

1. (El problema del escape) Una malla de  $n \times n$  es una gráfica compuesta por  $n$  filas y  $n$  columnas de vértices. 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 vértices  $(i, j)$ , con  $i = 1$ ,  $i = n$ ,  $j = 1$  o  $j = n$ .

Dados  $m \leq n^2$  vértices de arranque  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$  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 de la frontera (distintos).

- 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.

**Respuesta:** Para mostrar que puede ser reducido a el problema de flujo máximo vamos a explicar cómo se va a modelar este problema para que pueda ser representado por una gráfica. Empezamos notando que ya tenemos una malla  $\mathbf{M}$  de tamaño  $n \times n$  la cuál ya puede ser vista como una gráfica, pero además tenemos dos tipos de vértices destacados: los de la frontera (que son las *salidas* o *destinos*) y los nodos de *arranque*. A los nodos de la frontera los llamaremos  $\mathbf{F}$  y a los de arranque  $\mathbf{A}$ .

Construiremos una gráfica  $\mathbf{G}$  la cuál va a tener a todos los nodos y aristas de  $\mathbf{M}$  y otros dos nodos destacados que serán uno de inicio ( $\mathbf{s}$ ) y uno de fin ( $\mathbf{e}$ ). El nodo  $\mathbf{s}$  va a ser adyacente a todos los nodos en  $\mathbf{A}$  y el nodo  $\mathbf{e}$  va a ser adyacente a todos los nodos en  $\mathbf{F}$ . Para terminar con el modelado del problema necesitamos que todos los nodos y aristas tengan un peso que será de 1.

La modelamos de esta forma para que podamos usar el nodo  $\mathbf{s}$  como un punto de partida y  $\mathbf{e}$  como punto final, así estamos obligados a que todas las trayectorias tengan que pasar primero por un nodo en  $\mathbf{A}$  y antes de llegar a  $\mathbf{e}$  tengan que pasar por un nodo en  $\mathbf{F}$ , de este modo representamos que se tomó escapó desde un nodo en  $\mathbf{A}$  hasta uno de  $\mathbf{F}$ . El peso que ponemos en las aristas y nodos es para limitar que las personas pasen una sola vez por la arista o nodo, y necesitamos de este peso para asegurarnos que las trayectorias que estamos buscando sean ajenas por vértices y también por aristas. Cabe destacar el peso en los nodos, ya que si no tuvieran entonces tendríamos una gráfica de flujos como las que se han estado ocupando en clase, pero necesitamos de estos pesos por lo explicado anteriormente.

Podemos decir que nuestra gráfica  $\mathbf{G}$  es una gráfica de flujos *tradicional* por que (aún que en lo único que cambia es en que los vértices tienen un peso) el peso de los vértices solo se utilizará de igual forma que la capacidad de las aristas y del mismo modo se disminuirá en uno cuando estemos utilizando dicha arista. Así que se vuelve un problema de flujo máximo por que debemos buscar el flujo máximo de nuestro vértice  $\mathbf{s}$  a  $\mathbf{e}$ .

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

**Respuesta:** Como vimos en el inciso anterior, tenemos ante nosotros un problema de flujo máximo con la variación de que tenemos un peso en los vértices. Entonces para resolver nuestro problema vamos a describir y modificar un poco el algoritmo de *Ford-Fulkerson* que tome en cuenta dicha variación.

2. Sea  $\mathbf{G} = (V, E)$  una gráfica conexa no dirigida. Dar un algoritmo  $O(V + E)$  para obtener un camino en  $\mathbf{G}$  que atraviese cada arista en  $E$  exactamente una vez en cada dirección. Describe como puedes

encontrar un camino fuera de un laberinto si tienes una cantidad muy grande de monedas.

**Respuesta:** Primero vamos a describir el algoritmo para atravesar dos veces cada arista.

Supondremos que cada nodo  $u$  tiene una lista *vecinos* con una referencia a los vecinos de  $u$ . También tendremos una lista *vecinos-explorados* que contendrá los vecinos que ya no necesitan ser explorados desde  $u$ . Por último utilizaremos una pila *padre*, que contendrá en el tope al último nodo por el que se llegó a  $u$  (empiezan con la pila vacía).

- a) Inicialmente vamos a tomar un nodo cualquiera, digamos  $r$  y empieza siendo el nodo actual.
- b) Para el nodo actual  $u$  vamos a verificar si su lista *vecinos* es igual a su lista *vecinos-explorados*, en caso de que sí:
  - a) Si *padre* ya está vacía, entonces termina el algoritmo.
  - b) Si no es vacía, entonces nuestro nodo actual va a ser el resultado de hacer pop en *padre* (sacamos el nodo de la pila) de  $u$ .

En caso de que aún *vecinos* sea distinto a *vecinos-explorados* vamos a seleccionar un nodo de *vecinos* que no esté en *vecinos-explorados*, digamos  $s$ , metemos a  $s$  en *vecinos-explorados*, hacemos a  $s$  el nodo actual y por último añadimos a  $u$  en la lista *vecinos-explorados* de  $s$ . Repetimos *b*).

Vamos a notar lo siguiente: las aristas solo se recorren cuando nos movemos a un nodo que no está en nuestra lista *vecinos-explorados*, y cuando volvemos hacia nuestro padre (por que ya no tenemos vecinos no explorados). También podemos ver que si se llegó por un nodo  $u$  a un nodo  $s$  entonces ya no nos movemos a  $u$  (desde  $s$ ) a menos que hayamos explorado todos los vecinos de  $s$ , por lo tanto nos movimos por la arista  $(u, s)$  una vez para llegar de  $u$  a  $s$  y otra vez para volver de  $s$  a  $u$ . Y no hay otra ocasión en la que nos movamos por las aristas. Además recorreremos siempre todas las aristas ya que hacemos una búsqueda por profundidad, por lo tanto siempre recorreremos todas y además pasamos por ellas dos veces.

El tiempo que tarda en ejecutarse este algoritmo es  $O(E)$  ya que recorre dos veces todas las aristas.

Ahora para el problema del laberinto basta con aplicar el algoritmo descrito y viendo cada intersección del laberinto como un nodo y cada pasillo entre intersecciones como una arista. También necesitamos una forma de guardar información (la pila *padre* y *vecinos-explorados*) en las intersecciones del laberinto, lo cuál vamos a hacer con ayuda de las monedas de la siguiente forma:

- a) Para los *vecinos-explorados*: Cuando añadimos un nodo a la lista *vecinos-explorados* de una intersección, vamos a representarlo con una moneda puesta en la entrada del pasillo que lleva al nodo que se quiere guardar.
- b) Para la pila *padre*: Cuando queramos guardar un elemento en la pila primero vamos a ver si hay dos o más monedas en alguno de los pasillos de la intersección por la que llegamos, si no hay en ninguno más de dos, ponemos una moneda extra para marcar que es un padre. si ya hay una o más monedas en un pasillo, entonces nos fijaremos en el pasillo, digamos  $c$ , con más monedas y colocaremos en el pasillo que es el nuevo padre una moneda más de las que había en  $c$ .
- c) Para cuando queramos hacer pop de la pila *padre* nos fijaremos en el pasillo  $c$  con más monedas, quitamos las monedas dejando solo una y nos movemos a la intersección que ese pasillo lleva.

El algoritmo se aplica hasta que se llegue a una intersección que tenga una salida.

3. Supongamos que una gráfica  $\mathbf{G}$  tiene un árbol generador de peso mínimo  $\mathbf{T}$  que ya fué calculado. Qué tan rápido se puede actualizar el árbol de peso mínimo si un nuevo vértice con aristas en agregado a  $\mathbf{G}$ ?

**Respuesta:** Nos va a tomar  $O(n \log n)$  siendo  $n$  el número de vértices originales de  $\mathbf{G}$ .

Para mostrarlo primero vamos a describir el algoritmo de **Prim**, que calcula el árbol generador de peso mínimo de una gráfica:

- Primero toma un vértice cualquiera y lo añade al árbol.
- De los nodos adyacentes al árbol les asigna una arista mínima (es la arista de peso mínimo que conecta al vértice con el árbol). Selecciona el vértice con la menor arista y lo añade al árbol.
- Repite el paso anterior hasta que ya no quedan vértices por añadir al árbol.

Para la selección del nuevo vértice con la arista mínima que se va a añadir al árbol, se hace uso de un *min-heap* con lo cuál obtenemos una complejidad de  $O(|E| \log n)$  para este algoritmo. Con esto dicho podemos seguir.

Vamos a unir el nuevo vértice solamente con el árbol generador  $\mathbf{T}$ . No usamos las aristas de  $\mathbf{G}$  ya que solo maximizarían o mantendrían igual el peso de  $\mathbf{T}$ . Lo unimos a  $\mathbf{T}$  y no a  $\mathbf{G}$  para así mantener la conexidad de la gráfica y al mismo tiempo mantenerla con la menor cantidad de aristas posibles sin perder el peso mínimo de la gráfica. El árbol generador  $\mathbf{T}$  tiene actualmente  $n - 1$  aristas, así que al añadir el nuevo vértice con sus aristas (puede a lo más tener  $n$  aristas) nos quedaría una gráfica  $\mathbf{G}_1 = (V_1, E_1)$  tal que  $|V_1| = n + 1$  y  $n \leq |E_1| \leq 2n + 1$ . Al aplicar el algoritmo de **Prim** tenemos entonces que su complejidad de tiempo es  $O((2n + 1) \log(n + 1)) = O(n \log n)$ .

4. Da un algoritmo eficiente para encontrar la longitud del ciclo mínimo de peso negativo en una gráfica.

**Respuesta:** Vamos a utilizar una variación del algoritmo de **Floyd-Warshall**.

Primero vamos a explicar como funciona dicho algoritmo:

Suponiendo que tenemos la matriz de adyacencia  $M$  de nuestra gráfica, donde guardaremos en cada casilla el peso de la arista que las une. También utilizaremos otra matriz  $O$  del mismo tamaño que  $M$ .

- Tomamos  $k = 1$
- Para toda  $i$  y  $j$  hacemos: si  $M_{i,k} + M_{k,j} < M_{i,j}$  entonces  $O_{i,j} = O_{k,j}$  y  $M_{i,j} = M_{i,k} + M_{k,j}$ .
- Si  $k \leq n$  (con  $n$  el número de vértices de  $\mathbf{G}$ ) entonces detenemos el algoritmo, en otro caso hacemos  $k = k + 1$  y repetimos el paso anterior.

Ahora a este algoritmo le haremos la modificación siguiente: vamos a detenernos cuando alguna casilla de la diagonal de  $M$  sea menor a 0, es decir que  $M_{i,i} < 0$  en alguna iteración  $k$ .

De esta forma obtenemos un camino de el vértice  $i$  a sí mismo, es decir un ciclo. Y además lo detenemos en el momento que encontremos uno de peso negativo. Como el algoritmo empieza encontrando caminos que pasan a lo más por un vértice, luego por dos, ya así sucesivamente, va aumentando en cada iteración en uno, entonces así nos aseguramos de que en cuanto encontremos una casilla de la diagonal negativa la  $k$  va a ser la mínima. Así que obtendríamos el tamaño mínimo de el ciclo de peso negativo de longitud mínima.

La complejidad de este algoritmo se mantiene  $O(n^3)$  pero podría detenerse antes por la condición de la casilla diagonal negativa, así si la  $k$  es más chica podríamos decir que es  $O(k * n^2)$ .