	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Jorge Solano
Asignatura:	Estructura de Datos y Algoritmos II
Grupo:	2
No de Práctica(s):	7
Integrante(s):	Martínez Ostoa Néstor Iván
Semestre:	2019-1
Fecha de entrega:	5/Octubre/2018
Observaciones:	

CALIFICACIÓN: _____

1. Introducción

Los grafos son estructuras de datos muy importantes en las ramas de las ciencias de la computación puesto que poseen una gran eficiencia para guardar información y navegar sobre de ella. Los grafos nos acompañan en todos nuestra comunicaciones día a día, desde las redes sociales hasta las telecomunicaciones. Para esta práctica, implementamos un algoritmo de búsqueda por profundidad. O, mejor conocido como Depth First Search (DFS). La característica de este algoritmo es que recorre todos los nodos del grafo con el objetivo de conocer todos los nodos que visita.

Este algoritmo explora de manera sistemáticamente todas las aristas de una grafo G, primero visitando los nodos adyacentes partiendo de un nodo raíz. Con cada iteración, o al terminar la lista de adyacencia de un nodo en particular, DFS generará un árbol.

2. Objetivos

Conocer e identificar las características necesarias para entender e implementar el algoritmo de búsqueda por profundidad en un grafo G.

3. Desarrollo

3.1 Código DFS en Python

```
# Depth First Search algorithm
def depthFirstSearch(g):
    global times
    for u in range(g.numNodes+1):
            colors.append("White")
            predecessors.append(None)
            distances.append(0)
            finishes.append(0)
    times = 0
    for u in range(g.numNodes):
        if u > 0:
            if colors[u] == "White":
                print("Visiting u : " + str(u))
                visit(u)
```

```
# Function that visits all adjacent nodes to a node U
def visit(u):
    global times
    global colors
    global distances
    global finishes
    global predecessors

    colors[u] = "Gray"
    times += 1
    distances[u] = times
    # print("U " + str(u))
    # print(colors[u])
    v = g.edges[u]
    while v != None:
        V = v.to
        print(str(u) + ":" + colors[u] + " -> " + str(V) + ":" + colors[V])
        if colors[V] == "White":
            predecessors[V] = u
            visit(V)
        v = v.previous

    colors[u] = "Black"
    times = times + 1
    finishes[u] = times
```

3.2 Clases extra utilizadas

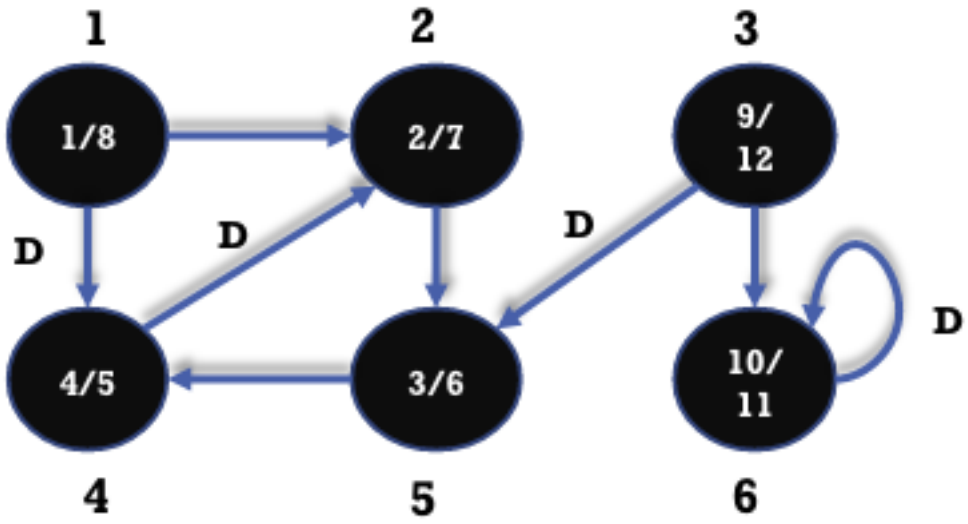
```
# Main function
def main():
    setDirectedCostGraph()
    setNumNodesNumEdgesAndCreateGraph()
    depthFirstSearch(g)
    printUpdatedGraph(g)
```

```python
# Basic set up functions to start and create the graph
def setDirectedCostGraph():
    g.isDirected = True if int(input("Directed (1) ? Not Directed (2)? : ")) == 1 else False
    g.hasCost = True if int(input("Cost (1) ? No Cost (2)? : ")) == 1 else False

def setNumNodesNumEdgesAndCreateGraph():
    MAXV = int(input("Number of nodes: "))
    s=(MAXV,MAXV)
    matrix = np.zeros(s)
    startGraph(g, MAXV)
    g.numNodes = MAXV
    g.numEdges = int(input("Number of edges: "))
    createGraph(g, matrix, totalGraph)
    printGraph(g)
```

3.3 Prueba de funcionamiento

```python
main()
```

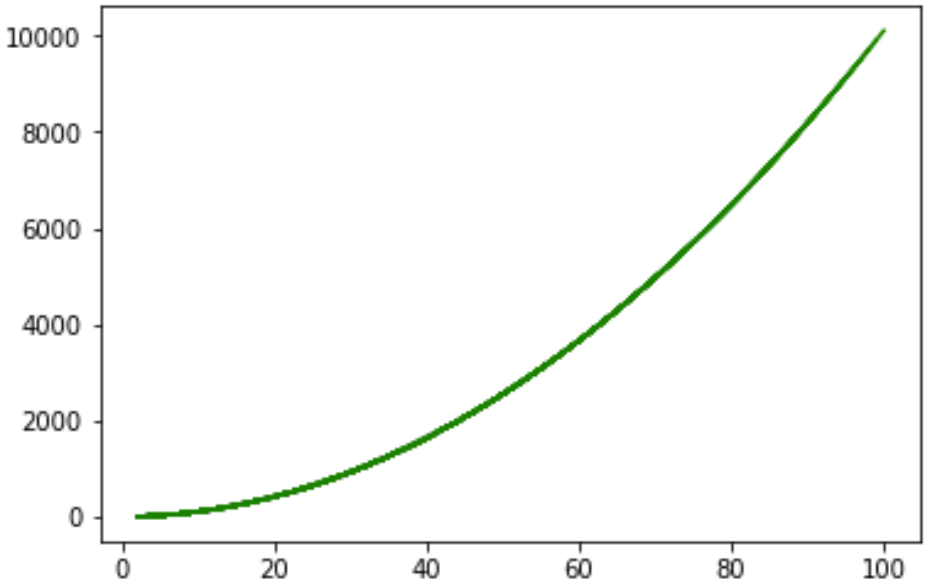
```
Directed (1) ? Not Directed (2)? : 1
Cost (1) ? No Cost (2)? : 2
Number of nodes: 6
Number of edges: 8
1        4 : 0    2 : 0
2        5 : 0
3        6 : 0    5 : 0
4        2 : 0
5        4 : 0
6        6 : 0

Visiting u : 1
1:Gray -> 4:White
4:Gray -> 2:White
2:Gray -> 5:White
5:Gray -> 4:Gray
1:Gray -> 2:Black
Visiting u : 3
3:Gray -> 6:White
6:Gray -> 6:Gray
3:Gray -> 5:Black
1 Black ( 1/8 )
2 Black ( 3/6 )
3 Black ( 9/12 )
4 Black ( 2/7 )
5 Black ( 4/5 )
6 Black ( 10/11 )
```



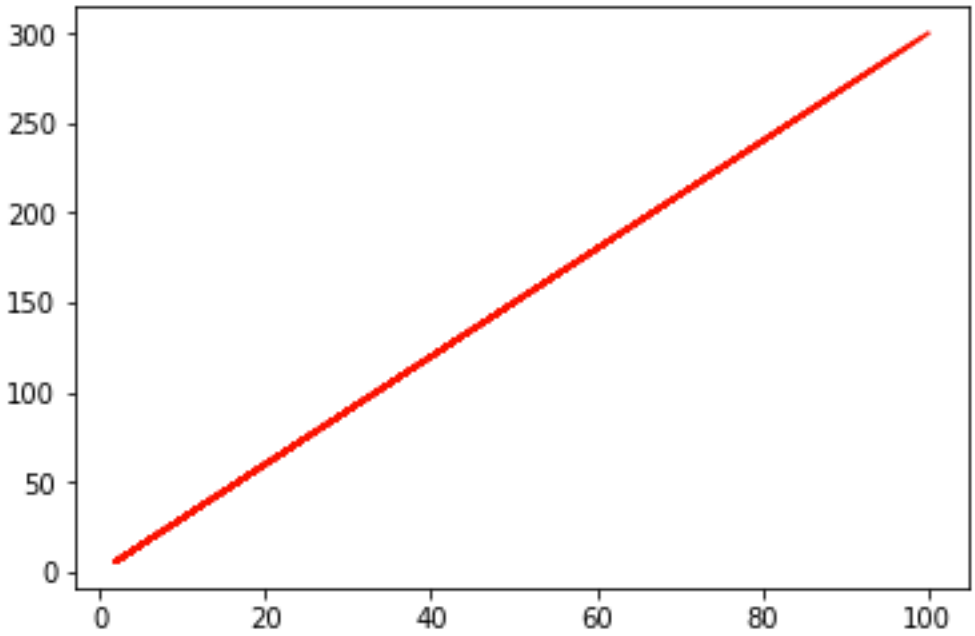
4. Gráficas

4.1 Peor Caso

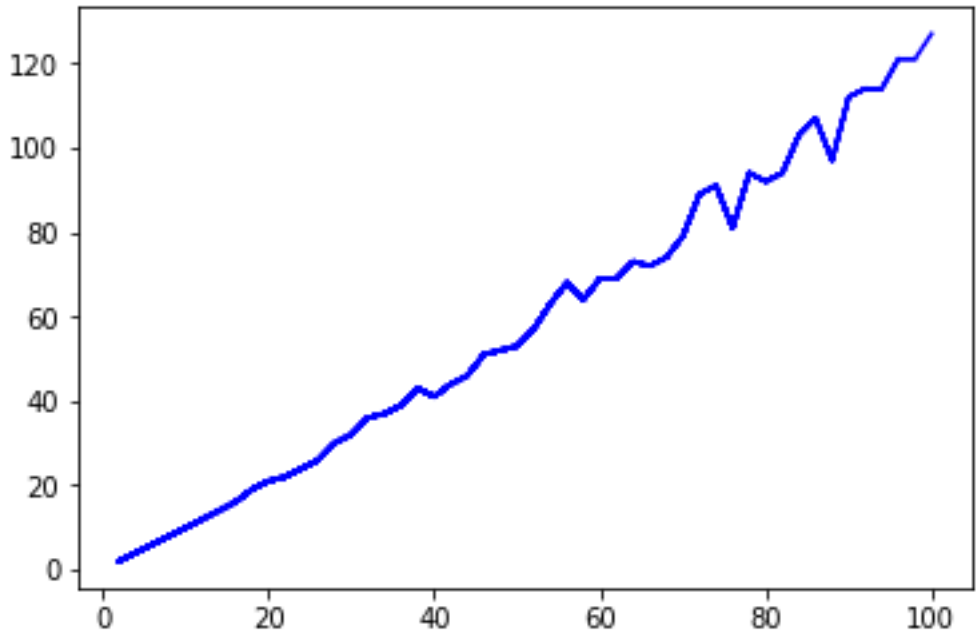
4.1.1 Grafo Completo



4.1.2 Grafo Cadena



4.2 Caso Promedio



5. Conclusiones

A diferencia de BFS, para este algoritmo, en el caso en el que tratamos con un grafo en cadena, la complejidad en DFS es lineal. Lo cual presenta una mejora sustentable al caso de BFS en el que hablamos de una complejidad cuadrática.

Es entendible que para un grafo completo tengamos un tiempo cuadrático, pues el algoritmo se encarga de recorrer todos los vértices.