

# Tarea 3

## Integrantes:

- David Alejandro Alquichire Rincón - [dalquichire@unal.edu.co](mailto:dalquichire@unal.edu.co)
- Kevin Felipe Marroquín Olaya - [kfmarroquino@unal.edu.co](mailto:kfmarroquino@unal.edu.co)
- Tomas David Rodríguez Agudelo - [trodrigueza@unal.edu.co](mailto:trodrigueza@unal.edu.co)

Librerías que utilizaremos:

```
1 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  $\beta = 0, 0.1, \dots, 0.9, 1$

## Solución:

Definamos el valor de  $k$ :

5

```
1 k
```

5

```
1 @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, 0.2), (0.1, 0.3), (0.1, 0.4), (0.2, 0.0), (0.2, 0.1), (0.2, 0.2), (0.2, 0.3), (0.2, 0.4), (0.3, 0.0), (0.3, 0.1), (0.3, 0.2), (0.3, 0.3), (0.3, 0.4), (0.4, 0.0), (0.4, 0.1), (0.4, 0.2), (0.4, 0.3), (0.4, 0.4)]

1 begin
2   g_1a = Graphs.grid([k, k]) # Grafo reticular k x k
3   vcolor_1a = [:white for i in 1:nv(g_1a)];
4   custom_layout_1a = [(i,j) for i = 0:0.1:(k-1)*0.1 ) for j = 0:0.1:(
5     (k-1)*0.1)];
6   end

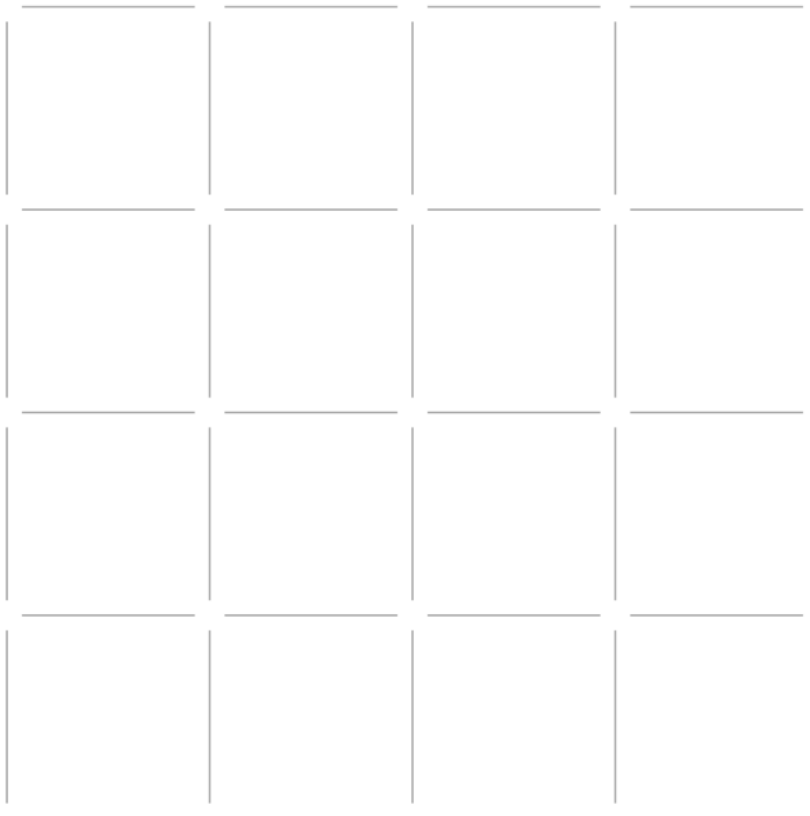
```

▶DataAspect()

```

1 begin
2   f_1a, ax_1a, p_1a = graphplot(g_1a; layout= custom_layout_1a,
3     node_color=vcolor_1a, edge_width=0.3, node_size=20);
4   hidedeclarations!(ax_1a)
5   hidedspines!(ax_1a)
6   ax_1a.aspect = DataAspect()
7   # sup ilabels = 1:400
8 end

```



Definamos el valor de  $\beta$ :

```
beta_1a = 0.01
```

```
1 beta_1a = 0.01
```

neighbor\_vertex (generic function with 1 method)

```
1 function neighbor_vertex(g, v, p)
2     k_plus = 0
3     k_minus = 0
4
5     for i in edges(g)
6         if src(i) == v
7             if p.node_color[][dst(i)] == :white
8                 k_minus += 1
9             elseif p.node_color[][dst(i)] == :black
10                k_plus += 1
11            end
12        elseif dst(i) == v;
13            if p.node_color[][src(i)] == :white
14                k_minus += 1
15            elseif p.node_color[][src(i)] == :black
16                k_plus += 1
17            end
18        end
19    end
20    return k_plus, k_minus
21 end
22
```

Implementamos el Gibbs sampler:

`gibbs_sampler` (generic function with 1 method)

```
1 # Construcción Gibbs Sampler
2
3 # -1 -> Blanco
4 # +1 -> Gris
5
6 function gibbs_sampler(g, p)
7     global k
8     v = rand( 1:k^2); # Seleccionar vértice aleatoriamente
9
10    k_plus_v, k_minus_v = neighbor_vertex(g, v, p) # Obtener número de vecinos
11                                           # con -1 y con +1.
12
13    u_n1 = rand()
14    # println(u_n1)
15    bound_upp = exp(2*beta_1a*(k_plus_v - k_minus_v)) / (exp(2*beta_1a*(k_plus_v -
16    k_minus_v)) + 1)
17    # println(bound_upp)
18
19    if u_n1 < bound_upp
20        p.node_color[][v] = :black
21        p.node_color = p.node_color[]
22    else
23        p.node_color[][v] = :white
24        p.node_color = p.node_color[]
25    end
26
27    # println(u_n1 < bound_upp)
28 end
```

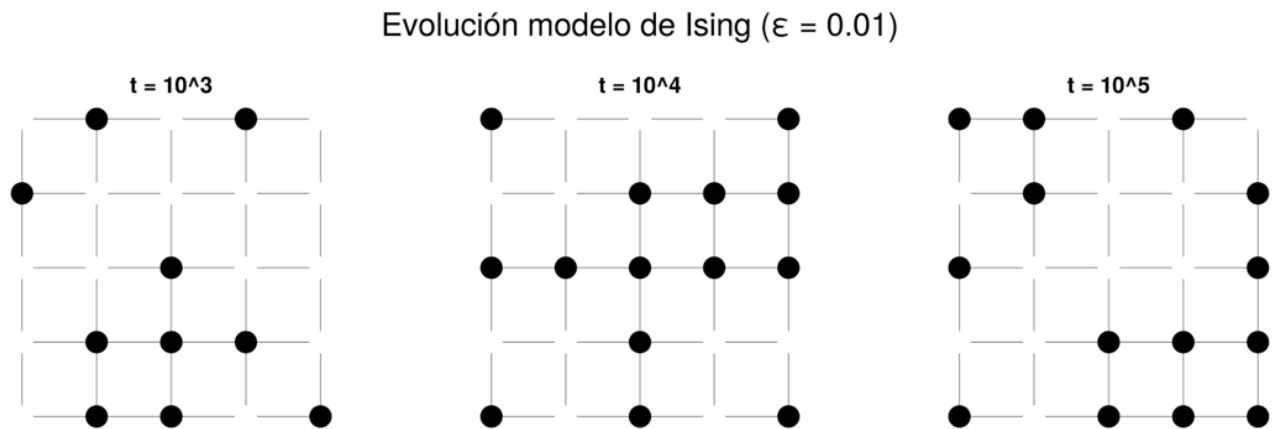
Corremos la cadena con un total de  $10^5$  pasos:

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

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

Makie.Label()

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



1 [fig\\_](#)

neighbor\_vertex\_ (generic function with 1 method)

gibbs\_sampler\_ (generic function with 1 method)

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

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

1 `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{}`

1 `samples = Dict{Float64, Vector{Vector{Vector{Symbol}}}}{}`

Generamos las muestras por cada valor de  $\beta$ :

```

1 for beta in beta_values
2   Random.seed!(100) # Semilla para replicar resultados
3   samples[beta] = Vector{Vector{Vector{Symbol}}}()
4
5   # Generar 100 muestras para cada beta
6   for _ in 1:100
7     node_colors = [:white for i in 1:nv(g_1a)]
8     sample_states = Vector{Vector{Symbol}}()
9
10    # Correr Gibbs sampler
11    for i in 1:10^5
12      node_colors = gibbs_sampler_(g_1a, node_colors, beta)
13
14      # Guardar estados
15      push!(sample_states, copy(node_colors))
16    end
17
18    # Guardar estados de la muestra
19    push!(samples[beta], sample_states)
20  end
21 end

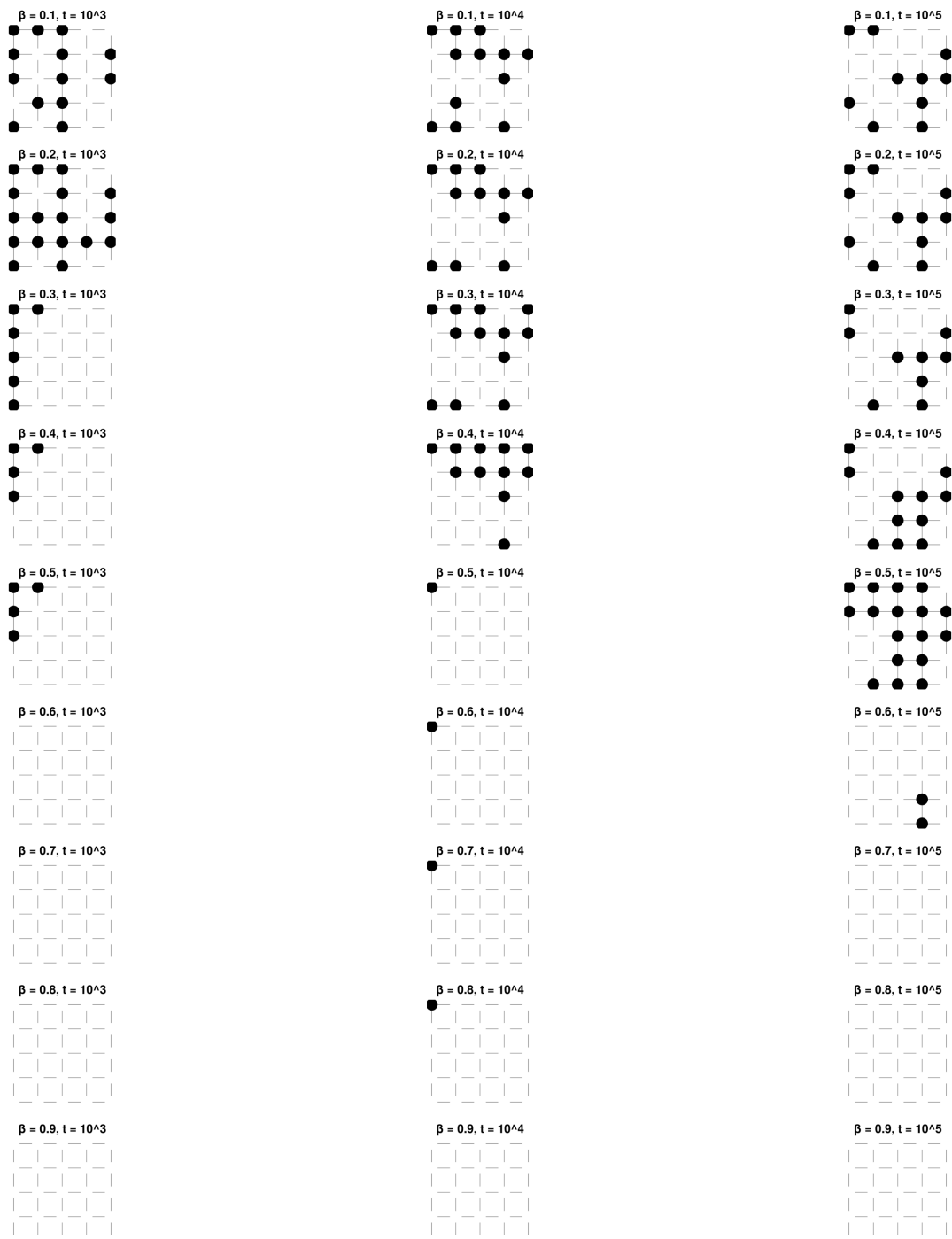
```

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  $\beta$ .

```

1 begin
2   fig = Figure(size = (1500, 1500))
3   axes = [Axis(fig[i, j], aspect = DataAspect()) for i in 1:9, j in 1:3]
4   for (i, beta) in enumerate(beta_values)
5     for (j, j1) in enumerate([10^3, 10^4, 10^5])
6       state_index = j1 # 1 -> 10^3, 2 -> 10^4, 3 -> 10^5
7       node_colors = samples[beta][end][state_index]
8
9       graphplot!(axes[i, j], g_1a; layout=custom_layout_1a,
10        node_color=node_colors, edge_width=0.3, node_size=20)
11
12       hidedeclarations!(axes[i, j])
13       hidedspines!(axes[i, j])
14       axes[i, j].title = "β = $beta, t = 10^$(j+2)"
15     end
16   end
17 end

```



1 [fig](#)

Veamos algunas estadísticas:

```

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

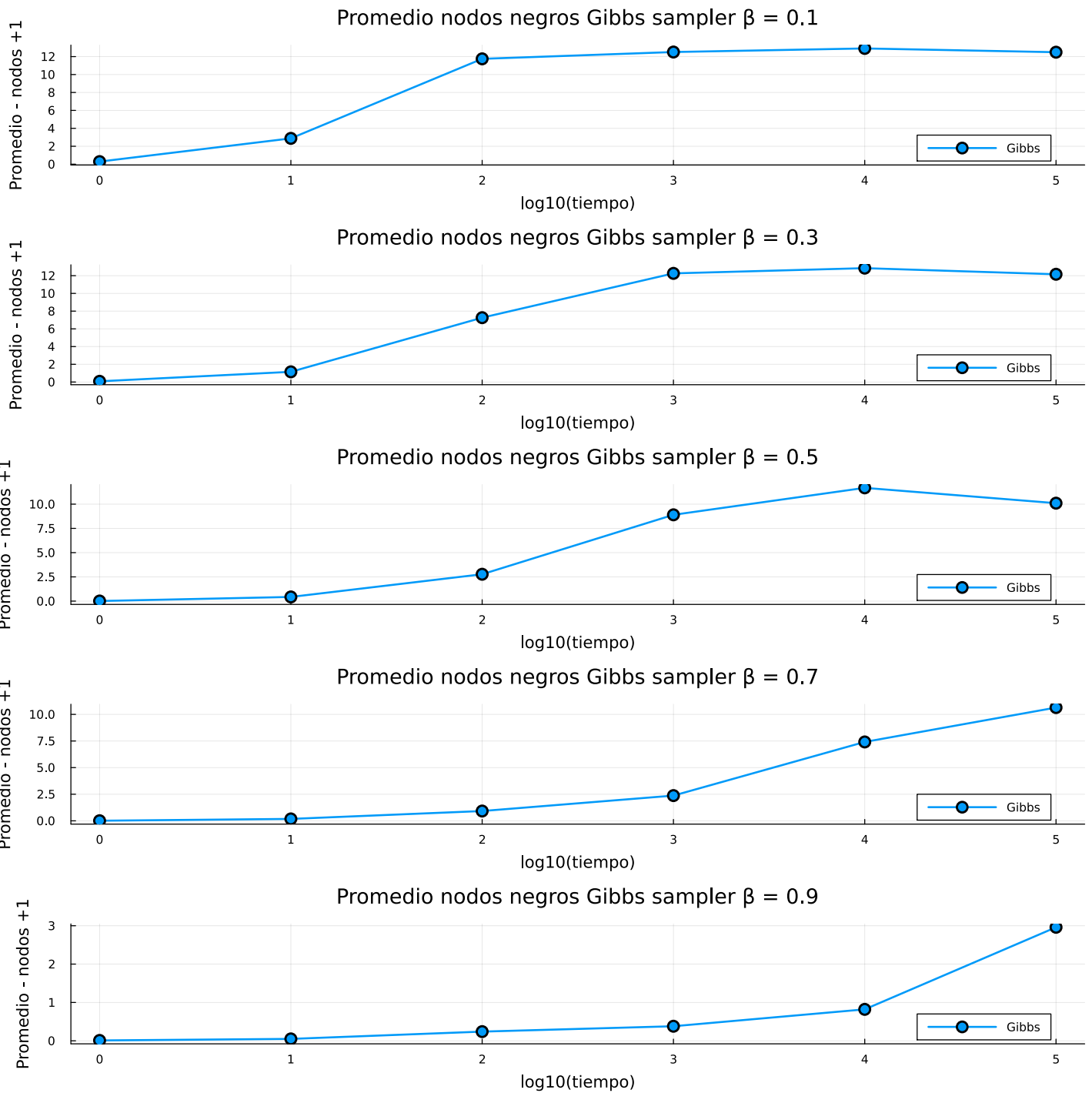
```

```

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

```





Lo cual tiene mucho sentido pues para  $\beta$  mayor a  $\beta_c$  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)

```
1 function metropolis_step(g, node_colors, beta)
2     v = rand(1:nv(g))
3     current_color = node_colors[v]
4     proposed_color = (current_color == :white) ? :black : :white
5
6     # Calcular el cambio en la energía
7     delta_E = 0
8     for neighbor in neighbors(g, v)
9         if node_colors[neighbor] == current_color
10             delta_E += 2
11         else
12             delta_E -= 2
13         end
14     end
15
16     # Calcular la probabilidad de aceptación
17     acceptance_prob = min(1, exp(-beta * delta_E))
18
19     # Decidir si aceptar el cambio
20     if rand() < acceptance_prob
21         node_colors[v] = proposed_color
22     end
23
24     return node_colors
25 end
```

run\_metropolis (generic function with 1 method)

```
1 function run_metropolis(g, beta, num_steps)
2     node_colors = [:white for _ in 1:nv(g)]
3     samples = Vector{Vector{Symbol}}{0}()
4
5     for step in 1:num_steps
6         node_colors = metropolis_step(g, node_colors, beta)
7
8         push!(samples, copy(node_colors))
9     end
10
11     return samples
12 end
```

Generaríamos las muestras así:

```

1 begin
2   Random.seed!(100) # Semilla para replicar resultados
3   samples_m = Dict{Float64, Vector{Vector{Vector{Symbol}}}}{ }
4
5   for beta in beta_values
6     samples_m[beta] = [run_metropolis(g_1a, beta, 10^5) for _ in 1:100]
7   end
8 end

```

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

```

1 begin
2   fig1 = Figure(size = (1000, 1000))
3   axes1 = [Axis(fig1[i, j], aspect = DataAspect()) for i in 1:9, j in 1:3]
4   for (i, beta) in enumerate(beta_values)
5     for (j, j1) in enumerate([10^3, 10^4, 10^5])
6       state_index1 = j1 # 1 -> 10^3, 2 -> 10^4, 3 -> 10^5
7       node_colors1 = samples_m[beta][end][state_index1]
8
9       graphplot!(axes1[i, j], g_1a; layout=custom_layout_1a,
10        node_color=node_colors1, edge_width=0.3, node_size=20)
11
12       hidedevelopments!(axes1[i, j])
13       hidedspines!(axes1[i, j])
14       axes1[i, j].title = "β = $beta, t = 10^$(j+2)"
15     end
16   end
17 end

```

Para ver algunas estadísticas:

```

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

```

**b)** Use el algoritmo de Propp-Wilson para obtener **100** muestras exactas del modelo de Ising, tomando los mismos valores para  $\beta$  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)

1 begin
2     global random_dict = Dict{Int, Float64}{}
3
4     function gibbs_sampler_pw(g, p, time)
5         v = rand( 1:k^2);
6
7         k_plus_v, k_minus_v = neighbor_vertex(g, v, p)
8
9         if !haskey(random_dict, time)
10             random_dict[time] = rand()
11         end
12
13         u_n1 = random_dict[time]
14
15         bound_upp = exp(2*beta_1a*(k_plus_v - k_minus_v)) / (exp(2*beta_1a*
16             (k_plus_v - k_minus_v)) + 1)
17
18         if u_n1 < bound_upp
19             p.node_color[][v] = :black
20             p.node_color = p.node_color[]
21         else
22             p.node_color[][v] = :white
23             p.node_color = p.node_color[]
24         end
25     end
26 end
```

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

Propp\_Wilson\_iter (generic function with 1 method)

```
1 # Ejecutar Propp-Wilson con Sandwiching
2
3 function Propp_Wilson_iter(n) # time of starting: 2^n
4     g_min = Graphs.grid([k, k])
5
6     vcolor_min = [:white for i in 1:nv(g_min)]
7     custom_layout_min = [(i,j) for i = 0:0.1:( (k-1)*0.1 ) for j = 0:0.1:(
8         (k-1)*0.1 ) ]
9     f_min, ax_min, p_min = graphplot(g_min; layout= custom_layout_min,
10         node_color=vcolor_min, edge_width=0.3, node_size=20);
11
12     hidedecorations!(ax_min)
13     hidespines!(ax_min)
14     ax_min.aspect = DataAspect()
15
16     g_max = Graphs.grid([k, k])
17
18     vcolor_max = [:black for i in 1:nv(g_max)]
19     custom_layout_max = [(i,j) for i = 0:0.1:( (k-1)*0.1 ) for j = 0:0.1:( (k-1)*0.1
20         ) ]
21     f_max, ax_max, p_max = graphplot(g_max; layout= custom_layout_max,
22         node_color=vcolor_max, edge_width=0.3, node_size=20);
23
24     hidedecorations!(ax_max)
25     hidespines!(ax_max)
26
27     ax_max.aspect = DataAspect()
28
29     for i in -2^n:0
30         gibbs_sampler_pw(g_min, p_min, i)
31         gibbs_sampler_pw(g_max, p_max, i)
32     end
33
34     return (f_max, ax_max, p_max), (f_min, ax_min, p_min)
35
36 end
```

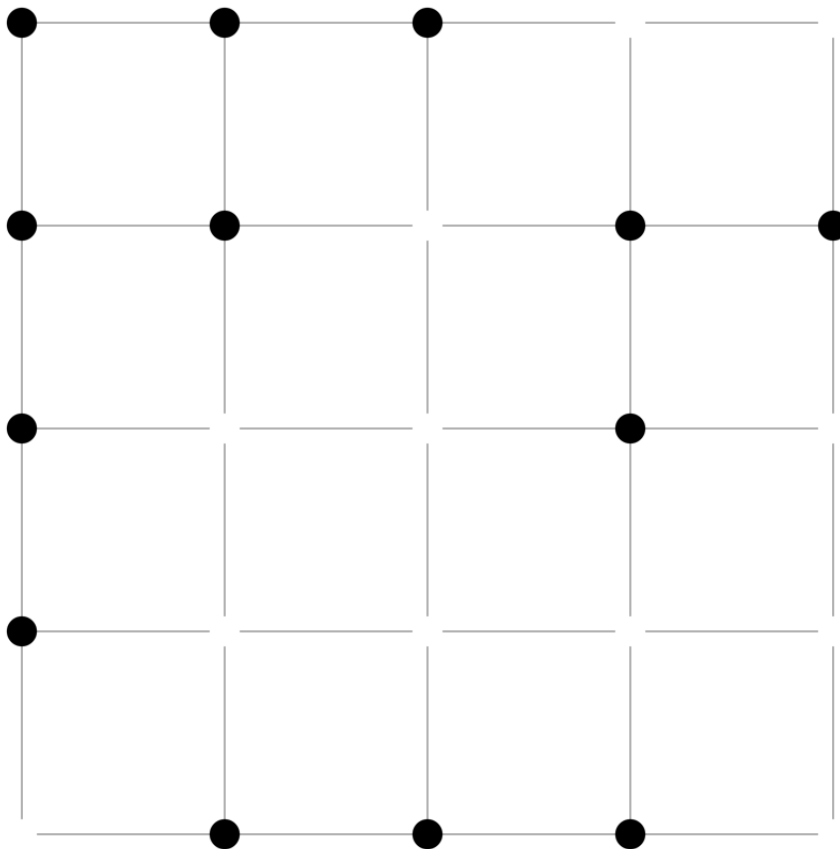
A continuación corremos el algoritmo. Una modificación notable es que las cadenas coalescerán en el momento en el que el número de nodos negros sea el mismo o difieran en uno a lo sumo, esto debido a que el tiempo de coalescencia para que las cadenas lleguen a exactamente el mismo estado parece ser muy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

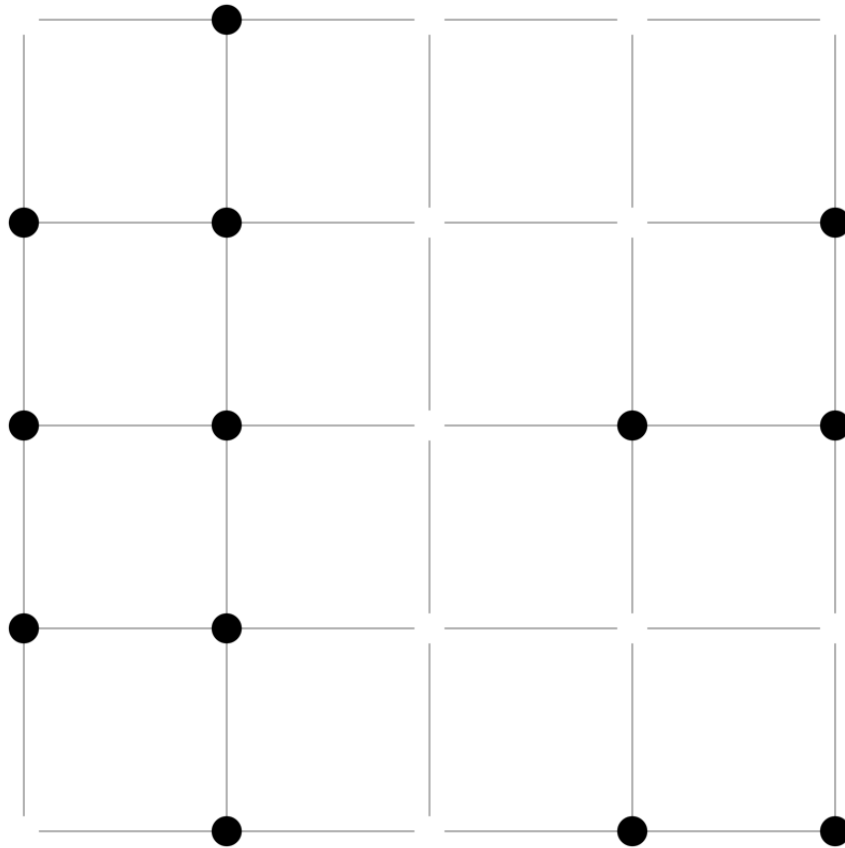
```

1 begin
2   m_ = 0
3   mi_ = 0
4   n = 1;
5   while true
6
7     Graph_max, Graph_min = Propp_Wilson_iter(n)
8
9     # println(Graph_max[3].node_color)
10    # println(Graph_min[3].node_color)
11
12    n += 1
13
14    if abs( count( i->(i == :black) , Graph_max[3].node_color[]) - count( i->
15      (i == :black), Graph_min[3].node_color[])) <= 1
16      m_ = Graph_max[1]
17      mi_ = Graph_min[1]
18      break
19    end
20  end
end

```

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





1 mi\_

gibbs\_sampler\_pw\_ (generic function with 1 method)

```

1 function gibbs_sampler_pw_(g, node_colors, beta, time)
2     v = rand(1:nv(g))
3     k_plus, k_minus = neighbor_vertex_(g, v, node_colors)
4
5     if !haskey(random_dict, time)
6         random_dict[time] = rand()
7     end
8     u_n1 = random_dict[time]
9
10    bound_upp = exp(2*beta*(k_plus - k_minus)) / (exp(2*beta*(k_plus - k_minus)) +
11 1)
12
13    if u_n1 < bound_upp
14        node_colors[v] = :black
15    else
16        node_colors[v] = :white
17    end
18
19    return node_colors
end

```

Propp\_Wilson\_iter\_ (generic function with 1 method)

```
1 function Propp_Wilson_iter_(n, beta)
2   g_min = Graphs.grid([k, k])
3   node_colors_min = [:white for i in 1:nv(g_min)]
4
5   g_max = Graphs.grid([k, k])
6   node_colors_max = [:black for i in 1:nv(g_max)]
7
8   for i in -2^n:0
9     node_colors_min = gibbs_sampler_pw_(g_min, node_colors_min, beta, i)
10    node_colors_max = gibbs_sampler_pw_(g_max, node_colors_max, beta, i)
11  end
12
13  return node_colors_max, node_colors_min
14 end
```

run\_propp\_wilson\_ (generic function with 1 method)

```
1 function run_propp_wilson_(beta, cv)
2   n = 1
3   while true
4     node_colors_max, node_colors_min = Propp_Wilson_iter_(n, beta)
5
6     n += 1
7
8     if abs(count(i -> i == :black, node_colors_max) - count(i -> i == :black,
node_colors_min)) <= 1 || (n > 10 && cv == true)
9       return node_colors_max, 2^n
10    end
11  end
12 end
```

Generamos las muestras utilizadas utilizando los valores  $\beta$  del punto anterior (0.1, 0.4, 0.9).

1 Enter cell code...

► Dict()

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



```

1 begin
2   for beta in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
3     samples_pw[beta] = Vector{Vector{Symbol}}{]()
4     coalescence_times[beta] = Vector{Int}{]()
5     for _ in 1:100
6       node_colors_, coalescence_time = run_propp_wilson_(beta, true)
7       push!(samples_pw[beta], node_colors_)
8       push!(coalescence_times[beta], coalescence_time)
9     end
10  end
11 end

```

```

1 begin
2 igs = []
3 for beta in [0.1, 0.3, 0.5, 0.7, 0.9]
4   avg_black_ = mean([count(x -> x == :black, sample) for sample in
5     samples_pw[beta]])
6   println("Para  $\beta$  = $beta, el número promedio de nodos negros es: $avg_black_")
7   push!(igs, histogram([count(x -> x == :black, sample) for sample in
8     samples_pw[beta]],
9     title="Distribución de nodos negros ( $\beta$  = $beta)",
10    xlabel="Número de nodos negros", ylabel="Frecuencia")))
11 savefig("propp_wilson_histogram_beta_$beta.png")
12 end
13 end

```

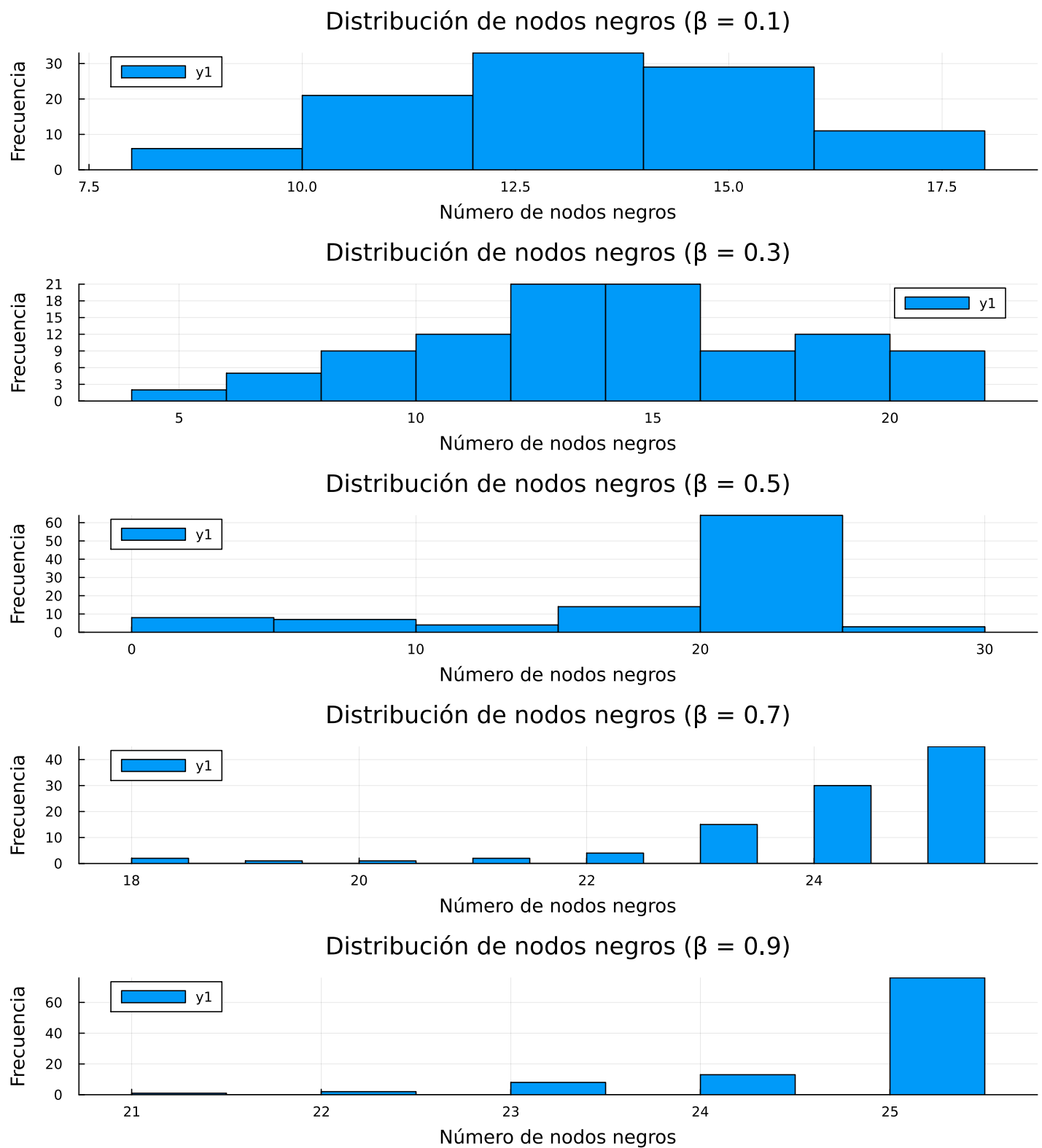
>-

```

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

```

?



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

Allí podemos visualizar el cambio de fase a partir del valor crítico  $\beta_c$ .

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)

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

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

promedio\_M (generic function with 1 method)

```
1 function promedio_M(samples)
2     return mean(calculate_M.(samples))
3 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

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

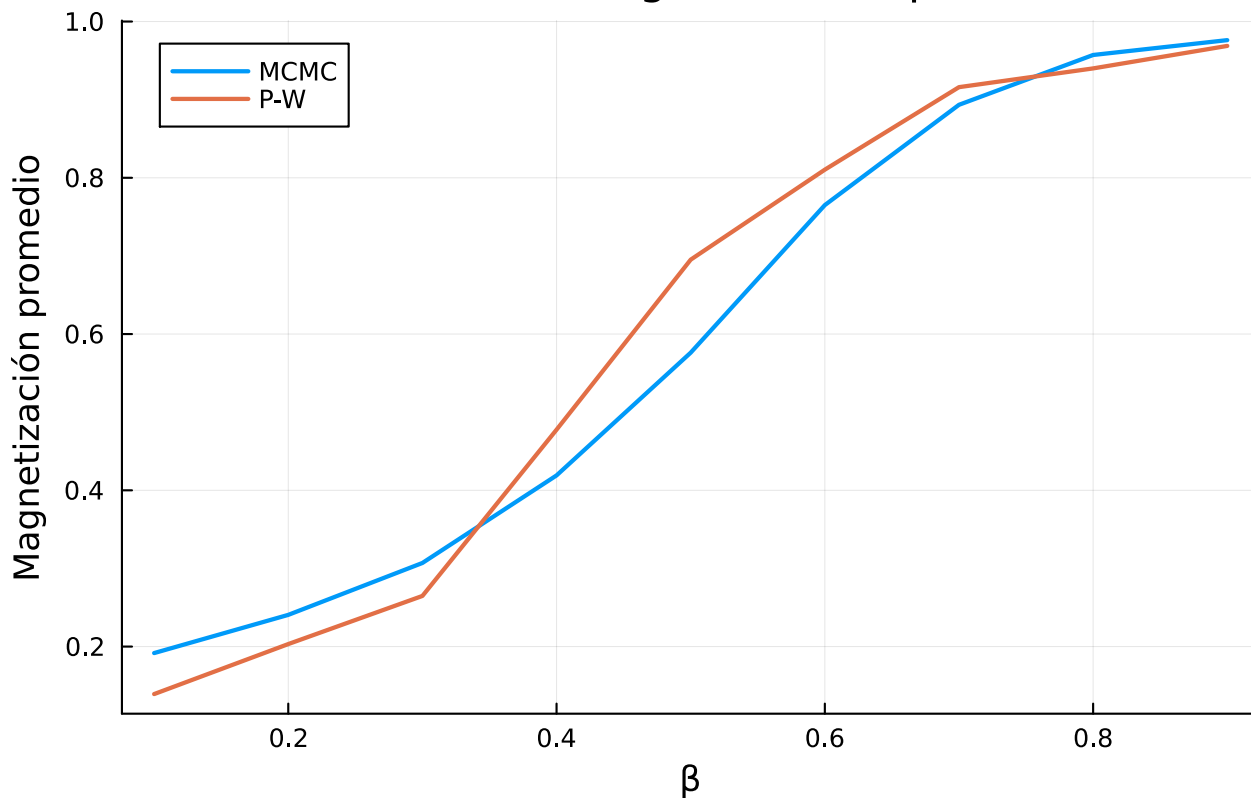
avg\_mag\_pw =

► Dict{0.4 ⇒ 0.4776, 0.7 ⇒ 0.916, 0.3 ⇒ 0.2648, 0.5 ⇒ 0.6952, 0.2 ⇒ 0.2032, 0.9 ⇒ 0.

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

Visualización tomando el valor absoluto de las magnetizaciones:

## MCMC vs PW: Magnetización promedio



```
1 begin
2   p_avg = Plots.plot(beta_values, [avg_mag_mcmc[beta] for beta in beta_values],
3     label = "MCMC", maker = :circle, linewidth=2, size = (600, 400))
4   Plots.plot!(beta_values, [avg_mag_pw[beta] for beta in beta_values], label = "P-
5     W", maker = :square, linewidth=2)
6   Plots.xlabel!(p_avg, " $\beta$ ")
7   Plots.ylabel!(p_avg, "Magnetización promedio")
8   Plots.title!(p_avg, "MCMC vs PW: Magnetización promedio")
9 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  $2^{25}$ . 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 enfriamiento 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"

```
1 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, 0.29935, 0.02671, 0.752099]
```

```
1 begin
2   locations__ = replace.(locations_, ",", ">=">".");
3
4   x_points = parse.(Float64, locations__."Coordenada X");
5   y_points = parse.(Float64, locations__."Coordenada Y");
6 end
```

```
► [0.0, 0.640195, 0.29346, 0.790026, 0.88871, 0.965121, 0.689975, 0.240618, 0.537452, 0.02671, 0.752099]
```

```
1 x_points
```

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

1 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)

```
1 # function: distance of path.
2 function dist_path(x::Vector, y::Vector)
3
4     dist = 0.0
5     n = length(x_points)
6
7     for i in 1:(n-1)
8         dist += sqrt( (y[i+1] - y[i])^2 + ( x[i+1] - x[i] )^2 )
9     end
10
11     return dist
12
13 end
```

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

new\_neighbor (generic function with 1 method)

```
1  # build a neighbor
2  function new_neighbor(x::Vector, y::Vector)
3
4      i = rand(2:76)
5      j = rand(i:76)
6
7      new_x = copy(x)
8      new_y = copy(y)
9
10     slice_reverse_x = reverse( new_x[i:j] )
11     slice_reverse_y = reverse( new_y[i:j] )
12
13     new_x[i:j] = slice_reverse_x
14     new_y[i:j] = slice_reverse_y
15
16     return new_x, new_y
17 end
18
```

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)

```
1 function simulated_annealing_1(x::Vector, y::Vector)
2
3     # Simulate Metropolis Chain
4     # cooling schedule (1, 1/10, 1/100, 1/1000, ...) and (1, 10, 100, 1000, ...)
5
6     curr_points = Dict{ "x" => x, "y" => y }
7
8     # final_neighboor = (p2_x, p2_y)
9
10    n = 1
11
12    # execute markov chain with transition matrix given in class
13
14    while n < 7
15
16        for i in 1:10^n # max 10^6
17
18            T = 1/(10^n)
19
20            p2_x, p2_y = new_neighboor(curr_points["x"], curr_points["y"])
21
22            e = dist_path(curr_points["x"], curr_points["y"])
23            e_ = dist_path(p2_x, p2_y)
24
25            u = rand()
26
27            if u < min( 1, exp( (e - e_)/T ) )
28                curr_points = Dict{ "x" => p2_x, "y" => p2_y }
29            #else
30            #     final_neighboor = (curr_points["x"], curr_points["y"])
31            end
32
33            end
34            n += 1
35        end
36    return curr_points
37 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 .

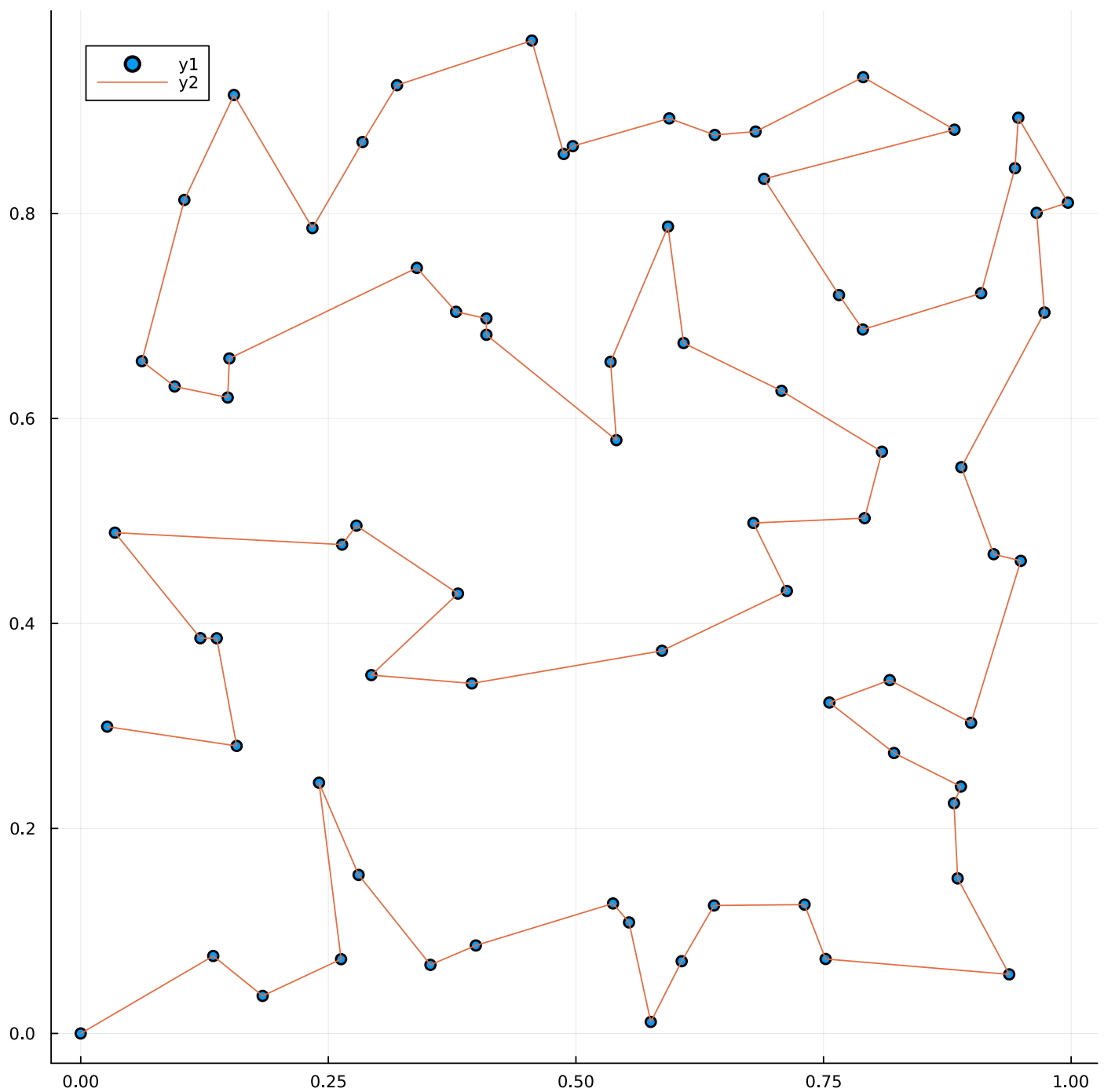
final\_neighboor =

►Dict{"x" => [0.0, 0.133813, 0.183704, 0.262884, 0.240618, 0.280562, 0.353121, 0.398956, 0.453121, 0.507273, 0.561425, 0.615577, 0.669729, 0.723881, 0.778033, 0.832185, 0.886337, 0.940489, 0.994641, 1.048793, 1.102945, 1.157097, 1.211249, 1.265401, 1.319553, 1.373705, 1.427857, 1.482009, 1.536161, 1.590313, 1.644465, 1.698617, 1.752769, 1.806921, 1.861073, 1.915225, 1.969377, 2.023529, 2.077681, 2.131833, 2.185985, 2.240137, 2.294289, 2.348441, 2.402593, 2.456745, 2.510897, 2.565049, 2.619201, 2.673353, 2.727505, 2.781657, 2.835809, 2.889961, 2.944113, 2.998265, 3.052417, 3.106569, 3.160721, 3.214873, 3.269025, 3.323177, 3.377329, 3.431481, 3.485633, 3.539785, 3.593937, 3.648089, 3.702241, 3.756393, 3.810545, 3.864697, 3.918849, 3.973001, 4.027153, 4.081305, 4.135457, 4.189609, 4.243761, 4.297913, 4.352065, 4.406217, 4.460369, 4.514521, 4.568673, 4.622825, 4.676977, 4.731129, 4.785281, 4.839433, 4.893585, 4.947737, 5.001889, 5.056041, 5.110193, 5.164345, 5.218497, 5.272649, 5.326801, 5.380953, 5.435105, 5.489257, 5.543409, 5.597561, 5.651713, 5.705865, 5.760017, 5.814169, 5.868321, 5.922473, 5.976625, 6.030777, 6.084929, 6.139081, 6.193233, 6.247385, 6.301537, 6.355689, 6.409841, 6.463993, 6.518145, 6.572297, 6.626449, 6.680601, 6.734753, 6.788905, 6.843057, 6.897209, 6.951361, 7.005513, 7.059665, 7.113817, 7.167969, 7.222121, 7.276273, 7.330425, 7.384577, 7.438729, 7.492881, 7.547033, 7.601185, 7.655337, 7.709489, 7.763641, 7.817793, 7.871945, 7.926097, 7.980249, 8.034401, 8.088553, 8.142705, 8.196857, 8.251009, 8.305161, 8.359313, 8.413465, 8.467617, 8.521769, 8.575921, 8.630073, 8.684225, 8.738377, 8.792529, 8.846681, 8.900833, 8.954985, 9.009137, 9.063289, 9.117441, 9.171593, 9.225745, 9.279897, 9.334049, 9.388201, 9.442353, 9.496505, 9.550657, 9.604809, 9.658961, 9.713113, 9.767265, 9.821417, 9.875569, 9.929721, 9.983873, 10.038025, 10.092177, 10.146329, 10.200481, 10.254633, 10.308785, 10.362937, 10.417089, 10.471241, 10.525393, 10.579545, 10.633697, 10.687849, 10.742001, 10.796153, 10.850305, 10.904457, 10.958609, 11.012761, 11.066913, 11.121065, 11.175217, 11.229369, 11.283521, 11.337673, 11.391825, 11.445977, 11.500129, 11.554281, 11.608433, 11.662585, 11.716737, 11.770889, 11.825041, 11.879193, 11.933345, 11.987497, 12.041649, 12.095801, 12.149953, 12.204105, 12.258257, 12.312409, 12.366561, 12.420713, 12.474865, 12.529017, 12.583169, 12.637321, 12.691473, 12.745625, 12.799777, 12.853929, 12.908081, 12.962233, 13.016385, 13.070537, 13.124689, 13.178841, 13.232993, 13.287145, 13.341297, 13.395449, 13.449601, 13.503753, 13.557905, 13.612057, 13.666209, 13.720361, 13.774513, 13.828665, 13.882817, 13.936969, 13.991121, 14.045273, 14.099425, 14.153577, 14.207729, 14.261881, 14.316033, 14.370185, 14.424337, 14.478489, 14.532641, 14.586793, 14.640945, 14.695097, 14.749249, 14.803401, 14.857553, 14.911705, 14.965857, 15.020009, 15.074161, 15.128313, 15.182465, 15.236617, 15.290769, 15.344921, 15.399073, 15.453225, 15.507377, 15.561529, 15.615681, 15.669833, 15.723985, 15.778137, 15.832289, 15.886441, 15.940593, 15.994745, 16.048897, 16.103049, 16.157201, 16.211353, 16.265505, 16.319657, 16.373809, 16.427961, 16.482113, 16.536265, 16.590417, 16.644569, 16.698721, 16.752873, 16.807025, 16.861177, 16.915329, 16.969481, 17.023633, 17.077785, 17.131937, 17.186089, 17.240241, 17.294393, 17.348545, 17.402697, 17.456849, 17.511001, 17.565153, 17.619305, 17.673457, 17.727609, 17.781761, 17.835913, 17.890065, 17.944217, 17.998369, 18.052521, 18.106673, 18.160825, 18.214977, 18.269129, 18.323281, 18.377433, 18.431585, 18.485737, 18.539889, 18.594041, 18.648193, 18.702345, 18.756497, 18.810649, 18.864801, 18.918953, 18.973105, 19.027257, 19.081409, 19.135561, 19.189713, 19.243865, 19.298017, 19.352169, 19.406321, 19.460473, 19.514625, 19.568777, 19.622929, 19.677081, 19.731233, 19.785385, 19.839537, 19.893689, 19.947841, 20.001993, 20.056145, 20.110297, 20.164449, 20.218601, 20.272753, 20.326905, 20.381057, 20.435209, 20.489361, 20.543513, 20.597665, 20.651817, 20.705969, 20.760121, 20.814273, 20.868425, 20.922577, 20.976729, 21.030881, 21.085033, 21.139185, 21.193337, 21.247489, 21.301641, 21.355793, 21.409945, 21.464097, 21.518249, 21.572401, 21.626553, 21.680705, 21.734857, 21.789009, 21.843161, 21.897313, 21.951465, 22.005617, 22.059769, 22.113921, 22.168073, 22.222225, 22.276377, 22.330529, 22.384681, 22.438833, 22.492985, 22.547137, 22.601289, 22.655441, 22.709593, 22.763745, 22.817897, 22.872049, 22.926201, 22.980353, 23.034505, 23.088657, 23.142809, 23.196961, 23.251113, 23.305265, 23.359417, 23.413569, 23.467721, 23.521873, 23.576025, 23.630177, 23.684329, 23.738481, 23.792633, 23.846785, 23.900937, 23.955089, 24.009241, 24.063393, 24.117545, 24.171697, 24.225849, 24.279999, 24.334151, 24.388303, 24.442455, 24.496607, 24.550759, 24.604911, 24.659063, 24.713215, 24.767367, 24.821519, 24.875671, 24.929823, 24.983975, 25.038127, 25.092279, 25.146431, 25.200583, 25.254735, 25.308887, 25.363039, 25.417191, 25.471343, 25.525495, 25.579647, 25.633799, 25.687951, 25.742103, 25.796255, 25.850407, 25.904559, 25.958711, 26.012863, 26.067015, 26.121167, 26.175319, 26.229471, 26.283623, 26.337775, 26.391927, 26.446079, 26.500231, 26.554383, 26.608535, 26.662687, 26.716839, 26.770991, 26.825143, 26.879295, 26.933447, 26.987599, 27.041751, 27.095903, 27.150055, 27.204207, 27.258359, 27.312511, 27.366663, 27.420815, 27.474967, 27.529119, 27.583271, 27.637423, 27.691575, 27.745727, 27.799879, 27.854031, 27.908183, 27.962335, 28.016487, 28.070639, 28.124791, 28.178943, 28.233095, 28.287247, 28.341399, 28.395551, 28.449703, 28.503855, 28.558007, 28.612159, 28.666311, 28.720463, 28.774615, 28.828767, 28.882919, 28.937071, 28.991223, 29.045375, 29.099527, 29.153679, 29.207831, 29.261983, 29.316135, 29.370287, 29.424439, 29.478591, 29.532743, 29.586895, 29.641047, 29.695199, 29.749351, 29.803503, 29.857655, 29.911807, 29.965959, 30.020111, 30.074263, 30.128415, 30.182567, 30.236719, 30.290871, 30.345023, 30.399175, 30.453327, 30.507479, 30.561631, 30.615783, 30.669935, 30.724087, 30.778239, 30.832391, 30.886543, 30.940695, 30.994847, 31.048999, 31.103151, 31.157303, 31.211455, 31.265607, 31.319759, 31.373911, 31.428063, 31.482215, 31.536367, 31.590519, 31.644671, 31.698823, 31.752975, 31.807127, 31.861279, 31.915431, 31.969583, 32.023735, 32.077887, 32.132039, 32.186191, 32.240343, 32.294495, 32.348647, 32.402799, 32.456951, 32.511103, 32.565255, 32.619407, 32.673559, 32.727711, 32.781863, 32.836015, 32.890167, 32.944319, 32.998471, 33.052623, 33.106775, 33.160927, 33.215079, 33.269231, 33.323383, 33.377535, 33.431687, 33.485839, 33.539991, 33.594143, 33.648295, 33.702447, 33.756599, 33.810751, 33.864903, 33.919055, 33.973207, 34.027359, 34.081511, 34.135663, 34.189815, 34.243967, 34.298119, 34.352271, 34.406423, 34.460575, 34.514727, 34.568879, 34.623031, 34.677183, 34.731335, 34.785487, 34.839639, 34.893791, 34.947943, 35.002095, 35.056247, 35.110399, 35.164551, 35.218703, 35.272855, 35.327007, 35.381159, 35.435311, 35.489463, 35.543615, 35.597767, 35.651919, 35.706071, 35.760223, 35.814375, 35.868527, 35.922679, 35.976831, 36.030983, 36.085135, 36.139287, 36.193439, 36.247591, 36.301743, 36.355895, 36.410047, 36.464199, 36.518351, 36.572503, 36.626655, 36.680807, 36.734959, 36.789111, 36.843263, 36.897415, 36.951567, 37.005719, 37.059871, 37.114023, 37.168175, 37.222327, 37.276479, 37.330631, 37.384783, 37.438935, 37.493087, 37.547239, 37.601391, 37.655543, 37.709695, 37.763847, 37.817999, 37.872151, 37.926303, 37.980455, 38.034607, 38.088759, 38.142911, 38.197063, 38.251215, 38.305367, 38.359519, 38.413671, 38.467823, 38.521975, 38.576127, 38.630279, 38.684431, 38.738583, 38.792735, 38.846887, 38.901039, 38.955191, 39.009343, 39.063495, 39.117647, 39.171799, 39.225951, 39.280103, 39.334255, 39.388407, 39.442559, 39.496711, 39.550863, 39.605015, 39.659167, 39.713319, 39.767471, 39.821623, 39.875775, 39.929927, 40.000000]

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

Ahora graficamos las colonias y el recorrido optimizado:





```

1 begin
2   gr(size = (800, 800))
3   Plots.scatter(x_points, y_points)
4   Plots.plot!(final_neighbour["x"], final_neighbour["y"])
5 end

```

Obteniendo la siguiente distancia:

7.376463295493744

```

1 dist_path(final_neighbour["x"], final_neighbour["y"])

```

**b)** Repita el ítem 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.

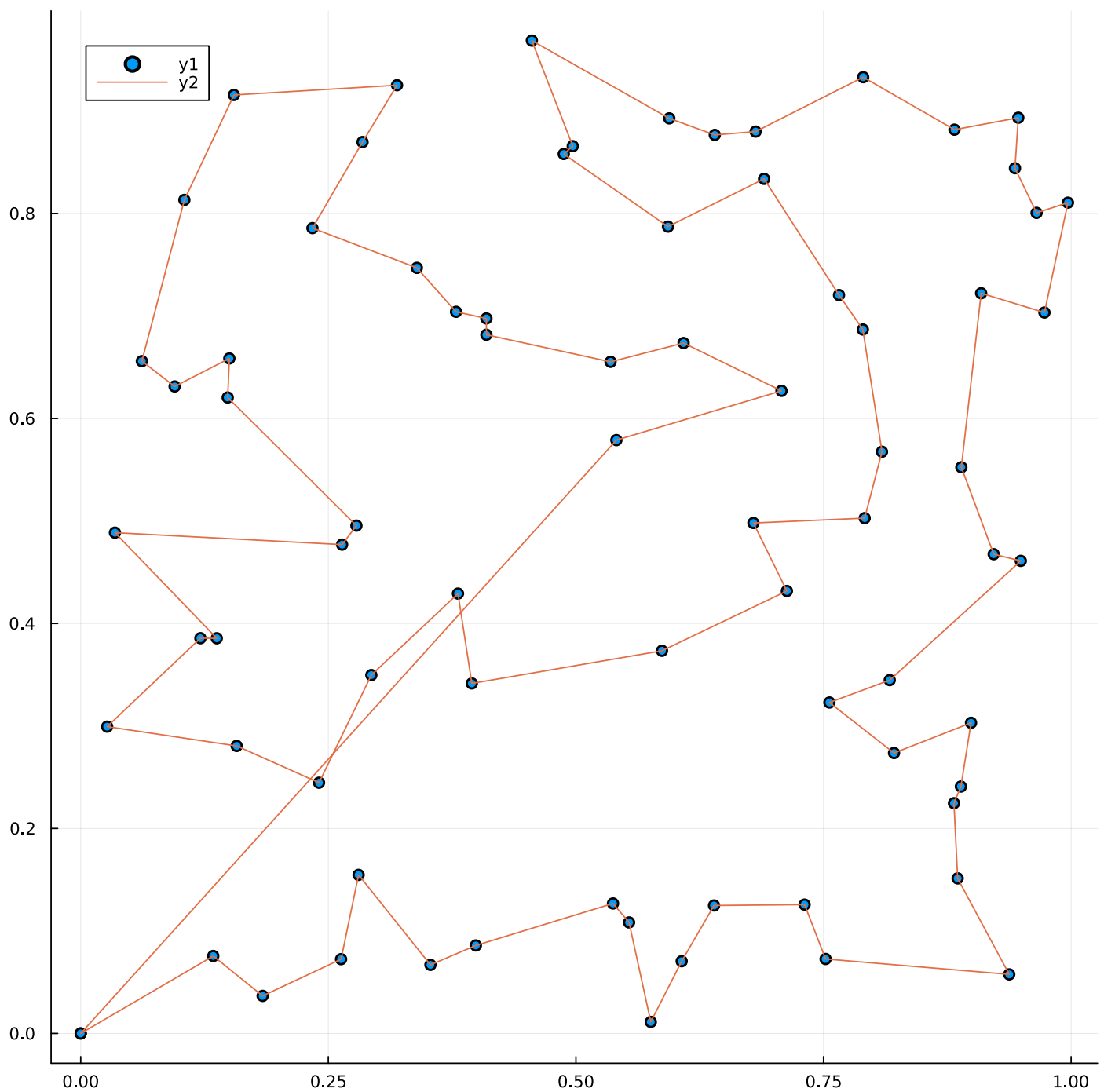
```
► [0.0, 0.876619, 0.34965, 0.932975, 0.24091, 0.800584, 0.833759, 0.24463, 0.126833, 0.2993
```

```
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, 0.026
```

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



```

1 begin
2   gr(size = (800, 800))
3   Plots.scatter(x_points_b, y_points_b)
4   Plots.plot!(final_neighbor_b["x"], final_neighbor_b["y"])
5 end

```

Obteniendo la siguiente distancia:

7.054698150065478

```

1 dist_path(final_neighbor_b["x"], final_neighbor_b["y"])
2
3 # Notar que para n = 7, ya no da un camino que minimice la distancia.

```

