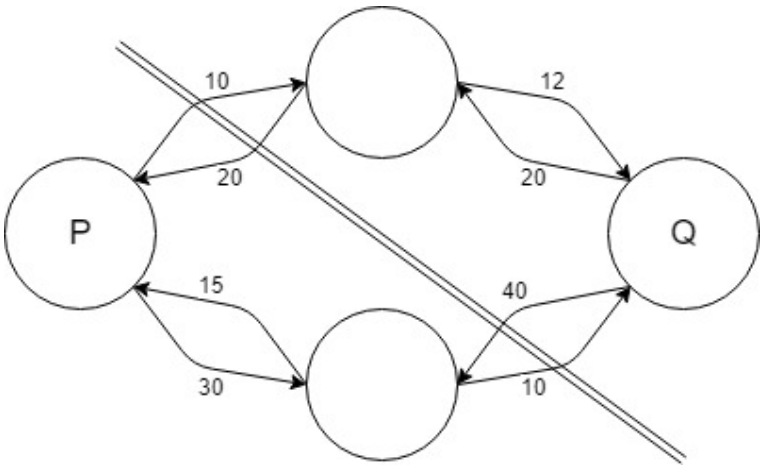
IIC2133 Estructuras de datos y algoritmos
1° **semestre 2019**

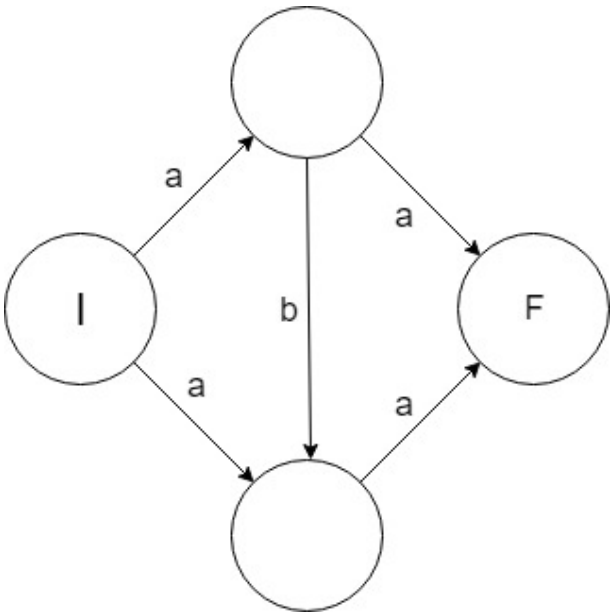
Ayudantía-Taller Flujo máximo y ordenación en O(n)

1. Flujo máximo

a) Explique cómo se relaciona el flujo máximo entre dos nodos P y Q de un grafo con los cortes del grafo que dejan a P y Q separados.
Solución: En un grafo dirigido, el flujo máximo desde P hasta Q es el mínimo flujo que cruza un corte desde P hasta Q. Por ejemplo en la figura siguiente el flujo que pasa desde la mitad izquierda a la derecha es solo 20 y es el corte con el mínimo flujo entre P y Q. No es relevante el flujo que pasa desde la derecha a la izquierda.

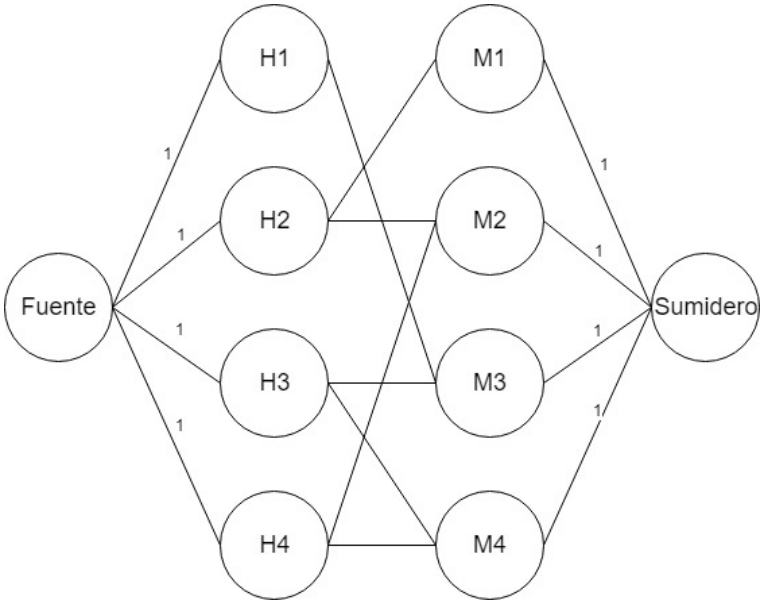


b) Explique por qué la complejidad del algoritmo de Ford-Fulkerson en el peor caso puede depender del peso de las aristas usando el siguiente grafo:

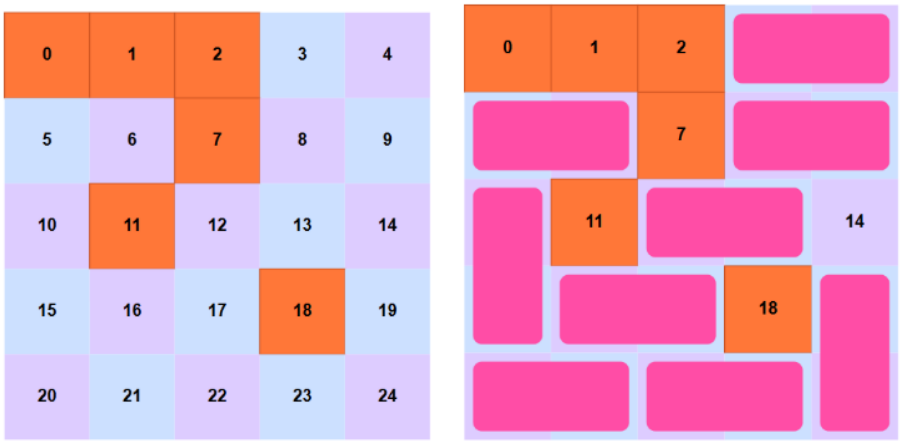


Solución: En el ejemplo de la imagen, si $a = 1000$ y $b = 1$ podemos elegir a ruta que pasa por la arista de costo b en todas las iteraciones. Esto aumentaría el flujo entre I y F de a 1 en cada iteración hasta llegar a 2000 de flujo.

c) Explique por qué los algoritmos de flujo máximo sirven para resolver problemas de emparejamiento.
Solución: Los problemas de flujo máximo sirven para resolver emparejamiento bipartito porque una pareja que une H_i con M_j puede ser representada por un flujo que pasa entre los nodos H_i y M_j. Además, podemos asegurar de que cada nodo se empareje una sola vez conectándolo con una arista de costo 1 a la fuente (en caso de estar al lado izquierdo) o al sumidero (en caso de estar al lado derecho). Sabemos que al calcular el flujo máximo tendremos el máximo de parejas por lo que resolvemos el problema de maximizar las parejas.



d) Tenemos un tablero como el de la figura y queremos saber cual es el máximo de dominós que podemos poner en el tablero para rellenarlo. Explique cómo resolver este problema como un problema de matching usando flujo máximo.



También explique qué significa el paso de flujo al traducirlo al tablero y qué significa pasar por una arista backward.

Solución: Este problema se puede resolver planteándolo como matching ya que cada celda impar (celeste) debe ser emparejada con una celda par (morada). Esto nos permite hacer un grafo bipartito donde las conexiones entre nodos de un lado y el otro son entre los vecinos en el tablero. Por ejemplo la celda 8 se conecta con la 3, 9 y 13 (la 7 no porque está deshabilidata). Si encontramos una pareja para cada celda del tablero entonces se puede cubrir de dominós. El paso de flujo entre un nodo celeste y uno morado indica que se coloca un dominó entre sus casillas correspondientes. Una arista backward en

el grafo permite deshacer una asignación que une dos casillas con un dominó, ya que deshace la unión entre 2 dominós para crear otra asignación.

2. Ordenación en O(n)

2.1. Counting sort

a) ¿Es counting sort estable? Justifique.
Solución: En la mayoría de sus implementaciones si es estable. La forma usual de implementarlo es crear una cola FIFO en cada casilla de un arreglo donde se insertan los elementos. Cuando coloco el valor en la casilla correspondiente se inserta en la cola y finalmente se juntan las colas para tener el arreglo ordenado. La cualidad FIFO de las colas hacen que se mantenga cualquier orden preexistente en los datos en caso e valores repetidos.

b) ¿Por qué counting sort puede ordenar datos más rápido que $O(n \cdot \log(n))$?¿Qué contras tiene el algoritmo?
Solución: La cota de ordenación $O(n \cdot \log(n))$ es solo válida para algorimos de ordenación que funcionan por comparación de elementos. Counting sort no compara los elementos a ordenar entre ellos sino que asigna las posiciones según su valor. Para poder ordenar con counting sort sin embargo es necesario que los valores a ordenar sean enteros (o se pueden pasar a un número entero) y que el valor máximo esté acotado. Esto permite que sea eficiente y que se pueda asignar una posición en un arreglo a cada número. Si el número más grande es muy alto, counting sort puede tomar más tiempo que un algoritmo por comparación.

c) ¿Qué pasa si se trata de usar counting sort con una lista de números reales?
Solución: Si se usa counting sort con números reales habría un problema ya que no se puede asignar un número a una posición de un arreglo con índice real. Sin embargo existen otros algoritmos de ordenación como bucket sort que permiten solucionar este problema agregando otras restricciones a los datos.

2.2. Radix sort

a) Explique qué pasa si no se usa un algoritmo de ordenación estable como sub rutina de radix sort en su versión LSD.
Solución: Si la subrutina de Radix sort no es estable se va a perder el orden realizado en la ordenación anterior al ordenar en la siguiente iteración. Esto hace que los datos finalmente queden ordenados solo por el dígito más significativo.

b) Explique por qué radix sort MSD es un buen algoritmo de ordenación para strings.
Solución: Usualmente al ordenar strings es raro que se tenga que iterar sobre el string completo para saber cual es el string mayor que el otro. Radix sort MSD permite

aprovechar esta propiedad ya que va a ordenar iterando por los primeros caracteres y no va a seguir cuando no sea necesario revisar más caracteres. Esto permite iterar solo lo necesario sobre el largo de los strings.

c) Explique por qué es conveniente cambiar de algoritmo de ordenación cuando hay pocos datos a ordenar en la versión MSD.
Solución: Cuando se ordena con radix sort MSD es necesario crear un arreglo que almacene los datos que se van a ordenar en cada iteración, y su tamaño es igual al número de distintos valores que puede tener un dígito (si se hace radix sort a binario solo hay 2 valores, a decimal hay 10 valores y si se hace a un string el arreglo mide lo mismo que todas las posibles letras del string). En particular cuando queden pocos elementos a ordenar en las distintas llamadas recursivas del algoritmo, se va a reservar muchas veces memoria para ordenar pocos elementos. Generalmente la operación de pedir memoria en un programa es pesada, por lo que ordenar pocos elementos usando un algoritmo de ordenación *in place* es la mejor opción.

d) (Actividad) Ordene los siguientes números usando radix sort desde el dígito menos significativo al más significativo.

- 3646
- 3276
- 8445
- 4687
- 2346
- 2386
- 2323
- 2335
- 2846
- 2723
- 2723
- 2623
- 3846

Solución: En cada iteración crearemos un arreglo de tamaño 10 para almacenar los números según el dígito que corresponde en la iteración:

a) Primera iteración: Los números quedan ordenados por su cuarto dígito y en caso de empate conservan el orden inicial: 2323, 2723, 2723, 2623, 8445, 2335, 3646, 3276, 2346, 2386, 2846, 3846, 4687

b) Segunda iteración: Los números se ordenan por su tercer dígito y en empate conservan el orden anterior: 2323, 2723, 2723, 2623, 2335, 8445, 3646, 2346, 2846, 3846, 3276, 2386, 4687

c) Tercera iteración: Los números se ordenan por su segundo dígito y en empate conservan el orden anterior: 3276, 2323, 2335, 2346, 2386, 8445, 2623, 3646, 4687, 2723, 2723, 2846, 3846

d) Cuarta iteración: Los números se ordenan por su primer dígito y en empate conservan el orden anterior: 2323, 2335, 2346, 2386, 2623, 2723, 2723, 2846, 3276, 3646, 3846, 4687, 8445