Project 1: Bayesian Structure Learning

Brandon DAgostino BDAG@STANFORD.EDU
AA228/CS238, Stanford University

1. Algorithm Description

The structure learning approach implemented in this project is a simple directed graph search using the K2 algorithm and Bayesian score. The approach follows closely with the reference implementation presented in Algorithm 5.2 in book section 5.2 and proceeds as follows:

1. A new unconnected graph is initialized with nodes corresponding to each variable in the dataset. The Bayesian score is computed for the empty graph, which is the initial best score.

2. For each node in a provided node ordering, the algorithm iteratively considers adding a directed edge from each preceding node in the ordering to the current node. If adding the edge improves the Bayesian score, the edge is added to the graph and the best score is updated. This process continues until no more edges can be added that improve the score or a maximum number of parents is reached.

   Both the node ordering and maximum number of parents are provided as input parameters to the algorithm, and default to the natural ordering of the variables and infinity (i.e., no limit), respectively.

3. The final graph structure and resulting Bayesian score is returned after all nodes have been processed.

The full Julia implementation is presented along with its supporting code in Section 3. The program runtime (excluding compilation time) and figure reference for the resulting Bayesian network of each dataset is presented below in Table 1.

Dataset	Runtime (s)	Figure
Small	0.229	1
Medium	8.899	2
Large	1363.306	3

Table 1: Program runtimes and figure references for each dataset.

2. Graphs

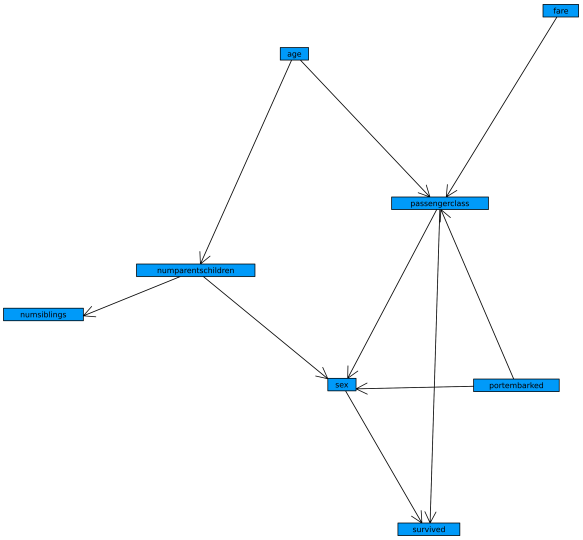

Figure 1: Bayesian Network for small dataset.

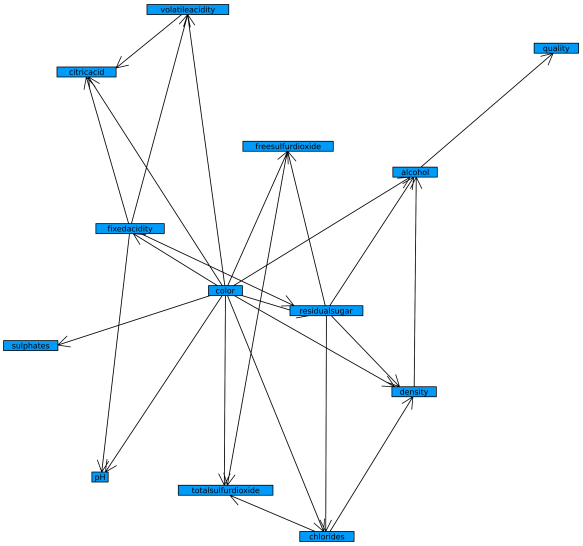

Figure 2: Bayesian Network for medium dataset.

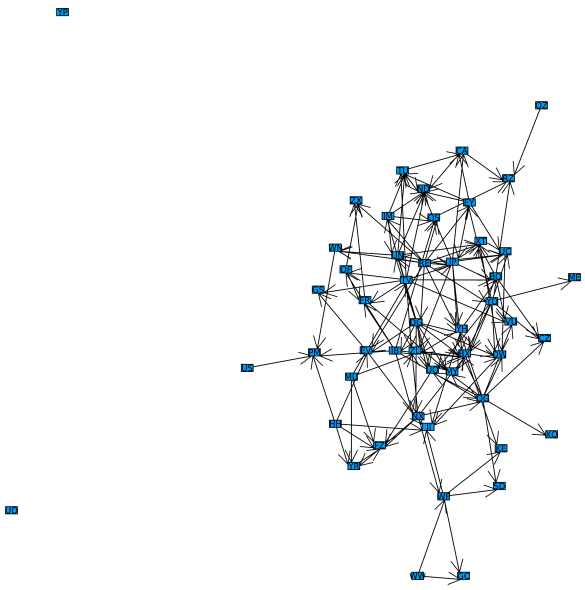

Figure 3: Bayesian Network for large dataset.

3. Code

```
using DelimitedFiles
using LinearAlgebra: dot

using Graphs
using GraphRecipes
using Plots
using SpecialFunctions: loggamma

struct Variable
    name::Symbol
    r::Int
end

Graphs.SimpleDiGraph(variables::AbstractVector{Variable}) = SimpleDiGraph(
    length(variables))

struct BayesianNetwork
    variables::AbstractVector{Variable}
    graph::SimpleDiGraph
end

"Compute the number of parental instantiations of ith variable (``q_i``) of
    Bayesian network."
parental_instantiations(bn::BayesianNetwork, i::Int) = foldl(*, [bn.variables
    [v].r for v in inneighbors(bn.graph, i)], init=1)
```

```
"Get the number of instantiations of the ith variable (``r_i``) of Bayesian
    network."
instantiations(bn::BayesianNetwork, i::Int) = bn.variables[i].r

"Compute counts (``m_{ijk}``) of Bayesian network from data."
function counts(bn::BayesianNetwork, data::Matrix{Int})
    n = length(bn.variables)
    r = [instantiations(bn, i) for i in 1:n]
    q = [parental_instantiations(bn, i) for i in 1:n]
    m = [zeros(q[i], r[i]) for i in 1:n]

    for o in eachcol(data)
        for i in 1:n
            k = o[i]
            parents = inneighbors(bn.graph, i)
            j = 1
            if !isempty(parents)
                # Convert Cartesian parental instantiation coordinates to
    linear index
                j = LinearIndices((r[parents]...,))[o[parents]...,]
            end
            m[i][j, k] += 1
        end
    end
    m
end

"Create uniform pseudocounts for Bayesian network."
function uniform_pseudocounts(bn::BayesianNetwork; value::AbstractFloat=1.0)
    n = length(bn.variables)
    r = [instantiations(bn, i) for i in 1:n]
    q = [parental_instantiations(bn, i) for i in 1:n]
    return [ones(q[i], r[i]) for i in 1:n] * value
end

"Compute Bayesian score for Bayesian network from data."
function bayesian_score(bn::BayesianNetwork, data::Matrix{Int})
    n = length(bn.variables)
    m = counts(bn, data)
    α = uniform_pseudocounts(bn)
    return sum([
        begin
            a = loggamma.(sum(α[i], dims=2))
            b = loggamma.(sum(α[i] + m[i], dims=2))
            c = sum(loggamma.(α[i] + m[i]), dims=2)
            d = sum(loggamma.(α[i]), dims=2)
            sum((a - b) + (c - d))
        end
        for i in 1:n
    ])
```

```
end

struct K2Search
    ordering::AbstractVector{Int}
    max_parents::Union{Some{Int},Nothing}
end

K2Search(ordering::AbstractVector{Int}) = K2Search(ordering, nothing)

function fit(method::K2Search, variables::AbstractVector{Variable}, data::
    Matrix{Int})
    G = SimpleDiGraph(variables)
    best_score = bayesian_score(BayesianNetwork(variables, G), data)

    # For each variable ``v`` in ordering
    for (i, v) in enumerate(method.ordering[2:end])
        println("$i: $v ($(variables[v]))")
        # While ``v`` is allowed to have more parents
        while length(inneighbors(G, v)) < something(method.max_parents, Inf)
            parent_candidate_to_score = Dict{Tuple{Int,Int},AbstractFloat}()

            # For each candidate parent of ``v``
            for parent_candidate in method.ordering[1:i]
                candidate_edge = (parent_candidate, v)

                # Skip if graph already contains candidate edge
                if has_edge(G, candidate_edge)
                    continue
                end

                # Compute score of graph with candidate edge
                add_edge!(G, candidate_edge)
                score = bayesian_score(BayesianNetwork(variables, G), data)
                parent_candidate_to_score[candidate_edge] = score

                # Remove candidate edge
                rem_edge!(G, candidate_edge...)
            end

            # Break if no candidate parent edges
            if isempty(parent_candidate_to_score)
                break
            end

            # If best candidate parent score is worse than current best score
    , move on to next variable in ordering.
            # Otherwise, update graph and best score with best parent
    candidate.
            best_candidate_parent_score, best_candidate_parent_edge = findmax
    (parent_candidate_to_score)
```

```
            if best_candidate_parent_score < best_score
                break
            end
            add_edge!(G, best_candidate_parent_edge)
            best_score = best_candidate_parent_score
        end
    end
    BayesianNetwork(variables, G), best_score
end

function main(args=ARGS)
    # Parse arguments
    input_filename = ""
    try
        input_filename = args[1]
    catch
        printstyled(stderr, "No input file provided\n", color=:red)
        exit(1)
    end
    keyword_args::Dict{Any,Any} = Dict(Pair(split(arg, '=', limit=2)...) for
    arg in args[2:end])
    for k in keys(keyword_args)
        v = keyword_args[k]
        if k == "--ordering"
            keyword_args[k] = map(x -> parse(Int, x), split(v, ','))
        elseif k == "--max-parents"
            keyword_args[k] = parse(Int, v)
        end
    end

    # Print arguments
    println("Input: $(input_filename)")
    println("Keyword Arguments: $(join(["$k => $v" for (k, v) in keyword_args
    ], ", "))")

    input_name = basename(splitext(input_filename)[1])

    # Load data
    data, variables = load_input(input_filename)
    println("Variables (index, name, r):")
    for (i, v) in enumerate(variables)
        println("  - $(i), $(v.name), $(v.r)")
    end
    num_observations = size(data, 2)
    println("Observations: $num_observations")

    # Configure fit parameters
    ordering = haskey(keyword_args, "--ordering") ? keyword_args["--ordering"
    ] : 1:length(variables)
```

```
max_parents = haskey(keyword_args, "--max-parents") ? Some(keyword_args["
--max-parents"]) : nothing
fit_parameters = K2Search(ordering, max_parents)
println("Fit Parameters: $fit_parameters")

# Find best Bayesian network
println("Fitting graph...")
(best_bn, best_score), profile... = @timed fit(K2Search(1:length(
variables)), variables, data)
total_time, compile_time, recompile_time = profile.time, profile.
compile_time, profile.recompile_time
adjusted_time = total_time - (compile_time + recompile_time)
time_output = "Fit graph in $total_time s ($adjusted_time s without
compilation)"
println(time_output)
println("Best Score: $best_score")

# Save output
graph_output_filename = string(input_name, ".gph")
plot_output_filename = string(input_name, ".svg")
score_output_filename = string(input_name, ".score")
time_output_filename = string(input_name, ".time")

## Graph
println("Saving graph to $(graph_output_filename)...")
es = map(e -> (string(variables[e.src].name), string(variables[e.dst].
name)), edges(best_bn.graph))
open(graph_output_filename, "w") do io
    for e in es
        println(io, join(e, ","))
    end
end

## Plot
println("Saving plot to $(plot_output_filename)...")
plot = graphplot(
    best_bn.graph,
    names=[v.name for v in variables],
    curves=false,
    nodeshape=:rect,
    method=:stress,
    size=(1000, 1000),
)
savefig(plot, plot_output_filename)

## Score
println("Saving score to $(score_output_filename)...")
write(score_output_filename, string(best_score))

## Time
```

```
    println("Saving time to $(time_output_filename)...")
    write(time_output_filename, time_output)

    println("Done!")
    return false
end

function load_input(input_filename::String)
    data, header = readdlm(input_filename, ',', Int, header=true)
    variables = vec([Variable(Symbol(v), maximum(data[:, i])) for (i, v) in
    enumerate(header)])
    Matrix(transpose(data)), variables
end

@main
```