Project 1: Bayesian Structure Learning

Qianying Wu wuqy@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

This algorithm combines the K2 search algorithm with a random permutation of the variable orderings that maximizes the Bayesian score.

- First, it randomly selects a fixed percentage (for example, 60%) of data from the data set as the training data set, and the complete data set is used as the validation data set.
- Then, a random shuffle of the variable ordering is executed, which will be applied to the K2 search algorithm in the next step.
- The K2 search algorithm is applied to the training data set with the aforementioned random variable ordering, which generates the optimum Bayesian graph that maximizes the Bayesian score given the training data. The K2 Search algorithm works as below.
  - Initiate with an empty graph and a specified topological ordering of variables.
  - Go through each variable based on the variable ordering, starting from the second variable.
  - For each variable, compare the Bayesian score before and after temporarily adding one additional edge between this variable and each variable before this variable on the topological ordering. Keep track of the variable that results in the highest Bayesian score after adding the edge. Here, instead of directly compute a Bayesian score from scratch, dynamic programming strategy is implemented which only calculate the differential of the new score on the previous score. This largely helps reducing the computation time.
  - Permanently adding the edge between this variable and the one variable that resulted in the highest Bayesian score, if the score is also higher than not adding an additional edge for this variable.
  - Returns the Bayesian network that gives the maximum Bayesian score based on the search algorithm.
- A validation Bayesian score is subsequently calculated based on the Bayesian graph generated above, and the validation data set that the graph is not trained on.
- The system keeps track of the maximum validation Bayesian score and the corresponding Bayesian network. If the resulting validation Bayesian score will be compared with the stored highest validation Bayesian score, and overwrite the stored highest validation Bayesian score if the new value is higher. And in the case that the validation

Bayesian score is the higher, the corresponding Bayesian graph also overwrites the graph on storage.

- The system repeats itself by random shuffles of the variable ordering for fixed amount of cycles (for example, 1000), and then output the Bayesian network that is on storage which gives the maximum validation Bayesian score among all the random variable orderings on the validation data set.

2. Run times

	Small	Medium	Large
Run time (s)	1.23093	5.90994	694.06231

3. Graphs

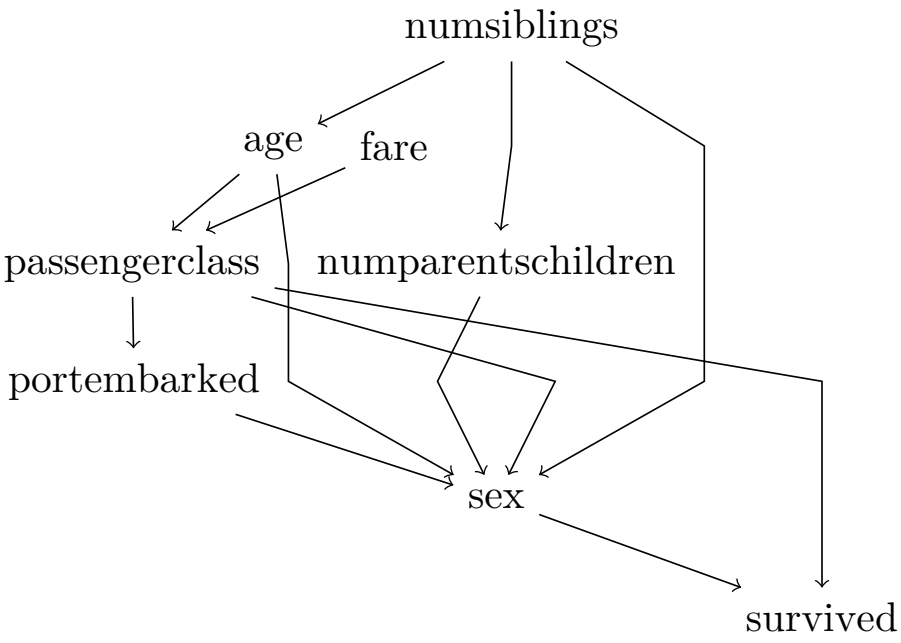

Figure 1: Small Graph

4. Code

```
using LinearAlgebra
using Graphs
using Printf
using DataFrames
using CSV
using TikzGraphs
```

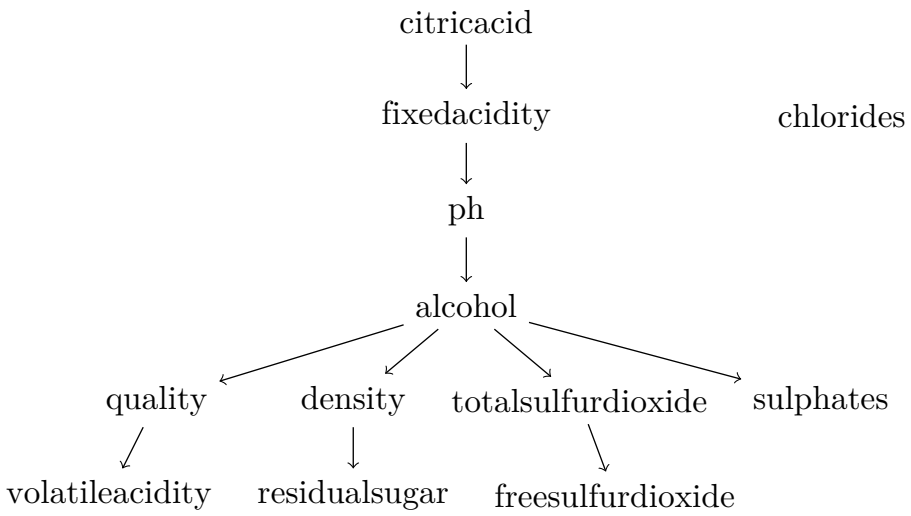

Figure 2: Medium Graph

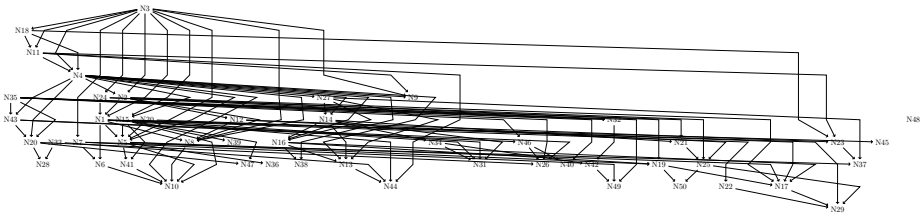

Figure 3: Large Graph

```julia
using SpecialFunctions
using TikzPictures # this is required for saving
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. idx2names is the ordering of the node names that
    you use. Basically, a dictionary that can map the node index to the node
    name.

"""
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
```

```
struct Variable
    name::Symbol
    # name::Symbol = :fixed
    r::Int # number of possible values
end


struct EnhanceK2Search
    not_important::Int
end

function compuateBayesianScore(g, D, vars)
    nVals = size(D, 1)
    nSamples = size(D, 2)

    # Compute the number of values for each variable
    nValCnts = [vars[i].r for i in 1:nVals]

    # Compute the number of instantiations of parents for each variable
    nParents = [1 for i in 1:nVals]
    for i in 1:nVals
        base = 1
        for j in inneighbors(g,i)
            base *= nValCnts[j]
        end
        nParents[i] = base
    end

    # Compute a where a is the pseodocounts
    # Assume uniform distrbution for the prior
    a = [[[1 for k in 1:nValCnts[i]] for j in 1:nParents[i]] for i in 1:nVals
    ]

    # Compute M where M the actual counts
    m = [[[0 for k in 1:nValCnts[i]] for j in 1:nParents[i]] for i in 1:nVals
    ]
    for j in 1:nSamples
        for i in 1:nVals
            val = D[i,j]
            index = 0
            base = 1
            for parent in inneighbors(g, i)
                index += (D[parent,j]-1) * base
                base *= nValCnts[parent]
            end
            index += 1
            m[i][index][val] += 1
        end
    end
```

```
    score = 0.0
    for i in 1:nVals
        for j in 1:nParents[i]
            aTmp = 0
            mTmp = 0
            for k in 1:nValCnts[i]
                score += loggamma(a[i][j][k] + m[i][j][k])
                score -= loggamma(a[i][j][k])
                aTmp += a[i][j][k]
                mTmp += m[i][j][k]
            end
            score += loggamma(aTmp)
            score -= loggamma(aTmp + mTmp)
        end
    end
    return score

end

function incrementalScore(D, from, to, vars, g)
    nSamples = size(D, 2)
    nVals = size(D, 1)
    # Compute the number of values for each variable
    nValCnts = [vars[i].r for i in 1:nVals]

    # Compute the number of instantiations of parents for each variable
    nParents = [1 for i in 1:nVals]
    for i in 1:nVals
        base = 1
        for j in inneighbors(g,i)
            base *= nValCnts[j]
        end
        nParents[i] = base
    end

    # calulate m[to, diffInstations, k]
    newM = [[0 for k in 1:nValCnts[to]] for j in 1:nParents[to]]
    oldM = [[0 for k in 1:nValCnts[to]] for j in 1:nParents[to]/nValCnts[from
    ]]

    # compute oldM
    rem_edge!(g, from, to)
    for j in 1:nSamples
        val = D[to,j]
        index = 0
        base = 1
        for parent in inneighbors(g, to)
            index += (D[parent,j]-1) * base
            base *= nValCnts[parent]
        end
```

```
            index += 1
            oldM[index][val] += 1
        end

        # compute newM
        add_edge!(g, from, to)
        for j in 1:nSamples
            val = D[to,j]
            index = 0
            base = 1
            for parent in inneighbors(g, to)
                index += (D[parent,j]-1) * base
                base *= nValCnts[parent]
            end
            index += 1
            newM[index][val] += 1
        end


        # Compute diff
        diffInstations = nParents[to] - nParents[to] / nValCnts[from]
        diff = 0
        diff += diffInstations * (loggamma(nValCnts[to])) - diffInstations *
       nValCnts[to] * (loggamma(nValCnts[to]))

        # println(nParents[to], ' ', nValCnts[from])
        for j in 1:Int(nParents[to]/nValCnts[from])
            mTmp = 0
            for k in 1:nValCnts[to]
                mTmp += oldM[j][k]
                diff -= loggamma(1 + oldM[j][k])
            end
            diff -= -loggamma(nValCnts[to] + mTmp)
        end

        for j in 1:nParents[to]
            mTmp = 0
            for k in 1:nValCnts[to]
                mTmp += newM[j][k]
                diff += loggamma(1 + newM[j][k])
            end
            diff += -loggamma(nValCnts[to] + mTmp)
        end

        return diff
    end


    function k2Search(D, indexes, vars)
        # nRow denotes the number of variables in the dataset.
```

```
    nRow = size(D, 1)
    # nCol denotes the number of samples in the dataset.
    nCol = size(D, 2)

    # run the loops and get the best graph in a greedy way
    g = SimpleDiGraph(nRow)
    y = compuateBayesianScore(g, D, vars)
    for i in 2:nRow
        while true
            yBest = -Inf
            jBest = -1
            for j in 1:(i-1)
                if !has_edge(g, indexes[j], indexes[i])
                    add_edge!(g, indexes[j], indexes[i])
                      yTmp = y + incrementalScore(D,indexes[j], indexes[i],
   vars, g)
                    if yTmp > yBest
                        yBest = yTmp
                        jBest = j
                    end
                    rem_edge!(g, indexes[j], indexes[i])
                end
            end
            if yBest > y
                y = yBest
                add_edge!(g, indexes[jBest], indexes[i])
            else
                break
            end
        end
    end
    return g
end


 function fit(method::EnhanceK2Search, vars, D)
    # Use 60% of D for learning, the other 20% for score comparison between
    different viariable ordering
    ncol = size(D,2)
    cols = collect(1:ncol)
    colsL = Random.randsubseq(cols, 0.6)
    colsC = setdiff(cols, colsL)
    DL = D[:, colsL]
    DC = D[:, colsC]

    # random permute a variable ordering
    nvars = size(D,1)
    varidx = collect(1:nvars)

    score_best = -Inf
```

```
    G_best = SimpleDiGraph(nvars)
    npermute = 1000

    for i in 1:npermute
        indexes = shuffle(varidx)

        # generate a BN using K2 and the subset of data DL
        G = k2Search(D, indexes, vars)

        # compute the score using the complete dataset D
        score = compuateBayesianScore(G, D, vars)

        if score_best < score
            score_best, G_best = score, G
        end
    end

    return G_best

end


function preprocess(D)
    nRow = size(D, 1)
    nCol = size(D, 2)
    for i in 1:nRow
        uniques = unique(D[i,:])
        map = Dict()
        for j in 1: length(uniques)
            map[uniques[j]] = j
        end
        for j in 1: nCol
            D[i,j] = map[D[i,j]]
        end
    end
    return D
end


function compute(infile, outfile)

    df = DataFrame(CSV.File(infile))
    nrow = size(df, 1)
    ncol = size(df, 2)
    vars = Vector{Variable}()
    for i in 1:ncol
        nunique = length(unique(df[:,i]))
        name = names(df)[i]
        push!(vars, Variable(Symbol(name), nunique))
    end
```

```
    idx = collect(1:ncol)
    idx2names = Dict(idx .=> names(df))
    g = SimpleDiGraph(ncol)
    method = EnhanceK2Search(0)

    D = (Matrix(df))
    D = permutedims(D)
    D = preprocess(D)

    @time g = fit(method, vars, D)

    write_gph(g::DiGraph, idx2names, outfile)

    t = TikzGraphs.plot(g, names(df))
    prefixIndex = 0
    posfixIndex = 0
    for i in 1:length(infile)
        if infile[i] == '/'
            prefixIndex = i
        end
        if infile[i] == '.'
            posfixIndex = i
        end
    end
    suffix = chop(infile, head = prefixIndex, tail=length(infile)-
    posfixIndex+1)
    filename = "graph_" * String(suffix)
    TikzPictures.save(PDF(filename), t)
    TikzPictures.save(SVG(filename), t)
    TikzPictures.save(TEX(filename), t)
end


ARGS = ["data/large.csv", "data/large.gph"]
# ARGS = ["data/medium.csv", "data/medium.gph"]
 # ARGS = ["data/small.csv", "data/small_out.gph"]
# ARGS = ["example/example.csv", "example/example_out.gph"]


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

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

compute(inputfilename, outputfilename)
```