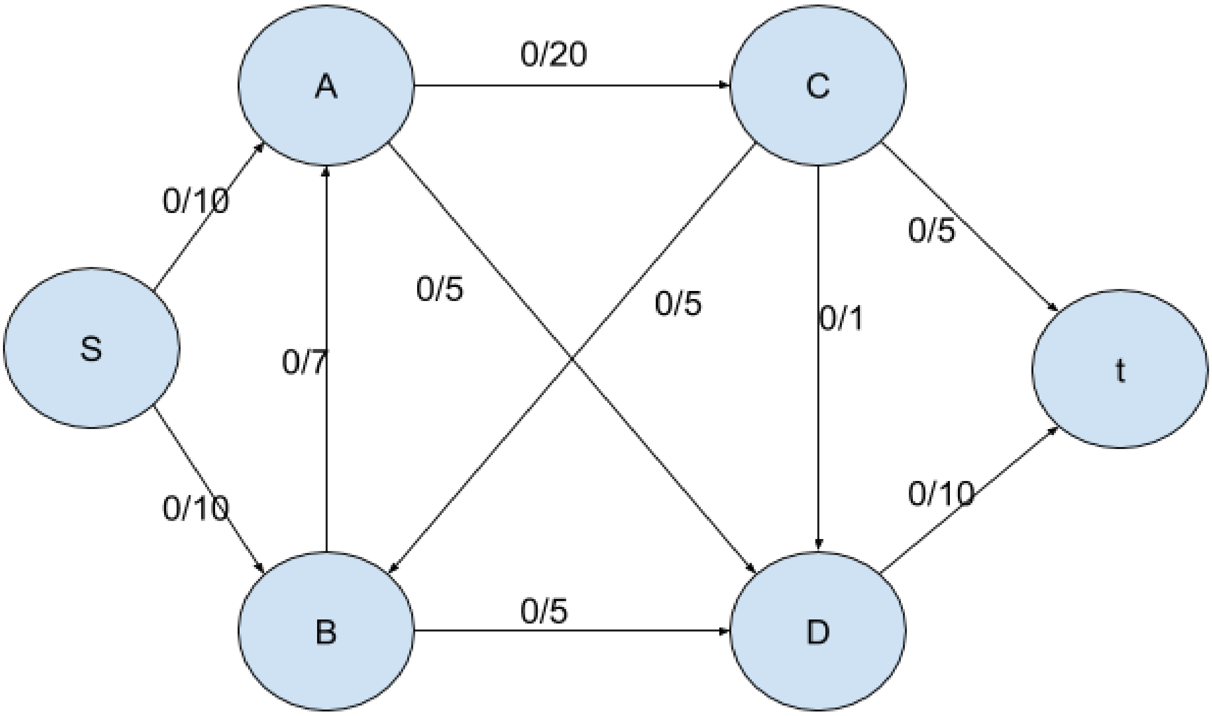
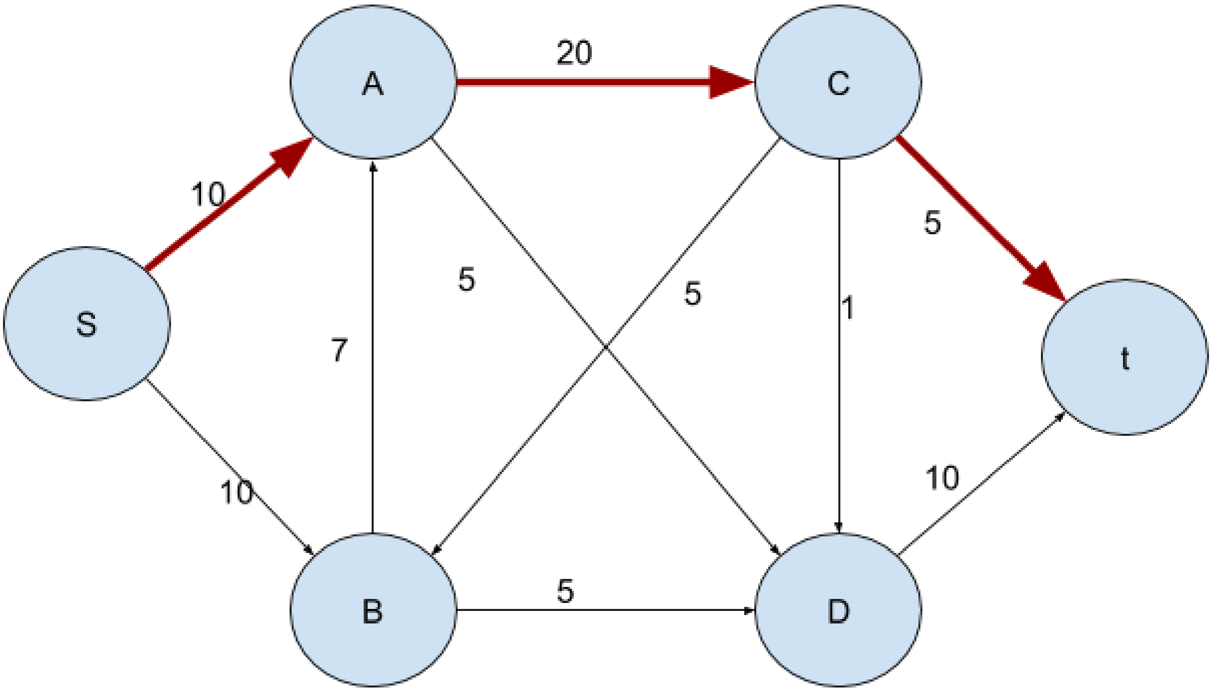
CS4522 Advanced Algorithms

Assignment 02 140106T - K.G.S. De Silva

1) (a) Using Ford Fulkerson to find max flow



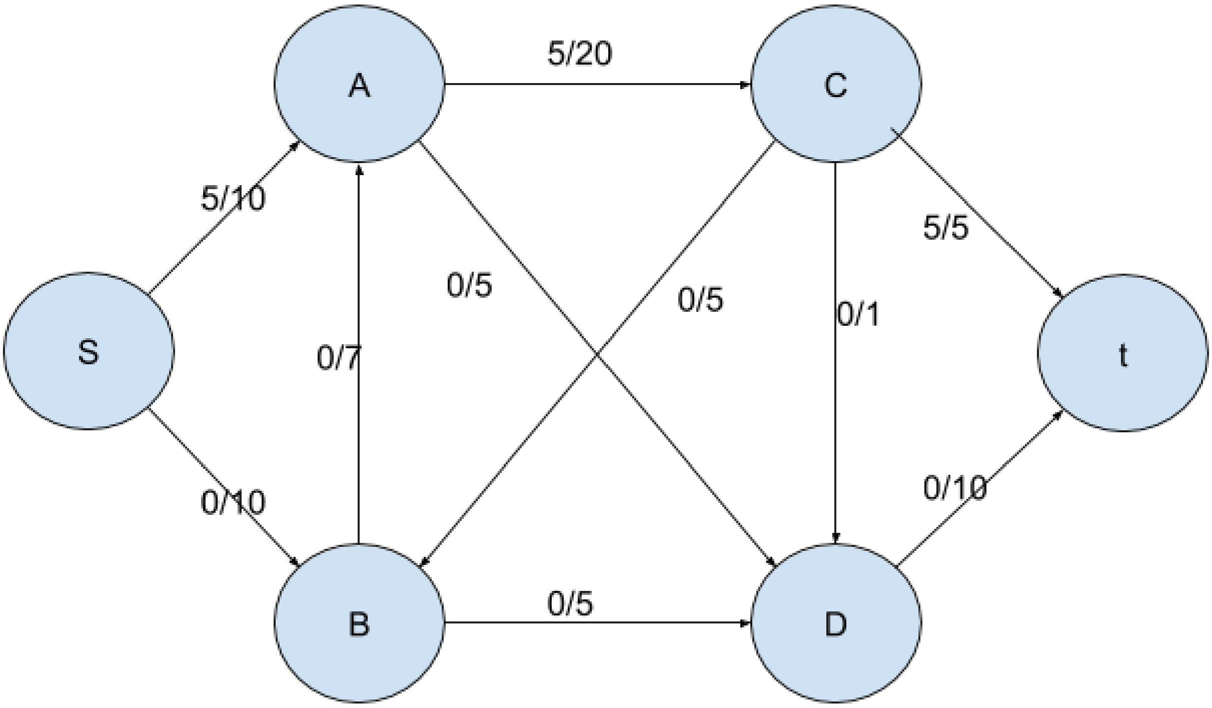
Flow Network



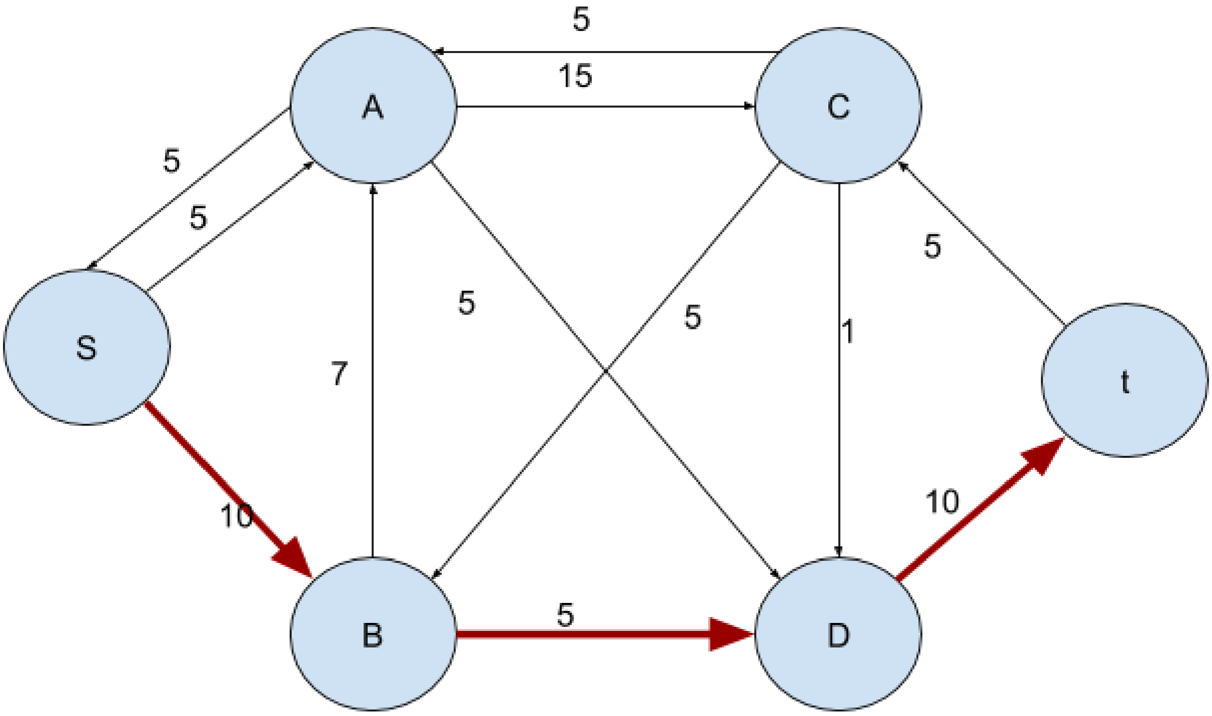
Residual Network 1 : Augmenting path is brown and weighted

$c_f(p) = min(10,\ 20, 5) = 5$

CS4522 Advanced Algorithms

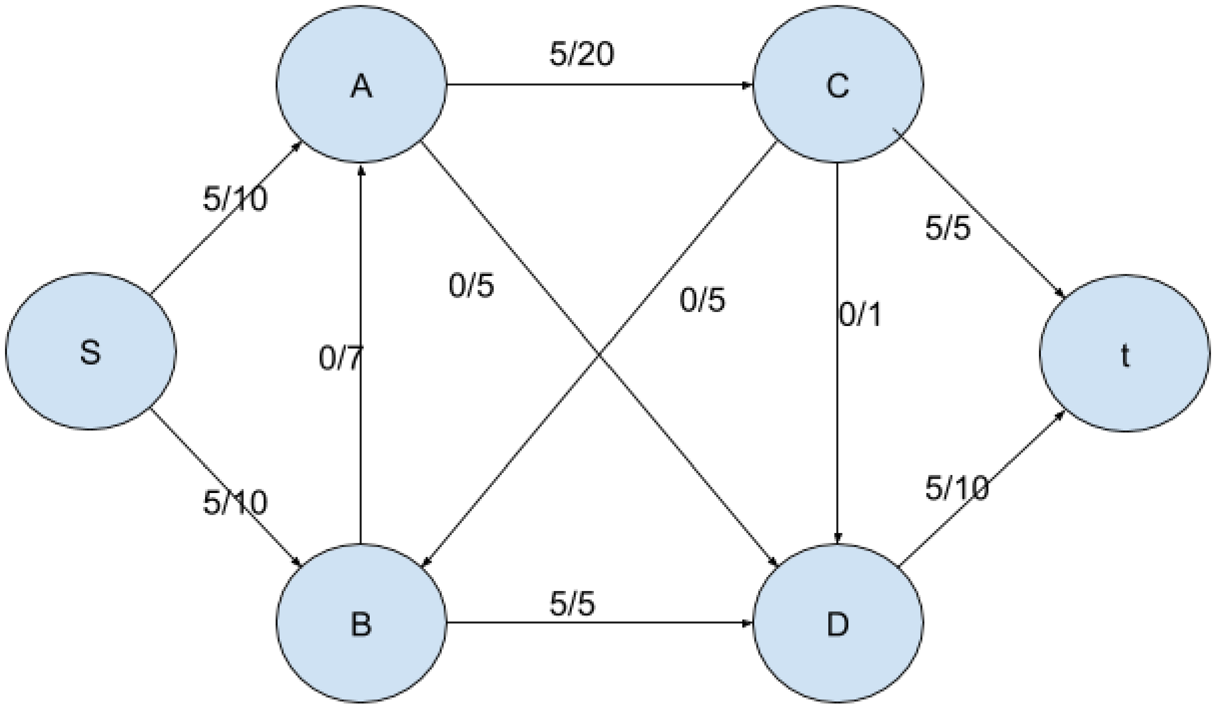


Flow Network 2

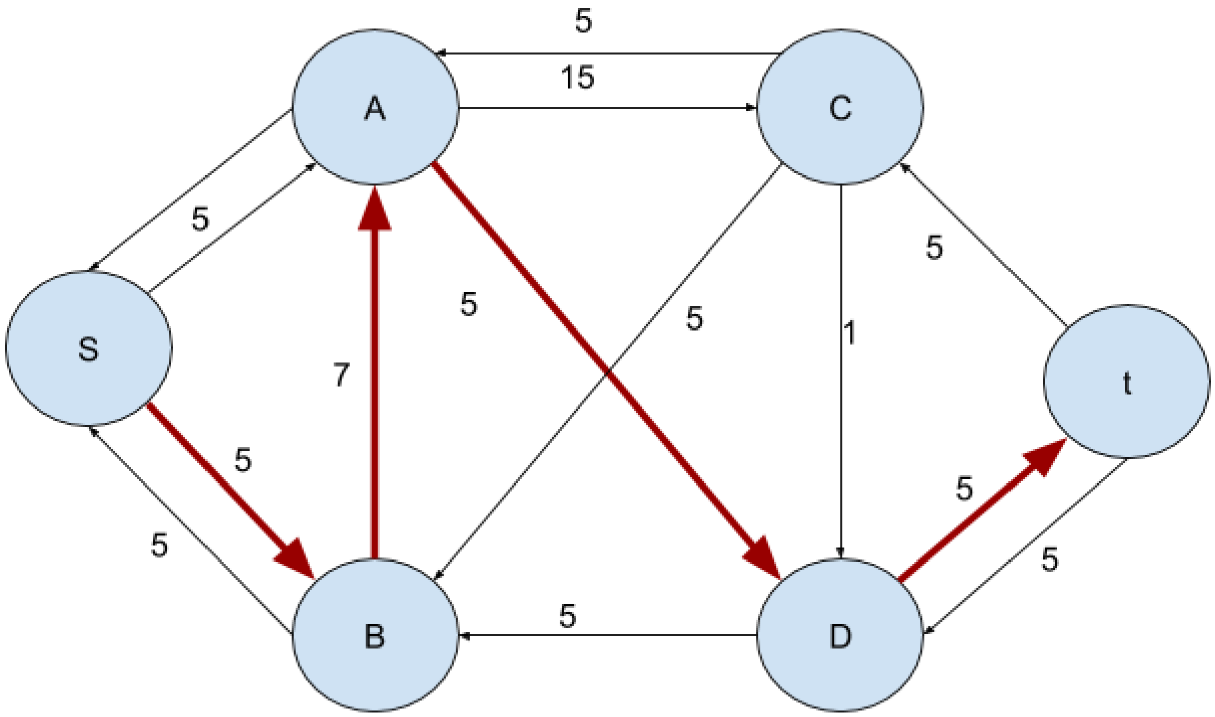


Residual Network 2 : Augmenting path is brown and weighted

$c_f(p) = min(10, 5, 10) = 5$

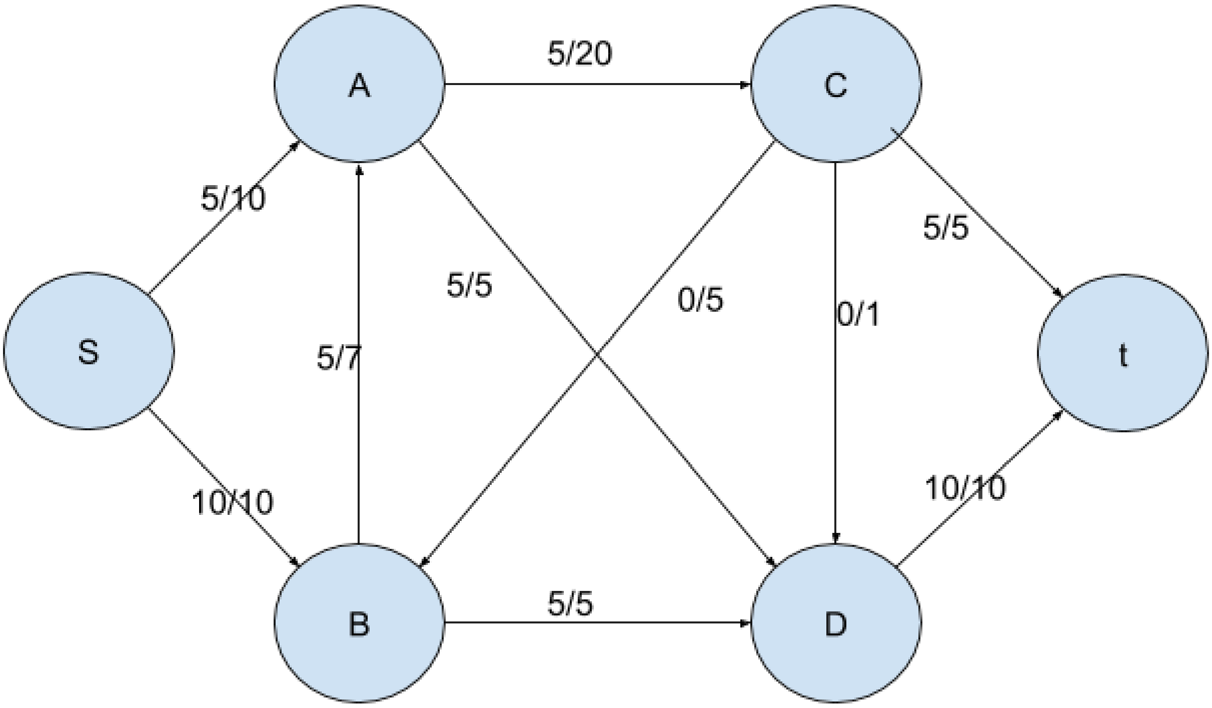


Flow Network 3

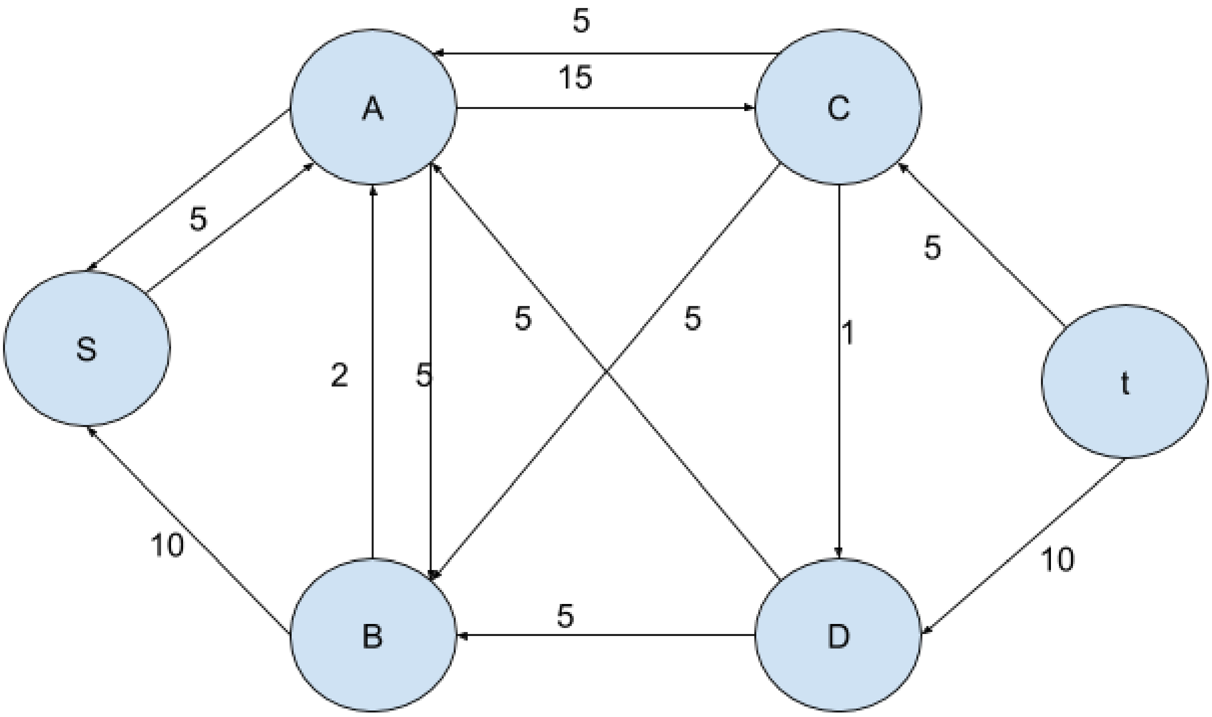


Residual Network 3 : Augmenting path is brown and weighted

$$c_f(p) = min(5, 7, 5, 5) = 5$$



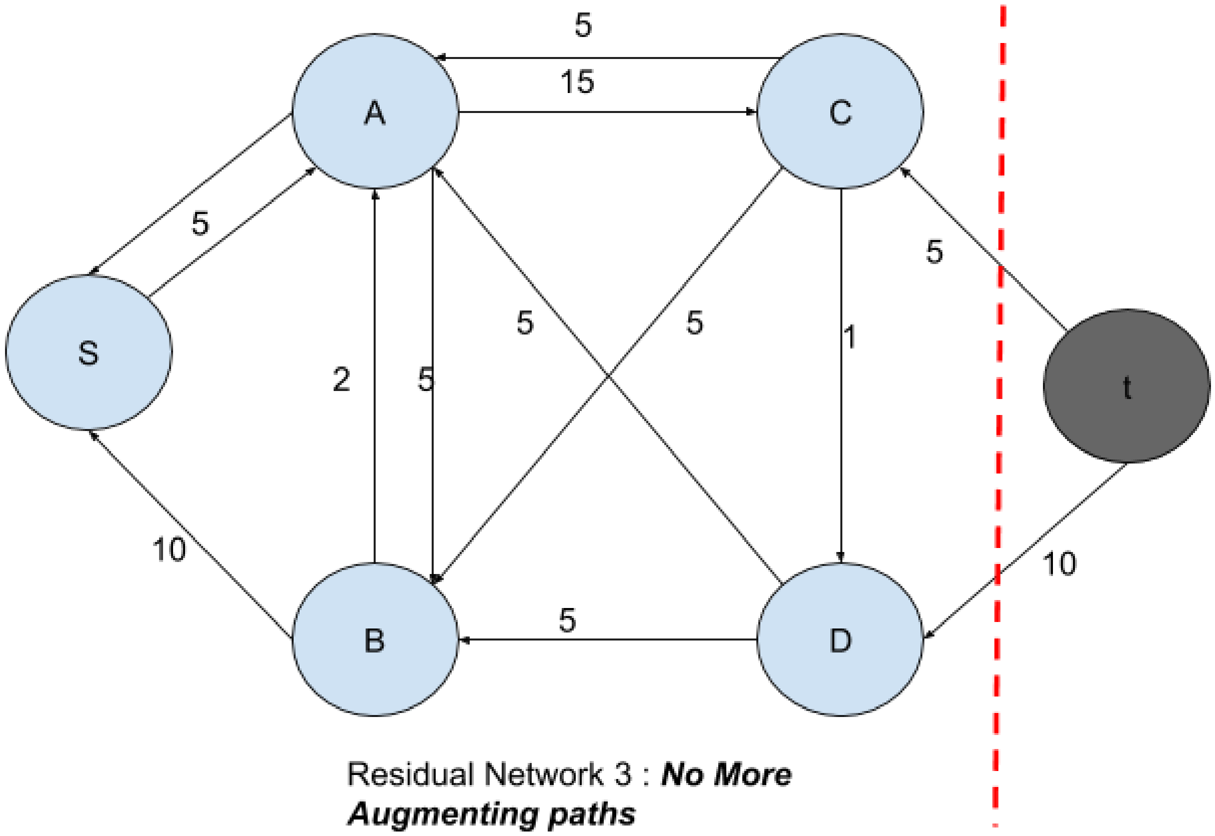
Flow Network 3



Residual Network 3 : ***No More Augmenting paths***

Max Flow across the network = 10 +5

(b) According to the Max-flow Min-cut Theorem
A Min cut has a capacity equal to the max flow ; (15)



Residual Network 3 : ***No More Augmenting paths***

2.

(a)

```
Assumption : Scheduler does not use any time
```

```
n = A.rows()                        // rows count
B = []                              // An array of size n
Parallel for i=1 to n:
    B[i] = 0
    for j=1 to n:
        B[i] += A[i][j]
```

The algorithm here uses two for loops, yet only one can be parallelized. The reason is, parallelizing the inner loop will create race conditions as it is trying to update the same memory location (B[i]) within a complete iteration of the inner loop. The outer Loop can be parallelized without no such race conditions. So, this is the best we can do for this algorithm in a parallelism point of view.

(b)

```
n = A.rows()                                       // Rows count
B = []                                         // An array of size n
```

```
MAT-SUM-Parallel (A,B):
     n = A.rows()
     B[i] = 0
     if(n==1):
          for j=1 to n:
               B[i] += A[i][j]
     else:
         mid = n/2
         Spawn MAT-SUM-Parallel (A[0:mid],B)
         MAT-SUM-Parallel (A[mid:n-mid],B)
         Sync
```

This recursive algorithm will update the globally defined ***Matrix (array) B*** given as the second parameter by summing the rows of the two dimensional ***Matrix A***. In each recursive call the ***Matrix A*** will be broken into two ***Matrices*** unless the number of rows in the ***Matrix A*** is one.

When dividing the Matrix A into two midpoint is considered. And one part is parallely computed while other is left for the main thread of each process call.

(c)

```
MAT-SUM-Parallel (A,B): ---------------------------------------- T(n)
     n = A.rows()        -----------------------------------------Θ(1)
     B[i] = 0            -----------------------------------------Θ(1)
     if(n==1):           -----------------------------------------Θ(1)
          for j=1 to n: ------------------------------------------Θ(n)
               B[i] += A[i][j] -----------------------------------Θ(1)
     else:
         mid = n/2 -----------------------------------------------Θ(1)
         Spawn MAT-SUM-Parallel (A[0:mid],B) -------------------T(n/2)
         MAT-SUM-Parallel (A[mid:n-mid],B) ---------------------T(n/2)
         Sync
```

$T = 2T(n/2) + \Theta(n)$

$Work = T_1(n) = 2T_1(n/2) + \Theta(N)$; N is the number of rows

$Span = T_\infty(n) = T_\infty(n/2) + \Theta(N)$; N is the number of rows

(d)

$$Work \;=\; T_1(n) \;= 2T_1(n\,/\,2) \;+\; \Theta(N) = \Theta(n) \;\times\; n = \Theta(n^2)$$

$$Span \;=\; T_\infty(n) \;= T_\infty(n\,/\,2) \;+\; \Theta(N) \;= \Theta(n)$$

$$Parallelism \;=\; T_1 /\, T_\infty = \Theta(n^2)\,/\,\Theta(n) \;=\; \Theta(n)$$

3.

(a) S1 strategy is not a good strategy. Think of a situation where an immediate push is carried out after a pop at k=L/2 point. Then again the size of the stack has to be doubled.
Example :

K = 5, L = 8

5	4	2	3	4	▨	▨	▨

After a pop

5	4	2	3

Push new element let's say 7

5	4	2	3	7	▨	▨	▨

So this approach doesn't keep cells for future pusheshes near L/2 pops. In an random amortized case probability of push and pop will be equal, resulting consecutive pushes and pops will increase the time cost.
But the strategy is more memory efficient as it utilizes the memory.

(b)
S2 strategy is better compared to the strategy S1 in performance aspect. This keeps cells for future pushes at any given time. But memory wise it is not that efficient.
Consider an example of popping an element at L/4, yet it will only reduce the the array in half. So, there are more empty cells in which future pushes can come in.

Eg : k =3 , L = 8

5	4	2	▨	▨	▨	▨	▨

After pop

5	4		

In an random amortised case the occurrences of recreating array will be low, resulting less time cost. But not as memory efficient as Strategy S1.

4.

(a)

```
def binary_search(A,elem):
   length_of_array = len(A)                  # Constant time in python
https://wiki.python.org/moin/TimeComplexity
   mid_point = length_of_array/2
   if(length_of_array == 1):
       if(A[0] == elem):
           return 0
       else:
           return -1
   elif(A[mid_point] == elem):
       return mid_point
   elif(A[mid_point] < elem):
       return mid_point + binary_search(A=A[mid_point:],elem=elem)
   else:
       return binary_search(A=A[:mid_point],elem=elem)
```

(b)

```
def binary_search_randomized(A,elem):
   length_of_array = len(A)
   break_point = random.randrange(0,length_of_array)
   if(length_of_array == 1):
       if(A[0] == elem):
           return 0
       else:
           return -1
   elif(A[break_point] == elem):
       return break_point
   elif (A[break_point] < elem):
       return break_point + binary_search_randomized(A=A[break_point:],
elem=elem)
   else:
       return binary_search_randomized(A=A[:break_point], elem=elem)
```

CS4522 Advanced Algorithms

Assignment 02 140106T - K.G.S. De Silva

(c) Algorithms implemented in python

Array Size N	Average Time for Search (seconds)	
	Binary Search	Randomized Search
10^5	0.843284845352	2.00084805489
10^6	11.8911921978	30.8364710808
10^7	171.556825876	336.787695885

Randomized search took nearly twice time as Binary Search. In randomized search worst case scenario is linear time. The reason for the huge time cost is that. Since we select the breaking point randomly the breaking point could be either near the largest value or near the lowest value.

In an occasion where the query element is near the max edge where random breaking point is near min edge worse case will be occured. There are many cases where this scenario can be occured.

In randomized search there can be situations where query element is near to the random value resulting very less time.

But in binary search always the mid point is selected, so that every query will equally benefit.

Discussed with : U.K.D. Akalanka - 140019E , G.A.B. Rathnayaka - 140528M