Project 1: Bayesian Structure Learning

Bradley Collicott collicott@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

Given the computational complexity of searching the graph space for the optimal Bayesian network structure, this algorithm searches the node-ordering space and creates a Bayesian network using the k2 algorithm for each ordering (1; 2). A modified particle swarm optimization (PSO) global optimization routine was developed based on (3). The general formulation of particle swarm optimization is:

1. Initialize n_p particles with a random position vector in the node-ordering space
2. Using the k2 algorithm, create the Bayesian network associated with each particle
3. Based on each particle's personal best (pbest) and the global best (gbest) score/ordering, update the velocity of each particle
4. Update the position of each particle using the velocity from Step 3
5. Mutate the position of a small percentage of the particles
6. Check the termination condition
7. Repeat from Step 2

A more detailed treatment of the particle swarm algorithm is given in the following subsection.

1.1 Particle Swarm Optimization

1.1.1 Initialization

Each particle is initialized with a random node ordering, a random velocity vector with values on [0, 1], and current and pbest scores of $-INF$.

1.1.2 Scoring

The k2 algorithm is used to create the graph to be scored using the Bayesian score. This classical Bayesian structure learning method accepts an ordering of nodes as the input and determines the optimal graph edges under the constraint that nodes can only be children of nodes that come before them in the initial ordering.

1.1.3 Velocity Update

The velocity is computed as a function of the particle's current position x_i, velocity v_i, and pbest position p_i as well as the gbest position p_g and a set of empirically determined coefficients c_1, c_2, and c_3. The coefficients represent the weighting of particle inertia, local information, and global information, respectively, in updating the particle's velocity.

$$v_i^{t+1} = c_1 v_i^t + c_2(p_i - x_i^t) + c_2(p_g - x_i^t) \tag{1}$$

1.1.4 Position Update

The position is updated as the sum of the previous position and the updated velocity.

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{2}$$

Since this operation results in floating point values, (3) proposes sorting the resulting values of x_i^{t+1} smallest-to-large and using the corresponding ordering of nodes as the updated position. This method was incorporated in the present algorithm.

1.1.5 Mutation

At each iteration of the particle swarm algorithm, a small percentage of the particles were randomly selected to have two random nodes in their ordering switch. The percentage was fixed at 10% for these simulations.

1.1.6 Termination Condition

The routine was terminated if the gbest score did not improve over 10 consecutive iterations

1.2 Execution Time

The code was parallelized to save on computation time by utilizing a 4 core processor. The resulting execution times are shown in the following table. Note that this method was too computationally expensive to run for the large dataset (attempted to run overnight, caused computer crash.), so the k2 method was applied to a random node ordering to obtain the final graph.

Small	Medium	Large
172.2 s	1266.0 s	13946.3 s

2. Graphs

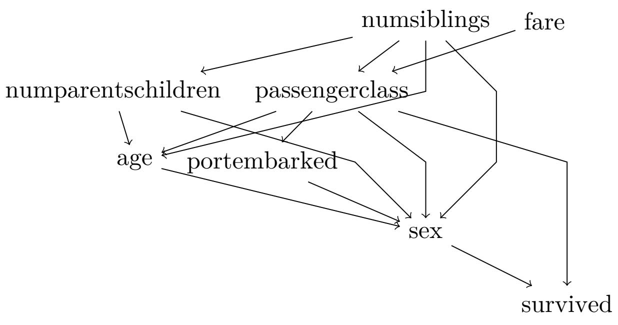

Figure 1: Small Dataset Graph

3. conclusion

A more thorough treatment of this algorithmic approach would have likely yielded better results. This includes tuning the weighting parameters for updating particle positions, adding consideration of the best solution in the neighborhood of each particle, and varying the number of particles per simulation.

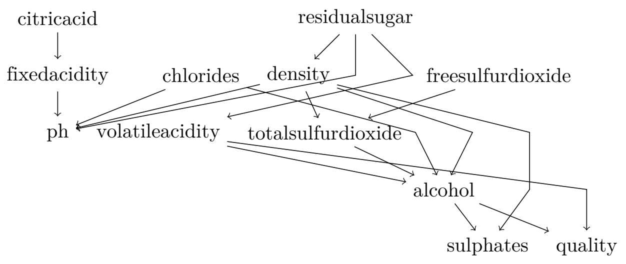

Figure 2: Medium Dataset Graph

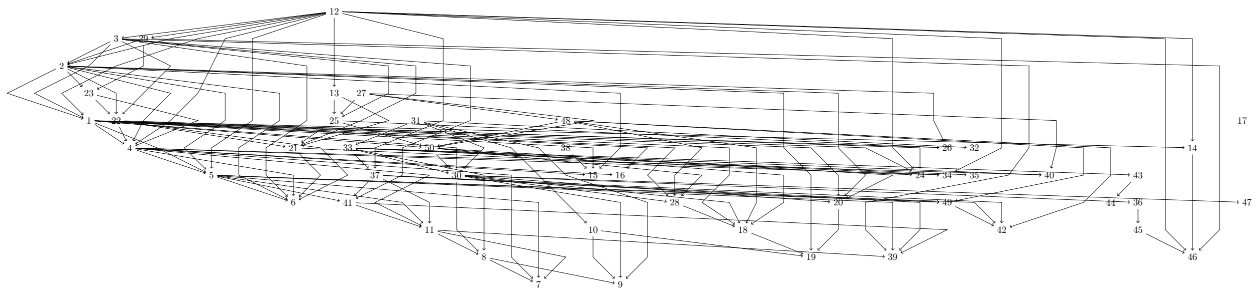

Figure 3: Large Dataset Graph

4. Code

```
########################################################################
#                            JULIA PACKAGES                            #
########################################################################
using LinearAlgebra
using LightGraphs
using DataFrames
using CSV
using Printf
using SpecialFunctions
using TikzGraphs
using TikzPictures
using Random

########################################################################
#                             USER STRUCTS                             #
########################################################################
struct Variable
    name::Symbol
    m::Int
end

mutable struct Particle
    i::Int                      # particle number
    pos::Vector{Int}            # current particle position
    vel::Vector{Float64}        # current particle velocity
    score::Float64              # current particle score
    pos_pbest::Vector{Int}      # best    particle position
    score_pbest::Float64    # best    particle score
    """
        Constructor Particle(i::Int, n_vars::Int)

    Initialize particle with bounded random position and velocity and
    -Inf score
    """
    function Particle(i::Int, n_vars::Int)
        zn = zeros(n_vars)
        pos_0 = shuffle([1:n_vars...])
        return new(i, pos_0, rand(0:1e-5:1,n_vars), -Inf, pos_0, -Inf)
    end
end

mutable struct PSO
    P::Vector{Particle}     # array of n_p particles
    n_p::Int                # number of particles used in process
    n_vars::Int             # number of variables in graph
    pos_gbest::Vector{Int}  # global best particle position
    score_gbest::Float64    # global best particle score
    mut_frac::Float64       # percentage of particles to mutate at each iteration
    mut_n::Int              # number of mutations (2 * number of swaps)
    search_fcn              # function handle of graph search function
    gbest_i::Int            # number of iterations with same gbest score
    gbest_i_max::Int        # termination condition
    """
        Constructor PSO(n_p::Int, n_vars::Int, search_fcn)

    Initialize particle swarm optimization process with i particles,
    n_vars graph variables, and search algorithm search_fcn. Set initial
    gbest position to zeros and gbest score to -Inf. Set default
    parameters as: mut_frac=0.1, mut_n=2, gbest_i_max=5.
    """
    function PSO(n_p::Int, n_vars::Int, search_fcn)
        P = [Particle(i, n_vars) for i in 1:n_p]
        return new(P, n_p, n_vars, zeros(Int, n_vars), -Inf, 0.1, 2, search_fcn, 0,
    5)
    end
end

##########################################################################
#                          OUTPUT FUNCTIONS                              #
##########################################################################
"""
    write_gph(dag::SimpleDiGraph, idx2names, filename)

Takes a SimpleDiGraph, a Dict of index to names and a output filename to write
the graph in ‘gph‘ format.
"""
function write_gph(dag::SimpleDiGraph, 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

"""
    write_score(score::Float64, filename::String)

Write score to <filename>.score.
"""
function write_score(score, filename)
    open(filename, "w") do io
        @printf(io, "%f", score)
    end
end

"""
    write_image(dag::SimpleDiGraph, filename::String)

Create visualization directed acyclical graph dag and save to
<filename>.pdf.
"""
function write_image(dag::SimpleDiGraph, filename)
    t = TikzGraphs.plot(dag)
    TikzPictures.save(PDF(filename),t)
end

"""
    output(
        graph_struct::SimpleDiGraph,
        best_score::Float64,
        var_names::Array{String},
        file_output::String
    )

Wrapper for outputting .gph, .score, and .pdf files.
"""
function output(
    graph_struct::SimpleDiGraph,
    best_score::Float64,
    var_names::Array{String},
    file_output::String
)

    file_output_graph = file_output*".gph"
    file_output_fig = file_output
    file_output_score = file_output*".score"

    write_gph(graph_struct, var_names, file_output_graph)
    write_image(graph_struct, file_output_fig)
    write_score(best_score, file_output_score)
end

########################################################################
```

```
#                           BAYES NET FUNCTIONS                            #
############################################################################

"""
    get_n_var_values(
        graph_vars::Vector{Variable},
        graph_struct::SimpleDiGraph,
        n_vars::Int
    )

Get the number of possible values (r_i) for each variable in graph_vars.
"""
function get_n_var_values(
    graph_vars::Vector{Variable},
    graph_struct::SimpleDiGraph,
    n_vars::Int
)
    # Number of possible values (instantiations) for each variable.
    n_var_values = [graph_vars[i].m for i in 1:n_vars]

    return n_var_values
end

"""
    get_n_pa_values(
        graph_vars::Vector{Variable},
        graph_struct::SimpleDiGraph,
        n_vars::Int,
        n_var_values
    )

Get the number of possible parental instantiations (q_i) at each node for
each possible node value (k).
"""
function get_n_pa_values(
    graph_vars::Vector{Variable},
    graph_struct::SimpleDiGraph,
    n_vars::Int,
    n_var_values
)
    # Number of possible values (instantiations) for the parents of each variable.
    n_pa_values = [prod([n_var_values[j] for j in inneighbors(graph_struct, i)]) for
     i in 1:n_vars]

    return n_pa_values
end

function sub2ind(siz, x)
    k = vcat(1, cumprod(siz[1:end-1]))
    return dot(k, x .-1) + 1
end

"""
```

```
    bayes_net_counts(
        graph_vars::Vector{Variable},
        graph_struct::SimpleDiGraph,
        dataset::Matrix{Int},
        n_vars::Int
    )

Given a set of variables, a graph structure, and discrete data set,
compute the count matrices for each variable.
"""
function bayes_net_counts(
    graph_vars::Vector{Variable},
    graph_struct::SimpleDiGraph,
    dataset::Matrix{Int},
    n_vars::Int
)
    # Number of possible values (instantiations) for variable and their parents
    n_var_values = get_n_var_values(graph_vars, graph_struct, n_vars)
    n_pa_values = get_n_pa_values(graph_vars, graph_struct, n_vars, n_var_values)

    # Pre-allocated matrix to store counts for variable instantiation i
    # given a parent instantiation j
    count_matrix = [zeros(n_pa_values[i], n_var_values[i]) for i in 1:n_vars]

    # eachcol creates a generator that iterates over the second dimension
    # of the dataset, returning the columns as AbstractVector views
    for obs_col in eachcol(transpose(dataset))
        # first loop:
        #   loop through each observation vector in the dataset
        for i_var in 1:n_vars
            # second loop:
            #   loop through each variable in the current observation
            #   vector
            # observation value relevant to the current looping variable
            obs_value = obs_col[i_var]

            # parents of current looping variable
            parents = inneighbors(graph_struct, i_var)

            # increment the counter for the current looping variable
            # given parental instantiation j and observation value k
            j = 1
            if !isempty(parents)
                j = sub2ind(n_var_values[parents], obs_col[parents])
            end
            count_matrix[i_var][j, obs_value] += 1.0
        end
    end
    return count_matrix
end

"""
    uniform_prior(
        graph_vars::Vector{Variable},
        graph_struct::SimpleDiGraph,
        n_vars::Int
    )
Returns alpha vector for uniform Dirchlet prior.
"""
function uniform_prior(
    graph_vars::Vector{Variable},
    graph_struct::SimpleDiGraph,
    n_vars::Int
)
    # Number of possible values (instantiations) for variable and their parents
    n_var_values = get_n_var_values(graph_vars, graph_struct, n_vars)
    n_pa_values = get_n_pa_values(graph_vars, graph_struct, n_vars, n_var_values)
    return [ones(n_pa_values[i], n_var_values[i]) for i in 1:n_vars]
end

"""
    bayesian_score_component(
        count_matrix::Vector{Matrix{Int}},
        alpha::Vector{Matrix{Int}}
    )
Helper function for computing the summation in the Bayesian score equation.
"""
function bayesian_score_component(
    count_matrix,
    alpha
)
    val =  sum(loggamma.(alpha + count_matrix))
    val -= sum(loggamma.(alpha))
    val += sum(loggamma.(sum(alpha, dims=2)))
    val -= sum(loggamma.(sum(alpha, dims=2) + sum(count_matrix, dims=2)))
    return val
end

"""
    bayesian_score(
        graph_vars::Vector{Variable},
        graph_struct::SimpleDiGraph,
        dataset::Matrix{Int},
        n_vars::Int
    )

Compute the Bayesian score associated with graph_struct according to
Algorithm 5.1 in Kochenderfer, Algorithms for Decision Making
"""
function bayesian_score(
    graph_vars::Vector{Variable},
    graph_struct::SimpleDiGraph,
    dataset::Matrix{Int},
    n_vars::Int
)
    count_matrix = bayes_net_counts(graph_vars, graph_struct, dataset, n_vars)
    alpha = uniform_prior(graph_vars, graph_struct, n_vars)
    # alpha = [ones(size(count_matrix[i]))for i in 1:5] # should do the same thing

    # Bayesian score calculated according to P. 96, Eq. 5.5
    return sum(bayesian_score_component(count_matrix[i], alpha[i]) for i in 1:n_vars
    )
end

########################################################################
#                      STRUCTURE LEARNING FUNCTIONS                    #
########################################################################
"""
    function k2(
        graph_vars::Vector{Variable},
        graph_struct::SimpleDiGraph,
        dataset::Matrix{Int},
        n_vars::Int,
        ordering::Vector{Int}
    )

k2 algorithm for Bayesian graph structure learning. Given an ordering of
graph_vars, add edges to the graph_struct that maximize the Bayesian
score for a given dataset.
"""
function k2(
    graph_vars::Vector{Variable},
    graph_struct::SimpleDiGraph,
    dataset::Matrix{Int},
    n_vars::Int,
    ordering::Vector{Int}
)
    for i in 1:n_vars
        parents = []
        current_node = ordering[i]
        prev_best_score = bayesian_score(graph_vars, graph_struct, dataset, n_vars)
        while true
            best_score = -Inf
            best_parent = 0
            for j in 1:i-1
                current_parent = ordering[j]
                if !has_edge(graph_struct, current_parent, current_node)
                    add_edge!(graph_struct, current_parent, current_node)
                    current_score = bayesian_score(graph_vars, graph_struct, dataset
    , n_vars)
                    if current_score > best_score
                        best_score = current_score
                        best_parent = current_parent
                    end
                    rem_edge!(graph_struct, current_parent, current_node)
                end
            end
            if best_score > prev_best_score
                prev_best_score = best_score
                add_edge!(graph_struct, best_parent, current_node)
            else
                break
            end
        end

    end
    return graph_struct, bayesian_score(graph_vars, graph_struct, dataset, n_vars)
end

"""
    Method particle_eval(pso::PSO)

Evaluate the score of each particle in the pso struct using the
embedded search_fcn.
"""
function particle_eval(pso::PSO)

    # increment
    pso.gbest_i += 1

    Threads.@threads for particle in pso.P
        _, particle.score = pso.search_fcn(particle.pos)
        if particle.score > particle.score_pbest
            particle.score_pbest = particle.score
            particle.pos_pbest = particle.pos
        end
        if particle.score_pbest > pso.score_gbest
            pso.score_gbest = particle.score_pbest
            pso.pos_gbest = particle.pos_pbest
            pso.gbest_i = 0
        end

    end

end

"""
    Method particle_update(pso::PSO)

Update the velocity and position of each particle.
"""
function particle_update(pso::PSO)
    for particle in pso.P
        # Compute the particle velocity update as the weighted combination
        # of current, pbest, and gbest states

        particle.vel = pso.C[1]*(particle.vel) +
                       pso.C[2]*(particle.pos_pbest - particle.pos) +
                       pso.C[3]*(pso.pos_gbest      - particle.pos)

        # The position update produces float values. Compute the new
        # particle position by sorting by the most negative values
```

```julia
        # computed in the x_i + v_i position update.
        particle.pos = particle.pos[sortperm(particle.pos + particle.vel)]
    end
end

"""
    Method particle_mutate(pso::PSO)

Mutate a mut_frac percentage of the particles by randomly swapping nodes
in the ordering list.
"""
function particle_mutate(pso::PSO)

    # compute number of particles to mutate
    n_p_mutate = Int64(ceil(pso.mut_frac*pso.n_p))

    # generate random particles to mutate
    mut_idx   = rand(1:pso.n_p, n_p_mutate, 1)

    #generate random nodes to swap
    mut_which = rand(1:pso.n_vars, n_p_mutate, pso.mut_n)

    k = 0
    for particle in pso.P[mut_idx]
        k += 1
        for j in 1:Int64(pso.mut_n/2)
            # swap each pair of random indices
            temp = particle.pos[mut_which[k,j]]
            particle.pos[mut_which[k,j]] = particle.pos[mut_which[k,j*2-1]]
            particle.pos[mut_which[k,j*2-1]] = temp
        end
    end
end

"""
    Method particle_swarm_optimization(pso::PSO)

Wrapper for running the particle swarm optimization routine. Runs until
termination condition has been met.
"""
function particle_swarm_optimization(pso::PSO)
    i = 0
    while pso.gbest_i < pso.gbest_i_max
        i += 1
        particle_eval(pso)
        particle_update(pso)
        particle_mutate(pso)
    end
end

########################################################################
#                           CODE EXECUTION                             #
########################################################################
```

```
function compute(file_dataset::String, file_output::String, n_p::Int)

    dataset_df = DataFrame(CSV.File(file_dataset))

    dataset = Matrix(dataset_df)
    var_names = names(dataset_df)

    n_vars = size(dataset,2)
    graph_struct = SimpleDiGraph(n_vars)
    graph_vars = [Variable(Symbol(var_names[i]),findmax(dataset[:,i])[1]) for i in
    1:n_vars]

    for i in 1:n_vars
        @printf("Variable: %s, # Values: %i\n", graph_vars[i].name, graph_vars[i].m)
    end

    search_fcn(ordering::Vector{Int}) = k2(graph_vars, SimpleDiGraph(n_vars),
    dataset, n_vars, ordering)
    pso = PSO(n_p, n_vars, search_fcn)

    particle_swarm_optimization(pso)

    graph_struct, best_score = pso.search_fcn(pso.pos_gbest)

    @printf("\tFinal Results: \n\t Score: %f \n\t Ordering: ", pso.score_gbest)
    for i in 1:pso.n_vars
        @printf("% i", pso.pos_gbest[i])
    end

    output(graph_struct, best_score, var_names, file_output)

end

runcases = ["small","medium","large"]
which_run = [2]

for i in which_run
    file_dataset = joinpath(@__DIR__,"..","data",runcases[i]*".csv")
    file_output = joinpath(@__DIR__,"..","output",runcases[i],runcases[i])

    @time compute(file_dataset, file_output, 4)
end
```

References

[1] Cooper, G. F., & Herskovits, E. (1992). A Bayesian Method for the Induction of Probabilistic Networks from Data. Machine Learning, 9(4), 309–347. https://doi.org/10.1023/A:1022649401552

[2] Ruiz, C. (2005). Illustration of the K2 algorithm for learning Bayes net structures. WPI Department of Computer Science, 1–7. http://web.cs.wpi.edu/ cs539/s07/Projects/k2_algorithm.pdf

[3] Aouay, S., Jamoussi, S., & Ayed, Y. Ben. (2013). Particle swarm optimization based method for Bayesian Network structure learning. 2013 5th International Conference on Modeling, Simulation and Applied Optimization, ICMSAO 2013, 0–5. https://doi.org/10.1109/ICMSAO.2013.6552569