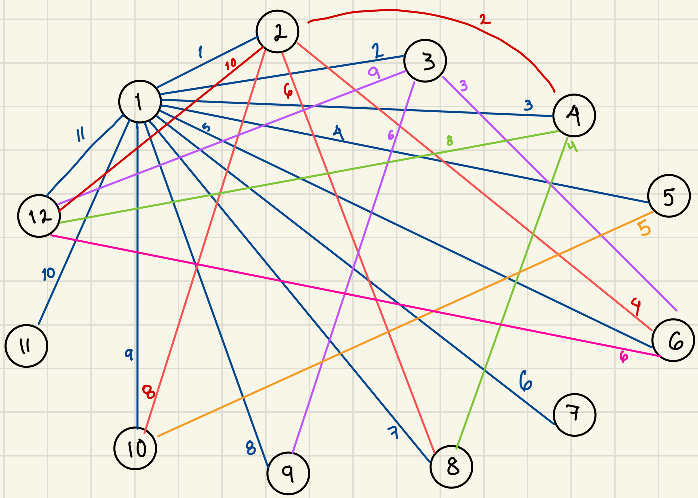
Examen MADI

Alumno: Axel Daniel Malváez Flores

1. (2.5 pts) Considera la gráfica de aristas ponderadas cuyos vértices son los números $1, 2, \ldots, 12$ y en donde dos números i, j tienen una arista entre ellos si i divide a j, o j divide a i. En ese caso, a esa arista se le pone peso $|i-j|$. Usa el algoritmo de Prim manualmente para encontrar un árbol de peso mínimo en esta gráfica.

Primero veamos cómo se ve la gráfica ponderada:



Ejecución del algoritmo:

1 Sea $v = 1$ el vértice elegido para iniciar el algoritmo

2 $T = [1]$

3 $A = \{2, \ldots, 12\}$

4 $A\_b = [\,]$

5 Iteración 1: escogemos la arista de menor peso $1 \xrightarrow{1} 2$

$A\_b = [12]$, $T = [1, 2]$, $A = \{3, \ldots 12\}$

6. Iteración 2: arista de menor peso $1 \xrightarrow{2} 3$

$A\_b = [12, 13]$, $T = [1, 2, 3]$, $A = \{4, \ldots, 12\}$

7 Iteración 3 : arista de menor peso $2 \overset{2}{—} 4$

A_b = [12, 13, 24], T = [1, 2, 3, 4], A = {5, ..., 12}

8 Iteración 4 : arista de menor peso $3 \overset{3}{—} 6$

A_b = [12, 13, 24, 36], T = [1, 2, 3, 4, 6], A = {5, 7, ..., 12}

9 Iteración 5 : arista de menor peso $4 \overset{4}{—} 8$

A_b = [12, 13, 24, 36, 48], T = [1, 2, 3, 4, 6, 8], A = {5, 7, 9 ..., 12}

10 Iteración 6 : arista de menor peso $1 \overset{4}{—} 5$

A_b = [12, 13, 24, 36, 48, 15], T = [1, 2, 3, 4, 5, 6, 8], A = {7, 9 ..., 12}

11 Iteración 7 : arista de menor peso $5 \overset{5}{—} 10$

A_b = [12, 13, 24, 36, 48, 15, 510], T = [1, 2, 3, 4, 5, 6, 8, 10], A = {7, 9, 11, 12}

12 Iteración 8 : arista de menor peso $1 \overset{5}{—} 7$

A_b = [12, 13, 24, 36, 48, 15, 510, 17], T = [1, ..., 6, 7, 8, 10], A = {9, 11, 12}

13 Iteración 9 : arista de menor peso $3 \overset{6}{—} 9$

A_b = [12, 13, 24, 36, 48, 15, 510, 17, 39], T = [1, .., 8, 9, 10], A = {11, 12}

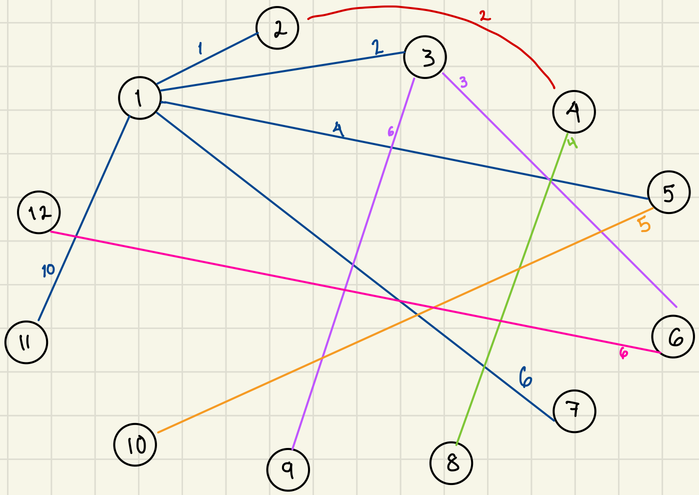
14 Iteración 10 : arista de menor peso $6 \overset{6}{—} 12$

A_b = [12, 13, 24, 36, 48, 15, 510, 17, 39, 612], T = [1, ..., 10, 12], A = {11}

15 Iteración 11 : arista de menor peso $1 \overset{10}{—} 11$

A_b = [12, 13, 24, 36, 48, 15, 510, 17, 39, 612, 111], T = [1, ..., 11, 12], A = { }

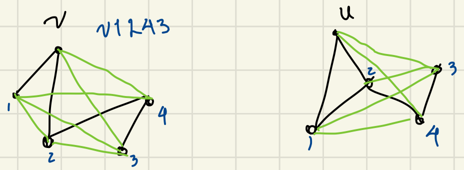
Regresamos el árbol de peso mínimo A_b = [12, 13, 24, 36, 48, 15, 510, 17, 39, 612, 111]



2. (2.5 pts) Se tiene una gráfica dada G en la cuál se sabe que dos vértices u y v **no** están conectados entre sí, es decir, no hay ningún camino posible de u a v. Queremos agregar a G más aristas, pero cuidando que u y v sigan desconectados.

Diseña un algoritmo que determine cuál es el máximo número de aristas que se pueden agregar de modo que u y v sigan desconectados. De entrada, recibe una gráfica en formato de vértices y aristas. De salida, debe de regresar tanto el máximo número de aristas que se pueden agregar, como la gráfica con las aristas agregadas, en formato de vértices y aristas. El algoritmo debe correr en tiempo a lo más $O(n^2)$.

Ejemplo:



aristas iniciales

aristas añadidas

Dado que la gráfica es inconexa entonces al menos tiene dos componentes conexas. Incluso si tuviera más de dos podríamos pensar en tener dos componentes no necesariamente conexas donde no exista un camino de u a v.

Algoritmo: Recibimos la gráfica

1. Creamos una lista que guarde las componentes $O(1)$

2. Aplicamos DFS para obtener una de las componentes conexas de G. $O(|V|+|E|)$.

Sub-algoritmo: Iniciamos DFS de manera recursiva y utilizando la técnica de backtracking. La función recibirá una gráfica en formato de vértices y aristas, un vértice inicial v, una lista de vértices visitados y una lista que guarda los vertices recorridos.

2.1. Agregamos a v en la lista de vértices recorridos

2.2. Iteramos sobre los vecinos de v, si u es vecino y no ha sido visitado hacemos DFS sobre u y si ya todos fueron visitados regresamos la lista de vértices recorridos y la lista de vértices visitados.

3. La lista de componentes tendrá dos elementos, el primero será el recorrido que regresa el paso anterior, el segundo será la lista de vértices de G que no aparecen en DFS.

4. Supongamos que las aristas se almacenan en un diccionario, entonces iteramos para cada componente C en la lista de componentes: $O(1)$ pues iniciamos suponiendo que hay únicamente dos componentes.

4.1 Declaramos un contador_v de vértices

$O(|V|^2)$

4.2 Declaramos un contador_e de aristas

4.3 Iteramos para cada vértice v en C: $O(n)$

En cada iteración obtenemos la lista de vértices adyacentes a v y verificamos la longitud de la lista $O(n)$ esta longitud la sumamos a nuestro contador de aristas. Finalmente sumamos 1 al contador de vértices.

4.4 Dividimos por 2 el contador de aristas pues las estamos contando doble.

5. Una vez que sabemos cuantas aristas y vértices hay en cada componente iteramos sobre el número de componentes: $O(1)$ pues hay 2 componentes y las operaciones realizadas dentro de la iteración son constantes.

La gráfica completa de esa componente tendrá a lo más

$$n = \binom{\text{contador\_v}_c}{2}$$

aristas, por tanto el número de aristas que necesitamos agregar a dicha componente son: $K_c = n - \text{contador\_e}_c$

contador de aristas para la componente c.

6. Creamos la nueva gráfica añadiendo a G las aristas restantes a cada componente. $O(n^2)$

7. Devolvemos la suma de todos los K_i de cada componente lo cual es la cantidad de aristas máxima que podemos agregar a la gráfica G y la nueva gráfica con las aristas añadidas.

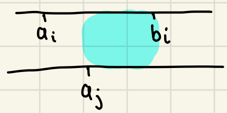
Complejidad total del algoritmo $O(|V|^2)$

4. (2.5 pts) Se tienen n intervalos finitos en $\mathbb{R}$, dados como $[a_i, b_i]$ para cada i en $1, 2, \ldots, n$. A partir de ellos, se crea una gráfica G de la siguiente manera: hay un vértice por cada intervalo y hay una arista si los dos intervalos correspondientes se intersectan. Es sencillo verificar si esta gráfica es la gráfica completa en tiempo $O(n^2)$. Veremos cómo hacerlo en menos tiempo.

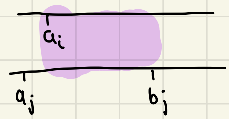
1) - Escribe qué desigualdades deben cumplir a_i, b_i, a_j, b_j para que los intervalos $[a_i, b_i]$ y $[a_j, b_j]$ se intersecten.
2) - Propón en pseudocódigo un algoritmo que use el inciso anterior para determinar si la gráfica G es completa en tiempo $O(n^2)$.
3) - Sea b el mínimo de los b_j. Muestra que todas las parejas de intervalos se intersectan si y sólo si b está en todos los intervalos.
4) - Propón en pseudocódigo un algoritmo que use el inciso anterior para determinar si la gráfica G es completa en tiempo $O(n)$.

1) Las desigualdades que notamos son:

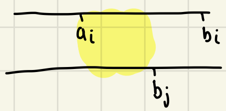
- $a_i \leq a_j \leq b_i$



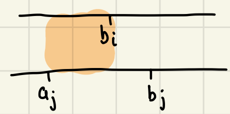
- $a_j \leq a_i \leq b_j$



- $a_i \leq b_j \leq b_i$



- $a_j \leq b_i \leq b_j$



2) Primero notemos que la condición que cumple la gráfica para que esta sea completa es que todos los intervalos deben intersectarse.

Iniciemos suponiendo que nos dan el conjunto de intervalos I.

Pseudocódigo:

$O(n^2)$ $O(n)$ $O(n)$ $O(1)$

```
for i in I:
    for j in I:
        if not (i[0] <= j[0] <= i[1] or j[0] <= i[0] <= j[1] or
                i[0] <= j[1] <= i[1] or j[0] <= i[1] <= j[1]) then
            return False
return True
```

3) P.D Sup. b mínimo de los b_j. Todos los intervalos se intersectan $\iff$ b está en todos los intervalos.

$\Leftarrow$: ∴ Si b está en todos los intervalos, entonces es trivial que todos los intervalos se intersectarán.

$\Rightarrow$ Sup. todos los intervalos se intersectan, entonces sea $K = \bigcap_i [a_i, b_i]$ el intervalo que resulta de la intersección. Veamos qué pasa si $b \notin K$. Sea K_i cota superior de K, entonces tenemos tres casos:

- $K_i < b$

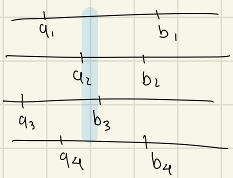
  Esto causa una contradicción a que b es el mínimo de los b_i

- $K_i > b$

  Tenemos dos casos. Sea K_j la cota inferior de K:

  1) $b \lneq K_j < K_i$

     Esta desigualdad no tiene sentido pues b es una cota superior mínima y no puede ser menor a una cota inferior



  2) $K_j \lneq b < K_i$

     Si $K_j = b$ por el caso anterior como b es cota superior mínima entonces es necesario que $K_i = b$ y tendríamos que la intersección $K = [b, b]$ y por tanto $b \in \bigcap_i [a_i, b_i]$, i.e b está en todos los intervalos.

- $K_i = b$

  Si sucede este caso entonces $K = [K_j, b] = \bigcap_i [a_i, b_i]$ y por tanto b está en todos los intervalos pues está en la intersección.

∴ Si todos los intervalos se intersectan, entonces b está en todos los intervalos.

Q.E.D.

4) Proponemos el siguiente algoritmo en pseudocódigo:

```
b min = 0                         O(1)
for a, b in I :                   O(n)
    if b < b min then    } O(1)
        b min = b
for i in I :                      O(n)
    if not (i[0] < b <= i[1]) then    O(1)
        return False
return True
```

$O(n)$

5. (+2 pts extra) Sea A la matriz de adyacencia de una gráfica G con n vértices. Toma enteros i, j en $\{1, \ldots, n\}$ y un entero positivo k. Demuestra por inducción que la entrada ij de la matriz A^k es igual a la cantidad de caminos que hay del vértice i al vértice j que tengan longitud k. Usa esto para proponer un algoritmo que calcule esta cantidad de caminos y analiza su tiempo de ejecución.

Dem Procedemos por inducción sobre k (longitud de los caminos)

- Caso base: $k=1$
  Entonces A^1 es la matriz de adyacencias original y esto nos dice que si en A_{ij} hay 1 arista y en particular (sup. que G es simple) hay un camino de longitud $k=1$, si A_{ij} es 0 entonces no hay arista de i a j y por tanto no habrá un camino de longitud 1.

- H.I: Supongamos que la entrada ij de A^{n-1} es la cantidad de caminos de i a j de longitud ~~n-1~~

- Paso Inductivo: P.D A^n cumple la propiedad

Sea $c = i, \dots, u, j$ un camino de i a j de longitud n, notemos que $i, \dots, u$ es un camino de longitud $n-1$, los cuales por H.I hay A^{n-1}_{iu}. Entonces el número de caminos de i a j de longitud n será la suma del número de caminos de longitud $n-1$ de i a cada vecino $v \in N(j)$:

$$\sum_{v \in N(j)} A^{n-1}_{iv} = \sum_{v \in N(j)} A^{n-1}_{iv} \cdot \underbrace{A_{vj}}_{\substack{\text{como } v \text{ es adyacente} \\ \text{a } j \text{ entonces este} \\ \text{valor es } 1.}} = \overbrace{A^n_{ij}}^{A^{n-1}A = A^n}$$

Q.E.D

Algoritmo propuesto:

1. Definimos una función multiplicación de dos matrices de $n \times n$, en pseudocódigo queda: $O(n^3)$

```
func mult(A, B):
    C = matriz de nxn
    for i from 0 to n-1:                        O(n^3)
        for j from 0 to n-1:                    O(n)
            for k from 0 to n-1:                O(n)
                C[i][j] = C[i][j] + A[i][k] * B[k][j]   O(n)
    return C
```

2. Definimos la función que cuenta el número de caminos de i a j con longitud k:

```
func cuenta_caminos(A, i, j, k):
    A_k = A
    for i from 1 to k:          O(1)
        A_k = mult(A_k, A)      O(n^3)
    return A_k[i][j]
```

$k\,O(n^3) = O(k n^3) = O(n^3)$

si $k = n$ entonces $O(n^4)$

Por lo tanto la complejidad del algoritmo será $O(n^3)$ mientras $k < n$, de otra forma la complejidad será $O(n^4)$