Project 1: Bayesian Structure Learning

Matthias Kura kura@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

1.1 Scoring Function

The scoring function `bayesian_score` uses the Bayesian score to obtain an estimate of the fitness of a graph to given data. First, a `create_pseudocount_matrices` function is used to obtain the prior and pseudocounts of the variables based on the number of parental instantiations r_i as well as the graph structure G and data matrix D. With the number of possible parental instantiations of node X_i, given as q_i, the function iterates over all nodes and increments the score an nested for loops for the terms summing over r and q.

1.2 Search Algorithm

The core search algorithm, `K2_search`, employs the K2 search algorithm with a given `order` and a maximum number of parents `max_par`. First, a simple directed graph with no edges is created. Iterating over the second to the last node given in the order vector, the function first initializes the node number `node`, the parental options `par_options` of node i, and an empty array `parents`, where the best parent options for the node are saved. While the maximum number of parents is not reached and the while loop is not exited from within, the function iterates over all parental options, adding edges to the graph to find one that increases the Bayesian score. Each best parent is then added to the graph and saved in the `par_best` vector. The best score is also updated. Once no more parent is found that increases the score, the while loop is exited.

Building on the K2 search function, the `K2_order` function randomizes the initial ordering to be used in the K2 algorithm. Here, a Mersenne Twister seed from the `Random` package is used to generate a repeatable random set of orders to test with the K2 algorithm. The best ordering, including the score and related graph are returned.

1.3 Settings

Based on the dataset, the algorithm settings are altered to tune the runtime and accuracy of the search. A summary of the search parameters is given in Table 1 below.

Table 1: Algorithm Settings

Setting	Small	Medium	Large
Number of Random Generated Orderings	1000	20	5
Maximum Number of Parents	7	12	49
MersenneTwister Seed	43	43	43

2. Graphs

Table 2 summarizes relevant information on the runtime and score of each dataset. The graph visualizations for the small, medium, and large datasets are given in Figure **??**, **??**, **??**, respectively.

Table 2: Graph Results

	Small	Medium	Large
Runtime [s]	12.500165939331055	15.215260982513428	599.6825940608978
Score	-3794.855597709796	-96569.22487170994	-428080.0160640493

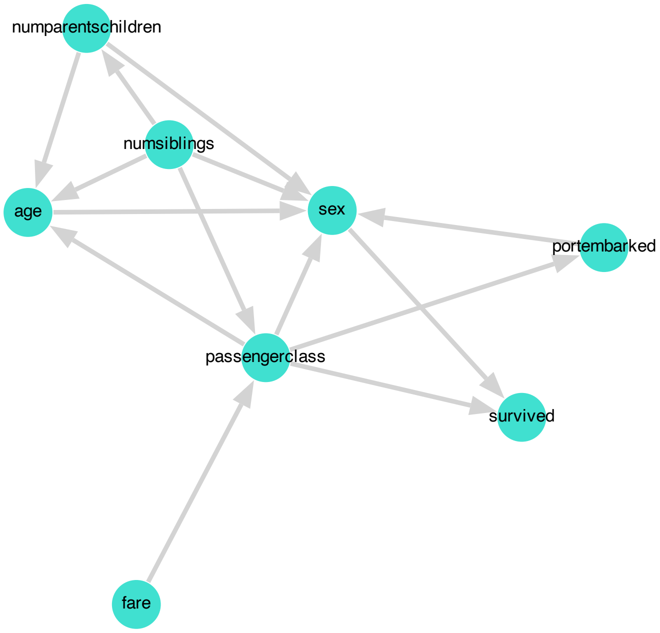

Figure 1: Graph for Small Dataset

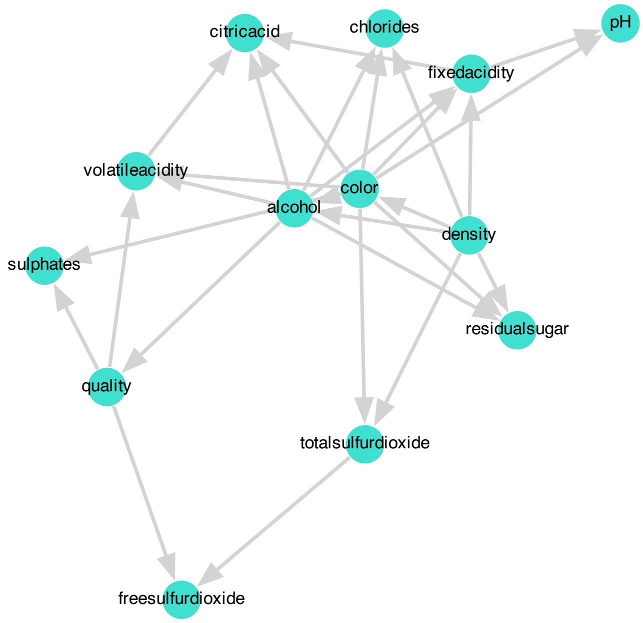

Figure 2: Graph for Medium Dataset

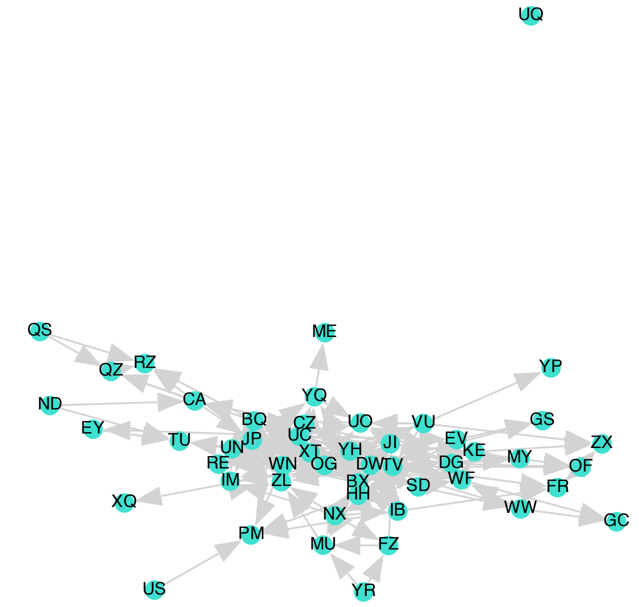

Figure 3: Graph for Large Dataset

3. Code

Listing 1: Bayesian Score Function

```
using Graphs
using GraphPlot
using Printf
using DataFrames
using CSV
using LinearAlgebra
using SpecialFunctions
using Compose, Cairo, Fontconfig
using Random
using Distributions
using Dates

"""
    write_gph(G::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(G::DiGraph, idx2names, filename)
    open(filename, "w") do io
        for edge in edges(G)
            @printf(io, "%s,%s\n", idx2names[src(edge)], idx2names[dst(edge)
    ])
        end
    end
end

function bayesian_score(r,G,D)
    score = 0.0
    n = nv(G)
    alpha, M = create_pseudocount_matrices(r, G, D)
    q = [size(alpha[i],1) for i in 1:n]
    for i in 1:nv(G)
        alpha_ij0 = sum(alpha[i], dims=2)
        m_ij0 = sum(M[i], dims=2)
        for j in 1:q[i]
            for k in 1:r[i]
                score += sum(loggamma.(alpha[i][j,k] + M[i][j,k]))  #
    nominator in r sum
                score -= sum(loggamma.(alpha[i][j,k]))  # denominator in r
    sum
            end
        end
        score += sum(loggamma.(alpha_ij0))  # nominator in q sum
        score -= sum(loggamma.(alpha_ij0 + m_ij0)) # denominator in q sum
    end
    return score
end

function create_pseudocount_matrices(r, G, D)
    n = nv(G)   # get number of nodes
    q = zeros(1,n)
    # number of instantiations of parents
    for i in 1:n
        par = inneighbors(G,i)
        if isempty(par)
            q[i] = 1
        else
            q[i] = prod(r[par])
        end
    end
    # create prior
    alpha = Vector{Matrix{Int64}}(undef,n)
    for i in 1:n
        alpha[i] = ones(Int64(q[i]), r[i])
    end

    ## create pseudocount matrix
    M = [zeros(Int64(q[i]), r[i]) for i in 1:n]
    for set in eachcol(D)
        for i in 1:n
            k = set[i] # recorded instantion of node i in data set
            par = inneighbors(G,i)
            if isempty(par)
                j = 1
            else
                j = 1   # instantiation table starts at 1
                inc_par = zeros(1, length(par)) # increment of index per
    parent
                for p in 1:length(par)
                    inc_par[p] = prod(r[par[1:p-1]])    # ordering here:
    equal instantations of last parent first, then second to last, ...
                    j += (set[par[p]]-1) * inc_par[p]
                end
            end
            M[i][Int64(j),k] += 1.0
        end
    end
    return alpha, M
end

function K2_search(order,max_par,var_ids,r,D)
    n = length(order)
    G = SimpleDiGraph(length(var_ids))
    score = -Inf
    for i in 2:n
        node = order[i]
        par_options = order[1:i-1]
        parents = []
        score = bayesian_score(r,G,D)

        while length(parents) < max_par
            par_best = []
            score_par_best = score

            # find the parent that gives best score improvement
            for p in par_options
                if !has_edge(G,p,node)
                    add_edge!(G,p,node)
                    score_new = bayesian_score(r,G,D)
                    if score_new > score_par_best
                        score_par_best = score_new
                        par_best = p
                    end
                    rem_edge!(G,p,node)
                end
            end
            # if any parent improved the score, add that parent
            if isempty(par_best)
                break
            else
                add_edge!(G,par_best,node)
                push!(parents,par_best)
                score = score_par_best
            end
        end
    end
    return G, score
end

function K2_order(n_order, max_par, seed, var_ids, r, D)
    rng = MersenneTwister(seed)
    orders = [shuffle(rng,var_ids) for _ in 1:n_order]
    best_score = -Inf;
    G_best = SimpleDiGraph(length(var_ids))
    best_order = orders[1]
    for i = 1:n_order
        # run K2 algorithm with different orderings
        G, score = K2_search(orders[i],max_par,var_ids,r,D)
        if score > best_score
            best_order = orders[i]
            best_score = score
            G_best = G
        end
    end
    return G_best, best_score, best_order
end

function compute(infile, outfile)
    ## GET DATA
    # read in csv file
    data_raw = CSV.read(infile, DataFrame)
    # create variables based on names in csv files and maximum value of each
    node
    var_names = names(data_raw)
    var_ids = 1:length(var_names)
    r = [maximum(unique(data_raw[:,i])) for i in 1:size(data_raw,2)] # get
    number of possible instantiations for each node
    D = transpose(Matrix(data_raw)) # data matrix has column for each set,
    one line per node (same as in book)

    ## RUN ALGORITHMS
    start_time = time()
    alg = "K2_order"
    if alg == "K2_order"
        if dataset == "small"
            n_order = 1000    # number of different random orderings
            max_par = length(var_names)-1     # maximum number of parents
        elseif dataset == "medium"
            n_order = 20
            max_par = length(var_names)-1
        elseif dataset == "large"
            n_order = 5
            max_par = length(var_names)-1
        end
        seed = 43
        G, score, order = K2_order(n_order,max_par,seed,var_ids,r,D)
    end

    ## Format output
    # txt report file
    report_file = "project1/output/"*dataset*".txt"
    runtime = time() - start_time
    current_dt = Dates.format(Dates.now(), "yyyy-mm-dd HH:MM:SS")
    open(report_file, "w") do f
        println(f, "=====================================")
        println(f, "Report File")
        println(f, "=====================================")
        println(f, "Dataset: " * dataset)
        println(f, "Date: " * current_dt)
        println(f, "Algorithm: " * alg)
        if alg == K2_order
            println(f, "Number of Orderings Tried: " * string(n_order))
            println(f, "Seed for MersenneTwister Random: " * string(seed))
        end
        println(f, "Ordering: " * string(order))
        println(f, "Maximum Number of Parents: " * string(max_par))
        println(f, "Score: " * string(score))
        println(f, "Runtime [s]: " * string(runtime))
        println(f, "====================================")
    end
    # show report
    read(report_file, String) |> println

    # Plot and graph gph
    p = gplot(G, nodelabel=var_names) # plot graph
    # Save using Compose
    draw(PDF("project1/output/"*dataset*".pdf", 16cm, 16cm), p)
    write_gph(G, Dict(i => var_names[i] for i in 1:length(var_names)),
    outfile)
end

if length(ARGS) == 2
    inputfilename = ARGS[1]
    outputfilename = ARGS[2]
else
    dataset = "small";
    inputfilename = "project1/data/" * dataset * ".csv"
    outputfilename = "project1/output/" * dataset * ".gph"
end

compute(inputfilename, outputfilename)
```