CS238 - Project 1

Bayesian Network Structure Learning

ALGORITHM DESCRIPTION

The structure learning algorithm combines the K2 search algorithm with a hill climbing method (local search refinement algorithm). This structure allows the best K2 result to be refined, further improving the structure. The algorithm generates random graph neighbors by adding or removing single edges, accepting changes only when they improve the Bayesian score. This greedy approach runs for up to 1500 iterations with early stopping if no improvement is found for 200 consecutive iterations.

Bayesian Scoring Function

My Bayesian scoring algorithm was inspired by Function 5.1 in the textbook:

- Is uses a Uniform Dirichlet Distribution to find a prior $\alpha_{ijk} = 1$ for all variables
- Stabilizes the variables using a Log-Gamma function
- Decomposes the total score as the sum of all the individual variable scores

$$logP(G \mid D) = logP(G) + \sum_{i=1}^{n} \sum_{j=1}^{q_i} \left(log\left(\frac{\Gamma(\alpha_{ij0})}{\Gamma(\alpha_{ij0} + m_{ij0})} \right) + \sum_{k=1}^{r_i} log\left(\frac{\Gamma(\alpha_{ijk} + m_{ijk})}{\Gamma(\alpha_{ijk})} \right) \right)$$

Where where m_{ijk} are counts from the data and α_{ijk} are prior pseudocounts. Equation 5.5.

K2 Search Algorithm

Inspired by Algorithm 5.2 from the textbook:

- Start with empty graph (no edges)
- Process variables according to ordering
- For each variable, greedily add parents that maximize Bayesian score
- Stop adding parents when score stops improving or max_parents reached
- Ensure no cycles are created

Random Ordering

Since K2's performance depends heavily on variable ordering, I run K2 multiple times with different random orderings:

- Small dataset (8 vars): 10 random orderings
- Medium dataset (13 vars): 8 random orderings
- Large dataset (50 vars): 5 random orderings

The best-scoring graph across all orderings is selected.

Local Search Refinement (Hill Climbing)

Starting from the best K2 result, I apply local search to further improve the structure:

- Generate random neighbor by adding or removing one edge
- Accept neighbor if Bayesian score improves
- Repeat for up to 1500 iterations
- Early stopping if no improvement for 200 iterations
- Always maintain acyclicity (reject edges that create cycles)

Parameter Choices

- max_parents: 3-4 (balances complexity vs. overfitting)
- K2 trials: 5-10 (more for smaller datasets)
- Local search iterations: 1500 (with early stopping)

EXPERIMENTAL RESULTS

Variables	Samples	Runtime	K2 Score	Best Score	Improvement	# of Edges
8 - small	889	1.78s	-3797.88	-3794.86	+3.02	14
13 - med	6497	5.25s	-97063.99	-97063.16	+0.83	30
50 - large	10000	157.89s	-433402.11	-428119.67	+5282.44	123

Small Dataset (Titanic)

The algorithm discovered meaningful relationships in the Titanic dataset, with passenger class and sex as key predictors of survival. The network includes 14 edges representing demographic and social relationships among passengers.

Medium Dataset (Wine Quality)

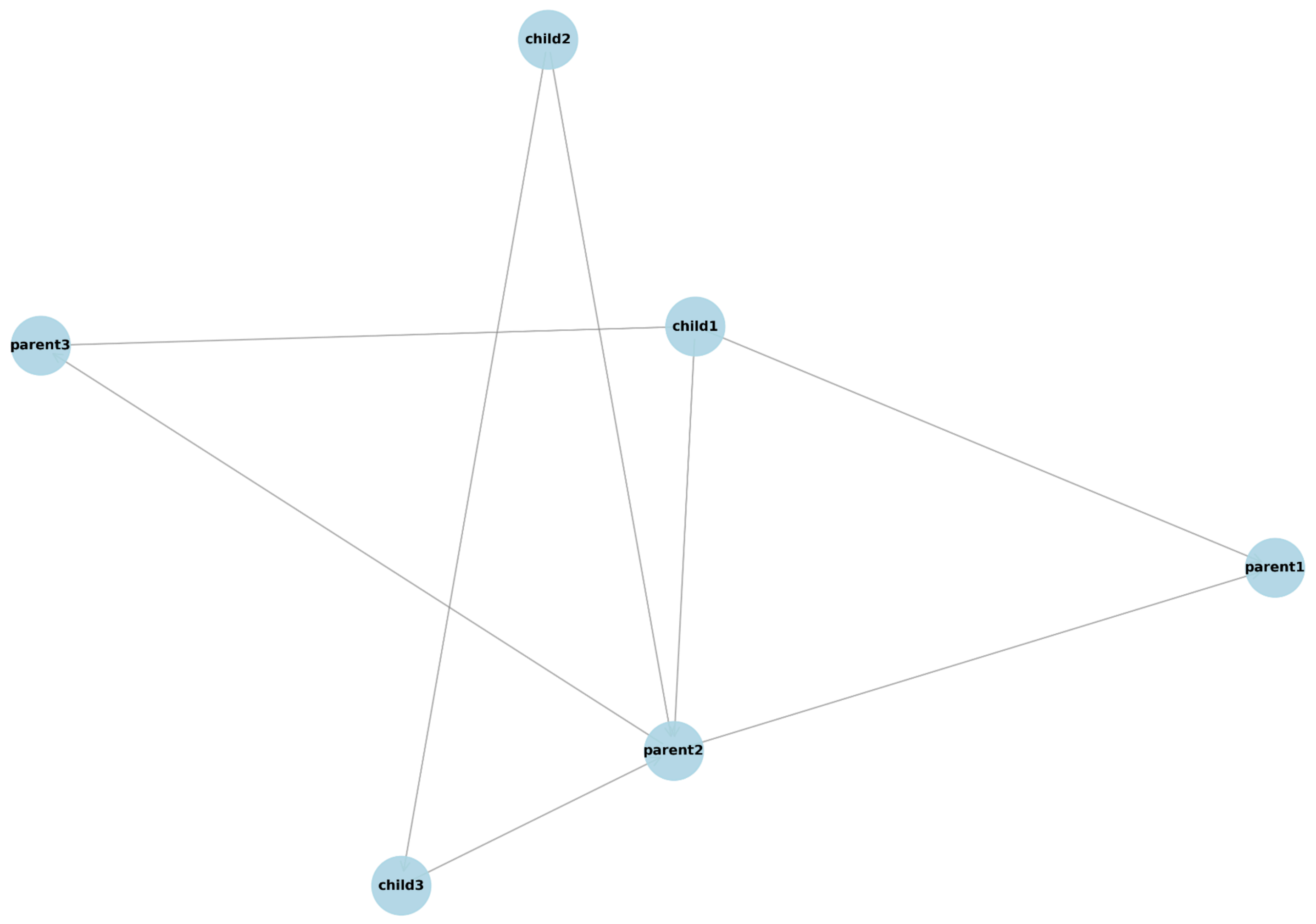
The learned structure captures wine chemistry relationships. Color emerges as a central variable influencing multiple chemical properties. Density serves as an integration point for dissolved components, and alcohol content appears as a key quality predictor.

Large Dataset (Mystery)

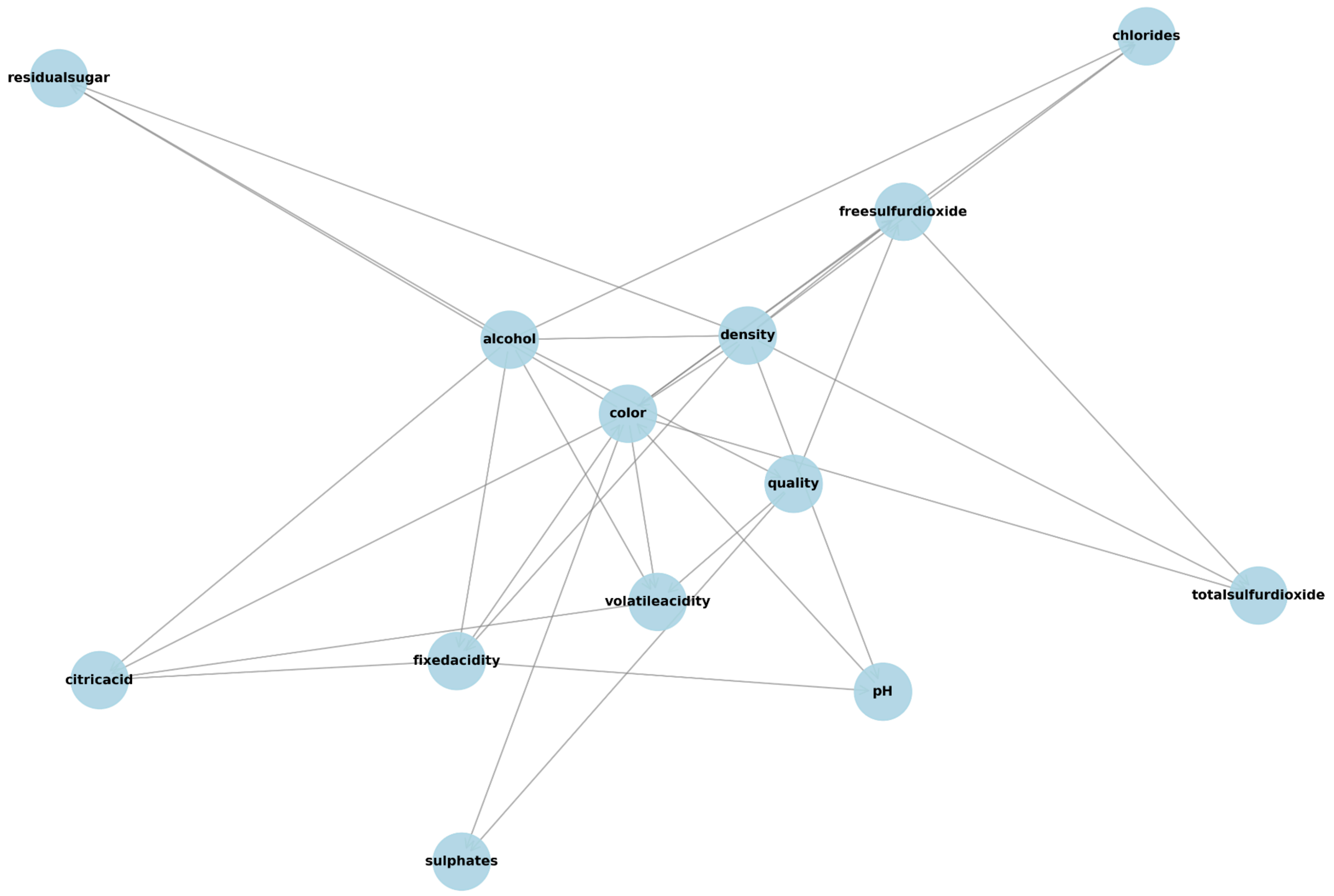
The network exhibits hierarchical structure with several hub variables showing high connectivity. With 123 edges for 50 variables, the learned structure balances model complexity with available data.

NETWORK STRUCTURES

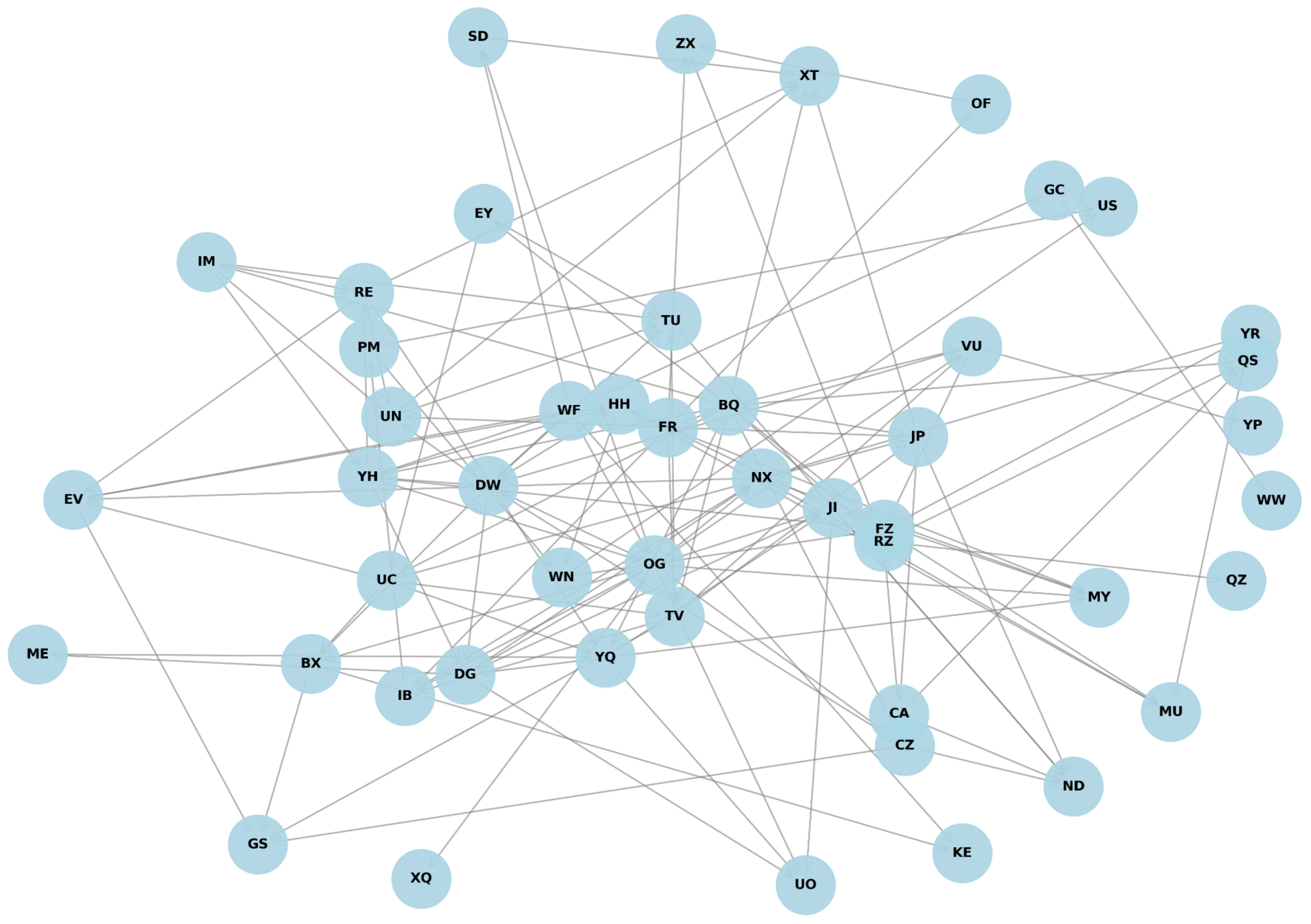
Small Dataset: Titanic Network



Medium Dataset: Wine Quality Network



Large Dataset: Network Structure



Code Implementation

```
using Graphs
using Printf
using CSV
using DataFrames
using SpecialFunctions
using Random

"""
   write_gph(dag::DiGraph, idx2names, filename)

Takes a DiGraph, a Dict of index to names and a output filename to write the graph in
`gph` format.
"""
function write_gph(dag::DiGraph, idx2names, filename)
   open(filename, "w") do io
       for edge in edges(dag)
           @printf(io, "%s,%s\n", idx2names[src(edge)], idx2names[dst(edge)])
       end
   end
end



"""
   preprocess_data(filename::String)

Prepocess data and returns:
- data (an mxn matrix containing samples and variables)
- names (a list with variable names)
- idx2names (a dictionary mapping indices to names)
- names2idx (a dictionary mapping names to indices)
- r (a list of # of discrete values for each variable: i.e., max value)
"""
function preprocess_data(filename::String)
   # Read file
   df = CSV.read(filename, DataFrame)

   # Extract variable names
   variables = names(df)
   # Number of variables
   n = length(variables)
```

```julia
    # Convert DataFrame to Matrix of integers
    data = Matrix{Int}(df)
    # Number of samples
    m = size(data, 1)

    # Index/name mappings
    idx2names = Dict(i => variables[i] for i in 1:n)
    names2idx = Dict(variables[i] => i for i in 1:n)

    # Determine r_i for each variable
    r = [maximum(data[:, i]) for i in 1:n]

    println("Preprocessed stats: $m samples, $n variables")
    println("Variable cardinality: $r")

    return data, variables, idx2names, names2idx, r
end


"""
    sub2ind(siz, x)

Transposed version of algorithm 4.1 (textbook). Returns 1-indexed linear index k

"""
function sub2ind(siz, x)
    k = x[end]
    for i in length(x)-1:-1:1
        k = (k - 1) * siz[i] + x[i]
    end
    return k
end


"""
    statistics(vars, G, D, r)
Computes matrix[i][j, k] from data and returns matrix, a vector of matrices where
matrix[i] is a q_i times r_i matrix of counts.

Transposed version of algorithm 4.1 from textbook.
"""
function statistics(vars, G, D, r)
```

```
    n = length(vars)
    m = size(D, 1)

    # Vector of matrices
    matrix = [zeros(Int, 1, r[i]) for i in 1:n]

    for i in 1:n
        # Get parents of var_i
        parents = inneighbors(G, i)

        # Case 1, no parents: q_1 = 1
        if isempty(parents)
            matrix[i] = zeros(Int, 1, r[i])
            for o in 1:m
                k = D[o, i]
                matrix[i][1, k] += 1
            end
        # Case 2, has parents: count for each instantiation
        else
            # q_i = product of paren cardinalities
            q_i = prod([r[p] for p in parents])
            matrix[i] = zeros(Int, q_i, r[i])

            for o in 1:m
                # Parent values for sample
                parent_val = [D[o, p] for p in parents]

                # Instantiation to linear index j
                parent_r = [r[p] for p in parents]
                j = sub2ind(parent_r, parent_val)

                k = D[o, i]
                matrix[i][j, k] += 1
            end
        end
    end

    return matrix
end


"""
```

```
   prior(vars, G, r)

Uniform Dirichlet prior alpha[i][j, k] = 1 for all i, j , k. Returns alpha,
a vector of matrices with all entries = 1.

Algorithm 4.2 from textbook.
"""
function prior(vars, G, r)
   n = length(vars)
   alpha = Vector{Matrix{Float64}}(undef, n)

   for i in 1:n
       parents = inneighbors(G, i)

       if isempty(parents)
           q_i = 1
       else
           q_i = prod([r[p] for p in parents])
       end

       alpha[i] = ones(Float64, q_i, r[i])
   end

   return alpha
end


"""
   bayesian_score_component(matrix_i, alpha_i)

Computes and returns bayesian score component for a variable. Uses the formula log[P(
G | D)].

Algorithm 5.1 from the textbook.
"""
function bayesian_score_component(matrix_i, alpha_i)
   partial_score = 0.0

   @views for j in axes(matrix_i, 1)
       alpha_ij0 = sum(alpha_i[j, :])
       m_ij0 = sum(matrix_i[j, :])
```

```
        partial_score += loggamma(alpha_ij0) - loggamma(alpha_ij0 + m_ij0)

        for k in axes(matrix_i, 2)
            # alpha_ijk and m_ijk
            a = alpha_i[j, k]
            m = matrix_i[j, k]

            partial_score += loggamma(a + m) - loggamma(a)
        end
    end

    return partial_score
end


"""
    bayesian_score(vars, G, D, r)

Computes and returns total bayesian score for graph G given data D.

Algorithm 5.1 from the textbook.
"""
function bayesian_score(vars, G, D, r)
    n = length(vars)
    matrix = statistics(vars, G, D, r)
    alpha = prior(vars, G, r)
    score = sum(bayesian_score_component(matrix[i], alpha[i]) for i in 1:n)

    return score
end



"""
    k2_search(vars, D, r, ordering; max_parents=3)

Adds parents to each variable in the given ordering. Returns learned DiGraph structure
G
"""

function k2_search(vars, D, r, ordering; max_parents=3)
    n = length(vars)
    G = SimpleDiGraph(n)
```

```
# Skip first variable
for k in 2:n
    i = ordering[k]

    candidates = ordering[1:k-1]

    current_best = bayesian_score(vars, G, D,r)
    improved = true

    # Adds parents while score improves and no max_parents
    while improved && length(inneighbors(G, i)) < max_parents
        improved = false
        best_parent = -1

        for j in candidates
            # j is a parent of i
            if has_edge(G, j, i)
                continue
            end

            add_edge!(G, j, i)

            # Checks cycles
            if is_cyclic(G)
                rem_edge!(G, j, i)
                continue
            end

            new_score = bayesian_score(vars, G, D, r)

            # Track best parents
            if new_score > current_best
                current_best = new_score
                best_parent = j
                improved = true
            end

            rem_edge!(G, j ,i)
        end

        if improved
```

```julia
                add_edge!(G, best_parent, i)
            end
        end
    end

    return G
end



"""
    random_graph_neighbor(G, n)

Returns a new digraph and operation (remove or add)
"""
function random_graph_neighbor(G, n)
    G_new = copy(G)

    # Try to add an edeg
    if ne(G) == 0 || (ne(G) <  n * (n-1) / 4 && rand() > 0.3)
        max_attempts = 50
        for _ in 1:max_attempts
            i = rand(1:n)
            j = rand(1:n)

            if i != j && !has_edge(G_new, i , j)
                add_edge!(G_new, i, j)

                if is_cyclic(G_new)
                    rem_edge!(G_new, i, j)
                    continue
                end

                return G_new, "add"
            end
        end

        # Try removing an edge
        if ne(G_new) > 0
            edge_list = collect(edges(G_new))
            edge_to_remove = rand(edge_list)
            rem_edge!(G_new, src(edge_to_remove), dst(edge_to_remove))
            return G_new, "remove"
        end
    else
        # Remove an edge
        if ne(G_new) > 0
            edge_list = collect(edges(G_new))
            edge_to_remove = rand(edge_list)
            rem_edge!(G_new, src(edge_to_remove), dst(edge_to_remove))
            return G_new, "remove"
        end
    end

    return G_new, "none"
end


"""
    hill_climbing(vars, D, r, G_init; k_max=1000, verbose=false)

Returns best graph found, using hill climbing algorithm.

"""
function hill_climbing(vars, D, r, G_init; k_max=1000, verbose=false)
    n = length(vars)
    G = copy(G_init)
    best_score = bayesian_score(vars, G, D, r)

    if verbose
        println("Initial score: $best_score")
    end

    no_improvement_count = 0

    for iteration in 1:k_max
        # Generate random neighbor
        G_new, operation = random_graph_neighbor(G, n)

        # Compute score of neighbor
        new_score = bayesian_score(vars, G_new, D, r)

        # Accept if score improves
        if new_score > best_score
            G = G_new
            best_score = new_score
            no_improvement_count = 0

            if verbose && iteration % 100 == 0
                println("Iteration $iteration: score = $best_score")
            end
        else
            no_improvement_count += 1
        end

        # Early stopping if no improvement for a while
        if no_improvement_count > 200
            if verbose
                println("Stopping early at iteration $iteration")
            end
            break
        end
    end

    if verbose
        println("Final score: $best_score")
    end

    return G
end


"""
    compute(infile, outfile)

Main fn to learn bayesian network structure
"""
function compute(infile, outfile)
    println("="^60)
    println("Learning structure: $infile -> $outfile")
    println("="^60)

    # Load data
    println("\n[1/3] Loading data...")
    data, var_names, idx2names, names2idx, r = preprocess_data(infile)
    n = length(var_names)
```

```julia
# Determine parameters based on problem size
if n <= 10
    num_k2_trials = 10
    max_parents = 4
elseif n <= 20
    num_k2_trials = 8
    max_parents = 3
else
    num_k2_trials = 5
    max_parents = 3
end

# Run K2 with multiple random orderings
println("\n[2/3] Running K2 with $num_k2_trials orderings...")
best_graph = SimpleDiGraph(n)
best_score = -Inf

for trial in 1:num_k2_trials
    ordering = randperm(n)
    G = k2_search(var_names, data, r, ordering, max_parents=max_parents)
    score = bayesian_score(var_names, G, data, r)

    println("  Trial $trial: score = $score")

    if score > best_score
        best_score = score
        best_graph = G
    end
end

println("\nBest K2 score: $best_score")

# Refine with local search
println("\n[3/3] Refining with local search...")
best_graph = hill_climbing(var_names, data, r, best_graph,
                           k_max=1500, verbose=true)
best_score = bayesian_score(var_names, best_graph, data, r)

# Write output
println("\nWriting output to $outfile")
write_gph(best_graph, idx2names, outfile)
```

```julia
    println("\n" * "="^60)
    println("FINAL RESULT")
    println("="^60)
    println("Best score: $best_score")
    println("Number of edges: $(ne(best_graph))")
    println("="^60)
end

# Main entry point
if length(ARGS) != 2
    error("Usage: julia project1.jl <infile>.csv <outfile>.gph")
end

inputfilename = ARGS[1]
outputfilename = ARGS[2]

Random.seed!(42)  # For reproducibility
start_time = time()

compute(inputfilename, outputfilename)

elapsed = time() - start_time
println("\nTotal runtime: $(round(elapsed, digits=2)) seconds")
```