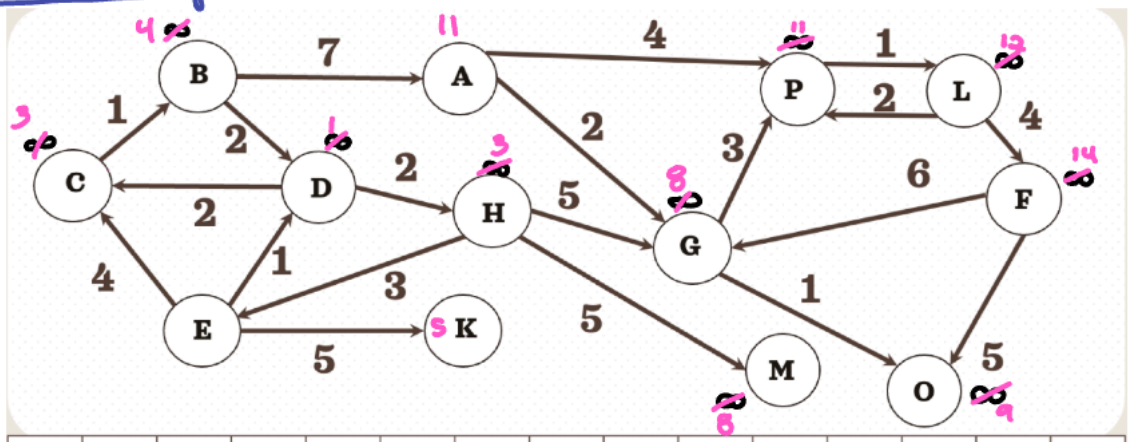
Problem 1:

Dijkstra's Algorithm



Shortest path: [E, D, C, H, B, K, M, G, O, A, P, L, F]

Iterations:

Visited [E]
- update K, D, C

Visited [E, D]
- update cost C, H

Visit [E, D, C]
- update cost to B

Visit [E, D, C, H]
- update cost of G + M

Visited [E, D, C, H, B]
- update A

Visit [E, D, C, H, B, K]
- update nothing

Visit [E, D, C, H, B, K, M]
- update nothing

Visited [E, D, C, H, B, K, M, G]
- update O + P

Visited [E, D, C, H, B, K, M, G, O]
- update nothing

Visited [E, D, C, H, B, K, M, G, O, A]
- update nothing

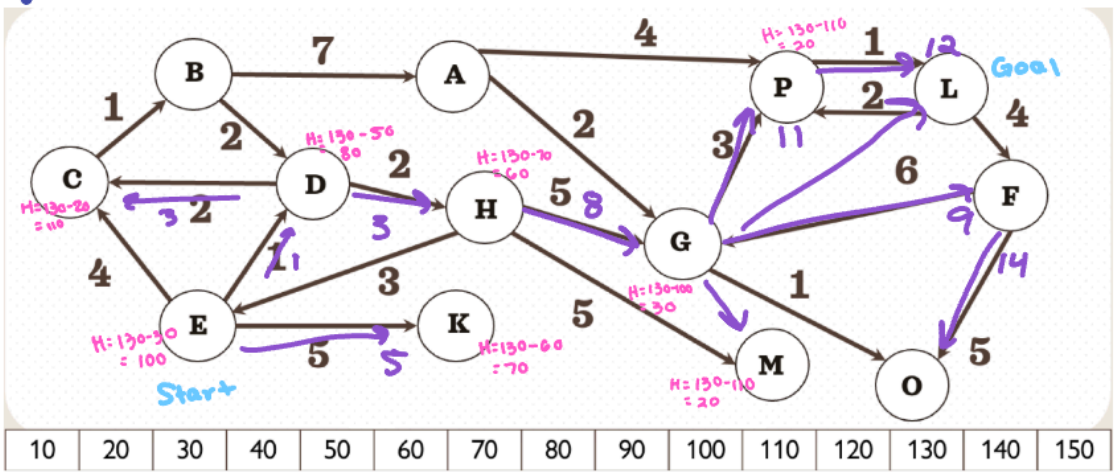
Visited [E, D, C, H, B, K, M, G, O, A, P]
- update L

Visited [E, D, C, H, B, K, M, G, O, A, P, L]
- update F

Visited [E, D, C, H, B, K, M, G, O, A, P, L, F]

Problem 2:

A* Algorithm



node	x-pos	f	g	h
E	30	100	0	100
D	50	81	1	80
C	20	113	3	110
K	60	75	5	70
H	70	63	3	60
G	90	38	8	30
F	120	19	9	10
O	130	14	14	0
M	110	29	9	20
P	110	31	11	20
L	130	12	12	0
A	80	∞	∞	50
B	30	∞	∞	100

E → D → H → G → L

Problem 3:

A* reaches H in fewer steps than Dijkstra. A* focuses on H, and only visits E and then H. This is because the h(H)=0 (since it is the goal) and it's f=3 which is lower than it's neighbors, so A* skips other nodes and immediately goes E → H. Dijsktra, however, does not consider the h(x) value, only g(x), therefore it visits E and D before reaching H.

Problem 7:

Directed or Undirected (DirectedOrUndir)

- Time complexity: O(n^2) where n is the number of rows and columns in the input matrix. The code checks if the matrix is square and only contains 0s and 1s, which takes O(n^2) time since we are iterating through our nested for loop for each row and column.
- Space complexity: O(n^2) where n is the number of rows and columns in our input matrix. The matrix is stored as a 2D array with a size of (n * n). This space is used to store whether each pair of vertices has a direct connection.

Every Path (EveryPath)

- Time complexity: When the graph is built, it takes O(V+E) time where V is the number of vertices and E is the number of edges.
- Space complexity: O(V+E+L) where V is the number of vertices, E is the number of edges, and L the length of the recursive stack. The adjacency list representation takes O(V+E) time, whereas the recursive stack grows up to length of L, which takes O(L) time.

Draw Graph

- Time complexity: O(n^2) where n is the number of (vertex, x) pairs, which is also the number of nodes in the graph. Adding the n amount of nodes makes O(n) calls to vertices.indexOf(vd), which is O(n) itself. Therefore, that loop takes O(n^2) time.
- Space complexity: O(n), where n is the number of (vertex, x) pairs, which is also the number of nodes. The auxiliary structures (list, regex matcger, GraphStream node and edge storage) used in the code also are linear.