# Project 1: Bayesian Structure Learning

**Matthew Nguyen** MBFLY@STANFORD.EDU
*AA228, Stanford University*

## 1. Algorithm Description

The algorithm used primarily revolves around the use of a Chow-Liu tree. The goal of the algorithm is to find the best tree-structured bayesian network in which each node has more than one parent and maximizes likelihood of observed data.
The Chow-Liu tree uses mutual information to measure how strongly each pair of variables is dependent on one another. In order to do this, it has to compute the mutual information for every variable pair, building a weighted graph with variables as nodes and MI values as edge weights. It then finds the maximum spanning tree by choosing a set of edges that connects all of the nodes with the largest total MI before picking a root and building the edges from there. The objective function for Chow-Liu is given by:
$KL(P^*|P_T) = \Sigma_x P^*(x) \log(\frac{P^*(x)}{P_T(x)})$
Where $KL$ is a Kullback-Leibler divergence, and the true distribution is $P^*$
If we define: $I(X_I^*, X_j) = \Sigma_{x_i,x_j} = P(x_i, x_j) \log \frac{P(x_i,x_j)}{P(x_i)P(x_j)}$
To find the best tree, $T^*$, we use:
$T^* = argmax_{T \in trees} \Sigma_{(i,j)\in T} I(X_i; X_j)$

However, one major limitation of Chow-Liu is that each variable can only have one parent. Since we want to form a generalized bayesian network, we need to add some refinement to reach a more generalized bayesian network.
The next refinement step has three possible moves: introducing a dependency, removing a dependency, or reversing a dependency. After doing so, we compute the score as outlined. Then we run this optimization step a few times, recomputing the DAG until we find some optimum or run out of compute time as the algorithm is limited in processing power.
To add some additional efficiency to the code, the algorithm only tests edges with high MI in previous steps to reduce re-computing, it caches local scores, incrementally rescores parts of the DAG rather than the whole DAG, and it only randomly samples some candidate pairs as the DAG is optimized.

## 2. Graphs

### 2.1 small.csv

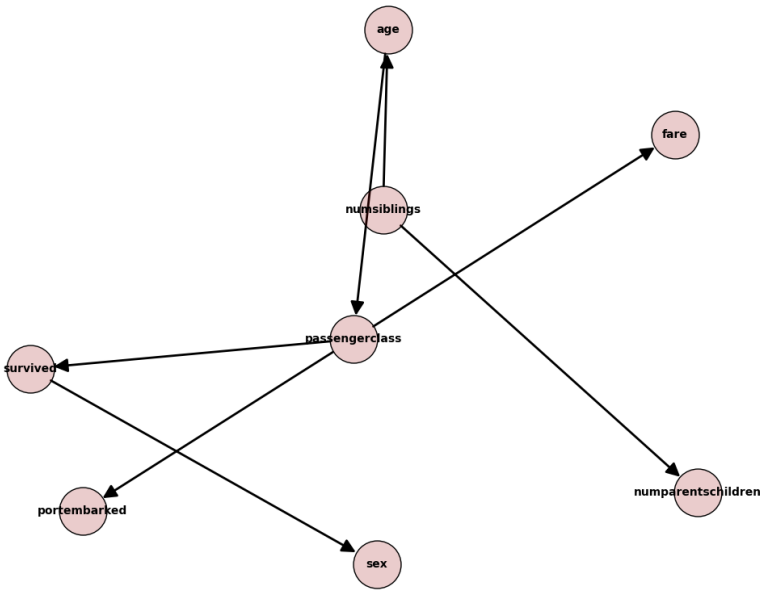


Figure 1: Small

### 2.2 medium.csv

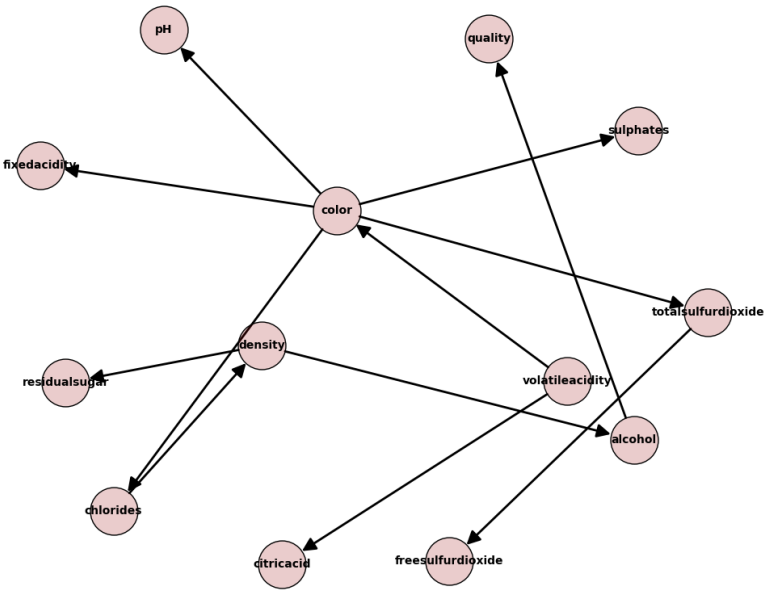


Figure 2: Medium

### 2.3 large.csv

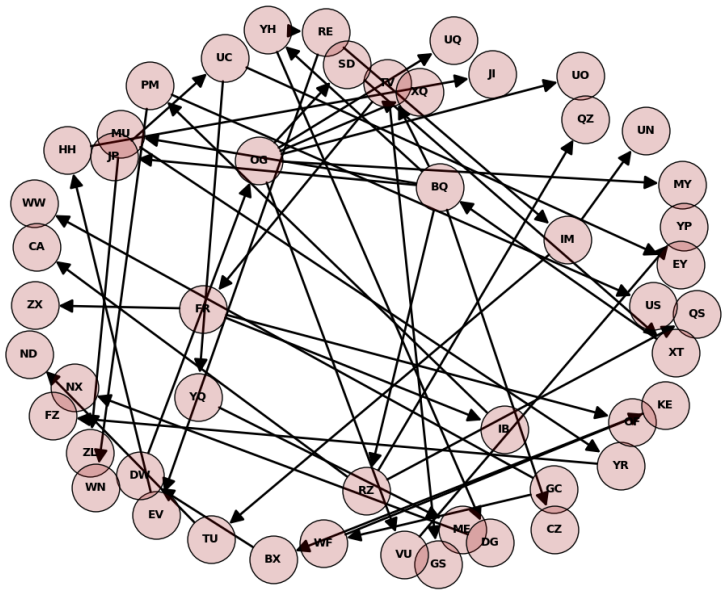


Figure 3: Large

## 3. Code

### 3.1 Project1.py

```
import sys

import pandas as pd
import numpy as np
import itertools
import networkx as nx
import random
from math import log
from functools import lru_cache

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):
    data = pd.read_csv(infile)
    nodes = list(data.columns)
    np.random.seed(0)
    random.seed(0)
    n = len(nodes)

    def mutual_information(x, y):
        joint = data.groupby([x, y]).size().div(len(data))
        px = data[x].value_counts(normalize=True)
        py = data[y].value_counts(normalize=True)
        mi = 0.0
        for (xi, yi), pxy in joint.items():
            mi += pxy * log((pxy + 1e-9) / ((px[xi] * py[yi]) + 1e-9), 2)
        return mi

    edges = []
    for a, b in itertools.combinations(nodes, 2):
        w = mutual_information(a, b)
        edges.append((a, b, w))

    edges.sort(key=lambda x: x[2], reverse=True)
    top_pairs = [(a, b) for a, b, _ in edges[: max(5, len(edges)//4)]]

    #ChowLiu Tree
    G = nx.Graph()
    G.add_weighted_edges_from(edges)
    T = nx.maximum_spanning_tree(G, weight="weight")

    root = nodes[0]
```

```
dag = nx.DiGraph()
dag.add_nodes_from(nodes)
for parent, child in nx.bfs_edges(T, source=root):
    dag.add_edge(parent, child)

print(f"ChowLiu tree with {len(dag.edges())} edges.")

#Cached Local Scoring
@lru_cache(maxsize=None)
def local_score(node, parents_tuple):
    parents = list(parents_tuple)
    if not parents:
        probs = data[node].value_counts(normalize=True)
        return np.sum(np.log(probs + 1e-6))
    joint = data.groupby(parents + [node]).size()
    parent_counts = data.groupby(parents).size()
    s = 0.0
    for idx, count in joint.items():
        parent_total = parent_counts[idx[0]] if len(parents)==1 else
parent_counts[idx[:-1]]
        p = count / parent_total
        s += np.log(p + 1e-6)
    penalty = 0.05 * len(parents) * np.log(len(data) + len(nodes))
    return s - penalty

def full_score(graph):
    return sum(local_score(node, tuple(graph.predecessors(node))) for
node in graph.nodes)

best_graph = dag.copy()
best_score = full_score(best_graph)
print(f"Score: {best_score:.3f}")

patience = 4
patience_counter = 0
iteration = 0
max_moves = 100

while patience_counter < patience:
    improved = False
    best_candidate_graph = None
    best_candidate_score = best_score

    move_candidates = random.sample(top_pairs, min(len(top_pairs),
max_moves))

    for a, b in move_candidates:
        for action in ["add", "remove", "reverse"]:
            g = best_graph.copy()
```

```
                if action == "add" and not g.has_edge(a, b):
                    g.add_edge(a, b)
                elif action == "remove" and g.has_edge(a, b):
                    g.remove_edge(a, b)
                elif action == "reverse" and g.has_edge(a, b):
                    g.remove_edge(a, b)
                    g.add_edge(b, a)
                else:
                    continue

                if not nx.is_directed_acyclic_graph(g):
                    continue

                delta = 0.0
                if g.has_node(b):
                    new_parents = tuple(g.predecessors(b))
                    old_parents = tuple(best_graph.predecessors(b))
                    delta += local_score(b, new_parents) - local_score(b,
    old_parents)
                if action == "reverse" and g.has_node(a):
                    new_parents = tuple(g.predecessors(a))
                    old_parents = tuple(best_graph.predecessors(a))
                    delta += local_score(a, new_parents) - local_score(a,
    old_parents)
                candidate_score = best_score + delta

                if candidate_score > best_candidate_score + 1e-6:
                    best_candidate_score = candidate_score
                    best_candidate_graph = g

        if best_candidate_graph is not None:
            best_graph = best_candidate_graph
            best_score = best_candidate_score
            iteration += 1
            patience_counter = 0
            improved = True
            print(f"{iteration}: score={best_score:.3f}, edges={len(best_graph
    .edges())}")
        else:
            patience_counter += 1


    print(f"Final edges = {len(best_graph.edges())}, Final score = {
    best_score:.3f}")


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()
```

### 3.2 graphing code

```
import sys
import networkx as nx
import matplotlib.pyplot as plt


def visualize_gph(filename):
    edges = []
    with open(filename, "r") as f:
        for line in f:
            parts = [x.strip() for x in line.strip().split(",")]
            if len(parts) == 2:
                edges.append(tuple(parts))

    G = nx.DiGraph(edges)

    pos = nx.spring_layout(G, seed=42, k=1.5, scale=3.0)

    plt.figure(figsize=(10, 8))
    nx.draw(
        G, pos,
        with_labels=True,
        node_color=(0.6, 0, 0, 0.2),
        edgecolors="black",
        node_size=1800,
        arrows=True,
        arrowsize=25,
        width=2,
        arrowstyle='-|>',
        font_weight="bold",
        font_size=10
    )

    plt.axis("off")
    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    if len(sys.argv) != 2:
```

```
        sys.exit(1)
    visualize_gph(sys.argv[1])
```