# Universidad Nacional Autónoma de México
# Facultad de Ciencias
# Complejidad Computacional 2024-1
# Práctica 01

Profesora: María de Luz Gasca Soto
Ayudante: Brenda Margarita Becerra Ruíz
Ayudante: Malinali González Lara

Fecha de entrega: 9 de Marzo del 2024

**López Diego Gabriela 318243485**

## 1. Ejercicios

1. Problema de decisión: Ruta más corta: Dada una gráfica no dirigida G = (V,E), u, v vértices en V, y k entero positivo, ¿Existe una uv−trayectoria en G de peso menor a k?

   - Forma canonica:
     Dada una gráfica NO dirigida G = (V,E) donde V es el cojunto de vértices y E el conjunto de aristas. Consideremos los vértices u, v ∈V y k $\geq$ 1, ¿Existe alguna uv-trayectoria en G tal que la suma del peso de todas las aristas que lo conforman sea menor que k?
   - Algoritmo no−determinístico polinomial.

     I. Dado un archivo .txt, guardamos en una lista el conjunto de vértices y en otra lista el conjunto de aristas que definen nuestra gráfica.

     II. Guardamos en alguna estructura de datos como un diccionario, los vértices y a cada uno le asignaremos otro diccionario. Estos otros diccionarios internos, se encargaran de contener los vértices de los cuales son adyacentes y el respectivo peso (aleatorio entre 1 a 5) que tiene la arista con dicho vértice. En esta parte, consideramos que G es no dirigida, por lo cual si tenemos el par de aristas (A,B) y le asignamos algún peso, también lo hacemos para el par (B,A).

     III. Escogemos de forma aleatoria el vértice inicial u y final v de nuestra trayectoria. Nos aseguramos que ambos vértices sean diferentes y que pertenezcan al conjunto de vértices V de la gráfica.

     IV. Creamos una variable pesoK con un valor fijo de 20

     V. **Fase 1 de nuestro algoritmo no determinista: Conjetura**.
     Crearemos alguna estructura de datos digamos *trayectoria* que nos sea de ayuda para almacenar los vértices y aristas que iremos generando para llegar del vértice u a v de nuestra gráfica. Como el vértice inicial es u, lo agregamos como primer elemento. Partiendo de u, avanzaremos a algún vértice vecino de forma aleatoria y lo agregaremos a la variable *trayectoria*. Repetiremos el procedimiento hasta dar con el vértice final v. Si ocurre lo anterior, quiere decir que hemos terminado y encontrado una uv−trayectoria.
     Aquí recordemos que nuestra gráfica es no dirigida, por lo que en nuestra trayectoria podemos visitar más de una vez el mismo vértice.

     VI. **Fase 2 de nuestro algoritmo no determinista: Verificación.**
     En esta parte de nuestro algoritmo, procedemos a verificar si la trayectoria generada en el paso anterior cumple con el siguiente criterio

     *¿Existe alguna uv−trayectoria en la gráfica G tal que la suma del peso de todas las aristas que la conforman sea menor estricto a k?*

     Por lo cual, sumaremos el peso de cada una de las aristas que construyen la trayectoria. Si y solo si, el peso total de la trayectoria es menor que K entonces **SI** cumple el criterio.

Vemos que nuestro algoritmo es no determinista ya que cumple con las dos fases correspondientes, es decir, la fase 1 donde se va contruyendo diferentes soluciones en cada ejecución a pesar de siempre manejemos la misma gráfica y variable K y aunque los vértices u y v se escogen de manera aleatoria, en cualquier otra ejecución que tengamos los mismos vértices u y v, el resultado será diferente. De igual forma, se tiene la fase 2 que verifica si cumple con algún criterio dado. En este caso, si el peso de una trayectoria en particular es menor que k.

Por otro lado, vemos que nuestro algoritmo pertenece a la clase NP (polinomial no determinista) pues en la fase de verificación se tiene complejidad de tiempo lineal O($n$) donde n es el número de aristas en la trayectoria. Aunque se tenga O(n) cumple con ser complejidad de tiempo polinomial pues es un polinomio de grado 1.

- Implementación del algoritmo

```
import random

def crear_trayectoria(grafica, u, v):
    #Verificamos que u y v pertenezcan al conjunto de vertices de la grafica
    if u not in grafica:
        print(f"El vertice de inicio {u} no existe en la grafica. Intente con un vertice valido.")
        exit()
    elif v not in grafica:
        print(f"El vertice final {v} no existe en la grafica. Intente con un vertice valido.")
        exit()

    #FASE 1 CONJETURA : Creamos una trayectoria del vertice u al vertice v, de forma no determinista.
    trayectoria = [u]  # La trayectoria comienza en el v rtice de inicio
    vertice_actual = u

    while vertice_actual != v:
        #Obtenemos las claves del diccionario del elemento [vertice actual] que son los vecinos
        #del vertice actual y lo convertimos a tipo lista
        vecinos = list(grafica[vertice_actual].keys())
        #Elegimos un vertice aleatorio de los vecinos
        vertice_aleatorio = random.choice(vecinos)
        trayectoria.append(vertice_aleatorio)
        vertice_actual = vertice_aleatorio # Nos movemos al siguiente v rtice

    return trayectoria


def verificar(grafica, k, trayectoria):
    # Fase 2 VERIFICACION :Verificamos si la trayectoria dada cumple con tener peso menor a k
    peso_trayectoria = 0
    num_aristas = len(trayectoria)

    for i in range(num_aristas - 1):
        #Obtenemos el peso de la arista que conecta el vertice i con el vertice i+1
        peso_arista = grafica[trayectoria[i]][trayectoria[i+1]]
        peso_trayectoria += peso_arista

    if peso_trayectoria < k:
        print("La trayectoria dada >>SI<< tiene peso menor a k. El peso de la trayectoria es", peso_trayectoria)
    else:
        print("La trayectoria dada >>NO<< tiene peso menor a k. El peso de la trayectoria es", peso_trayectoria)


#Leemos un archivo .txt donde contendra los vertices en la primera linea
#y las aristas en las siguientes lineas (cada arista en cada linea)
try:
    archivo = open("Grafo1.txt", "r")
    vertices = archivo.readline().strip().split(',') #Leemos la primera linea y la separamos por comas
    vertices = [str(i) for i in vertices] #Convertimos los vertices a string
    aristas = []

    for linea in archivo: #Continuamos con el resto del archivo, leemos las aristas y las separamos por comas
```

```
        arista = tuple(linea.strip().split(','))
        aristas.append(arista)

    archivo.close()
except FileNotFoundError:
    print("No se encontr  el archivo Grafo1.txt")
    exit()


# Creamos un diccionario para representar la grafica
grafica = {}

for vertice in vertices:
    grafica[vertice] = {}

# Asignamos aleatoriamente un peso entre 1 y 5 a cada arista de G
for arista in aristas:
    vertice1, vertice2 = arista
    peso = random.randint(1, 5)
    grafica[vertice1][vertice2] = peso
    grafica[vertice2][vertice1] = peso  # Por tratarse de una grafica no dirigida, la arista (
    v1, v2) es la misma que (v2, v1)

aleatorio1 = random.randint(0, len(vertices)-1)
inicio_u = vertices[aleatorio1]  #V rtice de inicio aleatorio
aleatorio2 = random.randint(0, len(vertices)-1)

#Nos aseguramos que el vertice final sea diferente al vertice de inicio
while aleatorio2 == aleatorio1:
    aleatorio2 = random.randint(0, len(vertices)-1)

final_v = vertices[aleatorio2]  #V rtice final aleatorio
peso_k = 20   #Peso. Queremos ver si existe una uv-trayectoria que cumpla con un peso menor a
    k

print("\nLa grafica G es " , grafica
        , "\nEl vertice de inicio es ", inicio_u
        , "\nEl vertice final es ", final_v
        , "\nEl peso k es: ", peso_k)

trayectoria_adivinadora = crear_trayectoria(grafica, inicio_u, final_v)
print("La uv-trayectoria es: ", trayectoria_adivinadora)
verificar(grafica, peso_k, trayectoria_adivinadora)
```

2. 3−SAT

- Forma canonica:
  Dada una fórmula normal conjuntiva con 3 variables en cada clausula, ¿Existe alguna asignación de valores de verdad a las variables que haga que la formula sea verdadera?
- Algoritmo no−determinístico polinomial.

  I. Dado el archivo .txt, obtenemos la fórmula en FNC
  (x + -y + z) * (x + y + z) * (-x + -y + -z) * (-x + y + -z) donde * representará el operador booleano **and**, + **or** y - **not**.

  II. Luego, conseguimos el conjunto de literales de nuestra fórmula. En este caso, {x,y,z}

  III. **Fase 1 de nuestro algoritmo no determinista: Conjetura**
  Para esta parte, simplemente asignaremos de manera aleatoria un valor de verdad (True o False) para cada una las literales de la fórmula.

  IV. **Fase 2 de nuestro algoritmo no determinista: Verificación**
  De acuerdo a la asignación de valores de verdad que le otorgamos a cada una de las variables, solamente sustituimos y evaluamos toda la fórmula. Si obtenemos que es **Verdadera** entonces el críterio dado anteriormente se cumpliria, en caso contrario, no.

  De igual forma, nuestro algoritmo cumple con ser no deterministico polinomial (clase NP-completo) ya que en la fase conjetura se encarga de asignar de forma no deterministica, valores de verdad a cada una de las variables de la fórmula. Por otro lado, la fase de verificación cumple con tener tiempo polinomial pues se tiene complejidad de tiempo O(n) ya que va aumentando proporcionalmente al número de clausulas en la fórmula.

- Implementación del algoritmo

```python
import random

def conjetura(variables):
    asignaciones = {}
    #Asignamos valores de verdad a cada una de las literales de forma no determinista (aleatoria)
    for variable in variables:
        asignaciones[variable] = random.choice([True, False])
    return asignaciones

def verificacion(asignaciones, formula):
    #Funcion que se encarga de evaluar toda la formula, verifica si es V o F
    sustitucion_clausulas = []

    for clausula in formula:
        for literal, asignacion in asignaciones.items():
            # Sustituimos cada literal por su respectivo valor de verdad
            clausula = clausula.replace(literal, str(asignacion))
            clausula = clausula.replace('-', 'not ')
        clausula = clausula.replace('+', 'or')
        sustitucion_clausulas.append(clausula)

    #Almacenamos los valores de verdad de cada clausula
    valores_clausulas = []
    for clausula in sustitucion_clausulas:
        valores_clausulas.append(eval(clausula))

    #Ahora evaluamos si TODA la formula es V o F
    return all(valores_clausulas)


#Leemos un archivo .txt que contiene la formula FNC
#Recibimos un texto que contiene la formula en la primera linea
#(x + -y + z) * (x + y + z) * (-x + -y + -z) * (-x + y + -z)
try:
    archivo = open("3sat.txt", "r")
    clausulas = archivo.readline().strip()
    archivo.close()
except FileNotFoundError:
    print("El archivo no existe")
    exit()


lista_clausulas = clausulas.split("*")

#Extraemos las literales de la formula
conjunto_literales = set()

for clausula in lista_clausulas:
    clausula = clausula.strip().replace("(", "").replace(")", "")
    literales = clausula.split("+")
    for literal in literales:
        literal = literal.strip().replace("-", "")
        conjunto_literales.add(literal)

conjunto_literales = sorted(conjunto_literales)
lista_literales = list(conjunto_literales)

asignaciones = conjetura(lista_literales)
valor_de_verdad_formula = verificacion(asignaciones, lista_clausulas)

print(f"\nFormula en FNC:{clausulas}")
print(f"Las literales de la formula: {lista_literales}")
print(f"Asignaciones de valores de verdad para las literales: {asignaciones}")
print(f"Con lo anterior, la formula en FNC es: {valor_de_verdad_formula}")
```

## 2. Referencias


- NP (complejidad). (s/f). Academia-lab.com. Recuperado el 3 de marzo de 2024, de `https://academia-lab.com/enciclopedia/np-complejidad/`