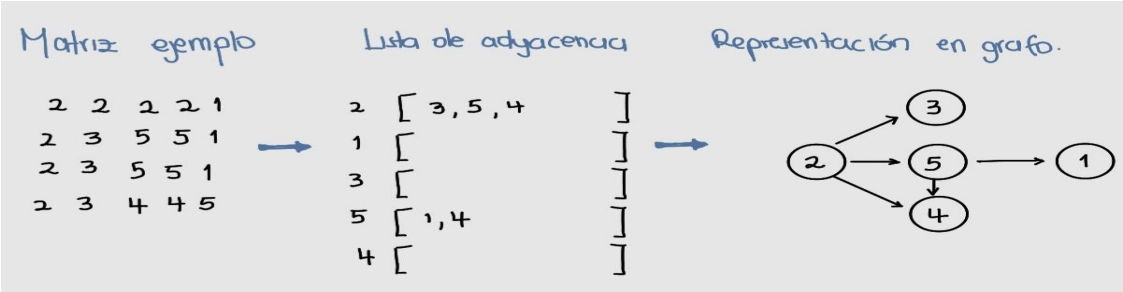
Proyecto #2

Diseño y análisis de algoritmos - Universidad de los Andes
Ronald Yesid Diaz Pardo - 202111309 - r.diazp, Nicolas Diaz Montaña - 202021006 - n.diaz9, Andres Felipe Guerrero - 202015143 - a.guerreros

Explicación de algoritmo para solucionar el problema

Se ha decidido modelar el problema por medio de una **lista de adyacencias** (grafo) y a está, se le aplicará un algoritmo que permita obtener un **ordenamiento topológico a cada subgrafo de ese grafo.** Esto, para saber el orden adecuado en el que se tienen que usar las llaves para abrir la máquina en el caso que se pueda.

Inicialmente, se recorre la matriz en dos ocasiones. La primera vez se busca el área rectangular que cubren las cajas correspondientes a cada llave, esto se hace localizando la esquina superior izquierda y la esquina inferior derecha. De esta forma se sabe no solo el tamaño del área rectangular de cada llave, sino también la ubicación del mismo en la matriz. En el segundo recorrido, se identifica qué áreas rectangulares de llaves están contenidas en otras. Esta información se guarda en una lista de adyacencias. Esta es la forma elegida para representar el grafo dirigido donde los nodos son las llaves y las conexiones hacen referencia a las demás llaves que están contenidas en ella. A continuación un ejemplo de las representaciones anteriormente mencionadas:



Ya con la lista de adyacencias creada se verifica si el grafo tiene ciclos o no, para esto se utilizó DFS. En esta parte se visita cada uno de los nodos del grafo y se anota en una lista (puede ser cualquier estructura de datos, en nuestro caso un diccionario) si ya ha sido visitado o no. Este proceso se realiza de forma iterativa hasta que no haya más nodos que visitar. Si durante el proceso se encuentra un nodo que ya está visitado dentro de la lista de visitados, significa que ese instante existe un ciclo y se termina el proceso. La detección de ciclos en el grafo ayuda a determinar si existe una configuración en la que no es posible abrir todas las cajas sin quedar atrapado en un bucle. Si no hay ciclos, significa que se pueden abrir todas las cajas de manera secuencial sin problemas.

Una vez comprobado que no existen ciclos se procede a realizar un **ordenamiento topológico**. Causó un gran interés ordenar las llaves de esta forma para asegurar la disposición de las llaves para abrir la máquina sin que esta se dañe. Para esto se selecciona un tipo de caja (llave) y por medio de la lista de adyacencia, vemos que tipo de cajas cerradas (llaves) están contenidas dentro de las dimensiones del set de esa llave. Para cada tipo de caja encontrada se repite el proceso. A medida que se avanza, se registra el orden en el que se abren las cajas. Después de explorar todas los tipos de cajas, se invierte ese registro para obtener el orden correcto en el que se abrieron las cajas.

Para hacer ese ordenamiento topológico de manera eficiente, se decidió organizar la lista de adyacencias por el número de sus edges (len(graph[v])) de manera descendente y tener una lista de vértices dentro del orden final que se retornara, al este se le va agregando vértices de cada orden topológico a medida que los hallamos para cada subgrafo.

Se propuso otra solución, donde las cajas que cubren una mayor dimensión de la máquina se deben abrir primero para encontrar la solución del mensaje. En otras palabras, se deben ir abriendo las cajas con mayor dimensión dentro de la máquina hasta las cajas de menor dimensión. Para lograr lo anterior el algoritmo identifica dentro de la máquina (que en nuestro caso sería la matriz que se nos da en la entrada) junto con las llaves que se nos dan (las cuales sería la entrada k), las columnas y filas (dimensiones) que cubren un set de cajas de una llave k. Una vez se tengan las dimensiones. Se organizan las llaves de mayor a menor con respecto al área de las cajas (la cual se encuentran a partir de sus dimensiones). El algoritmo siempre va de la llave con el set de mayor dimensión al menor. El orden en el que se recorren esos sets depende de cómo fueron sorteadas las llaves por medio de la función sort(). Pueden haber varios sets con las mismas dimensiones, pero como tal no importa cuál se abra primero siempre y cuando no afecta a otro set. Finalmente esta solución se ha descartado debido a que no en todos los casos el set mayor debe abrirse primero que la menor, Además de que la complejidad aumenta al tener que recorrer las llaves (n log n) veces.

1. **Análisis de complejidades espaciales y temporal**

Complejidad temporal:

Lo primero que se hace es un recorrido O(n^2) sobre la matriz, en este primer recorrido se busca que área rectangular cubre las cajas correspondientes a cada llave. Luego se realiza un segundo recorrido con una complejidad de O(n^2* k) , este ciclo recorre la matriz y para cada elemento de la matriz, mira si se ubica en cualquiera de las áreas halladas en el primer recorrido**.**

En segundo lugar, se hace una verificación de existencia de ciclos, para lo cual se usa el algoritmo de DFS, con una complejidad en el peor caso de O(V+E) lo cual es equivalente en nuestra terminología a O(k+k) = O(k).

En tercer lugar, se hace un ordenamiento de la lista de adyacencias, este algoritmo en python tiene una complejidad de O(v*logv) = O(k*logk).

En cuarto lugar, para cada subgrafo se halla el orden topológico, para el cual se usa el algoritmo DFS, con la complejidad mencionada anteriormente. Por lo que, en el peor caso habrán k cajas diferentes, donde cada caja tenga conflicto con aproximadamente k-1 cajas diferentes y por lo tanto ese sea su número de edges E, resultando así en una complejidad de O(V+E) = O(2k) = O(k).

En quinto lugar, para imprimir la respuesta correcta, se hace un for loop inverso sobre la lista de órdenes. Lo cual nos da una complejidad de O(k) en todos los casos.

Finalmente, uniendo todas las complejidades tendríamos una complejidad final = O(k+k+k*logk+k+n^2*k) = O(k+klogk+kn^2) = O(max(k, klogk, kn^2)) = **O(kn^2)**.

Complejidad espacial

En lo que respecta a complejidad espacial, identificamos que la complejidad espacial más significativa en el peor caso es la del grafo, ya que hay k vértices donde existe la posibilidad de que cada vértice contenga k-1 áreas diferentes y viceversa, lo cual implica una complejidad espacial de O(k^2). No identificamos otra debido a que hacemos uso de sets de python que ocupan memoria pero que solo tendrian k datos en el peor caso, lo cual es menor a k^2. Además, las demás estructuras siguen complejidades espaciales similares.

2. Respuestas a escenarios de comprensión de problemas algorítmicos

Escenario #1: "*Se puede aplicar la misma llave hasta dos veces*"

I. Nuevos retos que presupone este escenario #1

El impacto principal radica en la posibilidad de que una llave puede abrir una caja más de una vez, en otras palabras cubre la misma fila o columna en más de una ocasión. Esto introduce complejidades en la estructura del grafo y en la lógica de detección de ciclos, ya que las conexiones entre las llaves deben reflejar adecuadamente estas repeticiones. Además, la impresión de las dimensiones cubiertas por cada llave debe ser ajustada para considerar múltiples intervalos de filas y columnas cubiertas por la misma llave.

II. Cambios a la solución para adaptarse al escenario #1

En función de encontrar las dimensiones de las llaves, las estructuras de datos de las columnas recorridas y las filas recorridas deben permitir múltiples intervalos dado a que estamos visitando las cajas con una llave específica más de una vez. Por el lado de la creación del grafo, la construcción del mismo debe adaptarse para considerar conexiones entre llaves que pueden cubrir la misma fila o columna más de una vez, osea ahora tendríamos que mejor tener una matriz de adyacencia para tener la estructura del grafo de una forma más completa y precisa. La detección de ciclos por medio de DFS también necesita ser revisada para manejar correctamente la complejidad adicional introducida por las repeticiones de filas y columnas cubiertas por una llave, ya que la repetición de cobertura puede generar ciclos en el grafo de relaciones entre las llaves y eso en el código original retorna NO SE PUEDE. Entonces la lógica dentro de la función del DFS en la detección de ciclos debe ser adaptada para evitar falsas detecciones o patrones incorrectos de apertura.

Escenario #2: "*Se pueden diseñar estructuras en forma de ele (L)s*"

III. Nuevos retos que presupone este escenario #2

En el escenario donde se pueden diseñar estructuras en forma de (L), el impacto principal radica en la disposición específica de las cajas en forma de "L", lo que implica ajustes en la relación de contención entre las llaves. Esta disposición especial puede generar conexiones más complejas en el grafo, ya que las cajas en forma de "L" pueden tener interacciones muy específicas en las cuales puede afectar al algoritmo de recubrimiento del grafo al tener una posibilidad mayor de tener ciclos. La detección de ciclos y la construcción del grafo deben adaptarse para reflejar correctamente estas estructuras en forma de "L", asegurando que el programa pueda manejar la disposición única de las cajas de manera precisa.

IV. Cambios a la solución para adaptarse al escenario #2

Las conexiones en el grafo deben reflejar de manera fiel las relaciones entre las llaves. Esto se logra al añadir lógica adicional para identificar las posiciones donde se podrían formar "L" y establecería conexiones adicionales con las cajas relevantes. En otras palabras, ahora no solamente tenemos que tener en cuenta que otras cajas contienen los sets de llaves (osea sus intersecciones) si no que también tenemos que tener en cuenta si se intersectan estas estructuras en forma de "L" con ese set.

Esto implica un cambio en la manera de calcular las areas, donde ahora tambien se incluye la posibilidad de que estas tengan forma de Ls, y a partir de esto se construye el grafo y se detectan encuentros de diferentes figuras con otras por medio del grafo, para el cual se verificaran los ciclos y ordenes topologicos ya que se cuenta con la estructura fiel del grafo que se propuso inicialmente.

Por otro lado, en la detección de ciclos, la función DFS debe ser modificada para manejar la complejidad adicional introducida por las estructuras en forma de "L", para garantizar una detección precisa de ciclos en el grafo que refleje la disposición única de las cajas en forma de "L". Para adaptar nuestro algoritmo DFS a este escenario

específico, se deben considerar las intersecciones particulares. Durante el recorrido, la función DFS debe ser capaz de reconocer y seguir las conexiones que forman estas estructuras únicas y evitar falsas detecciones o no detectar correctamente ciclos que involucren cajas dispuestas en forma de "L". Para eso se podría, mirar cada posición en la matriz de cajas, luego conectar la caja actual con las otras cajas en la misma fila y columna, evitando conexiones innecesarias. Tomando en cuenta que cuando se encuentren cajas en forma de "L", conectemos la caja actual con las otras cajas que forman el "L".