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 $x_1 = 10$

$$A = \begin{pmatrix} \alpha & a^T \\ c & B \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ y \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$Then: Ax = b \Rightarrow \begin{pmatrix} \alpha & a^T \\ c & B \end{pmatrix}\begin{pmatrix} x_1 \\ y \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} \alpha & a^T \\ c & B \end{pmatrix}\begin{pmatrix} 10 \\ y \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$\Rightarrow \begin{matrix} 10\alpha + a^T y = b_1 \\ 10c + By = b_2 \end{matrix} \Rightarrow \begin{matrix} a^T y = b_1 - 10\alpha \\ By = b_2 - 10c \end{matrix}$$

Thus, we have shown that $By = b_2 - 10c$

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 y_f and the unknown labels as y_u

Considering the form from before: $Ax = b \Rightarrow \begin{pmatrix} A_{ff} & A_{fu} \\ A_{uf} & A_{uu} \end{pmatrix}\begin{pmatrix} y_f \\ y_u \end{pmatrix} = \begin{pmatrix} b_f \\ b_u \end{pmatrix}$, Where A_{ff} are the rows and columns of A associated with fixed labels y_f, similarly for A_{fu}, A_{uf}, and A_{uu}.

We can then split this in two parts:

$$A_{ff}y_f + A_{fu}y_u = b_f$$
$$A_{uf}y_f + A_{uu}y_u = b_u$$

We know that yf are the fixed values, so we denote them as $y_f = fixed\ values$

Then, substituting into the second equation:

$$A_{uf}(fixed\ values) + A_{uu}y_u = b_u$$
$$A_{uu}y_u = b_u - A_{uf}(fixed\ values)$$

Under the assumption that A_{uu} is nonsingular:

$$y_u = A_{uu}^{-1}\left(b_u - A_{uf}(fixed\ values)\right)$$

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