Problem Modeling

To model the problem as a graph, it represents the input data as a dictionary of paths between nodes, where each key in the dictionary is a combination of two nodes (e.g. "AB" or "BA"), and the value is a list of properties for the path between those two nodes (e.g. color, line type).

It then creates another dictionary, where the keys are combinations of a node and a property for example, "cAred" or "tBdashed" where c represents the color red at node A, and t represents the line type dashed at node B, and the values are the nodes that can be reached from the key node via a path with that property. It was then later updated since some of the input code had nodes with two letters, so I created a dictionary with "_" syntax. For example, "A_B" to "DZ_Dp".

Using these dictionaries, the code constructs an adjacency list that represents the graph of the problem, where each node is a combination of a node and a property, and each edge is a path with a specific property between two nodes.

To model the graph that starts from A to G, we can use the same tactic as mentioned above. This outputs the shortest path of A B D E F D B C F G.

The code then uses the breadth-first search algorithm to traverse the graph and find the shortest path between two given nodes. I counted how many nodes it took from the start and final, and took the least amount of node to print the shortest path.

This works since the program reads the input file with information about a directed graph, and implements a BFS algorithm to find the shortest path between two nodes of the graph. The input file contains lines with the format "A B color type", where A and B are nodes of the graph, and color and type are attributes of the edge between nodes A and B. The program creates two dictionaries: dpath that stores the attributes of each edge, and dconnect that stores the destination nodes for each combination of node, color, and type.

The program then defines a function findnext that receives an edge key and returns the list of keys of the edges that are adjacent to the input edge. The adjacency list of the graph is constructed by iterating through all edges in dpath and calling the findnext function for each edge. The adjacency list is stored in the dictionary adj_list_all.

Next, the program creates a new dictionary adj_list that contains only the non-empty adjacency lists from adj_list_all, and defines three dictionaries visited, parent, and level, and a queue for implementing the BFS algorithm. The program starts the BFS algorithm from a given starting node s, and stores the visited nodes, their level from the starting node, and their parent node in the respective dictionaries. The BFS algorithm uses the adjacency list from adj_list to traverse the graph, and stores the visited nodes in bfs_traversal_output.

Finally, the program implements a function to find the shortest path from any node to a given target node by backtracking from the target node to the starting node using the parent dictionary, and stores the path in the list path. The path is then printed to the console.

```
"""

Update version can Run with town more than 1 letter

But fail when start and end contain multiple branch

#### start option:   "A_B"      "A_Q"     "A_P"
#### end option  :   "Db_Dp"    "Do_Dp"
#TstartPair = "A_Q"
#TfinalPair = "Do_Dp"



"""


import sys


filein = sys.argv[1]

with open(filein,"r") as rf:
   rln = rf.readlines()

lscol0 = [mm.split()[0] for mm in rln[1:]]
lscol1 = [mm.split()[1] for mm in rln[1:]]

ls0    = rln[0].rstrip().split()
Tstart = ls0[2]
Tend   = ls0[3]


lsTstartPair = []
lsTfinalPair = []
for ii in rln[1:]:
   ii1 = ii.rstrip().split()
   if Tstart==ii1[0]:
      lsTstartPair += ["{}_{}".format(Tstart,ii1[1]) ]
   if Tstart==ii1[1]:
      lsTstartPair += ["{}_{}".format(Tstart,ii1[0]) ]
```

```
   if Tend==ii1[1]:
      lsTfinalPair += [ "{}_{}".format(ii1[0],Tend) ]
   if Tend==ii1[0]:
      lsTfinalPair += [ "{}_{}".format(ii1[1],Tend) ]

#print ("\n")
#print ("   Possible Pair start:",lsTstartPair)
#print ("   Possible Pair end:",lsTfinalPair)


runall = "yes"

if "{}_{}".format(Tstart,Tend) in lsTstartPair:
   pathresultfinal = "{} {}".format(Tstart,Tend)
elif "{}_{}".format(Tstart,Tend) in lsTfinalPair:
   pathresultfinal = "{} {}".format(Tstart,Tend)

else:


   #if Tstart in lscol0: TstartPair = "{}_{}".format(Tstart ,
lscol1[lscol0.index(Tstart)])
   #if Tstart in lscol1: TstartPair = "{}_{}".format(Tstart ,
lscol0[lscol1.index(Tstart)])
   #
   #if Tend in lscol0: TfinalPair = "{}_{}".format(lscol1[lscol0.index(Tend)] ,
Tend)
   #if Tend in lscol1: TfinalPair = "{}_{}".format(lscol0[lscol1.index(Tend)] ,
Tend)



   #### start option:   "A_B"      "A_Q"     "A_P"
   #### end option  :   "Db_Dp"    "Do_Dp"
   #TstartPair = "A_Q"
   #TfinalPair = "Do_Dp"

   #print ('start from: {}    end: {}'.format(TstartPair,TfinalPair))



   ## create dictionary 2 ways that contain property
   dpath = {}   ### key = TownFrTo _ TownToFr = [Color,linetype]
```

```
   dconnect = {}   ### key = "c"/"t" _ Color/linetype _ Town _
Color/linetype       --> value = [destinasi kota]   pas nyari kota darinya
dihilangkan di list

   for i in rln[1:]:
      ii = i.rstrip().split()


      ky1 = "{}_{}".format(ii[0],ii[1])
      ky2 = "{}_{}".format(ii[1],ii[0])
      prop = [ii[2],ii[3]]

      if ky1 not in dpath: dpath[ky1]=prop
      else: print ("  double entry read already exist ",ii)

      if ky2 not in dpath: dpath[ky2]=prop
      else: print ("  double entry read already exist ",ii)

      ########    kota + color + (RGB)   -- value kota dest
      con1     = "{}_{}_{}".format('c',ii[0],ii[2])
      ########    kota + type + (HCTB)   -- value kota dest
      con2     = "{}_{}_{}".format('t',ii[0],ii[3])
      if con1 in dconnect: dconnect[con1] += [ii[1]]
      else               : dconnect[con1]  = [ii[1]]
      if con2 in dconnect: dconnect[con2] += [ii[1]]
      else               : dconnect[con2]  = [ii[1]]

      ########    kota + color + (RGB)   -- value kota dest
      con3     = "{}_{}_{}".format('c',ii[1],ii[2])
      ########    kota + type + (HCTB)   -- value kota dest
      con4     = "{}_{}_{}".format('t',ii[1],ii[3])
      if con3 in dconnect: dconnect[con3] += [ii[0]]
      else               : dconnect[con3]  = [ii[0]]
      if con4 in dconnect: dconnect[con4] += [ii[0]]
      else               : dconnect[con4]  = [ii[0]]


   #for ky in dpath:
   #   print (ky, dpath[ky])
   #   # B_A ['R', 'C']
   #   # A_B ['R', 'C']
   #   # B_E ['B', 'C']
   #   # E_B ['B', 'C']
   #   # B_C ['B', 'T']
   #   # C_B ['B', 'T']
```

```
   #   # C_D ['G', 'T']
   #   # D_C ['G', 'T']
   #
   #for ky in dconnect:
   #   print ("---",ky, dconnect[ky])
   #   # --- c_B_R ['A', 'F']
   #   # --- t_B_C ['A', 'E']
   #   # --- c_A_R ['B']
   #   # --- t_A_C ['B']
   #   # --- c_B_B ['E', 'C']
   #   # --- c_E_B ['B']
   #   # --- t_E_C ['B', 'O']
   #   # --- t_B_T ['C', 'F']
   #   # --- c_C_B ['B']
   #   # --- t_C_T ['B', 'D']
   #   # --- c_C_G ['D']


   def findnextTown(findnext):
      findnext1 = findnext.split("_")[1]
      findnext0 = findnext.split("_")[0]


      #lsTfound1 =
dconnect["{}_{}_{}".format('c',findnext1,dpath[findnext][0])]   ### find next
town from findnext0_findnext1
      #lsfound1 = []                  ### combine next find town with end of
first town
      #for kk in lsTfound1:
      #   lsfound1 += [ "{}_{}".format(findnext1,kk) ]


      lsfound1 = [ "{}_{}".format(findnext1,kk) for kk
in  dconnect["{}_{}_{}".format('c',findnext1,dpath[findnext][0])] ]
      if "{}_{}".format(findnext1,findnext0) in
lsfound1:  lsfound1.remove("{}_{}".format(findnext1,findnext0))

      lsfound2 = [ "{}_{}".format(findnext1,kk) for kk
in  dconnect["{}_{}_{}".format('t',findnext1,dpath[findnext][1])] ]
      if "{}_{}".format(findnext1,findnext0) in
lsfound2:  lsfound2.remove("{}_{}".format(findnext1,findnext0))

      for kk in lsfound2:
         if kk not in lsfound1: lsfound1+= [kk]
```

```
      return lsfound1


   ### Make Dictionary with key=TownFrTo and value=next TownFrom(end of town key
search) to Next Town
   adj_list = {}
   for i in rln[1:]:
      ii = i.rstrip().split()

      #print ( "{}_{}".format(ii[0]+ii[1]) , findnextTown(
"{}_{}".format(ii[0]+ii[1]) ) )
      #print ( "{}_{}".format(ii[1]+ii[0]) , findnextTown(
"{}_{}".format(ii[1]+ii[0]) ) )

      lsres1 = findnextTown( "{}_{}".format(ii[0],ii[1]) )
      lsres2 = findnextTown( "{}_{}".format(ii[1],ii[0]) )
      #if lsres1 != []: adj_list[ ii[0]+ii[1] ] = lsres1
      #if lsres2 != []: adj_list[ ii[1]+ii[0] ] = lsres2

      adj_list[ "{}_{}".format(ii[0],ii[1]) ] = lsres1
      adj_list[ "{}_{}".format(ii[1],ii[0]) ] = lsres2



   ##########################################################################
   ##########################################################################
   ######   this part start to calculate possible path by combination
   ######      and then select the smallest
   ##########################################################################
   ##########################################################################


   nominpath = 99999999999999999999999999999.

   from queue import Queue

   pathresultfinal = "NO PATH"

   for istartpair in lsTstartPair:
      for ifinalpair in lsTfinalPair:


         TstartPair = istartpair
         TfinalPair = ifinalpair
```

```
        #print ('\n==========   Version From:"{}" to
"{}"   =========='.format(TstartPair,TfinalPair))

        #bfs code
        visited = {}
        level   = {}
        parent  = {}
        bfs_traversal_output = []
        queue = Queue()

        for node in adj_list.keys():
           visited[node] = False
           parent[node] = None
           level[node] = -1

        #TstartPair = 'AB'
        visited[TstartPair] = True
        level[TstartPair] = 0
        queue.put(TstartPair)

        while not queue.empty():
           u = queue.get()
           bfs_traversal_output.append(u)

           for v in adj_list[u]:
              try:
               if not visited[v]:
                 visited[v] = True
                 parent[v] = u
                 level[v] = level[u]+1
                 queue.put(v)
              except:
               pass


        #shortest path of from  any node from source code
        #v = 'ij'
        path = []
        while TfinalPair is not None:
           path.append(TfinalPair)
           TfinalPair = parent[TfinalPair]
        path.reverse()

        #print (path)
        #
```

```
        #if path == [] or len(path)<= 1: pathresult = 'NO PATH'
        #else                           :
        #   lspath     = [ll.split("_")[0] for ll in path]
        #   pathresult = " ".join(lspath) + " " + path[-1].split("_")[-1]
        #
        #print ( "          no Town Transit : {}".format(len(path)) )
        #print ( "          Shortest Path -->",pathresult  )

        #### this is to select final path
        if path == [] or len(path)<= 1: pass #pathresultfinal = "NO PATH"
        else :
           if len(path) < nominpath:
              lspath     = [ll.split("_")[0] for ll in path]
              nominpath = len(path)
              pathresultfinal = " ".join(lspath) + " " + path[-1].split("_")[-1]


print (pathresultfinal)
```