Algorithm Theory Assignment 3

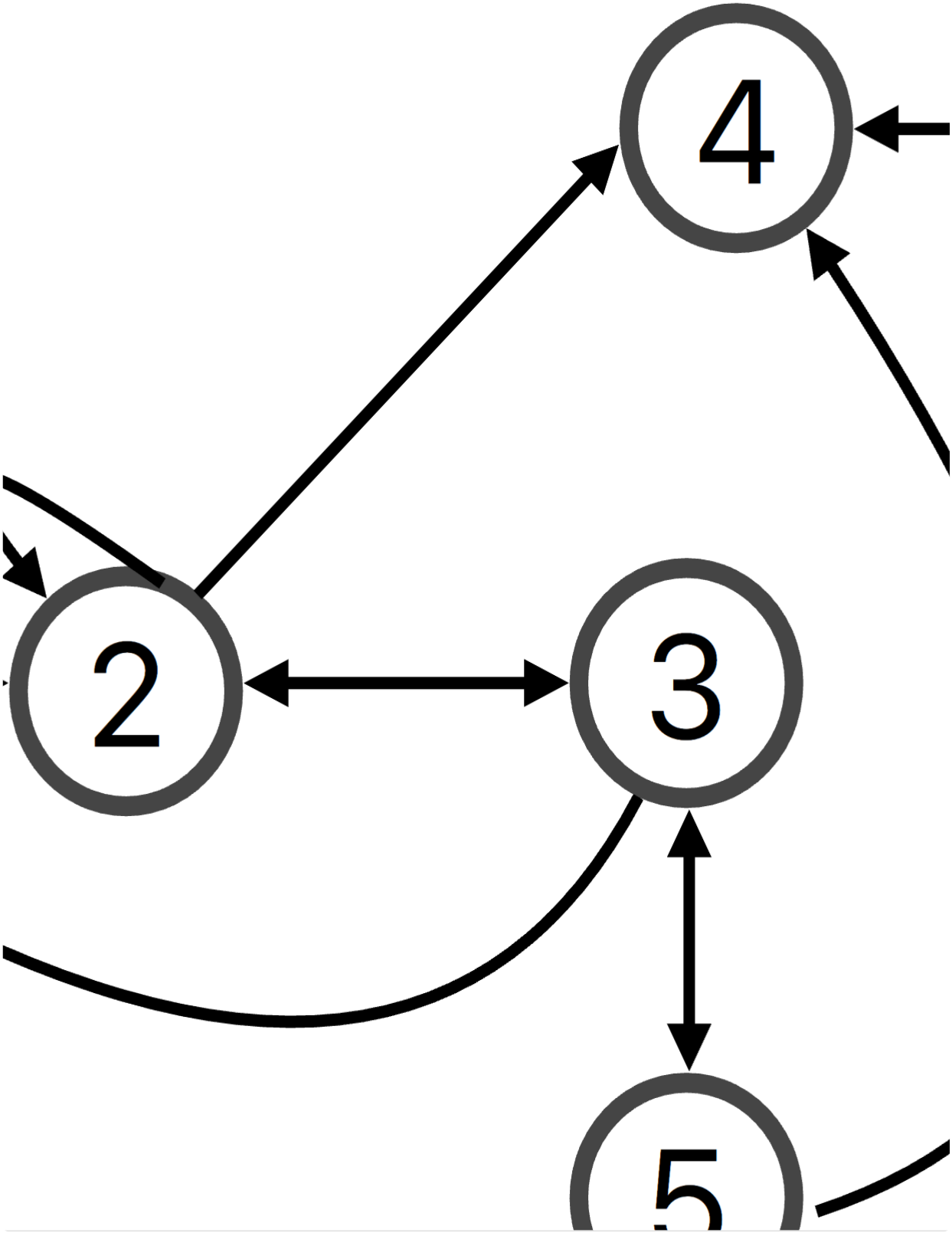
1. Problem Basic Graph

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

a)

1. convert to adjacency list. Note, here I choose to 1-index rows and columns, and interpret the connection matrix in row i column j as the connection from component i to component j.

$$\begin{gathered} 1 \to [2] \\ 2 \to [2, 3, 4] \\ 3 \to [1, 2, 5] \\ 4 \to [5, 6] \\ 5 \to [3, 4] \\ 6 \to [4] \end{gathered}$$

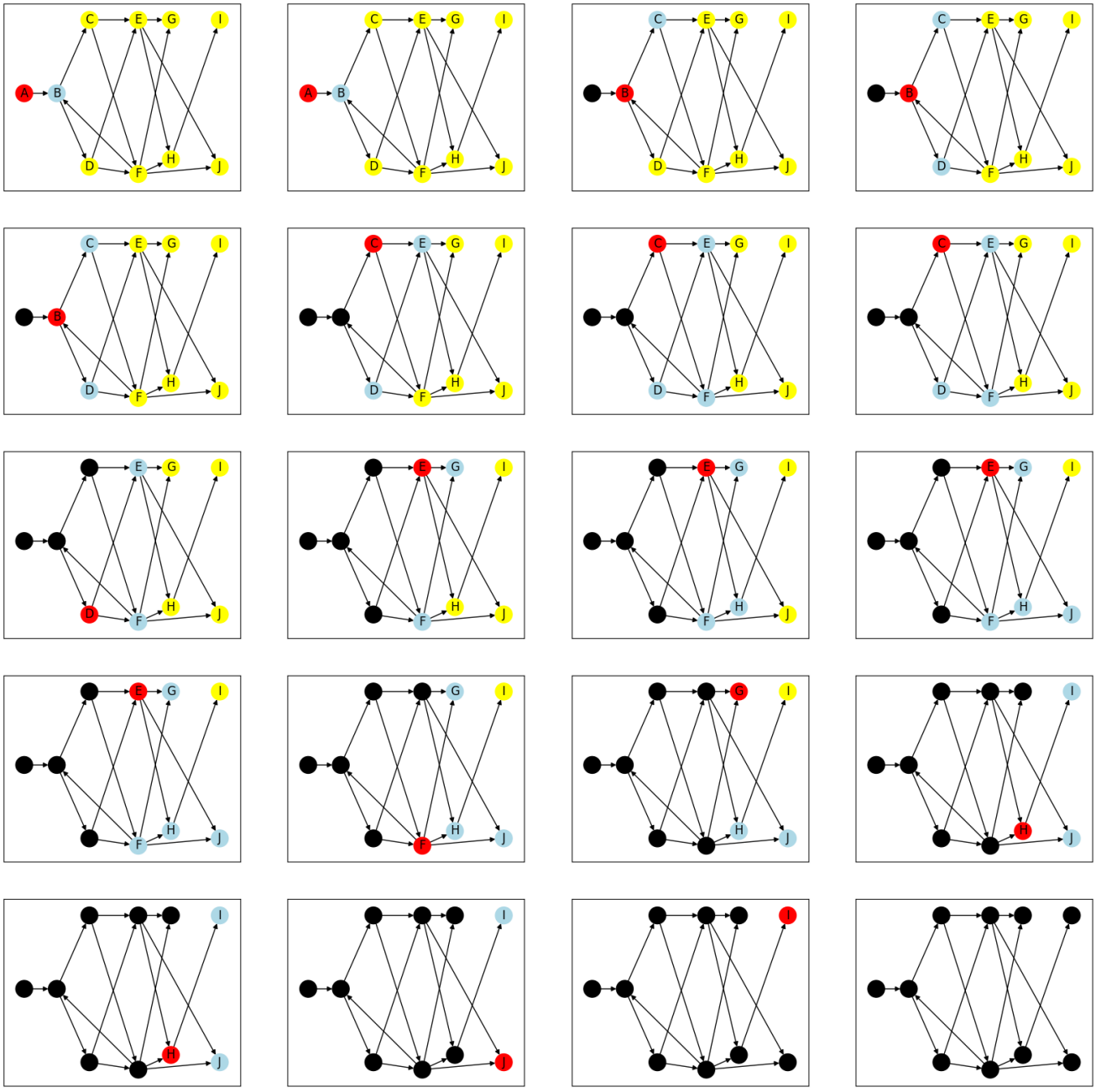

4
2
3
5

3.

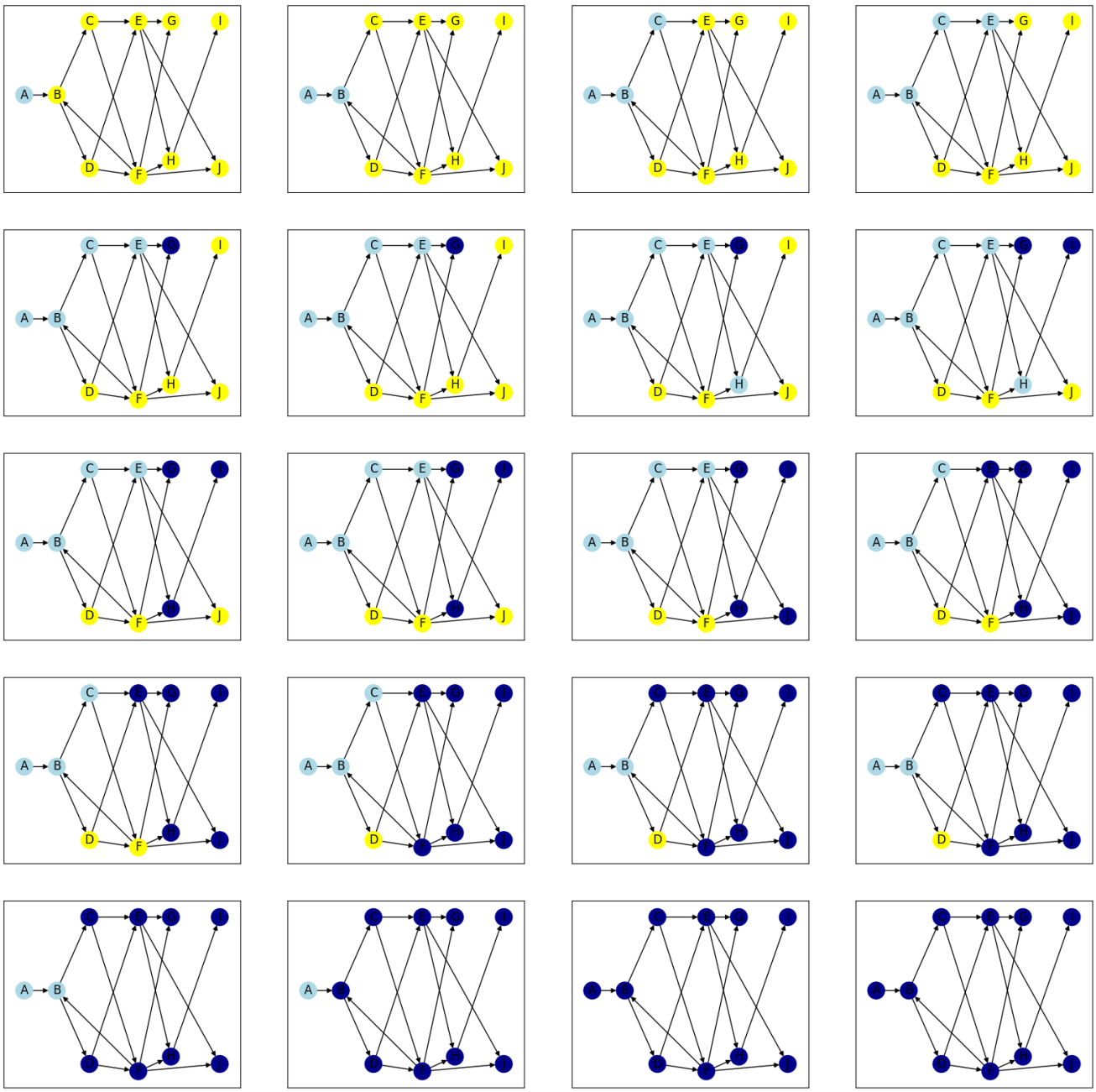
$$A \to [B]$$
$$B \to [C, D]$$
$$C \to [E, F]$$
$$D \to [E, F]$$
$$E \to [F, G, J]$$
$$F \to [B, G, J]$$
$$G \to \emptyset$$
$$H \to [I]$$
$$I \to \emptyset$$
$$J \to [I]$$

b) Show how the BFS and DFS algorithms would look traversing the given graph:

BFS



DFS



Note that there are some redundant plots in both figures.

c) The edge to remove is the edge $F \to B$

By performing a topological sort on the graf this edge removed, we get:

['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'I']

d)

Propose the following algorithm:

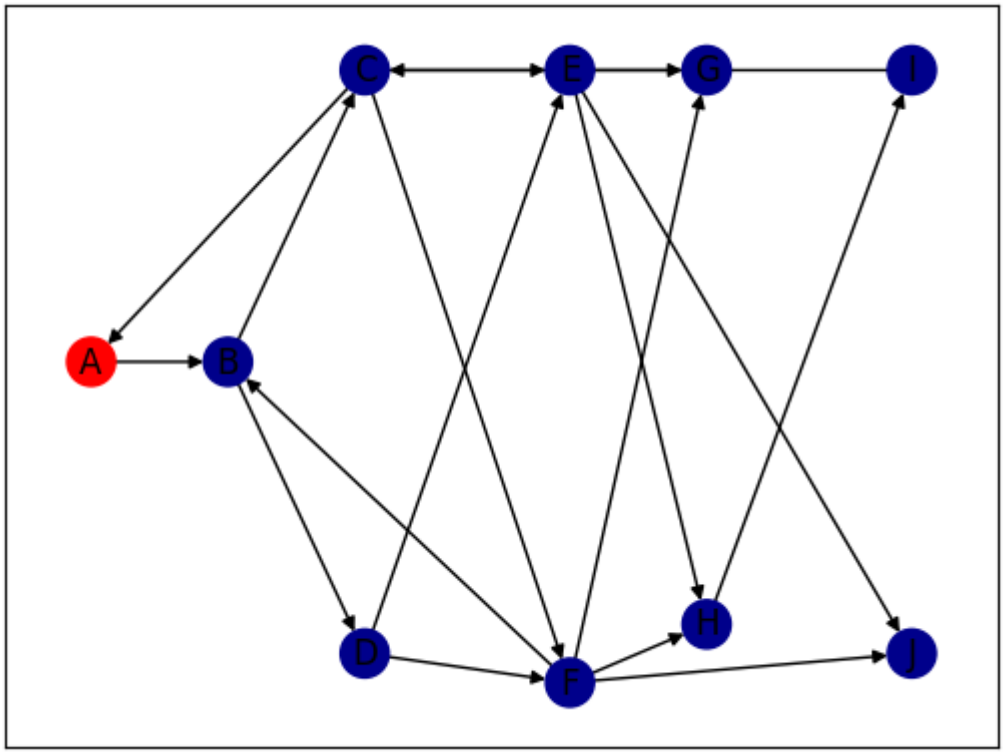
Perform dfs search. During node traversal, if a discovered vertex is already visited, we mark the edge pointing to this node as a backedge.

Then we list all the backedges and drop them from the graph.

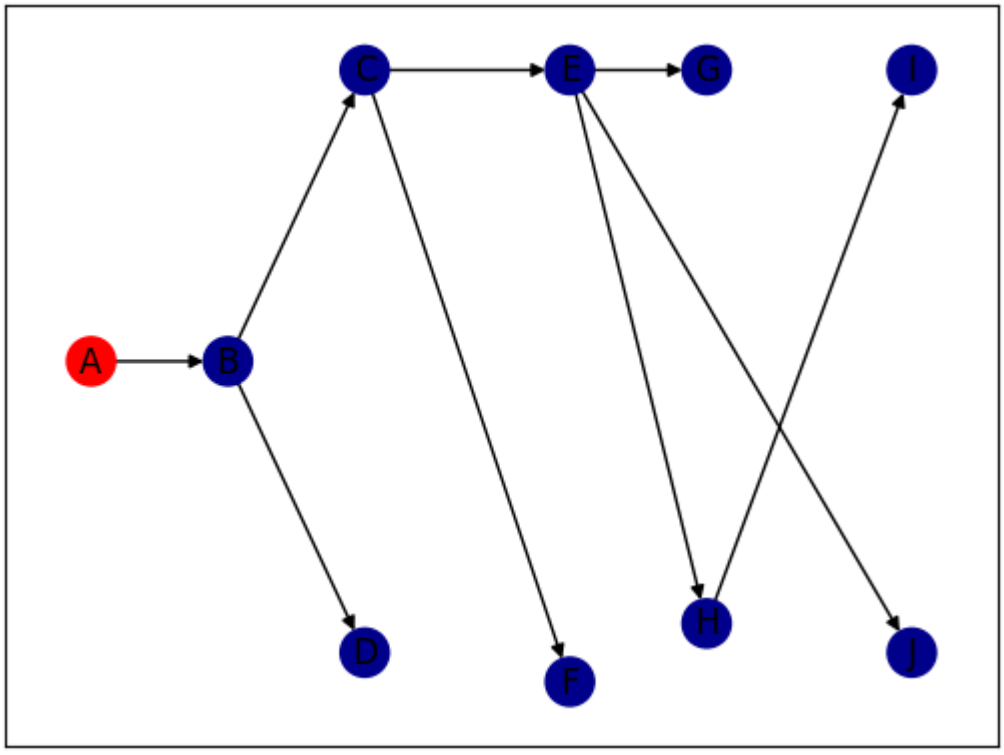
```
def dfs_backedges(graph):
    time = 0
    def dfs_visit(graph, node):
        """Recursive dfs graph traversers
        parameters:
            graph[nx.graph] grap to traverse
            node[str]: current node to traverse from"""

        nonlocal time
        time += 1
        nx.set_node_attributes(graph, {node: {'discover': time}})
        nx.set_node_attributes(graph, {node: {'color': 'lighblue'}})
        for child in graph.successors(node):
            if graph.nodes[child]['color'] == 'yellow':
                nx.set_node_attributes(graph, {child: {'parent':
node}})
                dfs_visit(graph, child)
            else:
                nx.set_edge_attributes(graph, {(node, child):
{'backedge': True}})

        nx.set_node_attributes(graph, {node: {'color': 'darkblue'}})
        time +=1
        nx.set_node_attributes(graph, {node: {'time':time}})

    for vertex in graph:
        if graph.nodes[vertex]['color'] == 'yellow':
            dfs_visit(graph, vertex)

    return graph
```

```
g = dfs_backedges(graph3)
backedges = [edge for edge in g.edges if
g.get_edge_data(*edge).get('backedge')]

graph3.remove_edges_from(backedges)
```

Graph with additional edges



Graph with "back edges" removed



Problem 2)

a)
The minimal sum of edges connecting every node is 26 which is below the constraint of 30

b)
The modified algorithm involved keeping count of the connections to the specific node D.
This will not always create a globally optimal solution, case in point if we have the network consisting of a node D with > 3 spokes of weight 1, none of which have any other connections. The solution to task 2 b) will not find a solution to this graph. As it is bound to leave some nodes out of the graph.

c)
From the solution graph we can try to exchange the largest included edges, with the smallest non included edges, and see if we can get a better overall solution. In this case of the solution four task A) we only have the edges F -> H (w: 7) to switch out with A -> B (w. 5). This would result in us obtaining a sum of 24 which is less than b' = 25.

3. a)

proppsed algorithm:

```
def find_champions(graph: nx.graph):
    champions = []

    def traverse(graph: nx.graph,start_node: str):
        nonlocal nodes_left
        nonlocal champions
        for child in graph.successors(start_node):
            if child in champions: # if we can reach a champion, then
we have a new another champion
                champions.append(start_node)
                return
            if child in nodes_left:
                nodes_left -= set(child)
                traverse(graph, child)

    for node in graph:
        nodes_left = set(graph.nodes).difference(node)
        traverse(graph,node)
        # print(node, nodes_left)
        if len(nodes_left) == 0:
            champions.append(node)

    print(champions)
```

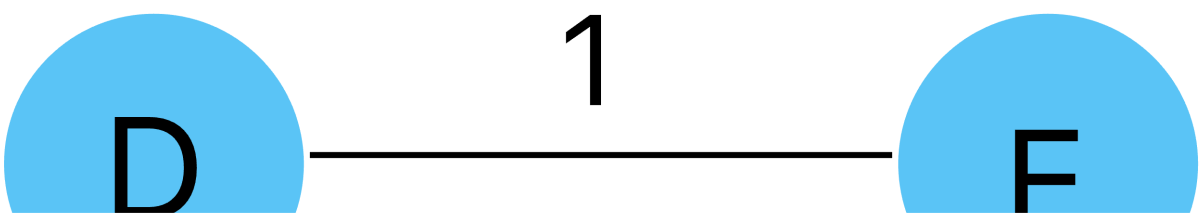
running this for the same graph as provided in the task sheet yields:
['A', 'B', 'C']

b)
Find groups that have defeated each other:
This simply become the task of finding strongly connected components.
Using networkx this task becomes simple:

```
list(nx.strongly_connected_components(graph))
```

the running time for this is then simply $O(V + E)$
from the definition of the SCC algorithm
(unless nx uses some faster algorithm)

4. a)

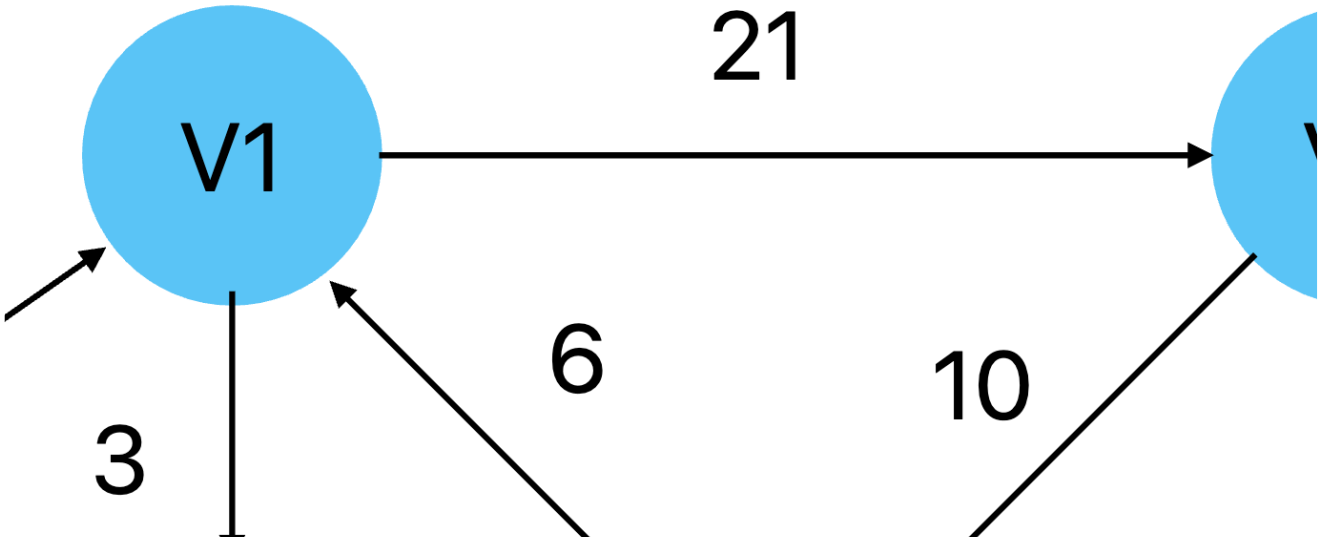


If we were to run djikstras algorithm over this starting from node E, it would yield essentially a cost of -1 going to node C. Hence a "shortest path" to node E to B could be: E,D,A,C B instead of the true shortest E -> B route.

b)
The simplest solution to the negativ edge problem can be to take the smallest negtavie edge weight that occurs in the graph: $v = min_i(E_i)$ and add the absolute value of this number to every edge such that ever edge value is ≥ 0
Another is to use the bellman ford algorithm instead of djikstra's.

5.

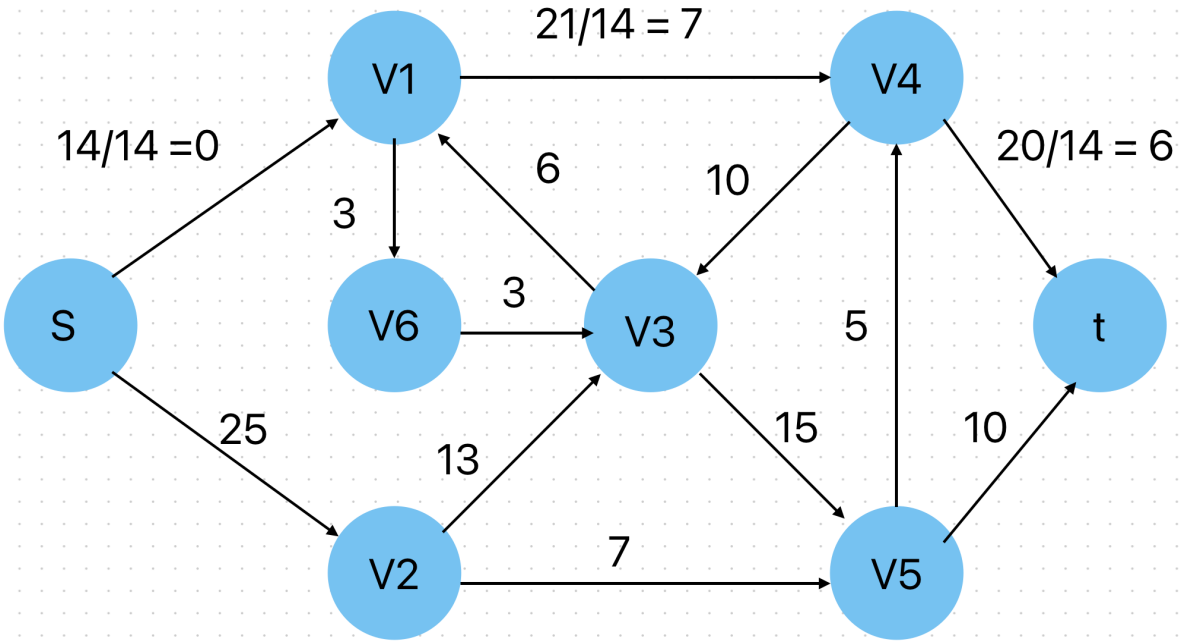
a) Resolving the antiparallel edge issue:
consider the graph:



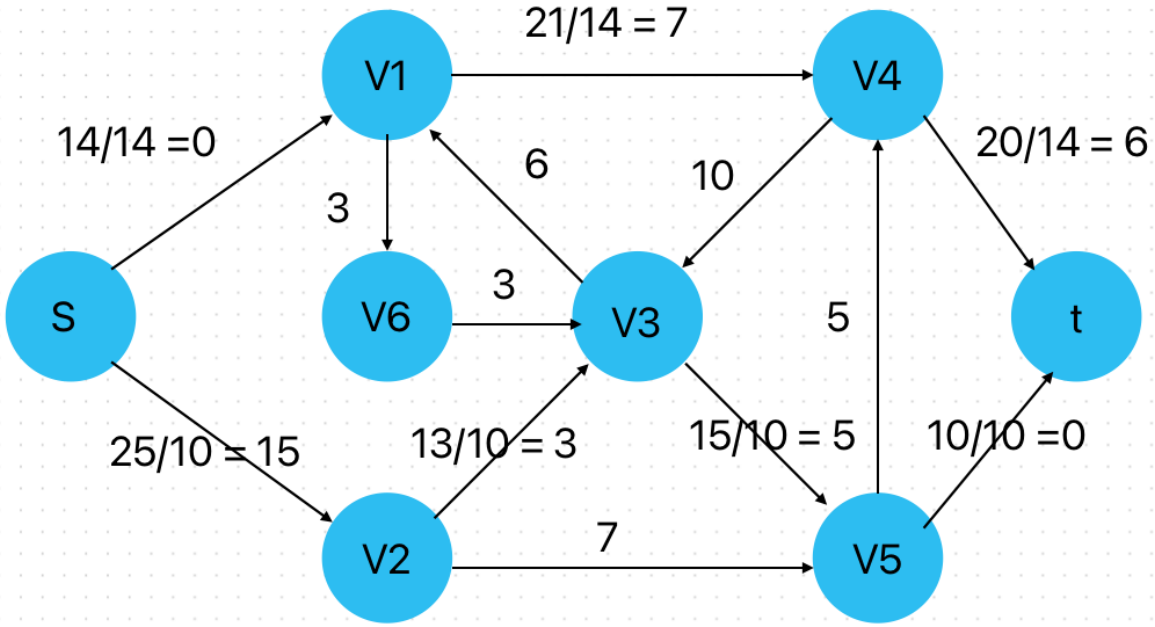
Here the antiparallel edge issue ir resolved by introducing a new vertex V& which represents the edge (V_1, V_3)

b)
To compute max flow using the Ford Fulkerson algorithm we begin by setting the total flow as zero, and then finding an augmenting path.

In this initial case we can find the initial augmenting path:
$(s, v_1, c = 14), (v_1, v_4, c = 21), (v_4, t, c = 20)$
The smallest capacity in this path is 14.
The edges of this identified path can hence have a flow of 14, and we can take the difference with the capacity to get the residuals.
We then a new graph:

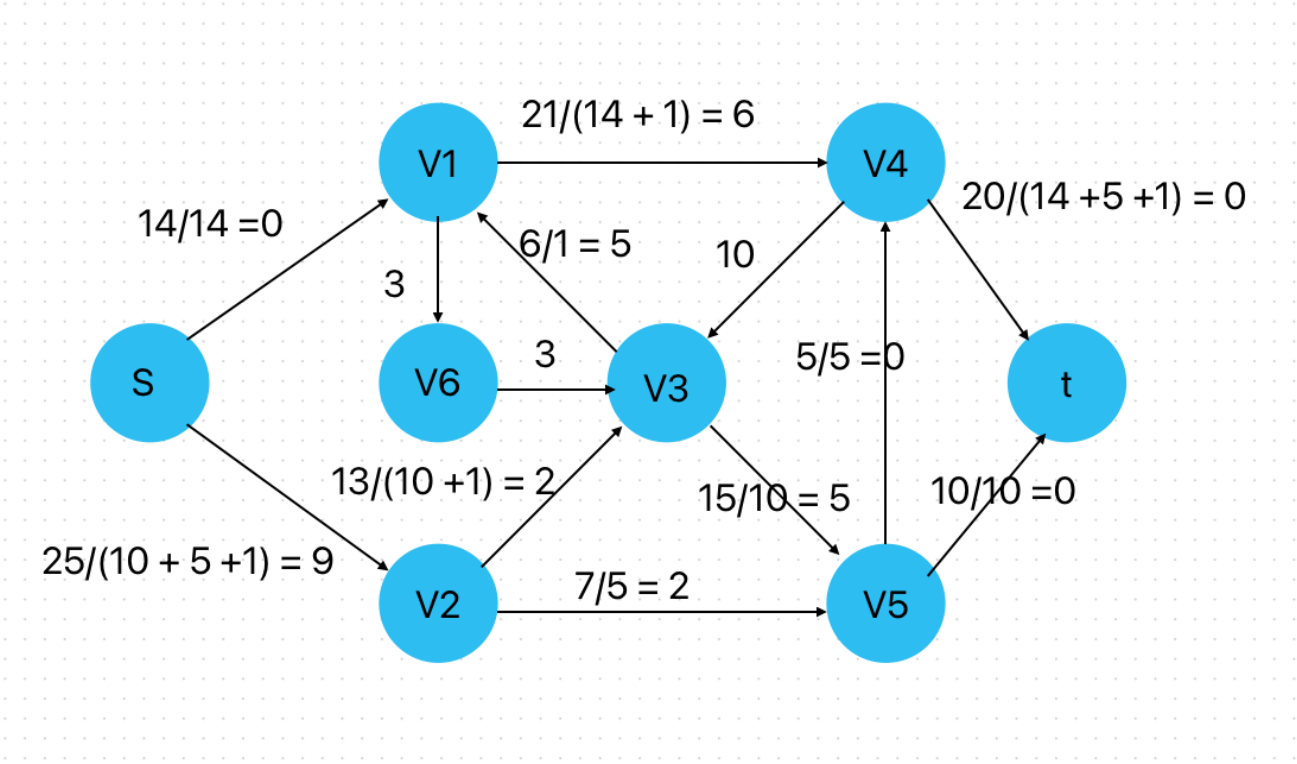


Now we can find another augmenting path.
Going by maximum available capacity we find:
$(v_5, t, c = 10)(v_3, v_5, c = 15), (v_2, v_3, c = 13), (s, v_2, c = 25)$
The minimum capacity of this flow is 10. We update:

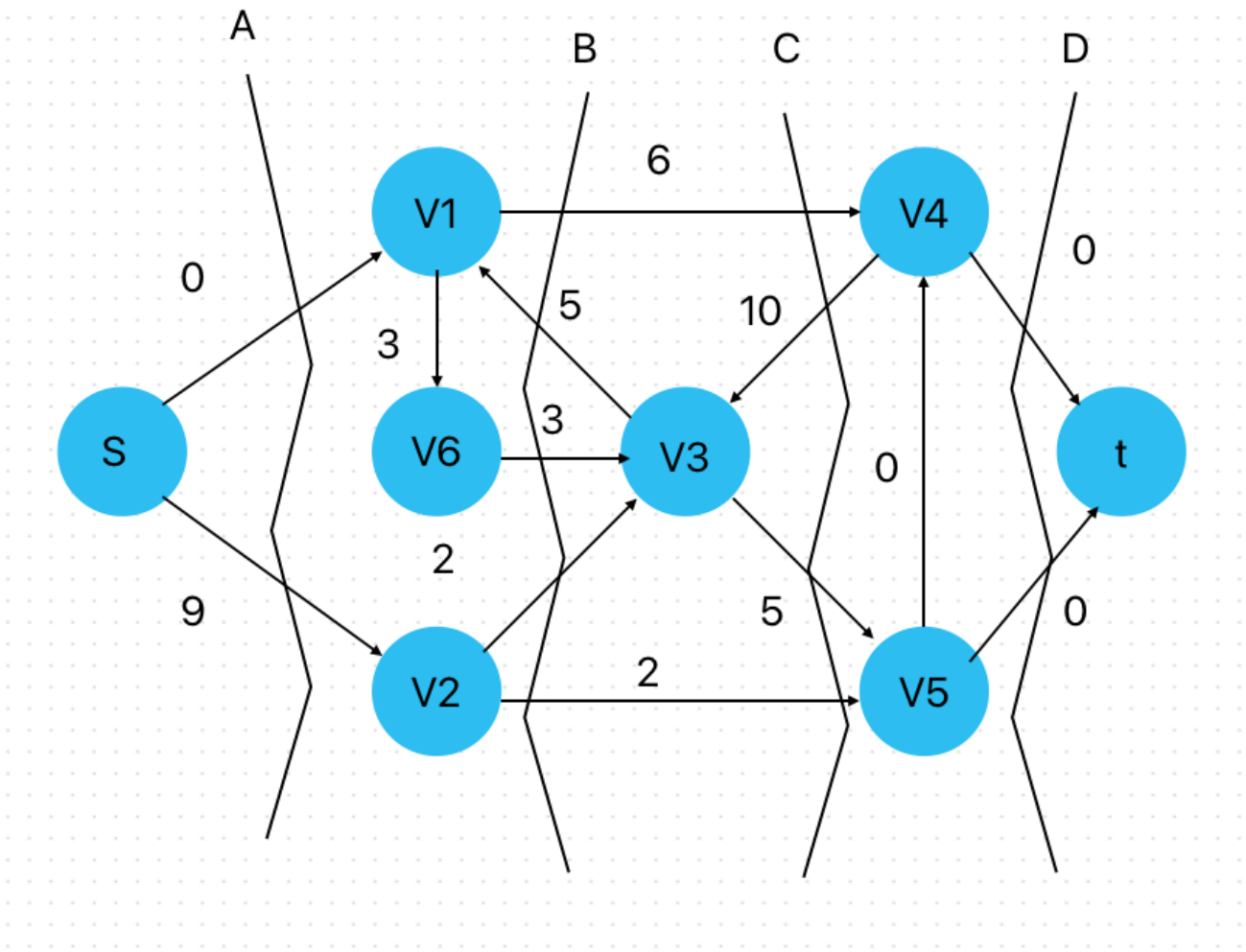


Now, by we can observe that the remaining possible flow is 6, which must go through edge (v_4, t)
To achieve this we can have a flow of 5 going the route of $(s \to v_2 \to v_5 \to v_4 \to t)$

And then at last a flow of 1 through $(s \to v_2, \to v_3 \to v_4 \to t)$
And we achieve the maximum flow of 30:



c)
consider the cuts [A,B,C,D] as marked over the solution to task b) in the next picture



Clearly, cut D marks the bottleneck of the flow graph.

d)

The ford fulkerson algorithm can be thought of as a series of "shortest path" calculations. That is, for a graph with a given capacity, we find the "shortest" paths being the augmenting paths, update the graph with new capacities and run again. As such, the algorithm must run some pathfinding algorithm X number of times, dependent on the properties of the graph. Say that we use djikstras algorithm to find augmenting paths, which has a running time of $O(VE)$. Further say that the overall solution to the problem involves finding X augmenting paths. Then the running time is expected to be $X \times O(VE) = O(XVE)$