Project 1: Bayesian Structure Learning

Kapil Dheeriya KAPD@STANFORD.EDU
AA228/CS238, Stanford University

1. Algorithm Description

I used a hybrid of a greedy K2 and an opportunistic local hill-climbing search algorithm to find an optimal structure of a directed acyclic graph that represents a bayesian network that fits the given datasets. I assumed a uniform dirichelet prior: $\alpha_{ijk} = 1$.

I first computed a total bayesian score by computing local bayesian scores at each node then summing them. I did not use the count/pseudocount matrices approach for computational efficiency. I used a classic K2 approach with an added L2 regularization term applied to the local bayesian score. This term is proportional to the nodes' number of parents squared to penalize graph complexity. My K2 algorithm also capped the maximum number of parents for a node at three. The K2 algorithm works as follows: starting from a graph with no edges, the given variables are repeatedly iterated over in reverse order as given (because survived in small.csv and quality in medium.csv should be at the bottom of the graph). First, we choose a node to be a child, then we choose a valid parent from the remaining nodes. If when we add an edge from parent to child, the local bayesian score of the child increases, we store that parent as the best parent. After looping over all possible parents and finding the best parent that increases the score the most, we add an edge from best parent to child. This process repeats for all nodes. In the end we get a directed acyclic graph on which we can perform an opportunistic hill climb to further optimize the score.

The opportunistic hill climb repeatedly chooses at random a valid child parent pair from the list of nodes. If there is already an edge between the pair, the edge is removed. If there is not an edge, it adds an edge. If this removal or addition improves the local bayesian score of the child, the change is kept. This process is repeated 200 times. More iterations can be used, but for the sake of speed I did not add more. This additional local search did slightly improve the final score.

My algorithm running on the small dataset executed within one second. Running it on the medium dataset took around 15 seconds, and the large dataset took around 2 minutes.

2. Graphs

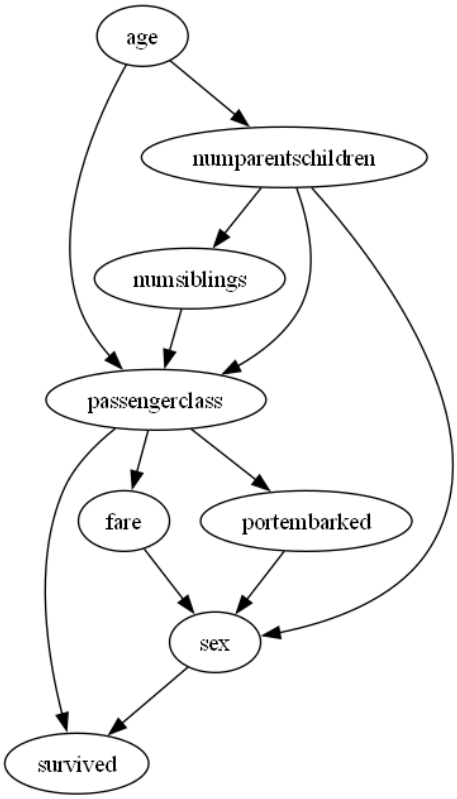

Figure 1: Small graph.

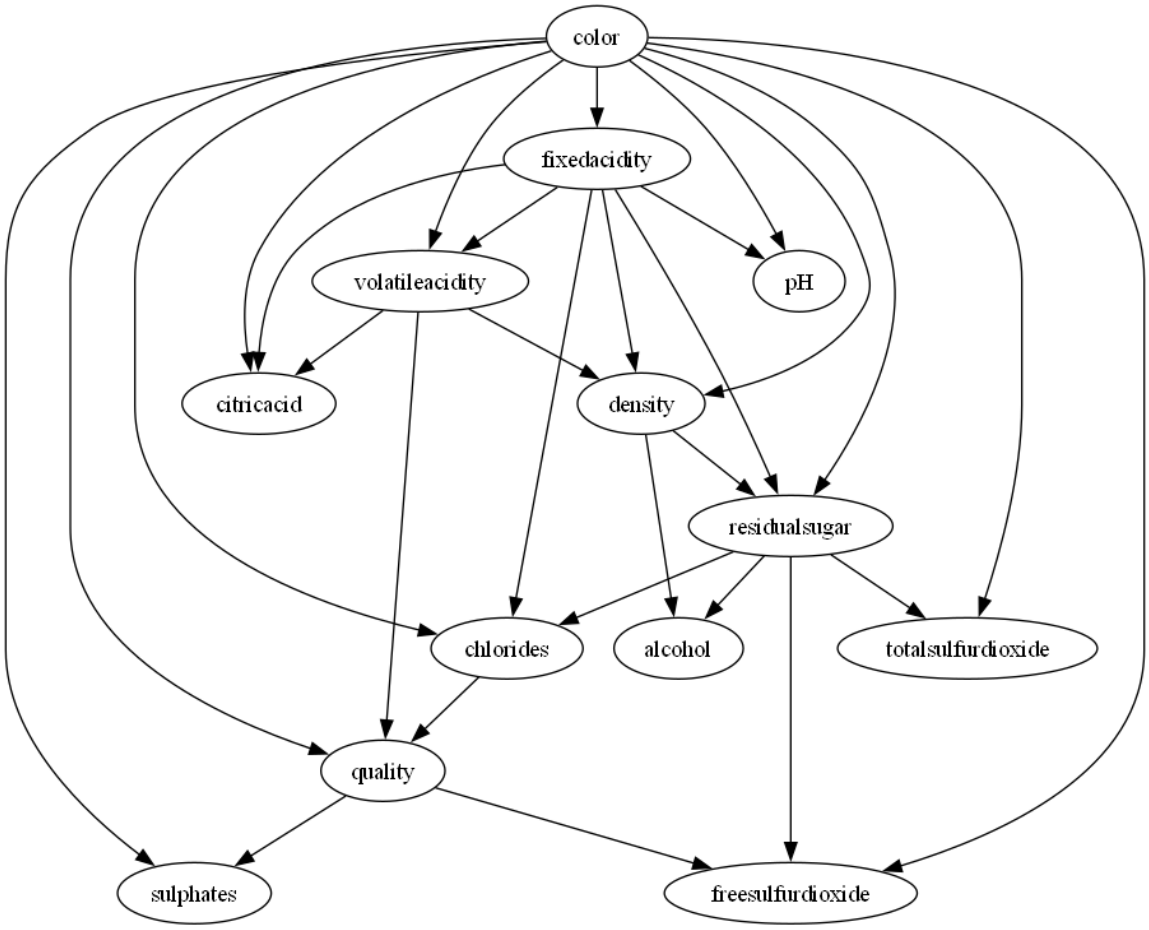

Figure 2: Medium graph.

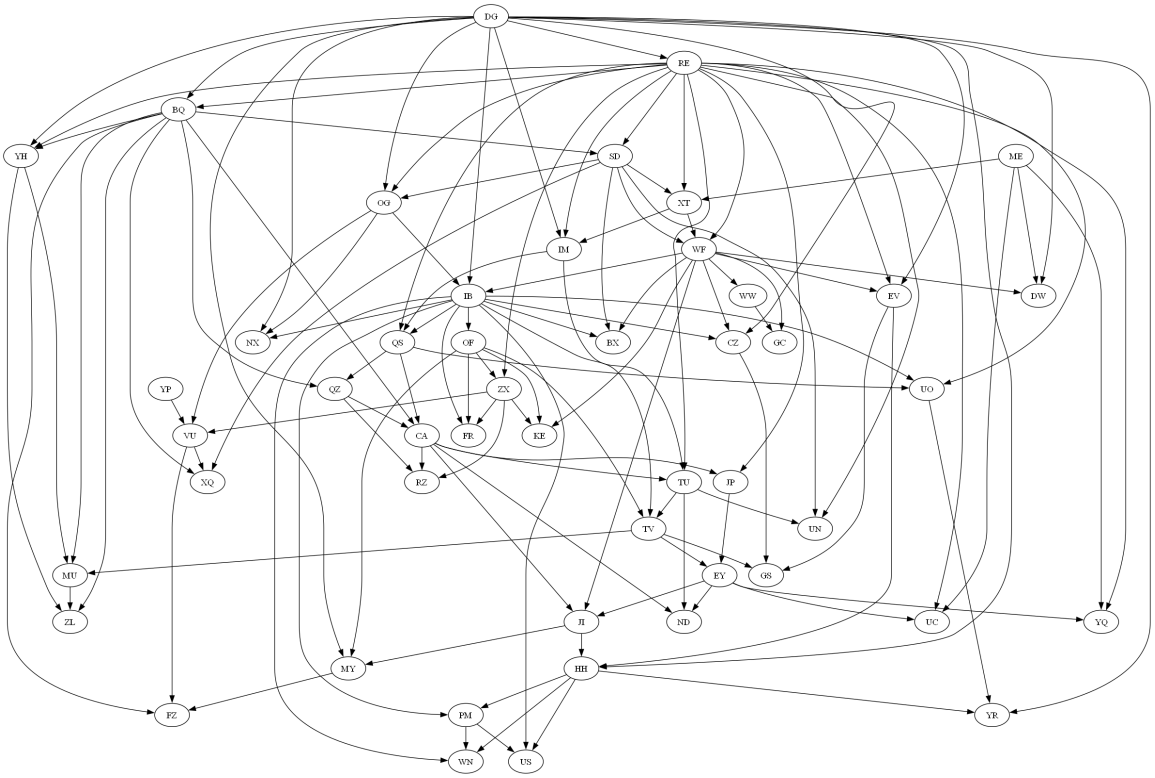

Figure 3: Large graph.

3. Code

```
import sys
import math
import random
import numpy as np
import pandas as pd
import networkx as nx
from itertools import product
from scipy.special import gammaln



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

def read_gph(infile):
    with open(infile, 'r') as file:
        return nx.read_edgelist(file, delimiter=',', create_using=nx.DiGraph
    (), nodetype=str)

def write_dot_file(dag, filename):
    with open(filename, 'w') as f:
        f.write("digraph G {\n")
        for edge in dag.edges():
            f.write(f'    "{edge[0]}" -> "{edge[1]}";\n')
        f.write("}\n")


#Compute bayesian score at the given node
def node_bayesian_score(dag, xi, data, reg_factor=0):
    score = 0
    #Number of instantiations of xi
    ri = len(np.unique(data[xi]))


    parents = list(dag.predecessors(xi))
    #regularization (penalty to score based on num parents)
    if not reg_factor: score -= reg_factor * (len(parents))**2

    #No parents gives a simple calculation that does not require summing
    zeros
    if not len(parents):
        #assuming uniform prior, all a_ijk = 1, so sum(a_ij0) over range(ri)
    = ri
        a_i10 = ri
        #The sum of the counts of each instantiation of xi is the number of
    samples
```

```
        m_i10 = len(data)
        m_i1k = data[xi].value_counts()
        score += gammaln(a_i10) - gammaln(a_i10 + m_i10) + sum(gammaln(1 +
    m_i1k))
        return score

    #Cartesian product of each parent's instantiations giving a list of all
    possible combos of parent instantiations
    parent_inst_list = product(*[data[parent].unique() for parent in parents
    ])
    #Loop over all possible sets of parent values
    for parent_inst in parent_inst_list:
        #select out the data consistent with the current set of parent values
        parent_data = data
        for j in range(len(parents)):
            is_consistent = parent_data[parents[j]] == parent_inst[j]
            parent_data = parent_data[is_consistent]
        #Count the number of occurences of observations consistent with
    current parent vals
        m_ijk = parent_data[xi].value_counts()
        m_ij0 = len(parent_data)
        score += gammaln(ri) - gammaln(ri + m_ij0) + sum(gammaln(1 + m_ijk))

    return score

def total_bayesian_score(dag, data):
    return sum(node_bayesian_score(dag, xi, data) for xi in list(dag.nodes))


def K2_search(dag, data, max_parents=3, reg_const= 5, max_iter = 10, tol=10e
    -2):
    x = list(dag.nodes)
    x.reverse()
    scores = {xi : node_bayesian_score(dag, xi, data, reg_const) for xi in x}
    bScore = sum(scores.values())

    delta = 1.0 + tol # to ensure while loop starts
    counter = 0
    foundBetterGraph = True
    cur_dag = dag.copy()

    while foundBetterGraph and delta >= tol and counter <= max_iter:
        foundBetterGraph = False
        oldScore = bScore

        for child in x:
            cur_score = scores[child]
            new_score = cur_score
            best_parent = None
```

```
            if len(list(cur_dag.predecessors(child))) >= max_parents:
    continue

            for parent in x:
                if not dag.has_edge(parent, child) and parent != child:
                    dag_candidate = cur_dag.copy()
                    dag_candidate.add_edge(parent, child)
                    if not nx.is_directed_acyclic_graph(dag_candidate):
    continue

                    candidate_score = node_bayesian_score(dag_candidate,
    child, data, reg_const)
                    if candidate_score > cur_score:
                        new_score = candidate_score
                        best_parent = parent

            if best_parent != None:
                cur_dag.add_edge(best_parent, child)
                bScore += new_score - cur_score
                scores[child] = new_score
                foundBetterGraph = True

        delta = bScore - oldScore
        counter += 1

    print(f'iterations: {counter}, delta: {delta}')
    return cur_dag


def local_directed_graph_search(dag, data, max_parents=3, reg_const = 5,
    max_iter=200):
    nodes = list(dag.nodes)
    n = len(dag)
    cur_dag = dag.copy()

    for k in range(max_iter):
        i = random.randint(0, n - 1)
        j = (i + random.randint(1, n - 1)) % n
        vars = data.columns
        child = vars[j]
        parent = vars[i]
        y = node_bayesian_score(cur_dag, child, data, reg_const)

        new_dag = cur_dag.copy()
        if cur_dag.has_edge(parent, child):
            new_dag.remove_edge(parent, child)

        elif len(list(new_dag.predecessors(child))) < max_parents:
            new_dag.add_edge(parent, child)
```

```
            if not nx.is_directed_acyclic_graph(new_dag): continue
            y_new = node_bayesian_score(new_dag, child, data, reg_const)

            if y_new > y:
                cur_dag = new_dag

    return cur_dag


def compute(infile, gph_outfile, dot_outfile):
    data = pd.read_csv(infile)
    dag = nx.DiGraph()
    dag.add_nodes_from(data.columns)


    K2_dag = K2_search(dag, data)
    print(f'K2 Total Bayesian Score {total_bayesian_score(K2_dag, data)}')
    #learned_dag = K2_search(dag, data)
    loc_dag = local_directed_graph_search(K2_dag, data)
    # Compute total score of learned DAG
    total_score = total_bayesian_score(loc_dag, data)
    print(f"K2+Local Total Bayesian Score: {total_score}")
    write_gph(loc_dag, gph_outfile)
    write_dot_file(loc_dag, dot_outfile)

def main():
    if len(sys.argv) != 4:
        raise Exception("usage: python project1.py <infile>.csv <outfile>.gph
     <dotoutfile>.dot")

    inputfilename = sys.argv[1]
    outputfilename = sys.argv[2]
    dot_outputfilename = sys.argv[3]
    compute(inputfilename, outputfilename, dot_outputfilename)


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