Diseño y Análisis de Algoritmos

Brian Ameht Inclán Quesada
Davier Sanchez Bello
Maykol Luis Martínez Rodríguez

Septiembre 2024

Contents

1 Videojuego

Se quiere crear un nivel para un videojuego. El nivel consiste en n habitaciones dispuestas en un círculo. Las habitaciones están numeradas del 1 al n. Cada habitación contiene exactamente una salida: completar la j-ésima habitación te permite pasar a la $(j+1)$-ésima habitación (y completar la n-ésima habitación te permite pasar a la 1-era habitación).

Se te da la descripción del multiconjunto de n cofres: el i-ésimo cofre tiene un valor de tesoro c_i. Cada cofre puede ser de uno de dos tipos:

- **Cofre normal**: cuando un jugador entra en una habitación con este cofre, recoge el tesoro y procede a la siguiente habitación.
- **Cofre trampa**: cuando un jugador entra en una habitación con este cofre, el cofre se lo come vivo y él pierde.

El jugador comienza en una habitación aleatoria, con cada habitación teniendo la misma probabilidad de ser elegida. Las ganancias del jugador son iguales al valor total de los tesoros que haya recogido antes de perder.

1.1 Definición del Problema

Se permite elegir el orden en que los cofres se colocan en las habitaciones. Para cada k de 1 a n, se debe colocar los cofres de tal manera que:

- Cada habitación contenga exactamente un cofre.
- Exactamente k cofres sean trampas.
- El valor esperado de las ganancias del jugador sea el mínimo posible.

El valor esperado se puede ver en la forma $\frac{P}{Q}$ donde P y Q son enteros no negativos y $Q \neq 0$.

1.2 Valor Esperado

El problema se trata de minimizar el **valor esperado** de las ganancias del jugador, dado que comienza en una habitación aleatoria. El valor esperado de las ganancias se puede definir como:

$$E = \frac{\text{suma total de ganancias sobre todas las posiciones}}{n}$$

Donde n es el número total de habitaciones (o cofres). Como buscamos minimizar este valor esperado, es equivalente a **minimizar la suma total de las ganancias**, ya que n es constante. **Por lo tanto, reformulamos el problema como la minimización de la suma total de las ganancias**

1.3 Rotación y Distribución de Cofres

Para simplificar el análisis, **rotamos las habitaciones** de tal forma que la última habitación siempre contenga una trampa. Esto no cambia la naturaleza del problema, pero facilita la distribución de cofres regulares y trampas en el análisis.

Si tenemos k trampas, las habitaciones se dividen en k grupos, donde cada grupo contiene cnt_i cofres regulares seguidos de una trampa. Los cnt_i satisfacen la restricción:

$$\text{cnt}_i \geq 0 \quad \text{y} \quad \sum_{i=1}^{k} \text{cnt}_i = n - k$$

Esto significa que los $n - k$ cofres regulares se distribuyen en los k intervalos entre las trampas.

1.3.1 Ejemplo

Supongamos que tenemos $n = 10$ habitaciones y $k = 3$ trampas. El jugador comienza en una habitación aleatoria, y las trampas están distribuidas. Los valores de los cofres son:

$$[100, 200, \textbf{Trampa}, \textbf{Trampa}, 50, 300, \textbf{Trampa}, 500, 400, \textbf{Trampa}]$$

Distribución de intervalos:

$$\underbrace{100, 200}_{\text{cnt}_1=2} \quad \underbrace{\text{Trampa, Trampa}}_{\text{cnt}_2=0} \quad \underbrace{50, 300}_{\text{cnt}_3=2} \quad \underbrace{500, 400}_{\text{cnt}_4=2}$$

Respetando:

$$\sum_{i=1}^{4} \text{cnt}_i = 6 \quad (\text{ya que } n - k = 10 - 4 = 6)$$

1.4 Coeficientes y Secuencias de Ganancias

El coeficiente de un cofre c es el número de veces que el jugador puede recoger ese cofre desde las distintas posiciones iniciales de un intervalo antes de caer en una trampa. En un intervalo con cnt_i cofres regulares, el último cofre puede ser recogido cnt_i veces, el penúltimo $\text{cnt}_i - 1$ veces, y así sucesivamente, esto debido a la naturaleza del juego de avanzar a la siguiente habitacion si no me encuentro un cofre Trampa.

Cada cofre tiene un coeficiente que refleja cuántas veces puede ser recogido dependiendo de su proximidad a la trampa. Los cofres más cercanos a la siguiente trampa tienen coeficientes más altos, ya que son alcanzables desde más posiciones iniciales. Esto significa que, cuanto más cerca esté el cofre de la siguiente trampa, mayor es su contribución potencial a las ganancias totales del jugador.

La suma total de ganancias en un intervalo es:

$$\sum_{j=1}^{\text{cnt}_i} j \cdot v_j$$

Donde v_j es el valor del j-ésimo cofre del intervalo. Cuanto más cerca esté el cofre de la siguiente trampa, mayor será su contribución potencial a las ganancias.

Esto se debe a que el número de veces que un cofre puede ser recogido depende directamente de su posición en relación con la siguiente trampa: los cofres más cercanos a la siguiente trampa pueden ser recogidos desde más posiciones iniciales. Al multiplicar el valor de cada cofre por su coeficiente, estamos sumando el impacto real de cada cofre en las ganancias totales, reflejando cómo su posición y valor combinan para maximizar o minimizar las ganancias en el intervalo.

1.5 Balanceo de los intervalos

Cuando tenemos dos intervalos de cofres con diferentes tamaños, podemos **mejorar** la distribución de las ganancias moviendo cofres del intervalo más grande al más pequeño. Esto ayuda a minimizar el total de ganancias del jugador, ya que los coeficientes de los cofres cambian dependiendo del intervalo al que pertenecen.

El concepto clave es que los **coeficientes** afectan la suma total de ganancias: un cofre en el intervalo más grande tiene un coeficiente más alto (y por lo tanto contribuye más a la suma total de ganancias), mientras que si movemos ese cofre al intervalo más pequeño, su coeficiente será más bajo, lo que reducirá las ganancias totales.

1.5.1 Demostración del balanceo

1. **Identificar los intervalos más grande y más pequeño**:
   - Supongamos que el intervalo más grande tiene y cofres, y el más pequeño tiene x cofres, donde $x \leq y - 2$.
   - Queremos reducir esta diferencia moviendo un cofre del intervalo más grande al más pequeño.
2. **Secuencias de coeficientes**:
   - En el intervalo más pequeño (x), los cofres tienen los coeficientes $[1, 2, \ldots, x]$.
   - En el intervalo más grande (y), los cofres tienen los coeficientes $[1, 2, \ldots, y]$.
   - La contribución a las ganancias está dada por:

     $$\sum_{j=1}^{x} j \cdot v_j \quad \text{para el intervalo pequeño y} \quad \sum_{j=1}^{y} j \cdot v_j \quad \text{para el grande.}$$
3. **Mover el cofre**:
   - Antes del movimiento, el último cofre del intervalo grande tiene un coeficiente y, y el último cofre del intervalo pequeño tiene un coeficiente x.
   - Si movemos el cofre del intervalo grande al pequeño, el coeficiente de ese cofre baja de y a $x + 1$. Esto significa que el cofre ahora se recoge menos veces, **reduciendo la suma total de ganancias**.
4. **Justificación matemática**:
   - La secuencia de coeficientes en el intervalo más pequeño, después de mover el cofre, será $[1, 2, \ldots, x, x + 1]$.
   - La secuencia en el intervalo más grande será $[1, 2, \ldots, y - 1]$.
   - Al reducir el coeficiente más alto de y a $x + 1$, hemos logrado una reducción efectiva en la suma total de ganancias, ya que el cofre valioso del intervalo más grande se recoge menos veces. Esto **minimiza** el impacto de ese cofre en las ganancias totales.
   - **Demostración**

     Cuando movemos un cofre del intervalo grande al intervalo pequeño, la contribución de ese cofre a la suma total de ganancias cambia, ya que su **coeficiente** cambia.

- **Antes del movimiento**: El cofre v_j está en el intervalo grande, donde su coeficiente es y, por lo que su contribución a las ganancias es $v_j \times y$.
- **Después del movimiento**: El cofre se mueve al intervalo pequeño, donde su nuevo coeficiente es $x + 1$, lo que significa que su contribución ahora es $v_j \times (x + 1)$.

La **reducción en las ganancias** es la diferencia entre la contribución original y la nueva contribución:

$$\Delta\text{Ganancia} = v_j \times (y - (x + 1))$$

Donde:

- v_j es el valor del cofre ($v_j \geq 0$).
- y es el coeficiente en el intervalo grande ($x \leq y - 2$).
- $x + 1$ es el nuevo coeficiente en el intervalo pequeño ($x > 0$).

Este resultado muestra que al mover un cofre del intervalo más grande al más pequeño, estamos disminuyendo su coeficiente, lo que significa que el cofre se recogerá menos veces, y por lo tanto, su contribución total a las ganancias se reducirá. La fórmula $v_j \times (y-(x+1))$ refleja precisamente cuánto disminuyen las ganancias cuando el cofre es movido.

Si la diferencia entre y y $x+1$ es grande, la reducción en las ganancias será mayor, por lo cual mover cofres de intervalos grandes a pequeños es una estrategia efectiva para minimizar las ganancias totales.

1.6 Asignación óptima de cofres regulares

Después de balancear los intervalos, el siguiente paso es asignar los cofres regulares de manera **óptima**. El objetivo es minimizar las ganancias esperadas del jugador.

1.6.1 Proceso de asignación

1. **Unión de coeficientes**: Después de definir todos los cnt_i (los tamaños de cada intervalo), debemos unir todas las secuencias de coeficientes desde los diferentes intervalos. Esto nos da una secuencia que abarca todas las posiciones posibles donde los cofres pueden ser asignados. Los coeficientes de cada intervalo forman secuencias $[1, 2, \ldots, \text{cnt}_i]$ para

cada i. Una vez que unimos estas secuencias de coeficientes, obtenemos la secuencia global de coeficientes:

$$\bigcup_{i=1}^{n-k} [1, 2, \ldots, \text{cnt}_i].$$

2. **Ordenar los cofres por valor**: Para minimizar las ganancias, es importante colocar los cofres más valiosos en las posiciones con los coeficientes más bajos, es decir, donde el jugador tendrá menos oportunidades de recogerlos. Esto se hace ordenando los valores de los cofres en **orden no creciente** (de mayor a menor). La razón detrás de esto es simple: si los cofres más valiosos se recogen menos veces, contribuyen menos a las ganancias totales.

3. **Asignar los valores a los coeficientes**: Después de unir y ordenar los coeficientes de los intervalos, se asignan los cofres en función de esta secuencia. Los cofres con mayor valor se colocan en los coeficientes más bajos. Esto garantiza que los cofres más valiosos se recojan menos veces y, por lo tanto, **minimicen las ganancias**.

4. **Cofres con coeficiente 0**: Los k cofres que van en las trampas **no contribuyen a las ganancias** del jugador porque el jugador cae en la trampa antes de recogerlos. Por lo tanto, asignamos **coeficiente 0** a los cofres que se colocan en las posiciones donde hay trampas. **Estos cofres son los primeros k cofres de la lista de cofres ordenada**, ya que, para minimizar las ganancias del jugador, es crucial asegurarnos de que los cofres más valiosos no sean recogidos. Colocarlos en las trampas con coeficiente 0 asegura que el jugador no obtenga ninguna ganancia de ellos.

1.6.2 Porqué es óptimo?

El **objetivo** es minimizar la ganancia esperada del jugador, lo que implica que **los cofres más valiosos** deben tener la **menor contribución posible**. Colocar estos cofres en posiciones donde tienen **coeficiente 0** (es decir, en las trampas) garantiza que no generen ninguna ganancia, ya que el jugador nunca los recoge.

Si distribuimos los cofres de manera diferente (por ejemplo, asignando un cofre valioso a una posición con un coeficiente alto), se podría **mejorar** esa disposición moviendo cofres más valiosos a posiciones con coeficientes más bajos o asignándolos a las trampas (coeficiente 0), lo que automáticamente

reduciría las ganancias totales. **Ordenar los cofres de mayor a menor valor** y asignarlos a los coeficientes más bajos o a las trampas (coeficiente 0) es la manera **óptima** de proceder para minimizar las ganancias esperadas.

Por lo tanto, asignar los **coeficientes más bajos** a los cofres menos valiosos y el **coeficiente 0** a los cofres más valiosos (colocados en las trampas) es la **estrategia óptima** para minimizar las ganancias esperadas.

1.7 Complejidad $O(n^2)$

El cálculo de las ganancias del jugador, basado en la división en intervalos y el uso de trampas, tiene una complejidad de $O(n^2)$. Esta complejidad surge debido a cómo se asignan los cofres a intervalos y trampas, multiplicando sus valores por los coeficientes correspondientes. A continuación, se explica el porqué.

1.7.1 Explicación de la solución:

1. **Ordenación de los cofres por valor**:
   - Primero, se ordenan todos los cofres en orden descendente para asegurar que los cofres más valiosos se coloquen en las posiciones menos favorables (donde el jugador tiene menos oportunidades de recogerlos).
   - Esta ordenación toma $O(n \log n)$, pero este paso es menor en comparación con el cálculo posterior de las ganancias.
2. **División entre k y redondeo hacia abajo $\left\lfloor \frac{i}{k} \right\rfloor$**:
   - Para cada valor de k (el número de trampas), dividimos los cofres en k intervalos. Cada intervalo tendrá un conjunto de coeficientes que indican cuántas veces el jugador puede recoger los cofres antes de caer en una trampa.
   - El redondeo $\left\lfloor \frac{i}{k} \right\rfloor$ evidencia cómo los cofres se dividen en estos intervalos. El índice i se divide entre k, lo que asigna un coeficiente a cada cofre.
3. **Asignación de coeficientes**:
   - Para los **primeros k cofres** (es decir, $i < k$), su coeficiente es 0, porque están en las posiciones que ocupan las trampas. Esto significa que el jugador no puede recoger estos cofres, ya que cae en una trampa antes de llegar a ellos. Esta parte del algoritmo

evidencia que los cofres de mayor valor se colocan en posiciones de trampas.

- Los **siguientes k cofres** reciben un coeficiente de 1, lo que indica que estos cofres pueden ser recogidos una vez antes de que el jugador caiga en una trampa. Este patrón sigue para el resto de los cofres: cada conjunto de k cofres recibe un coeficiente incrementado en 1, lo que refleja cuantas veces pueden ser recogidos antes de que el jugador caiga en la siguiente trampa.
- En términos simples, los cofres más valiosos (los primeros en la lista ordenada) reciben coeficientes más bajos simulanodo asi que en la ubicacion en los intervalos estan mas alejados de la siguiente trampa, y a medida que los coeficientes aumentan, quiere decir que en algun intervalo el cofre que le corresponde está mas pegado a la siguiente trampa pr lo cual se recoje mas veces en ese intervalo pero como son cofres menos valiosos contribuye menos que otro cofre con mayor valor.

4. **Cálculo de las ganancias**:

   - Para cada valor de k, multiplicamos el valor de cada cofre por su coeficiente $\left\lfloor \frac{i}{k} \right\rfloor$, sumando los resultados.
   - El **redondeo hacia abajo** ocurre porque los intervalos se dividen en tamaños de k. Los cofres dentro del primer intervalo no se recogen antes de caer en una trampa, por lo que sus coeficientes son 0. Los siguientes intervalos tienen coeficientes 1, luego 2, y así sucesivamente. El redondeo refleja esta estructura, asegurando que los cofres en las primeras posiciones (que corresponden a las trampas) no contribuyan a las ganancias.

5. **Repetición del cálculo para cada k**:

   - Este proceso se repite para cada k, desde 1 hasta n. Cada vez que calculamos para un k, recorremos todos los cofres y multiplicamos sus valores por $\left\lfloor \frac{i}{k} \right\rfloor$, lo que toma $O(n)$ tiempo para cada k.

1.7.2 Complejidad total:

Como el cálculo de las ganancias para cada k toma $O(n)$ debido a que iteramos sobre todos los cofres, y repetimos este proceso n veces (una vez por cada k), la complejidad total es:

$$O(n) + O(n) + O(n) + \cdots + O(n) \text{ (repetido } n \text{ veces)} = O(n^2)$$

1.7.3 Resumen:

- **Para cada valor de** k, se multiplican los valores de los cofres por $\left\lfloor \frac{i}{k} \right\rfloor$, reflejando cuántos intervalos completos se han recorrido antes de llegar a ese cofre.
- **Los primeros** k **cofres** reciben un coeficiente de 0, ya que caen en posiciones de trampas, lo que asegura que los cofres de mayor valor no contribuyen a las ganancias.
- **Este cálculo se repite** n **veces**, lo que nos da una complejidad total de $O(n^2)$.

1.8 Optimización

1.8.1 Optimización de la complejidad $O(n^2)$

Hasta este punto, hemos visto que el cálculo de las ganancias para cada valor de k nos lleva a una complejidad de $O(n^2)$. Esto ocurre porque, para cada k, estamos dividiendo los cofres en intervalos, sumando sus valores y repitiendo este proceso n veces. Para cada cofre, calculamos su contribución multiplicando su valor por el coeficiente $\left\lfloor \frac{i}{k} \right\rfloor$, lo que genera un alto costo computacional.

Sin embargo, podemos **optimizar** este proceso sin cambiar la idea básica del algoritmo de ordenación de cofres. La clave para mejorar la eficiencia está en **reducir el costo del cálculo de las sumas de los cofres en los intervalos** mediante el uso de **sumas prefijas**.

1.8.2 PREFIX SUMS

Para reducir la complejidad del algoritmo, agregamos una fase de **precomputo en** $O(n)$ mediante el uso de **sumas prefijas** o **prefix sums**. Esto es especialmente útil cuando necesitamos realizar múltiples consultas sobre la suma de intervalos en una secuencia de datos, como ocurre en nuestro problema.

1.8.3 ¿Qué hacen las sumas prefijas?

Las sumas prefijas nos permiten calcular la suma de cualquier intervalo de elementos en **tiempo constante** $O(1)$, después de un preprocesamiento inicial de $O(n)$. En nuestro caso, el **preprocesamiento** consiste en crear un arreglo auxiliar que contiene, en la posición i, la suma de todos los cofres desde la posición 0 hasta i.

Con esta estructura, la suma de los cofres en cualquier intervalo $[i, j]$ se puede obtener con la fórmula:

$$\text{sumintervalo}(i, j) = \text{prefix}[j] - \text{prefix}[i - 1]$$

Esto nos permite calcular la suma de cualquier intervalo de cofres de manera eficiente, sin tener que recorrerlos uno por uno en cada iteración.

1.8.4 Beneficio de las sumas prefijas en el algoritmo

El verdadero propósito de las **sumas prefijas** es facilitar el cálculo de las sumas de los cofres en cada intervalo, agrupando aquellos que comparten el mismo coeficiente. Esto se hace con el fin de multiplicar la suma total de esos cofres por su coeficiente correspondiente.

Con esto, podemos calcular las sumas mucho más rápido. La idea es que podemos obtener la suma de los primeros k cofres que están ordenados, los siguientes k, y así sucesivamente en $O(1)$, evitando la necesidad de recalcular las sumas de nuevo en cada iteración. Esto es batsante útil ya que por ejemplo para los cofres que tienen coeficiente 0 (aquellos que caen en las trampas), podemos multiplicar su suma por 0 directamente, como si estuviéramos sacando un factor común, y hacer lo mismo para los cofres con coeficientes 1, 2, etc.

1.8.5 Reducción de la complejidad total

La **complejidad total** del algoritmo se reduce de $O(n^2)$ a $O(n \log n)$ de la siguiente manera:

1. **Ordenar los cofres**:
    - La ordenación de los cofres por su valor en orden descendente toma $O(n \log n)$.
2. **Preprocesamiento de las sumas prefijas**:
    - Construir el arreglo de sumas prefijas toma $O(n)$, ya que solo necesitamos una pasada sobre los cofres para calcular estas sumas acumuladas.
3. **Cálculo de las ganancias para cada k**:
    - Si n no es divisible por k, el último bloque tendrá una longitud menor que k, pero esto no cambia el análisis general. Para cada k, el número de intervalos es precisamente k, podemos obtener

cada suma de intervalo en $O(1)$ utilizando las sumas prefijas y multiplicarlas por su coeficiente, , debido a que esto se hace de k cofres en k cofres en la lsita de los cofres ordenados, llegamos a la conclusion de que en el peor caso, para cada k, el cálculo de las ganancias toma $\frac{n}{k}$ operaciones.

Sumatoria para todos los valores de k:

La complejidad total es la suma de los tiempos de cálculo para todos los valores de k desde 1 hasta n. Esto implica sumar:

$$\sum_{k=1}^{n} \frac{n}{k}$$

Esta es una sumatoria armónica, cuyo comportamiento asintótico es:

$$O\left(\sum_{k=1}^{n} \frac{n}{k}\right) = O(n \log n)$$

Por lo tanto, la complejidad total del algoritmo, después de optimizar con sumas prefijas, se reduce a:

$$O(n \log n)$$

1.9 Test de Rendimiento y Comparación

1.9.1 Descripción del Test

Se realizó un test de rendimiento con el objetivo de comparar las salidas y los tiempos de ejecución entre dos versiones de un algoritmo: una versión optimizada y otra no optimizada. El test consistió en ejecutar ambas versiones del algoritmo para diferentes tamaños de entrada, que fueron aumentando progresivamente desde un tamaño inicial de 100 hasta un tamaño máximo de $2 \cdot 10^5$ elementos.

Cada iteración del test generó una lista de números aleatorios entre 1 y 10^6, y tanto la versión optimizada como la no optimizada procesaron la misma lista. Se midió el tiempo de ejecución de cada versión en cada iteración.

1.9.2 Objetivo

El objetivo de este test es observar cómo el tiempo de ejecución de ambas versiones del algoritmo se comporta a medida que crece el tamaño de la entrada. Se espera que la versión optimizada tenga un mejor rendimiento,

especialmente con tamaños de entrada grandes. Al final del test, se plotearon los tiempos de ejecución para visualizar la diferencia en rendimiento entre las dos versiones.

1.9.3 Resultados

A continuación se muestra el gráfico que compara los tiempos de ejecución de la versión optimizada y la no optimizada:

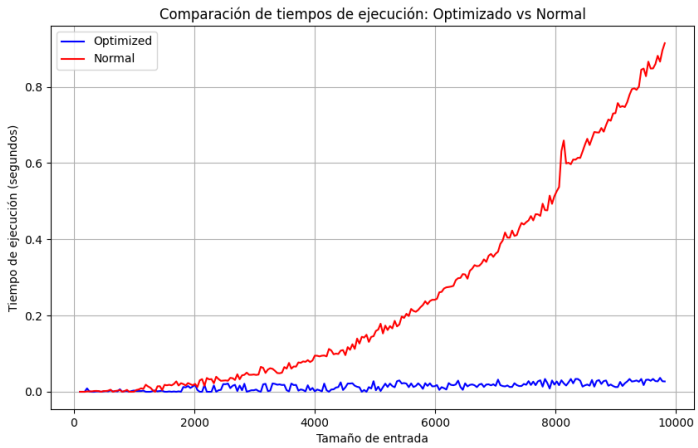

Figure 1: Comparación de tiempos de ejecución entre la versión optimizada y la no optimizada para una entrada maxima de 10^5.

1.10 Probabilidades y valor esperado

El valor esperado de las ganancias está relacionado directamente con las **probabilidades** de recoger cofres desde una posición inicial aleatoria. Dado que el jugador comienza en una habitación aleatoria, cada habitación tiene la misma probabilidad de ser elegida.

Para un cofre v_j en una posición j-ésima dentro de un intervalo con cnt_i cofres, la probabilidad de recoger ese cofre desde una posición inicial aleatoria es:

$$P(\text{recoger } v_j) = \frac{j}{n}$$

Donde:

- j es el número de veces que se puede recoger el cofre v_j antes de llegar a una trampa en un intervalo.

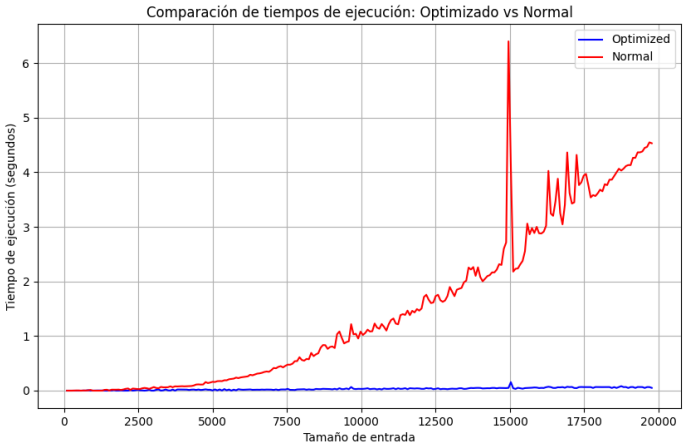

Figure 2: Comparación de tiempos de ejecución entre la versión optimizada y la no optimizada para una entrada maxima de $2 \cdot 10^5$.

- n es el número total de habitaciones.

El **valor esperado** de las ganancias del jugador se puede calcular sumando las contribuciones de cada cofre, ponderadas por su coeficiente y su valor:

$$E_k = \frac{1}{n} \sum_{i=1}^{k} \sum_{j=1}^{\text{cnt}_i} j \cdot v_j$$

El valor esperado de las ganancias del jugador se puede calcular para cada k (donde k es el número de trampas) sumando las contribuciones de cada cofre que no es trampa, ponderadas por su coeficiente y su valor.

1.10.1 ¿Por qué se minimizan las ganancias con este enfoque?

El enfoque de asignar los cofres más valiosos a las posiciones con coeficientes más bajos asegura que se minimice la contribución de esos cofres a la suma total de ganancias, dado que las probabilidades de recogerlos son menores en esas posiciones.

1.11 Conclusión

El valor esperado de las ganancias se minimiza al seguir estos pasos:

1. **Balancear los intervalos** moviendo cofres de intervalos grandes a pequeños.

2. **Asignar los cofres más valiosos** a las posiciones con los coeficientes más bajos, minimizando su impacto en las ganancias.

3. **Utilizar sumas acumuladas (prefix sums)** para calcular las ganancias de manera eficiente.

Este enfoque permite calcular las ganancias con una complejidad total de $O(n \log n)$, asegurando que el jugador tenga las menores ganancias posibles.

2 Grid

Dada una matriz de $n \times n$ con k secciones rectangulares pintadas de negro y la posibilidad de pintar de blanco secciones rectangulares de cualquier dimensión $(h \times w)$ con un costo $\min(h, w)$, ¿cuál es el menor costo necesario para que todas las celdas queden pintadas de blanco de nuevo?

Primero resolvamos una instancia más sencilla del problema, donde inicialmente no hay k secciones enteras pintadas de negro, sino t celdas individuales pintadas de negro.

2.1 Aprovechando los costos

El costo de las operaciones tal y como fue definido cumple las siguientes propiedades:

1. Es óptimo en cuanto a costo pintar secciones de tamaño (h, w) con $\max(h, w) = n$. Esto se debe a que si $\max(h, w) < n$, el costo sería el mismo que de haber tomado $\max(h, w) = n$, pero nunca estaríamos accediendo a más celdas negras.

2. Sea S una sección, y sean $S_1, S_2, \ldots, S_i$ subsecciones contiguas de S, obtenidas de dividir a S a lo largo de su dimensión más pequeña. Entonces, el costo de pintar de blanco por separado a cada una de las subsecciones es el mismo que pintar S en una sola operación.

Particularmente, dividir una sección a lo largo de la menor de sus dimensiones en subsecciones cuya menor dimensión tiene longitud 1, y luego pintar cada una de estas subsecciones por separado tiene el mismo costo que haber pintado toda la sección en una sola operación y alcanza el mismo número de casillas negras.

Uniendo ambas propiedades, llegamos a que es óptimo solo pintar secciones de tamaño $(1, n)$ o $(n, 1)$; es decir, solo pintar filas o columnas. Luego, podemos redefinir el problema como:

Dada una matriz de $n \times n$ con k celdas pintadas de negro, ¿cuál es el mínimo número de filas o columnas que debemos pintar de blanco para que toda la matriz quede de color blanco?

2.2 Concretando más

Solo nos interesará considerar pintar de blanco las filas o columnas que contienen al menos una celda negra. Sea C el conjunto de tales filas o columnas. Nos interesa encontrar el menor subconjunto de C tal que incluya todas las celdas negras.

Además, no nos interesa el dato de las dimensiones de la matriz, ya que siempre operamos sobre filas o columnas enteras, ni nos interesan las celdas blancas, puesto que, una vez que una celda es convertida en blanco, no es posible modificar su estado mediante ninguna operación.

Entonces podemos redefinir el problema de una manera más cómoda como:

Dado t puntos en una grilla y el conjunto C de filas y columnas que contienen al menos un punto, ¿cuál es el subconjunto de C de menor tamaño que aún contiene a todos los t puntos dentro de sí?

2.3 Soluciones de Fuerza Bruta

Llamemos **subconjunto bueno** de C a aquellos subconjuntos tales que para todo punto de la grilla, en C existe al menos una fila o columna que lo contiene. Verificar si un subconjunto es bueno tiene complejidad $O(t)$, pues implica comprobar punto a punto.

Una primera solución de fuerza bruta sería tomar como espacio de búsqueda el espacio de los subconjuntos de C, y recorrerlo en orden creciente de tamaño de los subconjuntos, hasta encontrar un subconjunto bueno.

El conjunto C tiene a lo más $2t$ elementos, ya que a lo más t filas diferentes y t columnas diferentes pueden contener puntos.

Luego, la complejidad de esta primera solución es $O(2^{2t} \cdot t)$, ya que existen 2^{2t} subconjuntos de C y probar cada uno es $O(t)$.

2.4 DFS sobre el espacio de los subconjuntos buenos

Sea S un subconjunto bueno, notemos que añadir filas y columnas a S no hace que deje de ser bueno, por tal motivo es posible añadir filas y columnas a S hasta llegar a C, sin pasar en ningún momento por un subconjunto que no sea bueno. Análogamente, para todo subconjunto bueno S, es posible alcanzarlo partiendo de C y realizando una secuencia de eliminaciones de

elementos de C sin que ninguno de los subconjuntos intermedios por los que transitemos deje de ser bueno. Por tal motivo, podemos presentar esta segunda solución.

Representemos los subconjuntos de C como vértices de un grafo, y establezcamos una arista entre cada par de vértices tales que uno puede ser generado a partir del otro con la eliminación de solo un elemento. Comenzando en el nodo correspondiente al propio conjunto C, realicemos un DFS, deteniéndonos cada vez que el nodo a visitar no sea bueno. Luego, el DFS realizará O(número de subconjuntos buenos) operaciones. ¿Cuántos subconjuntos buenos hay?

Tomemos el caso en que todos los puntos se encuentran ubicados a lo largo de una sola fila. Entonces habrá t columnas diferentes. Luego habrá 2^t subconjuntos de C que contienen a la fila en cuestión y, por tanto, son buenos; además será bueno el conjunto que contiene a todas las columnas pero no a la fila. Luego, en este caso habría $2^t + 1$ conjuntos buenos. Como podemos apreciar, esta solución es también exponencial. (Falta probar rigurosamente que es $O(2^t \cdot t)$).

2.5 Representando como grafo

Cada punto se encuentra solo en una fila y una columna. Cada fila y cada columna tienen en común a lo más un punto. Es decir, los puntos establecen una relación entre las filas y las columnas. Formalmente:

Una **relación binaria** entre dos conjuntos A y B es un subconjunto de su producto cartesiano. Los puntos son un subconjunto del producto cartesiano entre las filas y las columnas.

Luego, probemos crear un grafo G, donde a cada fila y a cada columna corresponde un vértice, y solo existen aristas entre aquellos nodos representantes de filas y columnas que tienen un punto en su intersección.

2.6 Entra el Minimum Vertex Cover

Notemos que con esta representación nuestro problema se reduce al **Minimum Vertex Cover**. El problema del minimum vertex cover se define como: dado un grafo, encontrar el menor subconjunto de los vértices (filas o columnas en nuestro problema), necesarios para que todas las aristas del grafo (puntos en nuestro problema) tengan al menos uno de sus dos extremos incluido.

El problema del **Minimum Vertex Cover** es **NP-completo**; pero en grafos bipartitos tiene una solución polinómica, derivada del **Teorema de König**, que enuncia que en grafos bipartitos el minimum vertex cover tiene

el mismo tamaño que el **maximum matching**. Luego, hemos reducido el problema a calcular el **maximum matching** del grafo generado.

El algoritmo de **Hopcroft-Karp** computa el maximum matching de un grafo bipartito en $O(|E|\sqrt{|V|})$. Notemos que para el grafo que hemos creado se cumple que $|V| = t$ y $|E| = 2t$, por lo que la complejidad de nuestro algoritmo sería $O(t\sqrt{t})$ si usamos Hopcroft-Karp y $O(t^2)$ si usamos Dinic. No obstante, notemos que a su vez t es $O(n^2)$, puesto que una sección pintada de negro puede contener hasta n^2 celdas en su interior. Luego, la complejidad de nuestro algoritmo sería $O(n^4)$.

Si en lugar de Hopcroft-Karp, usáramos el más general y sencillo de implementar algoritmo de **Edmonds-Karp**, la complejidad sería mayor, ya que la complejidad de Edmonds-Karp es $O(|V||E|^2)$, lo que aplicado a nuestro problema lo llevaría al orden de $O(n^5)$. Esta complejidad es muy limitante.

2.7 Compresión

No obstante, busquemos aprovechar la forma en que fueron dispuestas en secciones las celdas negras e intentemos realizar una compresión.

Si bien las k secciones pueden abarcar n filas, solo existirán $2k$ filas frontera (que son la primera o la última de una sección). Luego, podemos agrupar a todas las filas en $2k$ rangos. El i-ésimo rango abarcará a todas las filas entre la i-ésima fila frontera (incluida) y la fila frontera $i + 1$ (excluida). Podemos realizar la operación análoga con las columnas.

Dos celdas que compartan rango en cuanto a fila y en cuanto a columna están pintadas de la misma manera. Esto se debe a que, si están pintadas de color diferente, una de ellas estará dentro de una de las secciones pintadas iniciales, mientras que la otra está fuera de cualquier sección pintada. Pero si una está adentro y la otra afuera, entre ellas debe haber una fila frontera, y si hubiera una fila frontera, ya sea horizontal o vertical entre ellas, entonces no compartirían el rango de fila o de columna, respectivamente.

Sea una **sección compuesta** únicamente por celdas que compartan rango de fila y columna. Realizar la operación de pintar todo un rango de filas o de pintar todo un rango de columnas nunca provocará que en la sección aparezcan celdas de colores diferentes (nuevamente, porque esto implicaría la presencia de una frontera dentro de la sección).

Luego, cada sección compuesta por celdas que comparten rango de fila y columna es monocromática respecto a las operaciones sobre rangos enteros de filas o columnas, de manera análoga a cómo las celdas individuales son monocromáticas respecto a las operaciones sobre filas o columnas.

Así, podemos crear a partir del problema original una nueva grilla donde cada fila o columna corresponde a un rango de filas o columnas, y cada celda

corresponde a una sección de celdas del problema original que comparten rango de filas y de columnas. Esta nueva grilla, en lugar de tener dimensiones $n \times n$, tiene dimensiones $k \times k$, con lo cual hemos dejado a la variable n correr libre, sin restricciones.

2.8 Adaptando el flujo

No obstante, aún falta un detalle para poder resolver el problema. En esta nueva grilla, el costo de eliminar una fila o una columna es la longitud del rango al que representa. Luego, el problema no será precisamente encontrar el subconjunto de menor tamaño, sino el de **menor costo** de las filas y las columnas, tal que contengan dentro de sí a todas las celdas negras de la grilla.

Si realizamos la misma representación en forma de grafo, asignándole un vértice a cada fila o columna, y una arista entre cada fila y cada columna; y además asignamos a cada vértice un costo (el de eliminar la fila o columna correspondiente), entonces el problema se reduce a encontrar el **Minimum Cost Vertex Cover**, que no es lo mismo que el **Minimum Vertex Cover**.

2.9 Minimum Cost Vertex Cover

El problema del **Minimum Cost Vertex Cover** es una generalización del problema del **Minimum Vertex Cover** y, al igual que este último, es **NP-completo**, pero existe una solución en tiempo polinomial para él.

La solución en tiempo polinomial parte de la generalización del **teorema de König** a grafos ponderados, que es el **Teorema de Evergény** (Schrijver, 2003, p. 318). El Teorema de Evergény, en esencia, afirma que el costo del Minimum Cost Vertex Cover es igual al peso del **Maximum Weight Matching** de un grafo equivalente.

Luego, solo nos queda transformar el grafo que tenemos, donde los vértices están ponderados, en un grafo donde las aristas tengan asignadas una capacidad y sea posible computar el flujo máximo. Para ello, sencillamente conectaremos un nodo *source* a todos los vértices correspondientes a filas, asignando por capacidad a su arista el ancho de la fila en cuestión. De manera análoga, conectaremos todos los nodos columna a un nodo *sink*. Para las demás aristas, colocaremos capacidad infinita.

Finalmente, podemos calcular el flujo máximo. Para ello, usaremos el **algoritmo de Edmonds-Karp**. La complejidad final de nuestra solución será entonces $O(k^5)$, donde k es el número de secciones iniciales.

2.10 Casos de Pruebas y Resultados

Se realizaron pruebas de rendimiento y exactitud para comparar las implementaciones de los algoritmos de **fuerza bruta**, **solución final optimizada**, y **solución descomprimida** en varios tamaños de entrada. A continuación, se presentan los resultados de los casos de prueba:

2.11 Descripción de los Casos de Prueba

1. **Tamaño pequeño de** n:
    - Se realizaron pruebas con tamaños de n entre 10 y 15, utilizando el algoritmo de fuerza bruta para comparar el rendimiento con las otras dos soluciones.
    - El valor de m se estableció como el 10% de n.
2. **Tamaño grande de** n:
    - Se probaron valores grandes de n (entre 20 y 500), comparando las dos implementaciones más eficientes.
    - El valor de m se mantuvo constante en 50 para simular la complejidad adicional con un número fijo de secciones pintadas.

2.12 Gráficos de Resultados

2.12.1 Comparación de Tiempos de Ejecución (n pequeño)

El siguiente gráfico muestra el tiempo de ejecución de las tres implementaciones (Fuerza Bruta, Solución Final, y Solución Descomprimida) en tamaños de n pequeños:

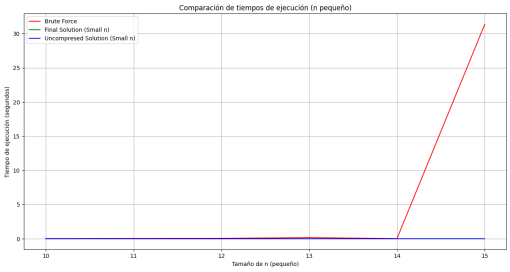

Figure 3: Comparación de Tiempos (n pequeño)

2.12.2 Comparación de Tiempos de Ejecución (n grande)

Este gráfico muestra los tiempos de ejecución para las soluciones más eficientes (Solución Final y Solución Descomprimida) en tamaños grandes de n:

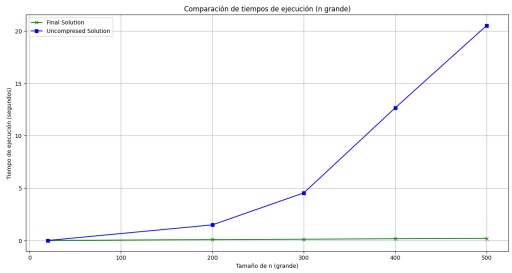

Figure 4: Comparación de Tiempos (n grande)

2.13 Análisis de Resultados

- **Tiempos de ejecución en n pequeño**: Como se esperaba, el algoritmo de fuerza bruta es significativamente más lento que las otras dos soluciones en cuanto los tamaños de n aumentan.
- **Tiempos de ejecución en n grande**: A partir de valores grandes de n, la **solución final** mostró un mejor rendimiento en comparación con la solución descomprimida, aunque ambas mantuvieron una diferencia notable con la fuerza bruta.

3 El secreto de la Isla

Eliminando las particularidades del problema (aldeas, guardianes, etc.) tenemos que: Dado un grafo $G = (V, E)$, debemos encontrar el subconjunto de vértices D más pequeño, tal que cada vértice de V pertenece a D o es adyacente a al menos un vértice de D.

Al problema anterior se le denomina problema del conjunto dominante (*dominating set problem*), un problema NP-completo.

3.1 Dominating Set Problem

Formalmente, el conjunto dominante se define como: Dado un grafo no dirigido $G = (V, E)$, un subconjunto de vértices $D \subseteq V$ se llama conjunto

dominante si para cada vértice $u \in V \setminus D$ hay un vértice $v \in D$ tal que $(u, v) \in E$.

El número de dominancia de G se define como:

$$Y(G) := \min\{|S| : S \text{ es un conjunto dominante de } G\}$$

El *dominating set* en un grafo con vértices aislados, los incluye por fuerza. Luego, el problema de computar el *dominating set* de un grafo con vértices aislados puede ser resuelto computando el *dominating set* del mismo grafo sin los nodos aislados y luego incorporando estos al resultado. Por tanto, pasemos a demostrar que computar el *dominating set* de un grafo sin vértices aislados es NP-Hard. Para ello, parece natural realizar una reducción desde *Vertex Cover* hasta *Dominating Set*, puesto que cada cubrimiento de un grafo sin nodos aislados es un *dominating set* de este; aunque no necesariamente el *Minimum Dominating Set*.

3.2 Demostrando que es NP-Completo

Mostremos que el problema de decisión de decidir si existe un *dominating set* de tamaño k en un grafo es un problema NP-Completo.

El problema pertenece a NP, ya que una solución candidata puede ser verificada en tiempo polinomial, con tan solo computar el conjunto de vértices dominados y verificar que su tamaño sea igual al número total de vértices del grafo.

Ahora encontremos una reducción en tiempo polinomial desde el problema del *Vertex Cover* al problema del *Dominating Set*. Esto nos permitirá afirmar que *Dominating Set* es NP-Hard, con lo cual completaríamos la demostración de que es NP-Completo.

Para un grafo $G(V, E)$, definamos el grafo G', como el resultante de mantener todos los vértices y aristas de G y añadir un vértice w por cada arista $\langle u, v\rangle$. Tal vértice w tendrá solamente aristas a los vértices u y v. Sea D un *dominating set* de tamaño k de G'. Si $D \subset V$, entonces, D es un *vertex cover* de G, puesto que para cualquier nodo de V que no esté en D, este se encontrará a distancia 1 de un nodo en D. Es decir, para toda arista, al menos uno de sus extremos está en D. Si $D \nsubseteq V$, es posible obtener un *dominating set* D' de tamaño k en G' que sí cumpla $D \subset V$, tan solo con sustituir todo vértice w no perteneciente a V, con uno de los dos vértices a los que está conectado. Sea u tal vértice. u estará conectado a todos los nodos a los que lo estaba w, ya que w solo estaba conectado a u y a un nodo v tal que existe una arista entre u y v, por lo cual u también está conectado a él. Este conjunto D', sí será *vertex cover* de G.

Por otro lado, sea C un *vertex cover* de tamaño k del grafo G. Entonces, todo vértice de G se encuentra a una distancia de a lo más 1 de un vértice del *vertex cover*, ya que para cada arista al menos uno de sus extremos está incluido en el *vertex cover* y en este grafo no hay nodos aislados. Además, como para cada vértice w que no está en G pero sí en G', se cumple que tiene una arista hacia los dos nodos de una arista, y al menos uno de los nodos de la arista está en C, entonces w se encuentra a una distancia 1 de al menos un nodo en C.

El algoritmo de conversión es meramente añadir un vértice y dos aristas por cada arista original, lo cual es polinómico. Luego, como decidir si existe un *Vertex Cover* de tamaño k en un grafo G es equivalente a aplicar la conversión planteada, obteniendo el grafo G', y decidir si existe un *Dominating Set* de tamaño k en G'; entonces, el problema de *Dominating Set* es NP-Completo.

3.3 Encontrar la solución exacta

Para encontrar la solución exacta del problema del **Conjunto Dominante**, es necesario evaluar todas las combinaciones posibles de subconjuntos de vértices, lo cual puede tener una complejidad exponencial de $O(2^n)$, siendo n el número de vértices en el grafo. Sin embargo, existen algoritmos que mejoran esta complejidad.

3.3.1 Algoritmos que reducen la complejidad

Se han desarrollado algoritmos que mejoran el tiempo de resolución exacta utilizando técnicas avanzadas de **ramificación y poda**, logrando reducir la complejidad a $O(1.5^n)$. Estos algoritmos son ideales para grafos de tamaño moderado, permitiendo resolver el problema exacto en menos tiempo que los algoritmos ingenuos de fuerza bruta.

3.4 Algoritmos de Aproximación

Dado que encontrar la solución exacta es computacionalmente costoso para grafos grandes, los **algoritmos de aproximación** son una alternativa eficaz. Uno de los más comunes es el **algoritmo greedy**, que selecciona nodos de manera ávida, cubriendo en cada paso la mayor cantidad posible de nodos no cubiertos.

El **algoritmo greedy** no garantiza la solución óptima, pero ofrece una solución cercana en un tiempo razonable. La aproximación que ofrece está relacionada con el logaritmo del *grado máximo* (Δ) del grafo.

A continuación, se presenta la demostración de que el algoritmo greedy tiene una aproximación de $O(\ln(\Delta))$:

3.4.1 Aproximación del Algoritmo Greedy

Contexto del Problema

Estamos resolviendo el problema del **Conjunto Dominante** en un grafo, donde queremos seleccionar un subconjunto de nodos tal que:

- Cada nodo en el grafo esté cubierto directamente (pertenezca al conjunto) o tenga un vecino que esté en el conjunto.
- El objetivo es minimizar el tamaño de este conjunto dominante.

Idea del Algoritmo Greedy

El **Algoritmo Greedy** selecciona nodos de forma ávida, eligiendo siempre el nodo que cubre más nodos no cubiertos en cada paso, hasta que todos los nodos estén cubiertos. El objetivo es probar que este proceso nos da una solución que es, como máximo, $\ln(\Delta)$ veces más grande que la solución óptima.

Análisis Amortizado: Distribución del Costo

Cuando seleccionamos un nodo para el conjunto dominante, en lugar de asignar todo el costo de esa selección a ese nodo, distribuimos el costo entre todos los *nodos no cubiertos* que ese nodo acaba de cubrir.

Por ejemplo:

- Si seleccionamos un nodo v y este cubre a 5 nodos no cubiertos (incluyéndose a sí mismo si también estaba sin cubrir), asignamos una fracción del costo de $\frac{1}{5}$ a cada uno de esos 5 nodos.

Descomposición del grafo en estrellas

Supongamos que tenemos una solución óptima al problema, es decir, un conjunto dominante S^*. Sabemos que en S^*, cada nodo que no pertenece al conjunto dominante tiene un vecino que sí pertenece. Esto nos permite dividir el grafo en *"estrellas"*, donde:

- Cada estrella tiene un *nodo dominante* de S^* como "centro".
- Los *nodos no cubiertos* por otros están "conectados" a este centro.

El costo de la solución óptima es cubrir 1 estrella por cada nodo en S^*, lo que significa que el costo de cubrir todos los nodos de la estrella es 1.

Ejemplo de cómo se ve una estrella en un grafo:

```
  o         <- Nodo central (dominador)
 /|\
o o o      <- Nodos no cubiertos (vecinos del nodo central)
```

En este ejemplo:

- El nodo central o (dominador) está en el conjunto dominante S^*.
- Los nodos conectados debajo son *nodos no cubiertos* que dependen del nodo central para estar cubiertos en el grafo.

Costo amortizado del algoritmo greedy

En el *Algoritmo Greedy*, seleccionamos nodos basándonos en cuántos nodos no cubiertos pueden ser cubiertos en cada paso.

- Sea v el nodo dominante de una estrella en la solución óptima S^*.
- Sea $w(v)$ el número de nodos no cubiertos en la estrella de v.

Cuando el algoritmo greedy selecciona un nodo, cubre a $w(v)$ nodos, y asigna una fracción del costo de $\frac{1}{w(v)}$ a cada uno de los nodos cubiertos. Esto se hace porque si v puede cubrir muchos nodos, el costo se reparte entre todos ellos.

Una vez que un nodo ha sido cubierto, no recibe más costos.

Peor caso y costo total

En el peor caso, el algoritmo greedy selecciona nodos uno a uno, cubriendo un nodo en cada paso. El costo total que recibe cada nodo en una estrella es la suma de fracciones de costo, como:

$$\sum_{i=1}^{\delta(v)+1} \frac{1}{i} = \frac{1}{\delta(v)+1} + \frac{1}{\delta(v)} + \cdots + \frac{1}{2} + \frac{1}{1} = H(\delta(v)+1),$$

donde $H(\delta(v)+1)$ es la *función armónica*, que se comporta asintóticamente como:

$$H(\delta(v)+1) \approx \ln(\delta(v)+1) + O(1),$$

donde $\delta(v)$ es el número de vecinos del nodo v, y por tanto $H(\delta(v)+1)$ se aproxima a $\ln(\Delta)$, siendo Δ el grado máximo en el grafo.

El costo amortizado total por estrella en el peor caso es aproximadamente $\ln(\Delta)$. Esto significa que el número de nodos en el conjunto dominante calculado por el *Algoritmo Greedy* es como máximo $\ln(\Delta)$ veces mayor que el tamaño del conjunto dominante óptimo.

Por lo tanto, el algoritmo proporciona una aproximación de $\ln(\Delta)$ para el problema del conjunto dominante.

3.5 Resultados de los Tests

A continuación se muestran los gráficos obtenidos tras la ejecución de los tests que comparan las soluciones exactas y aproximadas, así como el análisis del logaritmo del grado máximo.

3.5.1 Comparación de Soluciones Exactas y Aproximadas

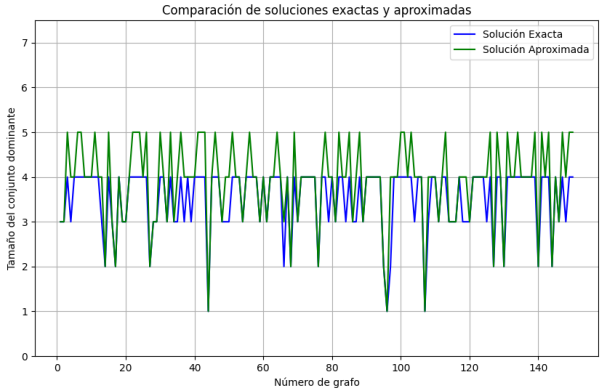

Figure 5: Comparación de Soluciones Exactas y Aproximadas

3.5.2 Logaritmo del Grado Máximo y Solución Aproximada

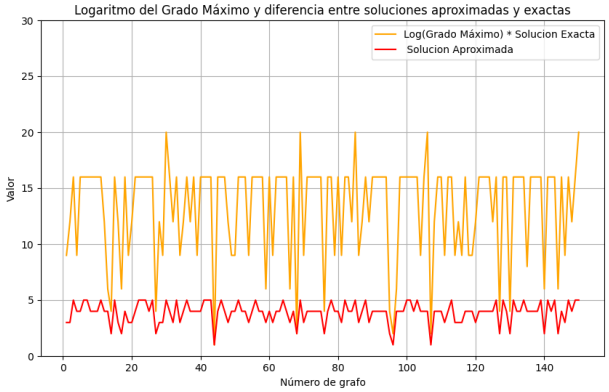

Figure 6: Logaritmo del Grado Máximo y Diferencia entre Soluciones

Los gráficos muestran claramente las diferencias entre las soluciones obtenidas de manera exacta y las soluciones aproximadas mediante el algoritmo greedy.

El análisis del logaritmo del grado máximo también proporciona información sobre la calidad de las aproximaciones obtenidas.

References

[1] Dominating Set, Chapter 7. Disponible en: `https://ac.informatik.uni-freiburg.de/teaching/ss_12/netalg/lectures/chapter7.pdf`

Schrijver, A. (2003). *Combinatorial optimization: Polyhedra and efficiency* (Vol. 24). Springer-Verlag.