

by: Khr
Kamran Khan

Graded Assignment 1

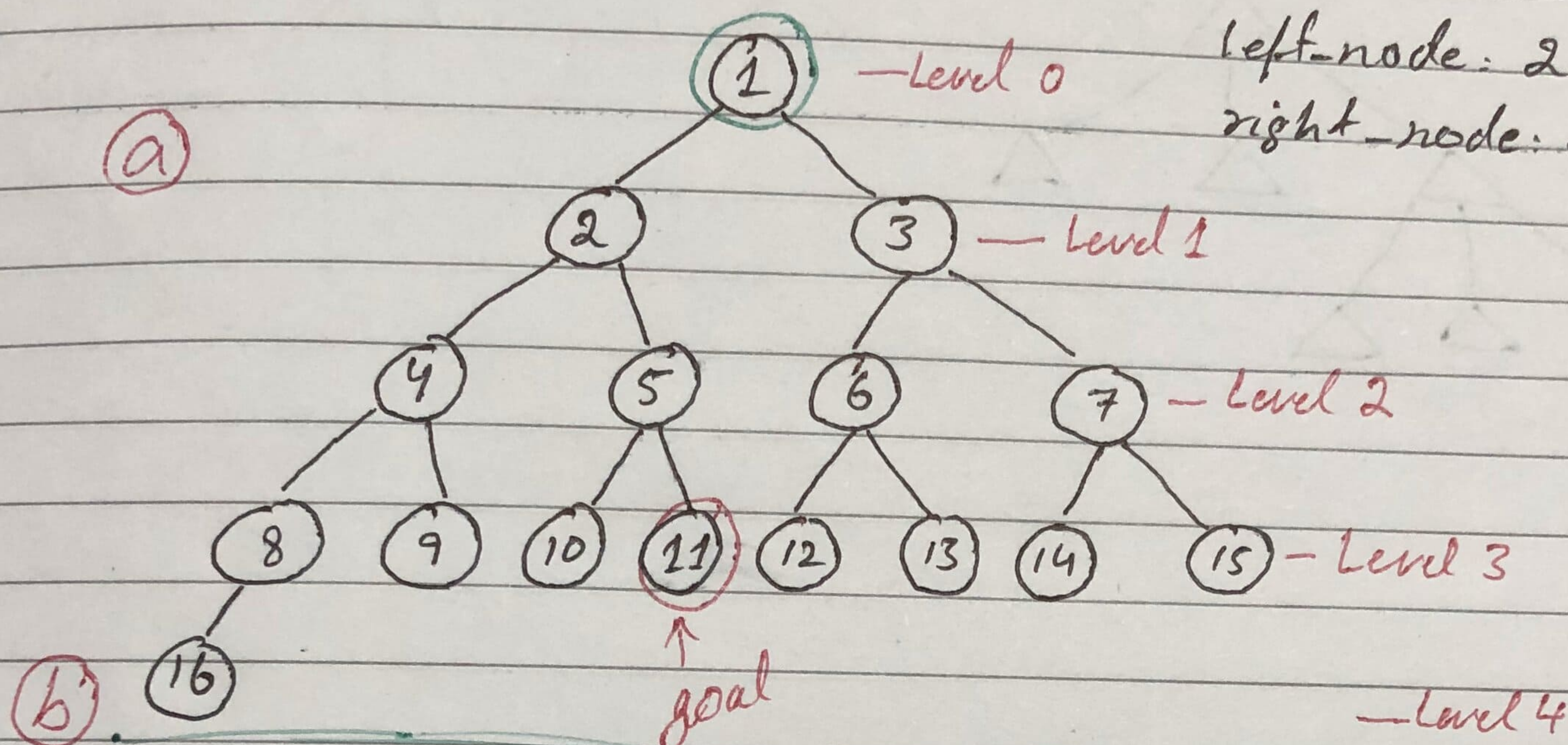
Q1:-

Start:- node 1
goal: node 11

$K = 1, \dots, 70$

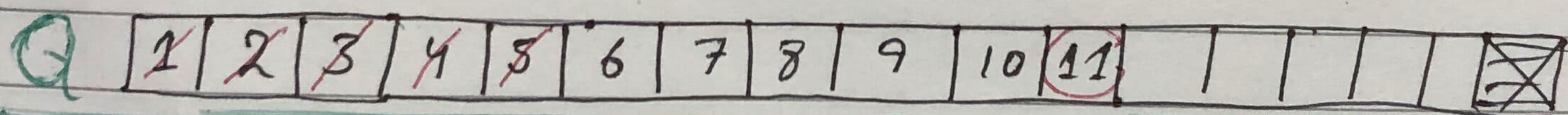
left-node: $2k$

right-node: $2k+1$



→ s: 1
→ g: 11
Limit = 3

BFS:- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11



LDFS:- 1, 2, 4, 8, 9, 5, 10, 11

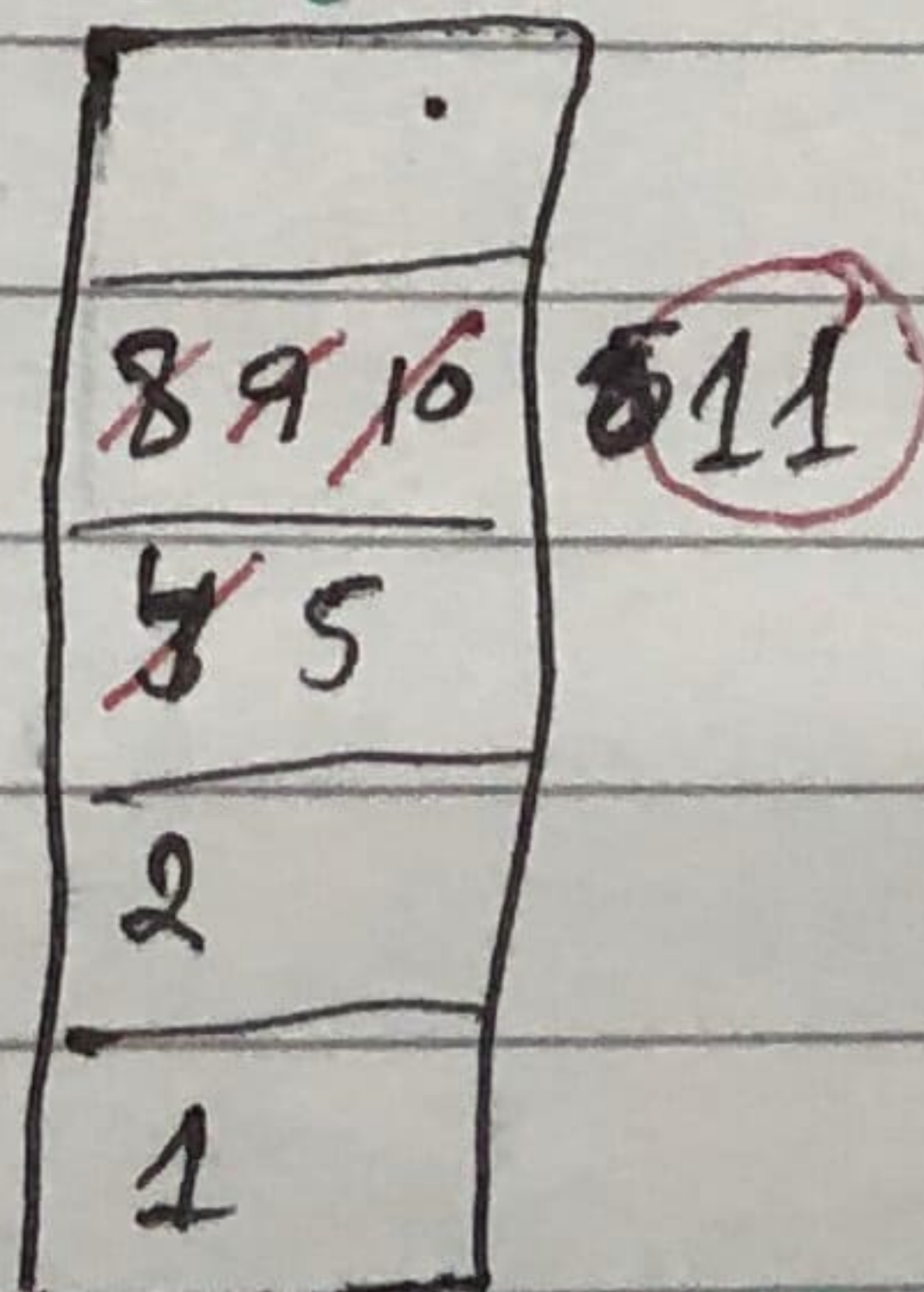
start = 1

goal = 11

limit = 3

Depth-Limited-Search

Stack



Iterative Deepening Search:

iteration	
d=0 1st iteration	[1]
d=1 2nd iteration	[1→2→3]
d=2 3rd iteration	[1→2→4→5→3→6→7]
d=3 4th iteration	[1→2→4→8→9→5→10→11]

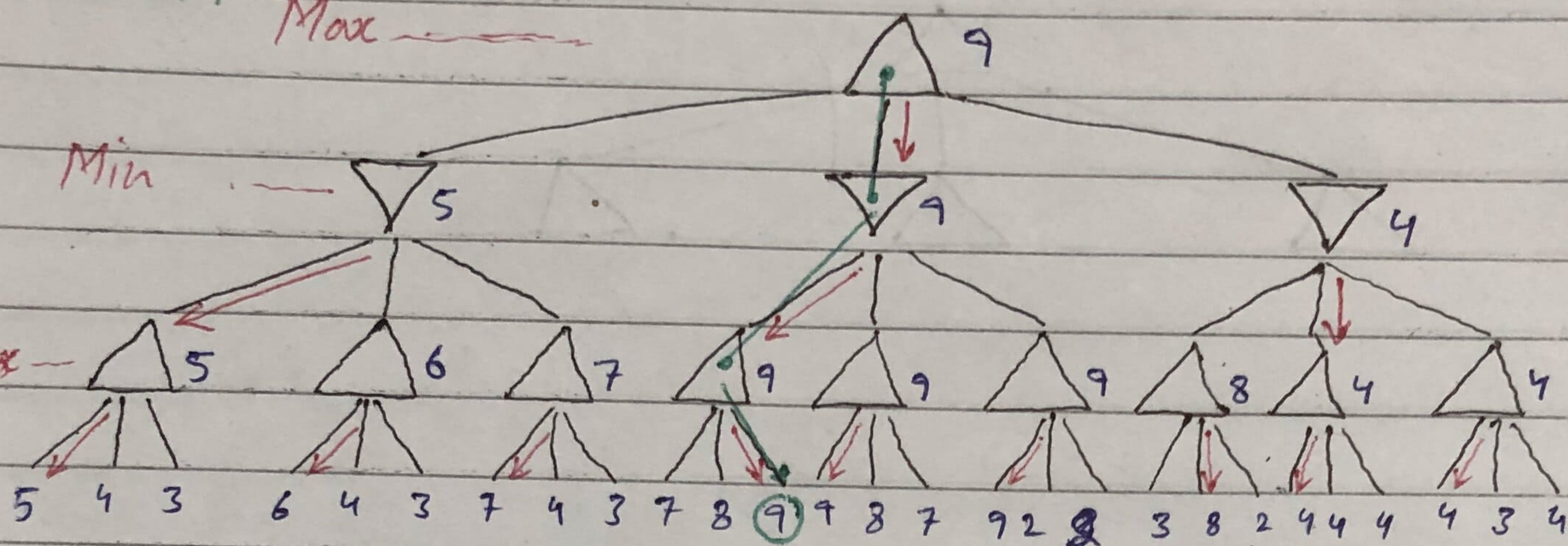
Q2:-

apt

Max

Min

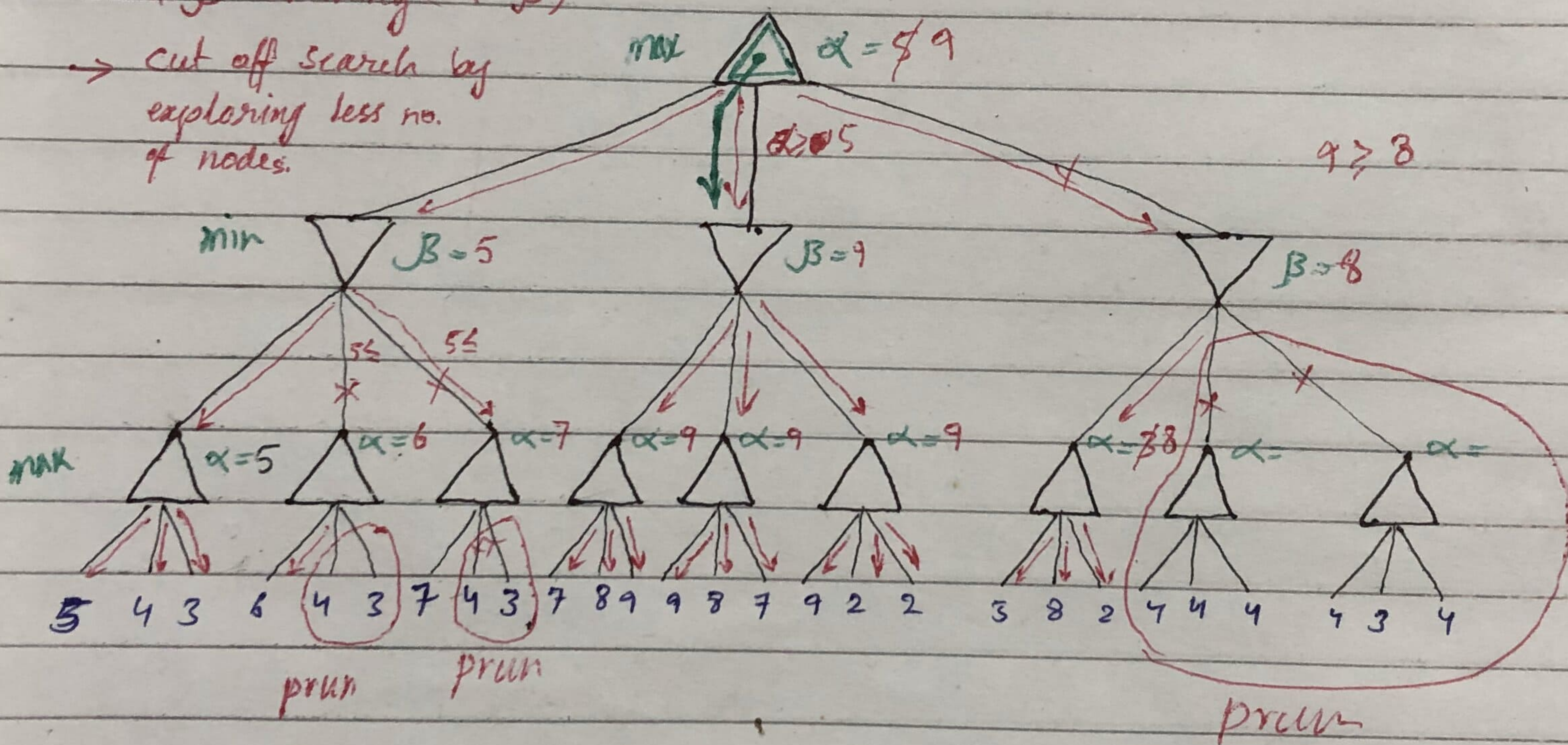
Max



b pt-

α β Pruning (α - β)

→ cut off search by exploring less no. of nodes.



$$O(b^{d/2})$$

for min β (default value $+\infty$)
for max α (default value $-\infty$)

α , β pruning is advance version of Minimax
 α , β pruning is much efficient than minimax algorithm.

Q 3:-

For that we want to introduce a new Z , such that if A 's domain has N values and B 's domain has another N values, Z 's domain is the cartesian product of A domain and B domain $N \times N$ values. Z 's domain is constructed value pair from A and B domain.

By doing this we have 3 binary constraints:

- 1) The values of A must be same as the first values of the pair Z .
- 2) The values of B must be the same as ~~the~~ the 2nd values of pair Z .
- 3) And the sum of the both A and B values pair in Z equals to C .

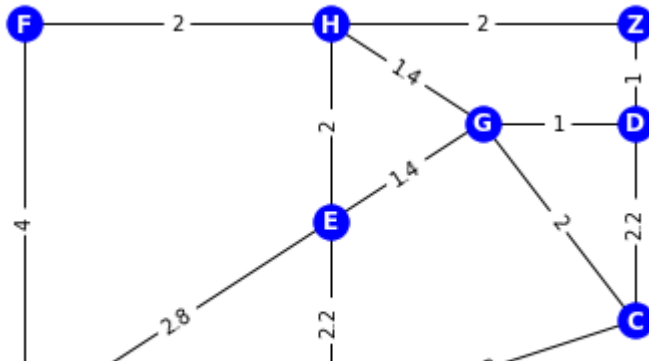
* When working with more than three variables the same procedure can be followed by doing it "step by step".

For example, if we have variables A, B, C and D , we would first reduce A, B and C to binary constraints, then include D and repeat the process until we have only binary constraints. Any unary constraint can be eliminated by moving the effects of the constraints to the domain of the variable.


```
1 ##### GRADED ASSIGNMENT - 01 #####
2 ##### VISUALIZATION #####
3
4 import networkx as nx
5 import matplotlib.pyplot as plt
6
7 G = nx.Graph()
8
9 G.add_node("A", pos=(0, 0))
10 G.add_node("B", pos=(1, 0))
11 G.add_node("C", pos=(2, .5))
12 G.add_node("D", pos=(2, 1.5))
13 G.add_node("E", pos=(1, 1))
14 G.add_node("F", pos=(0, 2))
15 G.add_node("G", pos=(1.5, 1.5))
16 G.add_node("H", pos=(1, 2))
17 G.add_node("Z", pos=(2, 2))
18
19 G.add_edge("Z", "D", weight=1)
20 G.add_edge("Z", "H", weight=2)
21 G.add_edge("H", "E", weight=2)
22 G.add_edge("H", "F", weight=2)
23 G.add_edge("H", "G", weight=1.4)
24 G.add_edge("G", "D", weight=1)
25 G.add_edge("G", "E", weight=1.4)
26 G.add_edge("D", "C", weight=2.2)
27 G.add_edge("C", "G", weight=2)
28 G.add_edge("C", "B", weight=2.2)
29 G.add_edge("B", "E", weight=2.2)
30 G.add_edge("G", "E", weight=1.4)
31 G.add_edge("E", "A", weight=2.8)
32 G.add_edge("F", "A", weight=4)
33
34 print("h(Z): 5.7\th(H): 4.5\th(D): 5")
35 print("h(F): 4\t\th(G): 4.2\th(E): 2.8\th(C): 4.1\th(B): 2.0\th(A): 0")
36
37 pos=nx.get_node_attributes(G,'pos')
38 labels = nx.get_edge_attributes(G,'weight')
39
40 nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
41
42 nx.draw(G, pos, with_labels=True, font_weight='bold',
43         edge_color='black', font_color='white', node_color='blue')
44 plt.savefig('simpleGraph.svg')
45 plt.show()
46
```



h(Z): 5.7	h(H): 4.5	h(D): 5
h(F): 4	h(G): 4.2	h(E): 2.8
h(C): 4.1	h(B): 2.0	h(A): 0



```

1 ##### GRADED ASSIGNMENT - 01 #####
2 from collections import deque #####
3 ##### IMPLEMENTATION #####
4
5
6
7 class Graph:
8
9     # inialize the class adjacency list with user defined adjacency list.
10    def __init__(self, adjacency_list):
11        self.adjacency_list = adjacency_list
12
13    # to retrieve the neighbors of a given node
14    def get_neighbors(self, _node):
15        return self.adjacency_list[_node]
16
17    # heuristic function with heuristics for respective nodes
18    def h(self, n):
19        # dictionary of the Nodes and respective heuristics
20        H = { 'Z': 5.7,
21              'D': 5,
22              'H': 4.5,
23              'G': 4.2,
24              'C': 4.1,
25              'F': 4,
26              'E': 2.8,
27              'B': 2.0,
28              'A': 0 }
29        return H[n] #retrieve the heuristic value of given node
30
31    def aStarGrphic(self, start_node, stop_node):
32
33        # is visited, but who's neighbors haven't all been inspected,
34        # starts off with the start node
35        open_list = set([start_node])
36
37        # has been visited and who's neighbors have been inspected
38        closed_list = set([])

```

```
39
40     g = {} # g contains current distances from start_node to all other nodes
41
42     g[start_node] = 0 # start node doesnt have parent node distance
43
44     # parents contains an adjacency map of all nodes
45     parents = {}
46     parents[start_node] = start_node
47
48     while len(open_list) > 0:
49         n = None
50
51         # find a node with the lowest value of f() - evaluation function
52         for v in open_list:
53             if n == None or g[v] + self.h(v) < g[n] + self.h(n):
54                 n = v;
55
56         if n == None:
57             print('Path does not exist!')
58             return None
59
60         # if the current node is the stop_node
61         # then we begin reconstructin the path from it to the start_node
62         if n == stop_node:
63             reconst_path = []
64
65             while parents[n] != n:
66                 reconst_path.append(n)
67                 n = parents[n]
68
69             reconst_path.append(start_node)
70             reconst_path.reverse()
71
72             print('Path found: {}'.format(reconst_path))
73             return reconst_path
74
75         # for all neighbors of the current node do
76         for (m, weight) in self.get_neighbors(n):
77             # if the current node isn't in both open_list and closed_list
78             # add it to open_list and note n as it's parent
79             if m not in open_list and m not in closed_list:
80                 open_list.add(m)
81                 parents[m] = n
82                 g[m] = g[n] + weight
83
84             # otherwise, check if it's quicker to first visit n, then m
85             # and if it is, update parent data and g data
86             # and if the node was in the closed_list, move it to open_list
87             else:
88                 if g[m] > g[n] + weight:
89                     g[m] = g[n] + weight
90                     parents[m] = n
```

```

91
92         if m in closed_list:
93             closed_list.remove(m)
94             open_list.add(m)
95
96         # remove n from the open_list, and add it to closed_list
97         # because all of his neighbors were inspected
98         open_list.remove(n)
99         closed_list.add(n)
100
101     print('Path does not exist!')
102     return None
103
104
105
106
107 # heuristic function with respective value of the alphabit
108 def hDif(self, n):
109     # dictionary of the Nodes and respective heuristics
110     H = { 'Z': 26,
111           'D': 4,
112           'H': 8,
113           'G': 7,
114           'C': 3,
115           'F': 6,
116           'E': 5,
117           'B': 2,
118           'A': 1 }
119
120     return H[n] - H['A'] #retreive the heuristic value of given node
121
122 def aStarAlphbiticValue(self, start_node, stop_node):
123
124     # is visited, but who's neighbors haven't all been inspected,
125     # starts off with the start node
126     open_list = set([start_node])
127
128     # has been visited and who's neighbors have been inspected
129     closed_list = set([])
130
131     g = {} # g contains current distances from start_node to all other nodes
132
133     g[start_node] = 0 # start node doesnt have parent node distance
134
135     # parents contains an adjacency map of all nodes
136     parents = {}
137     parents[start_node] = start_node
138
139     while len(open_list) > 0:
140         n = None
141
142         # find a node with the lowest value of f() - evaluation function

```

```

142 # find a node with the lowest value of  $\tau()$  - evaluation function
143 for v in open_list:
144     if n == None or g[v] + self.hDif(v) < g[n] + self.hDif(n):
145         n = v;
146
147 if n == None:
148     print('Path does not exist!')
149     return None
150
151 # if the current node is the stop_node
152 # then we begin reconstructin the path from it to the start_node
153 if n == stop_node:
154     reconst_path = []
155
156     while parents[n] != n:
157         reconst_path.append(n)
158         n = parents[n]
159
160     reconst_path.append(start_node)
161     reconst_path.reverse()
162
163     print('Path found: {}'.format(reconst_path))
164     return reconst_path
165
166 # for all neighbors of the current node do
167 for (m, weight) in self.get_neighbors(n):
168     # if the current node isn't in both open_list and closed_list
169     # add it to open_list and note n as it's parent
170     if m not in open_list and m not in closed_list:
171         open_list.add(m)
172         parents[m] = n
173         g[m] = g[n] + weight
174
175     # otherwise, check if it's quicker to first visit n, then m
176     # and if it is, update parent data and g data
177     # and if the node was in the closed_list, move it to open_list
178     else:
179         if g[m] > g[n] + weight:
180             g[m] = g[n] + weight
181             parents[m] = n
182
183         if m in closed_list:
184             closed_list.remove(m)
185             open_list.add(m)
186
187 # remove n from the open_list, and add it to closed_list
188 # because all of his neighbors were inspected
189 open_list.remove(n)
190 closed_list.add(n)
191
192 print('Path does not exist!')
193 return None

```



```

194
195
196 adjacency_list = {
197     'Z': [('H', 2), ('D', 1)],
198     'H': [('E', 2), ('F', 2), ('G', 1.4)],
199     'D': [('C', 2.2)],
200     'C': [('G', 2), ('B', 2.2)],
201     'G': [('D', 1), ('E', 1.4)],
202     'F': [('A', 4)],
203     'B': [('E', 2.2)],
204     'E': [('A', 2.8)]
205 }
206
207 graph1 = Graph(adjacency_list)
208
209 print("This is the implementation of the graphic heuristics value A*:\n")
210
211 for node in graph1.aStarGrphic('Z', 'A'):
212     print("-> {}".format(node), end=" ");
213
214
215 print("\n\nThis is the implementation of the alphabetic value A star:\n")
216
217 for node in graph1.aStarAlphabeticValue('Z', 'A'):
218     print("-> {}".format(node), end=" ");
219

```

☞ This is the implementation of the graphic heuristics value A*:

Path found: ['Z', 'H', 'E', 'A']
-> Z -> H -> E -> A

This is the implementation of the alphabetic value A star:

Path found: ['Z', 'H', 'E', 'A']
-> Z -> H -> E -> A

```

1 ##### PRACTICE ASSIGNMENT #####
2 import networkx as nx #####
3 import matplotlib.pyplot as plt #####
4 #####
5
6 G = nx.Graph()
7
8 G.add_node('A', pos=(1.5, 6))
9 G.add_node('B', pos=(.5, 5))
10 G.add_node('F', pos=(2.5, 5))
11 G.add_node('C', pos=(0, 3))
12 G.add_node('D', pos=(1, 3))
13 G.add_node('G', pos=(2, 3))
14 G.add_node('H', pos=(3, 3))
15 G.add_node('E', pos=(.5, 1))

```



```

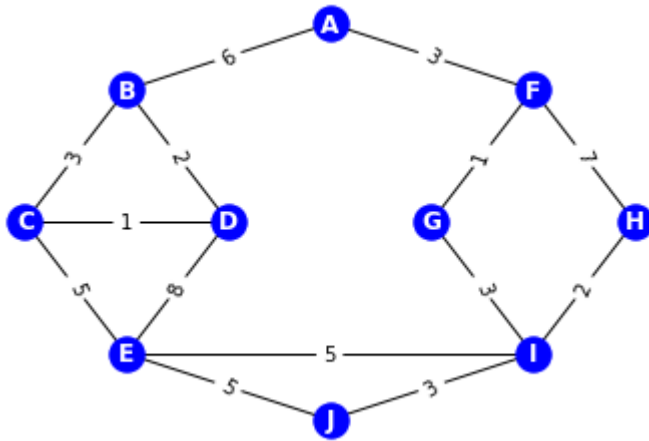
16 G.add_node('I',pos=(2.5,1))
17 G.add_node('J',pos=(1.5,0))
18
19 G.add_edge('F', 'A', weight=3)
20 G.add_edge('A', 'B', weight=6)
21 G.add_edge('B', 'D', weight=2)
22 G.add_edge('B', 'C', weight=3)
23 G.add_edge('D', 'C', weight=1)
24 G.add_edge('C', 'E', weight=5)
25 G.add_edge('D', 'E', weight=8)
26 G.add_edge('E', 'J', weight=5)
27 G.add_edge('F', 'G', weight=1)
28 G.add_edge('F', 'H', weight=7)
29 G.add_edge('G', 'I', weight=3)
30 G.add_edge('H', 'I', weight=2)
31 G.add_edge('I', 'E', weight=5)
32 G.add_edge('I', 'J', weight=3)
33
34 pos=nx.get_node_attributes(G,'pos')
35 labels = nx.get_edge_attributes(G,'weight')
36 nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
37
38 nx.draw(G, pos, with_labels=True, font_weight='bold',edge_color='black',
39         font_color='white', node_color='blue')
40 plt.savefig('simpleGraph.svg')
41 plt.show()
42
43
44
45 #####AI Lab query 3 - a:#####
46 def get_connected_nodes(n):
47     return list(G[n])
48
49 print("#####AI Lab query 3 - a:#####")
50 n_node= input("Adjacent nodes of:\n").upper()
51 if n_node in G:
52     print(get_connected_nodes(n_node))
53 else:
54     print(n_node," does not exist!")
55
56 #####AI Lab query 3 - b:#####
57 def get_edge(u, v):
58     return G.get_edge_data(u, v)
59
60 print("#####AI Lab query 3 - b:#####")
61
62 print("Get the edge of two nodes:")
63 u = input("Node 1: ").upper()
64 v = input("Node 2: ").upper()
65 print(get_edge(u, v))
66
67 #####AI Lab query 3 - c:#####

```



```
0/ #####AI Lab query 3 - c.#####
68 print("#####AI Lab query 3 - b:#####")
69 print("Shortest path:")
70 _start = input("Node 1: ").upper()
71 _goal = input("Node 2: ").upper()
72
73 #nx.shortest_path(G, source=None, target=None, weight=None, method='dijkstra')
74 print("Shortest path using Dijkstra:")
75 print(nx.shortest_path(G, _start, _goal, weight=None, method='dijkstra'))
76
77 print("Shortest path using A*:")
78 print(nx.astar_path(G, _start, _goal, heuristic=None, weight='weight'))
79
80 #####AI Lab query 4:#####
81 def are_connected(node_1, node_2):
82     return node_1 in G[node_2]
83
84 print("#####AI Lab query 4:#####")
85 print("Check the connectivity of two nodes:")
86 node_1 = input("Node 1: ").upper()
87 node_2 = input("Node 2: ").upper()
88 print(are_connected(node_1, node_2))
89
90 #####AI Lab query 5:#####
91 def is_valid_path(path, s, g):
92     return path in nx.all_simple_paths(G, s, g)
93
94 print("#####AI Lab query 5:#####")
95 print("Check the path is valid or not:")
96 xPath = ['A', 'F', 'G', 'I', 'J']
97 print("['A', 'F', 'G', 'I', 'J']")
98 if is_valid_path(xPath, "A", "J"):
99     print("path is valid")
100 else:
101     print("path is not valid")
```





#####AI Lab query 3 - a:#####

Adjacent nodes of:

A

['F', 'B']

#####AI Lab query 3 - b:#####

Get the edge of two nodes:

Node 1: B

Node 2: C

{'weight': 3}

#####AI Lab query 3 - b:#####

Shortest path:

Node 1: A

Node 2: J

Shortest path using Dijkstra:

['A', 'B', 'C', 'E', 'J']

Shortest path using A*:

['A', 'F', 'G', 'I', 'J']

#####AI Lab query 4:#####

Check the connectivity of two nodes:

Node 1: C

Node 2: D

True

#####AI Lab query 5:#####

Check the path is valid or not:

['A', 'F', 'G', 'I', 'J']

path is valid