Project 1: Bayesian Structure Learning

Deep Dayaramani deep.daya@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

For structure learning in this project, 2 methods of local graph search, better known as hill climbing, were used.

1.1 Traditional Hill Climbing

The first method, was a traditional method of local graph search, which involves starting with a graph and considering all of its neighbors which are one edge away. This involves finding the nearest neighbor considering each combination of nodes and then scoring each of them. Within the method, a combination of nodes was found and then a score was calculated for cases where there was an edge between the nodes, there wasn't an edge and if there was a reverse edge between the nodes. It is a very computationally heavy method of finding a Bayesian Structure and hence it was used for the small and medium data sets provided along with getting a starting structure for the large dataset.

1.2 Greedy Hill Climbing

The second method is a greedy implementation of hill climbing, where instead of considering each node combination, we randomly choose 2 nodes and then consider the score of the graph. If the score exceeds the current score, we choose that graph and repeat the process. This method of hill climbing ignores a lot of structures which may be better candidates in the hopes of reaching a local maxima or even the global maxima faster!

1.3 Bayesian Score

The score used in this algorithm is the Bayesian Score of a graph given by

$$\log(P(G|D)) = log(P(G)) + \sum_{i=1}^{n}(\sum_{j=1}^{q_i}\log(\frac{\Gamma(\alpha_{ij0})}{\Gamma(\alpha_{ij0}+m_{ij0})} + \sum_{k=1}^{r_i}\frac{\Gamma(\alpha_{ijk}+m_{ijk})}{\Gamma(\alpha_{ijk})} \quad (1)$$

1.4 Shortcomings and Solutions

To reduce computational time to calculate this score, an updater function for the count matrix, a matrix which keeps track of parental instantiations and counts of each node within the data. This helped a lot for the large data set, as the program didn't have to re-calculate the time intensive process of making the count matrix again and again.

For the large dataset, the greedy implementation of the hill climbing program were applied only after the traditional method failed to converge after 12 hours of implementation.

On the other hand the greedy implementation passed the most successful graph of the traditional implementation in 45 minutes.

Looking at the scores which other people have gotten, it seems that the large dataset arrived at a local maxima, which was close to the global maxima. This can be further corrected by random restarts which was also implemented into the algorithm.

2. Graphs

2.1 Small Dataset

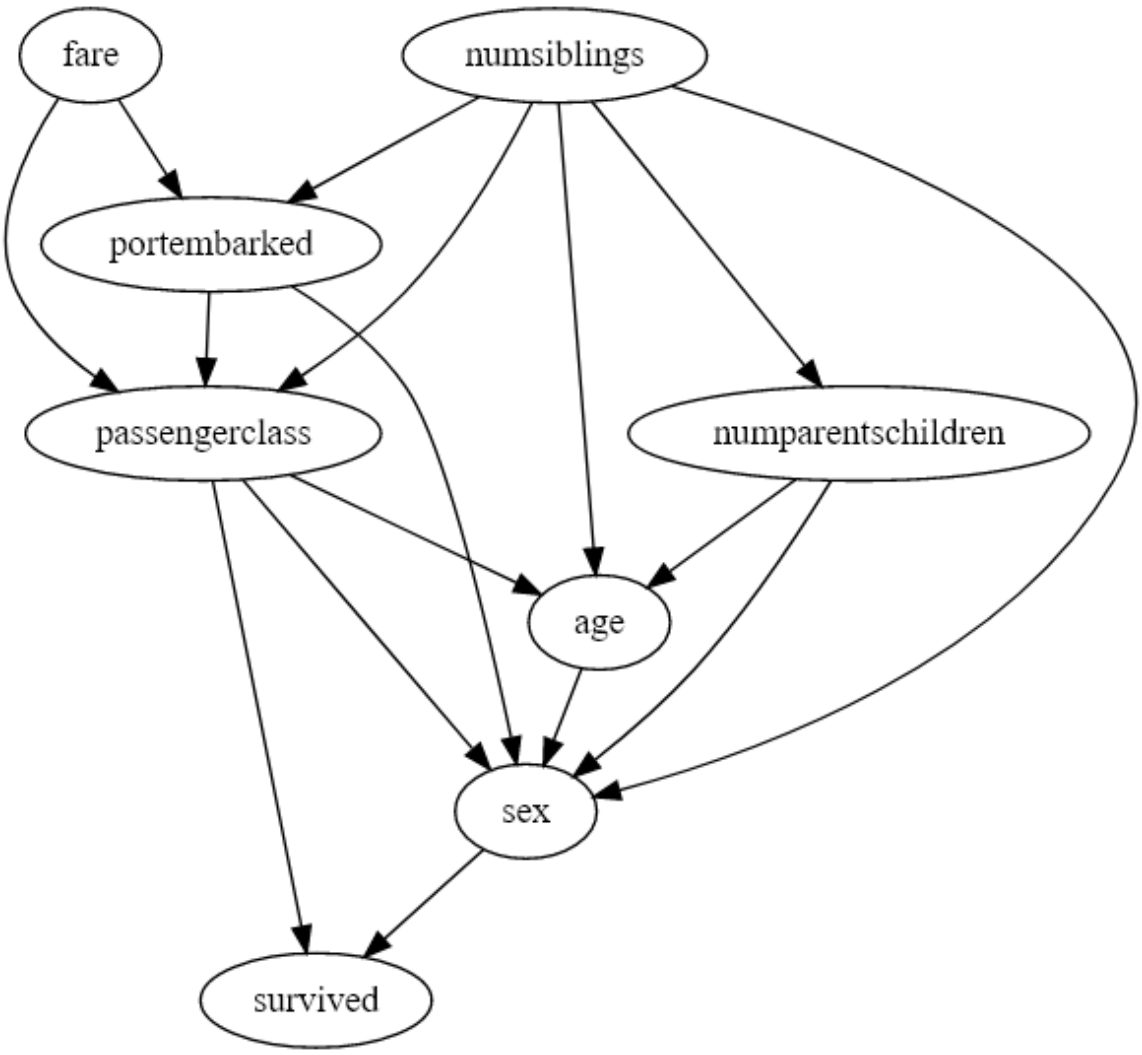

Figure 1: Small Dataset Graph. Score = -3799.4840101250225 & Time = 43.15s

2.2 Medium Dataset

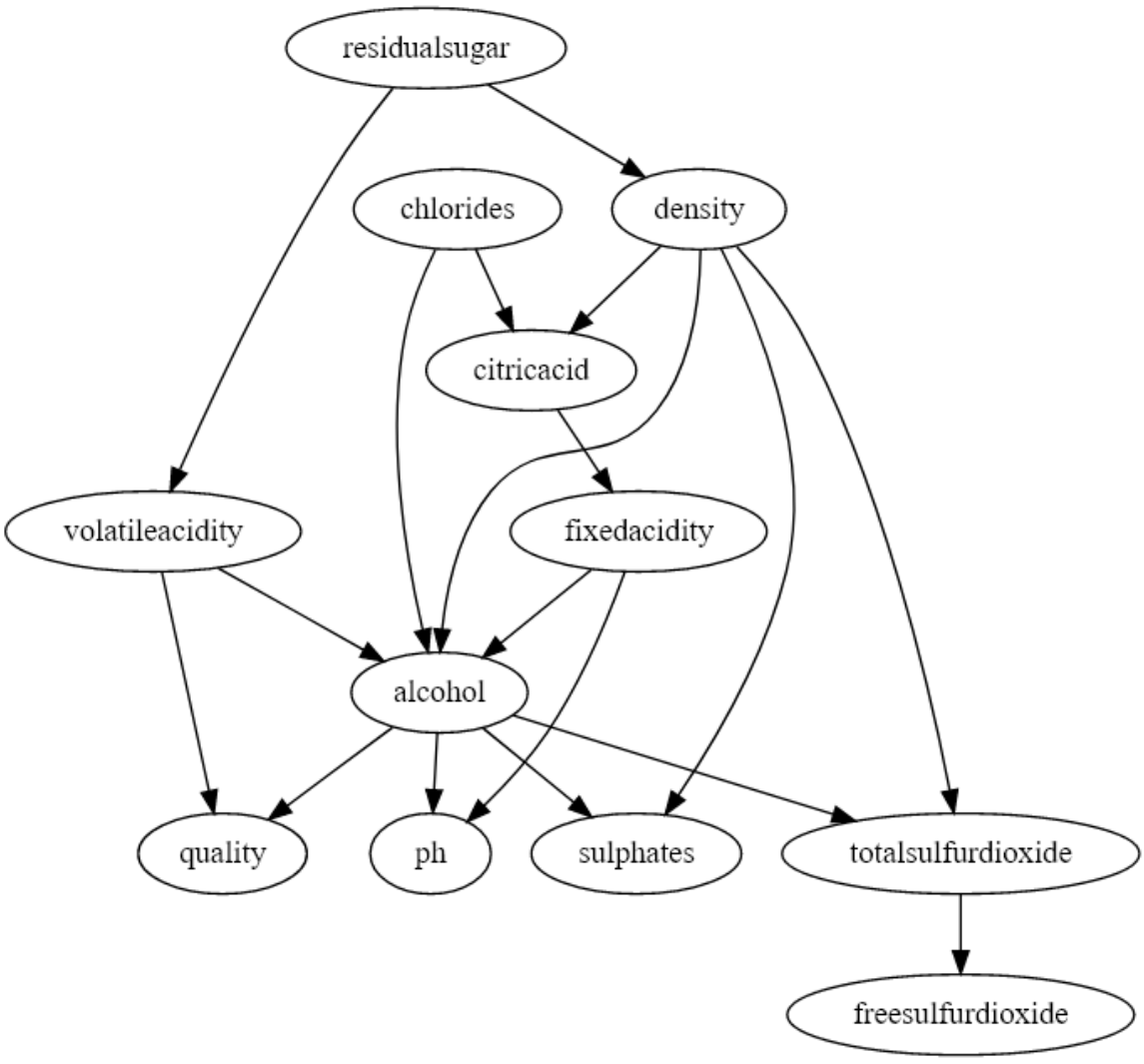

Figure 2: Medium Dataset Graph. Score = -41957.562988812075 & Time = 585.41s

2.3 Large Dataset

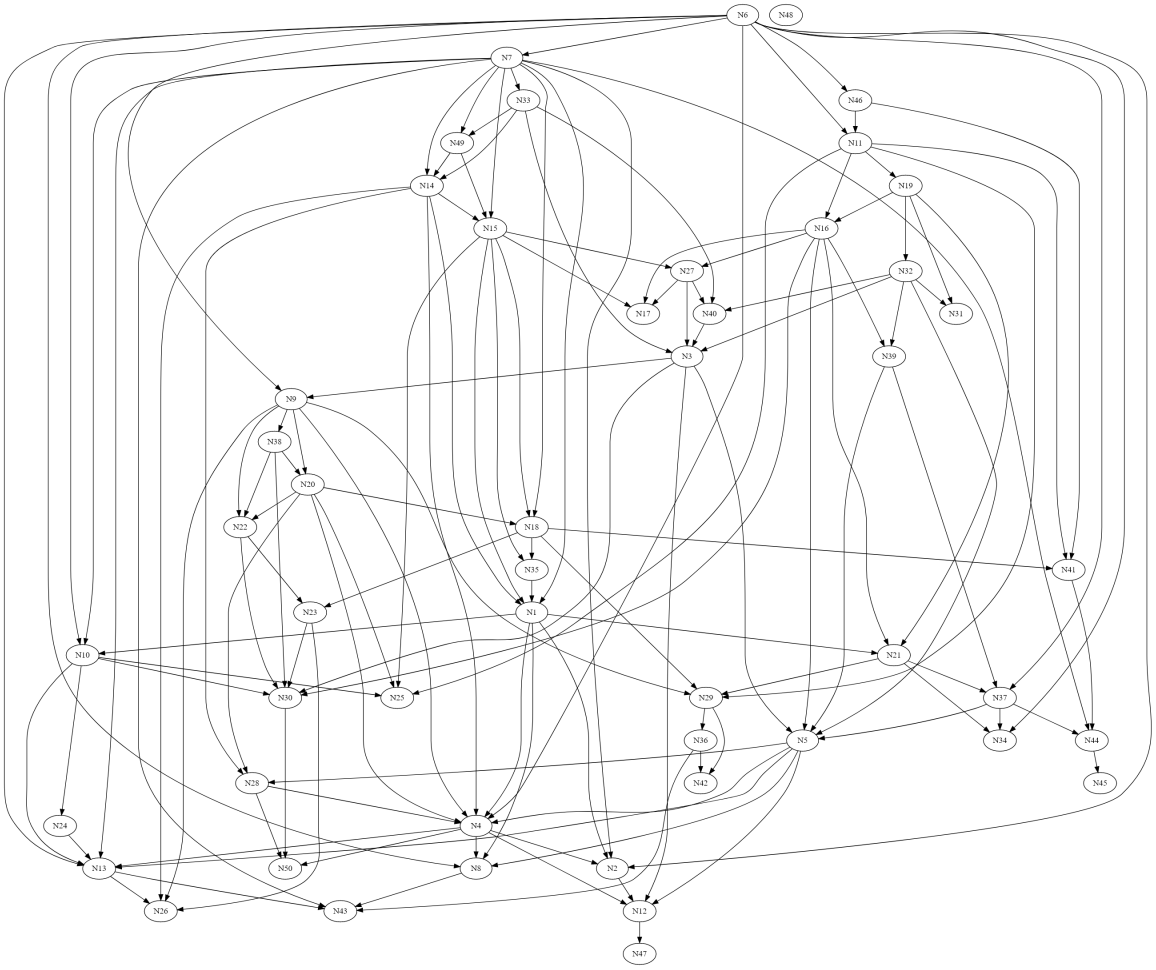

Figure 3: Large Dataset Graph. Score = -430087.3098036853 & Time = 5.14 hours.

3. Code

```
import random
import sys
import networkx as nx
import numpy as np
from scipy.special import loggamma
from random import randint
import math
from copy import deepcopy as clone
import pandas as pd
import time
import matplotlib.pyplot as plt


def write_gph(dag, filename):
    """ Writes a graph's edges to a file."""
    with open(filename, 'w') as f:
        for edge in dag.edges():
            f.write("{}, {}\n".format(edge[0], edge[1]))


global visited_states
global M_best
visited_states = set()

def compute(infile, outfile, method="simple", *args):
    """Finds the best graph structure given data and a method for doing the
    local graph search."""
    # Start Initial Graph and get data
    data = data_from_file(infile)
    G = initial_graph(data)

    #For Large DataSet, load in a previously reached graph and use it for
    generating a random graph
    if infile == "data/large.csv":
        f = open("largey.gph", 'r')
        edges = [f.read()]
        edges = edges[0].split("\n")
        epoch = []
        for x in edges:
            epoch.append(x.split(" "))
        for i in range(len(epoch)):
            epoch[i] = [epoch[i][0].replace(",",""), epoch[i][1]]
        G.add_edges_from(epoch)
        f = open("largey.score","r")
        current_score = float(f.read())
        G = randomized_new_graph(G, data)
        current_score = bayesian_score(G, data)
    else:
```

```
        # Else we generate a random start point
        G = randomized_new_graph(G, data)
        current_score = bayesian_score(G, data)
    # Establish best_scores and best_next_neighbor
    best_score = current_score + 0
    best_next_neighbor = G
    graphs_changed = 0

    # If we feed in extra args for Randomized Restarts
    if len(args) != 0:
        restarts = args[0]
    iterations = 0
    M = None
    first_time = time.time()
    # Check possible next states
    while True:
        iterations += 1
        time_start = time.time()
        print("Iteration {0}".format(iterations))
        # Find possible next states
        if infile == "data/large.csv":
            possible_next_states = possible_states_greedy(G, data, M)
        else:
            possible_next_states = possible_states_traditional(G, data, M)
        print("----%s seconds----" % (time.time() - time_start))

        # Check length of next states and if 0, we have converged to a local\
    global maxima
        if len(possible_next_states) == 0:
            if method == "simple":
                print("Total Program takes %s seconds" % (time.time() -
    first_time))
                break
            elif method == "random_restart":
                if bayesian_score(best_next_neighbor, data) < bayesian_score(
    G, data):
                    best_next_neighbor = G
                    best_score = current_score
                if restarts > 0:
                    G = randomized_new_graph(G, data)
                    restarts -= 1
                else:
                    print("Total Program takes %s seconds" % (time.time() -
    first_time))
                    break
        else:
            best_next_neighbor = list(possible_next_states.keys())[0]
            best_score = possible_next_states[best_next_neighbor]
            G = best_next_neighbor
```

```
            # Regularly save Graph to save progress in case of errors
            write_gph(G, outfile)

            # We update M instead of creating a new one so as to reduce
    computational load and time
            M = M_best
            graphs_changed += 1
        print("Structures Changed: {0}".format(graphs_changed))
        print("Best Score: {0}".format(best_score))
        print("Difference in Scores: {0}".format(best_score - current_score))

    # Save graph, draw Network and save dot file
    write_gph(G, outfile)
    nx.draw(G, with_labels=True)
    nx.nx_pydot.write_dot(G, 'outfile.txt')
    plt.show()

    # Check if any cycles were created in the final result
    assert len(list(nx.simple_cycles(G))) == 0


def fully_random_dag(G, data):
    """
    This function generates a fully connected random DAG- Directed Acyclic
    Graph G with N(N-1)/2 edges where N
    is the number of nodes
    :param G: Graph input. Simple DiGraph with no edges
    :param data: data from csv file to generate edges
    :return: Fully Connected DAG
    """

    # Randomize columns
    copy = list(data.columns)
    random.shuffle(copy)
    list_of_shuffled_nodes = copy
    set_of_nodes = set(data.columns)

    # Add edges
    for i, col in enumerate(list_of_shuffled_nodes):
        set_of_nodes.remove(col)
        G.add_edges_from([(col, k) for k in set_of_nodes])

    # Check for cycles
    assert (len(list(nx.simple_cycles(G)))) == 0
    return G


def randomized_new_graph(G, data):
    """
```

```
    Generates a random new start point, taking the current Graph G and data,
    and adding and removing a random
    amount of edges
    """
    # To make sure no cycles are created
    repeat = False
    while not repeat:
        G_copy = clone(G)

        # Randomly choose edges to add/remove
        for i in range(randint(1, len(list(G.nodes)))):
            # Randomly choose origin and destination nodes.
            org = data.columns[randint(0, len(data.columns)-1)]
            setBoi = set(list(G.neighbors(org))).union(set(org))
            dest_set = set(data.columns) - setBoi
            dest = list(dest_set)[randint(0, len(dest_set)-1)]

            # Check if edge exists, and if so remove it, or else add an edge
            if G_copy.has_edge(org, dest) or G_copy.has_edge(dest, org):
                if G_copy.has_edge(org, dest):
                    G_copy.remove_edge(org, dest)
                elif G_copy.has_edge(dest, org):
                    G_copy.remove_edge(dest, org)
            else:
                G_copy.add_edge(org, dest)
            repeat = len(list(nx.simple_cycles(G_copy))) == 0
    return G_copy


def possible_states_traditional(G, data, M_temp = None):
    """ Generates the Best Possible State using Traditional Hill Climbing for
     the graph G, data,
    and a count matrix if it has already been calculated"""
    possible_next_states = {}
    M_current = statistics(G, data) if M_temp is None else M_temp

    # Find score
    current_score = bayesian_score(G, data, M_current)
    highest_score = current_score

    global best_graph
    best_graph = None

    global M_best
    M_best = clone(M_current)

    copy = list(data.columns)
    for i, col in enumerate(copy):
        print(col)
        start_time = time.time()
```

```
        other_columns = list(data.columns.delete(i))
        for other_col in other_columns:
            G_copy = clone(G)
            G_copy_2 = clone(G)
            edge_present = False
            reverse_edge = False

            # Check for edge, if present remove or else add.
            if G_copy.has_edge(col, other_col) or G_copy.has_edge(other_col,
    col):
                if G_copy.has_edge(col, other_col):
                    G_copy.remove_edge(col, other_col)
                elif G_copy.has_edge(other_col, col):
                    G_copy.remove_edge(other_col, col)
                    reverse_edge = True
            else:
                G_copy.add_edge(col, other_col)
                G_copy_2.add_edge(other_col, col)
                edge_present = True

            # If move creates cycle, then score as -Inf or else calculate
    score
            if len(list(nx.simple_cycles(G_copy))) != 0:
                new_score = -math.inf
            else:
                if reverse_edge:
                    M_1 = clone(M_current)
                    M_1 = M_updater(G_copy, data, M_1, col)
                else:
                    M_1 = clone(M_current)
                    M_1 = M_updater(G_copy, data, M_1, other_col)
                new_score = bayesian_score(G_copy, data, M_1)

            if edge_present:
                if len(list(nx.simple_cycles(G_copy_2))) != 0:
                    new_score_2 = -math.inf
                else:
                    M_2 = clone(M_current)
                    M_2 = M_updater(G_copy_2, data, M_2, col)
                    new_score_2 = bayesian_score(G_copy_2, data, M_2)

                # Checks if score is higher and if graph is not visited
    already
                if (new_score_2 > current_score) & (tuple(G_copy_2.edges) not
     in visited_states):
                    visited_states.add(tuple(G_copy_2.edges))
                    if new_score_2 > highest_score:
                        highest_score = new_score_2
                        M_best = clone(M_2)
                        best_graph = G_copy_2
```

```
            if (new_score > current_score) & (tuple(G_copy.edges) not in
    visited_states):
                visited_states.add(tuple(G_copy.edges))
                if new_score > highest_score:
                    highest_score = new_score
                    M_best = clone(M_1)
                    best_graph = G_copy
        print("-----%s seconds for 1 Col-----"% (time.time() - start_time))

    # Stores the best graph and score in a dictionary
    if best_graph is not None:
        possible_next_states[best_graph] = highest_score
    return possible_next_states


def possible_states_greedy(G, data, M_temp = None):
    """Generates best possible state using Greedy Random Hill Climbing for
    Graph G and data."""
    possible_next_states = {}
    M_current = statistics(G, data) if M_temp is None else M_temp
    current_score = bayesian_score(G, data, M_current)
    highest_score = current_score
    best_graph = None
    M_best = clone(M_current)
    copy = list(data.columns)
    # Shuffle Columns
    random.shuffle(copy)
    for i, col in enumerate(copy):
        print(col)
        start_time = time.time()
        other_columns = list(data.columns.delete(i))
        # Boolean for ending iteration
        endIt = False
        # Shuffle Other Columns
        random.shuffle(other_columns)
        for other_col in other_columns:
            G_copy = clone(G)
            G_copy_2 = clone(G)
            edge_present = False
            reverse_edge = False
            if G_copy.has_edge(col, other_col) or G_copy.has_edge(other_col,
    col):
                if G_copy.has_edge(col, other_col):
                    G_copy.remove_edge(col, other_col)
                elif G_copy.has_edge(other_col, col):
                    G_copy.remove_edge(other_col, col)
                    reverse_edge = True
            else:
                G_copy.add_edge(col, other_col)
                G_copy_2.add_edge(other_col, col)
```

```
                edge_present = True
            if len(list(nx.simple_cycles(G_copy))) != 0:
                new_score = -math.inf
            else:
                if reverse_edge:
                    M_1 = clone(M_current)
                    M_1 = M_updater(G_copy, data, M_1, col)
                else:
                    M_1 = clone(M_current)
                    M_1 = M_updater(G_copy, data, M_1, other_col)
                new_score = bayesian_score(G_copy, data, M_1)
            if edge_present:
                if len(list(nx.simple_cycles(G_copy_2))) != 0:
                    new_score_2 = -math.inf
                else:
                    M_2 = clone(M_current)
                    M_2 = M_updater(G_copy_2, data, M_2, col)
                    new_score_2 = bayesian_score(G_copy_2, data, M_2)
                if (new_score_2 > current_score) & (tuple(G_copy_2.edges) not
     in visited_states):
                    visited_states.add(tuple(G_copy_2.edges))
                    if new_score_2 > highest_score:
                        highest_score = new_score_2
                        M_best = clone(M_2)
                        best_graph = G_copy_2
                        # Breaks the loop
                        endIt = True
                        break
            if (new_score > current_score) & (tuple(G_copy.edges) not in
    visited_states):
                visited_states.add(tuple(G_copy.edges))
                if new_score > highest_score:
                    highest_score = new_score
                    M_best = clone(M_1)
                    best_graph = G_copy
                    endIt = True
                    break
        if endIt:
            break
        print("-----%s seconds for 1 Col-----"% (time.time() - start_time))
    if best_graph is not None:
        possible_next_states[best_graph] = highest_score
    return possible_next_states
```

```
def data_from_file(infile):
    """ Gets data from provided csv in INFILE"""
    data = pd.read_csv(infile)
    return data
```

```
def initial_graph(data, Gtemp=None):
    """ Generates an Initial DiGraph and if provided adds edges in the graph.
    """
    G = nx.DiGraph()
    if Gtemp is not None:
        G.add_nodes_from(list(Gtemp.nodes))
        G.add_edges_from(list(Gtemp.edges))
    else:
        G.add_nodes_from(data.columns)
    return G


def statistics(G, data):
    """Finds the count matrices for the graph G and data. """
    n = len(data)
    r = {cols: max(data[cols].unique()) for cols in data.columns}
    q = {cols: int(np.prod([r[j] for j in list(G.predecessors(cols))])) for
    cols in data.columns}
    M = {i: {} for i in data.columns}
    for i in range(n):
        # print(i)
        for cols in data.columns:
            k = data.loc[i, cols] - 1
            parents = list(G.predecessors(cols))
            parent_vals = [data.loc[i, parent] for parent in parents]
            if len(parents) == 0:
                j = 1
            else:
                j = ''.join(str(val) for val in parent_vals)
            if j not in M[cols].keys():
                M[cols][j] = [0]*r[cols]
                M[cols][j][k] += 1
            else:
                M[cols][j][k] += 1
    len_M_i = {i:len(M[i]) for i in data.columns}

    # Checks if length of the matrices are right or appends them with zeros
    for i in data.columns:
        if len_M_i[i] != q[i]:
            for l in range(q[i]-len_M_i[i]):
                M[i]['{0}{1}'.format(i,l)] = [0] * r[i]
    return M

def M_updater(G, data, M_old, destination):
    """ Updates the count matrices using an older M, and the destination node
     at which the edge was
    removed or added in the given Graph G and data."""
    r = {cols: max(data[cols].unique()) for cols in data.columns}
```

```
    q = {cols: int(np.prod([r[j] for j in list(G.predecessors(cols))])) for
    cols in data.columns}
    M_old[destination] = {}
    for i in range(len(data)):
        k = data.loc[i, destination] - 1
        parents = list(G.predecessors(destination))
        parent_vals = [data.loc[i, parent] for parent in parents]
        if len(parents) == 0:
            j = 1
        else:
            j = ''.join(str(val) for val in parent_vals)
        if j not in M_old[destination].keys():
            M_old[destination][j] = [0] * r[destination]
            M_old[destination][j][k] += 1
        else:
            M_old[destination][j][k] += 1
    len_M_i = len(M_old[destination])
    if len_M_i != q[destination]:
        for l in range(q[destination] - len_M_i):
            M_old[destination]['{0}{1}'.format(destination, l)] = [0] * r[
    destination]
    return M_old


def bayesian_score_component(M, alpha):
    """ Helper function to calculate the sum of Loggamma functions for
    calculate the score."""
    M_values= [M[i] for i in M.keys()]
    p = np.sum(loggamma(alpha + M_values))
    p -= np.sum(loggamma(alpha))
    p += np.sum(loggamma(np.sum(alpha, axis=1)))
    p -= np.sum(loggamma(np.sum(alpha, axis=1) + np.sum(M_values,axis=1)))
    return p

\documentclass[twoside,11pt]{article}

\usepackage{aa228-jmlr2e}
\usepackage{listings}
\begin{document}

def bayesian_score(G, data, M_o = None):
    """Calculates the Bayesian score for the given graph G and data. """
    M = statistics(G, data) if M_o is None else M_o
    # print("fin2")
    alpha = prior(G, data)
    # print("Score")
    L = [bayesian_score_component(M[i], alpha[i]) for i in data.columns]
    return sum(L)
```

```
def prior(G, data):
    """Calculates the uniform prior for the given data and graph G."""
    n = len(data.columns)
    r = {cols: max(data[cols].unique()) for cols in data.columns}
    q = {cols: int(np.prod([r[j] for j in list(G.predecessors(cols))])) for
    cols in data.columns}
    return {i : np.ones((q[i], r[i])) for i in data.columns}


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]
    compute(inputfilename, outputfilename)


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