Project 1: Bayesian Structure Learning

John Dalloul jdalloul@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

For the structure learning included in the below code, the K2 algorithm was used as described in section 5.2 of the course textbook. The parameters for this algorithm included: topological variable ordering generated by randomly shuffling the variables, a single starting point, and a uniform Dirichlet prior distribution.

The algorithm functions by beginning with an empty graph and greedily adding edges that increase the graph's calculated Bayesian score, which represents the probability of the given dataset given the graph's structure. The algorithm proceeds until it has gone through each variable and assessed the potential addition of a parent.

Below are the times and scores for running the algorithm on each of the datasets:

1. small.csv:
   (a) Time taken: 0.8 seconds
   (b) Graph score: -3817.7
2. medium.csv:
   (a) Time taken: 1.6 seconds
   (b) Graph score: -42369.4
3. large.csv:
   (a) Time taken: 22 minutes
   (b) Graph score: -423359.0

2. Graphs

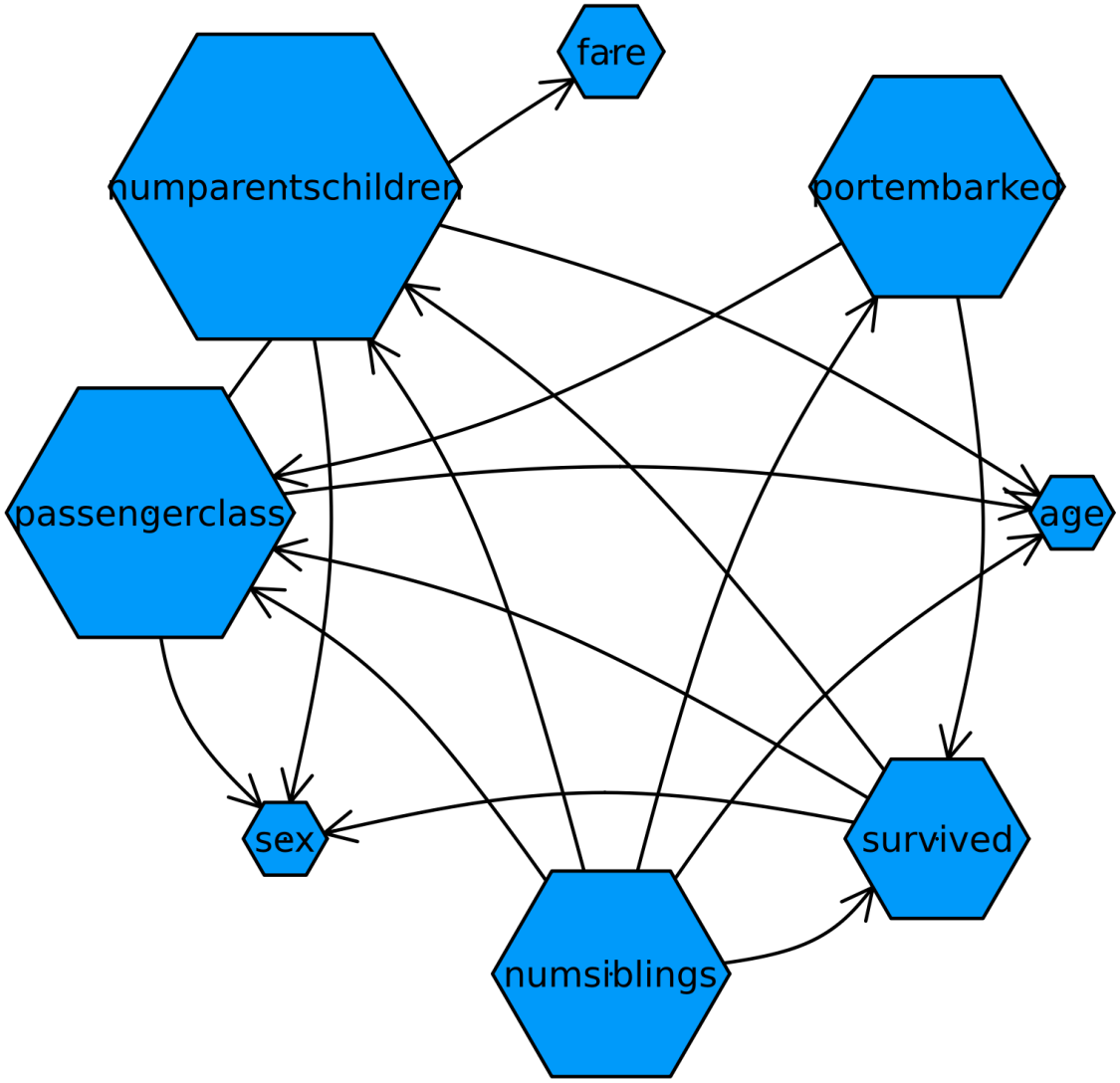

Figure 1: K2 output on small.csv

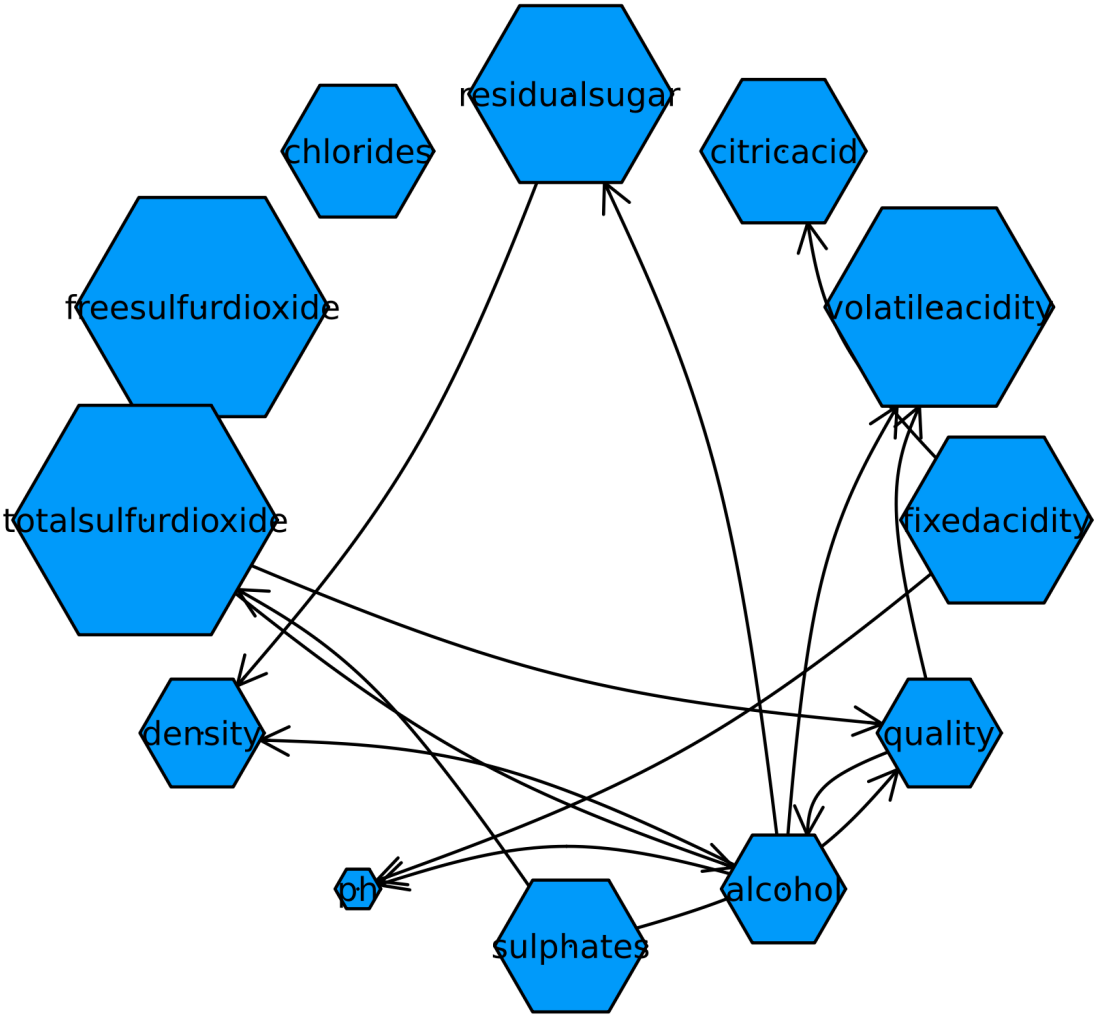

Figure 2: K2 output on medium.csv

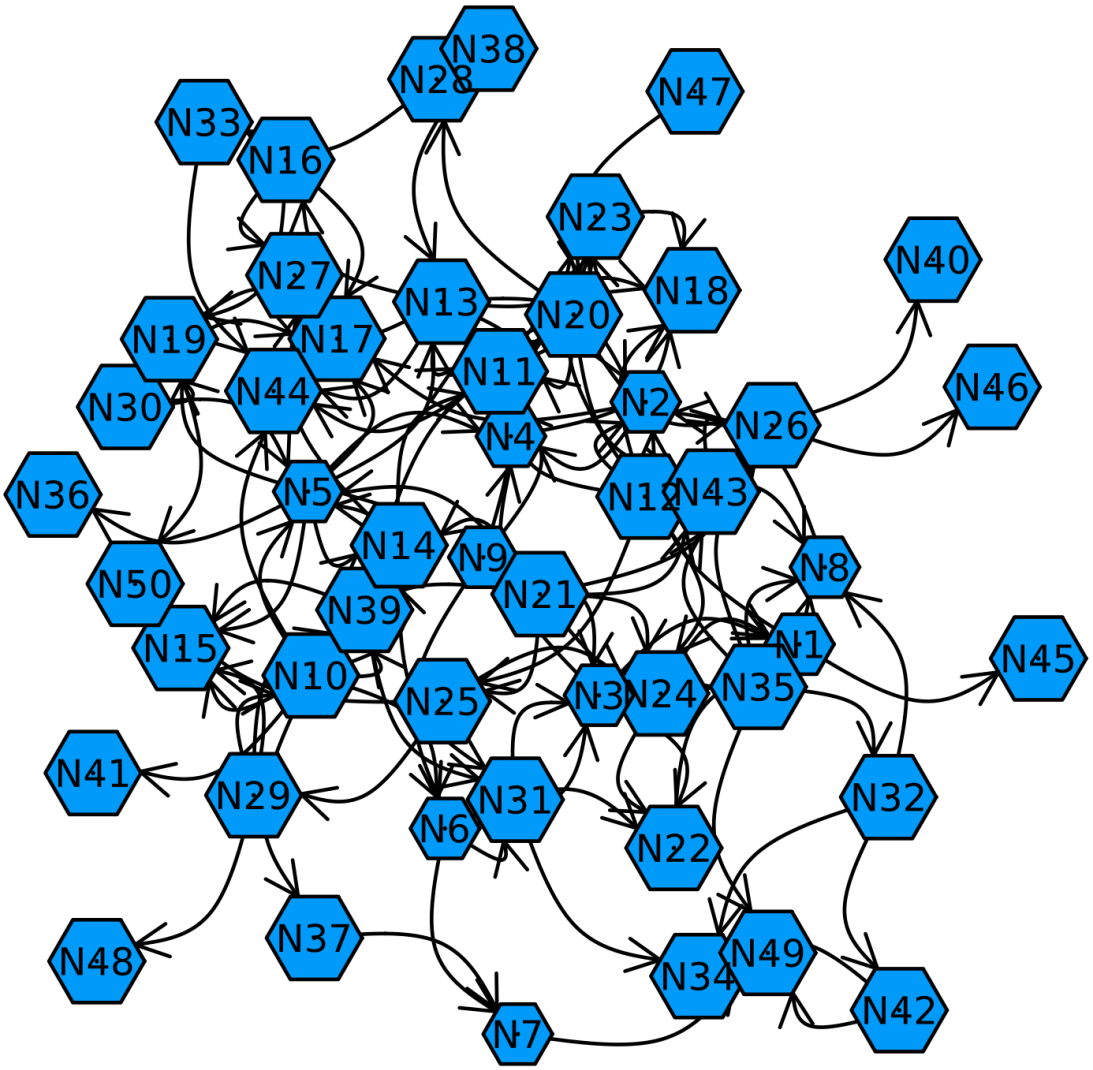

Figure 3: K2 output on large.csv

3. Code

```
using Graphs
using Printf
using CSV # reading in data
using DataFrames # reading in data
using SpecialFunctions # for loggamma
using Random # shuffling variables
using GraphRecipes
using Plots

"""
    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

"""
Takes in the path of a file containing an edgelist and creates a
    SimpleDiGraph
from it. Returns the graph object and a dictionary mapping the name of a
    vertex
to its index in the graph.

Inspiration taken from Ed post #368.
"""
function load_gph(graphfile, datafile)
    vars = split(readline(datafile), ',')
    names_to_idx = Dict(vars[i] => i for i in eachindex(vars))
    g = SimpleDiGraph(length(vars))
    open(graphfile, "r") do f
        while ! eof(f)
            line = readline(f)
            source, dest = split(line, ',')
            add_edge!(g, names_to_idx[source], names_to_idx[dest])
        end
    end

    return g
end
```

```
"""
Takes in a graph (SimpleDiGraph) and data matrix and returns an array
    containing
the (parental instantiations) x (number of variable values) counts.

Inspiration taken from Algorithm 4.1 in the course textbook.
"""
function get_counts_from_data(g, data)
    n = size(data, 2) # number of variables = number of cols
    r = [maximum(col) for col in eachcol(data)] # num values from max in
    dataset
    q = [prod([r[j] for j in inneighbors(g, i)]) for i in 1:n] # num parental
     instantiations
    M = [zeros(q[i], r[i]) for i in 1:n] # for each variable, init parent x
    val

    for sample in eachrow(data) # for each sample
        for i in 1:n # for each variable
            k = sample[i] # variable value is the same as the value's index k
            parents = inneighbors(g, i)
            j = 1
            if !isempty(parents)
                # convert parent values (cartesian indices) to linear index
                # for parental instantiations
                lin = LinearIndices(Tuple(r[parents]))
                j = lin[sample[parents]...]
            end
            M[i][j, k] += 1
        end
    end
    return M
end

"""
Takes in counts from a dataset (array of n variables, each containing an
    array
of j parental instantiations by k variable values) and Dirichlet prior counts
(same shape as the dataset counts) and calculates/returns the bayesian score.
"""
function calculate_bayesian_score(M, priors)
    # for each combination of i, j, k
    score = sum(
        [
            sum(loggamma.(M[i] + priors[i])) - sum(loggamma.(priors[i]))
            for i in eachindex(M)
        ]
    )

    # for each combination of i, j, summing across k indices (e.g. m_ij0)
    score += sum(
```

```
        [
            (sum(loggamma.(sum(priors[i], dims = 2))
                - loggamma.(sum(M[i], dims = 2) + sum(priors[i], dims = 2))))
            for i in eachindex(M)
        ]
    )

    return score
end

"""
Takes in a graph (SimpleDiGraph) and a data matrixand calculates/returns the
bayesian score (probability of the data given the graph structure).
"""
function compute(g, data)
    counts = get_counts_from_data(g, data)
    priors = [ones(size(var_counts)) for var_counts in counts] # uniform
    prior
    return calculate_bayesian_score(counts, priors)
end

"""
Takes in an topological ordering of variables and a data matrix and runs the
    k2
local search algorithm, greedily adding parents to each node that increase
    the
graph's bayesian score.
"""
function run_k2_search(var_order, data)
    g = SimpleDiGraph(size(data, 2))
    for (k, i) in enumerate(var_order[2:end]) # for each variable
        y = compute(g, data)
        while true
            y_best, j_best = -Inf, 0
            for j in var_order[1:k] # loop through candidate parents
                if !has_edge(g, j, i)
                    add_edge!(g, j, i)
                    curr_y = compute(g, data)
                    if curr_y > y_best
                        y_best, j_best = curr_y, j # better to add j as
    parent
                    end
                    rem_edge!(g, j, i)
                end
            end
            if y_best > y
                y = y_best
                add_edge!(g, j_best, i)
            else
                break
```

```
            end
        end
    end
    return g, compute(g, data) # return final graph and its bayesian score
end

"""
Takes in a file containing data and the search method to use (e.g. "k2",
    which
is the only one implemented for baseline) and runs the search method to find
    the
optimal graph structure.
"""
function find_best_graph(datafile, searchmethod)
    # init data and mapping of variable to index from datafile
    data_df = CSV.read(datafile, DataFrame)
    data = Matrix(data_df)
    vars = names(data_df)
    names_to_idx = Dict(vars[i] => i for i in eachindex(vars))
    if searchmethod == "k2"
        var_order = shuffle(eachindex(vars)) # random topological ordering
        g, score = run_k2_search(var_order, data)
        return g, score, names_to_idx
    end
end

# start of the program

fn = "draw"

if fn == "score"
    # Calculate the bayesian score of a graph for a specified dataset
    graphfile = "project1\\example\\example.gph"
    datafile = "project1\\example\\example.csv"
    outputfile = "test2.txt"

    g = load_gph(graphfile, datafile)
    data = Matrix(CSV.read(datafile, DataFrame))

    score = compute(g, data)
    open(outputfile, "w") do f
        write(f, string(score))
    end
elseif fn == "learn"
    # Determine the optimal graph structure for a specified dataset
    datafile = "data\\large.csv"
    outputfile = "large_output"
    searchmethod = "k2"

    graph_outputfile = outputfile * ".gph"
```

```
    score_outputfile = outputfile * ".txt"

    println("Finding best graph ...")
    g, score, names_to_idx = find_best_graph(datafile, searchmethod)

    println("Found graph. Writing to files and plotting ...")
    # write score and mapping of variable names -> indices to output
    open(score_outputfile, "w") do f
        write(f, string(score), '\n')
        for key in keys(names_to_idx)
            @printf(f, "%s => %s\n", key, names_to_idx[key])
        end
    end

    # plot graph
    idx_to_names = Dict(names_to_idx[var_name] => var_name for var_name in
    keys(names_to_idx))
    node_names = [idx_to_names[i] for i in 1:length(idx_to_names)]
    nodesize = [max(outdegree(g, v), 2) for v in vertices(g)]
    display(graphplot(g, names = node_names, nodesize = 0.2, method = :stress
    ))

    # write graph edgelist to file
    write_gph(g, idx_to_names, graph_outputfile)
elseif fn == "draw"
    # plot the graph defined by a specified edgelist and datafile
    graphfile = "large_output.gph"
    datafile = "data\\large.csv"

    g = load_gph(graphfile, datafile)
    display(graphplot(g, names = node_names, nodesize = 0.2, method = :stress
    ))
end
```