Problem 1: Calculating Long-Term Fraction of Time

We know that the long-term fraction of time of a random walk starting from node 4 is converges asymptotically to the dominant eigenvector, P.

This can be calculated by the following code:

using LinearAlgebra

using SparseArrays

## Taking only the first 7 components

A = [0 1 1 1 1 0 1

1 0 0 0 0 0 0

0 0 0 0 1 1 1

1 1 0 0 1 1 0

1 1 1 1 0 0 0

0 0 1 0 0 0 1

0 0 0 0 0 1 0]

## Get the inverse of the diagonal degree matrix

d = vec(sum(A, dims=2))

D\_inv = Diagonal(1.0 ./ d)

## Find P

P = Matrix(A' \* D\_inv)

vals, vecs = eigen(P)

## Find the eigenvector corresponding to the largest eigenvalue

max\_val\_ind = findmax(abs.(vals))[2]

println("Largest Eigenvector:")

probs = Float64.(vecs[:, max\_val\_ind]/sum(vecs[:, max\_val\_ind]))

Which returns the following dominant eigenvector:

Largest Eigenvector:

7-element Vector{Float64}:

0.08090614886731418

0.04854368932038853

0.19417475728155334

0.038834951456310766

0.09061488673139159

0.3106796116504853

0.23624595469255635

This is a unique solution for a strongly connected graph, so there is no difference when starting from node 1 versus node 4, the long-term fraction of time is the same.

Problem 2: Implementing Sparse Operation Function

1. General case matrix-vector product

function mv\_prod(B, operation, A, x)

## We consider that if non-edges are 2 and edges are 3, split this into 2 arrays,

## Where the second array is the sparse adjacency matrix A

## e.g., for one row: [2 2 2 2 2 2 2 ...] .+ [ 0 0 0 1 1 0 1 ...] = [2 2 2 3 3 2 3 ...] = M

## therefore taking the matrix-vector product M \* x, we can say in the general case,

## where B is the matrix of the integer offset of size nxn, and A is the original sparse matrix

## (B .+/- A) \* x = (B \* x) .+/- (A \* x) = (B \* sum(x)) .+/- (A \* x)

if operation == "+"

return (B \* sum(x)) .+ (A\*x)

elseif operation == "-"

return (B \* sum(x)) .- (A\*x)

end

end

2. Power method function to find the largest eigenvalue based on the wiki-cats graph.

function power\_method(M, max\_iter=10000, tol=10e-4)

## Power method using Von Mises algorithm and Rayleigh quotient to find the largest eigenvalue of M

## Create random vector and normalize

x = sparse(rand(size(M,2)))

x\_k = x / norm(x)

λ\_p = 0.0

for iter in 1:max\_iter

## Calculate matrix vector product for offset of 2 .+ A

x\_k\_1 = mv\_prod(2, "+", M, x\_k)

x\_k\_1 /= norm(x\_k\_1)

## Calculate Rayleigh quotient

λ = dot(x\_k\_1, x\_k') / dot(x\_k', x\_k)

## Check tolerance

e\_res = norm(x\_k\_1 .- λ\*x\_k)

# Δλ = λ - λ\_p ## Check difference between iters

# println("Iter: $iter \tresidual: $e\_res\t| Δλ = $Δλ") ## Display output

if e\_res < tol

return λ, iter, x\_k\_1

end

## Normalize and update

λ\_p = λ

x\_k = x\_k\_1

end

error("convergence was not reached within $max\_iter iterations for tolerance level $tol")

end

To implement, I ran this function over 10 tests to see if the eigenvalue would change significantly between function calls.

iters\_to\_coverge = []

λ\_values = []

tol = 10e-4

max\_iters = 10000

tests = 10

for t in 1:tests

local eigenvalue

local iterations

local eigenvector

eigenvalue, iterations, eigenvector = power\_method(A, max\_iters, tol)

println("Test $t converged to $eigenvalue in $iterations iterations")

push!(iters\_to\_coverge, iterations)

push!(λ\_values, eigenvalue)

end

avg\_iters = ceil(mean(iters\_to\_coverge))

avg\_λ = mean(λ\_values)

println("Average Results over $tests tests based on tolerance of $tol")

println("Average largest eigenvalue: $avg\_λ")

println("Average iterations to converge: $avg\_iters")

This resulted in the following output:

Test 1: Converged to 0.999999999993244 in 2 iterations

Test 2: Converged to 0.9999999999932964 in 2 iterations

Test 3: Converged to 0.9999999999932822 in 2 iterations

Test 4: Converged to 0.9999999999931708 in 2 iterations

Test 5: Converged to 0.9999999999934256 in 2 iterations

Test 6: Converged to 0.9999999999934432 in 2 iterations

Test 7: Converged to 0.99999999999331 in 2 iterations

Test 8: Converged to 0.9999999999932749 in 2 iterations

Test 9: Converged to 0.999999999993165 in 2 iterations

Test 10: Converged to 0.999999999993299 in 2 iterations

Average Results over 10 tests based on tolerance of 0.001

Average largest eigenvalue: 0.9999999999932913

Average iterations to converge: 2.0

Problem 3: Implementation of semi-supervised learning with fixed-value nodes

a. Given that

Thus, we have shown that

b.

For the semi-supervised learning case, we fix some labels at their known values, and leave the rest unknown. I will denote the fixed labels as and the unknown labels as

Considering the form from before: , Where Aff are the rows and columns of A associated with fixed labels yf, similarly for Afu, Auf, and Auu.

We can then split this in two parts:   
We know that yf are the fixed values, so we denote them as   
  
Then, substituting into the second equation:

Under the assumption that Auu is nonsingular:

Therefore, we have solved the system for the remaining unknowns while keeping labels fixed at their values in the semi-supervised case.