Project 1: Bayesian Structure Learning

Akshar Sarvesh asarvesh@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

I first opted to run a K2 algorithm to learn the structure of the graphs. Starting from a graph with all of the nodes and no edges, I sorted the nodes by name randomly, and then iterated across all the parent/children pairs avoiding adding an edge where the child came earlier in the list than the parent to ensure cycle prevention. After getting this working on a single run of the small dataset, I then dropped this in a for loop changing the seed on every iteration to re-randomize the list of names

However, after running this for 20 seeds and obtaining a wide spread of scores, I realized that by adhering to the arbitrary ordering of the nodes, I was limiting what edges could be added too significantly, all for the sake of avoiding cycles which could be prevented in other ways. So at the cost of a slight drop in efficiency, I switched to more of a local search algorithm. I generated all possible edges I could *add* to the graph, but did not consider removing or reversing edges, like in the typical local search algorithm.

To ensure I did not create any cycles, I implemented a 2D boolean array where an entry [i, j] would represent that node i could reach node j, or node j was a descendant of i. When inspecting a candidate edge between a parent and a child, the array[child][parent] was true, then this edge would create a cycle, and we would discard. This would double the possible candidate edges on every iteration of the search where an iteration is greedily picking the singular edge that would increase the score the most. After picking this edge, committing it to the graph would entail changing the boolean array, going through every ancestor of the parent and descendant of the child and have each of those pairs be set to true.

Because we would search through every possible edge, that would be n^2 checks, and then at the end of committing one edge, we would have to do another n^2 operation to update the array. However, the act of recomputing the score is so much slower, that this added feature is theoretically not slower at all than having to run a K2 algorithm multiple times on different seeds. This algorithm is deterministic, although it still isn't guaranteed to find a global max score, it is likelier to find a better score than K2 on any individual random seed.

Optimizations

There were a few things I also implemented to speed things up the process as well.

1. First of all, I noticed that if a new edge was added, only a few terms relating to its parents changing needed to be modified in the score - so to test a candidate edge would take significantly reduced time by calculating each edge's contribution individually and recalculating only those parts when necessary

2. Furthermore, by keeping a dictionary of how much each node was contributing to the score, I would only have to calculate the *new* score of the candidate edge.

3. Looking at the contribution to the score each node holds, those values only change if the parents of the node change; otherwise, using the same data, regardless of other changes in the graph, the score of each candidate edge involving an individual node as a child would not change in delta. So then, between iterations I could save every node's best parent/delta tuple, choose to use the best one, and then only have to recalculate on the chosen node's delta in the next iteration. Only its set of parents changed. Of course, in order to ensure valid edges calculated in the past don't create cycles because of changes in the graph now, before committing an old edge we scan it once more quickly, and if it's actually unsafe then we recompute it next iteration - in the worst case we're doing the same number of recomputations per edge commit, but in most cases we're able to get away with huge time gains.

Runtime

1. The small graph took a few seconds. It is deterministic, as expected and I know this because I ran it a few times and got the same results every time.

2. The medium graph took a few minutes. This makes sense because it's a larger graph, so there are exponentially more edges to check and more iterations to learn for because the graph would have to end with more edges than the small graph (30 compared to 16, specifically)

3. The large graph took much more time, I ran it overnight before adding optimizations and noted it was going to take too long to converge, so I added code to allow it to pick up from where it left off by reading in a file, added optimizations, and then let it run. With the heavy optimizations (especially the pruning) it took about an hour and a half to run the whole thing.

2. Graphs

So here (on the next pages) are my visualizations for all of my graphs. I used the recommended networkx library to get the latex for them, and drew them in a circle because that maximized the visual clarity. Although I will say they should make a framework to handle DAGs specifically, because with the knowledge these don't have cycles, you can topologically sort the nodes and not have to make this visually so confusing.

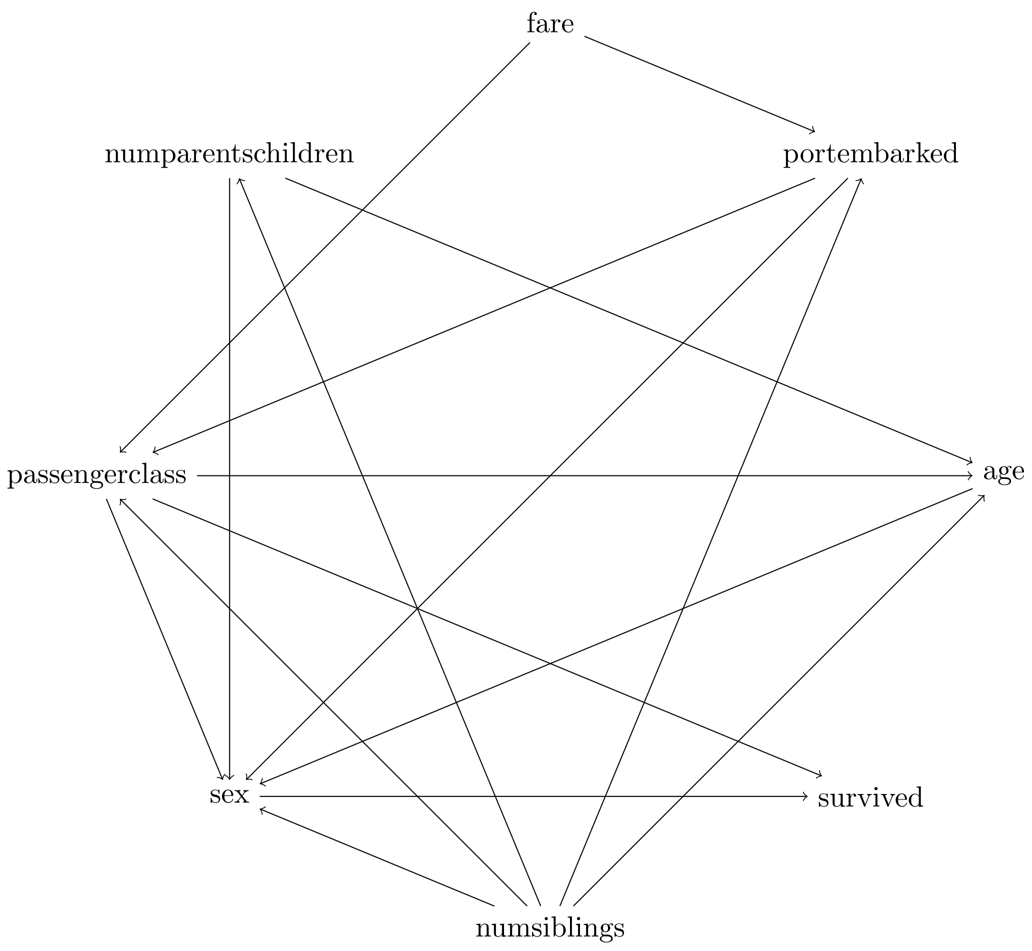

Figure 1: small

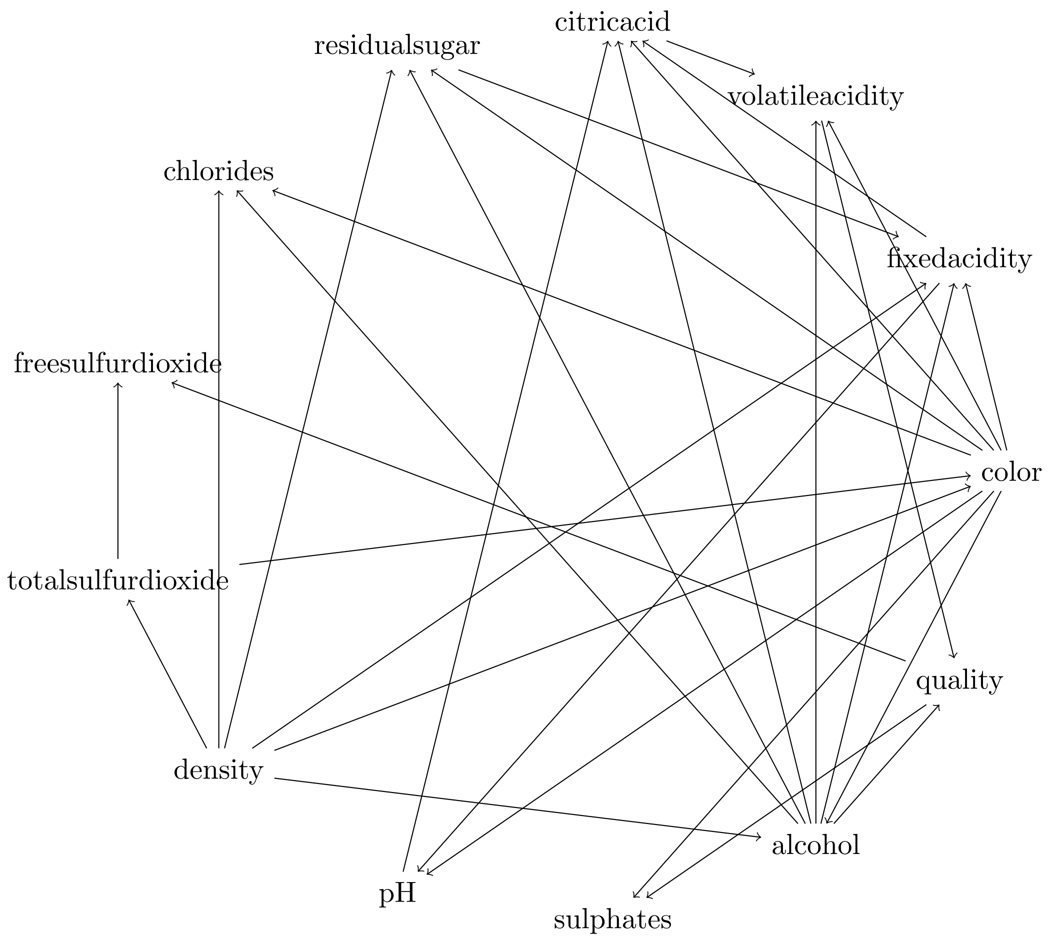

Figure 2: medium

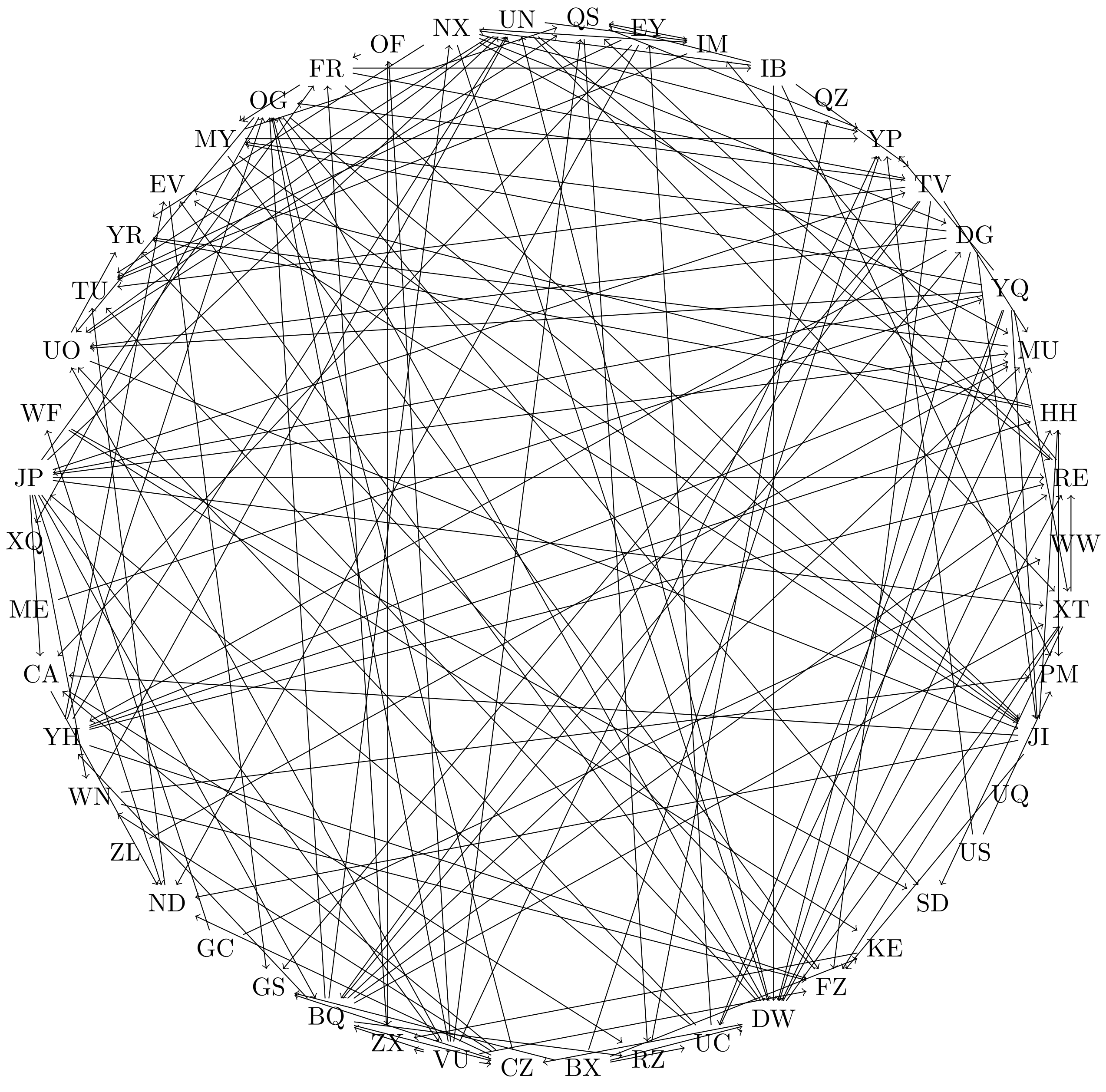

Figure 3: large

3. Code

```python
from math import lgamma
from itertools import product
import pandas as pd
import random

def get_state_names(df: pd.DataFrame):

    state_names = {}
    for col in df.columns:
        if isinstance(df[col].dtype, pd.api.types.CategoricalDtype):
            state_names[col] = list(df[col].cat.categories)
        else:
            state_names[col] = sorted(df[col].dropna().unique().tolist())
    return state_names

def ensure_categorical(df: pd.DataFrame, state_names: dict):
    """
    Return a copy of df with columns converted to Categorical using provided
    state_names.
    """
    out = df.copy()
    for col, states in state_names.items():
        out[col] = pd.Categorical(out[col], categories=states, ordered=False)
    return out

def save_graph(g: dict[str, list[str]], path: str):
    lines = []
    for child, parents in g.items():
        for parent in parents:
            lines.append(f"{parent},{child}")

    with open(path, "w") as f:
        f.write("\n".join(lines))

    print(f"[save_g] Saved {len(lines)} edges to {path}")

def local_k2_score(data, child: str, parents: list[str], state_names: dict[
    str, list]):

    log_score = 0.0
    r_i = len(state_names[child])
    alpha_ijk = 1

    parent_state_lists = [state_names[p] for p in parents] if parents else
    [()]
    parent_configs = list(product(*parent_state_lists)) if parents else [()]
    for j_cfg in parent_configs:
        subset = data
        if parents:
            for p, val in zip(parents, j_cfg):
                subset = subset[subset[p] == val]
        counts = subset[child].value_counts().reindex(state_names[child],
    fill_value=0).astype(int)
        N_ij = int(counts.sum())
        term = lgamma(r_i * alpha_ijk) - lgamma(N_ij + r_i * alpha_ijk)
        term += sum(lgamma(n_k + alpha_ijk) - lgamma(alpha_ijk) for n_k in
    counts)
        log_score += term
    return float(log_score)

def recompute_k2(data, g, parent, child, state_names, score_components) ->
    float:
    old = score_components[child]
    new = local_k2_score(data, child, g[child] + [parent], state_names)
    return new - old

def k2_score(df_cat: pd.DataFrame, g: dict[str, list[str]], state_names: dict
    [str, list], score_components: dict[str, float]) -> float:
    out = 0
    for child in g:
        score_components[child] = local_k2_score(df_cat, child, g[child],
    state_names)
        out += score_components[child]
    return out

def build_graph_from_edges(
    text: str,
    data: pd.DataFrame = None,
) -> dict[str, list[str]]:

    g: dict[str, list[str]] = {}

    if data is not None:
        g = {col: [] for col in data.columns}

    lines = [ln.strip() for ln in text.strip().splitlines() if ln.strip()]
    for i, line in enumerate(lines, start=1):
        parts = [p.strip() for p in line.split(",")]

        parent, child = parts

        if parent not in g[child]:
            g[child].append(parent)

    return g


def k2_search(case: str):
```

```
data = pd.read_csv(f'data/{case}.csv')
state_names = get_state_names(data)
data = ensure_categorical(data, state_names)

with open(f'{case}.gph', 'r') as f:
    g = build_graph_from_edges(f.read(), data)

vars_list = list(data.columns)
idx = {v: i for i, v in enumerate(vars_list)}
n = len(vars_list)
canReach = [[False]*n for _ in range(n)]

for child, parents in g.items():
    c = idx[child]
    for parent in parents:
        p = idx[parent]
        canReach[p][c] = True

for k in range(n):
    for i in range(n):
        if canReach[i][k]:
            row_i = canReach[i]
            row_k = canReach[k]
            for j in range(n):
                if row_k[j]:
                    row_i[j] = True

def would_cycle(parent: str, child: str) -> bool:
    u, v = idx[parent], idx[child]
    if u == v:
        return True
    return canReach[v][u]

def commit_edge_and_update(parent: str, child: str):
    u, v = idx[parent], idx[child]
    # ancestors of parent
    anc = [a for a in range(n) if a == u or canReach[a][u]]
    # descendants of child
    desc = [d for d in range(n) if d == v or canReach[v][d]]
    for a in anc:
        row = canReach[a]
        for d in desc:
            row[d] = True
    g[child].append(parent)

best_path = f'{case}.gph'
score_components = {}
best_score = k2_score(data, g, state_names, score_components)
print(best_score)
```

```
    score = best_score

    children_best_delta  = {c: 0.0  for c in vars_list}
    children_best_parent = {c: None for c in vars_list}

    recompute = set(vars_list)

    while True:
        best_edge = None
        best_edge_score = 0.0

        for child in vars_list:
            if child in recompute:
                local_best_delta = 0.0
                local_best_parent = None

                for parent in vars_list:
                    if parent == child or parent in g[child] or would_cycle(
parent, child):
                        continue

                    candidate = recompute_k2(data, g, parent, child,
state_names, score_components)
                    if candidate > local_best_delta:
                        local_best_delta = candidate
                        local_best_parent = parent

                children_best_delta[child]  = local_best_delta
                children_best_parent[child] = local_best_parent

                print(f'Scanned child {child}. Best edge delta: {
children_best_delta[child]:.6f}')

            # pick global best using cached (or just-updated) values
            if children_best_delta[child] > best_edge_score:
                best_edge_score = children_best_delta[child]
                best_parent = children_best_parent[child]
                best_edge = (best_parent, child) if best_parent is not None
else None


        if best_edge is not None and best_edge_score > 0:
            p, c = best_edge

            if would_cycle(p, c):
                # This can happen if constraints changed between compute and
commit.
                # Invalidate this child's cached choice and recompute next
loop.
                children_best_delta[c] = 0.0
                children_best_parent[c] = None
                # Recompute only this child next iteration
            else:
                commit_edge_and_update(p, c)
                score_components[c] += best_edge_score
                score += best_edge_score
                print(f'New Best Score: {score}')
                save_graph(g, 'large_cache.gph')
            recompute = {c}

        else:
            break

    print(f'Final best score: {score}')
    if score > best_score:
        save_graph(g, best_path)




if __name__ == "__main__":
    cases = ['small', 'medium', 'large']
    for case in cases:
        k2_search(case)
    # data = pd.read_csv("data/medium.csv")
    # file = open('medium.gph', 'r')
    # g = build_graph_from_edges(file.read(), data)

    # state_names = infer_state_names(data)
    # data = ensure_categorical(data, state_names)
    # print(g)
    # score_components = dict()
    # score = k2_score(data, g, state_names, score_components)
    # print(score)
```

So that was all the actual computation code, and for fun, here is the code I wrote to draw the graphs too

```
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd

from project1.project1 import build_graph_from_edges

def load_graph(path):

    graph_file = open(f'{path}.gph', 'r')
    data = pd.read_csv(f'data/{path}.csv')

    graph_data = graph_file.read()
    gDict = build_graph_from_edges(graph_data, data)

    G = nx.DiGraph()
    G.add_nodes_from(gDict.keys())

    for child in gDict.keys():
        for parent in gDict[child]:
            G.add_edge(parent, child)

    return G

def draw_graph(path):
    G = load_graph(path)
    nx.write_latex(G, f'{path}.tex', caption=f'{path}', as_document=True)
    # nx.draw_circular(G, with_labels=True, font_weight='bold')
    # plt.show()

if __name__ == "__main__":
    cases = ['small', 'medium','large']
    for case in cases:
        draw_graph(case)
```