MA3705. Algoritmos Combinatoriales 2020.
Profesor: José Soto
Escriba(s): Luis Fuentes y Javier Santidrián Salas.
Fecha: 21 de septiembre de 2020.

Cátedra 5

1. Complejidad de Algoritmos Conocidos

1.1. Recuerdo: Algoritmos polinomiales y fuertemente polinomiales

Empezaremos esta clase recordando que la entrada de un algoritmo se puede medir de varias maneras, entre ellas:

- B: Número de bits de la entrada.
- N: Número de datos de la entrada.
- n: Número de vértices de un grafo entrada.
- m: Número de aristas de un grafo entrada.

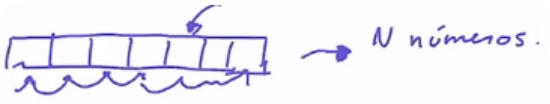
Cabe destacar que N permite controlar mejor el tamaño de la entrada que B, y con ello mejorar el funcionamiento del algoritmo (por ejemplo, conviene considerar N en el caso en que hayan pocos datos con muchos bits y en el caso en que hayan pocos datos con pocos bits, pero donde al algoritmo funcione bien si el número de datos es pequeño).
Por otro lado, para poder hablar sobre la eficiencia de un algoritmo, es fundamental definir lo siguiente:

- **Algoritmo Polinomial** (o débilmente polinomial) es uno con complejidad $O(B^k)$ para algún k fijo, es decir es polinomial en el número de bits de la entrada.
- **Algoritmo Fuertemente Polinomial** es uno con complejidad $O(N^k)$ para algún k fijo, es decir es polinomial en el número de datos de la entrada.

Notar que si un algoritmo es fuertemente polinomial entonces es polinomial. Una forma de ver esto, es que normalmente cada dato se escribe con un número polinomial de bits, por lo que si el algoritmo es polinomial en el número de datos, lo seguirá siendo en el número de bits. Cabe destacar que la recíproca en general no es cierta (podríamos tener un solo dato con muchos bits, donde claramente no será polinomial en el número de datos, a pesar de serlo en el número de bits).

1.2. Ejemplo de algoritmos conocidos

Recordemos que en un arreglo se puede acceder a cada celda en tiempo constante. Si tenemos N números naturales en un arreglo, para encontrar el máximo de ellos requerimos tiempo $O(N)$ (comportamiento asintótico lineal), pues debemos revisarlos todos.



Por otro lado, recordar que en una lista las celdas no tienen un identificador, por lo que para acceder a ellas hay un "puntero"que nos dice cuál es la celda siguiente a la anterior y así sucesivamente. Luego, si tenemos una lista de N números, para calcular el máximo tenemos que hacer el mismo procedimiento que para un arreglo (revisarlos todos), tomando tiempo $O(N)$.

Cabe destacar que insertar un objeto es más simple en una lista que en un arreglo, puesto que para un arreglo éste tiene que volver a crearse, mientras que para una lista solo se agrega un nuevo puntero a la nueva celda.

Si ahora quisieramos ordenar un arreglo, tenemos el algoritmo de Mergesort, que toma tiempo $O(NlogN)$ y el de Quicksort, que toma tiempo $O(N^2)$. Destacamos que no existen algoritmos (en el modelo de comparación) de complejidad $o(NlogN)$.

1.3. BFS

Búsqueda en amplitud (BFS):

Entrada: $G = (V, E)$, $r \in V$
$U \leftarrow \{r\}$, $F \leftarrow \emptyset$
Cola $\leftarrow \emptyset$.
Insertar aristas de $\delta(r)$ en Cola.
mientras Cola $\neq \emptyset$. **hacer**
  Extraer primer e de Cola.
  si $e \in \delta(U)$, $u \in U$, $v \notin U$
  entonces
    $U \leftarrow U + v$
    $F \leftarrow F + e$
    Insertar aristas de $\delta(v)$ en Cola
  fin
fin
devolver (U, F)

Estudiemos la complejidad de este algoritmo:

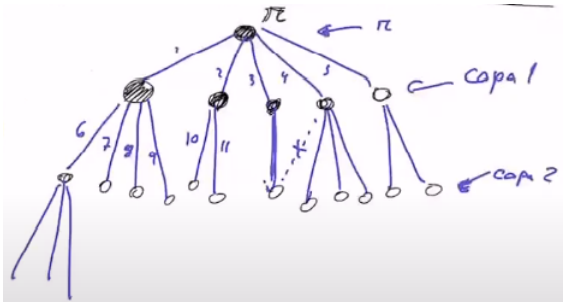
- Cuando se dice que se crea una *Cola* se refiere a hacer una lista de objetos donde se puede acceder al primero rápidamente (no es un arreglo pues la lista va creciendo). Cuando se habla de U se refiere a un arreglo que tiene una celda para cada vértice, donde el vértice r indica con un 1 si éste pertenece a U y con un 0 si no. Esto se hace así pues interesa saber en tiempo $O(1)$ si un vértice está o no en U (esto no se puede realizar en una lista pues hay que avanzar hasta que uno lo encuentre). Con ello, el tiempo que toma (ignorando los tiempos de inicialización de arreglos, pues suponemos que la entrada ya viene con un arreglo de los vértices y aristas) crear la *Cola* vacía, U con el vértice r y F vacía, es $O(1)$.
- El tiempo que toma insertar las aristas de $\delta(r)$ en la *Cola* es $O(deg(r))$ (número de aristas incidentes a r).
- Cada vez que sale e de *Cola* reviso sus dos extremos con tiempo $O(1)$ y si ocurre que hay un extremo dentro y otro fuera de U, agrego el extremo que está afuera a U con tiempo $O(1)$ y agrego e a la lista F, también con tiempo $O(1)$. Además, agregar las aristas de $\delta(v)$ a la *Cola* toma tiempo $O(deg(v))$.

Hagamos un mal conteo de este algoritmo: sabemos que todo lo que está antes del loop (mientras) toma tiempo $O(deg(r))$; pero todas las aristas entran eventualmente a la *Cola*, por ende m veces se entra a la cola (iteraciones) y en cada iteración se realiza trabajo. Así, podemos estimar que el trabajo del algoritmo se realiza a lo más en tiempo $O(m \cdot grado\ máximo)$, lo cual es mucho (nos concentraremos en el caso $n \leq m$, es decir un grafo conexo). Un mejor conteo sería el siguiente: cada vez que una arista entra a la *Cola*, paga por todo el trabajo que obliga al algoritmo hacer (en el futuro); pero cada arista entra a lo más 2 veces a la *Cola* (una por cada lado) y todo lo que se relaciona con esta arista toma $O(1)$, es decir cada arista realiza trabajo constante. Luego, el algoritmo realiza trabajo en tiempo $O(m)$ total.

Cabe destacar que en la realidad, en la entrada de BFS uno realiza una lectura completa, y hace $O(n+m)$ trabajo (pues hay que leer los vértices y las aristas) y luego en adelante toma $O(m)$ (de acuerdo al buen conteo). Así, el total de tiempo

que toma el algoritmo es $O(n+m)$, y esto sirve incluso si $n > m$ (notar que para el caso $n \leq m$ el orden de tiempo se traduce en $O(m)$).

Dando una segunda mirada, recordemos que el algoritmo BFS lo que hace es tomar un vértice r y comienza a trabajar sobre las aristas que salen de él, visitando los vértices por capas: primero visita a r, después entran a U todos los vértices que están a distancia 1 de r (primera capa), después entran a U todos los que están a distancia 2 de r (segunda capa) y así sucesivamente. Cada vez que se encuentra una arista con un extremo en las capas anteriores entrará a la *Cola* pero no hará nada de trabajo cuando salga (pues $e \notin \delta(U)$).

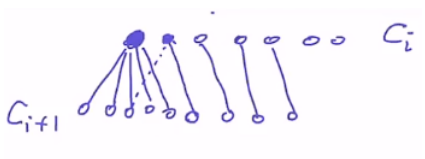


1.4. BFS Modificado

Búsqueda en amplitud (BFS):

Entrada: $G = (V, E)$, $r \in V$
$U \leftarrow \{r\}$, $F \leftarrow \emptyset$, Nivel$(r) \leftarrow 0$,
$C_0 \leftarrow \{r\}$, $C_1, \ldots, C_{n-1} \leftarrow \emptyset$.
para i *de* 0 *a* $n-1$ **hacer**
  mientras $C_i \neq \emptyset$ **hacer**
    Extraer primer u de C_i.
    para *cada* $v \in N(u)$ **hacer**
      si $v \notin U$ **entonces**
        $U \leftarrow U + v$, $F \leftarrow F + e$
        Padre$(v) \leftarrow u$, Nivel$(v) \leftarrow i+1$,
        Insertar v a C_{i+1}
      fin
    fin
  fin
fin
devolver (U, F)

La segunda manera de implementar BFS es por capas, asignándole un nivel a cada vértice: la capa 0 será r y en principio las siguientes estarán vacías. Luego, una vez que se visita la capa i (mientras la capa i tenga vértices que no se han procesado), se mira el primer vértice de la capa i y se agregan todos los vecinos que no están en capas anteriores (es decir, que no han sido visitados antes) a la capa $i+1$.



Con esta idea de BFS modificado, se deja **propuesto** demostrar por inducción en i que:

$$\mathcal{C}(i) := \{u \in V : Nivel(u) = i\} = \{u \in V : d(r,u) = i\}$$

Así, BFS puede servir para calcular los largos mínimos de un vértice a todos los otros (solamente cuando estamos hablando de número de aristas). Lo interesante es que este algoritmo sigue siendo $O(n+m)$. Así, en $O(n+m)$ tiempo podemos determinar todas las distancias desde una raíz a cualquier vértice (siempre las distancias en número de aristas).

Otra cosa muy útil es que uno puede recuperar el camino que va desde un vértice a la raíz: cada vez que un vértice u agrega sus vecinos nuevos a la capa siguiente, cada uno de los vecinos nuevos recuerda quien fue el que lo agregó al árbol, es decir a u (denominado $Padre$). Así, el camino desde un vértice a la raíz, se recupera preguntando quien es el $Padre$ del vértice actual y después quién es el $Padre$ del $Padre$ y así sucesivamente.

1.5. DFS

Búsqueda en profundidad (DFS):

Entrada: $G = (V, E)$, $r \in V$
$U \leftarrow \{r\}$, $F \leftarrow \emptyset$
Pila $\leftarrow \emptyset$.
Insertar aristas de $\delta(r)$ en Pila.
mientras Pila $\neq \emptyset$. **hacer**
  Extraer último e de Pila.
  si $e \in \delta(U)$, $u \in U$, $v \notin U$
  entonces
    $U \leftarrow U + v$
    $F \leftarrow F + e$
    Insertar aristas de $\delta(v)$ en Pila
  fin
fin
devolver (U, F)

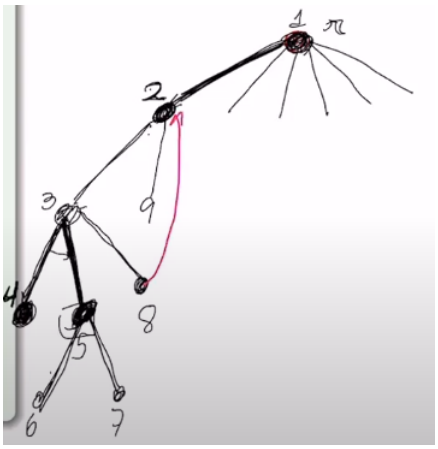
El análisis de DFS es igual al de BFS, donde también tenemos $O(n+m)$ trabajo. La única diferencia, es que aquí se usa una $Pila$ (y se extrae la última arista) y en BFS una $Cola$ (y se extrae la primera arista), aunque éstas pueden implementarse de la misma manera: una lista con un puntero en el primero y en el último, donde uno puede agregar o extraer alguien al principio o al final (al principio en BFS y al final en DFS).

En DFS las aristas que entran a la $Pila$ son las mismas, es decir todas las incidentes a r, pero la manera en que van visitándose los vértices es diferente. La idea de este algoritmo de búsqueda es como el de recorrer un laberinto: uno recuerda por donde ha pasado y lo que hace es devolverse por el camino que ha trazado, y sigue avanzando por la siguiente arista más próxima por la cual me devolví.

Notar que si el grafo fuera un árbol es fácil ver como DFS lo visita. Si no, entonces algunas aristas que se visitarán irán al pasado, es decir, a alguna que ya ha sido visitada.

En base a lo anterior, se deja el siguiente **propuesto**:
Sea $T = (V, F)$ un árbol DFS. Demostrar que todas las aristas de $E \setminus F$ conectan a un vértice u con un vértice v en el único $u - r$ camino en T. Probar que esto no es necesariamente cierto para BFS.

El resultado anterior es muy útil. Por ejemplo, si interesa entender los ciclos del grafo, saber que las aristas que están fuera del árbol van al pasado o hacia arriba (DFS), es mucho más cómodo que saber que las aristas pueden saltar desde una rama del árbol a otra (BFS).



1.6. Algoritmo de Prim

Recordemos que el **Algoritmo de Prim** calcula un MST o árbol de costo mínimo. Tenemos un grafo $G = (V, E)$ conexo y una función de costo $x : E \to \mathbb{R}$. Queremos calcular el árbol de costo mínimo.

¿Qué es lo que hace Prim? Parta de $r \in V$, mire todos los vértices visitados, mire las aristas que salen desde el conjunto de vértices visitados hacia afuera y escoja la arista de menor peso.
El algoritmo de Prim es bastante intuitivo en sí, se mira el conjunto visitado y se extiende el conjunto buscando cuál es el vértice que tenga una arista, hacia el conjunto de los visitados, de menor peso.
Si hacemos un análisis de cuánto tiempo se demora Prim en terminar, puede que no encontremos la mejor forma. Sin embargo, podemos hacer un análisis simple del tiempo:

Algoritmo de Prim (Prim 1957 - Jarník 1930):

Entrada: $G = (V, E)$ conexo, $r \in V$
Elegir $r \in V$;
$U \leftarrow \{r\}$
$F \leftarrow \varnothing$
mientras $\delta(U) \neq \varnothing$ **hacer**
  Sea $e = uv \in \delta(U),\ u \in U,\ v \notin U$ tal que e es la arista de menor peso en $\delta(U)$
  $U \leftarrow U + v$
  $F \leftarrow F + e$
fin
devolver $T = (U, F)$

- En realizar el primer paso, es decir, inicializar el algortimo, demoramos $O(n+m)$, pues necesitamos inicializar las aristas, los vértices y los arreglos.
- En cada iteración se debe encontrar la arista de menor peso que sale de los vértices visitados, por lo tanto, ¿Cuántas iteraciones tendría el algoritmo? A lo más n, es decir, $O(n)$. Y en cada iteración, ¿Cuánto demoramos en encontrar la arista de menor peso en $\delta(U)$? A lo más m, una forma de hacerlo es visitar todas las aristas del grafo, si esta está en $\delta(U)$ la guardamos y recordamos cuál es la mejor arista de este conjunto. Por lo tanto demora $O(m)$, por visitar todas las aristas del grafo recordando cuál es la mejor de $\delta(U)$ que se ha visitado.
- Finalmente, nuestras iteraciones toman tiempo $O(n \cdot m)$, pues el procedimiento que demora $O(m)$ se realiza alrededor de un tiempo $O(n)$.

Sin embargo, esta forma de implementar Prim es pésima pues realizamos demasiado trabajo, intentemos implementarlo de una forma mejor.
En lugar de hacer lo anterior, tener los visitados hasta ahora y en cada etapa calcular todas las aristas de U y ver cuál es la mejor, realizaremos otra cosa; para cada vértice W que esté fuera de U, acordemonos de la arista más barata que conecta w con U, y la llamaremos candidato de W.
Formalmente, para cada $w \in V \setminus U$:

$$cand(w) := uv \Leftrightarrow c(wu) = \min\{c(e) : e \in E[U; \{w\}]\}$$

¿Por qué nos sirve? Porque en el fondo si sabemos los candidatos, basta tomar el mínimo de estos candidatos para obtener la arista más barata, así en lugar de calcular el mejor de m cosas calculamos el mejor de n cosas, siendo en la mayoría de

las cosas $n < m$. El único problema es que debemos recalcular candidatos.
Entonces, el nuevo algoritmo de Prim, se ve de la siguiente forma:

Algoritmo de Prim (segunda implementación):

Entrada: $G = (V, E)$ conexo, $r \in V$
Elegir $r \in V$; $U \leftarrow \{r\}$; $F \leftarrow \varnothing$
mientras $U \neq V$ **hacer**
  (re)calcular para todo $w \notin U$, cand(w).
  Elegir uw con $u \in U$, $w \notin U$ en
   $\arg\min\{v \in V \setminus U : c(\text{cand}(v))\}$
  $U \leftarrow U + v$
  $F \leftarrow F + uv$
fin
devolver $T = (U, F)$

En general, el algoritmo es bastante directo:

- Iniciala igual que antes, con un orden $O(n + m)$.
- Mientras no hayamos visitado todo el grafo, recalculamos los candidatos de w. Una vez que tengamos los candidatos, escojemos el mejor candidato, es decir, la arista uw, con $u \in U$ y $w \notin U$ en $v \in V \setminus U : c(cand(v))$. Entonces ese es el mejor candidato y además es la arista que agregamos, el extremo que está fuera entra a U.
- Tenemos los vértices visitados U, supongamos que se agrega una arista uv con $v \notin U$, luego, el nuevo conjunto U corresponde a $U + v$, ¿Cómo recalculamos los candidatos? La idea es buscar la arista más barata al nuevo conjunto U, es decir, a $U + v$, entonces ¿Cuál es la arista más barata de un vértice w ahora? Si $vw \in E$, es decir, la arista existe, y el costo de vw es menor o igual que el costo del candidato de w, $c(vw) \leq c(cand(w))$, entonces vamos a redifinir el candidato de w a ser vw. En otro caso, nos quedamos con cualquier otro valor que tengamos.

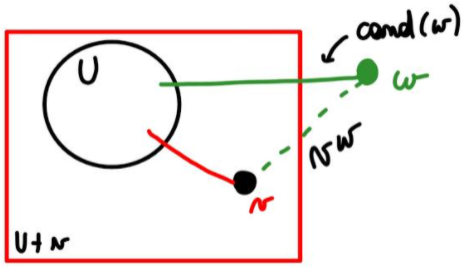


Figura 1: Representación Gráfica

- Lo anterior, para cada vértice, toma un tiempo de orden $O(1)$.
- En total, inicializar toma un tiempo de orden $O(n+m)$, más n iteraciones $(O(n))$, donde en cada iteración se realiza un trabajo que demora orden $O(n)$. Luego:

$$O(n + m) + O(n \cdot n) = O(n^2 + m) = O(n^2)$$

  Donde la primera igualdad conserva las cosas más grandes, y la segunda igualdad viene del hecho de que m siempre es más chico que n^2 pues m es el número de aristas y siempre se tiene a lo más n^2 aristas.

1.7. Mejores Implementaciones

El algoritmo de Prim se puede implementar mejor con mejores estructuras de dato:

1. Como lo hicimos anteriormente, usando arreglos y listas enlazadas (candidatos), podemos hacerlo en orden $O(n^2)$.

2. Los **Heaps** (o montículos) son estructuras de datos que permiten extraer el mínimo de un montículo de manera rápida o bien, actualizar los valores (decrecer) de manera rápida.
   - Si utilizamos Heaps Binarios, la implementación es del orden $O((n+m)\log n)$.
   - Si utilizamos Heaps de Fibonacci, la implementación es del orden $O(m + n\log n)$.

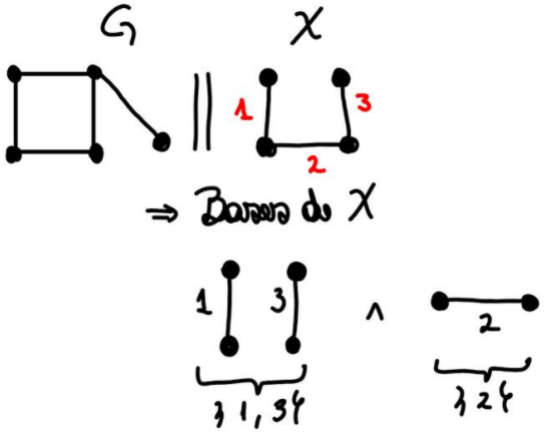
2. Sistemas de Independencia

2.1. Sistemas de independencia

- Un **Sistema** $(S, \mathcal{X})$ se compone de una conjunto de referencia S, el cual es finito, y de una familia $\mathcal{X}$ de subconjuntos contenido en S, de manera que $\mathcal{X} \subseteq 2^S$.
- Diremos que un sistema $(S, \mathcal{I})$ es un **Sistema de independencia** si:
  1. El vacío es independiente: $\emptyset \in \mathcal{I}$.
  2. $\forall X \subseteq Y \subseteq S, Y \in \mathcal{I} \Longrightarrow X \in \mathcal{I}$
- Todos los conjuntos en $\mathcal{I}$ se conocen como **conjuntos independientes.**
- Ejemplos:
  1. Consideremos $S = [n]$ e $\mathcal{I} = \{X \subseteq S : |X| \leq 10\}$, claramente $(S, \mathcal{I})$ es un sistema de independencia.
  2. Si $G = (V, E)$ grafo, consideremos $\mathcal{I} = \{X \subseteq E : X \textit{acíclico}\}$, entonces $(S, \mathcal{I})$ es un sistema de independencia.
  3. Sea $G = (V, E)$ grafo, $M \subseteq E$ es matching, si (V, M) tiene grado máximo 1. Si consideramos el conjunto $\mathcal{I} = \{X \textit{ matching de } E\}$, entonces $(E, \mathcal{I})$ es un sistema de independencia.
- Para $X \subseteq S$ llamamos **base** de X a cualquier $B \subseteq X$ independiente y maximal para inclusión.

2.2. Ejemplos de Sistemas de Independencia

1. Si tomamos a $X = \{3, 4, 5\}$, entonces la base de $X = X$.
2. Si tomamos el conjunto de los *acíclicos*, la base de X corresponde a los bosques generadores.
3. ¿Qué pasa con los Matching?



Entonces, las bases de $X = \{\{1, 3\}, \{2\}\}$

Hay sistemas en donde todas las bases presentan el mismo tamaño, y así mismo hay sistemas en donde las bases tienen diferentes tamaños. Por ejemplo, en el caso de los acíclicos, todas las bases presentan el mismo tamaño.
¿Por qué nos interesa el tamaño de las bases? Resulta que al tener la propiedad de que todas las bases tengan el mismo tamaño, los sistemas de independencia que lo cumplen, se comportan muy parecido a *espacios vectoriales*.