Project 1: Bayesian Structure Learning

Arpit Dwivedi dwivedi7@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

I experimented with several strategies to learn the Bayesian structure for the provided small, medium, and large datasets. After running each algorithm and evaluating the results, I submitted the one that achieved the highest score for each dataset. Below is a summary of the algorithm used and its runtime for each dataset:

1.1 Small Dataset

- **Algorithm:** For this dataset I have first passed the data through K2 search to get initialized structure with the upper limit on mumber of parents to be 5. The initialized structure was passed to the opportunistic version of Local Search modified with simulated annealing. Instead of always moving to the neighbor with greatest fitness, the search can visit neighbors with lower fitness at k_t iteration of local search with probability ϵ/k_t, where epsilon was chosen to be 0.17.
- The runtime for the same came out to be 27.477636098861694 sec with the final Bayesian Score as -3794.8555977098003.

1.2 Medium Dataset

- **Algorithm:**For this dataset I have first passed the data through *mutual.py* to get the mutual information between nodes as per following formula:

$$I(X;Y) = \sum_{x,y} \hat{P}(x,y) \log \left(\frac{\hat{P}(x,y)}{\hat{P}(x)\hat{P}(y)} \right)$$

  where $\hat{P}$ $(\cdot)$ is the observed frequencies in the dataset. Top $[\frac{n(n-1)}{4}]$.i.e..n - number of nodes, were used as initial edges for the graph. The initialized structure was passed to the opportunistic version of Local Search modified with simulated annealing. Instead of always moving to the neighbor with greatest fitness, the search can visit neighbors with lower fitness at k_t iteration of local search with probability ϵ/k_t, where epsilon was chosen to be 0.17. randomized exploration strategy
- The runtime for the same came out to be 822.2017841339111 sec with the final Bayesian Score as -96886.944280578.

1.3 Large Dataset

- **Algorithm:** For this dataset I have first passed the data through K2 search to get initialized structure with the upper limit on mumber of parents to be 5. The initialized structure was passed to the opportunistic version of Local Search modified with

simulated annealing. Instead of always moving to the neighbor with greatest fitness, the search can visit neighbors with lower fitness at k_t iteration of local search with probability ϵ/k_t, where epsilon was chosen to be 0.17.

- The runtime for the large datset came out to be 14899.134365558624 sec with the final Bayesian Score as -440669.22905314964.

2. Graphs

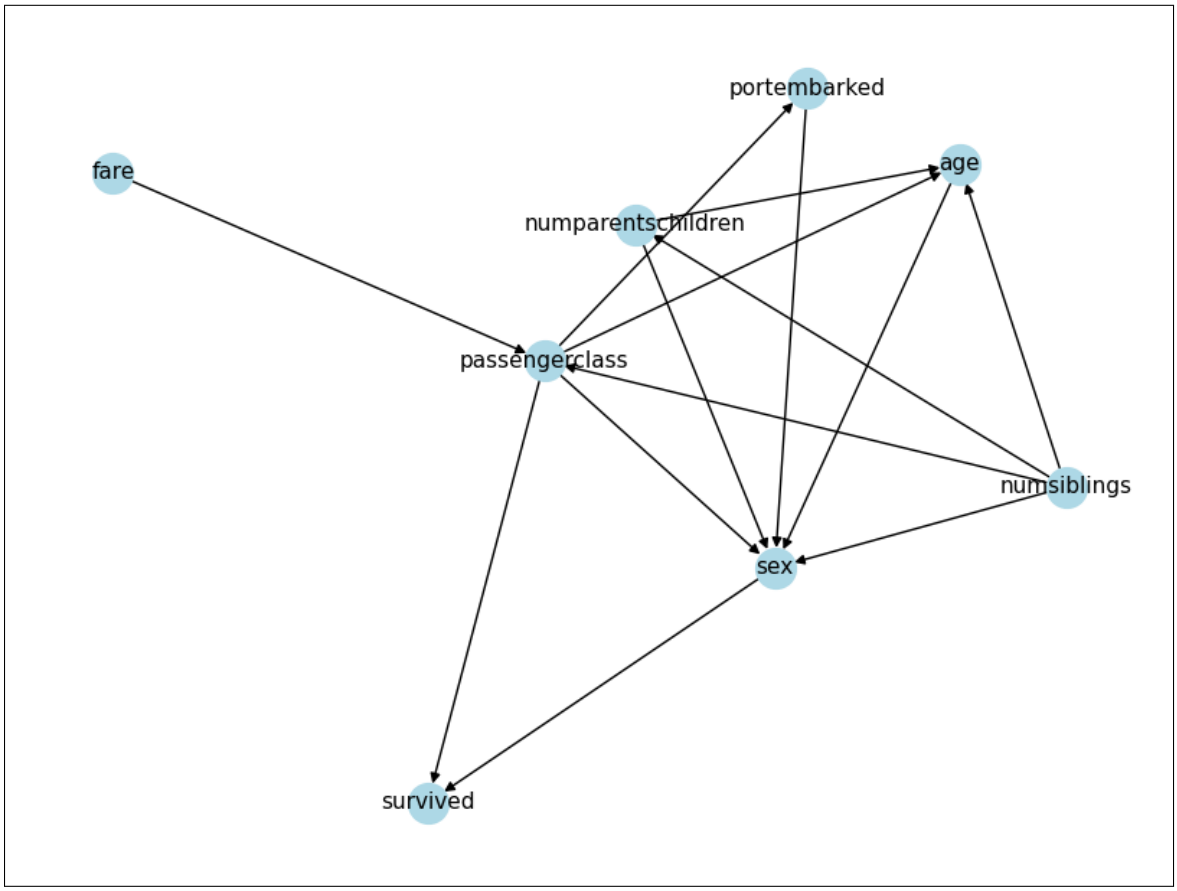

Figure 1: Graph : Small Dataset

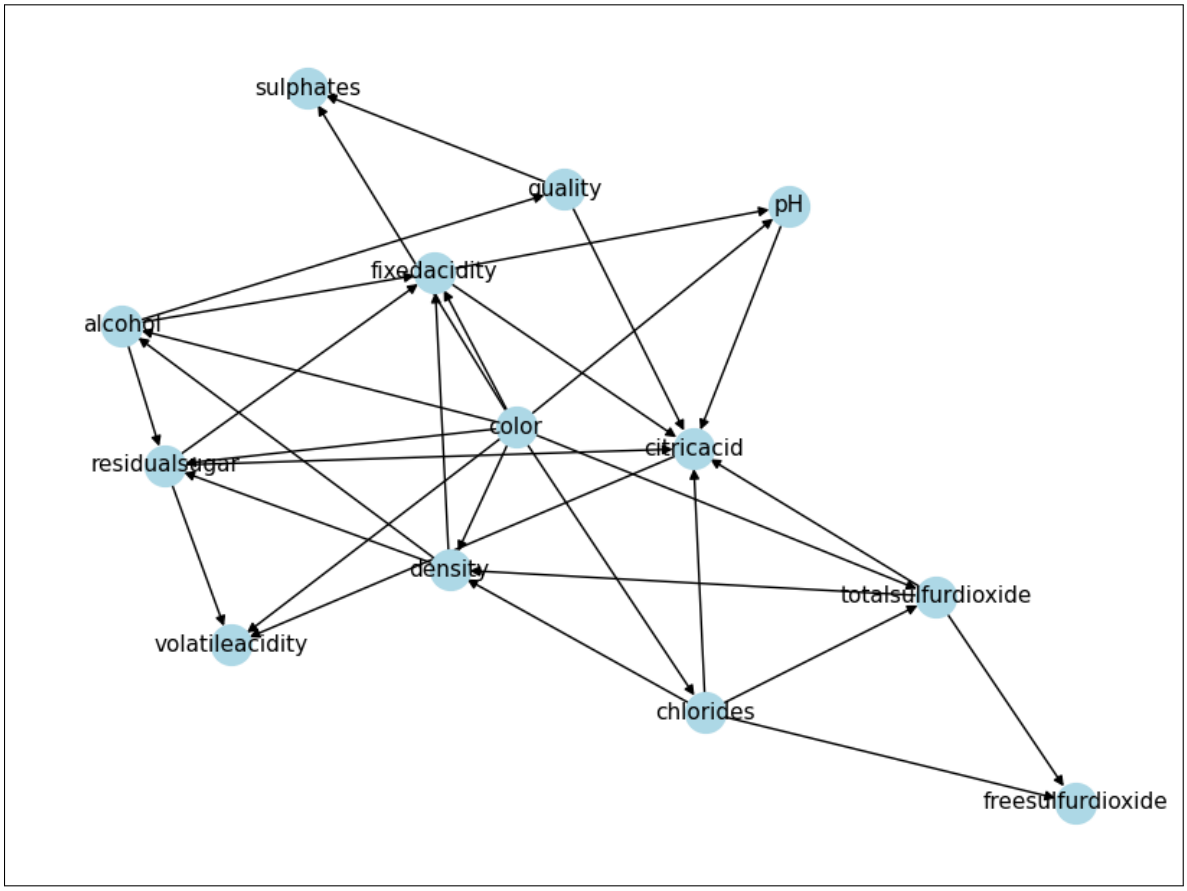

Figure 2: Graph : Medium Dataset

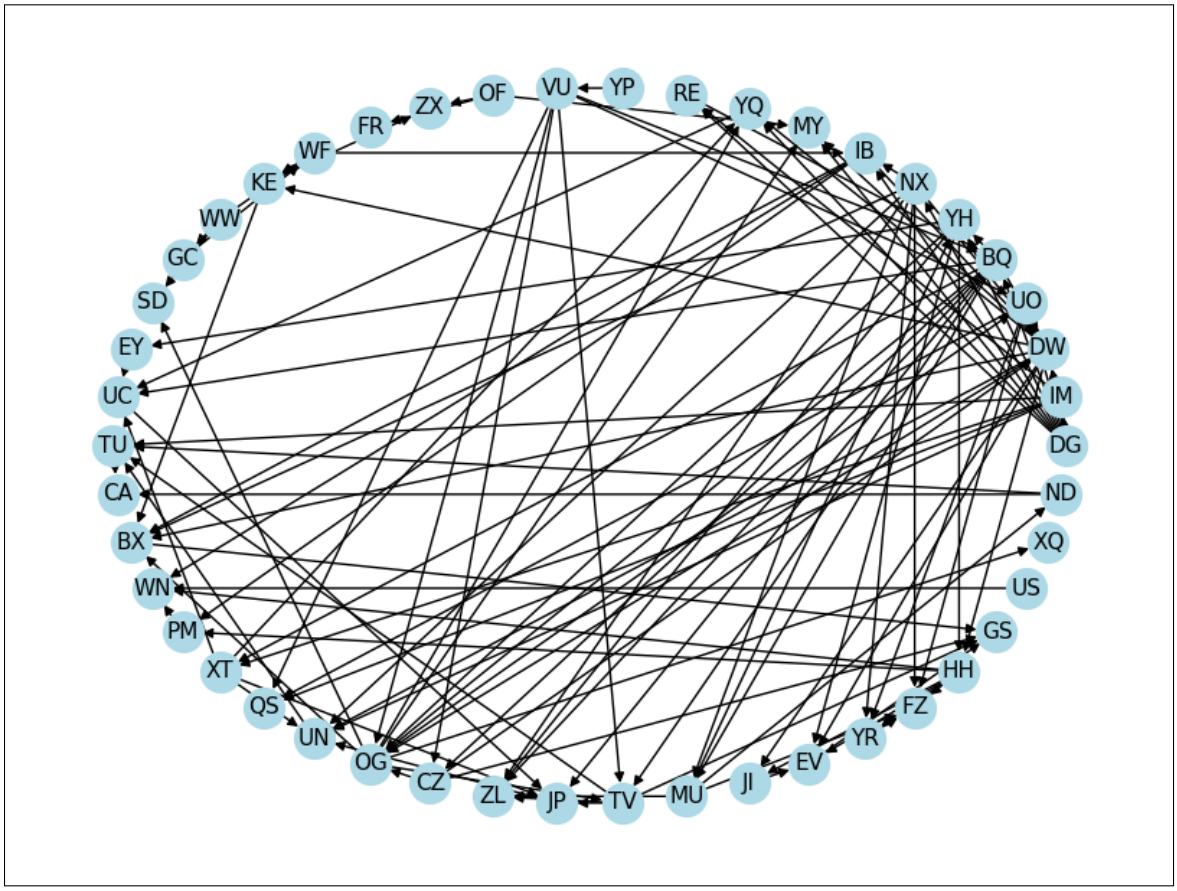

Figure 3: Graph : Large Dataset

3. Code

*project*1.*py* file:

```
import sys
import csv
import networkx as nx
import numpy as np
import pandas as pd
import math
import random
import copy
import time
from scipy.special import gammaln
from matplotlib import pyplot as plt
from mutual import *


class StructureLearning:
    def __init__(self,inputfile, outputfile):
```

```
        self.inputfilepath = inputfile
        self.outputfilepath = outputfile
        self.upperlimit_parents = 5
        self.epsilon = 0.17
        self.BS = []

        #Reading the CSV file
        self.read_csv()

        #Creating the Directed Graph Structure
        self.GraphInitialize()

        #Implement K2 search to get first grpah network for local search
        start_time = time.time()
        self.K2_Search()
        end_time = time.time()

        # G_deepcopy = copy.deepcopy(self.DAG)
        # egdelistK2 = G_deepcopy.edges()

        # Calculate the time difference
        time_taken = end_time - start_time
        print('This is the total run time:', time_taken)

        #Implement Local Search
        # self.Init_for_local_search()   # if k2 is called dont call this function
        self.local_search()



        ############################################################
        ############################################################
        ## Implement Gentic Crossover and call local search again
        # G_deepcopy2 = copy.deepcopy(self.DAG)
        # egdelistlocal = G_deepcopy2.edges()
        # set_p = egdelistlocal
        # self.GC(egdelistK2,egdelistlocal)
        # self.local_search()

        # for i in range(10):
        #     G_deepcopy2 = copy.deepcopy(self.DAG)
        #     set_n = G_deepcopy2.edges()

        #     self.GC(set_p, set_n)
        #     self.local_search()
        #     set_p = set_n


        ############################################################
        ############################################################
```

```
        end_time_local = time.time()
        time_taken2 = end_time_local - start_time
        print('This is the total run time after local search:', time_taken2)


        #Write the gph file
        self.write_gph()

        plt.plot(self.BS)
        plt.show()

    def GC(self,set_1,set_2):
        # print(set_1)
        # print(set_2)

        min_len = min(len(set_1), len(set_2))
        set_1 = list(set_1)[:min_len]
        set_2 = list(set_2)[:min_len]

        # Randomly choose a crossover point
        crossover_point = random.randint(1, min_len - 1)

        # Create new edge set by combining parts from each parent
        child_edges = set_1[:crossover_point] + set_2[crossover_point:]
        # print(child_edges)
        self.DAG.remove_edges_from(list(self.DAG.edges))

        self.DAG.add_edges_from(child_edges)


    def write_gph(self):
        with open(self.outputfilepath, 'w') as f:
            for edge in self.DAG.edges():
                f.write("{}, {}\n".format(edge[0], edge[1]))



    def read_csv(self):
        try:
            df = pd.read_csv(self.inputfilepath)
            self.Nodes = df.columns.tolist()
            # print("Nodes:", self.Nodes[0])

            self.Sample_Data = df.iloc[:].to_numpy()
            # print(self.Sample_Data.shape)    # The shape of data is (N,m): N
    sample points and m variables(nodes)


        except FileNotFoundError:
            print(f"File {self.inputfilepath} not found!")
```

```
def GraphInitialize(self):
    self.DAG = nx.DiGraph()
    self.DAG.add_nodes_from(self.Nodes)

    # print('These are the Nodes of the Grpah:',self.DAG.nodes)
    self.D = self.Sample_Data.transpose()  #required shape of Data matrix for
algorithm (m,n)
    self.n = self.D.shape[0]  #Number of variables

    #Number of instantiations of each n variables by finding number of unique
arguments
    self.r_values = {node: np.max(self.D[idx]) for idx, node in enumerate(self
.Nodes)}
    self.idx_values = {node: idx for idx, node in enumerate(self.Nodes)} #
assigning indexes to nodes
    nx.set_node_attributes(self.DAG, self.r_values, name='r')
    nx.set_node_attributes(self.DAG, self.idx_values, name='i')
    self.r = [self.DAG.nodes[self.Nodes[i]]['r'] for i in range(self.n)]

def random_DAG(self):
    for i in self.Nodes:
        for j in self.Nodes:
        # Randomly decide if there should be an edge from node i to node j
            if i !=j :
                if random.random() > 0.5:
                    self.DAG.add_edge(i, j)
                    if nx.is_directed_acyclic_graph(self.DAG):
                        if random.random() > 0.5:
                            self.DAG.remove_edge(i, j)
                        else:
                            self.DAG.remove_edge(i, j)



def K2_Search(self):
    ############Have to give a variable ordering ###############
    ############################################################
    New_Node_ordering =random.sample(self.Nodes, len(self.Nodes))


    for node_i in New_Node_ordering[1:]:
        y = self.BayesianScore()
        self.BS.append(y)
        ############
        print('Bayesian Score changes during K2 search:',y)
        ############
        while True:
            y_best, j_best = -np.inf, 0
            for nodeJ in New_Node_ordering[:self.DAG.nodes[node_i]['i']]:
                if self.DAG.has_edge(nodeJ, node_i) == 0:
```

```
                    if self.DAG.in_degree(node_i) < self.upperlimit_parents: #
Upper limiting the number of parents
                        self.DAG.add_edge(nodeJ, node_i)                    #
Adding the nodes
                        y_prime = self.BayesianScore()
                        if y_prime > y_best:
                            y_best, j_best = y_prime, self.DAG.nodes[nodeJ]['i
']

                        self.DAG.remove_edge(nodeJ, node_i)


            if y_best > y:
                y = y_best
                self.DAG.add_edge(self.Nodes[j_best], self.Nodes[self.DAG.
nodes[node_i]['i']])  #Permamnently adding the nodes
            else:
                break   #the logic is if the first edge added doesn't increase
 the score that means
                        #the the second parent will not increase further so
break thw while loop


def Bayesian_score_component(self,M,al):
    p = np.sum(gammaln(al + M))
    p -= np.sum(gammaln(al))
    p += np.sum(gammaln(np.sum(al, axis=1)))
    p -= np.sum(gammaln(np.sum(al, axis=1) + np.sum(M, axis=1)))
    # print(M,p)

    return p

def BayesianScore(self):
    if nx.is_directed_acyclic_graph(self.DAG):
        M, alpha = self.statistics()
        Score = np.sum([self.Bayesian_score_component(M[i], alpha[i]) for i in
 range(self.n)])

        return Score
    else:
        return -float('inf')


#Extracting the counts from the data
def statistics(self):
    #Calculating qi's for each node
    qi = np.ones(self.n, dtype=int)
    for idx, node in enumerate(self.Nodes):
        for paren in self.DAG.predecessors(node):
            qi[idx] = qi[idx]*self.DAG.nodes[paren]['r']
```

```
        M = [np.zeros((qi[i], self.r[i]),dtype=int) for i in range(self.n)]
        alpha = [np.ones((qi[i], self.r[i]),dtype=int) for i in range(self.n)]

        '''
        Now we will iterate over i,j,k elemets of matrix M and update the counts
        i: variable node
        j: which parent instantiation: this has to be calculated
        k: instatntion of variable node
        '''
        for sample in self.Sample_Data:
            for i, node in enumerate(self.Nodes):
                # print(f'i : {i}: node {node} : sample : {sample[i]}')
                k = int(sample[i]-1)

                parents = list(self.DAG.predecessors(node))

                j = 0     #preinitialization of second index
                if len(parents) != 0:   #Non-empty set of parent nodes
                    siz = [self.DAG.nodes[p_n]['r'] for p_n in parents]
                    x = [sample[self.DAG.nodes[p_n]['i']] for p_n in parents]
                    k_p = np.concatenate(([1], np.cumprod(siz[:-1])))
                    j = int(np.dot(k_p, np.array(x) - 1))


                M[i][j,k] += 1.0


        return M, alpha

    def GG(self,tp):
        for row in tp:
            if self.DAG.has_edge(row[0], row[1]) or self.DAG.has_edge(row[1], row
[0]):
                continue
            else:
                self.DAG.add_edge(row[0],row[1])
                if nx.is_directed_acyclic_graph(self.DAG):
                    pass
                else:
                    self.DAG.remove_edge(row[0],row[1])

    def Init_for_local_search(self):
        ################################## Initialize the grpah randomly

        # self.random_DAG()

        ############Initialize according to mutual information among nodes

        tp = top_pairs(self.inputfilepath,int(self.n*(self.n-1)/4)) #meduim data
set
        # tp = top_pairs(self.inputfilepath,1*int(self.n))  #small data set
```

```
        self.GG(tp)
        ################################
        pass


    def local_search(self):

        k = 0
        Converged_ = False
        while not Converged_:
            k+=1
            y = self.BayesianScore()
            self.BS.append(y)
            ############
            print('Bayesian Score changes during local search:',y)
            ############

            n = self.DAG.number_of_nodes()
            i = random.randint(0,n-1)
            j = (i + random.randint(1,n-1))%n

            if self.DAG.has_edge(self.Nodes[i], self.Nodes[j]):
                yprime = []
                self.DAG.remove_edge(self.Nodes[i], self.Nodes[j])  #remove edge
                yprime.append(self.BayesianScore())

                self.DAG.add_edge(self.Nodes[j], self.Nodes[i])    #add reversed
    edge
                yprime.append(self.BayesianScore())

                #Update
                if yprime[0] > yprime[1] and yprime[0] > y:
                    #Bayesian Score of grpah without edge is greater than reversed
     one and initial network
                    #Go ahead without the edge
                    self.DAG.remove_edge(self.Nodes[j], self.Nodes[i])   #removed
    and no edge now

                    ########################################################
                    #####with Simulated annealing exlore less score structure
                    ########################################################
                    if yprime[1]>y:
                        if np.random.rand() < self.epsilon/k:
                            self.DAG.add_edge(self.Nodes[j], self.Nodes[i])    #
    add reversed edge



                elif yprime[1] > yprime[0] and yprime[1] > y:
                    #Bayesian Score of grpah with reversed one is greater than
    without edge and initial network
```

```
                #Go ahead with revered edge


                ##########################################################
                #####with Simulated annealing exlore less score structure
                ##########################################################
                if yprime[0]>y:
                    if np.random.rand() < self.epsilon/k:
                        self.DAG.remove_edge(self.Nodes[j], self.Nodes[i])
#remove reversed edge

                pass

            elif yprime[1] == yprime[0] and yprime[1] > y:
                r = random.randint(0,1)
                if r ==0:
                    self.DAG.remove_edge(self.Nodes[j], self.Nodes[i])   #
remove

                else:
                    pass

            elif y >= max(yprime):
                #restore to initial
                self.DAG.remove_edge(self.Nodes[j], self.Nodes[i])    #
removed the reversed edge
                self.DAG.add_edge(self.Nodes[i], self.Nodes[j])       # added
back the initial edge


        elif self.DAG.has_edge(self.Nodes[j], self.Nodes[i]):
            continue

        else:
            yprime = []
            self.DAG.add_edge(self.Nodes[i], self.Nodes[j])  #add edge   [0]
            yprime.append(self.BayesianScore())

            self.DAG.remove_edge(self.Nodes[i], self.Nodes[j]) #remove edge

            self.DAG.add_edge(self.Nodes[j], self.Nodes[i])    #reversed edge
 [1]

            yprime.append(self.BayesianScore())

            #Update
            if yprime[0]>yprime[1] and yprime[0] > y:
                #Bayesian Score of grpah added edge is greater than reversed
one and initial network
                #Go ahead with added edge
```

```
                self.DAG.remove_edge(self.Nodes[j], self.Nodes[i])   #remove
back
                self.DAG.add_edge(self.Nodes[i], self.Nodes[j])      #add
normal direc

                ########################################################
                #####with Simulated annealing exlore less score structure
                ########################################################
                if yprime[1]>y:
                    if np.random.rand() < self.epsilon/k:
                        self.DAG.remove_edge(self.Nodes[i], self.Nodes[j])   #
remove back
                        self.DAG.add_edge(self.Nodes[j], self.Nodes[i])    #
reversed edge

            elif yprime[1]>yprime[0] and yprime[1] > y:
                #Bayesian Score of grpah with reversed one is greater than
added edge and initial network
                #Go ahead with revered edge

                ########################################################
                #####with Simulated annealing exlore less score structure
                ########################################################
                if yprime[0]>y:
                    if np.random.rand() < self.epsilon/k:
                        self.DAG.remove_edge(self.Nodes[j], self.Nodes[i])   #
remove back
                        self.DAG.add_edge(self.Nodes[i], self.Nodes[j])    #
reversed edge

                pass

            elif yprime[1] == yprime[0] and yprime[1] > y:
                r = random.randint(0,1)
                if r ==0:
                    self.DAG.remove_edge(self.Nodes[j], self.Nodes[i])  #
remove
                    self.DAG.add_edge(self.Nodes[i], self.Nodes[j])      #addd
 normal direc

                else:
                    pass

            elif y >= max(yprime):
                #restore to initial

                self.DAG.remove_edge(self.Nodes[j], self.Nodes[i])      #
remove

        if len(self.BS) > 100:
```

```
                if abs(y - self.BS[-40]) <= 1e-6:
                    Converged_ = True
                    print(' Local Search converged to Local Optima')



def main():
    if len(sys.argv) != 3:
        raise Exception("usage: python project1.py <infile>.csv <outfile>.gph")

    inputfilename = sys.argv[1]
    outputfilename = sys.argv[2]

    Structure_DAG = StructureLearning(inputfilename, outputfilename)
    # compute(inputfilename, outputfilename)


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

mutual.py file:

```
import numpy as np
import pandas as pd
from scipy.stats import entropy
import networkx as nx

def joint_prob(df, col1, col2):
    joint_dist = pd.crosstab(df[col1], df[col2], normalize=False)  #counts
    the occurance combined
    return joint_dist.values


def marginal_prob(df, col):
    marginal_dist = df[col].value_counts(normalize=False)
    return marginal_dist.values

# Function to compute mutual information I(X; Y)
def mutual_information(df, col1, col2):
    joint = joint_prob(df, col1, col2)
    marginal_x = marginal_prob(df, col1)
    marginal_y = marginal_prob(df, col2)

    # Compute mutual information I(X; Y)
    mi = 0
    for i in range(joint.shape[0]):
        for j in range(joint.shape[1]):
            if joint[i, j] > 0:  # Avoid division by zero
                mi += joint[i, j] * np.log(joint[i, j] / (marginal_x[i] *
    marginal_y[j]))

    return mi

# Function to compute mutual information for each pair of columns
def compute_mutual_info(df):
    columns = df.columns
    # print(columns)
    mi_matrix = pd.DataFrame(index=columns, columns=columns)

    for col1 in columns:
        for col2 in columns:
            if col1 != col2:
                mi_matrix.loc[col1, col2] = mutual_information(df, col1, col2
    )
            else:
                mi_matrix.loc[col1, col2] = np.nan

    return mi_matrix
```

```
def top_pairs(path,k):
    df = pd.read_csv(path)

    mi_matrix = compute_mutual_info(df)
    print(mi_matrix)
    mi_pairs = mi_matrix.stack().reset_index()
    mi_pairs.columns = ['Variable 1', 'Variable 2', 'Mutual Information']


    top_pairs = mi_pairs.sort_values(by='Mutual Information', ascending=False
    ).head(k)



    return top_pairs.to_numpy()
```