Project 1: Bayesian Structure Learning

Naomi Park ngpark@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

The structure learning algorithm implemented in this project is formally known as a **greedy hill-climbing search** that's guided by the Bayesian score (computed using a uniform Dirichlet prior). The formal mathematical description of the algorithm is outlined on page 2.

At a high level, it begins with an *empty* directed acyclic graph (DAG), which serves as the baseline, and iteratively explores neighbor structures created by adding, removing, or reversing a single edge. For each candidate DAG, the Bayesian score is evaluated to measure how well the structure explains the observed data. Only modifications that improve the score are accepted, ensuring consistent improvement until convergence to a local optimum.

During the search itself, I checked for and rejected cycles after every structural change. In addition, each node's parent set is restricted to a small maximum size to control computational complexity. Convergence was reached when no single edge modification improved the Bayesian score, indicating a local maximum. I monitored and printed the runtime for each dataset to get a sense of the algorithm's efficiency across increasing graph sizes. The final learned graph corresponds to the highest-scoring structure found within these constraints.

I chose to use a greedy hill-climbing search because it provides a straightforward yet effective way of exploring the large space of possible Bayesian network structures while guaranteeing consistent improvement in the score at each step. This allowed me to easily monitor the algorithm's performance my terminal output. In addition, the greedy hill-climbing algorithm enabled me limit the number of allowed parents. This limiting feature enables us to have control over the algorithm's runtime. As a result, this approach was easily adaptable, had reasonable computational efficiency, and the performance was sufficient enough to outperform the baseline.

In terms of my implementation process, the first step I took was to calculate the Bayesian score based on the uniform Dirichlet prior as defined in *Algorithms for Decision Making* (Kaelbling, Littman, & Parr, 2023, Section 5.1), as the Bayesian score is what the greedy hill-climbing search uses to construct each DAG. I ran the score calculation code on each .csv file to start, and then used my base implementation of the greedy hill-climbing search to calculate the empty graph before integrating the algorithm into the main *project1.py* file. Finally, I was able to call my greedy hill-climbing search on each graph within the *compute()* function, and I compared the final score for each the small, medium, and large dataset to that of the empty graph for each respective dataset.

Mathematical Description of the Greedy Hill-Climbing Search Algorithm

Let $\mathcal{G}$ denote the set of all directed acyclic graphs (DAGs) defined over variables $X_1, \ldots, X_n$. Each graph $G \in \mathcal{G}$ is evaluated by the Bayesian score

$$S(G) = \log P(D \mid G),$$

computed using a uniform Dirichlet prior. We restrict the search space to DAGs satisfying (i) acyclicity and (ii) a maximum in-degree K, i.e., $|\mathrm{Pa}_G(X_i)| \leq K$ for all nodes X_i. For any $G \in \mathcal{G}$, define its neighborhood $\mathcal{N}(G)$ as the set of DAGs that can be obtained from G by a single structural modification:

$$\mathcal{N}(G) = \Big\{ G' \text{ obtained from } G \text{ by one of } \begin{cases} \mathrm{ADD}(u \to v), \\ \mathrm{REMOVE}(u \to v), \\ \mathrm{REVERSE}(u \to v), \end{cases} \quad \text{s.t. } G' \text{ is acyclic and } |\mathrm{Pa}_{G'}(X_i)| \leq K \Big\}.$$

The greedy hill-climbing search proceeds as follows:

1. Initialize G_0 as the empty graph and set $t = 0$.
2. For each iteration, find
$$G^\star = \arg \max_{G' \in \mathcal{N}(G_t)} S(G').$$
3. If $S(G^\star) > S(G_t)$, set $G_{t+1} = G^\star$ and increment t. Otherwise, terminate and return G_t as the learned structure.

This procedure guarantees a non-decreasing sequence of scores $S(G_0) \leq S(G_1) \leq \cdots$ and terminates once no single-edge modification can further improve the Bayesian score, indicating convergence to a local optimum.

Runtime for Graph Generation:
Small: 9.32 seconds
Medium: 203.24 seconds
Large: 20055.53 seconds

2. Graphs

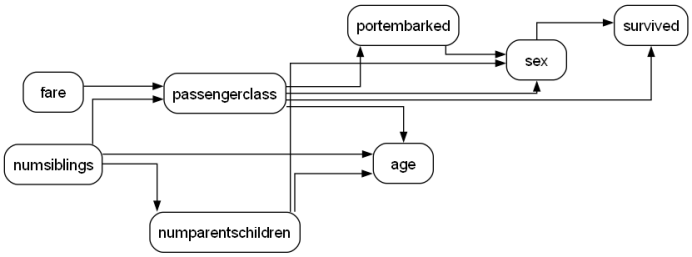

Figure 1: Visual small graph output.

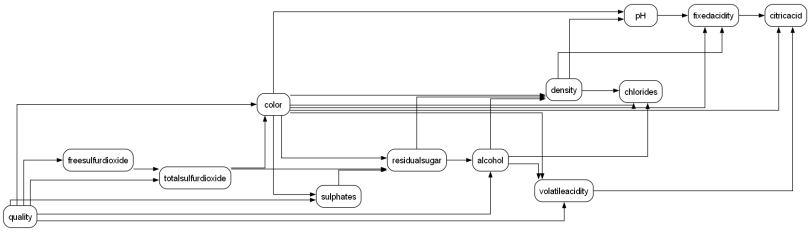

Figure 2: Visual medium graph output.

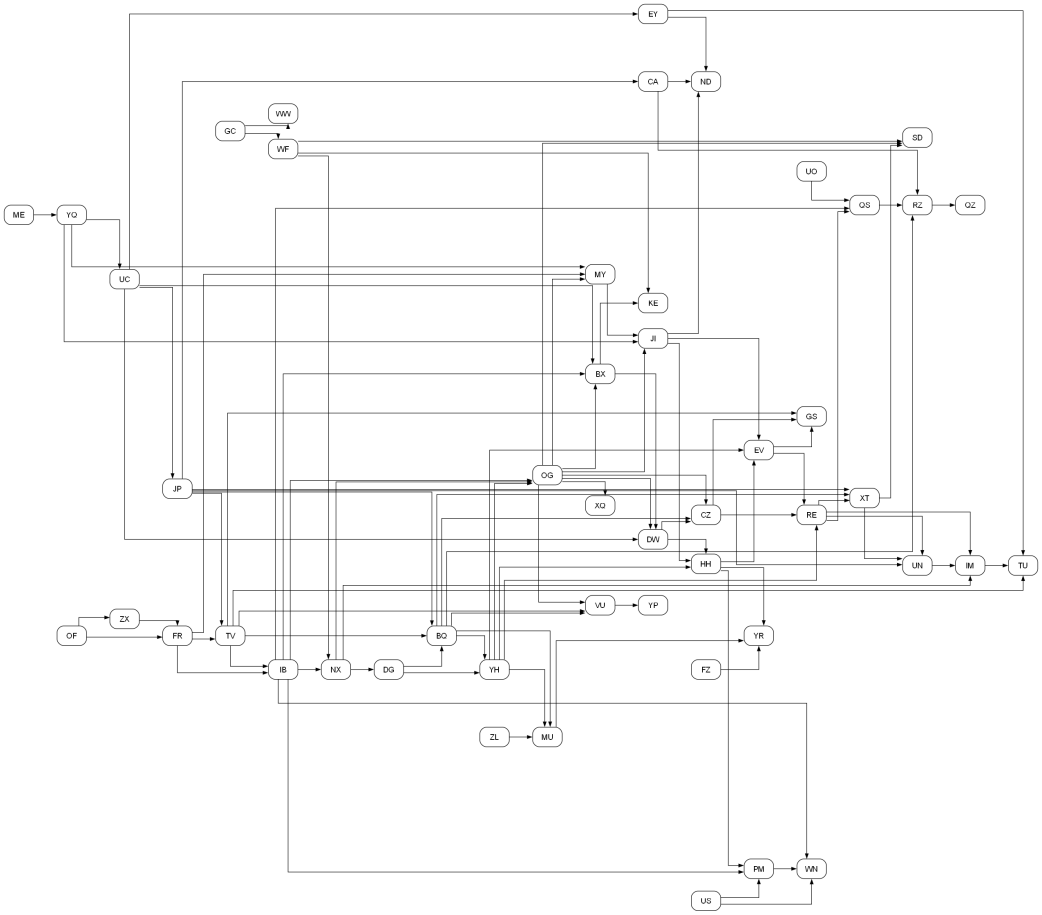

Figure 3: Visual large graph output.

3. Code

project1.py (main file):

```
import sys
import matplotlib.pyplot as plt
import networkx as nx
from bn_scoring import load_discrete_data
from structure_learning import hill_climb
import time


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]]))
```

```
def compute(infile, outfile):
    # WRITE YOUR CODE HERE
    # FEEL FREE TO CHANGE ANYTHING ANYWHERE IN THE CODE
    # THIS INCLUDES CHANGING THE FUNCTION NAMES, MAKING THE CODE MODULAR,
    BASICALLY ANYTHING
    df = load_discrete_data(infile) #load dataset
    columns = list(df.columns)

    idx2names = {i: col for i, col in enumerate(columns)} #maps integer node
    IDs to variable names (i.e. 0 --> "age")
    names2idx = {v: k for k, v in idx2names.items()} #maps variable names to
    integer node IDs ("age" --> 0)

    start_time = time.time()

    dag, best_score = hill_climb(df, max_iters=100, max_parents=3) #run
    structure learning algo and learn structure of DAG
    end_time = time.time()
    runtime = end_time - start_time

    plt.figure(figsize=(8, 6))
    pos = nx.spring_layout(dag, seed=0)
    nx.draw_networkx_nodes(dag, pos, node_size=900)
    nx.draw_networkx_labels(dag, pos, font_size=9)
    nx.draw_networkx_edges(dag, pos, arrows=True, arrowstyle='-|>', arrowsize
    =12)
    plt.axis('off')
    plt.tight_layout()
    plt.savefig(outfile.replace('.gph', '.png'), dpi=200)
    # plt.show()  #interactive window

    dag_indexed = nx.relabel_nodes(dag, names2idx, copy=True) #convert names
    to node IDs
    write_gph(dag_indexed, idx2names, outfile) #convert node IDs back to
    names

    print(f"Structure algorithm finished running. Best score = {best_score:.2
    f}")
    print(f"Runtime = {runtime:.2f} seconds")
    print(f"Graph written to {outfile}.")
    print(f"Edges: {list(dag.edges())}")

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()
```

bn_scoring.py (Bayesian Score Calculation File):

```
from typing import Dict, List, Tuple
import pandas as pd
from math import lgamma
from itertools import product

def load_discrete_data(path: str) -> pd.DataFrame:
    df = pd.read_csv(path)
    for c in df.columns:
        df[c] = df[c].astype(int)
    return df

def cardinalities(df: pd.DataFrame):
    return {col: int(df[col].max()) for col in df.columns}

def all_parent_configs(df: pd.DataFrame, parents: List[str], r: Dict[str, int
    ]):
    if not parents:
        return [()]
    from itertools import product
    ranges = [range(1, r[p] + 1) for p in parents]
    return list(product(*ranges))

def counts_for_node_given_parents(df: pd.DataFrame, node: str, parents: List[
    str], r: Dict[str, int]):
    result = {cfg: {k: 0 for k in range(1, r[node] + 1)} for cfg in
    all_parent_configs(df, parents, r)}
    if not parents:
        vc = df[node].value_counts()
        for k, cnt in vc.items():
            result[()][int(k)] = int(cnt)
        return result
    grouped = df.groupby(parents + [node]).size().reset_index(name='count')
    for _, row in grouped.iterrows():
        cfg = tuple(int(row[p]) for p in parents)
        k = int(row[node])
        result[cfg][k] = int(row['count'])
    return result

def bayesian_score_dirichlet_uniform(df: pd.DataFrame, graph: Dict[str, List[
    str]]) -> float:
    r = cardinalities(df)
    score = 0.0
```

```
    for node in df.columns:
        parents = graph.get(node, [])
        parents = [p for p in parents if p in df.columns and p != node]
        cfg_counts = counts_for_node_given_parents(df, node, parents, r)
        r_i = r[node]
        alpha_ij = r_i
        for cfg, k_counts in cfg_counts.items():
            N_ij = sum(k_counts.values())
            score += lgamma(alpha_ij) - lgamma(alpha_ij + N_ij)
            for k in range(1, r_i + 1):
                N_ijk = k_counts.get(k, 0)
                score += lgamma(1 + N_ijk) - lgamma(1)
    return float(score)
```

structure_learning.py (Greedy Hill-Climbing Search File):

```
import itertools
import networkx as nx
from bn_scoring import load_discrete_data, bayesian_score_dirichlet_uniform

def score_dag(df, dag):
    """Convert DAG into a structure of child:[parents] (i.e. map each child
    to its list of parents) and then compute the Bayesian score."""
    parent_map = {n: [] for n in dag.nodes}
    for u, v in dag.edges:
        parent_map[v].append(u)
    return bayesian_score_dirichlet_uniform(df, parent_map)

def neighbors(dag, max_parents=3):
    """Generate all DAGs that differ by one edge (add/remove/reverse)."""
    nodes = list(dag.nodes)
    existing = set(dag.edges)

    #try removing each edge as follows:
    for u, v in list(dag.edges):
        g2 = dag.copy()
        g2.remove_edge(u, v)
        yield g2

    #try adding each possible edge (if not already present) as follows:
    for u, v in itertools.permutations(nodes, 2):
        if (u, v) in existing or (v, u) in existing:
            continue
        g2 = dag.copy()
        g2.add_edge(u, v)
        if nx.is_directed_acyclic_graph(g2) and len(list(g2.predecessors(v)))
     <= max_parents:
            yield g2

    #try reversing existing edges as follows:
```

```
    for u, v in list(dag.edges):
        g2 = dag.copy()
        g2.remove_edge(u, v)
        g2.add_edge(v, u)
        if nx.is_directed_acyclic_graph(g2) and len(list(g2.predecessors(u)))
     <= max_parents:
            yield g2

def hill_climb(df, max_iters=200, max_parents=3):
    """Greedy hill climbing using the Bayesian score."""
    cols = list(df.columns)
    dag = nx.DiGraph()
    dag.add_nodes_from(cols)

    best_score = score_dag(df, dag)
    improved = True
    iteration = 0

    while improved and iteration < max_iters:
        improved = False
        iteration += 1
        best_neighbor, best_neighbor_score = None, best_score

        for g2 in neighbors(dag, max_parents=max_parents):
            s2 = score_dag(df, g2)
            if s2 > best_neighbor_score:
                best_neighbor_score = s2
                best_neighbor = g2

        if best_neighbor is not None:
            dag = best_neighbor
            best_score = best_neighbor_score
            improved = True
            print(f"Iteration {iteration}: improved score = {best_score:.2f}")

    return dag, best_score
```