Flow Free

Jorge Luis Esposito Albornoz[1] Juan Sebastián Herrera Guaitero[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{jesposito,jsebastianherrera}@javeriana.edu.co

24 de noviembre de 2022

Resumen

En este documento se presenta la documentación correspondiente a la implementación desarrollada para realizar el videojuego Flow Free. **Palabras clave:** Flow, puzzle.

Índice

1. Formalización del problema

Un rompecabezas o puzzle es un juego cuyo objetivo es formar una figura combinando correctamente las partes de esta, que se encuentran en distintos pedazos o en nuestro caso en cuadrícula que se compone de diferentes "Flows".

1.1. Definición del problema

El juego presenta rompecabezas de enlaces numéricos. Cada rompecabezas tiene una cuadrícula de cuadrados con pares de puntos de colores que ocupan algunos de los cuadrados. El objetivo es conectar puntos del mismo color dibujando "tuberías" entre ellos, de forma que toda la cuadrícula esté ocupada por tuberías. Sin embargo, las tuberías no pueden cruzarse. La dificultad viene determinada principalmente por el tamaño de la cuadrícula.

Flow free se define a partir de:

1. Dada una matriz de M de elementos $a \in \mathbb{Z}$, donde cada elemento representa un color en la cuadrícula.

Permita a través de la consola completar los "flows" o "tuberías" teniendo en cuenta las reglas anteriormente descritas.

- **Entradas:**
  - $M^{nxn} \mid \forall M_{ij} \in \mathbb{Z} \ \wedge \ \ni M_{ij} = M_{i+aj+b}$
- **Salida:**
  - $M^{nxn} \mid \forall M_{ij}, M_{ij} \neq 0$

1.2. Requerimientos

- El sistema debe dejar en claro al usuario los distintos puntos a conectar mediante distintos colores y un código identificador.
- El sistema debe generar diversas posibilidades de mapas aleatoriamente.
- El sistema debe permitir la interacción con el usuario por medio de entradas del teclado.
- El nivel debe finalizar automáticamente cuando el usuario complete todos los caminos.

1.3. Restricciones

- La terminal donde se ejecute el programa debe ocupar como mínimo la mitad de la pantalla.
- La terminal debe soportar 256 colores.
- Se deben completar todos los puntos para poder completar el nivel.
- Los caminos no se pueden interceptar.
- Se debe llenar todas las casillas del mapa.
- El usuario que ejecute el programa debe tener instalado Python 3 y la librería Curses.

2. Idea de solución

La idea de la solución se planteó utilizando CSP, conocida en español como PSR, que es una simple, pero poderosa idea utilizada en inteligencia artificial.

2.1. Problemas de Satisfacción de Restricciones (CSP)

Componentes del estado:

1. **Variables** Valores de A-Z incluyendo "_" que representa una espacio sin colorear.
2. **Dominios:** Valores posibles para las variables, en nuestro caso los posibles valores son los colores que se representan como letras de la A-Z.
3. **Restricciones:**
   - Si es un endpoint(Letras en mayúsculas)
     - No puede haber más de uno del mismo color conectado(Solo una pareja por color)
     - No hay caminos disponibles para conectarse con el endpoint
   - Si no es endpoint(Letras en minúscula)
     - No hay caminos disponibles para moverse

Objetivo: Encontrar un estado que satisface las restricciones (Asignación de valores a las variables, que satisfaga las restricciones)

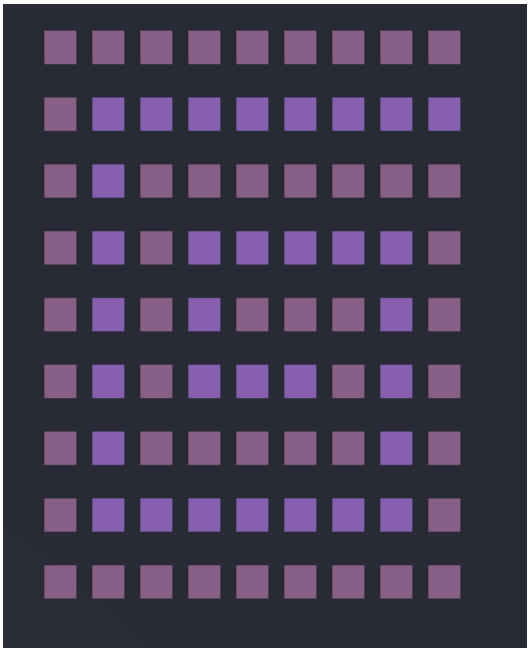

Figura 1: Estado cumplido

Nota: Los valores se manejan como letras, sin embargo, cuando se imprimen se toman todas las letras como minúsculas en su representación ASCII que representa un color en el mapa.

2.2. Algoritmo de solucion

Algoritmo 1 getNext

```
1: function getNext(self,x,y)
2:     if x = self.rows − 1 ∧ y = self.cols − 1 then
3:         return None
4:     end if
5:     if x = self.rows − 1 then
6:         return [0, y + 1]
7:     else
8:         return [x + 1, y]
9:     end if
10: end function
```

Algoritmo 2 CheckConstraints

```
1: function checkConstraints(self,x,y,nbors)
2:     let valid be True
3:     nbors+ ← self.graph[x][y]
4:     for j to nbors do
5:         if j.symbol = "_" then
6:             continue
7:         end if
8:         cnbors ← findNeighbors(j.x, j.y)
9:         if j.symbol.isupper() then
10:            symbolCount ← include(j.symbol.lower(), cnbors)
11:            blankCount ← include("_", cnbors)
12:            if symbolCount > 1 then
13:                valid ← False
14:            end if
15:            if blankCount = 0 ∧ symbolCount ≠ 1 then
16:                valid ← False
17:            end if
18:        else
19:            symbolCount ← include(j.symbol, cnbors)
20:            symbolCount+ ← include(j.symbol.upper(), cnbors)
21:            blankCount ← include("_", cnbors)
22:        end if
23:        if symbolCount > 2 then
24:            valid ← False
25:        end if
26:        if blankCount < 1 ∧ symbolCount = 1 then
27:            valid ← False
28:        end if
29:        if blankCount < 2 ∧ symbolCount = 0 then
30:            valid ← False
31:        end if
32:    end for
33:    return valid
34: end function
```

Algoritmo 3 findNeighbors

```
1: function FINDNEIGHBORS(self,x,y)
2:     let nbors be a list
3:     if x > 0 then
4:         nbors+ ← self.graph[x−1][y]
5:     end if
6:     if y > 0 then
7:         nbors+ ← self.graph[x][y−1]
8:     end if
9:     if x < self.rows − 1 then
10:        nbors+ ← self.graph[x+1][y]
11:    end if
12:    if y < self.cols − 1 then
13:        nbors+ ← self.graph[x][y+1]
14:    end if
15:    return nbors
16: end function
```

Algoritmo 4 solver

```
1: procedure SOLVER(self,x,y)
2:     let done be False
3:     nextSquare+ ← GETNEXT(x, y)
4:     nbors ← FINDNEIGHBORS(x, y)
5:     if self.graph_ij.symbol ≠ "_" ∧ nextSquare ≠ None then
6:         done ← SOLVER(nextSquare_0, nextSquare_1)
7:     else
8:         for i to self.options do
9:             self.graph_xy.symbol ← i
10:            self.count+ ← 1
11:            valid ← CHECKCONSTRAINTS(x, y, nbors)
12:            if valid then
13:                if nextSquare ≠ None then
14:                    return True
15:                else
16:                    done ← SOLVER(nextSquare_0, nextSquare_1)
17:                    if done then
18:                        return done
19:                    end if
20:                end if
21:            end if
22:        end for
23:        if ≠ done then
24:            self.graph_xy.symbol ← "_"
25:        end if
26:    end if
27:    return done
28: end procedure
```

2.3. Análisis de complejidad

Resolver un problema de satisfacción de restricciones con un algoritmo de búsqueda, en nuestro caso backtracking, es exponencial en el peor de los casos $O(d^n)$, donde n y d son el número de variables y el tamaño máximo del dominio.

3. Implementación

3.1. Modelado de datos

En esta sección se presetará las estructuras de datos utilizadas para la solución.

3.1.1. Diagramas de clases

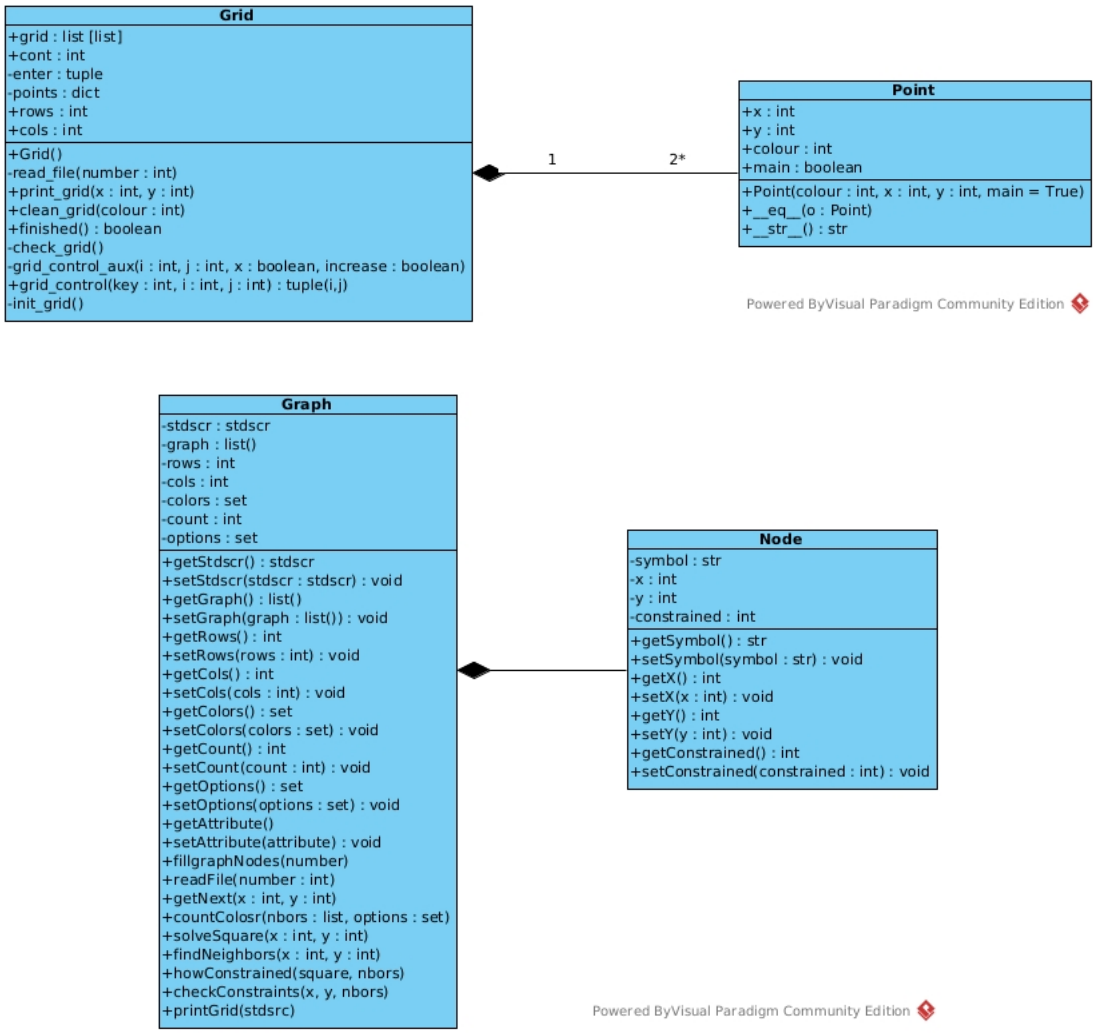

Figura 2: Diagrama de clases

3.1.2. Diagrama de secuencia

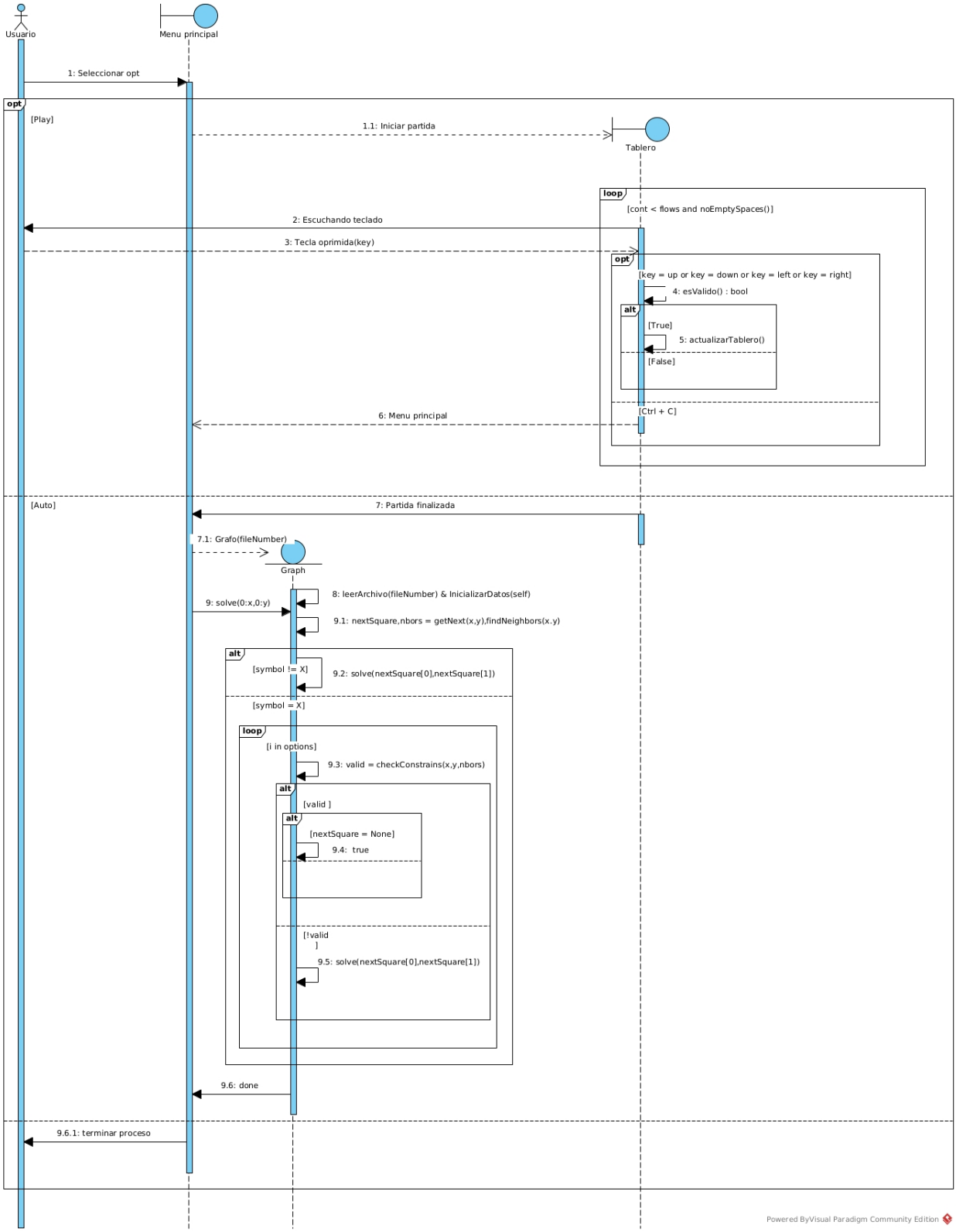

Figura 3: Diagrama de secuencia

4. Cómo jugar?

4.0.1. Keybinding

Tecla/Teclas	Funciones
Flecha izquierda	Moverse una columna hacia la izquierda
Flecha derecha	Moverse una columna hacia la derecha
Flecha arriba	Moverse una fila hacia arriba
Flecha abajo	Moverse una fila hacia abajo
Enter	Seleccionar el color actual(solo funciona con puntos principales)
Ctrl + c	Enviar un signal para volver al menú principal)

4.0.2. Protocolo

1. Diríjase a la ruta en donde se encuentra el proyecto con nombre FlowFree.
2. Asegúrese de tener Python 3 instalado y contar con la siguiente librería:
   - **Linux o macOS:**
     - El paquete ya viene instalado por default.
   - **Windows:**
     - windows-curses

   En caso de no tenerla, instalarla de la siguiente forma:

   pip install "nombre_paquete"

   Nota: La terminal utilizada para correr el programa debe soportar 256 colores.
3. Ejecutar el siguiente comando:

   *python*3 *main.py*

Figura 4: Menú principal

Moverse de acuerdo a las Keybindings para elegir una opción con la tecla "enter".

- **Play**: Se generá un número aleatorio para elegir un archivo del 1-14.

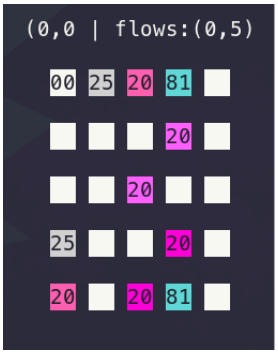

Figura 5: Mapa

- En la parte superior izquierda de la **Figura 3** se muestra las coordenadas actuales en donde se encuentra $(0, 0)$.
- En la parte superior derecha de la **Figura 3** se muestra el número de "flows" que han sido completados.

1. Al moverse de acuerdo a las Keybindings si se para en la posición de algún punto que este pintado de un color diferente al blanco y da "enter" el color se selecciona.

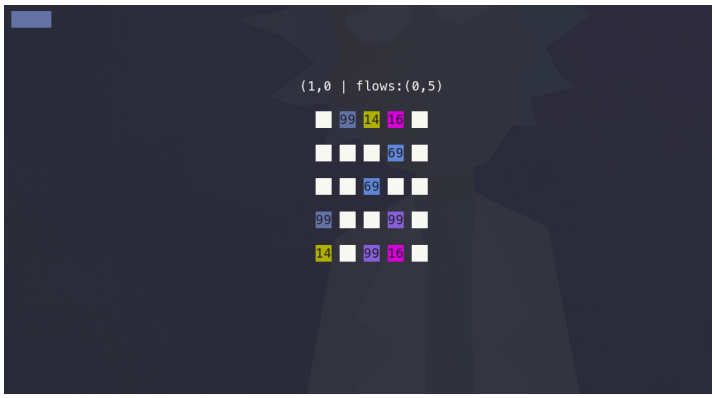

Figura 6: Color elegido

2. Una véz elegido el color en la esquina superior izquierda de la pantalla de acuerdo a la **Figura 4** se mostrará el color seleccionado.

   2.1 El usuario va a poder moverse de acuerdo a las Keybidings y se pintarán los cuadros en la cuadrícula con el color seleccionado.

   2.2 Para completar un "flow", el usuario debe moverse de acuerdo a las Keybindings y conectar el "flow" con el punto destino, una véz lo complete el programa, inmediatamente deselecciona el color previamente elegido.

   2.3 En caso de querer deseleccionar un color, se debe oprimir "enter" en un cuadro que no sea un punto de conexión a un flow.

3. El juego solo acaba cuando todos los "flows" estén completados y la cuadrícula no tenga ningún espacio vacío.
4. Tenga en cuenta que usted puede volver al menú principal en cualquier momento con la combinación Ctrl + c.

- **Auto**:

1. Se genera un mapa aleatoriamente eligiendo un archivo de 1-14.

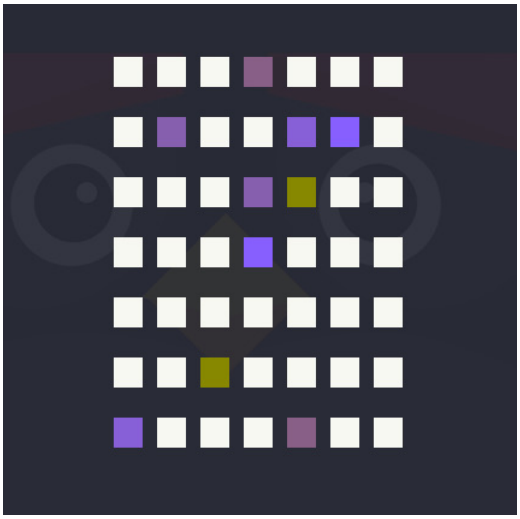

Figura 7: Mapa auto

2. Presionar cualquier tecla para que de la solución.

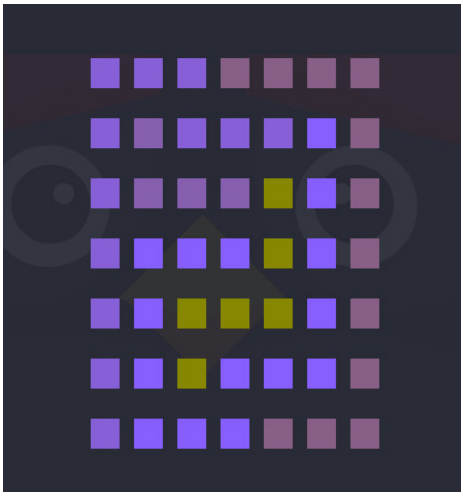

Figura 8: Mapa solucionado

Nota: No se colocó un sleep para ver el paso a paso, puesto que podía demorar más de 100 segundos para ver el proceso.

- **Exit**: El proceso se termina.