by: ~~Kh~~
Kamran Khan

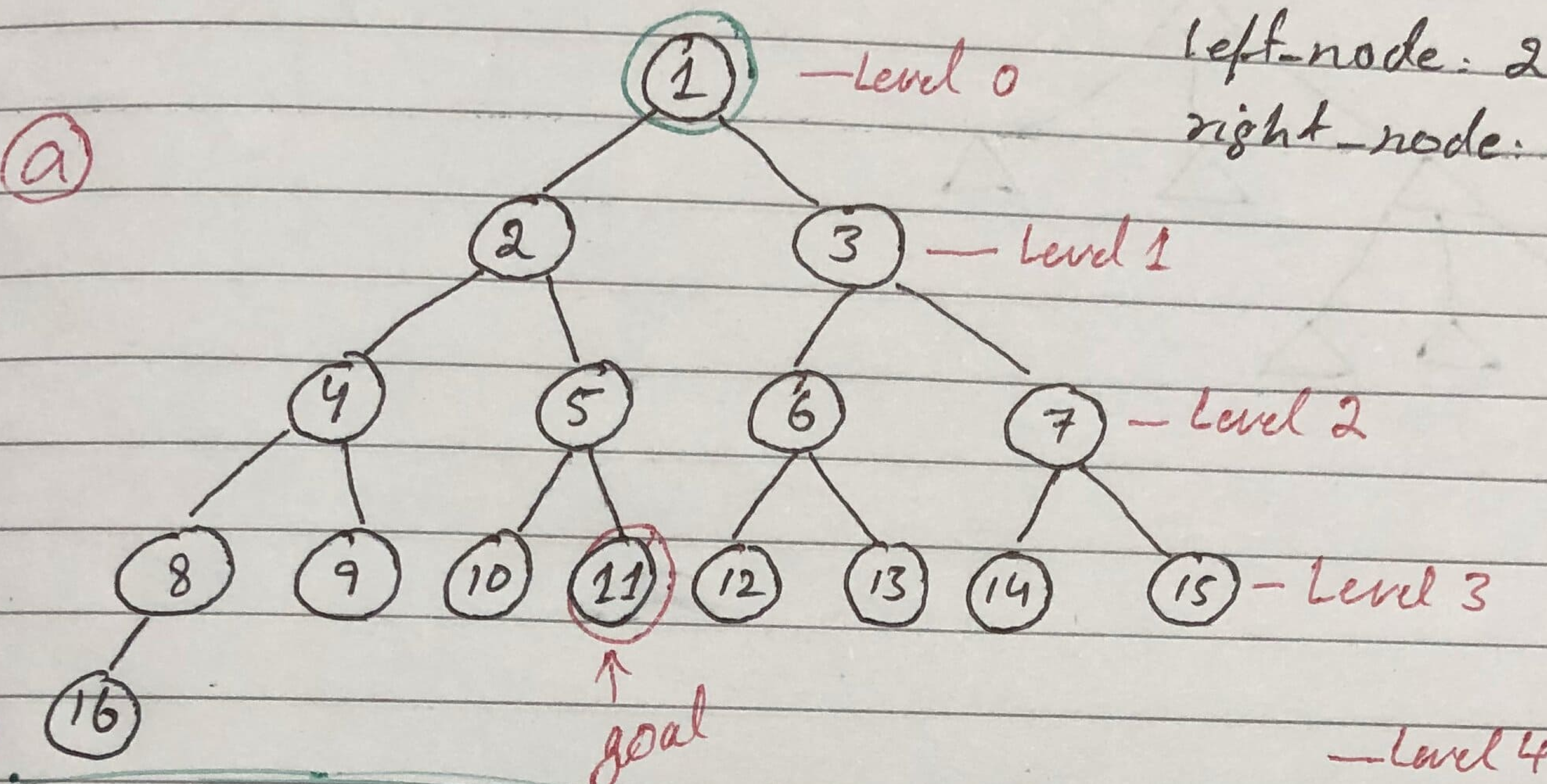
Graded Assignment 1

Q1:-

Start:- node 1
goal: node 11

K = 1 … 7
left_node: 2k
right_node: 2k+1

(a)



(b)

→ s: 1
→ g: 11
~~Limit = 3~~

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

Q

~~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

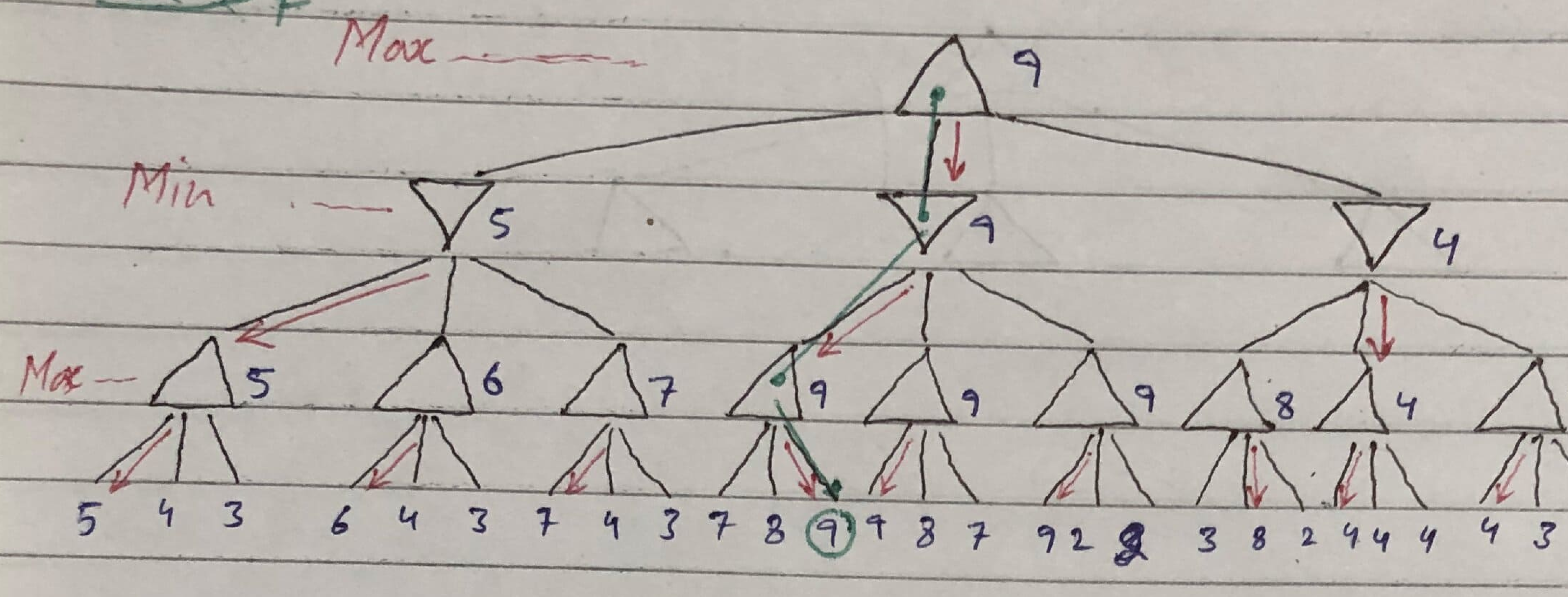
~~8~~ ~~9~~ ~~10~~ 11
~~4~~ 5
2
1

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]

Q 2:-

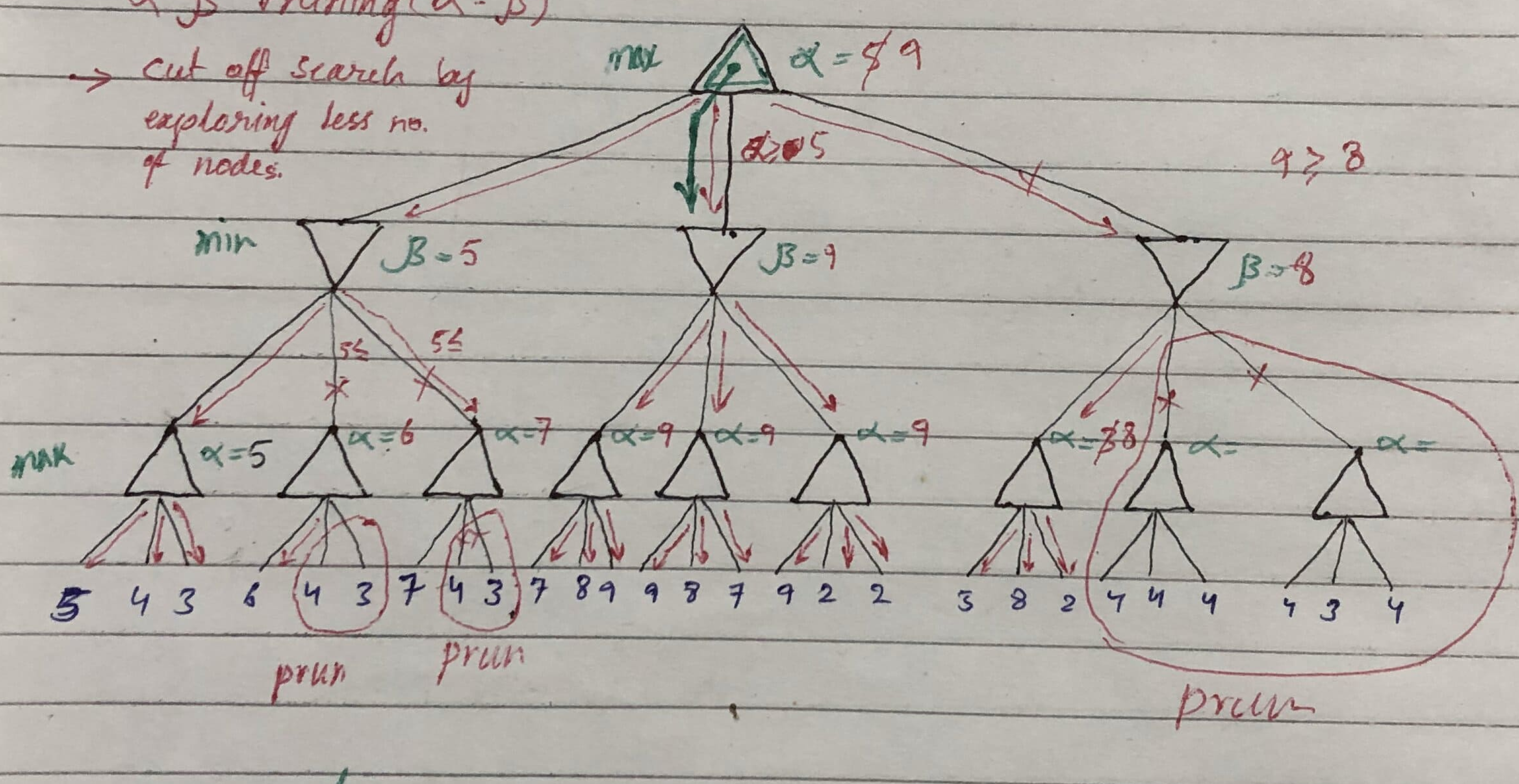
a) :-



b) :-

α β Pruning (α - β)

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



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

for min β (default value +∞)
for max α (default value −∞)

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 and B's domain has another N values, Z's domain is the cartisian product of of A domain and B domain N×N values

Z's ~~the~~ 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 2d values of pair Z.

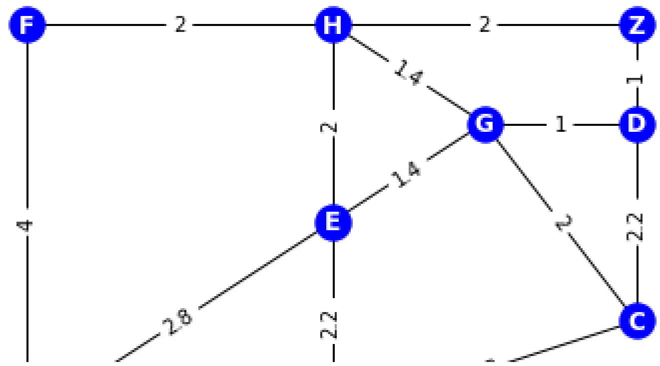
3) And the sum of the both A and B values pair in Z equals to C.

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

For example, if ~~we~~ 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.

```python
######################### GRADED ASSIGNMENT - 01 ##############################
############################  VISUALIZATION ###################################

import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()

G.add_node("A", pos=(0, 0))
G.add_node("B", pos=(1, 0))
G.add_node("C", pos=(2, .5))
G.add_node("D", pos=(2, 1.5))
G.add_node("E", pos=(1, 1))
G.add_node("F", pos=(0, 2))
G.add_node("G", pos=(1.5, 1.5))
G.add_node("H", pos=(1, 2))
G.add_node("Z", pos=(2, 2))

G.add_edge("Z", "D", weight=1)
G.add_edge("Z", "H", weight=2)
G.add_edge("H", "E", weight=2)
G.add_edge("H", "F", weight=2)
G.add_edge("H", "G", weight=1.4)
G.add_edge("G", "D", weight=1)
G.add_edge("G", "E", weight=1.4)
G.add_edge("D", "C", weight=2.2)
G.add_edge("C", "G", weight=2)
G.add_edge("C", "B", weight=2.2)
G.add_edge("B", "E", weight=2.2)
G.add_edge("G", "E", weight=1.4)
G.add_edge("E", "A", weight=2.8)
G.add_edge("F", "A", weight=4)

print("h(Z): 5.7\th(H): 4.5\th(D): 5")
print("h(F): 4\t\th(G): 4.2\th(E): 2.8\nh(C): 4.1\th(B): 2.0\th(A):  0")

pos=nx.get_node_attributes(G,'pos')
labels = nx.get_edge_attributes(G,'weight')

nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

nx.draw(G, pos, with_labels=True, font_weight='bold',
        edge_color='black', font_color='white', node_color='blue')
plt.savefig('simpleGraph.svg')
plt.show()

```

```
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
```




```
######################### GRADED ASSIGNMENT - 01 ##############################
from collections import deque                      ##############################
##########################  IMPLEMENTATION ####################################



class Graph:

  # inialize the class adjacency list with user defined adjacency list.
  def __init__(self, adjacency_list):
    self.adjacency_list = adjacency_list

  # to retreive the neighbors of a given node
  def get_neighbors(self, _node):
    return self.adjacency_list[_node]

  # heuristic function with heuristics for respective nodes
  def h(self, n):
    # dictionary of the Nodes and respective heuristics
    H = { 'Z': 5.7,
         'D': 5,
         'H': 4.5,
         'G': 4.2,
         'C': 4.1,
         'F': 4,
         'E': 2.8,
         'B': 2.0,
         'A':  0 }
    return H[n] #retreive the heuristic value of given node

  def aStarGrphic(self, start_node, stop_node):

        # is visited, but who's neighbors haven't all been inspected,
        # starts off with the start node
        open_list = set([start_node])

        # has been visited and who's neighbors have been inspected
        closed_list = set([])
```

```
    g = {} # g contains current distances from start_node to all other nodes

    g[start_node] = 0   # start node doesnt have parent node distance

    # parents contains an adjacency map of all nodes
    parents = {}
    parents[start_node] = start_node

    while len(open_list) > 0:
        n = None

        # find a node with the lowest value of f() - evaluation function
        for v in open_list:
          if n == None or g[v] + self.h(v) < g[n] + self.h(n):
            n = v;

        if n == None:
          print('Path does not exist!')
          return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
          reconst_path = []

          while parents[n] != n:
            reconst_path.append(n)
            n = parents[n]

          reconst_path.append(start_node)
          reconst_path.reverse()

          print('Path found: {}'.format(reconst_path))
          return reconst_path

        # for all neighbors of the current node do
        for (m, weight) in self.get_neighbors(n):
            # if the current node isn't in both open_list and closed_list
            # add it to open_list and note n as it's parent
          if m not in open_list and m not in closed_list:
            open_list.add(m)
            parents[m] = n
            g[m] = g[n] + weight

            # otherwise, check if it's quicker to first visit n, then m
            # and if it is, update parent data and g data
            # and if the node was in the closed_list, move it to open_list
          else:
            if g[m] > g[n] + weight:
              g[m] = g[n] + weight
              parents[m] = n
```

```

              if m in closed_list:
                closed_list.remove(m)
                open_list.add(m)

          # remove n from the open_list, and add it to closed_list
          # because all of his neighbors were inspected
          open_list.remove(n)
          closed_list.add(n)

      print('Path does not exist!')
      return None




# heuristic function with respective value of the alphabit
def hDif(self, n):
  # dictionary of the Nodes and respective heuristics
  H = { 'Z': 26,
       'D': 4,
       'H': 8,
       'G': 7,
       'C': 3,
       'F': 6,
       'E': 5,
       'B': 2,
       'A':  1 }

  return H[n] - H['A'] #retreive the heuristic value of given node

def aStarAlphbiticValue(self, start_node, stop_node):

      # is visited, but who's neighbors haven't all been inspected,
      # starts off with the start node
      open_list = set([start_node])

      # has been visited and who's neighbors have been inspected
      closed_list = set([])

      g = {} # g contains current distances from start_node to all other nodes

      g[start_node] = 0   # start node doesnt have parent node distance

      # parents contains an adjacency map of all nodes
      parents = {}
      parents[start_node] = start_node

      while len(open_list) > 0:
          n = None

          # find a node with the lowest value of f() - evaluation function
```

```
        # find a node with the lowest value of f() - evaluation function
        for v in open_list:
          if n == None or g[v] + self.hDif(v) < g[n] + self.hDif(n):
            n = v;

        if n == None:
          print('Path does not exist!')
          return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
          reconst_path = []

          while parents[n] != n:
            reconst_path.append(n)
            n = parents[n]

          reconst_path.append(start_node)
          reconst_path.reverse()

          print('Path found: {}'.format(reconst_path))
          return reconst_path

        # for all neighbors of the current node do
        for (m, weight) in self.get_neighbors(n):
            # if the current node isn't in both open_list and closed_list
            # add it to open_list and note n as it's parent
          if m not in open_list and m not in closed_list:
            open_list.add(m)
            parents[m] = n
            g[m] = g[n] + weight

            # otherwise, check if it's quicker to first visit n, then m
            # and if it is, update parent data and g data
            # and if the node was in the closed_list, move it to open_list
          else:
            if g[m] > g[n] + weight:
              g[m] = g[n] + weight
              parents[m] = n

            if m in closed_list:
              closed_list.remove(m)
              open_list.add(m)

        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None
```

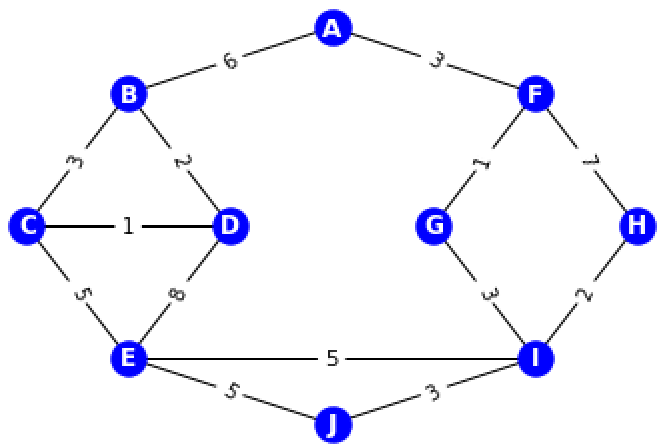
```
adjacency_list = {
    'Z': [('H', 2), ('D', 1)],
    'H': [('E', 2), ("F", 2), ("G", 1.4)],
    'D': [('C', 2.2)],
    'C': [("G", 2), ("B", 2.2)],
    'G': [("D", 1), ("E", 1.4)],
    'F': [("A", 4)],
    'B': [("E", 2.2)],
    'E': [("A", 2.8)]
}

graph1 = Graph(adjacency_list)

print("This is the implementation of the graphic heuristics value A*:\n")

for node in graph1.aStarGrphic('Z', 'A'):
  print("-> {}".format(node), end=" ");


print("\n\nThis is the implementation of the alphabitic value A star:\n")

for node in graph1.aStarAlphbiticValue('Z', 'A'):
  print("-> {}".format(node), end=" ");

```

```
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 alphabitic value A star:

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

```
######################### PRACTICE ASSIGNMENT #################################
import networkx as nx                         #################################
import matplotlib.pyplot as plt               #################################
###############################################################################

G = nx.Graph()

G.add_node('A',pos=(1.5,6))
G.add_node('B',pos=(.5,5))
G.add_node('F',pos=(2.5,5))
G.add_node('C',pos=(0,3))
G.add_node('D',pos=(1,3))
G.add_node('G',pos=(2,3))
G.add_node('H',pos=(3,3))
G.add_node('E',pos=(.5,1))
```

```
G.add_node('I',pos=(2.5,1))
G.add_node('J',pos=(1.5,0))

G.add_edge('F', 'A', weight=3)
G.add_edge('A', 'B', weight=6)
G.add_edge('B', 'D', weight=2)
G.add_edge('B', 'C', weight=3)
G.add_edge('D', 'C', weight=1)
G.add_edge('C', 'E', weight=5)
G.add_edge('D', 'E', weight=8)
G.add_edge('E', 'J', weight=5)
G.add_edge('F', 'G', weight=1)
G.add_edge('F', 'H', weight=7)
G.add_edge('G', 'I', weight=3)
G.add_edge('H', 'I', weight=2)
G.add_edge('I', 'E', weight=5)
G.add_edge('I', 'J', weight=3)

pos=nx.get_node_attributes(G,'pos')
labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

nx.draw(G, pos, with_labels=True, font_weight='bold',edge_color='black',
        font_color='white', node_color='blue')
plt.savefig('simpleGraph.svg')
plt.show()



###########AI Lab query 3 - a:################
def get_connected_nodes(n):
  return list(G[n])

print("###########AI Lab query 3 - a:################")
n_node= input("Adjacent nodes of:\n").upper()
if n_node in G:
  print(get_connected_nodes(n_node))
else:
  print(n_node," does not exist!")

###########AI Lab query 3 - b:################
def get_edge(u, v):
  return G.get_edge_data(u, v)

print("###########AI Lab query 3 - b:################")

print("Get the edge of two nodes:")
u = input("Node 1: ").upper()
v = input("Node 2: ").upper()
print(get_edge(u, v))

###########AI Lab query 3 - c:################
```

```
###########AI Lab query 3 - c:################
print("###########AI Lab query 3 - b:################")
print("Shortest path:")
_start = input("Node 1: ").upper()
_goal = input("Node 2: ").upper()

#nx.shortest_path(G, source=None, target=None, weight=None, method='dijkstra')
print("Shortest path using Dijkstra:")
print(nx.shortest_path(G, _start, _goal, weight=None, method='dijkstra'))

print("Shortest path using A*:")
print(nx.astar_path(G, _start, _goal, heuristic=None, weight='weight'))

###########AI Lab query 4:################
def are_connected(node_1, node_2):
  return node_1 in G[node_2]

print("###########AI Lab query 4:################")
print("Check the connectivity of two nodes:")
node_1 = input("Node 1: ").upper()
node_2 = input("Node 2: ").upper()
print(are_connected(node_1, node_2))

###########AI Lab query 5:################
def is_valid_path(path, s, g):
  return path in nx.all_simple_paths(G, s, g)

print("###########AI Lab query 5:################")
print("Check the path is valid or not:")
xPath = ['A', 'F', 'G', 'I', 'J']
print("['A', 'F', 'G', 'I', 'J']")
if is_valid_path(xPath, "A", "J"):
  print("path is valid")
else:
  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
```