TAREA 4: Analisis y Diseño de Algoritmos

Andrey Arguedas Espinoza - 2020426569

November 21, 2022

1. Considere el siguiente algoritmo voraz para resolver el problema de multiplicación encadenada de matrices: **30 ptos.**

   Se tienen matrices A_1, A_2, ..., A_n con dimensiones d_0 x d_1, d_1 x d_2, ..., d_{n-1} x d_n. Buscar la más pequeña de las dimensiones, llamarla d_i, y poner paréntesis entre las matrices A_{i-1} y A_i, de modo que se multipliquen así: (A_1 x ... x A_{i-1}) x (A_i ... x A_n). Aplicar el algoritmo recursivamente para decidir cómo multiplicar (A_1 x ... x A_{i-1}) y (A_i ... x A_n).

   Mostrar que este algoritmo voraz no necesariamente encuentra la forma óptima de multiplicar una secuencia de matrices.

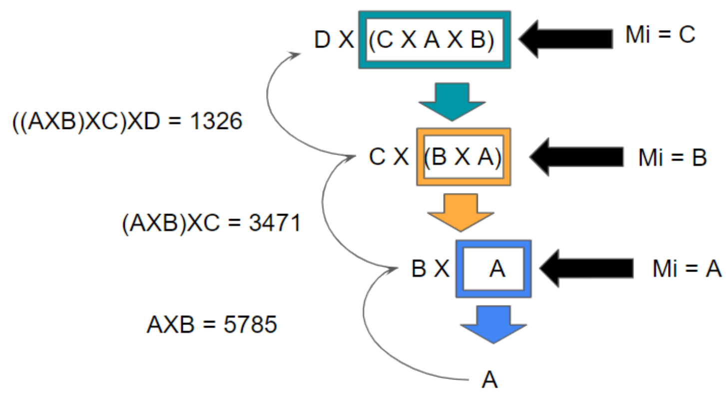
Para demostrar que el algoritmo dado no siempre encuentra la fomra óptima **necesitamos encontrar un contraejemplo**, para esto utilizaremos las siguientes matrices que poseen las siguientes dimensiones:

$\boxed{A = 13x5}$; $\boxed{B = 5x89}$; $\boxed{C = 89x3}$; $\boxed{C = 3x34}$

Seguido generamos las formas asociativas posibles de multiplicar estas matrices y vemos cuantas multiplicaciones serían necesarias:

((AB)C)D = 10582 multiplicaciones
(AB)(CD) = 54201 multiplicaciones
(A(BC)D) = 2856 multiplicaciones
A((BC)D) = 4055 multiplicaciones
A(B(CD) = 26418 multiplicaciones

Ahora si corrieramos el algoritmo propuesto sobre las 3 matrices se nos forma de la siguiente manera:



Asosiación generada : ((AXB)XC)XD

Total : 10582 multiplicaciones

***Nota: Las flechas es cuando retorna de los llamados recursivos.**

Finalmente podemos observar que este algoritmo no genera siempre la solución más óptima ya que la más optima era (AX(BXC)XD) con solo 2856 multiplicaciones necesarias.

2. Suponga que se tiene una red (grafo conexo no dirigido) en la cual cada arista e_i tiene asociada un ancho de banda b_i. Si se tiene una ruta P, de s a v, entonces se define: **30 ptos.**

    capacidad(P(s,v)) = ancho de banda mínimo de todas las aristas que forman la ruta P de s a v
    = min { b_i | e_i en ruta P de s a v }.

    Además, se define

    capacidad(s,v) = $\max_{P(s,v)}$capacidad(P).

    Esencialmente, ***capacidad(s,v)*** es iqual a la capacidad de la ruta con más capacidad que va de s a v.

    Dar un algoritmo que calcule capacidad(s_0,v) para cada vértice v, donde s_0 es un vértice fijo inicial. Muestre que su algoritmo es "correcto" y analice su tiempo de ejecución.

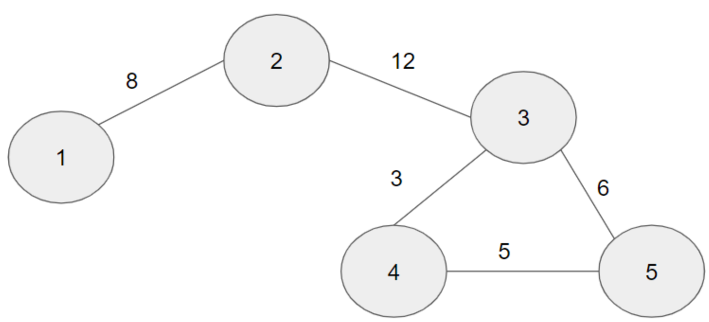
    Diseñarlo de modo que no tome más de $O(n^2)$, y con las estructuras de datos adecuadas tome O(m log n).
    Debe basarse en alguno de los algoritmos vistos en clase.

Para poder resolver este ejercicio nos basaremos en ciertas cosas del algoritmo de Dijkstra y BFS pero realizando las siguientes modificaciones:

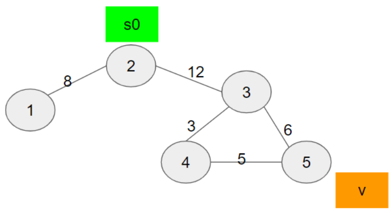
* En lugar de la distancia buscamos maximizar el minimo de la capacidad, es decir encontrar la ruta mas capaz de s0 a v.

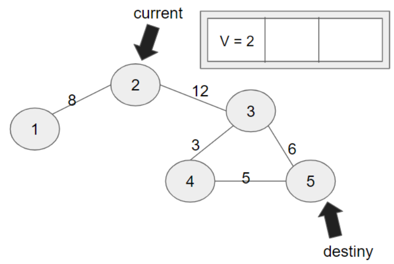
* Llevar una lista extra para guardar las rutas, el cual llamaremos paths para saber como llegar de s0 a v con la ruta con mayor capacidad.

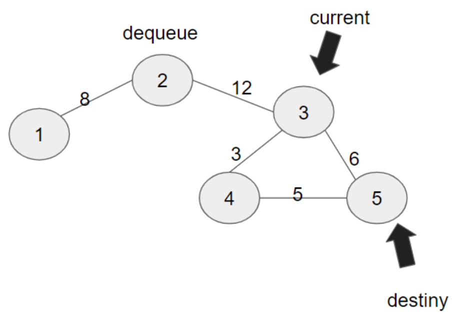
* Utilizar un priority queue para mejorar el rendimiento del algoritmo e ir maximizando la capacidad minima de las rutas. El prioirity queue se encarga de ordenar de mayor a menor a los vecinos de que nada por el cual vamos a pasar.

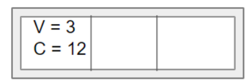


Con el grafo anterior nuestro algoritmo haría lo siguiente para encontrar la capacidad(2,5):

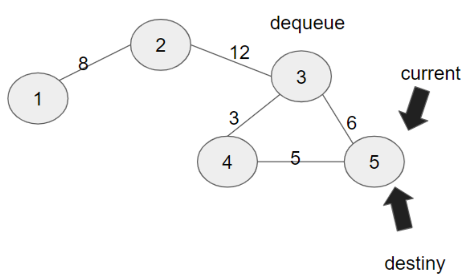


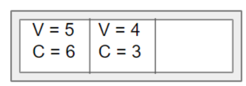




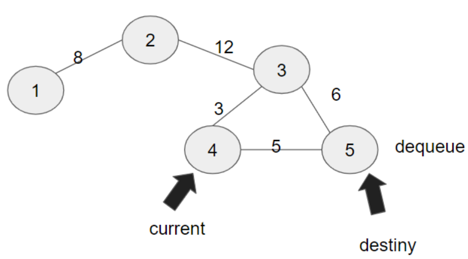


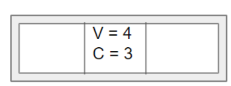
Path = [3]





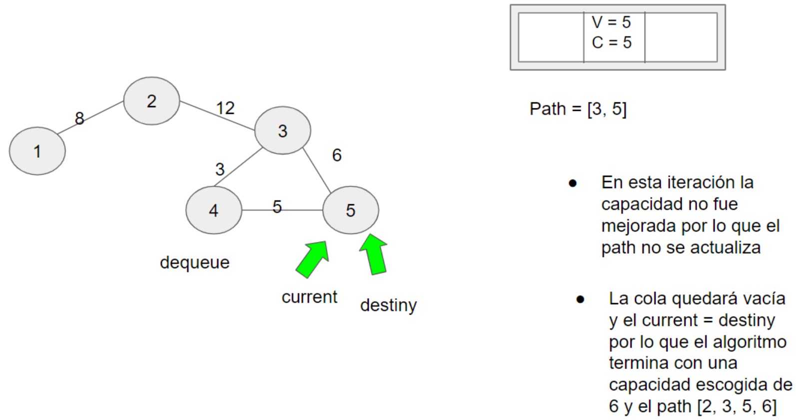
Path = [3, 5]





Path = [3, 5]

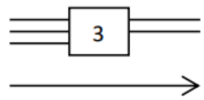
- En esta iteración la capacidad no fue mejorada por lo que el path no se actualiza



Pseudocodigo de nuestra implementación

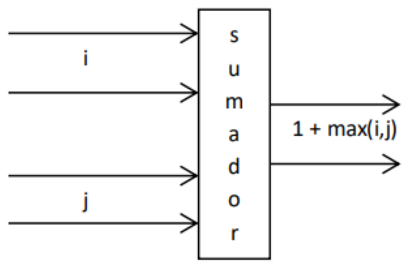
```
function capacidad(s0, v)
    current = s0
    destiny = v # A donde necesitamos llegar
    path = []
    capacity = 0
    visited = []
    # Crea el queue, esta estrucutra ordena al insertar los elementos
    # Los ordena de mayor a menor mediante el valor del arco entre v1 -> v2
    priority_queue = build_priority_queue()
    visited.add(current)
    #priority_queue.enqueue(current, infinity)
    while priority_queue is not empty or? current is not destiny:
        current = priority_queue.dequeue()

        for each node adjacent of current:
            #La cola encola en orden dado el valor del arco current -> node
            priority_queue.enqueue(node, edges(current, node))

        if current is not in visited:
            newCapacity = max(capacity, edges(current, priority_queue.top()))
            if newCapacity > capacity:
                path.add(priority_queue.top())
                capacity = newCapacity
            visited.add(current)

    return (capacity, path)
```

3. Un *n-tally* es un circuito que admite n bits como entrada y produce $1 + \lfloor \log_2 n \rfloor$ bits como salida. Cuenta (en binario) el número de bits iguales a 1 que hay en la entrada. Por ejemplo, si n=9 y las entradas son 0 1 1 0 0 1 0 1 1, la salida es 0101 (hay cinco bits encendidos).



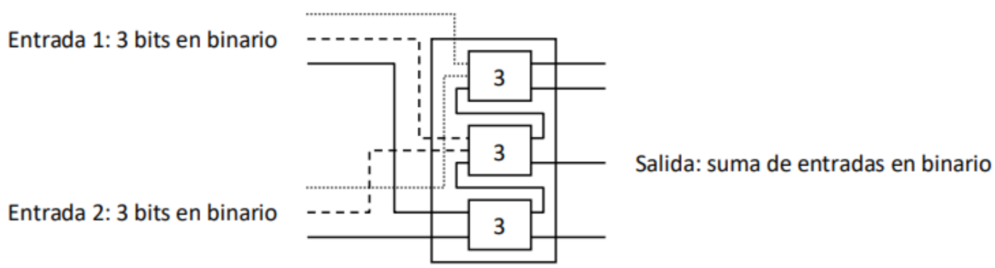
3-tally

Un *sumador-(i,j)* es un circuito que tiene una entrada de i bits, una entrada de j bits, y una salida de [1+max(i,j)] bits, Suma sus dos entradas en binario. Por ejemplo, si i=3, j=5 y las entradas son 101 y 10111 respectivamente, la salida es 011100.



sumador-(i,j)

Siempre es posible construir un *sumador-(i,j)* empleando exactamente max(i,j) circuitos *3-tally*. Por esta razón, el *3-tally* suele denominarse *sumador completo*.



sumador-(3,3)

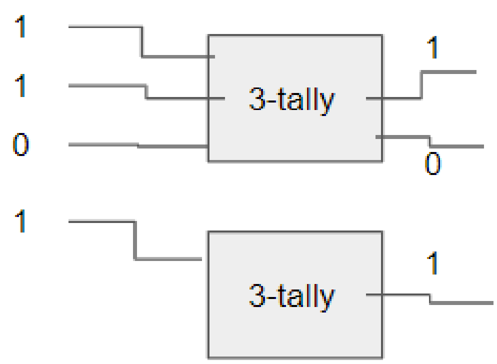
a) Utilizando *3-tallys* y *sumadores-(i,j),* mostrar la forma de construir un *n-tally* eficiente. **15 ptos.**

b) Dar la recurrencia, incluidas las condiciones iniciales, para el número de *3-tallys* que se necesitan para construir el *n-tally* anterior. Se deben incluir en la cuenta los *3-tally* que forman parte de aquellos *sumadores-(i,j)* que pueda utilizar. **15 ptos.**

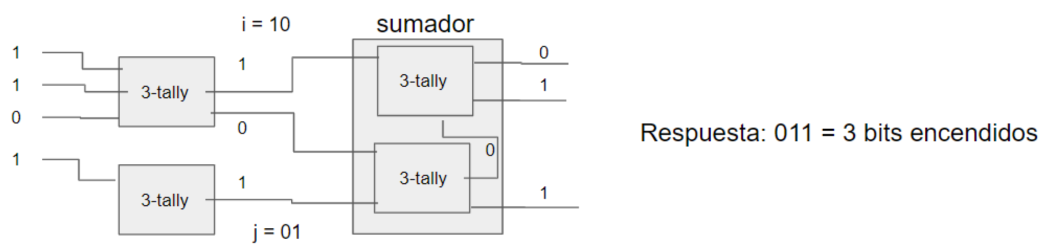
c) Dar la expresión θ para el número de *3-tally* que se necesitan para construir un n-tally. Justificar. **10 ptos.**

Para poder construir el n-tally primero tenemos que ver como funcionan el 3 tally y el sumador con una entrada pequena para encontrar una forma de combinarlas que nos sirva, pra eso veremos el ejemplo de la entrada **1101**:

Lo primero que podemos observar es que para poder ingresar esta entrada y ponerla en 3-tallys se necesitan 2, ya que por cada 3-tally solo podemos ingresar 3 bits, con esto podemos deducir que inicialmente necesitamos $\lceil n/3 \rceil$ 3-tallys.



Ahora podemos convetir estos resultados que nos dan los 3-tallys e insertarlos en un sumador, el cual necesta max(2,1) 3-tallys.

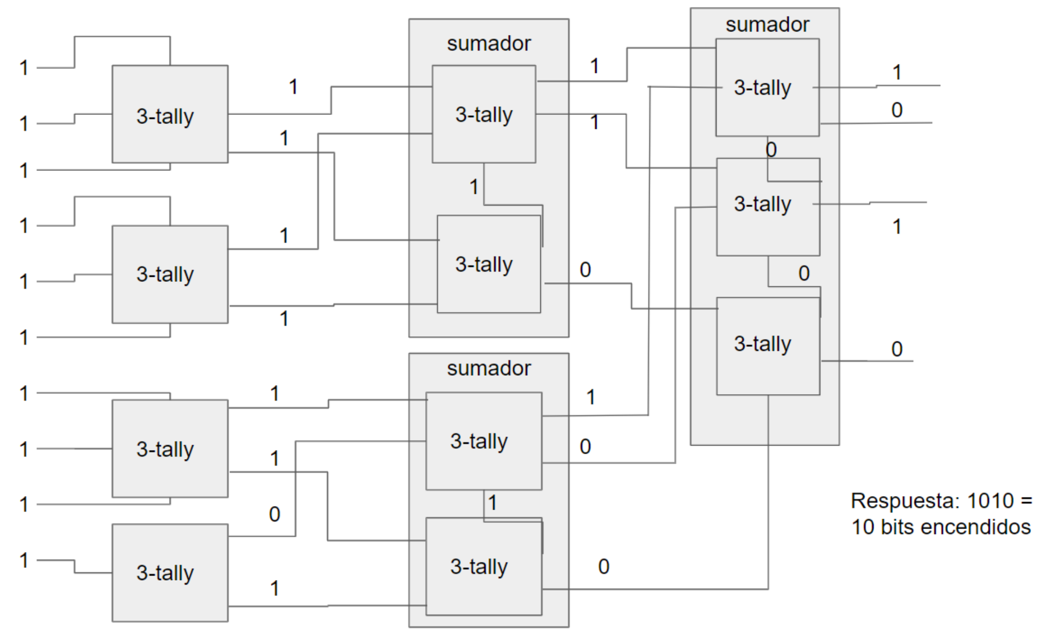


Para este caso vemos que el n-tally funciona, sin embargo este caso es muy simple, nuestra solución para **n** casos consiste en lo siguiente:

- Ingresar la entrada en $\lceil n/3 \rceil$ 3-tallys

- Los resutados de lo anterior sumarlos en $\lceil resultados/3 \rceil$ /2 sumadores hasta que solo quede un sumador y esa será la respuesta.

Ejemplo para la entrada **1111111111**



Finalmente tenemos que la cantidad de 3-tallys necesarios para el n-tally son: $\lceil n/3 \rceil + 2 * \lceil n/3 \rceil - 1$