

Index No: 190077A

Name: Bandara L.A.R.L.

1. Question 01

a)

```
# Assumed that if the matrix is 1x1 there is 1 path
# Number of paths to the cell (i,j) can be computed using the number of paths
# to its left neighbor cell (i,j - 1) and top neighbor cell (i - 1,j).
# Paths to left neighbor cell (i,j - 1) = count_paths(i, j-1)
# Paths to top neighbor cell (i - 1,j) = count_paths(i-1, j)
```

```
def count_paths(m,n):
    if m==1 or n==1:
        return 1
    else:
        return count_paths(m-1,n) + count_paths(m,n-1)
```

b)

The time complexity of the algorithm in (a) above is $O(2^{(m+n-2)})$ because the algorithm is a recursive algorithm that calls itself twice.

The algorithm will run until it reaches the base case ($m==1$ or $n==1$) and the number of recursive calls is $2^{(m+n-2)} - 1$

$$T(m,n) = O(2^{(m+n-2)})$$

c)

```
def count_paths_dp(m, n):
    # Create a 2D array to store path counts
    dynamic_array = [[1] * n] * m

    # Iterate through the dynamic_array
    for i in range(1, m):
        for j in range(1, n):
            # Calculate the number of paths to reach (i, j)
            dynamic_array[i][j] = dynamic_array[i-1][j] + dynamic_array[i][j-1]

    # Return the count of paths from (1, 1) to (m, n)
    return dynamic_array[m-1][n-1]
```

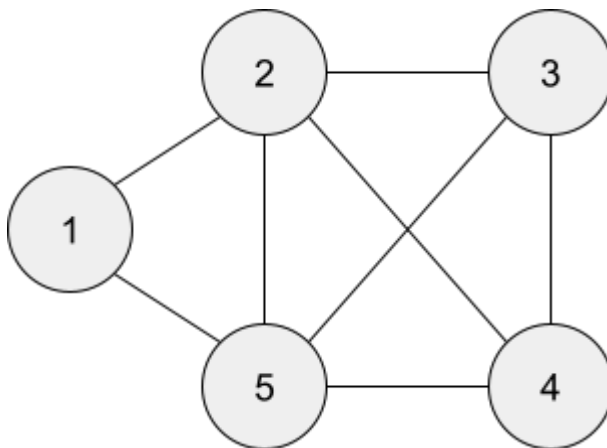
d)

The time complexity of the algorithm in (c) above is $O(m*n)$ because the algorithm iterates through the dynamic_array $m*n$ times.

Even though the (c) is improved in time complexity, it will consume more space for the computation.

Question 02

Graph:




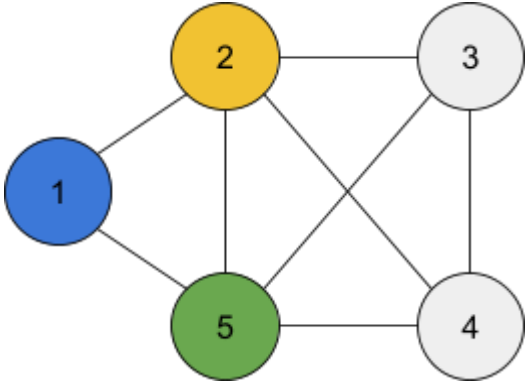

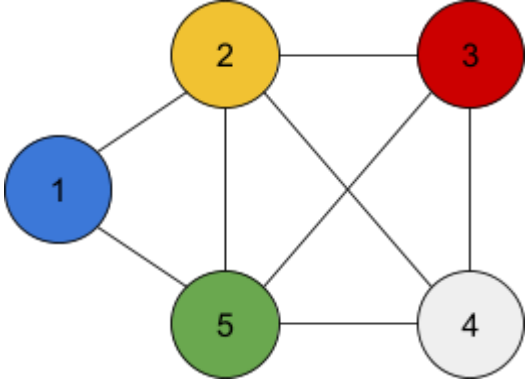

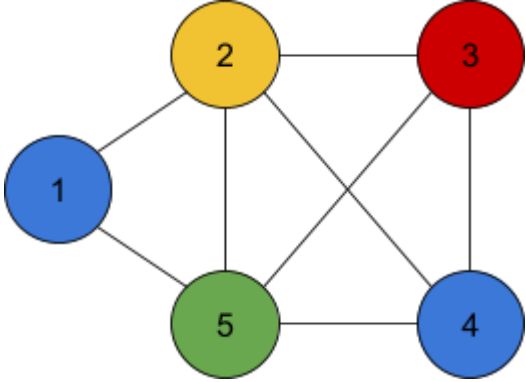
a. By applying the greedy coloring algorithm:

Vertex selection order : 1, 2, 3, 4, 5

Color order:

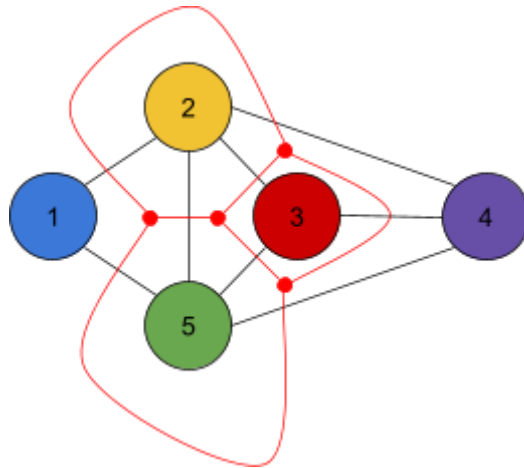


Effectd Vertex	Color Palette	Graph
1		
2		

5		
3		
4		

chromatic number of G , $X(G) = 4$

b.



Since the graph can be converted into a planar graph, it can have face coloring as shown in the above chart.

c.

M^2 Matrix: (i,j) of M^2 represents paths with length 2 from i^{th} node to j^{th} node.

M^3 Matrix: (i,j) of M^3 represents paths with length 3 from i^{th} node to j^{th} node.

d.

$$\begin{aligned} \# \text{ triangles} &= \text{trace}(M^3) / 6 \\ &= (2+8+6+6+8) / 6 \\ &= 5 \end{aligned}$$

e.

$$\begin{aligned} \# \text{ of closed triplets} &= 3 \times 5 \\ &= 15 \end{aligned}$$

$$\begin{aligned} \# \text{ of triplets} &= 1 + 6 + 6 + 3 + 3 \\ &= 19 \end{aligned}$$

Therefore,

$$\begin{aligned} \text{Global Clustering Coefficient} &= 15 / 19 \\ &= 0.79 \end{aligned}$$

f.

$$\begin{aligned} \text{Local clustering coefficient } C_2 &= (2 \times 4) / (4 \times 3) \\ &= 0.67 \end{aligned}$$

Question 03

Suppose you are given two matrices $A=(a_{ij})$ and $B=(b_{ij})$ each of size $n \times n$ and you have to perform matrix addition to compute a new matrix $C=(c_{ij})$ as $c_{ij} = a_{ij} + b_{ij}$ for all $i,j = 1,2, \dots, n$.

- a. Give a multi-threaded algorithm to compute C in parallel using parallel loops (i.e., parallel for). Parallelize as much as possible.

P-Matrix-Add(A, B):

$n = A.rows$

Let C be a new $n \times n$ matrix

parallel for $i = 1$ to n :

parallel for $j = 1$ to n :

$C_{ij} = A_{ij} + B_{ij}$

Since there are no occurrences of parallel instructions accessing the same memory location at the same time and performing write operations, there will be no race conditions. Therefore we can add parallel operation for both for loops.

- b. Give a multi-threaded algorithm to compute C in parallel using nested parallelism (i.e., spawn) with a divide-and-conquer strategy. State any assumptions and explain briefly how it works. Parallelize as much as possible

(Assumed that n is a power of 2)

P-Matrix-Add-Recursive(C, A, B):

$n = A.rows$

if $n == 1$: # base case

$C_{11} = A_{11} + B_{11}$

else:

Let T be a new $n \times n$ matrix

Partition A, B, C and T into $n/2 \times n/2$ submatrices $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}, C_{11}, C_{12}, C_{21}, C_{22}$ and $T_{11}, T_{12}, T_{21}, T_{22}$ respectively

spawn P-Matrix-Add-Recursive(C_{11}, A_{11}, B_{11})

spawn P-Matrix-Add-Recursive(C_{12}, A_{12}, B_{12})

spawn P-Matrix-Add-Recursive(C_{21}, A_{21}, B_{21})

P-Matrix-Add-Recursive(C_{22}, A_{22}, B_{22})

sync

return

- c. Determine the work, span and parallelism for the parallel algorithm in (b) above.

Work:

- Let span for partitioning is $\Theta(1)$
- Performs 4 recursive multiplications of $n/2 \times n/2$ matrices
- Finally $\Theta(n^2)$ work from adding $n \times n$ matrices.
Work, $T_1(n) = 4T_1(n/2) + \Theta(1)$
From master theorem,
 $T_1(n) = \Theta(n^2)$

Span:

- Let span for partitioning is $\Theta(1)$
 - Span, $T_\infty(n) = T_\infty(n/2) + \Theta(1)$
 $T_\infty(n) = \Theta(\lg n)$

Parallelism:

$$\begin{aligned} T_1(n) / T_\infty(n) &= \Theta(n^2) / \Theta(\lg n) \\ &= \Theta(n^2 / \lg n) \end{aligned}$$

Question 04

(a) Parallel algorithm with m processors ($m \ll n$):

```
Product(A,m):
  n = A.length
  k = (n/m)
  Initialize new array P with length m
  parallel for i = 1 to m:
    P[i] = 1
    if i != m:
      for j = 1 to k:
        P[i] = P[i] * A[k(i-1) + j]
    else:
      for j = k(m-1) to n:
        P[i] = P[i] * A[j]
  value = 1
  for i = 1 to m:
    value = value * P[i]
  return value
```

The number of operations in this algorithm is $O(n)$, as each processor computes the product of n/m elements, and there are m processors.

The number of steps in this algorithm is $O(\log m)$, where m is the number of processors.

This is because, in each step, the number of active processors is halved until the base case.

(b)

parallel algorithm with an unlimited number of processors:

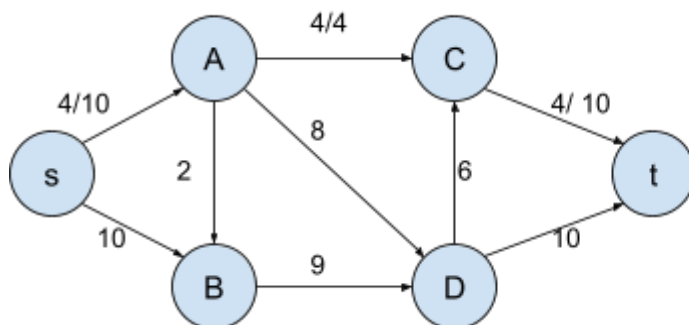
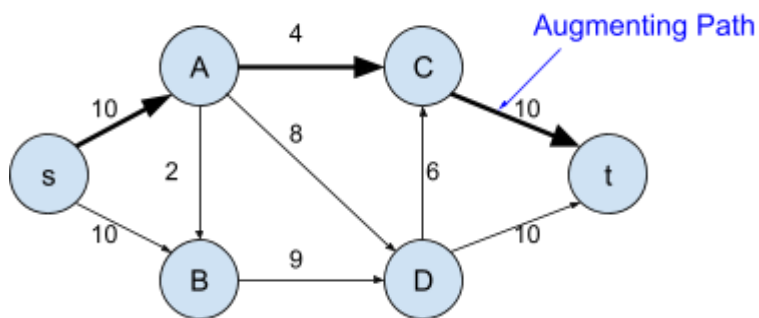
```
Parallel-Product(A, i, j):
  if (i == j):
    return A[i]
  m = (i+j) // 2
  a = spawn Parallel-Product(A, i, m)
  b = Parallel-Product(A, m+1, j)
  sync
  return a*b
```

This is a divide and conquer algorithm.

The number of steps is $O(\log n)$, and the number of operations is $O(n)$.

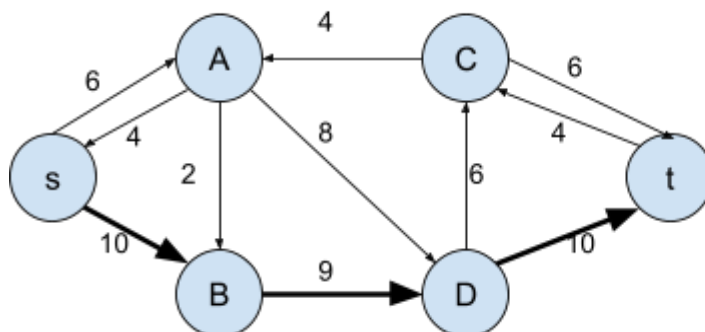
Question 05

A.
(i)

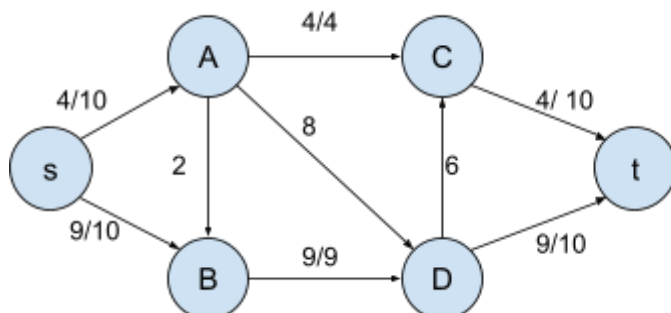


Flow Network

Resulting flow = 4

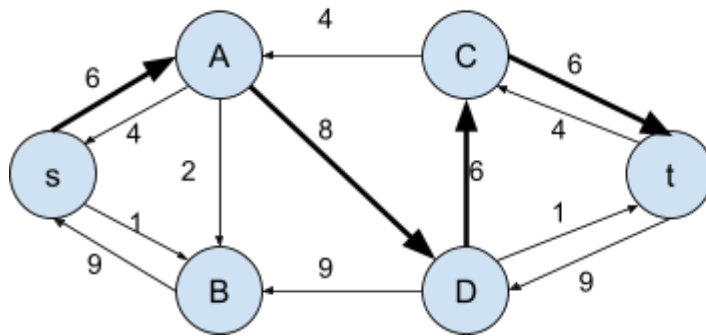


Residual Network

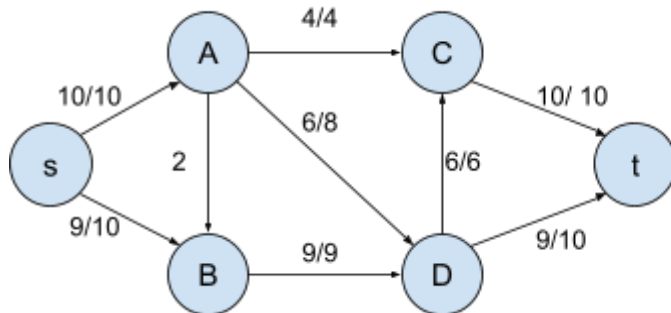


Flow Network

Resulting flow = 4 + 9 = 13

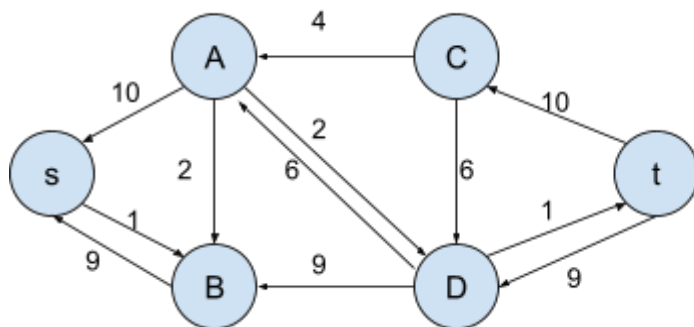


Residual Network



Flow Network

Resulting flow = $13 + 6 = 19$

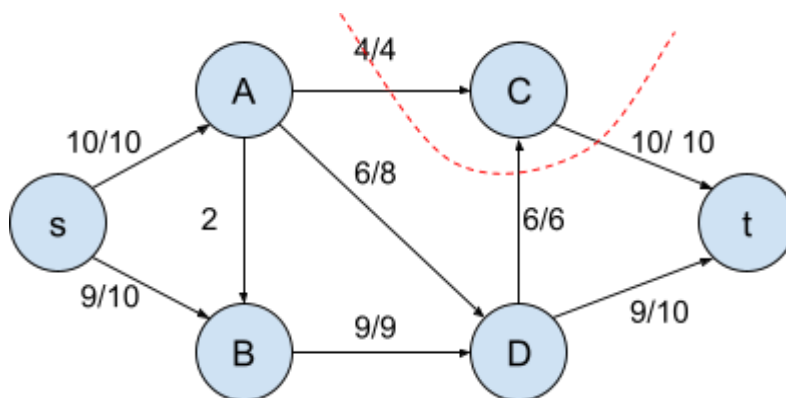


Residual Network

* s-t paths are saturated.
Therefore, flow = 19

(ii)

The max-flow min-cut theorem states that in a flow network, the amount of maximum flow is equal to the capacity of the minimum cut. Therefore, below are the min-cuts that can be found in the above flow network.



(b)

To solve the maximum matching problem in a bipartite graph, we need to convert the given graph into a flow network and then find the maximum flow in the network.

Here are the steps to solve this problem:

1. Convert the given graph into a flow network by adding source and sink nodes.
2. In this case, there is no specific sink or source we might have to create a super source and a super sink outside of the two sets and create edges.
3. Assign capacities to the edges such that the capacity of an edge is 1 if it is between two nodes in the original graph and 0 otherwise. (For each edge (i, j) , we will have one variable x_{ij} that takes on value 1 if $(i, j) \in E$ or 0 if otherwise.)
4. Find the maximum flow in the network using the Ford-Fulkerson algorithm.
5. The size of the maximum matching in the original graph will be equal to the value of the maximum flow in the network.

The resultant Maximum flow network will give the way to assign as many tasks as possible.

(c)

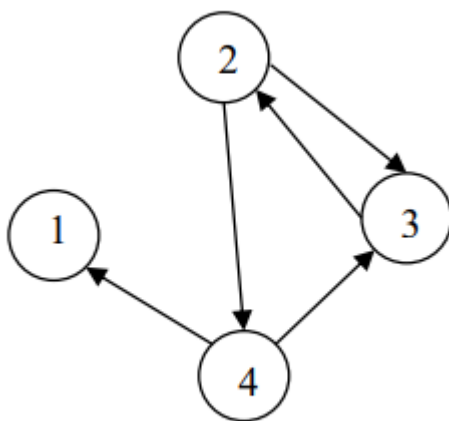


Fig. Q5.3

Assumed weight=1 to each edge of E.

Floyd-Warshall algorithm run:

$$i = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & 1 \\ \infty & 1 & \infty & \infty \\ 1 & \infty & 1 & \infty \end{bmatrix}$$

$$ii = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 2 & 2 & 1 & 1 \\ \infty & 1 & \infty & 2 \\ 1 & 2 & 1 & \infty \end{bmatrix}$$

$$iii = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 2 & 2 & 1 & 1 \\ \infty & 1 & \infty & 2 \\ 1 & 2 & 1 & 3 \end{bmatrix}$$

$$iv = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 2 & 2 & 1 & 1 \\ \infty & 1 & \infty & 2 \\ 1 & 2 & 1 & 3 \end{bmatrix}$$

Transitive Closure:

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 1 & 1 & 1 & 1 \\ \infty & 1 & \infty & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Code:

<https://github.com/lahirub99/Advanced-Algorithms/>

Resources:

https://www.brainkart.com/article/Maximum-Matching-in-Bipartite-Graphs_8054/

[lecture17scribe.pdf \(duke.edu\)](#)