

# Algoritmo de fuerza bruta

## Algoritmo - Flood Fill

Cuéllar Hernández Cinthya Sofía

223931799

Hernández Santos Karen Cecilia

219770168

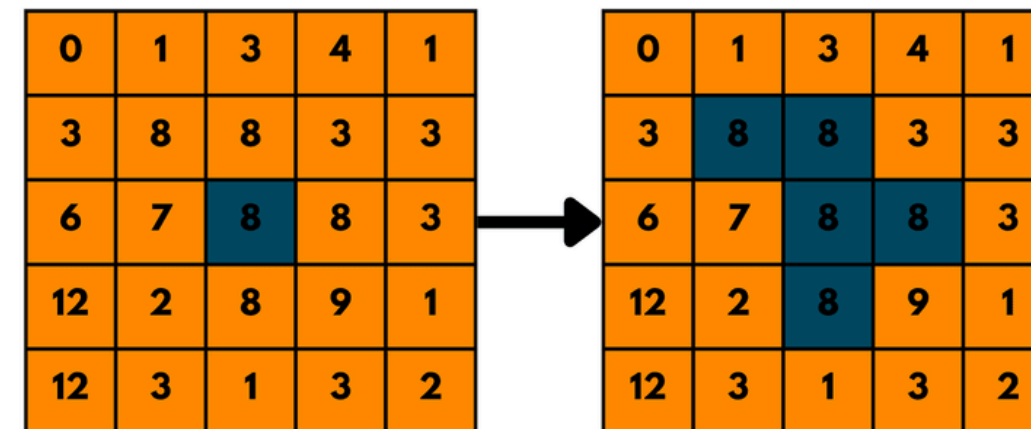
Valentín Gallardo José Eduardo

218452685

# Flood Fill

Flood Fill es un algoritmo que recorre y rellena áreas conectadas en una matriz.

Se usa comúnmente en gráficos, mapas y procesamiento de imágenes.



0	1	3	4	1
3	8	8	3	3
6	7	8	8	3
12	2	8	9	1
12	3	1	3	2

0	1	3	4	1
3	8	8	3	3
6	7	8	8	3
12	2	8	9	1
12	3	1	3	2

# Importancia

## *del algoritmo*

Este algoritmo Flood Fill es fundamental en la computación gráfica y en problemas de recorrido de estructuras de datos, tiene una alta aplicabilidad en múltiples áreas como la edición de imágenes y videojuegos. Este algoritmo es importante porque combina la eficiencia razonable en el manejo de grandes áreas con una amplia aplicabilidad en gráficos, procesamiento de imágenes y teoría de grafos.

# Código

```
1  # Importa la librería tkinter para la interfaz gráfica y la renombra como 'tk'
2  import tkinter as tk
3
4  # Importa 'deque' (cola doblemente terminada) desde el módulo collections
5  # Se usa para el algoritmo eficiente de flood fill
6  from collections import deque
7
8  # Define una clase principal para la aplicación de Flood Fill
9  class FloodFillApp:
10     # Método constructor que se ejecuta al crear una instancia de la clase
11     def __init__(self, root):
12         # Guarda la ventana principal (root) como atributo de la clase
13         self.root = root
14         # Establece el título de la ventana
15         self.root.title("Flood Fill Visual - Tkinter")
16
17         # CONFIGURACIÓN DEL GRID
18         # Tamaño de la grilla (no se usa directamente, pero está definido)
19         self.grid_size = 20
20         # Ancho de la grilla en celdas (30 columnas)
21         self.grid_width = 30
22         # Alto de la grilla en celdas (20 filas)
23         self.grid_height = 20
24         # Tamaño de cada celda en píxeles
25         self.cell_size = 20
```

# Código

```
27 # LISTA DE COLORES DISPONIBLES
28 # Cada color corresponde a un número: 0=blanco, 1=negro, 2=rojo, etc.
29 self.colors = ["white", "black", "red", "green", "blue", "yellow", "purple"]
30 # Color actual seleccionado para el relleno (inicia con rojo)
31 self.current_color = 2 # red
32
33 # CREACIÓN DE LA MATRIZ DEL GRID
34 # Crea una matriz 2D (20 filas x 30 columnas) llena de ceros (color blanco)
35 # [[0, 0, 0, ...], [0, 0, 0, ...], ...] - 20 listas de 30 ceros cada una
36 self.grid = [[0 for _ in range(self.grid_width)] for _ in range(self.grid_height)]
37
38 # Llama al método para crear los elementos de la interfaz gráfica
39 self.create_widgets()
40
```

# Código

```
41 # Método para crear todos los elementos visuales de la interfaz
42 def create_widgets(self):
43     # Crea un frame (marco) principal dentro de la ventana root
44     main_frame = tk.Frame(self.root)
45     # Empaqueta el frame con padding de 10 píxeles en X y Y
46     main_frame.pack(padx=10, pady=10)
47
48     # CREACIÓN DEL CANVAS (LIENZO)
49     # Crea un canvas (área de dibujo) dentro del frame principal
50     self.canvas = tk.Canvas(
51         main_frame, # Parent: el frame principal
52         width=self.grid_width * self.cell_size, # Ancho: 30 * 20 = 600px
53         height=self.grid_height * self.cell_size, # Alto: 20 * 20 = 400px
54         bg="white" # Color de fondo blanco
55     )
56     # Empaqueta el canvas en la interfaz
57     self.canvas.pack()
58
```



# Código

```
59 # CREACIÓN DEL FRAME DE CONTROLES
60 # Crea un frame para contener los botones de colores
61 control_frame = tk.Frame(main_frame)
62 # Empaqueta el frame de controles con padding vertical de 10 píxeles
63 control_frame.pack(pady=10)
64
65 # CREACIÓN DE BOTONES DE COLORES
66 # Itera sobre los colores desde el índice 2 hasta el final
67 # enumerate(self.colors[2:], 2) -> (2, "red"), (3, "green"), etc.
68 for i, color in enumerate(self.colors[2:], 2):
69     # Crea un botón para cada color
70     btn = tk.Button(
71         control_frame, # Parent: frame de controles
72         bg=color, # Color de fondo del botón
73         width=3, # Ancho del botón en caracteres
74         height=1, # Alto del botón en líneas de texto
75         # Comando que se ejecuta al hacer click: llama a set_color con el índice
76         command=lambda c=i: self.set_color(c)
77     )
78     # Empaqueta el botón a la izquierda con padding horizontal de 2 píxeles
79     btn.pack(side=tk.LEFT, padx=2)
```

# Código

```
81 # CREACIÓN DE ETIQUETA INFORMATIVA
82 # Crea una etiqueta de texto con instrucciones
83 info_label = tk.Label(
84     main_frame, # Parent: frame principal
85     text="Click Izquierdo: Flood Fill | Click Derecho: Dibujar paredes", # Texto
86     font=("Arial", 10) # Fuente y tamaño
87 )
88 # Empaqueta la etiqueta con padding vertical de 5 píxeles
89 info_label.pack(pady=5)
90
91 # CONFIGURACIÓN DE EVENTOS DEL MOUSE
92 # Vincula el evento de click izquierdo (<Button-1>) al método on_left_click
93 self.canvas.bind("<Button-1>", self.on_left_click)
94 # Vincula el evento de click derecho (<Button-3>) al método on_right_click
95 self.canvas.bind("<Button-3>", self.on_right_click)
96
97 # Dibuja el grid inicial en el canvas
98 self.draw_grid()
99
```



# Código

```
100 # Método para cambiar el color actual seleccionado
101 def set_color(self, color_index):
102     """Cambia el color actual"""
103     # Actualiza el atributo current_color con el nuevo índice
104     self.current_color = color_index
105
106 # Método para dibujar todo el grid en el canvas
107 def draw_grid(self):
108     """Dibuja el grid en el canvas"""
109     # Borra todos los elementos previos del canvas
110     self.canvas.delete("all")
111
112     # Itera sobre todas las filas del grid
113     for y in range(self.grid_height):
114         # Itera sobre todas las columnas del grid
115         for x in range(self.grid_width):
116             # Obtiene el color correspondiente al valor en grid[y][x]
117             color = self.colors[self.grid[y][x]]
118
119             # Dibuja un rectángulo (celda) en el canvas
120             self.canvas.create_rectangle([
121                 x * self.cell_size, # Coordenada X superior izquierda
122                 y * self.cell_size, # Coordenada Y superior izquierda
123                 (x + 1) * self.cell_size, # Coordenada X inferior derecha
124                 (y + 1) * self.cell_size, # Coordenada Y inferior derecha
125                 fill=color, # Color de relleno
126                 outline="gray" # Color del borde
127             ])
```

# Código

```
129 # Método para convertir coordenadas de pantalla a coordenadas de grid
130 def get_grid_pos(self, event):
131     """Convierte coordenadas del mouse a coordenadas del grid"""
132     # Calcula la coordenada X del grid: posición X del mouse dividido por tamaño de celda
133     x = event.x // self.cell_size
134     # Calcula la coordenada Y del grid: posición Y del mouse dividido por tamaño de celda
135     y = event.y // self.cell_size
136     # Retorna las coordenadas convertidas
137     return x, y
138
139 # ALGORITMO FLOOD FILL (EL CORAZÓN DEL PROGRAMA)
140 def flood_fill(self, x, y, new_color):
141     """Algoritmo Flood Fill con cola"""
142     # Obtiene el color original de la posición inicial
143     old_color = self.grid[y][x]
144     # Si el color nuevo es igual al antiguo, no hace nada (evita loops infinitos)
145     if old_color == new_color:
146         return
147
148     # Crea una cola (deque) e inserta la posición inicial
149     queue = deque([(x, y)])
```

# Código

```
151 # Mientras haya elementos en la cola...
152 while queue:
153     # Saca el primer elemento de la cola (coordenadas actuales)
154     cx, cy = queue.popleft()
155
156     # VERIFICACIÓN DE LÍMITES Y CONDICIONES
157     # Si está fuera de los límites del grid 0 no es el color original...
158     if (cx < 0 or cx >= self.grid_width or # Fuera de límites en X
159         cy < 0 or cy >= self.grid_height or # Fuera de límites en Y
160         self.grid[cy][cx] != old_color):    # No es el color original
161         # Salta esta iteración y continúa con el siguiente elemento
162         continue
163
164     # CAMBIA EL COLOR DE LA CELDA ACTUAL
165     self.grid[cy][cx] = new_color
166
167     # AGREGA LOS 4 VECINOS A LA COLA
168     queue.append((cx + 1, cy)) # Vecino de la derecha
169     queue.append((cx - 1, cy)) # Vecino de la izquierda
170     queue.append((cx, cy + 1)) # Vecino de abajo
171     queue.append((cx, cy - 1)) # Vecino de arriba
```

# Código

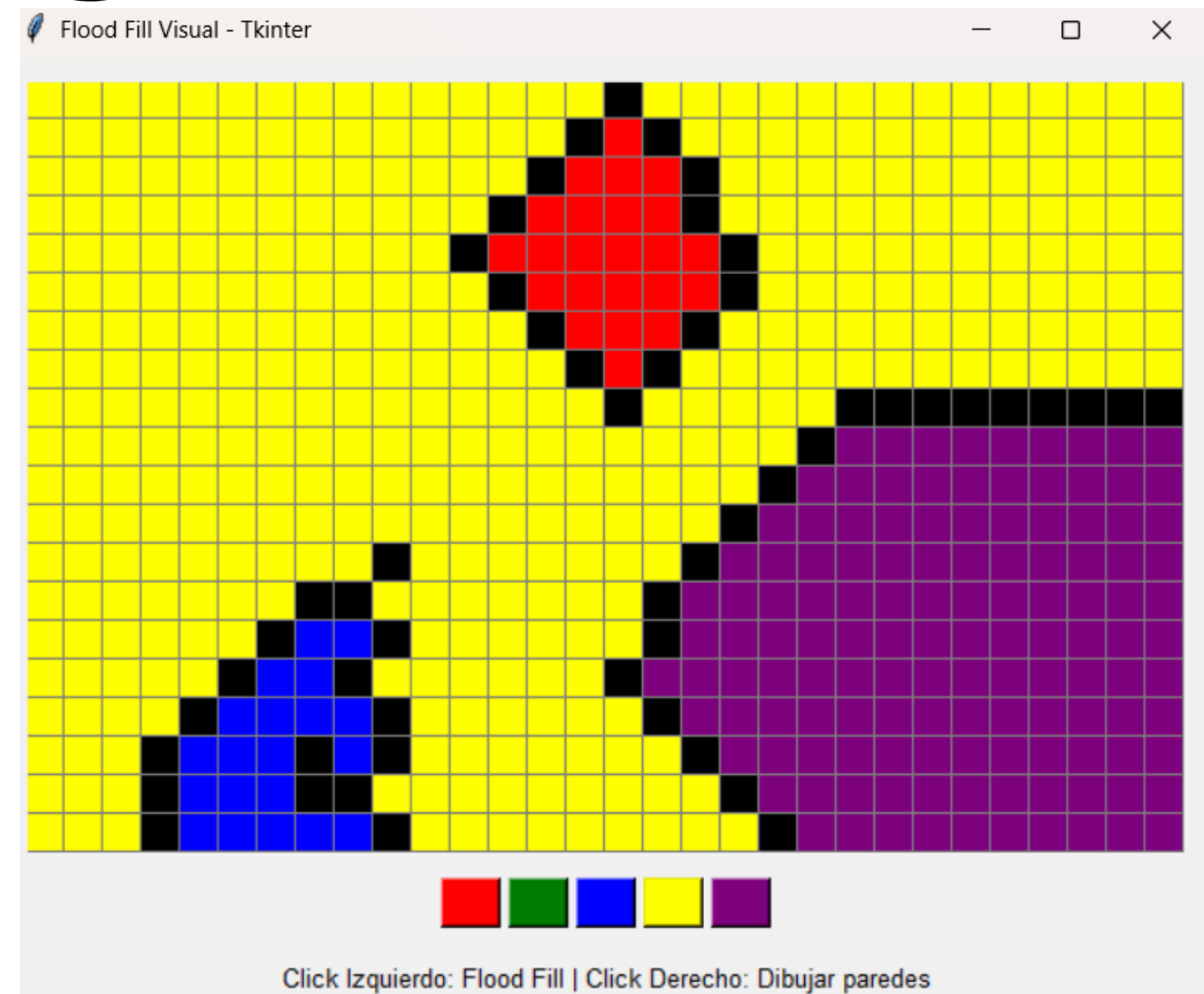
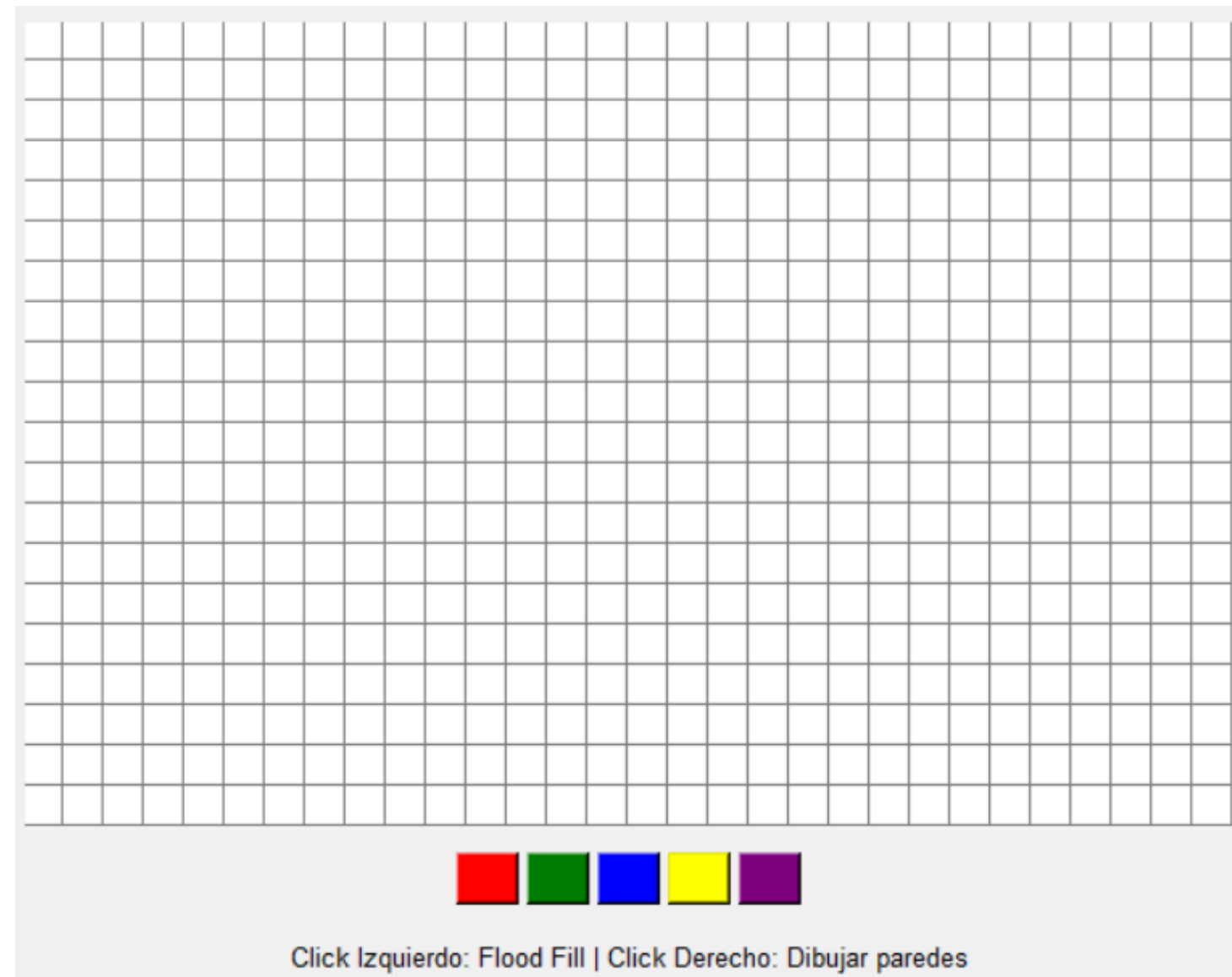
```
173 # Manejador de evento para click izquierdo
174 def on_left_click(self, event):
175     """Maneja click izquierdo - Flood Fill"""
176     # Convierte coordenadas de pantalla a coordenadas de grid
177     x, y = self.get_grid_pos(event)
178     # Ejecuta el algoritmo flood fill desde la posición clickeada
179     self.flood_fill(x, y, self.current_color)
180     # Redibuja el grid para mostrar los cambios
181     self.draw_grid()
182
183 # Manejador de evento para click derecho
184 def on_right_click(self, event):
185     """Maneja click derecho - Dibujar pared"""
186     # Convierte coordenadas de pantalla a coordenadas de grid
187     x, y = self.get_grid_pos(event)
188     # Cambia el valor en el grid a 1 (color negro - pared)
189     self.grid[y][x] = 1 # Negro (pared)
190     # Redibuja el grid para mostrar los cambios
191     self.draw_grid()
```

# Código

```
192
193 # BLOQUE PRINCIPAL DE EJECUCIÓN
194 # Si este archivo se ejecuta directamente (no importado como módulo)...
195 if __name__ == "__main__":
196     # Crea la ventana principal de Tkinter
197     root = tk.Tk()
198     # Crea una instancia de nuestra aplicación, pasando la ventana principal
199     app = FloodFillApp(root)
200     # Inicia el loop principal de la interfaz gráfica (mantiene la ventana abierta)
201     root.mainloop()
```



# Código



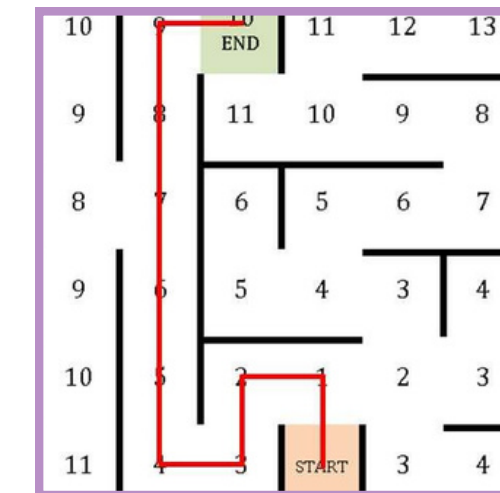
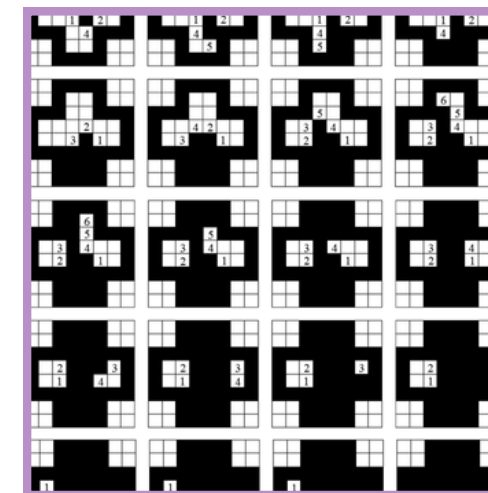
# Aplicaciones

## *en la vida real*

Se aplica en el procesamiento de imágenes, como la herramienta "bote de pintura" en programas de edición, para rellenar áreas de color;

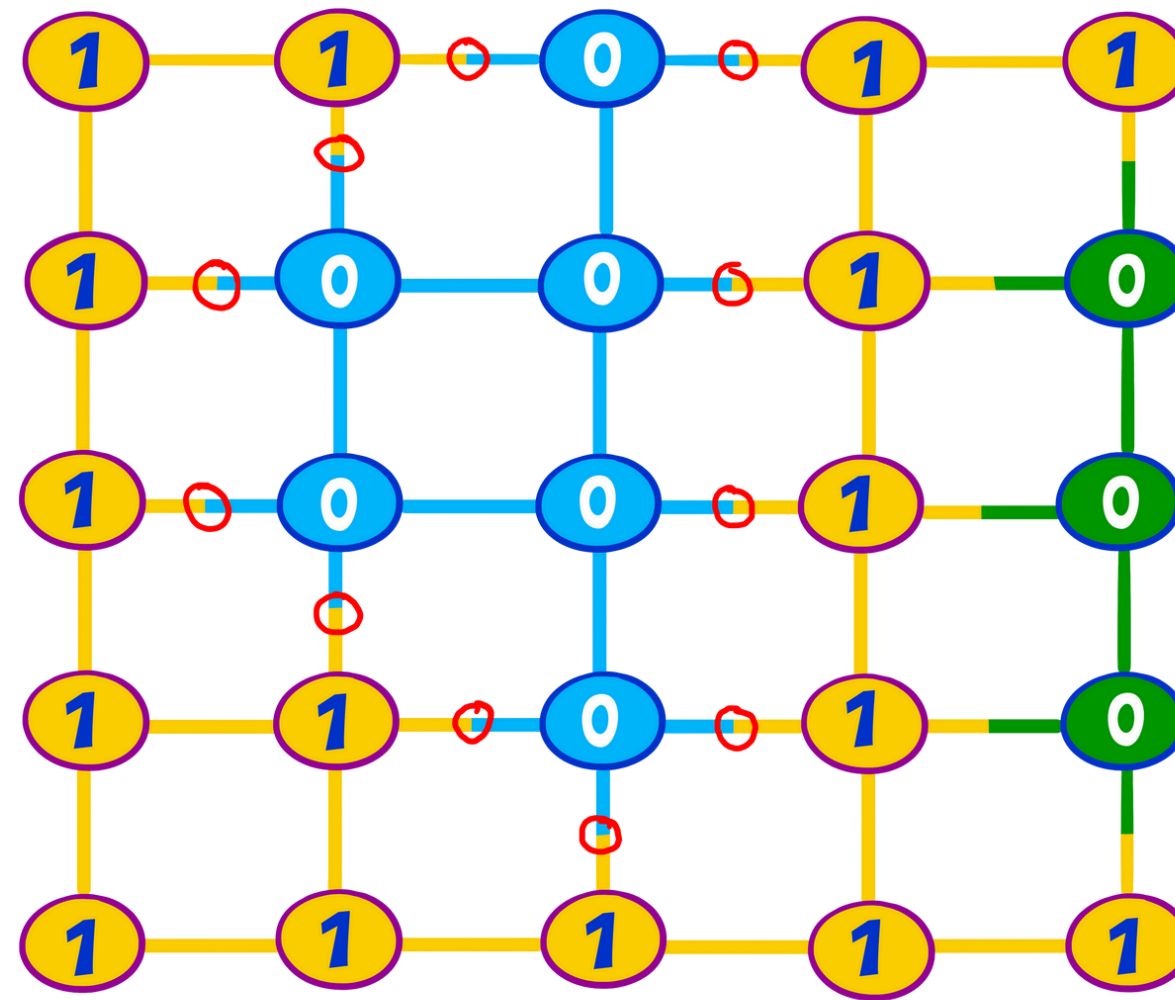
Tiene aplicaciones en videojuegos, como mapeo de zonas accesibles, en juegos tipo "Role-Playing Games" se usa para determinar qué áreas puede recorrer un personaje.

	0	1	2	3	4	5	6	7	8
0	#	#	#	#	#	#	#	#	#
1	#	-	-	-	#	-	-	-	#
2	#	-	-	-	#	-	-	-	#
3	#	-	-	#	-	-	-	-	#
4	#	#	#	-	f	-	#	#	#
5	#	-	-	f	f	#	-	-	#
6	#	-	-	f	#	-	-	-	#
7	#	-	-	f	#	-	-	-	#
8	#	#	#	#	#	#	#	#	#



# Diagrama

FLOOD FILL



# Complejidad *computacional*

## Temporal:

$O(n \cdot m)$  en el peor caso (donde  $n \times m$  es el tamaño de la matriz).

## Espacial:

- Versión recursiva: la pila de llamadas puede crecer hasta  $O(k)$  en el peor caso (riesgo de stack overflow si  $k$  es grande).
- Versión iterativa con pila/cola: también es  $O(k)$  pero más controlado y menos propenso a desbordamientos.

# Comparación con otras técnicas más eficientes

Técnica	Cómo funciona	Ventajas principales	Limitaciones
<b>Flood Fill Recursivo</b>	Explora celda por celda expandiéndose en 4 u 8 direcciones.	<ul style="list-style-type: none"> <li>• Simple, fácil de implementar.</li> <li>• Aplicable en imágenes pequeñas o medianas.</li> </ul>	<ul style="list-style-type: none"> <li>• Riesgo de desbordamiento (stack overflow)</li> <li>• En grandes áreas puede ser lento porque revisa celdas una por una.</li> </ul>
<b>Flood Fill Iterativo</b>	En vez de usar recursión, se maneja una estructura de datos.	<ul style="list-style-type: none"> <li>• Más seguro en áreas grandes.</li> <li>• Mejor control de la memoria.</li> <li>• Evita el límite de recursión.</li> </ul>	Igual de lento que el recursivo
<b>Scanline Fill</b>	En lugar de pixel por pixel, rellena líneas horizontales completas y luego busca vecinos arriba y abajo.	<ul style="list-style-type: none"> <li>• Mucho más eficiente, pues reduce llamadas y operaciones.</li> <li>• Evita recorrer repetidamente celdas ya pintadas.</li> </ul>	Más difícil de programar e implementar.
<b>Union-Find</b>	Trata cada región como un conjunto y agrupa píxeles conectados.	<ul style="list-style-type: none"> <li>• Detección de regiones grandes</li> <li>• Permite consultas rápidas sobre si dos celdas pertenecen a la misma región.</li> </ul>	No rellena directamente, más complejo



# Conclusión

El algoritmo Flood Fill es ideal para rellenar zonas en imágenes o mapas formados por cuadrículas, especialmente cuando la región tiene un color uniforme y no es demasiado grande. Se usa comúnmente en programas de dibujo, en tareas simples de edición de imágenes y en juegos donde se necesita identificar áreas conectadas.

Su uso es limitado y no recomendable para imágenes de ultra alta resolución o áreas enormes debido a su alto consumo de memoria. Su mayor limitación es esa falta de eficiencia en grandes escalas, pero se puede solucionar implementándolo de forma iterativa, usando una cola para manejar los datos. Esto hace que aproveche mejor la memoria y mantenga el mismo funcionamiento sin riesgo de fallos.



**¡Gracias por  
su atención!**