**Universidad Nacional Autónoma de México**
**Facultad de Ciencias**
**Análisis de Algoritmos | 7083**
Tarea 4 : | Greedy y flujos
Sosa Romo Juan Mario | 320051926
21/10/24

1. **Un algoritmo glotón para regresar el cambio de n unidades usando el mínimo número de monedas es el siguiente: Dar al cliente una moneda de mayor denominación, digamos d. Repite lo anterior para regresar el cambio de n-d unidades.**

   **Para cada una de las siguientes denominaciónes, determina si el algoritmo greedy antes mecionado mínimiza el número de monedas para dar el cambio. Si es así pruébalo, y si no lo es muestra un contraejemplo.**

   Esta probablemente no salio, si quiere no la califique pero la intente por si viene en el examen :v

   (a) **Monedas de Estados Unidos 50, 25, 10, 5 y 1 centavos**

   Demostración

   Seguramente no me va a salir porque demostrar que un algoritmo greedy funciona no es tan sencillo. Pero vamos a intentarlo.

   Voy a basar esta demostracion en la "Guide to Greedy Algorithms" de la Universidad de Stanford para la clase CS161 del 2013; basicamente podemos intentar la prueba por "Greedy Stays Ahead" que basicamente es probar que nuestro algoritmo es al menos tan bueno como el optimo o podemos intentarlo con "Exchange Argument" que es probar que podemos transformar cualquier solucion optima en la solucion que conseguimos con nuestro algoritmo.

   En este caso, vamos a intentar con "Greedy Stays Ahead", comienza definiendo mi solucion, voy a llamar a $G =< g_1, g_2, g_3, g_4, g_5 >$ como la solucion que encuentra mi algoritmo greedy siendo cada uno de los $g_i$ la cantidad de monedas de esa denominación que se necesitan para dar el cambio, por otro lado voy a definir $O =< o_1, o_2, o_3, o_4, o_5 >$ como la solucion optima, siendo cada uno de los $o_i$ la cantidad de monedas de esa denominación que se necesitan para dar el cambio, la idea es demostrar que $|G| \leq |O|$ o dicho de otra forma que la suma de cada una de sus coordenadas es menor o igual a la suma de las coordenadas de $O$.

   Ahora se viene lo chido, tenemos que demostrar que nuestro algoritmo siempre se queda adelante (o dicho de mejor manera al menos no se queda atras) vamos a hacer una pseudoinduccion para demostrarlo.

   Vamos a comenzar la induccion, sobre el tamaño de la solucion, es claro que si tenemos $n \leq 4$ entonces nuestro algoritmo greedy es optimo, ya que solo puede tomar 4 monedas de uno a lo mas, la solucion optima no puede ser mejor que eso, es decir $g_1 \leq o_1$

   Ahora si $5 \leq n \leq 9$ nuestro algoritmo greedy comenzara por restarle una de 5, y tras esto se quedara con un problema de tamaño a lo mas 4, por lo que por hipotesis de induccion sabemos que nuestro algoritmo greedy es optimo, esto es, $g_2 \leq o_2$.

   Si $10 \leq n \leq 14$ nuestro algoritmo greedy comenzara por restarle una de 10, y tras

esto se quedara con un problema de tamaño a lo mas 4, por lo que por hipotesis de induccion sabemos que nuestro algoritmo greedy es optimo, esto es, $g_3 \leq o_3$.

Si $15 \leq n \leq 19$ nuestro algoritmo greedy comenzara por restarle una de 10 y una de 5, y tras esto se quedara con un problema de tamaño a lo mas 4, por lo que por hipotesis de induccion sabemos que nuestro algoritmo greedy es optimo, esto es, $g_3 \leq o_3$.

Si $20 \leq n \leq 24$ nuestro algoritmo greedy comenzara por restarle 2 de 10 y se quedara con un problema de tamaño a lo mas 4, por lo que por hipotesis de induccion sabemos que nuestro algoritmo greedy es optimo, esto es, $g_3 \leq o_3$.

Si $25 \leq n \leq 29$ nuestro algoritmo greedy comenzara por restarle una de 25 y se quedara con un problema de tamaño a lo mas 4, por lo que por hipotesis de induccion sabemos que nuestro algoritmo greedy es optimo, esto es, $g_4 \leq o_4$.

Lo mismo pasara para casos hasta llegar a 49 (49-25=24 que ya es un caso anterior y es optimo), es decir caera en uno de los casos anteriores, todo eso para demostrar que $g_4 \leq o_4$.

Finalmente si $50 \leq n$ nuestro algoritmo greedy comenzara por restarle k monedas de 50, y se quedara con algun subproblema de tamaño a lo mas 49, por lo que por hipotesis de induccion sabemos que nuestro algoritmo greedy es optimo, esto es, $g_5 \leq o_5$.

Importante para este paso es notar que al ser multiplos unos de otros muchos casos en realidad son medio redundantes, sabemos que si algo le podemos restar una de 50 entonces le podemos restar 2 de 25, 5 de 10, 10 de 5 o 50 de 1, pero para cada una de estas otras soluciones se pasan en cantidad de monedas, cosa que voy a mostrar no pasa siempre, ademas una solucion optima del problema contiene solucioens optimas de sus subproblemas.

Ahora si vamos a demostrar que es optimo, para ello vamos a intentarlo por contradiccion:

Digamos que $|G| > |O|$ esto pasaria si solo si $\exists g_i > o_i$ para algun $i$, sin embargo, como mostramos arriba nuestro algoritmo greedy siempre se queda adelante (o mas bien no se queda atras) por lo que esto no puede pasar, por lo que $|G| \leq |O|$ y por lo tanto nuestro algoritmo greedy es optimo.

(b) **Monedas Inglesas 30, 24, 12, 6, 3, 1, 1/2 y 1/4 peniques**

Contraejemplo

En este caso es claro que existe un contraejemplo en donde el algoritmo no funciona, por ejemplo, si se tiene que regresar 48 peniques, el algoritmo daría 30, 12, 6, lo cual no es la mejor opción, ya que se pueden dar 24, 24.

Si se intentara demostrar con la idea del 1a, llegarias a la contradiccion durante la induccion, ya que las soluciones optimas de los subproblemas no siempre lleva a una solucion optima.

(c) **Monedas Portuguesas 1, 2.5, 5, 10, 20, 25, 50 escudos**

Contraejemplo

En este caso también es obvio que no va a funcionar el algoritmo glotón, ya que si por ejemplo se quiere dar el cambio de 40 unidades, primero se dara una moneda de 25, una de 10 y una de 5, en total 3 monedas. Sin embargo, si se diera 2 monedas de 20, se tendría un total de 2 monedas, lo cual es menor que 3.

Si se intenta seguir la idea de la demostración 1a, llegariamos a una contradiccion durante la induccion, asi demostrando que el algoritmo glotón no es optimo.

(d) **Monedas marcianas, 1, p, $p^2$, ..., $p^k$, con $p > 1$ y $k \geq 0$**

Demostracion

Igualmente esta no va a salir pero es una induccion, se parece bastante al 1 porque son multiplos con buena distancia pero ocupo mas lineal para probarlo fuertemente, vamos a dejar a p fijo y k puede ser basicamente cualquier cosa que cumpla.

**Caso base:** $n = 1$

Si el monto es de 1 moneda, entonces el algoritmo va a devolver 1 moneda, claramente es optimo ya que es el unico camino posible.

**Hipótesis inductiva:** $m \leq n$

Supongamos que el algoritmo es optimo para $m \leq n$ monedas.

**Paso inductivo:** $n + 1$

- El algoritmo selecciona la moneda de mayor valor que no exceda el monto restante. Sea esta moneda $p^i$ con $0 \leq i \leq k$ y $p^i \leq n + 1$.
- Tras seleccionar esa moneda el monto restante es $n + 1 - p^i$, con $0 \leq r \leq p^i$.
- Por hipótesis inductiva, el algoritmo es optimo para $r \leq n$ monedas.

No se muy bien como formalizar que el algoritmo es optimo, esencialmente como tenemos que $p^{i+1} = p^i * p$ entonces si no elegimos la moneda mas grande para igualar su valor habra que elegir al menos $p$ monedas de $p^i$ para igualar su valor, lo cual es peor que elegir una sola moneda de $p^{i+1}$. En este caso la solucion optima contiene soluciones optimas de subproblemas..

2. **Construya el árbol de Huffman para codificar el siguiente texto:**

**"El azote, hijo mío, se inventó para castigar afrontando al racional y para avivar la pereza del bruto que carece de razón; pero no para el niño decente y de vergüenza que sabe lo que le importa hacer y lo que nunca debe ejecutar, no amedrentado por el rigor del castigo, sino obligado por la persuasión de la doctrina y el convencimiento de su propio interés."**

No voy a explicar el algoritmo de Huffman, pues se vio en clase pero voy a hacer el procedimiento y luego mostrar con un árbol de Huffman online que esta bien hecho.

Creamos el arbol de Huffman

- **Paso 1:** Contamos la frecuencia de cada letra en el texto. (puede cambiar un poquito si consideras tabuladores o si yo conte mal xd)

␣ : 65
e : 39
a : 34
o : 27
r : 25
n : 21
i : 17
l : 15
t : 13
d : 13
c : 12
p : 11
s : 9
u : 9
v : 5
g : 5
z : 4
, : 4
m : 4
y : 4
b : 4
q : 4
ó : 3
h : 2
j : 2
E : 1
í : 1
f : 1
; : 1
ñ : 1
ü : 1
é : 1
. : 1

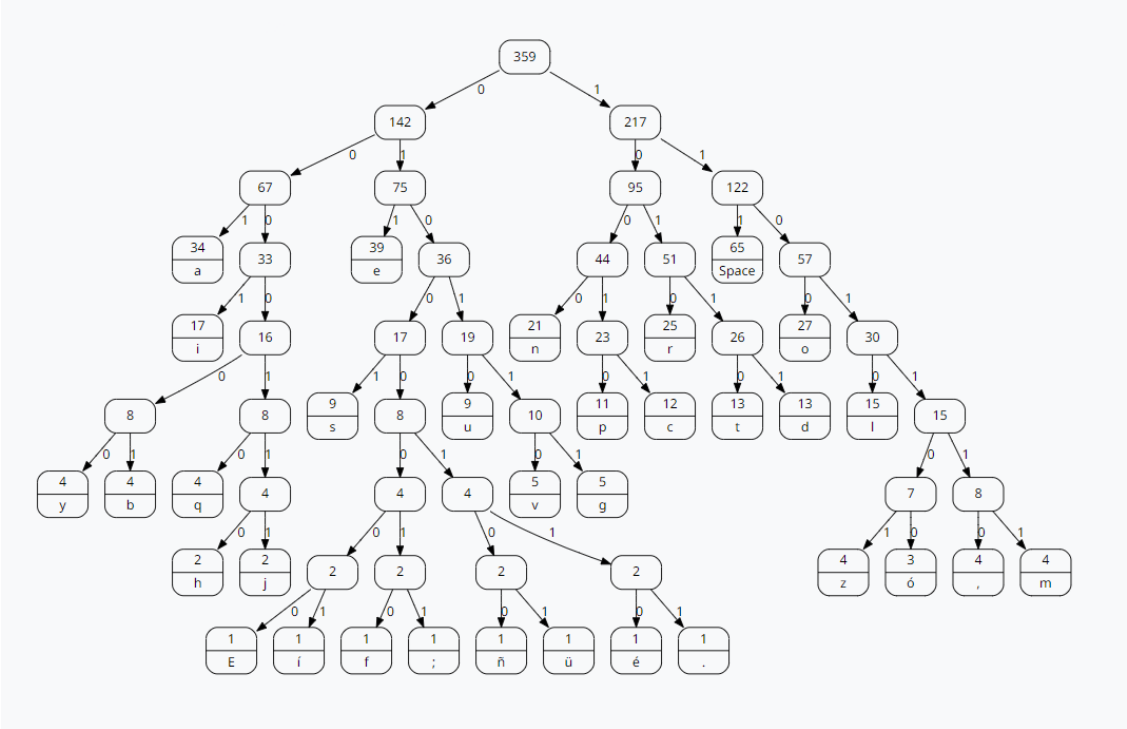
- **Paso 2:** Creamos una lista con los nodos de cada letra y su frecuencia. (este paso literalmente solo es hacer eso entonces no muestro nada)
- **Paso 3:** Tomamos 2 arboles con las frecuencias mas bajas y los unimos en un nuevo arbol con la suma de las frecuencias, la raiz de este nuevo arbol es la suma de las frecuencias y los hijos son los arboles que unimos. Ademas, se etiqueta cada rama con un 0 si esta a la izquierda o un 1 si esta a la derecha, (este paso es el mas largo

y tedioso, asi que solo muestro el resultado final)

- **Paso 4:** Repetimos el paso 3 hasta que solo quede un arbol.

No se si no se podia pero yo utilice un graficador en linea, igualmente el link del graficador es este y el resultado es este:

**NOTA:** El graficador no le importa tanto si es izquierda o derecha al a hora de mostrar el resultado grafico (por eso aveces pone 0 a la derecha) pero internamente si lo esta haciendo solo lo dibuja al reves.




Pero bueno por si acaso lo explico un poquito, hasta abajo vemos que los de frecuencia 1 se empezaron uniendo entre si generando arboles con raiz 2, a su vez se unieron entre si para generar arboles con raiz 4, aveces, cuando no hay arboles con la misma raiz, se toman los 2 de menor raiz digamos 8 y 9 se juntan para una raiz 17, y asi sucesivamente hasta que solo queda un arbol de raiz 359.

Ahora, como mencionamos ir a la izquierda desde la raiz agrega un 0 a la codificacion del caracter y a la derecha un 1, entonces, si queremos saber la codificacion de una letra, simplemente seguimos el camino desde la raiz hasta la letra y anotamos los 0s y 1s que tomamos, este camino es unico aunque la codificacion no lo sea (existen varias codificaciones de Huffman para este texto). Entonces por ejemplo el espacio tiene 111 como codificacion mientras que el . tiene una codificacion de 01000111

3. ***Mei Hua Zhuang* es una técnica de enfrentamiento de Kung Fu, que consiste en n postes grandes parcialmente hundidos en el suelo, con cada poste $p_i$ en la posición $(x_i, y_i)$. Los estudiantes practican técnicas de artes marciales pasando de la parte superior de un poste a la parte superior de otro poste. Pero para mantener el equilibrio, cada paso debe tener más de $d$ metros pero menos de $2d$ metros. Diseñe un algoritmo eficiente para encontrar si es que existe una ruta segura desde el poste $p_s$ al poste $p_t$.**

Crear grafo con ordenamiento y BB

La primera idea es que vamos a querer un grafo que vaya conectando en base a la condicion de distancias, pero lo vamos a hacer con cuidado.

Lo primero que vamos a hacer es ordenar a nuestros puntos por coordenada x, esto nos va a permitir que al buscar aquellos que cumplan la condicion de distancia no tengamos que buscar a todos si no a una fraccion lineal de n, conseguir esta lista ordenada toma $O(n\ logn)$

Ahora vamos a crear un grafo con n nodos, cada nodo va a ser conectado si sigue la condicion de distancia con otro nodo, pero tambien tenemos la lista ordenada por x lo que nos permite que no tengamos que comparar a todas las parejas si no comenzando desde el nodo actual hasta aquellos que cumplan la condicion de distancia, por tanto digamos si estamos checando el nodo $p_5$ en la lista ordenada y vemos que el $p_6$ ya tiene distancia mayor a $2d$ entonces no tiene sentido seguir buscando, en el resto, en escencia estamos haciendo una busqueda binaria en la lista ordenada, ademas usas la otra reestriccion de que la distancia debe ser mayor a $d$ para acortar de ambos lados, como es BB y tiene que hacerlo con 2 reestricciones es 2 $logn$, con n nodos entonces hacemos esto $O(n\ logn)$ para este paso.

Ahora tenemos que buscar el camino, esto es relativamente sencillo ya que podemos hacer un BFS, sobre el grafo que creamos, si es que encontramos un camino que llegue al nodo $p_t$ entonces si existe un camino seguro, si no entonces no existe, el BFS toma $O(|E|+|V|)$ sustituyendo sabiendo que $|E| \leq n^2$ (esto es claro si lo piensas porque si un vertice ya tiene $n^2-1$ vecinos estos a su vez no pueden todos tener un numero cuadratico de vecinos pues necesitan mantenerse a la distancia pefecta del primer vertice y del resto entre todos) y $|V| \in O(n)$ entonces el BFS toma $O(n)$

Por lo tanto el algoritmo toma $O(n\ logn)$

4. **El juego "sube y baja" tiene un tablero de n celdas, donde se busca viajar de la celda 1 a la celda n. Para moverse, un jugador lanza un dado de seis caras para determinar cuántas celdas debe avanzar. Este tablero también contiene rampas y escaleras que conectan ciertos pares de celdas. Un jugador que cae en una rampa cae inmediatamente a la celda en el otro extremo. Un jugador que cae en una escalera viaja inmediatamente hasta la celda en la parte superior de la escalera. Suponga que ha manipulado el dado para que tenga el control total del número de cada lanzamiento. Proporciona un algoritmo eficiente para encontrar el mínimo número de lanzamientos de dados para ganar.**

### Usando BFS

Aunque queria usar un algoritmo gloton la verdad es que no creo que sea conveniente, por ejemplo si tenemos en un turno una escalera muy grande, ignoramos el resto de posibilidades y la usamos, pero esta nos puede dejar en un lugar peor, por ejemplo digamos que puedes subir mucho si tiras 1 y en el resto normal, digamos que subes a la mitad, el greedy lo toma y en ese nivel solo hay normal hasta el final, en cambio en el primer nivel si tirabas 2 veces 6 habia una escalera que te llevaba al final, ya no puedes acceder a esta mas que si tiras otros 11 en el greedy.

Otra alternativa a considerar seria ordenar las escaleras y rampas de mayor a menor y de menor a mayor respectivamente, aunque esto quizas lleve a la solucion optima, no tiene tan buena complejidad, piensa que si por ejemplo hay n escaleras por nivel de tamaño 1, el ordenarlas nos llevaria a una complejidad de $O(n(n\ logn)) = O(n^2\ logn)$ pues en cada

nivel ordenas las escaleras (podrias mejorar esto si solo ordenas las que estan adelante pero me parece sigue quedandose en cuadratico en el peor caso entonces mejor ir a la segura).

Por lo tanto la mejor opcion es usar BFS, pues este nos garantiza la solucion optima, ademas podemos terminar antes si llegamos al final, pues el BFS nos garantiza que si llegamos al final lo hacemos en el menor numero de pasos posibles.

Ahora para esto, podemos usar un grafo, podriamos hacerlo sin el pero es mas simple, cada celda $i$ del tablero es un nodo y desde cada celda $i$ podemos llegar a $i+1, i+2, i+3, i+4, i+5, i+6$ si no hay escaleras o rampas, si las hay, entonces desde $i$ podemos llegar a la celda a la que nos lleva la escalera o rampa.

Ahora pasar de una celda a otra (nodo a nodo) tiene un costo fijo de 1 lanzamiento de dado, por lo que podemos usar BFS para encontrar el camino mas corto, usando una cola para realizarlo, empezamos en la celda 1, aparte de la cola usamos un arreglo de visitados, para no volver a visitar una celda (aqui no hay pesos entonces no importa si ya pudimos llegar) vamos marcando celdas por lo mismo, aqui mismo podemos ir guardando la cantidad de tiros que nos costo llegar a este nodo.

El BFS trabaja por niveles, su complejidad es de $O(|V| + |E|)$ en este caso sabemos que la cantidad de vertices es $n \times n$ pues es un tablero de $n$ celdas y cada celda puede tener a lo mucho 6 vecinos, por lo que $|V| \in O(n^2)$ y $|E| \in O(n^2)$, por lo que la complejidad del BFS es $O(n^2 + n^2) = O(n^2)$, por lo que es relativamente eficiente.

Funciona porque creamos un grafico no ponderado que simula el tablero, y BFS nos da el camino explorando rutas mas cortas primero (rutas con menos lanzamientos).

5. **Supongamos que tenemos un conjunto de $n$ ciudades $c_1, \ldots, c_n$ y una tabla $D[1, \ldots n, 1, \ldots n]$ tal que $D[i, j]$ es la longitud de una carretera que une a la ciudad $c_i$ con la ciudad $c_j$. (este valor puede ser $\infty$ si no hay carretera entre las ciudades). Encuentre un algoritmo eficiente que encuentre la ruta más corta entre las ciudades $c_1$ y $c_n$ tal que dicha ruta no pasa por mas de $k$ ciudades distintas (a $c_1$ y a $c_n$). Justique su respuesta.**

PD con Floyd Warshall.

Yo habia hecho un algorritmo bonito usando una matriz adicional de tamaño $n \times k$ en donde iba poniendo la distancia mínima de $c_1$ a $c_i$ pasando por $j$ ciudades, pero encontre Floyd Warshall y creo que funciona mejor.

Ya tenemos la matriz $D = dist[i][j][0]$ donde $dist[i][j][0]$ es la distancia de $c_i$ a $c_j$ sin pasar por ninguna ciudad (si se considera la ciudad inicial como 1 entonces haremos solo hasta llegar a dist[i][j][k-1]).

Ahora, para cada $t$ de 1 a $k$ (o hasta $k-1$ si se considera la ciudad inicial como 1) haremos lo siguiente:

$$dist[i][j][t] = \min(dist[i][j][t], dist[i][l][t-1] + dist[l][j][t-1])$$

Esto esta como curiosito pero lo que estamos haciendo es considerar el camino mas corto desde $c_i$ a $c_j$ pasando por $t$ ciudades intermedias y ver si es mejor que el camino mas corto que ya teniamos.

Escencialmente estamos encontrando el camino mas corto entre todos los pares de vertices cuya longitud es a lo mas $k$.

El algoritmo de Floyd Warshall tiene complejidad $O(n^3)$ esto pues tiene 3 ciclos anidados, el primero recorre con t de 1 a $k$, que en el peor caso es $n$, el segundo recorre usando i e itera sobre todos los vertices, finalmente el tercer bucle recorre usando j e itera sobre todos los vertices, esto no es terrible pues estamos pidiendo mas de lo que pedimos usualmente y por ejemplo Dijkstra tiene complejidad $O(n^2 \log n)$ en el peor caso aun sin considerar el $k$.

Para recuperar el camino podemos ir guardando ademas del peso de la arista que nos lleva a $j$ desde $l$ en la matriz $dist$ tambien el vertice $l$ que nos lleva a $j$ desde $l$ y asi podemos reconstruir el camino.

6. **El profesor López tiene 2 hijos los cuales no se llevan nada bien. Los chiquillos se odian tanto que no sólo se niegan a caminar juntos a la escuela, si no que además se niegan a caminar en cualquier acera en la que el otro hermano haya puesto pie ese día. Los chiquillos no tienen problemas con que sus caminos coincidan en algunas esquinas. Afortunadamente, tanto la casa del profesor como la escuela están en una esquina, fuera de eso el profesor no está seguro si será posible meter a los 2 hijos en la misma escuela. Muestre cómo modelar el problema de decidir si es posible enviar a los 2 hijos a la misma escuela como un problema de flujos.**

Modelando el problema como un problema de flujos:

Comenzamos por modelar las intersecciones de las calles como nodos, debido a que pueden pasar ambos hijos no tienen restricciones, ademas, el nodo de la casa del profesor es el nodo fuente y el nodo de la escuela es el nodo sumidero.

Ahora agregaremos las aceras como aristas, diremos que tienen capacidad de 1 pues si ya paso uno de los 2 el otro no puede usarla, unimos las intersecciones con las aceras que los conectan.

Ahora tenemos un grafo dirigido con capacidad en las aristas, los nodos fuentes y sumideros deben tener al menos 2 aristas salientes y entrantes respectivamente para que el flujo pueda pasar por ellos.

Para decidir si es posible que ambos hijos lleguen a la escuela sin violar restricciones, basta con encontrar el flujo máximo en el grafo, si el flujo máximo es 2 entonces es posible que ambos hijos lleguen a la escuela, de lo contrario no es posible, esto porque existen 2 caminos disjuntos entre la casa del profesor y la escuela que puede tomar cada hijo.

Para encontrar el flujo máximo podemos usar el algoritmo de Ford-Fulkerson o cualquier otro algoritmo de flujo máximo.

7. **Supongamos que tenemos un flujo óptimo en una red $N$ con n vértices, (con capacidades enteras) de un nodo fuente $s$ a un nodo destino $t$.**

   (a) **Supongamos que la capacidad de una sola arista $e$ se incrementa en una unidad. De un algoritmo de tiempo $O(n+E)$ para actualizar nuestro flujo. $E$ es el número de aristas de $N$.**

   Usando FF con BFS

   Primero que nada hay que notar que al ya tener un flujo optimo en la red asumo se refiere a maximo pues minimo podria solo no enviar nada y ya.

   Como el flujo ya es maximo, entonces sigue el **Teorema de Flujo Máximo-Corte Mínimo** que dice que si f un flujo circula en una red de flujo con origen s y destino t,

f es el flujo máximo de G si la red residual de G no contiene trayectorias aumentantes, esto nos garantiza que cuando incrementemos una arista en una unidad solo hay una oportunidad para que un camino nuevo se habilite y sera unico. (f*=1 maximo cuando aumentemos la arista)

Tras actualizar la capacidad de la arista, revisamos si existe un nuevo camino aumentante en la red residual, para ello usamos BFS, esto toma $O(|V| + |E|) = O(n + E)$

Si existe este camino aumentante, entonces incrementamos el flujo en una unidad si es que afecta y en el peor de los casos hay que actualizar las capacidades de todas las aristas del camino aumentante, esto toma $O(|V|)$

Por lo tanto, el algoritmo toma $O(n + E)$ en total.

(b) **Supongamos que la capacidad de una sola arista $e$ se decrementa en una unidad. De un algoritmo de tiempo $O(n + E)$ donde $E$ es el número de aristas de $N$.**

BFS

Aqui igual estoy asumiendo que optimo se refiere a maximo porque minimo podria no enviar nada y no hay negativos.

Comenzamos por decrementar la capacidad de la arista $e$ en una unidad. Ahora tenemos 2 casos, si el flujo a traves de la arista $e$ es menor o igual que la nueva capacidad de $e$, entonces el flujo óptimo no se ve afectado. En caso contrario, debemos reducir el flujo en la arista $e$ a la nueva capacidad de $e$.

Ahora debemos checar si podemos enviar más flujo desde $s$ a $t$ (reubicar esa unidad a otro camino). Para esto podemos usar BFS, si encontramos un camino de $s$ a $t$ entonces podemos enviar a lo mas una unidad de flujo por ese camino, pues solo decrementamos la capacidad de una arista en una unidad, si no hay camino acabamos, esto toma $O(|E| + |V|)$.

Ahora puede que necesitemos ajustar las capacidades de las aristas en el camino encontrado, esto toma $O(|V|)$, por lo que el algoritmo toma $O(|E| + |V|) = O(n + E)$.

Es evidente que si es que hay camino y hicimos todo el proceso al final no puede existir otro camino (seria sumarle mas de 1 al flujo y no hay decimales) pues esto nos daria un flujo mayor al optimo, lo cual seria una contradiccion.

8. **(El problema del escape) Una malla de $nxn$ es una gráfica compuesta por $n$ filas y $n$ columnas de vértices cómo se muestra en la figura 1. Denotamos el vértice en la i-ésima fila y j-ésima columna como $(i, j)$. Todos los vértices en una malla tienen exactamente cuatro vecinos, excepto aquellos en la frontera, que son los vertices $(i, j)$, donde $i = 1, n$ o $j = 1, n$.**

**Dados $m \leq n^2$ vértices de arranque $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ en la malla, el problema del escape es decidir si existen $m$ trayectorias ajenas por vértices que conecten a cada vértice de arranque con algún vértice en la frontera (distintos).**

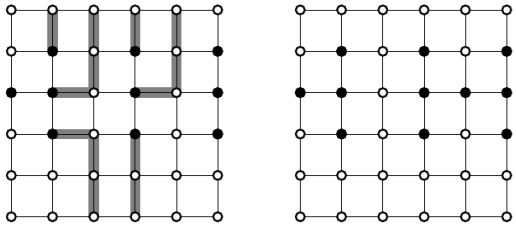


Figure 1: Izquierda malla con escapatoria. Derecha: una sin escapatoria

(a) **Considere una red de flujos cuyos vértices, así como las aristas, tengan capacidades. Esto es, el flujo positivo que entra a cualquier vértice está sujeto a una restricción de capacidad. Muestre que determinar el flujo máximo con capacidades tanto en los vértices como aristas puede ser reducido a un problema de flujo máximo ordinario en una red de flujo con tamaño similar.**

Transformacion agregando aristas

Como ya sabemos, el problema de flujo maximo ordinario solo considera restricciones en las aristas, por lo que vamos a pasar las restricciones de los vertices a una nueva arista, de tal forma que si un vertice tiene una restriccion de capacidad $c$, entonces vamos a agregar una arista de tal forma que la capacidad de esta sea $c$, ahora explico con mas detalle como se haria esto.

Lo primero que vamos a hacer es dividir a nuestros nodos en 2, digamos si tenemos el nodo $v$, entonces vamos a tener 2 nodos $v_{in}$ y $v_{out}$, ahora vamos a agregar una arista de $v_{in}$ a $v_{out}$ con capacidad $c$, donde $c$ es la capacidad del nodo $v$, despues vamos a considerar todas las aristas que salian de $v$ a $v'$ y las vamos a agregar a $v_{out}$ dirigidas a los $v'_{in}$ de tal forma que la capacidad de estas aristas sea la misma que la de las aristas originales (se ve que las aristas que llegan a un vertice se consideran cuando hacemos este proceso para el vertice de salida, osea las aristas que llegan a $v_{in}$ de $v''_{out}$ se van a considerar cuando agregemos las de salida de $v''$).

Después de aplicar la transformación anterior, la nueva red de flujo solo tiene restricciones en las aristas (incluyendo las aristas que conectan los nodos $v_{in}$ y $v_{out}$), por lo que podemos aplicar el algoritmo de flujo maximo ordinario para encontrar el flujo maximo de la red original.

Esta red tiene 2 veces el numero de nodos que la red original, y +n aristas, donde n es el numero de nodos de la red original, pero en general la complejidad de este algoritmo es la misma que la del algoritmo de flujo maximo ordinario.

(b) **Describe un algoritmo eficiente que de solución al problema del escape y analice su tiempo de ejecución.**

Usando inciso anterior mas Ford Fulkerson

De entrada si $m > ((n-2)*4)$ entonces no hay escapatoria, ya que hay mas nodos de arranque que nodos de la frontera, por lo que no se puede conectar a todos los nodos de arranque con la frontera.

Comenzamos usando el inciso anterior de esta pregunta para transformar el problema del escape a un problema de flujo maximo ordinario, para posteriormente usar Ford Fulkerson para encontrar el flujo maximo de la red de flujo resultante.

Entonces, para cada vertice $(i,j)$ lo vamos a descomponer en 2 nodos $(i,j)_{in}$ y $(i,j)_{out}$, y vamos a agregar una arista de $(i,j)_{in}$ a $(i,j)_{out}$ con capacidad 1, igualmente vamos a considerar todas las aristas que salian de $(i,j)$ a $(i',j')$ y las vamos a agregar a $(i,j)_{out}$ dirigidas a $(i',j')_{in}$ de tal forma que la capacidad de estas aristas sea 1, ahora vamos a considerar las aristas que llegaban a $(i,j)$ de $(i',j')$ y las vamos a agregar a $(i',j')_{out}$ dirigidas a $(i,j)_{in}$ de tal forma que la capacidad de estas aristas sea 1, esta parte escencialmente es aplicar $a$ para transformar la malla a una grafica dirigida con capacidades en las aristas.

Sin embargo aun nos falta agregar fuentes y sumideros, para esto vamos a agregar un nodo fuente $S$ (de source) y lo vamos a conectar a todos los nodos de arranque $(i,j)_{out}$ con una arista de capacidad 1, y vamos a agregar un nodo sumidero $T$ (de target) y lo vamos a conectar a todos los nodos de la frontera $(i,j)_{out}$ con una arista de capacidad 1 (los de la frontera ya se dijeron cuales son en la pregunta).

Ahora vamos a aplicar Ford Fulkerson a la red de flujo resultante, y si el flujo maximo es igual a $m$ entonces si hay una escapatoria, de lo contrario no la hay.

El algoritmo FF ya se vio en clase pero sus pasos son los siguientes:

- Encontramos un camino de $S$ a $T$ en la red residual. (Usando BFS toma $O(|V|+|E|)$)
- Seelccionar al min de las capacidades de las aristas del camino encontrado. (Toma $O(|V|)$)
- Actualizar las capacidades de las aristas del camino encontrado, es decir decrementamos el flujo de aristas de camino y actualizamos el reflujo. (Toma $O(|V|)$)
- Repetimos los pasos anteriores hasta que no haya camino de $S$ a $T$ en la red residual.

La complejidad de Ford Fulkerson es $O(f^*(|E|+|V|))$ donde $f^*$ es el flujo maximo, en el peor caso $f^* = m = ((n-2)*4)$ (porque hay 4 fronteras cada una con n-2 salidas, ya que las esquinas no nos sirven) y $|E| \approx n^2$ (porque hay nxn en la malla por 4 aristas originales mas unas pocas de la transformación) ademas $|V| \approx 2n^2+2$ (porque hay 2 nodos por cada vertice de la malla mas 2 nodos de la fuente y el sumidero), por lo que la complejidad del algoritmo maso menos es $O(((n-2)*4)(n^2+2n^2+2)) = O(n(n^2)) = O(n^3)$

9. **Sea $G$ una gráfica con $n$ vértices. Un subconjunto $S$ de los vértices de $G$ es independiente si cualesquiera 2 elementos de $S$ no son adyacentes. En general el problema de encontrar el conjunto independiente de una gráfica, es un problema NP-Completo. En algunos otros casos, este problema puede resolverse eficientemente.**
**Sea $G$ un camino, i.e. los vértices de G son $v_1, v_2, \ldots, v_n$ y $v_i$ es adyacente a $v_{i+1}$ para $i = 1, 2, \ldots, n-1$. Supongamos además que cada vértice tiene asignado un peso un peso $p_i$. Encuentre un algoritmo que resuelva el problema de encontrar el conjunto indpendiente en un camino $G$. Por ejemplo, si $G$ tiene 5 vertices $v_1, v_2, v_3, v_4, v_5$ y sus pesos son 1,8,6,3,6, el conjunto independiente de peso máximo es $v_2, v_5$ y tiene peso 14.**

La verdad no se si entendi este problema pero voy a reutilizar el algoritmo que usamos en la tarea pasada para encontrar al conjunto de personas que invitar a una fiesta. La idea es usar PD para guardar el valor de incluir vs no incluir al nodo.

Usando PD

Primero que nada vamos a usar un arreglo de tamaño $n$ en donde en cada punto tendremos un par (incluir, no incluir) comenzamos por el primer nodo, y en este caso como no hemos procesado ningun otro vamos a llenarlo con ($p_i$,0).

Ahora para cada nodo siguiente del camino vamos a usar la siguiente funcion de recurrencia:

$$\text{PD}[i] = \{(p_i + \text{PD}[i-1].\text{noIncluir}, max(\text{PD}[i-1].\text{noIncluir}, \text{PD}[i-1].\text{Incluir}))\}$$

Escencialmente lo que estamos haciendo es conseguir para cada nodo su valor hasta ese punto de agregarlo o no agregarlo. Al final de todo vamos para conseguir el conjunto independiente, checamos para el ultimo nodo si es mejor incluirlo o no, si lo incluimos, le restamos el valor de su peso al valor total y lo agregamos al conjunto, y nos movemos al nodo anterior, sabiendo que el valor que nos queda despues de restarle tiene que ser igual al valor que teniamos en algun nodo anterior, asi podemos ir consiguiendo el conjunto independiente.

**Ejemplo:**

Usando $G$ tiene 5 vertices $v_1, v_2, v_3, v_4, v_5$ y sus pesos son 1,8,6,3,6.

$$\begin{aligned} DP &= [(1,0),(,),(,),(,),(,)] \\ &= [(1,0),(8,1),(,),(,),(,)] \\ &= [(1,0),(8,1),(7,8),(,),(,)] \\ &= [(1,0),(8,1),(7,8),(11,8),(,)] \\ &= [(1,0),(8,1),(7,8),(11,8),(14,11)] \end{aligned}$$

En este caso el conjunto independiente es $v_2, v_5$ y tiene peso 14. Se puede recuperar como dijimos viendo que 14 es mayor que 11 entonces se incluye, se resta el peso de 5 (14-6=8), pasamos al nodo anterior, buscamos 8 y vemos que no lo incuimos, en el tercer nodo tampoco lo incluimos, en el segundo nodo como el 8 esta en incluirlo lo incluimos, restamos el peso de 2 (8-8=0) y podemos no incuir al resto de los nodos.

Como podemos ver en cada paso se hace una operacion constante pues solo es consultar a su vecino anterior y esto lo hace para cada nodo, por lo que el tiempo de ejecucion es $O(n)$.

10. **Diseña un algoritmo de tiempo $O(|V|+|E|)$ determine si una gráfica dirigida $G = (V, E)$ contiene o no un ciclo.**

### DFS Modificado

Para determinar si una gráfica dirigida $G = (V, E)$ contiene un ciclo, se puede utilizar una modificación del algoritmo de búsqueda en profundidad (DFS).

El algoritmo de DFS primero que nada consiste en recorrer todos los nodos utilizando una pila. Empezando por un nodo, lo agregamos a la pila, lo sacamos marcamos como visitado (podemos usar un arreglo booleano o mas facil agregarle informacion a los vertices, tipo (valor, visitado, listaAdyacencias)) y agregamos su lista de adyacencias a la pila. Despues vamos sacando del tope, si el nodo no ha sido visitado, lo marcamos como visitado y

repetimos con su lista de adyacencias, escencialmente visitando a lo mas profundo que podemos antes de visitar a sus vecinos, este algoritmo tiene complejidad de $O(|V| + |E|)$ cada vertice se visita solo una vez pues se marca como visitado y todas las aristas se visitan pues cuando checamos un nodo tenemos que ver a todos sus vecinos para saber si ya estan o no checados.

Para determinar si una gráfica dirigida $G = (V, E)$ contiene un ciclo, se puede utilizar una modificación del algoritmo de DFS. La modificación consiste en agregar un arreglo de booleanos que nos indique si un nodo ha sido visitado en el recorrido actual (podemos buscar a un nodo por su indice), la idea es la misma, ir marcando vertices pero esta vez solo vamos a ir marcando el recorrido actual y cuando hagamos backtrack tambien quitamos esos nodos de la lista de visitados. Si en algun momento encontramos un nodo que ya ha sido visitado en el recorrido actual, entonces hemos encontrado un ciclo.

Como el algoritmo de DFS tiene complejidad $O(|V| + |E|)$, la modificación, solo tiene que agregar un arreglo de booleanos que tiene complejidad $O(|V|)$, por lo que la complejidad del algoritmo modificado se queda en $O(|V| + |E|)$.

Se puede justificar que funciona pues si existe un camino que contenga un ciclo entonces en algun momento vamos a visitar un nodo que ya habiamos visitado en el recorrido actual, ademas, el no revisitar nodos que ya han sido visitados en el recorrido actual podria parecer que podria no dejarnos ver ciclos pero como estamos usando DFS si el ciclo existe por debajo de un nodo ya visitado entonces lo habriamos cachado en ese otro recorrido y no hay necesidad de volver a visitarlo.