Universidad Nacional Autónoma de México

Centro de Física Aplicada y Tecnología Avanzada

Números de Van der Waerden y su análogo heterocromático

T E S I S

QUE PARA OPTAR POR EL GRADO DE:

Licenciado en tecnología

PRESENTA:

Angel Balderas Paredes

DIRECTORA DE TESIS:

Dra. Amanda Montejano Cantoral

CODIRECTOR DE TESIS:

Dr. Edgardo Roldan Pensado

Santiago de Querétaro, Querétaro, 2016

A aquel al que este trabajo le sea útil

Reconocimientos

En general, agradezco a la UNAM, a mi familia y profesores.

En particular, bueno, en particular la historia es un poco más elaborada.

Agradezco a mi madre, con quien estoy en deuda por soportarme de la mejor manera, y a mi hermana por soportarme.

A Amanda porque, sin intentarlo, me mostró que no soy tan tonto como creía. Por eso y por todas las horas que mató explicándome matemáticas y hasta algo de redacción; por el mecenazgo en forma de computadora y beca, apoyo crucial para la culminación del proyecto. De verdad, gracias.

También quiero mostrar mi gratitud a Edgardo por todo lo que me dejó aprender en esas largas sesiones de trabajo y por ese entusiasmo perenne por resolver problemas del que espero haberme contagiado.

Quisiera reconocer también al Dr. Remy y al personal administrativo de la licenciatura por la atención que me brindaron. Al PAPIIT IN114016 por la beca otorgada y a CONACyT 219827. Además a la UNAM por razones más que obvias y al CINNMA por esas memorables veces en la casa amarilla.

Por último, quisiera agradecer las horas que se tomaron los miembros del sínodo para revisar este trabajo. Sin ellos, el presente texto hubiese reclamado el embellecimiento y correcciones que brindaron con sus atentas lupas y sus finísimos guantes. Que conste que si cabe alguna corrección será toda la culpa del autor y de ellos nada, pues sería inhumano haberles pedido más. Gracias pues a las Doctoras Hansberg, Millán y Montejano; gracias a los Doctores Raggi y Roldán.

Índice general

Índice de tablas VII

1. Preliminares 5

1.1. Brevario sobre complejidad computacional 5

1.2. Principios matemáticos 9

1.2.1. El principio de inducción 9

1.2.2. El principio de las casillas 12

1.3. Gráficas y coloraciones 16

1.3.1. Gráficas. 16

1.3.2. Coloraciones. 17

1.4. El teorema de Ramsey 19

1.4.1. Números de Ramsey, su dificultad y su relación con las computadoras 25

2. Números de Van der Waerden 29

2.1. Números de van der Waerden 29

2.2. Algoritmos sobre números de Van der Waerden 32

2.2.1. Algoritmo para hallar las progresiones aritméticas más largas . . 32

2.2.2. Algoritmo para obtener una cúspide 34

2.2.3. El algoritmo de la cúspide 36

2.2.4. El algoritmo de los comodines 38

3. Números de anti-Van der Waerden **43**

3.1. Caso heterocromático 43

3.2. Números conocidos 44

3.3. Un algoritmo de búsqueda de progresiones aritméticas heterocromáticas 45

3.4. Algoritmo de los comodines 47

3.4.1. El algoritmo de los comodines 48

3.4.2. Optimización 50

3.4.3. Implementación 51

3.5. Nuevos números 53

4. Conclusiones **55**

A. Código **57**

A.1. Algoritmo de la cúspide 57

A.2. Algoritmo de los comodines 60

Bibliografía **67**

Índice de tablas

2.1. Los números de Van der Waerden, $w(k_1, k_2; 2)$ 31
2.2. Los números de Van der Waerden, $w(2, k_2, k_3; 3)$ 31
2.3. Los números de Van der Waerden, $w(3, k_2, k_3; 3)$ 32

3.1. Los números de anti-Van der Waerden 44
3.2. Orden lexicografico de derecha a izquierda 50
3.3. Los nuevos números de anti-Van der Waerden 53

Introducción

La teoría de Ramsey en ocasiones se describe como *"la búsqueda de estructuras monocromáticas en universos coloreados"* o, también, como *"el estudio de la conservación de propiedades bajo particiones de conjuntos"*. Esta teoría goza de un creciente interés por parte de la comunidad matemática, siendo una fuente constante de inpiración y desafíos intelectuales, además de obtener atención de otras áreas como la computación y la toma de decisiones [1]. El teorema de Van der Waerden, que será revisado con cuidado en la Sección 2, es un resultado clasificado dentro la teoría de Ramsey; calcular los números de Van der Waerden, que se definen a partir del teorema homónimo, son un problema que se ha abordado haciendo uso de, entre otras cosas, algoritmos.

Sobre tal tema, el de calcular los números de Van der Waerden a través de algoritmos, este documento busca, en su humilde condición de tácita incompletitud, compilar y compartir el fruto del ingenio de aquellos que aceptaron pensar, repensar y luego programar.

Para introducir el problema central del que se ocupa el presente trabajo de tesis, es necesario entender antes el concepto de progresión aritmética: una *k-progresión aritmética*, o *k-PA* para acortar, es una secuencia de k números que se caracteriza porque dos números consecutivos cualesquiera difieren en una cantidad fija llamada *diferencia*. Dicha secuencia debe tener, además, un elemento mínimo al que se denomina *base* de la progresión aritmética. Por ejemplo, el conjunto $\{1, 2, 3\}$ es una 3-progresión aritmética con diferencia 1 y base 1, y el conjunto $\{-2, 1, 4, 7\}$ es una 4-progresión aritmética con diferencia 3 y base -2.

También necesitaremos entender qué son los números de Van der Waerden: consideremos el conjunto $\{1, 2, 3, ..., w\}$ e imaginemos que podemos asignar un color a cada uno de sus elementos, supongamos rojo o azul. Los *números de Van der Waerden* se encuentran al preguntar si existe un número w que garantice que, sin importar cómo se asignen los colores al conjunto $\{1, 2, 3, ..., w\}$, siempre encontraremos una k-progresión aritmética que tenga todos los números del mismo color, ya sean rojos o azules. Dicho número es conocido como el *número de Van der Waerden*, $w = w(k; 2)$, donde el 2 es el número de colores, en este caso, rojo o azul. Para ejemplificar lo anterior, mantengamos las cosas sencillas y escojamos $k = 3$. Así, con dos colores, $w = w(3; 2) = 9$ es el entero positivo mínimo tal que, sin importar cuál de los dos colores le asignemos a cada elemento del conjunto $\{1, 2, 3, ..., 9\}$, siempre vamos a encontrar una 3-PA cuyos componentes sean todos rojos o todos azules.

De manera general, los números de Van der Waerden $w = w(k; r)$ (cuya definición precisa se dará en la Sección 2.1), existen para cualesquiera k y r, siendo k el tamaño de la progresión aritmética y r el número de colores. Daremos más detalles en el capítulo siguiente sobre la afirmación anterior que es conocida como el *Teorema de Van der Waerden*.

Determinar los números de Van der Waerden es un problema tipo Ramsey (es decir, se busca un número mínimo para determinar el tamaño que debe tener un conjunto donde se pueda encontrar alguna propiedad específica) y, como tal, es un problema muy sencillo cuando se trata de números pequeños. Si, en cambio, uno se ocupa de números apenas más grandes, el panorama es sobrecogedor y obtenerlos no es, ni de lejos, trivial.

Incluso en tiempos como los que corren cuando la tecnología computacional es uno de los mayores orgullos de la humanidad, encontrar los números de Van der Waerden recurriendo a las computadoras resulta una tarea tan costosa que si nuestra vida dependiera de ello, convendría usar nuestros recursos en pensar como engañar a la muerte más que en encontrarlos. Para mostrar dicho punto basta con decir que la totalidad de los números de Van der Waerden, conocidos y no triviales, se pueden contar usando nada más que los dedos de las manos, y sobrarían dedos [2]. Estos siete números se

pueden encontrar en la sección 2.1).

Si bien es cierto que resolver el misterio de los números mencionados arriba es una tarea titánica, también es cierto que los humanos son tozudos y no ha faltado intención de aprovechar el poder de procesamiento que brindan las máquinas, produciendo así resultados interesantes.

A pesar de tales esfuerzos, se pueden encontrar temas que parecen estar poco explorados, por ejemplo: la existencia de algoritmos que permitan, a través de las computadoras, calcular los números de anti-Van der Waerden (la definición de números de anti-Van der Waerden se encuentra en la Sección 3.1), ya que tras una revisión del estado del arte solo se encontró un algoritmo ocupado en dicha tarea.

El presente trabajo surgió con la intención de cumplir los siguientes objetivos:

1. Brindar una introducción accesible al estudio de los números de Van der Waerden y su análogo heterocromático, los números de anti-Van der Waerden.

2. Estudiar los diferentes algoritmos existentes para el caso monocromático.

3. Explorar las posibilidades de adaptar los algoritmos del caso monocromático al caso heterocromático.

4. Comparar los resultados que se pueden obtener usando los nuevos algoritmos con los reportados en la literatura.

Se puede adelantar que al término de este proyecto se obtuvieron resultados satifactorios sobre cada uno de los objetivos recién enunciados. De hecho, cabe destacar que con el esfuerzo puesto en el presente trabajo, se logró contribuir al estado del arte con un algoritmo que permitió obtener nuevos números de Van der Waerden en un tiempo razonable.

Para alcanzar dichos objetivos, se propuso realizar una revisión del estado del arte de los algoritmos usados para el cálculo de los números de Van der Waerden y de los números de anti-Van der Waerden (el análogo heterocromático). Luego de cada revisión se profundizó en algunos de los algoritmos, se analizaron sus puntos fuertes y sus

flaquezas; se buscaron también semejanzas y diferencias entre los casos monocromático y heterocromático, con el fin de explorar la posibilidad de adaptar dichos algoritmos a este último.

Al final, se muestran los algoritmos y los resultados de su implementación, terminando con la discusión de los resultados. Se agregó también un apéndice con el código que se usó para implementar los distintos algorimos.

Capítulo 1

Preliminares

Este capítulo comenzará con la introducción de los conceptos necesarios para el resto de la tesis. Consistirá de cuatro secciones: la primera sobre complejidad, la segunda sobre algunos principios matemáticos, seguida de una sección sobre gráficas y coloraciones, para terminar con una sección dedicada al teorema de Ramsey. Las secciones están ordenadas de manera que cada una se construye usando los conceptos de las secciones previas.

1.1. Brevario sobre complejidad computacional

La tesis presente se enfocará en algunos de los algoritmos usados para calcular los números de Van der Waerden y de anti-Van der Waerden. Por ello, resulta adecuado comenzar con una sección dedicada al estudio de la complejidad computacional. La estrategia de presentación consistirá en enunciar todas las definiciones para terminar con un ejemplo que las ilustre.

A la hora de estudiar y diseñar algoritmos una de las preocupaciones principales es la cantidad de recursos que va a consumir la ejecución del procedimiento en cuestión. Recursos como la cantidad de memoria usada y número de pasos.

Una de las principales herramientas usadas con el fin de estimar el consumo de recursos es la notación asintótica. Dicha notación hace uso, entre otras, de tres definiciones: la notación $O(g(n))$, $\Omega(g(n))$ y $\Theta(g(n))$. Son usadas porque permiten realizar estimaciones del máximo o mínimo número de recursos requeridos para la ejecución de

un algoritmo, es útil la notación porque dichas estimaciones suelen representarse como una función, digamos f; además la función suele ser del número de elementos, n, de la entrada. Así, el tiempo estimado estará dado por función $f(n)$, donde $n \in \mathbb{N}$. Cada una de las tres O, Θ y Ω, definen un conjunto como sigue:

Definición 1.1.1. *Cuando decimos que $f(n) = O(g(n))$, significa que $f(n)$ pertenece al conjunto:*

$$\begin{aligned} O(g(n)) = &\{h : N \to N \mid \exists c \in N \, y \, n_0 \in N \\ &tales \, que \, 0 \leq h(n) \leq cg(n) \, para \, todo \, n \geq n_0\} \end{aligned} \tag{1.1}$$

Definición 1.1.2. *Decimos que $f(n) = \Omega(g(n))$ si $f(n)$ pertenece al conjunto:*

$$\begin{aligned} \Omega(g(n)) = &\{h : N \to N \mid \exists c \in N \, y \, n_0 \in N \\ &tales \, que \, 0 \leq cg(n) \leq h(n) \; para \; todo \, n \geq n_0\} \end{aligned} \tag{1.2}$$

Definición 1.1.3. *Decimos $f(n) = \Theta(g(n))$ cuando $f(n)$ pertenece al conjunto:*

$$\begin{aligned} \Theta(g(n)) = &\{h : N \to N \mid \exists c_1, c_2 \in N \, y \, n_0 \in N \\ &tales \; que \, c_1 g(n) \leq h(n) \leq c_2 g(n) \; para \; todo \, n \geq n_0\} \end{aligned} \tag{1.3}$$

Para ejemplificar el uso de las funciones anteriores vamos a realizar el análisis de un algoritmo de búsqueda lineal de casos. Dicho algoritmo tomará como entrada el vector $A = \langle a_1, a_2, \ldots, a_n \rangle$ de enteros y un valor por buscar v, también entero. La salida del algoritmo será el número p, que es el número de veces que v aparece en A. El algoritmo se muestra a continuación:

Algoritmo 0 Linear Search algorithm

Input: (A, v) A es el vector de números disponibles para la búsqueda y v es el valor a buscar.

Output: El número p, donde p es el número de veces que v aparece en A.

1:		//	Tiempo	Veces
2:	**for** $i = 1$ to $i = n$ **do**	//	c_1	$(n+1)$
3:	**if** $A[i] = v$ **then**	//	c_2	n
4:	$o := o + 1$	//	c_3	t
	return p	//	c_4	1

Para analizar el algoritmo uno podría preocuparse por el tiempo exacto de CPU que el algoritmo tomará, pues no todas las operaciones se ejecutan en exactamente un ciclo

de CPU o en una cantidad constante de ciclos, por ejemplo, *print i* seguramente tomará mucho más de un ciclo de CPU en prácticamente todas las computadoras, además de que tomaría más tiempo imprimir una cadena de caracteres que un solo entero.

La cuestión es que hacer un análisis que tome en cuenta el tiempo de CPU exacto requiere de una cantidad de trabajo que normalmente no queda justificada por el aumento de precisión de dicho análisis[1]. Un argumento análogo se tiene para el análisis del uso de memoria.

Para simplificar el trabajo sin que pierda sentido el análisis, uno puede optar por asignar un tiempo constante a cada una de las operaciones que realizará el algoritmo. Por ejemplo, en el caso del Algoritmo 0, se asumirá que el tiempo que tarda cada comparación del ciclo *for* toma un tiempo c_1, cada ejecución del enunciado *if* tomará un tiempo c_2, cada asignación de valor se ejecutará en un tiempo c_3 y el regresar el valor de salida del algoritmo tomará un tiempo c_4, siendo c_i una constante para $i \in \{1, 2, 3, 4\}$. Ya que establecimos cuánto tarda cada uno de los pasos del algoritmo, procederemos a ver cuántas veces se repite la ejecución de cada uno de éstos.

Tenemos que, aunque la comparación del *if* en el interior del el ciclo *for* se ejecutará n veces (una por cada elemento de A), solamente será positiva una cantidad t de veces[2], que será el mismo número de veces que se ejecutará la asignación de p. Cabe aclarar que la comparación del ciclo *for* sucederá $n + 1$ veces, n por el número de elementos que A tiene y 1 más porque es el momento en el que se da cuenta que $t > n$; regresar el valor se ejecutará una vez al final del algoritmo.

Antes de continuar con el análisis, uno tiene que preguntarse cómo estará constituida la entrada. Sabemos de antemano que tendrá n elementos, pero ¿Serán todos iguales o todos diferentes? ¿Estarán desordenados u ordenados? etc. Ahora es donde entra en juego el concepto de *el peor de los casos*.

Cuando se trabaja con algoritmos, se pueden tomar en cuenta el mejor de los casos, el caso promedio y el peor de los casos. El mejor de los casos ocurre cuando la entrada

[1] No se justifica para el diseño y análisis de algoritmos, sin embargo, cuando se trata de la implementación se debe ser más cauteloso.

[2] Puede suceder que $t = 0$ si $v \notin A$ o que $t = n$ si todos los elementos de A son iguales a v.

está constituida de manera que requiera la mínima cantidad de pasos necesaria para que el algoritmo termine con la respuesta correcta; el peor es cuando dicha constitución provoca que sea necesaria la mayor cantidad posible de operaciones; el caso promedio se usa para calcular cuál es la cantidad promedio de operaciones que se requerirán para terminar con la respuesta correcta. La principal ventaja de usar el caso promedio es que permite saber cuánto va a tardar, en promedio, la ejecución del algoritmo, la desventaja es que requiere de métodos bastante más complicados para ser obtenido; el mejor de los casos ocurrirá, en general, pocas veces y permite obtener una cota inferior para el tiempo de ejecución del algoritmo siendo analizado; el peor de los casos permite obtener una cota superior del tiempo que tardará un algoritmo en ejecutarse, es decir, garantiza que el algoritmo *nunca* tardará más con ninguna otra entrada. La principal ventaja, tanto del peor como del mejor de los casos, es que son frecuentemente más sencillos de construir que el caso promedio.

Con lo anterior en mente, se suele analizar un algoritmo con base en el peor de los casos.

El peor de los casos del Algoritmo 0 ocurre cuando todos los valores de A son iguales a v. Cuando eso sucede, $t = n$ y el tiempo total, $T(n)$, que tarda el algoritmo en ejecutarse es:

$$T(n) = c_1(n+1) + c_2n + c_3n + c_4 = (c_1 + c_2 + c_3)n + c_4 + c_1 \tag{1.4}$$

Vemos pues que el tiempo total de ejecución es $T(n) = C_1n+C_2$, con $C_1 = c_1+c_2+c_3$ y $C_2 = c_4+c_1$. Ahora usaremos las Definiciones 1.1.1 - 1.1.3 para analizar nuestro tiempo $T(n)$, y propondremos que la función $g(n)$ sea el término de mayor grado de nuestra ecuación, esto es, $g(n) = n$. Entonces:

- $T(n) = O(n)$ pues, recordando la Definición 1.1.1, tenemos que $C_1n + C_2 \leq Cn$, o lo que es igual, $C_1 + \dfrac{C_2}{n} \leq C$, para todo $n \geq n_0 = 1$, si escogemos $C = C_1 + C_2$.
- $T(n) = \Omega(n)$ pues, recordando la Definición 1.1.2, tenemos que $C_1n + C_2 \geq Cn$, para todo $n \geq n_0 = 1$, si escogemos $C = C_1$.

- Recordando la Definición 1.1.3 y usando los dos incisos anteriores tenemos que $T(n) = \Theta(n)$.

Cuando la función $T(n)$ es un polinomio se puede, en general, definir $g(n)$ como el término de mayor grado de $T(n)$.

Lo anterior es todo lo que se necesita saber sobre complejidad para la lectura del presente trabajo de tesis. Procederemos ahora a una revisión de los conceptos matemáticos por usar.

1.2. Principios matemáticos

En esta sección se estudiarán el principio de inducción y el de las casillas. El estudio de lo anterior servirá como base y preámbulo para los temas en las secciones y capítulos posteriores.

1.2.1. El principio de inducción

El *principio de inducción* es una herramienta matemática usada para demostrar proposiciones que dependen de los elementos de un conjunto bien ordenado, aunque nosotros nos conformaremos estudiándolo en situaciones donde dicho conjunto es el de los números naturales.

Es muy importante tener en cuenta que la prueba por principio de inducción se usa cuando ya se tiene una proposición definida y no como un primer paso para atacar el problema a probar. Para realizar una prueba echando mano del principio de inducción se pueden seguir, en general, las siguientes directivas:

Prueba por inducción de una proposición $P(n)$ con $n \in \mathbb{N}$:

1. **Caso base**: Primero se establece el caso base, es decir, probamos nuestra proposición para el natural más pequeño que se pueda.[1]

[1] La base de inducción suele ser $P(0)$ o $P(1)$, pero cualquier caso inicial funcionará.

2. **Hipótesis de inducción**: La hipótesis de inducción consiste en asumir que la proposición P se cumplirá para cualquier número entre el caso base y el caso $n-1$.

3. **Paso de inducción**: El paso final consiste en demostrar que $P(n-1)$ implica $P(n)$, o lo que es lo mismo, en usar $P(n-1)$ para demostrar $P(n)$. Una vez hecho lo anterior se termina la prueba.[1]

El método de inducción se analoga comunmente con el efecto dominó, pues se comienza colocandodo la primera ficha (verificando el primer paso), luego se alinean las fichas bajo la suposición de que cada ficha que caiga tirará a la siguiente (se hace la hipótesis de inducción) y se termina dando el empujón necesario para comprobar cómo las fichas caen una tras otra (análogo a lo que ocurre durante el paso de inducción).

Parte del encanto de realizar una prueba por inducción es que se asemeja mucho a la forma en que los humanos aprendemos: primero realizamos la tarea en los escenarios más sencillos, abstraemos lo que creemos que es el patrón general de la situación y suponemos que va a funcionar siempre. Al final usamos nuestro modelo abstraído para todos los casos que se nos presenten.

Con el efecto dominó la experiencia es indispensable para predecir cuándo alguna configuración va a funcionar; en el aprendizaje, la experiencia permite abstraer modelos más confiables. Lo mismo ocurre en el área de las matemáticas: en general, la experiencia juega un papel determinante a la hora de ejecutar una prueba correcta, en particular, es absolutamente necesaria a la hora de hacerlo por medio del método de inducción pues se tiene que ser extremadamente cuidadoso a la hora de establecer la hipótesis y en el momento de verificar la implicación del paso inductivo.

Para mostrar el uso del principio de inducción como método de demostración, vamos a recurrir a una historia bastante popular, una que se cuenta sucedió a finales del siglo XVIII y que tiene como protagonista a uno de los más grandes matemáticos de todos

[1] Este es el paso crucial y por eso hay que tener especial cuidado con lo que se hace en este punto de la prueba.

los tiempos: Carl Friedrich Gauss[1].

Ejemplo 1.2.1. Un día frío, como la mayoría de los días en la mayoría de las provincias alemanas, estaba el joven Gauss en un salón más bien decadente para los estándares modernos. Estaba él, como siempre, sentado en silencio.

Ese día su profesor se encontraba particularmente animado en clase, sentía que podía marcar la diferencia en la vida de aquellos pobres niños si es que conseguía inspirarlos de la manera adecuada. Decidió comenzar probando la destreza matemática de sus alumnos y propuso un problema más o menos sencillo que consistía en encontrar la suma de los enteros del 1 al 100. Los tres alumnos que terminasen primero y bien, podrían irse a casa sin tarea para el día.

El joven Gauss, tan talentoso como era, terminó casi de inmediato y descubrió que el resultado de cualquier suma de 1 a n estaría dado por $\frac{n}{2}(n+1)$. Debió ser la mirada estupefacta de su profesor o quizá entendía que no era tan obvio para los demás como para él, así que intentó explicar en el pizarrón cómo había llegado a su resultado. El profesor, que no era tonto, entendió de inmediato pero vio, también de inmediato, otra oportunidad para probar e inspirar a sus alumnos. Como el salón estaba lleno de caras de sorpresa, confusión y un poco de envidia, el profesor dejó como tarea que explicasen por qué Gauss tenía razón. El premio era, de nuevo, un día sin deberes para el que trajese la mejor explicación.

Al día siguiente, de entre todas las tareas, la primera que llamó su atención fue la siguiente:

"Primero calculamos un caso inicial, cuando $n = 1$. Así tenemos que la suma de 1 hasta 1 es 1, usando la fórmula $\frac{n(n+1)}{2} = \frac{2}{2} = 1$. Vemos que se cumple (**caso base**).

Lo que sigue será asumir que se cumple para cualquier número $n-1$. Es lo mismo decir que suponemos que la suma de los enteros de 1 hasta $n-1$ es igual a $\frac{(n-1)(n)}{2}$ (**hipótesis de inducción**).

[1] A pesar de la popularidad del relato, algunos detalles importantes no se pueden corroborar con registro histórico, como el tipo de problema y el método de solución [3]. Nótese que se hizo uso de una extensa libertad creativa en la versión aquí presentada.

Ahora voy a demostrar que sí se cumple para n, usando nuestra suposición:

$$1 + 2 + \cdots + (n-1) + n = (1 + 2 + \cdots + (n-1)) + n \tag{1.5}$$

Usando nuestra hipótesis de inducción y un poco de álgebra básica, de la Ecuación 1.5 obtenemos:

$$1 + 2 + \cdots + (n-1) + n = \frac{(n-1)n}{2} + n = \frac{(n-1)n + 2n}{2} = \ldots$$
$$\frac{n(n-1+2)}{2} = \frac{n(n+1)}{2}$$

Esto funciona porque demostramos que funciona para cualquier número n *si* funciona para el número $(n-1)$. Pero también demostramos que funciona para $n = 1$. Por eso, funcionará para $n = 2$, y por ello para $n = 3$, y así hasta cualquier número entero mayor que 1; se van cayendo los casos como fichas de dominó."

El profesor estaba impresionado porque un alumno había descubierto el principio de inducción. Aunque habría que averiguar quien, pues la explicación no estaba firmada. Tal vez esa sería otra buena tarea...

1.2.2. El principio de las casillas

Ahora toca estudiar un principio que tiene sus raíces bien enterradas en el sentido común, pero que enunciado y entendido correctamente se convierte en una herramienta muy poderosa. Para ilustrar el principio de las casillas imaginemos la siguiente situación.

En un reclusorio recién estrenado: los colchones huelen a nuevo, el acero inoxidable todavía sirve de espejo y las paredes siguen intactas. Imaginemos que dicho reclusorio cuenta con un total de n celdas.

Visitando la prisión en su primer fin de semana nos damos cuenta de que, durante el pase de lista, el número de reos es igual a n, resultando pues que tenemos tantos reos como celdas. Todos felices, nadie pelea porque cada uno tiene su propio rinconcito (su propia celda, para los que no perdonen el desliz literario).

Regresando un mes después nos encontramos con que han llegado nuevos internos pero ninguno ha salido. Eso quiere decir que tenemos más de n reos, es decir, más prisioneros que celdas ¿Será que cada reo estará solo en su celda o habrá quienes tengan que compartir? Si la respuesta parece obvia es porque lo es, si hay más reos que celdas, al menos una de las celdas tendrá que guardar a más de un reo.

Este es el principio de las casillas, que con términos un poco más matemáticos que mundanos se plantea como sigue:

Proposición 1.2.1 (El principio de las casillas)**.** *Si un conjunto con $n+1$ elementos, siendo $n \in \mathbb{N}$, se parte en n subconjuntos diferentes, al menos uno de ellos contendrá más de un elemento.*

Para entender la proposición anterior, es necesario dar la definición de *partición* de un conjunto S:

Definición 1.2.2. *Hacer una* partición *de o particionar un conjunto S en m subconjuntos S_i, con $0 < i \leq m$ y $S_i \neq \emptyset$, es dividir el conjunto S en m subconjuntos de manera que $S_i \cap S_j = \emptyset$ para $0 < i, j \leq m$, con $i \neq j$. Es decir, dividimos un conjunto S en subconjuntos disjuntos no vacíos.*

Regresando al lugar de encierro, cinco años hace ya que lo visitamos por última vez, nos encontramos con que la sociedad carcelaria ha crecido y evolucionado un poco más.

Ahora, hay muchos más reos que antes y los guardias, para facilitarse las cosas, simplemente los repartieron equitativamente en todas las celdas disponibles, de manera que hoy en día cada celda contiene a r reclusos distintos. Los reclusos decidieron que cada celda tenía que protegerse a sí misma y, como consecuencia, se formó una pandilla en cada celda.

Resumiendo, en la cárcel hay n pandillas (cada pandilla en su propia celda) con r miembros cada una.

Volviéndonos un poco políticos: todas las pandilla tienen el mismo número de miembros y se forma un equilibrio de poderes: los días transcurren en paz.

Cuando regresamos una semana después observamos que hubo muchos encarcelamientos durante la madrugada anterior, por alguna ola de violencia pre-apocalíptica, haciendo que en la cárcel ahora haya más de rn presos encarcelados.

A la hora de mandar los reos a su celda y tratando de hacerlo en partes iguales, los guardias terminan por darse cuenta que al menos en una de las celdas van a tener que meter a más de r reos. Sí, el equilibrio de poderes en la cárcel se ha vuelto a desbalancear, pero no nos preocupemos porque esta es la última vez que la visitamos.

Revisemos ahora la expresión matemática del caso anterior.

Proposición 1.2.3 (El principio de las casillas general)**.** *Si un conjunto con más de rn elementos, siendo $r, n \in \mathbb{N}$, se parte en n subconjuntos diferentes, al menos uno de ellos contendrá más de r elementos.*

Vamos a probarlo.

Demostración. Probaremos la proposición anterior por contradicción. Dado un conjunto S tal que $|S| > rn$, consideramos la partición en n subconjuntos, tal que $S = S_1 \cup S_2 \cup \cdots \cup S_n$. Así, con propósito de encontrar una contradicción, supondremos que $|S_i| \leq r$ para todo $i = 1, 2, \ldots, n$. Es decir, que ningún subconjunto de S tiene más de r elementos. Si lo anterior es cierto y sumamos todas las magnitudes $|S_i|$:

$$\begin{aligned} |S_1| + |S_2| + \cdots + |S_n| &\leq nr \\ |S| &\leq nr \end{aligned}$$

Lo anterior contradice nuestra afirmación inicial de que $|S| > rn$. □

Ejemplo 1.2.2. Propongamos la siguiente situación: En un viaje a un pueblito fantasma de Las Vegas, el lector se encuentra con la aparición de un viejo y conocido mal apostador de mediados del siglo XIX. Es media noche, la cantina tiene aire acondicionado, además, hay botana y alcohol de sobra, así que el lector decide pasar la noche allí. Luego de una plática larga y tendida, el viejo espectro le dice al lector:

"Te propongo un juego. Si me ganas te digo dónde está escondido el botín más grande de Las Vegas" —seguramente se referirá a la vieja Las Vegas, pero no importa, oro es oro— "Pero si pierdes, vas a tener que venir cada año, este mismo día, a jugar y platicar conmigo".

Luego de pensarlo un poco, eso de tener que escoger entre un tesoro y el pretexto perfecto para ir todos los años a Las Vegas, al lector le parece más bien un ganar-ganar. Decide aceptar.

"Muy bien, muy bien" —el fantasma se frota sus heladas e inmateriales manos— "el juego consiste en lo siguiente: te voy a dar cinco opciones para que escojas dos, las otras tres serán mías. Me ganarás si, usando todos los dados, consigues hacer que se repita algún tiro. Pero que conste que todos mis dados son diferentes, así que no es lo mismo, por ejemplo, un cinco en un dado que un cinco en otro; que conste también que por ser fantasma puedo escoger cómo van a caer los dados". Pareciera que los fantasmas no dan paso sin huarache pero no nos apresuremos.

"Puedes escoger cualquiera de las siguientes cosas:

1. El número de dados que vamos a usar.
2. El número de caras que tendrán los dados.
3. El número de tiros que serán hechos durante el juego.
4. El número de años de castigo que tendrás que sufrir no pueden ser menos de veinte.
5. Cualquier otra cosa que no interfiera con las reglas anteriores, tanto explícitas como implícitas".

Apresurado por ganar, el lector le puede decir al fantasma: pues escojo cualquier otra cosa, y esa cosa es ganar sin jugar.

El fantasma se sonrie —"Que no interfiera con las otras reglas, y la primera implícita es que ganas o que pierdes jugando. Otra de esas, listillo, y te doy un balazo fantasmal que te dolerá hasta los huesos"—.

Tiempo hay para pensar ¿Se le ocurre al lector una estrategia infalible para ganar? Deténgase aquí a meditar si no ha encontrado alguna solución pues las vacaciones de al menos los próximos veinte años dependen de ello, además una respuesta viene enseguida.

Una solución: Podemos pedirle al espectro usar la última y la del número de tiros. Usando la última le decimos que queremos que escoja antes de nosotros todos los números que le tocan. Así, supongamos que escoge usar d dados con c caras cada uno, y un castigo de 200 años. Al tratarse de d dados distinguibles con c caras cada uno, el número de combinaciones posibles es c^d; si, recordando el principio de las casillas, pedimos que el número de tiros sea $c^d + 1$, garantizamos que al menos una combinación se repita.

1.3. Gráficas y coloraciones

En el transcurso de la siguiente sección se mostrará un repaso de algunos conceptos básicos de teoría de gráficas que serán necesarios para continuar con la lectura del presente trabajo. Los lectores familiarizados con el tema pueden pasar a la siguiente.

1.3.1. Gráficas.

A continuación se presentarán las definiciones de gráfica, subgráfica y gráfica completa, así como de grado de un vértice.

Definición 1.3.1. *Una* gráfica, $G = (V, E)$, *es un par de conjuntos. Uno de ellos es un conjunto V denominado conjunto de vértices, donde cada* vértice *se puede imaginar como un punto; el conjunto E es un conjunto conformado por parejas de vértices llamadas* aristas, *que para nuestros fines pueden ser imaginadas a su vez como líneas, rectas o curvas, donde cada línea une un par de vértices o puntos.*

Definición 1.3.2. *Una* subgráfica $G' = (V', E')$ *de $G = (V, E)$ es una gráfica que cumple con las siguientes propiedades: $V' \subseteq V$ y $E' \subseteq E$, con la condición de que todos los elementos de E' quedan definidos por vértices en V'.*

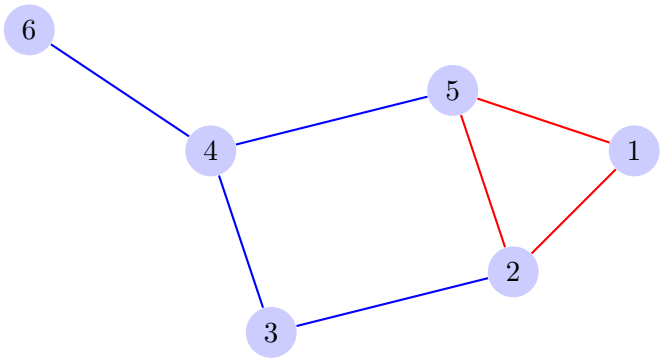

Figura 1.1: Ejemplo de una gráfica $G = \{V, E\}$ coloreada. El conjunto $V = \{1, 2, 3, 4, 5, 6\}$ de vértices y el conjunto $E = \{(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5), (4, 6)\}$ de aristas.

Definición 1.3.3. *Una* gráfica completa de orden n, *denotada por* K_n, *es una gráfica en la cual el conjunto de aristas contiene todas las parejas entre dos vértices del conjunto* V. *Equivalentemente, todos los vértices se encuentran conectados unos con otros.*

Definición 1.3.4. *El* grado de un vértice *es el número de aristas que lo contienen.*

Para ilustrar los conceptos anteriores haremos uso de la gráfica G mostrada en la Figura 1.1: Cada uno de los círculos enumerados representa un vértice distinto, y cada línea representa una arista conformada por los dos vértices que dicha línea une. En el ejemplo, la figura formada por $G' = \{V', E'\}$, donde $V' = \{1, 2, 5\}$ y $E' = \{(1, 2), (1, 5), (2, 5)\}$, es una subgráfica de G. Esta misma gráfica es la gráfica completa K_3 y cada uno de sus vértices tiene grado 2 porque cada vértice está contenido en dos aristas distintas.

1.3.2. Coloraciones.

Aparte de los conceptos de gráficas presentados, necesitaremos también algunos referentes a la teoría de coloraciones. A continuación vienen algunas definiciones.

Definición 1.3.5. *Una* r-coloración *de un conjunto* S, *coloreado con colores de un conjunto* C *con* $|C| = r$, *es una función*

$$\chi : S \to C. \tag{1.6}$$

Dado un conjunto coloreado, resulta interesante estudiar algunos subconjuntos de este: como aquellos cuyos elementos tienen todos el mismo color, o esos que tienen todos los colores diferentes. A saber:

Definición 1.3.6. *Dada una coloración* $\chi : S \to C$, *con* $|C| = r$, *y un subconjunto* $S' = \{s'_1, s'_2, \ldots, s'_m\}$, *con* $S' \subseteq S$, *decimos que el conjunto* S' *es:*

1. monocromático *si* $\chi(s'_1) = \chi(s'_2) = \cdots = \chi(s'_m)$.

2. heterocromático *si* $\chi(s'_i) \neq \chi(s'_j)$ *para cualesquiera* $s_i, s_j \in S'$ *con* $i \neq j$.

Usualmente las coloraciones las visualizamos precisamente como una asignación de colores. Por ejemplo, regresando a la Figura 1.1, tenemos que $S = E$ y el conjunto de colores es $C = \{$ rojo , azul $\}$, con la 2-coloración $\chi(E)$ definida como:

$$\chi(E) = \begin{cases} \chi((1,2)) = \chi((1,5)) = \chi((2,5)) = \text{rojo} \\ \chi((2,3)) = \chi((3,4)) = \chi((4,5)) = \chi((4,6)) = \text{azul}. \end{cases}$$

En dicha figura, la subgráfica $G' = \{V', E'\}$, donde $V' = \{1, 2, 5\}$ y $E' = \{(1,2), (1,5), (2,5)\}$, es monocromática de color rojo y la subgráfica $G'' = \{V'', E''\}$, con $V'' = \{1, 2, 3\}$ y $E'' = \{(1,2), (2,3)\}$ es heterocromática.

Es conveniente mencionar en este punto que una coloración es una función que va de un conjunto cualquiera a otro. En capítulos posteriores, tanto S como C serán conjuntos cuyos elementos son enteros o conjuntos de enteros. Específicamente:

Definición 1.3.7. *Una* r-coloración *de un conjunto* $[n] = \{1, 2, \ldots, n\}$ *con colores* $[r] = \{1, 2, \ldots, r\}$, *es una función*

$$\chi : [n] \to [r]. \tag{1.7}$$

En el presente trabajo, para representar las coloraciones y varicoloraciones de enteros (la definición de varicoloración se encuentra en la Definición 1.3.8), se escribirán dos renglones: en el renglón de la parte superior se escribirá el elemento del conjunto

$[n]$ y debajo de cada elemento se escribirá el color o varicolor correspondiente a dicha posición.

A modo de ejemplo tómese la siguiente 2-coloración de enteros:

$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 1 & 2 & 2 & 1 & 1 & 2 \end{array} \tag{1.8}$$

También haremos uso de una generalización de coloración, a la que se denomina varicoloración [4], a saber:

Definición 1.3.8. *Dados un entero n y un conjunto de colores $C = \{1, 2, \ldots, r\}$, una* varicoloración λ *se define como sigue:*

$$\lambda : \{1, \ldots, n\} \to S(C)\backslash\emptyset \tag{1.9}$$

Donde $n \in \mathbb{N}$ y $S(C)$ es el conjunto de todos los subconjuntos posibles de C, también denominado conjunto potencia de C.

Por ejemplo, si tenemos dos colores, $\{1, 2\}$, entonces $S(\{1, 2\}) = \{\{1\}, \{2\}, \{1, 2\}\}$. Como ya se mencionó con anterioridad, una varicoloración es una generalización del concepto de coloración: en la coloración asignamos un color, y solo uno, a cada entero; en la varicoloración podemos asignar más de uno por medio de los subconjuntos $T \in S(\{1, \ldots, r\})\backslash\emptyset$. Llamaremos varicolor a T, como analogía con las coloraciones y los colores, nótese que un color es un varicolor de tamaño 1. Un ejemplo de varicoloración sería el siguiente:

$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \{1\} & \{1, 2\} & \{2\} & \{1\} & \{1, 2\} & \{2\} & \{2\} \end{array}$$

Y con esto damos por terminada esta sucinta revisión sobre gráficas y coloraciones.

1.4. El teorema de Ramsey

En esta sección vamos a estudiar el teorema de Ramsey para dos colores. Antes, para facilitar la tarea de la demostración, vamos a revisar el lema que sigue:

Lema 1.4.1. *Dados cuatro números* $A, B, R_1, R_2 \in \mathbb{N}$*, si* $R_1 + R_2 = n$ *y* $A + B = n - 1$ *entonces* $A \geq R_1$ *o* $B \geq R_2$.

Demostración. Probaremos el lema anterior por contradicción. Para ello supondremos que $A < R_1$ y $B < R_2$. Entonces:

$$\begin{aligned} A &\leq R_1 - 1 \\ B &\leq R_2 - 1 \end{aligned}$$

Sumando ambas desigualdades queda:

$$A + B \leq R_1 + R_2 - 2$$

Como $R_1 + R_2 = n$ entonces $A + B \leq n - 2$, que es una contradicción a $A + B = n - 1$ □

Enunciaremos a continuación el teorema de Ramsey para dos colores.

Teorema 1.4.2 (Teorema de Ramsey para dos colores)**.** *Dados dos enteros positivos cualesquiera, digamos* k *y* l*, existe un mínimo número entero* $R = R(k, l)$*, tal que toda 2-coloración de las aristas de* K_R *contiene al menos una* K_k *monocromática del primer color o una* K_l *monocromática del segundo color.*

Los mínimos enteros $R = R(k, l)$, definidos en el Teorema 1.4.2, son denominados *números de Ramsey.*

Antes de pasar a la demostración del Teorema 1.4.2, y con la finalidad de entender mejor, vamos a revisar un ejemplo muy popular que habla sobre conocidas y desconocidas:

Ejemplo 1.4.1. Demuéstrese la siguiente afirmación: *En una fiesta de seis personas, digamos Adriana, Amanda, Beatriz, Edgardo, Miguel y Saúl, al menos tres personas se conocen todas entre sí o al menos tres son desconocidos todos entre sí.*

Antes de comenzar la demostración vamos a transformar el problema aprovechando las herramientas que conocemos: Usaremos la gráfica completa K_6 y diremos que cada

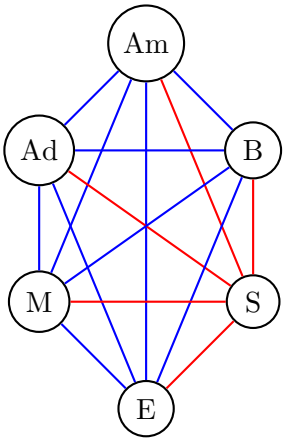

Figura 1.2: 2-coloración[1] de $E(K_6)$.

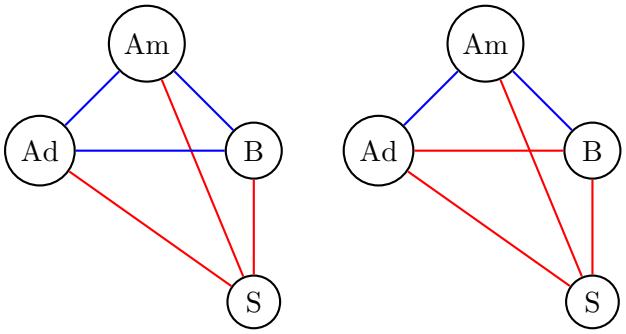

Figura 1.3: Subgráfica K_4 en la que Ad, Am y B se conocen (izquierda). Subgráfica K_4 con cambio de color en la arista (Ad,B) en la que S, B y Ad son desconocidas (derecha).

uno de los vértices representa una persona (ver Figura 1.2); si las dos personas se conocen, la arista que los une será de color azul, en caso contrario, será roja.

Ya que convertimos el problema a una gráfica completa cuyo conjunto de aristas está 2-coloreado, vemos que para demostrar que al menos tres personas se conocen todas entre sí o no se conocen todas entre sí bastará con encontrar un triángulo monocromático. Continuemos con la demostración.

Demostración. Tomamos un vértice cualquiera, digamos S, y las 5 aristas incidentes a éste. Debido a que solo tenemos dos posibles colores, se tiene que, por el principio de las casillas, al menos tres aristas deberán tener el mismo color. Nos quedaremos con

[1] La coloración se escogió de manera arbitraria. Aunque pudo haber sido cualquier otra.

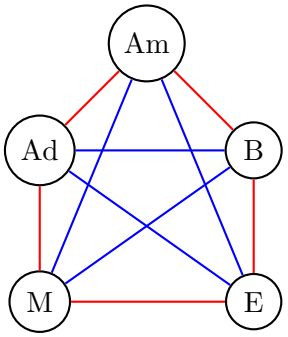

Figura 1.4: 2-coloración de K_5 libre de triángulos monocromáticos.

esas tres aristas del mismo color y, sin pérdida de generalidad supondremos que son rojas. Nos quedamos también con los vértices conectados a dichas aristas, digamos Ad, Am y B y con todas las aristas que hay entre ellos.

Tenemos pues que S es un desconocido de Am, Ad, y B (debido a nuestra convención de colores), y por ello se nos presentan solamente dos escenarios posibles para terminar la demostración: o cualquier pareja de las tres son desconocidos, en cuyo caso esas dos (junto con S) formarán un triángulo rojo de desconocidos (ver Figura 1.3 (izquierda)); o tanto Am, Ad y B se conocen, por lo que entre sus vértices todos las aristas serán azules y se formará un triángulo monocromático, es decir, se conocen las tres (ver Figura 1.3 (derecha)).

En cualquiera de los dos casos se forma un triángulo monocromático, con lo que se demuestra que, sin importar cómo se coloreen las aristas de la gráfica K_6, siempre se encontrará un triángulo monocromático. Lo anterior es equivalente a decir que habrá al menos tres completos desconocidos o al menos tres buenos conocidos en cualquier fiesta de seis personas. □

¿Y sucedería lo mismo si la fiesta fuese de 5 personas? La respuesta es no. De entre todas las posibles fiestas de cinco personas existe una, la representada por la Figura 1.4, que no contiene ningún triángulo monocromático.

Antes de regresar a la demostración del teorema de Ramsey para dos colores veamos más de cerca lo que acabamos de hacer usando la lente del Teorema 1.4.2. Por medio del Ejemplo 1.4.1 demostramos que toda 2-coloración de las aristas de K_i, con $i \geq 6$,

contendrá al menos un triángulo monocromático, es decir, que $R(3,3) \leq 6$; con la Figura 1.4 demostramos que existe al menos una 2-coloración de las aristas de K_5, que no contiene ningún triángulo, o lo que es lo mismo, que $5 < R(3,3)$. Tenemos pues:

$$5 < R(3,3) \leq 6 \tag{1.10}$$

Lo anterior implica que $R(3,3) = 6$.

Demostraremos también que:

Lema 1.4.3. *$R(2,l) = l$ y $R(k,2) = k$.*

Solo demostraremos $R(2,l) = l$ porque $R(k,2) = k$ es análogo. En fin, lo que debemos demostrar es que, sin importar cómo coloreemos las aristas de K_l con dos colores, siempre terminaremos formando una K_2 del primer color o una K_l del segundo color.

Demostración. Al colorear K_l con dos colores, podemos dividir todas las coloraciones en dos casos:

1. Al menos una arista será coloreada con el primer color.
2. Ninguna arista es coloreada usando el primer color.

Si fuese el caso primero existe al menos una arista del primer color y, con ello, una K_2 monocromática de dicho color, con lo cual terminamos. Si fuese el segundo en que ninguna arista es del primer color, se implica que todas las aristas de K_l son del segundo color. Entonces $R(2,l) \leq l$.

Para demostrar que $R(2,l) > (l-1)$ bastará con encontrar una 2-coloración de las aristas de K_{l-1} que no contenga una K_2 del primer color o una K_l del segundo color, como la que resulta de colorear todas las aristas de K_{l-1} del segundo color. Con esto se implica que $R(2,l) = l$.

De manera semejante se demuestra que $R(k,2) = k$. □

Ahora sí regresamos a la demostración del teorema de Ramsey para dos colores. Para ello, bastará con demostrar la existencia de una cota superior para $R(k,l)$. La demostración se llevará a cabo por medio de inducción sobre $k+l$; en el proceso haremos uso de los Lemas 1.4.1 y 1.4.3. Por último, con fines didácticos, propondremos que el primer color sea el rojo y el segundo sea el azul.

Demostración. Vamos a probar que existe $R(k,l)$ para todo $k,l \geq 2$ por medio de inducción sobre $k+l$.

Nuestro caso base es cuando $k+l=4$, caso que se encuentra demostrado en el Lema 1.4.3 pues este muestra que tanto $R(k,2)$ como $R(2,l)$, para $k,l \geq 2$ existen.

A partir de lo anterior, contruimos nuestra hipótesis de inducción: supondremos que tanto $R(k,l-1)$ como $R(k-1,l)$ existen.

Ahora tenemos que demostrar, usando la hipótesis de inducción, que $R(k,l)$ existe.

Procedemos proponiendo $n = R(k-1,l) + R(k,l-1)$ (debido a nuestra hipótesis de inducción sabemos que n existe). Ahora vamos a tomar un vértice v cualquiera de K_n. Sabemos que se trata de una gráfica completa y, por tanto, v tiene grado $n-1$. Dividiremos las $n-1$ aristas conectadas a v como $A =$ número de aristas rojas conectadas a v y $B =$ número de aristas azules conectadas a v. Ahora usamos el Lema 1.4.1, con $A = A$, $B = B$, $R_1 = R(k-1,l)$ y $R_2 = R(k,l-1)$, para demostrar que:

$$\begin{aligned} A &\geq R(k-1,l) \\ &o \\ B &\geq R(k,l-1) \end{aligned}$$

Sin pérdida de generalidad suponemos $A \geq R(k-1,l)$ y tomaremos la subgráfica K_A generada por todos los vértices conectados a v por medio de una arista roja. Por la hipótesis de inducción sabemos que K_A que contiene o bien K_l azul, en cuyo caso terminamos, o bien contiene K_{k-1} roja. En este último caso recordamos que el vértice v está conectado, por medio de aristas rojas, a una K_A y, por eso mismo, a todos los vértices de K_{k-1}. Así, la gráfica resultante de conectar v por medio de aristas rojos

a todos los vértices de K_{k-1}, es un gráfica K_k con todos sus aristas de color rojo, demostrando así que $R(k,l) \leq R(k,l-1) + R(k-1,l)$.

□

1.4.1. Números de Ramsey, su dificultad y su relación con las computadoras

Hagamos un experimento mental basado en un mundo apocalíptico que alguna vez usó Erdős[1] como ejemplo [2], uno donde la vida se ponga en riesgo (una vida hipotética, por supuesto): Supongamos que ocurre un día en que usted despierta a media noche, abre los ojos y no puede moverse pero si escucha un zumbido que de repente se interrumpe para dar paso a una voz suave y dulce "Buenas noches terrícolas, es un placer conocerlos por fin. Hay leyendas allá afuera que dicen que son ustedes tremendamente creativos y que su inteligencia no tiene límites. Nosotros somos una raza de Alfa Centauri y no venimos en son de paz– *No venimos en son de paz* resuena en su cabeza, pero ese eco es todo culpa de la preocupación— "Venimos más bien a jugar un juego. Este juego se llama: denme el número de Ramsey para $k = r$ y $l = r$ o los aniquilamos. Empecemos con $r = 6$".

Antes que hacerle caso a una visión de media noche, prefiere todo el mundo culpar al estrés o a una esquizofrenia espontánea y temporal. Pero al día siguiente la noticia está en todos los canales y redes sociales disponibles en ese futuro hipotético. Está junto a otra que dice que el reclusorio del ejemplo anterior desapareció sin más explicación que la de una detonación termonuclear, cuya radiación leída con un contador Geiger estándar y traducida del código morse dice: Empiecen ahora.

Bueno, relajémonos un poco y regresemos a un pasado donde todo era mejor (mejor porque no estábamos al borde de un apocalipsis alienígena). La historia de los números

[1] **Paul Erdős (1913 – 1996)**. Fue uno de los matemáticos más importantes y prolíficos del siglo XX. Citando uno de los obituarios en su honor: *"Nunca, dicen los matemáticos, ha habido un individuo como Paul Erdős. Fue uno de los matemáticos más grandes de este siglo, propuso y resolvió peliagudos problemas en teoría de números y otras áreas. Fundó además el campo de las matemáticas discretas, campo que sirve como base de la ciencia de la computación. Él fue uno de los matemáticos más prolíficos en la toda la historia, con más de 1,500 artículos con su nombre. Y fue también, dicen sus amigos, uno de los más inusuales."* [5] (Traducción del autor)

de Ramsey comienza con el nacimiento de Frank Plumpton Ramsey un día 22 de febrero de 1903 en Cambridge, hijo de Arthur Stanley Ramsey, matemático y director del Magdalene College, y de Mary Agnes Stanley, quien se graduó con honores en historia moderna.

Ramsey fue un hombre con una mirada preguntona pero severa y una sonrisa más pícara que burlona, fue filósofo, matemático y economista; además de eso, era también un amante de la literatura, un adepto del psicoanálisis, un apóstol de Cambridge, un políglota talentoso, fue incluso un romántico descarado y depresivo. Era pues un curioso empedernido (quizá la última implique todas las anteriores, y más, pero mejor no arriesgarse).

Fue su vida muy interesante, su muerte, repentina y trágica. Probó en 1928 el teorema que lleva su nombre, unos dos años antes de morir; lo probó con el propósito de usarlo como herramienta en la solución de un problema de lógica formal. Y aunque ahora sabemos que no hubiese sido útil para el propósito original de Ramsey, el teorema se quedó aguardando por algún matemático curioso. Y matemáticos curiosos nunca han faltado.[1]

Regresemos ya al mundo donde los extraterrestres están por exterminarnos, como en toda situación de supervivencia tenemos tres opciones: congelarnos, huir o pelear.

Huir queda descartado pues nuestros proyectos espaciales apenas pueden llevarnos a la luna, mucho menos pensar en escapar de una nave que puede moverse entre sistemas solares. Pelear es arriesgar mucho pues una guerra nuclear en nuestro planeta solo terminaría por matarnos a todos. Así que, en general, el mundo completo decide detenerse y aprovechar todos los recursos disponibles para cumplir con las demandas; en particular, hubo que dejar intactos los sistemas necesarios para mantener a la sociedad andando, además de quienes no se pudieron despegar de las redes sociales y otras ociosas costumbres.

Lo que sigue es hacer un chequeo de nuestras posibilidades. Para lograrlo tomaremos como referencia el poder computacional estimado en 2007 [7] y, felizmente, supondremos

[1] Se puede encontrar una muy interesante biografía en [6].

que ese poder se duplica cada año. Así, la capacidad de la tecnolgía humana de procesar 6.4×10^{18} operaciones por segundo en 2007 llegaría hoy a 3.28×10^{21}. Tomaremos también como referencia las cotas dadas para $R(6,6)$ hasta la fecha: $102 \leq R(6,6) \leq 165$.

Si analizásemos por fuerza bruta una sola 2-coloración de K_n para comprobar la existencia de una K_6 monocromática, nos encontraríamos revisando todas las K_6 generadas por K_n, es decir, todos los subconjuntos de 6 vertices de un conjunto con n vértices. Si usamos las cotas conocidas tenemos que $n \in \{102, 103, \ldots, 165\}$.

El proceso de revisión que describimos arriba, sería el primer paso para poder verificar que en alguna de las gráficas analizadas existe un contraejemplo (es decir, una 2-coloración de K_n que no contenga K_6 monocromática) y así poder mejorar la cota inferior. Decimos que es un primer paso porque habría que hacer eso mismo para cada una de las coloraciones posibles de K_n. Pero comencemos despacio.

El número total, T, de K_6 generadas por K_n estará dado por $T = \dfrac{n!}{(n-6)!6!}$. Para $n = 102$ el número de subgráficas sería $T \approx 2.7 \times 10^9$. Aparte de lo anterior, habría que visitar cada una de las aristas de K_6 y compararla contra todas los demás. Supongamos que visitar cada subgráfica K_6 nos toma 1 sola operación y que podemos comparar todas las aristas contra todas en una sola operación también. En ese caso, tendríamos que realizar algo así como $O = 2 \times T$ operaciones para comprobar que una 2-coloración de K_{102} no tiene K_6 monocromática. Pero el número de 2-coloraciones posibles de las aristas de K_{102} es 2^{102}, por ello, el número de operaciones necesarias sería $(2^{102})(2.7 \times 10^9) \approx 1.4 \times 10^{40}$. Y eso es solo para revisar cuando $n = 102$.

Pero los aliens tenían razón en que somos una raza bastante creativa, así que ahora seremos generosos y convocaremos otra lluvia de suposiciones. Empezaremos suponiendo que el esfuerzo de todos los matemáticos disminuyó la cota superior a 121 y aumentó la inferior a 120; asumiremos también que todos los computólogos juntos encontraron un algoritmo para reducir a el número de operaciones necesarias para revisar cada coloración a una millonésima parte del procedimiento del párrafo anterior, esto es, $O \approx 7.3E^3$ por cada 2-coloración de K_{120}. Supondremos finalmente que juntos,

computólogos y matemáticos, redujeron el problema computacional a revisar solo una billonésima parte de las 2-coloraciones de K_{120}. Al resto de los mortales nos tocó, entre otras cosas, rezar.

Luego del tiempo necesario para la logística, el cálculo y las oraciones, va el embajador de la humanidad a entregar el resultado (en realidad se comunicó la respuesta usando una frecuencia secreta acordada previamente).

"La respuesta es correcta" —Se escucha telepáticamente en la cabeza de todo el mundo y la celebración tiene un gran estallido inicial. Segundos después el mundo completo enmudece— "Ahora queremos el número de Ramsey para $r = 7$".

No hay mucho que pensar, es tiempo de pasar al plan B. Los humanos tenemos que pelear.

Capítulo 2

Números de Van der Waerden

Lo que sigue es ver la teoría necesaria para entender los números de Van der Waerden de manera formal y, con ello, ver también cuales son los números que se conocen actualmente. También se introducirán algunos de los algoritmos usados para su determinación, así como unas pocas definiciones necesarias para simplificar su expresión. Se seleccionaron algoritmos que por su sencillez pueden ser analogados al caso heterocromático. Sin más, procedamos.

2.1. Números de van der Waerden

En el Capítulo 1 se introdujeron de manera intuitiva tanto la noción de una k-progresión aritmética como aquella de los números de Van der Waerden. Es tiempo ya de definirlos formalmente, comencemos con la de una k-progresión aritmética:

Definición 2.1.1. *Una* ***progresión aritmética de*** k ***términos****, también denominada* k***-progresión aritmética****, con diferencia* d *es un conjunto de números enteros con la siguiente estructura*

$$\{a, a+d, a+2d, ..., a+(k-1)d\},$$

donde $a \in \mathbb{Z}$ *y* $d, k \in \mathbb{Z}^+$.

Antes de pasar al teorema de Van der Waerden, se aclara que se hará uso de las definiciones dadas en 1.3.2. En este capítulo la coloración será una función cuyo dominio

es el conjunto $[n]$ y cuyo rango es $[r]$. Además se tomarán como subconjuntos las progresiones aritméticas contenidas en el dominio.

Regresando a lo que nos incumbe, hay que destacar que la obtención de los números de Van der Waerden es un problema tipo Ramsey que se deriva del Teorema 2.1.2, mejor conocido como teorema de Van der Waerden.

Teorema 2.1.2 (**Teorema de Van der Waerden**, versión finita)**.** *Dados dos números enteros positivos cualesquiera, digamos k y r, existe un mínimo número entero $w = w(k;r)$ tal que cualquier r-coloración de $[w]$, siendo $[w] = \{1, 2, 3, \ldots, w\}$, contiene al menos una k-progresión aritmética monocromática.*

Los números $w = w(k;r)$ son los llamados números de Van der Waerden. Cabe destacar que el teorema anterior prueba solamente la existencia de dichos números, no los calcula.

Aunque los números de Van der Waerden parezcan tan inocentes, al igual que los números de Ramsey, son tremedamente difíciles de obtener. Todos los números de Van der Waerden conocidos y no triviales[1], son los siguientes:

$$w(3;2) = 9, \quad w(4;2) = 35, \quad w(5;2) = 178, \quad w(6;2) = 1132,$$
$$w(3;3) = 27, \quad w(4;3) = 293,$$
$$w(3;4) = 76.$$

Además de los números de Van der Waerden definidos por el Teorema 2.1.2 existe una variante, los números mixtos de Van der Waerden, que a algunos les parecerá se asemejan más a los números de Ramsey, pues permiten asociar a cada color $t \in [r]$ un k_t. Se definen como sigue:

Definición 2.1.3. *Dados $r \in \mathbb{N}$ y una secuencia de enteros positivos $\{k_1, k_2, \ldots, k_r\}$, el* número mixto de van der Waerden, *$w = w(k_1, k_2, \ldots, k_r; r)$, es el mínimo entero tal*

[1]Los números de Van der Waerden triviales son $w(2;r) = r + 1$. Lo anterior es porque, por el principio de las casillas, al menos dos números de $[r + 1]$ tendrán el mismo color sin importar cómo se coloree $[r + 1]$. Por otro lado, un conjunto $[r]$ elementos se puede colorear usando un color para cada número, evitando así una 2-PA monocromática. Por ello $r < w(2;r) \leq r + 1$.

que toda r-coloración de $[w]$ contiene al menos una k_t-PA monocromática de color t, para algún $t \in \{1, 2, \ldots, r\}$

Observemos que, de acuerdo con el Teorema 2.1.2 y la Definición 2.1.3:

$$w(k; r) = w(\underbrace{k, k, \ldots, k}_{r \text{ veces}}; r)$$

Por ejemplo $w(3; 2) = w(3, 3; 2)$ porque $w(3; 2)$ es el entero mínimo positivo que garantiza que cualquier 2-coloración de $[w]$ contendrá una 3-PA de color 1 o una 3-PA de color 2, que es por definición $w(3, 3; 2)$.

El Teorema 2.1.2 nos dice que $w(k; r)$ existe, lo cual implica la existencia de los números mixtos de Van der Waerden ya que $w(k_1, k_2, \ldots, k_r) \leq w(k; r)$ donde $k = max(k_1, \ldots, k_r)$.

Se conocen más números mixtos de Van de Waerden, 119 a la fecha de esta redacción (enero 2017). Todos los números conocidos para $r \leq 3$ están dados en las Tablas 2.1-2.3.

$k_1 \backslash k_2$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	6	7	10	11	14	15	18	19	22	23	26	27	30	31	34	35	38
3	9	18	22	32	46	58	77	97	114	135	160	186	218	238	279	312	349
4	18	35	55	73	109	146											
5	22	55	178	206													
6	32	73	206	1132													

Tabla 2.1: Números de Van der Waerden conocidos con dos colores, $\boldsymbol{w(k_1, k_2; 2)}$.

$k_2 \backslash k_3$	3	4	5	6	7	8	9	10	11	12	13	14
3	14	21	32	40	55	72	90	108	129	150	171	202
4	21	40	71	82	119	157						
5	32	71	180	246								

Tabla 2.2: Números de Van der Waerden conocidos con tres colores y $k_1 = 2$, $\boldsymbol{w(2, k_2, k_3; 3)}$.

Por último, tenemos que $w(4, 4, 4; 3) = 293$.

Ahora toca hablar de algunos algoritmos usados para calcular los números de Van der Waerden. Nos concentraremos en aquellos que han servido de inspiración para el presente trabajo.

$k_2 \backslash k_3$	3	4	5	6
3	27	51	80	107
4	51	89		

Tabla 2.3: Números de Van der Waerden conocidos con tres colores y $k_1 = 3$, $\boldsymbol{w(3, k_2, k_3; 3)}$.

2.2. Algoritmos sobre números de Van der Waerden

El primer algoritmo es usado para encontrar las progresiones aritméticas monocromáticas más largas y, aunque no tiene como propósito hallar los números de Van der Waerden, es indispensable para calcular los mismos.

2.2.1. Algoritmo para hallar las progresiones aritméticas más largas

En [8] se enuncia un algoritmo eficiente para encontrar la progresión aritmética más larga contenida un conjunto de n elementos. Muestra Erickson que dicho algoritmo tiene complejidad $O(n^2)$.

Schweitzer muestra, en [4], una adaptación de dicho método pensada para encontrar la progresión aritmética monocromática más larga de un color t contenida en el conjunto $\{1, 2, \ldots, n\}$. En este trabajo se mostrará la adaptación de dicho algoritmo para obtener simultáneamente todas las progresiones aritméticas más largas de todos los colores.

El algoritmo toma como entradas la coloración χ de $\{1, 2, \ldots, n\}$ y el número de colores r.

Durante el proceso se usan además: Un arreglo l de r elementos, en cuyo t-ésimo elemento se guardará la longitud de la progresión aritmética monocromática más larga de color t, con $t \in \{1, \ldots, r\}$; se usa también una matriz L de $n \times n$ elementos que contendrá, en $L[i, j]$[1], la longitud de la progresión aritmética monocromática, de color $\chi[i]$[2], tal que el primer elemento es i y el segundo elemento, en caso de ser del mismo color, es j, con $i, j \in \{1, 2, \ldots, n\}$ e $i < j$. Si ambos tienen colores distintos se establece

[1] En este texto, dada una matriz A, $A[i, j]$ representa el elemento en el i-ésimo renglón y la j-ésima columna.

[2] Durante las discusiones de los algoritmos $\chi[i]$ y $\chi(i)$ serán intercambiables, debido a que la coloración puede y suele representarse como un arreglo cuando el algoritmo se implementa.

que la longitud de la PA monocromática es 1.

La salida del algoritmo será el arreglo l que contendrá la longitud de las progresiones aritméticas más largas para cada uno de los colores disponibles.

El algoritmo procede evaluando cada una de las posibles parejas i, j con $i < j$, tal que $j - i = d$.

Para cada pareja de valores se sigue este procedimiento: si el color de i es distinto del color de j, entonces $L[i, j] := 1$, si tenemos que el color de i y el de j son iguales y, además, que el elemento $j + d \leq n$ entonces $L[j - d, j] := L[j, j + d] + 1$. Para el caso final, los colores de i y j son iguales pero $j + d > n$, por ello $L[j - d, j] := 2$.

Nótese que la idea más importante es que el número de miembros de la progresión aritmética $\{j - d, j, \dots\}$, es igual al número de miembros de $\{j, j + d, \dots\}$ más 1 (por el elemento $j - d$ que está siendo agregado). El algoritmo descrito, que en general se conoce como programación dinámica, se muestra en Algoritmo 1.

Algoritmo 1

Input: Una coloración χ del conjunto $\{1, \dots, n\}$ y el número de colores, r, de la coloración χ

Output: Un arreglo l de longitud r, tal que $l[t]$ es la longitud de la progresión aritmética más larga de color $t \in \{1, \dots, r\}$.

```
1:  for t = 0 to t = r do
2:      l[t] := 0
3:  for j = n down to j = 1 do
4:      for d = 1 to d = j − 1 do
5:          if χ[j − d] ≠ χ[j] then
6:              L[j − d, j] := 1
7:          else if j + d ≤ n then
8:              L[j − d, j] := L[j, j + d] + 1
9:          else
10:             L[j − d, j] := 2
11:     if L[j − d, j] > l[χ[j − d]] then
12:         l[χ[j − d]] := L[j − d, j]
    return l
```

Cabe decir que valdrá la pena almacenar los valores $L[i, j]$ pues se usarán en el algoritmo de la Sección 2.2.2.

2.2.2. Algoritmo para obtener una cúspide

Antes de estudiar el algoritmo agrandaremos nuestra lista de definiciones con un par que necesitaremos de inmediato. La primera es la de una coloración libre.

Definición 2.2.1. Coloración libre: *dado un vector* $\langle k_1, \ldots, k_r \rangle$*, decimos que una coloración* $\chi : [n] \to [r]$ *es* libre *si no contiene ninguna* k_i*-progresión aritmética monocromática de color* i*, para toda* $i \in [r]$*.*

A modo de ejemplo retomemos la coloración 1.8 dada como ejemplo en la sección 1.3.2 y agreguemos un elemento más:

$$\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 1 & 2 & 2 & 1 & 1 & 2 & 1 \end{array} \tag{2.1}$$

Si verificamos la coloración 2.1 con el vector $\langle 3, 3 \rangle$, nos daremos cuenta de que no es libre pues los elementos 2, 5 y 8 forman una 3-progresión aritmética con el color 1. Pero si la verificamos usando el vector $\langle 4, 3 \rangle$, el resultado será que sí es una coloración libre pues no hay una 4-progresión aritmética de color 1 ni una 3-progresión aritmética de color 2.

La siguiente definición que nos compete es la de cúspide de una coloración.

Definición 2.2.2. *Sean* $r, k_1, k_2, \ldots, k_r$ *enteros positivos y* w_{lb} *un entero positivo tal que* $w_{lb} \leq w(k_1, \ldots, k_r; r)$*. Sea* χ *una coloración de* $[i]$ *libre respecto al vector* $\langle k_1, \ldots, k_r \rangle$*, con* $i < w_{lb}$*. Llamamos cúspide a la posición* $i' \in \{i+1, \ldots, w_{lb}\}$ *si, para todo* $t \in \{1, \ldots, r\}$*,* i' *cumple que existe una* $(k_t - 1) - PA$ *monocromática,* $\{a_1, a_2, \ldots, a_{k_t - 1}\}$*, en* $[i]$ *tal que* $\{a_1, a_2, \ldots, a_{k_t-1}, i'\}$ *es una* k_t*-PA.*

Una cúspide i' es la garantía de que la coloración actual de $[i]$ *no* es útil para nuestra búsqueda. Para ello es necesario haber encontrado antes una coloración libre de $[w_{lb}]$, con $w_{lb} > i$. La cúspide es una posición, i', entre $i+1$ y w_{lb}, que al ser coloreada con cada uno de los colores $\{1, \ldots, r\}$ va a generar una k_t-PA monocromática con los elementos de $[i]$; por lo anterior dicha coloración de $[i]$ no sirve porque tenemos garantizado que no

podremos encontrar una coloración libre en $[i']$ y por ello, siendo $i' \leq w_{lb}$, no podríamos aumentar nuestra actual cota inferior w_{lb}.

Por ejemplo, dados $r = 2$, $k_1 = k_2 = 3$, $i = 6$ y $w_{lb} = 8 < w(3,3;2) = 9$, consideremos la coloración parcial libre (parcial porque no todos los elementos tienen asignado un color):

$$\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 1 & 2 & 2 & 1 & 2 & & \end{array} \tag{2.2}$$

Ya que i' puede ser 7 u 8 consideramos ambos casos: con $i' = 7$ no hay ninguna 3-PA de color 1 o 2 cuyo último elemento sea i'; en cambio, con $i' = 8$ hay al menos una 3-PA monocromática de cada color, siendo de color 1 la dada por las posiciones $\{2, 5, 8\}$ y de color 2 la dada por las posiciones $\{4, 6, 8\}$, garantizando que con la coloración actual no podríamos llegar más allá del 8. Por ello, la posición 8 es una cúspide.

Por último, decimos que una progresión aritmética $\{a, a+d, \ldots, a+(k-1)d\}$ *apunta* a m si $m = a + kd$.

Regresando al algoritmo que sirve para encontrar una cúspide, éste realizará una búsqueda lineal para cada uno de los elementos de la matriz L generada por el Algoritmo 1 de la sección anterior.

Como entrada se necesita el número de colores r, las longitudes máximas permitidas $\langle k_1, \ldots, k_r \rangle$, una cota inferior w_{lb}, la matriz L y una coloración libre χ de $[i]$, con $i < w_{lb}$. La idea clave es que si la longitud $L[j,h]$ es de tamaño $k_{\chi(j)} - 1$, entonces verificamos que la PA registrada en $L[j,h]$ apunte a alguna posición $i' \in \{i+1, \ldots, w_{lb}\}$ y, si apunta a tal posición, registramos dicha posición i' usando la matriz llamada *aimed_by_mono*, de dimensiones $r \times w_{lb}$ y definida por:

$$aimed\_by\_mono = \begin{cases} 1, \text{ si hay una } (k_t - 1)\text{-PA de color } t \text{ apuntando a la} \\ \quad \text{posición } i' \in \{i+1, \ldots, w_{lb}\} \\ 0, \text{ si no la hay.} \end{cases} \tag{2.3}$$

Si a la posición i' apunta al menos una $(k_t - 1)$-PA de cada color $t \in \{1, \ldots, r\}$, de acuerdo con la Definición 2.2.2, i' es cúspide. El algoritmo devolverá dicha cúspide i'.

El algoritmo que devolverá la primera cúspide encontrada según vaya realizando el recorrido, se muestra en Algoritmo 2.

Algoritmo 2

Input: El número de colores r, las longitudes máximas permitidas $\langle k_1, \ldots, k_r \rangle$, una cota inferior w_{lb}, una coloración χ de $[i]$ libre, con $i < w_{lb}$ y la matriz L, donde $L[j, h]$ es la longitud de la progresión aritmética más larga con elementos iniciales j, h.

Output: Un entero *culprit* que es la posición de la primera cúspide encontrada. En caso de no encontrar ninguna, devuelve 0.

```
1:  for j = 1 to j = i do
2:      for t = 1 to t = r do
3:          aimed_by_mono[t][p] := 0
4:  culprit := 0
5:  for j = n down to j = 1 do
6:      for d = 1 to d = j − 1 do
7:          if L[j − d, j] < k_{χ(j−d)} − 1 then
8:              continue
9:          else                                   // L[j − d, j] == k_{χ(j−d)−1}
10:             aimed := (j − d) + L[j − d, j] · d
11:             if (i < aimed) & (aimed ≤ w_lb) then
12:                 if aimed_by_mono[χ(j − d)][aimed] == 0 then
13:                     aimed_by_mono[χ(j − d)][aimed] := 1
14:                     colors_used[aimed] := colors_used[aimed] + 1
15:                     if colors_used[aimed] == r then
16:                         culprit := aimed
17:                         return culprit
18:                     else
19:                         continue
20:                 else                           // Esta posición ya tiene registrado este color
21:                     continue
22: return 0
```

2.2.3. El algoritmo de la cúspide

Imaginemos que se quiere obtener el número de Van der Waerden $w = w(k_1, \ldots, k_r; r)$. Una forma de hacerlo es revisar cada una de las r^w_{lwb} coloraciones posibles y partir des-

de allí, donde w_{lwb} representa la cota inferior conocida para dicho número. Aunque ya quedó claro en la Sección 1.4.1 que eso nos traería consecuencias desastrosas el día que los alienígenas decidan divertirse cruelmente. Una segunda idea sería comenzar a colorear cada uno de los elementos y detenernos cuando encontremos una k_t-PA, cambiar el color de la posición y probar de nuevo. Si los colores se acaban, habría que regresar, elemento por elemento, hasta alguno que nos permita cambiar su color. Luego habrá que repetir el procedimiento.

Este último enfoque es denomidado de ramificación y poda (*Branch and Bound*), y es el procedimiento estándar a la hora de lidiar con problemas como el de encontrar coloraciones. Si bien este último enfoque nos permite deshacernos, desde temprano, de las coloraciones que contengan una progresión aritmética más larga de lo debido y es, por tanto, más eficiente que el de fuerza bruta, también es cierto que solo permite calcular números de Van der Waerden muy pequeños.

Dicho procedimiento evolucionó en lo que se ha dado por llamar el algoritmo de la cúspide. Dicho algoritmo fue introducido por primera vez en [9]. Este algoritmo es básicamente un algoritmo de ramificación y poda al que se agrega una restricción más.

El principio de dicho algoritmo es muy simple: comienza coloreando cada uno de los elementos y se detiene cuando se encuentra un progresión aritmética que se exceda, cambia el color del último elemento, el elemento i, y prueba de nuevo. El número máximo de elementos coloreados se guarda en una variable w_{lb} que funge como la cota inferior que luego se usará para buscar una cúspide. Una vez que se usaron todos los colores disponibles para probar en el elemento i y todavía no se tiene una coloración libre de $\{1, \ldots, i\}$, habrá que regresar hacia atrás, disminuyendo i de a uno, hasta encontrar un elemento que se pueda colorear diferente, como se hace en un algoritmo de vuelta atrás común. La novedad del método de las cúspides es que ya no basta con preguntar si la coloración χ de $\{1, \ldots, i\}$ es libre, sino que también se pregunta si esa coloración, la de $\{1, \ldots, i\}$, apunta hacia alguna cúspide i' en el intervalo $i < i' \leq w_{lb}$. Si apunta a alguna cúspide significa que cualquier coloración libre que hagamos, usando el color actual de i, tendrá una longitud menor que w_{lb}, por lo que no tendrá sentido

continuar con el color que i tiene en este momento.

El algoritmo mostrado en Algoritmo 3, hace uso de los algoritmos revisados en las dos secciones anteriores. El algoritmo toma como entradas el número de colores r, el vector de longitudes máximas $\langle k_1, \ldots, k_r \rangle$, el número i, una cota inferior w_{lb} y una coloración libre χ de $\{1, \ldots, i\}$. El algoritmo se invocará usando inicialmente $i = w_{lb} = 0$ y la coloración libre inicial $\chi = \{\}$. La salida del algoritmo será el número w_{lb} tal que toda r-coloración de $\{1, \ldots, w_{lb} + 1\}$ contiene al menos una k_t-PA de algún color $t \in \{1, \ldots, r\}$. Es decir que para cuando el algoritmo termine tendremos $w_{lb}(k_1, \ldots, k_r; r) + 1 = w(k_1, \ldots, k_r; r)$, siendo w el número de Van der Waerden.

Algoritmo 3 Culprit_algorithm

Input: $(i, \chi, w_{lb}, r, \langle k_1, \ldots, k_r \rangle)$ El número de colores r, el vector de longitudes máximas $\langle k_1, \ldots, k_r \rangle$, el número i, una cota inferior w_{lb} y una coloración χ de $\{1, \ldots, i\}$ libre.
Output: El número mínimo w_{lb} tal que toda r-coloración de $\{1, \ldots, w_{lb} + 1\}$ contiene al menos una k_t-PA de algún color $t \in \{1, \ldots, r\}$

1: **for** $t = 1$ to $t = r$ **do**
2: $\quad \chi(i + 1) := t$
3: $\quad$ **if** No hay k_t-PA monocromática de color t en $\{1, \ldots, i + 1\}$ para algún $t \in \{1, \ldots, r\}$ y no hay cúspide i', con $i + 1 < i' \leq w_{lb}$ **then**
4: $\quad\quad w_{lb} := max\{w_{lb}, i + 1\}$
5: $\quad\quad w_{lb} :=$ Culprit_algorithm$(i + 1, \chi, w_{lb}, r, \langle k_1, \ldots, k_r \rangle)$
return w_{lb}

2.2.4. El algoritmo de los comodines

El último algoritmo para calcular los números de Van der Waerden que revisaremos es el algoritmo de los comodines.

Recordando la Definición 1.3.8, donde se introduce el concepto de varicoloración, podemos proceder a definir un par de conceptos que ayudarán a simplificar la estructura del algoritmo. Dichos conceptos son los de color prohibido y color inocuo.

Definición 2.2.3. *Dada una varicoloración $\lambda : [n] \to S([r])\backslash\emptyset$ y un color $t \in [r]$, decimos que dicho color t es:*

- *Un* color prohibido *para la posición i, si al asignar $\lambda(i) := \{t\}$ se genera una*

progresión aritmética monocromática de color t y tamaño prohibido k_t.

- *Un* color inocuo *para la posición i, si al hacer $\lambda(i) := \{t\}$ no se genera una k_t-PA monocromática de color t.*

Hay que aclarar que se considera progresión aritmética monocromática de color t, si todos los elementos de dicha progresión son están varicoloreados con $\{t\}$.

Recordando la varicoloración 1.3.2 de dos colores presentada en la Sección 1.3.2:

$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \{1\} & \{1,2\} & \{2\} & \{1\} & \{1,2\} & \{2\} & \{2\} \end{array} \tag{2.4}$$

Usando la varicoloración del Ejemplo 2.4, con $k_1 = k_2 = 3$, podemos observar que, si hacemos $\lambda(5) = \{2\}$, las posiciones 5, 6 y 7 formarán una 3-PA de color 2. Por ello decimos que el color 2 está prohibido para la posición 5. En cambio, si hacemos $\lambda(5) = \{1\}$ no se generará ninguna 3-PA monocromática de color 1. Decimos pues que el color 1 es inocuo en la posición 5.

Antes de pasar al algoritmo estableceremos una nomenclatura para recolorear una posición i con un varicolor T: Dada una varicoloración de $[n]$ con r colores, una posición i de $[n]$ y un varicolor $T \in S(\{1,\ldots,r\})$, decimos que cambiamos el varicolor $\lambda(i)$ por T cuando escribimos $\lambda_{i\to T}$.

Ahora sí, es tiempo de pasar al algoritmo de los comodines.

El algoritmo de los comodines se basa en que el conjunto que contiene todas las coloraciones libres está generalizado en un conjunto de igual o menor tamaño que contiene varicoloraciones libres. Decimos que una *varicoloración es libre* si todas las especificaciones[1] de la misma son libres, es decir si dicha varicoloración no contiene colores prohibidos. Este algoritmo funciona mejor que el de la cúspide porque usando varicoloraciones en vez de coloraciones se reduce el número de iteraciones necesarias para encontrar el número de Van der Waerden.

[1]Una especificación, λ', de una varicoloración λ de $[n]$, se caracteriza porque $\lambda'(i) \subseteq \lambda(i)$ para $i \in [n]$.

Nuestro procedimiento toma como entrada al conjunto de todas las varicoloraciones libres de $[n]$, denotado por $\mathbb{L}_n$. Luego agrega en cada varicoloración libre la posición $n+1$ coloreada con el conjunto $[r]$ y las agrega todas al conjunto de coloraciones pendientes de revisión S. Con cada una de las nuevas varicoloraciones de tamaño $n+1$, primero se eliminan todos los colores prohibidos de la varicoloración en cuestión y después se verifica en cuál de los siguientes tres casos se encuentra: la varicoloración es libre y la guardamos en el conjunto $\mathbb{L}_{n+1}$ de todas las varicoloraciones libres de $[n+1]$; la varicoloración no se especifica en una coloración libre y simplemente la descartamos; o existe una posición i con $|\lambda(i)| \geq 2$ tal que no todos los colores son inocuos en dicha posición. En el último caso ramificaremos nuestra varicoloración cambiando el elemento $\lambda(i)$ y coloreándolo con cada uno de los colores no inocuos, y con el conjunto que contiene todos los colores inocuos en la posición i; estas nuevas varicoloraciones, una nueva para cada color no inocuo y una nueva para el conjunto con los colores inocuos, se agregarán al conjunto de varicoloraciones pendientes de revisión S. Cuando la lista de varicoloraciones pendientes de revisión esté vacía, tendremos todas las coloraciones libres guardadas en el conjunto $\mathbb{L}_{n+1}$. El algoritmo repetirá los pasos anteriores asignando $n = n+1$ y se detendrá cuando el conjunto $\mathbb{L}_{n+1}$ al final de las iteraciones se encuentre vacío, pues esto significa que no encontró ninguna varicoloración libre de tamaño $n+1$, siendo éste número, $n+1$, el buscado número de Van der Waerden.

El Algoritmo 4 solo considera los pasos usados para tomar como entrada $\mathbb{L}_n$ y dar como salida $\mathbb{L}_{n+1}$:

Algoritmo 4 Wildcards_algorithm

Input: El conjunto de colores $C = \{1, \ldots, r\}$ y un conjunto de varicoloraciones de longitud n, $\mathbb{L}_n$.

Output: Un conjunto $\mathbb{L}_{n+1}$ de todas las varicoloraciones libres de longitud $n+1$ que sean extensiones de las varicoloraciones en $\mathbb{L}_n$.

1: $\mathbb{L}_{n+1} := \emptyset$
2: $S := \emptyset$
3: **for** all $\lambda \in \mathbb{L}_n$ **do**
4: $\quad S := S \cup \{\lambda_{n+1 \to C}\}$
5: **while** $S \neq \{\}$ **do**
6: $\quad$ pick $\lambda \in S$

7: **while** Exista una posición i con color t prohibido **do**
8: $\lambda := \lambda_{i \to T \setminus \{t\}}$
9: **if** λ es libre **then**
10: $\mathbb{L}_{n+1} := \mathbb{L}_{n+1} \cup \{\lambda\}$
11: **else if** Existe una posición i coloreada con $T = \lambda(i)$ con $|T| \geq 2$ y no todos los colores $t \in T$ son inocuos en la posición i **then**
12: $T' := \{t \in T \mid t \text{ es inocuo en la posición } i\}$
13: $S := S \cup \{\lambda_{i \to T'}\}$
14: **for** all $t \in T \setminus T'$ **do**
15: $S := S \cup \{\lambda_{i \to \{t\}}\}$
return $\mathbb{L}_{n+1}$

El algoritmo de los comodines fue introducido y optimizado por Schweitzer en [4]. Este algoritmo resalta por su sencillez y nuevo enfoque, siendo la primordial inspiración para el algoritmo con el que contribuye el presente trabajo de tesis, algoritmo que estudiaremos en la sección siguiente.

Capítulo 3

Números de anti-Van der Waerden

El tema de este penúltimo capítulo es el de los números de anti-Van der Waerden. Se estudiará su definición y se presentarán los números conocidos para luego mostrar el estado del arte en los algoritmos de búsqueda de dichos números; en las últimas dos secciones se introducirá el algoritmo propuesto y se enumerarán los nuevos números encontrados.

3.1. Caso heterocromático

Comenzaremos este capítulo definiendo a qué nos referimos cuando hablamos del caso heterocromático de los números de Van der Waerden, caso particular que se conoce como los números de anti-Van der Waerden.

Haremos uso de las definiciones dadas en 1.3.2. En este capítulo, al igual que en el anterior, la coloración será una función cuyo dominio es un conjunto $[n]$ y cuyo rango es $[r]$, se tomarán como subconjuntos las progresiones aritméticas contenidas en el dominio.

Podemos definir un número de anti-Van der Waerden a partir de lo siguiente:

Definición 3.1.1. *Dados dos números enteros positivos cualesquiera, digamos n y k, existe un número entero mínimo $aw = aw([n], k)$ denominado* número de anti-Van der Waerden, *tal que cualquier aw-coloración exacta*[1] *de $[n]$ contiene al menos una*

[1]Decimos que una coloración es exacta si se trata de una función sobreyectiva. En el caso que estamos estudiando, si la coloración de $[n]$ usa todos los colores.

progresión aritmética heterocromática con k términos.

Podemos observar que para todo n, $aw([n], 2) = 2$, debido a que para obtener una 2-progresión aritmética heterocromática basta con usar dos colores distintos, pues cualquier pareja de elementos de $[n]$ con dos colores distintos, digamos $\{s_i, s_j\}$ con $s_i < s_j$, será una 2-progresión aritmética heterocromática con diferencia $s_j - s_i$.

3.2. Números conocidos

Siguiendo el estilo del capítulo anterior, procederemos a presentar algunos de los números conocidos en la Tabla 3.1 [10].

k \n	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
3	3														
4	4														
5	4	5													
6	4	6													
7	4	6	7												
8	5	6	8												
9	4	7	8	9											
10	5	8	9	10											
11	5	8	9	10	11										
12	5	8	10	11	12										
13	5	8	11	11	12	13									
14	5	8	11	12	13	14									
15	5	9	11	13	14	14	15								
16	5	9	12	13	15	15	16								
17	5	9	13	13	15	16	16	17							
18	5	10	14	14	16	17	17	18							
19	5	10	14	15	17	17	18	18	19						
20	5	10	14	16	17	18	19	19	20						
21	5	11	14	16	17	19	20	20	20	21					
22	6	12	14	17	18	20	21	21	21	22					
23	6	12	14	17	19	20	21	22	22	22	23				
24	6	12	15	18	20	20	22	23	23	23	24				
25	6	12	15	19	21	21	23	23	24	24	24	25			

Tabla 3.1: Números de anti-Van der Waerden conocidos.

La lista completa puede consultarse en [10], donde se aclara que se han calculado

los valores $aw([n], 3)$ para $n \leq 58$ y $aw([n], k)$ para $k < n \leq 25$. Sin embargo, en [11] se muestra una prueba que determina el valor exacto de $aw([n], 3)$ para todo n, a saber:

$$aw([n], 3) = \begin{cases} m+2, & \text{si } n = 3^m, \\ m+3, & \text{en cualquier otro caso.} \end{cases} \tag{3.1}$$

En la Sección 3.3 se explicará el procedimiento usado para el cálculo de los números dados en [10].

3.3. Un algoritmo de búsqueda de progresiones aritméticas heterocromáticas

En esta sección hablaremos sobre el procedimiento usado para encontrar los números reportados en [10]. El procedimiento trata de un algoritmo de ramificación y poda que usa, entre otras, las siguientes restricciones:

1. **Restricción 1:** La primera vez que aparece el color p sucederá antes que la primera vez que aparece el color q si y solo si $p < q$, para $q, p \in [r]$.

2. **Restricción 2:** La coloración debe ser una función sobreyectiva. Es decir, la coloración debe usar todos los colores.

Estas restricciones se mencionan por separado debido a que también serán útiles para el algoritmo mostrado en la Sección 3.4.1. A continuación ahondaremos en la explicación del algoritmo de ramificación y poda.

El algoritmo de ramificación y poda es un procedimiento estándar cuando se abordan problemas como este. Consiste en ir construyendo la solución buscada de manera que cada elemento agregado a la construcción limita las opciones de los otros elementos a través de restricciones impuestas a priori. En el caso de una coloración, $c : [n] \to [r]$, un acercamiento consistiría en asignar los colores en algún orden definido. Por ejemplo ir desde la primera hasta la última posicón, donde cada nuevo elemento coloreado

limitará las posibilidades de los elementos siguientes. La eficiencia de dichos algoritmos reside en escoger adecuadamente tales restricciones. Stolee presenta, en [12], el algoritmo explicado a continuación.

Lo primero consistirá en definir e inicializar los *dominios* $D(i) = [r]$ para todo $i \in [n]$. El dominio de la i–ésima posición, $D(i)$, contiene todos los colores que no generan una k-PA heterocromática. Así, cada vez que se asigna un color a i, buscaremos alguna k-PA, que llamaremos P, tal que P contiene $k-1$ elementos de color distinto. Ahora pensemos en un elemento $j \in P$, donde j es el único elemento sin colorear en P. Entonces podemos restringir el dominio de j asignando a $D(j)$ el valor de $D(j) \cap c(P \backslash j)$, donde $c(P \backslash j)$ representa los colores ya usados en el resto de la progresión aritmética.

Con el procedimiento anterior de definir los dominios, podemos establecer otra restricción:

3. **Restricción 3:** No puede ocurrir que una posición sin colorear j tenga $|D(j)| = 0$ pues significa que dicha posición no se puede colorear.

La última restricción mostrada en [12] requiere, para computar $aw([n], k)$, el cómputo previo de los números $aw([m], k)$ para $m < n$, pues establece la siguiente dependencia:

4. **Restricción 4:** Si $[i]$ se colorea usando solamente l colores, entonces $\{i, \ldots, n\}$ debe poder ser coloreado usando al menos $r - l + 1$ colores.

Lo anterior es equivalente a decir que $aw([n-i], k)$ debe ser menor o igual que $r - l + 1$, cosa que resulta razonable porque, de no ser así, la segunda parte de la coloración, $\{i, \ldots, n\}$, contendría una k-PA heterocromática sin importar qué colores se asignen a cada miembro de $\{i, \ldots, n\}$.

Con las cuatro restricciones anteriores, se va coloreando cada posición i con alguno de los colores disponibles en $D(i)$ y, si ninguna restricción se quebrantó, se asigna $i := i+1$ para continuar coloreando; si alguna restricción es violada, entonces se colorea i con el siguiente color disponible de $D(i)$, hasta que no quedan colores disponibles para i y se procede a asignar $i := i - 1$.

Cuando i alcanza el valor de n y ninguna restricción se traspasa al asignarle color, significa que se ha encontrado una coloración de $[n]$ con r colores que queda libre de k-PA heterocromáticas, coloración que se guarda y se continúa con el procedimiento, cabiando el color de i. El proceso termina cuando todos los colores de $D(1)$ han sido usados para la posición 1. Dicho acercamiento se denomina, de manera general, arc Consistency para Constraint Satisfaction Problems.

3.4. Algoritmo de los comodines

Ahora es turno de hablar del algoritmo con que se espera contribuir a la investigación sobre el tema. Este algoritmo fue ideado, partiendo de los conceptos introducidos por Schweitzer [4] y, por ello, ha sido bautizado con el nombre correspondiente, aunque esta vez se trate del caso heterocromático. Antes de presentar el algoritmo, será necesario replantear las definiciones de color prohibido y color no inocuo para adaptarlas al caso heterocromático.

Definición 3.4.1. *Dada una varicoloración $\lambda : [n] \to S([r])\backslash\emptyset$ y un color $t \in [r]$:*

- *Decimos que el color t es un* color prohibido *para la posición i, si al asignar $\lambda(i) := \{t\}$ se genera una k-PA heterocromática.*
- *Decimos que el color t es un* color inocuo *en la posición i, si al hacer $\lambda(i) := \{t\}$ no se genera ninguna k_t-PA heterocromática.*

Se dice que una k_t-PA es heterocromática si tanto $\lambda(i) \neq \lambda(j)$ como $|\lambda(i)| = |\lambda(j)| = 1$ para cualesquiera $i, j \in k_t$-PA.

Con estas dos adaptaciones menores podemos continuar con el algoritmo. Igual que la vez pasada, las dos ideas más poderosas son que una varicoloración puede especificarse en muchas coloraciones, reduciendo así el número de iteraciones, y que los colores inocuos y los no inocuos pueden ser separados para facilitar el proceso de eliminación de los colores prohibidos, reduciendo de esta manera una gran cantidad de ramificaciones no válidas.

3.4.1. El algoritmo de los comodines

Este algoritmo toma como entrada un conjunto, S, de varicoloraciones con n elementos que cumplan con las restricciones 1 y 2, el número de colores r y el tamaño de la progresión aritmética k; la salida el algoritmo será el conjunto de varicoloraciones de $[n]$ libres con r colores, al que se denotará $\mathbb{L}$. Una varicoloración se considera *libre* si respeta las restricciones 1 y 2, y si ninguna de sus especificaciones contiene una k-PA heterocromática.

Para conseguirlo se asignará $S := \mathbb{L}$ y cada varicoloración λ contenida en S deberá pasar a través de tres filtros:

El primero se encargará de eliminar todos los colores prohibidos presentes en λ, lo que conseguirá revisando las progresiones aritméticas de $[n]$ y eliminando los colores prohibidos contenidos en ellas; continuará hasta que las haya revisado todas o haya encontrado alguna k-PA heterocromática donde todos los elementos tienen tamaño uno. En el primer caso pasa al siguiente filtro, en el segundo la varicoloración es descartada pues tiene un color prohibido que no se puede eliminar.

El segundo filtro verificará que se respeten las restricciones. Si en λ existe alguna posición i que no respete el orden de aparición de colores por un color $t \in \lambda(i)$, se recoloreará $\lambda_{i\to\lambda(i)\setminus t}$ y se agregará la varicoloración al conjunto de varicoloraciones pendientes de revisión, S. Si antes de recolorearlo $|\lambda(i)| = 1$ o si la varicoloración no usa los r colores, entonces esa varicoloración se descarta. En caso de respetar ambas restricciones se procede al tercer filtro.

El tercer filtro consiste en revisar que la varicoloración esté libre de especificaciones que contengan k-PA heterocromáticas De ser así, se trata de una varicoloración libre y se guarda en $\mathbb{L}$. Al igual que el primer filtro, esto se consigue revisando todas las k-PA en busca de una que se especifique en una k-PA heterocromática. Cuando se encuentra se busca la posición más alta, i, con $|\lambda(i)| > 1$; de esta posición se enumeran los colores inocuos, T', para luego colorear $\lambda_{i\to\setminus T'}$ y se agrega λ a S para volver a ser revisado. Para cada color no inocuo t de $\lambda(i) = T'$, se agrega otra varicoloración, λ', a S con:

$$\lambda'(j) = \begin{cases} \lambda(j), & si \quad j \neq i, \\ \{t\}, & si \quad j = i. \end{cases}$$

El algoritmo esta enunciado en Algoritmo 5.

Algoritmo 5 Wildcards_algorithm_rainbow

Input: Un conjunto S de varicoloraciones de n elementos, el número de colores r y el tamaño máximo permitido k de una k-PA heterocromática.

Output: Un conjunto $\mathbb{L}$ de varicoloraciones libres especificadas de las varicoloraciones contenidas en S.

1: **while** $S \neq \{\}$ **do**
2: pick $\lambda \in S$
3: **while** Exista una posición i con color t prohibido en λ **do**
4: $\lambda := \lambda_{i \to T \setminus \{t\}}$
5: **if** λ tiene una posición i con algún color t que no respete la restricción 1 y $|\lambda(i)| > 1$ **then**
6: $S := \lambda_{i \to \lambda(i) \setminus \{t\}}$
7: **else if** λ tiene una posición i con algún color t que no respete la restricción 1 y $|\lambda(i)| = 1$ o no respeta la restricción 2 **then**
8: (continue)
9: **if** λ es libre **then**
10: $\mathbb{L} := \mathbb{L} \cup \{\lambda\}$
11: **else if** Existe una posición i coloreada con $T = \lambda(i)$ con $|T| \geq 2$ y no todos los colores $t \in T$ son inocuos en la posición i **then**
12: $T' := \{t \in T \mid t$ es inocuo en la posición $i\}$
13: **if** $|T'| \neq \emptyset$ **then**
14: $S := S \cup \{\lambda_{i \to T'}\}$
15: **for** all $t \in T \setminus T'$ **do**
16: $S := S \cup \{\lambda_{i \to \{t\}}\}$
return $\mathbb{L}$

El procedimiento mostrado en el Algoritmo 5 se puede usar para encontrar el número de anti-Van der Waerden $aw([n], k)$ usando como entrada $r = k$, la varicoloración $\lambda(i) = [r]$ para $1 \leq i \leq n$ y k. Cada vez que obtengamos $\mathbb{L}$, mientras $\mathbb{L}$ no esté vacío, hacemos $r = r + 1$ y $\lambda(i) = [r]$ para $1 \leq i \leq n$. Cuando $\mathbb{L} = \emptyset$ luego de ejecutar el algoritmo, $aw([n], k) = r$.

3.4.2. Optimización

En esta sección se abordarán algunos detalles de diseño del algoritmo y de la implementación, detalles que fueron pensados para aumentar la eficiencia del algoritmo.

El primero de ellos toma ventaja de la Restricción 1 para hacer una optimización inicial. Debido al orden de aparición impuesto, se puede comenzar con una varicoloración como sigue: Dados k, r y n:

$$\lambda(i) := \{\{1\}, \{1,2\}, \ldots, \{1,2,\ldots,r-1\}, \{1,2,\ldots,r\} \ldots, \{1,2,\ldots,r\}\}.$$

Ahora hablaremos sobre el orden de revisión de todas las k-PA. Consiste en iterar a través de todas las k-PA's usando dos ciclos: uno que va de $Indice = k$ hasta n y otro que va desde $Diff = Diff\_Max$ hasta 1, con $Diff\_Max = \left\lfloor \frac{Indice - 1}{k-1} \right\rfloor$, donde $Indice$ es el último término de la progresión aritmética y $Diff$ es la diferencia de dicha k-PA. Iterar en este orden es importante porque nos permite definir un orden lexicográfico de derecha a izquierda, en el que el último término de cada progresión aritmética, y su respectiva diferencia, definen el orden de revisión. Por ejemplo, todas las 3-PA en $[n]$, con $n = 7$, son revisadas en el orden mostrado en la Tabla 3.2.

$Diff$			$Indice$
1	1	2	**3**
1	2	3	**4**
2	1	3	**5**
1	3	4	**5**
2	2	4	**6**
1	4	5	**6**
3	1	4	**7**
2	3	5	**7**
1	5	6	**7**

Tabla 3.2: Ejemplo de orden de revisión lexicográfico con base en el último término $Indice$ y diferencia $Diff$ para todas las 3-PA en $[n]$, con $n = 7$. A la izquierda de la tabla se encuentra la diferencia y a la derecha se encuentra la progresión aritmética revisada.

El orden propuesto es útil debido a que permite definir todas las ramificaciones con colores inocuos en la parte más izquierda de $[n]$. Haciendo lo anterior se consigue

que se eliminen de manera eficiente los colores prohibidos pues ramifica las posiciones de izquierda a derecha. Además de eso, permite ir creando puntos de resguardo: Una vez que ha verificado que la k-PA, con último término $Indice$ y diferencia $Diff$, no se especifica en una k-PA heterocromática, no es necesario volver a revisarla durante el resto de las iteraciones porque ya sabemos que es libre, disminuyendo el número de iteraciones necesarias para revisar todas las k-PA cada vez que se pasa la varicoloración por el tercer filtro. Dicho punto de resguardo sirve tanto para la búsqueda de las especificaciones heterocromáticas como para la de los colores prohibidos.

Otro detalle menester de mención es el hecho de que el algoritmo usado para la revisión de las progresiones aritméticas revisa, para cada posición de dicha progresión, únicamente los colores que no hayan sido usados ya por otros miembros de la misma progresión.

Aprovechando la definición de color prohibido, uno puede simplificar la búsqueda limitando a 1 el número máximo de varicolores con más de un color para cualquier k-AP. Haciendo lo anterior se disminuye la complejidad del peor de los casos de $O(r^k)$ a $O(k^2)$ para cada progresión aritmética.

Por último, se usaron enteros para representar los varicolores pues las únicas operaciones de conjuntos necesarias son asignación y diferencia; para los miembros solo se necesitaban acceso, eliminación y adición. Todas las operaciones antes mencionadas son fáciles de implementar a través de operaciones con bits.

3.4.3. Implementación

La implementación del Algoritmo 5 y de las optimizaciones mostradas en la Sección 3.4.2 se realizó usando el lenguaje de programación C++. El código se encuentra anexado en el Apéndice A.

El código se ejecutó en una computadora con sistema operativo Ubuntu 14.04.5 LTS, procesador Intel Atom N270 a 1.60GHz y 1 GB de RAM. Se usó el compilador g++ v4.8.4 con bandera de optimización O3.

A modo de comparación se tomaron como referencia los tiempos resgistrados en la

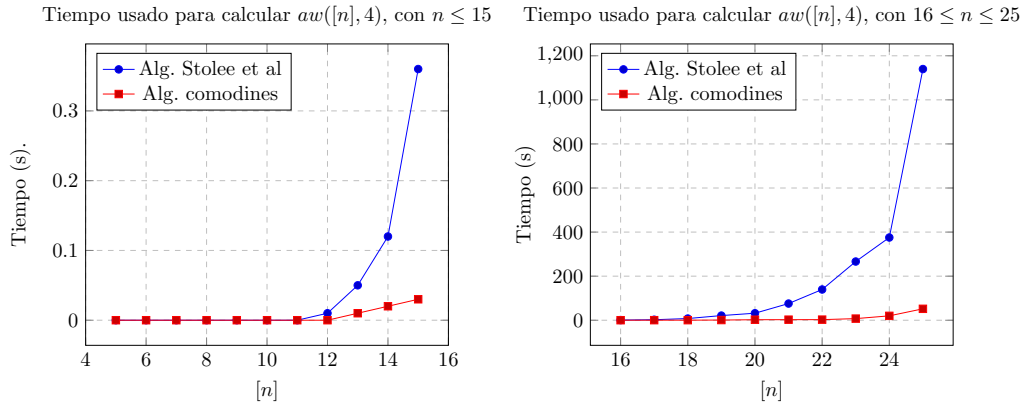

Figura 3.1: Gráficas de los tiempos regristrados con $k = 4$ para $n \leq 15$ y $16 \leq n \leq 25$, respectivamente izquierda y derecha. La línea azul son los tiempos reportados por Stolee y la línea roja los tiempos obtenidos usando el algoritmo de los comodines.

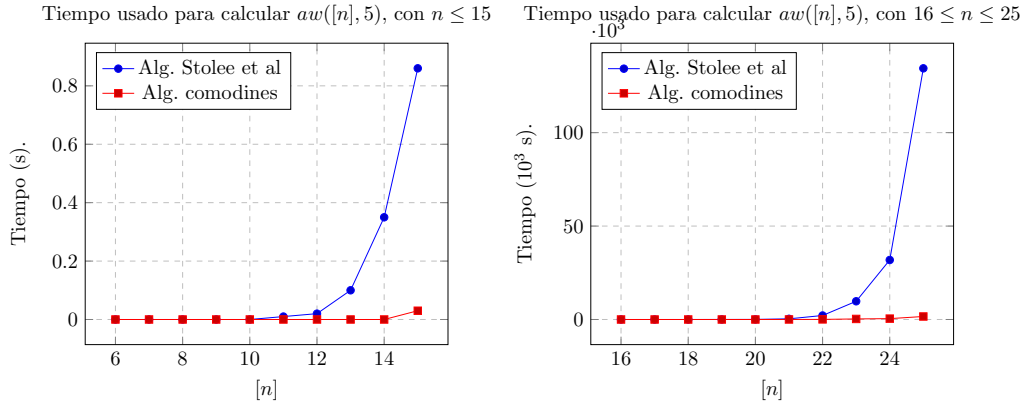

Figura 3.2: Gráficas de los tiempos regristrados con $k = 5$ para $n \leq 15$ y $16 \leq n \leq 25$, respectivamente izquierda y derecha. La línea azul son los tiempos reportados por Stolee y la línea roja los tiempos obtenidos usando el algoritmo de los comodines.

base de datos de Stolee [12], con $k = 4$, y se graficaron en las Figuras 3.1 y 3.2.

3.5. Nuevos números

Los nuevos números de anti-Van der Waerden que se muestran en la tabla 3.3, se encontraron usando el algoritmo de los comodines para el caso heterocromático, en menos de 12 horas. Cabe destacar que para el cálculo de dichos números se usó una computadora con un procesador AMD A10-7300 Turbo CORE, con 12 GB de RAM y con el sistema operativo Kali linux en una live USB.

$k\backslash n$	4	5
26	12[1]	16
27	12	16
28	12	17
29	12	
30	12	

Tabla 3.3: Nuevos números de anti-Van der Waerden calculados.

[1] $aw([26], 4) = 12$ fue calculado en [12].

Capítulo 4

Conclusiones

Como se planteó en la Introducción, los objetivos con los que se inició el trabajo fueron los siguientes:

1. Brindar una introducción accesible al estudio de los números de Van der Waerden y, su análogo heterocromático, los números anti-Van der Waerden.

2. Estudiar las diferencias entre los algoritmos existentes para el caso monocromático.

3. Explorar las posibilidades de adaptar los algoritmos monocromáticos al caso heterocromático.

4. Comparar los resultados que se pueden obtener con dichos algoritmos.

Al llegar a este punto, a la conclusión de este académico viaje, los objetivos con que inició quedaron satisfechos y rebasados pues se pudo hacer una contribución al estado del arte, aunque no fuese sino otro grano de arena para la playa de las matemáticas. Se puede apreciar que el algoritmo creado, según lo muestran las gráficas 3.1 y 3.2, tiene un mucho mejor tiempo de ejecución que los reportados en la literatura, lo que permitió calcular los siete nuevos números de Van der Waerden registrados en la tabla 3.1.

Si bien el proyecto satisfizo los objetivos iniciales, andando así el camino que se propuso desde el comienzo, también es cierto que hizo que nuevas rutas quedaron al

descubierto, nuevas posibilidades, trabajo a futuro en palabras llanas. Entre dichas posibilidades se encuentran: el refinamiento de la implementación del algoritmo, explorar la posibilidad de paralelización y adaptarlo para la búsqueda de los números anti-Van der Waerden en coloraciones balanceadas.

Así queda claro que ideas sobran, ganas sobran, pero como en todas las empresas del hombre, el tiempo apremia.

Gracias lector, por haber leído.

Apéndice A

Código

En este apéndice se compartirá el código usado para la implementación del algoritmo de la cúspide, el algoritmo de los comodines en el caso heterocromático. Una implementación del algoritmo de los comodines para el caso monocromático se realizó con fines de aprendizaje, pero la implementación realizada por Schweitzer se puede encontrar en http://people.mpi-inf.mpg.de/~pascal/software/.

A.1. Algoritmo de la cúspide

El Algoritmo 3 o algoritmo de la cúspide, fue implementado en el lenguaje de programación C. El código es el que sigue:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned size_of_uint = (unsigned) sizeof(unsigned);
unsigned steps=0;
void van_der_waerden(void);
unsigned culprit_algorithm_pascal(unsigned n_max, unsigned i, unsigned chi[
    n_max], unsigned wlb, unsigned r, unsigned k_s[r+1]);
unsigned is_chi_valid(unsigned n_max, unsigned i, unsigned wlb, unsigned r,
    unsigned chi[n_max], unsigned k_s[r+1]);
unsigned get_longest_AP(unsigned n_max, unsigned i, unsigned r, unsigned int
     chi[n_max], unsigned L[i+1][i+1], unsigned k_s[r+1]);
unsigned get_culprit(unsigned n_max, unsigned i, unsigned wlb, unsigned r,
    unsigned chi[n_max], unsigned L[i+1][i+1], unsigned k_s[r+1]);

void main(void){
        van_der_waerden();
        printf("\n%u\n",steps);
}

void van_der_waerden(void){
        unsigned r=2;
        unsigned n_max = 200, w=0;
        unsigned chi[n_max];
```

```
        unsigned j;

        for(j=0; j<n_max; j++){
                chi[j] = 999;
        }

        unsigned cases[][4]={{0,3,7}};
        unsigned number_of_cases =(unsigned) sizeof(cases)/sizeof(cases[0]);
        printf("New coloring:\n");
        for(j=0; j < number_of_cases; j++)
        {
                printf("w( %u, %u) = ",cases[j][1],cases[j][2]);
                w = culprit_algorithm_pascal( n_max, 0, chi, 0, r, cases[j])
                    ;
                printf("w = %u\n", w);

        }


}

unsigned culprit_algorithm_pascal(unsigned n_max, unsigned i, unsigned chi[
    n_max], unsigned wlb, unsigned r, unsigned k_s[r+1]){
        unsigned t;
        unsigned j;
        for(t=1; t <= r; t++){


                chi[i+1]=t;

                if( is_chi_valid( n_max, i+1, wlb, r, chi,  k_s) ){
                        if (wlb < i+1) { wlb = i+1; printf("\n %u",wlb); }
                        wlb = culprit_algorithm_pascal( n_max, i+1, chi, wlb
                            , r, k_s);
                }
        }
        return wlb;
}

unsigned is_chi_valid(unsigned n_max, unsigned i, unsigned wlb, unsigned r,
    unsigned chi[n_max], unsigned k_s[r+1])
{
        unsigned t, L[i+1][i+1];

        if( get_longest_AP(n_max, i, r, chi, L, k_s) ){ //If there's at
            least one monochromatic k-AP.
                return 0;
        }else{
                        //If longest[t] > k[t] wasn't found for all every
                            color then it's free. Now it's time to check if
                            there is any culprit.
                if( get_culprit(n_max, i, wlb, r, chi, L, k_s) ){
                        return 0;
                }else return 1;
        }
}

unsigned get_longest_AP(unsigned n_max, unsigned i, unsigned r, unsigned int
     chi[n_max], unsigned L[i+1][i+1], unsigned k_s[r+1]){
        unsigned int longest[r+1];
        unsigned int j, d, h, t; //j is used for initialization and as
            established in the algorithm. d is used as established in the
            algorithm.
        unsigned int j_menos_d=0;//used for index calculations.
        unsigned int j_mas_d=0;//used for index calculations.
        unsigned int current_index=0;//index of L[j-d,j]
        unsigned int chi_j_menos_d=0;//color of chi[j-d]
```

```
        for(j = 1; j <= r; j++){ //initialize every usable element in
            longest to 0
                longest[j] = 0;
        }
        for(j = 1; j <= i; j++){ //initialize every usable element in L to 0
                for (h = 1; h <= i; h++){
//                      printf("j=%u, h=%u\n",j,h);
                        L[j][h] = 0;
                }
        }

        longest[ chi[i] ] = 1;

        for(j = i; j != 1; j-- ){//Check every pair of j-d, j E {1, ..., n}
                for (d = 1; d < j; d++){
                        steps++;
                        j_menos_d = j - d;
                        chi_j_menos_d = chi[j_menos_d]; //color of chi[j-d]
                        if(chi_j_menos_d != chi[j] ){//if chi[j-d] != chi[j]
                                L[j_menos_d][j] = 1;
                        }
                        else if( j+d <= i){ //if there's a third member of
                            the AP
                                L[j_menos_d][j] = L[j][j+d] + 1; // L[j-d,j]
                                    = L[j,j+d] + 1
                        }
                        else{ //If there's no third member but chi[j-d] ==
                            chi[j]
                                L[j_menos_d][j] = 2;
                        }
                        if( L[j_menos_d][j] > longest[chi_j_menos_d] ){//If
                            the current AP is longer than the longest saved
                            for that color then save it.
                                longest[chi_j_menos_d] = L[j_menos_d][j];
                                if( longest[chi_j_menos_d] == k_s[
                                    chi_j_menos_d] ){
                                        return longest[chi_j_menos_d];
                                }else continue;
                        }else continue;
                }
        }
        return 0;
}

unsigned get_culprit(unsigned n_max, unsigned i, unsigned wlb, unsigned r,
    unsigned chi[n_max], unsigned L[i+1][i+1], unsigned k_s[r+1]){
        if( i >= wlb ) return 0;//If wlb isn't greater that i.
        unsigned j,d,h;
        unsigned aimed_by_monochromatic[r+1][wlb+1], colors_used[wlb+1];
        unsigned j_menos_d = 0, chi_j_menos_d=0, k_menos_1=0;
        unsigned aimed = 0;

        for(j=1; j <= wlb; j++){
                colors_used[j]=0;
                for(d=1; d <= r; d++){
                        aimed_by_monochromatic[d][j]=0;
                }
        }


        for(j=i; j != 1; j--){//Nested loops to check every L[j-d,j] for j E
             {1,...,n}, d E {1,...,j-1};
                for(d=1; d < j; d++){
                        j_menos_d = j - d;
                        chi_j_menos_d = chi[j_menos_d]; //color of chi[j-d]
```

```
                        k_menos_1 = k_s[chi_j_menos_d] - 1;//k_menos_1 = k[
                            chi[j-d] ]-1

                        if( L[j_menos_d][j] < k_menos_1 ){//if the length of
                             the AP is not at least as long as k_menos_1
                            then nothing is done.
                                continue;
                        }
                        else{//if the length of the AP is equal to k_menos_1
                            .
                                aimed = j_menos_d + L[j_menos_d][j] * d;
                                if( (i < aimed) && (aimed <= wlb) ){//Check
                                    if it aims to {i+1, ..., wlb}.
                                        if( aimed_by_monochromatic[
                                            chi_j_menos_d][aimed] == 0 ){//
                                            If it's the first time (k-1)-AP
                                            of color chi_j_menos_d is found.
                                                aimed_by_monochromatic[
                                                    chi_j_menos_d][aimed]
                                                    =1;
                                                colors_used[aimed]++;
                                                if(colors_used[aimed] == r){
                                                        return aimed;
                                                }else continue;
                                        }else continue;//if the last member
                                            is bigger or equal just ignore
                                            it.
                                }else continue;// if the AP doesn't aim to n
                                     then nothing s done.
                        }
                }
        }
        return 0;
}
```

A.2. Algoritmo de los comodines

El algoritmo de los comodines, según fue explicado en la Sección 3.4, se implemento en el lenguaje de programación C++ como se muestra a continuación.

Archivo `main.cpp`:

```
#include <iostream>
#include <stack>
#include <algorithm>
#include <ctime>

#define MAX_COLORES 22
#define MAX_CONJCOLORES 4194304

using namespace std;

#include "Rainbow.hpp"

int main()
{
    //Precachear
    vector<int> Num1s(MAX_CONJCOLORES);
    vector<int> IndMax1(MAX_CONJCOLORES);
    f_Num1s(Num1s);
```

```
    f_IndMax1(IndMax1);

    int n=12;
    int k=4;
    int r=5;


        scanf("%i %i %i", &n, &k, &r);//Values to calculate

        while( n != 0 ){
            stack<VC,vector<VC> > L;
        vector<vector<int> > Libres;

            VC v(n,k,r);
                cout << "\n\nVamos a probar" << " n = " << n << ", k = " <<
                    k << ", r = " << r << endl;
            L.push(v);
            cout << "La coloracion libre es:" << endl;
            while(!L.empty())
        {
                RevisaUltimoDeLista(L,Libres,Num1s,IndMax1,k,r);
                if(Libres.size() >= 1){
                                r++;
                    break;
                        }
        }
            if(Libres.empty()){
            cout << "No hay coloraciones libres!!!" << endl;
                        cout << "aw(n=" << n << ",k=" << k << ")=" << r <<
                            endl;
                        scanf("%i %i %i", &n, &k, &r);//Look for values
                    }
        }
    return 0;
}
```

Archivo `Rainbow.cpp`:

```
#include <iostream>
#include <algorithm>
#include <ctime>
#include <stack>
#include <cmath>

using namespace std;

#include "Rainbow.hpp"

int f_Num1s(int i)
{
    if(i == 0)
        return 0;
    if(i%2 == 1)
        return 1+f_Num1s(i>>1);
    return f_Num1s(i>>1);
}

void f_Num1s(vector<int> &Num1s)
{
    Num1s[0]=0;
    for(unsigned int i=1; i<Num1s.size(); ++i)
    {
        if(i%2 == 1)
        {
            Num1s[i] = 1+Num1s[i>>1];
        }
```

```
        else
        {
            Num1s[i] = Num1s[i>>1];
        }
    }
}

int f_IndMax1(int i)
{
    int maximo=-1;
    while(i>0)
    {
        i>>=1;
        ++maximo;
    }
    return 1<<maximo;
}

void f_IndMax1(vector<int> &IndMax1)
{
    IndMax1[0] = 0;
    IndMax1[1] = 1;
    for(unsigned int i=2; i<IndMax1.size(); ++i)
    {
        IndMax1[i] = (IndMax1[i>>1]<<1);
    }
}

VC::VC(int n, int k, int r) : vari(n)
{
    chkdif = 1;
    chkfin = k-1;
    vari[0] = 1;
    for(int i=1; i<r; ++i)
    {
        vari[i] = (1<<(i+1))-1;
    }
    for(int i=r; i<n; ++i)
    {
        vari[i] = vari[r-1];
    }
}

void VC::Imprimir()
{
    for(auto i : vari)
        cout << i << " ";
    cout << endl;
}

void ImprimeColores(int vc)
{
    while(vc != 0)
    {
        int m1 = __builtin_ctz(vc);
        cout << m1;
        vc -= (1<<m1);
    }
    cout << " ";
}

bool nextPA(int &fin, int &dif, int n, int k)
{
    if(dif > 1)
    {
        --dif;
        return true;
```

```
    }
    else
    {
        if(fin == n-1)
        {
            return false;
        }
        ++fin;
        dif = fin/(k-1);
        return true;
    }
}

int OrdenLexi(vector<int> &vari, const vector<int> &Num1s, const vector<int>
     &IndMax1, int r)
{
    int ret = 2;//No se ha modificado nada
    int n = vari.size();
    int max1 = 1; //Asumimos que vari[0]==1
    for(int i = 1; i < n; ++i)
    {
        if((max1<<1) < (IndMax1[vari[i]]))
        {
            vari[i] &= (4*max1-1);//Borra todos los colores a dist>1 de
                IndMax1[var[i]]
            if(vari[i] == 0)
                return 0;
            ret = 1;//Ya modifiqu
        }
        //Actualizo max1
        if(((max1<<1) & vari[i]) != 0)
        {
            max1 <<= 1;
        }
        //Si ya us todos los colores
        if(max1 == (1<<(r-1)))
        {
            return ret;
        }
    }
    //Si llegue al final sin usar todos los colores
    return 0;
}

int QuitarProhibidosPA(vector<int> &vari, const vector<int> &Num1s, const
    vector<int> &IndMax1, int k, int &fin, int &dif)
{
    int CuantosVaris = 0;
    int UnicoVari = 0;
    int MultiOr = 0; //Va a ser el OR de los de tamao 1
    for(int i=0,elem=fin; i<k; ++i, elem -= dif)
    {
        if(Num1s[vari[elem]] > 1)
        {
            ++CuantosVaris;
            if(CuantosVaris > 1)//Si no voy a hacer nada
                return 2;
            UnicoVari = elem;
        }
        else
        {
            if((MultiOr & vari[elem]) != 0)//Si hay un color repetido, no
                hago nada
                return 2;
            MultiOr |= vari[elem];
        }
```

```
    }
    if(CuantosVaris == 0)
    {
        return 0;//Todos los colores son diferentes
    }
    else//Hay exact 1 Vari y es UnicoVari
    {
        int temp = vari[UnicoVari] & MultiOr;
        if(vari[UnicoVari] == temp)//Si no quito a nadie
            return 2;
        vari[UnicoVari] = temp;//Quita los prohibidos
        if(vari[UnicoVari] == 0)
            return 0;
        //Si llegu aqu es porque hubo un cambio, ajusto fin y dif
        if(UnicoVari < k)
        {
            dif = 2;
            fin = k-1;
        }else
        {
            fin = UnicoVari;
            dif = (fin-1)/(k-1)+1;
        }
        return 1;
    }
}

bool QuitaOrdena(VC &v, const vector<int> &Num1s, const vector<int> &IndMax1
    , int k, int r)
{
    if(OrdenLexi(v.vari,Num1s,IndMax1,r) == 0)
    {
        return false;
    }
    while(true){
        int fin = v.chkfin, dif = v.chkdif;
        bool QuiteUnProhibido = false;
        do{
            int h = QuitarProhibidosPA(v.vari,Num1s,IndMax1,k,fin,dif);
            if(h == 0)
                return false;
            if(h == 1)
                QuiteUnProhibido = true;
        }while(nextPA(fin,dif,v.vari.size(),k));//Cuando acaba resetea fin y
             dif
        if(!QuiteUnProhibido)
            return true;//Si no quitamos colores, no tiene caso volver a
                ordenar lexi
        int ol = OrdenLexi(v.vari,Num1s,IndMax1,r);
        if(ol == 0)
            return false;
        if(ol == 2)
            return true;//Si no cambio, no tiene caso quitar prohibidos
    }
    return true;
}

bool HayEspHetero(const VC &v, int CAs, int fin, int dif, int k, const
    vector<int> &Num1s, const vector<int> &IndMax1)
{
    if(k == 1)
    {
        if((v.vari[fin] & (~CAs)) != 0)//Si el el ltimo menos CAs no es vaco
            , hay especificacin htero
            return true;
        else
            return false;
```

```
    }
    //Si k>1
    int CPosib = v.vari[fin-(k-1)*dif] & (~CAs);
    if(CPosib == 0)//Si no hay colores posibles para hacer htero
        return false;
    while(CPosib != 0)
    {
        int SigC = IndMax1[CPosib];//El ltimo color posible
        int SigCAs = CAs | SigC;//Le aumento este color, s que s le estoy
            aumentando algo
        if(HayEspHetero(v,SigCAs,fin,dif,k-1,Num1s,IndMax1))//Si encontr
            htero
            return true;
        CPosib -= SigC;//Se lo quito para siguiente caso
    }
    return false;//Si nunca lo encontr
}

void RevisaUltimoDeLista(stack<VC,vector<VC> > &L, vector<vector<int> > &
    Libres, vector<int> &Num1s, const vector<int> &IndMax1, int k, int r)
{
    VC v = L.top();
    L.pop();
    if(!QuitaOrdena(v,Num1s,IndMax1,k,r))//Si v no es buena
        return;
    int pv = v.chkfin - (k-1)*v.chkdif;//Primer ndice vari
    int MultiOr = 0;//El Or de los primeros de tamao 1
    int kk = k-1;//Los que quedan a la derecha de pv
    for(; pv <= v.chkfin; pv+=v.chkdif, --kk)
    {
        if(Num1s[v.vari[pv]] > 1)
            break;//Si encontr el primero de tamao >1
        if((v.vari[pv] & MultiOr) != 0)//Si este color ya lo us, entonces no
             hay hteros
        {
            if(!nextPA(v.chkfin,v.chkdif,v.vari.size(),k))//Si adems estoy
                en la ltima revisin
            {
                                RevisaVariForzados(v.vari, IndMax1);
                                v.Imprimir();
                Libres.push_back(v.vari);//Es libre!!
            }
            else
            {
                L.push(v);
            }
            return;
        }
        MultiOr |= v.vari[pv];//Aumento este color en MultiOr
    }
    //Ya calcul pv,MultiOr,kk
    int Inocuos = MultiOr & v.vari[pv];//Inocuos que obviamente son inocuos
    int CPosib = (v.vari[pv] & (~Inocuos));//Colores que falta revisar
    while(CPosib != 0)//Voy a pasar por todos los colores posibles
    {
        int C = IndMax1[CPosib];//Primer color a revisar
        if(HayEspHetero(v,MultiOr | C,v.chkfin,v.chkdif,kk,Num1s,IndMax1))
        {
            v.vari[pv] = C;
            L.push(v);//Agrego este ocuo
        }
        else
        {
            Inocuos |= C;//Aumento color Inocuo
        }
        CPosib -= C;//Ya revis el color
```

```
    }
    //Ahora agregamos el inocuo
    if(Inocuos == 0)
    {
        return;
    }
    v.vari[pv] = Inocuos;
    if(!nextPA(v.chkfin,v.chkdif,v.vari.size(),k))
    {
        if(OrdenLexi(v.vari,Num1s,IndMax1,r) != 0)
        {
                        RevisaVariForzados(v.vari, IndMax1);
                        v.Imprimir();
            Libres.push_back(v.vari);
for(auto i : v.vari) ImprimeColores(i);
cout << endl;
        }
    }
    else
    {
        L.push(v);
    }
}

void RevisaVariForzados(vector<int> &vari, const vector<int> &IndMax1) {
        //Limpia los colores forzados por el orden lexicografico cuand solo
            hay una posicin que contenga algn color t.
        int Probe=0, Doubles=0;
        int n = vari.size();
        for (int index = 0; index < n; index++) { //Revisa en cada posicin
                int Max = IndMax1[vari[index]];
                if (Probe < (Probe | Max)) {//Si es la primera vez que el
                    color aparece
                        Probe |= Max;
                }
                else {//Si ya haba aparecido antes
                        Doubles |= Max; //Es doble
                }
        }
        int ToClean = ~Doubles & Probe;//Los colore por limpiar son aquellos
             que no son dobles
        for (int index = 0; index < n; index++) {//Segunda pasada para
            limpiar.
                int Max = IndMax1[vari[index]];
                if ((ToClean & Max) != 0) {//Si esta posicin se debe limpiar
                        vari[index] = Max; //Limpia
                        ToClean -= Max;
                }
        }
}
```

Bibliografía

[1] V. Rosta, "Ramsey theory applications," *The Electronic Journal of Combinatorics*, vol. 1000, pp. DS13–Dec, 2004. 1

[2] R. L. G. J. H. Spencer, "Ramsey Theory," *Scientific American*, vol. 263, pp. 112–117, 1990. 2, 25

[3] B. Hayes, "Gauss's Day of Reckoning A famous story about the boy wonder of mathematics has taken on a life of its own," *American Scientist*, vol. 94, no. 3, p. 200, 2006. 11

[4] P. Schweitzer, "Problems of Unknown Complexity: Graph Isomorphism and Ramsey theoretic numbers," Ph.D. dissertation, Universität des Saarlandes & Max-Planck-Institut für Informatik, 2009. 19, 32, 41, 47

[5] G. Kolata, "Paul Erdős, 83, a Wayfarer In Math's Vanguard, Is Dead," *New York Times*, 1996, http://www.nytimes.com/1996/09/24/us/paul-erdos-83-a-wayfarer-in-math-s-vanguard-is-dead.html. 25

[6] D. H. Mellor, "Cambridge Philosophers I: FP Ramsey," *Philosophy*, vol. 70, no. 272, pp. 243–262, 1995. 26

[7] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *science*, vol. 332, no. 6025, pp. 60–65, 2011. 26

[8] J. Erickson, "Finding longest arithmetic progressions," http://jeffe.cs.illinois.edu/pubs/arith.html, 1999. 32

[9] M. D. Beeler, "A new Van der Waerden number," *Discrete Applied Math.*, vol. 6, no. 2, p. 207, 1983. 37

[10] S. Butler, C. Erickson, L. Hogben, K. Hogenson, L. Kramer, R. L. Kramer, J. C.-H. Lin, R. R. Martin, D. Stolee, N. Warnberg *et al.*, "Rainbow arithmetic progressions," *arXiv preprint arXiv:1404,7232*, 2014. 44, 45

[11] M. Y. Zhanar Berikkyzy, Alex Schulte, "Anti-van der Waerden numbers of 3-term arithmetic progressions," *arXiv preprint arXiv:1604.08819*, 2016. 45

[12] D. Stolee, "Rainbow Arithmetic Progressions I: Search Algorithm," https://computationalcombinatorics.wordpress.com/2014/04/30/rainbow-arithmetic-progressions-i-search-algorithm/, 2014. 46, 52, 53

[13] B. M. Landman and A. Robertson, *Ramsey Theory on the integers*, 2nd ed. American Mathematical Society, 2014.

[14] R. Stevens and R. Shantaram, "Computer-generated Van der Waerden partitions," *Math. Comput.*, vol. 32, pp. 635–636, 1978.

[15] T. Ahmed, "Some more Van der Waerden numbers," *J. integer Seq.*, vol. 16, 2013.

[16] ——, "On computation of exact Van der Waerden numbers," *Integers*, vol. 11, 2011.

[17] B. Llano and A. Montejano, "Rainbow-free colorings for x + y = cz in Zp," *Discrete Math.*, vol. 312, pp. 2566–2573, 2012.

[18] M. Kouril, "A Backtracking Framework for Beowulf Clusters with an Extension to Multi-cluster Computation and SAT Benchmark Problem Implementation," Ph.D. dissertation, University of Cincinnati, 2006.

[19] V. Chvatal, *Combinatorial Structures and their Applications.* Gordon and Breach, 1970.

[20] V. Jungić, J. Licht, M. Mahdian, J. Nešetřil, and R. Radoičić, "Rainbow arithmetic progressions and anti-Ramsey results," *Combinatorics, Probability and Computing*, vol. 12, no. 5+ 6, pp. 599–620, 2003.

[21] T. Ahmed, "Two new van der Waerden numbers: w (2; 3, 17) and w (2; 3, 18)," *Integers*, vol. 10, no. 4, pp. 369–377, 2010.

[22] H. S. Tanbir Ahmed, Oliver Kullmann, "On the van der Waerden numbers w(2;3,t)," *arXiv preprint arXiv:1102.5433*, 2014.

[23] M. Kouril, "Computing the van der Waerden number W(3,4) = 293," *Integers*, vol. 12, p. 46, 2012.

[24] T. Ahmed, "On computation of exact van der Waerden numbers," *Integers*, vol. 12, no. 3, pp. 417–425, 2012.