Project 1: Bayesian Structure Learning

Zhemin Huang zheminh@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

I use the K2 search algorithm with multiple random starting variable orderings to construct the best graph with the highest Bayesian score. K2 search is a greedy algorithm for learning the structure of Bayesian networks. It starts with a predefined ordering of variables and iteratively adds edges between parent and child nodes to maximize a Bayesian score, which measures the likelihood of the network given the data conditions.

For each round, I randomly shuffle the nodes representing the variables to generate a starting order. To optimize the process, instead of recalculating the entire graph's Bayesian score during each iteration, I focus only on the node i being processed. When an edge (j, i) is added, only the Bayesian score of node i changes. Thus, optimizing the score of each individual node incrementally leads to optimizing the score of the entire graph.

The algorithm works by initially setting the Bayesian score of node i without any parent nodes. For each potential parent j, we calculate the Bayesian score of i after adding the parent node j. If no parent j results in a positive score increase, the loop terminates. Otherwise, we continue adding parent nodes to node i until no further improvements are possible. Afterward, the algorithm moves to the next node $i + 1$ in the ordering.

2. Running Time

For the small dataset, we try 100 times for different orderings.

For the medium and large datasets, we try 10 times for different orderings.

Here is the average running time of one trial of each dataset:

- Small dataset: 0.53 sec
- Medium dataset: 35 sec
- Large dataset: 80 min

3. Graphs

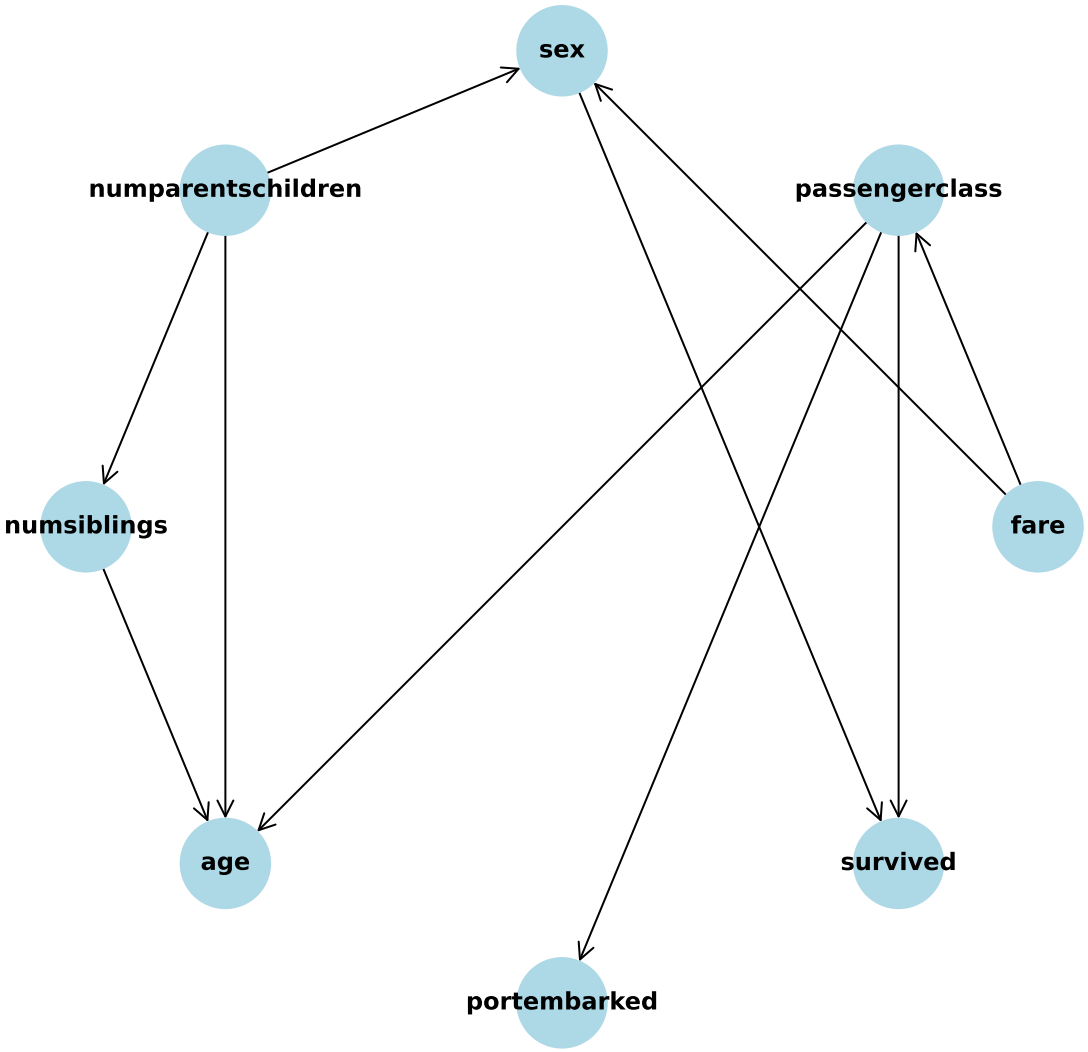

Figure 1: Bayesian network structure of the small dataset

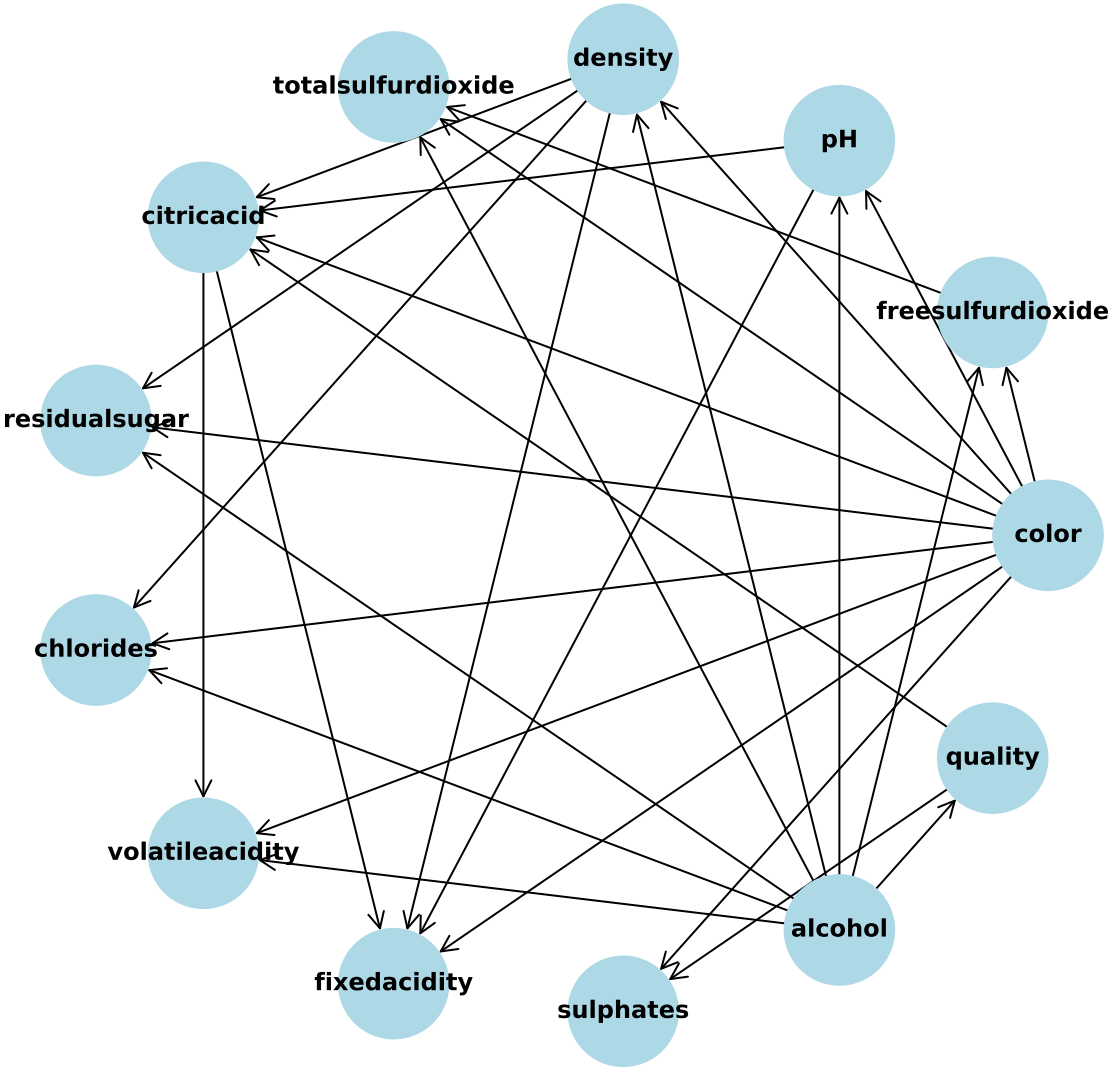

Figure 2: Bayesian network structure of the medium dataset

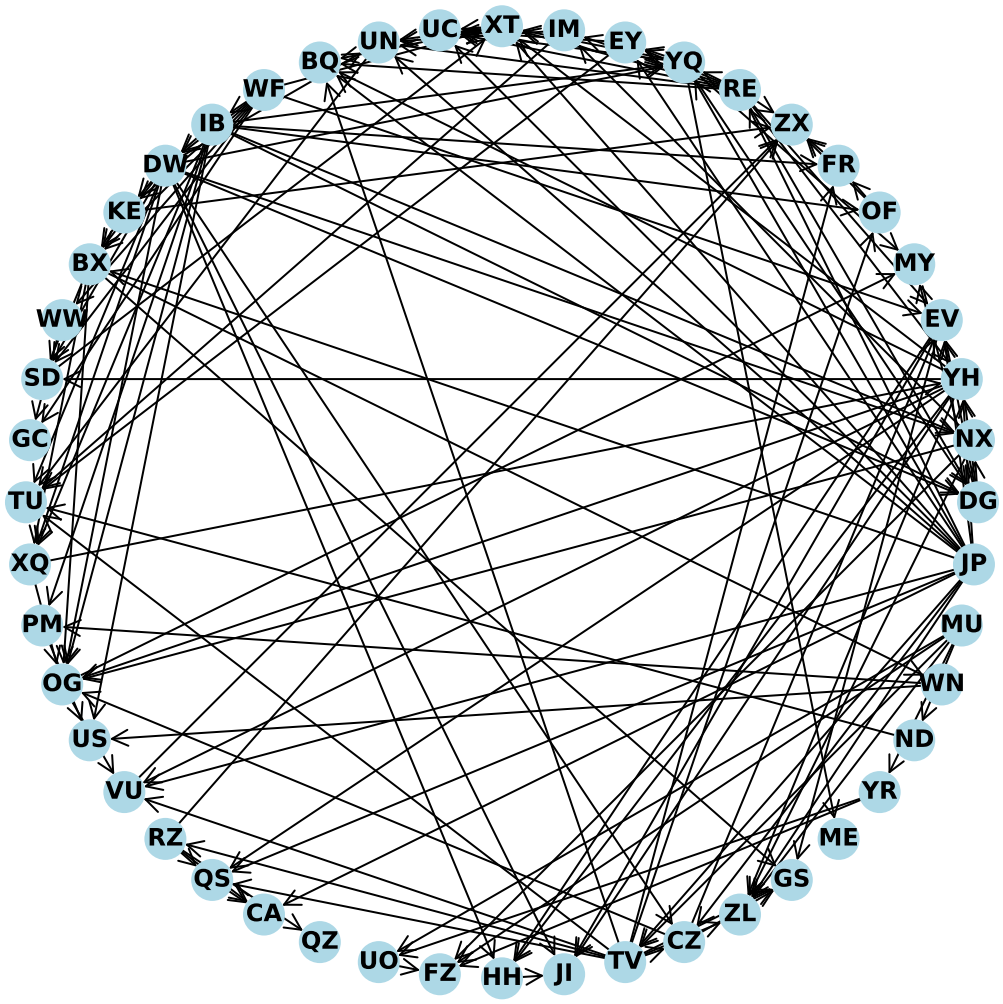

Figure 3: Bayesian network structure of the large dataset

4. Code

4.1 Calculating Bayesian Score

```
import sys
import pandas as pd
import networkx as nx
import numpy as np
from scipy.special import loggamma
```

```
from networkx.drawing.nx_agraph import write_dot
import os
import time
import random

class Variable():
    def __init__(self, name: str, r: int):
        self.name = name
        self.r = r


def statistics(vars: list[Variable], graph: nx.DiGraph, data: np.ndarray) ->
    list[np.ndarray]:
    n = len(vars)
    r = np.array([var.r for var in vars])
    q = np.array([int(np.prod([r[j] for j in graph.predecessors(i)])) for i
    in range(n)])
    M = [np.zeros((q[i], r[i])) for i in range(n)]

    for o in data.T:
        for i in range(n):
            k = o[i]
            parents = list(graph.predecessors(i))
            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]:
    n = len(variables)
    r = [var.r for var in variables]
    q = np.array([int(np.prod([r[j] for j in graph.predecessors(i)])) for i
    in range(n)])
    return [np.ones((q[i], r[i])) for i in range(n)]


def bayesian_score_component(M: np.ndarray, alpha: np.ndarray) -> float:
    p = np.sum(loggamma(alpha + M))
    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, axis=1)))
    return p


def bayesian_score(vars: list[Variable], G: nx.DiGraph, D: np.ndarray) ->
    float:
    n = len(vars)
    M = statistics(vars, G, D)
```

```
    alpha = prior(vars, G)
    return np.sum([bayesian_score_component(M[i], alpha[i]) for i in range(n)
    ])
```

4.2 K2 Search Algorithm

```
def fit(self, variables: list[Variable], data: np.ndarray) -> nx.DiGraph:
    directed_graph = nx.DiGraph()
    directed_graph.add_nodes_from(range(len(variables)))
    for node_index, current_node in enumerate(self.ordering[1:]):
        current_score = bayesian_score(variables, directed_graph, data)
        while True:
            best_score, best_parent_node = -np.inf, 0
            for candidate_node in self.ordering[:node_index]:
                if not directed_graph.has_edge(candidate_node, current_node):
                    directed_graph.add_edge(candidate_node, current_node)
                    candidate_score = bayesian_score(variables,
    directed_graph, data)
                    if candidate_score > best_score:
                        best_score, best_parent_node = candidate_score,
    candidate_node
                    directed_graph.remove_edge(candidate_node, current_node)
            if best_score > current_score:
                current_score = best_score
                directed_graph.add_edge(best_parent_node, current_node)
            else:
                break
    return directed_graph
```

4.3 Compute Function

```
def compute(infile, outfile):
    df = pd.read_csv(infile)
    data = df.to_numpy().T - 1
    columns = df.columns

    value_counts = {column: max(df[column]) for column in df.columns}
    variables = [Variable(key, value) for key, value in value_counts.items()]

    num_columns = df.shape[1]

    ordering = list(range(num_columns))
    M = K2Search(ordering)

    start_time = time.time()
    graph = M.fit(variables, data)
    end_time = time.time()
```

```
score = bayesian_score(variables, graph, data)

print("score:", score)
print(f"Elapsed time: {end_time - start_time} seconds")

labels = {i: columns[i] for i in range(len(columns))}


best_graph = None
best_score = 0

for i in range(trial_time):
    ordering = list(range(num_columns))
    random.shuffle(ordering)
    M = K2Search(ordering)

    start_time = time.time()
    graph = M.fit(variables, data)
    end_time = time.time()

    score = bayesian_score(variables, graph, data)

    print("score:", score)
    print(f"Elapsed time: {end_time - start_time} seconds")

    if score > best_score or best_graph == None:
        best_graph = graph
        best_score = score

graph = nx.relabel_nodes(best_graph, labels)

with open(outfile, 'w') as f:
    for u, v in graph.edges():
        f.write(f"{u},{v}\n")

# dot_file = 'graph.dot'
# write_dot(graph, dot_file)

# os.system(f'dot -Tpng {dot_file} -o graph.png')

# draw circular
nx.draw_circular(G, with_labels=True, node_color='lightblue', node_size
 =2000, font_size=12, font_weight='bold',
             arrowstyle='->', arrowsize=20)
plt.savefig("graph.pdf")
```