PROYECTO FLOW: ENTREGA FINAL

Análisis de Algoritmos
Departamento de Ingeniería de Sistemas
Pontificia Universidad Javeriana

Realizado por
Alejandro Morales Contreras

24 de noviembre de 2022

Índice

1. Introducción

El presente pretende servir como documento de diseño y documentación sobre el proyecto del curso de análisis de algoritmos. El proyecto consiste en la implementación de un algoritmo que juegue "Flow". Este juego consiste en resolver rompecabezas *numberlink*. Cada rompecabezas consiste en conectar todos los pares de puntos del mismo color dibujando tuberías no intersectables hasta que todo el tablero está ocupado.

Para la entrega final, se propone la implementación de un algoritmo capaz de resolver el juego. Para esta entrega, se propone un algoritmo de aproximación. En este documento se presenta entonces el diseño de la interfaz (sección 2), se muestran detalles de la implementación (sección 3), se presenta todo el proceso de análisis de algoritmos (sección 4) y una breve documentación (sección 5).

2. Diseño

2.1. Arquitectura del Sistema

Para implementar el juego mediante una interfaz gráfica de usuario, se hace necesario definir una arquitectura para diseñar el Sistema. En este caso, se va a utilizar el patrón MVC junto con un mediador de eventos para comunicar los componentes entre sí. En la figura 1 se presenta la arquitectura del Sistema.

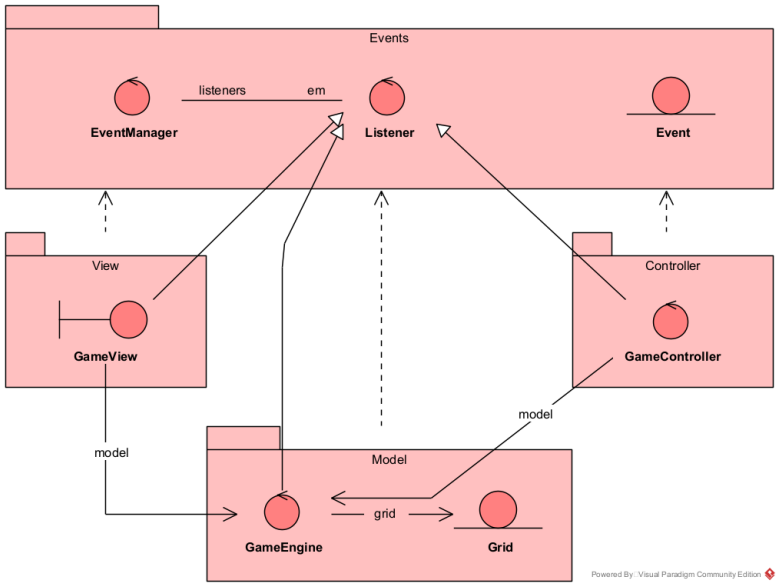

Figura 1: Arquitectura del Sistema

A continuación se presenta una breve descripción de cada uno de los componentes que hacen parte de esta arquitectura.

2.1.1. Model

El modelo guarda toda la lógica y estructura del estado del juego. Para lograr esto, se define un *GameEngine* que se encarga de almacenar el estado del juego, así como modificarlo acorde a los eventos de modificación del modelo. Nótese que en el modelo se tiene la clase *Grid*, la cual tiene toda la lógica de cómo funciona el juego.

2.1.2. View

La vista se encarga de representar el estado actual del juego gráficamente en la pantalla. *GameView* tiene toda la lógica asociada a pintar el modelo cada vez que se recibe un evento de actualización de pantalla.

2.1.3. Controller

El controlador se encarga de recibir todas las interacciones del usuario y llevarlas a modificar el modelo. *GameController* tiene la lógica asociada a recibir los eventos de interacción del usuario y transformarlos en eventos de modificación del modelo.

2.1.4. Events

Mediante los eventos se coordina toda la comunicación entre cada uno de los componentes. El *EventManager* se encarga de manejar la cola de eventos y notificar a todos sus *listeners asociados*. El *Listener* es notificado cada vez que se genera un evento, y también tiene la posibilidad de colocar eventos en la cola. Nótese que *GameEngine*, *GameView* y *GameController* heredan de este.

2.2. Modelo de interacción del Sistema

La interacción del Sistema y sus componentes está mediada por el manejador de eventos. En general, estos son los eventos de interacción importantes que analizar:

- Inicialización, generado al ejecutar por primera vez el juego y el cual se encarga de inicializar todos los componentes del Sistema.
- Tick, tick del reloj el cual se encarga de avisar a la Vista que debe actualizarse.
- Interacción del usuario, generado cuando el usuario interactúa con el Sistema y modifica el estado del modelo.

A continuación se presenta una breve representación de cada una de estas interacciones.

2.2.1. Inicialización y Tick

En la figura 2 se presenta el modelo de interacción de inicialización y tick del reloj. La inicialización es dirigida por el modelo, el cual crea el evento de inicialización para los otros componentes. Después, entra en un ciclo hasta que el juego acabe que genera continuamente los ticks. La Vista utiliza estos para recuperar el estado del modelo y actualizarse.

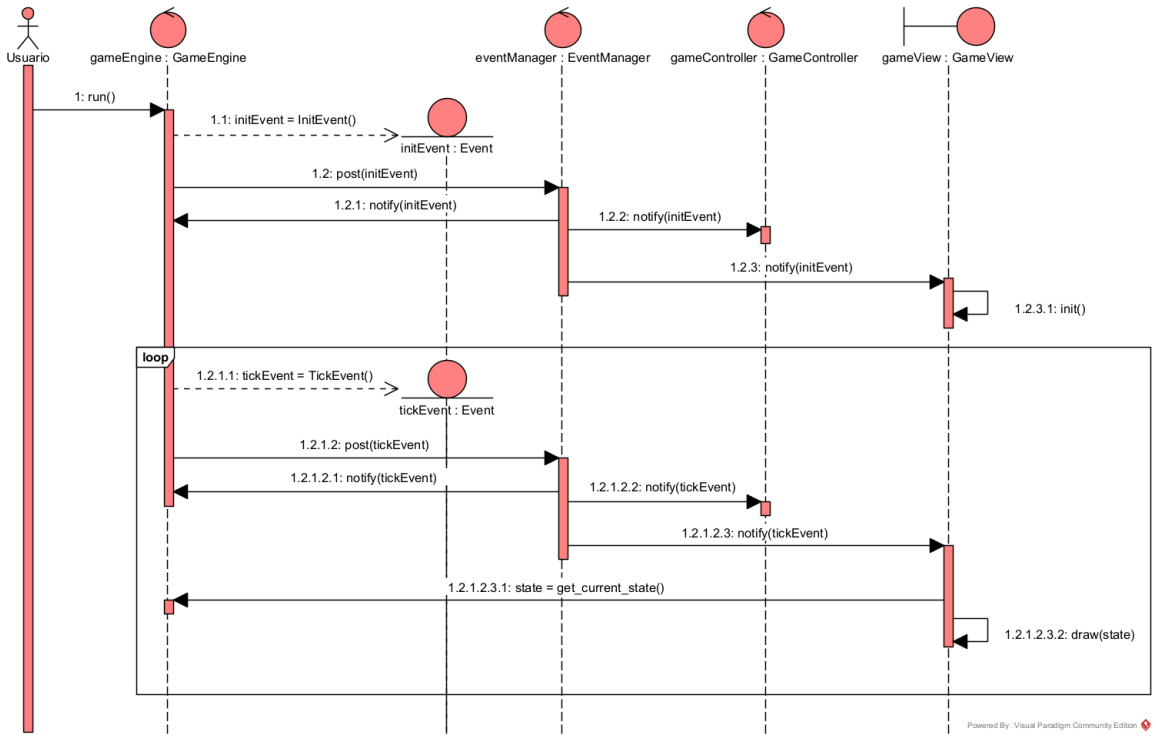

Figura 2: Interacción de inicialización y tick del reloj

2.2.2. Interacción del usuario

En la figura 3 se presenta el modelo de interacción del usuario. En general, el usuario interactúa con el controlador, el cual se encarga de traducir esta interacción en un evento de modificación del modelo (estado) del Sistema.

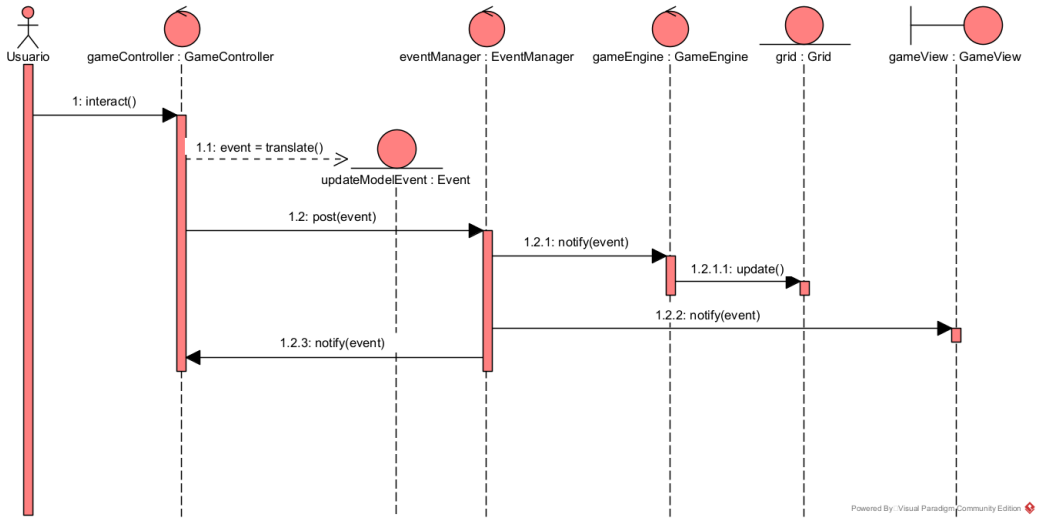

Figura 3: Interacción del usuario

2.3. Lógica del juego

Como se mencionó en la sección 2.1.1, la lógica de cómo funciona el juego está mediada por el *Grid*, el cual hace parte del estado del modelo. Como se vió en la sección 2.2.2, la modificación del modelo (y por ende, del estado del juego) está dirigida por las interacciones del usuario.

2.3.1. Representación del tablero

Para empezar, se define el prototipo que modela el *Grid*, presentado en la figura 4.

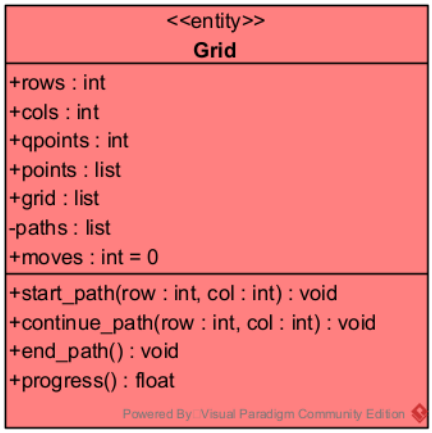

Figura 4: Prototipo de *Grid*

El tablero se modela a partir de una cantidad de filas (*rows*), una cantidad de columnas (*cols*), una cantidad de puntos (*qpoints*) y el posicionamiento (row, col) de cada par de puntos (*points*). A partir de esta información, es posible construir el tablero (*grid*): una matriz de $rows \times cols$. También se representan los caminos actuales (*paths*): un arreglo de (row, col) para cada punto disponible.

Cada celda del tablero es representada con un *estado*, acorde a su color (un número entero) y su interacción (o no) con las demás celdas (dos números enteros de posicionamiento). Para entender el posicionamiento de una celda, véase la figura 5.

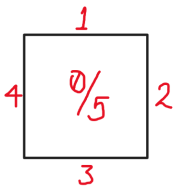

Figura 5: Posiciones de una celda

Las posiciones cardinales norte, este, sur y oeste de una celda se representan con 1, 2, 3 y 4 respectivamente. Así mismo, el centro de celda puede ser representando con un 0 (inicio / fin) o un 5 (fin parcial). Estas posiciones se utilizan para representar como se posiciona la celda en el tablero. El posicionamiento entonces se refiere a de dónde viene el camino a la celda, y a dónde va el camino desde la celda.

Para entender mejor este concepto, supóngase que se tiene un tablero 3×3 con dos colores (rojo y azul) representados por los enteros 1 y 2 (con 0 como vacío) como el que se presenta en la figura 6. Nótese en este que el posicionamiento de todas las celdas es $(0, 0)$, lo cual significa que no se han movido o no están conectadas por ningún camino.

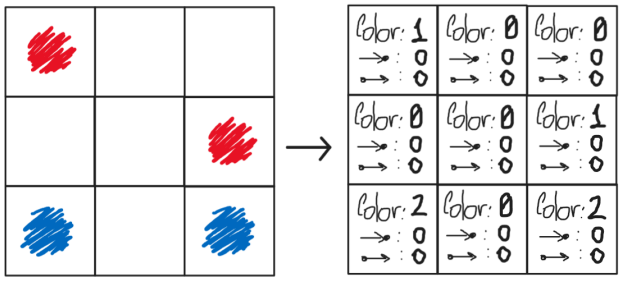

Figura 6: Ejemplo de un tablero inicial

Tracemos un primer camino para el color rojo, desde su punto ubicado en la esquina superior izquierda como el que se presenta en la figura 7. Nótese que el punto rojo ahora está representado como $(0, 3)$, indicando que es un punto inicial que va hacia el sur. La última celda del camino rojo es $(3, 5)$, indicando que un camino llega por el sur y se queda en el centro de la celda.

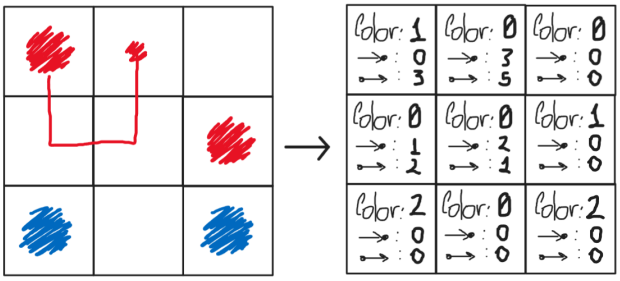

Figura 7: Ejemplo de un tablero primer movimiento

Un segundo camino para el color azul que conecta ambos puntos se presenta en la figura 8. Nótese que el punto inicial es $(0, 2)$ y el punto final es $(4, 0)$, representando este último que un camino llega desde el oeste y termina en este punto.

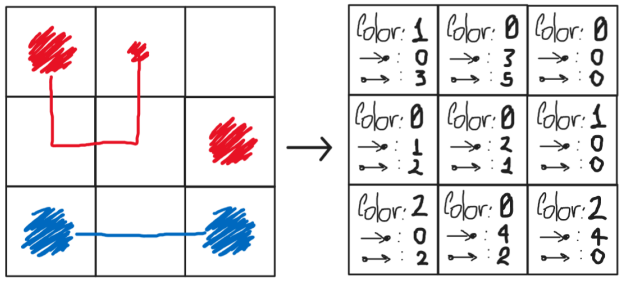

Figura 8: Ejemplo de un tablero segundo movimiento

2.3.2. Interacción con el tablero

La interacción del usuario con el tablero se compone de 3 operaciones principales:

- Iniciar un camino: el usuario hace un primer click sobre una celda que no está vacía. Esta interacción inicia un camino con el color de la celda sobre la que se para.
- Continuar un camino: el usuario continúa su click, moviendo el mouse por distintas celdas del tablero. Esta interacción continúa el camino del mismo color creado en el paso anterior.
- Terminar un camino: el usuario levanta el click. Esta interacción termina el camino que había sido iniciado.

2.3.3. Iniciar un camino

En la figura 9 se presenta un flujograma con la lógica del juego al iniciar un camino.

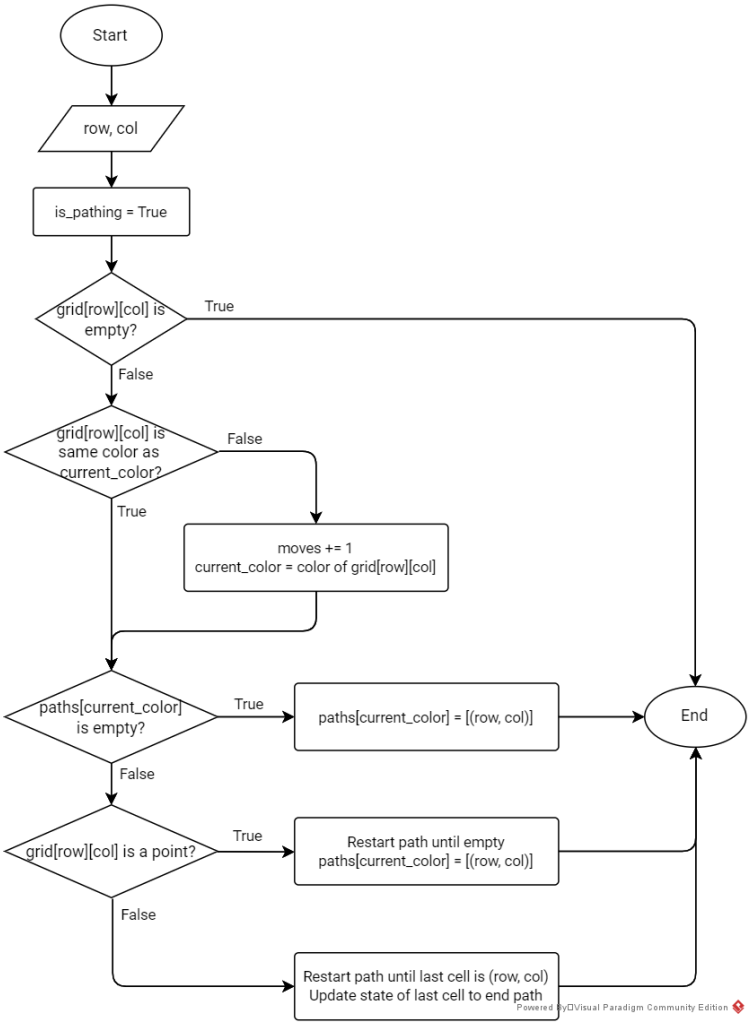

Figura 9: Flujograma de iniciar un camino

2.3.4. Continuar un camino

En la figura 10 se presenta un flujograma con la lógica del juego al continuar un camino.

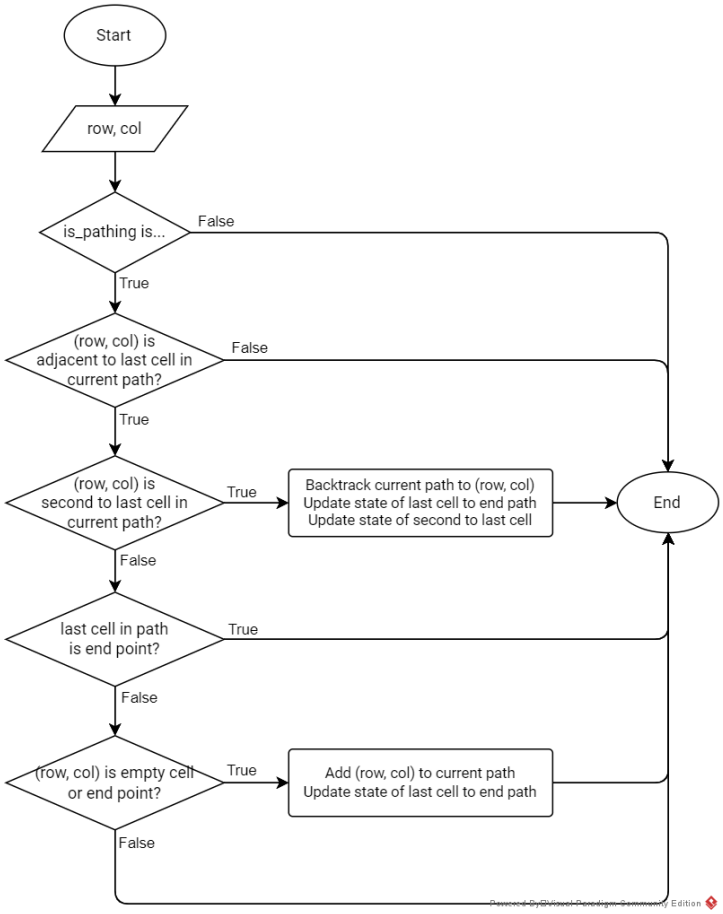

Figura 10: Flujograma de continuar un camino

2.3.5. Terminar un camino

En la figura 11 se presenta un flujograma con la lógica del juego al terminar un camino.

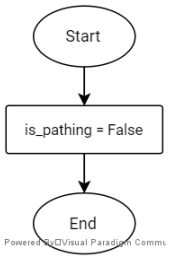

Figura 11: Flujograma de terminar un camino

3. Implementación

El juego se desarrolla en Python v3.10. Para la implementación de las interfaces gráficas, se hace uso de la librería PyGame v2.1.2, la cual cuenta con múltiples módulos para facilitar la creación de videojuegos en Python. La implementación se hace acorde a la arquitectura definida en la sección 2.1, con el uso de la librería PyGame en la vista y el controlador.

3.1. Vista previa del juego

Una vista previa del juego se presenta en la figura 12.

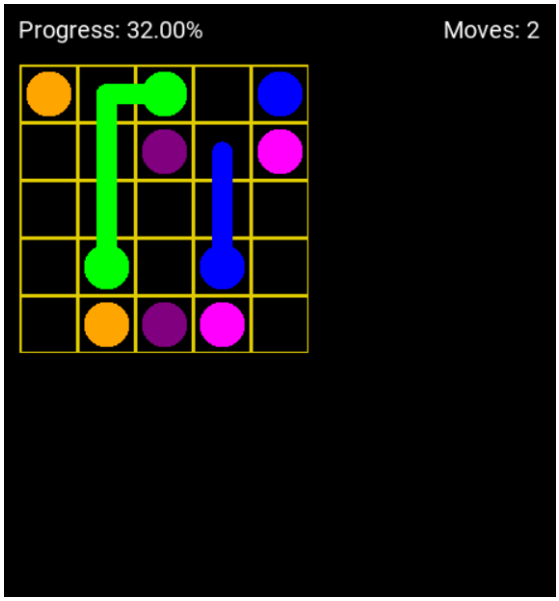

Figura 12: Vista previa del juego

3.2. Funcionalidades del Sistema

Antes de empezar, el usuario tiene la posibilidad de:

- Iniciar un juego con un tablero de niveles predeterminados
- Iniciar un juego con un tablero proporcionado en un archivo
- Iniciar un juego con un tablero creado al azar

Durante el juego, el usuario puede:

- Crear caminos manteniendo presionado con el click derecho o izquierdo del mouse
- Reiniciar el tablero presionando la tecla R
- Ver en todo momento el progreso y cantidad de movimientos

3.3. Restricciones del Sistema

Se definen las siguientes restricciones:

- La cantidad mínima de puntos es 2 y la cantidad máxima es 16
- El tablero mínimo es de 2×2 y el máximo es de 15×15

3.4. Configuración de los tableros

Como se mencionó en la sección 3.2, existen múltiples formas de iniciar un juego. Todas estas dependen de una configuración del tablero, la cual se representa como un diccionario en Python que contiene los mismos datos para representar un tablero (ver sección 2.3.1). Por ejemplo, el mismo tablero inicial de la figura 6 es representado por la siguiente configuración:

```
{
   "rows": 3,
   "cols": 3,
   "qpoints": 2,
   "points": [
        [
            [0, 0], [1, 2]
        ],
        [
            [2, 0], [2, 2]
        ]
   ]
}
```

Así mismo, esta configuración puede venir de niveles predeterminados, de un archivo o ser creada al azar.

3.4.1. Tableros de niveles predeterminados

El juego viene con unos niveles predeterminados definidos dentro de un archivo JSON. Estos niveles aseguran que es posible ganar y existe una única solución. Refiérase a la sección 5.2.1 para ver como ejecutarlos.

3.4.2. Tableros por archivos

Es posible definir una configuración similar a la presentada anteriormente dentro de un archivo JSON para proceder a jugar dicho tablero. Sin embargo, no es posible asegurar que se puede ganar. Refiérase a la sección 5.2.2 para ver como ejecutarlos.

3.4.3. Tableros al azar

Es posible definir una cantidad arbitraria de filas, columnas y puntos para que el juego genere un tablero al azar y poder proceder a jugarlo. Sin embargo, no es posible asegurar que se puede ganar. Refiérase a la sección 5.2.3 para ver como ejecutarlos.

4. Análisis

4.1. Formalización del Problema

Sean $n, m \in \mathbb{N}$ números naturales en el rango $[2, 9]$, sea $k \in \mathbb{N}$ un número natural en el rango $[2, 9]$ y sea $P = \langle\ p_i \in \mathbb{N}^{2\times 2} \mid i \in [1, k] \land j \in [1, 2] \land p_i^{[j]} = \langle x_i^{[j]}, y_i^{[j]} \rangle \land x_i \in [1, n] \land y_i \in [1, m]\ \rangle$ como k pares de puntos; es posible representar $T \in \mathbb{N}^{n\times m}$ una matriz compuesta por números naturales en el rango 0 a k tales que $T_{x_i^{[j]}, y_i^{[j]}} = i\ \forall i \in [1, k]\ \forall j \in [1, 2]$. Esto es, una matriz con las coordenadas dadas por el conjunto de puntos P representadas como un i-índice del par de puntos o 0 cuando no existe la coordenada en P.

El objetivo del problema es entonces *conectar* todos los pares de puntos sobre el tablero T de tal forma que $T_{x,y} \neq 0\ \forall x\ \forall y$, no exista coordenada vacía en T. Así mismo, debe cumplirse que es posible ir desde $p_i^{[1]}$ hacia $p_i^{[2]}$ sobre T para todo i. Defínase un vecino de $T_{x,y}$ como una coordenada adyacente a (x, y) en cualquiera de las cuatro direcciones cardinales norte $(T_{x-1,y})$, sur $(T_{x+1,y})$, oeste $(T_{x,y-1})$ o este $(T_{x,y+1})$; siempre y cuando el vecino esté definido en el rango de búsqueda $\mathbb{N}^{n\times m}$. Una manera de determinar que este camino existe es que para todo $T_{x,y} = i$, este posee al menos dos vecinos con su mismo valor i si $(x, y) \notin P$ o un vecino si $(x, y) \in P$.

4.1.1. Definición del Problema "Flow"

El problema "Flow" se define a partir de:

- **Entradas:**
  - $n \in \mathbb{N} \mid n \in [2, 9]$, una cantidad de filas
  - $m \in \mathbb{N} \mid m \in [2, 9]$, una cantidad de columnas
  - $k \in \mathbb{N} \mid k \in [2, 9]$, una cantidad de pares de puntos
  - $P = \langle\ p_i \in \mathbb{N}^{2\times 2} \mid i \in [1, k] \land j \in [1, 2] \land p_i^{[j]} = \langle x_i^{[j]}, y_i^{[j]} \rangle \land x_i \in [1, n] \land y_i \in [1, m]\ \rangle$, una secuencia de pares de puntos
- **Salidas:**
  - $T \in \mathbb{N}^{n\times m} \mid T_{x,y} \in [1, k]\ \forall x, y \land T_{x,y}$ posee al menos 2 vecinos si $(x, y) \notin P \lor T_{x,y}$ posee 1 vecino si $(x, y) \in P$.

4.1.2. Análisis del problema

Este problema se conoce en la literatura como un rompecabezas tipo *Numberlink*, y se ha comprobado que es un problema de tipo decisión y de clase NP-completo. El algoritmo que lo soluciona "perfectamente" es permutativo y dadas las definiciones presentadas en la sección 4.1.1, esta solución tardaría un tiempo demasiado grande (complejidad temporal superpolinomial) en encontrar la solución. Es por esto que se propone abordar el problema mediante un algoritmo de aproximación.

4.2. Algoritmo de Solución

4.2.1. Idea general de la solución

La primera idea que surge de un problema de este tipo, es desarrollar una solución por fuerza bruta. Sin embargo, como se mencionó en la sección anterior, este algoritmo puede demorar mucho debido a las condiciones establecidas. Entonces se empieza a abordar un algoritmo de aproximación.

El primer paso que se tomó fue el definir un tablero inicial dadas las restricciones de entrada como el que se muestra en la figura 13. Este particularmente, ya que se considera un desafío trazar el camino de los puntos azules sin interrumpir el de los amarillos.

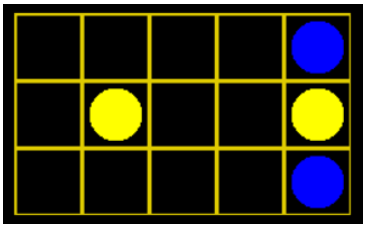

Figura 13: Tablero inicial

Deliberadamente, se procedió a tratar de simular un algoritmo que resuelva primero el camino azul y después el amarillo. Para lograr esto, se empezaron a definir unos costos asociados a las celdas de acuerdo a la distancia de Manhattan ($d(a, b) = |a_1 - b_1| + |a_2 - b_2|$) con el punto de partida y el punto de llegada, como se muestra en la figura 14.

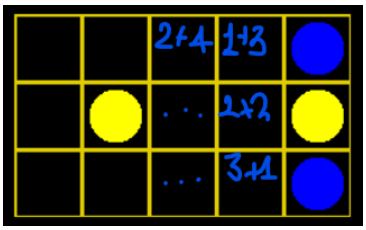

Figura 14: Cálculo de costos inicial

Sin embargo, se encontró un problema, y es que después de trazar el mejor camino azul y proceder a trazar el camino amarillo, no se encontraría ninguno, como se muestra en la figura 15.

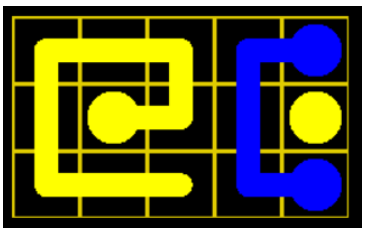

Figura 15: Camino bloqueado

Esta situación implicó la necesidad de implementar una especie de backtracking o el cálculo de mejores costos para tener mayores repercusiones sobre bloquear un camino de

otro color. Se consideró entonces la posibilidad de acumular costos *inversos*. Esto es, un camino para un color tiene el costo usual sumado con el costo inverso de los otros colores. Por ejemplo, el color amarillo tiene un costo de $1 + 2$ en la celda $(1, 2)$, debido a que sus puntos tienen esa distancia a esa celda. El costo inverso sería el mayor costo (cantidad de celdas) menos el costo normal del color. De esta forma, se desincentiva bloquear los mejores caminos de otros colores, como se muestra en la figura 16.

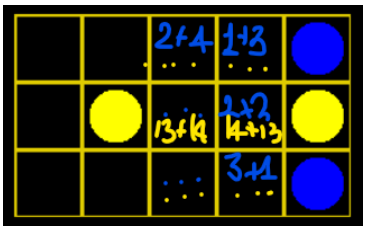

Figura 16: Cálculo de costos inversos

Sin embargo, rápido se desechó la anterior idea, debido a que una cantidad k de puntos implica hacer $k*2$ cálculos para saber el costo de una sola celda. Así mismo, situaciones como la que se presenta en la figura 17 en donde existen varios colores cercanos, se desincentivaría el uso del camino correcto y se volvería a coger el camino más corto.

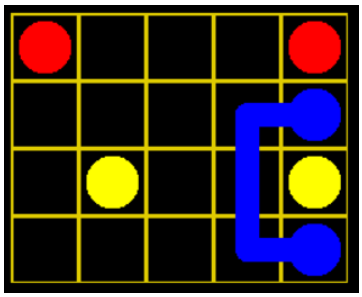

Figura 17: Efecto adverso por más colores

En este punto, se cayó en cuenta que la primera idea del algoritmo, de un doble costo asociado a la distancia origen y destino se parecen mucho al algoritmo de búsqueda A*, con la búsqueda del mejor camino entre 2 nodos. Es por esto que se toma esta idea y se decide utilizar el algoritmo A* para buscar el mejor camino entre cada par de puntos de cada color. Por cada par de puntos, se inicia en un punto y se empiezan a recorrer los vecinos siguiendo la lógica de prioridades y costos así:

- $g(c) = g(c-1) + 1$, el costo de llegar a la siguiente celda como el costo de la celda padre más 1
- $h(c) = d(c, e)$, el costo heurístico de ir desde la siguiente celda hasta la celda fin como la distancia Manhattan entre estas
- $f(c) = g(c) + h(c)$

Al aplicar estas reglas y el algoritmo básico de A*, se encuentra que se está en una situación muy parecida a la inicial. Después de calcular el camino azul exactamente igual a como en la figura 15 y determinar que es imposible trazar el camino amarillo, se necesita una forma de hacer backtracking sobre el camino azul para probar otro camino. A* no ofrece la posibilidad de determinar el N-avo mejor camino, por lo que siempre se va a recalcular el mismo camino azul.

Se propone entonces afectar los costos después de calcular un camino, para desincentivar recorrer el mismo camino la siguiente vez que se calcule A*. Para lograr esto, se implementa un efecto parecido a un mapa de calor, donde los caminos (y celdas) que se calculan continuamente con el algoritmo crecen en un costo que incentiva al algoritmo a buscar otros caminos.

La idea entonces es manejar una matriz de costos añadidos por cada color disponible, que se actualiza cada vez que se calcula un camino de un color. La matriz de costos añadidos crece en $n * m$ en cada celda visitada, con el fin de que sea más lucrativo recorrer todos los demás nodos que volver a pasar por uno ya visitado. Se puede entonces volver a representar $g(c)$ como:

- $g(c) = g(c-1)+1+a(c)$, el costo de llegar a una celda es el costo del camino existente más el costo añadido por repetir un camino sobre esa celda
- con $a(c) = a(c) + n * m$ cada vez que c hace parte del camino solución

Finalmente, el algoritmo recorre todos los puntos disponibles calculando A* sobre el par de puntos. Cuando encuentra un camino, incrementa los pesos y pasa al siguiente punto. Si el siguiente punto queda en un punto muerto o se repite, se regresa al punto anterior para recalcular un nuevo camino. Los costos añadidos ayudan a que A* no calcule siempre el mismo camino, sino que recorra el espacio de búsqueda para encontrar caminos óptimos no explorados.

4.2.2. Escritura del algoritmo

El algoritmo se basa principalmente en dos funcionalidades: encontrar un camino entre un par de puntos, y encontrar todos los caminos del tablero. Para encontrar todos los caminos, se utiliza la primera funcionalidad.

Algorithm 1 Calcular camino entre un par de puntos

```
1: procedure SolvePoint(T, P, C, p)
2:     start ← P[p][0] ∧ end ← P[p][1]
3:     let q be a PriorityQueue
4:     q.put(0, start)
5:     let v be a Map
6:     v[start] ← (0, Null)
7:     ended ← 0
8:     while not q.Empty() ∧ ended = 0 do
9:         c ← queue.get()
10:        if c = end then
11:            ended ← 1
12:        end if
13:        for n in Neighbors(c) do
14:            g ← v[c][0] + 1 + C[p][n]
15:            h ← Distance(n, end)
16:            f ← g + h
17:            if n not in v ∨ v[n][0] > g then
18:                q.put(f, n)
19:                v[n] ← (g, c)
20:            end if
21:        end for
22:    end while
23:    if end not in v then
24:        return [ ]
25:    end if
26:    path ← [ ]
27:    c ← end
28:    while c is not Null do
29:        path ← c ∪ path
30:        c ← v[c][1]
31:        C[p][c] ← C[p][c] + AddedCost
32:    end while
33:    return path
34: end procedure
```

Algorithm 2 Solucionar tablero

```
1: procedure SOLVE(T, P, C, k)
2:     p ← 1
3:     while 1 ≤ p ≤ k do
4:         T.REMOVEPATH(p)
5:         path ← SOLVEPOINT(T, P, C, p)
6:         if path = [ ] then
7:             p ← p − 1
8:             continue
9:         end if
10:        T.ADDPATH(path)
11:        if T.PROGRESS() = 1 then
12:            return 1
13:        end if
14:        point ← point + 1
15:    end while
16:    return 0
17: end procedure
```

4.3. Experimentación

Para la experimentación, se cuenta con 25 niveles importados de la aplicación *Flow Free*, los cuáles garantizan una solución única. Son tableros desde 5×5 hasta 9×9, 5 niveles por tamaño.

4.3.1. Prueba de completitud

El objetivo de esta prueba es determinar si el algoritmo implementado es capaz de resolver los 25 niveles disponibles. El protocolo de experimentación es:

1. Se cargan los 25 niveles en el formato de configuración de tablero como se presenta en la sección 3.4.

2. Cada nivel se corre el algoritmo con el comando `python main.py -l LEVEL -s -d`. Más detalles en las secciones 5.2 y 5.3.

3. Se determina si el algoritmo fue capaz de solucionar el tablero

4.3.2. Prueba de tiempo de ejecución

El objetivo de esta prueba es determinar el tiempo que le toma al algoritmo solucionar distintas configuraciones de tableros. El protocolo de experimentación es:

1. Se cargan los 25 niveles en el formato de configuración de tablero como se presenta en la sección 3.4.

2. Se seleccionan los siguientes niveles que el algoritmo puede solucionar:
    - Nivel 5: tablero 5×5 con 4 pares de puntos
    - Nivel 10: tablero 6×6 con 4 pares de puntos
    - Nivel 15: tablero 7×7 con 6 pares de puntos
    - Nivel 18: tablero 8×8 con 6 pares de puntos
    - Nivel 25: tablero 9×9 con 8 pares de puntos
3. Se corre el algoritmo de solución 3 veces por cada nivel y se mide el tiempo de ejecución.
4. Se promedian los tiempos de ejecución por cada nivel escogido.

4.4. Resultados

4.4.1. Prueba de completitud

La prueba se sigue de acuerdo al protocolo establecido en la sección 4.3.1. Se determina que el algoritmo no es capaz de resolver los siguientes niveles / configuraciones:

- Nivel 17: tablero 8×8 con 6 pares de puntos
- Nivel 19: tablero 8×8 con 7 pares de puntos
- Nivel 20: tablero 8×8 con 6 pares de puntos
- Nivel 22: tablero 9×9 con 8 pares de puntos
- Nivel 24: tablero 9×9 con 7 pares de puntos

4.4.2. Prueba de tiempo de ejecución

La prueba se sigue de acuerdo al protocolo establecido en la sección 4.3.2. Los resultados se presentan en la siguiente tabla:

Grid	**Average time (s)**
$n = 5, m = 5, k = 4$	0.002658
$n = 6, m = 6, k = 4$	0.004479
$n = 7, m = 7, k = 6$	0.016763
$n = 8, m = 8, k = 6$	0.173835
$n = 9, m = 9, k = 8$	1.515127

Como se puede evidenciar, conforme el tamaño del tablero crece, el tiempo de ejecución crece a gran ritmo. Esto parece indicar que la complejidad del algoritmo obedece la de un polinomio de alto grado.

5. Documentación

5.1. Instalación

5.1.1. Requisitos previos

La máquina debe contar con la versión de Python 3.10 o superior.

5.1.2. Windows

Clonar el repositorio donde se aloja el código

```
> git clone https://github.com/amoralesc/flow.git
> cd flow
```

Crear un entorno virtual para instalar las dependencias

```
> python -m venv venv
```

(Opcional) Activar los permisos de ejecución de scripts

```
> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Activar el entorno virtual

```
> venv\Scripts\activate
```

Instalar las dependencias

```
> pip install -r requirements.txt
```

5.1.3. Linux / Mac

Clonar el repositorio donde se aloja el código

```
$ git clone https://github.com/amoralesc/flow.git
$ cd flow
```

Crear un entorno virtual para instalar las dependencias

```
$ python -m venv venv
```

Activar el entorno virtual

```
$ source venv/bin/activate
```

Instalar las dependencias

```
$ pip install -r requirements.txt
```

5.2. Ejecución

Para ver todas las opciones de ejecución posibles:

```
$ python main.py -h
```

5.2.1. Tableros de niveles predeterminados

Para visualizar las configuraciones de tablero de los niveles disponibles:

```
$ cat data/levels.json
```

Para ejecutar un tablero de un nivel predeterminado:

```
$ python main.py -l LEVEL
```

donde LEVEL es un entero que representa la posición del nivel en el arreglo de niveles disponibles (indexado desde 1).

5.2.2. Tableros por archivos

Configurar previamente el tablero siguiendo los lineamientos de 3.4 dentro de un archivo. Para ejecutar el tablero desde un archivo:

```
$ python main.py -f FILE
```

donde FILE es la ruta hasta el archivo que contiene la configuración del tablero.

5.2.3. Tableros al azar

Para ejecutar un tablero al azar:

```
$ python main.py -r ROWS COLS POINTS
```

donde ROWS, COLS y POINTS son 3 enteros representando la cantidad de filas, columnas y puntos respectivamente.

5.3. Solver

El programa cuenta con un solver que implementa el algoritmo descrito en la sección 4.2. Este solver se puede correr con cualquier de las opciones de ejecución presentadas anteriormente, agregando la bandera `-s`. Por ejemplo, para correr el solver sobre el nivel 1:

```
$ python main.py -l 1 -s
```

Así mismo, activando la bandera de debug `-d`, es posible ver el paso a paso que el solver realiza para solucionar el tablero:

```
$ python main.py -l 1 -s -d
```

El solver no garantiza encontrar una solución, así la configuración del tablero tenga una (o varias).

5.4. Documentación del código

El código se encuentra completamente documentado, y es posible referirse a este para entender las distintas partes de la implementación.