DOI: 10.24275/uama.6749.9407

Cálculo de matrices intercaladas óptimas

Tesis presentada a la
Universidad Autónoma Metropolitana Azcapotzalco
como requerimiento parcial para obtener el grado de
Maestro en Optimización
por

Ing. Rubén Alejandro González Yáñez

Asesores:
Dr. Francisco Javier Zaragoza Martínez
Departamento de Sistemas, UAM Azcapotzalco

Dr. Rodrigo Alexander Castro Campos
Departamento de Sistemas, UAM Azcapotzalco

Ciudad de México, México
7 de septiembre de 2022

Dedicatoria

A mis papás y a mi hermanita, por su amor y apoyo incondicional.

Agradecimientos

A mis papás por su amor y apoyo en cada instante de mi vida, a mi hermanita por su ánimo constante y a mis asesores por su dedicación perseverante.

Resumen

Considere una matriz de tamaño $r \times s$ cuyos elementos pertenecen a un conjunto ordenado de colores numerados del 1 al n. Dicha matriz es intercalada si es latina y cada una de sus submatrices de tamaño 2×2 tiene dos colores distintos (intercalación) o cuatro colores distintos (cointercalación). Sea $f(r, s)$ el número mínimo de colores que se necesitan para colorear una matriz intercalada de tamaño $r \times s$, la conjetura de Yuzvinsky postula que $f(r, s)$ coincide con la función $r \circ s$ de Hopf y Stiefel.

En esta tesis se presenta un acercamiento computacional para probar que la conjetura de Yuzvinsky se cumple para matrices de tamaño de hasta 32×32. Se busca encontrar un contraejemplo de la conjetura de Yuzvinsky, es decir, se busca encontrar al menos una matriz intercalada que utilice un color menos de los que postula la conjetura, es decir, a lo más $r \circ s - 1$ colores. Si dicho contraejemplo no existe, se considera que la conjetura se cumple. Una búsqueda exhaustiva completa requeriría explorar n^{rs} posibilidades al intentar colorear una matriz intercalada de tamaño $r \times s$ con n colores, por lo que se necesitan buenas estrategias para poder resolver este problema.

Esta tesis presenta el diseño e implementación de dos modelos de programación lineal entera para resolver el problema con el solucionador lineal Gurobi. También se presenta un algoritmo de búsqueda con retroceso que logra determinar eficientemente los colores válidos para una celda de la matriz. Posteriormente se presenta un algoritmo que explora el espacio de búsqueda de forma híbrida (búsqueda en amplitud y en profundidad) y que es capaz de aprovechar el paralelismo disponible en la computadora. Posteriormente se presenta el uso del algoritmo híbrido para la generación de las matrices intercaladas simétricas no isomorfas y cómo sus resultados pueden usarse para obtener matrices simétricas de tamaños más grandes. Los algoritmos pudieron probar que la conjetura de Yuzvinsky es cierta para trece casos de los que quedaban abiertos. La tesis muestra que las estrategias utilizadas se pueden modificar o mejorar para poder desarrollar nuevas estrategias más sofisticadas y aún más rápidas.

Índice general

1. Introducción **1**

1.1. Definiciones preliminares 1

1.2. Estado de la técnica 2

1.3. Objetivos 4

1.3.1. Objetivo general 4

1.3.2. Objetivos específicos 4

2. Modelos de programación entera **5**

2.1. Primer modelo 6

2.1.1. Variables de decisión del primer modelo 6

2.1.2. Restricciones del modelo de programación entera 6

2.2. Segundo modelo 7

2.2.1. Variables de decisión del segundo modelo 7

2.2.2. Restricciones del modelo de programación entera 8

2.3. Restricciones adicionales 9

2.4. Resultados experimentales 9

3. Matrices intercaladas óptimas **11**

3.1. Determinación de colores asignables en una celda 12

3.1.1. Algoritmo en tiempo cúbico 12

3.1.2. Algoritmo en tiempo cuadrático 13

3.1.3. Algoritmo en tiempo lineal 14

3.2. Prevención de isomorfismo 16

3.2.1. Resultados 17

3.3. Búsqueda concurrente 17

3.4. Resultados experimentales 20

4. Matrices intercaladas simétricas **23**
4.1. Búsqueda concurrente 23
4.1.1. Detección de isomorfismo 24
4.1.2. Resultados de las matrices intercaladas simétricas 26
4.1.3. Generación de matrices intercaladas a partir de las simétricas 27
4.2. Resultados experimentales 28

5. Conclusiones **29**

A. Código fuente de los algoritmos implementados **31**
A.1. Implementación del modelo 1 de programación entera lineal utilizando la interfaz de programación de C++ de Gurobi 31
A.2. Implementación del modelo 1 de programación entera lineal con las restricciones adicionales utilizando la interfaz de programación de C++ de Gurobi . . 36
A.3. Implementación del modelo 2 de programación entera lineal utilizando la interfaz de programación de C++ de Gurobi 42
A.4. Implementación del modelo 2 de programación entera lineal con las restricciones adicionales utilizando la interfaz de programación de C++ de Gurobi . . 46
A.5. Implementación del algoritmo base de Backtracking 52
A.6. Implementación del algoritmo de Backtracking para las matrices simétricas con isomorfismo 58
A.7. Implementación del algoritmo de Backtracking que utiliza las matrices precalculadas 70
A.8. Implementación del algoritmo de Backtracking que utiliza las matrices simétricas precalculadas para generar matrices simétricas más grandes 76

Bibliografía **91**

Capítulo 1

Introducción

1.1. Definiciones preliminares

Una matriz de $r \times s$ es un arreglo en dos dimensiones con r filas y s columnas, como el mostrado en la ecuación (1.1).

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1s} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2s} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3s} \\ \dots & \dots & \dots & \dots & \dots \\ a_{r1} & a_{r2} & a_{r3} & \dots & a_{rs} \end{pmatrix} \tag{1.1}$$

Supondremos que los elementos de la matriz pertenecen a un conjunto ordenado de *colores*. Los elementos de este conjunto están numerados del 1 al n.

Una matriz *latina* es una matriz donde todos los elementos de cada fila y todos los elementos de cada columna son distintos, como la matriz L mostrada en la ecuación (1.2).

$$L = \begin{pmatrix} 1 & 2 & 5 & 4 \\ 5 & 1 & 4 & 3 \\ 3 & 4 & 2 & 5 \end{pmatrix} \tag{1.2}$$

Una matriz *intercalada* es una matriz latina donde todas sus submatrices de 2×2 tienen dos colores distintos (*intercalación*) o cuatro colores distintos (*cointercalación*), como la

matriz M mostrada en la ecuación (1.3).

$$M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 5 & 6 & 7 & 8 \end{pmatrix} \tag{1.3}$$

Dos matrices intercaladas M_1 y M_2 son isomorfas si es posible permutar las filas, las columnas y los colores de M_1 para obtener M_2.

Se desea minimizar la cardinalidad del conjunto de colores que permite construir una matriz intercalada de $r \times s$. Se dice que una matriz intercalada es de tipo $[r, s, n]$ si es de $r \times s$ y tiene n colores. Una matriz intercalada de tipo $[r, s, n]$ es *óptima* si n es el número mínimo de colores necesarios para colorear cualquier matriz de $r \times s$. Un ejemplo de una matriz intercalada óptima de tipo $[3, 4, 4]$ es:

$$N = \begin{pmatrix} 2 & 3 & 4 & 1 \\ 3 & 2 & 1 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix} \tag{1.4}$$

A los interesados en aplicaciones de matrices intercaladas los dirigimos a [11].

1.2. Estado de la técnica

En 1941, Hopf [3] y Stiefel [9] introdujeron la función $\circ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definida como:

$$r \circ s = \min \left\{ n \in \mathbb{N} \mid \binom{n}{k} \equiv 0 \bmod 2,\ \forall k : n - r < k < s \right\}. \tag{1.5}$$

La función además puede calcularse recursivamente con la fórmula de Pfister [8]:

$$r \circ s = \begin{cases} s \circ r & \text{si } r > s \\ s & \text{si } r = 1 \\ 2^k & \text{si } 2^{k-1} < r \leq s \leq 2^k \text{ para alguna } k \in \mathbb{N} \\ 2^k + r \circ (s - 2^k) & \text{si } r \leq 2^k < s \text{ para alguna } k \in \mathbb{N}. \end{cases} \tag{1.6}$$

Sea $f(r, s)$ el número mínimo de colores que se necesitan para colorear una matriz intercalada de $r \times s$, la conjetura de Yuzvinsky postula que $f(r, s) = r \circ s$.

Claramente $f(r, s) \leq rs$, pues rs es la cantidad de colores empleados en una matriz

intercalada que usa un color distinto para cada elemento. Además, $f(r,s) \geq \text{máx}(r,s)$ ya que ninguna fila o columna puede tener colores repetidos. Ya que las matrices intercaladas son cerradas bajo transposición, se tiene que $f(r,s) = f(s,r)$. Las matrices intercaladas también son cerradas bajo toma de submatrices, por lo que f es una función monótona, es decir, si $r \leq R$ y $s \leq S$ entonces $f(r,s) \leq f(R,S)$.

Yiu [10] verificó la conjetura de Yuzvinsky cuando $r, s \leq 16$. Lam [4] la probó para matrices intercaladas cuadradas, junto con algunos otros casos especiales. En el mismo documento Lam dió un resultado implicando que la conjetura es asintóticamente verdadera para $\frac{2}{3}$ de los pares de enteros (r,s). Gitler, Reyes y Zaragoza [1] mejoraron la cota asintótica de Lam para $\frac{5}{6}$ de los pares de enteros (r,s) y probaron la conjetura para $r \leq 8$.

De acuerdo a [1] se tiene la siguiente lista de los casos abiertos de las matrices de tamaño hasta 32×32, es decir, que faltan por verificar si cumple o no la conjetura de Yuzvinsky en estas matrices:

- $r = 9$ y $s \in \{17, 18, 19, 20, 21, 22, 23\}$.
- $r = 10$ y $s \in \{17, 19, 21\}$
- $r = 11$ y $s \in \{17, 18, 21\}$
- $r \in \{12, 14, 15\}$ y $s = 17$
- $r = 13$ y $s \in \{17, 18, 19\}$

Para resolver cada uno de estos casos, se podría intentar encontrar una matriz intercalada que use $r \circ s - 1$ colores o menos, la cual constituiría un contraejemplo. Si tal contraejemplo no existe, la conjetura sería cierta para ese caso. El siguiente resultado con respecto a contraejemplos minimales apareció primero en [12].

Teorema 1. *Sea M una matriz intercalada de tipo $[r,s,n]$ que es un contraejemplo de la Conjetura de Yuzvinsky con mínimo $r+s$. Se sigue que $n = r \circ s - 1 = r + s - 2$.*

En la construcción de matrices intercaladas óptimas se puede considerar la frecuencia de los colores utilizados para descartar algunos casos, ya sea porque no es posible asignar alguna frecuencia de colores o porque algunas frecuencias implicarían utilizar más colores de los deseados. Tomando en cuenta la frecuencia de los colores se tiene lo siguiente.

Sea M una matriz intercalada de tipo $[r,s,n]$ y sea a un color con frecuencia t en M. Se puede suponer sin pérdida de generalidad que el color a aparece a lo largo de la diagonal

principal de M. Considérese la siguiente partición de M:

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \tag{1.7}$$

En ella, A es una submatriz de tamaño $t \times t$, B es una submatriz de tamaño $t \times (s - t)$, C es una submatriz de tamaño $(r - t) \times t$, y D es una submatriz de tamaño $(r - t) \times (s - t)$.

Teorema 2 ([5, 6, 10])**.** *La matriz*

$$M' = \begin{pmatrix} A \ B \ C^T \end{pmatrix} \tag{1.8}$$

es intercalada de tipo $[t, s + r - t, n']$ *con* $n' \leq n$*. Esto es,* $n \geq f(t, s + r - t)$*.*

Una matriz intercalada es de tipo extendido $[r, s, n, t]$ si es de tipo $[r, s, n]$ y el color que más aparece tiene frecuencia t. El teorema anterior descarta la existencia de matrices intercaladas para algunas combinaciones de valores de r, s, n, t. En el mismo sentido, se observa lo siguiente:

Lema 1. *No existen matrices intercaladas de tipo extendido* $[r, s, n, t]$ *para* $t > \text{mín}(r, s)$*.*

Lema 2. *No existen matrices intercaladas de tipo extendido* $[r, s, n, t]$ *para* $t < \lceil \frac{rs}{n} \rceil$*.*

1.3. Objetivos

1.3.1. Objetivo general

Diseñar modelos e implementar algoritmos para encontrar matrices intercaladas óptimas.

1.3.2. Objetivos específicos

1. Diseñar e implementar un algoritmo de búsqueda con retroceso para encontrar matrices intercaladas óptimas.

2. Proponer un modelo de programación entera para encontrar matrices intercaladas óptimas.

3. Diseñar e implementar un algoritmo para encontrar matrices intercaladas simétricas.

4. Diseñar e implementar un algoritmo paralelo para encontrar matrices intercaladas óptimas.

Capítulo 2

Modelos de programación entera

En este capítulo se definen dos modelos de programación lineal entera para el problema de determinar si es posible encontrar una matriz intercalada de tipo $[r, s, n]$. Cada modelo se presenta con las restricciones necesarias para garantizar latinicidad e intercalación. También se presenta un conjunto de restricciones adicionales para cada modelo que reducen el espacio de búsqueda. Cabe mencionar que, por simplicidad, las restricciones que se muestran en cada uno de los modelos no están en la forma estándar, pero es posible reescribirlas para que sí lo estén. Cada modelo se generó y resolvió con la interfaz de programación en C++ del solucionador de programación lineal entera Gurobi [2].

Dadas constantes r, s y n, los modelos desarrollados están planteados de forma que tengan solución sólo si es posible construir una matriz de tipo $[r, s, m]$ con $m \leq n$. La función objetivo sólo es relevante si se desea minimizar el valor de m. Para intentar probar la conjetura de Yuzvinsky, se podría elegir $n = r \circ s$ (valor para el cual se sabe que siempre existen soluciones) y revisar que el valor mínimo de m sea igual a n. De forma alternativa, se podría elegir $n = r \circ s - 1$ y verificar que el modelo no tiene solución. Dicho esto y para reducir el tamaño del espacio de búsqueda, es posible definir variables z_c que denoten si el color c fue usado, de modo que se agregue la restricción $z_1 \geq z_2 \geq \cdots \geq z_n$ para evitar que existan colores intermedios sin usar y donde posiblemente se minimice $m = \sum_{c=1}^{n} z_c$.

Además, cabe mencionar que una submatriz de tamaño 2×2 se puede representar con las coordenadas de sus cuatro celdas y se puede ver gráficamente como:

$$\begin{matrix} (i_1, j_1) & \dots & (i_1, j_2) \\ \vdots & & \vdots \\ (i_2, j_1) & \dots & (i_2, j_2). \end{matrix} \tag{2.1}$$

2.1. Primer modelo

2.1.1. Variables de decisión del primer modelo

El primer modelo tiene tres familias de variables de decisión. Las variables x_{ij}^c son binarias y sirven para saber si el color c ha sido asignado a la celda (i, j).

$$x_{ij}^c = \begin{cases} 1 & \text{si el color } c \text{ fue asignado a la celda } i, j, \\ 0 & \text{en otro caso.} \end{cases} \tag{2.2}$$

Las variables $y_{i_1 i_2 j_1 j_2}^c$ son binarias y sirven para saber si el color c aparece en alguna de las cuatro celdas de la submatriz de tamaño 2×2 con contraesquinas (i_1, j_1) y (i_2, j_2), donde $1 \leq i_1 < i_2 \leq r$, $1 \leq j_1 < j_2 \leq s$ y $1 \leq c \leq n$.

$$y_{i_1 i_2 j_1 j_2}^c = \begin{cases} 1 & \text{si } c \text{ aparece en la submatriz con contraesquinas } (i_1, j_1), (i_2, j_2), \\ 0 & \text{en caso contrario.} \end{cases} \tag{2.3}$$

Las variables auxiliares $k_{i_1 j_1 i_2 j_2}$ son enteras y hay una para cada submatriz de tamaño 2×2 con contraesquinas (i_1, j_1), (i_2, j_2), donde $1 \leq i_1 < i_2 \leq r$ y $1 \leq j_1 < j_2 \leq s$. La cantidad de colores distintos de la submatriz debe ser 2 o 4 y se buscará que esto sea igual a $2k_{i_1 j_1 i_2 j_2}$, por lo que la variable auxiliar deberá valer 1 o 2.

2.1.2. Restricciones del modelo de programación entera

En este primer modelo se tienen cinco familias de restricciones que cubren las dos características de las matrices intercaladas, latinicidad e intercalación. La familia (2.4) de rs restricciones sirve para que cada celda tenga exactamente un color.

$$\sum_{c=1}^{n} x_{ij}^c = 1, \qquad \forall\ 1 \leq i \leq r, 1 \leq j \leq s \tag{2.4}$$

La familia (2.5) de sn restricciones obliga a que cada color aparezca a lo más una vez por fila.

$$\sum_{i=1}^{r} x_{ij}^c \leq 1, \qquad \forall\ 1 \leq j \leq s, 1 \leq c \leq n \tag{2.5}$$

La familia (2.6) de rn restricciones obliga a que cada color aparezca a lo más una vez por

columna.

$$\sum_{j=1}^{s} x_{ij}^{c} \leq 1, \qquad \forall\ 1 \leq i \leq r, 1 \leq c \leq n \tag{2.6}$$

La familia (2.7) de rsn restricciones sirve para determinar si el color c aparece en alguna de las cuatro celdas de la submatriz con esquinas (i_1, j_1), (i_1, j_2), (i_2, j_1), (i_2, j_2).

$$y_{i_1 i_2 j_1 j_2}^{c} = \text{máx}(x_{i_1 j_1}^{c}, x_{i_1 j_2}^{c}, x_{i_2 j_1}^{c}, x_{i_2 j_2}^{c}), \qquad \forall\ 1 \leq i_1 < i_2 \leq r, 1 \leq j_1 < j_2 \leq s, 1 \leq c \leq n \tag{2.7}$$

La familia (2.8) de rs restricciones sirve para garantizar que los colores utilizados sean dos o cuatro. La suma de las variables $y_{i_1 i_2 j_1 j_2}^{c}$ de la submatriz da como resultado el número de colores que aparecen en la submatriz y se obligará a que dicha suma sea dos o cuatro usando las variables auxiliares del tipo $k_{i_1 j_1 i_2 j_2}$.

$$\sum_{c=1}^{n} y_{i_1 i_2 j_1 j_2}^{c} = 2k_{i_1 j_1 i_2 j_2}, \qquad \forall\ 1 \leq i_1 < i_2 \leq r, 1 \leq j_1 < j_2 \leq s \tag{2.8}$$

La implementación de este modelo se puede encontrar en el apéndice A.1.

2.2. Segundo modelo

2.2.1. Variables de decisión del segundo modelo

El segundo modelo utiliza dos familias de variables de decisión. Las variables x_{ij}^{c} son binarias y sirven para saber si el color c ha sido asignado a la celda (i, j).

$$x_{ij}^{c} = \begin{cases} 1 & \text{si el color } c \text{ fue asignado a la celda } i, j, \\ 0 & \text{en otro caso.} \end{cases} \tag{2.9}$$

Las variables auxiliares $k_{i_1 j_1 i_2 j_2}$ son binarias y hay una para cada submatriz de tamaño 2×2 con contraesquinas (i_1, j_1), (i_2, j_2), donde $1 \leq i_1 < i_2 \leq r$ y $1 \leq j_1 < j_2 \leq s$. Si una diagonal de la submatriz tiene una cantidad impar de colores distintos, la contradiagonal también. De forma similar, si una diagonal de la submatriz tiene una cantidad par de colores distintos, la contradiagonal también. Debe cumplirse que $k_{i_1 j_1 i_2 j_2}$ tenga la paridad de la cantidad de colores distintos de ambas diagonales.

2.2.2. Restricciones del modelo de programación entera

En este segundo modelo se tienen siete familias de restricciones para cubrir las dos características necesarias, latinicidad e intercalación. La familia (2.10) de rs restricciones sirve para que cada celda tenga exactamente un color.

$$\sum_{c=1}^{n} x_{ij}^{c} = 1, \qquad \forall\ 1 \leq i \leq r, 1 \leq j \leq s \tag{2.10}$$

La familia 2.11 de sn restricciones obliga a que cada color aparezca a lo más una vez por fila.

$$\sum_{i=1}^{r} x_{ij}^{c} \leq 1, \qquad \forall\ 1 \leq j \leq s, 1 \leq c \leq n \tag{2.11}$$

La familia (2.12) de rn restricciones obliga a para que cada color parezca a lo más una vez por columna.

$$\sum_{j=1}^{s} x_{ij}^{c} \leq 1, \qquad \forall\ 1 \leq i \leq r, 1 \leq c \leq n \tag{2.12}$$

La familia (2.13) de rsn restricciones y la familia (2.14) de rsn restricciones sirven para detectar cuando la diagonal o la contradiagonal, respectivamente, tienen el mismo color. Si esto ocurre entonces se obliga a que la variable auxiliar valga 1 y esto quiere decir que hay una intercalación.

$$x_{i_1 j_1}^{c} + x_{i_2 j_2}^{c} \leq 1 + k_{i_1 j_1 i_2 j_2}, \qquad \forall\ 1 \leq i_1 < i_2 \leq r, 1 \leq j_1 < j_2 \leq s, 1 \leq c \leq n \tag{2.13}$$

$$x_{i_1 j_2}^{c} + x_{i_2 j_1}^{c} \leq 1 + k_{i_1 j_1 i_2 j_2}, \qquad \forall 1 \leq i_1 < i_2 \leq r, 1 \leq j_1 < j_2 \leq s, 1 \leq c \leq n \tag{2.14}$$

La familia (2.15) de rsn restricciones y la familia (2.16) de rsn restricciones sirven para detectar cuando la diagonal o la contradiagonal, respectivamente, tienen colores distintos. Si esto ocurre entonces se obliga a que la variable auxiliar valga 0 y esto quiere decir que hay una cointercalación.

$$x_{i_1 j_1}^{c} + \sum_{d \neq c} x_{i_2 j_2}^{d} \leq 2 - k_{i_1 j_1 i_2 j_2}, \qquad \forall\ 1 \leq i_1 < i_2 \leq r, 1 \leq j_1 < j_2 \leq s, 1 \leq c \leq n \tag{2.15}$$

$$x_{i_1 j_2}^{c} + \sum_{d \neq c} x_{i_2 j_1}^{d} \leq 2 - k_{i_1 j_1 i_2 j_2}, \qquad \forall\ 1 \leq i_1 < i_2 \leq r, 1 \leq j_1 < j_2 \leq s, 1 \leq c \leq n \tag{2.16}$$

La implementación de este modelo se puede encontrar en el apéndice A.3.

2.3. Restricciones adicionales

Es posible reducir la cantidad de matrices candidatas que Gurobi debe considerar si se agregan restricciones que descarten matrices isomorfas. A su vez, es posible buscar (o determinar la inexistencia de) matrices intercaladas de tipo extendido $[r, s, n, t]$. A continuación se presenta un conjunto de familias de restricciones que incorporan las ideas previas y que pueden simplemente agregarse a los modelos previos.

La familia (2.17) de s restricciones obliga a que la primera fila tenga colores $1, \ldots, s$.

$$x_{1j}^{j} = 1, \qquad \forall\ 1 \leq j \leq s \tag{2.17}$$

La familia (2.18) de t restricciones obliga a que el 1 aparezca en la diagonal con la frecuencia t dada.

$$x_{ii}^{1} = 1, \qquad \forall\ 1 \leq i \leq t \tag{2.18}$$

La familia (2.19) de n restricciones impide que algún color sobrepase la frecuencia t dada.

$$\sum_{i=1}^{r} \sum_{j=1}^{s} x_{ij}^{c} \leq t, \qquad \forall\ 1 \leq c \leq n \tag{2.19}$$

La familia (2.20) de tn restricciones fuerza a que la submatriz cuadrada con la diagonal de 1 sea simétrica.

$$x_{ij}^{c} = x_{ji}^{c}, \qquad \forall\ 1 \leq i, j \leq t, 1 \leq c \leq n \tag{2.20}$$

La familia (2.21) de $r - 1$ restricciones fuerza a que la primera columna tenga colores incrementales.

$$1 + \sum_{c=1}^{n} c x_{i1}^{c} \leq \sum_{c=1}^{n} c x_{(i+1)1}^{c}, \qquad \forall\ 1 \leq i \leq r - 1 \tag{2.21}$$

Las implementaciones de estos modelos se pueden encontrar en los apéndices A.2 y A.4.

2.4. Resultados experimentales

Para las pruebas experimentales se utilizó el solucionador Gurobi ejecutándose en un servidor con dos CPU AMD Opteron 6174 corriendo a 2.2 GHz, con 128 GB de RAM y 24 núcleos en total. Se corrió Gurobi con los primeros dos modelos, sin restricciones adicionales, para el caso de 7×9 donde la función de Hopf-Stiefel da una cantidad de colores igual a $f(7, 9) = 15$. Se sabe que no existen contraejemplos con 14 colores para este caso, por lo que la conjetura se

cumple. En el modelo se especificó usar un máximo de 14 colores, esperando que la búsqueda de contraejemplos terminara sin encontrar alguno. Además se tomó $f = 6$ como frecuencia máxima de los colores mediante los lemas 1 y 2. Se permitió que Gurobi intentara determinar la infactibilidad del modelo por un tiempo aproximado de dos horas para cada modelo, sin que Gurobi pudiera terminar la búsqueda. Posteriormente se les agregaron las restricciones adicionales a cada modelo y se volvieron a correr con el mismo caso seleccionado por un tiempo aproximado de dos horas cada uno y tampoco fue capaz de terminar la búsqueda. Se concluyó que esta técnica no es la mejor opción para resolver los casos abiertos.

Capítulo 3

Matrices intercaladas óptimas

La construcción de matrices intercaladas óptimas se puede hacer usando un algoritmo de búsqueda con retroceso, el cual intenta construir la matriz intercalada celda por celda, probando todas las asignaciones válidas de colores a cada paso. El conjunto de colores válidos para una celda se puede calcular tomando en cuenta los colores asignados a celdas previas, de esta forma se evitan algunas ramas de la recursión y el algoritmo de búsqueda con retroceso es más rápido. Un color puede colocarse en una celda sólo si permite conservar la latinicidad y la intercalación de la matriz. El decidir si un color viola o no la latinicidad de una matriz T es sencillo, ya que basta mantener actualizadas dos tablas: una tabla F_T que indique si un color c ha sido o no usado en una fila i (y si sí, en qué columna j de esa fila), y una tabla C_T que indique si un color c ha sido usado en una columna j (y si sí, en qué fila i de esa columna). A continuación se muestra un ejemplo de matriz T y de sus dos tablas asociadas.

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 4 & 5 & 2 \end{pmatrix} \qquad F_T = \begin{pmatrix} 1 & 2 & - \\ 2 & 1 & 3 \\ 3 & - & - \\ - & 3 & 1 \\ - & - & 2 \end{pmatrix} \qquad C_T = \begin{pmatrix} 1 & 2 & - \\ 2 & 1 & 3 \\ - & - & 1 \\ 3 & - & 2 \\ - & 3 & - \end{pmatrix} \tag{3.1}$$

El decidir eficientemente si un color viola o no la intercalación de la matriz es bastante más complicado. En este capítulo se presentan tres algoritmos para conocer los colores válidos que pueden asignarse a la celda con coordenadas (i, j), donde se considera que una asignación de color es válida si no afecta a las submatrices de tamaño 2×2 formadas por las asignaciones previas. Todos los algoritmos suponen que la matriz se llena por filas de arriba para abajo, y que cada fila se llena de izquierda a derecha.

3.1. Determinación de colores asignables en una celda

Cualquier algoritmo de búsqueda con retroceso que intente encontrar una matriz intercalada de tipo $[r, s, n]$ deberá probar hasta n colores distintos en cada una de las rs entradas de la matriz, es decir, su tiempo de ejecución será $O(n^{rs})$. Sin embargo, conviene decidir lo más rápido posible cuáles de los n colores no se deben intentar en cada entrada. A continuación describimos tres algoritmos para esto. En los ejemplos que se muestran a continuación, es posible que un color sea evidentemente inválido según el criterio de latinicidad de la matriz, pero para poder usar ejemplos pequeños, la explicación se concentra en detectar si un color es válido según el criterio de intercalación.

3.1.1. Algoritmo en tiempo cúbico

Este algoritmo consiste en revisar, para color desde 1 hasta n, si todas las submatrices de tamaño 2×2 que tengan su esquina inferior derecha en (i, j) son compatibles con el color a considerar. Para ver más detalladamente el funcionamiento de este algoritmo, se puede ver un ejemplo sobre la celda que está marcada con un ? en la siguiente matriz T.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & ? & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.2}$$

Se empieza revisando el 1 y después de esto se revisa cada submatriz de tamaño 2×2 para ver si el 1 afecta las asignaciones previas.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & 1 & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.3}$$

En este caso se puede ver que, además de violar la latinicidad de la matriz, hay al menos una submatriz que se ve afectada con dicha asignación, es decir, que la submatriz no es una intercalación y tampoco una cointercalación, por lo que se descarta este color y se prueba con el siguiente. Al finalizar, el algoritmo obtiene una lista de colores válidos para asignar en la celda. Para conocer la complejidad de este algoritmo, se puede notar que hay

$O(ij)$ contraesquinas posibles de la submatriz y que se revisan $O(n)$ colores, por lo que este algoritmo tiene una complejidad de $O(ijn)$.

3.1.2. Algoritmo en tiempo cuadrático

Este algoritmo también consiste en revisar, para color desde 1 hasta n, si todas las submatrices de tamaño 2×2 que tengan su esquina inferior derecha en (i, j) son compatibles con el color a considerar. Sin embargo, este algoritmo busca ser más eficiente haciendo uso de las tablas auxiliares que permiten conocer en qué columna de una fila o en qué fila de una columna se encuentra un color.

Para ver más detalladamente el funcionamiento de este algoritmo, se supondrá que se está decidiendo qué color colocar en la celda marcada con un ? de la matriz T.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & ? & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.4}$$

Se comenzará revisando si el color 1 es válido para la celda $(3, 4)$. Para hacer esto, comenzaremos a examinar los elementos que están en la fila 3. La primera celda de esa fila tiene el color 3 y se revisa si en la columna 4 también aparece. Como esto ocurre, se revisa la contraesquina complementaria $(3, 1)$ para verificar que ahí también haya un 1. Como el color de esa celda es diferente, se formaría una intercalación errónea y se descarta el color 1. Es decir, el color 1 viola la propiedad de intercalación de la matriz, independientemente de que también violaba la laticinidad de la matriz.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & 1 & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.5}$$

Ahora se revisará si el color 2 es válido para la celda $(3, 4)$. Para hacer esto, comenzaremos a examinar los elementos que están en la fila 3. La primera celda de esa fila tiene el color 3 y se revisa si en la columna 4 también aparece. Como esto ocurre, se revisa la contraesquina complementaria para verificar que ahí también haya un 2. Como el color de esa celda si es

un 2, la revisión continúa.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & 2 & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.6}$$

La siguiente celda de la fila 3 tiene un 4 y se revisa si en la columna 4 también aparece. Como esto ocurre, se revisa la contraesquina complementaria $(1, 2)$ para verificar que ahí también haya un 2. Como el color de esa celda si es un 2, la revisión continúa.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & 2 & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.7}$$

La siguiente celda de la fila 3 tiene un 1 y se revisa si en la columna 4 también aparece. Como esto no ocurre, entonces las submatrices con esas esquinas inferiores deberán tener todos sus colores distintos. Si se desea colocar un 2 en la celda $(3, 4)$, entonces éste no debe aparecer en la columna 3. Como es cierto que el 2 no aparece en esa columna, la revisión continuaría, pero como ya se examinaron todas las celdas de la fila 3, entonces el 2 sí es un color válido en la celda $(3, 4)$.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & 2 & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.8}$$

Al finalizar, el algoritmo obtiene una lista de colores válidos para asignar en la celda. Para conocer la complejidad de este algoritmo, se observa que hay $j-1$ elementos previos en la fila i y que se revisan $O(n)$ colores, por lo que este algoritmo tiene una complejidad de $O(jn)$.

3.1.3. Algoritmo en tiempo lineal

Este algoritmo es similar al anterior, pero en lugar de ejecutar una revisión para cada color individual, el algoritmo calcula todos los colores factibles en una única revisión. El algoritmo

supone que el valor máximo de n es comparable con el tamaño del entero de la computadora (por ejemplo, de 64 bits) y que entonces es posible realizar operaciones de conjuntos de colores en tiempo constante con arreglos de bits. El algoritmo parte de un conjunto de colores disponibles, el cual representado como un arreglo de bits D, lo actualiza durante la única revisión que ejecuta y entrega el conjunto final.

Para ver más detalladamente el funcionamiento de este algoritmo, se supondrá que se está decidiendo qué color colocar en la celda marcada con un ? de la matriz T.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & ? & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.9}$$

$$D = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Comenzaremos a examinar los elementos que están en la fila i. La primera celda de esa fila tiene el color 3 y se revisa si en la columna j también aparece. Como esto ocurre, se revisa la contraesquina complementaria. Como el color de esa celda es 2, entonces la intercalación obliga a que la celda (i, j) deba tener el color 2.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & ? & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.10}$$

$$D = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

La siguiente celda de la fila i tiene un 4 y se revisa si en la columna j también aparece. Como esto ocurre, se revisa la contraesquina complementaria. Como el color de esa celda es

2, entonces la intercalación obliga a que la celda (i, j) deba tener el color 2.

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & ? & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.11}$$

$$D = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

La siguiente celda de la fila i tiene un 1 y se revisa si en la columna j también aparece. Como esto no ocurre, entonces las submatrices con esas esquinas inferiores deberá tener todos sus colores distintos. Ningún color de la columna 3 puede aparecer en la celda (i, j).

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & ? & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix} \tag{3.12}$$

$$D = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Como ya se examinaron todas las celdas de la fila i, el algoritmo termina. Para conocer la complejidad de este algoritmo, se observa que hay j elementos previos en la fila i, por lo que este algoritmo tiene una complejidad de $O(j)$.

3.2. Prevención de isomorfismo

Es posible intentar reducir la cantidad de matrices que se intentan generar usando las siguientes ideas:

1. Forzar a que ningún otro color aparezca más veces que el color 1. Esto es válido porque algún color debe ser el de máxima cardinalidad y los colores se pueden permutar para que el 1 sea el que tenga esta propiedad.

2. Forzar a que el color 1 sólo pueda aparecer en la diagonal, a partir de la esquina superior izquierda de la matriz. A su vez, esto forza a que la submatriz cuadrada con la diagonal

llena de 1 sea simétrica. Esto se puede hacer porque se pueden permutar las filas y columnas de la matriz y obtener una isomorfa.

3. Forzar a que los colores nuevos se utilicen por primera vez en orden $1, 2, 3$, etc. Esto implica que la primera fila tendrá colores incrementales consecutivos. Esto se puede hacer porque los colores se pueden permutar para que tengan esta propiedad.

4. Forzar a que la primera columna tenga colores incrementales. Si una matriz que cumpla los tres puntos anteriores no cumpliera que la primera columna es incremental, entonces se pueden permutar las dos primeras filas que violen dicha propiedad, para después reenumerar nuevamente los colores de acuerdo a la regla anterior. Después de una cantidad finita de estas correcciones, la matriz resultante cumplirá la propiedad deseada en la primera columna [11].

3.2.1. Resultados

Al ejecutar el algoritmo de búsqueda con retroceso, se obtiene un primer resultado para el primer caso abierto, que es la matriz de tamaño 9×17. Por los argumentos teóricos planteados en la sección 1.2, sólo se necesita determinar si existen matrices intercaladas válidas para los tipos extendidos $[9, 17, 24, 7]$ y $[9, 17, 24, 8]$. El cuadro 3.1 muestra una comparativa de tiempos de ejecución del algoritmo de búsqueda con retroceso usando cada algoritmo de verificación de asignaciones válidas. Al finalizar la ejecución, se prueba que la conjetura de Yuzvinsky es cierta para este caso al no encontrar matrices factibles.

r	s	n	t	Cúbico	Cuadrático	Lineal
9	17	24	7	-	15437 s	2696 s
9	17	24	8	114 s	44 s	8 s

Tabla 3.1: Tiempos de búsqueda para matrices de tamaño 9×17 con 24 colores.

3.3. Búsqueda concurrente

La construcción de matrices intercaladas óptimas se puede hacer combinando búsqueda en amplitud y búsqueda en profundidad durante la búsqueda con retroceso para explorar el árbol de búsqueda. A esto lo llamamos el algoritmo híbrido [A.5].

Este algoritmo primero explora de forma iterativa los primeros niveles del árbol de búsqueda. Los nodos encontrados los va metiendo a una cola, para extraerlos después y procesarlos.

Cuando la cola ya tiene suficientes elementos, entonces el algoritmo cambia de estrategia y explora concurrentemente los subárboles enraizados por esos nodos, donde cada árbol se explora de forma recursiva.

La figura (3.1) muestra una parte del árbol de búsqueda para matrices 5×5. Los espacios en blanco son celdas de la matriz que faltan por decidir. Cada nodo hereda todas las decisiones (asignaciones anteriores) que ya se tomaron en el nodo precursor. Un nodo que es sucesor directo toma una decisión sobre la siguiente celda a considerar.

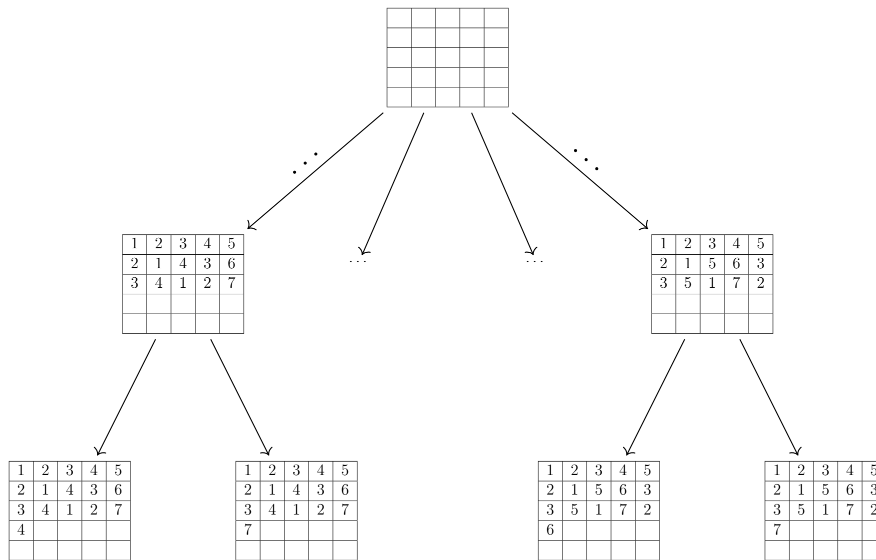

Figura 3.1: Exploración del árbol de búsqueda.

En la figura (3.2) se puede observar que el algoritmo comienza explorando el árbol piso a piso. Desde cada nodo se mete a todos sus sucesores a la cola; posteriormente se sacan y se procesan, todo este proceso es de forma iterativa. Como los pisos cada vez son más grandes, la cola de nodos pendientes crece cada vez más.

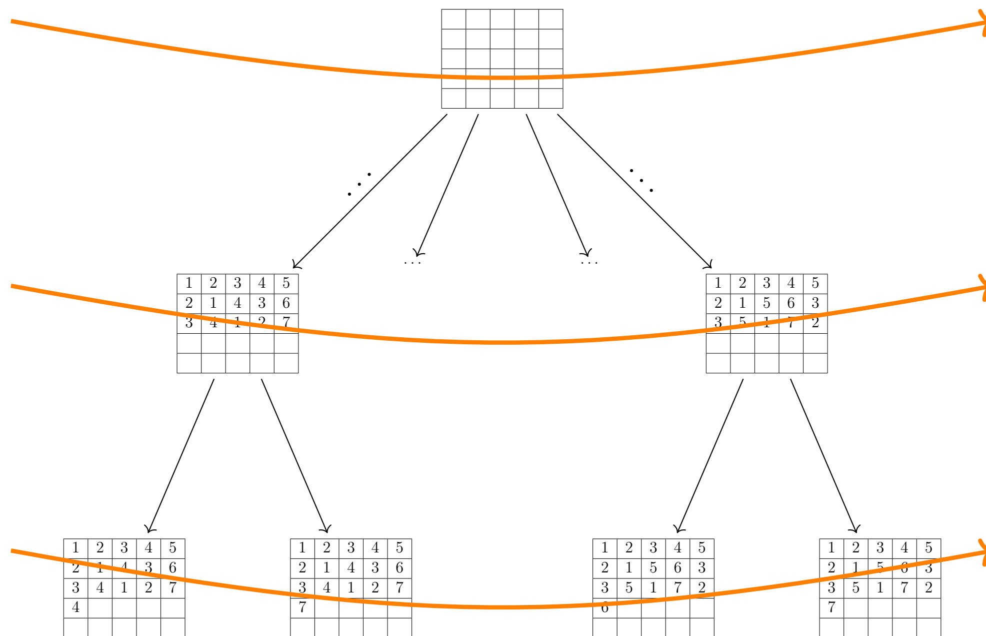

Figura 3.2: Exploración en amplitud del árbol de búsqueda.

En la figura (3.3) se puede observar que cuando la cola ya está muy llena, entonces se detiene el algoritmo iterativo y se crean hilos para explorar los nodos pendientes de forma recursiva. Conforme los hilos van explorando los nodos que les tocaron, la cola se va vaciando. El algoritmo termina cuando la cola se vacía y todos los hilos terminan. Experimentalmente, se cambia de búsqueda en amplitud a búsqueda en profundidad cuando la cola tiene algunas decenas de miles de nodos. La biblioteca Intel Threading Building Blocks (TBB) [7] para C++ se encarga de asignarles nodos a los hilos.

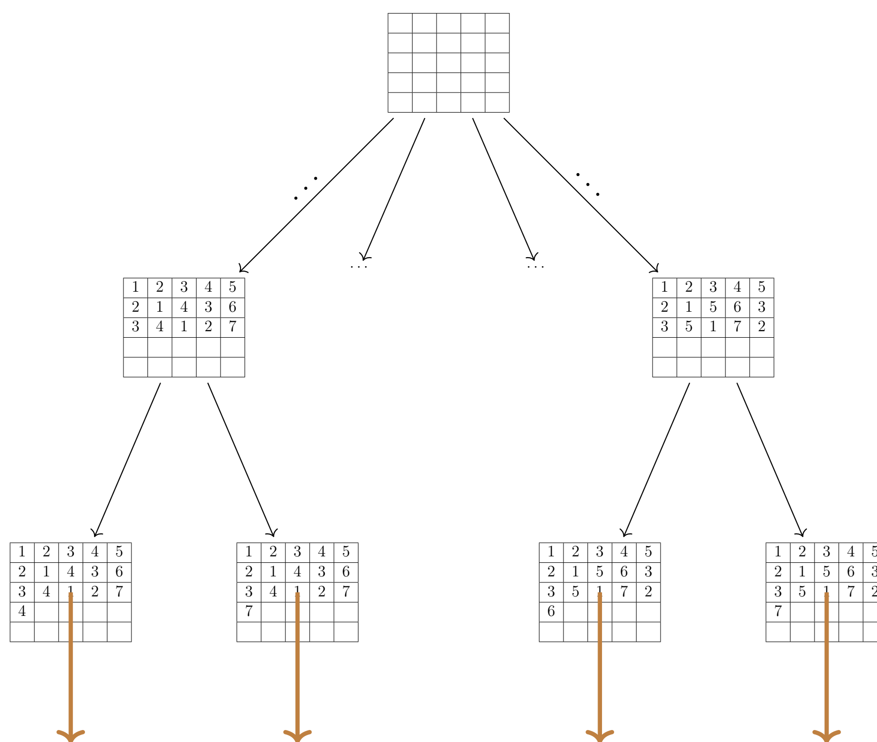

Figura 3.3: Exploración en profundidad del árbol de búsqueda.

3.4. Resultados experimentales

En la tabla 3.2 se listan los casos abiertos y se indica cuáles son los tipos extendidos de matrices intercaladas para los que se necesita determinar computacionalmente la existencia o inexistencia de una matriz con $n = r \circ s - 1$ colores. Cuando se requiera comprobar computacionalmente esto último, se indica el tiempo que toma probar que la conjetura de Yuzvinsky es cierta utilizando el algoritmo descrito en este capítulo.

La notación en la tabla 3.2 es la siguiente. Una f indica que el valor de t es inválido por alguno de los lemas 1 o 2, mientras que una h indica que el valor de t es inválido por el teorema 2. El resto de los casos deben revisarse computacionalmente. Cuando el algoritmo logró completar el cálculo, se indica el tiempo requerido para determinar que la conjetura de Yuzvinsky es cierta al no haber un contraejemplo. Los casos que deben revisarse, pero para los que el algoritmo no terminó en un tiempo razonable, se marcan con ★.

Las ejecuciones se realizaron en una computadora multiprocesador con dos CPU AMD Opteron 6174 corriendo a 2.2 GHz. La computadora tiene 128 GB de RAM y 24 núcleos.

r	s	rs	n	rs/n	$t=7$	$t=8$	$t=9$	$t=10$	$t=11$	$t=12$	$t=13$	$t=14$	$t=15$
9	17	153	24	6.38	381 s	2 s	h	f	f	f	f	f	f
9	18	162	25	6.48	733 s	14 s	h	f	f	f	f	f	f
10	17	170	25	6.80	358468 s	3526 s	h	h	f	f	f	f	f
9	19	171	26	6.58	11519 s	366 s	h	f	f	f	f	f	f
11	17	187	26	7.19	f	31588 s	h	984 s	h	f	f	f	f
9	20	180	27	6.67	52513 s	2664 s	h	f	f	f	f	f	f
10	19	190	27	7.04	f	★	h	h	f	f	f	f	f
11	18	198	27	7.33	f	★	h	h	h	f	f	f	f
12	17	204	27	7.56	f	★	h	h	h	h	f	f	f
9	21	189	28	6.75	2953120 s	58159 s	h	f	f	f	f	f	f
13	17	221	28	7.89	f	★	h	★	★	★	h	f	f
9	22	198	29	6.83	★	★	h	f	f	f	f	f	f
10	21	210	29	7.24	f	★	h	h	f	f	f	f	f
13	18	234	29	8.07	f	f	h	h	★	★	h	f	f
14	17	238	29	8.21	f	f	h	h	★	★	h	h	f
9	23	207	30	6.90	h	★	h	f	f	f	f	f	f
11	21	231	30	7.70	f	★	h	★	h	f	f	f	f
13	19	247	30	8.23	f	f	h	★	h	★	h	f	f
15	17	255	30	8.50	f	f	h	★	h	★	h	★	h

Tabla 3.2: Tiempos de búsqueda con algoritmo concurrente.

Capítulo 4

Matrices intercaladas simétricas

Una matriz simétrica es una matriz de tamaño $r \times r$, con la particularidad de que el valor de cada celda a_{ij} es igual al valor de la celda a_{ji}. Obsérvese que, debido a la propiedad de intercalación, los elementos de la diagonal $a_{11}, \ldots, a_{rr}$ deben ser todos iguales.

$$A = \begin{pmatrix} 1 & a_{12} & \ldots & a_{1r} \\ a_{12} & 1 & \ldots & a_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1r} & a_{2r} & \ldots & 1 \end{pmatrix} \tag{4.1}$$

4.1. Búsqueda concurrente

En principio, se puede usar el algoritmo híbrido explicado anteriormente para la generación de matrices simétricas de tamaño $r \times r$, agregando la restricción de simetría [A.6]. Un ejemplo de una matriz intercalada simétrica de tamaño 5×5 es:

$$A_s = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & 2 & 7 \\ 4 & 3 & 2 & 1 & 8 \\ 5 & 6 & 7 & 8 & 1 \end{pmatrix} \tag{4.2}$$

Adicionalmente, se puede modificar el algoritmo híbrido para que use las matrices simétricas de tamaño $(r-1) \times (r-1)$ previamente calculadas para generar las matrices simétricas de tamaño $r \times r$ [A.8]. Esto se hace colocando cada matriz precalculada en la esquina superior izquierda y procediendo a ejecutar el algoritmo híbrido para rellenar lo que falta, es decir,

asignar las celdas marcadas con un ? como se muestra a continuación:

$$\begin{pmatrix} 1 & a_{12} & a_{13} & \dots & a_{1(r-1)} & ? \\ a_{12} & 1 & a_{23} & \dots & a_{2(r-1)} & ? \\ a_{13} & a_{23} & 1 & \dots & a_{3(r-1)} & ? \\ \vdots & \vdots & \vdots & \ddots & \vdots & ? \\ a_{1(r-1)} & a_{2(r-1)} & a_{3(r-1)} & \dots & 1 & ? \\ ? & ? & ? & ? & ? & 1 \end{pmatrix} \tag{4.3}$$

Por ejemplo, si se desean generar todas las matrices simétricas de tamaño 5×5, se pueden utilizar las dos matrices simétricas de tamaño 4×4.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & ? \\ 2 & 1 & 4 & 3 & ? \\ 3 & 4 & 1 & 2 & ? \\ 4 & 3 & 2 & 1 & ? \\ ? & ? & ? & ? & 1 \end{pmatrix} \qquad A = \begin{pmatrix} 1 & 2 & 3 & 4 & ? \\ 2 & 1 & 5 & 6 & ? \\ 3 & 5 & 1 & 7 & ? \\ 4 & 6 & 7 & 1 & ? \\ ? & ? & ? & ? & 1 \end{pmatrix} \tag{4.4}$$

Durante la generación de matrices intercaladas simétricas se tiene interés en lo siguiente:

- Detectar isomorfismo en matrices simétricas.
- Usar las matrices simétricas únicas de tamaño $r \times r$ para calcular matrices intercaladas (no necesariamente simétricas) de tamaños mayores a $r \times r$.

4.1.1. Detección de isomorfismo

Una vez que se tiene la lista de las matrices intercaladas simétricas precalculadas, se necesita saber si hay matrices isomorfas y descartarlas si las hay. Para ello se utiliza un algoritmo de verificación de isomorfismo. El algoritmo de detección hace lo siguiente:

1. Se hace una clasificación de cada una de las matrices en grupos. Dicha clasificación se hace con base en el número de intercalaciones y la frecuencia de cada color (con un vector de frecuencias ordenado de menor a mayor, de modo que la clasificación sea insensible a renombramiento de colores).

2. Se ordenan las matrices de un mismo grupo lexicográficamente y luego se revisa isomorfismo entre las matrices de dicho grupo. Si una matriz es isomorfa a una matriz anterior de su mismo grupo, entonces se descarta.

Para la clasificación de una matriz se cuenta el número de intercalaciones y se contabiliza la frecuencia de cada color en un vector, para posteriormente ordenar tal vector de forma ascendente. A continuación se muestra una matriz de ejemplo con 14 intercalaciones y también se muestra su vector de frecuencias.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & 2 & 7 \\ 4 & 3 & 2 & 1 & 8 \\ 5 & 6 & 7 & 8 & 1 \end{pmatrix} \tag{4.5}$$

2	2	2	2	4	4	4	5
5	6	7	8	2	3	4	1

Cada matriz de un grupo se revisa contra las anteriores del mismo grupo. Por ejemplo:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 6 \\ 3 & 4 & 1 & 2 & 7 \\ 4 & 3 & 2 & 1 & 8 \\ 5 & 6 & 7 & 8 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 6 & 5 & 4 \\ 3 & 6 & 1 & 7 & 8 \\ 4 & 5 & 7 & 1 & 2 \\ 5 & 4 & 8 & 2 & 1 \end{pmatrix} \tag{4.6}$$

Como las matrices A y B no son iguales, se permutará la matriz B para intentar hacerla coincidir con A. Primero se permutan las columnas de B y la misma permutación se aplica a las filas de B para mantener la diagonal. Posteriormente se hace un renombramiento de colores de B, el cual se lleva a cabo leyendo el orden de aparición de colores en la matriz permutada B_p. Supongamos que la permutación que se desea es $(1,2,3,4,5) \rightarrow (3,1,4,2,5)$.

$$B_p = \begin{pmatrix} 1 & 3 & 7 & 6 & 8 \\ 3 & 1 & 4 & 2 & 5 \\ 7 & 4 & 1 & 5 & 2 \\ 6 & 2 & 5 & 1 & 4 \\ 8 & 5 & 2 & 4 & 1 \end{pmatrix} \tag{4.7}$$

El orden en que aparecen los colores es: 1, 3, 7, 6, 8, 4, 2, 5, por lo que la matriz permutada y con su respectivo renombramiento es:

$$B'_p = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 6 & 7 & 8 \\ 3 & 6 & 1 & 8 & 7 \\ 4 & 7 & 8 & 1 & 6 \\ 5 & 8 & 7 & 6 & 1 \end{pmatrix} \tag{4.8}$$

Esto se hace para todas las permutaciones posibles. Si alguna B'_p es idéntica a A, entonces A y B son isomorfas.

4.1.2. Resultados de las matrices intercaladas simétricas

En la tabla (4.1) se muestra la cantidad de matrices simétricas no isomorfas que encontró el matemático Yiu [10] y la cantidad de matrices simétricas no isomorfas que se encontraron computacionalmente.

Caso	Yiu	Algoritmo	Tiempo	Memoria aproximada
1	1	1	$<$ 1 seg	1 byte
2	1	1	$<$ 1 seg	4 bytes
3	1	1	$<$ 1 seg	9 bytes
4	2	2	$<$ 1 seg	32 bytes
5	2	2	$<$ 1 seg	150 bytes
6	3	3	$<$ 1 seg	540 bytes
7	5	5	$<$ 1 seg	2.2 Kilobytes
8	9	9	$<$ 1 seg	9.6 Kilobytes
9	13	13	$<$ 1 seg	42.6 Kilobytes
10	-	27	$<$ 1 seg	191.1 Kilobytes
11	-	54	$<$ 1 seg	1 Megabyte
12	-	139	6 seg	5.5 Megabytes
13	-	480	3 min	32.9 Megabytes
14	-	3409	1 hr	230.9 Megabytes
15	-	81526	18 hrs	2 Gigabytes

Tabla 4.1: Resultados de Yiu contra resultados del algoritmo híbrido.

Se puede ver que se calcularon las matrices simétricas de tamaño hasta 15×15. Cabe mencionar que una razón para calcular hasta ese tamaño es que, en los casos abiertos de la conjetura de Yuzvinsky hasta 32×32 con determinadas frecuencias, sólo hacen falta calcular las matrices de tamaño 15×15.

4.1.3. Generación de matrices intercaladas a partir de las simétricas

Para calcular una matriz intercalada no necesariamente cuadrada de tamaño $r \times s$ con frecuencia máxima t, se pueden utilizar las matrices simétricas de tamaño $t \times t$ [A.7]. Esto se puede hacer porque la frecuencia máxima de un color es t y se pone en la diagonal principal.

Para ver más claramente la idea, se puede ver la matriz A mostrada en la ecuación (4.9). Si se quieren calcular las matrices de tamaño 9×17 con frecuencia máxima 7, simplemente se toma cada una de las matrices simétricas de tamaño 7×7, se coloca en la esquina superior izquieda de la matriz más grande y se intenta asignar lo que falta con el algoritmo híbrido.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ 2 & 1 & 4 & 3 & 6 & 5 & 8 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ 3 & 4 & 1 & 2 & 7 & 8 & 5 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ 4 & 3 & 2 & 1 & 8 & 7 & 6 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ 5 & 6 & 7 & 8 & 1 & 2 & 3 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ 6 & 5 & 8 & 7 & 2 & 1 & 4 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ 7 & 8 & 5 & 6 & 3 & 4 & 1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \end{pmatrix} \quad (4.9)$$

4.2. Resultados experimentales

En la tabla (4.2) se muestra nuevamente la tabla (3.4) pero ahora usando el algoritmo híbrido que utiliza las matrices simétricas precalculadas. Los tiempos de ejecución bajaron drásticamente y se pudieron resolver más casos en un tiempo razonable.

r	s	rs	n	rs/n	$t=7$	$t=8$	$t=9$	$t=10$	$t=11$	$t=12$	$t=13$	$t=14$	$t=15$
9	17	153	24	6.38	4 s	0 s	h	f	f	f	f	f	f
9	18	162	25	6.48	7 s	0 s	h	f	f	f	f	f	f
10	17	170	25	6.80	7968 s	27 s	h	h	f	f	f	f	f
9	19	171	26	6.58	101 s	1 s	h	f	f	f	f	f	f
11	17	187	26	7.19	f	46780 s	h	0 s	h	f	f	f	f
9	20	180	27	6.67	478 s	8 s	h	f	f	f	f	f	f
10	19	190	27	7.04	f	448 s	h	h	f	f	f	f	f
11	18	198	27	7.33	f	67217 s	h	h	h	f	f	f	f
12	17	204	27	7.56	f	★	h	h	h	h	f	f	f
9	21	189	28	6.75	24223 s	195 s	h	f	f	f	f	f	f
13	17	221	28	7.89	f	★	h	★	31478 s	2 s	h	f	f
9	22	198	29	6.83	79295 s	574 s	h	f	f	f	f	f	f
10	21	210	29	7.24	f	207578 s	h	h	f	f	f	f	f
13	18	234	29	8.07	f	f	h	h	208165 s	103 s	h	f	f
14	17	238	29	8.21	f	f	h	h	★	72503 s	h	h	f
9	23	207	30	6.90	h	2977 s	h	f	f	f	f	f	f
11	21	231	30	7.70	f	★	h	1324 s	h	f	f	f	f
13	19	247	30	8.23	f	f	h	★	h	1627 s	h	f	f
15	17	255	30	8.50	f	f	h	★	h	★	h	230 s	h

Tabla 4.2: Tabla de tiempos usando matrices simétricas precalculadas.

Capítulo 5

Conclusiones

El cálculo de matrices intercaladas exige hacer uso de estrategias mucho más eficientes que la búsqueda exhaustiva a ciegas. Para intentar encontrar una matriz intercalada de tamaño $r \times s$ con a lo más n colores, una búsqueda ciega requeriría explorar exactamente n^{rs} posibilidades. El primer caso abierto era de tamaño 9×17 con 24 colores, por lo que se tendrían $24^{9\times 17} = 1{,}487031 \times 10^{211}$ posibilidades y se requerirían aproximadamente 4×10^{209} años de cálculo en una computadora moderna. Aunque resulta difícil determinar si la complejidad de un algoritmo de búsqueda con retroceso es asintóticamente mejor, sí es posible diseñar algoritmos que examinen una cantidad de matrices varios órdenes de magnitud por debajo del peor caso.

Como primera estrategia de solución se diseñaron e implementaron dos modelos de programación entera a resolverse con Gurobi. Durante los experimentos, los modelos se probaron con un caso de tamaño 7×9, que es más pequeño que el primer caso abierto, cabe mencionar que en este caso y para el que ya se sabe que conjetura de Yuzvinsky se cumple; se dejaron correr los modelos por más de dos horas cada uno y no pudieron resolvelo, por lo que se concluyó en no continuar con esta estrategia para resolver los casos abiertos.

Como una siguiente estrategia se diseñó un algoritmo de búsqueda con retroceso, pero como el espacio de búsqueda es muy grande, fue necesario tratar de hacer las asignaciones de color inteligentemente. Para ello se utilizó un algoritmo eficiente para conocer qué colores se podrían asignar en una celda sin afectar las asignaciones previas. Aunque con este algoritmo se resolvió al menos el primer caso abierto de tamaño 9×17, fue necesario utilizar otra estrategia para reducir el tiempo de búsqueda para matrices más grandes.

Como una siguiente estrategia para acelerar el tiempo de búsqueda, se diseñó un algoritmo híbrido que combina búsqueda en amplitud y búsqueda en profundidad para poder hacer uso de la concurrencia del hardware. Esta estrategia logró acelerar el algoritmo y se resolvieron siete casos abiertos, que son 9×17, 9×18, 10×17, 9×19, 11×17, 9×20 y 9×21.

Haciendo la observación de que un color con frecuencia máxima t puede aparecer en diagonal principal y que esto forza una matriz simétrica, se pueden precalcular todas las matrices simétricas de tamaño $t \times t$ para reutilizarlas al construir matrices más grandes. Esto reduce el tiempo de búsqueda porque solo se va llenando lo que falta. Añadiendo esta estrategia al algoritmo híbrido se logró reducir el tiempo en los primeros siete casos resueltos con el algoritmo híbrido que no usa esta estrategia y además se lograron resolver los casos 9×22, 10×19, 11×18, 10×21, 13×18 y 9×23, finalizando con trece casos resueltos y quedando seis casos abiertos que son:

- $r = 11$ y $s = 21$
- $r \in \{12, 14, 15\}$ y $s = 17$
- $r = 13$ y $s \in \{17, 19\}$

Se puede concluir que utilizar concurrencia en el algoritmo de búsqueda es una excelente opción para poder mejorar el rendimiento, pero la reutilización de trabajo previo también puede acelerar considerablemente la búsqueda de las matrices. Con estas ideas se podría diseñar alguna nueva estrategia que sea capaz de revisar los casos faltantes.

Apéndice A

Código fuente de los algoritmos implementados

A.1. Implementación del modelo 1 de programación entera lineal utilizando la interfaz de programación de C++ de Gurobi

```
#include <bit>
#include <array>
#include <cstdio>
#include <string>
#include <vector>
#include <iostream>
#include <gurobi_c++.h>

template<typename... T>
std::string formato(const char* s, const T&... v) {
   char bufer[32 + 1];
   sprintf(bufer, s, v...);
   return std::string(bufer);
}

constexpr int f(unsigned r, unsigned s) {
```

```
   if (r > s) {
      return f(s, r);
   } else if (r == 1) {
      return s;
   } else if (int st = std::bit_ceil(s); st / 2 < r) {
      return st;
   } if (int rt = std::bit_ceil(r); rt < s) {
      return rt + f(r, s - rt);
   }
   return -1;
}

int main( ) try {
   // lectura de la entrada
   int r, s, delta; std::cin >> r >> s >> delta;
   int n = f(r, s) + delta;

   // construcción del modelo
   GRBEnv ambiente;
   GRBModel modelo(ambiente);

   GRBVar x[r][s][n];
   for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         for (int c = 0; c < n; ++c) {
            x[i][j][c] = modelo.addVar(0, 1, 0, GRB_BINARY, formato("x%d_%
   d_%d", i, j, c));
         }
      }
   }

   GRBVar y[r][s][r][s][n];
   for (int i1 = 0; i1 < r - 1; ++i1) {
      for (int j1 = 0; j1 < s - 1; ++j1) {
         for (int i2 = i1 + 1; i2 < r; ++i2) {
```

```
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               for (int c = 0; c < n; ++c) {
                  y[i1][j1][i2][j2][c] = modelo.addVar(0, 1, 0, GRB_BINARY,
formato("y%d_%d_%d_%d_%d", i1, j1, i2, j2, c));
               }
            }
         }
      }
   }

   GRBVar k[r][s][r][s];
   for (int i1 = 0; i1 < r - 1; ++i1) {
      for (int j1 = 0; j1 < s - 1; ++j1) {
         for (int i2 = i1 + 1; i2 < r; ++i2) {
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               k[i1][j1][i2][j2] = modelo.addVar(1, 2, 0, GRB_INTEGER,
formato("k%d_%d_%d_%d_%d", i1, j1, i2, j2));
            }
         }
      }
   }

   // Cada celda tiene exactamente un color
   for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         GRBLinExpr suma;
         for (int c = 0; c < n; ++c) {
            suma += x[i][j][c];
         }
         modelo.addConstr(suma == 1);
      }
   }

   // Cada color aparece a lo mas una vez por fila
   for (int i = 0; i < r; ++i) {
```

```
    for (int c = 0; c < n; ++c) {
       GRBLinExpr suma;
       for (int j = 0; j < s; ++j) {
          suma += x[i][j][c];
       }
       modelo.addConstr(suma <= 1);
    }
 }

 // Cada color aparece a lo mas una vez por columna
 for (int j = 0; j < s; ++j) {
    for (int c = 0; c < n; ++c) {
       GRBLinExpr suma;
       for (int i = 0; i < r; ++i) {
          suma += x[i][j][c];
       }
       modelo.addConstr(suma <= 1);
    }
 }

 // Detectar si un color se usó en una submatriz
 for (int i1 = 0; i1 < r - 1; ++i1) {
    for (int j1 = 0; j1 < s - 1; ++j1) {
       for (int i2 = i1 + 1; i2 < r; ++i2) {
          for (int j2 = j1 + 1; j2 < s; ++j2) {
             for (int c = 0; c < n; ++c) {
                modelo.addGenConstrOr(y[i1][j1][i2][j2][c], std::array{ x[
 i1][j1][c], x[i1][j2][c], x[i2][j1][c], x[i2][j2][c] }.data( ), 4);
             }
          }
       }
    }
 }

 // La cantidad de colores distintos en una submatriz debe ser par
```

```
   for (int i1 = 0; i1 < r - 1; ++i1) {
      for (int j1 = 0; j1 < s - 1; ++j1) {
         for (int i2 = i1 + 1; i2 < r; ++i2) {
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               GRBLinExpr suma;
               for (int c = 0; c < n; ++c) {
                  suma += y[i1][j1][i2][j2][c];
               }
               modelo.addConstr(suma == 2 * k[i1][j1][i2][j2]);
            }
         }
      }
   }

   modelo.write(formato("modelo1_%d_%d_%d.lp", r, s, delta));
   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
      modelo.write(formato("modelo1_%d_%d_%d.sol", r, s, delta));
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Status no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}

// g++ -std=c++20 -O3 modelo1.cpp -lgurobi_c++ -lgurobi90 -o modelo1
// ./modelo1
```

A.2. Implementación del modelo 1 de programación entera lineal con las restricciones adicionales utilizando la interfaz de programación de C++ de Gurobi

```
/*
   Segunda version del modelo 1, se agregan frecuencias, 1ra fila, 1ra
   columna, diagonal de ceros y submatriz simétrica
*/

#include <bit>
#include <array>
#include <cstdio>
#include <string>
#include <vector>
#include <iostream>
#include <gurobi_c++.h>

template<typename... T>
std::string formato(const char* s, const T&... v) {
   char bufer[32 + 1];
   sprintf(bufer, s, v...);
   return std::string(bufer);
}

constexpr int f(unsigned r, unsigned s) {
   if (r > s) {
      return f(s, r);
   } else if (r == 1) {
      return s;
   } else if (int st = std::bit_ceil(s); st / 2 < r) {
      return st;
   } if (int rt = std::bit_ceil(r); rt < s) {
      return rt + f(r, s - rt);
   }
```

```
    return -1;
}

int main( ) try {
    // lectura de la entrada
    int r, s, delta, frecuencia; std::cin >> r >> s >> delta >> frecuencia;
    int n = f(r, s) + delta;

    if (s != std::min(r, s)) {
        std::swap(r, s);
    }
    if (frecuencia > std::min(r, s)) {
        std::cout << "La frecuencia supera la dimension mas chica\n";
        return 0;
    }

    // construcción del modelo
    GRBEnv ambiente;
    GRBModel modelo(ambiente);

    GRBVar x[r][s][n];
    for (int i = 0; i < r; ++i) {
        for (int j = 0; j < s; ++j) {
            for (int c = 0; c < n; ++c) {
                x[i][j][c] = modelo.addVar(0, 1, 0, GRB_BINARY, formato("x%d_%
    d_%d", i, j, c));
            }
        }
    }

    GRBVar y[r][s][r][s][n];
    for (int i1 = 0; i1 < r - 1; ++i1) {
        for (int j1 = 0; j1 < s - 1; ++j1) {
            for (int i2 = i1 + 1; i2 < r; ++i2) {
                for (int j2 = j1 + 1; j2 < s; ++j2) {
```

```
            for (int c = 0; c < n; ++c) {
               y[i1][j1][i2][j2][c] = modelo.addVar(0, 1, 0, GRB_BINARY,
formato("y%d_%d_%d_%d_%d", i1, j1, i2, j2, c));
            }
         }
      }
   }
}

GRBVar k[r][s][r][s];
for (int i1 = 0; i1 < r - 1; ++i1) {
   for (int j1 = 0; j1 < s - 1; ++j1) {
      for (int i2 = i1 + 1; i2 < r; ++i2) {
         for (int j2 = j1 + 1; j2 < s; ++j2) {
            k[i1][j1][i2][j2] = modelo.addVar(1, 2, 0, GRB_INTEGER,
formato("k%d_%d_%d_%d_%d", i1, j1, i2, j2));
         }
      }
   }
}

// Cada celda tiene exactamente un color
for (int i = 0; i < r; ++i) {
   for (int j = 0; j < s; ++j) {
      GRBLinExpr suma;
      for (int c = 0; c < n; ++c) {
         suma += x[i][j][c];
      }
      modelo.addConstr(suma == 1);
   }
}

// Cada color aparece a lo mas una vez por fila
for (int i = 0; i < r; ++i) {
   for (int c = 0; c < n; ++c) {
```

```
      GRBLinExpr suma;
      for (int j = 0; j < s; ++j) {
         suma += x[i][j][c];
      }
      modelo.addConstr(suma <= 1);
   }
}

// Cada color aparece a lo mas una vez por columna
for (int j = 0; j < s; ++j) {
   for (int c = 0; c < n; ++c) {
      GRBLinExpr suma;
      for (int i = 0; i < r; ++i) {
         suma += x[i][j][c];
      }
      modelo.addConstr(suma <= 1);
   }
}

// Detectar si un color se usó en una submatriz
for (int i1 = 0; i1 < r - 1; ++i1) {
   for (int j1 = 0; j1 < s - 1; ++j1) {
      for (int i2 = i1 + 1; i2 < r; ++i2) {
         for (int j2 = j1 + 1; j2 < s; ++j2) {
            for (int c = 0; c < n; ++c) {
               modelo.addGenConstrOr(y[i1][j1][i2][j2][c], std::array{ x[
i1][j1][c], x[i1][j2][c], x[i2][j1][c], x[i2][j2][c] }.data( ), 4);
            }
         }
      }
   }
}

// La cantidad de colores distintos en una submatriz debe ser par
for (int i1 = 0; i1 < r - 1; ++i1) {
```

```
        for (int j1 = 0; j1 < s - 1; ++j1) {
            for (int i2 = i1 + 1; i2 < r; ++i2) {
                for (int j2 = j1 + 1; j2 < s; ++j2) {
                    GRBLinExpr suma;
                    for (int c = 0; c < n; ++c) {
                        suma += y[i1][j1][i2][j2][c];
                    }
                    modelo.addConstr(suma == 2 * k[i1][j1][i2][j2]);
                }
            }
        }
    }

    // Ningun color puede sobrepasar la frecuencia dada
    for (int c = 0; c < n; ++c) {
        GRBLinExpr suma;
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < s; ++j) {
                suma += x[i][j][c];
            }
        }
        modelo.addConstr(suma <= frecuencia);
    }

    // Forzar a que la primera fila tenga colores 0, 1, 2, etc
    for (int j = 0; j < s; ++j) {
        modelo.addConstr(x[0][j][j] == 1);
    }

    // Forzar a que el 0 aparezca en la diagonal con la frecuencia dada
    for (int i = 0; i < frecuencia; ++i) {
        modelo.addConstr(x[i][i][0] == 1);
    }

    // Forzar a que la submatriz cuadrada con la diagonal llena de 0 sea simé
```

```
   trica
   for (int i = 0; i < frecuencia; ++i) {
      for (int j = 0; j < frecuencia; ++j) {
         for (int c = 0; c < n; ++c) {
            modelo.addConstr(x[i][j][c] == x[j][i][c]);
         }
      }
   }

   // Forzar a que la primera columna tenga colores incrementales
   GRBLinExpr previo = 0;
   for (int i = 0; i < r; ++i) {
      GRBLinExpr actual;
      for (int c = 0; c < n; ++c) {
         actual += c * x[i][0][c];
      }
      modelo.addConstr(previo <= actual);
      previo = std::move(actual);
   }

   modelo.write(formato("modelo1x_%d_%d_%d.lp", r, s, delta));
   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
      modelo.write(formato("modelo1x_%d_%d_%d.sol", r, s, delta));
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Status no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}
```

```
// g++ -std=c++20 -O3 modelo1x.cpp -lgurobi_c++ -lgurobi90 -o modelo1x
// ./modelo1x
```

A.3. Implementación del modelo 2 de programación entera lineal utilizando la interfaz de programación de C++ de Gurobi

```
#include <bit>
#include <array>
#include <cstdio>
#include <string>
#include <vector>
#include <iostream>
#include <gurobi_c++.h>

template<typename... T>
std::string formato(const char* s, const T&... v) {
   char bufer[32 + 1];
   sprintf(bufer, s, v...);
   return std::string(bufer);
}

constexpr int f(unsigned r, unsigned s) {
   if (r > s) {
      return f(s, r);
   } else if (r == 1) {
      return s;
   } else if (int st = std::bit_ceil(s); st / 2 < r) {
      return st;
   } if (int rt = std::bit_ceil(r); rt < s) {
      return rt + f(r, s - rt);
   }
   return -1;
}
```

```
int main( ) try {
   // lectura de la entrada
   int r, s, delta; std::cin >> r >> s >> delta;
   int n = f(r, s) + delta;

   // construcción del modelo
   GRBEnv ambiente;
   GRBModel modelo(ambiente);

   GRBVar x[r][s][n];
   for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         for (int c = 0; c < n; ++c) {
            x[i][j][c] = modelo.addVar(0, 1, 0, GRB_BINARY, formato("x%d_%
   d_%d", i, j, c));
         }
      }
   }

   GRBVar k[r][s][r][s];
   for (int i1 = 0; i1 < r - 1; ++i1) {
      for (int j1 = 0; j1 < s - 1; ++j1) {
         for (int i2 = i1 + 1; i2 < r; ++i2) {
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               k[i1][j1][i2][j2] = modelo.addVar(0, 1, 0, GRB_BINARY,
   formato("k%d_%d_%d_%d_%d", i1, j1, i2, j2));
            }
         }
      }
   }

   // Cada celda tiene exactamente un color
   for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
```

```
        GRBLinExpr suma;
        for (int c = 0; c < n; ++c) {
            suma += x[i][j][c];
        }
        modelo.addConstr(suma == 1);
    }
}

// Cada color aparece a lo mas una vez por fila
for (int i = 0; i < r; ++i) {
    for (int c = 0; c < n; ++c) {
        GRBLinExpr suma;
        for (int j = 0; j < s; ++j) {
            suma += x[i][j][c];
        }
        modelo.addConstr(suma <= 1);
    }
}

// Cada color aparece a lo mas una vez por columna
for (int j = 0; j < s; ++j) {
    for (int c = 0; c < n; ++c) {
        GRBLinExpr suma;
        for (int i = 0; i < r; ++i) {
            suma += x[i][j][c];
        }
        modelo.addConstr(suma <= 1);
    }
}

// Si la diagonal y la contradiagonal tiene el mismo color, se obliga a
 que k se prenda
for (int i1 = 0; i1 < r - 1; ++i1) {
    for (int j1 = 0; j1 < s - 1; ++j1) {
        for (int i2 = i1 + 1; i2 < r; ++i2) {
```

```
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               for (int c = 0; c < n; ++c) {
                  modelo.addConstr(x[i1][j1][c] + x[i2][j2][c] <= 1 + k[i1][
j1][i2][j2]);
                  modelo.addConstr(x[i1][j2][c] + x[i2][j1][c] <= 1 + k[i1][
j1][i2][j2]);
               }
            }
         }
      }
   }

   // Si la diagonal o la contradiagonal tienen colores distintos, se obliga
    a que k se apague
   for (int i1 = 0; i1 < r - 1; ++i1) {
      for (int j1 = 0; j1 < s - 1; ++j1) {
         for (int i2 = i1 + 1; i2 < r; ++i2) {
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               for (int c = 0; c < n; ++c) {
                  GRBLinExpr diag1 = x[i1][j1][c];
                  GRBLinExpr diag2 = x[i1][j2][c];
                  for (int d = 0; d < n; ++d) {
                     if (c != d) {
                        diag1 += x[i2][j2][d];
                        diag2 += x[i2][j1][d];
                     }
                  }
                  modelo.addConstr(diag1 <= 2 - k[i1][j1][i2][j2]);
                  modelo.addConstr(diag2 <= 2 - k[i1][j1][i2][j2]);
               }
            }
         }
      }
   }
```

```
   modelo.write(formato("modelo2_%d_%d_%d.lp", r, s, delta));
   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
      modelo.write(formato("modelo2_%d_%d_%d.sol", r, s, delta));
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Status no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}

// g++ -std=c++20 -O3 modelo2.cpp -lgurobi_c++ -lgurobi90 -o modelo2
// ./modelo2
```

A.4. Implementación del modelo 2 de programación entera lineal con las restricciones adicionales utilizando la interfaz de programación de C++ de Gurobi

```
/*
   Segunda version del modelo 2, se agregan frecuencias, 1ra fila, 1ra
   columna, diagonal de ceros y submatriz simétrica
*/

#include <bit>
#include <array>
#include <cstdio>
#include <string>
#include <vector>
#include <iostream>
#include <gurobi_c++.h>
```

```
template<typename... T>
std::string formato(const char* s, const T&... v) {
   char bufer[32 + 1];
   sprintf(bufer, s, v...);
   return std::string(bufer);
}

constexpr int f(unsigned r, unsigned s) {
   if (r > s) {
      return f(s, r);
   } else if (r == 1) {
      return s;
   } else if (int st = std::bit_ceil(s); st / 2 < r) {
      return st;
   } if (int rt = std::bit_ceil(r); rt < s) {
      return rt + f(r, s - rt);
   }
   return -1;
}

int main( ) try {
   // lectura de la entrada
   int r, s, delta, frecuencia; std::cin >> r >> s >> delta >> frecuencia;
   int n = f(r, s) + delta;

   if (s != std::min(r, s)) {
      std::swap(r, s);
   }
   if (frecuencia > std::min(r, s)) {
      std::cout << "La frecuencia supera la dimension mas chica\n";
      return 0;
   }

   // construcción del modelo
```

```
GRBEnv ambiente;
GRBModel modelo(ambiente);

GRBVar x[r][s][n];
for (int i = 0; i < r; ++i) {
   for (int j = 0; j < s; ++j) {
      for (int c = 0; c < n; ++c) {
         x[i][j][c] = modelo.addVar(0, 1, 0, GRB_BINARY, formato("x%d_%
d_%d", i, j, c));
      }
   }
}

GRBVar k[r][s][r][s];
for (int i1 = 0; i1 < r - 1; ++i1) {
   for (int j1 = 0; j1 < s - 1; ++j1) {
      for (int i2 = i1 + 1; i2 < r; ++i2) {
         for (int j2 = j1 + 1; j2 < s; ++j2) {
            k[i1][j1][i2][j2] = modelo.addVar(0, 1, 0, GRB_BINARY,
formato("k%d_%d_%d_%d_%d", i1, j1, i2, j2));
         }
      }
   }
}

// Cada celda tiene exactamente un color
for (int i = 0; i < r; ++i) {
   for (int j = 0; j < s; ++j) {
      GRBLinExpr suma;
      for (int c = 0; c < n; ++c) {
         suma += x[i][j][c];
      }
      modelo.addConstr(suma == 1);
   }
}
```

```
// Cada color aparece a lo mas una vez por fila
for (int i = 0; i < r; ++i) {
   for (int c = 0; c < n; ++c) {
      GRBLinExpr suma;
      for (int j = 0; j < s; ++j) {
         suma += x[i][j][c];
      }
      modelo.addConstr(suma <= 1);
   }
}

// Cada color aparece a lo mas una vez por columna
for (int j = 0; j < s; ++j) {
   for (int c = 0; c < n; ++c) {
      GRBLinExpr suma;
      for (int i = 0; i < r; ++i) {
         suma += x[i][j][c];
      }
      modelo.addConstr(suma <= 1);
   }
}

// Si la diagonal y la contradiagonal tiene el mismo color, se obliga a
que k se prenda
for (int i1 = 0; i1 < r - 1; ++i1) {
   for (int j1 = 0; j1 < s - 1; ++j1) {
      for (int i2 = i1 + 1; i2 < r; ++i2) {
         for (int j2 = j1 + 1; j2 < s; ++j2) {
            for (int c = 0; c < n; ++c) {
               modelo.addConstr(x[i1][j1][c] + x[i2][j2][c] <= 1 + k[i1][
j1][i2][j2]);
               modelo.addConstr(x[i1][j2][c] + x[i2][j1][c] <= 1 + k[i1][
j1][i2][j2]);
            }
```

```
            }
         }
      }
   }

   // Si la diagonal o la contradiagonal tienen colores distintos, se obliga
    a que k se apague
   for (int i1 = 0; i1 < r - 1; ++i1) {
      for (int j1 = 0; j1 < s - 1; ++j1) {
         for (int i2 = i1 + 1; i2 < r; ++i2) {
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               for (int c = 0; c < n; ++c) {
                  GRBLinExpr diag1 = x[i1][j1][c];
                  GRBLinExpr diag2 = x[i1][j2][c];
                  for (int d = 0; d < n; ++d) {
                     if (c != d) {
                        diag1 += x[i2][j2][d];
                        diag2 += x[i2][j1][d];
                     }
                  }
                  modelo.addConstr(diag1 <= 2 - k[i1][j1][i2][j2]);
                  modelo.addConstr(diag2 <= 2 - k[i1][j1][i2][j2]);
               }
            }
         }
      }
   }

   // Ningun color puede sobrepasar la frecuencia dada
   for (int c = 0; c < n; ++c) {
      GRBLinExpr suma;
      for (int i = 0; i < r; ++i) {
         for (int j = 0; j < s; ++j) {
            suma += x[i][j][c];
         }
```

```
    }
    modelo.addConstr(suma <= frecuencia);
}

// Forzar a que la primera fila tenga colores 0, 1, 2, etc
for (int j = 0; j < s; ++j) {
    modelo.addConstr(x[0][j][j] == 1);
}

// Forzar a que el 0 aparezca en la diagonal con la frecuencia dada
for (int i = 0; i < frecuencia; ++i) {
    modelo.addConstr(x[i][i][0] == 1);
}

// Forzar a que la submatriz cuadrada con la diagonal llena de 0 sea simé
trica
for (int i = 0; i < frecuencia; ++i) {
    for (int j = 0; j < frecuencia; ++j) {
        for (int c = 0; c < n; ++c) {
            modelo.addConstr(x[i][j][c] == x[j][i][c]);
        }
    }
}

// Forzar a que la primera columna tenga colores incrementales
GRBLinExpr previo = 0;
for (int i = 0; i < r; ++i) {
    GRBLinExpr actual;
    for (int c = 0; c < n; ++c) {
        actual += c * x[i][0][c];
    }
    modelo.addConstr(previo <= actual);
    previo = std::move(actual);
}
```

```
   modelo.write(formato("modelo2x_%d_%d_%d.lp", r, s, delta));
   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
      modelo.write(formato("modelo2x_%d_%d_%d.sol", r, s, delta));
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Status no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}

// g++ -std=c++20 -O3 modelo2x.cpp -lgurobi_c++ -lgurobi90 -o modelo2x
// ./modelo2x
```

A.5. Implementación del algoritmo base de Backtracking

```
/**
Backtracking con parámetros en forma de macro. Backtracking base.
/**/

#include <algorithm>
#include <bit>
#include <bitset>
#include <chrono>
#include <deque>
#include <execution>
#include <iostream>
#include <math.h>
#include <stdint.h>
#include <string.h>

#if !defined(VALOR_R) || !defined(VALOR_S) || !defined(VALOR_DELTA) || !
```

```
    defined(VALOR_T)
    static_assert(false, "Deben definirse macros VALOR_R, VALOR_S,
    VALOR_DELTA, VALOR_T");
#else
    constexpr int f(unsigned r, unsigned s) {
        if (r > s) {
            return f(s, r);
        } else if (r == 1) {
            return s;
        } else if (int st = std::bit_ceil(s); st / 2 < r) {
            return st;
        } if (int rt = std::bit_ceil(r); rt < s) {
            return rt + f(r, s - rt);
        }
        return -1;
    }

    constexpr int r = std::max(VALOR_R, VALOR_S), s = std::min(VALOR_R,
    VALOR_S), n = f(r, s) + VALOR_DELTA, t = VALOR_T;      // preferir pocas
    columnas, sino se usarían muchos colores desde la primera fila

    struct datos {
        int8_t matriz[32][32], frecuencia[32] = { }, pos_color_en_columna
    [32][32];
        uint32_t prohibir_fila[32] = { }, prohibir_columna[32] = { },
    prohibir_frecuencia[32] = { }, prohibir_columna_acumulado[32] = { };

        datos( ) {
            std::fill(&pos_color_en_columna[0][0], &pos_color_en_columna
    [32][0], -1);
        }
    };

    template<int i, int j>
    auto prohibir_celda(datos& m) {
```

```
   if ((m.prohibir_fila[i] & m.prohibir_columna[j]) != 0) {
      uint32_t res = 0;
      for (int y = 0; y < j; ++y) {
         int8_t pos_color = m.pos_color_en_columna[j][m.matriz[i][y]];
         res |= (pos_color != -1 ? ~(1 << m.matriz[pos_color][y]) : m.
prohibir_columna[y]);
      }
      return res;
   } else {
      return m.prohibir_columna_acumulado[i];
   }
}

template<typename trabajador, int i, int j, bool continuar_diag>
void resuelve(datos& m, int usados) {
   /*
      La primera fila debe ser creciente consecutiva.
      La primera columna debe ser creciente.
      El cero debe tener la máxima frecuencia (pueden empatar en
frecuencia, pero no superarlo). Los ceros deben aparecer en la diagonal,
aunque no necesitan cubrirla completamente.
      La submatriz cuadrada inducida por la diagonal de ceros debe ser
simétrica.
      No nos saltaremos colores no usados.

      Los prohibidos son:
      - Los usados de fila y columna.
      - El cero fuera de la diagonal.
      - Fuera de la submatriz simétrica, los prohibidos por frecuencia
incluirán los colores que se hayan usado la misma cantidad de veces que
el cero.
   */

   auto prohibidos = (
      (i == 0 && continuar_diag ? ~(1 << j) :
```

```
      (i >= j && continuar_diag ? ~(1 << (i == j ? 0 : m.matriz[j][i])) :
      m.prohibir_fila[i] | m.prohibir_columna[j] | m.prohibir_frecuencia[
i] | prohibir_celda<i, j>(m) | (1 << 0) | (j == 0 ? (1 << (m.matriz[i -
1][0] + 1)) - 1 : 0)
   ))), temp = m.prohibir_columna_acumulado[i];

   for (auto fin = std::min(n, usados + 1);;) {
      auto color = std::countr_one(prohibidos);
      if (color >= fin) {
         break;
      }

      prohibidos |= (1 << color);
      usados += (color == usados);
      m.matriz[i][j] = color;

      m.frecuencia[color] += 1;
      m.prohibir_fila[i] |= (1 << color);
      m.prohibir_columna[j] |= (1 << color);
      m.pos_color_en_columna[j][color] = i;
      m.prohibir_columna_acumulado[i] |= m.prohibir_columna[j];
      if constexpr(j + 1 < s) {
         trabajador::template procesa<i, j + 1, continuar_diag>(m, usados
);
      } else {
         trabajador::template procesa<i + 1>(m, usados);
      }
      m.prohibir_columna_acumulado[i] = temp;
      m.pos_color_en_columna[j][color] = -1;
      m.prohibir_columna[j] &= ~(1 << color);
      m.prohibir_fila[i] &= ~(1 << color);
      m.frecuencia[color] -= 1;
   }
}
```

```
template<typename trabajador, int i>
void resuelve(datos& m, int usados) {
   if constexpr(i == r) {
      static int cuenta = 0;
      std::cout << "Matriz #" << ++cuenta << " encontrada con " << usados
 << " colores usados\n";
      for (int x = 0; x < r; ++x) {
         for (int y = 0; y < s; ++y) {
            std::cout << int(m.matriz[x][y]) << " ";
         }
         std::cout << "\n";
      }
   } else {
      if (i < s && m.frecuencia[0] == i && (t == -1 || t > i)) {
       // si podemos continuar la diagonal de ceros y no la hemos
interrumpido, lo hacemos
         trabajador::template procesa<i, 0, true>(m, usados);
      }

      if (t == -1 || t <= i) {
       // si podemos interrumpir la diagonal, lo hacemos
         for (int c = 0; c < n; ++c) {
            m.prohibir_frecuencia[i] |= ((m.frecuencia[c] == m.frecuencia
[0]) << c);
         }
         trabajador::template procesa<i, 0, false>(m, usados);
         m.prohibir_frecuencia[i] = 0;
      }
   }
}

struct trabajador_recursivo {
   template<int i, int j, bool continuar_diag>
   static void procesa(datos& m, int usados) {
      resuelve<trabajador_recursivo, i, j, continuar_diag>(m, usados);
```

```
   }

   template<int i>
   static void procesa(datos& m, int usados) {
      resuelve<trabajador_recursivo, i>(m, usados);
   }
};

using ptr_iter = void(*)(datos&, int);
struct parametros {
   ptr_iter continuacion_recursiva;
   ptr_iter continuacion_iterativa;
   datos m;
   int usados;
};

std::deque<parametros> cola;
struct trabajador_iterativo {
   template<int i, int j, bool continuar_diag>
   static void procesa(datos& m, int usados) {
      cola.push_back(parametros(&resuelve<trabajador_recursivo, i, j,
continuar_diag>, &resuelve<trabajador_iterativo, i, j, continuar_diag>, m
, usados));
   }

   template<int i>
   static void procesa(datos& m, int usados) {
      cola.push_back(parametros(&resuelve<trabajador_recursivo, i>, &
resuelve<trabajador_iterativo, i>, m, usados));
   }
};

int main(int argc, const char* argv[]) {
   auto t0 = std::chrono::high_resolution_clock::now( );
   if (argc >= 2 && strcmp(argv[1], "secuencial") == 0) {
```

```
         datos m; int usados = 0;
         resuelve<trabajador_recursivo, 0>(m, usados);
      } else {
         cola.push_back(parametros(&resuelve<trabajador_recursivo, 0>, &
   resuelve<trabajador_iterativo, 0>, datos( ), 0));
         do {
            auto actual = cola.front( ); cola.pop_front( );
            actual.continuacion_iterativa(actual.m, actual.usados);
         } while (0 < cola.size( ) && cola.size( ) < 32 * 1024);

         std::for_each(std::execution::par_unseq, cola.begin( ), cola.end( )
   , [](auto& actual) {
            actual.continuacion_recursiva(actual.m, actual.usados);
         });
      }
      auto t1 = std::chrono::high_resolution_clock::now( );

      std::cout << std::chrono::duration<double>(t1 - t0).count( ) << "
   segundos\n";
   }
#endif
```

A.6. Implementación del algoritmo de Backtracking para las matrices simétricas con isomorfismo

```
/**
Backtracking con parámetros en forma de macro, se usa para matrices simé
   tricas con isomorfismo.
/**/

#include <tbb/concurrent_vector.h>
#include <tbb/parallel_for_each.h>
#include <bit>
#include <map>
#include <array>
```

```
#include <atomic>
#include <deque>
#include <bitset>
#include <chrono>
#include <utility>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <execution>
#include <syncstream>
#include <type_traits>
#include <math.h>
#include <stdint.h>
#include <string.h>

#define VALOR_R 1

#if !defined(VALOR_R)
   static_assert(false, "Deben definirse macros VALOR_R");
#else
   namespace std {
      inline int countr_one(__uint128_t v) {
         return (uint64_t(v) != ~uint64_t(0) ? countr_one(uint64_t(v)) : 64
   + countr_one(uint64_t(v >> 64)));
      }
   }

   constexpr int r = VALOR_R, s = VALOR_R, n = 1 + (VALOR_R * VALOR_R -
   VALOR_R) / 2, t = VALOR_R;
   using uintflag_t = std::conditional_t<(n < 32), uint32_t, std::
   conditional_t<(n < 64), uint64_t, __uint128_t>>;

   template<typename T, size_t F, size_t C>
   using matrix = std::array<std::array<T, C>, F>;
```

```
tbb::concurrent_vector<matrix<int8_t, r, s>> encontradas;
struct datos {
   matrix<int8_t, r, s> matriz;
   int8_t frecuencia[128] = { }, pos_color_en_columna[128][128];
   uintflag_t prohibir_fila[32] = { }, prohibir_columna[32] = { },
prohibir_frecuencia[128] = { }, prohibir_columna_acumulado[32] = { };

   datos( ) {
      std::fill(&pos_color_en_columna[0][0], &pos_color_en_columna
[128][0], -1);
   }
};

template<int i, int j>
auto prohibir_celda(datos& m) {
   if ((m.prohibir_fila[i] & m.prohibir_columna[j]) != 0) {
      uintflag_t res = 0;
      for (int y = 0; y < j; ++y) {
         int8_t pos_color = m.pos_color_en_columna[j][m.matriz[i][y]];
         res |= (pos_color != -1 ? ~(uintflag_t(1) << m.matriz[pos_color
][y]) : m.prohibir_columna[y]);
      }
      return res;
   } else {
      return m.prohibir_columna_acumulado[i];
   }
}

template<typename trabajador, int i, int j, bool continuar_diag>
void resuelve(datos& m, int usados) {
   /*
      La primera fila debe ser creciente consecutiva.
      La primera columna debe ser creciente.
      El cero debe tener la máxima frecuencia (pueden empatar en
frecuencia, pero no superarlo). Los ceros deben aparecer en la diagonal,
```

```
aunque no necesitan cubrirla completamente.
      La submatriz cuadrada inducida por la diagonal de ceros debe ser
simétrica.
      No nos saltaremos colores no usados.

      Los prohibidos son:
      - Los usados de fila y columna.
      - El cero fuera de la diagonal.
      - Fuera de la submatriz simétrica, los prohibidos por frecuencia
incluirán los colores que se hayan usado la misma cantidad de veces que
el cero.
   */

   auto prohibidos = (
      (i == 0 && continuar_diag ? ~(uintflag_t(1) << j) :
      (i >= j && continuar_diag ? ~(uintflag_t(1) << (i == j ? 0 : m.
matriz[j][i])) :
      m.prohibir_fila[i] | m.prohibir_columna[j] | m.prohibir_frecuencia[
i] | prohibir_celda<i, j>(m) | (uintflag_t(1) << 0) | (j == 0 ? (
uintflag_t(1) << (m.matriz[i - 1][0] + 1)) - 1 : 0)
   ))), temp = m.prohibir_columna_acumulado[i];

   for (auto fin = std::min(n, usados + 1);;) {
      auto color = std::countr_one(prohibidos);
      if (color >= fin) {
         break;
      }

      prohibidos |= (uintflag_t(1) << color);
      usados += (color == usados);
      m.matriz[i][j] = color;

      m.frecuencia[color] += 1;
      m.prohibir_fila[i] |= (uintflag_t(1) << color);
      m.prohibir_columna[j] |= (uintflag_t(1) << color);
```

```
      m.pos_color_en_columna[j][color] = i;
      m.prohibir_columna_acumulado[i] |= m.prohibir_columna[j];
      if constexpr(j + 1 < s) {
         trabajador::template procesa<i, j + 1, continuar_diag>(m, usados
);
      } else {
         trabajador::template procesa<i + 1>(m, usados);
      }
      m.prohibir_columna_acumulado[i] = temp;
      m.pos_color_en_columna[j][color] = -1;
      m.prohibir_columna[j] &= ~(uintflag_t(1) << color);
      m.prohibir_fila[i] &= ~(uintflag_t(1) << color);
      m.frecuencia[color] -= 1;
   }
}

template<typename trabajador, int i>
void resuelve(datos& m, int usados) {
   if constexpr(i == r) {
      encontradas.push_back(m.matriz);
   } else {
      if (i < s && m.frecuencia[0] == i && (t == -1 || t > i)) {
       // si podemos continuar la diagonal de ceros y no la hemos
interrumpido, lo hacemos
         trabajador::template procesa<i, 0, true>(m, usados);
      }

      if (t == -1 || t <= i) {
       // si podemos interrumpir la diagonal, lo hacemos
         for (int c = 0; c < n; ++c) {
            m.prohibir_frecuencia[i] |= ((m.frecuencia[c] == m.frecuencia
[0]) << c);
         }
         trabajador::template procesa<i, 0, false>(m, usados);
         m.prohibir_frecuencia[i] = 0;
```

```
        }
    }
}

struct trabajador_recursivo {
    template<int i, int j, bool continuar_diag>
    static void procesa(datos& m, int usados) {
        resuelve<trabajador_recursivo, i, j, continuar_diag>(m, usados);
    }

    template<int i>
    static void procesa(datos& m, int usados) {
        resuelve<trabajador_recursivo, i>(m, usados);
    }
};

using ptr_iter = void(*)(datos&, int);
struct parametros {
    ptr_iter continuacion_recursiva;
    ptr_iter continuacion_iterativa;
    datos m;
    int usados;
};

std::deque<parametros> cola;
struct trabajador_iterativo {
    template<int i, int j, bool continuar_diag>
    static void procesa(datos& m, int usados) {
        cola.push_back(parametros(&resuelve<trabajador_recursivo, i, j,
continuar_diag>, &resuelve<trabajador_iterativo, i, j, continuar_diag>, m
, usados));
    }

    template<int i>
    static void procesa(datos& m, int usados) {
```

```
      cola.push_back(parametros(&resuelve<trabajador_recursivo, i>, &
resuelve<trabajador_iterativo, i>, m, usados));
   }
};

/* Empieza todo lo necesario para la prueba de isomorfismo */

std::ostream& operator<<(std::ostream& os, const matrix<int8_t, r, s>& m)
 {
   for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         os << int(m[i][j]) << " ";
      }
      os << "\n";
   }
   return os;
}

std::array<int, n> calcula_frecuencias(const matrix<int8_t, r, s>& m) {
   std::array<int, n> tabla = { };
   for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         tabla[m[i][j]] += 1;
      }
   }
   return tabla;
}

int calcula_intercalaciones(const matrix<int8_t, r, s>& m) {
   int res = 0;
   for (int i1 = 0; i1 < r - 1; ++i1) {
      for (int j1 = 0; j1 < s - 1; ++j1) {
         for (int i2 = i1 + 1; i2 < r; ++i2) {
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               res += (m[i1][j1] == m[i2][j2]);
```

```
            }
         }
      }
   }
   return res;
}

struct metadatos {
   int intercalaciones;
   std::array<int, n> frecuencias;

   metadatos(const matrix<int8_t, r, s>& m)
   : intercalaciones(calcula_intercalaciones(m)), frecuencias(
calcula_frecuencias(m)) {
      std::sort(frecuencias.begin( ), frecuencias.end( ));
   }

   bool operator<(const metadatos& a) const {
      return std::pair(intercalaciones, frecuencias) < std::pair(a.
intercalaciones, a.frecuencias);
   }
};

std::vector<int> colores_frec_maxima(const matrix<int8_t, r, s>& m) {
   std::array<int, n> tabla = calcula_frecuencias(m);
   std::vector<int> lista_colores;
   for (int i = 0; i < n; ++i) {
      if (tabla[i] == tabla[0]) {
         lista_colores.push_back(i);
      }
   }
   return lista_colores;
}

matrix<int8_t, r, s> normaliza_colores(const matrix<int8_t, r, s>& m) {
```

```
    matrix<int8_t, r, s> res;
    std::array<int8_t, n> asignacion;
    asignacion.fill(-1);
    for (int i = 0, c = 0; i < r; ++i) {
        for (int j = 0; j < s; ++j) {
            if (asignacion[m[i][j]] == -1) {
                asignacion[m[i][j]] = c++;
            }
            res[i][j] = asignacion[m[i][j]];
        }
    }
    return res;
}

matrix<int8_t, r, s> permuta_filas(const matrix<int8_t, r, s>& m, const
std::array<int, r>& permutacion) {
    matrix<int8_t, r, s> res;
    for (int i = 0; i < r; ++i) {
        for (int j = 0; j < s; ++j) {
            res[i][j] = m[permutacion[i]][j];
        }
    }
    return res;
}

matrix<int8_t, r, s> permuta_columnas(const matrix<int8_t, r, s>& m,
const std::array<int, r>& permutacion) {
    matrix<int8_t, r, s> res;
    for (int i = 0; i < r; ++i) {
        for (int j = 0; j < s; ++j) {
            res[i][j] = m[i][permutacion[j]];
        }
    }
    return res;
}
```

```
std::array<int, r> permutacion_col_diagonal(const matrix<int8_t, r, s>& m
, int c) {
   std::array<int, r> permutacion;
   for (int i = 0; i < r; ++i) {
      permutacion[i] = std::find(m[i].begin( ), m[i].end( ), c) - m[i].
begin( );
   }
   return permutacion;
}

bool es_isomorfa(const matrix<int8_t, r, s>& m1, const matrix<int8_t, r,
s>& m2) {
   std::array<int, r> permutacion;
   std::iota(permutacion.begin( ), permutacion.end( ), 0);
   for (const auto& c : colores_frec_maxima(m2)) {
      auto temp = normaliza_colores(permuta_columnas(m2,
permutacion_col_diagonal(m2, c)));
      do {
         if (m1 == normaliza_colores(permuta_filas(permuta_columnas(temp,
 permutacion), permutacion))) {
            return true;
         }
      } while (std::next_permutation(permutacion.begin( ), permutacion.
end( )));
   }
   return false;
}

std::vector<matrix<int8_t, r, s>> matrices_unicas(const std::vector<
matrix<int8_t, r, s>>& grupo) {
   std::vector<matrix<int8_t, r, s>> res;
   for (const auto& actual : grupo) {
      if (std::all_of(res.begin( ), res.end( ), [&](const auto& guardada)
 {
```

```
         return !es_isomorfa(actual, guardada);
      })) {
         res.push_back(actual);
      }
   }
   return res;
}

int main(int argc, const char* argv[]) {
   auto t0 = std::chrono::high_resolution_clock::now( );
   if (argc >= 2 && strcmp(argv[1], "secuencial") == 0) {
      datos m; int usados = 0;
      resuelve<trabajador_recursivo, 0>(m, usados);
   } else {
      cola.push_back(parametros(&resuelve<trabajador_recursivo, 0>, &
resuelve<trabajador_iterativo, 0>, datos( ), 0));
      do {
         auto actual = cola.front( ); cola.pop_front( );
         actual.continuacion_iterativa(actual.m, actual.usados);
      } while (0 < cola.size( ) && cola.size( ) < 32 * 1024);

      std::for_each(std::execution::par_unseq, cola.begin( ), cola.end( )
, [](auto& actual) {
         actual.continuacion_recursiva(actual.m, actual.usados);
      });
   }
   auto t1 = std::chrono::high_resolution_clock::now( );
   std::cout << std::chrono::duration<double>(t1 - t0).count( ) << "
segundos para generar matrices\n";

   /* Revisar isomorfismo */
   auto t2 = std::chrono::high_resolution_clock::now( );
   std::cout << "Agrupando " << encontradas.size( ) << " matrices...\n";
   std::sort(std::execution::par_unseq, encontradas.begin( ), encontradas
.end( ));
```

```
   std::map<metadatos, std::vector<matrix<int8_t, r, s>>> mapa_metadatos;
   for (const auto& actual : encontradas) {
      mapa_metadatos[metadatos(actual)].push_back(actual);
   }
   auto t3 = std::chrono::high_resolution_clock::now( );
   std::cout << std::chrono::duration<double>(t3 - t2).count( ) << "
segundos para agrupar matrices\n";

   auto t4 = std::chrono::high_resolution_clock::now( );
   tbb::concurrent_vector<matrix<int8_t, r, s>> unicas;
   std::cout << mapa_metadatos.size( ) << " grupos...\n";
   tbb::parallel_for_each(mapa_metadatos.begin( ), mapa_metadatos.end( ),
 [&](auto& entrada) {        // por alguna razón, std::for_each(std::
execution::par_unseq, ...) no está jalando bien con std::map
      auto& [meta, grupo] = entrada;
      std::osyncstream(std::cout) << "   Revisando grupo de " << grupo.
size( ) << " matrices...\n";
      for (const auto& actual : matrices_unicas(grupo)) {
         unicas.push_back(actual);
      }
   });
   auto t5 = std::chrono::high_resolution_clock::now( );
   std::cout << std::chrono::duration<double>(t5 - t4).count( ) << "
segundos para filtrar matrices\n";

   std::cout << "Imprimiendo " << unicas.size( ) << " matrices...\n";
   std::sort(std::execution::par_unseq, unicas.begin( ), unicas.end( ));
   for (int indice = 0; indice < unicas.size( ); ++indice) {
      std::cout << "Matriz #" << indice + 1 << ":\n" << unicas[indice];
   }
}
#endif

// g++ -std=c++20 -D VALOR_R=6 -O3 -ltbb backtracking_isomorfismo.cpp -o
   isomorfas
```

A.7. Implementación del algoritmo de Backtracking que utiliza las matrices precalculadas

```
/**
Backtracking que usa matrices precalculadas.
/**/

#include <algorithm>
#include <bit>
#include <bitset>
#include <chrono>
#include <deque>
#include <execution>
#include <fstream>
#include <iostream>
#include <thread>
#include <math.h>
#include <stdint.h>
#include <string.h>

#if !defined(VALOR_R) || !defined(VALOR_S) || !defined(VALOR_DELTA) || !
   defined(VALOR_T)
   static_assert(false, "Deben definirse macros VALOR_R, VALOR_S,
   VALOR_DELTA, VALOR_T");
#else
   constexpr int f(unsigned r, unsigned s) {
      if (r > s) {
         return f(s, r);
      } else if (r == 1) {
         return s;
      } else if (int st = std::bit_ceil(s); st / 2 < r) {
         return st;
      } if (int rt = std::bit_ceil(r); rt < s) {
         return rt + f(r, s - rt);
```

```
    }
    return -1;
}

constexpr int r = std::max(VALOR_R, VALOR_S), s = std::min(VALOR_R,
VALOR_S), n = f(r, s) + VALOR_DELTA, t = VALOR_T;      // preferir pocas
columnas, sino se usarían muchos colores desde la primera fila

template<typename T, size_t F, size_t C>
using matrix = std::array<std::array<T, C>, F>;

struct datos {
    matrix<int8_t, r, s> matriz;
    int8_t frecuencia[n] = { }, pos_color_en_columna[s][n], usados;
    uint32_t prohibir_fila[r] = { }, prohibir_columna[s] = { },
prohibir_frecuencia[n] = { }, prohibir_columna_acumulado[r] = { };
    const matrix<int8_t, t, t>& simetrica;

    datos(matrix<int8_t, t, t>& sim, int u)
    : simetrica(sim), usados(u) {
        std::fill(&pos_color_en_columna[0][0], &pos_color_en_columna[s][0],
 -1);
    }
};

template<int i, int j>
auto prohibir_celda(datos& m) {
    if ((m.prohibir_fila[i] & m.prohibir_columna[j]) != 0) {
        uint32_t res = 0;
        for (int y = 0; y < j; ++y) {
            int8_t pos_color = m.pos_color_en_columna[j][m.matriz[i][y]];
            res |= (pos_color != -1 ? ~(1 << m.matriz[pos_color][y]) : m.
prohibir_columna[y]);
        }
        return res;
```

```
   } else {
      return m.prohibir_columna_acumulado[i];
   }
}

template<typename trabajador, int i, int j>
void resuelve(datos& m) {
   if constexpr(i == r) {
      static int cuenta = 0;
      std::cout << "Matriz #" << ++cuenta << " encontrada con " << m.
usados << " colores usados\n";
      for (int x = 0; x < r; ++x) {
         for (int y = 0; y < s; ++y) {
            std::cout << int(m.matriz[x][y]) << " ";
         }
         std::cout << "\n";
      }
      std::cout << "Posiblemente mas matrices... abortando\n";
      std::exit(0);
   } else {
      if constexpr(j == 0) {
         m.prohibir_frecuencia[i] = 0;
         for (int c = 0; c < n; ++c) {
            m.prohibir_frecuencia[i] |= ((m.frecuencia[c] == t) << c);
         }
      }

      /*
         La primera fila debe ser creciente consecutiva.
         La primera columna debe ser creciente.
         El cero debe tener la máxima frecuencia (pueden empatar en
frecuencia, pero no superarlo). Los ceros deben aparecer en la diagonal,
aunque no necesitan cubrirla completamente.
         La submatriz cuadrada inducida por la diagonal de ceros debe ser
 simétrica.
```

```
        No nos saltaremos colores no usados.

        Los prohibidos son:
        - Los usados de fila y columna.
        - El cero fuera de la diagonal.
        - Fuera de la submatriz simétrica, los prohibidos por frecuencia
 incluirán los colores que se hayan usado la misma cantidad de veces que
el cero.
      */

      auto prohibidos =
         (i < t && j < t ? ~(1 << m.simetrica[i][j]) : 0) |
         m.prohibir_fila[i] | m.prohibir_columna[j] | m.
prohibir_frecuencia[i] | prohibir_celda<i, j>(m) | (i != 0 && j == 0 ? (1
 << (m.matriz[i - 1][0] + 1)) - 1 : 0) | (i >= t && j < t ? m.
prohibir_fila[j] : 0)
      , temp = m.prohibir_columna_acumulado[i];

      for (auto fin = std::min(n, m.usados + 1);;) {
         auto color = std::countr_one(prohibidos);
         if (color >= fin) {
            break;
         }
         bool nuevo = (color == m.usados);
         prohibidos |= (1 << color);

         m.matriz[i][j] = color;
         m.usados += nuevo;
         m.frecuencia[color] += 1;
         m.prohibir_fila[i] |= (1 << color);
         m.prohibir_columna[j] |= (1 << color);
         m.pos_color_en_columna[j][color] = i;
         m.prohibir_columna_acumulado[i] |= m.prohibir_columna[j];
         trabajador::template procesa<i + (j + 1 == s), (j + 1) % s>(m);
         m.prohibir_columna_acumulado[i] = temp;
```

```
            m.pos_color_en_columna[j][color] = -1;
            m.prohibir_columna[j] &= ~(1 << color);
            m.prohibir_fila[i] &= ~(1 << color);
            m.frecuencia[color] -= 1;
            m.usados -= nuevo;
         }
      }
   }

   struct trabajador_recursivo {
      template<int i, int j>
      static void procesa(datos& m) {
         resuelve<trabajador_recursivo, i, j>(m);
      }
   };

   using ptr_iter = void(*)(datos&);
   struct parametros {
      ptr_iter continuacion_recursiva;
      ptr_iter continuacion_iterativa;
      datos m;
   };

   std::deque<parametros> cola;
   struct trabajador_iterativo {
      template<int i, int j>
      static void procesa(datos& m) {
         cola.push_back(parametros(&resuelve<trabajador_recursivo, i, j>, &
   resuelve<trabajador_iterativo, i, j>, m));
      }
   };

   template<typename RI, typename F>
   void parallel_for_each_(RI ini, RI fin, F&& f) {
      std::atomic<size_t> indice(0); size_t tam = fin - ini;
```

```
    std::thread hilos[std::thread::hardware_concurrency( )];
    for (auto& hilo : hilos) {
       hilo = std::thread([&] {
          for (size_t usar; (usar = indice++) < tam;) {
             f(ini[usar]);
          }
       });
    }
    for (auto& hilo : hilos) {
       hilo.join( );
    }
 }

 int main(int argc, const char* argv[]) {
    std::ifstream ifs(std::to_string(t) + ".txt");
    if (!ifs.is_open( )) {
       std::cout << "No se pudo abrir el archivo";
       return 0;
    }

    int matrices;
    ifs >> matrices;

    std::vector<matrix<int8_t, t, t>> simetricas(matrices);
    for (int mi = 0; mi < matrices; ++mi) {
       int colores;
       ifs >> colores;
       for (int i = 0; i < t; ++i) {
          for (int j = 0; j < t; ++j) {
             int temp;
             ifs >> temp;
             simetricas[mi][i][j] = temp;
          }
       }
       if (colores <= n) {
```

```
            cola.push_back(parametros(&resuelve<trabajador_recursivo, 0, 0>,
    &resuelve<trabajador_iterativo, 0, 0>, datos(simetricas[mi], colores)));
         }
      }

      auto t0 = std::chrono::high_resolution_clock::now( );
      if (argc >= 2 && strcmp(argv[1], "secuencial") == 0) {
         std::for_each(cola.begin( ), cola.end( ), [](auto& actual) {
            actual.continuacion_recursiva(actual.m);
         });
      } else {
         int usados = cola.size( );
         do {
            auto actual = cola.front( ); cola.pop_front( );
            actual.continuacion_iterativa(actual.m);
         } while (0 < cola.size( ) && cola.size( ) < usados * 32 * 1024);
         parallel_for_each_(cola.begin( ), cola.end( ), [](auto& actual) {
            actual.continuacion_recursiva(actual.m);
         });
      }
      auto t1 = std::chrono::high_resolution_clock::now( );
      std::cout << std::chrono::duration<double>(t1 - t0).count( ) << "
   segundos\n";
   }
#endif
```

A.8. Implementación del algoritmo de Backtracking que utiliza las matrices simétricas precalculadas para generar matrices simétricas más grandes

```
/**
Backtracking que usa matrices precalculadas simétricas.
/**/
```

```
#define TBB_PREVIEW_CONCURRENT_ORDERED_CONTAINERS 1
#include <tbb/concurrent_map.h>
#include <tbb/concurrent_vector.h>
#include <tbb/parallel_for_each.h>
#include <algorithm>
#include <bit>
#include <bitset>
#include <chrono>
#include <deque>
#include <execution>
#include <fstream>
#include <iostream>
#include <syncstream>
#include <thread>
#include <math.h>
#include <stdint.h>
#include <string.h>

#if !defined(VALOR_R)
   static_assert(false, "Deben definirse macros VALOR_R");
#else
   namespace std {
      inline int countr_one(__uint128_t v) {
         return (uint64_t(v) != ~uint64_t(0) ? countr_one(uint64_t(v)) : 64
   + countr_one(uint64_t(v >> 64)));
      }
   }
   namespace impl {
      template<typename T>
      std::atomic<T>& as_atomic(T& v) {
         static_assert(sizeof(T) == sizeof(std::atomic<T>) && alignof(T) ==
   alignof(std::atomic<T>));
         return reinterpret_cast<std::atomic<T>&>(v);
      }
   }
```

```
constexpr int r = VALOR_R, s = VALOR_R, n = 1 + (VALOR_R * VALOR_R -
VALOR_R) / 2, t = VALOR_R - 1;
using uintflag_t = std::conditional_t<(n < 32), uint32_t, std::
conditional_t<(n < 64), uint64_t, __uint128_t>>;

template<typename T, size_t F, size_t C>
using matrix = std::array<std::array<T, C>, F>;

std::ostream& operator<<(std::ostream& os, const matrix<int8_t, r, s>& m)
 {
   for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         os << int(m[i][j]) << " ";
      }
      os << "\n";
   }
   return os;
}

// INICIO ISOMORFISMO

std::array<int, n> calcula_frecuencias(const matrix<int8_t, r, s>& m) {
   std::array<int, n> tabla = { };
   for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         tabla[m[i][j]] += 1;
      }
   }
   return tabla;
}

int calcula_intercalaciones(const matrix<int8_t, r, s>& m) {
   int res = 0;
   for (int i1 = 0; i1 < r - 1; ++i1) {
```

```
      for (int j1 = 0; j1 < s - 1; ++j1) {
         for (int i2 = i1 + 1; i2 < r; ++i2) {
            for (int j2 = j1 + 1; j2 < s; ++j2) {
               res += (m[i1][j1] == m[i2][j2]);
            }
         }
      }
   }
   return res;
}

struct metadatos {
   int intercalaciones;
   std::array<int, n> frecuencias;

   metadatos(const matrix<int8_t, r, s>& m)
   : intercalaciones(calcula_intercalaciones(m)), frecuencias(
calcula_frecuencias(m)) {
      std::sort(frecuencias.begin( ), frecuencias.end( ));
   }

   bool operator<(const metadatos& a) const {
      return std::pair(intercalaciones, frecuencias) < std::pair(a.
intercalaciones, a.frecuencias);
   }
};

matrix<int8_t, r, s> normaliza_colores(const matrix<int8_t, r, s>& m) {
   matrix<int8_t, r, s> res;
   std::array<int8_t, n> asignacion;
   asignacion.fill(-1);
   for (int i = 0, c = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         if (asignacion[m[i][j]] == -1) {
            asignacion[m[i][j]] = c++;
```

```
        }
        res[i][j] = asignacion[m[i][j]];
      }
    }
    return res;
}

std::vector<int> colores_frec_maxima(const matrix<int8_t, r, s>& m) {
    std::array<int, n> tabla = { };
    for (int i = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
        tabla[m[i][j]] += 1;
      }
    }

    std::vector<int> lista_colores;
    for (int i = 0; i < n; ++i) {
      if (tabla[i] == tabla[0]) {
        lista_colores.push_back(i);
      }
    }
    return lista_colores;
}

std::array<int, r> permutacion_col_diagonal(const matrix<int8_t, r, s>& m
, int c) {
    std::array<int, r> permutacion;
    for (int i = 0; i < r; ++i) {
      permutacion[i] = std::find(m[i].begin( ), m[i].end( ), c) - m[i].
begin( );
    }
    return permutacion;
}

std::array<int, r> composicion_permutaciones(const std::array<int, r>&
```

```
permutacion1, const std::array<int, r>& permutacion2) {
   std::array<int, r> temp;
   for (int i = 0; i < r; ++i) {
      temp[i] = permutacion1[permutacion2[i]];
   }
   return temp;
}

bool misma_matriz(const matrix<int8_t, r, s>& m1, const matrix<int8_t, r,
 s>& m2, const std::array<int, r>& permutacion_fila, const std::array<int
, r>& permutacion_col) {
   std::array<int8_t, n> asignacion; asignacion.fill(-1);
   for (int i = 0, c = 0; i < r; ++i) {
      for (int j = 0; j < s; ++j) {
         int temp = m2[permutacion_fila[i]][permutacion_col[j]];
         if (asignacion[temp] == -1) {
            asignacion[temp] = c++;
         }
         if (m1[i][j] != asignacion[temp]) {
            return false;
         }
      }
   }
   return true;
}

bool es_isomorfa(const matrix<int8_t, r, s>& m1, const matrix<int8_t, r,
s>& m2) {
   std::array<int, r> permutacion;
   std::iota(permutacion.begin( ), permutacion.end( ), 0);
   for (const auto& c : colores_frec_maxima(m2)) {
      auto permutacion_diag = permutacion_col_diagonal(m2, c);
      do {
         if (misma_matriz(m1, m2, permutacion, composicion_permutaciones(
permutacion_diag, permutacion))) {
```

```
            return true;
         }
      } while (std::next_permutation(permutacion.begin( ), permutacion.
end( )));
   }
   return false;
}

template<typename IT>
bool es_isomorfa(IT ini, IT fin, const matrix<int8_t, r, s>& m) {
   return std::any_of(ini, fin, [&](const auto& guardada) {
      return es_isomorfa(m, guardada);
   });
}

std::vector<matrix<int8_t, r, s>> matrices_unicas(tbb::concurrent_vector<
matrix<int8_t, r, s>>&& v) {
   std::sort(v.begin( ), v.end( ));
   std::vector<matrix<int8_t, r, s>> res;
   for (auto& actual : v) {
      if (!es_isomorfa(res.begin( ), res.end( ), actual)) {
         res.push_back(actual);
      }
   }
   return res;
}

// FIN ISOMORFISMO

struct shared_spinlock {
   int estado_raw = 0;

   void lock_shared( ) {
      auto& estado = impl::as_atomic(estado_raw);
      for (estado += 1; estado <= 0;) {
```

```
            continue;
         }
      }

      void unlock_shared( ) {
         impl::as_atomic(estado_raw) -= 1;
      }

      void lock( ) {
         auto& estado = impl::as_atomic(estado_raw);
         for (int esperado = 0; !estado.compare_exchange_strong(esperado,
   -1000); esperado = 0) {
            continue;
         }
      }

      void unlock( ) {
         impl::as_atomic(estado_raw) += 1000;
      }
   };

   tbb::concurrent_map<metadatos, std::tuple<int, shared_spinlock, tbb::
   concurrent_vector<matrix<int8_t, r, s>>>> encontradas;

   struct datos {
      matrix<int8_t, r, s> matriz;
      int8_t frecuencia[n] = { }, pos_color_en_columna[s][n], usados;
      uintflag_t prohibir_fila[r] = { }, prohibir_columna[s] = { },
   prohibir_columna_acumulado[r] = { };
      const matrix<int8_t, t, t>& simetrica;

      datos(matrix<int8_t, t, t>& sim, int u)
      : simetrica(sim), usados(u) {
         std::fill(&pos_color_en_columna[0][0], &pos_color_en_columna[s][0],
    -1);
```

```
    }
};

template<int i, int j>
auto prohibir_celda(datos& m) {
    if ((m.prohibir_fila[i] & m.prohibir_columna[j]) != 0) {
        uintflag_t res = 0;
        for (int y = 0; y < j; ++y) {
            int8_t pos_color = m.pos_color_en_columna[j][m.matriz[i][y]];
            res |= (pos_color != -1 ? ~(uintflag_t(1) << m.matriz[pos_color
][y]) : m.prohibir_columna[y]);
        }
        return res;
    } else {
        return m.prohibir_columna_acumulado[i];
    }
}

template<typename trabajador, int i, int j>
void resuelve(datos& m) {
    if constexpr(i == r) {
        auto normalizada = normaliza_colores(m.matriz);
        /*auto& [construidos_raw, vector] = encontradas[m.matriz];
        if (auto& construidos = impl::as_atomic(construidos_raw); !
es_isomorfa(vector.begin( ), vector.begin( ) + construidos.load( ),
normalizada)) {
            vector.push_back(normalizada);
            construidos += 1;
        }*/
        auto& [construidas_raw, mutex, vector] = encontradas[m.matriz];
        auto& construidas = impl::as_atomic(construidas_raw);
        mutex.lock_shared( );
        for (auto ini = vector.begin( ), fin = ini + construidas.load( );
ini != fin; ++ini) {
            if (auto& actual = *ini; es_isomorfa(normalizada, actual)) {
```

```
            bool intentar = (normalizada < actual);
            mutex.unlock_shared( );
            if (intentar) {
               mutex.lock( );
               if (normalizada < actual) {
                  actual = normalizada;
               }
               mutex.unlock( );
            }
            return;
         }
      }
      mutex.unlock_shared( );
      vector.emplace_back(normalizada);
      construidas += 1;
   } else {
      /*
         La primera fila debe ser creciente consecutiva.
         La primera columna debe ser creciente.
         El cero debe tener la máxima frecuencia (pueden empatar en
frecuencia, pero no superarlo). Los ceros deben aparecer en la diagonal,
aunque no necesitan cubrirla completamente.
         La submatriz cuadrada inducida por la diagonal de ceros debe ser
 simétrica.
         No nos saltaremos colores no usados.

         Los prohibidos son:
         - Los usados de fila y columna.
         - El cero fuera de la diagonal.
         - Fuera de la submatriz simétrica, los prohibidos por frecuencia
 incluirán los colores que se hayan usado la misma cantidad de veces que
el cero.
      */

      auto prohibidos = (
```

```
         i < t && j < t ?
            ~(uintflag_t(1) << m.simetrica[i][j]) :
            m.prohibir_fila[i] | m.prohibir_columna[j] | prohibir_celda<i
, j>(m) | (j == 0 ? (uintflag_t(1) << (m.matriz[i - 1][0] + 1)) - 1 : 0)
| (i >= j ? ~(uintflag_t(1) << (i == j ? 0 : m.matriz[j][i])) : 0)
      ), temp = m.prohibir_columna_acumulado[i];

      for (auto fin = std::min(n, m.usados + 1);;) {
         auto color = std::countr_one(prohibidos);
         if (color >= fin) {
            break;
         }
         bool nuevo = (color == m.usados);
         prohibidos |= (uintflag_t(1) << color);

         m.matriz[i][j] = color;
         m.usados += nuevo;
         m.prohibir_fila[i] |= (uintflag_t(1) << color);
         m.prohibir_columna[j] |= (uintflag_t(1) << color);
         m.pos_color_en_columna[j][color] = i;
         m.prohibir_columna_acumulado[i] |= m.prohibir_columna[j];
         trabajador::template procesa<i + (j + 1 == s), (j + 1) % s>(m);
         m.prohibir_columna_acumulado[i] = temp;
         m.pos_color_en_columna[j][color] = -1;
         m.prohibir_columna[j] &= ~(uintflag_t(1) << color);
         m.prohibir_fila[i] &= ~(uintflag_t(1) << color);
         m.usados -= nuevo;
      }
   }
}

struct trabajador_recursivo {
   template<int i, int j>
   static void procesa(datos& m) {
      resuelve<trabajador_recursivo, i, j>(m);
```

```
    }
};

using ptr_iter = void(*)(datos&);
struct parametros {
    ptr_iter continuacion_recursiva;
    ptr_iter continuacion_iterativa;
    datos m;
};

std::deque<parametros> cola;
struct trabajador_iterativo {
    template<int i, int j>
    static void procesa(datos& m) {
        cola.push_back(parametros(&resuelve<trabajador_recursivo, i, j>, &
resuelve<trabajador_iterativo, i, j>, m));
    }
};

template<typename RI, typename F>
void parallel_for_each_(RI ini, RI fin, F&& f) {
    std::atomic<size_t> indice(0); size_t tam = fin - ini;
    std::thread hilos[std::thread::hardware_concurrency( )];
    for (auto& hilo : hilos) {
        hilo = std::thread([&] {
            for (size_t usar; (usar = indice++) < tam;) {
                f(ini[usar]);
            }
        });
    }
    for (auto& hilo : hilos) {
        hilo.join( );
    }
}
```

```
int main(int argc, const char* argv[]) {
   std::ifstream ifs(std::to_string(t) + ".txt");
   if (!ifs.is_open( )) {
      std::cout << "No se pudo abrir el archivo";
      return 0;
   }

   int matrices;
   ifs >> matrices;

   std::vector<matrix<int8_t, t, t>> simetricas(matrices);
   for (int mi = 0; mi < matrices; ++mi) {
      int colores;
      ifs >> colores;
      for (int i = 0; i < t; ++i) {
         for (int j = 0; j < t; ++j) {
            int temp;
            ifs >> temp;
            simetricas[mi][i][j] = temp;
         }
      }
      if (colores <= n) {
         cola.push_back(parametros(&resuelve<trabajador_recursivo, 0, 0>,
 &resuelve<trabajador_iterativo, 0, 0>, datos(simetricas[mi], colores)));
      }
   }

   auto t0 = std::chrono::high_resolution_clock::now( );
   if (argc >= 2 && strcmp(argv[1], "secuencial") == 0) {
      std::for_each(cola.begin( ), cola.end( ), [](auto& actual) {
         actual.continuacion_recursiva(actual.m);
      });
   } else {
      int usados = cola.size( );
      do {
```

```
         auto actual = cola.front( ); cola.pop_front( );
         actual.continuacion_iterativa(actual.m);
      } while (0 < cola.size( ) && cola.size( ) < std::log2(std::log2(
usados)) * 32 * 1024);
      parallel_for_each_(cola.begin( ), cola.end( ), [](auto& actual) {
         actual.continuacion_recursiva(actual.m);
      });
   }
   auto t1 = std::chrono::high_resolution_clock::now( );
   std::cout << std::chrono::duration<double>(t1 - t0).count( ) << "
segundos para generar matrices\n";

   /* Revisar isomorfismo final*/
   tbb::concurrent_vector<matrix<int8_t, r, s>> unicas;
   std::cout << encontradas.size( ) << " grupos...\n";
   tbb::parallel_for_each(encontradas.begin( ), encontradas.end( ), [&](
auto& entrada) {        // por alguna razón, std::for_each(std::execution
::par_unseq, ...) no está jalando bien con map
      auto& [meta, grupo] = entrada;
      std::osyncstream(std::cout) << "   Revisando grupo de " << std::get
<2>(grupo).size( ) << " matrices...\n";
      for (const auto& actual : matrices_unicas(std::move(std::get<2>(
grupo)))) {
         unicas.push_back(actual);
      }
   });
   auto t2 = std::chrono::high_resolution_clock::now( );
   std::cout << std::chrono::duration<double>(t2 - t1).count( ) << "
segundos para filtrar matrices\n";

   std::cout << "Imprimiendo " << unicas.size( ) << " matrices...\n";
   std::sort(std::execution::par_unseq, unicas.begin( ), unicas.end( ));
   for (int indice = 0; indice < unicas.size( ); ++indice) {
      std::cout << "Matriz #" << indice + 1 << ":\n" << unicas[indice];
   }
```

```
    }
#endif
```

Bibliografía

[1] Gitler, I., E. Reyes y F. Zaragoza: *A Step Towards Yuzvinsky's Conjecture.* The Electronic Journal of Combinatorics, 24(4), 2017.

[2] Gurobi Optimization, Inc. (ed.): *Gurobi Optimizer Reference Manual*. 2020. `https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.0/refman.pdf`.

[3] Hopf, H.: *Ein topologischer Beitrag zur reellen Algebra.* Comment. Math. Helv., 13:219–239, 1940/41.

[4] Lam, K. Y.: *Intercalate coloring of matrices and the Yuzvinsky conjecture*. Bol. Soc. Mat. Mexicana, 23(1):79–86, 2017.

[5] Lam, K. Y. y P. Y. H. Yiu: *Sums of squares formulae near the Hurwitz-Radon range.* Contemp. Math., 58:51–56, 1987.

[6] Lam, K. Y. y P. Y. H. Yiu: *Geometry of normed bilinear maps and the 16-square problem.* Math. Ann., 284:437–447, 1989.

[7] Padua, D. (ed.): *TBB (Intel Threading Building Blocks)*, págs. 2029–2029. Springer US, Boston, MA, 2011, ISBN 978-0-387-09766-4. `https://doi.org/10.1007/978-0-387-09766-4_2080`.

[8] Pfister, A.: *Zur Darstellung von* -1 *als Summe von Quadraten in einem Körper.* J. London Math. Soc., 40:159–165, 1965.

[9] Stiefel, E.: *Über Richtungsfelder in den projecktiven Räumen und einen Satz aus der reellen Algebra.* Comment. Math. Helv., 13:201–208, 1941.

[10] Yiu, P. Y. H.: *On the product of two sums of 16 squares as a sum of squares of integral bilinear forms.* Quart. J. Math. Oxford, 41(2):463–500, 1990.

[11] Zaragoza, F.: *Coloraciones Mínimas de Matrices Intercaladas*. Tesis de Maestría, Centro de Investigación y de Estudios Avanzados del I.P.N., Agosto 1997.

[12] Zaragoza, F.: *Intercalate matrices and algebraic varieties*. Morfismos, 2(1):67–81, 1998.