Project 1: Bayesian Structure Learning

Anthony Weng ad2weng@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

For this project, I implemented and employed the *hill climb search algorithm*. Beginning with an edgeless graph, neighboring DAGs of the current DAG[1] are generated and scored. The highest-scoring neighbor is selected, and its score compared to the score of the source DAG. If $s_{neighbor} - s_{source} > 0$, then this highest-scoring neighbor becomes the new source DAG, and the process is repeated. Once this score comparison condition is no longer satisfied, the most recent source DAG is returned.

I also built various heuristics into my search algorithm, all of which can be toggled on or off and some of which can be further parameterized. These heuristics either seek to move the search algorithm convergence away from settling into a local optima too quickly or to limit the search space to make the problem more computationally tractable. A description of the implemented heuristics follows:

1. **Simulated annealing:**
    - *Description:* With a probability computed as the inverse of the square root of the loop number, the best scoring neighbor DAG will be ignored and replaced with a randomly chosen neighbor DAG for the next search iteration.
    - *Purpose:* Discourage early search convergence on a local optima.
    - *In the code:* toggled on/off by the `USE_ANNEALING` constant.
2. **TABU list:**
    - *Description:* Maintain a list of the k-most recent DAGs used as the source graph for neighbor generation in previous search iterations. If the best-scoring neighbor from a later search iteration is a member of this list, ignore it and select a random neighbor DAG for the next search iteration instead.
    - *Purpose:* Discourage early search convergence on a local optima.
    - *In the code:* toggled on/off by the `USE_TABU` constant, with `TABU_LIST` being a global variable of size `TABU_LIST_MAX_SIZE` (a constant).
3. **Max parents:**
    - *Description:* A constant specifying the maximum number parents any node is allowed to have in the generated neighbor graphs we will explore and score. If the neighbor generation function finds an edge operation to cause a node to exceed

1. I adhere to the convention that a DAG neighborhood consists of the graphs which reachable within one edge operation-i.e., an edge addition, deletion, or reversal-of the source DAG.

this constant, it will not include the associated neighbor graph in the list of all to-be-scored neighbors of the source DAG.

- *Purpose:* Limit search space to ensure tractability.
- *In the code:* set with the `MAX_PARENTS` constant (to disable it, simply set this constant to a very large real number).

4. **Max neighbors:**

   - *Description:* A list with three constants specifying the maximum number of neighbor graphs to be generated by each type of edge operation in the neighbor generation process. To ensure some randomness in which neighbor graphs get generated, the source graph node and edge lists (which are iterated through to generate neighboring DAGs) are randomly shuffled prior to neighbor generation.
   - *Purpose:* Limit search space to ensure tractability.
   - *In the code:* toggled on/off by the `USE_MAX_NEIGHBORS` constant, with the `MAX_NEIGHBORS` global variable being a list with three entries, each one specifying the maximum number of neighbors to be generated from a source DAG for each possible edge operation in the order: [# edge add, # edge subtract, # edge flip]. A specific neighbor generation function, `generate_subset_neighbors()`, separate from the otherwise-employed `generate_neighbors()`, is also implemented and conducts the random shuffling described above.

1.1 Algorithm Runtimes

- File: `small.csv`
  - Runtime: 19.46 seconds.
  - Options: YES annealing, YES TABU list (size = 1), NO max parents, NO max neighbors.
- File: `medium.csv`
  - Runtime: 403.01 seconds.
  - Options: YES annealing, YES TABU list (size = 1), NO max parents, NO max neighbors.
- File: `large.csv`
  - Runtime: 7193.58 seconds.
  - Options: YES annealing, YES TABU list (size = 1), YES max parents (3), YES max neighbors (maximum of 25 from each type of edge operation).

2. Graphs

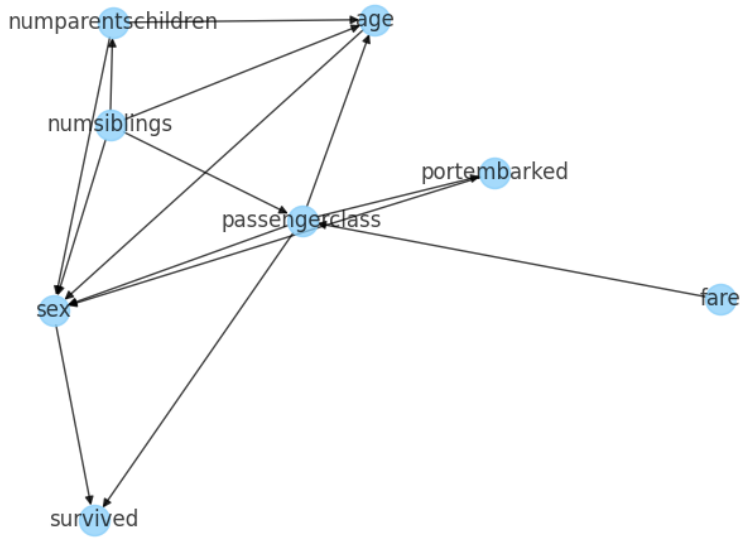

Figure 1: Graph visualization for Bayesian network learned from `small.csv`.

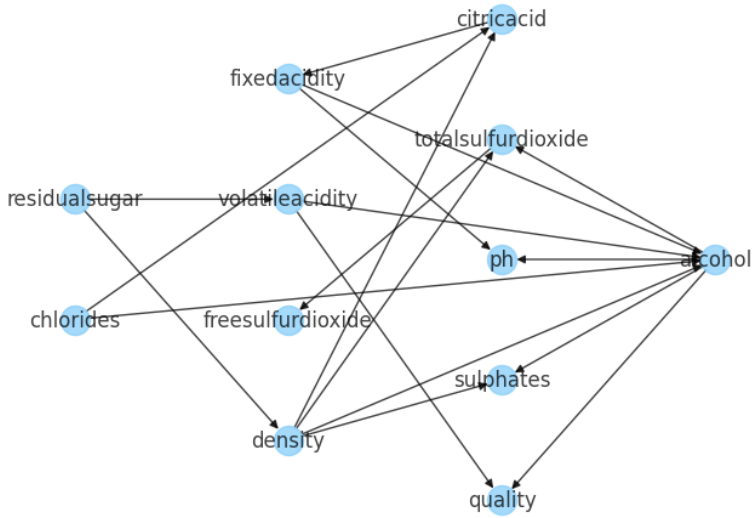

Figure 2: Graph visualization for Bayesian network learned from `medium.csv`.

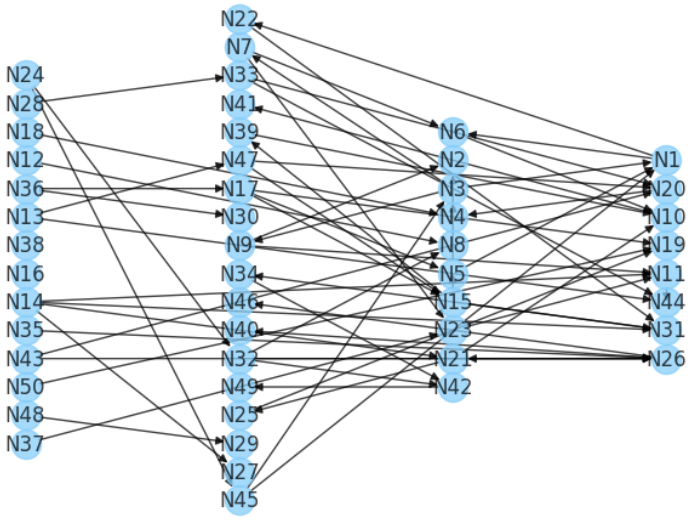

Figure 3: Graph visualization for Bayesian network learned from `large.csv`.

3. Code

```
#######################
###     IMPORTS     ###
#######################
import sys
import time
import pandas as pd
import numpy as np
import math
import random as rd
import matplotlib.pyplot as plt
import scipy.special as sp
import networkx as nx
```

```
###################
###   GLOBALS   ###
###################
# VAR/SETTING:              # DESCRIPTION:
MIN_VAL = 1                 # constant describing the minimum value any
    variable in the graph can assume.
max_vals = None             # dict. mapping: var_idx -> maximum value/num.
    potential values (r_i).
idx2names = {}              # dict. mapping: var_idx -> var_name.P
MAX_PARENTS = 100           # constant controlling the number of parents any
    node in a given neighbor graph can have before the search ignores that
    graph. to disable, set MAX_PARENTS = large constant.
USE_TABU = True             # constant controlling whether to use the
    TABU_LIST or not
TABU_LIST = []              # list of recently visited edge sets which will
    steer the hill climb search away from re-visiting recently seen DAGs.
TABU_LIST_MAX_SIZE = 1      # constant describing the maximum size we want
    the above mentioned TABU_LIST to be.
USE_ANNEALING = True        # constant controlling whether or not to use
    simulatned annealing in the hill climb search.
USE_MAX_NEIGHBORS = False    # constant controlling whether or not to limit
    the number of neighbors generated during each iteration of the hill climb
    search.
MAX_NEIGHBORS = [25,25,25]  # list specifying the maximum number of neighbors
     to be generated from each operation (i.e., [# edge add, # edge subtract,
    # edge flip]) if USE_MAX_NEIGHBORS is set to 'True.'
rd.seed(238)                # fixed seed number to ensure replicability
    during testing; feel free to comment out!
```

```
def init_graph(inputfilename):
    """
    Initializes a DAG, maximum values dictionary, idx2names dict, and
    dataframe from the data found at 'inputfilename'.
```

```
    """
    # read in the data file:
    df = pd.read_csv(inputfilename)
    # determine max value for each variable in the network:
    global max_vals
    max_vals = df.max().to_dict()
    # initialize: an empty (directed) graph
    g = nx.DiGraph()
    # for each variable in the data file, init. a graph node and log its
    index-name association:
    for idx, var_name in enumerate(list(max_vals.keys())):
        g.add_node(idx)
        idx2names[idx] = var_name
        max_vals[idx] = max_vals.pop(var_name)
    # return the initialized graph, complete set of nodes, and observations
    dataframe:
    return g, set(g.nodes()), df
```

```
def generate_subset_neighbors(g: nx.DiGraph, original_nodes: set):
    """
    Generates and returns the edge sets of a (partially) random subset of
    DAGs which are within one operation of the supplied DAG, g.
    An operation can be an: edge addition; edge deletion; edge reversal; on a
     **SINGLE** edge.
    """

    # retrieve the current set of nodes & edges (some as lists and some as
    sets for shuffling purposes):
    source_edge_set, source_node_list, source_edge_list = set(g.edges()),
    list(g.nodes()), list(g.edges())
    # counters for the number of neighbors generated from each operation:
    add_neighbors, subtract_neighbors, flip_neighbors = 0, 0, 0
    # init. the ret. val:
    neighbor_edge_sets = []

    # randomly permutate the containers which will be iterated through to
    generate neighbors:
    rd.shuffle(source_node_list)
    rd.shuffle(source_edge_list)

    # first set of ret vals: graphs within one edge addition of g
    for node in source_node_list:
        # limiting the number of neighbors which can be generated through the
     'edge add' operator:
        if add_neighbors < MAX_NEIGHBORS[0]:
            # rule: for a node n, you cannot add an arc from n to any of n's
    ancestor or n itself without creating a cycle.
            node_ancestors = nx.ancestors(g, node).union({node})
            valid_new_neighbors = list(original_nodes - node_ancestors)
```

```
            # shuffle the valid_new_neighbors too:
            rd.shuffle(valid_new_neighbors)

            for new_neighbor in valid_new_neighbors:
                # need to double check this b/c more than MAX_NEIGHBORS[0]
    neighbors can be added from a single node's 'edge add' neighbors:
                if add_neighbors < MAX_NEIGHBORS[0]:
                    # omit any neighboring DAGs for who adding an edge would
    cause the edge-receiving node to have too many parents:
                    if len(list(g.predecessors(new_neighbor))) < MAX_PARENTS:
                        # further omit any neighbors which result from adding
     edges which already exist in the source graph:
                        if (node, new_neighbor) not in source_edge_set:
                            neighbor_edge_set = source_edge_set.union(
                                {(node, new_neighbor)})
                            neighbor_edge_sets.append(neighbor_edge_set)
                            add_neighbors += 1
                else:
                    break
        else:
            break

    # second set of ret vals: graphs within one edge deletion of g
    for edge in source_edge_list:
        # limiting the number of neighbors which can be generated through the
     'edge subtract' operator:
        if subtract_neighbors < MAX_NEIGHBORS[1]:
            neighbor_edge_set = source_edge_set.copy()
            neighbor_edge_set.remove(edge)
            neighbor_edge_sets.append(neighbor_edge_set)
            subtract_neighbors += 1
        else:
            break

    # third (and final) set of ret vals: graphs within one edge reversal of g
    for edge in source_edge_list:
        # limiting the number of neighbors which can be generated through the
     'edge flip' operator:
        if flip_neighbors < MAX_NEIGHBORS[2]:
            neighbor_edge_set = source_edge_set.copy()
            neighbor_edge_set.remove(edge)
            flipped_edge = edge[::-1]
            node_with_potential_new_parent = flipped_edge[0]

            # omit neighboring DAG if flipping this edge would cause the new
    edge-receiving node to have too many parents:
            if len(list(g.predecessors(node_with_potential_new_parent))) + 1
    < MAX_PARENTS:
                neighbor_edge_set.add(flipped_edge)
```

```
                # initialize a new graph with the same nodes as the source
    DAG & one edge reversed:
                potential_neighbor_graph = nx.DiGraph(incoming_graph_data=
    neighbor_edge_set)
                potential_neighbor_graph.add_nodes_from(original_nodes)

                # check if the neighboring graph is a DAG:
                if nx.is_directed_acyclic_graph(potential_neighbor_graph):
                    neighbor_edge_sets.append(set(potential_neighbor_graph.
    edges()))
                flip_neighbors += 1
            else:
                break

    return neighbor_edge_sets
```

```
def generate_neighbors(g: nx.DiGraph, original_nodes: set):
    """
    Generates and returns the edge sets of all DAGs which are within one
    operation of the supplied DAG, g.
    An operation can be an: edge addition; edge deletion; edge reversal; on a
     **SINGLE** edge.
    """
    # retrieve the current set of nodes & edges:
    source_node_set, source_edge_set = set(g.nodes()), set(g.edges())
    # init. the ret. val:
    neighbor_edge_sets = []

    # first set of ret vals: graphs within one edge addition of g
    for node in source_node_set:
        # rule: for a node n, you cannot add an arc from n to any of n's
    ancestor or n itself without creating a cycle.
        node_ancestors = nx.ancestors(g, node).union({node})
        valid_new_neighbors = original_nodes - node_ancestors

        for new_neighbor in valid_new_neighbors:
            # omit any neighboring DAGs for who adding an edge would cause
    the edge-receiving node to have too many parents:
            if len(list(g.predecessors(new_neighbor))) < MAX_PARENTS:
                # further omit any neighbors which result from adding edges
    which already exist in the source graph:
                if (node, new_neighbor) not in source_edge_set:
                    neighbor_edge_set = source_edge_set.union(
                        {(node, new_neighbor)})
                    neighbor_edge_sets.append(neighbor_edge_set)

    # second set of ret vals: graphs within one edge deletion of g
    for edge in source_edge_set:
        neighbor_edge_set = source_edge_set.copy()
```

```
            neighbor_edge_set.remove(edge)
            neighbor_edge_sets.append(neighbor_edge_set)

        # third (and final) set of ret vals: graphs within one edge reversal of g
        for edge in source_edge_set:
            neighbor_edge_set = source_edge_set.copy()
            neighbor_edge_set.remove(edge)
            flipped_edge = edge[::-1]
            node_with_potential_new_parent = flipped_edge[0]

            # omit neighboring DAG if flipping this edge would cause the new edge
        -receiving node to have too many parents:
            if len(list(g.predecessors(node_with_potential_new_parent))) + 1 <
        MAX_PARENTS:
                neighbor_edge_set.add(flipped_edge)

                # initialize a new graph with the same nodes as the source DAG &
        one edge reversed:
                potential_neighbor_graph = nx.DiGraph(incoming_graph_data=
        neighbor_edge_set)
                potential_neighbor_graph.add_nodes_from(original_nodes)

                # check if the neighboring graph is a DAG:
                if nx.is_directed_acyclic_graph(potential_neighbor_graph):
                    neighbor_edge_sets.append(set(potential_neighbor_graph.edges
        ()))

        return neighbor_edge_sets
```

```
################################################
###      BAYESIAN SCORE FUNCTION HERE:      ###
################################################

def bayesian_score(g_edge_set: set, original_nodes: set, data: pd.DataFrame):
    """
    Returns the Bayesian score for the DAG specified by g_edge_set and
    g_node_set over the samples in 'data'.
    CITATION: I adapt the code presented in the textbook (Algorithms for
    Decision Making) on pgs. 75, 81, and 98.
    """
    # create the graph from the specified edge set:
    g = nx.DiGraph(incoming_graph_data=g_edge_set)
    g.add_nodes_from(original_nodes)

    # generate and update the matrix M where m_ijk corresponds to the count
    of the observations where
    # variable i assumes its k'th value and its parents are in their j'th
    instantiation.
    def compute_M_counts(g: nx.DiGraph):
```

```
    r = [max_vals[var_idx] for var_idx in max_vals.keys()]
    q = [math.prod([r[parent_var_idx] for parent_var_idx in nx.DiGraph.
predecessors(g, var_idx)]) for var_idx in max_vals.keys()]
    M = [np.zeros(shape=(q[var_idx], r[var_idx])) for var_idx in max_vals
.keys()]

    for sample in data.itertuples(index=False):
        for var_idx in idx2names.keys():
            k = sample[var_idx] - 1  # b/c Python is 0-indexed
            j = 0  # should be one, but what do you know - Python is 0-
indexed
            parents = list(nx.DiGraph.predecessors(g, var_idx))

            if len(parents) > 0:
                parent_max_vals = tuple([max_vals[parent] for parent in
parents])
                coordinate = tuple([sample[parent] - 1 for parent in
parents])
                j = np.ravel_multi_index(coordinate, parent_max_vals)

            M[var_idx][j][k] += 1

    return M

# generate a matrix of the same shape as M which captures our uniform
prior belief (e.g., a_ijk = 1 for all ijk)
def gen_prior_counts(g: nx.DiGraph):
    r = [max_vals[var_idx] for var_idx in max_vals.keys()]
    q = [math.prod([r[parent_var_idx] for parent_var_idx in nx.DiGraph.
predecessors(g, var_idx)]) for var_idx in max_vals.keys()]
    a = [np.ones(shape=(q[var_idx], r[var_idx])) for var_idx in max_vals.
keys()]

    return a

# compute the bayesian score compute for variable x_i:
def bayesian_score_component(M_i, a_i):
    p  = np.sum(sp.gammaln(a_i + M_i ))
    p -= np.sum(sp.gammaln(a_i))
    p += np.sum(sp.gammaln(np.sum(a_i, axis=1)))
    p -= np.sum(sp.gammaln(np.sum(a_i, axis=1) + np.sum(M_i, axis=1)))

    return p

M, a = compute_M_counts(g), gen_prior_counts(g)
bayesian_score = sum([bayesian_score_component(M[var_idx], a[var_idx])
for var_idx in max_vals.keys()])

return bayesian_score
```

```
##############################################
###   STRUCTURE LEARNING ALGORITHM HERE:   ###
##############################################

def hill_climb_search(g: nx.DiGraph, original_nodes: set, data: pd.DataFrame)
    :
    """
    Conducts a simple greedy hill climb search to find a locally optimal DAG
    to describe the data in 'data'.
    """
    # initialize ret val & search parameters:
    est_dag, curr_score, score_improvement, loop_counter = g, bayesian_score(
    g, original_nodes, data), np.Inf, 0

    # loop until convergence:
    while score_improvement > 0:
        # increment the loop counter & print it out:
        loop_counter += 1
        print("I'm currently on search loop number: {ln}".format(ln=
    loop_counter))

        # gen. a list of the edge sets of all DAGs within one operation of
    our current DAG:
        if (USE_MAX_NEIGHBORS == True):
            neighbor_dag_edge_sets = generate_subset_neighbors(est_dag,
    original_nodes)
        else:
            neighbor_dag_edge_sets = generate_neighbors(est_dag,
    original_nodes)

        # score the neighboring DAGs by Bayesian score:
        neighbor_dag_and_scores = [(bayesian_score(edge_set, original_nodes,
    data), edge_set)
                                   for edge_set in neighbor_dag_edge_sets]
        best_neighbor_dag = max(neighbor_dag_and_scores, key=lambda a: a[0])

        # retrieve the neighbor with the highest Bayesian score:
        best_neighbor_score, best_neighbor_edge_set = best_neighbor_dag[0],
    best_neighbor_dag[1]

        # check to see if there's a score improvement:
        score_improvement = best_neighbor_score - curr_score

        # ignore score and move to a random neighbor if the best neighboring
    DAG is in the TABU_LIST:
        if (USE_TABU == True):
            if (len(TABU_LIST) >= TABU_LIST_MAX_SIZE):
                if best_neighbor_edge_set in TABU_LIST:
```

```
                    neighbor_idx = rd.randint(0, len(neighbor_dag_and_scores)
 - 1)
                    alt_dag = nx.DiGraph(incoming_graph_data=
neighbor_dag_and_scores[neighbor_idx][1])
                    alt_dag.add_nodes_from(original_nodes)
                    est_dag, curr_score = alt_dag, neighbor_dag_and_scores[
neighbor_idx][0]
                else:
                    TABU_LIST.pop(0)
                    TABU_LIST.append(best_neighbor_edge_set)
            else:
                TABU_LIST.append(best_neighbor_edge_set)

        # ignore score and move to a random neighbor if the simulated
annealing check passes:
        if (USE_ANNEALING == True):
            # annealing probability decreases with inverse of sqrt of loop
number
            if rd.random() < (1 / math.sqrt(loop_counter)):
                neighbor_idx = rd.randint(0, len(neighbor_dag_and_scores) -
1)
                alt_dag = nx.DiGraph(incoming_graph_data=
neighbor_dag_and_scores[neighbor_idx][1])
                alt_dag.add_nodes_from(original_nodes)
                est_dag, curr_score = alt_dag, neighbor_dag_and_scores[
neighbor_idx][0]

        # if there's a score improvement, init. another search iteration with
 the improved DAG:
        if score_improvement > 0:
            best_alt_dag = nx.DiGraph(incoming_graph_data=
best_neighbor_edge_set)
            best_alt_dag.add_nodes_from(original_nodes)
            est_dag, curr_score = best_alt_dag, best_neighbor_score

    # return the (local) optimal DAG:
    return est_dag, curr_score
```

```
def main():
    """
    Entry point of program.
    """
    # argument parsing:
    if len(sys.argv) != 3:
        raise Exception("usage: python project1.py <infile>.csv <outfile>.gph
")
    inputfilename, outputfilename = sys.argv[1], sys.argv[2]

    # if things parse well, we're off to the races! make sure to time things:
```

```
    start_time = time.time()

    # initialize a graph with a node for each variable and no edges:
    g, nodes, data = init_graph(inputfilename)

    # estimate a network structure using hill climb search:
    est_dag, final_bn_score = hill_climb_search(g, nodes, data)

    # output the learned DAG:
    write_gph(est_dag, idx2names, outputfilename)

    # sanity check + results! what's the score of the final estimated DAG?
    how long did it take to find it?
    # and is the DAG a DAG? and draw the final (hopefully) DAG too!
    print("\n" + "=" * 25 + "\n" + " " * 5 + "OUTPUT SUMMARY:" + "\n" + "=" *
     25)
    print("Final score is: {score}".format(score=final_bn_score))
    print("It took me: --- %.2f --- seconds to find the final graph." % (time
    .time() - start_time))
    print("The final graph is a DAG: " + str(nx.is_directed_acyclic_graph(
    est_dag)) + "\n")
    plt.figure(1)
    nx.draw(est_dag, labels=idx2names, with_labels=True, node_color="
    lightskyblue", alpha=0.75)
    plt.show()

if __name__ == "__main__":
    main()
```