Proyecto

```
using Graphs, Random, Plots, Statistics, StatsPlots, GraphMakie, ColorSchemes, Makie
, CairoMakie, FileIO, ImageMagick, Images, PlutoUI, Distributions
```

Modelo 1

Competencia de dos ideas opuestas

(Parámetros globales)

Consideraciones del modelo:

1. Estructura de la red social: Representada como un grafo donde los nodos son agentes y las aristas son conexiones sociales.

2. Estados de los agentes: Cada agente puede estar en uno de tres estados: creer en una idea ϕ, creer en la idea contraria $\neg\phi$, o ser indiferente $\perp$.

3. Transiciones entre estados: Los agentes pueden cambiar de estado basado en los mensajes que reciben de sus vecinos.

- Hay dos probabilidades/parámetros clave: λ (probabilidad de adoptar una idea) y μ (probabilidad de abandonar una idea).

4. Proceso de difusión:

- En cada paso, un agente es seleccionado al azar para transmitir su creencia.
- Los agentes conectados a este agente pueden cambiar su estado basado en el mensaje recibido.

6. Simulación Monte Carlo: Realizamos múltiples ejecuciones del modelo para obtener resultados estadísticamente significativos.

Implementación:

Definimos los posibles estados de cada individuo/agente, usamos un tipo enumerado `State` para representar los posibles estados de los agentes.

```
@enum State phi notphi ind
```

Definimos la estructura `Agent`, que cuenta con un número de identificación y el estado del agente. La estructura `Agent` es mutable, lo que permite cambiar su estado.

```
mutable struct Agent
    id::Int
    state::State
end
```

Función que simula el recibimiento de un mensaje y que, según los parámetros, el agente decidirá si adoptar o no la creencia de quien envió el mensaje (en caso de que el agente sea indiferente). En caso de que cuente con un estado distinto al mensaje, este decidirá si seguir con su creencia actual o pasar a un estado de indiferencia.

```
receive_message! (generic function with 1 method)
```

```
function receive_message!(agent::Agent, message::State, λ::Float64, μ::Float64)
    if agent.state == ind
        if rand() < λ
            agent.state = message
        end
    elseif agent.state != message
        if rand() < μ
            agent.state = ind
        end
    end
end
```

Definimos la estructura `SocialNetwork`, compuesta por el grafo de la estructura y un vector con los agentes de la misma:

```
struct SocialNetwork
    graph::SimpleGraph
    agents::Vector{Agent}
end
```

Función para inicializar una red social, con todos los agentes en estado indiferente:

```
socialNetwork (generic function with 1 method)
```

```
function socialNetwork(num_agents::Int)
    graph = barabasi_albert(num_agents, 3) # cambiar esto según  el tipo de grafo
    agents = [Agent(i, ind) for i in 1:num_agents]
    SocialNetwork(graph, agents)
end
```

Para este caso en particular, generaremos la red social utilizando el algoritmo Barabási-Albert. Para ver más tipos de grafos visitar esto.

Implementamos una función para setear los estados semilla:

```
set_seed_nodes! (generic function with 1 method)
```

```
function set_seed_nodes!(network::SocialNetwork, phi_seed::Int, notphi_seed::Int)
    network.agents[phi_seed].state = phi
    network.agents[notphi_seed].state = notphi
end
```

En este modelo consideramos dos nodos semillas: uno con el estado ϕ y uno con el estado $\neg\phi$.

Implementamos una función que simula un paso de la cadena, en este se elige un nodo aleatorio y, en caso de que dicho nodo no sea indiferente, se comparte su creencia con sus vecinos:

```
simulate_step! (generic function with 1 method)
```

```
function simulate_step!(network::SocialNetwork, λ::Float64, μ::Float64)
    sender = rand(network.agents)
    if sender.state != ind # verificamos que quien envía no sea indiferente
        for neighbor in neighbors(network.graph, sender.id)
            receive_message!(network.agents[neighbor], sender.state, λ, μ)
        end
    end
end
```

Algo que intentaremos más adelante es mejorar la manera en la que elegimos el nodo aleatorio que manda el mensaje, pues tiene más sentido que se elijan estados no indiferentes.

La siguiente función nos permite contar el número de agentes adeptos a cada uno de los estados:

```
count_beliefs (generic function with 1 method)
```

```
function count_beliefs(network::SocialNetwork)
    beliefs = Dict(s => count(a -> a.state == s, network.agents) for s in
instances(State))
    return beliefs
end
```

Finalmente, implementamos una función que nos permite correr la simulación completa:

```
run_simulation (generic function with 1 method)
```

```
function run_simulation(num_agents::Int, num_steps::Int, λ::Float64, μ::Float64, num_runs::Int)
    results = []
    for _ in 1:num_runs
        network = socialNetwork(num_agents)
        set_seed_nodes!(network, 1, 2)  # Se toma 1 y 2 como nodos semillas

        run_results = []
        for step in 1:num_steps
            simulate_step!(network, λ, μ)
            push!(run_results, count_beliefs(network))
        end

        push!(results, run_results)
    end
    return results
end
```

Veamos un ejemplo de uso:

```
sim_results =
```

```
▸ [[Dict(notphi::State = 1 ⇒ 1, phi::State = 0 ⇒ 1, ind::State = 2 ⇒ 98), Dict(notphi::
```

```
sim_results = run_simulation(100, 1000, 0.5, 0.5, 100)
```

Implementamos una función para visualizar los resultados. Esta función crea un gráfico de líneas que muestra cómo evoluciona el número esperado de agentes en cada estado (ϕ, $\neg\phi$, $\bot$) a lo largo de los pasos de la simulación.

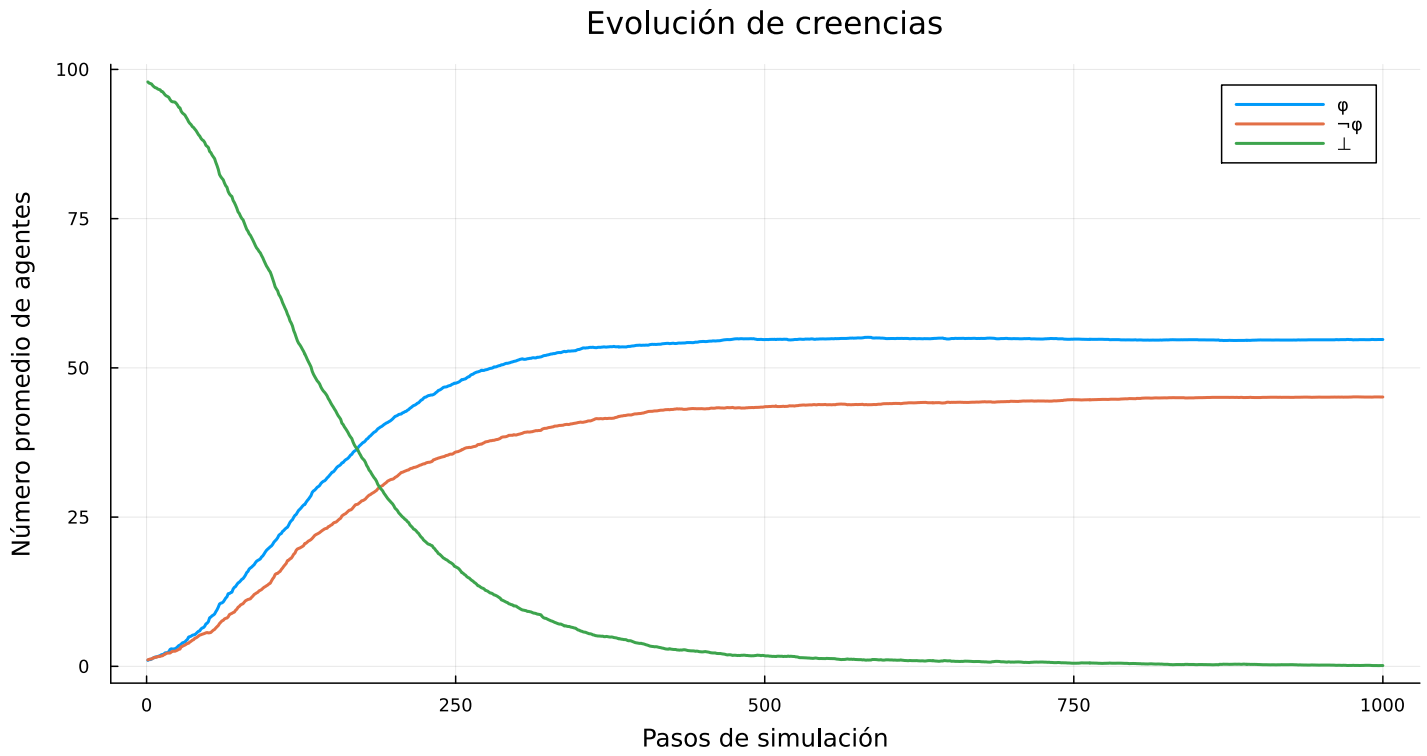
```
plot_belief_evolution (generic function with 1 method)
```

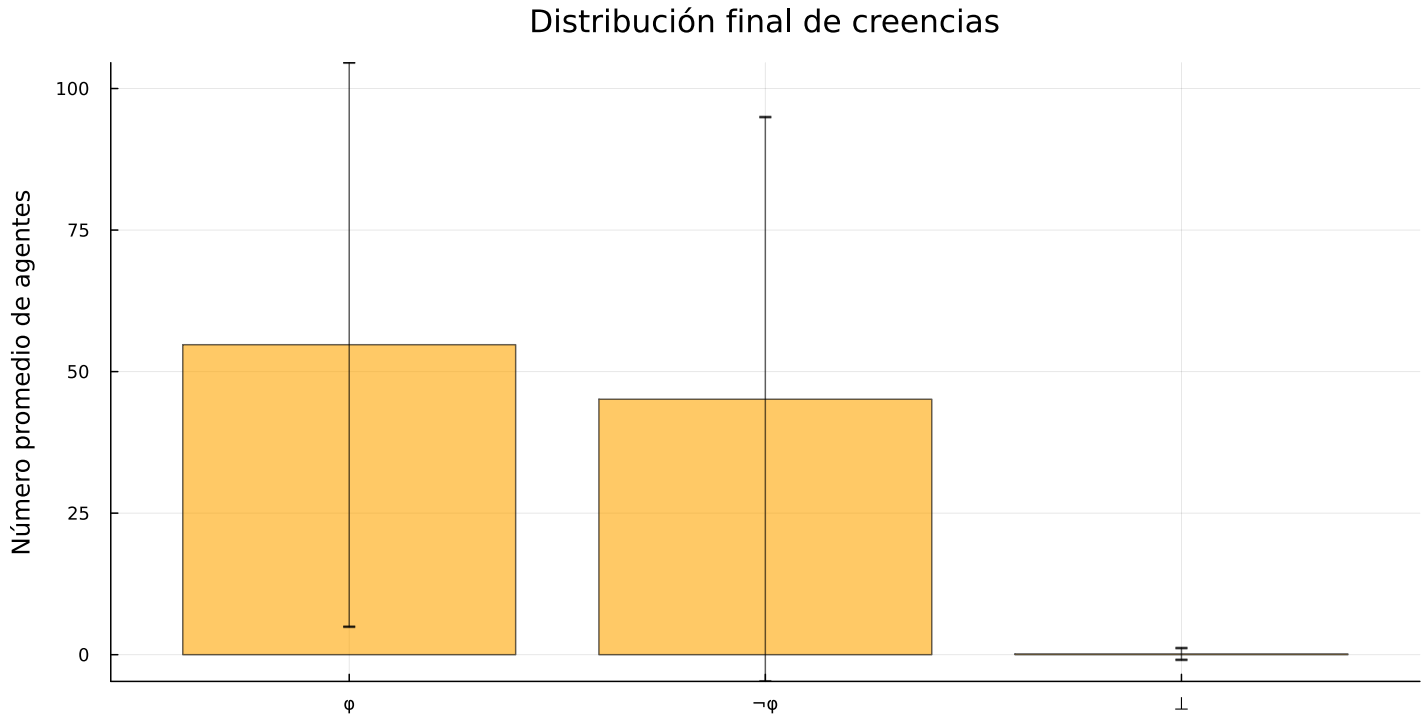
```
function plot_belief_evolution(results)
    num_steps = length(results[1])
    num_runs = length(results)

    phi_means = [mean([run[step][phi] for run in results]) for step in 1:num_steps]
    notphi_means = [mean([run[step][notphi] for run in results]) for step in 1:num_steps]
    ind_means = [mean([run[step][ind] for run in results]) for step in 1:num_steps]

    Plots.plot(1:num_steps, [phi_means notphi_means ind_means],
         label=["φ" "¬φ" "⊥"],
         title="Evolución de creencias",
         xlabel="Pasos de simulación",
         ylabel="Número promedio de agentes",
         lw=2)
end
```

Y una función para visualizar el estado final de la cadena. Esta función crea un diagrama que muestra la distribución del número de agentes en cada estado al final de las simulaciones.

```
plot_final_distribution (generic function with 1 method)
```

```
function plot_final_distribution(results)
    final_states = [last(run) for run in results]

    phi_counts = [state[phi] for state in final_states]
    notphi_counts = [state[notphi] for state in final_states]
    ind_counts = [state[ind] for state in final_states]

    means = [mean(phi_counts), mean(notphi_counts), mean(ind_counts)]
    stds = [std(phi_counts), std(notphi_counts), std(ind_counts)]

    labels = ["φ", "¬φ", "⊥"]

    p = Plots.bar(labels, means,
        yerr=stds,
        title="Distribución final de creencias",
        ylabel="Número promedio de agentes",
        legend=false,
        color=:orange,
        alpha=0.6)
end
```






```
begin
    p1 = plot_belief_evolution(sim_results)
    p2 = plot_final_distribution(sim_results)
    Plots.plot(p1, p2, layout=(2,1), size=(950,1000))
end
```

La anterior gráfica muestra la evolución del número esperado de agentes con la creencia ϕ, $\neg\phi$ e $\bot$. Del total de pasos que se realizaron, no en todos se envió un mensaje desde un nodo a otro, pues si en un paso se elegía un agente indiferente, este no hacía nada. Veamos en promedio cuantos mensajes se enviaban después del total de simulaciones, esto para darnos una idea del número de

mensajes que se requieren para que la cadena converja (empiricamente a un estado en el que la mitad de los agentes cree ϕ y la otra mitad cree $\neg\phi$). Para esto implementamos la función `count_mean_messages_sent`.

```
count_mean_messages_sent (generic function with 1 method)
```

```
function count_mean_messages_sent(simulation_results)
    num_runs = length(simulation_results)
    messages_per_run = zeros(Int, num_runs)

    for (run, result) in enumerate(simulation_results)
        messages = 0
        prev_state = result[1]
        for state in result[2:end]
            if state != prev_state
                messages += 1
            end
            prev_state = state
        end
        messages_per_run[run] = messages
    end

    mean_messages = mean(messages_per_run)
    return mean_messages, messages_per_run
end
```

```
begin
mean_messages, messages_per_run = count_mean_messages_sent(sim_results)
    println("Media de mensajes enviados: $(round(mean_messages, digits=2))")
    println("Rango de mensajes enviados: $(minimum(messages_per_run)) -
$(maximum(messages_per_run))")
end
```

```
Media de mensajes enviados: 77.46
Rango de mensajes enviados: 38 - 439
```

Finalmente, implementaremos una función para visualizar una de las simulaciones:

```
visualize_network_evolution (generic function with 1 method)
```

```
function visualize_network_evolution(simulation_results, network)
    layout = GraphMakie.spring(network.graph)

    fig = Figure(size = (800, 600))
    ax = GraphMakie.Axis(fig[1, 1])

    function update_colors(i)
        return map(simulation_results[i]) do state
            if state == phi
                :blue
            elseif state == notphi
                :red
            else  # state == ind
                :green
            end
        end
    end

    node_colors = Observable(update_colors(1))
    p = graphplot!(ax, network.graph,
        layout = layout,
        node_color = node_colors,
        node_size = 15,
        edge_width = 1,
        edge_color = :gray80
    )

    legend_elements = [
        MarkerElement(color = :blue, marker = :circle, markersize = 15),
        MarkerElement(color = :red, marker = :circle, markersize = 15),
        MarkerElement(color = :green, marker = :circle, markersize = 15)
    ]
    legend_labels = ["φ", "¬φ", "⊥"]
    Legend(fig[1, 2], legend_elements, legend_labels, "Creencias")

    ax.title = "Evolución de la Red Social y Distribución de Creencias"

    slider = PlutoUI.Slider(1:length(simulation_results), default=1,
show_value=true)

    function update_viz(i)
        node_colors[] = update_colors(i)
    end

    return fig, slider, update_viz
end
```

```
count_messages (generic function with 1 method)
```

```
# para contar los mensajes de una única simulación:
function count_messages(simulation_results)
    messages = 0
    prev_state = simulation_results[1]
    for state in simulation_results[2:end]
        if state != prev_state
            messages += 1
        end
        prev_state = state
    end
    return messages
end
```

Corremos una única simulación y la visualizamos:

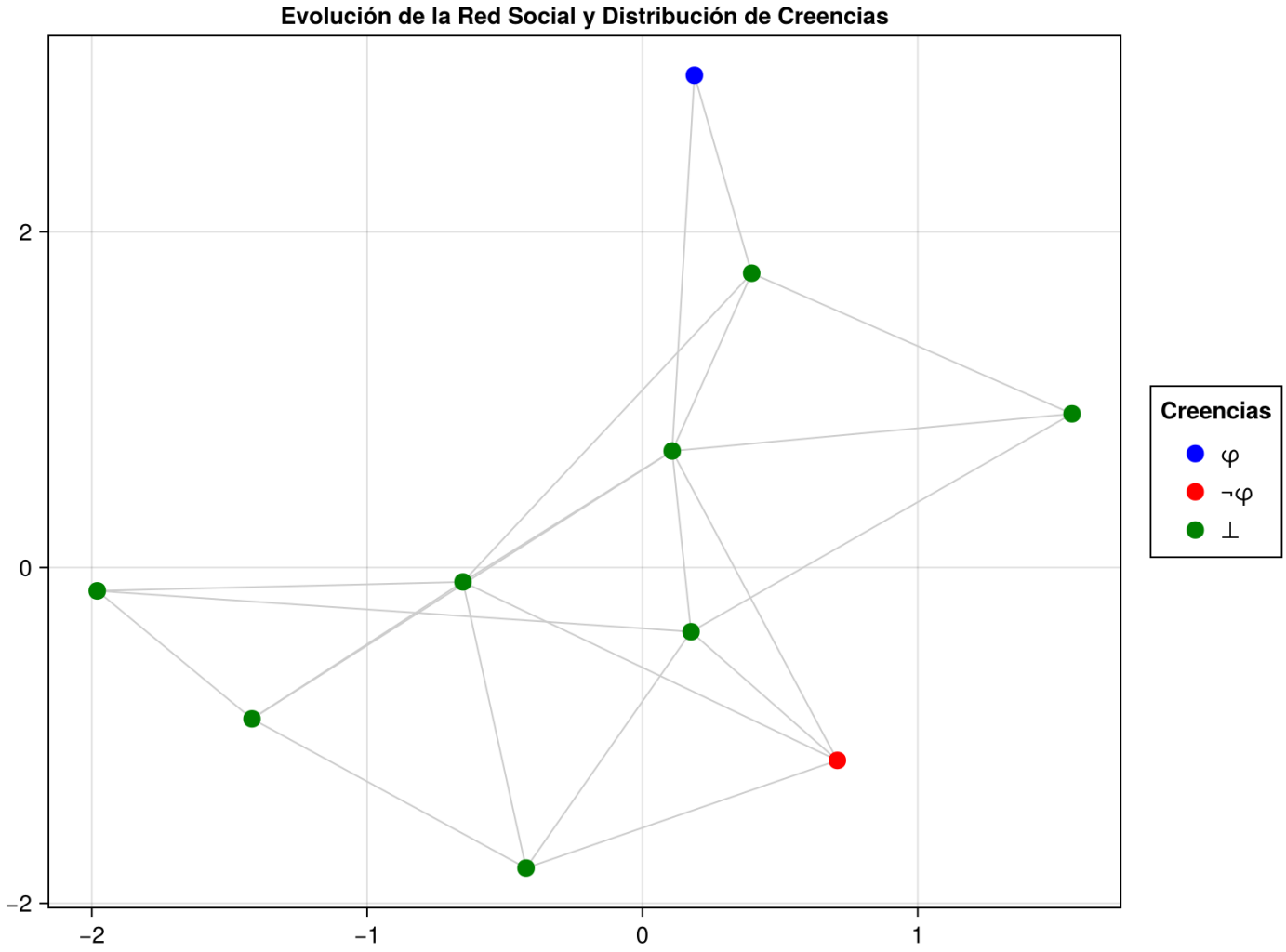
```
run_and_visualize_simulation (generic function with 1 method)
```

```
function run_and_visualize_simulation(num_agents::Int, num_steps::Int, λ::Float64, μ::Float64)
    network = socialNetwork(num_agents)
    set_seed_nodes!(network, 1, 2)

    simulation_results = []
    for _ in 1:num_steps
        push!(simulation_results, [agent.state for agent in network.agents])
        simulate_step!(network, λ, μ)
    end

    messages_sent = count_messages(simulation_results)
    println("Número de mensajes enviados: $messages_sent")

    fig, slider, update_viz = visualize_network_evolution(simulation_results, network)

    return fig, slider, update_viz
end
```

Veamos una simulación: (distinta de las obtenidas en el resultado anterior)

```
fig, slider, update_viz = run_and_visualize_simulation(10, 100, 0.5, 0.5);
```

```
Número de mensajes enviados: 7
```



Modelo 2

Competencia de dos ideas opuestas

(Parámetros locales)

Consideraciones del modelo:

Vamos a modificar el **modelo 1** para que cada agente tenga sus propios parámetros λ y μ. Esto podría permitir una mayor diversidad en el comportamiento de los agentes y potencialmente resultados más interesantes en la simulación.

Implementación:

Los estados posibles de cada agente se conservan: ϕ, $\neg\phi$ y $\bot$.

Modificamos la estructura `Agent` añadiendo los parámetros individuales de cada agente:

```
mutable struct Agent2
    id::Int
    state::State
    λ::Float64  # Probabilidad individual de adoptar una idea
    μ::Float64  # Probabilidad individual de abandonar una idea
end
```

La estructura de la red social se conserva, incluyendo su grafo y un vector con los agentes de la red.

```
struct SocialNetwork2
    graph::SimpleGraph
    agents::Vector{Agent2}
end
```

La función que simula el recibimiento de un mensaje ahora utiliza los parámetros λ y μ específicos de cada agente.

```
receive_message2! (generic function with 1 method)
```

```
function receive_message2!(agent::Agent2, message::State)
    if agent.state == ind
        if rand() < agent.λ
            agent.state = message
        end
    elseif agent.state != message
        if rand() < agent.μ
            agent.state = ind
        end
    end
end
```

Actualizamos la función `socialNetwork`: Ahora toma rangos para λ y μ, y asigna valores aleatorios dentro de esos rangos a cada agente.

```
socialNetwork2 (generic function with 1 method)
```

```
function socialNetwork2(num_agents::Int, λ_range::Tuple{Float64,Float64},
μ_range::Tuple{Float64,Float64})
    graph = barabasi_albert(num_agents, 3)
    agents = [Agent2(i, ind, rand(Uniform(λ_range...)), rand(Uniform(μ_range...)))
for i in 1:num_agents]
    SocialNetwork2(graph, agents)
end
```

Función para establecer los nodos semilla:

```
set_seed_nodes2! (generic function with 1 method)
```

```
function set_seed_nodes2!(network::SocialNetwork2, phi_seed::Int,
notphi_seed::Int)
    network.agents[phi_seed].state = phi
    network.agents[notphi_seed].state = notphi
end
```

Para simular un paso de la cadena:

```
simulate_step2! (generic function with 1 method)
```

```
function simulate_step2!(network::SocialNetwork2)
    sender = rand(network.agents)
    if sender.state != ind
        for neighbor in neighbors(network.graph, sender.id)
            receive_message2!(network.agents[neighbor], sender.state)
        end
    end
end
```

Para contar el número de agentes adeptos a cada uno de los estados:

```
count_beliefs2 (generic function with 1 method)
```

```
function count_beliefs2(network::SocialNetwork2)
    beliefs = Dict(s => count(a -> a.state == s, network.agents) for s in
instances(State))
    return beliefs
end
```

Actualizamos las funciones de simulación: `run_simulation` y `run_and_visualize_simulation` ahora toman rangos para λ y μ en lugar de valores fijos.

```
run_simulation2 (generic function with 1 method)
```

```
function run_simulation2(num_agents::Int, num_steps::Int,
λ_range::Tuple{Float64,Float64}, μ_range::Tuple{Float64,Float64}, num_runs::Int)
    results = []
    for _ in 1:num_runs
        network = socialNetwork2(num_agents, λ_range, μ_range)
        set_seed_nodes2!(network, 1, 2)
        run_results = []
        for step in 1:num_steps
            simulate_step2!(network)
            push!(run_results, count_beliefs2(network))
        end
        push!(results, run_results)
    end
    return results
end
```

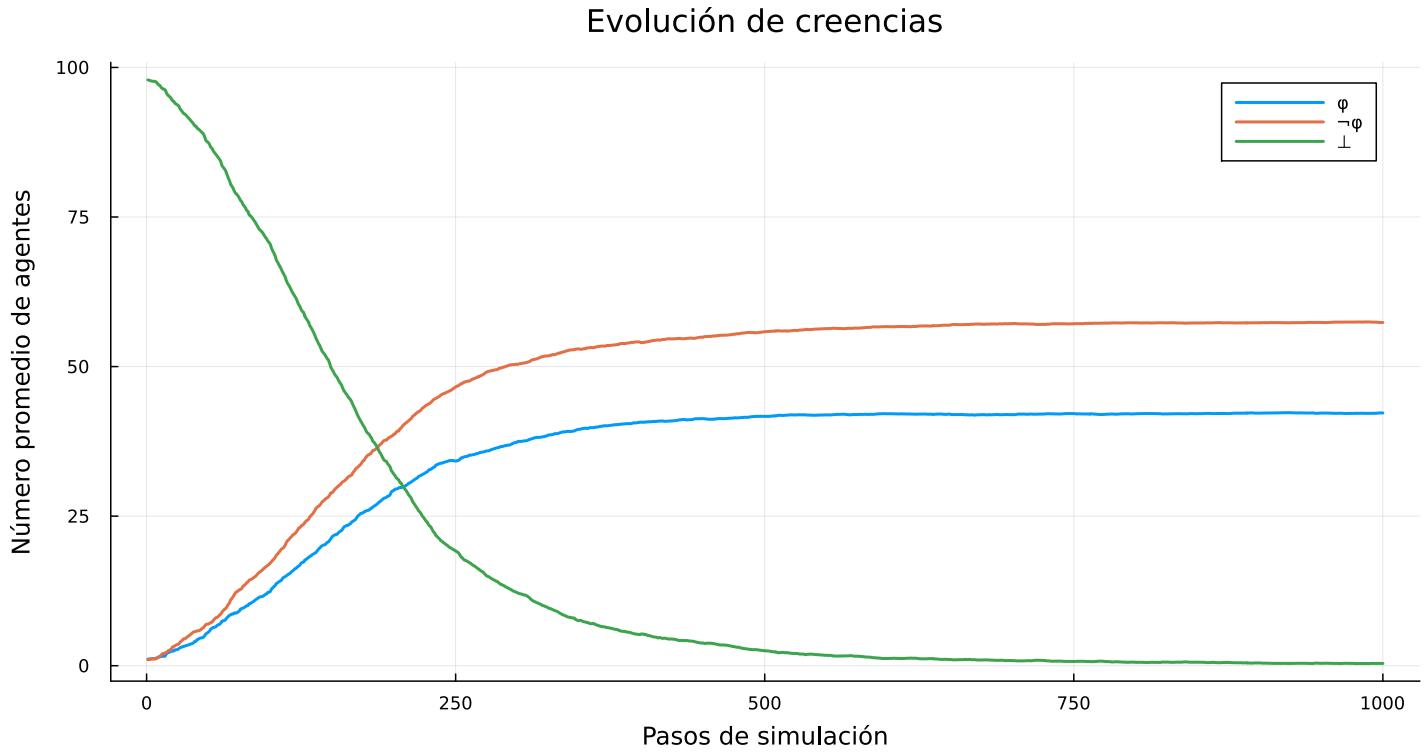
```
run_and_visualize_simulation2 (generic function with 1 method)
```

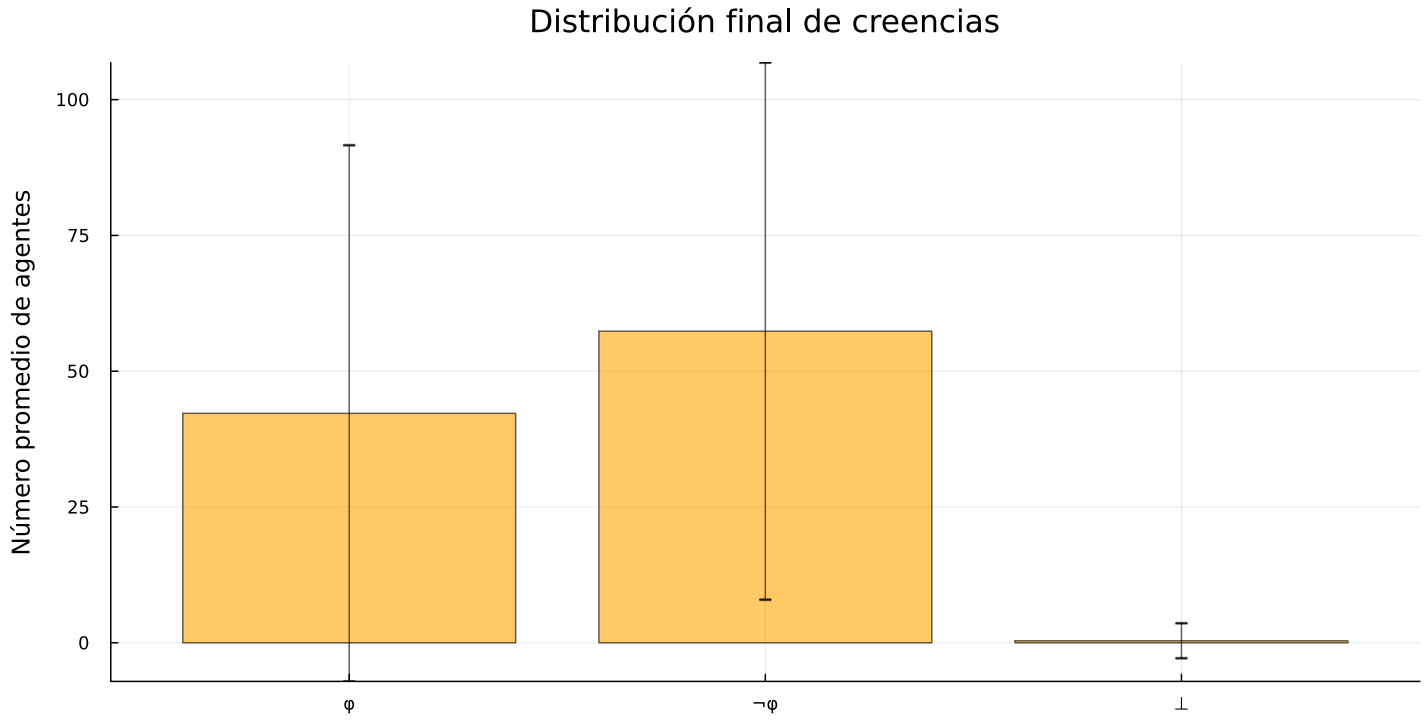
```
function run_and_visualize_simulation2(num_agents::Int, num_steps::Int,
λ_range::Tuple{Float64,Float64}, μ_range::Tuple{Float64,Float64})
    network = socialNetwork2(num_agents, λ_range, μ_range)
    set_seed_nodes2!(network, 1, 2)
    simulation_results = []
    for _ in 1:num_steps
        push!(simulation_results, [agent.state for agent in network.agents])
        simulate_step2!(network)
    end

    messages_sent = count_messages(simulation_results)
    println("Número de mensajes enviados: $messages_sent")

    fig, slider, update_viz = visualize_network_evolution(simulation_results,
network)
    return fig, slider, update_viz
end
```

```
sim_results2 =
```

```
▸ [[Dict(notphi::State = 1 ⇒ 1, phi::State = 0 ⇒ 1, ind::State = 2 ⇒ 98), Dict(notphi::
```

```
sim_results2 = run_simulation2(100, 1000, (0.3, 0.7), (0.3, 0.7), 100)
```






```julia
begin
    p1_ = plot_belief_evolution(sim_results2)
    p2_ = plot_final_distribution(sim_results2)
    Plots.plot(p1_, p2_, layout=(2,1), size=(950,1000))
end
```

```
begin
mean_messages2, messages_per_run2 = count_mean_messages_sent(sim_results2)
    println("Media de mensajes enviados: $(round(mean_messages2, digits=2))")
    println("Rango de mensajes enviados: $(minimum(messages_per_run2)) -
$(maximum(messages_per_run2))")
end
```

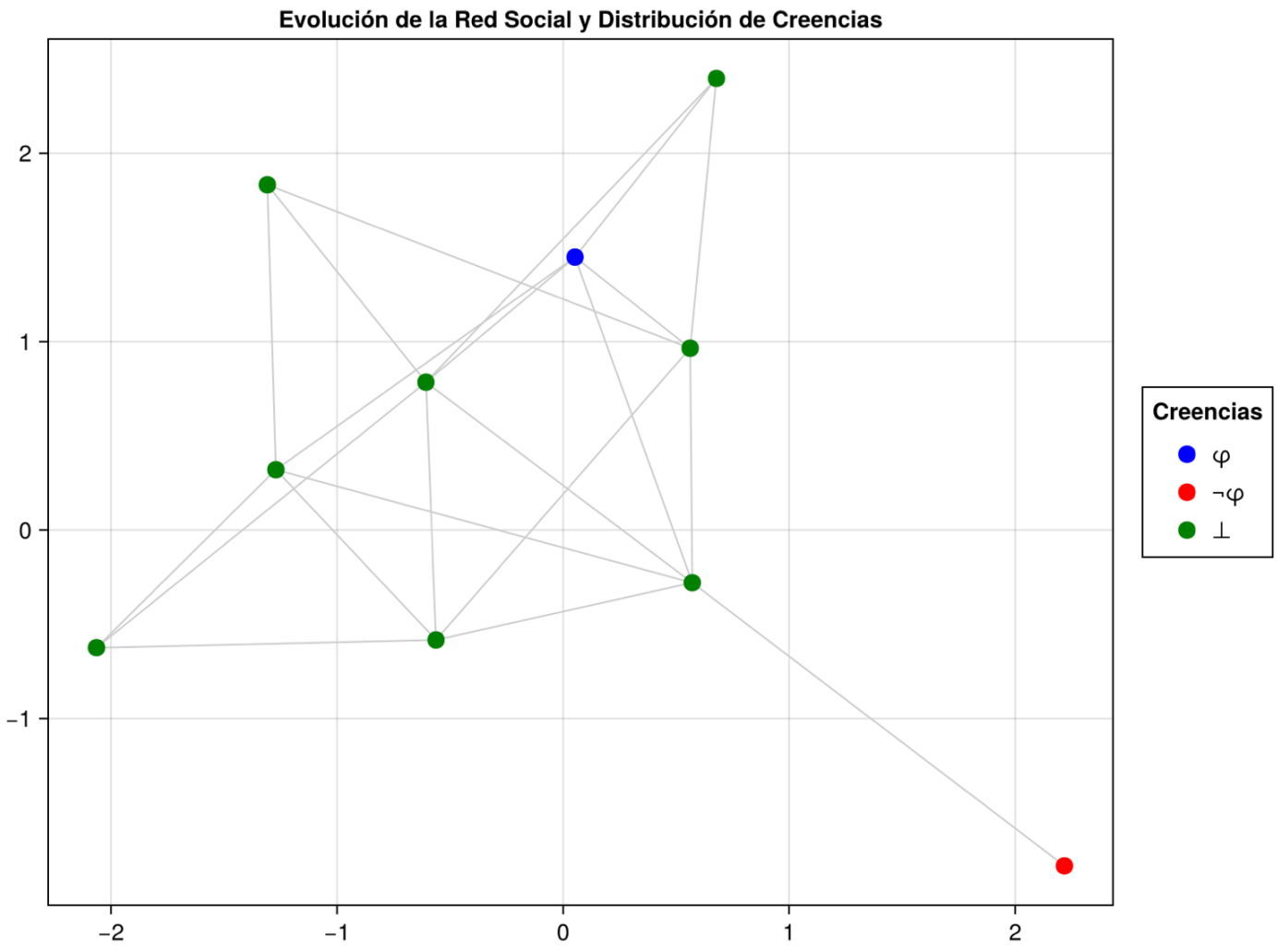
```
Media de mensajes enviados: 80.61
Rango de mensajes enviados: 42 - 420
```

Veamos una simulación: (distinta de las obtenidas en el resultado anterior)

```
fig2, slider2, update_viz2 = run_and_visualize_simulation2(10, 100, (0.3, 0.7),
(0.3, 0.7));
```

```
Número de mensajes enviados: 6
```

1



Modelo 3

Difusión de un mensaje

Consideraciones del modelo:

Este tercer modelo que se enfoca en la velocidad de propagación de un mensaje, considerando solo dos estados: informed (mensaje recibido) e uninformed (mensaje no recibido). Este modelo simplificado nos permitirá observar cómo se propaga un único mensaje a través de la red.

1. Solo dos estados: informed y uninformed.
2. Consideramos el parámetro `spread_probability` en la estructura Agent. Este parámetro ahora representa la probabilidad de que un agente decida compartir el mensaje con sus vecinos.
3. Implementamos la función `attempt_spread!` para reflejar este cambio. Ahora, cuando un agente informado intenta compartir el mensaje, usa su spread_probability para decidir si lo hace o no.
4. Actualizamos la función `simulate_step!` para que todos los agentes informados intenten compartir el mensaje con todos sus vecinos en cada paso.
5. Ajustamos las funciones de simulación y visualización para reflejar estos cambios.

Algunas cosas por ver:

- Cómo la distribución de las probabilidades de propagación afecta la velocidad general de propagación del mensaje.
- Si hay un umbral crítico de probabilidad de propagación por debajo del cual el mensaje no se propaga eficazmente.
- Cómo la estructura de la red interactúa con las probabilidades de propagación individuales.
- Si emergen "super-propagadores" (nodos que son particularmente efectivos en propagar el mensaje debido a su alta probabilidad de propagación y/o su posición en la red).

Implementación:

Definimos los posibles estados de cada individuo/agente, usamos un tipo enumerado `State` para representar los posibles estados de los agentes.

```
@enum State3 informed uninformed
```

Definimos la estructura que representa a un agente, esta cuenta con un id, el estado y la probabilidad que tiene el agente de esparcir el mensaje:

```
mutable struct Agent3
    id::Int
    state::State3
    spread_probability::Float64  # Probabilidad de decidir esparcir el mensaje
end
```

Definimos la estructura que representará la red social para este modelo:

```
struct SocialNetwork3
    graph::SimpleGraph
    agents::Vector{Agent3}
end
```

Definimos la siguiente función para simular el envío del mensaje desde un agente informado a uno no informado:

```
attempt_spread! (generic function with 1 method)
```

```
function attempt_spread!(sender::Agent3, receiver::Agent3)
    if sender.state == informed && receiver.state == uninformed
        if rand() < sender.spread_probability
            receiver.state = informed
        end
    end
end
```

La siguiente función nos permite inicializar la red social:

```
socialNetwork3 (generic function with 1 method)
```

```
function socialNetwork3(num_agents::Int,
spread_probability_range::Tuple{Float64,Float64})
    graph = barabasi_albert(num_agents, 3)
    agents = [Agent3(i, uninformed, rand(Uniform(spread_probability_range...))) for
i in 1:num_agents]
    SocialNetwork3(graph, agents)
end
```

La siguiente función sirve para establecer los nodos semilla:

```
set_seed_node3! (generic function with 1 method)
```

```
function set_seed_node3!(network::SocialNetwork3, seed::Int)
    network.agents[seed].state = informed
end
```

Para simular un paso de la cadena. En este todos los agentes informados intentan compartir el

mensaje con cada uno de sus vecinos.

```
simulate_step3! (generic function with 1 method)
```

```
function simulate_step3!(network::SocialNetwork3)
    for agent in network.agents
        if agent.state == informed
            for neighbor_id in neighbors(network.graph, agent.id)
                attempt_spread!(agent, network.agents[neighbor_id])
            end
        end
    end
end
```

Para contar el número de agentes informados:

```
count_informed (generic function with 1 method)
```

```
function count_informed(network::SocialNetwork3)
    count(agent -> agent.state == informed, network.agents)
end
```

Para correr la simulación:

```
run_simulation3 (generic function with 1 method)
```

```
function run_simulation3(num_agents::Int, num_steps::Int,
spread_probability_range::Tuple{Float64,Float64}, num_runs::Int)
    results = []
    for _ in 1:num_runs
        network = socialNetwork3(num_agents, spread_probability_range)
        set_seed_node3!(network, 1)
        run_results = [count_informed(network)]
        for _ in 1:num_steps
            simulate_step3!(network)
            push!(run_results, count_informed(network))
        end
        push!(results, run_results)
    end
    return results
end
```

A continuación, implementamos las funciones para visualizar los resultados:

```
plot_message_spread (generic function with 1 method)
```

```julia
function plot_message_spread(results)
    num_steps = length(results[1])
    num_runs = length(results)
    mean_informed = [mean([run[step] for run in results]) for step in 1:num_steps]
    std_informed = [std([run[step] for run in results]) for step in 1:num_steps]

    Plots.plot(0:(num_steps-1), mean_informed,
        ribbon=std_informed,
        fillalpha=0.3,
        title="Propagación del Mensaje",
        xlabel="Pasos de simulación",
        ylabel="Número de agentes informados",
        label="Media ± Desv. Estándar",
        legend=:bottomright)
end
```

```
visualize_network_evolution_ (generic function with 1 method)
```

```julia
function visualize_network_evolution_(simulation_results, network)
    layout = GraphMakie.spring(network.graph)
    fig = Figure(size = (800, 600))
    ax = GraphMakie.Axis(fig[1, 1])

    function update_colors(i)
        return [state == informed ? :red : :blue for state in simulation_results[i]]
    end

    node_colors = Observable(update_colors(1))
    graphplot!(ax, network.graph,
        layout = layout,
        node_color = node_colors,
        node_size = 15,
        edge_width = 1,
        edge_color = :gray80
    )

    legend_elements = [
        MarkerElement(color = :red, marker = :circle, markersize = 15),
        MarkerElement(color = :blue, marker = :circle, markersize = 15)
    ]
    legend_labels = ["Informado", "No informado"]
    Legend(fig[1, 2], legend_elements, legend_labels, "Estado")

    ax.title = "Propagación del Mensaje en la Red Social"

    slider = PlutoUI.Slider(1:length(simulation_results), default=1,
show_value=true)

    function update_viz(i)
        node_colors[] = update_colors(i)
    end

    return fig, slider, update_viz
end
```

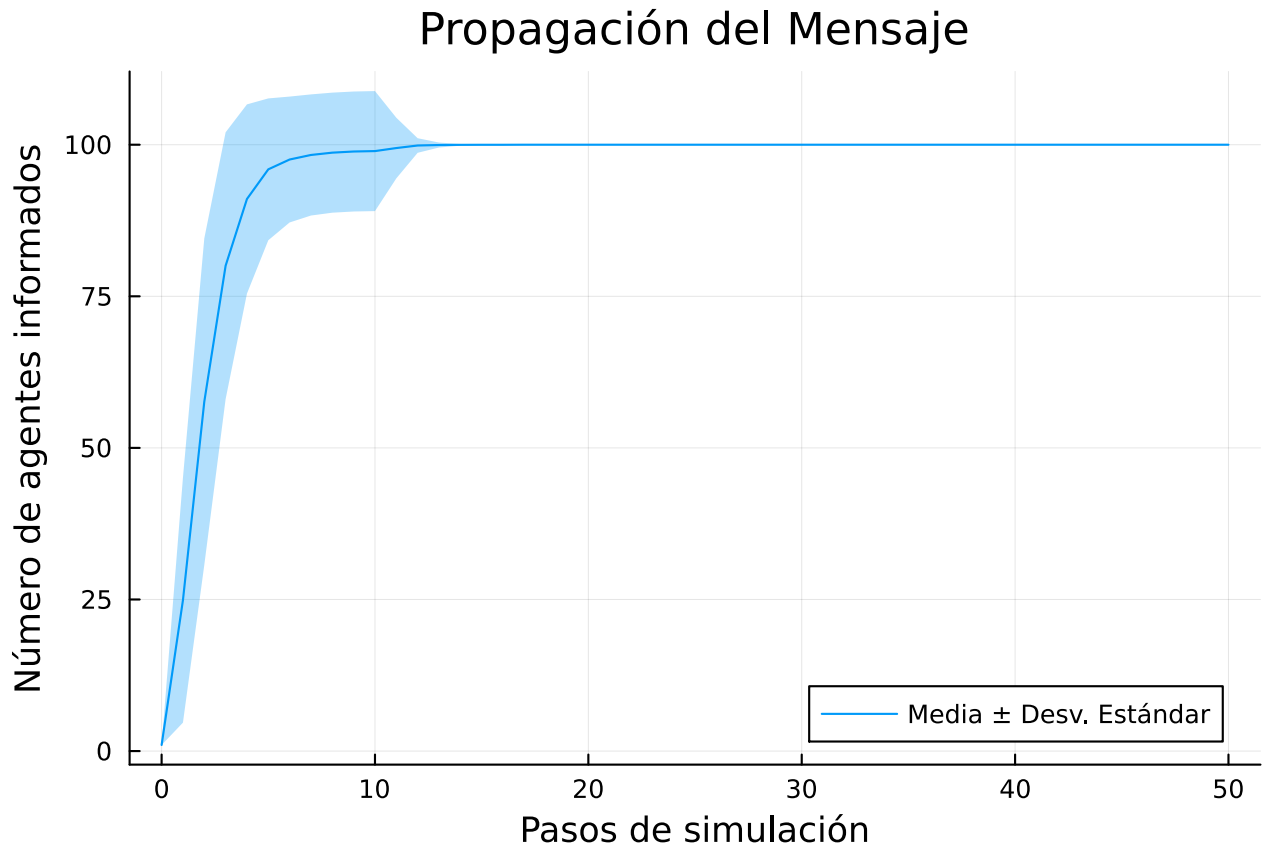
```
run_and_visualize_simulation_ (generic function with 1 method)
```

```
function run_and_visualize_simulation_(num_agents::Int, num_steps::Int,
spread_probability_range::Tuple{Float64,Float64})
    network = socialNetwork3(num_agents, spread_probability_range)
    set_seed_node3!(network, 1)
    simulation_results = [[agent.state for agent in network.agents]]
    for _ in 1:num_steps
        simulate_step3!(network)
        push!(simulation_results, [agent.state for agent in network.agents])
    end

    msf = count_messages(simulation_results)
    print("Mensajes enviados: $msf")

   fig, slider, update_viz = visualize_network_evolution_(simulation_results,
network)
    return fig, slider, update_viz
end
```

```
sim_results3 =
```

```
▸ [[1, 49, 88, 96, 96, 99, 100, 100, 100, ⋯ more ,100], [1, 39, 79, 91, 95, 100, 100, 100, 1
```

```
sim_results3 = run_simulation3(100, 50, (0.1, 0.5), 100)
```




```
plot_message_spread(sim_results3)
```

```
count_messages_in_simulation (generic function with 1 method)
```

```
function count_messages_in_simulation(simulation_results)
    messages = 0
    prev_state = simulation_results[1]
    for state in simulation_results[2:end]
        messages += count(prev_state .!= state)
        prev_state = state
    end
    return messages
end
```

```
expected_messages_sent_ (generic function with 1 method)
```

```
function expected_messages_sent_(results)
    num_runs = length(results)
    message_counts = zeros(Int, num_runs)

    for i in 1:num_runs
        message_counts[i] = count_messages_in_simulation(results[i])
    end

    expected_messages = mean(message_counts)
    std_dev_messages = std(message_counts)

    return expected_messages, minimum(message_counts), maximum(message_counts)
end
```

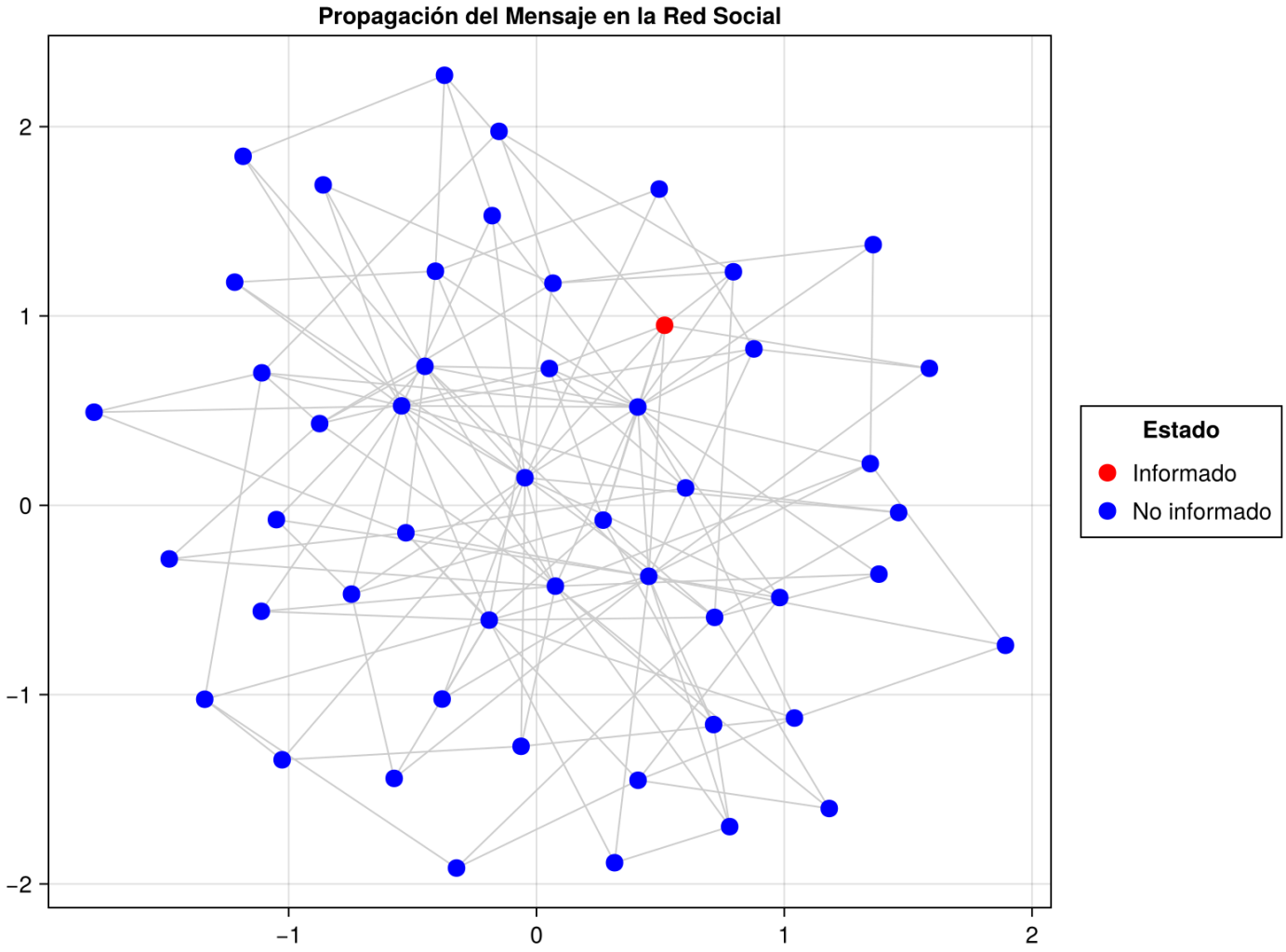
```
begin
mean_messages3, min3, max3 = expected_messages_sent_(sim_results3)
    println("Media de mensajes enviados: $(round(mean_messages3, digits=2))")
    println("Rango de mensajes enviados: $(min3) - $(max3)")
end
```

```
Media de mensajes enviados: 6.09
Rango de mensajes enviados: 4 - 10
```

Veamos una simulación: (distinta de las obtenidas en el resultado anterior)

```
fig3, slider3, update_viz3 = run_and_visualize_simulation_(50, 20, (0.1, 0.5));
```

```
Mensajes enviados: 6
```


Propagación del Mensaje en la Red Social
2
1
0
−1
−2
−1
0
1
2
Estado
Informado
No informado