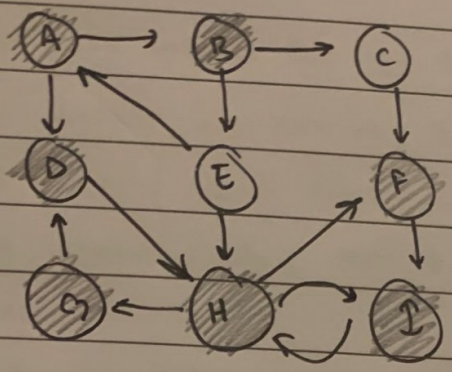
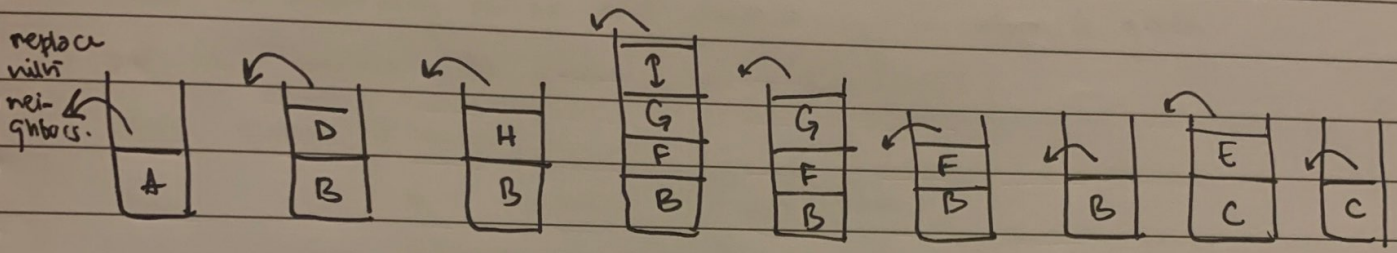
KAYYAN MINHAJ
20k-0143 | BCS-5F

Date

DAA
ASSIGNMENT #4

Q1(a)





visited: A | D | H | I | G | F | B | E | C

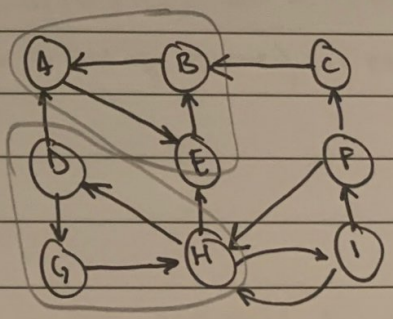
As we pop from stack we write the vertex as visited.

(b) No sets written?

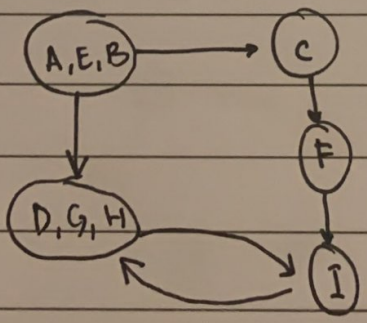
(c) Strongly Connected Components.

- A-E-B
- D-G-H
- I
- F
- C

G^T =>



Component Graph



Date

Q2.

(a) Linear time implies the size of the input. (n num of vertices). Assuming the graph has n vertices, we can use a modified version of BFS to mark adjacent nodes 2 diffent marks (color) than their neighbours. If we can mark entire graph with alternating symbols / colors then it is a bipartite graph.

ALGORITHM

1.) select an arbitrary vertex and label it X / or colour it green.

1.5) put the vertex into queue (queue.push(node)).

2.) while queue != empty {

n = neighbour of top node (parent)

if (n is not labelled): label it Y / or colour it red. & push in queue

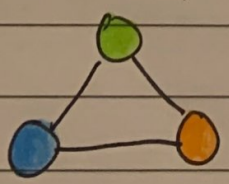
if (n is same label/color as its parent): exit program; it is not bipartite

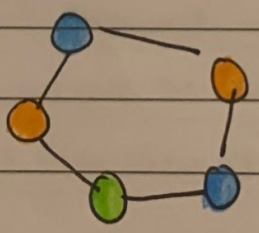
}

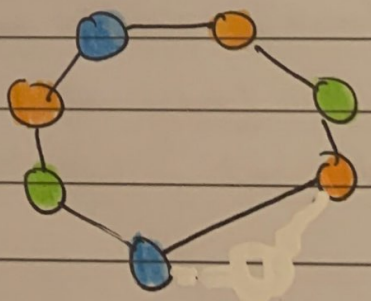
3.) it is bipartite, end program.

The running time will be similar to BFS which is O(|V| + |E|) but since we ensure O(1) work |V| times (because we are inserting in & out of queue) we can determine it as linear.

(b) we will need a minimum of 3 colors, and we can see this by drawing a cycle of 3 nodes, ensuring no node has same color as its adjacent neighbours.





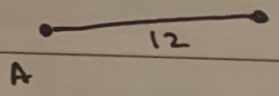


Date

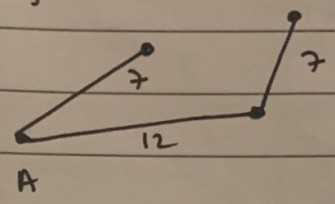
Q3.

(a) At each step just select maximum weighted edge to achieve max spanning. steps.

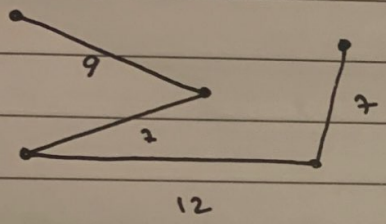
(i)



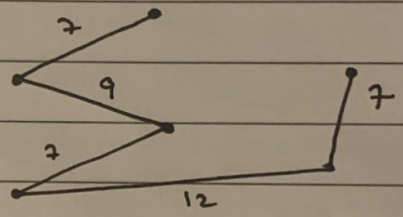
(ii) & (iii)



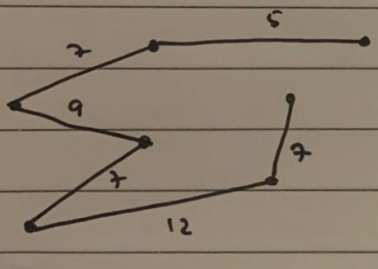
(iv)



(v)



(vi)



MAXIMUM SPANNING TREE

cost: 47

(b) ~~No, they will always yield same results since both are algorithms for MAX/MIN spanning trees~~

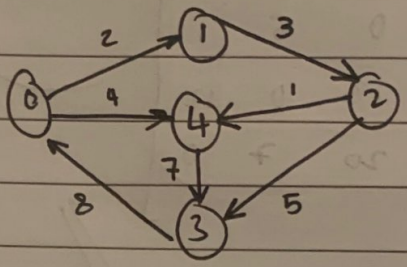
↳ if all weights are diffent from one another then both Prim's & Kruskal will give same answer. Else if some weights are same (like in above graph) then either both algo can give same ans or diffent since they have diffent ways of choosing max weight that satisfies same condition.

Date _____

Q4. All pair shortest path (Floyd Warshall).

D4 //

	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	∞	∞	0	∞
4	∞	∞	∞	7	0



D0 //

	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	∞	0	12
4	∞	∞	∞	7	0

D1 //

	0	1	2	3	4
0	0	2	5	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

D2 //

	0	1	2	3	4
0	0	2	5	10	4
1	∞	0	3	8	4
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

D3 //

	0	1	2	3	4
0	0	2	5	10	4
1	16	0	3	8	4
2	13	15	0	5	1
3	8	10	13	0	12
4	15	17	20	7	0

NEXT PAGE.

Date

D_4 //

	0	1	2	3	4
0	0	2	5	10	4
1	16	0	3	8	4
2	13	15	0	5	1
3	8	10	13	0	12
4	15	17	20	7	0

Final Answer
(All pair shortest path).

//

> Floyd Warshall uses the dynamic programming paradigm with $O(|V|^3)$ time complexity and $O(|V|^2)$ space complexity.

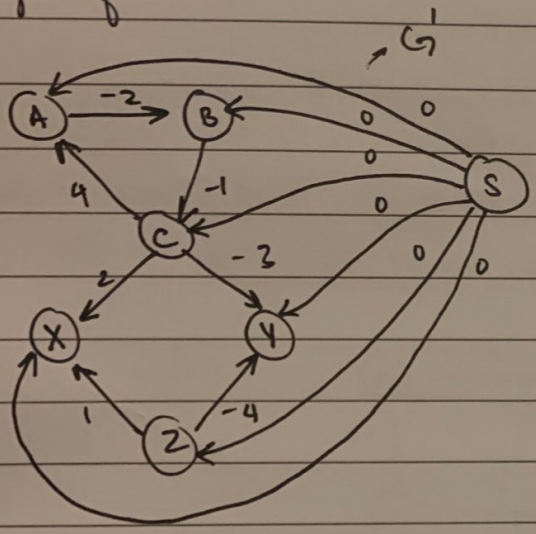
Date

Qs. JOHNSONS ALGORITHM SUMMARY

→ Johnsons Algorithm is an All pair shortest path algorithm (using single source).

→ Can be applied on directed graphs with -ve weights (where dijkstra fails).

→ We can add a new source node that is connected with ALL vertices and has weight of 0



→ Now we compute shortest paths from 's' to all other vertices

→ This will give us finite shortest path distances

→ adding S does not give us any new paths.

→ Now we can use Bellman Ford to compute shortest path.

→ P_v = length of a shortest S-v paths.

→ for each edge $E=(u,v)$ define $c'_e = c_e + P_u - P_v$

→ After new reweighting, all edge weights became non-negative (reweighting preserves shortest path)

→ If there is a negative cost cycle, that cycle has to be in original graph G.

→ After reweighting, run Dijkstras Algorithm in G with edge lengths $\{c'_e\}$ to compute shortest paths

→ We will get shortest paths with respect to reweighted edges but we need in respect to original weights so we will simply subtract $(P_u + P_v)$ from shortest paths

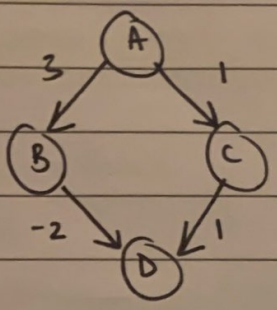
$$d(u,v) = d'(u,v) - (P_u + P_v)$$

(we get this from dijkstras)

→ running time = $O(mn \log(n))$

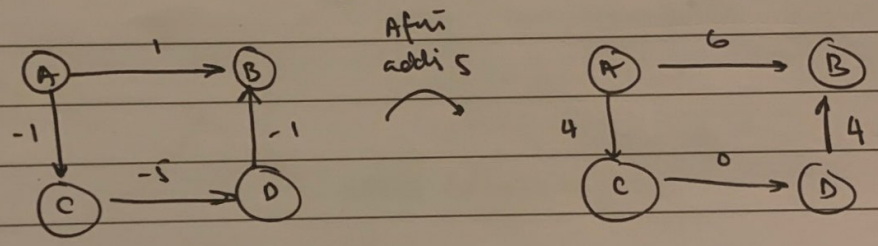
Date _____

Q6. (a)



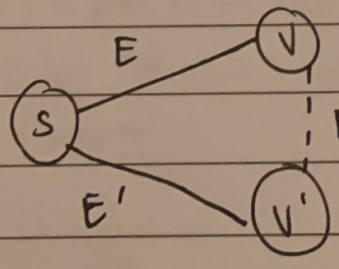
if we apply Dijkstra to this graph it will produce incorrect result b/c it will give cost of going from A to D as 2 (1+1) but in reality it can be 1 (3-2) but since Dijkstra does not apply on -ve weights, it will disregard the edge from (B)—(D).

(b) No, this will not work because if actual shortest path has more edges than potential shortest paths, the paths with more edges have their weights increased by more than the path with fewer edges ex.



Shortest path from (A)—(B) should be A – C – D – B (-7) but if we add 5 to each edge, this increases actual shortest path to 8 (which is NOT shortest) & algo will return this.

(c) Assuming there is no negative weight cycle in graph G & all -ve weights are connected to source. If some vertex $v \neq s$ is connected with some -ve weight edge e, the shortest path from s to v must cover the -ve weight edge e.



Both E & E' are -ve. Assume shortest path from S to V is $S \to V' \to V$, then we can say

$$E' + P < E$$

& we know $E + E' + P < 2E < 0$ which is incorrect hence contradic

Date

Q7.

BREADTH FIRST SEARCH

(i) Adjacency Matrix

↳ $O(|V|^2)$

↳ everytime we want to find edges adjacent to a node, we would have to traverse the entire array which is of length $|V|$? (num. of vertices).

(ii) Adjacency List.

↳ $O(|V| + |E|)$

↳ in adjacency list edges are already available to us so it takes times proportional to number of adjacent vertices which on summation over all vertices is $|E|+|V|$

DEPTH FIRST SEARCH

(i) Adjacency Matrix

(ii) Adjacency List

same as BFS!!!

KRUSKAL

(i) Adjacency Matrix

↳ $O(|V|^2 + E \log(E))$

↳ it takes $O(|V|^2)$ to find all the edges and $O(E \log E)$ to sort the edges so we can find smallest weights.

(ii) Adjacency List

↳ $O(E \log(|V|))$

↳ considering we need to sort the edges into non-decreasing order by weight 'w' for each edge.

Date _____

PRIMS

(i) Adjacency Matrix

↳ $O(V^3)$

↳ This is the traditional way where we have an array in addition to the matrix which has minimum distance of that vertex to a node.

Without using this array, we would have to iterate through all E edges every single time which at worst contains V^2 edges so time complexity would be $O(V^3)$

(ii) Adjacency List

↳ $O((E+V) * \log(V))$

- $(E+V)$: similar to BFS we need total vertices & edges
- $\log(V)$: used to extract current value of min heap.