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**

**Introducción2**

**Grafo…………………………………………………………………………………………………………………………………………………3**

Imagen3

Descripción3

**Instrucciones4**

Leyendo el código4

**Funciones/Clases/Objetos…………………………………………………………………………………………………………………5**

Orden de ejecución y Descripción5

**Ejemplos…………………………………………………………………………………………………………………………………………11**

**Restricciones……………………………………………………………………………………………………………………………………15**

**Tabla de Imágenes**

[Figure 1. Grafo Dirigido 3](#_Toc127917468)

[Figure 2. Código del Grafo 5](#_Toc127917469)

[Figure 3. Código de Búsqueda a lo Ancho. 5](#_Toc127917470)

[Figure 4. Código Búsqueda Limitada 6](#_Toc127917471)

[Figure 5. Código de Búsqueda por Profundidad 6](#_Toc127917472)

[Figure 6. Código de Búsqueda Iterada 7](#_Toc127917473)

[Figure 7. Algoritmo de Dijkstra 7](#_Toc127917474)

[Figure 8. Búsqueda Bidireccional 8](#_Toc127917475)

[Figure 9 Parte 1 main.py 9](#_Toc127917476)

[Figure 10 Parte 2 main.py 10](#_Toc127917477)

[Figure 11 Parte 3 main.py 1](#_Toc127917478)1

[Figure 12 Parte 4 main.py 12](#_Toc127917481)

[Figure 13 Parte 5 main.py 13](#_Toc127917482)

[Figure 14 & 15 Ejecución 1 14](#_Toc127917483)

[Figure 15 & 16 Ejecución 1 15](#_Toc127917483)

[Figure 17 & 18 Ejecución 2 16](#_Toc127917483)

[Figure 19 & 20 Ejecución 3 17](#_Toc127917483)

**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:

1. 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.
2. Búsqueda por Profundidad
3. Búsqueda en Profundidad Limitada
4. Búsqueda en Profundidad Iterada
5. Algoritmo de Dijkstra
6. 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**

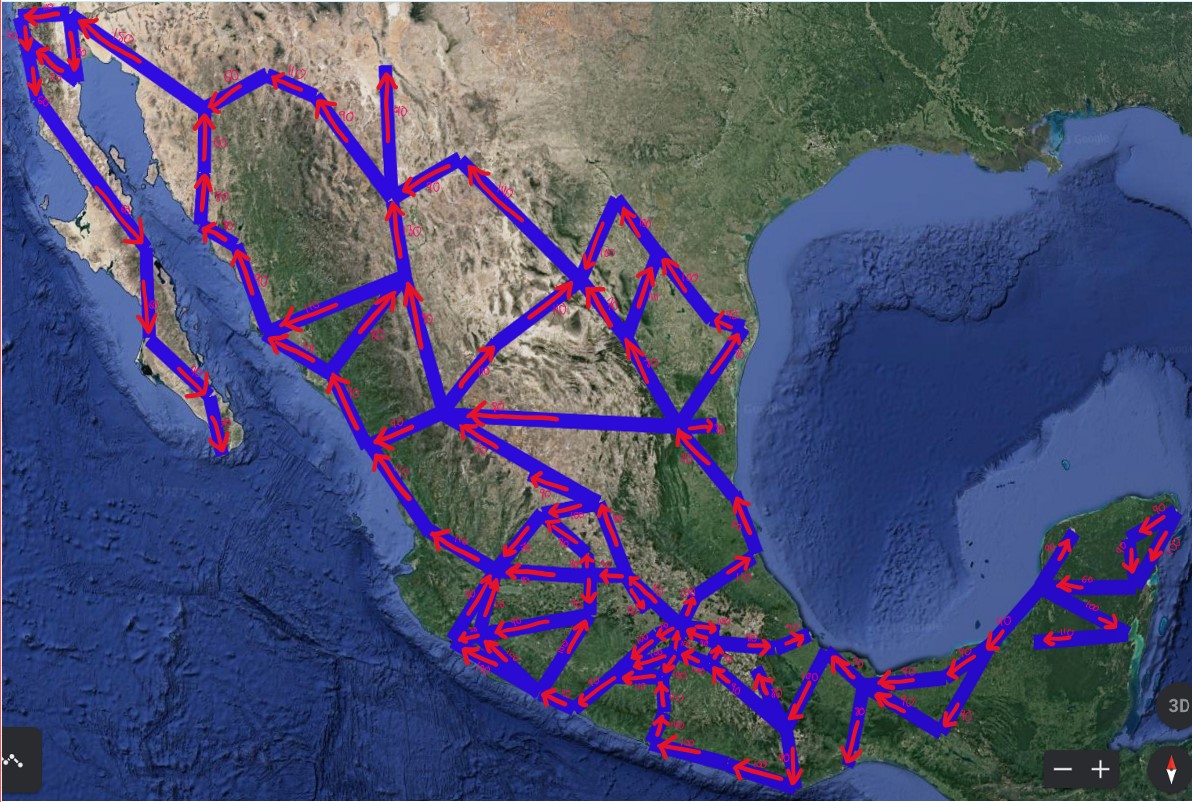
****

Figure . 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**

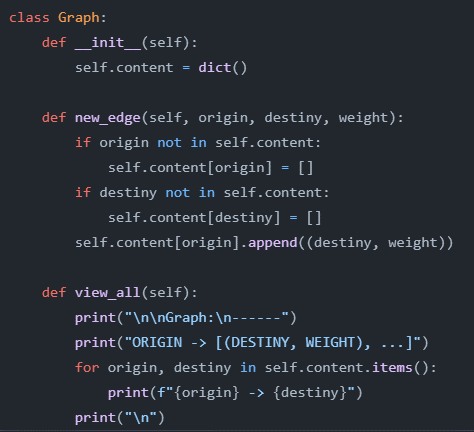
Graph.py

Figure . 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:

1. “ \_\_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.
2. “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.

1. “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

Figure . Código de Búsqueda a lo Ancho.

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

1. “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.
2. “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

Tiene 2 funciones:

1. “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.
2. “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 . Código Búsqueda Limitada

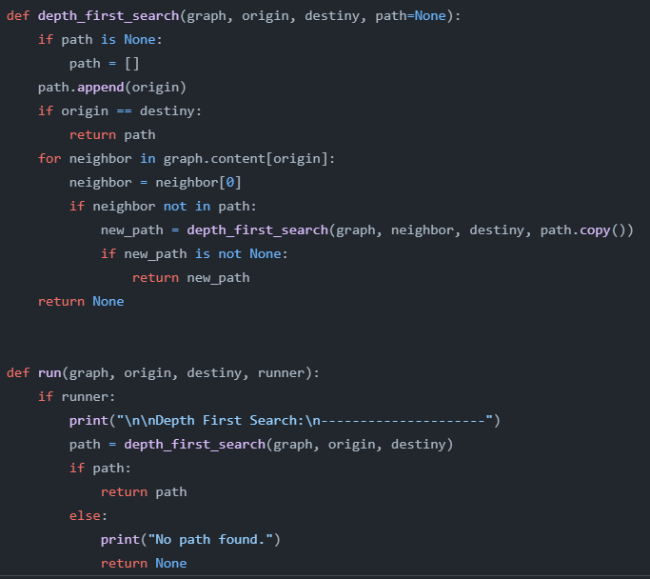
depth.py

Figure . Código de Búsqueda por Profundidad

Tiene 2 funciones:

1. “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.
2. “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.

iterative\_depth.py

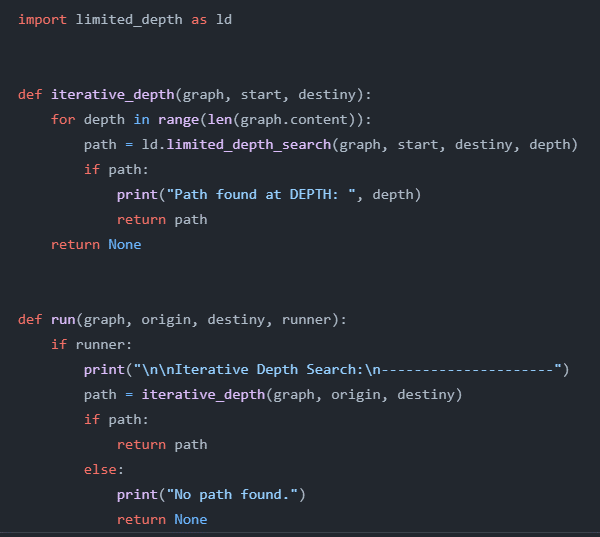
Primero se importa a “limited\_depth” que contiene a “limited\_depth\_search”.

Figure . Código de Búsqueda Iterada

Creamos 2 funciones:

1. “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.
2. “run” es igual al caso anterior.

dijkstra.py

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 . Algoritmo de Dijkstra

bidirectional.py

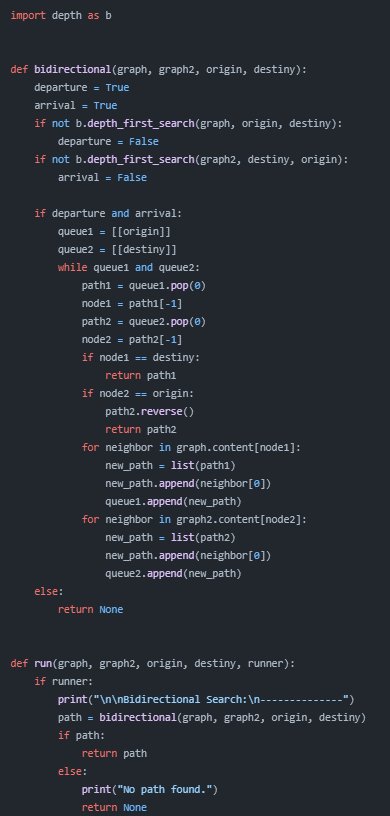
1. “breadth” se importa el breadth en la primera línea del código.

Figure . Búsqueda Bidireccional

1. “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.
2. “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.

Text

Description automatically generated

Figure 9. Parte 1 main.py

**Text

Description automatically generated**

Figure 10. Parte2 main.py

**Text

Description automatically generated**

Figure 11. Parte3 main.py

**Text

Description automatically generated**

Figure 12. Parte4 main.py

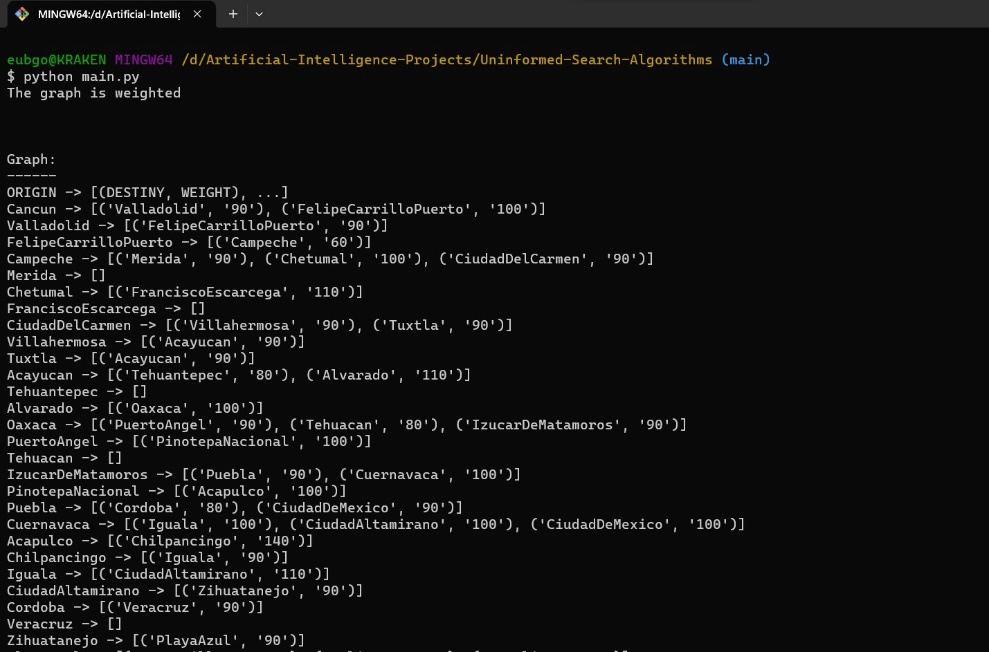
**Text

Description automatically generated**

Figure 13. Parte5 main.py

**Ejemplos de ejecución**

**Ejemplo 1)**

****

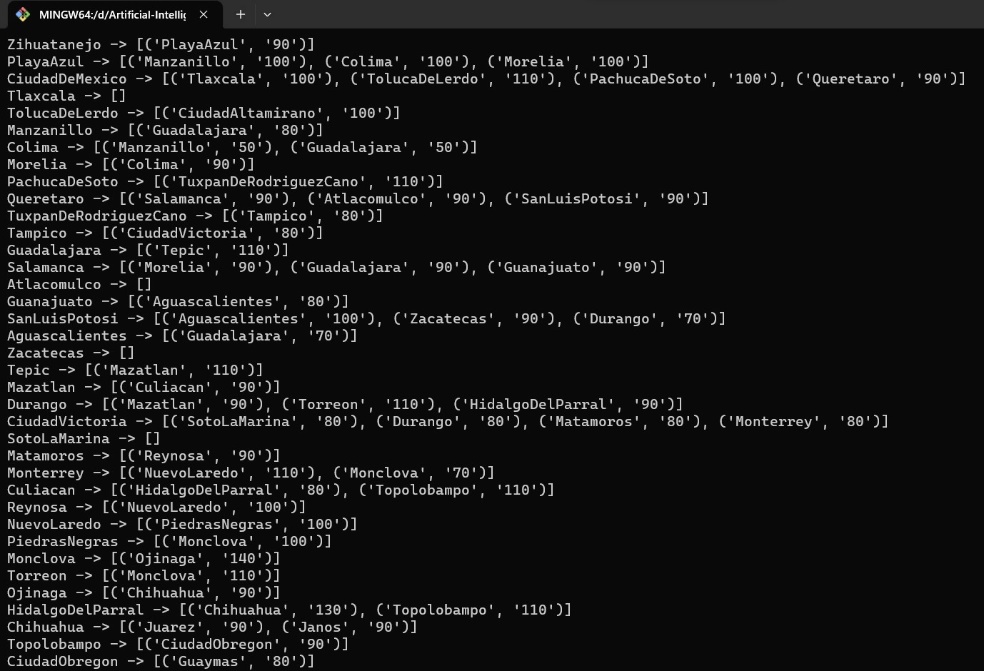
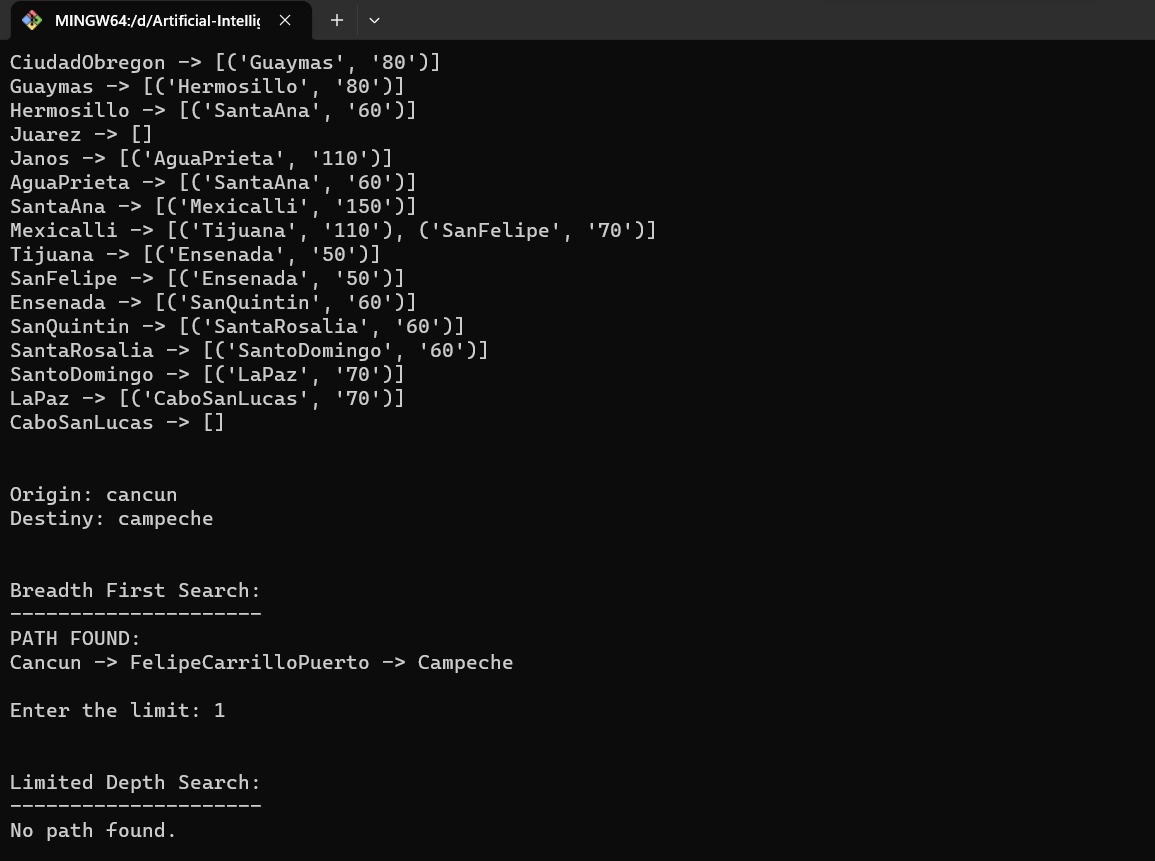
****

Figure 14 & 15. Ejecución 1

****

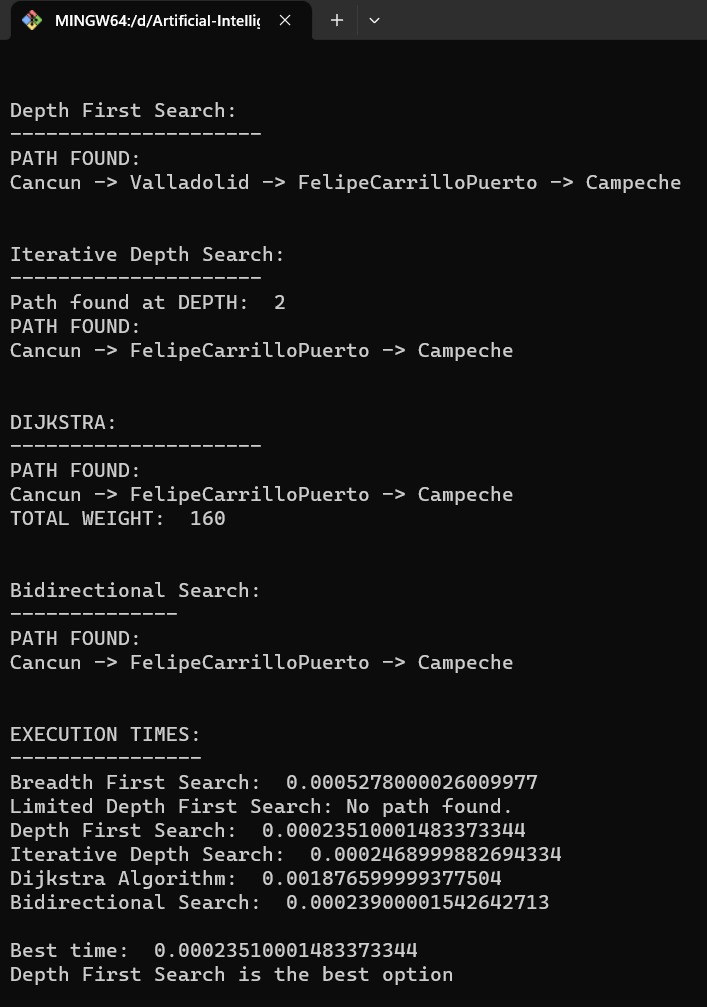
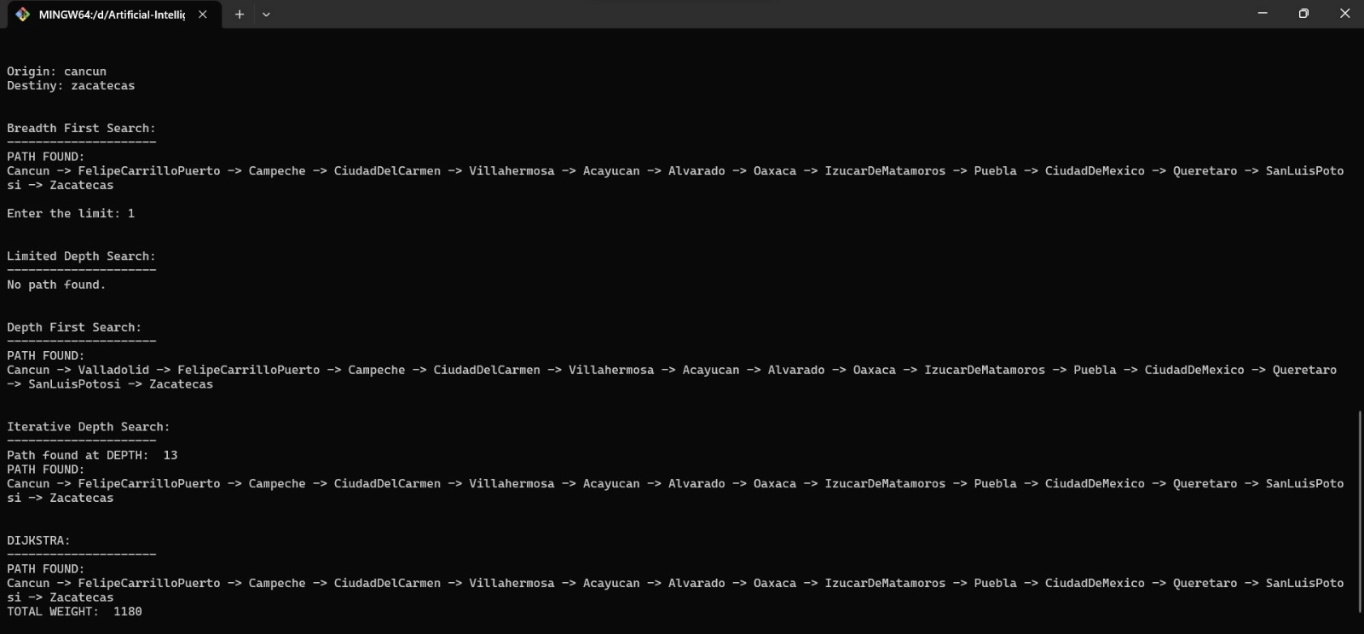
****

Figure 15 & 16. Ejecución 1

**Ejemplo 2)**

****

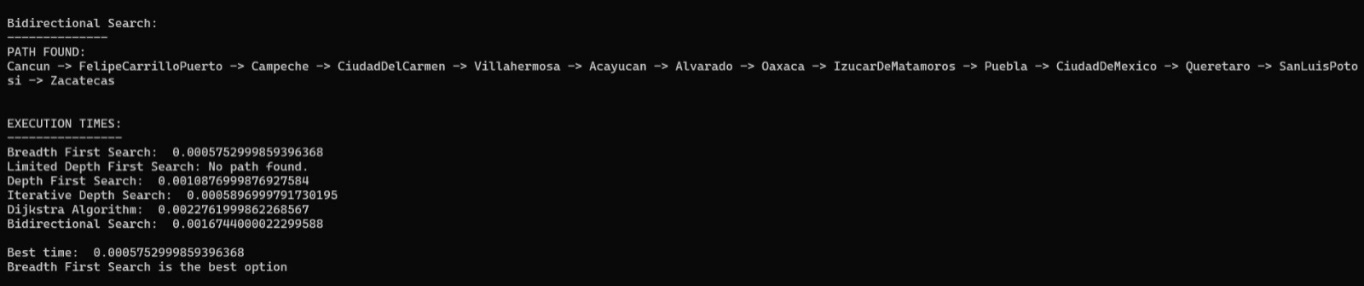
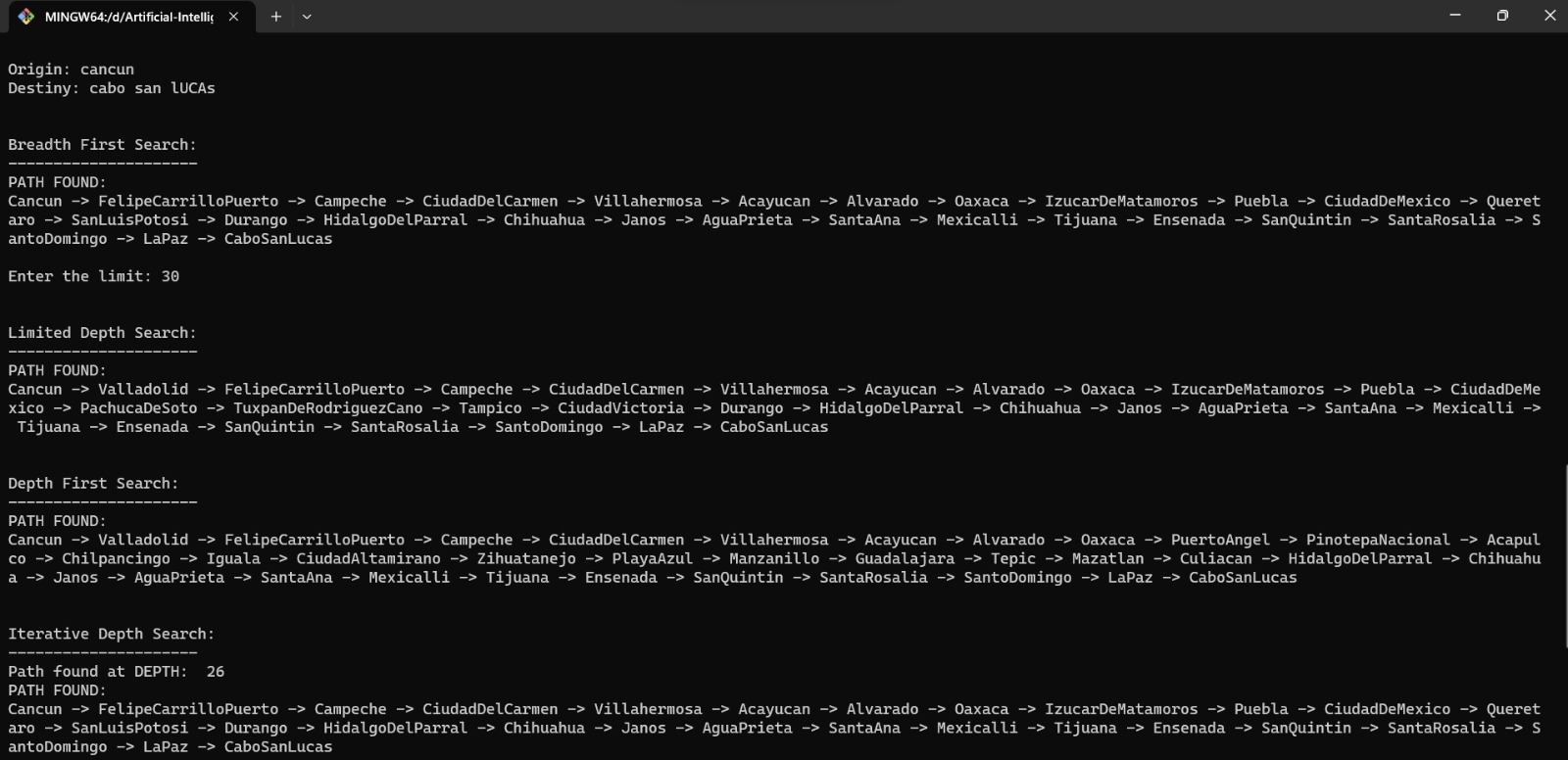
****

Figure 17 & 18. Ejecución 2

**Ejemplo 3)**

****

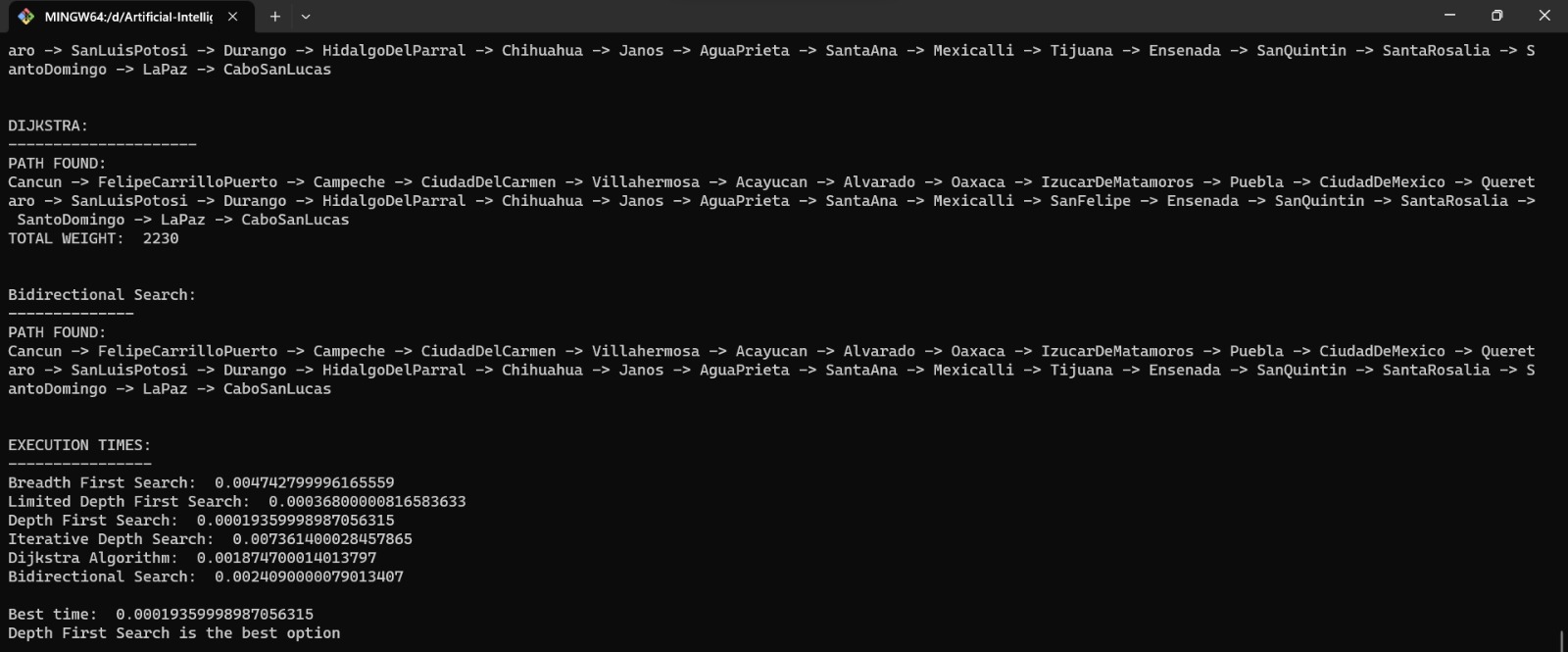
****

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.