Daniel Aguilera - 202010592

William Mendez - 202012662

Juliana Galeano - 202012128

Boris Reyes - 202014743

Tarea 5

- Cree un algoritmo correcto y eficiente para hacer una conversión entre las siguientes estructuras de datos. Suponga que G no es dirigido con n vértices y m aristas. Cree el programa en GCL (no es necesaria la pre/post condición) y analice la complejidad.

  ♣ Convertir de una matriz de adyacencias a una lista de adyacencias.

  ♣ Convertir de una lista de adyacencias a una matriz de adyacencias.

- **Convertir de una matriz de adyacencias a una lista de adyacencias**

```
fun adjListConversion(A: Matrix of int) ret B: HashMap<int, List of int>
    var B: HashMap<int, List of int >
    var i, j: int
        var m: list of int
    i,j := 0,0;                                          → c1
    do i < A.length() -> ;                               → c2(n + 1) + c.length
            m := [ ]                                     → nc1
            B.put(i+1, m);                               → n(c.put + c3)
            do j < A[i].length() ->                      → nc2(m + 1)
                    if A[i][j] = 1                       → nm(c2)
                        B.get(i+1).append(j+1);          → nm(c.get + 2c3 + c.append)
                    fi
                    j := j+1;                            → nm(c1 + c3)
            od
            i := i+1;                                    → n(c1 + c3)
            j := 0;                                      → nc1
    od
    ret B
```

$c1 + c2(n + 1) + c.length + nc1 + n(c.put + c3) + nc2(m + 1) + nm(c2) + nm(c.get + 2c3 + c.append) + nm(c1 + c3) + n(c1 + c3) + nc1$

$3c3nm + c1nm + 2nc2m + nmc.get + nmc.append + 2c3n + nc.put + c1 + 3nc1 + 2nc2 + c2 + c.length$

$nm(3c3 + c1 + 2c2 + c.get + c.append) + n(2c3 + c.put + 3c1 + 2c2 + c.length) + c1 + c2$

Reemplazamos $3c3 + c1 + 2c2 + c.get + c.append$ por k1, $2c3 + c.put + 3c1 + 2c2 + c.length$ por k2, y $c1 + c2$ por k3

$$\boldsymbol{nmk1 + nk2 + k3}$$

Con esto podemos ver que nuestro GCL es **O(nm)**

- Convertir de una lista de adyacencias a una matriz de adyacencias.

```
fun matrixConversion(A: HashMap<int, List of int>) ret B: Matrix of int
var B: Matrix of int
var M: Array of int
var i, j, k: int
i,j := 0,0;                                          → c1
M := A.keySet().toArray()
do i < M.length()->                                  → n(c2)
    do j <A.get(M[i]).length() ->                    → nc2(c.get)
        k := A.get(M[i])[j]                          → nm(c.get + c1)
        B[i][k] := 1;                                → nm(c1)
        j := j+1;                                    → nm(c1 + c3)
    od
    j := 0;                                          → n(c1)
    i := i+1;                                        → n(c1 + c3)
od
ret B
```

$\rightarrow c1 + nc2 + nc2(c.get) + nm(c.get + c1) + nm(c1) + nm(c1 + c3) + n(c1) + n(c1 + c3)$

$\rightarrow c1 + 2nc1 + 2c3 + nc2 + nc2\,c.get + 2nmc1 + nm\,c.get$

$\rightarrow c1 + 2nc1 + 2c3 + nc2 + nc2\,c.get + 2nmc1 + nm\,c.get$

$\rightarrow c1 + n(2c1 + c2 + c2 + c.get) + 2c3 + nm(2c1 + c.get)$

Reemplazamos $2c1 + c2 + c2 + c.get$ por k2 , $2c1 + c.get$ por k1 y c1 + 2c3 por k3

$$nmk1 + nk2 + k3$$

Al igual que el anterior podemos ver que tiene una complejidad de **O(nm)**

- Implemente (Python, Java) el algoritmo de Prim's y Kruskal utilizando como estructura de datos de grafo:
  ♣ 1. Lista de adyacencias para Prim's.
  ♣ 2. Matriz de adyacencias para Kruskal.
  ♣ Documente su implementación y como se implementó cada una de las estructuras (hay múltiples estrategias).
  ♣ Verifique la complejidad computacional de su implementación.

1. **Prim's**

   Para implementar la lista de adyacencias se utilizó una lista de Python, la cual tiene n posiciones dependiendo del número de vértices (el número de vértices se obtiene del usuario al inicializar el grafo), además en cada posición de la lista se guarda otra lista que contiene las adyacencias del vértice de esa posición. Es importante aclarar que estamos trabajando sobre un grafo no dirigido en el que todos sus arcos tienen pesos.

   Para el algoritmo de Prim's se utiliza un minHeap, primero por cada uno de los vértices del grafo se crea un nodo en el Heap el cual contiene su vértice correspondiente en el grafo y una key que corresponderá al peso mínimo (todos los nodos se inicializan con su llave en tamaño máximo), luego con el árbol ya creado se recorren los diferentes nodos mientras el árbol no esté vacío. Por cada nodo recorremos sus vértices adyacentes (solo se verifica un nodo si este aún hace parte del heap, esto se verifica por medio de una lista con booleanos), por cada adyacente obtenemos su posición en el grafo y el peso del arco que une a ese adyacente con el nodo que estamos revisando, si este peso es menor que la llave guardada en el nodo del árbol, se remplaza la llave por el peso, se reorganiza el árbol y se guarda este adyacente en una lista que servirá para dar el resultado final del algoritmo que será el mínimo árbol de expansión. La lista del resultado se actualiza con cada recorrido hasta que obtengamos solo los pesos mínimos que nos permiten recorrer todo el grafo.

```
def Prims(self):
        inHeap = [False] * (self.no_vertices)                    C1
        resultSet = [None] * (self.no_vertices)                  C1
        key = [0] * (self.no_vertices)                           C1
        heapNodes = [None] * (self.no_vertices)                  C1
        i = 0                                                    C1

        while (i < self.no_vertices) :                          C2+n(adentro)
            heapNodes[i] = HeapNode()                            C1
            heapNodes[i].vertex = i                              C1
            heapNodes[i].key = sys.maxsize                       C1
            resultSet[i] = ResultSet()                           C1
            resultSet[i].parent = -1                             C1
            inHeap[i] = True                                     C1
            key[i] = sys.maxsize                                 C1
            i += 1                                               C1+C3
```

```
heapNodes[0].key = 0                                        C1
minHeap = Heap(self.no_vertices)                            C1
j = 0                                                       C1
while (j < self.no_vertices) :                          C2+ n(adentro)
    minHeap.insert(heapNodes[j])                             log(n)
    j += 1                                                  C1+C3

i = 0                                                       C1
while (minHeap.isEmpty() == False) :                   C2+n(adentro)
    extractedNode = minHeap.extractMin()               C1+ 1
    extractedVertex = extractedNode.vertex                  C1
    inHeap[extractedVertex] = False                         C1
    while (i < len(self.vertices[extractedVertex])) :       C2 + n
        edge = self.vertices[extractedVertex][i]            C1
        if (inHeap[edge.v]) :                               C2
            v = edge.v                                      C1
            w = edge.weight                                 C1
            if (key[v] > w) :                               C2
                key[v] = w                                  C1
                self.decreaseKey(minHeap, w, v)             log(n)
                resultSet[v].parent = extractedVertex       C1
                resultSet[v].weight = w                     C1
        i += 1                                              C1 + C3
    i = 0                                                   C1
result = 0                                                  C1
node = 1                                                    C1
```

$5C1+ C2 + n(8(C1)+C3)+3(C1)+ C2 + n(\log(n)+C1+C3) + C1 + C2 + n(C1+1 + 2C1+C2+ n (C1 + C2 + 2(C1) + C2 + C1 + \log(n) + 3(C1) + C3) + C1) +2(C1)$

$9C1 + 3C2 + 13nC1 + nC2 + 2nC3 + n + n\log(n) + 7n^2C1 + 2n^2C2 + n^2C3 + n^2\log(n)$

Bajo el orden natural de los números, las constantes se omiten puesto que resultan irrelevantes dejando la expresión:

$4n + n\log(n) + 3n^2 + n^2\log(n)$

Por orden natural de las expresiones, se obtiene que la complejidad es de:

$$O(n^2 \log(n))$$

2. **Kruskal**

Para implementar la matriz de adyacencias se utilizó una matriz de Python en la estructura del grafo en la cual se almacenan según los índices i, j en la matriz los pesos del arco si los hay, y si no se almacena un -1. Esta matriz se mantiene de forma simétrica para considerar el caso del grafo dirigido.
Además, en la clase Graph se almacena un diccionario con los valores de los vértices y una variable llamada vertices_no que almacena la cantidad de vértices del grafo.
En el programa se implementa un método para revisar si agregar un arco de v1 a v2 introduce un ciclo, este método almacena en las llaves de un diccionario los vértices visitados y una cola tipo LIFO y realiza una búsqueda DFS iniciando en v1 en la que si se llega a v2 se sabe que existiría un ciclo si se agrega el arco, este algoritmo tiene una complejidad de O(v), donde v es el número de vértices del grafo.
La implementación de Kruskal diseñada crea un grafo de resultado que será el árbol de recubrimiento, crea una lista de tuplas para los arcos del árbol original de la forma (peso, vertice1, vertice2) llamado edges que será ordenado de mayor a menor según el peso del arco. Finalmente se crea una variable nEdges que almacena la cantidad de arcos que contiene el árbol de recubrimiento, pues sabemos que necesitamos V – 1 arcos para un árbol de V vértices.
Primero, el algoritmo almacena todos los vértices en edges, lo cual tiene una complejidad de V*V. Luego se repetirá V -1 veces lo siguiente: Se toma el último valor de la lista edges y se elimina ya que se sabe que es el de menor peso, se revisa si agregar este nodo genera un ciclo en el grafo de resultado, si no, se agrega y se suma uno a la varable nEdges. Luego de tener V- 1 vértices se retorna el árbol de resultado

```
def kruskal(self):
    resul = Graph(self.vertices_no)                          O(V)
    edges = []                                               c1
    nEdges = 0                                                       c1

    for i in range(self.vertices_no):                        O(V)
      for j in range(i, self.vertices_no):
        O(V)
        if self.adjacencies[i][j] != -1:                     c2
          edges.append([self.adjacencies[i][j], i, j])       c1

    edges = sorted(edges, key=lambda x: x[0], reverse=True)  O(E*Log(E))

    while nEdges < self.vertices_no - 1:                     c2 + O(E)
      weight, v1, v2 = edges.pop()                           c1 + O(1)
      if resul.checkCycle(v1, v2) is False:                  O(V)
        resul.add_edge(v1+1, v2+1, weight)                   O(1)
        nEdges = nEdges + 1                                  c1 + c3

    return resul
```

Complejidad:

$$O(V)+c1+c1+O(V)(O(V)(c2+c1))+O(E*Log(E))+(c2+O(E))(c1+O(1)+O(V)+O(1)+c1+c3)$$

=

O(V)+k1+O(V)(O(V)(k2))+O(E*Log(E))+(c2+O(E))(O(1)+O(V)+k3)

=

O(V)+k1+(k2)O(V^2)+O(E*Log(E))+ k5+ c2*O(V)+k4 + O(E)O(1)+ O(E)O(V)+k3O(E)

=

O(V)+O(V^2)+O(E*Log(E))+O(V)+O(E)+O(EV)+O(E)

=

O(2V+2E+EV+V^2+E*Log(E))