Project 1 - Bayesian Structure Learning

Algorithm

The algorithm entails a hybrid k2 search plus a local search method. At the very beginning, it initializes the graph with all variables but no directed edges. It then enters a loop where K2 search is deployed, which iterates over the variables in an ascending order, greedily adding parents to the nodes in a way that maximally increases the score. *Opportunistic* local search method is used afterwards, which generates a single random neighbor and accepts if its score is greater than that of the current graph. The maximum number of the local search is set at 100 iterations. The loop is exited till Bayesian score after the K2 and local search could not improve any further, returning the optimal graph structure with maximum score.

Bayesian score is used for both methods with unit uniform Dirichlet prior assumptions. K2 search also deployed an arbitrary ascending order as prior is assumed to be uniform.

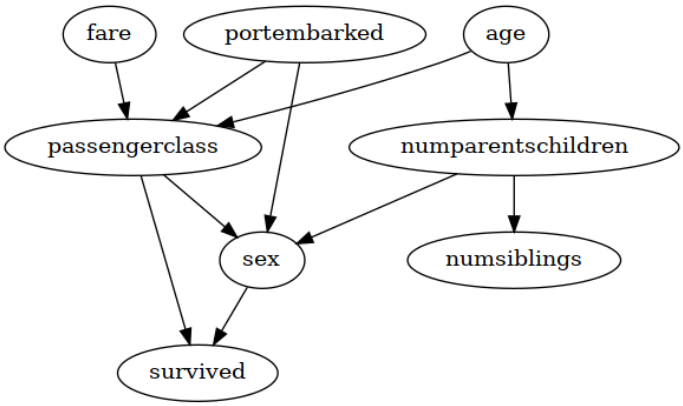
Running time & bayesian score for each problem

Small : 5.946	Score : -3835.679
Medium : 82.080	Score : -42060.476
Large : 14373.34	Score : -408087.152

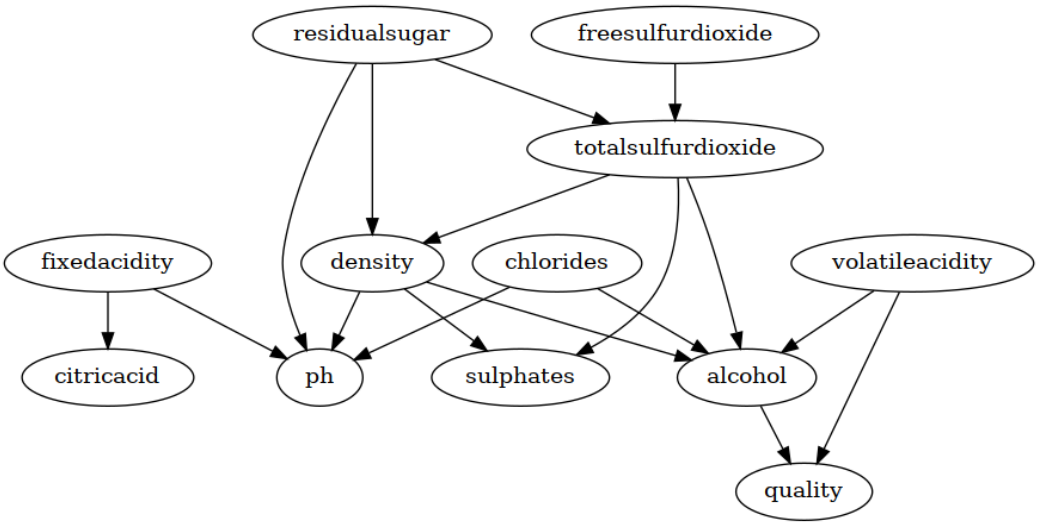
Discussion

I added this session to share some of the observations along this project study. First, I noticed that the graphs produced by K2 search all seem to follow an ascending order, this tends to produce more locality for the K2 research results, where some random ordering for parent addition could potentially help to improve the search results. Second, I noticed that the local search post K2 does not necessarily improve Bayesian score, thus graph structure. This could potentially be improved by deploying more random start between each iteration. Third, the current way of switch implementation regarding node index and node object due to nx.graphs data structure could have been done in a leaner way - make node object as node index. However, it was not intuitive to me how to incorporate node object names into visualization, if the leaner way was deployed. Last, I think a simplified Baysian Score computation can be used to reduce compute time.

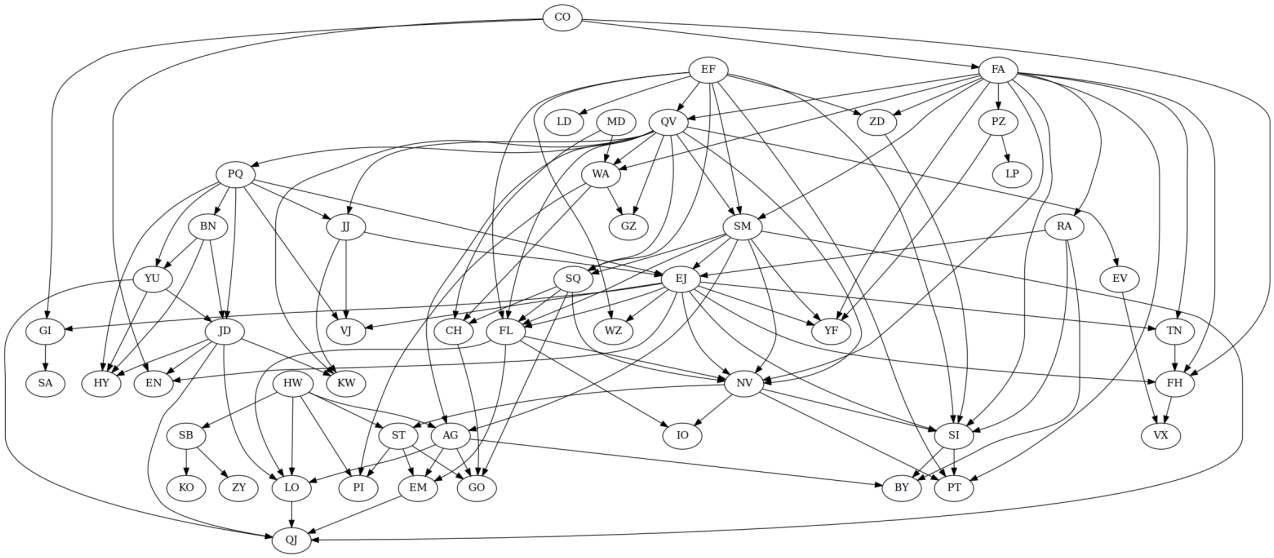
Visualization of each graph



Small



Medium



Large

Code (Python)

The code bodies compose the main algorithm session and the necessary utility functions. Some of the utility functions are downloaded from and modified upon the python code repository of the textbook.

main.py

```
import sys
import pandas as pd
import time
from scipy.special import loggamma
import numpy as np
import networkx as nx
from networkx.drawing.nx_pydot import write_dot
import graphviz
from ch02 import Variable
from ch05 import bayesian_score


def write_gph(G, filename):
    edges = []
    for key, values in nx.to_dict_of_lists(G).items():
        for value in values:
            edges.append("{}, {}\n".format(key, value))

    with open(filename, 'w') as f:
        f.writelines(edges)

def init_graph(data):
    G = nx.DiGraph()
    G.add_nodes_from(data.columns)
    return G

def init_variables(data):
    variables = []
    for i in data.columns:
        variables.append(Variable(i, max(data[i])))
    return variables

def init_data(data):
    return data.to_numpy().T-1

def draw_graph(G, filename):
    with open(filename,'w') as f:
        write_dot(G,f.name)
```

```
        img_file = graphviz.render('dot','png',f.name)

def compute(infile, outfile1, outfile2):
    rawdata = pd.read_csv(infile)
    G = init_graph(rawdata)
    variables = init_variables(rawdata)
    data = init_data(rawdata)
    score = bayesian_score(variables,G,data)
    print(score)

    start = time.time()

    while True:
        k2(variables,G,data)
        localsearch(variables,G,data,100)
        new_score = bayesian_score(variables, G, data)
        print(new_score)
        if(new_score <= score):
            break
        score = new_score

    write_gph(G, outfile1)
    draw_graph(G, outfile2)
    end = time.time()
    print(end-start)


def k2(variables, G, data):
    for k, i in enumerate(G):
        y = bayesian_score(variables, G, data)
        while True:
            y_best, j_best = -np.inf, list(G)[0]
            for j in range (k):
                nj = list(G)[j]
                if not G.has_edge(nj, i):
                    G.add_edge(nj, i)
                    y_prime = bayesian_score(variables, G, data)
                    if nx.is_directed_acyclic_graph(G) and y_prime > y_best:
                        y_best, j_best = y_prime, nj
                    G.remove_edge(nj, i)
            if y_best > y:
                y = y_best
                G.add_edge(j_best, i)
            else:
```

```
            break

def localsearch(variables, G,  data, k_max):
    y = bayesian_score(variables, G, data)
    for k in range(k_max):
        graph_prime = rand_graph_neighbor(G)
        if len(list(nx.simple_cycles(graph_prime))) == 0:
            y_prime = bayesian_score(variables, graph_prime, data)
        else:
            y_prime = -np.inf
        if y_prime > y:
            y, G = y_prime, graph_prime

def rand_graph_neighbor(graph) -> nx.DiGraph:
    n = graph.number_of_nodes()
    i = np.random.randint(low=0, high=n)
    j = (i + np.random.randint(low=1, high=n) - 1) % n
    ni = list(graph)[i]
    nj = list(graph)[j]
    graph_prime = graph.copy()
    if graph.has_edge(ni, nj):
        graph_prime.remove_edge(ni, nj)
    else:
        graph_prime.add_edge(ni, nj)
    return graph_prime

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

    inputfilename = sys.argv[1]
    outputfilename1 = sys.argv[2]
    outputfilename2 = sys.argv[3]
    compute(inputfilename, outputfilename1, outputfilename2)

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

ch2.py

```
class Variable():
    """
    A variable is given a name (represented as a string) and may take on an integer from 0 to r -
1
    """
```

```
    def __init__(self, name: str, r: int):
        self.name = name
        self.r = r  # number of possible values

    def __str__(self):
        return "(" + self.name + ", " + str(self.r) + ")"
```

ch5.py

```
def bayesian_score_component(M: np.ndarray, alpha: np.ndarray) -> float:
    # Note: The `loggamma` function is provided by `scipy.special`
    alpha_0 = np.sum(alpha, axis=1)
    p = np.sum(loggamma(alpha + M))
    p -= np.sum(loggamma(alpha))
    p += np.sum(loggamma(alpha_0))
    p -= np.sum(loggamma(alpha_0 + np.sum(M, axis=1)))
    return p


def bayesian_score(variables: list[Variable], graph: nx.DiGraph, data: np.ndarray) -> float:
    """
    An algorithm for computing the Bayesian score for a list of `variables` and a `graph` given `data`.
    This method uses a uniform prior alpha_{ijk} = 1 for all i, j, and k as generated by algorithm 4.2 (`prior`).
    Chapter 4 introduced the `statistics` and `prior` functions.

    Note: log(Γ(alpha)/Γ(alpha + m)) = log Γ(alpha) - log Γ(alpha + m), and log Γ(1) = 0.
    """
    n = len(variables)
    M = statistics(variables, graph, data)
    alpha = prior(variables, graph)
    return np.sum([bayesian_score_component(M[i], alpha[i]) for i in range(n)])
```

ch4.py

```
def statistics(variables: list[Variable], graph: nx.DiGraph, data: np.ndarray) -> list[np.ndarray]:
    """
    A function for extracting the statistics, or counts, from a discrete data set `data`,
    assuming a Bayesian network with `variables` and structure `graph`.
    The data set `data` is an n x m matrix, where n is the number of variables and
    m is the number of data points. This function returns an array M of length n.
    The ith component consists of a q_i x r_i matrix of counts.

    ASSUMES DATA IS ZERO-INDEXED: instead of the entries begin classes {1, 2, 3}, they must be {0, 1, 2}
```

```
    Note: `np.ravel_multi_index` indexes the parental instantiations differently than the textbook's
`sub2ind`
    """
    n = len(variables)
    r = np.array([var.r for var in variables])
    q = np.array([int(np.prod([r[list(graph).index(j)] for j in graph.predecessors(node)])) for node in
graph])
    M = [np.zeros((q[i], r[i])) for i in range(n)]
    for o in data.T:
        for i, node in enumerate(graph):
            k = o[i]
            parents = list([list(graph).index(j) for j in graph.predecessors(node)])
            # print(parents)
            j = 0
            if len(parents) != 0:
                j = np.ravel_multi_index(o[parents], r[parents])
            M[i][j, k] += 1.0
    return M


def prior(variables: list[Variable], graph: nx.DiGraph) -> list[np.ndarray]:
    """
    A function for generating a prior alpha_{ijk} where all entries are 1.
    The array of matrices that this function returns takes the same form
    as the counts generated by `statistics`.
    To determine the appropriate dimensions, the function takes as input the
    list of variables `variables` and structure `graph`.
    """
    n = len(variables)
    r = [var.r for var in variables]
    q = np.array([int(np.prod([r[list(graph).index(j)] for j in graph.predecessors(node)])) for node in
graph])
    return [np.ones((q[i], r[i])) for i in range(n)]
```