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

```
optimal_solution=[]
min_cost = float('inf')
def backtrack(matriz, rectangulos, solution=[],
    solution_cost=0):
    global optimal_solution
    global min_cost

    # 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)

    # Caso base: si la matriz esta completamente vacia,
    # devolver el costo de la solucion
    if matriz_esta_vacia(matriz):
        if solution_cost < min_cost:
            optimal_solution= solution.copy()
            min_cost = solution_cost
        return solution_cost

    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 indice del rectangulo actual
            # a la solucion
            solution.append(i)

            # Llamamos recursivamente con la solucion
            # actualizada
            a = backtrack(matriz, rectangulos, solution,
                solution_cost + peso_rectangulo(rect[0],
```

```
36                  rect[1], rect[2], rect[3]))
37
38              # Restauramos el estado de la matriz
39              restaurar_rectangulo(matriz, rect[0], rect[1],
40                  rect[2], rect[3])
41
42              # Eliminamos el indice del rectangulo actual
43              # de la solucion
44              solution.pop()
45
46              # Actualizamos la respuesta con el minimo entre
47              # la solucion actual y la respuesta
48              response = min(response, a)
49
50      return response
```

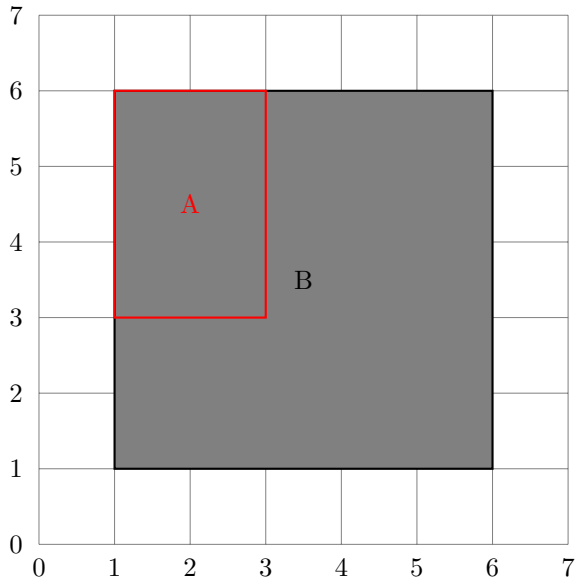
1.3. Primeras ideas

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.

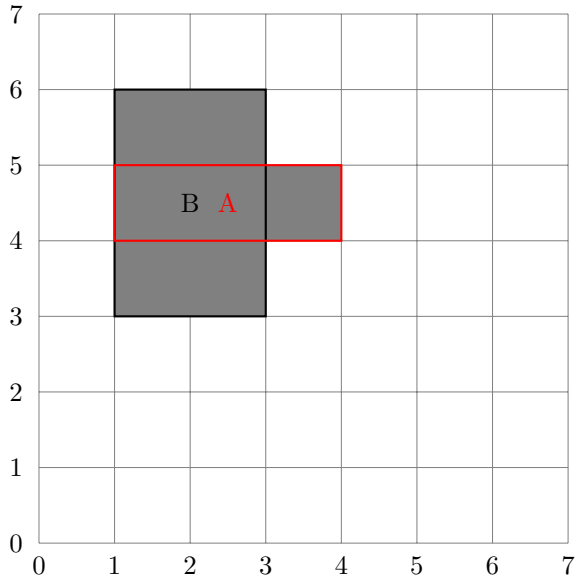
Aquí tendríamos otro problema y es en que orden se pintan los cuadrados para asegurarnos de que siempre se pintan solo los necesarios?

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.



- 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 área con un costo de $O(\log m)$ y elegir en cada momento el más grande, siempre que no haya sido borrado.

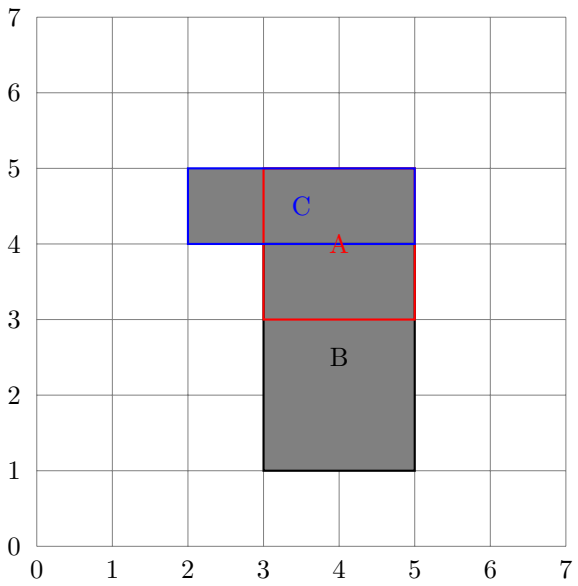
Este método nos acercó a la solución con un costo computacional polinomial, pues solo sería el costo de ordenar los rectángulos($O(\log m)$) más el costo de borrarlos todos ($O(m) \cdot O(k) = O(m \cdot k)$), donde k es la cantidad de cuadrados que conforman el rectángulo, que en el caso peor es $O(n^2)$. Por tanto este algoritmo tendría un tiempo de $O(\log m) + O(mn^2) = O(\text{máx}(\log m, mn^2) = O(mn^2)$

Esta solución tiene un caso que nos falla, el cual abordaremos más adelante.

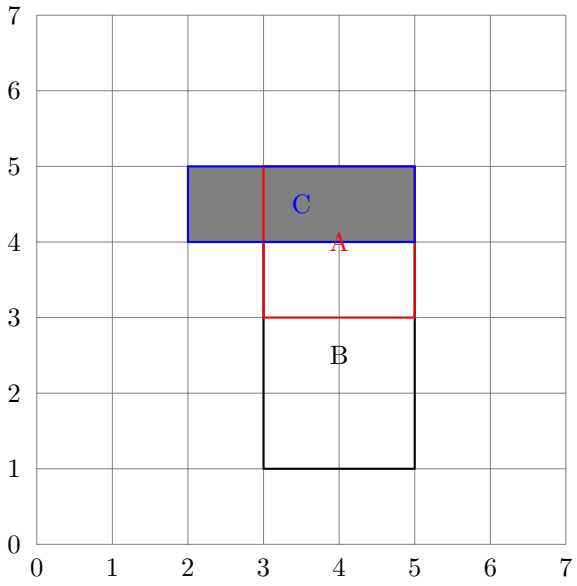
1.3.1. Problemas con este enfoque

El enfoque planteado hasta ahora, aunque no define una solución completa, si especifica como podemos enfrentar los posibles casos al plantear la solución final. Pero este tiene un problema.

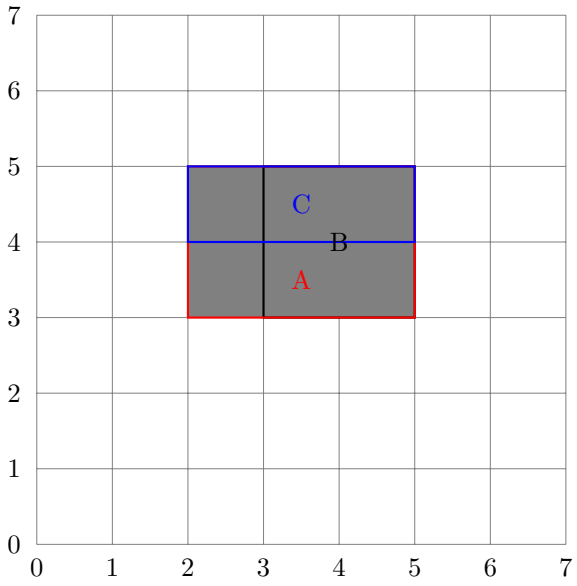
Pensemos que siguiendo este enfoque empezando del rectángulo más pequeño al más grande, si encontrarmos un rectángulo A de tamaño $i \times j$, y rectángulo mayor B de tamaño $i \times j + 1$, desplazado una posición en alguno de los ejes, por tanto B cubre a A, completamente, exepto una línea. Pero a su vez está linea es cubierta por un rectángulo C de tamaño $i + 1 \times 1$.



Observámos que si borramos el cuadrado más grande osea B, nos quedaría solo C por borrar, sin embargo A no se ha borrado completamente, por tanto el algoritmo borraría primero A y luego C, lo cual es incorrecto.



Para arreglar este problema se nos ocurrión en vez de borrar el rectángulo de mayor área, borrar el que mas área pinte en cada momento. Esto aumentaría la complejidad el algoritmo pues cada vez que se borre un rectángulo habría que recalcular cuantos cuadraditos están pintando en ese momento los rectángulos restantes. Pero este enfoque también fue erróneo pues asume que el rectángulo de mayor área siempre será borrado por él, y no es el caso, pues si es cubierto por otros que deben borrarse antes, por estar pintando casillas que solo ellos pueden borrar, entonces el más grande no debe borrarse.



Obsérvece aquí que el rectángulo de mayor área sería B, formado por 4 cuadraditos, y tanto C como A están formados por 3, por tanto a B no habría que borrarlo, pues C y A deben ser borrados para borrar las casillas que no borra B, y al borrarse ya borran B.

1.4. Greedy

Al recorrer los rectángulos de menor a mayor, si este tiene al menos un uno, entonces lo pintamos, sino restamos uno a todas sus casillas del rectángulo. Esto garantiza que si cuando un rectángulo mas grande pase por una casilla que previamente era mayor que uno pero para ese momento ya su valor es uno, indica que él es el ultimo rectángulo que queda que debería pintar dicha casilla, ya que los mas pequeños que él "borraron"su color de esa casilla con la seguridad que un rectángulo más adelante lo pintará por él.

1.4.1. Analisis de correctitud

Hasta ahora tenemos claro que:

- Si un rectángulo $A_{w \times h}$ tiene al menos un cuadrado que solo es pintado de negro por él mismo entonces este rectángulo hay que pintarlo de completamente con un costo de mín (w, h).
- Si un rectángulo $A_{w_1 \times h_1}$ es **cubierto completamente** por al menos un rectángulo $B_{w_2 \times h_2}$ tal que $w_1 \leq w_2$ y $h1 \leq h_2$, o sea, por un rectángulo mayor o igual a él que lo cubra completamente, entonces no es necesario pintar A ya que pintando B se cubre A, y el costo sería mín (w_2, h_2).

- Si un rectángulo $A_{w \times h}$ está cubierto completamente pero por más de un rectángulo, entonces si pintamos A sólo pintaríamos parcialmente los demás rectángulos, por lo que el costo total sería el costo de pintar A más el costo de pintar los demás, pero si delegamos la tarea de pintar A a los restantes rectángulos, el costo total solo sería el de pintar los demás.

- Se garantiza que pintando los rectángulos de menor a mayor con criterio de ordenación por área éste dará una respuesta correcta porque:

- Los más pequeños solo hay que pintarlos si no están cubiertos completamente por uno mas grande.

- Si un rectangulo grande está cubierto por varios rectángulos más pequeños y a su vez, el rectángulo grande está cubriendo a los más pequeños (vease como que el grande es una composición de rectángulos mas pequeños), entonces el coste óptimo es pintar el grande. Sea $A_{w \times h}$ el rectángulo grande y asumamos que hay n rectángulos más pequeños $B^i_{w_i \times h_i}$, tal que $\forall B^i_{w_i \times h_i}, w_i = w \vee h_i = h$, o sea, que todos los rectángulos B cubran a A a todo los alto o a todo lo ancho.

Con este emplo garantizamos que solo hallan rectángulos en horizontal o en vertial, pero que no hallan intermedios (figura de abajo). Si se cumple esto entonces el pintar los rectangulos mas pequeños entonces la suma de sus lados más pequeños h_i o w_i, entonces $\sum h_i = h \wedge \sum w_i = w$. En este caso, si $\forall w_i \leq h_i$, pero $h \leq w$ entonces la solución óptima sería w (porque es la suma de los lados más pequeños), sin embargo esto es un error ya que la solución optimas es h, por lo que es mejor pintar A directamente. Si por el contrario, $\forall w_i \leq h_i$, y $w \leq h$ entonces la solución óptima si es w, entonces da igual pintar A antes que los B^i que viceversa, la solución es la misma.

Si el caso anterior no se cumple y A está formado por varios rectángulos B^i pero no todos sus lados son iguales a w o a h como en el caso anterior, quiere decir que hay rectángulos intermedios, (TODO: poner la foto debajo), por lo que la $\sum w_i \geq w \vee \sum h_i \geq h$, y la solución optima en estos dos casos simpre va a ser pintar A primero.

Con lo anterior y siguendo la estrategia de descontar uno cada vez que encontramos un rectángulo que es cubierto por uno o varios mas grandes aseguramos como bien dijimos anteriormente, que en cada paso que restamos uno a cada casilla es como si estubiéramos .[el]iminando.[es]e rectángulo sin ningún costo, ya que uno o varios mas grandes que él lo cubrirán.

1.4.2. Análisis de complejidad

Sea n el tamaño del cuadrado y m la cantidad de rectángulos, donde cada rectángulo tiene como tamaño máximo $n \times n$. El algoritmo consta de 3 partes:

- **Ordenar los rectángulos de menor a mayor por área**: Usando el propio algoritmo de ordenamiento del lengugaje es $O(m \log m)$.

- **Pintar los rectángulos**: Por cada uno de los rectángulos, sumar uno a todas las casillas del mismo. $O(mn^2)$.
- **Calcular la solución**: Por cada rectángulo, recorrer sus casillas buscando al menos una casilla con valor de uno, y si es así entonces despintar este rectángulo (restarle uno a cada casilla). En el caso de que ningún rectángulo cubra a otro entonces el costo es $O(m(n^2 + n^2)) = O(mn^2)$

Con estos tres pasos, la complejidad total es:

$$O(m \log m) + O(mn^2) + O(mn^2) = O(m \log m) + O(mn^2) = O(mn^2)$$

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

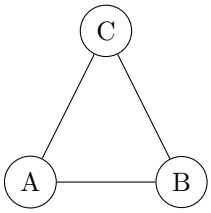


Figura 1: Grafo G

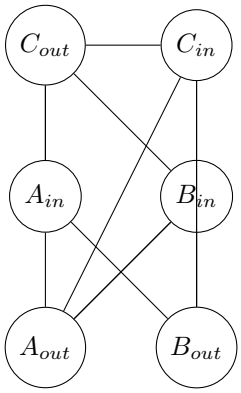


Figura 2: Grafo resultante G_f

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

Jorge es profesor de programación. En sus ratos libres, le gusta divertirse con las estadísticas de sus pobres estudiantes reprobados. Los estudiantes están separados en n grupos. Casualmente, este año, todos los estudiantes reprobaron alguno de los dos exámenes finales: P (POO) y R (Recursividad).

Esta tarde, Jorge decide entretenerse separando a los estudiantes suspensos en conjuntos de tamaño k que cumplan lo siguiente: En un mismo conjunto, todos los estudiantes son del mismo grupo i ($1 \leq i \leq n$) o suspendieron por el mismo examen P o R.

Conociendo el grupo y la prueba suspendida de cada estudiante, y el tamaño de los conjuntos, ayude a Jorge a saber cuántos conjuntos de estudiantes suspensos puede formar.

3.2. Solución Bruta:

Utilizando Backtrack realizamos todas las combinaciones donde el orden no importa de tamaño k posibles entre el conjunto de alumnos. En nuestro caso base contabamos solamente las que cumplieran las condiciones enunciadas (todos eran de la misma aula o habían suspendido la misma asignatura), luego devolviamos el valor resultante de contar todas las posibles combinaciones.

Complejidad Como generamos todas las posibles combinaciones de tamano k del conjunto de estudiantes de tamano n, la complejidad resultante del algoritmo sería de: $O(\frac{n!}{k!(n-k)!})$

Correctitud Al utilizar backtrack se explora todo el espacio de las posibles combinaciones de tamaño k del conjunto original, por lo que se expploran todas las combinaciones de elementos incluídas las válidas para este ejercicio por lo que se obtiene como resultado de filtrar por las condiciones de validez solo las combinaciones que interesan al ejercicio. Es decir se cuenta efectivamente todas las posibles combinaciones de tamaño k que cumplan las condiciones que necesita Jorge.

3.3. Solución Combinatoria:

Bajo la asunción e que ningún estudiante desaprobó ambas asignaturas, podemos utilizar el principio de inclusión exclusión para obtener el resultado de la cantidad total de grupos de tamaño k que se pueden obtener de un conjunto de n estudiantes:

Descripción del Algoritmo

El siguiente algoritmo permite contar el número de grupos posibles de tamaño k entre estudiantes, considerando sus aulas y asignaturas suspendidas. El enfoque consiste en crear un diccionario donde las llaves sean tuplas del tipo '(aula, asignatura)' y luego hacer uso de combinaciones para determinar cuántos grupos se pueden formar bajo estas condiciones.

- Primero, se crea un diccionario que almacena la lista de estudiantes para cada aula, asignatura y para la combinación de ambos.
- Luego, se cuentan todas las combinaciones posibles de tamaño k para cada aula y cada asignatura.
- Finalmente, se ajusta el número de combinaciones restando aquellas que pertenecen a la misma aula y que han suspendido la misma asignatura para evitar contar duplicados.

Código en Python

El siguiente bloque de código implementa el algoritmo en Python:

Listing 1: Código Python para contar grupos de estudiantes

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

    # Contar estudiantes por grupo y examen
    group_count = defaultdict(list)
    exam_count = defaultdict(list)
    combined_count = defaultdict(list)

    for student_id, (group, exam) in enumerate(students):
        group_count[group].append(student_id)
        exam_count[exam].append(student_id)
        combined_count[(group, exam)].append(student_id)

    total_sets = 0

    comb_aula=0
```

```
    comb_exam=0
    comb_comb=0
    # Contar grupos de tama o k por aula
    for group, student_ids in group_count.items():
        count = len(student_ids)
        if count >= k:
            tmp = comb(count, k)
            total_sets += tmp
            comb_aula+=tmp

    # Contar grupos de tama o k por examen
    for exam, student_ids in exam_count.items():
        count = len(student_ids)
        if count >= k:
            tmp = comb(count, k)
            total_sets += tmp
            comb_exam+=tmp

    # Contar grupos de tama o k por aula y examen
    for (group, exam), student_ids in combined_count.items()
        :
        count = len(student_ids)
        if count >= k:
            tmp = comb(count, k)
            total_sets -= tmp
            comb_comb+=tmp

    return total_sets
```

Explicación

El algoritmo realiza las siguientes operaciones:

1. Inicializa tres diccionarios:
   - ‘group$_c ount$‘ : $Almacenalosestudiantesporcadaaula.$‘$exam_c ount$‘ : $Almacenalosestudiantesporcadaasignaturasuspendida.$
   - ‘combined$_c ount$‘ : $Almacenalosestudiantesporcadacombinación$‘$(aula, asignaturasuspendida)$‘.
2. Recorre la lista de ‘students‘ para llenar los diccionarios.
3. Calcula todas las combinaciones posibles de tamaño k en cada uno de los tres contextos mencionados:
   - Suma el número de combinaciones posibles por aula (‘comb$_a ula$‘).$Sumaelnúmerodecombinacionesposiblesporasignatura$(‘$comb_e xam$‘).
   - Resta las combinaciones posibles por cada ‘(aula, asignatura suspendida)‘ para evitar duplicados (‘comb$_c omb$‘).

Finalmente, la función retorna el número total de conjuntos posibles de tamaño k.

Análisis de Complejidad y Correctitud

Análisis de Complejidad El análisis de complejidad del algoritmo se puede dividir en varias partes, considerando tanto el tiempo como el espacio requerido:

- **Inicialización de los Diccionarios:**
  - El tiempo para recorrer la lista de estudiantes y construir los diccionarios es $O(n)$, donde n es el número de estudiantes en la lista `students`.
  - La inserción de cada estudiante en los diccionarios se realiza en tiempo constante $O(1)$.

  Por lo tanto, la complejidad temporal de esta etapa es $O(n)$.

- **Conteo de Combinaciones:**
  - Calcular combinaciones de tamaño k usando `comb()` tiene un costo de $O(k)$.
  - El cálculo de combinaciones se realiza para cada grupo, cada asignatura y cada combinación '(aula, asignatura suspendida)'.

  La complejidad para calcular todas las combinaciones es:

  $$O(G \cdot k) + O(E \cdot k) + O(C \cdot k)$$

  Donde:
  - G es el número de grupos únicos.
  - E es el número de asignaturas suspendidas únicas.
  - C es el número de combinaciones '(aula, asignatura suspendida)'.

- **Complejidad Total:**

  La complejidad total del algoritmo en el peor de los casos es:

  $$O(n + G \cdot k + E \cdot k + C \cdot k)$$

  Donde:
  - n es el número de estudiantes.
  - G, E, y C son las definiciones anteriores.

  En términos generales, la complejidad se puede aproximar como $O(n + k \cdot \text{máx}(G, E, C))$.

Análisis de Correctitud El análisis de correctitud del algoritmo se basa en verificar que cada etapa se realice según lo especificado y que los resultados generados correspondan con las expectativas:

1. **Inicialización de los Diccionarios:** Se recorre la lista de `students`, identificando correctamente el `group` y `exam` de cada estudiante, y se insertan en los diccionarios correspondientes:
   - `group_count`: Almacena los estudiantes por cada aula.
   - `exam_count`: Almacena los estudiantes por cada asignatura suspendida.
   - `combined_count`: Almacena los estudiantes por cada combinación '(aula, asignatura suspendida)'.
2. **Cálculo de Combinaciones:**
   - El algoritmo verifica que cada grupo o asignatura tenga al menos k estudiantes antes de calcular combinaciones.
   - La función `comb()` se utiliza para calcular el número de combinaciones $C(n, k)$.
3. **Restar Combinaciones Duplicadas:** El algoritmo ajusta el número total de combinaciones restando aquellas que pertenecen a la misma aula y que suspendieron la misma asignatura:
   - Las combinaciones duplicadas se almacenan en `combined_count`, y se resta el número de combinaciones posibles para evitar duplicados.
4. **Retorno del Resultado:** El valor retornado por la función (`total_sets`) corresponde al número final de combinaciones de tamaño k, habiendo sumado y restado adecuadamente para evitar duplicados.

En conclusión, el algoritmo es correcto, ya que construye y manipula los datos de manera adecuada, calcula las combinaciones correctamente, y evita contar duplicados a través de un proceso lógico de suma y resta de combinaciones. La complejidad total es eficiente para tamaños de entrada moderados, pero puede volverse costoso si el número de grupos, asignaturas o combinaciones es muy grande en comparación con k.