

Universidad Panamericana



Facultad de Ingeniería

Inteligencia Artificial

Profesor: Ari Yair Barrera Animas

Integrantes:

Mauricio Iván Ascencio Martínez_____0249220

Enrique Ulises Báez Gómez Tagle_____0241823

Daniel Eduardo Leal Córdova_____0241338

Ana Teresa Vega Zerecero_____0239131

México, CDMX a 22 de febrero del 2023

Table of Contents

Tabla de Figuras.....	1
Introducción	2
Grafo.....	3
Imagen	3
Descripción	3
Instrucciones	4
Leyendo el código	4
Funciones/Clases/Objetos.....	5
Orden de ejecución y Descripción.....	5
Ejemplos.....	11
Restricciones.....	15

Tabla de Imágenes

Figure 1. Grafo Dirigido	3
Figure 2. Código del Grafo	5
Figure 3. Código de Búsqueda a lo Ancho	5
Figure 4. Código Búsqueda Limitada	6
Figure 5. Código de Búsqueda por Profundidad	6
Figure 6. Código de Búsqueda Iterada	7
Figure 7. Algoritmo de Dijkstra	7
Figure 8. Búsqueda Bidireccional	8
Figure 9 Parte 1 main.py	Error! Bookmark not defined.
Figure 10 Parte 2 main.py	10
Figure 11 Parte 3 main.py	Error! Bookmark not defined.1
Figure 12 Parte 4 main.py	Error! Bookmark not defined.
Figure 13 Parte 5 main.py	Error! Bookmark not defined.
Figure 14 & 15 Ejecución 1.....	Error! Bookmark not defined.
Figure 15 & 16 Ejecución 1.....	Error! Bookmark not defined.
Figure 17 & 18 Ejecución 2.....	Error! Bookmark not defined.
Figure 19 & 20 Ejecución 3.....	Error! Bookmark not defined.

Introducción

El grafo que se presenta a continuación es una simulación de una ruta que se quiere hacer en la República Mexicana.

El origen (nodo de inicio) es Cancún y el destino (nodo final) es el Cabo San Lucas principalmente, pero se puede tomar cualquier otra ciudad del mapa.

Con esta información, vamos a hacer un código cuya meta es recibir los nodos de inicio, final y el límite de profundidad deseado, para procesar el grafo con alguno de los métodos de búsqueda vistos en clase. El orden de los métodos será el siguiente:

- a) Búsqueda a lo Ancho: si el grafo entregado no es ponderado, solo se mostrará en pantalla el camino regresado por este algoritmo. En caso contrario ejecutará todos.
- b) Búsqueda por Profundidad
- c) Búsqueda en Profundidad Limitada
- d) Búsqueda en Profundidad Iterada
- e) Algoritmo de Dijkstra
- f) Búsqueda Bidireccional

Para poder demostrar el alcance completo de este proyecto, se usa el mismo grafo primero con los pesos diferentes que se especifican en el mapa y posteriormente, con los valores de los pesos iguales.

Grafo Dirigido

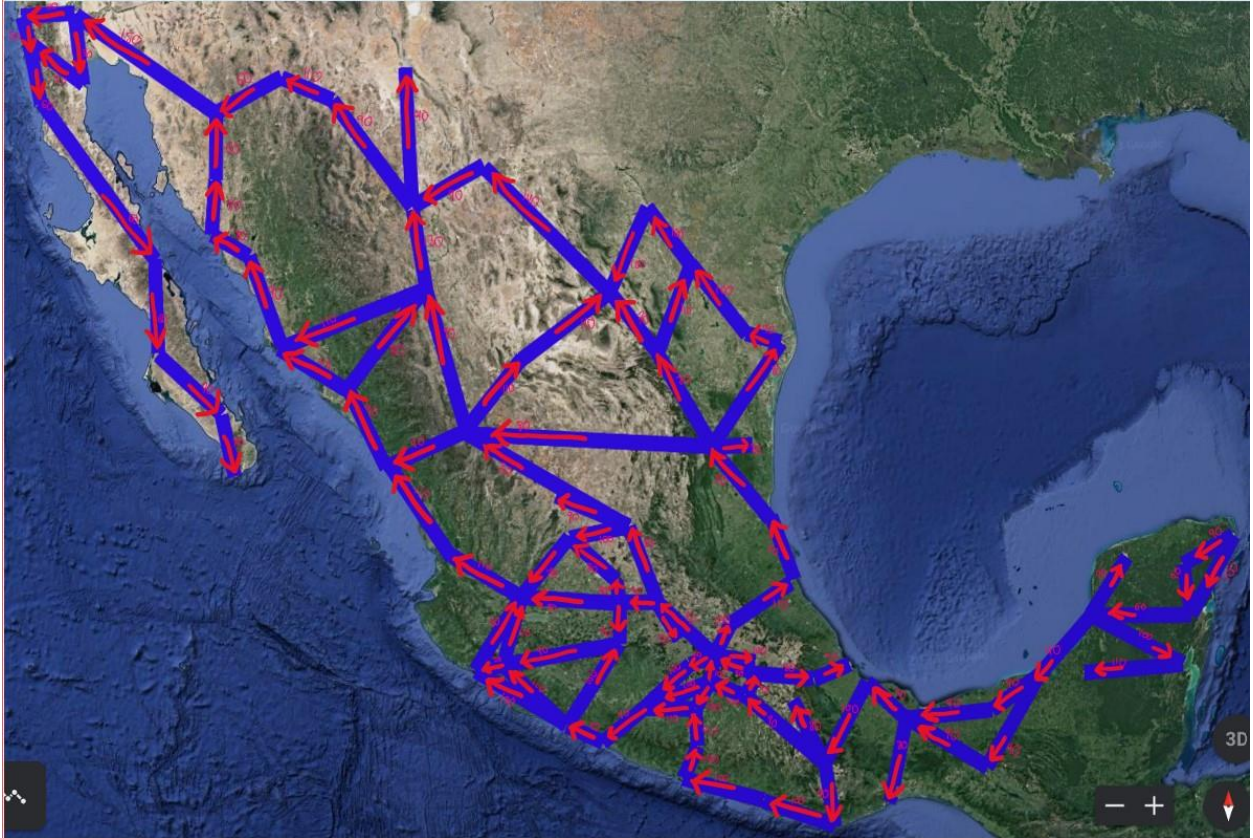


Figure 1. Grafo Dirigido

Formamos el grafo con dirección de Este a Oeste y Sur a Norte, como se muestra en la figura. Cada uno de los caminos solo va hacia una dirección específica y por al menos un camino se puede llegar a cada una de las ciudades.

Instrucciones

La carpeta enviada es un ZIP que contiene 11 archivos, 2 de ellos son los grafos dirigidos y con los pesos mientras que el resto de los archivos son los códigos.

Pasos:

1. Descomprimir el ZIP.
2. Dentro de la carpeta, dar clic derecho, seleccionar “abrir terminal” y observar que se abre la terminal de la computadora.
3. Correr el archivo principal desde la consola, con el comando *python main.py*
4. Ingresar el nodo de origen y el nodo de destino, recomendamos poner que la ruta sea de Cancún a Cabo San Lucas, ya que son los 2 extremos del mapa y se puede observar mejor el funcionamiento del código.
5. Saldrá el algoritmo de anchura y la ruta por la cual llegar. En el caso de ser un grafo ponderado, pedirá el límite de profundidad, el cual tiene que ingresar el usuario para continuar con los demás algoritmos.
6. Mostrará las demás rutas y métodos de ejecución que tiene el proyecto.

NOTA 1: en el código del main, se puede cambiar entre los 2 grafos requeridos (el ponderado y el no ponderado). Para poder usar el grafo de su preferencia, ubicarse en las líneas 19 y 20 del código, donde podrá comentar y descomentar el grafo que quiera usar u ocultar.

NOTA 2: Si se pone el valor del límite menor al valor en el que se llega al nodo deseado, no regresa camino.

Funciones, Clases y Objetos

Graph.py

```
class Graph:
    def __init__(self):
        self.content = dict()

    def new_edge(self, origin, destiny, weight):
        if origin not in self.content:
            self.content[origin] = []
        if destiny not in self.content:
            self.content[destiny] = []
        self.content[origin].append((destiny, weight))

    def view_all(self):
        print("\n\nGraph:\n-----")
        print("ORIGIN -> [(DESTINY, WEIGHT), ...]")
        for origin, destiny in self.content.items():
            print(f"{origin} -> {destiny}")
        print("\n")
```

Figure 2. Código del Grafo

Define una clase llamada Graph que representa a nuestro grafo, que es almacenado en el atributo content del objeto Graph.

Tiene tres métodos:

- A) “__init__” es el constructor de la clase y se llama cuando se crea un nuevo objeto de la clase. Inicializa el atributo content como diccionario vacío.
- B) “new_edge” agrega una nueva arista al grafo. Toma los argumentos: origin, destiny y weight y hace una entrada en el diccionario del contenido para el nodo de origen si aún no existe.

Luego, agrega una tupla a la lista correspondiente. La tupla tiene el nodo final o destino y el peso de la

arista.

- C) “view_all” se usa para poder visualizar el grafo. Imprime una cabecera y luego itera sobre las claves del diccionario content. Para cada clave o nodo del grafo, se imprime el nodo y la lista de tuplas que corresponde a sus aristas y cada tupla es un nodo destino y el peso de la arista.

breadth.py

```
def breadth_first_search(graph, origin, destiny):
    queue = [[origin]]
    while queue:
        path = queue.pop(0)
        node = path[-1]
        if node == destiny:
            return path
        for neighbor in graph.content[node]:
            new_path = list(path)
            new_path.append(neighbor[0])
            queue.append(new_path)
    return None

def run(graph, origin, destiny, runner):
    if runner:
        print("\n\nBreadth First Search:\n-----")
        path = breadth_first_search(graph, origin, destiny)
        if path:
            return path
        else:
            print("No path found.")
            return None
```

Figure 3. Código de Búsqueda a lo Ancho.

Tiene 2 funciones para buscar un camino desde el origen hasta el destino:

- A) “breadth_first_search” toma 3 argumentos: el grafo como diccionario, el nodo de origen y el nodo de destino. Crea una cola con una lista que tiene al nodo de origen y mientras la cola no esté vacía, la función separa la primera ruta de la cola, identifica el último nodo y comprueba si es el nodo de destino y si es, devuelve la ruta. Si no, la función busca los vecinos del nodo y agrega una nueva ruta que incluya al vecino al final de la cola. En caso de no encontrar un camino, la función devuelve None.

- B) “run” tiene 4 argumentos: el grafo, el nodo de origen, el nodo de destino y la bandera "runner". Si "runner" es verdadera, se llama a "breadth_first_search" y busca un camino

desde el origen hasta el destino usando esta búsqueda. Si se encuentra un camino, la función devuelve la ruta. Si no se encuentra un camino, la función devuelve None. Pero si "runner" es falsa, la función no hace nada y devuelve None.

limited_depth.py

```
def limited_depth_search(graph, origin, destiny, limit):
    if origin == destiny:
        return [origin]
    if limit == 0:
        return None
    for node in graph.content[origin]:
        path = limited_depth_search(graph, node[0], destiny, limit - 1)
        if path:
            return [origin] + path
    return None

def run(graph, origin, destiny, runner, limit):
    if runner:
        print("\n\nlimited Depth Search:\n-----")
        path = limited_depth_search(graph, origin, destiny, limit)
        if path:
            return path
        else:
            print("No path found.")
            return None
```

Tiene 2 funciones:

A) "limited_depth_search" se usa para encontrar un camino entre el origen y el destino en el grafo y devuelve el camino como una lista de nodos. Si el límite se alcanza sin encontrar el destino, devuelve None.

B) "run" llama a limited_depth_search e imprime el resultado si runner es verdadero. Si se encuentra un camino, lo devuelve, y si no se encuentra, devuelve None.

Figure 4. Código Búsqueda Limitada

depth.py

```
def depth_first_search(graph, origin, destiny, path=None):
    if path is None:
        path = []
    path.append(origin)
    if origin == destiny:
        return path
    for neighbor in graph.content[origin]:
        neighbor = neighbor[0]
        if neighbor not in path:
            new_path = depth_first_search(graph, neighbor, destiny, path.copy())
            if new_path is not None:
                return new_path
    return None

def run(graph, origin, destiny, runner):
    if runner:
        print("\n\nDepth First Search:\n-----")
        path = depth_first_search(graph, origin, destiny)
        if path:
            return path
        else:
            print("No path found.")
            return None
```

Tiene 2 funciones:

A) "depth_first_search" tiene el argumento "path" que registra el camino que se está realizando. Si "path" es None, se crea una nueva lista vacía. Luego, el nodo de origen se adjunta al camino. Si el nodo actual es el de destino, regresa el camino. De otra manera, para cada vecino del nodo actual, si aún no está en el camino, se crea uno nuevo al llamar a "depth_first_search" recursivamente con ese vecino como el nuevo origen el mismo destino y una copia del camino actual. Si el nuevo camino no es None, regresa y si no se encuentra camino, regresa None.

B) "run" tiene el argumento "runner" que es del tipo booleano para determinar si la función debe imprimir el tipo de búsqueda. Si "runner" es True, se imprime la búsqueda, llama a "depth_first_search" y regresa el camino resultante si es que tiene. Si no hay camino, regresa None.

Figure 5. Código de Búsqueda por Profundidad

iterative_depth.py

```
import limited_depth as ld

def iterative_depth(graph, start, destiny):
    for depth in range(len(graph.content)):
        path = ld.limited_depth_search(graph, start, destiny, depth)
        if path:
            print("Path found at DEPTH: ", depth)
            return path
    return None

def run(graph, origin, destiny, runner):
    if runner:
        print("\n\nIterative Depth Search:\n-----")
        path = iterative_depth(graph, origin, destiny)
        if path:
            return path
        else:
            print("No path found.")
            return None
```

Figure 6. Código de Búsqueda Iterada

Primero se importa a “limited_depth” que contiene a “limited_depth_search”.

Creamos 2 funciones:

- A) “iterative_depth” itera a través de los valores de lo largo del grafo y para cada una de las iteraciones llama a “limited_depth_search” con el límite de profundidad actual. Si se regresa el camino, va a indicar el camino y su profundidad.
- B) “run” es igual al caso anterior.

dijkstra.py

```
def dijkstra(graph, origin, destiny):
    distances = {node: float('inf') for node in graph.content}
    distances[origin] = 0
    visited = []
    shortest_paths = {}

    while len(visited) != len(graph.content):
        current_node = None
        current_distance = float('inf')
        for node in graph.content:
            if distances[node] < current_distance and node not in visited:
                current_node = node
                current_distance = distances[node]
        if current_node is None:
            break
        if current_node not in visited:
            visited.append(current_node)
        for neighbor, weight in graph.content[current_node]:
            distance = int(current_distance) + int(weight)
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                shortest_paths[neighbor] = current_node

    if distances[destiny] == float('inf'):
        return None, None

    path = []
    node = destiny
    while node != origin:
        path.append(node)
        node = shortest_paths[node]
    path.append(origin)
    path.reverse()

    if path:
        return path, distances[destiny]
    else:
        print("No path found.")
        return None, None

def run(graph, origin, destiny, runner):
    if runner:
        print("\n\nDIJKSTRA:\n-----")
        path, weight = dijkstra(graph, origin, destiny)
        if path:
            return path, weight
        else:
            print("No path found.")
            return None, None
```

Este código tiene la función “run” que funciona como en los anteriores códigos.

Adicionalmente, tiene la función “dijkstra”, que implementa el algoritmo primero inicializando “distances”, que contiene la distancia más corta del nodo inicial a los demás del grafo. También se inicializa la lista vacía “visited” para almacenar nuestros nodos ya visitados y, además, “shortest_paths” para poder almacenar el nodo previo de cada nodo en el camino más corto. Después se ejecuta un “while” hasta que se hayan visitado todos los nodos del grafo. Dentro de este, está el nodo actual que tiene la distancia más corta desde el nodo de origen y se van actualizando las distancias de los nodos vecinos y luego se devuelve la ruta más corta encontrada.

Figure 7. Algoritmo de Dijkstra

bidirectional.py

```
import depth as b

def bidirectional(graph, graph2, origin, destiny):
    departure = True
    arrival = True
    if not b.depth_first_search(graph, origin, destiny):
        departure = False
    if not b.depth_first_search(graph2, destiny, origin):
        arrival = False

    if departure and arrival:
        queue1 = [[origin]]
        queue2 = [[destiny]]
        while queue1 and queue2:
            path1 = queue1.pop(0)
            node1 = path1[-1]
            path2 = queue2.pop(0)
            node2 = path2[-1]
            if node1 == destiny:
                return path1
            if node2 == origin:
                path2.reverse()
                return path2
            for neighbor in graph.content[node1]:
                new_path = list(path1)
                new_path.append(neighbor[0])
                queue1.append(new_path)
            for neighbor in graph2.content[node2]:
                new_path = list(path2)
                new_path.append(neighbor[0])
                queue2.append(new_path)
        else:
            return None

def run(graph, graph2, origin, destiny, runner):
    if runner:
        print("\n\nBidirectional Search:\n-----")
        path = bidirectional(graph, graph2, origin, destiny)
        if path:
            return path
        else:
            print("No path found.")
            return None
```

Figure 8. Búsqueda Bidireccional

A) “breadth” se importa el breadth en la primera línea del código.

B) “bidirectional” implementa la búsqueda en el grafo. Toma como entrada a graph y graph2 (uno el inverso del otro), así como el nodo de origen y el nodo de destino. Devuelve el camino más corto del origen al destino si existe uno, o None si no existe.

C) “run” sigue funcionando de la misma manera que en los otros códigos.

main.py

En cuanto a objetos, se crean varios de la clase “Graph”, llamando a un constructor y guardar una referencia. Se crean los objetos “graph” y “graph2”.

Ya para las funciones, se definen en el módulo “functions” que se importa al principio del código. Están “check_origin ()” y “check_destiny()”. También se usa “timeit.default_timer ()” para medir el tiempo de ejecución de las búsquedas de rutas, y varias de “breadth”, “limited_depth”, “depth”, “dijkstra”, “iterative_depth” y “bidirectional” que se importan al principio del código. La función principal del programa es “main ()”, que realiza diversas tareas, como cargar un archivo de texto que describe un gráfico, buscar caminos a través de ese gráfico y mostrar información sobre el rendimiento de diferentes algoritmos de búsqueda.

```
1  import functions as f
2  import timeit as t
3  import string
4
5  from Graph import *
6  import breadth
7  import limited_depth
8  import depth
9  import dijkstra
10 import iterative_depth
11 import bidirectional
12
13
14 def main():
15     weighted = True
16     weights = []
17     times = []
18
19     filename = "weighted_graph.txt"
20     # filename = "non_weighted_graph.txt"
21
22     graph = Graph()
23     graph2 = Graph()
24     with open(filename) as file:
25         lines = file.readlines()
26
27     for i in range(1, len(lines)):
28         origin, destiny, weight = lines[i].split()
29         weights.append(weight)
30         graph.new_edge(origin, destiny, weight)
31         graph2.new_edge(destiny, origin, weight)
32
33     for i in range(len(weights)):
34         if weights[i] != weights[0]:
35             print("The graph is weighted\n")
36             break
```

Figure 9. Parte 1 main.py

```

37         if i == len(weights) - 1:
38             print("The graph is not weighted\n")
39             weighted = False
40
41     graph.view_all()
42
43     print("Origin: ", end="")
44     origin = input()
45     origin2 = origin.translate({ord(c): None for c in string.whitespace})
46     runner1 = f.check_origin(graph, origin2)
47     print("Destiny: ", end="")
48     destiny = input()
49     destiny2 = destiny.translate({ord(c): None for c in string.whitespace})
50     runner2 = f.check_destiny(graph, destiny2)
51
52     runner = runner1 and runner2
53
54     breadth_reached = False
55     limited_reached = False
56     depth_reached = False
57     iterative_reached = False
58     dijkstra_limited_reached = False
59     bidirectional_reached = False
60
61     origin = string.capwords(origin).translate({ord(c): None for c in string.whitespace})
62     destiny = string.capwords(destiny).translate({ord(c): None for c in string.whitespace})
63
64     stime = t.default_timer()
65     path = breadth.run(graph, origin, destiny, runner)
66     ftime = t.default_timer()
67     times.append(ftime - stime)
68     f.show_path(path)
69     if path:
70         breadth_reached = True
71

```

Figure 10. Parte2 main.py

```

72     if weighted and runner:
73         limit = int(input("\nEnter the limit: "))
74         stime = t.default_timer()
75         path = limited_depth.run(graph, origin, destiny, runner, limit)
76         ftime = t.default_timer()
77         times.append(ftime - stime)
78         f.show_path(path)
79         if path:
80             limited_reached = True
81
82         stime = t.default_timer()
83         path = depth.run(graph, origin, destiny, runner)
84         ftime = t.default_timer()
85         times.append(ftime - stime)
86         f.show_path(path)
87         if path:
88             depth_reached = True
89
90         stime = t.default_timer()
91         path = iterative_depth.run(graph, origin, destiny, runner)
92         ftime = t.default_timer()
93         times.append(ftime - stime)
94         f.show_path(path)
95         if path:
96             iterative_reached = True
97
98         stime = t.default_timer()
99         path, weight = dijkstra.run(graph, origin, destiny, runner)
100        ftime = t.default_timer()
101        times.append(ftime - stime)
102        f.show_path_dijkstra(path, weight)
103        if path:
104            dijkstra_limited_reached = True
105

```

Figure 11. Parte3 main.py

```

106         stime = t.default_timer()
107         path = bidirectional.run(graph, graph2, origin, destiny, runner)
108         ftime = t.default_timer()
109         times.append(ftime - stime)
110         f.show_path(path)
111         if path:
112             bidirectional_reached = True
113
114     if runner:
115         print("\n\nEXECUTION TIMES:\n-----")
116         if times:
117             if breadth_reached:
118                 print("Breadth First Search: ", times[0])
119             if weighted and times:
120                 if limited_reached:
121                     print("Limited Depth First Search: ", times[1])
122                 else:
123                     print("Limited Depth First Search: No path found.")
124                     times[1] = 999999999999
125
126             if depth_reached:
127                 print("Depth First Search: ", times[2])
128             else:
129                 print("Depth First Search: No path found.")
130                 times[2] = 999999999999
131
132             if iterative_reached:
133                 print("Iterative Depth Search: ", times[3])
134             else:
135                 print("Iterative Depth Search: No path found.")
136                 times[3] = 999999999999
137             if dijkstra_limited_reached:
138                 print("Dijkstra Algorithm: ", times[4])
139             else:
140                 print("Dijkstra Algorithm: No path found.")
141                 times[4] = 999999999999
142

```

Figure 12. Parte4 main.py

```

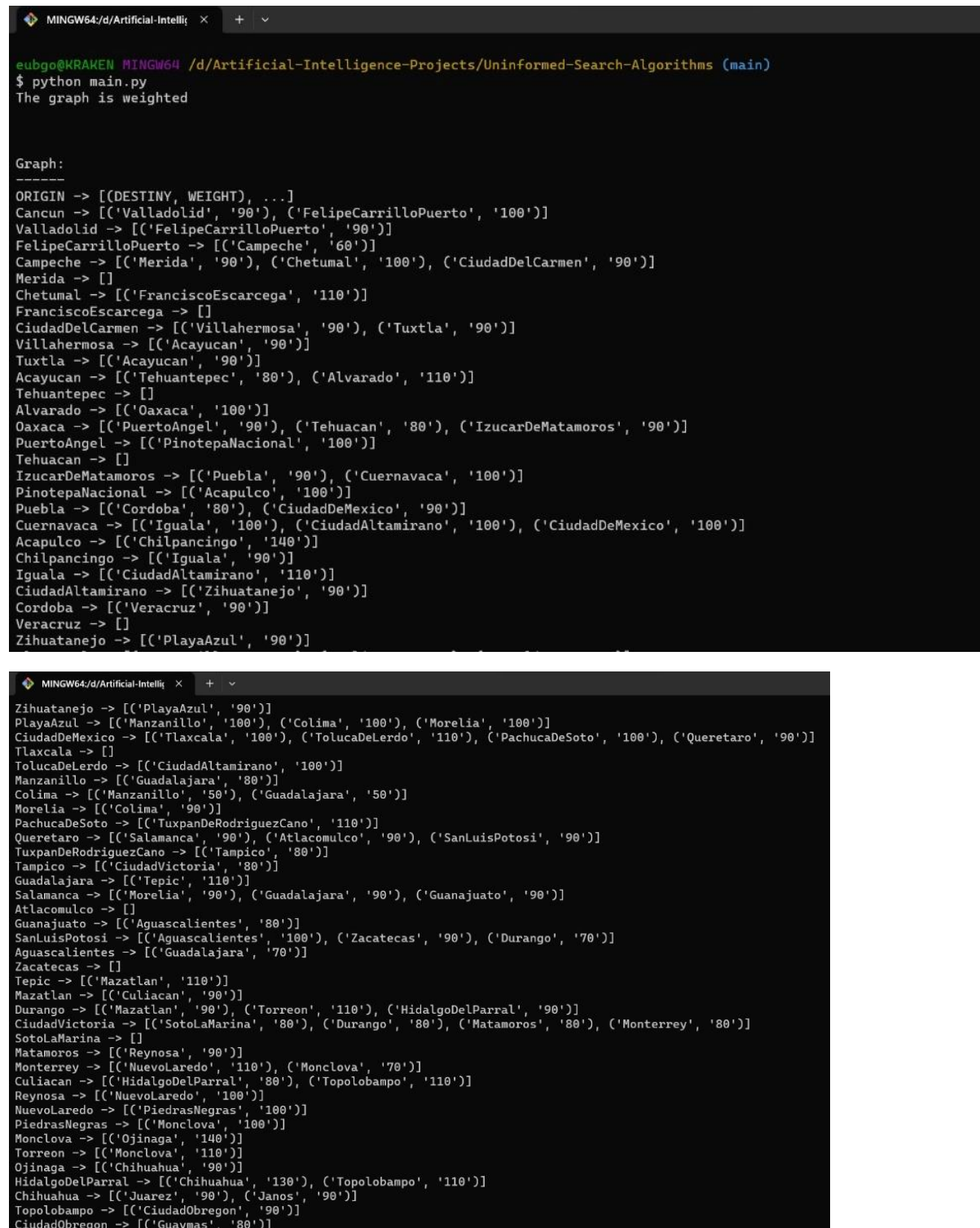
143         if bidirectional_reached:
144             print("Bidirectional Search: ", times[5])
145         else:
146             print("Bidirectional Search: No path found.")
147             times[5] = 999999999999
148
149         if times[1] == 999999999999 and times[2] == 999999999999 and times[3] == 999999999999 and \
150             times[4] == 999999999999 and times[5] == 999999999999:
151             exit()
152         else:
153             best = min(times)
154             print("\nBest time: ", best)
155             if best == times[0]:
156                 print("Breadth First Search is the best option")
157             elif best == times[1] and limited_reached:
158                 print("Limited Depth First Search is the best option")
159             elif best == times[2]:
160                 print("Depth First Search is the best option")
161             elif best == times[3]:
162                 print("Iterative Depth Search is the best option")
163             elif best == times[4]:
164                 print("Dijkstra is the best option")
165             elif best == times[5]:
166                 print("Bidirectional Search is the best option")
167
168
169     if __name__ == "__main__":
170         main()

```

Figure 13. Parte5 main.py

Ejemplos de ejecución

Ejemplo 1)



```
MINGW64/d/Artificial-Intelli... X + v

eubgo@KRAKEN MINGW64 /d/Artificial-Intelligence-Projects/Uninformed-Search-Algorithms (main)
$ python main.py
The graph is weighted

Graph:
-----
ORIGIN -> [(DESTINY, WEIGHT), ...]
Cancun -> [(Valladolid, '90'), (FelipeCarrilloPuerto, '100')]
Valladolid -> [(FelipeCarrilloPuerto, '90')]
FelipeCarrilloPuerto -> [(Campeche, '60')]
Campeche -> [(Merida, '90'), (Chetumal, '100'), (CiudadDelCarmen, '90')]
Merida -> []
Chetumal -> [(FranciscoEscarcega, '110')]
FranciscoEscarcega -> []
CiudadDelCarmen -> [(Villahermosa, '90'), (Tuxtla, '90')]
Villahermosa -> [(Acayucan, '90')]
Tuxtla -> [(Acayucan, '90')]
Acayucan -> [(Tehuantepec, '80'), (Alvarado, '110')]
Tehuantepec -> []
Alvarado -> [(Oaxaca, '100')]
Oaxaca -> [(PuertoAngel, '90'), (Tehuacan, '80'), (IzucarDeMatamoros, '90')]
PuertoAngel -> [(PinotepaNacional, '100')]
Tehuacan -> []
IzucarDeMatamoros -> [(Puebla, '90'), (Cuernavaca, '100')]
PinotepaNacional -> [(Acapulco, '100')]
Puebla -> [(Cordoba, '80'), (CiudadDeMexico, '90')]
Cuernavaca -> [(Iguala, '100'), (CiudadAltamirano, '100'), (CiudadDeMexico, '100')]
Acapulco -> [(Chilpancingo, '140')]
Chilpancingo -> [(Iguala, '90')]
Iguala -> [(CiudadAltamirano, '110')]
CiudadAltamirano -> [(Zihuatanejo, '90')]
Cordoba -> [(Veracruz, '90')]
Veracruz -> []
Zihuatanejo -> [(PlayaAzul, '90')]

Zihuatanejo -> [(PlayaAzul, '90')]
PlayaAzul -> [(Manzanillo, '100'), (Colima, '100'), (Morelia, '100')]
CiudadDeMexico -> [(Tlaxcala, '100'), (TolucaDeLerdo, '110'), (PachucaDeSoto, '100'), (Queretaro, '90')]
Tlaxcala -> []
TolucaDeLerdo -> [(CiudadAltamirano, '100')]
Manzanillo -> [(Guadalajara, '80')]
Colima -> [(Manzanillo, '50'), (Guadalajara, '50')]
Morelia -> [(Colima, '90')]
PachucaDeSoto -> [(TuxpanDeRodriguezCano, '110')]
Queretaro -> [(Salamanca, '90'), (Atzacmulco, '90'), (SanLuisPotosi, '90')]
TuxpanDeRodriguezCano -> [(Tampico, '80')]
Tampico -> [(CiudadVictoria, '80')]
Guadalajara -> [(Tepic, '110')]
Salamanca -> [(Morelia, '90'), (Guadalajara, '90'), (Guanajuato, '90')]
Atzacmulco -> []
Guanajuato -> [(Aguascalientes, '80')]
SanLuisPotosi -> [(Aguascalientes, '100'), (Zacatecas, '90'), (Durango, '70')]
Aguascalientes -> [(Guadalajara, '70')]
Zacatecas -> []
Tepic -> [(Mazatlan, '110')]
Mazatlan -> [(Culiacan, '90')]
Durango -> [(Mazatlan, '90'), (Torreon, '110'), (HidalgoDelParral, '90')]
CiudadVictoria -> [(SotoLaMarina, '80'), (Durango, '80'), (Matamoros, '80'), (Monterrey, '80')]
SotoLaMarina -> []
Matamoros -> [(Reynosa, '90')]
Monterrey -> [(NuevoLaredo, '110'), (Monclova, '70')]
Culiacan -> [(HidalgoDelParral, '80'), (Topolobampo, '110')]
Reynosa -> [(NuevoLaredo, '100')]
NuevoLaredo -> [(PiedrasNegras, '100')]
PiedrasNegras -> [(Monclova, '100')]
Monclova -> [(Ojinaga, '140')]
Torreon -> [(Monclova, '110')]
Ojinaga -> [(Chihuahua, '90')]
HidalgoDelParral -> [(Chihuahua, '130'), (Topolobampo, '110')]
Chihuahua -> [(Juarez, '90'), (Janos, '90')]
Topolobampo -> [(CiudadObregon, '90')]
CiudadObregon -> [(Guaymas, '80')]
```

Figure 14 & 15. Ejecución 1

```
MINGW64/d/Artificial-Intelli... x + v
CiudadOregon -> [['Guaymas', '80']]
Guaymas -> [['Hermosillo', '80']]
Hermosillo -> [['SantaAna', '60']]
Juarez -> []
Janos -> [['AguaPrieta', '110']]
AguaPrieta -> [['SantaAna', '60']]
SantaAna -> [['Mexicali', '150']]
Mexicali -> [['Tijuana', '110'], ['SanFelipe', '70']]
Tijuana -> [['Ensenada', '50']]
SanFelipe -> [['Ensenada', '50']]
Ensenada -> [['SanQuintin', '60']]
SanQuintin -> [['SantaRosalia', '60']]
SantaRosalia -> [['SantoDomingo', '60']]
SantoDomingo -> [['LaPaz', '70']]
LaPaz -> [['CaboSanLucas', '70']]
CaboSanLucas -> []

Origin: cancan
Destiny: campeche

Breadth First Search:
-----
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche

Enter the limit: 1

Limited Depth Search:
-----
No path found.
```

```
MINGW64/d/Artificial-Intelli... x + v

Depth First Search:
-----
PATH FOUND:
Cancun -> Valladolid -> FelipeCarrilloPuerto -> Campeche

Iterative Depth Search:
-----
Path found at DEPTH: 2
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche

DIJKSTRA:
-----
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche
TOTAL WEIGHT: 160

Bidirectional Search:
-----
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche

EXECUTION TIMES:
-----
Breadth First Search: 0.0005278000026009977
Limited Depth First Search: No path found.
Depth First Search: 0.00023510001483373344
Iterative Depth Search: 0.0002468999882694334
Dijkstra Algorithm: 0.001876599999377504
Bidirectional Search: 0.00023900001542642713

Best time: 0.00023510001483373344
Depth First Search is the best option
```

Figure 15 & 16. Ejecución 1

Ejemplo 2)

```
MINGW64/d/Artificial-Intellig X + v

Origin: cancion
Destiny: zacatecas

Breadth First Search:
=====
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queretaro -> SanLuisPotosi -> Zacatecas
Enter the limit: 1

Limited Depth Search:
=====
No path found.

Depth First Search:
=====
PATH FOUND:
Cancun -> Valladolid -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queretaro -> SanLuisPotosi -> Zacatecas

Iterative Depth Search:
=====
Path found at DEPTH: 13
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queretaro -> SanLuisPotosi -> Zacatecas

DIJKSTRA:
=====
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queretaro -> SanLuisPotosi -> Zacatecas
TOTAL WEIGHT: 1188
```

```
Bidirectional Search:
=====
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queretaro -> SanLuisPotosi -> Zacatecas

EXECUTION TIMES:
=====
Breadth First Search: 0.0005752999859396368
Limited Depth First Search: No path found.
Depth First Search: 0.0010876999876927584
Iterative Depth Search: 0.0005896999791730195
Dijkstra Algorithm: 0.0022761999862268567
Bidirectional Search: 0.0016744000022299588

Best time: 0.0005752999859396368
Breadth First Search is the best option
```

Figure 17 & 18. Ejecución 2

Ejemplo 3)

```

MINGW64/d/Artificial-Intelli
+ +
Origin: cancan
Destiny: cabo san LUCAS

Breadth First Search:
-----
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queret
aro -> SanLuisPotosi -> Durango -> HidalgoDelParral -> Chihuahua -> Janos -> AguaPrieta -> SantaAnna -> Mexicali -> Tijuana -> Ensenada -> SanQuintin -> SantaRosalia -> S
antoDomingo -> LaPaz -> CaboSanLucas

Enter the limit: 30

Limited Depth Search:
-----
PATH FOUND:
Cancun -> Valladolid -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMe
xico -> PachucaDeSoto -> TuxpanDeRodriguezCano -> Tampico -> CiudadVictoria -> Durango -> HidalgoDelParral -> Chihuahua -> Janos -> AguaPrieta -> SantaAnna -> Mexicali ->
Tijuana -> Ensenada -> SanQuintin -> SantaRosalia -> SantoDomingo -> LaPaz -> CaboSanLucas

Depth First Search:
-----
PATH FOUND:
Cancun -> Valladolid -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> PuertoAngel -> PinotepaNacional -> Acapul
co -> Chilpancingo -> Iguala -> CiudadAltamirano -> Zihuatanejo -> PlayaAzul -> Manzanillo -> Guadalajara -> Tepic -> Mazatlan -> Culiacan -> HidalgoDelParral -> Chihuahua
a -> Janos -> AguaPrieta -> SantaAnna -> Mexicali -> Tijuana -> Ensenada -> SanQuintin -> SantaRosalia -> SantoDomingo -> LaPaz -> CaboSanLucas

Iterative Depth Search:
-----
Path found at DEPTH: 26
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queret
aro -> SanLuisPotosi -> Durango -> HidalgoDelParral -> Chihuahua -> Janos -> AguaPrieta -> SantaAnna -> Mexicali -> Tijuana -> Ensenada -> SanQuintin -> SantaRosalia -> S
antoDomingo -> LaPaz -> CaboSanLucas

```

```
MINGW64/d/Artificial-Intelli × + ✓
aro -> SanLuisPotosi -> Durango -> HidalgoDelParral -> Chihuahua -> Janos -> AguaPrieta -> SantaAna -> Mexicali -> Tijuana -> Ensenada -> SanQuintin -> SantaRosalia -> SantoDomingo -> LaPaz -> CaboSanLucas

DIJKSTRA:
-----
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queret
aro -> SanLuisPotosi -> Durango -> HidalgoDelParral -> Chihuahua -> Janos -> AguaPrieta -> SantaAna -> Mexicali -> SanFelipe -> Ensenada -> SanQuintin -> SantaRosalia -> SantoDomingo -> LaPaz -> CaboSanLucas
TOTAL WEIGHT: 2230

Bidirectional Search:
-----
PATH FOUND:
Cancun -> FelipeCarrilloPuerto -> Campeche -> CiudadDelCarmen -> Villahermosa -> Acayucan -> Alvarado -> Oaxaca -> IzucarDeMatamoros -> Puebla -> CiudadDeMexico -> Queret
aro -> SanLuisPotosi -> Durango -> HidalgoDelParral -> Chihuahua -> Janos -> AguaPrieta -> SantaAna -> Mexicali -> Tijuana -> Ensenada -> SanQuintin -> SantaRosalia -> SantoDomingo -> LaPaz -> CaboSanLucas

EXECUTION TIMES:
-----
Breadth First Search: 0.004742799996165559
Limited Depth First Search: 0.00036800000016583633
Depth First Search: 0.00019359998987056315
Iterative Depth Search: 0.007361400028457865
Dijkstra Algorithm: 0.001874700014013797
Bidirectional Search: 0.0024090000079013407

Best time: 0.00019359998987056315
Depth First Search is the best option
```

Figure 19 & 20. Ejecución 3

Restricciones

La meta para el código de este parcial se alcanzó con éxito y se cumplieron todas las expectativas respecto al mismo.

Considerando los objetivos propuestos, no se le encontraron deficiencias en este código, pero una mejora que facilitaría el análisis de los resultados puede ser el mejorar el diseño de impresión del grafo para que sea mucho más fácil de comprender.