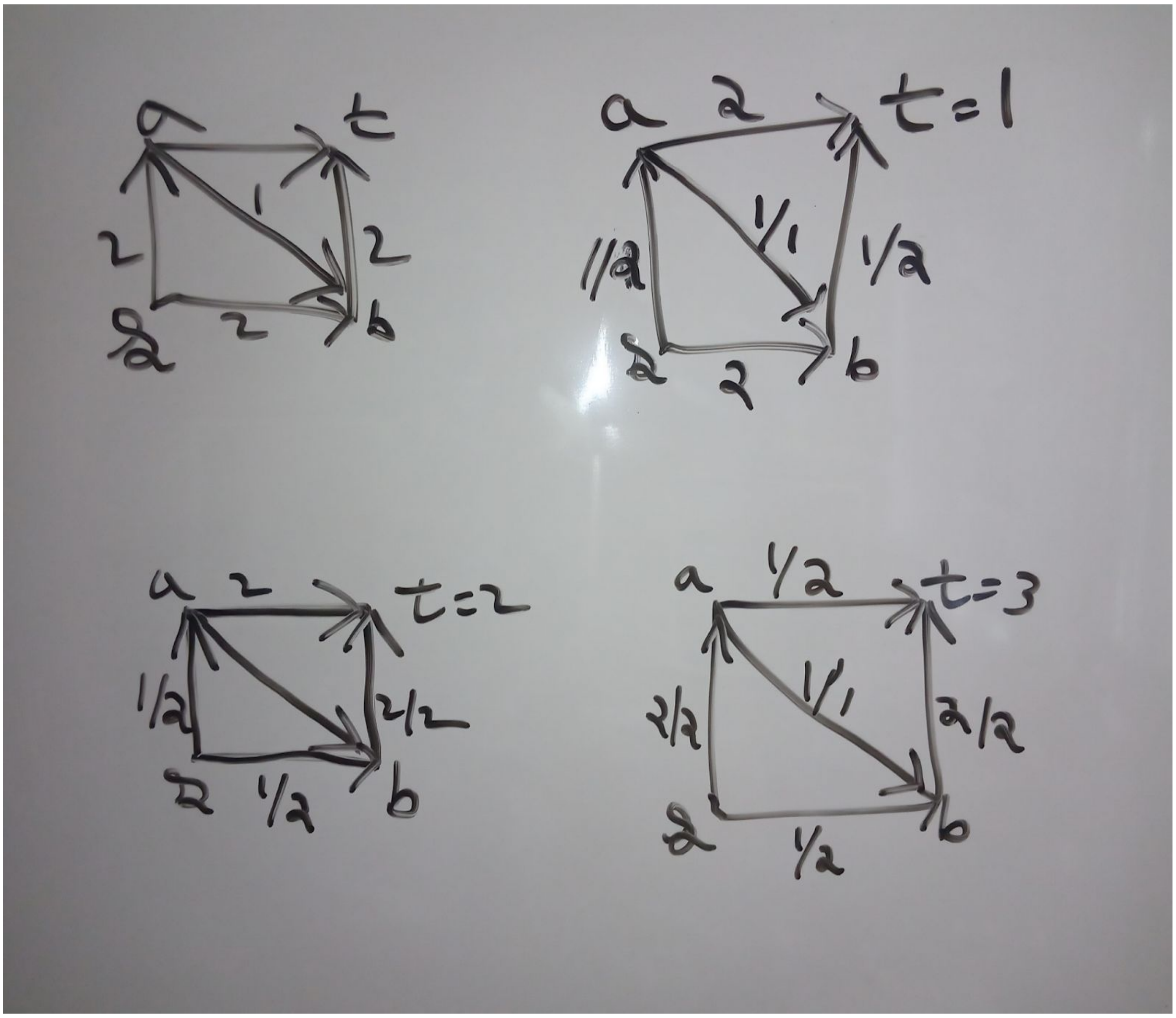
HW4

Pramod Kulkarni

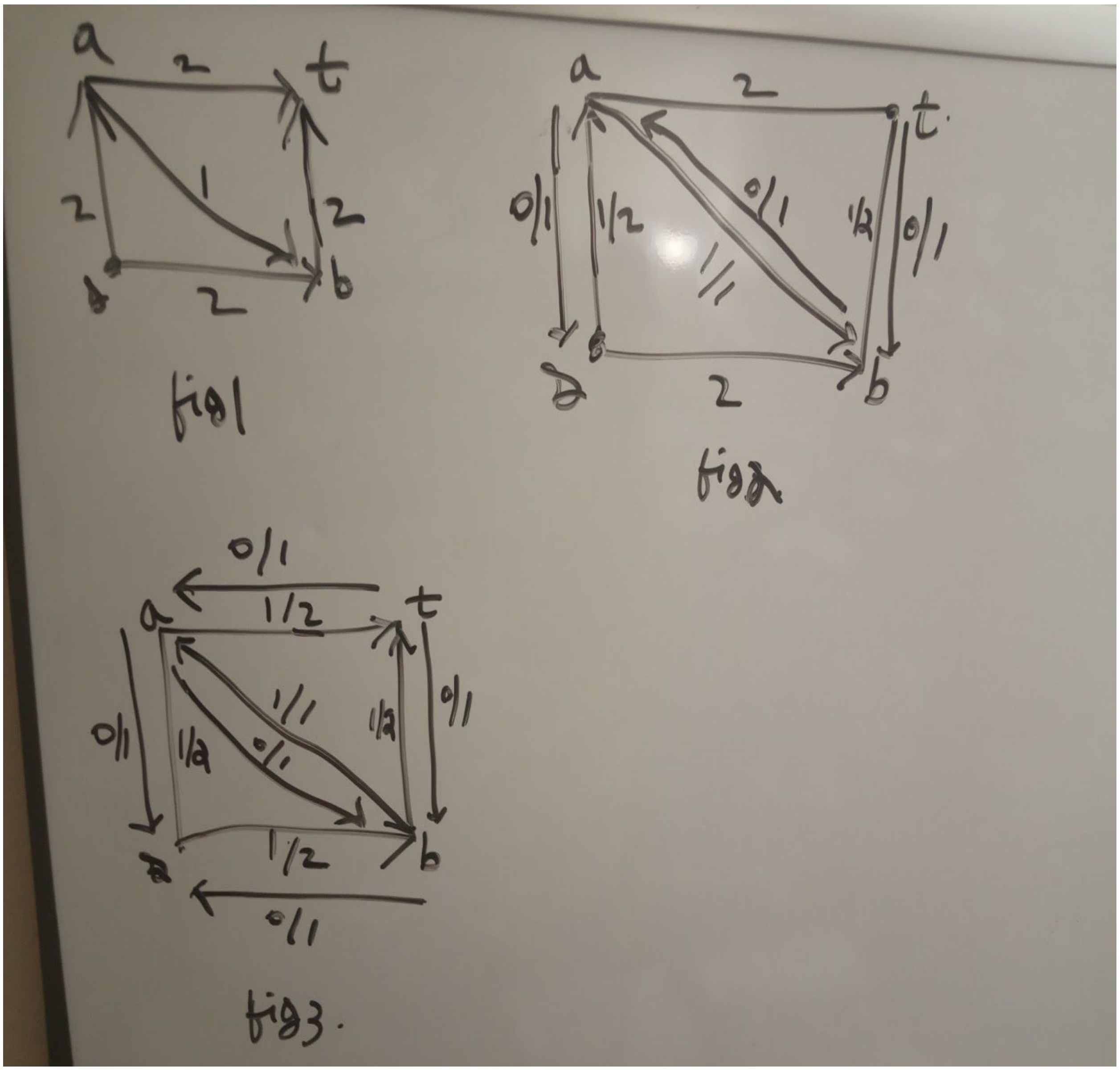
Problem 1:

Part a)



- Consider the graph above. Suppose algorithm selects s-a-b-t path. Minimum flow that can be sent through path s-a-b-t without violating capacity constraint is 1. So we push flow=1 and get to fig2.

- We cannot propagate any flow into path s-a-b-t as we have hit maximum capacity on edge ab. In the next iteration we may select path s-a-t. Minimum flow that can be sent through this path is 1. So we push flow=1 and get to fig.3
- From fig.3 we can send flow = 1 along s-b-t path.
- This gives flow of 3 when max flow is 4. So algorithm fails in this case.



- Let’s consider Ford-Fulkerson algorithm. We take s-a-b-t path and increase flow by 1 and construct a residual graph G’ from the original graph G reaching fig.2. The edge ba with capacity 1 means we can decrease flow in backward direction.
- We then might select s-b-a-t path. Minimum flow that can be sent without violating capacity constraints is 1. So

we push flow = 1 along that path and decrease the flow by considering edge ba leading us to fig.3

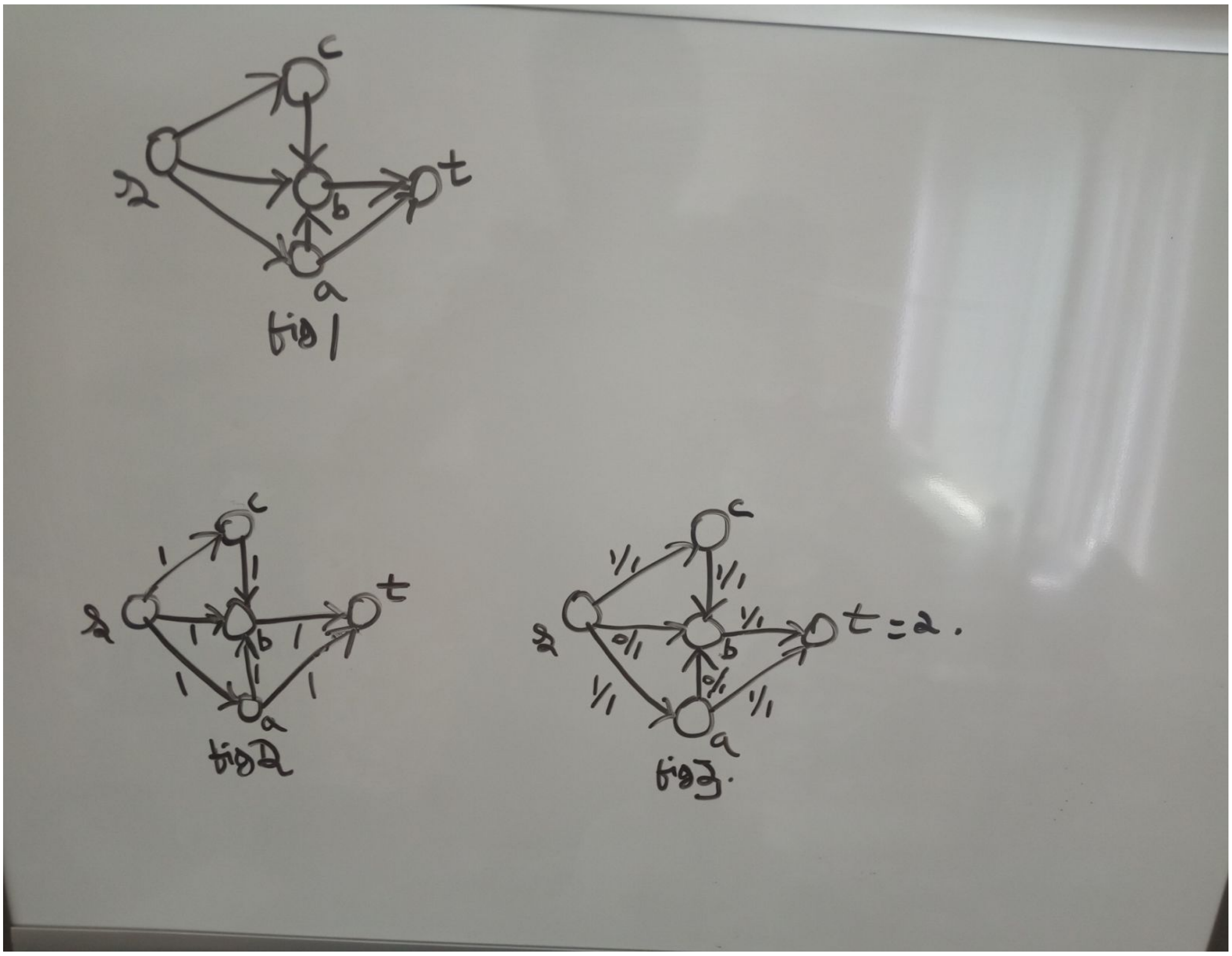
- We then might select s-a-t and s-b-t paths increasing flow = 1 respectively and getting a flow = 4 at t which is max flow.

Part b

Ford-Fulkerson gives us a chance to undo the wrong decision of selecting s-a-b-t path because there’s an edge (b,a) which helps in decreasing flow. This is because of construction of residual graph G’ which is not available in case of algorithm specified in question of part a.

Problem 2

Part a



- A set of paths in G are edge disjoint if edge eεE appears in at most one paths. Several edge disjoint paths may pass through a vertex.
- Given a graph G={V,E} with start vertex s and end vertex t assign each edge (u,v) a capacity of 1.
- Run Edmond Karp algorithm on 0-1 network.
- From fig.3 we see that max flow = number of disjoint paths.

- We select paths s-c-b-t and s-a-t giving us max flow of 2 and these paths are edge disjoints as no edge is common in both paths.

Correctness:

- We have to prove that max_flow = number of disjoint path.
- Consider base case when flow f = 0. Then maximum flow through graph is 0 and s-t paths with disjoint edges is 0 as well.
- Consider some maximum flow of m>0 through graph. Remove all the edges with f = 0.
- Find a s-t path(it exists because m>0) and remove all edges on this path.
- Maximum capacity of any edge is 1 hence maximum flow across any s-t path is 1. Hence after we remove one s-t we are left with m-1 flow.
- If m = 1 originally m - 1 = 0. We have proved already for base case 0. So when m = 1 we just have 1 s-t disjoint path.
- So max-flow = number of disjoint paths. If this works for max-flow of value 1 and 0 it works for m>1. Hence proved.

Running time:

- We're using Edmond-Karps max flow algorithm to find largest number of disjoint paths in Graph G.
- Running time of above algorithm will be same as Edmond-Karp which is O(|V| |E^2|).

Part b)

- Given graph G= (V,E), set all edge capacities to 1.
- Run Edmond-Karp to find max-flow from start vertex s to end vertex t. With start flow f =0.
- Find min-cut such that (S,T) = max-flow (m)
- Let A be the set of edges that cross-over from S to T
- The minimum number of edges to be deleted from Graph such that t becomes unreachable from s is |A|

Correctness:

- From **problem 2 part a** we proved that if we set all edge capacities to 1 and run max-flow algorithm then **number of disjoint paths = max flow m.**
- In the above algorithm we do a similar process and find min cut (S,T).
- If we have **max flow as m then there must be m disjoint paths**. That means there are **m edges crossing from S to T**.
- If we delete these edges then t becomes unreachable from s. That's because each path is dependent on the cross over edge to reach to t. After deleting all edges from A no such cross over edge is available to reach to t, hence no path from s to t.

Running time:

- Running time is the same as that of Edmond -Karp that is O(|V||E^2|)

Problem 3
Part a
Input:

- Unweighted,undirected graph G=(V, E)

Output:

- Check if G is bi-partite graph or not.

Algorithm:

- A graph is bi-partite if we can divide V into two sets of vertices A and B such that no edge can be found in the same set.
- We can check for bi-partiteness by assigning colours to each vertex.We use two colors RED and BLUE. If vertices on an edge have the same colour then we return false else true. We can use BFS for colouring.

1. Initialize all visited[all vertices] = false. Color[all vertices] = NULL.
2. Let s be the start node.
3. Call BFS(s)
4. BFS(s):

```
color[s] = red
visited[parent] = true
Initialize queue Q
Q.add(s)
While Q is not empty:
    parent = Q.pop()
    For all children c of parent :
        If visited[c]==false:
            Q.add(c)
            Visited[c] = true
            If color[parent]==RED:
                color[c]=BLUE
            If color[parent]==BLUE:
                color[c]=RED
        If color[c]==color[parent]:
            return false
return true
```

Correctness:

- We know that if the graph has a cycle with an odd number of vertices, then it’s not bipartite.
- We assign Red color to source and then for every child we assign BLUE color. In the next iteration we pop from queue and assign each child in level 2 a RED colour.
- If there are any edges between adjacent vertices in the same level it would lead to the formation of a cycle with an odd number of vertices in the graph G, but then as per our assumption such a graph is not bipartite and we return false. We identify such edges by checking if adjacent vertices have the same color.
- If there is no such edge in any level of our graph we return true else algorithm would have returned false and would not have reached this point.

To-Prove:

If a graph has a cycle with an odd number of vertices, then it is not bipartite

Proof:

- Let us assume a graph G=(V,E) has a cycle with an odd number of vertices and is bi-partite.
- If G is bi-partite then we should be able to divide set V into sets A and B such that no edge can be found within a set.
- Let v1,v2,v3,v4...vn be the number of vertices forming cycle and n is odd. We partition v1,v3,v5..vk into set A and v2,v4,v6..v(k-1) into set B.
- To form a cycle we must include edge vk,v1 which is an edge within set A.
- This contradicts the definition of bip-partite graph. Hence,our assumption was wrong.
- Therefore by proof of contradiction,If a graph has a cycle with an odd number of vertices, then it is not bipartite.

Running time:

- We use BFS algorithm to check if Graph G is bi-partite or not. So running time of above algorithm is same as BFS that is O(|V|+|E|)

Part b

- From our modified BFS algorithm in **part a** we return true if Graph G is bi-partite.
- Instead we can modify our algorithm to return partition of vertices. We will iterate through **color array** and partition vertices based on colour. Following is the modification:
- Initialize A = {} and B = {}
- For each vertex v in V:

```
        If color[v] == RED:
                A.add(v)
        else:
                B.add(v)
return A,B
```