Project 1: Bayesian Structure Learning

Trevor Perey tperey@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

For this project, Bayesian structure models were learned for three data sets of various sizes using a K2 search algorithm. For each data set, the column labels, which represent the variables, were extracted into a vector of `Variable` data structures containing the labels in the `name` fields, which were used as the node labels for the structure, and the numbers of possible values for each variable in the `r` fields. Additionally, the data from each set was extracted into a $n \times m$ matrix where n is the number of variables and m is the number of data samples. The vector of variables was then used to prescribe an ordering with which to step through these variables when learning the corresponding structure. This ordering was chosen to be that in which the variables were listed in the original data set. Next, the K2 search algorithm was applied. More specifically, an initially edgeless directed acyclic graph containing a node for each variable was assumed, and the initial Bayesian score for this graph was computed. To compute this score, a `statistics` function was first used to extract an array M whose entries m_{ijk} are the counts for the kth value of variable i for its jth parental instantiation for all i, j, k. This was done by iterating over each data sample, determining the value of each variable and its parents in that sample, and incrementing the appropriate count per Algorithm 4.1 in the text [1, pg. 75]. Then, the Bayesian score was computed using Algorithm 5.1 in the text [1, pg. 98]. Next, for each variable in the ordering, the Bayesian scores for new graphs containing an edge to each of the prior variables in the ordering, which are the acceptable parents, were considered. The developing graph was then updated to contain the edge leading to the greatest improvement in score. If the variable under consideration had multiple acceptable parents, after adding an edge to one, edges to the remainders were then also considered, and the edge leading to the greatest score improvement was kept. This process was repeated for a given variable until there were no remaining parents or possible improvement in score. The next variable in the ordering was then considered and the process repeated for all variables. The resulting graph was then chosen as the learned Bayesian structure. Notably, this algorithm ensures no cycles are generated as only prior variables in the ordering are allowed as parents. Furthermore, the algorithm was developed to allow for user-imposed limits on the maximum number of parents for each node, but these limits were not used in practice as they were found to lead to minimal improvements in computational timing in exchange for lower Bayesian scores for the final learned structure.

2. Graphs

Three data sets were considered: `small.csv`, `medium.csv` and `large.csv`. A plot visualizing the final structure obtained with the described algorithm for the `small` data set along with the corresponding score and runtime is given in Figure 1 below.

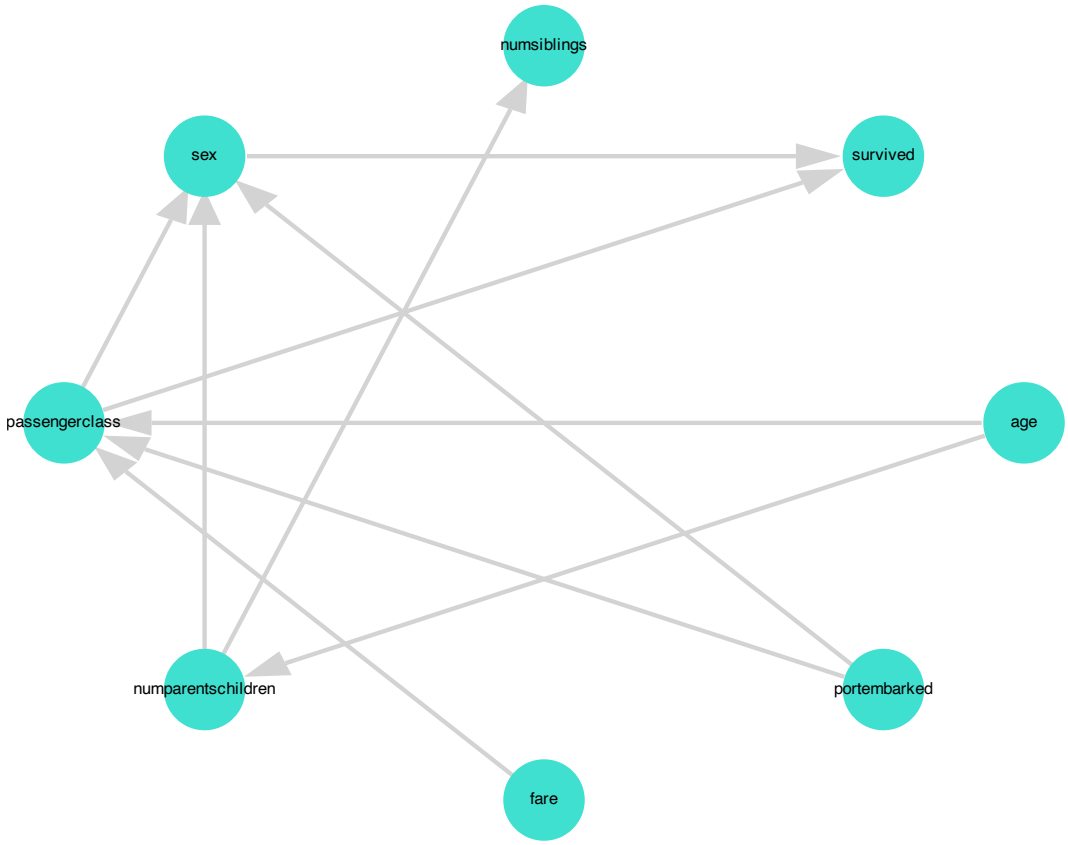

Figure 1: Learned Bayesian structure for the `small` data set. The final Bayesian score was –3835.67942521279, and the runtime was 0.690971 seconds.

A plot of the structure obtained for the `medium` data set with the corresponding score and runtime is provided in Figure 2 below.

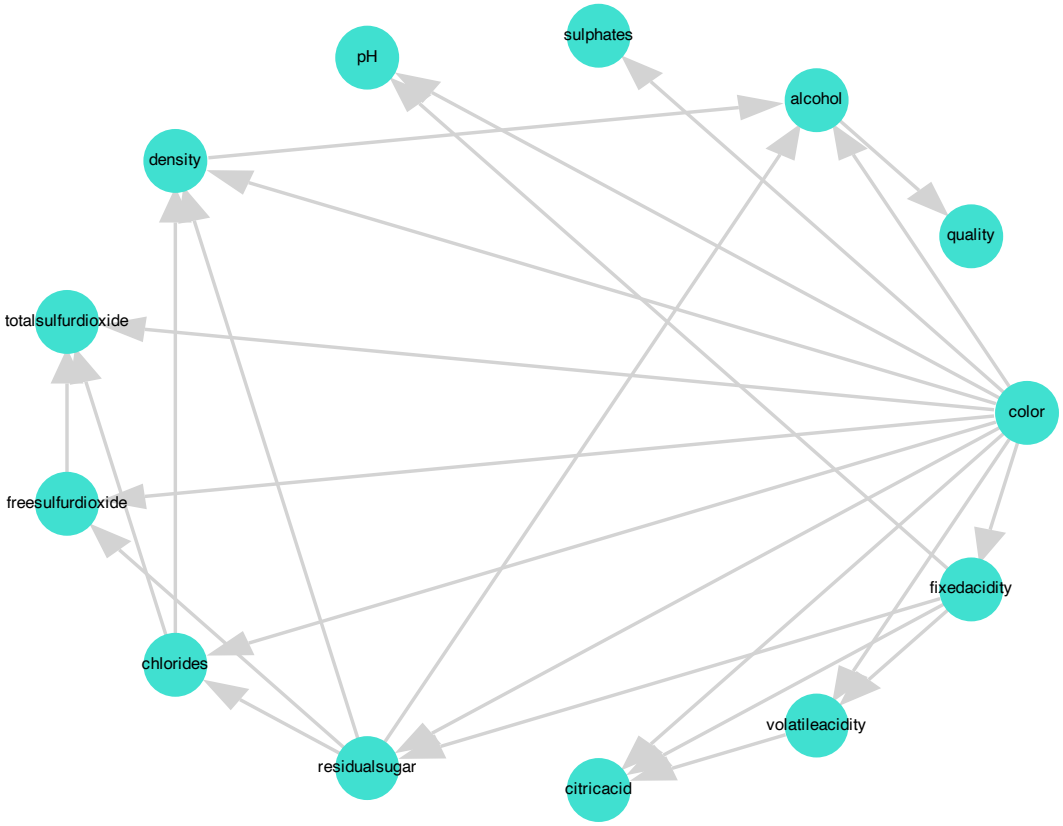

Figure 2: Learned Bayesian structure for the `medium` set. The final Bayesian score was −97918.44194908428, and the runtime was 2.798188 seconds.

A plot of the structure obtained for the `large` data set with the corresponding score and runtime is provided in Figure 3 below.

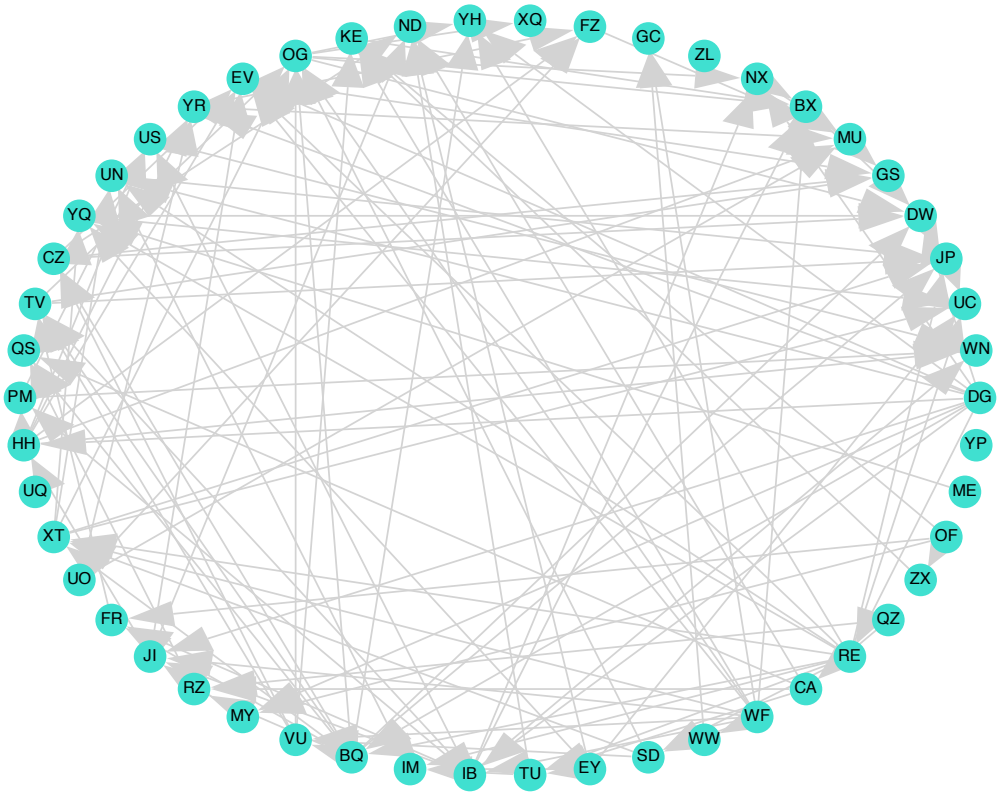

Figure 3: Learned Bayesian structure for the `large` set. The final Bayesian score was −430708.5510238648, and the runtime was 274.456486 seconds, or approximately 4.57 minutes.

3. Code

The code used to implement the described algorithm and generate the provided structures, plots, scores, and runtimes is provided below.

```
using Pkg
Pkg.activate(".")

using Graphs
using Plots
using Printf
using DataFrames
```

```
using CSV
using GraphRecipes
using LinearAlgebra
using SpecialFunctions

using GraphPlot
using Compose, Cairo, Fontconfig

"""
    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

#====================================
  STRUCTURES
====================================#

# Variable struct
struct Variable
    name::Symbol ## Variable name
    r::Int ## Number of possible values (1:r) for that variable
end

# Variable
struct K2Search
    ordering::Vector{Int} # Topological sort to use for K2.
    # Ints are not necessarily sorted
    # Correspond to order in original csv
end

#====================================
  EXTRACTION - Helper functions
====================================#

### DATA EXTRACTION ###
function extract_data(filepath)
    alldata = CSV.read(filepath, DataFrame) # Get dataframe with all data

    # Build up Vector of variables
```

```
    var_names = Symbol.(names(alldata)) # Extract column names into list of
    vars

    vars = Vector{Variable}(undef, length(var_names)) # Initialize a vector
    for storing Variables
    index = 1
    for col in names(alldata) # Iterate over each column
        vars[index] = Variable(Symbol(col), maximum(alldata[!,col])) # Create
     Variable with name and r
        index += 1 # Move to next index of vars
    end

    # Generate matrix of data
    data_matrix = Matrix(alldata)
    data_matrix = collect(transpose(data_matrix)) # statistics func expects
    vars x data

    return vars, data_matrix
end

### GRAPH EXTRACTION (from .gph file) ###
function extract_graph(graphpath)

    # Extract edges from graphpath
    lines = readlines(graphpath) # Read in the file
    edges = [ (Symbol(split(line, ",")[1]), Symbol(split(line, ",")[2])) for
    line in lines] #Convert to symbol pairs

    # Extract variables
    var_symbols = vcat([var for e in edges for var in e]...) # Use splat and
    vcat to break edges pairs into single array of symbols
    var_symbols = unique(var_symbols) # Reduce down to unique vars

    # Build graph
    var_index_map = Dict(var_symbols => index for (index, var_symbols) in
    enumerate(var_symbols))
    # Create tuples of (index, var) for all variables,
    # and then a mapping from each variable to its index

    g = SimpleDiGraph(length(var_symbols)) # Instantiate directed graph with
    nodes for unique variables
    for (parent, child) in edges # For all edges in list
        add_edge!(g, var_index_map[parent], var_index_map[child]) # Use
    mapping to create that edge
    end

    return var_symbols, g

end
```

```
#===================================
  SCORE - Helper functions
===================================#

### STATISTICS EXTRACTION (per pg. 75 of the text) ###

# Helper function for determining parental instatiation
function sub2ind(sze, x)
    k = vcat(1, cumprod(sze[1:end-1])) # Compute cumulative index product for
     each x (e.g. parent) value
    return dot(k, x.-1) + 1 # Compute linear index for particular parental
    instantiation
end


# Function for extracting counts M
function statistics(vars, G, D::Matrix{Int})

    n = size(D,1) # Number of variables
    r = [vars[i].r for i in 1:n] # Extract vector of ri's (number of values
    for each variable)
    q = [prod([r[j] for j in inneighbors(G,i)]) for i in 1:n] # Extract
    vector of qi's
    # (number of parental instantiations for each variable, which is product
    of r's of parents)

    # Build up counts matrix
    M = [zeros(q[i], r[i]) for i in 1:n] # n x qi x ri tensor. ith is qi x ri
     matrix of counts for variable i

    for o in eachcol(D) # Iterate through variables
        for i in 1:n # Iterate through values for a variable
            k = o[i] # Get current value

            # Determine parental instantiation
            parents = inneighbors(G,i) # Get parent indices
            j = 1 # Initialize j to 1 (in case no parents)
            if !isempty(parents)
                j = sub2ind(r[parents], o[parents]) # Pass the possible and
    actual parent values to sub2ind
            end
            M[i][j,k] += 1.0 # Have determined current value for variable i
    to be for parental instantiation j and value k. Increment.
        end
    end

    return M
end

### UNIFORM PRIOR (per pg. 81 of the text) ###
```

```
function prior(vars, G)
    n = length(vars) # Get number of variables
    r = [vars[i].r for i in 1:n] # Get vector of possible values ri for each
    variable idx2names
    q = [prod([r[j] for j in inneighbors(G,i)]) for i in 1:n] # Get vector of
     number of parental instantiations for each variable
    # qij = product of r's for parents of variable i
    return [ones(q[i], r[i]) for i in 1:n] # return qi x ri matrix of 1's (
    uniform) for each variable i
end

### BAYESIAN SCORE (per pg. 98 of text) ### <-- REWRITE

# Helper for each ith Bayes score component
function bayesian_score_component(M, )
    p = sum(loggamma.(+M)) # Equ 5.5, term 2 numerator
    # Adds priors and counts, takes loggamma of each element, then sums
    matrix elements.
    # This sums over all qi and ri as desired.

    p -= sum(loggamma.()) # Equ 5.5, term 2 denominator
    # Takes loggamma of prior elements, then sums matrix elements.
    # This sums over all qi and ri as desired.

    p += sum(loggamma.(sum(,dims=2))) # Equ 5.5, term 1 numerator
    # First, sums  over all ri for each j in qi. Gives vector of _ij0 for
    each j up to qi.
    # Then, takes loggamma of vector elements and sums them. This sums
    loggamma(_ij0) for all j up to qi as desired.

    p -= sum(loggamma.(sum(,dims=2) + sum(M,dims=2))) # Equ 5.5, term 1
    denominator
    # First, sum  and M over all ri for each j in qi. Gives vectors _ij0 and
    m_ij0 for each j up to qi and sums them.
    # Then, takes loggamma of vector elements and sums them. This sums
    loggamma(_ij0 + m_ij0) for each j up to qi as desired.
end

# Overall Bayes score
function bayesian_score(vars, G, D)
    n = length(vars) # Get number of variables
    M = statistics(vars, G, D) # Get counts m_ijk
     = prior(vars, G) # Define uniform prior

    return sum(bayesian_score_component(M[i],[i]) for i in 1:n)
    # Compute score components for each variable and sum them to return total
     score
end

#===================================
```

```
  FINDING BEST GRAPH
=====================================#

function K2fit(method::K2Search, vars, D, parent_lim::Int = 2)
    G = SimpleDiGraph(length(vars)) # Instantiate a graph with nodes for vars
    for (k,i) in enumerate(method.ordering[2:end]) # Iterate through nodes.
        # Note k = (1, length(ordering) - 1 ) corresponds to i = ordering(2:
    end)

        y = bayesian_score(vars, G, D) # Starting Bayes Score for assessing
    parents for node i

        while true
            y_best, j_best = -Inf, 0 # Initialize new best Bayes score,
    parent index

            for j in method.ordering[1:k] # Loop through allowable (previous)
     parents
                # Evalute each edge from parent j to current node i, and save
     best
                if !has_edge(G, j, i) # Verify edge doesn't already exist

                    add_edge!(G,j,i)
                    y_new = bayesian_score(vars, G, D) # Add and eval edge

                    if y_new > y_best # If score improvement
                        y_best, j_best = y_new, j #Update best score, parent
    index
                    end

                    rem_edge!(G, j, i) # Remove and test others
                end
            end
            # Now, have best possible parent

            # FIRST, enforce parent limit
            if ( (length(inneighbors(G,i))) >= parent_lim )
                break
            end

            # If within limit, decide to keep this edge
            if y_best > y # If improvement over prior score
                y = y_best # Update current score
                add_edge!(G, j_best, i) # Actually keep that edge

                # 'while true' will lead to checking of additional parents (
    from allowable)

            else # NO IMPROVEMENT, so assess next node
                break
```

```
            end

        end
        # Repeat for all nodes

    end

    return G # Return final graph
end

#====================================
  MAIN COMPUTATION
====================================#
function compute(infile, outfile)

    #===== K2 SEARCH with most basic ordering =====#
    # K2 search
    vars, D = extract_data(infile)
    ordering = collect(1:length(vars))
    search = K2Search(ordering)

    println("About to search")
    parent_limiter = length(vars) + 1
    @time begin
        G = K2fit(search, vars, D, parent_limiter)
    end

    # Plots (2 versions so can choose best)
    nodes = [vars[i].name for i in 1:length(vars)]
    plotname, _ = splitext(split(outfile, "/")[end]) # Parse out data file
    name

    plotname1 = plotname * "_plot_1.pdf" # Version 1, using GraphRecipes
    visualize_graph = plot( graphplot(G; names = nodes, method = :circular),
    size = (1000,700) )
    savefig(visualize_graph,plotname1)

    plotname2 = plotname * "_plot_2.pdf" # Version 2, using GraphPlot
    visualize_graph = gplot(G; nodelabel = nodes, layout = circular_layout)
    draw(PDF(plotname2, 28cm, 22cm), visualize_graph)

    # Compute and print score
    print("Score = ")
    score = bayesian_score(vars,G,D)
    println(score)

    scorename = plotname * ".score" # Save in a file
    open(scorename, "w") do file
        println(file, score)
    end
```

```
    # Write to file
    indices = Dict(i => variable for (i, variable) in enumerate(nodes))
    write_gph(G, indices, outfile)

end

if length(ARGS) != 2
    error("usage: julia project1.jl <infile>.csv <outfile>.gph")
end

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

compute(inputfilename, outputfilename)
```

4. References

[1] Kochenderfer, Mykel J., Wheeler, Tim A., and Wray, Kyle H. *Algorithms for Decision Making*. The MIT Press, 2022, `algorithmsbook.com/files/dm.pdf`.