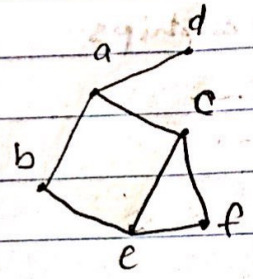
328 HW #9

1. Draw graph, size, order, deg(v), adj(v)

a. $E = \{\{a,b\},\{b,e\},\{a,c\},\{a,d\},\{c,e\},\{e,f\},\{c,f\}\}$



size = 7

order = $|V| = 6$ $\{d, a, c, b, e, f\}$

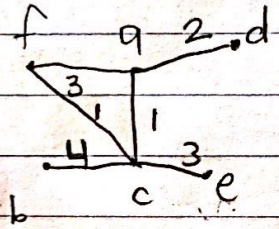
deg(a) = 3 deg(b) = 2 deg(c) = 3

deg(d) = 1 deg(e) = 3 deg(f) = 2

adj(a) = d, c, b adj(b) = a, e adj(c) = f, e, a

adj(d) = a adj(e) = f, c, b adj(f) = c, e

b. $E = \{\{a,f,3\},\{b,c,4\},\{a,c,1\},\{a,d,2\},\{c,e,3\},\{c,f,1\}\}$



size = 6 order = $|V| = 6$

$\{d, a, f, b, c, e\}$

deg(a) = 3 adj(a) = d, f, c

deg(b) = 1 adj(b) = c

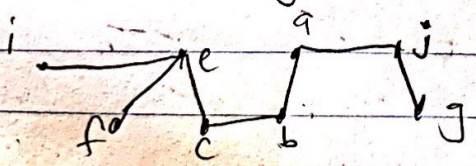
deg(c) = 4 adj(c) = e, a, b, f

deg(d) = 1 adj(d) = a

deg(e) = 1 adj(e) = c

deg(f) 2 adj(f) = c, a

c. $E = \{(j,a),(j,g),(a,b),(a,e),(b,c),(e,c),(e,f),(e,i)\}$



size = 7

order = 8 (i, f, e, c, b, a, j, g)

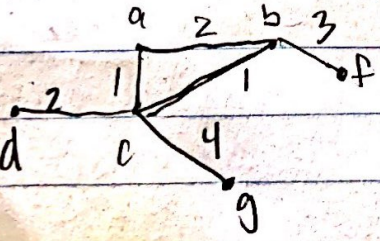
deg(a) = 3 deg(b) = 2 deg(c) = 2 deg(e) = 4 deg(f) = 1 deg(g) = 1

deg(i) = 1 deg(j) = 2

adj(a) = e, j, b adj(b) = a, c adj(c) = e, b adj(e) = f, c, i, a

adj(f) = e adj(g) = j adj(i) = e adj(j) = a, g

d. $\{(a,b,2),(a,c,1),(b,f,3),(c,b,1),(c,g,4),(d,c,2)\}$



size = 6

order = d, c, g, a, b, f = 6

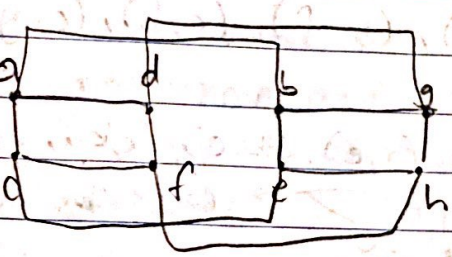
deg(a) = 2 deg(b) = 3 deg(c) = 4 deg(d) = 1 deg(f) = 1 deg(g) = 1
adj(a) = c, b adj(b) = a, c, f adj(c) = a, b, d, g
adj(d) = c adj(f) = b adj(g) = c

2. {(000, 100, 010, 001, 110, 011, 101, 111)}
a, b, c, d, e, f, g, h

Q_1 = 1 — 0

Q_2 = 00 — 01 / 10 — 11

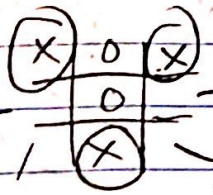
Q_3 =



Hamilton cycle → a, d, g, b, e, h, f, c, back to a

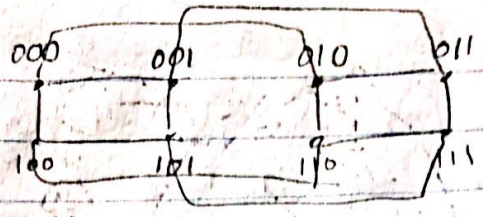
3. Provide formulas for both order/size of Q_n. Explain
The order of Qn would be 2^n because there will be 2^n three digit bits with the length of n.
As for the size of Q_n, we see that the number of bit length each graph of Q has, the amount of adjacent edges there are. Q_1 has length 1 with one adjacent edge, while Q_2 has a length of 2 with 2 adjacent edges

4. vertices: reachable tic-tac-toe board positions
Edges: valid moves



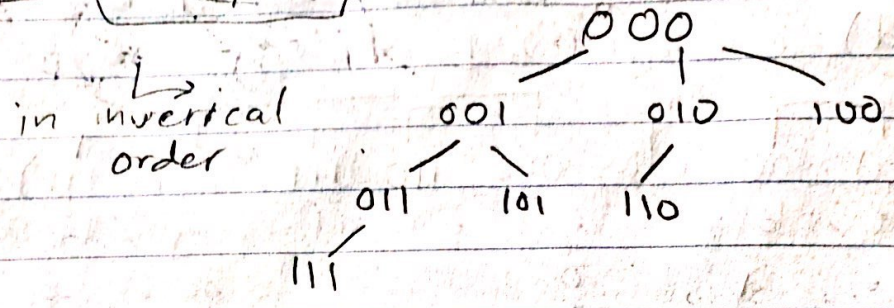
we had 3 diff. reachable positions (vertices) → 4 diff. valid moves (edges) for "O" player to move to
∴ in-degree = 3 and out-degree = 4

5.



Q3

BFS of Q3:

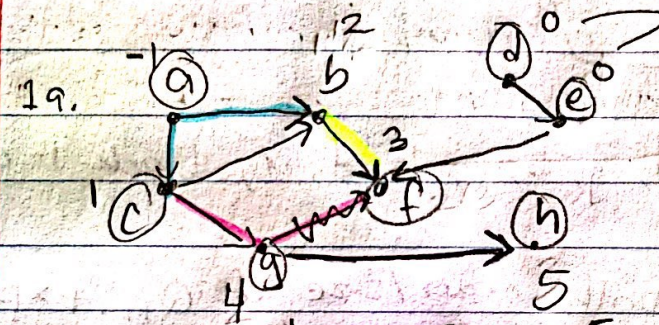


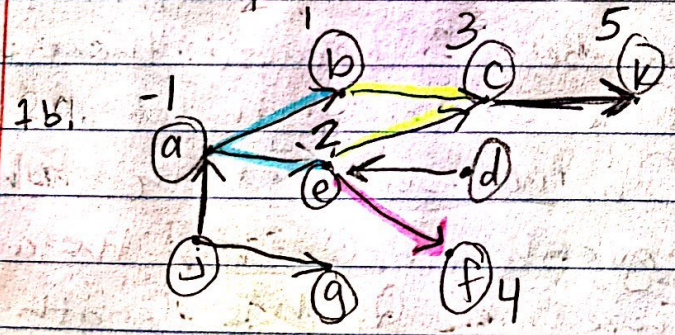
in vertical order

6. a. BFS algorithm:

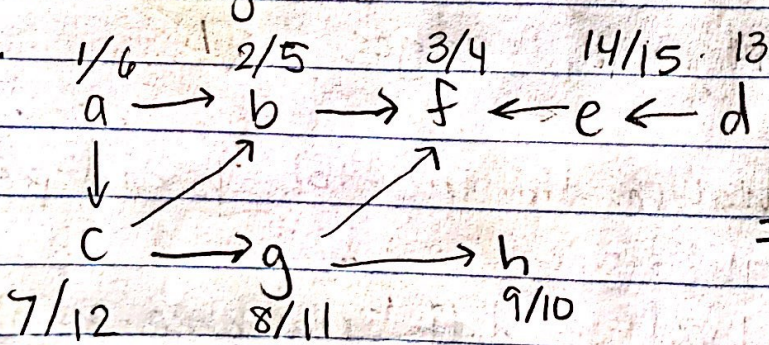
1. $E = \{(a,b), (a,c), (b,f), (c,b), (c,g), (d,e), (e,f), (g,f), (g,h)\}$

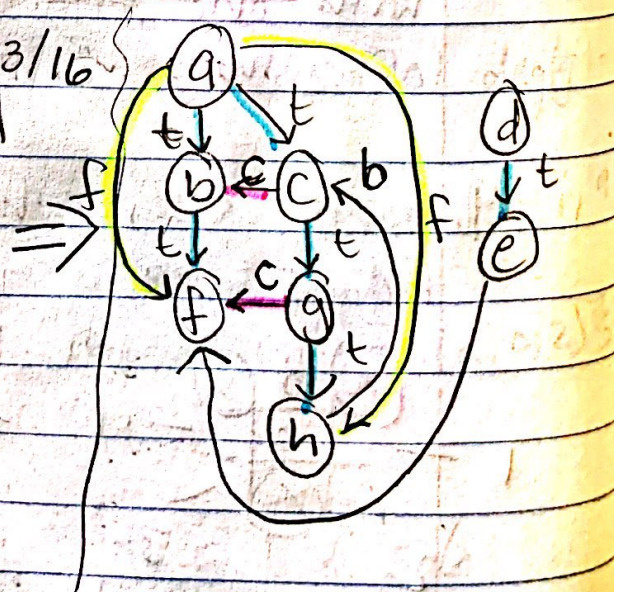
2. $\{(j,a), (j,g), (a,b), (a,e), (b,c), (c,k), (d,e), (e,c), (e,f)\}$



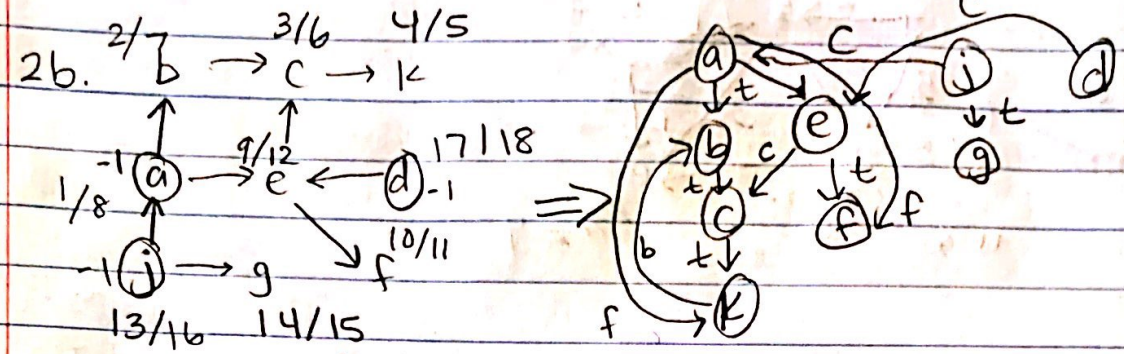


b. DFS algorithm





2b.



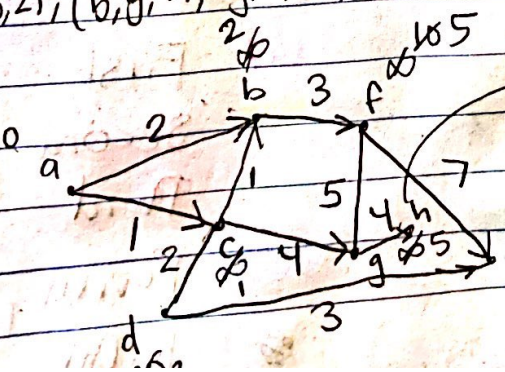
7. Running time of:

a) The running time of Breadth First search would be $O(V+E)$ because BFS looks at each of its neighbors and move onto each of the nodes. Therefore, we have to iterate throughout each of the edges and vertices of the graph. Since we aren't iterating over all the edges in the graph for each vertex, $T(n)$ can't be $V \cdot E$, but rather the sum of the two.

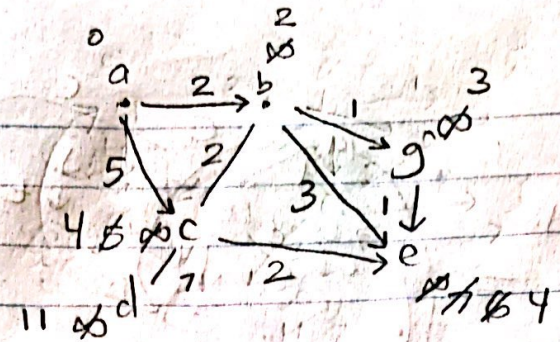
b) The running time of Depth-First Search represented by an adjacency matrix would be $O(V \cdot V)$ because DFS looks at every vertex then since we are looking at a matrix → $\begin{bmatrix} a & c \\ b & d \end{bmatrix}$, this would account for the other V in the runtime. The first V was used to search and DFS alone takes $O(V)$ so we get $O(V^2)$ time complexity.

8. $E = \{(a,b,2), (a,c,1), (b,f,3), (c,b,1), (c,g,4), (d,c,2), (d,e,3), (e,f,7), (g,f,5), (g,h,4)\}$

$E = \{(a,b,2), (b,g,1), (g,e,1), (b,e,3), (b,c,2), (a,c,5), (c,e,2), (c,d,7)\}$



8b.



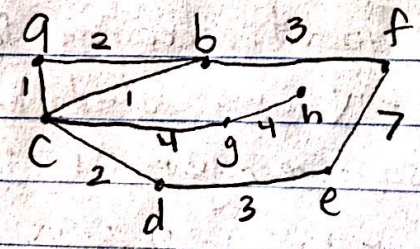
9. The average-case running time of Dijkstra's algorithm if nodes are stored in binary-heap would be loiner than the worst case of the algorithm because it is limited by the binary heaps. The $T(n)$ of worst-case would be $O(E \log V)$ where initializing and building the heap would both take $O(n)$. To find the minimum, it would take $O(V \log V)$ and to check that a certain node gets to the top of the queue then removes it, it takes $(E \log V)$

10. $E = \{(a,b,2), (a,c,1), (b,f,3), (c,b,1), (c,g,4), (d,c,2), (d,e,3), (e,f,7), (g,f,5), (g,h,4)\}$

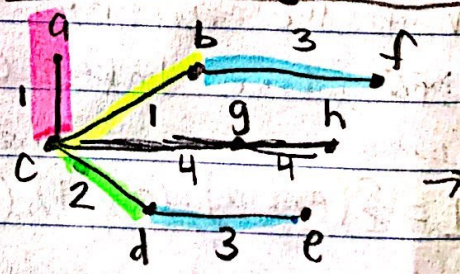
$E = \{\{a,b,3\}, \{a,d,4\}, \{a,e,4\}, \{b,c,10\}, \{b,e,2\}, \{c,f,6\}, \{c,g,1\}, \{d,e,5\}, \{d,h,6\}, \{e,f,$

Undirected

1a)

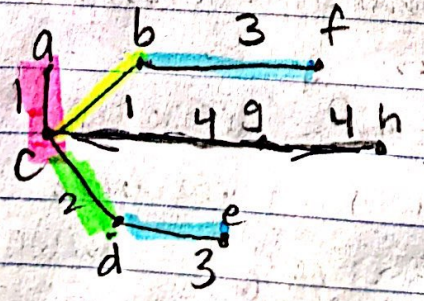


Kruskal's algorithm



→ 23

1b) Prim's Algorithm



→ 23

no cycles

First

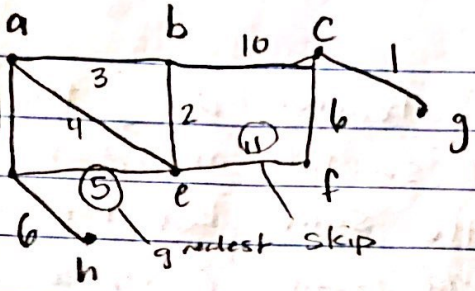
Second

Third

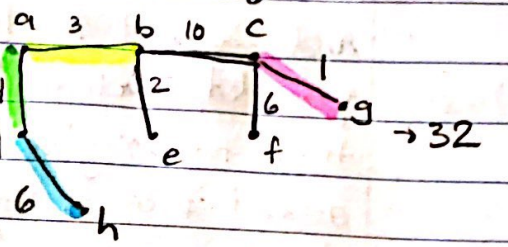
Fourth

(combined)

2a)

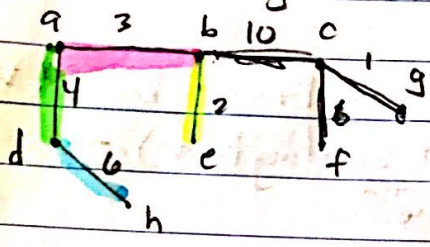


Kruskal's Algorithm



→ 32

2b) Prim's Algorithm



→ also weight of 32.

11. I think Prim's and Kruskal's algorithms work even if we have negative weights because we can do the same thing we did when we had negative numbers in an array that we were going to do quick sort on where we shifted all the numbers making all the negative numbers positive then shifting back after doing the algorithm. Furthermore, since these two algorithms are looking for the MST, I think negative numbers shouldn't affect the result.

12. How to use Prim's / Kruskal's to find maximum spanning tree? I'm guessing we just have to do the opposite of what we did for minimum spanning tree. We would take the same greedy approach but instead of picking the smallest weight, we pick the biggest weight and loop thru until all nodes have been visited. The only difference between how the two algorithms would be implemented is that Prim's would look towards edges that meet the visited vertices. On the other hand, Kruskal's method can skip around and pick the largest edge

and add it to the MST. However, using the hint provided, we could just switch around the order so that the weights stay the same, but the minimal weights aren't used as they would create a cycle which we can't have.

13. G = (V, E) → adjacency matrix → Prim's in $O(V^2)$ run time

```
Prim (int[][] adjMatrix, int startingNode) {
    int l = adj.length  → Queue  (initialize)
    int w = adj.[0].length
    for (i : j) {                          // go through matrix
        for (k : w) {
            if (minimum #) {  weight[i][k] < minimum
                add to spanning tree;
                store minimal num;
    print spanningTree;
```

This would be $O(V^2)$ because we iterate through each value of the matrix using rows and columns then check if our weight is the smallest amongst the other edges. Prim's being greedy allows us to do this and we store these values. I think we could have also used a queue similar to BFS and run through that iteratively.

14. The most efficient algorithm to find the shortest path between two vertices in a directed graph where the weights of all edges are equal would be DFS with a time complexity of $O(V+E)$ and also BFS since they both have the same runtimes. Furthermore, the best case is also the worst case and

vice versa meaning that either algorithm would be a good choice when wanting accurate fast results, given the prerequirements are met.

15. $G = (V, E)$ → check for cycle → for undirected graph run in $O(V)$ time:
Similar to lab 8 where we used the DFS function to check if the graph was a directed acyclic graph, we do the same here, since the undirected graph will have cycles if there are back edges. The back edges are already connected elsewhere and if a vertex connects to one of the ancestors this means that there is a cycle. This would be $O(V)$ because if we found a back edge we would have found it before traversing through the entire graph.

16. When the child node is already visited/marked with the same color as the parent node, that means that the simple graph has an odd cycle. If the color was opposite to that of its parent, it would be an even cycle.

```
Color Graph(G) {
    for v = adj[x]
        if v isn't colored
            color it the opposite of x
            Q.push(v)
        if v is colored & same color of x
            return true; // odd cycle (same color)
    return false; // not odd (opposite color)
```