Project 1: Bayesian Structure Learning

John Greaney-Cheng johngc@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

My algorithm is a combination of local search and simulated annealing with random large jumps and resets. I started with no edges in my initial graph. My algorithm would then do a local search of the current graph to find the neighbor with the highest Bayesian score. Neighbors are the same as the current graph with either one additional edge, one removed edge, or one reversed edge. If the neighbor has an added edge or a reversed edge, then the algorithm runs a topological sort to check if the neighbor is acyclic, and if there's a cycle then the neighbor is discarded.

After that, we compare the Bayesian scores of the current graph and its highest scoring neighbor. If the neighbor scores higher then we set our current graph to be the highest scoring neighbor guaranteed. Unlike the local search algorithm, if the neighbor scores lower then there's a chance we still set our current graph to be the lower scoring neighbor. The probability is $e^{\frac{\Delta}{T}}$ where Δ = highest neighbor score - current graph score, and T is the current temperature, which is a parameter from the simulated annealing aspect of the algorithm. The temperature starts high then cools down over time, which in simulated annealing is symbolic of looking everywhere in the beginning to avoid getting caught in local maximums and focusing solely on (hopefully) the global maximum at the end. In this case, as the temperature cools, $e^{\frac{\Delta}{T}}$ decreases significantly (because $\Delta < 0$), so we're more likely to take a lower scoring neighbor at the beginning of the algorithm than at the end. I added this aspect to try to avoid local maximums that the local search algorithm would be caught in.

Also, we keep track of the highest scoring graph our algorithm finds, and if the highest scoring neighbor has a better score than our recorded highest scoring graph, then we update. After updating the current and highest scoring graphs, we have a chance to "jump" i.e. to change our current graph by a significant amount by randomly adding and removing edges. This is because I want to consider as many graphs as possible, and jumping can introduce new graphs my algorithm may not have discovered otherwise. We jump with a probability $j\frac{T}{T_0}$ where j is a fixed jump probability, T is the current temperature and T_0 is the initial temperature. This means at the very beginning of the algorithm, we jump with probability j, and as time goes on, the temperature will cool and $\frac{T}{T_0}$ will decrease, so we have a lower probability to jump. For the jump, we take the current graph and add/remove a total of k randomly chosen edges where k is a random number between n and $2n$, n being the number of vertices in the graph. Before adding edges, we make sure adding them doesn't make the graph cyclic, otherwise we skip it.

We then have a chance to "reset" i.e. to set our current graph to be the highest scoring graph our algorithm has found. While jumping can be helpful to explore more, it can also take us away from a potential optimal solution. To combat this, we add this reset chance. Unlike the jump probability, the probability of resetting remains constant throughout the

algorithm. We then finish an iteration by updating the current temperature by multiplying it by a cooling rate. We then repeat, starting from a new local search of the current graph, for a fixed number of iterations. After that, we return the highest scoring graph our algorithm found. Below are all the numerical constants the algorithm used, and the runtimes for small.csv, medium.csv and large.csv.

Number Iterations $I = 600$
Initial Temperature $T_0 = 5.0$
Final Temperature $T_f = 0.1$
Cooling Rate $c = (\frac{T_f}{T_0})^I$, to ensure last iteration ended at final temperature
Jump Probability $j = 0.3$
Reset Probability $r = 0.1$
Max Parents = 4, this was overlooked in the explanation above, but the max number of parents for any vertex was capped to help reduce run times. This was checked whenever an edge was being added or reversed, alongside checking for cycles.

Run time for small.csv: 27 seconds
Run time for medium.csv: 16 minutes 24 seconds
Run time for large.csv: 1 hour 57 minutes

2. Graphs

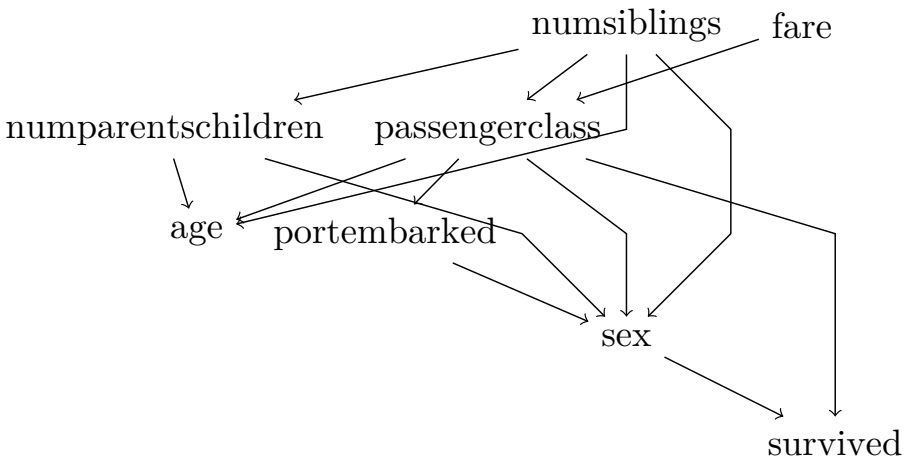

Figure 1: Small Graph Visualization

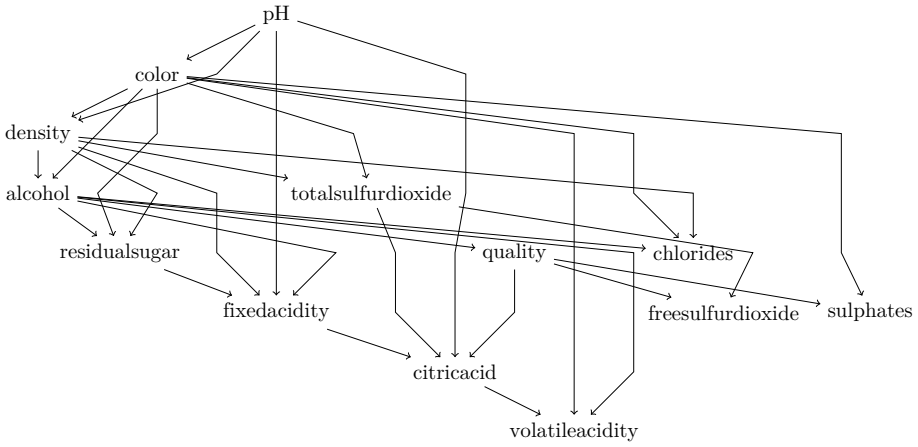

Figure 2: Medium Graph Visualization

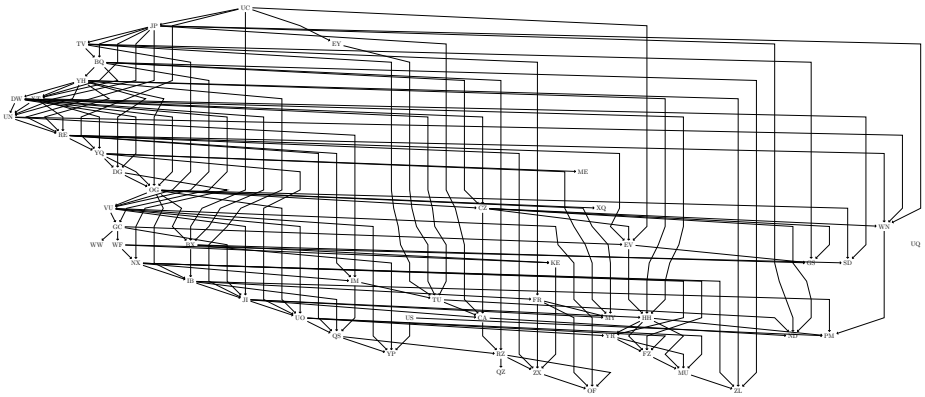

Figure 3: Large Graph Visualization

3. Code

```julia
using Graphs
using Printf
using GraphPlot
using CSV
using DataFrames
using SpecialFunctions
using Random
using TikzGraphs   # for TikZ plot output
using TikzPictures # to save TikZ as PDF


function get_m_count(i, j_tuple, k, df, m_dict)
    #This is a getter function for m_dict
    #Avoids recalculating known m_ijk values to save time
```

```
    #j indexing normally changes depending on #parents
    #To avoid this our parent indexing is now an n-length tuple
    #whose jth index is 0 if node j isn't a parent of node i
    #or its the value that node j is if it is a parent
    key = (i, j_tuple, k)
    if haskey(m_dict, key)
        return m_dict[key]
    end

    #Mask is #samples-length array of booleans, initialized
    mask = trues(nrow(df))
    for (node_idx, node_value) in enumerate(j_tuple)
        if node_value != 0
            #Any trues corresponding to samples with a parent value
            #different than specified become false
            mask .&= df[!, node_idx] .== node_value
        end
    end

    #Also need samples with node i == value k,
    #counts all those samples
    m_ijk = sum(df[mask, i] .== k)
    m_dict[key] = m_ijk
    return m_ijk
end

function get_bayesian_score(dag::DiGraph, m_dict, df, r_values)
    #Computes bayesian score of given graph
    score = 0.0
    alpha_ijk = 1.0
    n = nv(dag)

    for i in 1:n
        parents = inneighbors(dag, i)
        if isempty(parents)
            alpha_ij0 = alpha_ijk * r_values[i]
            m_ij0 = sum(get_m_count(i, ntuple(j -> 0, n), k, df, m_dict) for
    k in 1:r_values[i])
            score += lgamma(alpha_ij0) - lgamma(alpha_ij0 + m_ij0)
            for k in 1:r_values[i]
                m_ijk = get_m_count(i, ntuple(j -> 0, n), k, df, m_dict)
                score += lgamma(m_ijk + alpha_ijk) - lgamma(alpha_ijk)
            end
        else
            parent_ranges = [1:r_values[p] for p in parents]
            #If node i has two parents, one binary and the other trinary
            #then parent_ranges = [[1,2], [1,2,3]]
            for vals in Iterators.product(parent_ranges...)
                #For each tuple in cartesian product of parent parent_ranges
                #From earlier, (1, 1), (1, 2), ..., (2, 3)
```

```
                j_tuple = ntuple(j -> begin
                    idx = findfirst(==(j), parents)
                    idx !== nothing ? vals[idx] : 0
                end, n)
                #j_tuple takes tuple and extends it to be length of all nodes
                #so that any node that isn't a parent has 0 in tuple
                #this is all for consistent indexing for m_dict
                alpha_ij0 = alpha_ijk * r_values[i]
                m_ij0 = sum(get_m_count(i, j_tuple, k, df, m_dict) for k in
    1:r_values[i])
                score += lgamma(alpha_ij0) - lgamma(alpha_ij0 + m_ij0)
                for k in 1:r_values[i]
                    m_ijk = get_m_count(i, j_tuple, k, df, m_dict)
                    score += lgamma(m_ijk + alpha_ijk) - lgamma(alpha_ijk)
                end
            end
        end
    end

    return score
end

function is_acyclic(g::SimpleDiGraph)
    #Returns true if g is acyclic
    try
        topological_sort(g)
        return true
    catch
        return false
    end
end

function simulated_annealing(df, r_values, m_dict)
    #Given the input data, finds the dag with best bayesian score
    #combining local search w/simulated annealing and jumps
    #
    # In each iteration, this algorithm finds the best neighbor of
    # the current graph, it'll switch to it guaranteed if the neighbor is
    better
    # or randomly if the neighbor is worse (more frequently towards the
    beginning
    # which is the simulated annealing portion). There's also a chance to
    jump
    # to a far away graph to avoid getting caught in a local maxium, jump
    probability
    # decreases as time goes on. Finally, to make sure we don't jump away
    from our
    # best graph, we also add a chance to reset the current graph to the best
     graph
```

```
    max_iters = 600
    initial_temp = 5.0
    current_temp = initial_temp
    min_temp = 0.1
    cooling_rate = (min_temp / initial_temp)^(1/max_iters)
    max_parents = 4
    jump_prob = 0.3
    reset_prob = 0.1

    n = size(df, 2)
    current_dag = SimpleDiGraph(n)
    current_score = get_bayesian_score(current_dag, m_dict, df, r_values)
    best_dag = deepcopy(current_dag)
    best_score = current_score

    for iter in 1:max_iters
        best_neighbor = nothing
        best_neighbor_score = -Inf

        #Find neighbor with best bayesian score from neighbors
        for a in 1:n, b in 1:n
            #Iterate through every ordered node pair
            if a == b
                continue
            end
            if has_edge(current_dag, a, b)
                #If there's an edge, check neighbors where
                #that edge is removed or reversed
                remove_edge_dag = deepcopy(current_dag)
                rem_edge!(remove_edge_dag, a, b)
                score = get_bayesian_score(remove_edge_dag, m_dict, df,
r_values)
                if score > best_neighbor_score
                    best_neighbor_score = score
                    best_neighbor = remove_edge_dag
                end

                reverse_edge_dag = deepcopy(remove_edge_dag)
                if length(inneighbors(reverse_edge_dag, a)) < max_parents
                    #Make sure adding parent to a doesn't exceed max parents
                    add_edge!(reverse_edge_dag, b, a)
                    if is_acyclic(reverse_edge_dag)
                        score = get_bayesian_score(reverse_edge_dag, m_dict,
df, r_values)
                        if score > best_neighbor_score
                            best_neighbor_score = score
                            best_neighbor = reverse_edge_dag
                        end
                    end
                end
```

```
            else
                #If there's no edge from a to b,
                #check neighbor with that edge
                add_edge_dag = deepcopy(current_dag)
                if length(inneighbors(add_edge_dag, b)) < max_parents
                    #Make sure adding parent to b doesn't exceed max parents
                    add_edge!(add_edge_dag, a, b)
                    if is_acyclic(add_edge_dag)
                        score = get_bayesian_score(add_edge_dag, m_dict, df,
    r_values)
                        if score > best_neighbor_score
                            best_neighbor_score = score
                            best_neighbor = add_edge_dag
                        end
                    end
                end
            end
        end

        #Move to best neighbor with simulated annealing acceptance
        delta = best_neighbor_score - current_score
        if delta > 0 || rand() < exp(delta / current_temp)
            current_dag = best_neighbor
            current_score = best_neighbor_score
        end

        if current_score > best_score
            best_dag = deepcopy(current_dag)
            best_score = current_score
        end

        #Reset to best DAG with probability reset_prob
        if rand() < reset_prob
            current_dag = deepcopy(best_dag)
            current_score = best_score
        end

        #Jumps to very different dag to explore space
        #Jump probability shrinks as time goes on
        if rand() < jump_prob * (current_temp / initial_temp)
            num_edges = rand(n:2n)
            for _ in 1:num_edges
                a, b = rand(1:n, 2)
                if a != b
                    if has_edge(current_dag, a, b)
                        rem_edge!(current_dag, a, b)
                    elseif length(inneighbors(current_dag, b)) < max_parents
                        temp = deepcopy(current_dag)
                        add_edge!(temp, a, b)
                        if is_acyclic(temp)
```

```
                            add_edge!(current_dag, a, b)
                        end
                    end
                end
            end
            current_score = get_bayesian_score(current_dag, m_dict, df,
    r_values)
        end

        # Print iteration and best score
        @printf("Iter %d | Current=%.2f | Best=%.2f\n", iter, current_score,
    best_score)

        # Cool temperature
        current_temp *= cooling_rate
    end

    return best_dag
end

function write_gph(dag::DiGraph, idx2names, filename)
    #Saves graph as text in output file
    open(filename, "w") do io
        for edge in edges(dag)
            @printf(io, "%s,%s\n", idx2names[src(edge)], idx2names[dst(edge)
    ])
        end
    end
end

function plot_pdf(best_dag, labels, filename)
    #Plots graph
    p = plot(best_dag, labels)
    save(PDF(filename), p)
end

function compute(infile, outfile, plotfile)
    #Main Function
    #Computes Bayesian Network Structure to fit input data

    #Read input file and initialize useful variables
    #r_values[i] := #Different values for node i (max of i column)
    #m_dict stores previous computed m_ijk values, more on that in
    get_m_count
    df = CSV.read(infile, DataFrame)
    labels = names(df)
    idx2names = Dict(i => label for (i, label) in enumerate(labels))
    n = length(labels)
    r_values = [maximum(df[!, i]) for i in 1:n]
    m_dict = Dict{Tuple{Int, Tuple{Vararg{Int}}, Int}, Int}()
```

```
    #Computes Best Graph for input data using simulated annealing
    best_dag = simulated_annealing(df, r_values, m_dict)

    #Write and Plot Graph in Output Files
    write_gph(best_dag, idx2names, outfile)
    plot_pdf(best_dag, labels, plotfile)
end


if length(ARGS) != 3
    #Changed length from 2 to 3 to output graph plot pdf as well
    error("usage: julia project1.jl <infile>.csv <outfile>.gph <plotfile>.pdf
    ")
end

inputfilename = ARGS[1]
outputfilename = ARGS[2]
plotfilename = ARGS[3]

compute(inputfilename, outputfilename, plotfilename)
```