Nonograma

Galdos Rodriguez, Emmanuel
emmanuel.galdos@ucsp.edu.pe

Yari Merma, Erick Branco
erick.yari@ucsp.edu.pe

Alvarez Reyes, Javier Oswaldo
javier.alvarez@ucsp.edu.pe

Yucra Ccolqque, Pablo Cesar
pablo.yucra@ucsp.edu.pe

Sicos Barrera, Rony Rodrigo
rony.sicos@ucps.edu.pe

Junio 2022

Índice

1. Introducción

Los Nonogramas también llamados acertijos japoneses, son acertijos lógicos y fueron inventados por el diseñador Non Ishida. El objetivo es colorear las celdas de tal manera que los cuadros de colores en cada fila y columna correspondan al número y la estructura especificada, basándonos en los números correspondientes a cada fila y columna de la cuadrícula, podemos derivar qué secuencias de píxeles negros deben colorearse para obtener la imagen final. Resolver tal rompecabezas puede verse como un caso especial de tomografía discreta(1).
Estos problemas matemáticos pueden ser resueltos de diversas formas y métodos, pero en este artículo nos hemos propuesto dar a entender de forma más detallada cómo es posible resolver un Nonograma, además de dar una explicación clara de por qué este problema es una NP-Completo, partiendo de la reducción del problema *SAT* el cual sabemos claramente que si es un problema *NP-Completo*. La

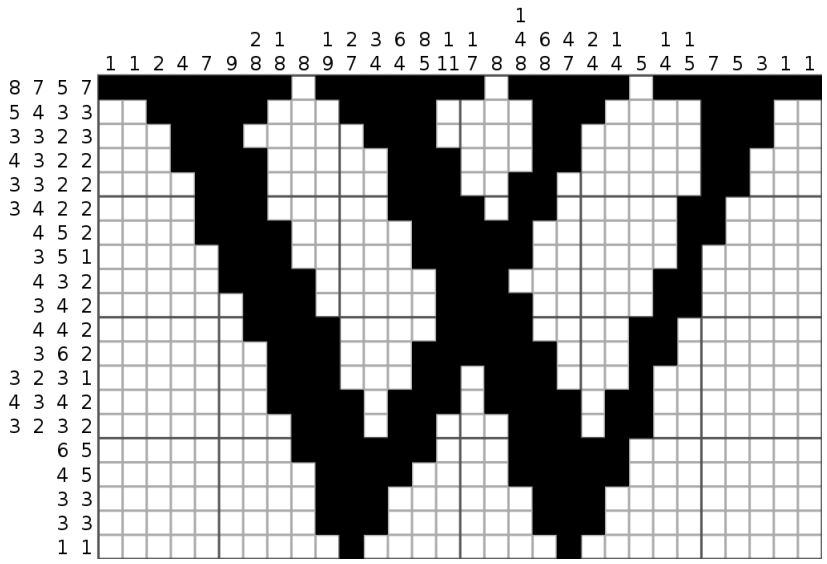


Figura 1: Un nonograma solucionado de la letra «W».

complejidad computacional de un problema a veces se puede determinar o estimar proporcionando un algoritmo para resolver el problema (para los llamados problemas P) o verificando que una solución candidata satisface el problema (para problemas NP). Otra forma de mostrar la complejidad es reducir un problema del que se conoce la complejidad al problema dado. Es decir, mostramos que si pudiéramos resolver instancias de nuestro nuevo problema, también podríamos resolver instancias del problema cuya complejidad se conoce. Esto resultará útil para problemas *NP-completos* y *NP-Difícil*. Usamos el concepto «Another Solution Problem» (**ASP**) para determinar la complejidad del problema de unicidad para nonogramas. Introducimos el problema de emparejamiento tridimensional(three dimensional matching problem o **3DM**) y mostramos su complejidad de manera similar a NCL acotado (Bounded Nondeterministic Constraint Logic problem). Usamos el hecho de que cuando un problema tiene otra solución y se reduce a un nuevo problema, ese nuevo problema también tiene otra solución. Con esta noción, demostramos que el problema de unicidad para Nonogramas también es *NP-completo*.

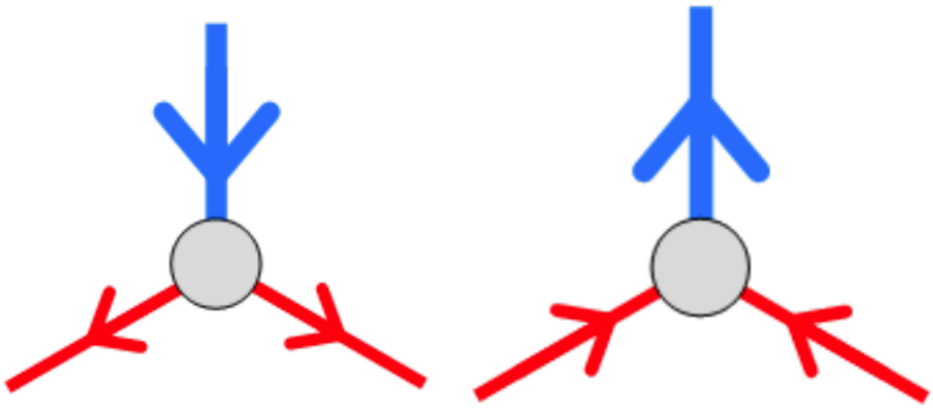
2. Conocimientos previos

2.1. Grafos de restricciones

Un gráfo de restricción es un par (T, m) en el que T es un gráfo simple, dirigido y ponderado, y m : $V \to \mathbf{R}$. El número $m(v)$ se denomina el flujo de entrada mínimo de v, y es un límite inferior para la entrada $t(v)$ de v, es decir, $t(v) \geq m(v)$.
Para que el concepto de grafos de restricciones sea útil en la aplicación, estos grafos deben tener algunas propiedades adicionales: nos restringimos a casos con función de peso w : $E \to \{1, 2\}$ y no

permitimos bucles automáticos. Para hacer dibujos útiles de tales grafos, las aristas con peso 1 se colorean en rojo y se dibujan relativamente delgados, y las aristas con peso 2 se colorean en azul y se dibujan relativamente gruesos. De ahora en adelante, hablaremos de aristas rojas y azules, al considerar los pesos de las aristas.

A diferencia de los grafos dirigidos ordinarios, las direcciones de las aristas no son fijas, pero



(a) Inicial (b) Invertido

Figura 2: El vértice **AND** en ambas configuraciones.

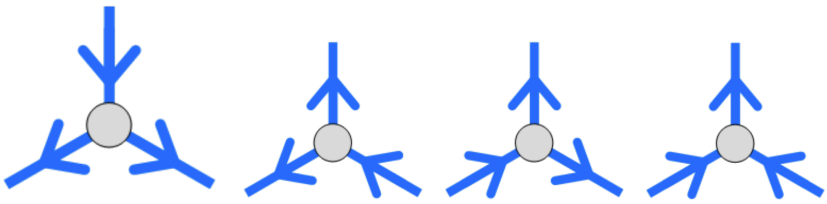
se pueden invertir. Un movimiento legal es la inversión de una sola arista y de un grafo G_0, de modo que el grafo resultante G_1 sigue siendo un grafo de restricción. Es decir, la condición $t(v) \geq m(v)$ se sigue cumpliendo para cada vértice v. De ahora en adelante solo invertiremos una arista cuando sea legal, para asi ser capaces de satisfacer los problemas posteriormente descritos.
Los vértices de los gráficos de restricciones pueden representar operadores lógicos. Para empezar, tomamos un vértice con tres aristas adyacentes y establecemos su entrada mínima $m(v)$ igual a 2. Esto da varias posibilidades. Un vértice con una arista azul hacia adentro y dos aristas rojas hacia afuera representa un operador AND. Por lo tanto, se denominará vértice AND y se muestra en la figura 2a.

La funcionalidad de este operador se vuelve clara cuando intentamos invertir la arista azul hacia adentro. Dado que la entrada mínima es igual a 2, esto solo se puede lograr cuando ambas aristas rojas ya se han invertido también (de lo contrario, el movimiento no sería legal). Esto se dibuja en la figura 2b. Como ambas aristas deben invertirse, podemos pensar en un AND lógico.

También se podría pensar en un vértice como un operador OR cuando es adyacente a tres aristas azules, con una arista apuntando hacia adentro y dos aristas hacia afuera, como se muestra en la figura 3a. Al invertir la único arista interior, basta con haber invertido cualquiera de las aristas de salida (o ambos, ya que no hablamos de un **OR exclusivo**). Esto corresponde a un **OR** lógico, como podemos ver en la figura 3b.

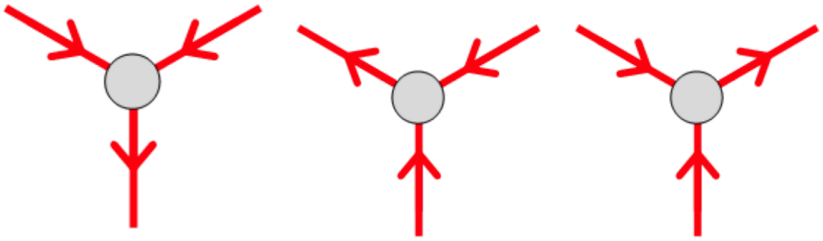
El vértice **CHOICE** tiene tres aristas rojas, uno que apunta hacia adentro y dos que apuntan hacia afuera. Cuando una arista apunta hacia afuera, las otras deben estar dirigidos hacia adentro, para satisfacer la entrada mínima de 2. Esto da como resultado el hecho de que cualquiera de los bordes internos se puede invertir, pero no ambos, y esto es a lo que se refiere «**CHOICE**». Tenga en cuenta que la arista exterior única ya debe invertirse en cada caso, aunque esto no cambia nada en la funcionalidad.

Este vértice representa la funcionalidad de un **NOT** lógico de la siguiente manera: asigne una



(a) Inicial (b) Toda inversión posible

Figura 3: El vértice **OR** en todas las configuraciones.



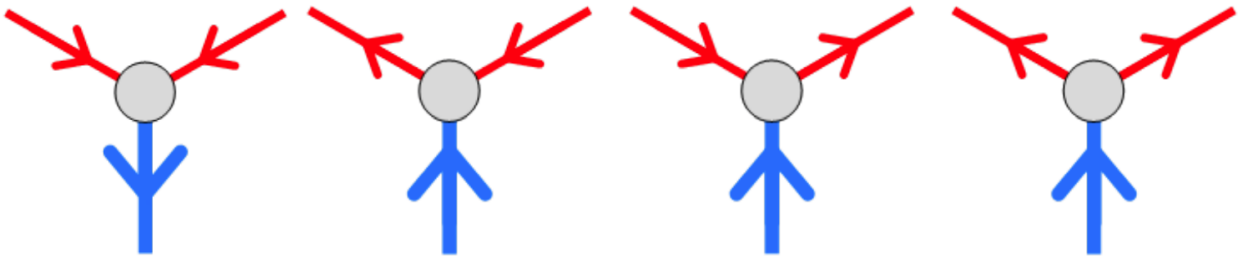
(a) Inicial (b) Ambas posibles reversiones

Figura 4: El vértice **AND** en ambas configuraciones.

variable lógica a una arista interior, digamos x, y su negación al otro, digamos $\overline{x}$. Esto significa que si la arista x apunta hacia afuera, la $\overline{x}$ tiene que apuntar hacia adentro y viceversa. Cuando imaginamos que apuntar hacia afuera como una variable es verdadera, inmediatamente se sigue que su forma negada tiene que ser falsa, y viceversa.

El vértice **FANOUT** tiene dos aristas rojas hacia adentro y una sola arista azul hacia afuera, vea la figura 5a. En principio, este vértice es el mismo que el vértice **AND**, aunque las funciones de las aristas han cambiado. El vértice tiene más o menos la misma funcionalidad que el vértice **CHOICE**. Sin embargo, con este vértice podemos invertir cualquiera de las aristas internas, así como ambos. Tenga en cuenta que todas las inversiones requieren que la arista exterior azul se invierta en la derecha, vea la figura 5b.

Hay que tener en cuenta que también es posible tener vértices con una arista roja y dos azu-



(a) Inicial (b) Ambas posibles reversiones

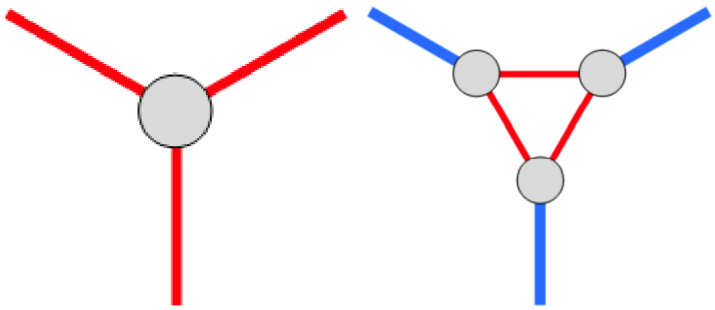
Figura 5: El vértice **FANOUT** en todas las configuraciones.

les. Estos vértices resultan poco útiles, por lo que los omitiremos. Los vértices descritos anteriormente se pueden conectar entre sí para crear un gráfo de restricciones. Entonces, para cada uno de los vértices posibles, tenemos que satisfacer la entrada mínima de 2 en cualquier caso. Sin embargo, en algunos casos es necesario conectar dos vértices por una sola arista, aunque la arista necesita tener diferentes pesos para alcanzar la funcionalidad de sus vértices adyacentes. Por supuesto, esto es imposible, por

lo que introducimos vértices con dos aristas adyacentes, una roja y otra azul. Llamamos a tal vértice vértice rojo-azul, y tiene un flujo de entrada mínimo de 1.
Esto significa que una arista apunta hacia adentro si y sólo si el otro apunta hacia afuera. Dado que el flujo de entrada mínimo es 1, no importa qué arista esté apuntando hacia adentro y, por lo tanto, estos pueden revertirse. Estos vértices son útiles cuando se construyen grafos en los que se debe preservar la orientación de una cierta arista, y mientras tanto se deben intercambiar los pesos.

Podemos preguntarnos si hay otros vértices útiles con grado 2 y entrada mínima 1. Se podría pensar en un vértice como un operador **NOT**, es decir, intercambiando la orientación de una arista por un vértice. Sin embargo, esto no es muy conveniente, ya que significa que ambas aristas apuntan hacia adentro o hacia afuera. Este último caso contradice el principio de entrada mínima, ya que sería igual a 0 y da como resultado una configuración ilegal. Por lo tanto, no sería posible invertir una de las aristas, por lo que este concepto de **NOT** lógico es inútil en el sentido de los gráficos de restricciones. Tenga en cuenta que ya hemos implementado un **NOT** lógico en cierto sentido al introducir el vértice CHOICE, que elimina la necesidad de un vértice **NOT** relacionado con la entrada (un vértice) y la salida (un vértice) estrictamente lógicos.

Los grafos que contienen vértices **AND** combinados con **OR**, junto con vértices rojo-azul, se denominan gráficos de restricción **AND/OR**. Tenga en cuenta que los vértices **CHOICE** y rojo-azul no están incluidos en la definición de los gráficos de restricción **AND/OR**. Sin embargo, podemos reemplazarlos con subgrafos **AND/OR** equivalentes, es decir, una estructura de vértices **AND** y **OR** conectados que mantendrán el comportamiento de estos vértices. El vértice CHOICE es equivalente al subgrafo **AND/OR** dibujado en la figura 6b. Tenga en cuenta que cada uno de las aristas rojas ha sido reemplazado por uno azul. Con un vértice rojo-azul colocado entre cada una de las aristas, todos los problemas deben ser omitidos.



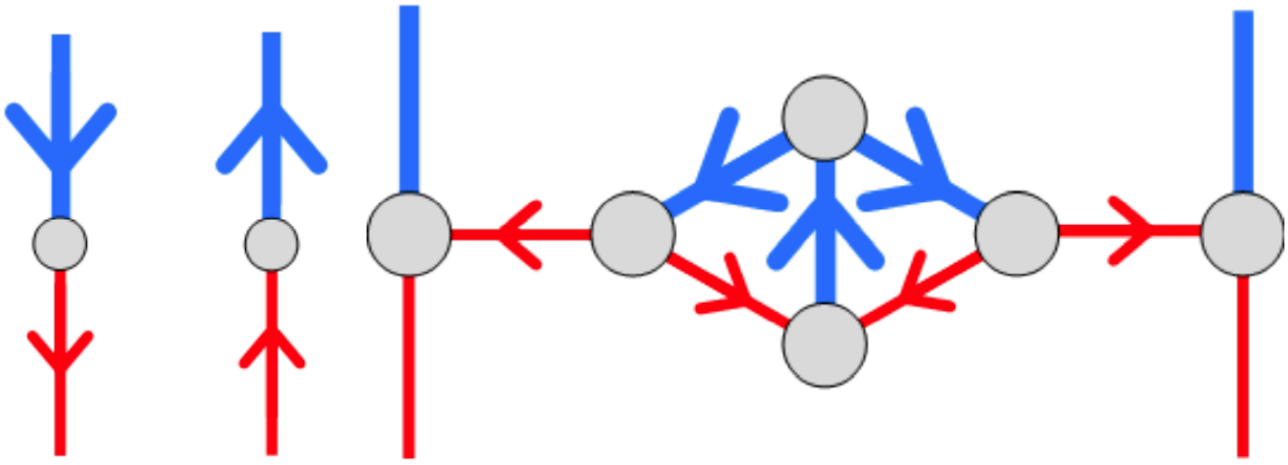
(a) Vértice original (b) Subgrafo equivalente

Figura 6: El reemplazo de un vértice CHOICE por un equivalente **AND/OR** subgrafo.

Debemos notar que los vértices rojo-azul siempre vienen en pares, ya que estarán conectados a los extremos de un vértice AND u OR. El subgrafo equivalente de dos vértices rojo-azul se muestra en la figura 7. Tenga en cuenta que en la figura 7b la orientación de las aristas originales está ausente, ya que esta orientación no depende de la estructura posterior del subgrafo. Esto debe ser verificado por el lector.

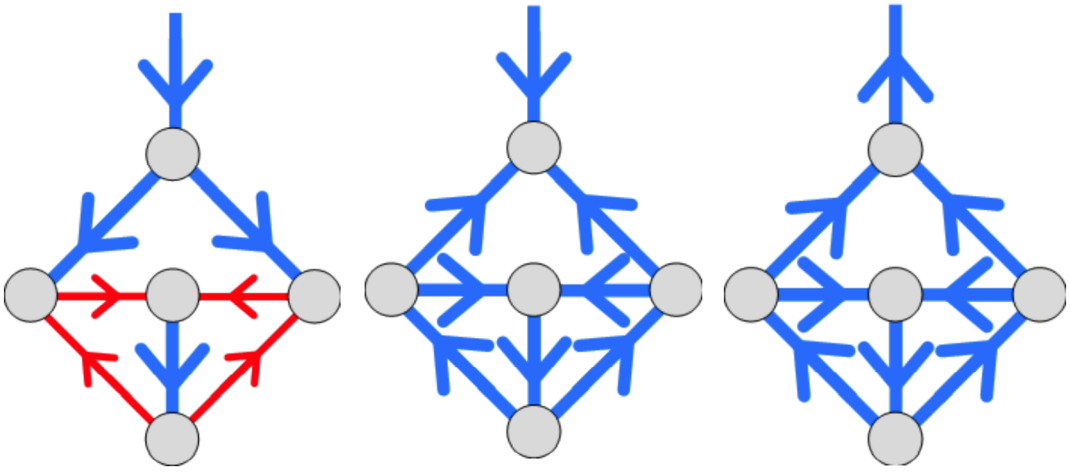
A veces tenemos que crear una arista, sin más conexión con otro vértice. Dichas aristas se denominan aristas sueltas y también se pueden reemplazar por un subgrafo equivalente. Hay que tener en cuenta que las aristas sueltos tienen varios propósitos posibles: la arista tiene la libertad de invertirse o su dirección debe determinarse con la derecha. En la figura 8a se muestra el subgráfico **AND/OR** equivalente de una arista bloqueada, mientras que las figuras 8b y 8c muestran las posibles direcciones de una arista libre suelta, reemplazado por un subgráfico equivalente.

Para varios problemas, tenemos que mirar específicamente los gráficos planos. Es decir, gráficos que no tienen autointersecciones en las aristas. Suponemos que las aristas siempre se dibujan en líneas rectas. Para evitar las autointersecciones, podemos reemplazar dicha intersección por un subgrafo que conserve la orientación de las aristas opuestas: el dispositivo de cruce. En la figura 9, el dispositivo cruzado se muestra en su forma dirigida y no dirigida. Tenga en cuenta que hay vértices con cuatro



(a) vértices (b) Subgrafo equivalente

Figura 7: El reemplazo de dos vértices rojo-azul por un equivalente **AND/OR** subgrafo.



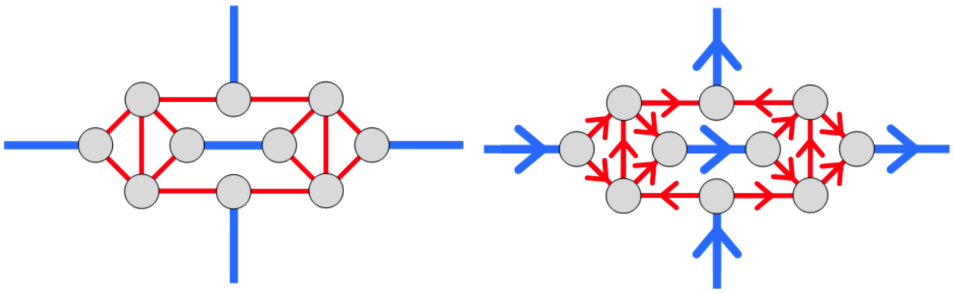
(a) Arista Bloqueada (b) Arista libre hacia adentro (c) Arista libre hacia afuera

Figura 8: Determinación y orientación de aristas sueltas.

aristas rojas en este subgráfico, lo que significa que al menos dos de sus aristas deben apuntar hacia adentro. Se deja al lector comprobar que, cuando se invierte una de las aristas sueltas interiores, se debe invertir también la arista opuesta y, por lo tanto, siempre se conservará la orientación de estas aristas.

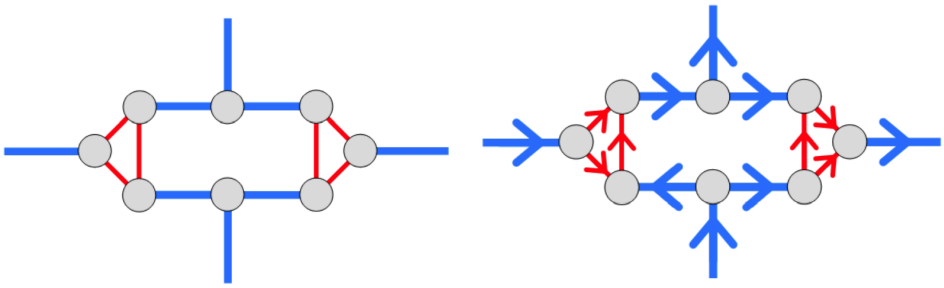
Tenga en cuenta que este gadget no es en ningún sentido un (sub)grafo restringido **AND/OR**, ya que hay vértices con cuatro aristas rojas involucrados. Por esta razón, necesitamos construir un subgrafo equivalente, o al menos un subgrafo con la misma funcionalidad. El gadget de medio cruce es un gadget para el que al menos dos de las aristas sueltas exteriores deben mirar hacia adentro. En la figura 10 se muestra una determinada configuración del gadget medio cruzado, así como su versión no dirigida.

Inmediatamente tenemos que demostrar que este dispositivo puede conservar la orientación de las aristas sueltas opuestos, como se muestra en la figura 11.
Sin embargo, hay algunas otras configuraciones posibles, a saber, un par de aristas sueltas opuestas giran hacia adentro y la otra apunta hacia hacia afuera, consulte la figura 12. Hay otras configuraciones posibles, ya que al menos dos de las aristas sueltos deben estar dirigidos hacia adentro. Esto debe ser verificado por el lector. La funcionalidad de este gadget al menos es suficiente para nuestro propósito,



(a) Direcciones indeterminadas (b) Direcciones determinadas

Figura 9: El dispositivo cruzado.

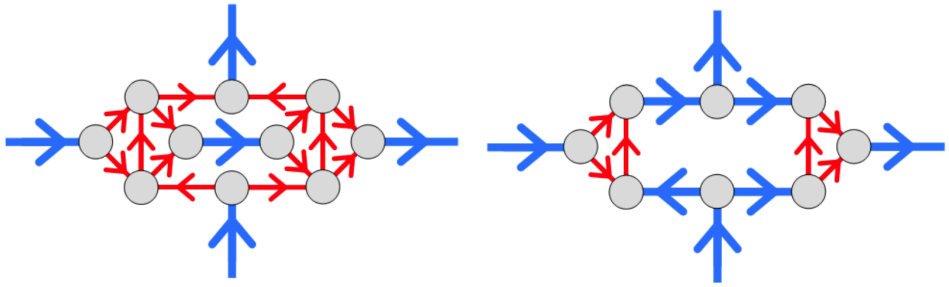


(a) Direcciones indeterminadas (b) Direcciones determinadas

Figura 10: El gadget medio cruzado.

es decir, mantener la orientación de las aristas que se cruzan. Cuando dos aristas sueltas ortogonales (es decir, no opuestos, ya que no podemos hablar de adyacencia) ambas apuntan hacia afuera, significa que sus aristas opuestas deben apuntar hacia adentro y, por lo tanto, su orientación se transmitirá a través del dispositivo.

Con todos los vértices, aristas, subgrafos y gadgets posibles, podemos construir varios gráficos



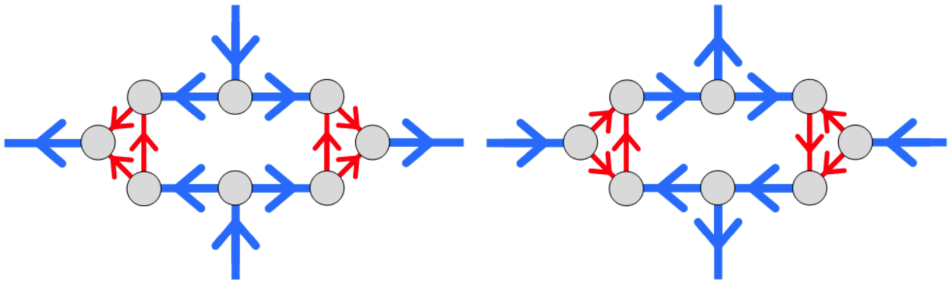
(a) El dispositivo cruzado (b) El gadget medio cruzado

Figura 11: Equivalencia de los dos gadgets.

de restricciones. Estos grafos se utilizarán para determinar la complejidad y la resolución de varios juegos y acertijos, como el acertijo Nonogram.

2.2. Problemas P y NP-Completos

Los problemas P y NP son grupos de lo que llamamos «problemas decidibles» o «problemas computables». En esta teoría, la clase P consiste de todos aquellos problemas de decisión que pueden ser resueltos en una máquina determinista secuencial en un período de tiempo polinomial en proporción a



(a) Primera configuración (b) Segunda configuración

Figura 12: Las otras dos configuraciones del gadget medio cruzado.

los datos de entrada. En la teoría de complejidad computacional, la clase P es una de las más importantes, la clase NP consiste de todos aquellos problemas de decisión cuyas soluciones positivas/afirmativas pueden ser verificadas en tiempo polinómico a partir de ser alimentadas con la información apropiada, o en forma equivalente, cuya solución puede ser hallada en tiempo polinómico en una máquina no determinista. Por lo tanto, la principal pregunta aún sin respuesta en la teoría de la computación está referida a la relación entre estas dos clases.
La relación entre las clases de complejidad NP y P es una pregunta por primera vez formulada por el científico computacional Stephen Cook que la teoría de la complejidad computacional aún no ha podido responder. En esencia, la pregunta ¿es P = NP completo ? significa: si es posible «verificar» rápidamente soluciones positivas a un problema del tipo SI/NO (donde «rápidamente» significa «en tiempo polinómico»), ¿es que entonces también se pueden «obtener» las respuestas rápidamente?

2.2.1. La clase P

P es conocido por contener muchos problemas naturales, incluyendo las versiones de decisión de programa lineal, cálculo del máximo común divisor, y encontrar una correspondencia máxima. Algunos problemas naturales son completos para P, incluyendo la conectividad (o la accesibilidad) en grafos no dirigidos. Una generalización de P es NP, que es la clase de lenguajes decidibles en tiempo polinómico sobre una máquina de Turing no determinista. De forma trivial, tenemos que P es un subconjunto de NP. Aunque no está demostrado, la mayor parte de los expertos creen que esto es un subconjunto estricto.

2.2.2. La clase NP

La clase NP está compuesta por los problemas que tienen un certificado sucinto (también llamado testigo polinómico) para todas las instancias cuya respuesta es un SÍ. La única forma de que tengan un tiempo polinomial es realizando una etapa aleatoria, incluyendo el azar de alguna manera para elegir una posible solución, y entonces en etapas posteriores comprueba si esa solución es correcta. En otras palabras, dada una solución para una cierta instancia, es posible comprobar que es válida en TIME (n^k). En el caso de SAT (Problema de satisfacibilidad booleana), dado una asignación de valores de verdad, se puede comprobar fácilmente si la fórmula es cierta o no. Una nMT puede .ªdivinar"la solución en $O(n)$ y verificarla en tiempo polinómico.

2.2.3. NP-Completo

Para abordar la pregunta de si $P = NP$, el concepto de la completitud de NP es muy útil. Informalmente, los problemas de NP-completos son los problemas más difíciles de NP, en el sentido de que son los más probables de no encontrarse en P. Los problemas de NP-completos son esos problemas NP-duros que están contenidos en NP, donde los problemas NP-duros son estos que cualquier problema en P puede ser reducido a complejidad polinomial. Por ejemplo, la decisión del Problema del viajante de comercio es NP-completo, así que cualquier caso de cualquier problema en NP puede ser transformado mecánicamente en un caso del Problema del viajante de comercio, de complejidad polinomial. El Problema del viajante de comercio es de los muchos problemas NP-completos existentes. Si cualquier

problema NP-completo estuviera en P, entonces indicaría que $P = NP$. Desafortunadamente, se sabe que muchos problemas importantes son NP-completos y a fecha de 2008, no se conoce ningún algoritmo rápido para ninguno de ellos.

2.3. Complejidad Computacional

Al mirar los algoritmos para resolver problemas, queremos decir algo útil sobre la posible eficiencia de tales algoritmos. El tiempo utilizado por un algoritmo puede ser representado por el tiempo de cálculo o el número de pasos de cálculo.

1. Se dice que un algoritmo toma tiempo polinomial si existe un polinomio f(n) que es un límite superior para el tiempo utilizado por este algoritmo, donde n es el tamaño de entrada de una instancia a la que se aplica el algoritmo. Con esta noción, ya podemos clasificar varios problemas en dos clases de complejidad.

2. Se dice que un problema pertenece a la clase polinomial o, abreviadamente, P si existe un algoritmo que resuelve el problema en tiempo polinomial.

3. Se dice que un problema está en la clase polinomial no determinista o abreviadamente NP si existe un algoritmo que verifica para una solución candidata en tiempo polinomial de tamaño de entrada n si es una solución al problema.

Entonces, decimos que un problema es P respectivamente NP. Cuando se cumple una declaración similar para la cantidad de espacio computacional, además de la cantidad de tiempo utilizado, decimos que el problema pertenece a la clase P-ESPACIO, o simplemente es *P -ESPACIO*. Tenga en cuenta que cualquier problema en P también es *P-ESPACIO*, ya que cualquier proceso que use un tiempo finito siempre usará una cantidad finita de espacio. Lo contrario no siempre es cierto: un proceso que usa solo una cantidad finita de espacio puede tomar un tiempo infinito. Un ejemplo bien conocido es un ciclo while. Aunque solo usa una cantidad finita de espacio (el código de programación y las variables creadas hasta ahora), puede durar para siempre mientras sus condiciones no se cumplan (todavía).
Se dice que un problema A es reducible a un problema B si existe un algoritmo de tiempo polinomial que vincula los resultados de A sobreyectivamente a los resultados equivalentes de B. Decimos que B está en la clase *NP-difícil* si este problema es reducible a cualquier problema en *NP* en tiempo polinomial. Decimos brevemente que un problema es *NP-difícil* en este caso. Además, un problema pertenece a la clase *NP-completo* si es tanto *NP* como *NP-difícil*. Con esta noción, tomamos ciertos problemas de los que se sabe que son *NP-completos* y los reducimos a nuevos problemas en tiempo polinomial. De esta manera, podemos demostrar que estos problemas también son *NP-completos*.

2.4. Problemas de Satisfacción (SAT)

Una fórmula booleana es una fórmula que contiene variables a las que se les puede asignar verdadero o falso, conocidas como variables booleanas. Estas variables se combinan con operadores lógicos para crear expresiones, que también pueden ser verdaderas o falsas. Estas definiciones están relacionadas con (9), Capítulo 3.

1. La conjunción $x \wedge y$ de dos variables x e y es verdadera cuando tanto x como y son verdaderas. Este operador se llama **AND** lógico.

2. La disyunción $x \vee y$ de x e y es verdadera cuando una de las variables es verdadera o ambas. Este operador se llama **OR** lógico.

3. La negación $\overline{x}$ de una variable x es verdadera si y solo si x es falsa. Se dice que el operador es un **NOT** lógico, y los símbolos x y $\overline{x}$ se denominan respectivamente literales positivos y literales negativos.

Las variables booleanas se pueden denotar con x, y, z, etcétera, así como con x_1, x_2, x_3, etcétera. Usaremos la última notación cuando necesitemos especificar fórmulas booleanas por sus números.

Los problemas de satisfacibilidad, en breve problemas *SAT*, se basan en la siguiente pregunta: dada una fórmula booleana, ¿existe una asignación para las variables para las que la fórmula es verdadera? En otras palabras, ¿puede satisfacerse esta fórmula?

Un enfoque de fuerza bruta está verificando cada posible combinación de asignaciones para cada variable. Como cada variable tiene dos valores posibles, el número de combinaciones es 2^n, donde n es el número de variables. Por lo tanto, no es inmediatamente obvio a qué clase de complejidad pertenece este problema. Para clasificar la complejidad de los problemas del *SAT*, nos fijamos en las fórmulas que están en forma normal conjuntiva (abreviado CNF). Es decir, la fórmula es una conjunción de cláusulas, donde cada cláusula es una disyunción de literales. Cada fórmula booleana se puede convertir a CNF, un algoritmo para hacer esto se da en (9), Sección 4.6.

El problema de satisfacibilidad más simple es el problema *2-SAT*. En este caso, cada una de las cláusulas no contiene más de dos literales. Resulta que hay algoritmos que resuelven este problema en tiempo polinomial(8), Section 2.2. No trataremos más este resultado en esta tesis, ya que estamos interesados en el problema *3-SAT*. Es decir, cada una de las cláusulas puede contener como máximo tres literales.

Ejemplo Un breve ejemplo es $(\overline{x} \vee y \vee z) \wedge (y \vee \overline{z} \vee w)$. Que consiste en dos cláusulas $(\overline{x} \vee y \vee z)$, y en $(y \vee \overline{z} \vee w)$. Lo cual es verdadero cuando, por ejemplo, x es verdadero, y es falso, z es verdadero y w es verdadero. Por lo tanto, esta fórmula es satisfactoria.

Ejemplo El problema *3-SAT* es *NP-completo* Esto significa que podemos mostrar la completitud de *NP* para otros problemas, si podemos reducir estos problemas al problema *3-SAT* en tiempo polinomial.

3. Nonogramas

3.1. Definición de un Nonograma

Esta sección es análoga a (6), Sección 2.
Los Nonogramas son acertijos logicos donde el objetivo es colorear las celdas de tal manera que los cuadros de colores en cada fila y columna correspondan al número y la estructura especificada.

Considere los símbolos 0 y 1 como valores que representan diferentes colores de píxeles. Nos referiremos al 1 como un píxel negro y al 0 como un píxel blanco. Además, el valor de un píxel puede ser indeterminado, especialmente al comienzo del proceso de resolución (es decir, todos los píxeles son indeterminados). En ese caso, tomamos «?» como el valor del píxel, y tendríamos $T = \{0, 1, ?\}$ como valores posibles.

Vamos a definir un nonograma como $N = \{I, D\}$ donde es una imagen I combinada con una descripción del monograma "D". Decimos que la imagen I se adhiere a una descripción D cuando cada fila y columna de I concuerdan con su descripción. Este es el objetivo principal de resolver un acertijo de Nonograma.

3.2. Problemas

Al observar con detenimiento múltiples Nonogramas, podemos identificar los siguientes problemas:

1. Existencia: Dado un Nonograma N, ¿Existe una solución ?.
2. Resolución: Dado un Nonograma N soluble, ¿Cómo podemos obtener una solución?.
3. Unicidad: Dada una solución para un Nonograma N, ¿Existe otra solución?.

Nótese que la solucionabilidad de un Nonograma infiere simplemente el mismo problema que la existencia de soluciones, ya que podemos decidir si el Nonograma es solucionable cuando hemos encontrado una determinada solución. Aunque, la naturaleza de estos problemas es bastante diferente. El problema de existencia es un problema de decisión, es decir, el problema se responde con un

«sí» o un «no». Sin embargo, el problema de la solución se responde con una solución completa. Por otro lado, determinar que existe una solución para un Nonograma determinado, no significa inmediatamente que también sea solucionable.

Si adaptamos el problema de resolución de modo que también se pueda aplicar a los nonogramas que no tienen solución, obtendremos que si el Nonograma no es solucionable, no obtendremos solución. Ya que solo podemos decir lo contrario si existe un algoritmo óptimo que resuelva cada Nonograma solucionable. Si este es el caso, y el algoritmo no devuelve una solución completa, podemos concluir que el nonograma no tiene solución. Por lo tanto, combinaremos los dos primeros problemas en uno. Este último problema a menudo se conoce como el «Problema de otra solución» o «Another Solution Problem» en inglés, por lo que denotaremos este problema con ASP.

4. El problema de la unicidad

4.1. El problema de emparejamiento tridimensional

Para determinar la complejidad del «Another Solution Problem» para Nonogramas (al cual nos referiremos de aquí en adelante como ASP de Nonogramas), presentamos otro problema para el cual podemos demostrar que es NP-completo.
Dados tres conjuntos disjuntos X, Y y Z con el mismo número de elementos q, y un conjunto $T \subseteq X \times Y \times Z$ de ternas ordenadas. ¿Existe un subconjunto $T' \subseteq T$ que consta exactamente de q elementos, tal que cada elemento de X, Y y Z respectivamente aparece en una sola de sus ternas? Tal subconjunto $T^{'}$ se llama emparejamiento de T.
Nuevamente, no es inmediatamente obvio a qué clase de complejidad pertenece este problema. Podríamos crear cada posible subconjunto T' de T que tenga q triples, y verificar si es un emparejamiento para T. Cuando los conjuntos X, Y y Z constan de q elementos, el conjunto T constará como máximo de q^3 triples. El número de posibles subconjuntos T' con q triples será como máximo

$$\binom{q^3}{q} = \frac{q^3!}{q! \cdot (q^3 - q)!}$$

Establecer el tamaño de entrada de la instancia igual a $n = q^3 + 3q$, el número de posibles subconjuntos T' no puede estar acotado superiormente por un polinomio finito $f(n)$.
Para reducir el problema de 3-SAT al problema de 3DM mediante una reducción de tiempo polinomial, construimos tres conjuntos X, Y y Z y un conjunto de triples ordenado $T \subseteq X \times Y \times Z$, tal que el requisito 3-SAT se cumple si y solo si 3DM está satisfecho.
Dada una fórmula booleana en 3-CNF con cláusulas $\{C_1, \ldots, C_m\}$ y las variables $\{x_1, \ldots, x_n\}$. Para cada variable x_i creamos un conjunto de elementos centrales

$$A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,2m-1}, a_{i,2m}\}$$

y elementos tip o simplemente tips.

$$B_i = \{b_{i,1}, b_{i,2}, \ldots, b_{i,2m-1}, b_{i,2m}\}$$

A partir de esto, creamos triples $t_{i,j} = \{a_{i,j}, a_{i,j+1 mod 2m}, b_{i,j}\}$. El índice $j + 1 mod 2m$ tiene el propósito de no exceder el número de elementos de A, que significa que el último triple es de la forma $t_{i,2k} = \{a_{i,2m}, a_{i,1}, b_{i,2m}\}$. Por lo tanto, cada elemento de A está contenido exactamente en dos triples. En adelante nos referiremos a los triples $t_{i,j}$ y tips $b_{i,j}$ con j par como triples pares (y tips), y con j impar como triples impares (y tips).
Por ejemplo, creamos una situación coincidente para la fórmula $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4)$. Esto significa que tenemos cuatro variables x_1, x_2, x_3, x_4 y dos cláusulas $C_1 = (x_1 \vee \overline{x_2} \vee x_3)$ y $C_2 = (x_2 \vee \overline{x_3} \vee x_4)$. Por lo tanto, tenemos nuestros conjuntos $A = \{a_{i,j}, 1 \leq i \leq 4, 1 \leq j \leq 4\}$ y $B = \{b_{i,j}, 1 \leq i \leq 4, 1 \leq j \leq 4\}$.Las ternas correspondientes son $t_{i,j} = \{a_{i,j}, a_{i,j+1 mod 4}, b_{i,j}\}$ con $1 \leq i \leq 4$ y $1 \leq j \leq 4$. Esto se muestra en la figura 13

La idea es adoptar triples impares o incluso triples en T'. Esto corresponderá a asignar $x_i = 0$

respectivamente $x_i = 1$ a una determinada variable. Para x_1, esto se muestra en la figura 14.

Para cada cláusula C_k creamos un par $c_k, c_K^{'}$. Luego combinamos estas parejas con los tips $b_{i,j}$ de la siguiente forma:

- Un triple $t_{c_k,i} = \{c_k, c_k^{'}, b_{i,j}\}$ con j impar se hace cuando x_i aparece en la cláusula C_k.
- Un triple $t_{c_k,i} = \{c_k, c_k^{'}, b_{i,j}\}$ con j par se hace cuando $\overline{x_i}$ aparece en la cláusula C_k.

Esto siempre es posible, ya que tenemos exactamente m cláusulas, y para cada variable m tips impares y m tips pares. En adelante nos referiremos a estos triples como cláusulas triples. Adoptando tal triple en T' corresponderá a satisfacer un literal. Si nosotros no somos capaces de adoptar al menos uno de estos triples, significa que no podemos asignar un valor de verdad a uno de los literales (y por lo tanto la cláusula) para representar la fórmula booleana satisfecha. Aunque se permite satisfacer más de un literal en una cláusula en el problema 3-SAT, tomaremos como máximo una cláusula triple para cada cláusula, ya que el propósito de 3DM es que todos los elementos ocurran solo una vez.

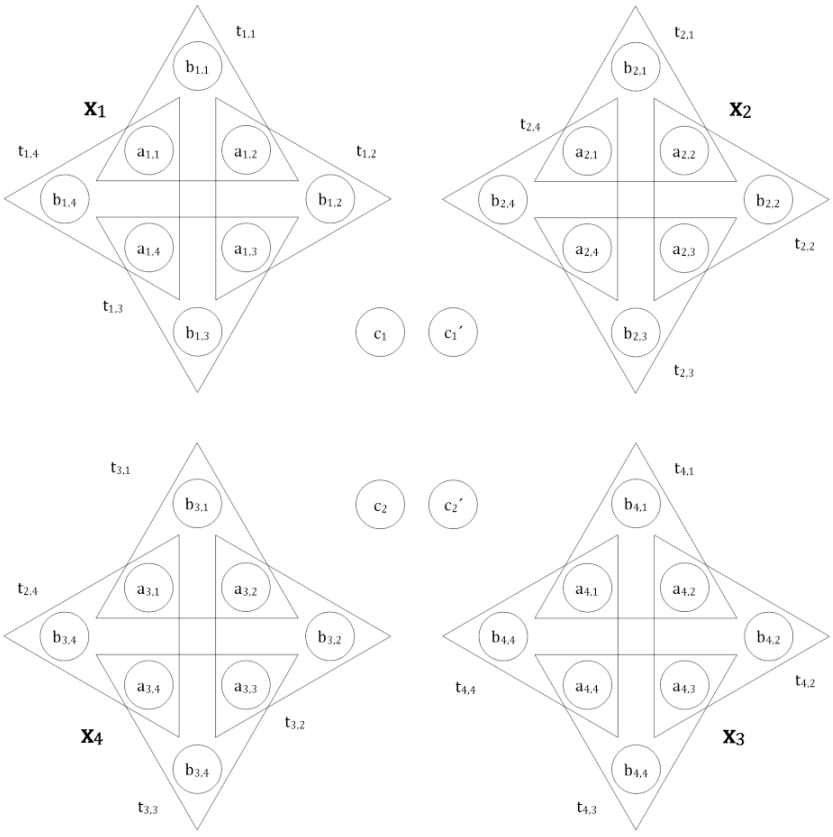


Figura 13: Los elementos central y los tips para $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4)$.

Como tenemos dos cláusulas, creamos las parejas c_1, c_1 y c_2, $c_2^{'}$. Por la primera cláusula, necesitamos los triples $t_{c1,1}$, $t_{c1,2}$, $t_{c1,3}$ con p_1, $p_1^{'}$, y respectivamente $b_{1,1}$, $b_{2,2}$ y $b_{3,1}$. Para la segunda cláusula, necesitamos $c_{2,1}$, $c_{2,2}$, $c_{2,3}$ con p_2, $p_2^{'}$ y respectivamente $b_{2,1}$, $b_{3,2}$ y $b_{4,1}$. Tenga en cuenta que para cualquier $b_{i,j}$ los números impares j pueden intercambiarse con cualquier otro número impar, y lo mismo vale para los números pares j. Los resultados se muestra en la figura 15.

En el problema 3DM, todos los elementos tienen que ocurrir exactamente una vez a lo largo

de todo los triples. Al sumar triples a $T^{'}$, los tips $b_{i,j}$ que no están contenidas en una de los literales no puede estar contenido en un triple. Para resolver esto, creamos parejas adicionales d_l, $d_l^{'}$, con $1 \leq l \leq 2n-k$ e inferir cada tip $b_{i,j}$. Tenemos que usar un método tan extenso, ya que no sabemos qué elementos de B van a estar contenidos en una cláusula triple, y cuáles no. De este modo, siempre estamos seguros de una posible coincidencia, siempre que se cumplan las cláusulas. En el ejemplo, creamos parejas p_m, $p_m^{'}$ con $3 \leq m \leq 8$ para los elementos restantes de B, y crea triples $\{p_m, p_m^{'}, b_{i,j}\}$ para cada $b_{i,j}$. El conjunto de triples T se compone de todos los triples descritos hasta ahora. Por la claridad de las imágenes, esto no está esbozado.

Decimos que una variable x_i se asigna VERDADERO si insertamos los triples pares en nuestra coincidencia, y FALSO si sumamos los impares. Esto significa que podemos agregar una cláusula triple que contiene $b_{i,j}$, con j impar, solo si para la variable correspondiente $x_i = 1$, y de manera similar $\overline{x_i} = 0$. Si no podemos sumar uno de los triples de la cláusula, entonces no podemos satisfacer la cláusula y, por lo tanto, toda la fórmula. esto corresponde a la situación del problema 3DM donde no todos los elementos de los conjuntos pueden ser contenido en exactamente un triple.

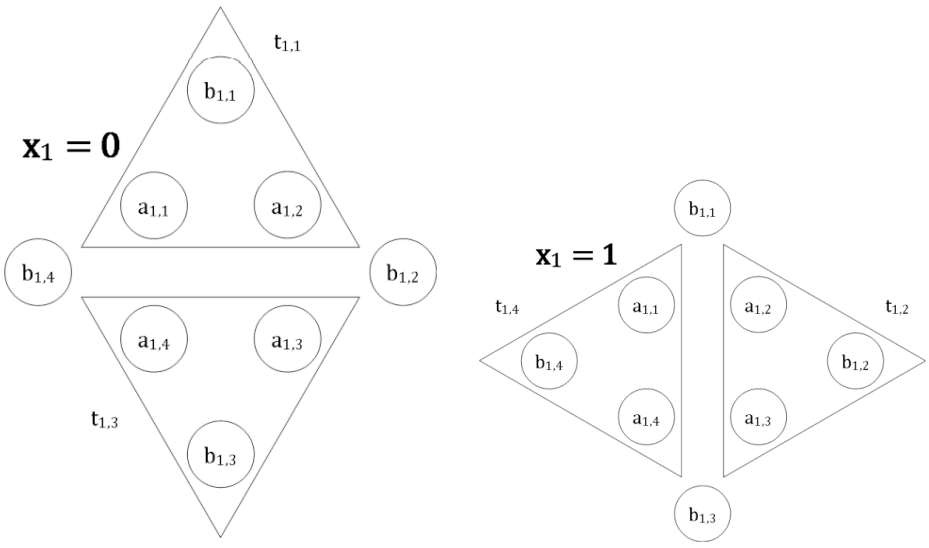
Ahora tratamos de determinar nuestra T, coincidente. Hay varias opciones, siempre y cuando se cumplan las cláusulas C_1 y C_2. Dos coincidencias posibles, sin elementos de limpieza, como se muestra en las figuras 16 y 17. Podemos ver que en cada caso, al menos una cláusula triple está contenida en la $T^{'}$ coincidente para cada cláusula, lo que significa que podemos asignar las variables x_1, x_2, x_3, x_4 de tal manera que la fórmula se cumple.
Ahora identificamos nuestros conjuntos X, Y y Z en el problema 3DM de la siguiente manera:

- $X = \{a_{i,j},\ impar\ j\} \cup \{c_k\} \cup \{d_l\}$,
- $Y = \{a_{i,j},\ par\ j\} \cup \{c_k^{'}\} \cup \{d_l\}$,
- $Z = \{b_{i,j}\}$.

Theorem 1. *El problema 3DM es NP-complete*

Esta prueba es análoga a la prueba en (2), Sección 8.6

Continuamos los pasos esbozados anteriormente. El último paso es mostrar que 3-SAT se re-



(a) Asignando falso a x1 (b) Asignando verdadero a x1

Figura 14: Las dos opciones posibles de núcleos triples

duce de hecho a 3DM de esta manera. Si es posible satisfacer cada cláusula en una fórmula booleana,

significa que a cada variable se le puede dar un valor tal que se satisfaga la fórmula booleana y, por lo tanto, podemos adoptar una cláusula triple en $T^{'}$ para cada cláusula. Para una variable x_i, todas las tips pares o tips impares están contenidas en un triple junto con elementos de A, y las otras están contenidas en un triple con $c_k, c_k^{'}$ o $d_l, d_l^{'}$ pares. Por lo tanto, cada elemento de X, Y y Z está contenido exactamente en un triple, lo que satisface el problema 3DM. Sin embargo, si no es posible satisfacer la fórmula, significa que hay al menos una variable x_i para la que no podemos encontrar un valor adecuado.

Para la situación de 3DM, significa que no se nos permite incluir unos tips par e impar en una cláusula triple. Si este es el caso para los tres literales en una cláusula, significa que esta cláusula no se puede cumplir y no se puede agregar ningún triple con c_k y $c_k^{'}$ a $T^{'}$. Por lo tanto, el problema de 3DM no se puede resolver.

El lector debe verificar que todos los pasos de reducción se pueden ejecutar dentro de un tiempo polinomial, lo que significa que el problema 3DM es NP-Difícil. El resto es para mostrar que el problema 3DM es NP. Para la verificación de un solo candidato que coincida con $T^{'}$, solo tenemos que verificar si cada elemento de X, Y y Z ocurre solo una vez a lo largo de $T^{'}$. Esto significa que el problema 3DM es NP y, en consecuencia, NP-completo.

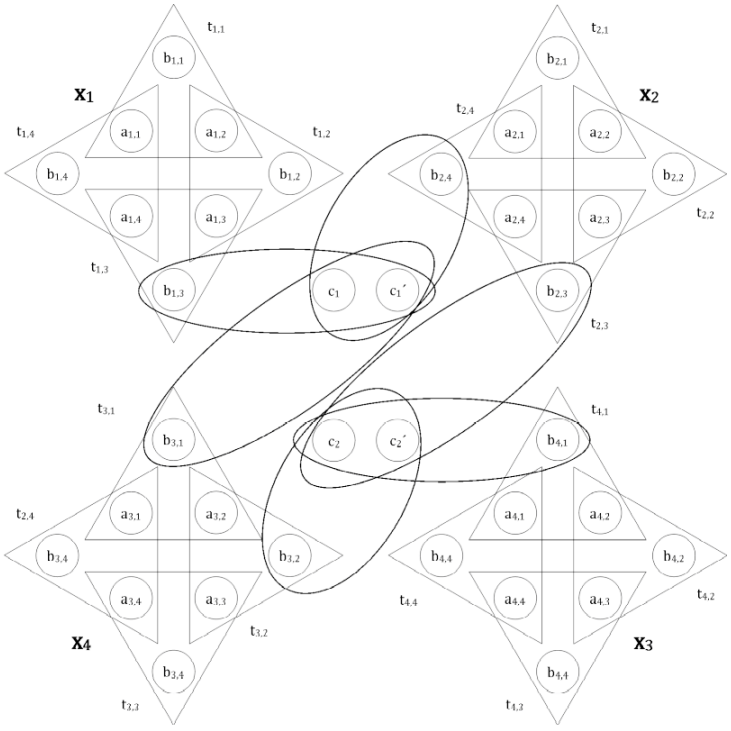


Figura 15: Los elementos adicionales de la cláusula y los triples para $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4)$.

4.2. Complejidad del problema de la unicidad

Según (11), Sección 1, una reducción parsimoniosa es una reducción del problema A al B, con un algoritmo que vincula los resultados de la biyectividad A con los resultados equivalentes de B. Por lo tanto, conserva el número de soluciones entre los problemas A y B. En otras palabras, el problema B tiene más de una solución siempre que el problema A las tenga. Recuerde que el ASP para un problema significa que tenemos que verificar si existe otra solución para la descripción dada. Por lo tanto, podemos concluir lo siguiente: si el problema A puede reducirse al problema B solución del problema con una reducción parsimoniosa, y el ASP para el problema A es NP-Difícil, entonces el ASP para el problema B también es NP-Difícil.

Theorem 2. *El ASP para 3DM es NP-completo*

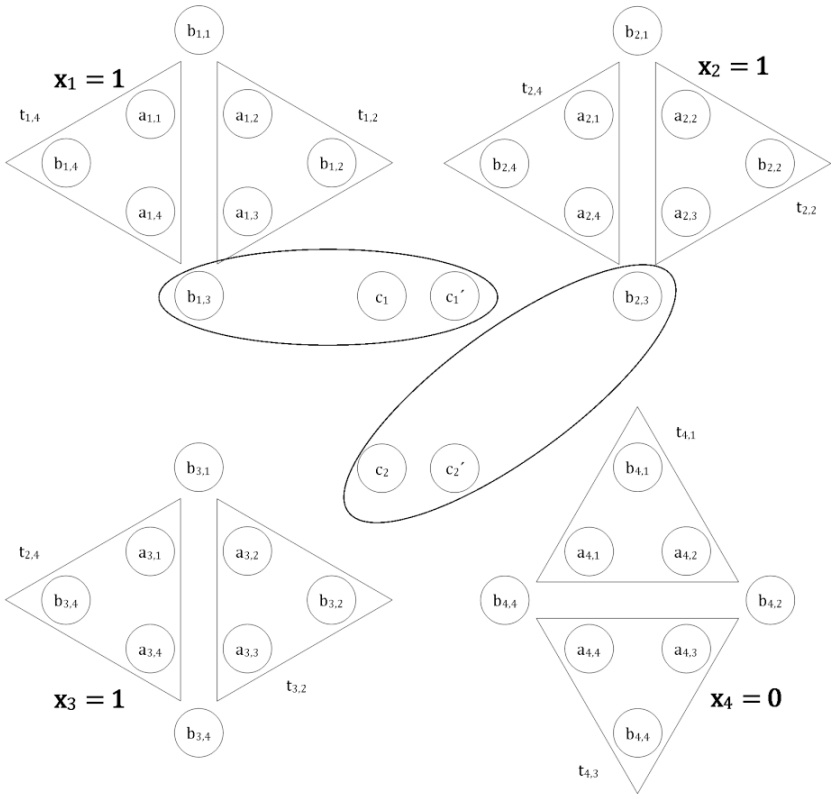


Figura 16: Una posible coincidencia T' para T

Uno puede imaginar que el ASP para 3DM es similar al propio 3DM y, por lo tanto, puede reducirse a 3DM. Los detalles de la prueba se pueden encontrar en (11), Sección 4.

Theorem 3. *El problema de unicidad de Nonograma es NP-completo*

Los detalles de esta prueba se pueden encontrar en (11), Sección 3. Aquí solo dibujaremos los contornos.
Mostramos la completitud de NP al reducir el problema 3DM al problema de existencia de Nonograma. Creamos una situación de Nonograma en la que obtenemos el mismo número de resultados que en 3DM, de manera que tenemos una reducción parsimoniosa. Dados tres conjuntos disjuntos $X = \{x_1, \ldots, x_q\}$, $Y = \{y_1, \ldots, y_q\}$ y $Z = \{z_1, \ldots, z_q\}$, que tienen todos q elementos, y un conjunto $T \subseteq X \times Y \times Z$ que consta de n triples, se construye el siguiente nonograma $N = (D, I)$:

1. La imagen I tiene un tamaño de $2n$ filas y $6q + 2$ columnas.
2. Excepto por la primera y la última columna, cada dos columnas adyacentes corresponden a un determinado elemento de X, Y o Z.
3. Cada fila impar corresponde a un determinado triple de M, de la siguiente manera:
   el primer y último píxel son de color negro, y tiene que haber números enteros de descripción, de forma que entre cada par de columnas haya una fila continua de píxeles negros correspondientes a elementos dentro de la terna mencionada.
4. Cada fila par consta de píxeles blancos
5. La descripción de cada columna consta de un número de unos, tal que la imagen se cumple

Un ejemplo con $q = 3$ se muestra en la figura 18

Cuando es posible encontrar otra T coincidente, tendremos una solución adicional para el Nonograma N. Esto significa que el número de coincidencias posibles para X, Y y Z es igual al número de soluciones para nuestro Nonograma. Por lo tanto, tenemos una reducción parsimoniosa de 3DM a

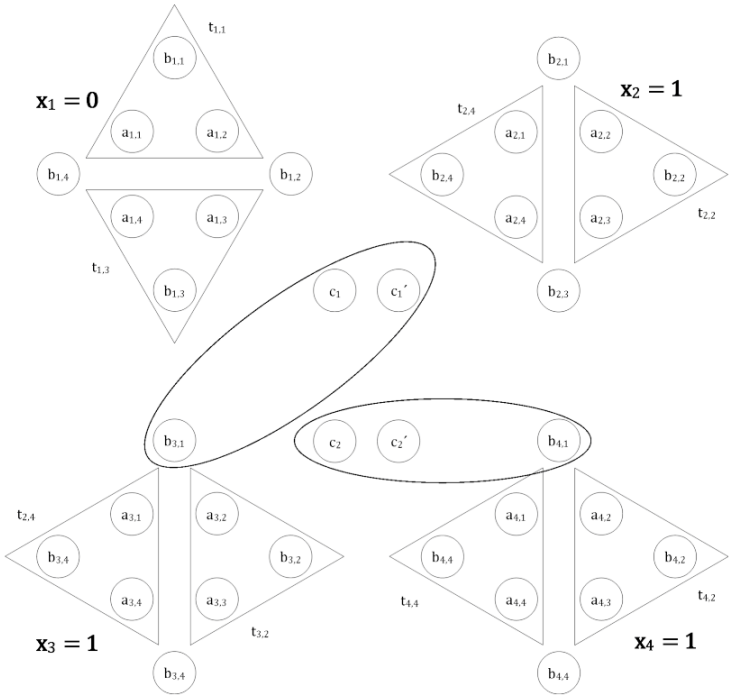


Figura 17: Otra posible coincidencia T' para T

Nonograma.

5. Soluciones propuestas

Ahora que hemos determinado la complejidad tanto de la solución como de los problemas de unicidad para los nonogramas en general, podemos centrarnos en el proceso de resolución en sí. En este capítulo, mostraremos varios posibles pasos de solución. Finalmente, aplicaremos varios ejemplos conocidos de nonogramas a diferentes solucionadores y compararemos el resultado. Además de los nonogramas que tienen una solución única o no, aquí distinguimos dos tipos de nonogramas. Los nonogramas simples se pueden resolver analizando cada fila o columna por separado. Si esto no es posible, debemos mirar la información de otras filas o columnas, o comenzar a buscar una solución. En esta sección, analizamos los pasos de resolución que se aplican a una fila o columna a la vez, por lo que en su mayoría se aplican a nonogramas simples. Otros nonogramas pueden tener algunos píxeles asignados, aunque el proceso de resolución terminará después de un tiempo.

5.1. Primera búsqueda en profundidad (DFS)

La clave para la búsqueda en profundidad primero es analizar todas las diferentes posibilidades en una determinada fila o columna, o posiblemente en toda la imagen, y en consecuencia determinar qué píxeles deben o no pueden colorearse en negro. En la medida de lo posible, intentaremos determinar tantos valores de píxeles como sea posible mediante determinación lógica. Si nos atascamos, podemos adivinar ciertos valores de píxeles y abandonar nuestra decisión cuando esto nos lleve a una contradicción.

5.1.1. Superposición

El primer paso para resolver Nonogramas es determinar qué píxeles de una fila o columna se ven obligados a ser de color negro, como resultado de su descripción. Recordemos que una descripción de

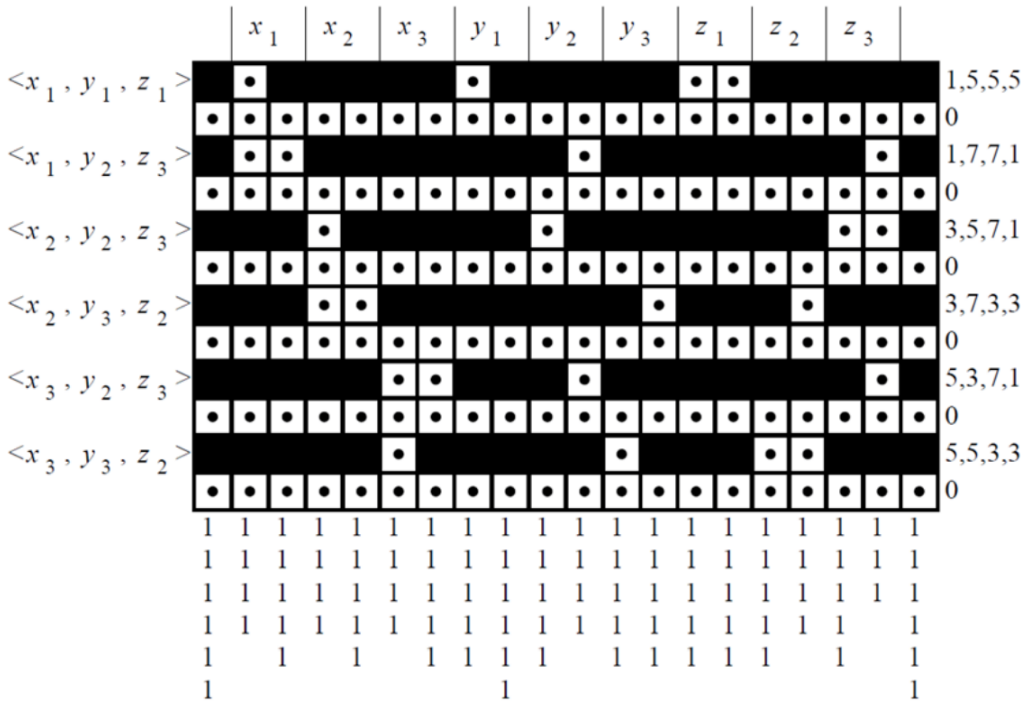


Figura 18: El Nonograma de ejemplo para la reducción de 3DM, tomado de (11) Section 3

fila o columna $d = d_1, d_2, ..., d_k$ cuando

$$k - 1 + \sum_{i=0}^{k} d_i \geq l \tag{1}$$

Cuando hay igualdad, significa que las secuencias de píxeles negros encajan exactamente en una fila o columna, y en ese caso inmediatamente tenemos la solución para toda la fila o columna. Sin embargo, cuando no hay igualdad, todavía es posible colorea varios píxeles de negro.

5.2. Pasos de resolución heurística

Para determinar a qué píxeles se les puede asignar un determinado valor, utilizamos rangos de ejecución. En una descripción de fila o columna, asignamos a cada entero de descripción un inicio de rango $R_{s,i}$ y un final de rango $R_{e,i}$. Es decir, el número de píxeles más bajo y más alto que puede contener píxeles negros de la ejecución correspondiente a la descripción entera dk. La idea de colorear píxeles por rangos de ejecución.

5.2.1. Configuración de rangos de ejecución

Dada una cierta descripción de fila o columna $d_1, d_2, ..., d_k$ para una fila o columna con píxeles $1, 2, ..., n$, establecemos el rango inicial comienza de la siguiente manera:

- El inicio del rango inicial $R_{s,q}$ de d_1 es 1.
- Para los enteros de descripción d_i con $i > 1$, el rango inicial comienza con $R_{s,i}$ vienen dados recursivamente por:

$$R_{s,i} = R_{s,i-1} + d_{i-1} + 1 \tag{2}$$

La idea detrás de esta asignación es que la posibilidad de que comience la ejecución di, que es la más cercana desde el principio, es cuando todas las ejecuciones anteriores se colocan tras otra lo

más cerca posible. Es decir, cuando cada dos ejecuciones posteriores están separadas por solo un píxel blanco. Establecemos el final del rango inicial de una manera similar. Esta vez, comenzamos a contar los posibles extremos de carrera que están más alejados de el principio:

- El extremo del rango inicial $R_{e,k}$ de d_k es n.
- Para los enteros de descripción d_i con $i > 1$, los extremos del rango inicial $R_{e,i}$ vienen dados recursivamente por:

$$R_{e,i-1} = R_{e,i} - d_i - 1 \tag{3}$$

5.2.2. Relleno de píxeles por rangos de ejecución

Si una determinada ejecución se superpone dentro de su propio rango, podemos usar esto para colorear los píxeles superpuestos de negro. Dado el inicio del rango $R_{s,i}$ y el final del rango $R_{e,i}$ para una descripción entera d_i, los píxeles superpuestos se pueden colorear como una serie de píxeles negros. Esta ejecución comienza en ese píxel que se encuentra exactamente d_i1 píxeles antes del final del rango, y termina en los píxeles que se encuentran $d_i - 1$ píxeles después del inicio del rango (se resta un píxel porque el rango comienza y termina también pueden ser parte de la carrera negra final):

- Primer píxel negro: $R_{e,i} - d_i - 1$
- Último píxel negro: $R_{e,i} + d_i - 1$

Algunos píxeles pueden no estar contenidos en ningún rango. Es decir, para un número de píxel p no hay un entero de descripción di para el cual $R_{s,i} \leq p \leq R_{e,i}$. Este es el caso cuando:

- $1 \leq p < R_{s,1}$ (el píxel se encuentra antes del primer rango),
- $R_{e,i} < p < R_{s,i+1}$ (el píxel se encuentra entre dos rangos),
- $R_{e,k} < p \leq n$ (el píxel se encuentra después del último rango).

Cuando para una descripción entera d_i tenemos $R_{s,i} + d_1 - 1 = R_{e,i}$, la ejecución está completa. Es decir, una ejecución de negro posterior consiste exactamente en la cantidad de píxeles negros que debería contener (y en los dos pasos anteriores, los píxeles negros ya se han llenado). Los píxeles individuales antes y después de la ejecución no pueden ser negros, ya que esto daría como resultado una ejecución negra más grande. Por lo tanto, estos píxeles pueden ser de color blanco.

5.2.3. Actualización de rangos de ejecución

Usar solo los rangos de ejecución iniciales no es muy inteligente, ya que esto solo usaría la información proporcionada por la descripción de la fila o columna en sí. Por lo tanto, tenemos que actualizar los rangos de ejecución de acuerdo con la nueva información que hemos obtenido en una determinada fila o columna.

1. Los espacios en blanco conocidos pueden garantizar que se necesite menos espacio para llenar los espacios negros y, por lo tanto, podemos determinar que los píxeles sean negros.
2. Los espacios negros conocidos pueden garantizar que una fila deba contener un espacio vacío, lo que determina que los píxeles sean blancos.
3. Busque píxeles blancos y negros consecutivos (no necesariamente en ese orden) y determine qué píxeles deben rellenarse.
4. Verificar si los píxeles negros conectados pueden caber dentro del espacio en blanco conocido
5. Fijación de píxeles para que sean negros o blancos, calculando las soluciones posibles más a la izquierda y más a la derecha.

5.3. Implementación de Algoritmo basado en fuerza bruta

El siguiente algoritmo fue implementado en c + +, y se basa en resolver un nonograma por fuerza bruta. Primero se llamará a la función "solve"que empezara llamando a general estrategia para saber si ay solucion, luego esta llamara al contenedor pair que permite recibir dos tipos de datos diferentes en este caso bol y long donde eliminará las estrategias que no ocupe, si pair en bool es negativo no hay solución, pero en cambio si este resulta ser positivo entonces procederá a resolverlo.

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>

#include <chrono>
using namespace std;
using std::cout; using std::endl;
using std::chrono::duration_cast;
using std::chrono::milliseconds;
using std::chrono::seconds;
using std::chrono::system_clock;

enum gridState_t {
    EMPTY,
    FILL,
    UNKNOWN
};

class nonogramGame {
public:
    nonogramGame(int h, int w) : height(h), width(w) {
        state = vector<vector<gridState_t>>(height, vector<gridState_t>(width, UNKNOWN
    ));
        rowStrategies = vector<vector<long long>>(height);
        columnStrategies = vector<vector<long long>>(width);
    }
    void solve();
    void printState();

    vector<vector<int>> rowConstraints;
    vector<vector<int>> columnConstraints;

private:
    void generateStrategies();
    void generateOneStrategy(vector<long long>& line, const vector<int>& constraint,
    const int size);
    bool generateRecursive(const int size, const vector<int>& sum, vector<long long>&
    stratList, long long& output, int start, int level);

    std::pair<bool, long long> eliminateStrategies();
    bool isValid(const vector<gridState_t>& states, long long strategy);

    bool isSolved();

    bool solveRecursive();

    int height, width;

    vector<vector<gridState_t>> state;
    vector<vector<long long>> rowStrategies;
    vector<vector<long long>> columnStrategies;
    vector<bool> rowSolved;
    vector<bool> columnSolved;
};

bool nonogramGame::isSolved() {
    for (int i = 0; i < height; ++i) {
        if (rowStrategies[i].size() > 1) {
            return false;
        }
```

```
61      }
62      return true;
63  }
64
65  void nonogramGame::printState() {
66      for (int i = 0; i < height; ++i) {
67          for (int j = 0; j < width; ++j) {
68              switch (state[i][j]) {
69              case EMPTY:
70                  cout << "_ ";
71                  break;
72              case FILL:
73                  cout << "0 ";
74                  break;
75              case UNKNOWN:
76                  cout << "1 ";
77                  break;
78              }
79          }
80          cout << endl;
81      }
82  }
83  /*solve empezara llamando a general estrategia para saber si ay solucion   luego llama
        al contenedor pair
84  que mermite recibir dos tipos de datos diferentes en este caso bol y long donde
        eliminara las estrategias q no ocupe
85  si pair en bool es negativo no ay solucion si es positivo lo resuelve*/
86  void nonogramGame::solve() {
87      cout << "Generando estrategias..." << endl;
88      generateStrategies();
89      cout << "Estrategias de poda..." << endl;
90      pair<bool, long long> pruneResults = eliminateStrategies();
91      if (!pruneResults.first) {
92          cout << "El rompecabezas es imposible." << endl;
93          return;
94      }
95
96      if (isSolved()) {
97          printState();
98      }
99      else {
100         cout << "Podada: " << pruneResults.second << " estrategias." << endl;
101         solveRecursive();
102     }
103 }
104
105 bool nonogramGame::solveRecursive() {
106     //Pick line with fewest strategies
107     bool vertical = false;
108     long long unsigned int strategies = UINT64_MAX;
109     int useLine = -1;
110     for (int i = 0; i < height; ++i) {
111         if (!rowSolved[i] && rowStrategies[i].size() < strategies && rowStrategies[i].
        size() > 1) {
112             strategies = rowStrategies[i].size();
113             useLine = i;
114         }
115     }
116     for (int i = 0; i < width; ++i) {
117         if (!columnSolved[i] && columnStrategies[i].size() < strategies &&
        columnStrategies[i].size() > 1) {
118             strategies = columnStrategies[i].size();
119             useLine = i;
120             vertical = true;
121         }
122     }
123     //Copy state
124     vector<vector<gridState_t>> origState(height, vector<gridState_t>(width));
125     vector<vector<long long>> origRowStrats(height);
126     vector<vector<long long>> origColStrats(width);
127     vector<bool> origRowSolved(height);
```

```
128     vector<bool> origColSolved(width);
129     for (int i = 0; i < height; ++i) {
130         for (int j = 0; j < width; ++j) {
131             origState[i][j] = state[i][j];
132         }
133     }
134     for (int i = 0; i < height; ++i) {
135         origRowStrats[i].reserve(rowStrategies[i].size());
136         for (long long j : rowStrategies[i]) {
137             origRowStrats[i].push_back(j);
138         }
139         origRowSolved[i] = rowSolved[i];
140     }
141     for (int i = 0; i < width; ++i) {
142         origColStrats[i].reserve(columnStrategies[i].size());
143         for (long long j : columnStrategies[i]) {
144             origColStrats[i].push_back(j);
145         }
146         origColSolved[i] = columnSolved[i];
147     }
148 
149     auto restore = [&]() {
150         for (int i = 0; i < height; ++i) {
151             for (int j = 0; j < width; ++j) {
152                 state[i][j] = origState[i][j];
153             }
154         }
155         for (int i = 0; i < height; ++i) {
156             rowStrategies[i].clear();
157             rowStrategies[i].reserve(origRowStrats[i].size());
158             for (long long j : origRowStrats[i]) {
159                 rowStrategies[i].push_back(j);
160             }
161             rowSolved[i] = origRowSolved[i];
162         }
163         for (int i = 0; i < width; ++i) {
164             columnStrategies[i].clear();
165             columnStrategies[i].reserve(origColStrats[i].size());
166             for (long long j : origColStrats[i]) {
167                 columnStrategies[i].push_back(j);
168             }
169             columnSolved[i] = origColSolved[i];
170         }
171     };
172 
173     if (!vertical) {
174         int count = 0;
175         //Test numbers
176         vector<pair<long long, int>> eliminated;
177         cout << " Usando fila " << useLine + 1 << endl;
178         for (long long strat : origRowStrats[useLine]) {
179             cout << "Probando la perspectiva de la estrategia" << count + 1 << "/" <<
    origRowStrats[useLine].size() << endl;
180             rowStrategies[useLine].clear();
181             rowStrategies[useLine].push_back(strat);
182             rowSolved[useLine] = true;
183             for (int i = 0; i < width; ++i) {
184                 state[useLine][i] = (strat & (1LLU << i)) != 0 ? FILL : EMPTY;
185             }
186             pair<bool, long long> eliminateResult = eliminateStrategies();
187             if (eliminateResult.first) {
188                 if (isSolved()) {
189                     printState();
190                     return true;
191                 }
192                 else {
193                     cout << "Strategy " << count << "/" << origRowStrats[useLine].size
    () << " prunes " << eliminateResult.second << " others." << endl;
194                     eliminated.push_back({ eliminateResult.second, count });
195                 }
196             }
```

```
197             else {
198                 cout << "This strategy cannot work" << endl;
199             }
200             restore();
201             count++;
202         }
203         sort(eliminated.begin(), eliminated.end(), [&](const pair<long long, int>& a,
    const pair<long long, int>& b) {
204             return a.first > b.first;
205             });
206 
207         count = 0;
208         for (auto& useStrats : eliminated) {
209             long long strat = origRowStrats[useLine][useStrats.second];
210             cout << " Usando fila  " << useLine + 1 << " Strategy: " << count + 1 << "
    /" << eliminated.size() << ": " << strat << endl;
211             rowStrategies[useLine].clear();
212             rowStrategies[useLine].push_back(strat);
213             rowSolved[useLine] = true;
214             for (int i = 0; i < width; ++i) {
215                 state[useLine][i] = (strat & (1LLU << i)) != 0 ? FILL : EMPTY;
216             }
217             eliminateStrategies();
218             if (solveRecursive()) {
219                 return true;
220             }
221             cout << "No further strategies possible, restoring..." << endl;
222             restore();
223             count++;
224         }
225     }
226     else {
227         int count = 0;
228         vector<pair<long long, int>> eliminated;
229         cout << "Using column " << useLine + 1 << endl;
230         for (long long strat : origColStrats[useLine]) {
231             cout << "Testing prospect of strategy " << count << "/" << origColStrats[
    useLine].size() << endl;
232             columnStrategies[useLine].clear();
233             columnStrategies[useLine].push_back(strat);
234             columnSolved[useLine] = true;
235             for (int i = 0; i < height; ++i) {
236                 state[i][useLine] = (strat & (1LLU << i)) != 0 ? FILL : EMPTY;
237             }
238             pair<bool, long long> eliminateResult = eliminateStrategies();
239             if (eliminateResult.first) {
240                 if (isSolved()) {
241                     printState();
242                     return true;
243                 }
244                 else {
245                     cout << "estrategia " << count << "/" << origColStrats[useLine].
    size() << " prunes " << eliminateResult.second << " others." << endl;
246                     eliminated.push_back({ eliminateResult.second, count });
247                 }
248             }
249             else {
250                 cout << "Esta estrategia no puede funcionar" << endl;
251             }
252             restore();
253             count++;
254         }
255         sort(eliminated.begin(), eliminated.end(), [&](const pair<long long, int>& a,
    const pair<long long, int>& b) {
256             return a.first > b.first;
257             });
258 
259         count = 0;
260         for (auto& useStrats : eliminated) {
261             long long strat = origColStrats[useLine][useStrats.second];
```

```
262             cout << "usando la columna" << useLine + 1 << " estrategia: " << count <<
    "/" << eliminated.size() << ": " << strat << endl;
263             columnSolved[useLine] = true;
264             columnStrategies[useLine].clear();
265             columnStrategies[useLine].push_back(strat);
266             for (int i = 0; i < height; ++i) {
267                 state[i][useLine] = (strat & (1LLU << i)) != 0 ? FILL : EMPTY;
268             }
269             eliminateStrategies();
270             if (solveRecursive()) {
271                 return true;
272             }
273             cout << "No hay m s estrategias posibles, restaurando..." << endl;
274             restore();
275             count++;
276         }
277     }
278     return false;
279 }
280
281 std::pair<bool, long long> nonogramGame::eliminateStrategies() {
282     rowSolved = vector<bool>(height, false);
283     columnSolved = vector<bool>(width, false);
284
285     int eliminated = 0;
286     long long totalEliminated = 0;
287     do {
288         eliminated = 0;
289         for (int i = 0; i < height; ++i) {
290             if (rowSolved[i]) { continue; }
291             vector<long long> strats;
292             strats.reserve(rowStrategies[i].size());
293             vector<gridState_t> currentState;
294             currentState.reserve(width);
295             for (int j = 0; j < width; ++j) {
296                 currentState.push_back(state[i][j]);
297             }
298             long long mustFill = -1, mustEmpty = -1;
299             for (long long it : rowStrategies[i]) {
300                 if (!isValid(currentState, it)) {
301                     ++eliminated;
302                 }
303                 else {
304                     strats.push_back(it);
305                     mustFill &= it;
306                     mustEmpty &= ~(it);
307                     ++it;
308                 }
309             }
310             if (strats.empty()) {
311                 return { false, 0 };
312             }
313             if (strats.size() < rowStrategies[i].size()) {
314                 rowStrategies[i].resize(strats.size());
315                 for (long long unsigned int j = 0; j < strats.size(); ++j) {
316                     rowStrategies[i][j] = strats[j];
317                 }
318             }
319             for (int j = 0; j < width; ++j) {
320                 if (mustFill & (1LLU << j)) {
321                     state[i][j] = FILL;
322                 }
323                 else if (mustEmpty & (1LLU << j)) {
324                     state[i][j] = EMPTY;
325                 }
326             }
327             if (rowStrategies.size() == 1) {
328                 rowSolved[i] = true;
329             }
330         }
331         for (int j = 0; j < width; ++j) {
```

```
            if (columnSolved[j]) { continue; }

            vector<long long> strats;
            strats.reserve(columnStrategies[j].size());

            vector<gridState_t> currentState;
            currentState.reserve(width);
            for (int i = 0; i < height; ++i) {
                currentState.push_back(state[i][j]);
            }
            long long mustFill = -1, mustEmpty = -1;
            for (long long it : columnStrategies[j]) {
                if (!isValid(currentState, it)) {
                    ++eliminated;
                }
                else {
                    strats.push_back(it);
                    mustFill &= it;
                    mustEmpty &= ~(it);
                    ++it;
                }
            }
            if (strats.empty()) {
                return { false, 0 };
            }
            if (strats.size() < columnStrategies[j].size()) {
                columnStrategies[j].resize(strats.size());
                for (long long unsigned int i = 0; i < strats.size(); ++i) {
                    columnStrategies[j][i] = strats[i];
                }
            }
            for (int i = 0; i < height; ++i) {
                if (mustFill & (1LLU << i)) {
                    state[i][j] = FILL;
                }
                else if (mustEmpty & (1LLU << i)) {
                    state[i][j] = EMPTY;
                }
            }
            if (columnStrategies[j].size() == 1) {
                columnSolved[j] = true;
            }
        }
        totalEliminated += eliminated;
    } while (eliminated);
    return { true, totalEliminated };
}

bool nonogramGame::isValid(const vector<gridState_t>& states, long long strategy) {
    size_t size = states.size();
    for (size_t i = 0; i < size; ++i) {
        bool bit = ((strategy & (1LLU << i)) != 0);
        if ((bit && states[i] == EMPTY) || (!bit && states[i] == FILL)) {
            return false;
        }
    }
    return true;
}
/*
generateStrategies llama a generateOneStrategy las n cantidad de veces tanto en las
    filas
como en las columnas para saber si ay solucion
*/
void nonogramGame::generateStrategies() {
    for (int i = 0; i < height; ++i) {
        generateOneStrategy(rowStrategies[i], rowConstraints[i], width);
    }
    for (int i = 0; i < width; ++i) {
        generateOneStrategy(columnStrategies[i], columnConstraints[i], height);
    }
}
```

```
402
403 void nonogramGame::generateOneStrategy(vector<long long>& line, const vector<int>&
        constraint, const int size) {
404     bool allZeros = true;
405     for (int c : constraint) {
406         if (c) { allZeros = false; break; }
407     }
408     if (allZeros) {
409         line.push_back(0);
410         return;
411     }
412
413     vector<int> sum(constraint.size() + 1, 0);
414     sum[constraint.size() - 1] = constraint.back() + 1;
415     for (int i = constraint.size() - 2; i >= 0; --i) {
416         sum[i] = sum[i + 1] + constraint[i] + 1;
417     }
418     long long output = 0;
419     if (!generateRecursive(size, sum, line, output, 0, 0)) {
420         cout << "Impossible!" << endl;
421     }
422 }
423
424 bool nonogramGame::generateRecursive(const int size, const vector<int>& sum, vector<
        long long>& stratList, long long& output, int start, int level) {
425     if (size_t(level) == sum.size() - 1) {
426         stratList.push_back(output);
427         return true;
428     }
429
430     int end = size - sum[level] + 1;
431     if (end < start) { return false; }
432     int num = sum[level] - sum[level + 1] - 1;
433     long long oldOutput = output;
434     long long mask = (1LLU << num) - 1;
435     for (int i = start; i <= end; ++i) {
436         output = oldOutput | (mask << i);
437         if (!generateRecursive(size, sum, stratList, output, i + num + 1, level + 1))
        {
438             return false;
439         }
440     }
441     output = oldOutput;
442     return true;
443 }
444
445 int main() {
446     unsigned int height, width;
447     do {
448         cout << " Ingrese la longitud y el ancho : ";
449         cin >> height >> width;
450         cin.ignore(256, '\n');
451     } while (height > 64 || width > 64 || height == 0 || width == 0);
452
453     nonogramGame game(height, width);
454
455     cout << " Restricciones de fila de entrada : " << endl;
456     for (int i = 0; i < height; ++i) {
457         string constraintString;
458         getline(cin, constraintString);
459
460         vector<int> constraint;
461         stringstream ss;
462         ss << constraintString;
463         while (!ss.eof()) {
464             int temp;
465             ss >> temp;
466             constraint.push_back(temp);
467         }
468         game.rowConstraints.push_back(constraint);
469     }
```

```
470     cout << "Restricciones de la columna de entrada:" << endl;
471     for (int i = 0; i < width; ++i) {
472         string constraintString;
473         getline(cin, constraintString);
474 
475         vector<int> constraint;
476         stringstream ss;
477         ss << constraintString;
478         while (!ss.eof()) {
479             int temp;
480             ss >> temp;
481             constraint.push_back(temp);
482         }
483         game.columnConstraints.push_back(constraint);
484     }
485     auto millisec_start_epoch = duration_cast<milliseconds>(system_clock::now().
    time_since_epoch()).count();
486     game.solve();
487     auto millisec_end_epoch = duration_cast<milliseconds>(system_clock::now().
    time_since_epoch()).count();
488     cout<<"time :"<<millisec_end_epoch - millisec_start_epoch << endl;
489 }
```

Listing 1: Algoritmo Fuerza Bruta

En la resolución del algoritmo por fuerza bruta lo que hacemos es usar el tamaño n*n que recibimos del usuario, para definir los recorridos que hacemos y de esa misma cantidad llamaremos a la función generateOneStrategy que recorrerá esas n cantidad de veces por fila y columna para poder saber si existe una solución para este problema, finalmente llamamos a la función generateRecursive para saber si el problema puede generar una solución recursivamente, a este algoritmo lo llamamos que trabaja por fuerza bruta dado a que su complejidad es mayor porque su complejidad aumenta por las constantes llamadas a los bucles y recursividad excesivos siendo el mejor de los casos lo que podríamos sacar si usáramos los teoremas anteriores perteneciente a la sección El problema de la unicidad.

5.4. Algoritmo aproximado

El siguiente algoritmo es nuestra presentacion de solución por aproximación, para el cual usamos recursividad que nos permite identificar entradas que posiblemente no sean Nonogramas, y si en este proceso encontramos una solución a la entrada se termina la recursividad, para devolver la solución creamos dos estructuras de datos para nuestra conveniencia, la clase nonogram para almacenar los vectores de vectores que son nuestras entradas, y la struct N creada dentro de la clase nonogram para almacenar y trabajar con los vectores que contienen nuestras entradas, asimismo la clase nonograma utiliza funciones de struct N para dar con la solución, para la cual tenemos un enum que almacenan los caracteres para imprimir, en cualquiera de los 3 casos: incertidumbre, positivo y negativo.

```
1  #include <iostream>
2  #include <vector>
3  #include <bitset>
4  #include <numeric>
5  #include <fstream>
6  #include <chrono>
7  #include <sstream>
8 
9  template<unsigned int _N, unsigned int _G> class Nonogram {
10   enum class ng_val : char {X='#',B='.',V='?'};
11 
12   template<unsigned int _NG> struct N {
13     N() {}
14     N(std::vector<int> ni,const int l) : X{},B{},Tx{},Tb{},ng(ni),En{},gNG(l){}
15     std::bitset<_NG> X, B, T, Tx, Tb;
16     std::vector<int> ng;
17     int En, gNG;
18 
19     void fn (const int n,const int i,const int g,const int e,const int l){
20       if (fe(g,l,false) and fe(g+l,e,true)){
21           if ((n+1) < ng.size()) {if (fe(g+e+l,1,false))
22              fn(n+1,i-e-1,g+e+l+1,ng[n+1],0);}
```

```
      else {
        if (fe(g+e+1,gNG-(g+e+1),false)){Tb &= T.flip(); Tx &= T.flip(); ++En;}
      }}
      if (l<=gNG-g-i-1) fn(n,i,g,e,l+1);
    }
    void fi (const int n,const bool g) {X.set(n,g); B.set(n, not g);}
    ng_val fg (const int n) const{return (X.test(n))? ng_val::X : (B.test(n))?
    ng_val::B : ng_val::V;}
    inline bool fe (const int n,const int i, const bool g){
     for (int e = n;e<n+i;++e) if ((g and fg(e)==ng_val::B)
     or (!g and fg(e)==ng_val::X))
      return false; else T[e] = g;
      return true;
    }
    int  fl (){
      if (En == 1) return 1;
      Tx.set(); Tb.set(); En=0;
      fn(0,std::accumulate(ng.cbegin(),ng.cend(),0)+ng.size()-1,0,ng[0],0);
      return En;
    }}; // end of N

 std::vector<N<_G>> ng;
 std::vector<N<_N>> gn;
 int En, zN, zG;


void setCell(unsigned int n, unsigned int i, bool g){ng[n].fi(i,g);
gn[i].fi(n,g);}
public:
  Nonogram(const std::vector<std::vector<int>>& n,
  const std::vector<std::vector<int>>& i,
  const std::vector<std::string>& g = {}) :
  ng{}, gn{}, En{}, zN(n.size()), zG(i.size()) {
    for (int n=0; n<zG; n++) gn.push_back(N<_N>(i[n],zN));
    for (int i=0; i<zN; i++) {
      ng.push_back(N<_G>(n[i],zG));
      if (i < g.size()) for(int e=0; e<zG or e<g[i].size(); e++)
        if (g[i][e]=='#') setCell(i,e,true);
    }}
  bool solve(){
    int i{}, g{};
    for (int l = 0; l<zN; l++) {
      if ((g = ng[l].fl()) == 0) return false; else i+=g;
      for (int i = 0; i<zG; i++) if (ng[l].Tx[i] != ng[l].Tb[i])
      setCell (l,i,ng[l].Tx[i]);
    }
    for (int l = 0; l<zG; l++) {
      if ((g = gn[l].fl()) == 0) return false; else i+=g;
      for (int i = 0; i<zN; i++) if (gn[l].Tx[i] != gn[l].Tb[i])
      setCell (i,l,gn[l].Tx[i]);
    }
    if (i == En)    return false; else En = i;
    if (i == zN+zG) return true;  else return solve();
  }
  const std::string toStr() const {
    std::ostringstream n;
    for (int i = 0; i<zN; i++){for (int g = 0; g<zG; g++){n <<
    static_cast<char>(ng[i].fg(g));}n<<std::endl;}
    return n.str();
  }
};

int main(){
  std::ifstream n ("nono.txt");
  if (!n) {
    std::cerr << "Unable to open nono.txt.\n";
    exit(EXIT_FAILURE);
  }
  std::string i;
  getline(n,i);
  std::istringstream g(i);
```

```
94    std::string e;
95    std::vector<std::vector<int>> N;
96      while (g >> e) {
97        std::vector<int> G;
98        for (char l : e) G.push_back((int)l-64);
99        N.push_back(G);
100     }
101   getline(n,i);
102   std::istringstream gy(i);
103   std::vector<std::vector<int>> G;
104     while (gy >> e) {
105       std::vector<int> N;
106       for (char l : e) N.push_back((int)l-64);
107       G.push_back(N);
108     }
109   Nonogram<32,32> myN(N,G);
110 }
```

Listing 2: Aproximado

6. Conclusiones

Después de introducir los Nonogramas y algunas nociones básicas de la teoría de grafos y la teoría de la complejidad, presentamos el teorema principal sobre la complejidad de la solucionabilidad de los Nonogramas. La demostración, que consiste en reducir el llamado Problema NCL Acotado Plenamente a esta solucionabilidad. Después se discute la complejidad del problema de la unicidad de soluciones. Ambos problemas (solubilidad y unicidad) resultan ser NP-completos. Tratamos con una estrategia básica para resolver nonogramas.

Referencias

[1] K.J. Batenburg and W.A. Kosters A *Discrete Tomography Approach to Japanese Puzzles, preprint, 2005.*

[2] J. Kleinberg, E. Tardos, *Algorithm Design, Pearson Education Inc., 2005.*

[3] M. Mesbahi, M. Egerstedt, *Graph Theoretic Methods in multiagent Networks, Princeton University Press, 2010.*

[4] R.A. Hearn, G*ames, Puzzles and Computation, Massachusetts Institute of Technology, 2006.*

[5] R.A. Hearn, E.D. Demaine, *The Nondeterministic Constraint Logic Model of Computation: Reductions and Applications, ICALP, Lecture Notes in Computer Science volume 2380, 401-413, Springer, 2002.*

[6] K.J. Batenburg, S. Henstra, W.A. Kosters, W.J. Palenstijn, *Constructing Simple Nonograms of Varying Difficulty, Corvinus University of Budapest, Department of Mathematics, Pure Mathematics and Applications, 20(1), 1-15, 2009.*

[7] C.H. Yu, H.L. Lee, L.H. Chen, *An efficient algorithm for solving nonograms, Springer Science+Business Media, Springer Verlag, Applied Intelligence 35(1): 18-31, 2011*

[8] D.S. Johnson, *"A Catalog of Complexity Classes", Algorithms and Complexity. Ed. J. Van Leeuwen, Amsterdam: Elsevier Science Publishers, 1992. 67-162*

[9] J. Barwise, J. Etchemendy, *Language, Proof and Logic, CSLI Publications, 2003.*

[10] S.R. Dunbarr, *Topics in Probability Theory and Stochastic Processes.*

[11] N. Ueda, T. Nagao, *NP-completeness Results for NONOGRAM via Parsimonious Reductions*

[12] W.H. Hesselink, *Dictaat Talen en Automaten, Rijksuniversiteit Groningen, 2012.*

[13] S. Salcedo-Sanz, E.G. Ortiz-Garcia et al., *Solving Japanese Puzzles with Heuristics, IEEE Symposium on Computational Intelligence and Games, 2007, 224-231, CIG, 2007.*

[14] K.J. Batenburg, W.A. Kosters, *Solving Nonograms by combining relaxations, Pattern Recognition, Volume 42, Issue 8, 16721683, Elsevier, 2009.*

[15] J. N. van Rijn, *Playing Games: The complexity of Klondike, Mahjong, Nonograms and Animal Chess, Master's thesis, Universiteit Leiden, 2012.*