Appendix A

graph.py

```
# This file is the main file for the ECE 760 Graphic Probability Model in 2018
# Fall, Texas A & M University.
# Author: Tianqi Li
# the file is the Problem 1 in hw2, which constructs a class called DAG and
# executes the d-separate searching ALG to return set Y of all vertices that is
# d-separated from the source vertex given observation Z.
from collections import deque


class DAG(object):
    # This class is used for Directed Acyclic Graph, which same vertex cannot
    # be met twice in a path

    def __init__(self, graph_dict=None):
        if graph_dict == None:
            graph_dict = {}
        self.graph_dict = graph_dict  # includes all nodes with their children
        self.parents = dict.fromkeys(self.graph_dict.keys())
        for key in self.parents:
            self.parents[key] = []  # initilize the self.parents list
        self.get_vertex()
        self.get_parent()

    def get_vertex(self):
        # get the list of all vertices, which is stored in self.vertices
        self.vertices = sorted(self.graph_dict.keys())

    def get_parent(self):
        # get the parents of all vertices, which is stored in self.parents
        for vertex in self.vertices:
            for i in self.vertices:
                if vertex in self.graph_dict[i]:
                    self.parents[vertex].append(i)

    def get_A(self, observe):
        # this function gets all parents of the observation Z, which is used to
        # check V-constructs
        L, A = deque(), list()
        for i in observe:
            L.append(i)
        while len(L) != 0:
```

```
        n = L.popleft()
        if n not in A:
            for par in self.parents[n]:
                L.append(par)
        A.append(n)
    return sorted(A)  # return the set A= {parents of Z}

def D_sep(self, sn, observe):
    # This algorithm returns
    # Y ={d-separated nodes from sn | observation set Z}
    # Reference: P75 Algorithm 3.1 in Probabilisitc Graphic Model: Principles and Techniques
    # Input: (sn: start node, observe: observation)
    # Output: Y ={d-separated nodes from sn | observation set Z}

    L, V, R = deque(), list(), list()
    # L: a queue of tuples of (node, direction) to visit
    # specificly, direction :
    # 'forward' same direction with the edge, 'backward' is opposite
    # V: visited nodes
    # R: all reachable nodes
    L.append((sn, 'backward'))
    A = self.get_A(observe)

    while len(L) != 0:
        (node, direction) = L.popleft()
        # pop the first node out of the queue
        if (node, direction) not in V:
            # the node + direction is not visited
            if node not in observe and node not in R:
                R.append(node)
            V.append((node, direction))  # mark node + direction visited
            if direction == 'backward' and node not in observe:
                for z in self.parents[node]:
                    # evidential trail
                    L.append((z, 'backward'))
                for z in self.graph_dict[node]:
                    # common cusal trail
                    L.append((z, 'forward'))
            elif direction == 'forward':
                if node not in observe:
                    # causal trail
                    for z in self.graph_dict[node]:
                        L.append((z, 'forward'))
                if node in A:
```

```
            # there is a activated V- structure, so trace the parents
            for z in self.parents[node]:
                L.append((z, 'backward'))
    return sorted(self.graph_dict.keys() - R - observe)  # Y =  All nodes - R – Z
```

Appendix B

pearl.py

```
# This file contains the inherit class Pearls, which contains most function to realize
# Pearl's message passing ALG in the DAG.
import numpy as np
from graph import DAG
import sys
import copy


class Pearls(DAG):

    def __init__(self,  graph_dict, marginal, CPD):
        '''
        initialize function in pearls ALG, whcih includes initialization of
        BN in DAG class, lamda value, lamda message of all nodes, initialization
        of pi value of all root nodes.
        '''

        DAG.__init__(self,  graph_dict)
        self.E, self.e = [], []
        self.value = {}
        for key in list(marginal.keys()):
            self.value[key] = list(marginal[key].keys())  # this is value set each node can take
        self.lamda = dict.fromkeys(graph_dict.keys())
        self.pi = dict.fromkeys(graph_dict.keys())
        self.lamda_msg = dict.fromkeys(graph_dict.keys())
        self.pi_msg = dict.fromkeys(graph_dict.keys())

        self.cpd = dict.fromkeys(graph_dict.keys())

        for x in self.vertices:
            self.lamda[x], self.lamda_msg[x], self.pi[x], self.pi_msg[x], self.cpd[x] = {}, {}, {}, {}, {}

#        given info of the graph
        self.marginal = marginal
        self.CPD = CPD

        self.get_root()
#        initialize lamda
        for x in self.vertices:
            for x_value in self.value[x]:
                # data strucutre for lamuda(X = x_value) is self.lamda[X][x_value]
```

```
            self.lamda[x][x_value] = 1

#       initialize lamda_msg
        for z in self.parents[x]:
            self.lamda_msg[z][x] = {}
            for z_value in self.value[z]:
                # data strucutre for lamuda_X(Z = z_value) is self.lamda_msg[Z][X][z_value]
                self.lamda_msg[z][x][z_value] = 1

#       initialize pi_msg
        for y in self.graph_dict[x]:
            self.pi_msg[x][y] = {}
            for x_value in self.value[x]:
                # data strucutre for pi_Y(Z = z_value) is self.lamda_msg[Z][Y][z_value]
                self.pi_msg[x][y][x_value] = 1


#       send pi_message
    for R in self.root:
        for r_value in self.value[R]:
            # data strucutre for pi(X = x_value) is self.pi[X][x_value]
            self.pi[R][r_value] = self.marginal[R][r_value]
            # data strucutre for P(X = x|e) is self.cpd[X][x]
            self.cpd[R][r_value] = self.marginal[R][r_value]

        for W in self.graph_dict[R]:
            self.send_pi_msg(R, W)

  def get_root(self):
    # this function returns the root nodes of the BN
    self.root = []
    for x in self.vertices:
        if self.parents[x] == []:
            self.root.append(x)
    return

  def send_pi_msg(self, Z, X):
    # Z is parent node, X is child node, message is Z -> X

    # get all parent of Z:
    U = copy.deepcopy(self.graph_dict[Z])
    # exclude Z itself, BE CAUTIOUS THAT WE NEED TO USE deepcopy to keep
    # original data save when using remove function
    U.remove(X)
```

```
        for z_value in self.value[Z]:
            self.pi_msg[Z][X][z_value] = self.pi[Z][z_value]
            if U != []:
                print('falut!!!!!!', Z)
                for u in U:
                    self.pi_msg[Z][X][z_value] = self.pi_msg[Z][X][z_value] * \
                        self.lamda_msg[Z][u][z_value]

        P_tilta = []
        if X not in self.E:

            for x_value in self.value[X]:
                [zs, probs] = self.CPD[X][x_value]
                total = self.calculate_pi(X, x_value, zs, probs)
                self.pi[X][x_value] = total
                P_tilta.append(self.lamda[X][x_value] * self.pi[X][x_value])

            self.normalize(X, P_tilta)

            for Y in self.graph_dict[X]:
                self.send_pi_msg(X, Y)

        for x_value in self.value[X]:
            if self.lamda[X][x_value] != 1:
                # here is to check if V-structure is turned on, by checking the
                # descendant's lamda value
                other_parents = copy.deepcopy(self.parents[X])
                other_parents.remove(Z)
                if other_parents != []:
                    for W in other_parents:
                        if W not in self.E:
                            print('falut!!w', W)
                            self.send_lamda_msg(X, W)

        return

    def calculate_pi(self, X, x, zs, probs):
        #       X: node name of X, x is value of the X node
        #       zs are node names of all other parents
        #       probs will be a k dimentional 2*...2*2 matrix for k parents
        k = len(zs)

        prob_matrix = np.array(probs)
        prob_matrix = prob_matrix.flatten()
```

```
    total = 0

    for i in range(3**k):

        p = prob_matrix[i]

        for j in range(k):

            if i - 2*3**(k-j-1) + 1 > 0:
                #  this means z_k value is 2
                try:
                    p = p * self.pi_msg[zs[j]][X][2]
                    i = i - 2*3**(k-j-1)
                except:
                    print(X, zs)
                    print([zs[j]], [X], [2])
                    print(self.pi_msg[zs[j]][X][2])
                    i = i - 2*3**(k-j-1)
            elif i - 3**(k-j-1) + 1 > 0:
                #  this means z_k value is 1
                p = p * self.pi_msg[zs[j]][X][1]

                i = i - 3**(k-j-1)
            else:
                #  this means z_k value is 0
                p = p * self.pi_msg[zs[j]][X][0]

        total += p

    return total

def normalize(self, X, P):
    # this function is to normalize the P_tilta
    sum_ = 0
    for i in self.value[X]:
        sum_ += P[i]
    for x_value in self.value[X]:
        self.cpd[X][x_value] = P[x_value] / sum_
    return

def calculate_lamda(self, X, x, Y):
    # similar to the calculate_pi function, but when doing product, x will
    # be passed
    zs = self.CPD[Y][0][0]
```

```
k = len(zs)

total = 0
x_index = zs.index(X)

for y in self.value[Y]:
    prob_matrix = np.asarray(self.CPD[Y][y][1]).flatten()

    for i in range(3**k):
        p = prob_matrix[i]
        for j in range(k):
            if i - 2*3**(k-j-1) + 1 > 0:
                #  this means x_k value is 2
                if j == x_index:
                    # pass x
                    if x != 2:
                        p = 0
                    else:
                        p = p
                else:
                    p = p * self.pi_msg[zs[j]][Y][2]
                i = i - 2*3**(k-j-1)
            elif i - 3**(k-j-1) + 1 > 0:
                #  this means x_k value is 1
                if j == x_index:
                    # pass x
                    if x != 1:
                        p = 0
                    else:
                        p = p
                else:
                    p = p * self.pi_msg[zs[j]][Y][1]

                i = i - 3**(k-j-1)
            else:
                #  this means x_k value is 0
                if j == x_index:
                    # pass x
                    if x != 0:
                        p = 0
                    else:
                        p = p
                else:
```

```
                    p = p * self.pi_msg[zs[j]][Y][0]

            total += p * self.lamda[Y][y]

        return total

    def send_lamda_msg(self, Y, X):
        #        Y(child) -> X(parent)

        P_tilta = []

        for x_value in self.value[X]:

            self.lamda_msg[X][Y][x_value] = self.calculate_lamda(X, x_value, Y)

            lamda = 1

#           calculate lamuda(x)
            for u in self.graph_dict[X]:
                lamda = lamda * self.lamda_msg[X][u][x_value]
            self.lamda[X][x_value] = lamda

            P_tilta.append(self.lamda[X][x_value] * self.pi[X][x_value])

#       get self.cpd
        self.normalize(X, P_tilta)

        for Z in self.parents[X]:
            if Z not in self.E:
                self.send_lamda_msg(X, Z)

        for U in self.graph_dict[X]:
            if U != Y:
                self.send_pi_msg(X, U)

        return

    def update_network(self, V, v_value_hat):

        self.E.append(V)
        self.e.append((V, v_value_hat))

        for v in self.value[V]:
            if v == v_value_hat:
```

```
            self.lamda[V][v] = 1
            self.pi[V][v] = 1
            self.cpd[V][v] = 1
          else:
            self.lamda[V][v] = 0
            self.pi[V][v] = 0
            self.cpd[V][v] = 0

        for Z in self.parents[V]:
          if Z not in self.E:
            self.send_lamda_msg(V, Z)

        for Y in self.graph_dict[V]:
          self.send_pi_msg(V, Y)

        return


def solver(g, marginal, CPD, questions):
    '''
    This function is for solving the CPD inference,
    P(X = x|E = e)
    '''
    Answers = []
    for q in questions:
        # initialize pearls BN
        A = Pearls(g, marginal, CPD)
        answer = 1
        [X, E, x, e] = q
        for i in range(len(E)):
            # update all evidence
            A.update_network(E[i], e[i])

        # get all X variable one by one
        for j in range(len(X)):
            answer = answer * A.cpd[X[j]][x[j]]
            A.update_network(X[j], x[j])
#    answer = answer * A.cpd[X[j]][x[j]]

#   answer = answer * A.cpd[X[-1]][x[-1]]
        print("P(", X, "=", x, " | e = ", E, "=", e, ") = ", answer)
        Answers.append(answer)
    return Answers
```

Appendix C

main.py

```
# This file is the main file for the ECE 760 Graphic Probability Model in 2018 Fall, Texas A & M University.
# Author: Tianqi Li
# the file is the part 2 for final project, which implement the Pearl's message
# passing ALG in a human error preditction model
#
# few notation in this code:
# pi_X(Z=z_value) is pi_msg[Z][X][z_value]
# lamda_X(Z = z_value) is lamda_msg[Z][X][z_value]

import numpy as np
from graph import DAG
import sys
from pearl import Pearls
import copy


def main():

    g = {"t": ["w"],
         "o": ["w"],
         "q": ["s"],
         "r": ["s"],
         "a": ["p"],
         "w": ["p"],
         "e": ["f"],
         "i": ["f"],
         "p": ["h"],
         "f": ["h"],
         "s": ["h"],
         "h": []
         }

    marginal = {"t": {0: 0.1, 1: 0.3, 2: 0.6},
                "o": {0: 0.05, 1: 0.2, 2: 0.75},
                "q": {0: 0.05, 1: 0.15, 2: 0.8},
                "r": {0: 0.1, 1: 0.2, 2: 0.7},
                "a": {0: 0.2, 1: 0.3, 2: 0.5},
                "w": {0: [], 1: [], 2: []},
                "e": {0: 0.1, 1: 0.4, 2: 0.5},
                "i": {0: 0.1, 1: 0.3, 2: 0.6},
```

```
        "p": {0: [], 1: [], 2: []},
        "f": {0: [], 1: [], 2: []},
        "s": {0: [], 1: [], 2: []},
        "h": {0: [], 1: []}
        }

CPD = {"t": [],
    "o": [],
    "q": [],
    "r": [],
    "a": [],
    "w": {0: [["t", 'o'], [[0.01, 0.05, 0.2], [0.05, 0.1, 0.8], [0.2, 0.8, 0.9]]],
        1: [["t", 'o'], [[0.09, 0.15, 0.6], [0.15, 0.8, 0.15], [0.6, 0.15, 0.09]]],
        2: [["t", 'o'], [[0.9, 0.8, 0.2], [0.8, 0.1, 0.05], [0.2, 0.05, 0.01]]]},
    "e": [],
    "i": [],
    "p": {0: [["a", 'w'], [[0.2, 0.05, 0.01], [0.6, 0.1, 0.1], [0.9, 0.8, 0.3]]],
        1: [["a", 'w'], [[0.5, 0.15, 0.09], [0.3, 0.7, 0.3], [0.09, 0.15, 0.5]]],
        2: [["a", 'w'], [[0.3, 0.8, 0.9], [0.1, 0.2, 0.6], [0.01, 0.05, 0.2]]]},
    "f": {0: [["e", 'i'], [[0.9, 0.8, 0.6], [0.4, 0.15, 0.1], [0.2, 0.1, 0.01]]],
        1: [["e", 'i'], [[0.09, 0.15, 0.3], [0.5, 0.7, 0.5], [0.6, 0.2, 0.09]]],
        2: [["e", 'i'], [[0.01, 0.05, 0.1], [0.1, 0.15, 0.4], [0.2, 0.7, 0.9]]]},
    "s": {0: [["q", 'r'], [[0.9, 0.7, 0.2], [0.7, 0.2, 0.1], [0.2, 0.1, 0.01]]],
        1: [["q", 'r'], [[0.09, 0.2, 0.6], [0.2, 0.6, 0.6], [0.6, 0.3, 0.09]]],
        2: [["q", 'r'], [[0.01, 0.1, 0.2], [0.1, 0.2, 0.3], [0.2, 0.6, 0.9]]]},
    "h": {0: [['p', 'f', 's'], [[[0.3, 0.5, 0.7], [0.5, 0.8, 0.9], [0.7, 0.9, 0.99]],
                    [[0.8, 0.4, 0.5], [0.3, 0.6, 0.8], [0.5, 0.8, 0.9]],
                    [[0.01, 0.1, 0.3], [0.2, 0.4, 0.5], [0.3, 0.5, 0.7]]]],
        1: [['p', 'f', 's'], [[[0.7, 0.5, 0.3], [0.5, 0.2, 0.1], [0.3, 0.1, 0.01]],
                    [[0.2, 0.6, 0.5], [0.7, 0.4, 0.2], [0.5, 0.2, 0.1]],
                    [[0.99, 0.9, 0.7], [0.8, 0.6, 0.5], [0.7, 0.5, 0.3]]]]},
    }
# all inference
A = Pearls(g, marginal, CPD)
print('the prior probability', A.cpd)
# get the Posterior probability
A.update_network('h', 1)
print('the posterior probability', A.cpd)
# calculae the h=1 given different root condition
print('calculae the h=1 given different root condition')
for r in A.root:
  A = Pearls(g, marginal, CPD)
  A.update_network(r, 0)
  print(r, 'caused the error probability is', A.cpd['h'][1])
```

```
if __name__ == "__main__":
    main()
```