Tarea 3

Integrantes:

- David Alejandro Alquichire Rincón - *dalquichire@unal.edu.co*
- Kevin Felipe Marroquín Olaya - *kfmarroquino@unal.edu.co*
- Tomas David Rodríguez Agudelo - *trodrigueza@unal.edu.co*

Librerias que utilizaremos:

```
using Pkg, PlutoUI, Graphs, GLMakie, GraphMakie, Makie.Colors,
GraphMakie.NetworkLayout, Random, CSV, DataFrames, Plots, Statistics
```

Primer Punto

a) Use un algoritmo MCMC, para generar 100 muestras *aproximadas* $(X_{10^3}, X_{10^4}, X_{10^5})$ del modelo de Ising, con inversos de temperatura $\beta = 0, 0.1, \cdots, 0.9, 1$

Solución:

Definamos el valor de k:

```
5
```

```
k
```

5

```
@bind k PlutoUI.Slider(5:20; default = 5, show_value = true) # Tamaño grilla
```

Para este notebook usaremos $k = 5$ debido a los tiempos de ejecución.

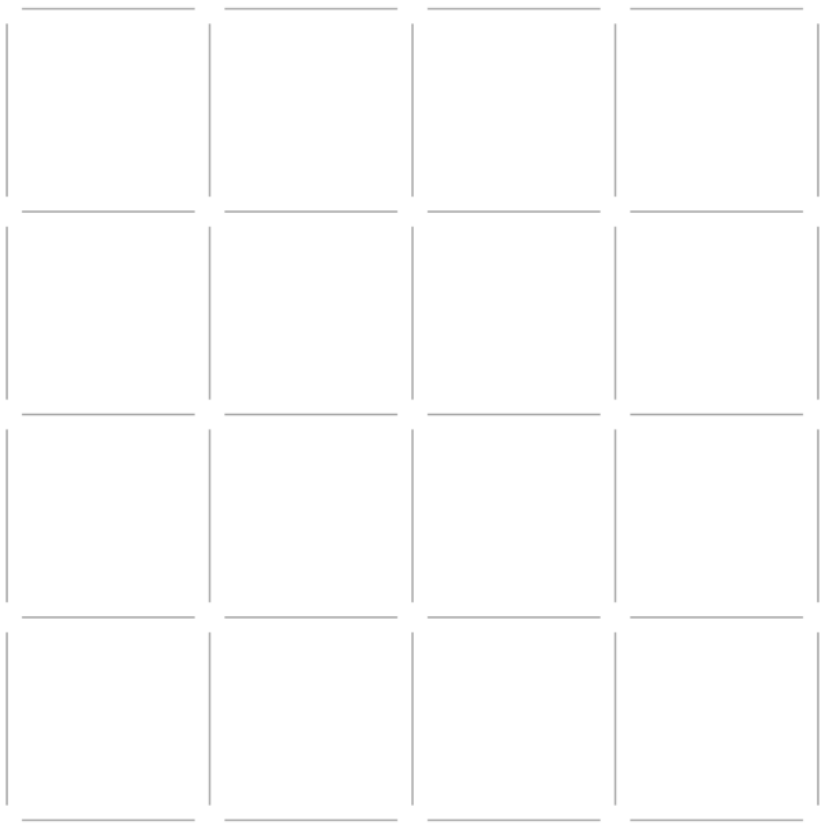
Inicialicemos el grafo reticular, haciendo uso de la librería *Graphs*:

▶ [(0.0, 0.0), (0.0, 0.1), (0.0, 0.2), (0.0, 0.3), (0.0, 0.4), (0.1, 0.0), (0.1, 0.1), (0.1,

```
begin
    g_1a = Graphs.grid([k, k]) # Grafo reticular kxk
    vcolor_1a = [:white for i in 1:nv(g_1a)];
    custom_layout_1a = [(i,j) for i = 0:0.1:( (k-1)*0.1 ) for j = 0:0.1:(
    (k-1)*0.1)];
end
```

▶ DataAspect()

```
begin
    f_1a, ax_1a, p_1a = graphplot(g_1a; layout= custom_layout_1a,
    node_color=vcolor_1a, edge_width=0.3, node_size=20);
    hidedecorations!(ax_1a)
    hidespines!(ax_1a)
    ax_1a.aspect = DataAspect()

    # sup ilabels = 1:400
end
```



Definamos el valor de β:

```
beta_1a = 0.01
```

```
beta_1a = 0.01
```

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

```
function neighbor_vertex(g, v, p)
    k_plus = 0
    k_minus = 0

    for i in edges(g)
        if src(i) == v
            if p.node_color[][dst(i)] == :white
                k_minus += 1
            elseif p.node_color[][dst(i)] == :black
                k_plus += 1
            end
        elseif dst(i) == v;
            if p.node_color[][src(i)] == :white
                k_minus += 1
            elseif p.node_color[][src(i)] == :black
                k_plus += 1
            end
        end
    end

    return k_plus, k_minus
end

```

Implementamos el Gibbs sampler:

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

```
# Construcción Gibbs Sampler

# -1 -> Blanco
# +1 -> Gris

function gibbs_sampler(g, p)
    global k
    v = rand( 1:k^2); # Seleccionar vértice aleatoriamente

    k_plus_v, k_minus_v = neighbor_vertex(g, v, p) # Obtener número de vecinos
                                                   # con -1 y con +1.

    u_n1 = rand()
    # println(u_n1)
    bound_upp = exp(2*beta_1a*(k_plus_v - k_minus_v))  / (exp(2*beta_1a*(k_plus_v - k_minus_v)) + 1)
    # println(bound_upp)

    if u_n1 < bound_upp
        p.node_color[][v] = :black
        p.node_color = p.node_color[]
    else
        p.node_color[][v] = :white
        p.node_color = p.node_color[]
    end

    # println(u_n1 < bound_upp)
end
```

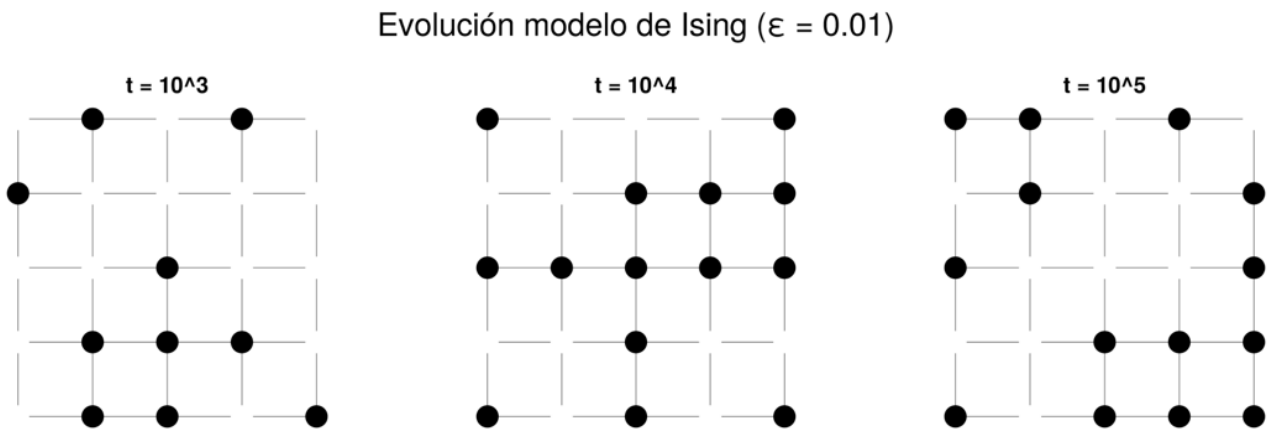
Corremos la cadena con un total de 10^5 pasos:

```
begin
    states = []
    Random.seed!(100) # Semilla para replicar resultados
    for i in range(1, 10^5)
        gibbs_sampler(g_1a, p_1a)
        if i in [10^3, 10^4, 10^5]
            push!(states, copy(p_1a.node_color[]))
        end
    end
end
```

Visualicemos X_{10^3}, X_{10^4} y X_{10^5}:

```
Makie.Label()
```

```
begin
    fig_ = Figure(size = (900, 300))
    axes_ = [Axis(fig_[1, i], aspect = DataAspect()) for i in 1:3]
    for (i, state) in enumerate(states)
        graphplot!(axes_[i], g_1a; layout=custom_layout_1a,
                node_color=state, edge_width=0.3, node_size=20)
        hidedecorations!(axes_[i])
        hidespines!(axes_[i])
        axes_[i].title = "t = 10^$(i+2)"
    end
    fig_[0, :] = Label(fig_, "Evolución modelo de Ising (ε = 0.01)", fontsize = 20)
end
```




```
fig_
```

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

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

Para generar las 100 muestras, consideremos $\beta = 0.1, 0.2, 0.3, \ldots, 0.8, 0.9$.

```
beta_values = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

```
beta_values = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

Inicializamos un diccionario para guardar las muestras:

```
samples = Dict()
```

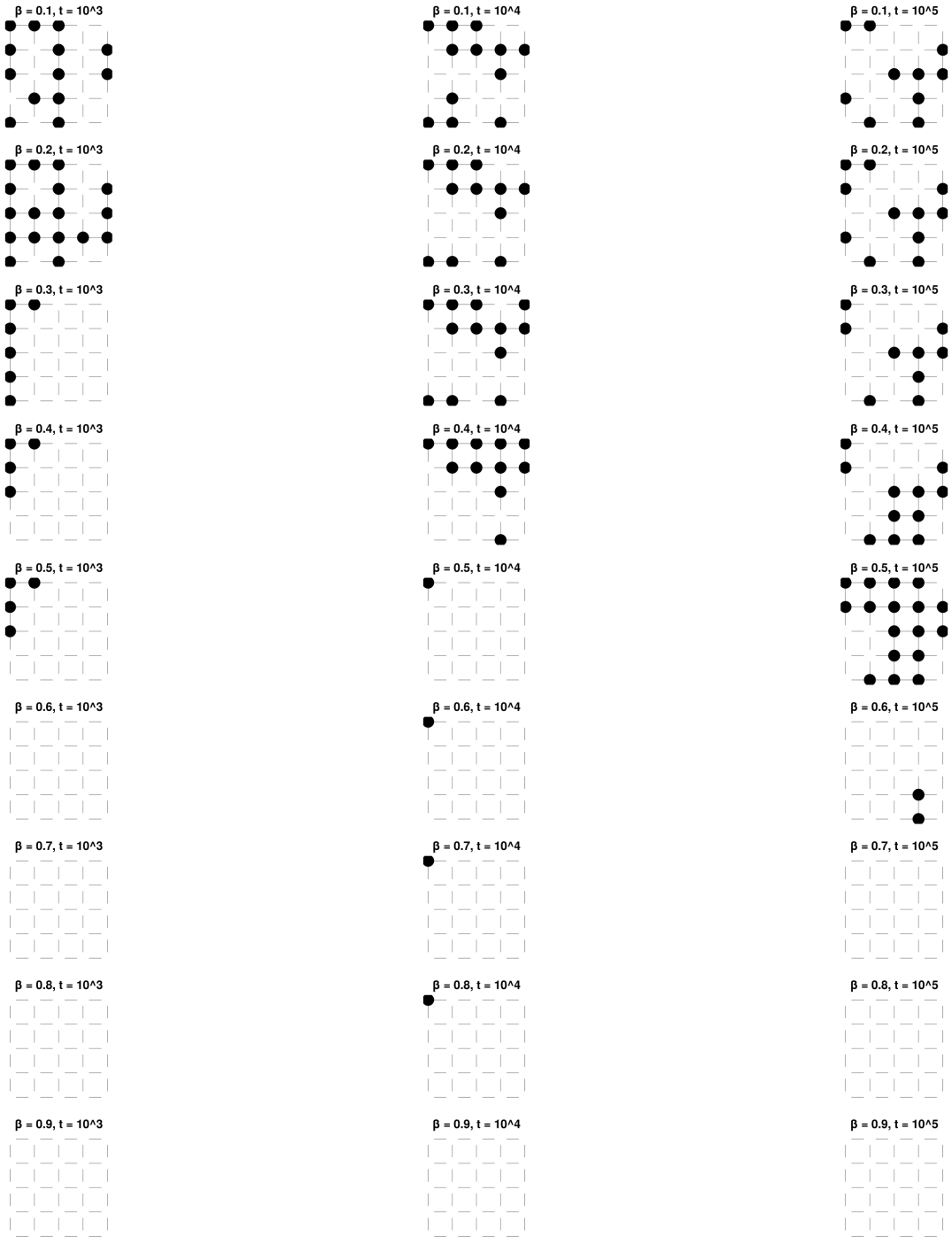
```
samples = Dict{Float64, Vector{Vector{Vector{Symbol}}}}()
```

Generamos las muestras por cada valor de β:

```
for beta in beta_values
    Random.seed!(100) # Semilla para replicar resultados
    samples[beta] = Vector{Vector{Vector{Symbol}}}()

    # Generar 100 muestras para cada beta
    for _ in 1:100
        node_colors = [:white for i in 1:nv(g_1a)]
        sample_states = Vector{Vector{Symbol}}()

        # Correr Gibbs sampler
        for i in 1:10^5
            node_colors = gibbs_sampler_(g_1a, node_colors, beta)

            # Guardar estados
            push!(sample_states, copy(node_colors))
        end

        # Guardar estados de la muestra
        push!(samples[beta], sample_states)
    end
end

```

Creamos una figura para mostrar los resultados, visualizaremos los estados X_{10^3}, X_{10^4} y X_{10^5} de la última muestra obtenida para cada β.

```
begin
    fig = Figure(size = (1500, 1500))
    axes = [Axis(fig[i, j], aspect = DataAspect()) for i in 1:9, j in 1:3]
    for (i, beta) in enumerate(beta_values)
        for (j, j1) in enumerate([10^3, 10^4, 10^5])
            state_index = j1  # 1 -> 10^3, 2 -> 10^4, 3 -> 10^5
            node_colors = samples[beta][end][state_index]

            graphplot!(axes[i, j], g_1a; layout=custom_layout_1a,
                   node_color=node_colors, edge_width=0.3, node_size=20)

            hidedecorations!(axes[i, j])
            hidespines!(axes[i, j])
            axes[i, j].title = "β = $beta, t = 10^$(j+2)"
        end
    end
end

```

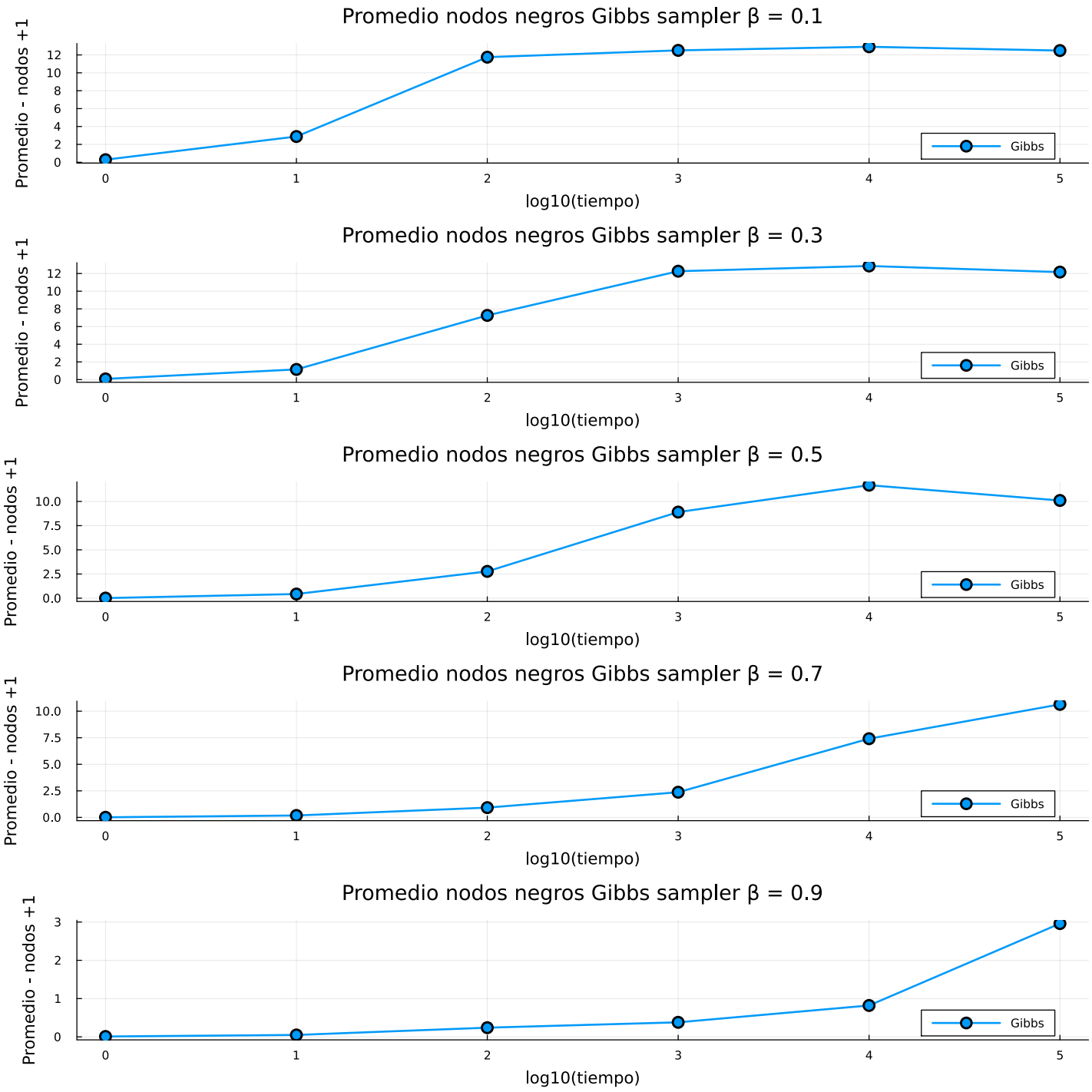



```
1 fig
```

Veamos algunas estadísticas:

```
for beta in beta_values
    for (j, j1) in enumerate([10^3, 10^4, 10^5])
        state_index = j1
        avg_black = mean([count(x -> x == :black, sample[state_index]) for sample in samples[beta]])
        println("Para β = $beta, t = 10^$(j+2), el promedio de nodos negros es: $avg_black")
    end
end
```

```
Para β = 0.1, t = 10^3, el promedio de nodos negros es: 12.51
Para β = 0.1, t = 10^4, el promedio de nodos negros es: 12.91
Para β = 0.1, t = 10^5, el promedio de nodos negros es: 12.49
Para β = 0.2, t = 10^3, el promedio de nodos negros es: 12.41
Para β = 0.2, t = 10^4, el promedio de nodos negros es: 12.71
Para β = 0.2, t = 10^5, el promedio de nodos negros es: 12.77
Para β = 0.3, t = 10^3, el promedio de nodos negros es: 12.26
Para β = 0.3, t = 10^4, el promedio de nodos negros es: 12.85
Para β = 0.3, t = 10^5, el promedio de nodos negros es: 12.16
Para β = 0.4, t = 10^3, el promedio de nodos negros es: 12.17
Para β = 0.4, t = 10^4, el promedio de nodos negros es: 12.59
Para β = 0.4, t = 10^5, el promedio de nodos negros es: 11.8
Para β = 0.5, t = 10^3, el promedio de nodos negros es: 8.9
Para β = 0.5, t = 10^4, el promedio de nodos negros es: 11.68
Para β = 0.5, t = 10^5, el promedio de nodos negros es: 10.1
Para β = 0.6, t = 10^3, el promedio de nodos negros es: 4.43
Para β = 0.6, t = 10^4, el promedio de nodos negros es: 11.6
Para β = 0.6, t = 10^5, el promedio de nodos negros es: 11.25
Para β = 0.7, t = 10^3, el promedio de nodos negros es: 2.37
Para β = 0.7, t = 10^4, el promedio de nodos negros es: 7.41
Para β = 0.7, t = 10^5, el promedio de nodos negros es: 10.64
Para β = 0.8, t = 10^3, el promedio de nodos negros es: 0.93
Para β = 0.8, t = 10^4, el promedio de nodos negros es: 2.1
Para β = 0.8, t = 10^5, el promedio de nodos negros es: 9.22
Para β = 0.9, t = 10^3, el promedio de nodos negros es: 0.38
Para β = 0.9, t = 10^4, el promedio de nodos negros es: 0.82
Para β = 0.9, t = 10^5, el promedio de nodos negros es: 2.96
```



Lo cual tiene mucho sentido pues para β mayor a β_c el valor de un spin tiende a dominar el retículo.

También mostraremos el código en caso de que quisiéramos implementar las cadeas utilizando Metropolis en lugar de Gibbs sampler:

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

```
function metropolis_step(g, node_colors, beta)
    v = rand(1:nv(g))
    current_color = node_colors[v]
    proposed_color = (current_color == :white) ? :black : :white

    # Calcular el cambio en la energía
    delta_E = 0
    for neighbor in neighbors(g, v)
        if node_colors[neighbor] == current_color
            delta_E += 2
        else
            delta_E -= 2
        end
    end

    # Calcular la probabilidad de aceptación
    acceptance_prob = min(1, exp(-beta * delta_E))

    # Decidir si aceptar el cambio
    if rand() < acceptance_prob
        node_colors[v] = proposed_color
    end

    return node_colors
end

```

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

```
function run_metropolis(g, beta, num_steps)
    node_colors = [:white for _ in 1:nv(g)]
    samples = Vector{Vector{Symbol}}()

    for step in 1:num_steps
        node_colors = metropolis_step(g, node_colors, beta)


            push!(samples, copy(node_colors))


    end

    return samples
end
```

Generaríamos las muestras así:

```
begin
    Random.seed!(100) # Semilla para replicar resultados
    samples_m = Dict{Float64, Vector{Vector{Vector{Symbol}}}}()

    for beta in beta_values
        samples_m[beta] = [run_metropolis(g_1a, beta, 10^5) for _ in 1:100]
    end
end
```

Así visualizaríamos los estados X_{10^3}, X_{10^4} y X_{10^5} de la última muestra obtenida para cada β:

```
begin
    fig1 = Figure(size = (1000, 1000))
    axes1 = [Axis(fig1[i, j], aspect = DataAspect()) for i in 1:9, j in 1:3]
    for (i, beta) in enumerate(beta_values)
        for (j, j1) in enumerate([10^3, 10^4, 10^5])
            state_index1 = j1  # 1 -> 10^3, 2 -> 10^4, 3 -> 10^5
            node_colors1 = samples_m[beta][end][state_index1]

            graphplot!(axes1[i, j], g_1a; layout=custom_layout_1a,
                   node_color=node_colors1, edge_width=0.3, node_size=20)

            hidedecorations!(axes1[i, j])
            hidespines!(axes1[i, j])
            axes1[i, j].title = "β = $beta, t = 10^$(j+2)"
        end
    end
end
```

Para ver algunas estadísticas:

```
for beta in beta_values
    for (j, j1) in enumerate([10^3, 10^4, 10^5])
        state_index = j1
        avg_black = mean([count(x -> x == :black, sample[state_index]) for sample
in samples_m[beta]])
        println("Para β = $beta, t = 10^$(j+2), el promedio de nodos negros es:
$avg_black")
    end
end
```

b) Use el algoritmo de Propp-Wilson para obtener 100 muestras exactas del modelo de Ising, tomando los mismos valores para β que en el inciso **a)**

Solución:

Modificamos brevemente la función `gibbs_sampler` para conservar los mismo números aleatorios que se van utilizado (los guardaremos en `random_dict`. Los guardaremos en las cadenas que generaremos usando el algoritmo Propp-Wilson:

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

```
begin
    global random_dict = Dict{Int, Float64}()

    function gibbs_sampler_pw(g, p, time)
        v = rand( 1:k^2);

        k_plus_v, k_minus_v = neighbor_vertex(g, v, p)

        if !haskey(random_dict, time)
            random_dict[time] = rand()
        end

        u_n1 = random_dict[time]

        bound_upp = exp(2*beta_1a*(k_plus_v - k_minus_v))  / (exp(2*beta_1a*
        (k_plus_v - k_minus_v)) + 1)

        if u_n1 < bound_upp
            p.node_color[][v] = :black
            p.node_color = p.node_color[]
        else
            p.node_color[][v] = :white
            p.node_color = p.node_color[]
        end

    end
end
```

Tomamos el mismo orden parcial que vimos en clase, y tomamos los retículos donde todos los vértices son blancos (O $-\mathbf{1}$) y donde todos los vértices son negros (O $+\mathbf{1}$).

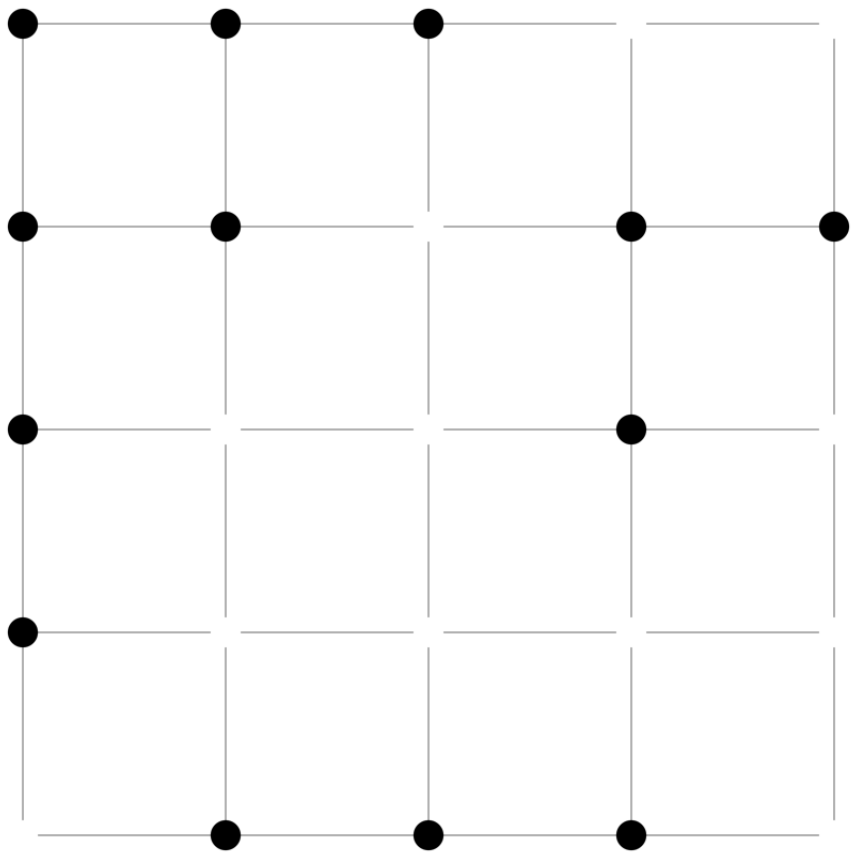
```
Propp_Wilson_iter (generic function with 1 method)
```

```
# Ejecutar Propp-Wilson con Sandwiching

function Propp_Wilson_iter(n) # time of starting: 2^n
    g_min = Graphs.grid([k, k])

    vcolor_min = [:white for i in 1:nv(g_min)]
    custom_layout_min = [(i,j) for i = 0:0.1:( (k-1)*0.1 ) for j = 0:0.1:( (k-1)*0.1 ) ]
    f_min, ax_min, p_min = graphplot(g_min; layout= custom_layout_min, node_color=vcolor_min, edge_width=0.3, node_size=20);

    hidedecorations!(ax_min)
    hidespines!(ax_min)
    ax_min.aspect = DataAspect()


    g_max = Graphs.grid([k, k])

    vcolor_max = [:black for i in 1:nv(g_max)]
    custom_layout_max =[(i,j) for i = 0:0.1:( (k-1)*0.1 ) for j = 0:0.1:( (k-1)*0.1 ) ]
    f_max, ax_max, p_max = graphplot(g_max; layout= custom_layout_max, node_color=vcolor_max, edge_width=0.3, node_size=20);

    hidedecorations!(ax_max)
    hidespines!(ax_max)

    ax_max.aspect = DataAspect()

    for i in -2^n:0
        gibbs_sampler_pw(g_min, p_min, i)
        gibbs_sampler_pw(g_max, p_max, i)
    end

    return (f_max, ax_max, p_max), (f_min, ax_min, p_min)

end

```

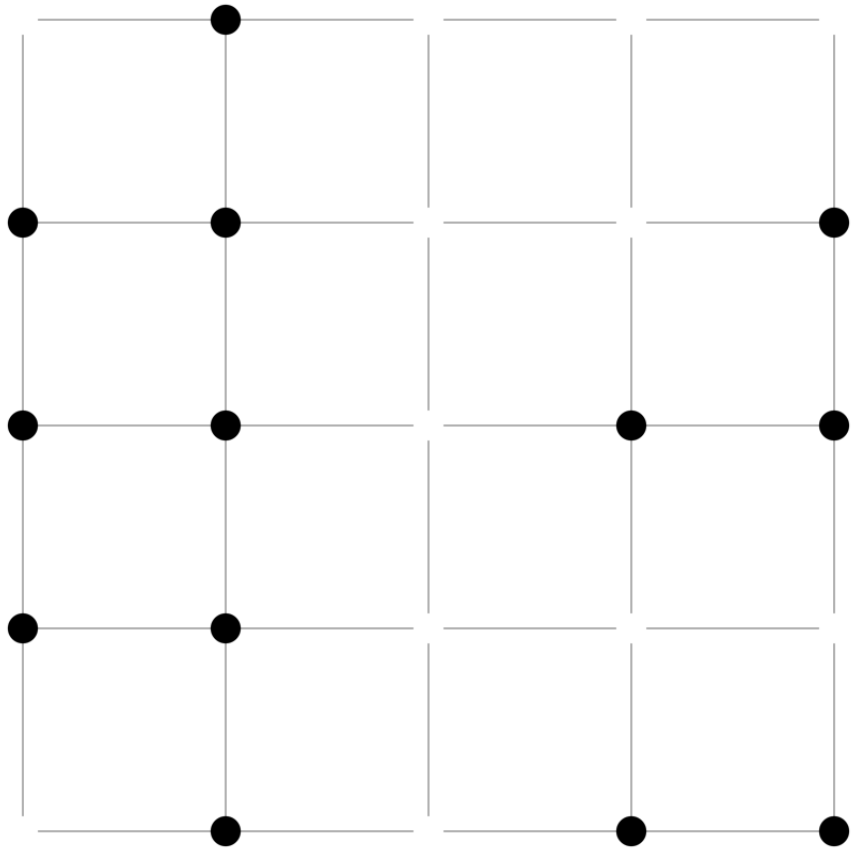
A continuación corremos el algoritmo. Una modificación notable es que las cadenas coalescerán en el momento en el que el número de nodos negros sea el mismo o difieran en uno a lo sumo, esto debido a que el tiempo de coalescencia para que las cadenas lleguen a exactamente el mismo estado parece ser muy grande. En términos prácticos conseguimos los resultados deseados.

```
begin
    m_ = 0
    mi_ = 0
    n = 1;
    while true

        Graph_max, Graph_min = Propp_Wilson_iter(n)

        # println(Graph_max[3].node_color)
        # println(Graph_min[3].node_color)

        n += 1

        if abs( count( i->(i == :black) , Graph_max[3].node_color[]) - count(  i->(i == :black), Graph_min[3].node_color[])) <= 1
            m_ = Graph_max[1]
            mi_ = Graph_min[1]
            break
        end
    end
end
```

A continuación podemos observar los estados finales de las dos cadenas corridas (max-min).



```
m_
```



```
mi_
```

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

```
function gibbs_sampler_pw_(g, node_colors, beta, time)
    v = rand(1:nv(g))
    k_plus, k_minus = neighbor_vertex_(g, v, node_colors)

    if !haskey(random_dict, time)
        random_dict[time] = rand()
    end
    u_n1 = random_dict[time]

    bound_upp = exp(2*beta*(k_plus - k_minus)) / (exp(2*beta*(k_plus - k_minus)) +
1)

    if u_n1 < bound_upp
        node_colors[v] = :black
    else
        node_colors[v] = :white
    end

    return node_colors
end
```

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

```
function Propp_Wilson_iter_(n, beta)
    g_min = Graphs.grid([k, k])
    node_colors_min = [:white for i in 1:nv(g_min)]

    g_max = Graphs.grid([k, k])
    node_colors_max = [:black for i in 1:nv(g_max)]

    for i in -2^n:0
        node_colors_min = gibbs_sampler_pw_(g_min, node_colors_min, beta, i)
        node_colors_max = gibbs_sampler_pw_(g_max, node_colors_max, beta, i)
    end

    return node_colors_max, node_colors_min
end
```

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

```
function run_propp_wilson_(beta, cv)
    n = 1
    while true
        node_colors_max, node_colors_min = Propp_Wilson_iter_(n, beta)

        n += 1

        if abs(count(i -> i == :black, node_colors_max) - count(i -> i == :black, node_colors_min)) <= 1 || (n > 10 && cv == true)
            return node_colors_max, 2^n
        end
    end
end
```

Generamos las muestras utilizadas utilizando los valores β del punto anterior ($\mathbf{0.1, 0.4, 0.9}$).

```
Enter cell code...
```

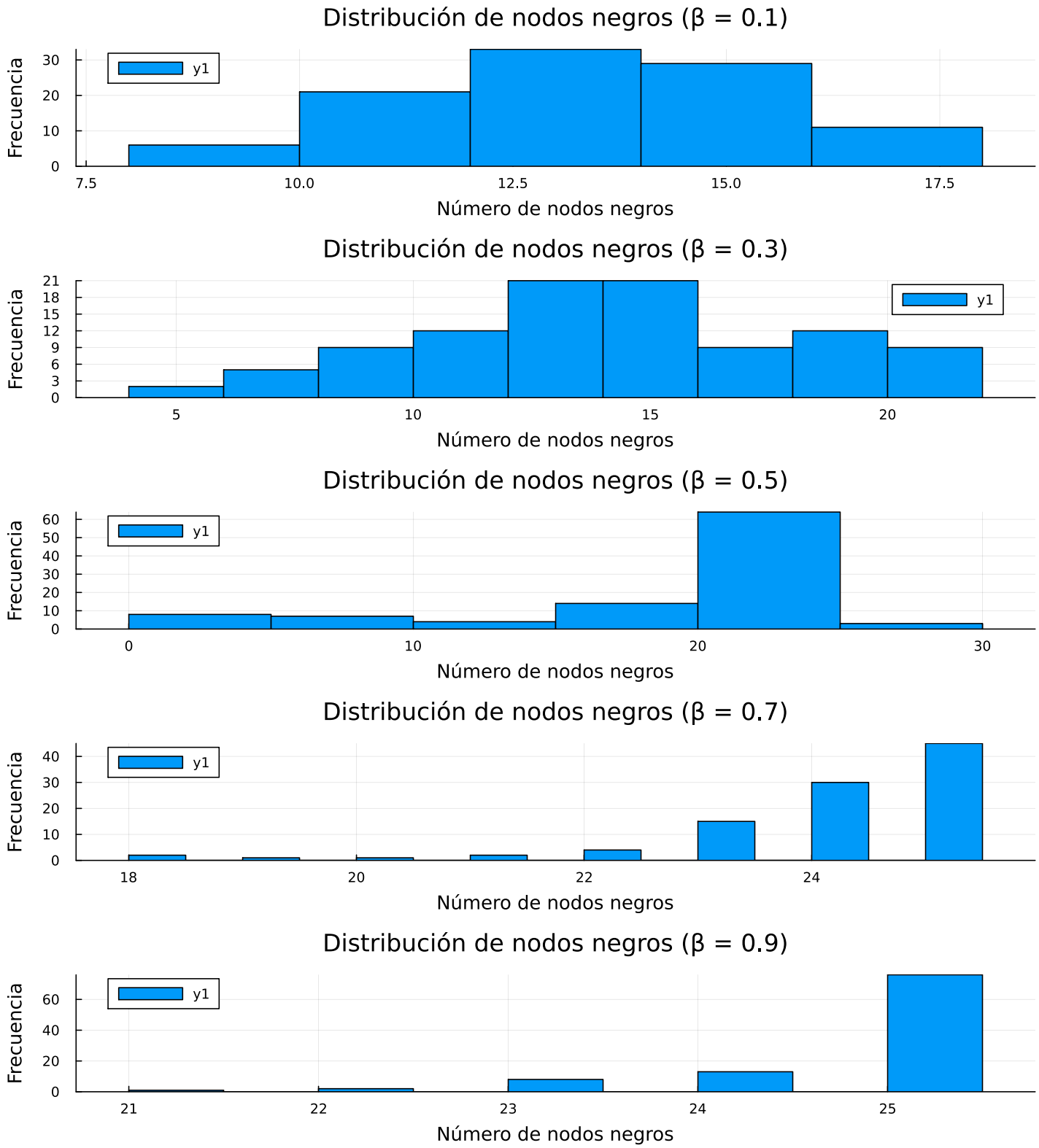
```
▸ Dict()
```

```
begin
    samples_pw = Dict{Float64, Vector{Vector{Symbol}}}()
    coalescence_times = Dict{Float64, Vector{Int}}()
end
```

```julia
begin
    for beta in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
        samples_pw[beta] = Vector{Vector{Symbol}}()
        coalescence_times[beta] = Vector{Int}()
        for _ in 1:100
            node_colors_, coalesence_time = run_propp_wilson_(beta, true)
            push!(samples_pw[beta], node_colors_)
            push!(coalescence_times[beta], coalesence_time)
        end
    end
end
```

```julia
begin
igs = []
for beta in [0.1, 0.3, 0.5, 0.7, 0.9]
    avg_black_ = mean([count(x -> x == :black, sample) for sample in samples_pw[beta]])
    println("Para β = $beta, el número promedio de nodos negros es: $avg_black_")

    push!(igs, histogram([count(x -> x == :black, sample) for sample in samples_pw[beta]],
                title="Distribución de nodos negros (β = $beta)",
                xlabel="Número de nodos negros", ylabel="Frecuencia"))
    savefig("propp_wilson_histogram_beta_$beta.png")
end
end
```

```
Para β = 0.1, el número promedio de nodos negros es: 12.82
Para β = 0.3, el número promedio de nodos negros es: 13.71
Para β = 0.5, el número promedio de nodos negros es: 18.72
Para β = 0.7, el número promedio de nodos negros es: 23.95
Para β = 0.9, el número promedio de nodos negros es: 24.61
```




```
Plots.plot(igs..., layout=plot_layout, size=(900, 1000))
```

Allí podemos visualizar el cambio de fase a partir del valor crítico β_c.

c) Estime con los incisos anteriores, $\mathbb{E}[M(\eta)]$, donde $M(\eta) = \frac{1}{|V_k|}\sum_{x\in V_k}\eta_x$

Reporte:

- Tiempo de coalescencia en P-W.
- Comparación de las estimativas obtenidas en **c)**.

Solución:

- Comparación de las estimativas:

Definimos la función para calcular la magnetización de una configuración dada:

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

```
function calculate_M(config)
    return abs(mean(config .== :black) - mean(config .== :white))
end
```

Definimos la función para calcular el promedio de la magnetización en un conjunto dado de muestras:

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

```
function promedio_M(samples)
    return mean(calculate_M.(samples))
end
```

Recordemos que guardamos las muestras obtenidas mediante MCMC (caso Gibbs sampler) y Prop-Wilson en los diccionarios `samples` y `samples_pw`, respectivamente.

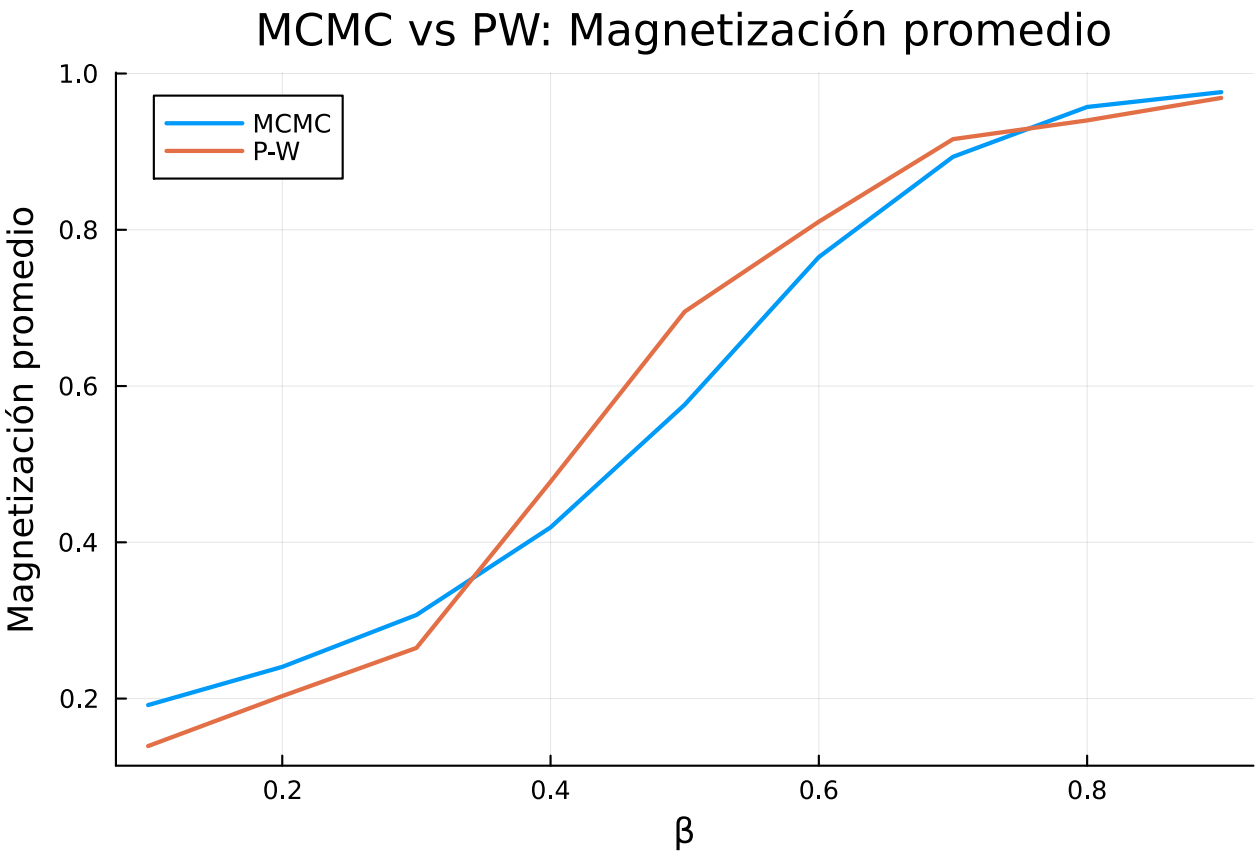
```
avg_mag_mcmc =
▸ Dict(0.4 => 0.419073, 0.7 => 0.893418, 0.3 => 0.307002, 0.5 => 0.576169, 0.2 => 0.240576
```

```
avg_mag_mcmc = Dict(beta => promedio_M(samples[beta][end]) for beta in beta_values)
```

```
avg_mag_pw =
▸ Dict(0.4 => 0.4776, 0.7 => 0.916, 0.3 => 0.2648, 0.5 => 0.6952, 0.2 => 0.2032, 0.9 => 0.
```

```
avg_mag_pw = Dict(beta => promedio_M(samples_pw[beta]) for beta in beta_values)
```

Visualización tomando el valor absoluto de las magnetizaciones:




```
begin
    p_avg = Plots.plot(beta_values, [avg_mag_mcmc[beta] for beta in beta_values],
    label = "MCMC", maker = :circle, linewidth=2, size = (600, 400))

    Plots.plot!(beta_values, [avg_mag_pw[beta] for beta in beta_values], label = "P-
    W", maker = :square, linewidth=2)

    Plots.xlabel!(p_avg, "β")
    Plots.ylabel!(p_avg, "Magnetización promedio")
    Plots.title!(p_avg, "MCMC vs PW: Magnetización promedio")
end
```

Para los tiempos de coalescencia, tuvimos que parar antes las cadenas debido al tiempo que se tomaban en converger, podemos estimar debido a los tiempos que en su momento duraba el problema ejecutándose que al menos para $\beta > \beta_c$ los tiempos son mayores a $\mathbf{2^{25}}$. En cualquier caso podemos observar en la gráfica anterior que se obtienen los resultados que se esperarían.

Segundo Punto

Una hormiga ha sido desajolada de su colonia ubicada en el punto $(0, 0)$ de la parcela $[0, 1] \times [0, 1]$. Decide entonces la hormiga visitar todas las otras 75 colonias de su parcela sin repetir.

Reporte:

- Esquema de enfriamento usado.
- Distancia mínima obtenida.
- Mapa generado para la hormiga.

Primero, leemos el archivo csv con las coordenadas:

locations_ =

	Coordenada X	Coordenada Y
1	"0"	"0"
2	"0,640194804"	"0,8766190504"
3	"0,2934596053"	"0,3496503006"
4	"0,7900258247"	"0,9329749235"
5	"0,8887102598"	"0,240910138"
6	"0,9651210325"	"0,8005837321"
7	"0,6899746576"	"0,8337593556"
8	"0,240618119"	"0,2446297373"
9	"0,5374522203"	"0,1268334208"
10	"0,02671012213"	"0,299350216"
⋮	more	
76	"0,7520991723"	"0,07265280872"

```
locations_ = CSV.read("ant_points.csv", DataFrame)
```

y las guardamos usando el tipo de dato `Float64` (flotantes):

▸ [0.0, 0.876619, 0.34965, 0.932975, 0.24091, 0.800584, 0.833759, 0.24463, 0.126833, 0.2993

```
begin
    locations__ = replace.(locations_, ","=>".");

    x_points = parse.(Float64, locations__."Coordenada X");
    y_points = parse.(Float64, locations__."Coordenada Y");
end
```

▸ [0.0, 0.640195, 0.29346, 0.790026, 0.88871, 0.965121, 0.689975, 0.240618, 0.537452, 0.026

```
x_points
```

```
▸ [0.0, 0.876619, 0.34965, 0.932975, 0.24091, 0.800584, 0.833759, 0.24463, 0.126833, 0.2993
```

```
y_points
```

a) Use "simulated annealing" para ayudarle a la hormiga a encontrar el camino más corto que recorra todas las parcelas.

Solución:

Creamos la función `dist_path` para calcular la distancia entre cada par de colonias de acuerdo a sus coordenadas

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

```
# function: distance of path.
function dist_path(x::Vector, y::Vector)

    dist = 0.0
    n = length(x_points)

    for i in 1:(n-1)
       dist += sqrt(  (y[i+1] - y[i])^2 + ( x[i+1] - x[i] )^2 )
    end

    return dist

end
```

Ahora con la siguiente función `new_neighboor` se le asigna un nuevo vecino a cada colonia a partir de la lista original

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

```
# build a neighboor
function new_neighboor(x::Vector, y::Vector)

    i = rand(2:76)
    j = rand(i:76)

    new_x = copy(x)
    new_y = copy(y)

    slice_reverse_x = reverse( new_x[i:j] )
    slice_reverse_y = reverse( new_y[i:j] )

    new_x[i:j] = slice_reverse_x
    new_y[i:j] = slice_reverse_y

    return new_x, new_y

end
```

Ahora creamos la función `simulated_annealing_1` usando el algoritmo de simulated annealing para optimizar la distancia entre cada par de colonias la cual convergera a una solución final a medida que la temperatura T baja.

```
simulated_annealing_1 (generic function with 1 method)
```

```
function simulated_annealing_1(x::Vector, y::Vector)

    # Simulate Metropolis Chain
    # cooling schedule (1, 1/10, 1/100, 1/1000, ...) and (1, 10, 100, 1000, ...)

    curr_points = Dict( "x" => x, "y" => y )

    #  final_neighboor = (p2_x, p2_y)

    n = 1

    # execute markov chain with transition matrix given in class

    while n < 7

        for i in 1:10^n # max 10^6

            T = 1/(10^n)

            p2_x, p2_y = new_neighboor(curr_points["x"], curr_points["y"])

            e = dist_path(curr_points["x"], curr_points["y"])
            e_ = dist_path(p2_x, p2_y)

            u = rand()

            if u < min( 1, exp( (e - e_)/T )  )
                curr_points = Dict( "x" => p2_x, "y" => p2_y )
            #else
            #   final_neighboor = (curr_points["x"], curr_points["y"])
            end

        end
        n += 1
    end
    return curr_points
end
```

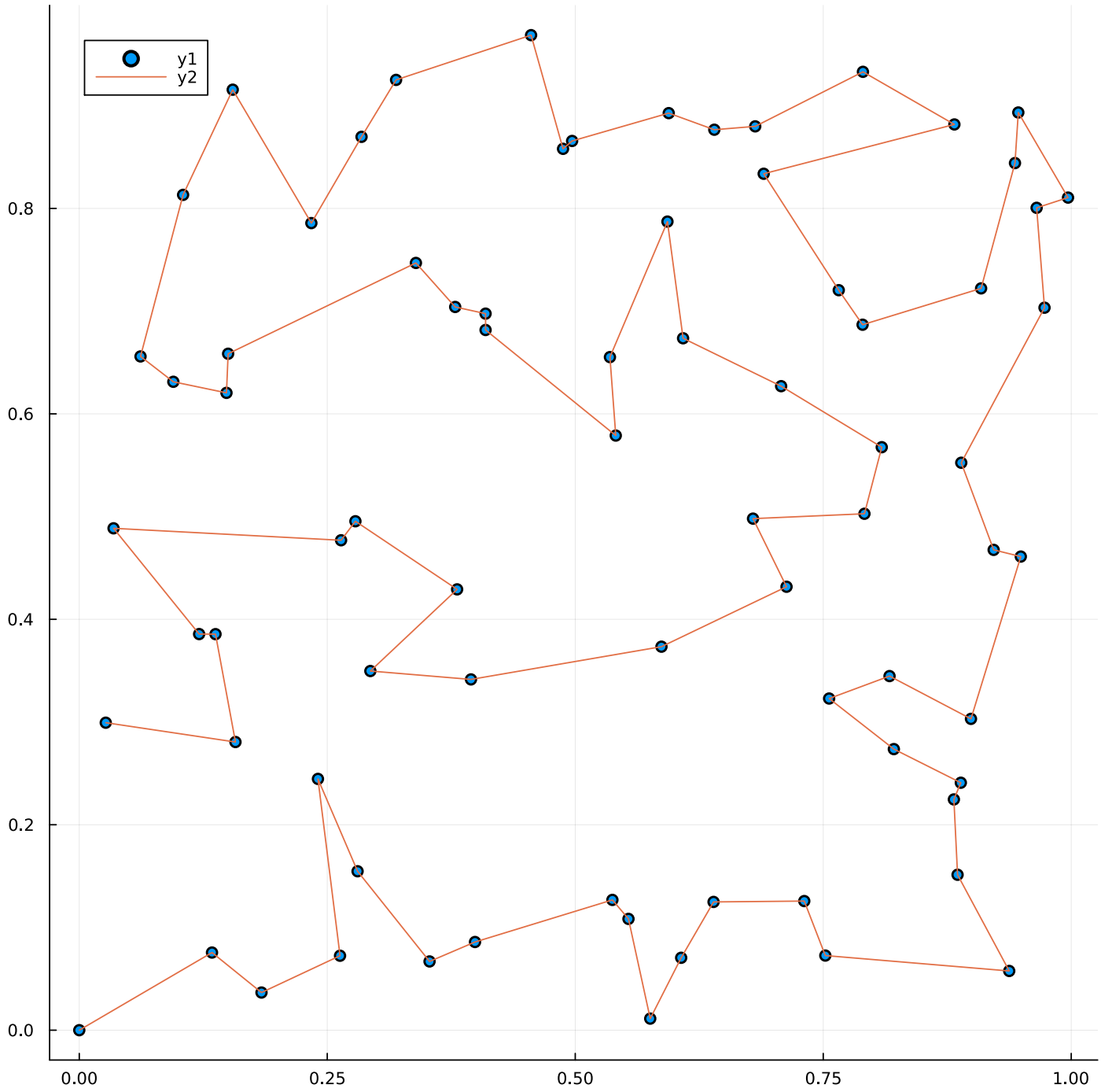
La función devolverá una solución optimizada (en forma de un diccionario con las nuevas coordenadas x, y), la cual se almacenará en `final_neighboor`.

```
final_neighboor =
```

```
▸ Dict("x" ⇒ [0.0, 0.133813, 0.183704, 0.262884, 0.240618, 0.280562, 0.353121, 0.398956, (
```

```
#gr(size = (800,800))
# scatter(x, y)
final_neighboor = simulated_annealing_1(x_points, y_points)
```

Ahora graficamos las colonias y el recorrido optimizado:




```
begin
    gr(size = (800, 800))
    Plots.scatter(x_points, y_points)
    Plots.plot!(final_neighboor["x"], final_neighboor["y"])
end
```

Obteniendo la siguiente distancia:

```
7.376463295493744
```

```
dist_path(final_neighboor["x"], final_neighboor["y"])
```

b) Repita el item a) si se sabe que la hormiga retornará a su colonia original, después de haber

recorrido todas las otras colonias.

Solución:

```
x_points_b =
```

▸ [0.0, 0.640195, 0.29346, 0.790026, 0.88871, 0.965121, 0.689975, 0.240618, 0.537452, 0.026

```
1 x_points_b = copy(x_points)
```

```
y_points_b =
```

▸ [0.0, 0.876619, 0.34965, 0.932975, 0.24091, 0.800584, 0.833759, 0.24463, 0.126833, 0.2993

```
1 y_points_b = copy(y_points)
```

Para este punto incluimos el punto de partida para obtener la ruta más corta en la cual la hormiga recorre todos los puntos y vuelve al punto de origen.

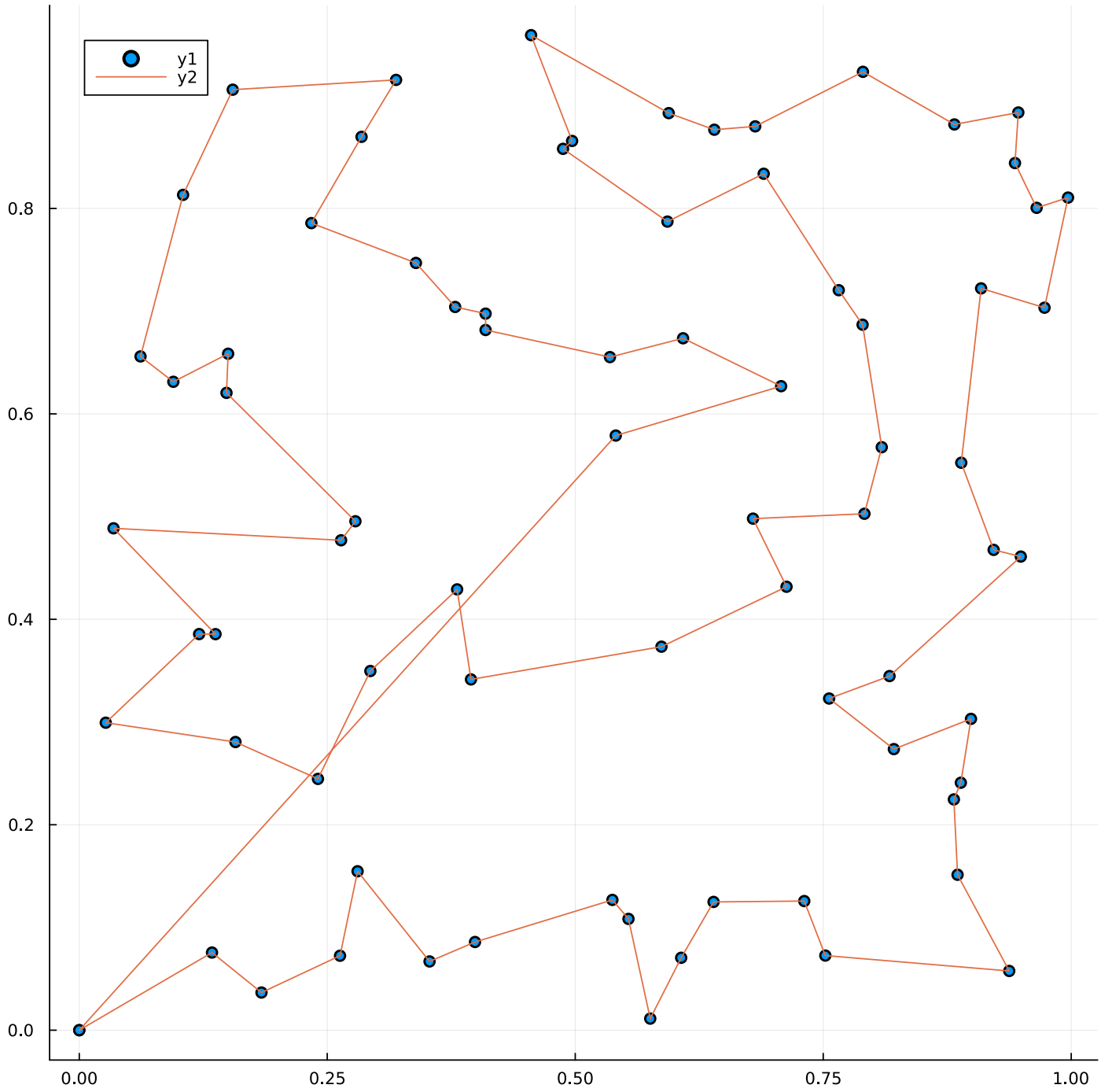
▸ [0.0, 0.876619, 0.34965, 0.932975, 0.24091, 0.800584, 0.833759, 0.24463, 0.126833, 0.2993

```
1 begin
2     push!(x_points_b, 0)
3     push!(y_points_b, 0)
4 end
```

```
final_neighboor_b =
```

▸ Dict("x" ⇒ [0.0, 0.133813, 0.183704, 0.262884, 0.280562, 0.353121, 0.398956, 0.537452, 0

```
1 final_neighboor_b = simulated_annealing_1(x_points_b, y_points_b)
```




```
begin
    gr(size = (800, 800))
    Plots.scatter(x_points_b, y_points_b)
    Plots.plot!(final_neighboor_b["x"], final_neighboor_b["y"])
end
```

Obteniendo la siguiente distancia:

```
7.054698150065478
```

```
dist_path(final_neighboor_b["x"], final_neighboor_b["y"])

# Notar que para n = 7, ya no da un camino que minimice la distancia.
```