IMPORTS

In [1]:

```
from collections import defaultdict,deque
import pandas as pd
from locationiq.geocoder import LocationIQ
from math import radians,sin,cos,acos
import json
from time import sleep
```

Creating Graph Class

In [2]:

```
class Graph:

    def __init__(self):
        self.graph=defaultdict(list)
    def addedge(self,u,v,w):          #Adds undirected edges to the graph
        self.graph[u].append((v,w))
        self.graph[v].append((u,w))


    def printGraph(self):
        print(self.graph)
    def neighbours(self,u):           #returns neighbours of a node
        neighbours = []
        for node in self.graph[u]:
                neighbours.append(node[0])
        return neighbours


    def pathcost(self,u,v):
        for node in self.graph:
            if node == u:
                for i in self.graph[node]:
                    if(i[0]==v):
                        return i[1]


g=Graph()
```

Reading the file and adding edges to the graph

In [3]:

```
df= pd.read_excel(r'C:\Users\DELL\Desktop\ug2\AI\Indian_capitals.xlsx',header=None)
```

In [4]:

```
for i,j,k in zip(df[0],df[1],df[2]):
    g.addedge(i,j,k)
```

Testing the graph functions

In [5]:

```
g.printGraph()
print(g.neighbours('Aizwal'))
g.pathcost('Bangalore','Chennai')
```

```
defaultdict(<class 'list'>, {'Agartala': [('Aizwal', 342), ('Dispur', 53
6)], 'Aizwal': [('Agartala', 342), ('Imphal', 400), ('Dispur', 462)], 'Imp
hal': [('Aizwal', 400), ('Dispur', 482), ('Kohima', 136)], 'Amaravathi':
[('Bangalore', 663), ('Chennai', 448), ('Bhubaneswar', 819), ('Raipur', 75
8)], 'Bangalore': [('Amaravathi', 663), ('Panaji', 578), ('Chennai', 333),
('Thiruvanathapuram', 730), ('Mumbai', 980), ('Hyderabad', 569)], 'Chenna
i': [('Amaravathi', 448), ('Bangalore', 333), ('Thiruvanathapuram', 771)],
'Bhubaneswar': [('Amaravathi', 819), ('Raipur', 544), ('Ranchi', 455), ('K
olkata', 441)], 'Raipur': [('Amaravathi', 758), ('Bhubaneswar', 544), ('Hy
derabad', 783), ('Mumbai', 1091), ('Bhopal', 614), ('Lucknow', 810), ('Ran
chi', 580)], 'Panaji': [('Bangalore', 578), ('Mumbai', 542)], 'Thiruvanath
apuram': [('Bangalore', 730), ('Chennai', 771)], 'Mumbai': [('Bangalore',
980), ('Hyderabad', 719), ('Panaji', 542), ('Gandhinagar', 553), ('Bhopa
l', 776), ('Raipur', 1091)], 'Bhopal': [('Gandhinagar', 599), ('Jaipur', 5
98), ('Lucknow', 615), ('Mumbai', 776), ('Raipur', 614)], 'Gandhinagar':
[('Bhopal', 599), ('Jaipur', 634), ('Mumbai', 553)], 'Ranchi': [('Bhubanes
war', 455), ('Kolkata', 395), ('Lucknow', 710), ('Patna', 327), ('Raipur',
580)], 'Kolkata': [('Bhubaneswar', 441), ('Ranchi', 395), ('Patna', 583),
('Gangtok', 675), ('Dispur', 1035)], 'Chandigarh': [('Lucknow', 742), ('Ja
ipur', 528), ('Shimla ', 113), ('Srinagar', 562)], 'Lucknow': [('Chandigar
h', 742), ('Dehradun', 552), ('Jaipur', 574), ('Bhopal', 615), ('Ranchi',
710), ('Patna', 539), ('Raipur', 810), ('Shimla ', 841)], 'Jaipur': [('Cha
ndigarh', 528), ('Gandhinagar', 634), ('Bhopal', 598), ('Lucknow', 574)],
'Dehradun': [('Lucknow', 552), ('Shimla ', 227)], 'Dispur': [('Shillong',
91), ('Imphal', 482), ('Aizawl', 462), ('Agartala', 536), ('Itanagar', 32
3), ('Kohima', 350), ('Kolkata', 1035)], 'Shillong': [('Dispur', 91)], 'It
anagar': [('Dispur', 323), ('Kohima', 323)], 'Kohima': [('Dispur', 350),
('Imphal', 136), ('Itanagar', 323)], 'Hyderabad': [('Amaravati', 271), ('B
angalore', 569), ('Raipur', 783), ('Mumbai', 719)], 'Amaravati': [('Hydera
bad', 271)], 'Patna': [('Kolkata', 583), ('Lucknow', 539), ('Ranchi', 32
7)], 'Gangtok': [('Kolkata', 675)], 'Shimla ': [('Chandigarh', 113), ('Deh
radun', 227), ('Lucknow', 841)], 'Srinagar': [('Shimla', 620), ('Chandigar
h', 562)], 'Shimla': [('Srinagar', 620)]})
[]
```

Out[5]:

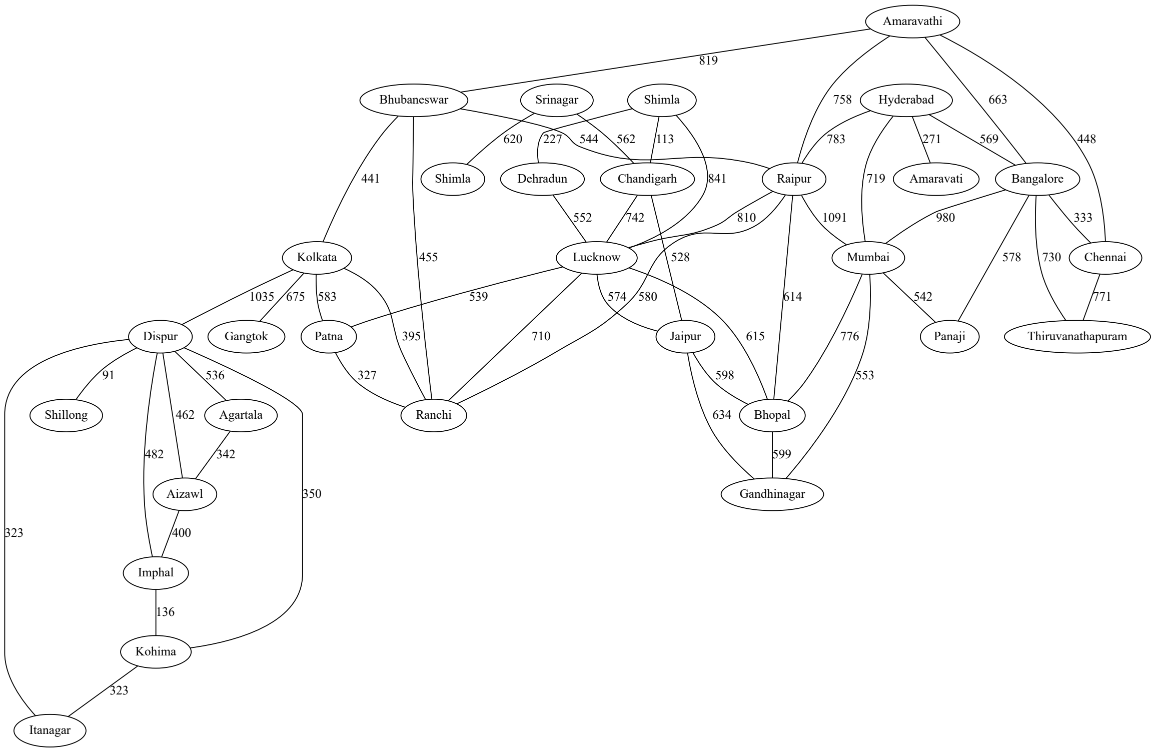
```
333
```

Graph for the given input file

In [6]:

```
import graphviz
from graphviz import Graph
gr = Graph('G')
for i,j,k in zip(df[0],df[1],df[2]):
    gr.edge(i,j,label=str(k))
gr

#Below plotted is the input graph
```

Out[6]:



Goal Test function

In [7]:

```
def goal_test(node): #returns True/False whether the argument is goal or not
    global goal
    if node == goal:
        return True
    else:
        return False
```

Defining problem class

In [8]:

```
class Problem:
    def __init__(self,initial):
        self.initial_state=initial

    def actions(self,source):      #returns the neighbours of a node
        return g.neighbours(source)
    def stepcost(self,src,dest):   #return pathcost between two nodes
        return g.pathcost(src,dest)
    def result(self,state,action): #returns the action
        if not state and not action:
            return self.initial_state
        return action
```

Defining child_node class

In [9]:

```
class Child_Node:
    def __init__(self,state,parent, action):

        self.parent = parent
        self.action = action
        if parent is None:
            self.path_cost =0
            self.state = problem.initial_state
        else:
            self.path_cost = parent.path_cost + problem.stepcost(parent.state,action)
            self.state = problem.result(parent.state,action)
```

Functions for Agent

In [10]:

```
def updatestate(state,percept): #returns initial state
    return percept[0]
def formulate_problem(state,goal,flag): #Creates and returns Problem instance for state
    if flag == 1:                 #flag 1 is for path-finding
        return Problem(state)
    elif flag == 2:               #flag 2 is for 8-puzzle
        return Problem_8puzzle(state)

def formulate_goal(state): #returns the goal
    return state
```

Main Agent Function

In [11]:

```
state = None
goal = None
problem = None

def SimpleProblemSolvingAgent(percept,search,flag): #returns the pathe from initial sta
te to goal state
    global state, goal, problem
    seq=[]
    state = updatestate(state,percept)
    if len(seq)==0:
        goal = formulate_goal(percept[1])
        problem = formulate_problem(state,goal,flag)
        seq = search(problem,flag)
        if seq == None:
            return None

    return seq[::-1]
```

Breadth first Search

In [12]:

```
def solution(node,flag): #Does backtracking of node ...returns path it took to reach th
ere
    path = []
    if flag == 1:
        while node.parent !=None:
            path.append(node.state)
            node = node.parent
        path.append(node.state)
        return path
    elif flag == 2:
        while node.parent !=None:
            path.append(node.action)
            node = node.parent
        path.append(node.action)
        return path



def check(frontierr,child): # Returns false if  child.state is in the frontier
    for n in frontierr:
        if n.state == child.state:
            return False
    return True

def breadth_first_search(problem,flag): # returns path required to the agent function

    node = Child_Node(problem.initial_state,None,None)

    if goal_test(node.state):
        return solution(node,flag)

    frontier = deque([])
    frontier.append(node)

    explored = []
    while True:
        if len(frontier)==0:
            return print('No such goal exists')
        node=frontier.popleft()
        explored.append(node.state)

        for action in problem.actions(node.state):

            child = Child_Node(problem,node,action)

            if child.state not in explored:
                if check(frontier,child):
                    if goal_test(child.state):
                        print(f'The path cost is {child.path_cost} (Path is printed in
 below cell)')

                        return solution(child,flag)
                    frontier.append(child)
```

Depth first Search

In [13]:

```
def depth_first_search(problem,flag): # returns path required to the agent function
    node = Child_Node(problem.initial_state,None,None)
    if goal_test(node.state):
        return solution(node,flag)

    frontier = deque([])
    frontier.append(node)
    explored = []

    while True:
        if len(frontier)==0:
            return print('No such goal exists')
        node=frontier.pop()
        explored.append(node.state)

        for action in problem.actions(node.state):

            child = Child_Node(problem,node,action)

            if child.state not in explored:
                if check(frontier,child):
                    if goal_test(child.state):
                        print(f'The path cost is {child.path_cost} (Path is printed in
 below cell)')
                        return solution(child,flag)
                    frontier.append(child)
```

Bi-directional BFS

In [14]:

```
def bidirectionalBFS(problem,flag):  # returns path required to the agent function
    global goal
    node = Child_Node(problem.initial_state,None,None)
    if goal_test(node.state):
        return solution(node)
    node_end = Child_Node(goal,None,None)
    node_end.state = goal

    frontier_init = []
    frontier_end = []
    explored_init = []
    explored_end = []
    frontier_init.append(node)
    frontier_end.append(node_end)


    while True:


        node = frontier_init.pop(0)
        explored_init.append(node.state)
        node_end =frontier_end.pop(0)
        explored_end.append(node_end.state)

        for action in problem.actions(node.state):

            child = Child_Node(problem,node,action)

            if child.state not in explored_init:
                if check(frontier_init,child):
                    frontier_init.append(child)


        for action in problem.actions(node_end.state):

            child = Child_Node(problem,node_end,action)

            if child.state not in explored_end:
                if check(frontier_end,child):
                    frontier_end.append(child)



        for i in frontier_init:
            for j in frontier_end:
                if i.state==j.state:
                    print(j.path_cost + i.path_cost)
                    return solution(j,flag)[::-1]+solution(i,flag)[1::]
```

Function to find heuristic between two cities

In [15]:

```
def findHeuristic(city,dest):  #returns the heuristic for arguments
    sleep(1)
    geocoder = LocationIQ("df1cea1ac0972a")
    c1 = geocoder.geocode(city)[0]
    c2 = geocoder.geocode(dest)[0]

    slat = radians(float(c1['lat']))
    slon = radians(float(c1['lon']))

    elat = radians(float(c2['lat']))
    elon = radians(float(c2['lon']))

    dist = 6371.01 * acos(sin(slat)*sin(elat)+cos(slat)*cos(elat)*cos(slon - elon))
    return dist
```

A-star

In [16]:

```
states=[]
distances=[]

def getHeuristic(src,goal): # This function stores the heuristic
    if src in states:
        return distances[states.index(src)]
    else:
        dist = findHeuristic(src,goal)
        states.append(src)
        distances.append(dist)
        return dist
def Astar(problem,flag): # returns path required to the agent function
    global goal

    node = Child_Node(problem.initial_state,None,None)
    if goal_test(node.state):
        return solution(node)

    frontier = []
    frontier.append(node)
    explored = []

    while True:
        if len(frontier)==0:
            return print('No such goal exists')

        heuristic = [i.path_cost+getHeuristic(i.state,goal) for i in frontier]
        min_index = heuristic.index(min(heuristic))



        node=frontier[min_index]
        frontier.remove(node)
        explored.append(node.state)


        for action in problem.actions(node.state):

            child = Child_Node(problem,node,action)


            if child.state not in explored:
                if check(frontier,child):
                    if goal_test(child.state):
                        print(f'The path cost is {child.path_cost} (Path is printed in
 below cell)')
                        return solution(child,flag)
                frontier.append(child)
```

Inputs and Function mapping

Example Outputs

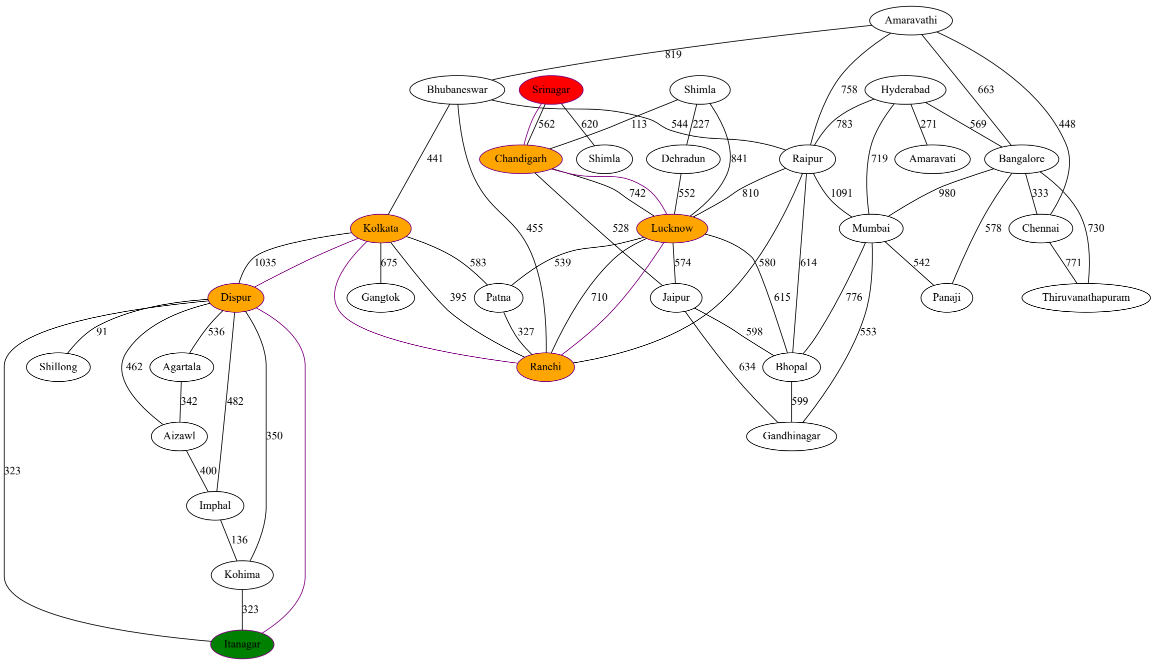
BFS

In [17]:

```
#Takes input for the required function  and maps it to search in agent function
func_map = {"bfs":breadth_first_search,"dfs":depth_first_search,"astar":Astar,"bdbfs":b
idirectionalBFS}
search = func_map[input(" breadth first search --bfs\n depth_first_search   --dfs\n Ast
ar                  --astar \n bidirectionalBFS     --bdbfs\nEnter search function name :
\n")]

percept = input("Enter source and destination : ").split(" ")

seq = SimpleProblemSolvingAgent(percept,search,1)

#Displays output

seq

for i in seq:
    if i==percept[0]:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'Red'})
    elif i==percept[1]:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'Green'})
    else:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'orange'})


for i in range(len(seq)-1):
    gr.edge(seq[i],seq[i+1],_attributes={'color':'Purple'})
gr

#The below output graph represents the path taken from source(Blue color) to destinatio
n(Green Color) through purple edges and orange nodes
#Black edges are original edges the extra purple edges are added to show the path witho
ut disturbing original edge
```

```
 breadth first search --bfs
 depth_first_search   --dfs
 Astar                --astar
 bidirectionalBFS     --bdbfs
Enter search function name :
bfs
Enter source and destination : Srinagar Itanagar
The path cost is 3767 (Path is printed in below cell)
```

Out[17]:



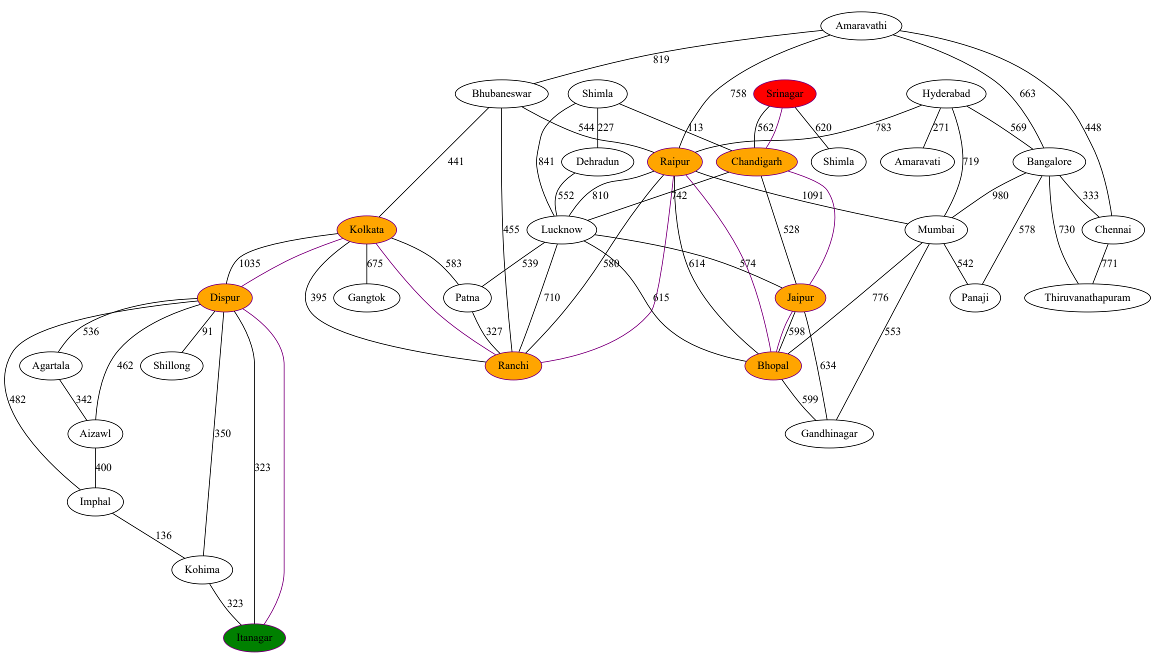
DFS

In [19]:

```
func_map = {"bfs":breadth_first_search,"dfs":depth_first_search,"astar":Astar,"bdbfs":b
idirectionalBFS}
search = func_map[input(" breadth first search --bfs\n depth_first_search   --dfs\n Ast
ar                 --astar \n bidirectionalBFS     --bdbfs\nEnter search function name :
\n")]

percept = input("Enter source and destination : ").split(" ")

seq = SimpleProblemSolvingAgent(percept,search,1)

#Displays output

seq

for i in seq:
    if i==percept[0]:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'Red'})
    elif i==percept[1]:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'Green'})
    else:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'orange'})


for i in range(len(seq)-1):
    gr.edge(seq[i],seq[i+1],_attributes={'color':'Purple'})
gr
```

```
 breadth first search --bfs
 depth_first_search   --dfs
 Astar                --astar
 bidirectionalBFS     --bdbfs
Enter search function name :
dfs
Enter source and destination : Srinagar Itanagar
The path cost is 4635 (Path is printed in below cell)
```

Out[19]:



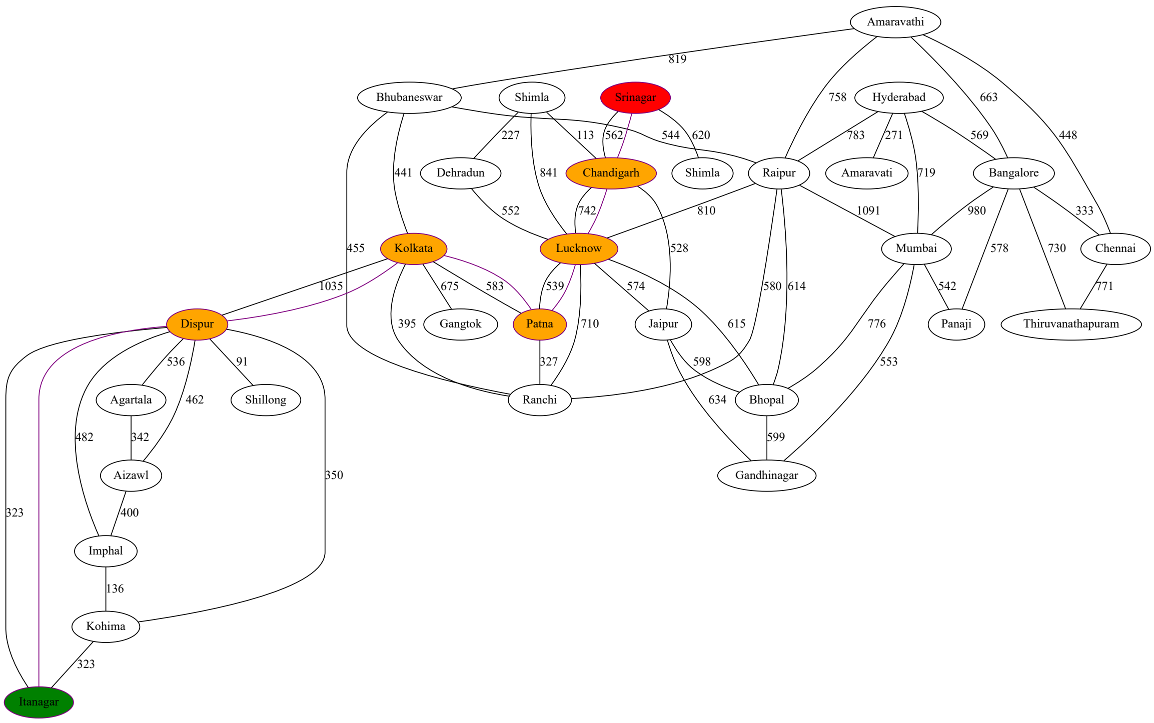
A-Star

In [17]:

```
func_map = {"bfs":breadth_first_search,"dfs":depth_first_search,"astar":Astar,"bdbfs":b
idirectionalBFS}
search = func_map[input(" breadth first search --bfs\n depth_first_search   --dfs\n Ast
ar                 --astar \n bidirectionalBFS     --bdbfs\nEnter search function name :
\n")]

percept = input("Enter source and destination : ").split(" ")

seq = SimpleProblemSolvingAgent(percept,search,1)

#Displays output

seq

for i in seq:
    if i==percept[0]:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'Red'})
    elif i==percept[1]:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'Green'})
    else:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'orange'})


for i in range(len(seq)-1):
    gr.edge(seq[i],seq[i+1],_attributes={'color':'Purple'})
gr
```

```
 breadth first search --bfs
 depth_first_search   --dfs
 Astar                --astar
 bidirectionalBFS     --bdbfs
Enter search function name :
astar
Enter source and destination : Srinagar Itanagar
The path cost is 3784 (Path is printed in below cell)
```

Out[17]:



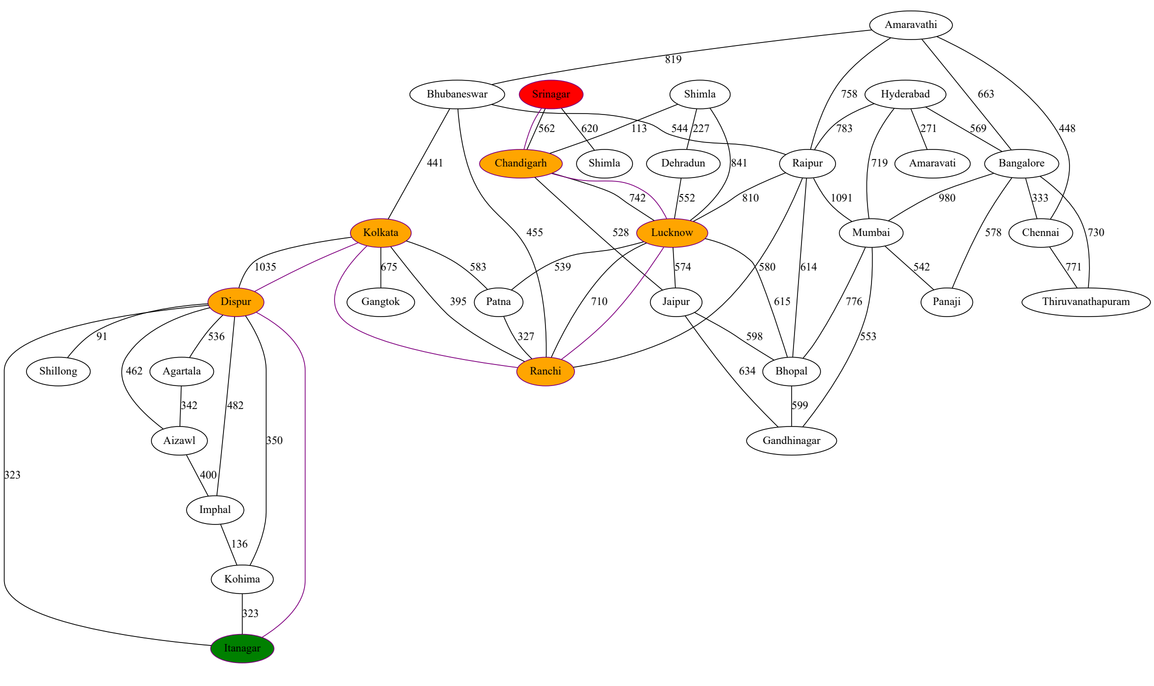
BiDirectional BFS

In [17]:

```
func_map = {"bfs":breadth_first_search,"dfs":depth_first_search,"astar":Astar,"bdbfs":b
idirectionalBFS}
search = func_map[input(" breadth first search --bfs\n depth_first_search   --dfs\n Ast
ar                  --astar \n bidirectionalBFS     --bdbfs\nEnter search function name :
\n")]

percept = input("Enter source and destination : ").split(" ")

seq = SimpleProblemSolvingAgent(percept,search,1)

#Displays output

seq

for i in seq:
    if i==percept[0]:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'Red'})
    elif i==percept[1]:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'Green'})
    else:
        gr.node(i,_attributes={'color':'Purple','style':'filled','fillcolor':'orange'})


for i in range(len(seq)-1):
    gr.edge(seq[i],seq[i+1],_attributes={'color':'Purple'})
gr
```

```
 breadth first search --bfs
 depth_first_search   --dfs
 Astar                --astar
 bidirectionalBFS     --bdbfs
Enter search function name :
bdbfs
Enter source and destination : Srinagar Itanagar
3767
```

Out[17]:



8-Puzzle Problem

In [18]:

```
class Problem_8puzzle: #problem class
    def __init__(self,initial):
        self.initial_state = initial
    def find_blank_square(self,state):  #returns index of blank square
        return state.index(0)
    def actions(self,state):   #returns set of possible actions
        possible_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
        index_blank_square = self.find_blank_square(state)
        if index_blank_square % 3 == 0:
            possible_actions.remove('LEFT')
        if index_blank_square < 3:
            possible_actions.remove('UP')
        if index_blank_square % 3 == 2:
            possible_actions.remove('RIGHT')
        if index_blank_square > 5:
            possible_actions.remove('DOWN')
        return possible_actions
    def result(self,state,action):  #returns the resulting state after the action is pe
rformed
        blank = self.find_blank_square(state)
        new_state = list(state)

        delta = {'UP': -3, 'DOWN': 3, 'LEFT': -1, 'RIGHT': 1}
        neighbor = blank + delta[action]
        new_state[blank], new_state[neighbor] = new_state[neighbor], new_state[blank]

        return list(new_state)
    def stepcost(self,src,dest):
        return 1
```

Greedy Best first Search

In [19]:

```
def Heuristic_8puzzle(state): #heuristic for 8-puzzle problem -- returns number of mism
atched tiles
    count = 0
    for src,dest in zip(state,goal):
        count=count+1
    return count

def greedy_bestfirstsearch(problem,flag):   #returns set of actions required to reach g
oal state
    global goal
    node = Child_Node(problem.initial_state,None,None)
    if goal_test(node.state):
        return solution(node)

    frontier = []
    frontier.append(node)
    explored = []

    while True:
        if len(frontier)==0:
            return print('No such goal exists')

        heuristic = [Heuristic_8puzzle(i.state) for i in frontier]
        min_index = heuristic.index(min(heuristic))


        node=frontier[min_index]
        frontier.remove(node)
        explored.append(node.state)


        for action in problem.actions(node.state):
            child = Child_Node(problem,node,action)
            if child.state not in explored:
                if check(frontier,child):
                    if goal_test(child.state):
                        return solution(child,flag)
                    frontier.append(child)
```

A-star for 8-puzzle problem

In [20]:

```
def Astar_8puzzle(problem,flag):                        #returns set of actions required to
 reach goal state
    global goal
    state_list=[]
    distance_list=[]
    node = Child_Node(problem.initial_state,None,None)
    if goal_test(node.state):
        return solution(node,flag)

    frontier = []
    frontier.append(node)
    explored = []

    while True:
        if len(frontier)==0:
            return print('No such goal exists')

        heuristic = [i.path_cost+Heuristic_8puzzle(i.state) for i in frontier]
        min_index = heuristic.index(min(heuristic))



        node=frontier[min_index]
        frontier.remove(node)
        explored.append(node.state)


        for action in problem.actions(node.state):

            child = Child_Node(problem,node,action)


            if child.state not in explored:
                if check(frontier,child):
                    if goal_test(child.state):
                        print(f'The path cost is {child.path_cost} (Path is printed in
 below cell)')
                        return solution(child,flag)
                    frontier.append(child)
```

Inputs and function mapping

Example Outputs

BFS

In [23]:

```
func_map_8puzzle = {"bfs":breadth_first_search,"dfs":depth_first_search,"astar":Astar_8
puzzle,"gdbfs":greedy_bestfirstsearch}
search_8puzzle = func_map_8puzzle[input("Enter search function name : ")]

percept_8puzzle = []
percept_8puzzle_init = input("Enter the initial states:").split(' ')
percept_8puzzle.append([int(x) for x in percept_8puzzle_init ])
percept_8puzzle_goal = input("Enter the goal states:").split(' ')
percept_8puzzle.append([int(x) for x in percept_8puzzle_goal ])

seq_8puzzle = SimpleProblemSolvingAgent(percept_8puzzle,search_8puzzle,2)
print("\n\n\nActions to be performed are: ")
print(seq_8puzzle[1::])
print("Note ---Actions are based on movement of empty block")
```

```
Enter search function name : bfs
Enter the initial states:2 4 3 1 5 6 7 8 0
Enter the goal states:1 2 3 4 5 6 7 8 0
The path cost is 8 (Path is printed in below cell)



Actions to be performed are:
['UP', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'DOWN']
Note ---Actions are based on movement of empty block
```

DFS

In [25]:

```
func_map_8puzzle = {"bfs":breadth_first_search,"dfs":depth_first_search,"astar":Astar_8
puzzle,"gdbfs":greedy_bestfirstsearch}
search_8puzzle = func_map_8puzzle[input("Enter search function name : ")]

percept_8puzzle = []
percept_8puzzle_init = input("Enter the initial states:").split(' ')
percept_8puzzle.append([int(x) for x in percept_8puzzle_init ])
percept_8puzzle_goal = input("Enter the goal states:").split(' ')
percept_8puzzle.append([int(x) for x in percept_8puzzle_goal ])

seq_8puzzle = SimpleProblemSolvingAgent(percept_8puzzle,search_8puzzle,2)
print("\n\n\nActions to be performed are: ")
print(seq_8puzzle[1::])
print("Note ---Actions are based on movement of empty block")
```

```
Enter search function name : dfs
Enter the initial states:1 2 3 4 5 6 7 0 8
Enter the goal states:1 2 3 4 5 6 7 8 0
The path cost is 1 (Path is printed in below cell)



Actions to be performed are:
['RIGHT']
Note ---Actions are based on movement of empty block
```

Astar

In [26]:

```
func_map_8puzzle = {"bfs":breadth_first_search,"dfs":depth_first_search,"astar":Astar_8
puzzle,"gdbfs":greedy_bestfirstsearch}
search_8puzzle = func_map_8puzzle[input("Enter search function name : ")]

percept_8puzzle = []
percept_8puzzle_init = input("Enter the initial states:").split(' ')
percept_8puzzle.append([int(x) for x in percept_8puzzle_init ])
percept_8puzzle_goal = input("Enter the goal states:").split(' ')
percept_8puzzle.append([int(x) for x in percept_8puzzle_goal ])

seq_8puzzle = SimpleProblemSolvingAgent(percept_8puzzle,search_8puzzle,2)
print("\n\n\nActions to be performed are: ")
print(seq_8puzzle[1::])
print("Note ---Actions are based on movement of empty block")
```

```
Enter search function name : astar
Enter the initial states:1 2 3 4 5 0 6 7 8
Enter the goal states:1 2 3 4 5 6 7 8 0
The path cost is 13 (Path is printed in below cell)



Actions to be performed are:
['DOWN', 'LEFT', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'LE
FT', 'DOWN', 'RIGHT', 'RIGHT']
Note ---Actions are based on movement of empty block
```

Greedy Best First Search

In [27]:

```
func_map_8puzzle = {"bfs":breadth_first_search,"dfs":depth_first_search,"astar":Astar_8
puzzle,"gdbfs":greedy_bestfirstsearch}
search_8puzzle = func_map_8puzzle[input("Enter search function name : ")]

percept_8puzzle = []
percept_8puzzle_init = input("Enter the initial states:").split(' ')
percept_8puzzle.append([int(x) for x in percept_8puzzle_init ])
percept_8puzzle_goal = input("Enter the goal states:").split(' ')
percept_8puzzle.append([int(x) for x in percept_8puzzle_goal ])

seq_8puzzle = SimpleProblemSolvingAgent(percept_8puzzle,search_8puzzle,2)
print("\n\n\nActions to be performed are: ")
print(seq_8puzzle[1::])
print("Note ---Actions are based on movement of empty block")
```

```
Enter search function name : gdbfs
Enter the initial states:2 4 3 1 5 6 7 8 0
Enter the goal states:1 2 3 4 5 6 7 8 0



Actions to be performed are:
['UP', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'DOWN']
Note ---Actions are based on movement of empty block
```

References

For data-structures in python ---> GeeksForGeeks

For code structures and pseudocodes ---> AIMA Book

For basic structure of 8-puzzle problem ---> github AIMA repository

For Graphviz(graph visualization) ----> graphviz official documentation