UNIVERSIDAD DE
COSTA RICA

Universidad de Costa Rica

Facultad de Ingeniería
Escuela de Ingeniería Eléctrica

IE0217: Estructuras de Datos Abstractas y Algoritmos para Ingeniería

Grupo 1

Profesor: Juan Carlos Coto Ulate

Proyecto: Flood Fill

Jorge Castrillo Rojas
B41570

II-2020

Índice de contenidos

1. Introducción

Dados dos colores distintos c y c', y un set A de color c encerrado por puntos cuyos colores sean diferentes de c y c', hallar un algoritmo que cambie todos los puntos de A y solo esos a c' [1, p. 28]. Un algoritmo que resuelva el problema anterior es llamado flood fill.

Con respecto a los algoritmos de llenado, su uso práctico más conocido es el de las herramientas de llenado en sofware de edición de gráficos.

Es difícil saber quién implementó el primer algoritmo de fill, pero destaca Richard Shoup en la decáda de los 70 con su computador SuperPaint [2, p. 129] en cuyo menú se puede apreciar una herramienta de llenado.

Figura 1: Sistema SuperPaint, tomado de [3]

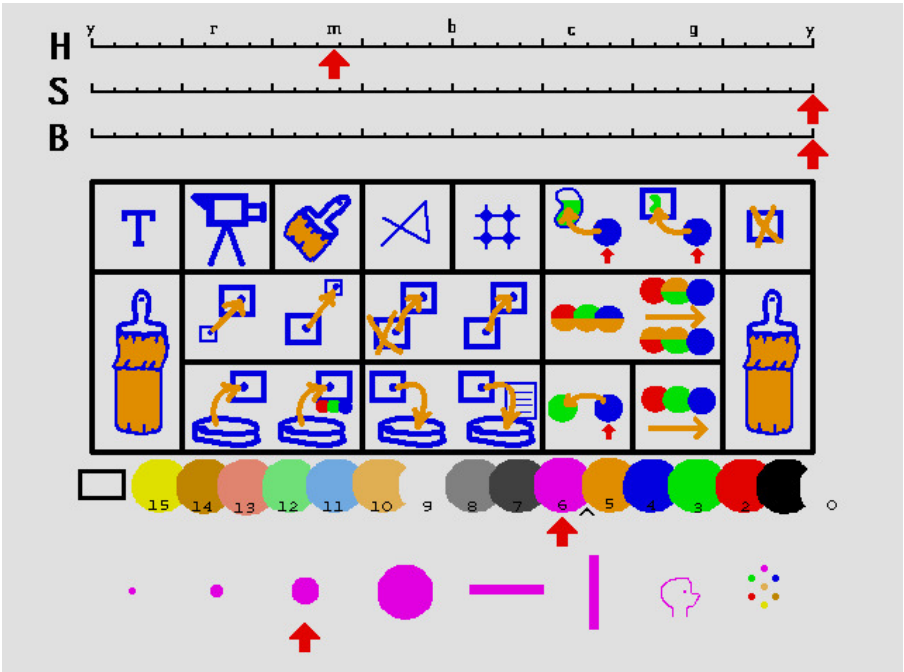

Figura 2: Menú de SuperPaint. La herramienta de llenado está en la primera fila, segunda opción desde la derecha. Tomado de [3]

El siguiente gran paso en el desarrollo de los algoritmos de llenao vino de la mano de Alvy Ray Smith en 1979, con el algoritmo de tint fill que funciona con imágenes difusas, con las que el flood fill tiene dificultades. Desde entonces se han desarrollado algoritmos más sofisticados que se usan hoy en día [2, p. 129-130].

2. Discusión

2.1. El algoritmo

En términos mśs comunes, consiste en llenar una imagen (por simplicidad la voy a llamar "imagen", pero estamos hablando de un arreglo bidimensional como una matriz), dado un nodo de partida (llamado también *semilla* o *seed*), un color de reemplazo y un color a reemplazar.

Dados estos parámetros, a partir del nodo de entrada se comienza nodo por nodo a cambiar el color inicial por el color de destino, hasta que todos los nodos interconectados entre sí del mismo color cambien al color de destino.

El algoritmo más básico reemplaza los nodos adyacentes en 4 direcciones, (x+1), (x-1), (y+1) y (y-1), es llamado *4-way*. Hay implementaciones llamadas *8-way*. que además de los anteriores, reemplazan los nodos (x+1, y+1), (x+1, y-1), (x-1, y+1) y (x-1, y-1).

Existen varias formas de implementar el algoritmo, lo más común es hacerlo de forma recursiva. El problema con la recursión es que algunos pixeles van a ser evaluados más de una vez, y los recursos necesarios pueden crashear un programa si la imagen es muy grande [4]. Algunas implementaciones usan un queue en donde se van almacenando los pixeles visitados. Otras implementaciones rellenan fila por fila para lidiar con las recursiones y no hacer uso intenso del stack.

El algoritmo acaba cuando no hay más nodos que puedan ser reemplazados.

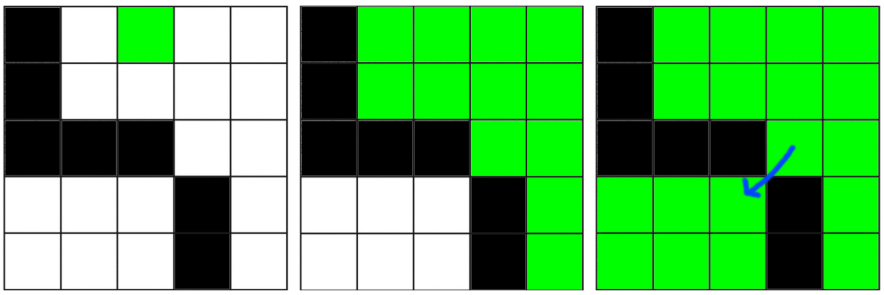

Figura 3: De izquierda a derecha: imagen y nodo inicial. 4-way flood fill. 8-way flood fill

En más detalle, el algoritmo hace 4 cosas:

1. Revisa si el nodo a pintar es válido. Esto quiere decir que el nodo esté dentro de los límites de la imagen y que sea del color que se quiere reemplazar.
2. Si el nodo es válido, le cambia el color y examina el siguiente nodo.
3. Si el nodo no es válido, hace return y revisa el nodo anterior en las otras direcciones en busca de validez.
4. Una vez todos los nodos válidos han sido evaluados en las cuatro direcciones, el algoritmo termina.

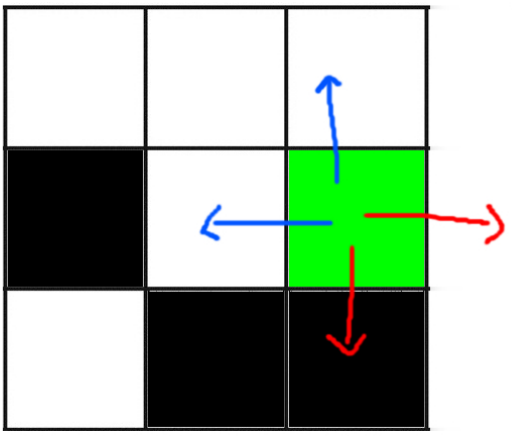

Figura 4: Azul: Nodos válidos. Rojo: Nodos no válidos

2.2. Usos

Como se mencionó en la introducción, es utilizado por las herramientas de relleno en programas como Paint y Photoshop. En este caso los nodos son los pixeles de la zona a la que le queremos cambiar el color.

El algoritmo es utilizado en el famoso juego buscaminas. En la figura 5, se puede apreciar como se puede expandir el área vacía (beige), y como está encerrado por las casillas con números que representan la cantidad de minas que hay alrededor. Es el mismo principio solo que no podemos ver lo que hay escondido.

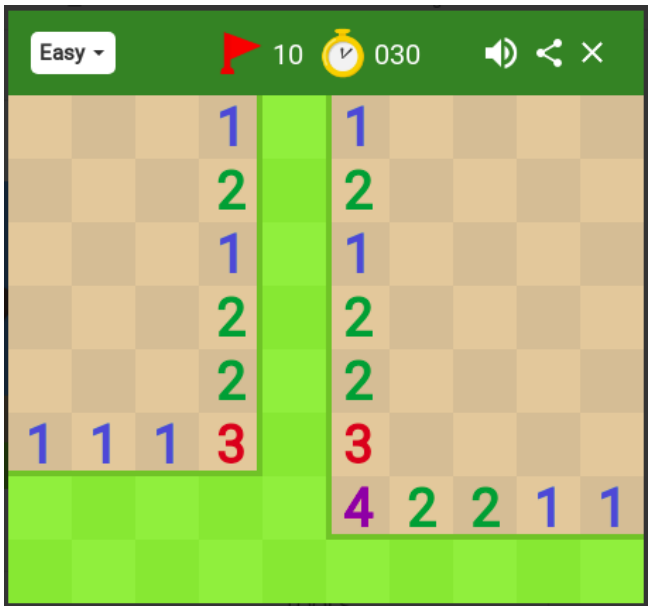

Figura 5: Buscaminas de Google

Otro juego en el que el algoritmo es utilizado es en Go. En este caso, al rodear perfectamente (ocupar sus libertades) una o varias piedras del color opuesto, estas son removidas del tablero, como se puede apreciar en la figura 6. En este caso, se usa flood fill para remover las piezas del color capturado y vaciar el espacio.

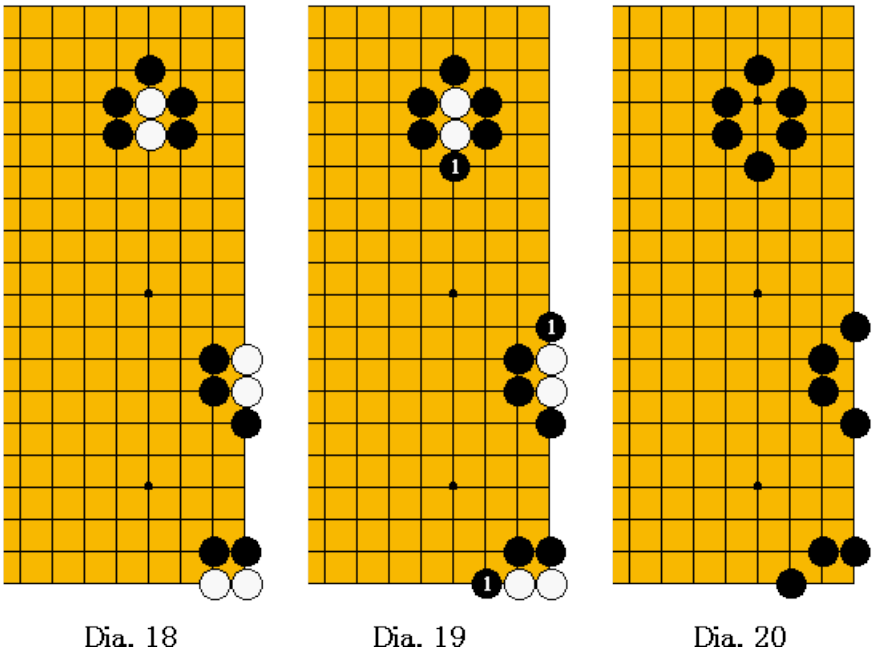

Figura 6: Proceso de captura de Go, tomado de [5]

Es también muy usado en robots para encontrar el camino más corto entre dos puntos de un laberintos aunque es un poco lento [6]. De igual manera se puede encontrar el o los caminos más cortos entre dos puntos (en un área delimitada), pero existen algoritmos más eficientes como el de Djikstra o el de Floyd. Tal vez la ventaja de flood fill en esa situación es su fácil implementación.

2.3. Complejidad

Si hacemos una recursión N veces, la complejidad es O(N). En el peor caso, para el flood fill recursivo todos los pixeles son evaluados 4 veces (una vez para cada dirección). Esto es entonces O(4N)=O(N) donde N es el número total de pixeles. También se puede definir en términos del tamaño de la matriz/imagen, en este caso si se tienen N pixeles de alto y M pixeles de largo, se tiene que la complejidad es O(4NM)=O(NM), y si es un cuadrado $O(N^2)$. Como se vio en clase, O(4N)=O(N) porque no afecta significativamente el resultado final.

2.4. Código

Se implementó un flood fill 4-way recursivo. En este caso, se usó una matriz de números que hace las veces de imagen, en la que cada número corresponde a un pixel.

```
def floodfill(imagen, x, y, color1, color2):
    #condiciones de (in)validez (return si se cumplen)
    if (x>=len(imagen) or y>=len(imagen[0]) or x<0 or y<0 or imagen[x][y]==color2 or imagen[x
    ][y]!=color1):
        return

    imagen[x][y]=color2 #reemplazo el color si es valido

    #recursion
    floodfill(imagen, x+1, y, color1, color2)
    floodfill(imagen, x-1, y, color1, color2)
    floodfill(imagen, x, y+1, color1, color2)
    floodfill(imagen, x, y-1, color1, color2)

    return imagen
```

La función floodfill recibe los siguientes parámetros:

- imagen: la matriz
- x: la coordenada en x del nodo de inicio
- y: la coordenada en y del nodo de inicio
- color1: el número que deseo reemplazar
- color2: el número por el que voy a reemplazar color1

El if se fija con un OR si un nodo está fuera de la matriz (*len(matrix[0])* se puede usar para hallar el tamaño de las columnas). O si sl nodo está dentro de la matriz, se fija si ya está pintado. O se fija si es de un color que no quiero pintar. Si alguna de las condiciones anteriores se cumple, hace return.

¿Return a qué? Para responder a esta pregunta se analiza el resto del código. En caso de que el if no se cumpla, osea que el nodo es válido, se procede normalmente y se pinta el nodo, Una vez el nodo válido fue pintado, comienza la recursión. Se llama a la función pero esta vez un pixel hacia la derecha (x+1, y). Si el nodo es válido, el if no se cumple, el nodo se pinta y se continua con el nodo (x+2, y). Pero si el if se cumple, se hace return y se intenta esta vez con (x-1, y). Después de sucesivas recursiones en otros nodos, se terminará examinando el nodo que está debajo del original (x, y-1). La función entonces devuelve la matriz modificada.

```
def floodfill_print(original, x, y, n):
    print('Ejemplo #',n)
    print('La matriz original es:')
    for item in original:
        print(item)
    ff = floodfill(original, x, y, 0, 1)
    print('La matriz despues de aplicar flood fill, con coordenadas de inicio xy =',x,y,'es:')
    for item in ff:
        print(item)
    print('-----------------------------------')
```

Esta función como su nombre lo indica simplemente imprime la matriz original y la matriz modificada en la terminal.

2.5. Ejemplos

2.5.1. Ejemplo base

Nodo inicial (1,1), color1=0, color2=1. Se puede apreciar en la figura 7 como cambió los 0 por 1 excepto los ceros de la zona inferior derecha, que están encerrados.

```
Ejemplo # 1
La matriz original es:
[2, 0, 0, 0, 0]
[2, 0, 0, 0, 2]
[0, 0, 2, 2, 0]
[0, 2, 0, 0, 0]
La matriz despues de aplicar flood fill, con coordenadas de inicio xy = 1 1 es:
[2, 1, 1, 1, 1]
[2, 1, 1, 1, 2]
[1, 1, 2, 2, 0]
[1, 2, 0, 0, 0]
---------------------------------
```

Figura 7: Ejemplo básico

2.5.2. Distinto nodo de entrada

En este caso el nodo inicial es (2,4), color1=0, color2=1. Se puede apreciar en la figura 8 como esta vez cambiaron los ceros de la esquina inferior derecha.

```
Ejemplo # 2
La matriz original es:
[2, 0, 2, 0, 0]
[2, 0, 0, 0, 2]
[0, 2, 2, 2, 0]
[0, 2, 0, 0, 0]
La matriz despues de aplicar flood fill, con coordenadas de inicio xy = 2 4 es:
[2, 0, 2, 0, 0]
[2, 0, 0, 0, 2]
[0, 2, 2, 2, 1]
[0, 2, 1, 1, 1]
---------------------------------
```

Figura 8: Diferente nodo de inicio

2.5.3. Distintos colores

En este caso, el nodo inicial es (1,1), color1=0, per el color es color2=5. Se puede apreciar en la figura 9 como los ceros se cambiaron por cincos.

```
Ejemplo # 3
La matriz original es:
[2, 0, 2, 0, 0]
[2, 0, 0, 0, 2]
[0, 2, 2, 2, 0]
[0, 2, 0, 0, 0]
La matriz despues de aplicar flood fill, con coordenadas de inicio xy = 1 1 es:
[2, 5, 2, 5, 5]
[2, 5, 5, 5, 2]
[0, 2, 2, 2, 0]
[0, 2, 0, 0, 0]
---------------------------------
```

Figura 9: Diferentes colores

2.5.4. Distinta matriz

En el caso de la figura 10, el argumento que cambia es la matriz de entrada. El nodo de inicio es (1,2), color1=0, color2=1.

```
Ejemplo # 4
La matriz original es:
[2, 0, 2, 2, 0, 2, 2, 0]
[2, 0, 0, 0, 0, 2, 0, 2]
[2, 2, 2, 2, 0, 2, 0, 0]
[2, 2, 2, 2, 0, 0, 0, 2]
[2, 2, 2, 2, 0, 2, 2, 0]
[2, 2, 2, 0, 0, 2, 2, 0]
[2, 2, 0, 0, 0, 2, 0, 0]
La matriz despues de aplicar flood fill, con coordenadas de inicio xy = 1 2 es:
[2, 1, 2, 2, 1, 2, 2, 0]
[2, 1, 1, 1, 1, 2, 1, 2]
[2, 2, 2, 2, 1, 2, 1, 1]
[2, 2, 2, 2, 1, 1, 1, 2]
[2, 2, 2, 2, 1, 2, 2, 0]
[2, 2, 2, 1, 1, 2, 2, 0]
[2, 2, 1, 1, 1, 2, 0, 0]
--------------------------------
```

Figura 10: Diferente matriz

3. Conclusiones

El algoritmo de flood fill es muy sencillo de implementar y tiene una gran variedad aplicaciones, una particularmente útil en software de edición gráfica con la herramienta de relleno, aunque también puede ser encontrada en videojuegos y hasta en robots.

Esta facilidad de implementación viene con un costo sin embargo, y es que utiliza muchos recursos por ser recursivo. Teniendo esto en cuenta, puede ser modificado para que no sea tan intensivo en el stack, y dependiendo de la imagen que se quiera editar se pueden hacer cambios para tener un mejor manejo de memoria.

4. Referencias

[1] M. Agoston, *Computer Graphics and Geometric Modelling*, ser. Computer Graphics and Geometric Modeling. Springer, 2005, p. 28. [Online]. Available: https://books.google.co.cr/books?id=fGX8yC-4vXUC

[2] A. Glassner, *Andrew Glassner's Other Notebook: Further Recreations in Computer Graphics*, 1st ed. A K Peters/CRC Press, 2002.

[3] R. Shoup, *The SuperPaint System (1973-1979)*, (accessed December 09, 2020). [Online]. Available: https://web.archive.org/web/20150924090732/http://www.rgshoup.com/prof/SuperPaint/index.html

[4] J. Lawhead, *Learning Geospatial Analysis with Python: Understand GIS fundamentals and perform remote sensing data analysis using Python 3.7, 3rd Edition.* Packt Publishing, 2019. [Online]. Available: https://books.google.co.cr/books?id=eBWzDwAAQBAJ

[5] Kiseido, *HOW TO PLAY GO*, (accessed December 13, 2020). [Online]. Available: https://www.kiseidopublishing.com/ff.htm

[6] H. Dang, J. Song, and Q. Guo, "An efficient algorithm for robot maze-solving," in *2010 Second International Conference on Intelligent Human-Machine Systems and Cybernetics*, vol. 2, 2010, pp. 79–82.