Project 1: Bayesian Structure Learning

Brian Dobkowski bdobkows@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

To find an optimal graph for each of the three provided datasets, a combination of the K2 Search and Opportunistic Local Search was used.

K2 Search begins with an ordering of the nodes in the graph, and iterates through them one by one. At each node, the algorithm greedily adds all other nodes as parents (following the ordering initially given to the algorithm) provided that adding each edge improves the Bayesian score.

Opportunistic Local Search begins with an initial graph and randomly generates a neighboring graph. If the neighboring graph is non-cyclic and improves the Bayesian score, the neighboring graph replaces the original graph, and the process is completed for a given number of iterations.

The approach taken in this design involves running a number of K2 search algorithms with different random initializations (i.e., ordering of nodes). After the best-scoring K2 algorithm is found, that graph is given to Opportunistic Local Search as an initial graph, and OLS runs for more iterations to try to improve the best K2 graph. Of course, the graph that OLS finds optimal must be at least as high-scoring as the best K2 graph. Initial parameter tunings involved running the K2 Search algorithm for 10 iterations, and running the Opportunistic Local Search algorithm for 1000 iterations.

Initially, doing 10 iterations of K2 Search (in series) was implemented, with no limit on the number of parents for each node. After the optimization algorithms ran on the large dataset for over 17 hours without coming near completion, it was clear some changes needed to be made. To improve the time situation, I reduced the number of K2 iterations to 8, and used all 8 CPU cores with Python multiprocessing to run these in parallel rather than series. I also put a cap (5) on the number of potential parents for each node. Unfortunately, the assignment of these parents depends heavily upon the ordering that is originally drawn at the start of the search algorithm. Also, the number of iterations for Opportunistic Local Search was reduced from 1000 to 400.

2. Algorithm Run Time

Dataset	Runtime (hh:mm::ss)
Small	00:01:08
Medium	00:12:33
Large	04:29:27

3. Graphs

See graphs below:

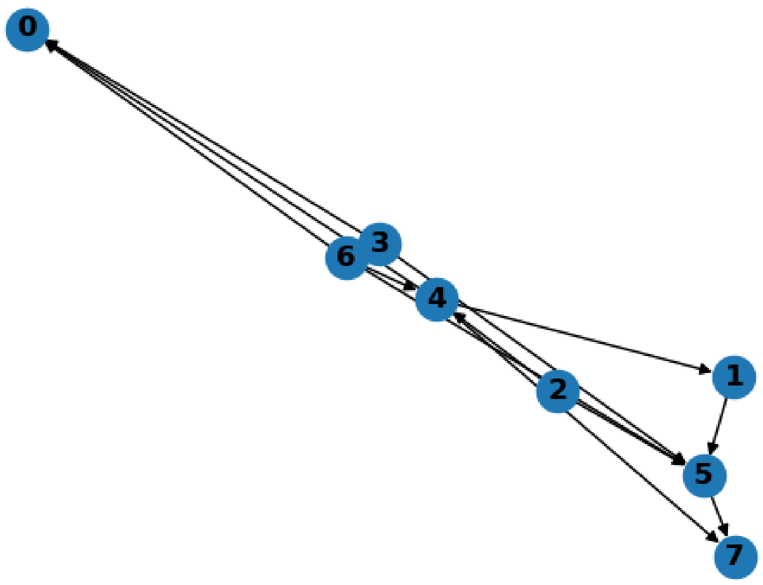

Figure 1: Solution for Small Graph

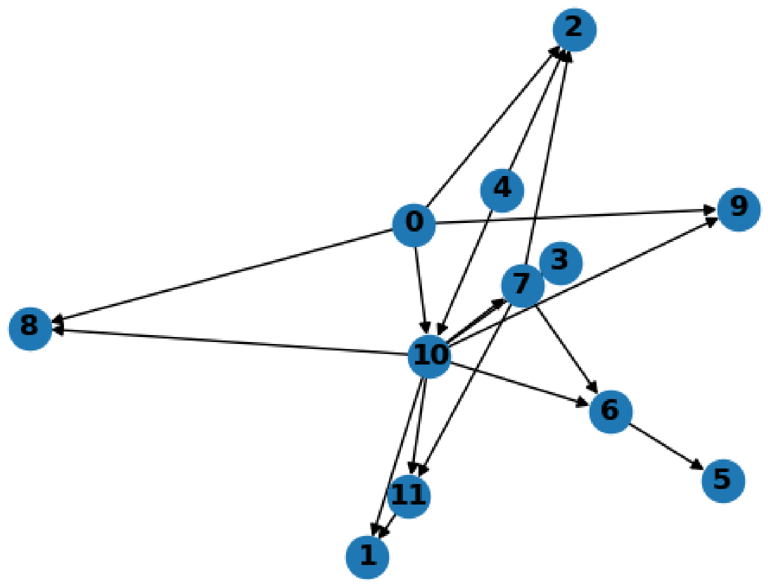

Figure 2: Solution for Medium Graph

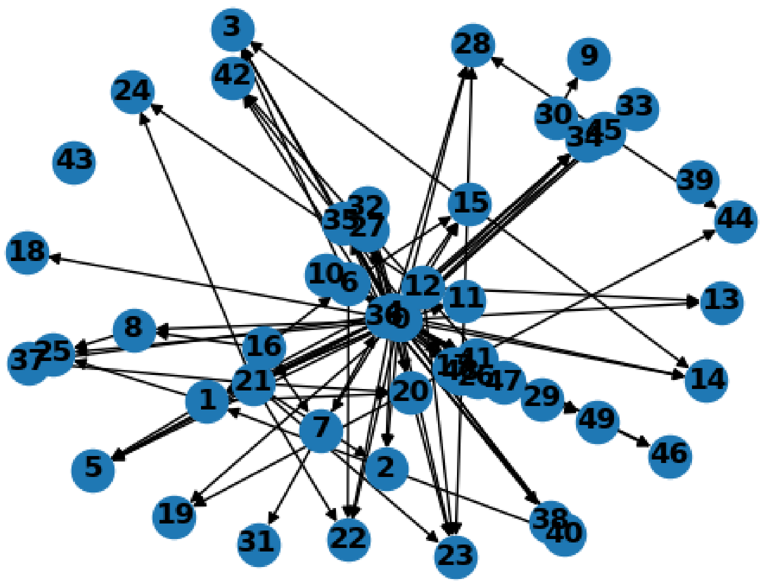

Figure 3: Solution for Large Graph

4. Code

```
#!/usr/bin/env python3

import sys
import numpy as np
import pandas as pd
import networkx as nx
from matplotlib import pyplot as plt
import scipy.special as sp
import random
from tqdm import tqdm
import datetime
from multiprocessing import Process, Manager

_K2_ITER_  = 8 # Number of iterations for K2 Search algorithm
_OLS_ITER_ = 400 # Number of iterations for Opportunistic Local Search

# Random variable class
class Variable:
    def __init__(self, values):
        self.values = values

# K2 Search Implementation
class K2Search:
    def __init__(self, variables, idx2names):
        self.variables = variables
        self.ordering  = random.sample(list(variables.keys()), len(variables.
    keys()))
        self.idx2names = idx2names
        self.bs        = -np.inf
        self.graph     = nx.DiGraph()

    # Randomize ordering for search
    def shuffle(self):
        self.ordering  = random.sample(list(self.variables.keys()), len(self.
    variables.keys()))

    def fit(self, data):
        DG = nx.DiGraph()
        DG.add_nodes_from(self.idx2names)
        print('Total nodes: {}'.format(len(self.ordering)))
        for k, i in tqdm(enumerate(self.ordering)):
            y = bayesian_score(DG, data, self.idx2names, self.variables)
            while True:
                y_best = -np.inf
                j_best = 0
                for num, j in enumerate(self.ordering[0:k]):
                    if not DG.has_edge(j, i):
                        DG.add_edge(j, i)
                        y_prime = bayesian_score(DG, data, self.idx2names,
    self.variables)
                        if y_prime > y_best:
                            y_best = y_prime
                            j_best = j
                        DG.remove_edge(j, i)
                    if num > 4:
                        break
                if y_best > y:
                    y = y_best
                    DG.add_edge(j_best, i)
                else:
                    break
        self.bs    = y
        self.graph = DG

# Opportunistic Local Search Implementation
class OppLocalSearch:
    def __init__(self, variables, idx2names):
        self.variables = variables
        self.ordering  = random.sample(list(variables.keys()), len(variables.
    keys()))
        self.idx2names = idx2names
        self.bs        = -np.inf
        self.graph     = nx.DiGraph()
        self.maxiter   = _OLS_ITER_

    def fit(self, data, initial_graph):
        G = initial_graph
        y = bayesian_score(G, data, self.idx2names, self.variables)
        for k in tqdm(range(self.maxiter)):
            G_prime = random_graph_neighbor(G)
            cycles = [c for c in nx.algorithms.simple_cycles(G_prime)]
            if cycles:
                y_prime = -np.inf
            else:
                y_prime = bayesian_score(G_prime, data, self.idx2names, self.
    variables)
            if y_prime > y:
                y = y_prime
                G = G_prime
        self.bs = y
        self.graph = G

# Returns neighbor graph for OLS
def random_graph_neighbor(G):
    n = len(G.nodes)
    i = np.random.randint(0, high=n)
    j = np.mod(i + np.random.randint(1, high=n) - 1, n)
    G_prime = G.copy()
    if G.has_edge(j, i):
        G_prime.remove_edge(j, i)
    else:
        G_prime.add_edge(j, i)
    return G_prime

# Writes .gph file
def write_gph(dag, idx2names, filename):
    with open(filename, 'w') as f:
        for edge in dag.edges():
            f.write("{}, {}\n".format(idx2names[edge[0]], idx2names[edge[1]]))

# General algorithm
def compute(infile, outfile):
    data      = pd.read_csv(infile)
    idx2names = dict(zip(range(len(data.columns)), list(data.columns)))
    variables = dict(zip(range(len(data.columns)),np.zeros(len(data.columns))
    ))
    for i in variables:
        variables[i] = np.max(data[idx2names[i]])
    bs_to_beat = -np.inf
    res        = K2Search(variables, idx2names)
    begin_time = datetime.datetime.now()

    # # Runs K2 Search _K2_ITER_ times, saves best result
    k2res = Manager().dict()
    proclist = []
    for p in range(_K2_ITER_):
        proc = Process(target=k2_iteration, args=(variables, idx2names, data,
     k2res, p))
        proclist.append(proc)
        proc.start()
    for proc in proclist:
        proc.join()
    for kres in k2res.values():
        if kres.bs > bs_to_beat:
            res = kres
            bs_to_beat = kres.bs

    print('Best K2 Result: {}'.format(res.bs))

    # Runs OLS using best K2 graph as initialization
    ols = OppLocalSearch(variables, idx2names)
    ols.fit(data, res.graph)

    print('ols result: {}'.format(ols.bs))
    print('\n================================')
    print('Final Bayesian Score: {}'.format(ols.bs))
    print(datetime.datetime.now() - begin_time)
    fname = infile.split('/')[-1]
    plot_graph(ols.graph, fname.split('.')[0])
    write_gph(ols.graph, idx2names, outfile)
    pass

# One iteration of K2 search, using a random ordering
def k2_iteration(variables, idx2names, data, k2res, procnum):
    obj = K2Search(variables, idx2names)
    obj.shuffle()
    obj.fit(data)
    print('k2 result: {}'.format(obj.bs))
    k2res[procnum] = obj

# Returns bayesian score of input graph
def bayesian_score(graph, data, idx2names, variables):
    # names2idx = {v: k for k, v in idx2names.items()}
    alpha = prior(variables, graph)
    M = statistics(variables, graph, data, idx2names)
    n = len(variables)
    return np.sum([bayesian_score_component(M[i], alpha[i]) for i in range(n)
    ])
```

```
# Summations for bayesian scoring
def bayesian_score_component(M, alpha):
    p  = np.sum(sp.loggamma(alpha + M))
    p -= np.sum(sp.loggamma(alpha))
    p += np.sum(sp.loggamma(np.sum(alpha, axis=1)))
    p -= np.sum(sp.loggamma(np.sum(alpha, axis=1) + np.sum(M, axis=1)))
    return p

# Dirichlet uniform prior used for all graphs
# Enables log(P(G)) term to be 0 in bayesian score
def prior(variables,G):
    n=len(variables)
    r=[variables[i] for i in range(n)]
    q=[int(np.prod([r[j] for j in G.predecessors(i)])) for i in range(n)]
    return [np.ones((q[i],r[i])) for i in range(n)]

# Returns matrix M, which has counts of each event outcome
# added onto the Dirichlet uniform prior distribution for given graph
def statistics(variables, G, df, idx2names):
    n=len(df.columns)
    r=[variables[i] for i in range(n)]
    q=[int(np.prod([r[j] for j in G.predecessors(i)])) for i in range(n)]
    M=[np.zeros((q[i],r[i])) for i in range(n)]
    for index, row in df.iterrows():
        for i in range(n):
            k=row[idx2names[i]] - 1
            parents=[pred for pred in G.predecessors(i)]
            j=np.array([0])
            parent_list = []
            for parent in parents:
                parent_list.append(parent)
            parents_array = np.array([x for x in parent_list])
            dims = ()
            for p in parents_array:
                dims += (r[p],)
            for parent in parents:
                row_sub = np.array([[row[idx2names[x]] - 1] for x in
    parent_list])
                j=np.ravel_multi_index(row_sub, dims, order='F')
                break
            M[i][j,k]+=1.0
    return M

# Simple graph visualization
def plot_graph(graph, sz):
    ax = plt.subplot()
    nx.draw(graph, with_labels=True, font_weight='bold')
    plt.savefig('./output/{}.png'.format(sz))
```

```
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()
```