Universidad de La Hábana

Facultad de Matemáticas y Ciencias de la Computación

Proyecto Final de DAA

Presentado por:

Alex Sánchez Saez C412
Carlos Manuel González C411
Jorge Alberto Aspiolea González C412

Septiembre 2024

Índice

1. Grid

1.1. Definición del problema

Un día iba Alex por su facultad cuando ve un cuadrado formado por $n \times n$ cuadraditos de color blanco. A su lado, un mensaje ponía lo siguiente: "Las siguientes tuplas de la forma (x_1, y_1, x_2, y_2) son coordenadas para pintar de negro algunos rectángulos. (coordenadas de la esquina inferior derecha y superior izquierda)" Luego se veían k tuplas de cuatro enteros. Finalmente decía: "Luego de tener el cuadrado coloreado de negro en las secciones pertinentes, su tarea es invertir el cuadrado a su estado original. En una operación puede seleccionar un rectángulo y pintar todas sus casillas de blanco. El costo de pintar de blanco un rectángulo de $h \times w$ es el mínimo entre h y w. Encuentre el costo mínimo para pintar de blanco todo el cuadrado."

En unos 10 minutos Alex fue capaz de resolver el problema. Desgraciadamente esto no es una película y el problema de Alex no era un problema del milenio que lo volviera millonario. Pero, ¿sería usted capaz de resolverlo también?

1.1.1. Entrada

La entrada del problema sería un entero n y m tuplas, tal que n sería la longitud de los lados del cuadrado; y las tuplas serían de la forma (x_1, y_1, x_2, y_2), con $0 \leq x1, x2, y1, y2 \leq n-1$, donde (x_1, y_1) sería las coordenadas de la esquina inferior derecha, y (x_2, y_2) las de las esquina superior izquierda.

1.1.2. Salida

Un número entero indicando el costo mínimo de revertir el color de todos los rectángulos a blanco.

1.1.3. Técnicas de solución empleadas

- Backtrack
- Greedy

1.2. Backtrack

Del problema tenemos un cuadrado en donde fueron pintados algunos rectángulos los cuales pueden solaparse entre ellos tanto parcial como completamente. Nuestra primera solución para atacar el problema fue crear la solución de fuerza bruta para poder probar nuestras posteriores soluciones.

Para esto implementamos un backtrack clásico, en donde iteramos por todas las tuplas de los rectángulos, y calculamos el costo de borrarlo o no borrarlo. En cada iteración guarda la mejor solución hasta el momento. El caso de parada sería cuando no haya más cuadrados negros, en ese caso se regresa por la rama del backtrack, hasta que se hayan probado todas las combinaciones de rectángulos.

1.2.1. Análisis de correctitud

Claramente este algoritmo da la solución correcta pues prueba todas las combinaciones posibles de borrar los rectángulos y se queda con la mejor, lo que en un tiempo no polinomial.

1.2.2. Analisis de complejidad

La complejidad de este algoritmo sería exponencial respecto a la cantidad de rectángulos, siendo de m!.

1.2.3. Algoritmo

```
def backtrack(matriz, rectangulos, solution=[],
    solution_cost=0):
    # Inicializamos la respuesta como infinito
    response = float('inf')


    for rect in solution:
        x1,y1,x2,y2 = rectangulos[rect]
        eliminar_rectangulo(matriz,x1,y1,x2,y2)

    if matriz_esta_vacia(matriz):
        costo_actual_solution = sum([peso_rectangulo(
            rectangulos[r][0],rectangulos[r][1],rectangulos[r
            ][2],rectangulos[r][3]) for r in solution])
        response= min(response,costo_actual_solution)

    for rect in solution:
        x1,y1,x2,y2 = rectangulos[rect]
        marcar_rectangulo(matriz,x1,y1,x2,y2,rect+1)


    for i, rect in enumerate(rectangulos):
        if i not in solution:
            # Agregamos el  ndice  del rect ngulo actual a
                la soluci n
            solution.append(i)
            # Llamamos recursivamente con la soluci n
                actualizada
            a = backtrack(matriz, rectangulos, solution,
                solution_cost + peso_rectangulo(rect[0], rect
                [1], rect[2], rect[3]))
            # Restauramos el estado de la matriz
            solution.pop()
            # Actualizamos la respuesta con el m nimo entre
                 la soluci n actual y la respuesta
            response = min(response, a)
```

```
29
30
31      return response
```

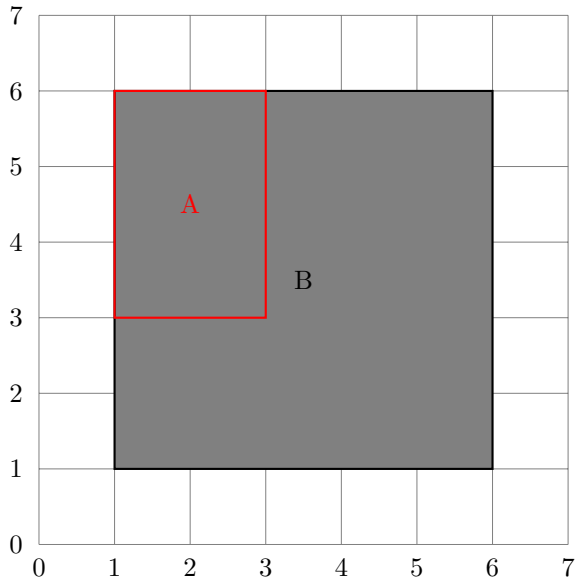
1.3. Primeras ideas

Podemos observar que si un rectángulo es cubierto por otros rectágunos, este no es necesario pintarlo ya que pintando a los que lo cubren se pintaría este también, evitándonos así un costo innecesario.

Aquí tendríamos otro problema y es en que orden se pintan los cuadrados para asegurarnos de que siempre se pintan solo los necesarios?

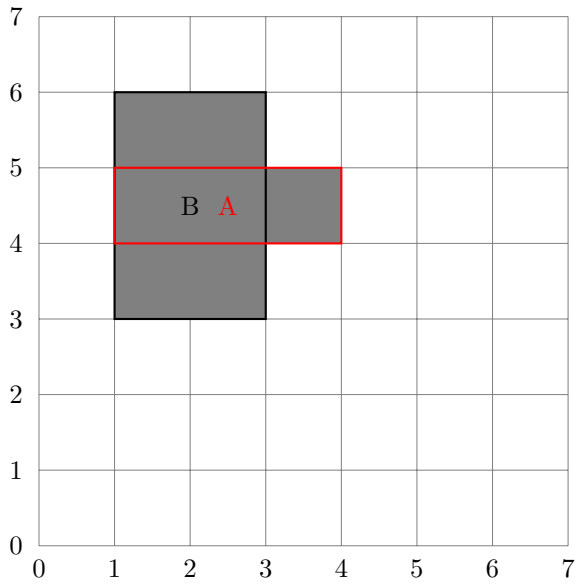
Para esto definamos algúnos puntos importantes.

- Si un rectángulo A esta completamente contenido dentro de otro B, no es necesario pintar A y sólo pintaríamos B. Esto es así porque aunque pintemos A, igual necesitaríamos pintar B, sin embargo si pintamos solo B este cubriría A, por lo que solo tendríamos el costo de pintar uno solo. Para cualquier Rectángulo si tiene un cuadrado que ningún otro rectángulo cubre, este estará presente en cualquier solución al problema, pues de lo contrario no se pintaría toda la superficie de blanco y no sería una solución.



- Si un rectángulo A tiene al menos una casilla la cual no es cubierta por ningún otro rectangulo, entonces hay que pintar A obligatioriamente. Esto es obvio ya que pintando los demas rectángulos solo pintaríamos una parte de A quedando algunas casillas en negro todavía que solo serán pintadas

de blanco si y solo si pintamos A directamente.



Sabiendo esto se nos ocurrio ordenar los rectángulos por la cantidad de casillas únicas que tengan, es decir la cantidad de casillas que cubre cada rectángulo tal que ningún otro cubre ese espacio, con un costo de $O(\log m)$ y elegir en cada momento el más grande según esta métrica. Queda un caso pendiente: ¿Que ocurre cuando no hay rectángulos con al menos un espacio que no cubran otros?, es decir cuando todos los rectángulos tienen métrica 0. En estos casos ocurre que todos se solapan entre sí, por lo que lo mejor sería tomar al mayor de estos, el que mas espacios cubra actualmente en la matriz y en caso de que hubiera mas de uno con el mismo tamaño, es decir haya más de uno con tamaño máximo, tomar el que tenga menor costo; con esto garantizamos que se tome el rectángulo que tenga mayor área, es decir que cubra al resto de rectángulos que como no hay espacios únicos entre ellos el más grande cubre completamente a al menos uno de los otros.

Con esto implementamos el siguiente algoritmo:

1. El algoritmo para solamente cuando la matriz está vacía
2. En cada paso ordenamos los rectángulos por la métrica anteriormente expuesta de Mayor a Menor
3. En cada paso tomamos el primer elemento del array de rectángulos y lo eliminamos de la matriz (ponemos todas las casillas que cubre en 0)

4. En el caso de que todos los rectángulos tengan métrica 0, entonces ordenamos por el que tenga mayor cantidad de rectángulos sin pintar en la matriz, en caso de que haya dos rectángulos con la misma cantidad de cuadros sin pintar , nos quedamos con el de menor peso

1.3.1. Algoritmo

```
def greedy_max_area(matriz, rectangulos: list):
    response = []

    # Ordenar rect ngulos primero por  rea  de mayor a
        menor y luego por coste de menor a mayor en caso de
        empate
    rectangulos=sorted(rectangulos,key=lambda rect:  (
        area_rect(matriz,rectangulos,*rect),covered_area(
        matriz,*rect)),reverse=True)


    costo = 0
    while not matriz_esta_vacia(matriz):
        # Seleccionar el primer rect ngulo, ya que est n
            ordenados por  rea  descendente y menor coste en
            caso de empate
        biggest_rect = rectangulos.pop(0)

        x1, y1, x2, y2 = biggest_rect

        # Ignorar rect ngulos que ya han sido eliminados
            completamente
        if rectangulo_borrado(matriz, x1, y1, x2, y2):
            continue

        # # Ignorar rect ngulos que est n completamente
            contenidos dentro de otros ya seleccionados
        # if rect_a_contenido(matriz, biggest_rect,
            rectangulos):
        #     continue

        # Eliminar el rect ngulo de la matriz y a adirlo a
             la soluci n
        eliminar_rectangulo(matriz, x1, y1, x2, y2)
        imprimir_matriz(matriz)
        response.append(biggest_rect)


        rectangulos=sorted(rectangulos,key=lambda rect:  (
            area_rect(matriz,rectangulos,*rect),covered_area(
            matriz,*rect)),reverse=True)

```

```
31          if all(area_rect(matriz, rectangulos, *rect) == 0
                for rect in rectangulos):
32              rectangulos = sorted(
33                      rectangulos,
34                      key=lambda rect: (-covered_area(matriz,
                            *rect), peso_rectangulo(*rect))
35                  )
36
37          costo += peso_rectangulo(x1, y1, x2, y2)
38
39      return costo
```

1.3.2. Análisis de correctitud

El algoritmo siempre para pues en cada paso se toma algún rectángulo y solo se dejan de tomar en el caso donde el espacio que ocupaba dicho rectángulo ya ha sido limpiado en espacios anteriores. Por las carácterísticas del ejercicio sabemos que siempre los rectángulos cubren totalmente el área a blanquear por lo que en el peor caso se tomarían todos los rectángulos dejando el área totalmente blanqueada por tanto la condición de parada del ciclo while se cumple para cada instancia de este problema en algún momento.

1.3.3. Demostrando que devuelve siempre una solución Óptima

Sea r_1,r_2,r_3....r_n un conjunto de rectángulos de costo mínimo y $r_{2,1}$,$r_{2,2}$,$r_{2,3}$....$r_{2,m}$ un conjunto obtenido por el algoritmo anteriormente expuesto.

Ahora supongamos:

$$\sum_{i=0}^{m} r_{2i} < \sum_{i=0}^{n} r_i \tag{1}$$

Hay dos posibles formas de que esto ocurra:

1. Existe una forma de cubrir totalmente la matriz con un costo menor al óptimo.
2. El algoritmo no cubrió completamente la matriz

Si ocurre (1), entonces r_i no sería la solución óptima, lo cuál es una contradicción.

Si ocurre (2), entonces el algoritmo habría parado sin estar la matriz vacía, esto es una contradicción pues el algoritmo solo para cuando la matriz está totalmente vacía.

Por tanto $\sum_{i=0}^{m} r_{2i} >= \sum_{i=0}^{n} r_i$
Supongamos entonces:

$$\sum_{i=0}^{m} r_{2i} > \sum_{i=0}^{n} r_i \tag{2}$$

En cada paso el algoritmo anterior toma los rectángulos que tengan la mayor cantidad de espacios sin pintar en la matriz tal que ningún otro rectángulo lo cubra, si estos rectángulos estuvieran en $r_{2,i}$ y no estuvieran en r_i, entonces r_i no cubriría toda la matriz pues estos rectángulos cumplen tener áreas tales que ningún otro los cubre. Por tanto para poder cubrir toda la matriz deben estar en $r_{2,i}$ y r_i.

Ahora verifiquemos que los nodos seleccionados en el caso cuando la métrica de todos los rectángulos es 0. En estos casos el algoritmo anterior toma los de mayor área, tales que si hay 2 de igual cantidad de zonas sin pintar se coloca primero el de menor peso. sea p el mayor rectángulo del subconjunto de rectángulos que cumplen r $\in r_{2,i}$ — métrica(r)=0 .

Supongamos que p $\notin r_i$, dado que p es el de mayor tamaño. y menor costo, como la métrica de cada rectángulo es 0 entonces todas las zonas de dicho rectángulo están cubierta por otros, si p $\notin r_i$, entonces para cubrir esa área están en r_i los rectángulos que tienen intersecciones con él, por lo que para cubrir la misma área se necesitan más rectángulos que o bien están totalmente contenidos en p o tienen un área que sobrepasa p , por lo que llamando o_i a estos rectángulos podemos afirmar:

$$\sum_{i=0}^{k} \text{peso}(o_i) \leq p \tag{3}$$

Dando como resultado que en $r_{2,i}$ cubra más área que r_i con un peso menor, lo que contradice que r_i sea la solución óptima. Por tanto todo rectángulo que está en $r_{2,i}$ está en r_i.

1.4. Análisis de Complejidad

La operación de la métrica de cada rectámgulo tiene costo O(n) pues hay que verificar las zonas comunes con el resto para saber si hay zonas que pertenecen únicamente al rectángulo que se está analizando.

El algoritmo de ordenación tiene costo O(nlogn) por lo que se hacen en el peor caso nlogn comparaciones, como cada comparación se hace en O(n), entonces el resultado de ordenar los rectángulos es O($n^2 * log(n)$)

Como hacemos esto por cada rectángulo hasta que la matriz esté vacía el costo total será de O($n^3 * log(n)$), el costo de ordenar los rectángulos en el caso de que todos tengan métrica 0 es O($n * log(n)$) pues cada operación se hace en tiempo constante. Por tanto el costo total del algoritmo será de O($n^3 * log(n)$).

2. El Laberinto

2.1. Definición del problema

En tiempos antiguos, esos cuando los edificios se derrumbaban por mal tiempo y la conexión mágica era muy lenta, los héroes del reino se aventuraban en el legendario laberinto, un intrincado entramado de pasillos, cada uno custodiado por una bestia mágica. Los pasillos sólo podían caminarse en un sentido pues un viento muy fuerte no te dejaba regresar. Se decía que las criaturas del laberinto, uniendo sus fuerzas mágicas (garras y eso), habían creado ciclos dentro de este, atrapando a cualquiera que entrara a ellos en una especie de montaña rusa sin final en la que un monstruo se reía de ti cada vez que le pasabas por al lado, una locura.

El joven héroe Carlos, se enfrentaba a una prueba única: desmantelar los ciclos eternos y liberar los pasillos del laberinto para que su gente pudiera cruzarlo sin caer en los bucles infinitos de burla y depravación.

Cada vez que el héroe asesinaba cruelmente (no importa porque somos los buenos) a la criatura que cuidaba una un camino, este se rompía y desaparecía. Orión era fuerte, pero no tanto, debía optimizar bien a cuántos monstruos enfrentarse. Ayude al héroe encontrando la mínima cantidad de monstruos que debe matar para eliminar todas las montañas rusas de burla y depravación.

2.1.1. Primer contacto

El problema puede ser visto comoo un Digrafo cíclico(si no hay ciclios no tiene sentido el ejercicio), donde las salas del laberinto serían los nodos, y los pasillos los arcos. Por lo que el problema se resume en encontrar la menor cantidad de aristas a quitar para hacer el grafo acíclico.

2.1.2. Algoritmos utilizados

- GreedyFAS
- SimpleFAS
- KwikSort

2.2. Comprobación en timepo Polinomial

Si queremos comprobar una solución $< G, k >$ es decir si eliminando k arcos eliminamos los ciclos. Simplemente eliminamos esos k arcos y comprobamos que no queden ciclos en el grafo resultante. Usando algoritmos como Floyd Warshall O(V^3), verificando aristas de retroceso con DFS(O(V(G)+E(G))), etc. Por lo que verificar una solución es posible en tiempo polinomial.

2.3. Reduciendo a Vertex Cover

Supongamos que tenemos un grafo G, convirtámoslo en un grafo dirigido. Siguiendo las siguientes reglas:

1. Por cada vertice $u \in V(G)$ creamos en G_f(grafo dirigido resultante) dos vértices u_{in} y u_{out} y conectamos u_{in} con u_{out}
2. Por cada arista $u, v \in E(G)$ conectamos u_{out} con v_{in} y v_{out} con u_{in} en G_f

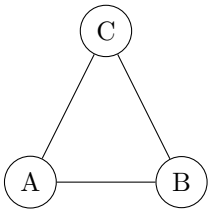


Figura 1: Grafo G

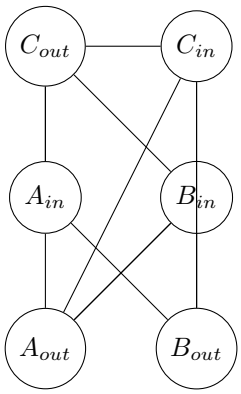


Figura 2: Grafo resultante G_f

Esta transformación es posible hacerla en tiempo polinomial siguiendo las dos reglas anteriores por cada nodo , por lo que la transformación se haría en O(V(G))

Por cada vértice en G se construye un ciclo en G_f, asumimos que el grafo no tiene vertices aislados de tener estos sería imposible hacer Vertex Cover y no tendría sentido la reducción. Por tanto si existiera un algoritmo que resolviera este problema en una complejidad polinomial, se pudiera transformar a Vertex Cover en una complejidad polinomial, por tanto se resolvería Vertex Cover en una complejidad polinomial, no puede pasar porque Vertex Cover está demostrado es NP-Hard. Por tanto el problema en cuestión es NP-Completo

Este problema es similar a un problema llamado "Feedback Arc Set" ya que ambos consisten en eliminar la mínima cantidad de arcos en un digrafo acíclico tal que el resultado sea un DAG. Este problema fue demostrado NP por Karp.

2.4. GreedyFAS

El algoritmo GreedyFAS es una aproximación heurística para resolver el problema del Feedback Arc Set (FAS), pero no garantiza que la solución sea óptima. Sin embargo, en la práctica, la solución obtenida puede estar razonablemente cerca del óptimo en muchos casos.

No da garantias teóricas de su cercanía al resultado óptimo real. Para cada vértice u en el grafo G, se define:

- $d^+(u)$: el número de arcos que salen de u (outdegree).
- $d^-(u)$: el número de arcos que entran en u (indegree).
- $\delta(u) = d^+(u) - d^-(u)$: una medida de cuán "fuente" o "sumidero" es el vértice.

2.4.1. Proceso

En cada iteración, el algoritmo elimina vértices de G siguiendo las siguientes reglas:

1. Elimina vértices que son sumideros ($d^+(u) = 0$) y conéctalos a una secuencia s_2.
2. Elimina vértices que son fuentes ($d^-(u) = 0$) y añádelos al final de una secuencia s_1.
3. Si no hay fuentes ni sumideros, elimina el vértice u con el mayor valor de $\delta(u)$ y añádelo a s_1.

2.4.2. Resultado

Cuando se eliminan todos los vértices, se obtiene una secuencia $s = s_1 + s_2$, donde los arcos orientados de derecha a izquierda (backward arcs) forman un conjunto de arcos de retroalimentación.

La complejidad temporal del algoritmo es O(V + E), donde V es el número de nodos y E es el número de aristas en el grafo. Esto se debe a que cada nodo y cada arista se procesan una vez.

2.5. SimpleFAS

El algoritmo SimpleFAS se basa en un algoritmo muy simple de 2-aproximación para el problema de conjunto de arcos de retroalimentación mínimo (MAS). El proceso se describe a continuación:

2.5.1. Proceso

1. Primero, se fija una permutación arbitraria P de los vértices de G.
2. Luego, se construyen dos subgrafos L y R:
   - L contiene los arcos (u, v) donde $u < v$ en P.
   - R contiene los arcos (u, v) donde $u > v$ en P.
3. Después de esta construcción, tanto L como R son subgrafos acíclicos de G.
4. Al menos uno de ellos tiene un tamaño que es al menos la mitad del mayor subgrafo acíclico. Por lo tanto, podemos devolver $m - \text{máx}(|L|, |R|)$ como el tamaño de un conjunto de arcos de retroalimentación para G.

2.5.2. Complejidad

La complejidad en tiempo del algoritmo SimpleFAS es $O(m + n)$, donde m es el número de arcos y n es el número de vértices en el grafo.

Sea OPT el tamaño del conjunto mínimo de arcos de retroalimentación (la solución óptima). Queremos probar que el conjunto de retroalimentación producido por SimpleFAS es como máximo $2 \times \text{OPT}$.

Observaciones:

1. Los subgrafos L y R son acíclicos por construcción, ya que L contiene los arcos donde $u < v$ según la permutación P, y R donde $u > v$. Los grafos acíclicos no contienen ciclos.

2. El grafo G se descompone en los arcos de L y R. Al menos uno de estos subgrafos contiene al menos la mitad de los arcos de un subgrafo acíclico máximo. Es decir:

$$\text{máx}(|L|, |R|) \geq \frac{|A_{\max}|}{2}$$

donde $|A_{\max}|$ es el número de arcos en el subgrafo acíclico máximo.

3. La solución óptima OPT es tal que eliminar OPT arcos deja un subgrafo acíclico de tamaño $|E| - \text{OPT}$.

Cota superior:

El algoritmo SimpleFAS devuelve $|E| - \text{máx}(|L|, |R|)$. Dado que $\text{máx}(|L|, |R|) \geq \frac{|E| - \text{OPT}}{2}$, tenemos:

$$\text{SimpleFAS} \leq |E| - \frac{|E| - \text{OPT}}{2}$$

Simplificando:

$$\text{SimpleFAS} \leq \frac{|E| + \text{OPT}}{2}$$

Dado que $|E| \geq \text{OPT}$, obtenemos:

$$\text{SimpleFAS} \leq 2 \times \text{OPT}$$

Por lo tanto, SimpleFAS es una 2-aproximación.

2.6. KwikSortFAS

Input: Arreglo lineal A, vértice lo, vértice hi
Output: Un conjunto de arcos de retroalimentación para G

2.6.1. Proceso

1. Si $lo < hi$ entonces:
    - Inicializar $lt \leftarrow lo$, $gt \leftarrow hi$, $i \leftarrow lo$.
    - Elegir un pivote aleatorio p en el rango $[lo, hi]$.
    - Mientras $i \leq gt$ hacer:
        - Si existe un arco (i, p):
            - Intercambiar lt con i.
            - Incrementar lt y i.
        - Sino, si existe un arco (p, i):
            - Intercambiar i con gt.
            - Decrementar gt.
        - Sino:
            - Incrementar i.
2. Llamar recursivamente a $KwikSortFAS(A, lo, lt - 1)$.
3. Si se realizó al menos un intercambio: - Llamar recursivamente a $KwikSortFAS(A, lt, gt)$.
4. Llamar recursivamente a $KwikSortFAS(A, gt + 1, hi)$.

Nota: El algoritmo utiliza un método de partición de 3 vías para ordenar, lo que permite manejar eficientemente los vértices desconectados.

2.6.2. Complejidad

Sea OPT el tamaño del conjunto mínimo de arcos de retroalimentación (la solución óptima). Queremos demostrar que el conjunto de retroalimentación producido por KwikSortFAS es como máximo $3 \times$ OPT.

Observaciones:

1. El algoritmo utiliza un método de ordenamiento que intenta minimizar la cantidad de arcos de retroalimentación al organizar los vértices en un arreglo lineal favorable.

2. Al finalizar el ordenamiento, cada arco que va en dirección contraria al ordenamiento se cuenta como un arco de retroalimentación. Dado que el algoritmo organiza los vértices basándose en la existencia de arcos entre ellos, el número de arcos de retroalimentación generados es proporcional al desorden inicial de los arcos.

3. En el peor de los casos, el número de arcos de retroalimentación producidos por KwikSortFAS puede ser hasta el triple del número de arcos que se

necesitarían eliminar para obtener un grafo acíclico. Esto se puede establecer formalmente como:

$$|F| \leq 3 \times \text{OPT}$$

Cota superior:

Al ordenar los vértices, si consideramos que un arreglo óptimo (o deseado) requeriría eliminar un número OPT de arcos para eliminar todos los ciclos, la naturaleza del algoritmo permite que, en el peor caso, hasta dos arcos adicionales puedan contribuir a la retroalimentación. Por lo tanto, se establece que:

$$|F| \leq |E| - |E_{\text{final}}| \leq 3 \times \text{OPT}$$

Por lo tanto, KwikSortFAS es una 3-aproximación.

3. El Profe

3.1. Definición del problema

Sean m estudiantes repartidos en m grupos, donde cada uno de ellos suspendió o la prueba P (POO) o la prueba R (Recursividad). Se nos pide calcular la cantidad de conjuntos de tamaño k que se pueden formar con todos los estudiantes tal que, en estos grupos, todos suspendieron la misma prueba o son de la misma aula.

3.2. Entrada

Una lista de m estudiantes, cada uno de ellos representado por un par de valores a, b donde a es el grupo al que pertenence y b la prueba que suspendió. Además un entero k, el tamaño de los grupos que se quieren formar.

3.3. Salida

Un entero c, la cantidad de conjuntos de tamaño k que se pueden formar con los estudiantes respetando las limitaciones dadas.

3.4. Técnicas de solución empleadas

- Backtrack
- Combinatoria
- Combinatoria + Dinámica

3.5. Backtrack

Nuestro primer enfoque como siempre es realizar el backtrack para recorrer el espacio de todas las soluciones posibles, descartando las soluciones incorrectas y contando las correctas, garantizándonos así que contemos todas las soluciones posibles.

Esta solución combinatoria la podemos realizar como sigue. Generamos todas los conjuntos de tamaño k posibles y, por cada uno comprobamos que que la solucion sea válida. Si es válida la sumamos al total de conjuntos posibles y continuamos, sino la descartamos y seguimos. Esta solución pasa por todos los conjuntos posibles, tanto por los buenos como por los malos, por lo tanto si contamos todos los buenos al final tendremos la solución correcta.

3.6. Análisis de complejidad

Generar todos los conjuntos de tamaño k posibles es $\binom{m}{k}$, con m la cantidad total de estudiantes, y para comprobar que la solución sea correcta tenemos que recorrer los k elementos chequeando que cumpla las restricciones. Por tanto la complejidad total sería:

$$O(\binom{m}{k} * k) \equiv O(\frac{m! * k}{k!(n-k)!}) \equiv O(\frac{m(m-1)\dots(m-k+1)}{(k-1)(k-1} \dots (k{-}k{+}1)) \equiv O(\frac{n^2}{k^2}) \equiv O(n^2)$$

3.7. Combinatoria

El ejericios se trata de contar la cantidad de conjuntos de tamaño k que se pueden construir y que cumplan algunas restricciones. Este ejercicio ya que es solo de conteo se puede usar un enfoque combinatorio para resolverlo. Específicamente se puede usar **El principio de Inclusión-Exclusión** para darle solución. Primero definamos algunas variables. Sea:

- G_i el conjunto de estudiantes que pertenecen al grupo i, $1 \leq i \leq n$.
- P el conjunto de estudiantes que suspendieron la prueba de POO.
- R el conjunto de estudiantes que suspendieron la prueba de Recursividad.
- k el tamaño de los conjuntos que se quieren formar tal que $\forall s \in X_j \implies \forall s \in P \vee \forall s \in R \vee \forall s \in XG_i$.

Podemos definir la ecuación del principio de inclusión exclusión como sigue:

$$|X| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |A \cap C| + |A \cap B \cap C|$$

Esta función aplicada a nuestro problema sería interpretada como:

- A es el la cantidad de formas de crear conjuntos de tamaño k donde todos sean de la misma aula. Eso expresado en combinatoria sería $|A| = \sum_{i=0}^{n} \binom{|G_i|}{k}$, o sea, por cada grupo, contar la cantidad de conjuntos de tamaño k que se pueden formar.

- B es la cantidad de estudiantes que suspendieron la prueba P, por lo que la cantidad de conjuntos que se pueden formar de tamaño k es $|B| = \binom{|P|}{k}$.

- C es la cantidad de estudiantes que suspendieron la prueba R, por lo que la cantidad de conjuntos que se pueden formar de tamaño k es $|C| = \binom{|R|}{k}$.

Estos son los primeros 3 miembros de la expresión, la cantidad de conjuntos que se pueden formar independientemente. Pero como pueden haber estudiantes que suspendieron la misma prueba y son de mismo grupo, estos conjuntos se contaron doble, cuando se contó en la parte de los estudiantes por grupo y despues en los estudiantes por prueba. Por tanto hay q restarle estos grupos repetidos al total. Entonces quedaría:

- $|A \cap B|$ es la cantidad de conjutos que se pueden formar con estudiantes que suspendieron la prueba P y son del mismo grupo. Esto se expresa como $|A \cap B| = \sum_{i=0}^{n} \binom{|G_i \cap P|}{k}$.

- $|A \cap C|$ es la cantidad de conjutos que se pueden formar con estudiantes que suspendieron la prueba R y son del mismo grupo. Esto se expresa como $|A \cap C| = \sum_{i=0}^{n} \binom{|G_i \cap R|}{k}$.

- $|B \cap C|$ es la cantidad de conjutos que se pueden formar con estudiantes que suspendieron las dos pruebas. En nuestro ejericio los estudiantes suspendieron una prueba u otra, por lo que $B \cap C =$ y por tanto $|B \cap C| = 0$ por lo que podemos descartarlo.

Con estas expresiones ya substraímos todos los grupos que se contaron doble. Ahora la última parte de la expresión es la cantidad de grupos de tamaño k que se pueden formar de estudiantes que suspendieron las dos pruebas y están en el mismo grupo. Aquí sabemos que no hay estudiantes que suspendieron la misma prueba, por lo que el conjunto es vacío, y la intercepción de este con cualquier otro conjunto también es vacío. Luego la última expresión quedaría como $|A \cap B \cap C| = |A \cap | = || = 0$, por tanto esta expresión se puede obviar.

Con todas estas definiciones hacia nuestro problema, podemos definir la solución de nuestro ejercicio con la expresión matemática siguiente:

$$|X| = \sum_{i=0}^{n} \binom{|G_i|}{k} + \binom{|P|}{k} + \binom{|R|}{k} - (\sum_{i=0}^{n} \binom{|G_i \cap P|}{k} + \sum_{i=0}^{n} \binom{|G_i \cap R|}{k})$$

Por tanto, computando esta expresión podemos calcular la solución de nuestro problema.

3.8. Análisis de complejidad

Para el analisis de complejidad hay que tener varios factores a tener en cuenta. En principio computar la expresión matemática anterior es $O(n)$ con n la cantidad de grupos a los que pertenecen los alumnos. Esta complejidad es $o(n)$ asumiendo que las expresiones combinatorias se calculen en $O(1)$ y las tres sumatorias se calculan en $O(n) + O(n) + O(n) = O(n)$.

Asumiendo esto la complejidad de nuestro algoritmo sería $O(n)$. Pero para computar las expresiones combinatorias hay varias cosas a tener en cuenta. La primera es que como la combinación de n en k se expresa como:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Esta expresión al final son solo operaciones matemáticas que la máquina las puede realizar en $O(1)$, en cuyo caso la complejidad total sí sería $O(1)$. El problema es que al calcular factoriales, los números son demasiado grandes y crecen exponencialmente. Es por esto que al calcular un simple $\binom{20}{7}$ conlleva hacer un cálculo de $20 \times 19 \times \cdots \times 8 = 482718652416000$ el cual ya se sale de 32 bits y 20 y 7 son números pequeños para nuestro problema, sin embargo $\binom{20}{7} = 77520$ que es muchísimo más pequeño. Es este costo de calcular el factorial el que hace que calcular el valor directamente no sea una opción válida. Para esto usaremos programación dinámica.

3.9. Combinatoria + Programación Dinámica

Hasta ahora la complejidad de nuestra solución es $O(n)$, asumiendo que las combinaciones se puedan realizar en $O(1)$, el cual en máquinas de 32 bits no es realista. Una alternativa a proder procesar combinaciones más grandes sin la limitación que implica calcular un factorial es usando *el triángulo de Pascal*. Este triángulo se de las siguientes propiedades de las combinaciones:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\binom{n}{1} = 1$$

$$\binom{n}{n} = 1$$

Con estas tres propiedades podemos definir una matriz C de tamaño $m \times k$ en donde $C[i][j] = \binom{i}{j}$, m la cantidad de alumnos y k el tamaño de los conjuntos que se desea formar. Entonces podemos definir todas las posiciones $C[n][i] = C[i][i] = 1$, $\forall 1 \leq i \leq m$, y las posiciones $C[i][j] = C[i-1][j-1] + C[i-1][j]$, $i \neq j \wedge j > 1$ utilizando la primera expresión. Luengo hemos pasdo de tener que calcular el factorial de un número para computar la combinación a precomputar las combinaciones mediante sumas.

Con este enfoque de programación dinámica podemos computar las combinaciones en $O(1)$ con un costo de precómputo de $O(m * k)$. Como el peor caso es que todos pertenezcan al mismo grupo o que todos los alumnos suspendan la misma prueba podemos definir que $m \leq k$ y por tanto $O(m * k) = O(n * k)$.

Precomputando las combinaciones mediante sumas nuestro espacio de combinaciones posibles que podemos calcular con enteros de 32 bits crece, pudiendo realizar combianciones con números más grandes y que tengan más sentido en nuestro problema.

Entonces, usando programación dinámica nuestra solución aumenta la complejidad de $O(n)$ a $O(n) + O(n * k) = O(n * k)$, pero también aumentamos el rango posible de soluciones que podemos tener.

3.9.1. Grupos muy grandes

Nuestra solución con combinatoria también está limitada por la memoria que se pueda almacenar, ya que tenemos que guardar una matriz que en el peor de los casos es de tamano $n * n$, y si n es muy grande éste da problemas de memoria.

Si los números son muy grandes, (lo cuál no creemos que sea un problema para nuestro problema en la vida real), se puede usar aritmética modular tomando como módulo un número primo que sea lo suficientemente grande pero que quepa en un entero de 32 o 64 bits. Con este enfoque podemos computar operaciones con números tan grande como queramos, pero esto trae un problema y es que al trabajar con módulos no será posible saber realmente cual es la solución real de la combianción, ya que tendrías infinitas posibilidades y no tendríamos certeza de cual sería el número. Suponiendo que p sea el número primo, a nuestra solución real y b nuestra solución modulada, tendremos que:

$$a \equiv b \pmod{p}$$

Como no conocemos a ya que lo que computamos fue b, entonces para computar a sería de la forma:

$$a = p * w + b$$

con $w \geq 1$. Y como no conocemos a es imposible definir un valor concreto para w, por lo que podemos definir infinitos valores de w y esto nos generaría infinitos valores de a.

Otro enfoque sería usar tratar los números como strings y realizar las operaciones de suma, resta, multiplicación y división *.ª mano"*, lo cual nos permitiría computar numeros muchísimos más grandes pero ejecutando esto en una máquina que trabaja con números de 32 bits, computar estos valores ya no sería constante.

3.10. Utilizando Programación Dinámica

La idea es sencilla, utilizamos memorisación para reducir la cantidad de cálculos redundantes. Creamos una matriz dp[grupo][asignatura][cantidad] que

en cada posicion guardará todos los grupos posibles de tamanño cantidad que se pueden formar con los estudiantes del grupo [grupo] suspensos en la asignatura [asignatura]. Podemos notar la relación de recurrencia:

$$\begin{aligned} \mathrm{dp}[g][e][c] = \quad & \underbrace{\mathrm{dp}[g][e][c-1]}_{\text{Agregar nuevo estudiante}} \\ & + \underbrace{\mathrm{dp}[g-1][e][c]}_{\text{Seleccionar del grupo anterior}} \\ & + \underbrace{\mathrm{dp}[g][e-1][c]}_{\text{Seleccionar del examen anterior}} \\ & - \underbrace{\mathrm{dp}[g-1][e-1][c]}_{\text{Evitar doble conteo}} \end{aligned} \tag{4}$$

Luego de precomputar la matriz podemos calcular todas las columnas de la forma dp[*][*][k] y sumarlos para obtener todos los grupos de tamaño k.

El algoritmo quedaria:

```python
def count_sets(students, k):
    from collections import defaultdict

    # Paso 1: Identificar grupos y asignaturas  nicas
    groups = set()
    exams = set()

    for student in students:
        group, exam = student
        groups.add(group)
        exams.add(1 if exam=='R' else 2)

    # Paso 2: Inicializaci n de la tabla de DP
    dp = defaultdict(lambda: defaultdict(lambda: defaultdict
        (int)))

    # Estado inicial: Hay exactamente 1 forma de seleccionar
         0 estudiantes
    for group in groups:
        for exam in exams:
            dp[group][1 if exam=='R' else 2][0] = 1

    # Paso 3: Rellenar la tabla DP considerando cada
        estudiante
    for student in students:
        group, exam = student
        for count in range(1, k + 1):
            # Actualizar DP considerando diferentes
                subproblemas
```

```
            dp[group][1 if exam=='R' else 2][count] = dp[
                group][0 if exam=='R' else 0][count - 1]
            dp[group][1 if exam=='R' else 2][count] += dp[
                group - 1][0 if exam=='R' else 0][count]
            dp[group][1 if exam=='R' else 2][count] += dp[
                group][(1 if exam=='R' else 2) - 1][count]
            dp[group][1 if exam=='R' else 2][count] -= dp[
                group - 1][ (1 if exam=='R' else 2)  - 1][
                count]

    # Paso 4: Obtener el resultado final sumando los casos
        de tama o k
    result = 0
    for group in groups:
        for exam in exams:
            result += dp[group][1 if exam=='R' else 2][k]

    return result
```

3.10.1. Análisis de Correctitud

Por lo anteriormente expuesto el Algoritmo calcula todas las posiblkes combinaciones de grupos y asignaturas utilizando el principio de inclusión exclusión, evitando recalcular estados anteriores y evitando realizar las operaciones combinatorias que pueden llegar a ser costosas O(n) si tomamos como entrada el número utilizado para calcular.

3.10.2. Análisis de complejidad

La precomputación de la matriz tendría costo O(n+k) donde n es la cantidad de estudiantes y k el tamaño de los subconjuntos que se quieren armar. Luego hacer la sumatoria de los subconjuntos de tamaño k tendría costo O(n). Por regla de la suma el costo total del algoritmo sería O(n+k)