Facultad de Ciencias, UNAM
Análisis de Algoritmos
Tarea 4

Rubí Rojas Tania Michelle

26 de enero de 2021

1. *Mei Hua Zhuang* es una técnica de entrenamiento 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.

   Solución: Representamos el conjunto de postes como una gráfica $G = (V, E)$. Cada uno de los vértices de G corresponderán a un poste p_i (cuyas coordenadas son (x_i, y_i)). Por otro lado, las aristas corresponderán a la adyacencia entre los postes que están en el suelo y tendrán un peso w_d de acuerdo a la distancia (en metros) entre un poste y otro.

   Si realizamos una búsqueda BFS desde el vértice p_s y en algún momento del recorrido descubrimos el vértice p_t, entonces existe un camino entre p_s y p_t. Realizándo una modificación para que el camino cumpla con la restricción de la distancia entre los postes, podemos traducir el procedimiento en el siguiente algoritmo:

   - Creamos una cola Q.
   - Agregamos el vértice p_s a la cola Q.
   - Marcamos a p_s como visitado.
   - Mientras Q no esté vacío:
     - Sacamos un elemento de la cola Q, digamos v.
     - Para cada vértice w adyacente a v en G:
       - Si el peso (distancia) entre w y v es mayor que d pero menor que $2d$, entonces:
         - Si w es igual a p_t, entonces regresamos `"Sí existe un camino"`.
         - En otro caso, si w no ha sido visitado, entonces lo marcamos como visitado y lo agregamos dentro de la cola Q.
       - En otro caso, si w no ha sido visitado, entonces lo marcamos como visitado.
   - Regresamos `"No existe un camino"`

   Este algoritmo funciona porque en cada iteración nos aseguramos de seguir un camino donde el peso entre los vértices se encuentra en un rango de $(d, 2d)$; y esto lo logramos gracias al recorrido BFS y una pequeña condición (el de los pesos) para saber cuáles vertices tomar en cuenta y cuáles no. Luego, como el único algoritmo que aplicamos es BFS (modificado por una condición), entonces la complejidad total del algoritmo es $O(V + E)$.

2. El presidente de un país cree que cada ciudad debe de tener acceso a una biblioteca, desafortunadamente el país se vio afectado por un temblor que destruyó todas las bibliotecas y bloqueó todos los caminos que había. Dadas n ciudades numeradas de 1 a n, con m caminos bidireccionales, se dice que una ciudad puede acceder a una biblioteca si tiene una construida o puede trasladarse a otra ciudad que contenga una. Considerando que los costos de reparación de un camino o de construcción de biblioteca son $\texttt{Costo}_c$

y $\texttt{Costo}_b$, respectivamente. Diseña un algoritmo de tiempo $O(n+m)$ que determine qué caminos reparar y qué bibliotecas construir tal que cada ciudad pueda acceder a una biblioteca y el costo sea mínimo.

SOLUCIÓN: Representamos al país como una gráfica $G = (V, E)$. Cada uno de los vértices de G corresponderán a una ciudad y las aristas corresponderán a los caminos bidireccionales que existen entre las ciudades.

Ahora bien, para que un ciudadano tenga acceso a una biblioteca, debemos tener en cuenta dos propiedades importantes:

- La ciudad en la que habita tiene una biblioteca.
- Puede viajar desde su ciudad hasta una que tenga una biblioteca.

Así, tenemos que cada componente conexa de la ciudad debe de tener al menos una biblioteca. Para cada componente, definimos un contador `cc` (será variable global), el cual se encargará de contabilizar el número de vértices en la componente. Luego, tenemos tres posibles casos:

- Si $\texttt{Costo}_c < \texttt{Costo}_b$, entonces el costo mínimo será construir una biblioteca y `cc-1` caminos para conectar a las ciudades (pues este número es suficiente para reparar los caminos que conectan a las ciudades). Como un ciudadano puede acceder a una biblioteca si puede viajar hacia ella, es decir, existe un camino desde su ciudad a la que tiene la biblioteca; entonces en este caso resulta más barato construir una sóla biblioteca (que es más cara) y reparar los caminos entre las ciudades para que puedan llegar a ella.
- Si $\texttt{Costo}_c > \texttt{Costo}_b$, entonces el costo mínimo es construir una biblioteca por cada ciudad. Como un ciudadano puede acceder a una biblioteca si se encuentra en su ciudad, entonces es más económico construir puras bibliotecas (que son las más baratas) en lugar de incrementar el costo reparándo caminos.
- Si $\texttt{Costo}_c = \texttt{Costo}_b$, entonces se pueden aplicar cualquiera de los dos procedimiento anteriores, pues como los costos son iguales, el resultado será el mismo.

Con esta idea, podemos utilizar DFS sobre la gráfica G para contar el número de vértices en cada componente conexa (gracias a que este recorrido genera un árbol por cada componente de G), y luego calcular el costo mínimo (de acuerdo a las consideraciones anteriores). Así, este procedimiento lo podemos traducir como el siguiente algoritmo:

- Inicializamos la variable `costo = 0`.
- Para cada vértice $v_i \in V(G)$ que no ha sido visitado:
  - Inicializamos la variable `cc = 0`.
  - Aplicamos el algoritmo DFS con el vértice v_i.
  - Si $\texttt{Costo}_c < \texttt{Costo}_b$, entonces `costo += Costo_b + Costo_c * (cc-1)`
  - En caso contrario, `costo += Costo_b * cc`
- Regresamos `costo`.

donde el algoritmo DFS se vería de la siguiente forma:

- Marcamos `origen` como visitado.
- Aumentamos al contador `cc` en una unidad.
- Para cada vértice v adyacente a `origen` en G:
  - Si v no ha sido visitado, entonces lo marcamos como visitado.
  - Llamámos recursivamente a DFS con v.

Ahora bien, este algoritmo funciona porque gracias a DFS podemos analizar cada una de las componentes conexas (ya que G no necesariamente debe ser un país donde todas sus ciudades están conectadas) y de acuerdo al número de vértices de cada una podemos calcular el costo mínimo (dependiéndo de los costos de construcción que nos den).

Como estamos usando DFS para calcular el número de vértices en cada componente conexa, entonces esto nos toma en total $O(n+m)$ (pues exploramos todas las aristas y vértices de G); y calcular el costo mínimo nos toma tiempo constante, pues sólo debemos realizar operaciones aritméticas. Por lo tanto, la complejidad total del algoritmo es de $O(n+m)$.

3. La ONU quiere mandar al espacio dos personas a la luna de países distintos. Dada una lista de pares (i, j) donde el $i-$ésimo astronauta es del mismo país que el $j-$ésimo, determina el número de pares posibles a formar.

   SOLUCIÓN: El hint para este problema es determinar el número de países que existen. Para lograr esto, utilizaremos `union-find`. Recordemos que éste es una estructura de datos que modela una colección de conjuntos disjuntos y está basado en dos operaciones:

   - `Find(A)`. Determina a cuál conjunto pertenece el elemento A. Esta operación puede ser usada para determinar si 2 elementos están o no en el mismo conjunto.
   - `Union(A,B)`. Une todo el conjunto al que pertenece A con todo el conjunto al que pertence B, dando como resultado un nuevo conjunto basado en los elementos tanto de A como de B.

   Si bien hay varias formas de implementarlo, usaremos aquella que es conocida como `bosque de conjuntos disjuntos`. Éste es una estructura donde cada conjunto está representado por un árbo con raíz. El representante es, naturalmente, la raíz. Cada nodo de cada árbol apunta únicamente a su padre, a excepción de la raíz (que apunta a sí misma). La función `Find(A)` busca la raíz del árbol en el que se encuentra el elemento A, mientras que la función `Union(A,B)` pone a la raíz del árbol que contiene a B como hijo de la raíz del árbol que contiene a A. De esta forma, cada astronauta será un vértice en el bosque. Como cada par (i, j) de astronautas pertenece al mismo país, entonces podemos agregar una arista entre sus vértices y unir los árboles a los que pertenecen. Así, un árbol del bosque representará a todos los astronautas del mismo país. Contando el número de árboles disjuntos y su tamaño t_p podemos obtener la subdivisión exacta de todos los astronautas. Como tenemos que elegir a dos personas de diferentes países, entonces podemos elegir a los posibles pares con el siguiente algoritmo:

   - Inicializamos `aux = num = 0`.
   - Para cada uno de los valores de `tamaño` t_p:
     - `num += aux * t_p`
     - `aux += t_p`
   - Regresamos `num`

   Finalmente, la implementación del bosque de conjuntos disjuntos en la que `Find(A)` no actualiza los punteros de los padres y en la que `Union(A,B)` no intenta controlar las alturas de los árboles, puede tener (en el peor caso) árboles con altura $O(n)$. En esta situación, ambas funciones nos tomarían tiempo $O(n)$ (se pueden hacer mejoras al algoritmo, pero nos quedaremos con esta). Por otro lado, calcular el número de pares posibles a formar nos toma también $O(k)$, donde k es el número de países. Por lo tanto, la complejidad total del algoritmo es de $O(n)$.

4. Supongamos que tenemos un conjunto de n ciudades $c_1, c_2, \ldots, c_n$, 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 ∞ si no hay carretera entre c_i y c_j). 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 más de k ciudades distintas a c_1 y c_n.

   SOLUCIÓN: Representamos el conjunto de ciudades como una gráfica $G = (V, E)$. Cada uno de los vértices corresponderá a una ciudad c_i. Por otro lado, las aristas corresponderán a las carreteras que unen a las

ciudades entre sí (para hacer la gráfica dirigida, colocamos dos aristas: una de ida y otra de regreso) y tendrán un peso w_d de acuerdo a la longitud de cada carretera. Consideramos además a la tabla D como nuestra matriz de adyacencias, donde cada entrada $D[i, j]$ corresponde al peso w_d de las aristas entre dos ciudades (notemos además que $D[i, j] = D[j, i]$ para no tener que repetir las aristas en la tabla de adyacencias) y si no hay una arista que una a las ciudades, entonces el peso puede ser ∞.

Como queremos encontrar un camino entre dos vértices tal que dicho camino no pase por más de k ciudades diferentes, entonces podríamos recorrer todos los caminos de longitud a lo más k desde c_1 hasta c_n.

Ejercicio incompleto :c

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

   Solución: Sea G una gráfica con un conjunto de vértices $V(G)$ y un conjunto de aristas $E(G)$. Dado el número de aristas, tenemos dos casos:

   - Si el número de aristas de G es menor que $|V(G)|$ entonces puede que tengamos o no un ciclo (pues se requiere que al menos existan tantas aristas como vértices para que necesariamente exista un ciclo). Para comprobar esto, recorremos la gráfica usando DFS: si durante el recorrido nos encontramos con una arista e que tiene un vértice visitado entonces hemos encontrado un ciclo, en caso contrario, no existe algún ciclo.
   - Si el número de aristas de G es mayor o igual que $|V(G)|$ entonces necesariamente tenemos un ciclo.
     - Si G es conexa, entonces recorremos la gráfica usando DFS (iniciándo en un vértice arbitrario). Como la gráfica es conexa, el árbol DFS resultante tendrá todos los vértices de la gráfica, y como un árbol tiene $|V(G)| - 1$ aristas, entonces existirá al menos una arista en G que no está en el árbol DFS de G (pues $|E(G)| \geq |V(G)|$). Esta arista da un ciclo en G.
     - Si G es disconexa, entonces sabemos que alguna de sus k componentes conexas tiene un ciclo (pues seguirá la misma lógica que en el inciso anterior). Así, vamos verificando vértice por vértice hasta encontrar una componente conexa (los vamos marcando como visitados). Una vez que la encontremos, aplicamos DFS y esperamos tener el mismo resultado que en el inciso anterior. Si no lo encontramos, como todos esos vértices ya fueron visitados, entonces seguimos probando este método hasta encontrar la componente que contiene el ciclo.

   Este algoritmo funciona porque en ambos casos usamos DFS para poder buscar el ciclo. Al usar este recorrido, nos aseguramos de no repetir vértices y de buscar todas las componentes conexas que podría tener nuestra gráfica, por lo que siempre buscamos en toda la gráfica el ciclo.

   Por otro lado, la complejidad del algoritmo efectivamente es $O(V)$. Recordemos que DFS nos toma $O(V+E)$ en tiempo. En el primer caso, como $|E(G)| < |V(G)|$, entonces DFS nos toma $O(V + E) = O(V)$ en tiempo. En el segundo caso, recorremos todos los vértices una sóla vez y si nos encontramos con alguna arista distinta de V, entonces ya la habremos visto antes, por lo que vemos más vértices que aristas, y así la complejidad también será de $O(V)$. Por lo tanto, la complejidad total del algoritmo es de $O(V)$.

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

   - 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.

     Solución: Sabemos que si existe un corte mínimo en el que la arista e no se encuentra, entonces no se puede aumentar el flujo máximo; por lo que no existirá ningún camino aumentante en la red residual. En caso contrario, posiblemente podemos aumentar el flujo en 1. Para averiguarlo, realizamos una nueva iteración del algoritmo `Ford-Fulkerson`. Si existe un camino aumentante, entonces aumentará en esta iteración (sino, el flujo no cambia). Como el flujo aumenta estrictamente, entonces en una sóla iteración del ciclo `while` en la línea 3 (ver 2) de `Ford-Fulkerson`, el flujo aumentará en una unidad. Para encontrar un camino aumentante utilizamos BFS, el cual nos toma $O(V + E') = O(n + E)$ en tiempo.

- 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.

  SOLUCIÓN: Si el flujo de la arista e ya estaba al menos 1 unidad por debajo de la capacidad, entonces no cambia. En caso contrario, debemos buscar un camino de s a t que contenga a e (pues aún tiene capacidad residual positiva), esto lo haremos realizándo BFS (lo cual nos toma $O(V + E) = O(n + E)$). Decrementamos el flujo de cada arista en ese camino en 1. Esto disminuye el flujo total en 1 (pero no afecta el flujo máximo). Luego, ejecutamos una iteración del ciclo `while` del algoritmo `Ford-Fulkerson` (el cual nos toma $O(n + E)$). Por lo argumentado en el inciso anterior, el flujo aumenta estrictamente, lo que implica que no encontraremos ningún camino que aumente, o bien, aumentaremos el flujo en 1 y terminamos.

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

   SOLUCIÓN: Representamos el mapa del pueblo del profesor Protón como una gráfica dirigida $G = (V, E)$, donde los vértices de G serán las esquinas y existirá una arista entre dos vértices en caso de que haya una banqueta que las une (creamos una arista también en la dirección contraria como lo hicimos en clase para los problemas de flujos). Después, definimos la capacidad de flujo de cada arista de la red de flujos como $c(u, v) = 1$, pues los chiquillos no quieren pisar la misma banqueta que su hermano. El nodo fuente s será la casa del profesor Protón, mientras que el nodo destino t será la escuela (ambos puntos del mapa son vértices ya que se encuentran en esquinas).

   El flujo de una arista $f(u, v)$ será de una unidad en caso de que un niño vaya por la banqueta que une las esquinas u, v. Por la restricción de capacidad $0 \leq f(u, v) \leq c(u, v)$, entonces estamos modelándo correctamente el hecho de que los chiquillos no usen la misma banqueta. Si existieran dos caminos distintos de la casa del profesor Protón a la escuela, entonces el flujo de la red tendría que ser mayor o igual a 2, ya que sería posible asignar un flujo de una unidad ($f(u, v) = 1$) para cada arista. Por otro lado, si sólo existiera un camino distinto de la casa a la escuela, entonces esto implicaría que existe un puente, digamos la arista (x, y), que al eliminarla dejaría desconectada a la casa de la escuela. Al haber definido la capacidad de cada arista de una unidad, entonces en particular tenemos que $c(x, y) = 1$ y así el máximo flujo que puede llegar al vértice x sería de una unidad. Luego, por la conservación de flujos, tenemos que el flujo desde la casa hasta el vértice y puede ser a lo más de una unidad, es decir,

$$f = \sum_{v \in V} f(s, y) \leq 1$$

   Por consecuente, tenemos que el hecho de determinar el flujo máximo y ver que éste sea mayor o igual a 2 le servirá al profesor para poder averiguar si puede llevar a los chiquillos a la misma escuela o no.

   Ahora bien, el algoritmo que resuelve el problema de encontrar el flujo máximo en una red de flujos (gráficas dirigidas y acíclicas) es el de `Ford-Fulkerson`, cuyo método general está basado en los caminos aumentantes:

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

Figura 1: Cormen, Introduction To Algorithms pag. 715

En donde la red residual mencionada es de manera intuitiva aquella que consiste de las aristas con capacidades que representan cómo han cambiado los flujos de las aristas de la gráfica G y un flujo f. De manera formal, la capacidad residual en una red de flujos $G = (V, E)$ con origen s y destino t, un flujo f en G y un par de vértices $u, v \in G$ se define de la siguiente forma:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in E \\ f(u,v) & (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Y así la red residual de G inducida por el flujo f será $G_f(V, E_f)$ donde

$$E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$$

Que es lo que hacíamos en clase al ir actualizándo el flujo de una arista, luego reduciéndo su capacidad y poniéndo el flujo restante en el otro sentido.

Finalmente, un pseudocódigo de la implementación de la técnica de `Ford-Fulkerson` sería el siguiente:

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

Figura 2: Cormen, Introduction To Algorithms pag. 724

Como lo indica el `Cormen`, la complejidad de la implementación dependerá de la línea 3 (donde se buscan los caminos aumentantes). Si se emplean búsquedas BFS o DFS, entonces cada iteración de la línea 3 usará tiempo $O(E)$, al igual que la inicialización de las líneas $1-2$. Luego, como el ciclo que corresponde a las líneas $4-8$ se ejecuta a lo más $|f^*|$ veces, donde f^* denota el flujo máximo en la red de flujos transformada, pues en cada iteración el flujo se ve incrementado al menos en una unidad. Así, la complejidad de todo el algoritmo sería de $O(E|f^*|)$ tiempo.

8. It is a natural to apply graph models and algorithms to spatial problems. Consider a black and white digitalized image of a maze; white pixels represents open areas and black spaces are walls. There are two spacial white pixels: one is designated the entrance and the other is the exit. The goal in this problem is to find a way of getting from the entrance to the exit, as ilustrated in Figure 1. Given a $2D$ array of black and white entries representing a maze with designated entrance and exit points, find a path from the entrance to the exit, if one exists.

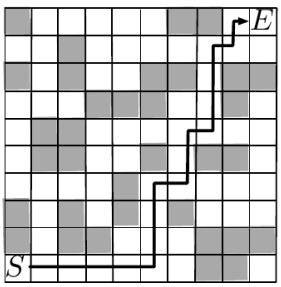

Figura 3: A shortest path from entrance to exit.

SOLUCIÓN: Representamos el laberinto como una gráfica $G = (V, E)$. Cada uno de los vértices de G corresponderán a un píxel blanco (zona abierta) y estarán indexados de acuerdo a sus respectivas posiciones en la matriz $2D$ (es decir, el vértice $v_{i,j}$ corresponde al píxel blanco en la entrada (i, j) del arreglo $2D$). Por otro lado, las aristas corresponderán a la adyacencia entre los píxeles blancos (es decir, dos píxeles blancos que sean adyacentes en la matriz $2D$ estarán unidos por una arista en la gráfica G).

Sea p_i y p_f la entrada y salida del laberinto, respectivamente. Si realizamos una búsqueda BFS desde el vértice p_i y en algún momento del recorrido descubrimos el vértice p_f, entonces existe un camino entre p_i y p_f. Para encontrar dicho camino, basta con ir guardando cada uno de los vértices que lo van formando en una lista. De esta manera, podemos traducir este procedimiento en el siguiente algoritmo:

- Creamos una cola Q y una lista P.
- Agregamos p_i a la cola Q.
- Marcamos p_i como visitado.
- Mientras Q no esté vacío:
  - Sacamos un elemento de la pila Q, digamos v.
  - Agregamos a v a la lista P.
  - Para cada vértice w adyacente a v en G:
    - Si w es igual a p_f, entonces agregamos p_f a la lista P y regresamos esta lista. Terminamos.
    - En otro caso, si w no ha sido visitado, entonces marcamos como visitado a w e insertamos w dentro de la cola Q.
- Regresamos la lista vacía (pues no encontramos un camino).

Este algoritmo funciona debido a que en cada iteración garantizamos tener los vértices que pertenecen al camino entre p_i y p_f (si es que existe) al utilizar BFS para recorrer la gráfica G y en el transcurso ir guardándo los vértices del camino que vamos recorriéndo en una lista P. Además, BFS encuentra el camino más corto: como exploramos todos los hijos inmediatos antes de pasar a los nietos, entonces garantiza que todos los nodos a una distancia determinada del nodo padre se exploren al mismo tiempo.

Ahora bien, sabemos que la complejidad de BFS es $O(V + E)$ y como las únicas operaciones extra que realizamos son la creación de una lista P e ir agregando elementos a ella (ambas en tiempo constante), entonces la complejidad total del algoritmo es de $O(V + E)$.