Project 1: Bayesian Structure Learning

AJ Phillips andrewjp@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

The algorithm implemented here to accomplish Bayesian structure learning was a variant of genetic local search with initial populations determined by either K2 searches with random variable orderings or edgeless graphs.

1.1 Computational Optimizations

One critical insight enabling much quicker computation of Bayesian scores for DAGs was that the only components of the Bayesian score that change for a basic graph operation are those components associated with variables that have changed parents. All other components of the Bayesian score remain the same. Therefore, recomputing only components of the Bayesian score corresponding to variables with changed parents was sufficient to compare the Bayesian score of neighboring DAGs to one another. This was implemented in code by storing the Bayesian score components and parents for each variable of the current DAG in data structures, then evaluating proposed basic graph operations only on the basis of variables with proposed parent changes. For introducing or removing an edge, this corresponded to evaluation on a single variable; for reversing an edge, this corresponded to evaluation on two variables.

Applying local search to each individual in the population after genetic recombination naturally lends itself to parallel computation, so multiprocessing was implemented to enable more rapid iteration through the local search stage of each generation.

1.2 K2 Search

K2 search was implemented as described in Chapter 5 of the textbook with the computational optimization discussed above. The hyperparameter `MAX_PARENTS` was used to impose an upper bound on the number of parents for any one variable to reduce the required computation.

1.3 Genetic Search

Each DAG was represented as a ternary string where the index of the digit in the string encoded a potential edge location and a 0, 1, or 2 encoded no edge, an edge from the first to the second variable, or an edge from the second to the first variable, respectively. One key aspect of genetic search was the recombination stage during which individuals in the population reproduced at varied rates according to their scores. After trying several different methods, a k-way tournament selection procedure was implemented, whereby `TOURN_SIZE` individuals were selected from the population at random and the individual with the highest Bayesian score was selected to become a parent. The two parents selected for reproduction were recombined by swapping strings at a randomly selected crossover point. Recombined individuals with cycles were invalid and replaced by new recombined individuals. The hyperparameters `generations`, `pop_size`, and `mut_prob` determined the total number of generations, the size of the population, and the probability that an individual would undergo a random mutation at each of its genes, respectively.

1.4 Local Search

A variation of local search was implemented for `k_max` total iterations. During every iteration, each edge in the DAG would be considered with probability `1-skip_prob`. All basic graph operations (introducing an edge, removing an edge, reversing an edge) were evaluated for all considered edges. If none of the operations improved the Bayesian score, no action was taken and the algorithm moved to the next iteration. Otherwise, the operation that most improved the Bayesian score was selected to update the DAG. This formulation was a hybrid of classic local search and opportunistic local search, providing the flexibility to reduce runtime at the expense of a less thorough neighborhood search by increasing `skip_prob`. Of course, basic graph operations that introduced cycles into the graph were invalid and excluded from evaluation.

1.5 Tuning Hyperparameters

The hyperparameters were informally selected according to observed performance on the datasets. The selected values for the small and medium datasets were `MAX_PARENTS=4`, `TOURN_SIZE=3`, `generations=10`, `pop_size=10`, `mut_prob=0`, `k_max=(# potential edge locations)`, and `skip_prob=0`. The runtime was prohibitively long applying these hyperparameters to the large dataset, so `skip_prob=0.98` was increased to reduce runtime at the expense of less thorough neighborhood searches and an initial population of edgeless graphs was used instead of K2 search. For completeness, results are also tabulated below applying these modifications to the small and medium datasets.

2. Running Time

Dataset	`skip_prob`	Bayesian Score	Running Time (s)	Running Time (min)
`small.csv`	0	-3794.8555977098	16.7	0.3
`medium.csv`	0	-41953.675685890805	333.7	5.6
`large.csv`	0	N/A	N/A	N/A
`small.csv`	0.98	-3858.837437343546	35.6	0.6
`medium.csv`	0.98	-42087.97332750539	117.2	2.0
`large.csv`	0.98	-404302.78964386426	6366.7	106.1

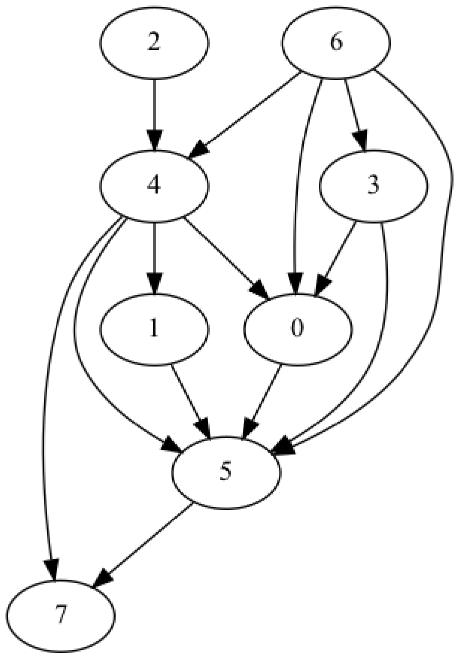

Figure 1: Bayesian network structure for `small.csv`.

3. Graphs

4. Code

```
import sys
import networkx as nx
import time
import numpy as np
import numpy.random as rd
import scipy.special as sp
import pygraphviz
from tqdm import tqdm
from joblib import Parallel, delayed, parallel_backend
from networkx.drawing.nx_agraph import write_dot


IN_DIR = "data"
OUT_DIR = "results"
MAX_PARENTS = 4
TOURN_SIZE = 3


class Variable:
```

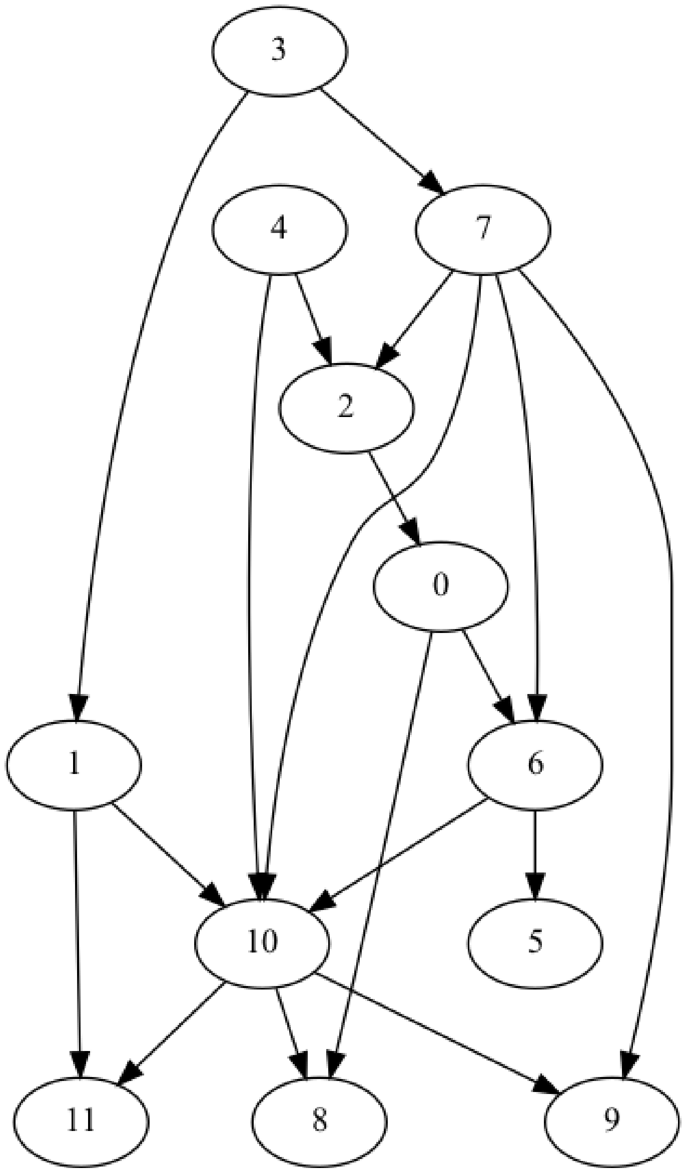

Figure 2: Bayesian network structure for `medium.csv`.

```
    def __init__(self, name, r):
        self.name = name
        self.r = r


def write_gph(dag, idx2names, filename):
    with open(filename, 'w') as f:
        for edge in dag.edges():
```

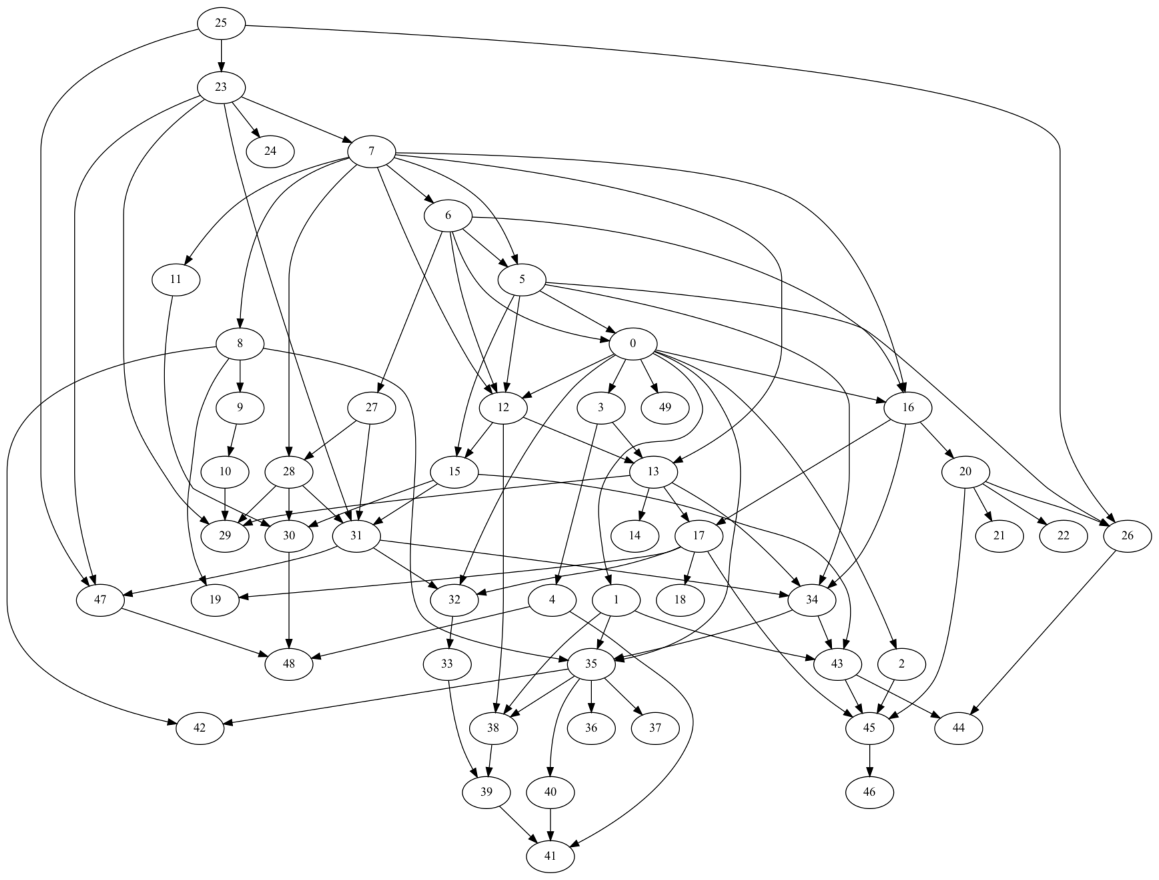

Figure 3: Bayesian network structure for `large.csv`.

```
            f.write("{},{}\n".format(idx2names[edge[0]],
             ↪  idx2names[edge[1]]))


def write_time(time, score, filename):
    with open(filename, 'w') as f:
        f.write(f"Execution time was {time} seconds for a score of
         ↪  {score}")


def compute(infile, outfile):
    # Read CSV
    varnames = np.loadtxt(f"{IN_DIR}/{infile}", dtype=str, delimiter=',',
     ↪  max_rows=1)
    D = np.loadtxt(f"{IN_DIR}/{infile}", dtype=int, delimiter=',',
     ↪  skiprows=1)
    # Build vars
    m,n = D.shape
    r = np.max(D, axis=0)
    vars = np.array([Variable(varnames[i],r[i]) for i in range(n)])
    idx2names = dict(zip(range(n),[vars[i].name for i in range(n)]))

    # Variant of genetic local search
    mut_prob = 0
    skip_prob = 0
    pop_size = 10
    generations = 10
    k_max = int(n*(n-1)/2)
    genetic_local_search(vars, D, r, pop_size, generations, k_max,
    ↪  mut_prob, skip_prob, idx2names, f"final_{outfile}")


def draw_graph(infile, outfile):
    # Read CSV
    varnames = np.loadtxt(f"{IN_DIR}/{infile}", dtype=str, delimiter=',',
    ↪  max_rows=1)
    D = np.loadtxt(f"{IN_DIR}/{infile}", dtype=int, delimiter=',',
    ↪  skiprows=1)

    # Build vars
    m,n = D.shape
    r = np.max(D, axis=0)
    vars = np.array([Variable(varnames[i],r[i]) for i in range(n)])
    idx2names = dict(zip(range(n),[vars[i].name for i in range(n)]))

    # Read graph
    G = nx.DiGraph()
    G.add_nodes_from(range(n))
    edges = np.loadtxt(f"{OUT_DIR}/{outfile}", dtype=str, delimiter=',')
    r,c = edges.shape
    G.add_edges_from([tuple([np.where(edges[i,j]==varnames)[0][0] for j in
    ↪  range(c)]) for i in range(r)])

    # Write dot
    write_dot(G, f"{OUT_DIR}/{outfile[:-4]}.dot")


def genetic_local_search(vars, D, r, pop_size, generations, k_max,
↪  mut_prob, skip_prob, idx2names, outfile):
    st = time.time()
```

```
n = len(vars)
num_edges = int(n*(n-1)/2)
highest_score = -np.inf

# Initial population determined by K2 searches with random variable
↪  orderings
genes = np.zeros((pop_size, num_edges))
orderings = np.zeros((pop_size,n), dtype=int)
for k in range(pop_size):
    ordering = np.arange(n)
    rd.shuffle(ordering)
    orderings[k] = ordering
with parallel_backend('multiprocessing'):
    results =
    ↪  Parallel(n_jobs=pop_size)(delayed(K2_search_optimized)(vars, D,
    ↪  r, orderings[k]) for k in range(pop_size))
    G, bayes = [i for i,j in results], [j for i,j in results]
    for k in range(pop_size):
        genes[k] = interpret_graph(G[k])
    print(f"Highest score for this generation is {np.max(bayes)}")
    highest_score = np.max(bayes)
    write_gph(G[np.argmax(bayes)], idx2names, f"{OUT_DIR}/{outfile}")
    et = time.time()
    elapsed_time = et - st
    write_time(elapsed_time, highest_score,
    ↪  f"{OUT_DIR}/{outfile[:-4]}.t")

# Initial population of edgeless graphs
# genes = np.zeros((pop_size, num_edges))

# Genetic local search
for _ in range(generations):
    bayes = np.zeros(pop_size)

    # Local search
    with parallel_backend('multiprocessing'):
        graphs =
        ↪  Parallel(n_jobs=pop_size)(delayed(build_graph)(n,genes[k])
        ↪  for k in range(pop_size))
        results = Parallel(n_jobs=pop_size)(delayed(local_search)(vars,
        ↪  D, r, graphs[k], k_max, skip_prob) for k in
        ↪  range(pop_size))
        G, bayes = [i for i,j in results], np.array([j for i,j in
        ↪  results])
    print(f"Highest score for this generation is {np.max(bayes)}")
    if np.max(bayes) > highest_score:
        print("Updating results")
        highest_score = np.max(bayes)
        write_gph(G[np.argmax(bayes)], idx2names,
        ↪   f"{OUT_DIR}/{outfile}")
        gene_best = interpret_graph(G[np.argmax(bayes)])
        et = time.time()
        elapsed_time = et - st
        write_time(elapsed_time, highest_score,
        ↪   f"{OUT_DIR}/{outfile[:-4]}.t")

# Genetic recombination
gene_recombinations = np.zeros((pop_size, num_edges))
for k in range(pop_size):

    # k-way tournament selection
    while True:
        gene_selection = np.zeros(2, dtype=int)
        k_individuals = rd.choice(pop_size, TOURN_SIZE,
        ↪   replace=False)
        gene_selection[0] =
        ↪   rd.choice(np.where(bayes==np.max(bayes[k_individuals]))[0])
        k_individuals = rd.choice(pop_size, TOURN_SIZE,
        ↪   replace=False)
        gene_selection[1] =
        ↪   rd.choice(np.where(bayes==np.max(bayes[k_individuals]))[0])

        gene_split = rd.choice(num_edges)

        gene_recombinations[k] = genes[gene_selection[0]]
        gene_recombinations[k,gene_split:] =
        ↪   genes[gene_selection[1],gene_split:]

        if nx.is_directed_acyclic_graph(build_graph(n,
        ↪   gene_recombinations[k])):
            print(f"Gene selection: {gene_selection}")
            print(f"Gene split: {gene_split}")
            break

# Genetic mutations
gene_mutations = np.zeros((pop_size, num_edges))
for k in range(pop_size):
    while True:
        gene_mutations[k] = gene_recombinations[k]
```

```
                for e in range(num_edges):
                    if rd.rand() < mut_prob:
                        gene_mutations[k,e] = rd.choice(3)
                if nx.is_directed_acyclic_graph(build_graph(n,
                ↪  gene_mutations[k])):
                    break

        genes = gene_mutations


def build_graph(n, gene):
    edges = []
    index = 0
    for i in range(n):
        for j in range(i+1,n):
            if gene[index] == 1:
                edges.append((i,j))
            elif gene[index] == 2:
                edges.append((j,i))
            index += 1
    G = nx.DiGraph()
    G.add_nodes_from(range(n))
    G.add_edges_from(edges)
    return G


def interpret_graph(G):
    n = G.number_of_nodes()
    num_edges = int(n*(n-1)/2)
    edges = [e for e in G.edges]
    gene = np.zeros(num_edges)
    index = 0
    for i in range(n):
        for j in range(i+1,n):
            if (i,j) in edges:
                gene[index] = 1
            elif (j,i) in edges:
                gene[index] = 2
            index += 1
    return gene


def local_search(vars, D, r, G, k_max, skip_prob=0):
    n = len(vars)
```

```
# Precomputations
component_scores = bayesian_score_optimized(vars, G, D)
parents = [np.array([j for j in G.predecessors(i)], dtype=int) for i in
↪  range(n)]

# Hybrid of classic local search and opportunistic local search
for k in tqdm(range(k_max)):
    delta_best, i_best, j_best, i_score_best, j_score_best, action =
    ↪  -np.inf, -1, -1, -np.inf, -np.inf, "none"

    for i in range(n):
        for j in range(i+1,n):

            # Determines which edges to randomly consider
            if rd.rand() < skip_prob:
                continue

            if G.has_edge(i,j):
                # Try remove
                G.remove_edge(i,j)
                if nx.is_directed_acyclic_graph(G):
                    new_parents = np.delete(parents[j],
                    ↪  np.where(parents[j]==i)[0][0])
                    j_score_prime = compute_graph_component(j,
                    ↪  new_parents, r, D)
                    delta = j_score_prime - component_scores[j]
                    if delta > delta_best:
                        delta_best, i_best, j_best, j_score_best,
                        ↪  action = delta, i, j, j_score_prime,
                        ↪  "removeij"
                G.add_edge(i,j)

                # Try reverse
                G.remove_edge(i,j)
                G.add_edge(j,i)
                if nx.is_directed_acyclic_graph(G):
                    new_parents = np.delete(parents[j],
                    ↪  np.where(parents[j]==i)[0][0])
                    j_score_prime = compute_graph_component(j,
                    ↪  new_parents, r, D)
                    new_parents = np.concatenate(([j], parents[i]))
                    i_score_prime = compute_graph_component(i,
                    ↪  new_parents, r, D)
                    delta = (i_score_prime + j_score_prime) -
                    ↪  (component_scores[i] + component_scores[j])
                    if delta > delta_best:
                        delta_best, i_best, j_best, i_score_best,
                        ↪  j_score_best, action = delta, i, j,
                        ↪  i_score_prime, j_score_prime, "reverseij"
                G.remove_edge(j,i)
                G.add_edge(i,j)

            elif G.has_edge(j,i):
                # Try remove
                G.remove_edge(j,i)
                if nx.is_directed_acyclic_graph(G):
                    new_parents = np.delete(parents[i],
                    ↪  np.where(parents[i]==j)[0][0])
                    i_score_prime = compute_graph_component(i,
                    ↪  new_parents, r, D)
                    delta = i_score_prime - component_scores[i]
                    if delta > delta_best:
                        delta_best, i_best, j_best, i_score_best,
                        ↪  action = delta, i, j, i_score_prime,
                        ↪  "removeji"
                G.add_edge(j,i)

                # Try reverse
                G.remove_edge(j,i)
                G.add_edge(i,j)
                if nx.is_directed_acyclic_graph(G):
                    new_parents = np.delete(parents[i],
                    ↪  np.where(parents[i]==j)[0][0])
                    i_score_prime = compute_graph_component(i,
                    ↪  new_parents, r, D)
                    new_parents = np.concatenate(([i], parents[j]))
                    j_score_prime = compute_graph_component(j,
                    ↪  new_parents, r, D)
                    delta = (i_score_prime + j_score_prime) -
                    ↪  (component_scores[i] + component_scores[j])
                    if delta > delta_best:
                        delta_best, i_best, j_best, i_score_best,
                        ↪  j_score_best, action = delta, i, j,
                        ↪  i_score_prime, j_score_prime, "reverseji"
                G.remove_edge(i,j)
                G.add_edge(j,i)

            else:
                # Try add
                G.add_edge(j,i)
```

```
            if nx.is_directed_acyclic_graph(G):
                new_parents = np.concatenate(([j], parents[i]))
                i_score_prime = compute_graph_component(i,
                 ↪  new_parents, r, D)
                delta = i_score_prime - component_scores[i]
                if delta > delta_best:
                    delta_best, i_best, j_best, i_score_best,
                     ↪  action = delta, i, j, i_score_prime,
                     ↪  "addji"
            G.remove_edge(j,i)

            # Try add
            G.add_edge(i,j)
            if nx.is_directed_acyclic_graph(G):
                new_parents = np.concatenate(([i], parents[j]))
                j_score_prime = compute_graph_component(j,
                 ↪  new_parents, r, D)
                delta = j_score_prime - component_scores[j]
                if delta > delta_best:
                    delta_best, i_best, j_best, j_score_best,
                     ↪  action = delta, i, j, j_score_prime,
                     ↪  "addij"
            G.remove_edge(i,j)

# Move on to next iteration without changing DAG if no operations
 ↪  improved Bayesian score
if delta_best < 0:
    continue

if action == "removeij":
    G.remove_edge(i_best,j_best)
    # Update bookkeeping
    component_scores[j_best] = j_score_best
    parents[j_best] = np.delete(parents[j_best],
     ↪  np.where(parents[j_best]==i_best)[0][0])
elif action == "removeji":
    G.remove_edge(j_best,i_best)
    # Update bookkeeping
    component_scores[i_best] = i_score_best
    parents[i_best] = np.delete(parents[i_best],
     ↪  np.where(parents[i_best]==j_best)[0][0])
elif action == "addij":
    G.add_edge(i_best,j_best)
    # Update bookkeeping
    component_scores[j_best] = j_score_best
```

```
                parents[j_best] = np.concatenate(([i_best], parents[j_best]))
            elif action == "addji":
                G.add_edge(j_best,i_best)
                # Update bookkeeping
                component_scores[i_best] = i_score_best
                parents[i_best] = np.concatenate(([j_best], parents[i_best]))
            elif action == "reverseij":
                G.remove_edge(i_best,j_best)
                G.add_edge(j_best,i_best)
                # Update bookkeeping
                component_scores[i_best] = i_score_best
                parents[i_best] = np.concatenate(([j_best], parents[i_best]))
                component_scores[j_best] = j_score_best
                parents[j_best] = np.delete(parents[j_best],
                 ↪  np.where(parents[j_best]==i_best)[0][0])
            elif action == "reverseji":
                G.remove_edge(j_best,i_best)
                G.add_edge(i_best,j_best)
                # Update bookkeeping
                component_scores[j_best] = j_score_best
                parents[j_best] = np.concatenate(([i_best], parents[j_best]))
                component_scores[i_best] = i_score_best
                parents[i_best] = np.delete(parents[i_best],
                 ↪  np.where(parents[i_best]==j_best)[0][0])

    return G, sum(component_scores)


def K2_search_optimized(vars, D, r, ordering):
    n = len(vars)
    G = nx.DiGraph()
    G.add_nodes_from(range(n))

    # Precomputations
    component_scores = bayesian_score_optimized(vars, G, D)
    parents = [np.array([j for j in G.predecessors(i)], dtype=int) for i in
    ↪  range(n)]

    for (k,i) in enumerate(tqdm(ordering[1:])):
        pars = 0
        while pars < MAX_PARENTS:
            y_best, j_best = -np.inf, -1
            for j in ordering[:k+1]:
                if not G.has_edge(j,i):
                    new_parents = np.concatenate(([j], parents[i]))
                    y_prime = compute_graph_component(i, new_parents, r, D)
                    if y_prime > y_best:
                        y_best, j_best = y_prime, j
            if y_best > component_scores[i]:
                G.add_edge(j_best,i)
                # Update bookkeeping
                component_scores[i] = y_best
                parents[i] = np.concatenate(([j_best], parents[i]))
                pars += 1
            else:
                break
    return G, sum(component_scores)


def compute_graph_component(i, new_parents, r, D):
    m,n = D.shape
    q = int(np.prod(r[new_parents]))
    M = np.zeros((q, r[i]), dtype=int)
    alpha = np.ones((q, r[i]), dtype=int)
    for o in range(m):
        k = D[o,i] - 1
        j = 0
        if new_parents.size > 0:
            j = np.ravel_multi_index(D[o,new_parents] - 1, r[new_parents])
        M[j,k] += 1
    return bayesian_score_component(M,alpha)


def bayesian_score_component(M, alpha):
    p = np.sum(sp.loggamma(alpha + M))
    p -= np.sum(sp.loggamma(alpha))
    p += np.sum(sp.loggamma(np.sum(alpha, axis=1)))
    p -= np.sum(sp.loggamma(np.sum(alpha, axis=1) + np.sum(M, axis=1)))
    return p


def bayesian_score_optimized(vars, G, D):
    n = len(vars)
    M = statistics(vars, G, D)
    alpha = prior(vars, G)
    return np.array([bayesian_score_component(M[i], alpha[i]) for i in
    ↪  range(n)])


def statistics(vars, G, D):
    m,n = D.shape
    r = np.array([vars[i].r for i in range(n)])
    q = np.array([int(np.prod([r[j] for j in G.predecessors(i)])) for i in
    ↪  range(n)])
    M = [np.zeros((q[i], r[i]), dtype=int) for i in range(n)]
    for o in range(m):
        for i in range(n):
            k = D[o,i] - 1
            parents = np.array([n for n in G.predecessors(i)], dtype=int)
            j = 0
            if parents.size > 0:
                j = np.ravel_multi_index(D[o,parents] - 1, r[parents])
            M[i][j,k] += 1
    return M


def prior(vars, G):
    n = len(vars)
    r = np.array([vars[i].r for i in range(n)])
    q = np.array([int(np.prod([r[j] for j in G.predecessors(i)])) for i in
    ↪  range(n)])
    return [np.ones((q[i], r[i]), dtype=int) for i in range(n)]


def K2_search(vars, D, ordering):
    n = len(vars)
    G = nx.DiGraph()
    G.add_nodes_from(range(n))
    for (k,i) in enumerate(tqdm(ordering[1:])):
        y = bayesian_score(vars, G, D)
        while True:
            y_best, j_best = -np.inf, -1
            for j in ordering[:k+1]:
                if not G.has_edge(j,i):
                    G.add_edge(j,i)
                    yprime = bayesian_score(vars, G, D)
                    if yprime > y_best:
                        y_best, j_best = yprime, j
                    G.remove_edge(j,i)
            if y_best > y:
                y = y_best
                G.add_edge(j_best,i)
            else:
                break
    return G
```

```
def bayesian_score(vars, G, D):
    n = len(vars)
    M = statistics(vars, G, D)
    alpha = prior(vars, G)
    return sum(bayesian_score_component(M[i], alpha[i]) for i in range(n))


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()
```