Project 1: Bayesian Structure Learning

Aaron Jin aaronjin@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

My implementation uses the K2 search algorithm for Bayesian network structure learning. As discussed in class, the K2 algorithm is a greedy search method that incrementally constructs our directed acyclic graph (DAG) by adding parent nodes to each variable in a predefined order to maximize the Bayesian score. We then use this score to evaluate how well the network structure explains our observed data. The implementation is divided into the following key components:

1.1 Data Preprocessing

The first step is data preprocessing. We know from the problem statement that all datasets are already discretized, ensuring that all variables are discrete with a finite number of states. Thus, our preprocessing steps is bifold: (1) categorical variables are encoded as integers starting from 0; and (2) discrete variables are ensured to be in integer format without additional discretization.

1.2 Scoring Function

The second step is the scoring function, where we calculate the Bayesian score for each variable given its current set of parents. For this step, we use the Gamma function to compute log-probabilities, which aligns with the Bayesian scoring criteria. For variables without parents, the score depends solely on the variable's distribution. The function calculates the log probability of the observed counts given the uniform prior. For variables with parents, the score accounts for each parent configuration. It aggregates the log probabilities across all possible parent states, ensuring that the network accurately captures dependencies between variables.

1.3 Search Algorithm

The third component is the K2 Search Algorithm itself. This greedy algorithm iterates over variables in a specified order, attempting to add the parent that most improves the Bayesian score at each step. The process continues iteratively until no further improvement is possible or a predefined maximum number of parents is reached for each variable. We start with an empty DAG where each variable has no parents. Then, for each variable, we evaluate potential parent candidates and select the one that provides the highest score improvement. We finally terminate by stopping the addition of parents to a variable when no candidate parent can improve the score or when the maximum number of parents is reached.

1.4 Graph Visualization

The final step is graph visualization. Utilizing Matplotlib and NetworkX, the learned Bayesian network is visualized as a directed graph. Nodes represent variables, and directed edges indicate parent-child relationships.

2. Running Time

I have recorded the execution times for the K2 search algorithm across our different datasets in the table below:

Table 1: Execution Times for Each Dataset

Dataset	**Execution Time**
small.csv	0.08 seconds
medium.csv	0.29 seconds
large.csv	5.38 seconds

Surprisingly, these results indicate that the algorithm is relatively both scalable and efficient, as it can handle increasing data sizes with manageable computation times.

3. Graphs

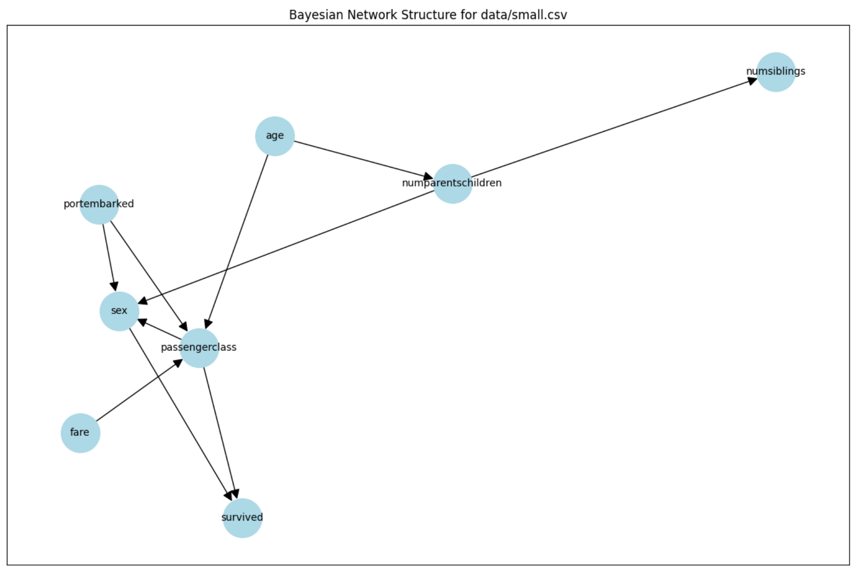

Figure 1: Bayesian Network Structure for Small Dataset

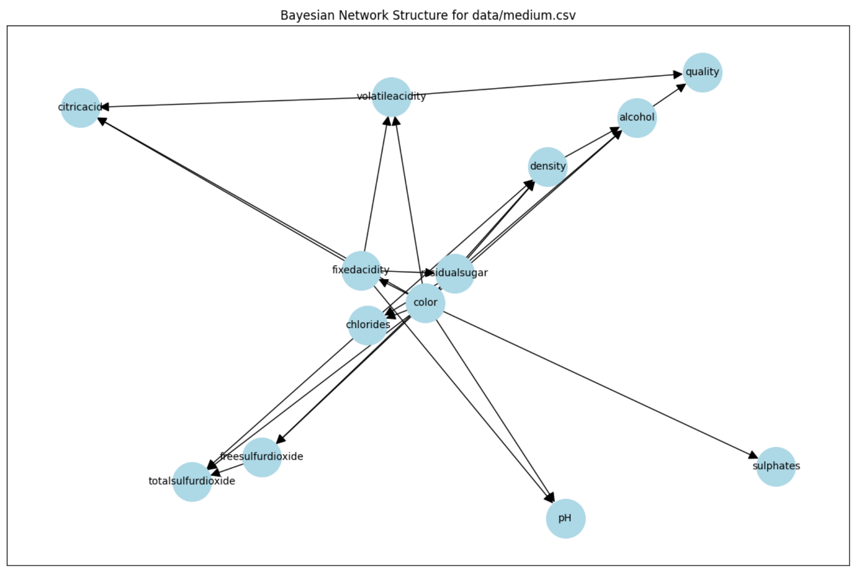

Figure 2: Bayesian Network Structure for Medium Dataset

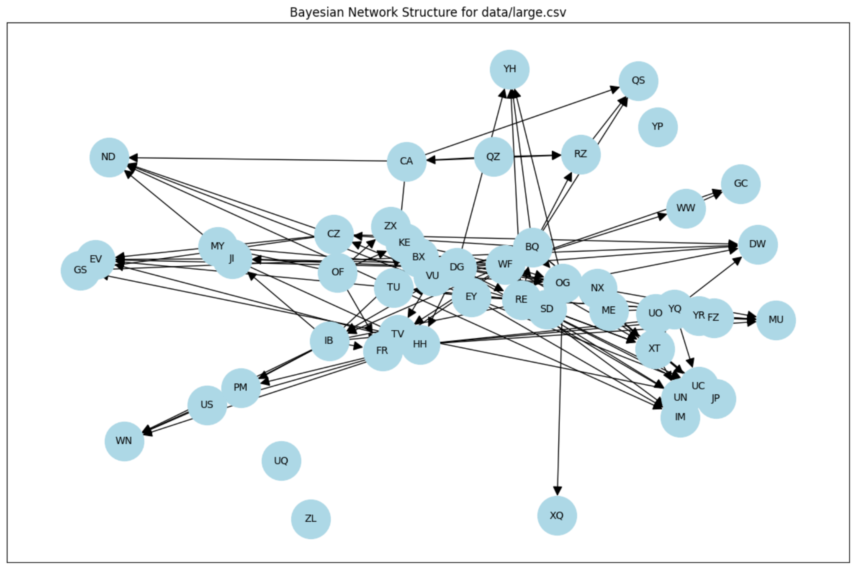

Figure 3: Bayesian Network Structure for Large Dataset

4. Code

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

```
import matplotlib.pyplot as plt
import time


# Step 1: Data preprocessing
def read_data(filename):
    data = pd.read_csv(filename)
    for col in data.columns:
        if data[col].dtype == 'object':
            # Categorical var
            data[col] = data[col].astype('category').cat.codes
        else:
            # Discrete var
            data[col] = data[col].astype('category').cat.codes

    # Debug: Print number of states per var
    print("Number of states per variable after encoding and discretization:")
    for col in data.columns:
        print(f"{col}: {data[col].nunique()} states")
    return data


def write_gph(dag, filename):
    with open(filename, 'w') as f:
        for parent, child in dag.edges():
            f.write(f"{parent},{child}\n")


def initialize_alpha(data):
    alpha = {}

    # Using uniform priors: alpha = 1 for each state
    for var in data.columns:
        num_states = data[var].nunique()
        alpha[var] = np.ones(num_states)

    return alpha


# Step 2: Scoring function
def compute_variable_score(data, var, parents, alpha):
    # Edge case: No parents (score only depends on var itself)
    if not parents:
        counts = data[var].value_counts().values
        total_count = len(data)
        alpha_sum = np.sum(alpha[var])

        score = gammaln(alpha_sum) - gammaln(alpha_sum + total_count)
        score += np.sum(gammaln(alpha[var] + counts) - gammaln(alpha[var]))
        return score

    # General case: With parents (compute scores for each parent config)
    else:
        grouped = data.groupby(parents)[var].value_counts().unstack(
    fill_value=0)
        counts_parent = grouped.sum(axis=1).values
        counts_child = grouped.values
        alpha_sum = np.sum(alpha[var])

        score_part1 = len(counts_parent) * gammaln(alpha_sum) - np.sum(
    gammaln(alpha_sum + counts_parent))
        score_part2 = np.sum(gammaln(counts_child + alpha[var]) - gammaln(
    alpha[var]))

        total_score = score_part1 + score_part2

        return total_score


def compute_bayesian_score(data, dag, alpha, variable_order):
    total_score = 0.0

    for var in variable_order:
        parents = list(dag.predecessors(var))
        score = compute_variable_score(data, var, parents, alpha)
        total_score += score

    return total_score


# Step 3: Search algorithm
def k2_search(data, alpha, max_parents=5):
    variables = list(data.columns)
    variable_order = variables.copy()
    dag = nx.DiGraph()
    dag.add_nodes_from(variables)
    total_score = 0.0

    unique_states = {var: data[var].unique() for var in variables}

    for i, var in enumerate(variable_order):
        cur_parents = []
        best_score = compute_variable_score(data, var, cur_parents, alpha)
        improved = True

        while improved and len(cur_parents) < max_parents:
            improved = False
            candidate_parents = [v for v in variable_order[:i] if v not in
    cur_parents]
```

```
            scores = []

            if not candidate_parents:
                break

            # Calculate vectorized computation of scores for all candidate
    parents
            candidate_scores = []
            for candidate in candidate_parents:
                temp_parents = cur_parents + [candidate]
                score = compute_variable_score(data, var, temp_parents, alpha
    )
                candidate_scores.append((candidate, score))

            # Find candidate with highest score
            if candidate_scores:
                best_candidate, best_candidate_score = max(candidate_scores,
    key=lambda x: x[1])

                if best_candidate_score > best_score:
                    cur_parents.append(best_candidate)
                    best_score = best_candidate_score
                    improved = True
                else:
                    improved = False

        # Add selected parents to DAG
        for parent in cur_parents:
            dag.add_edge(parent, var)

        total_score += best_score
        print(f"Variable '{var}': Parents added: {cur_parents}, Score: {
    best_score}")

    return dag, total_score


# Step 4: Graph visualization
def plot_graph(dag, infile, outfile):
    plt.figure(figsize=(12, 8))
    pos = nx.spring_layout(dag, seed=42)
    nx.draw_networkx(dag, pos, with_labels=True, node_size=1500, node_color='
    lightblue', arrowsize=20, font_size=10)
    plt.title(f"Bayesian Network Structure for {infile}")
    image_filename = outfile.replace('.gph', '.png')
    plt.tight_layout()
    plt.savefig(image_filename)
    plt.close()
```

```
def compute(infile, outfile):
    start_time = time.time()

    data = read_data(infile)
    alpha = initialize_alpha(data)
    dag, total_score = k2_search(data, alpha, max_parents=5)
    print(f"Total Bayesian score for {infile}: {total_score}")

    plot_graph(dag, infile, outfile)

    write_gph(dag, outfile)

    end_time = time.time()
    duration = end_time - start_time
    minutes, seconds = divmod(duration, 60)

    if minutes >= 1:
        print(f"Time taken: {int(minutes)} minutes and {seconds:.2f} seconds")
    else:
        print(f"Time taken: {seconds:.2f} seconds")


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()
```