Universidad Alfonso X El Sabio

Redes de transporte: Optimización de la red de transporte entre ciudades mediante grafos.

Estructura de Datos y Algoritmos II

Autores:
Jacobo Calviño
Lucia Cantos
María González

Diciembre 2023

Contents

Resumen

Este trabajo aborda la implementación de grafos y algoritmos fundamentales en *Python*, explorando su relevancia en la informática y su aplicación práctica. La aplicación practica seleccionada ha sido la creación de un sistema para optimizar la red de transporte entre ciudades. Para ello, los nodos representan las ciudades y las aristas las conexiones dónde cada peso equivale a la distancia entre ellas. Se implementa funciones clave como un método para encontrar la ruta más corta entre dos ciudades, mediante el algoritmo de *Dijkstra*. Además, integra un árbol binario de búsqueda, *BST*, para mantener un registro ordenado de las distancias entre ciudades; un método para seleccionar un conjunto mínimo de aristas que mantengan el grafo conectado, garantizando la distancia total mínima.

Corolario a este trabajo, se ha consolidado una comprensión sólida de los grafos, algoritmos y estructuras de datos clave, enfocándose en la aplicación práctica de estos conceptos para su representación y manipulación. Además, el uso de bibliotecas gráficas ha permitido un análisis visual que ofrece una perspectiva clara y concisa de las relaciones y estructura de los grafos implementados.

Creación de la clase Grafo.

0.0.1 Definición.

Una clase *Grafo* es una estructura de datos que modela un conjunto de nodos, vértices, y las relaciones entre ellos, aristas. Un grafo puede ser dirigido, donde las aristas tienen una dirección específica desde un nodo de origen a un nodo de destino, o no dirigido, donde las aristas no tienen dirección. Además, las aristas pueden o no tener pesos que representan alguna medida asociada, como la distancia entre nodos.

En el contexto de la programación orientada a objetos, una clase 'Grafo' podría tener métodos para agregar nodos y aristas, buscar caminos entre nodos, entre otros.

La representación de un grafo en un programa puede realizarse de diversas maneras, como mediante listas de adyacencia, matrices de adyacencia o estructuras más complejas dependiendo de los requisitos y eficiencia deseada.

0.0.2 Implementación en código.

El código define dos clases, 'Ciudad' y 'Grafo', que representan un grafo no dirigido donde las ciudades son nodos y las conexiones entre ciudades son aristas.

Atributos y métodos de la Clase Ciudad.

Utilizamos un método 'init()' que se llama automáticamente cuando se crea un nuevo objeto (Ciudad). Inicializa una ciudad con un nombre y un diccionario de conexiones, el cual se utiliza para almacenar las conexiones de la ciudad con otras ciudades y los pesos asociados. Usamos el método 'agregar_conexion()', que permite establecer conexiones entre la ciudad actual y

la ciudad proporcionada con el peso asociado. Esta conexión se almacena en el diccionario (conexiones) de la ciudad actual.

Algorithm 1: Método agregar_conexion(ciudad, peso)

```
Data: ciudad, peso
Result: Conexión agregada a la ciudad
if ciudad no está en conexiones then
    Agregar la ciudad y su peso a conexiones;
    return
end
else
    Actualizar el peso de la conexión existente con la ciudad;
    return
end
```

Atributos y métodos de la Clase Grafo.

Usamos el método 'agregar_ciudad()' que toma el objeto 'ciudad' y lo agrega a un diccionario del grafo. Este diccionario se utiliza para almacenar todas las ciudades del grafo.

Luego usamos un método para poder establecer una conexión entre dos ciudades con un peso establecido. El cual llama al método 'agregar_conexion' de las ciudades respectivas, y agrega la conexión a la lista de aristas. Con el método 'imprimir_grafo', imprimimos las conexiones de todas las ciudades en el grafo. Itera sobre el diccionario con todas las ciudades, y muestra el nombre de la ciudad y sus conexiones ordenadas alfabéticamente. Ahora con el grafo ya hecho, utilizamos la biblioteca NetworkX para crear un grafo a partir de las ciudades y sus conexiones en el grafo actual. Cada ciudad se agrega como un nodo, y cada conexión con su respectivo peso se agrega como una arista.

Una vez creado el grafo, utilizamos las librerías Matplotlib y NetworkXn, en el método 'dibujar_grafo', para poder dibujar el grafo. Este método calcula la posición de los nodos, dibuja el grafo con etiquetas y estilos específicos, y muestra el gráfico obtenido.

Algorithm 2: Clase Grafo

Atributos:

- ciudades (diccionario)
- aristas (lista)

Método agregar_ciudad(self, ciudad): begin
    Añadir la ciudad al diccionario de ciudades;

Método agregar_conexion(self, ciudad1, ciudad2, peso): begin
    Llamar a agregar_conexion de ciudad1 con ciudad2 y peso;
    Llamar a agregar_conexion de ciudad2 con ciudad1 y peso;
    Añadir la conexión a la lista de aristas;

Método imprimir_grafo(self): begin
    for *cada ciudad en ciudades* **do**
        Imprimir nombre de la ciudad y sus conexiones;

Método grafo_networkx(self): begin
    Crear un grafo vacío;
    for *cada ciudad en ciudades* **do**
        Añadir un nodo al grafo para la ciudad;
        for *cada conexión, peso en conexiones de la ciudad* **do**
            Añadir una arista al grafo con el peso;

Método dibujar_grafo(self): begin
    Crear un grafo usando grafo_networkx;
    Calcular la posición de los nodos en el grafo;
    Dibujar el grafo con etiquetas y estilos;
    Mostrar el gráfico;

0.1 Elección de la ruta más corta.

0.1.1 Definición.

El algoritmo de Dijkstra es un método utilizado para localizar el camino más corto desde un nodo de origen a todos los demás nodos en un grafo, con pesos positivos. Se basa en una estrategia voraz para determinar la ruta más corta.

El funcionamiento del algoritmo de Dijkstra es el siguiente.

• Dados un par de vértices no visitados, selecciona el vértice con la menor distancia desde la fuente y lo visita.

• A continuación, se actualiza la distancia de cada vecino. Lo mismo se hace para el vértice visitado, que tiene una distancia actual mayor que la suma y el peso del borde dado entre ellos.

• Los pasos 1 y 2 deben repetirse hasta que no queden vértices no visitados

El algoritmo de Dijkstra necesita la generación de un SPT (árbol de ruta más corta), tomando la fuente como la raíz del árbol.

0.1.2 Algoritmo de Dijkstra

En el inicio del método se inicializa un diccionario (distancias) con todas las ciudades del grafo, estableciendo la distancia inicial a cada ciudad como infinito, excepto la ciudad de inicio, cuya distancia se establece en 0. Esto crea un diccionario donde la clave es el nombre de la ciudad y el valor es la distancia actual conocida desde el inicio hasta es ciudad.

Después creamos una cola de prioridad utilizando una lista de tuplas, donde cada tupla contiene la distancia actual y el nombre de la ciudad. La cola de prioridad se implementa como una min-heap (montículo mínimo) utilizando la biblioteca headpq, importada al principio del código, de python. A parte la cola de prioridad garantiza que siempre se explore primero el nodo con la menor distancia acumulada, lo que hace que el algoritmo sea eficiente y encuentre las distancias mínimas de manera óptima en grafos ponderados.

Procedemos ejecutando un bucle mientras la cola de prioridad no esté vacía. En cada iteración se extrae el número mínimo de la cola de prioridad, en este caso la ciudad con la distancia mínima actual desde el inicio. Si la distancia actual es mayor que la distancia conocida para esa ciudad, se ignora y se continúa con la siguiente iteración del bucle.

Para cada ciudad vecina y su peso asociado en el grafo, se calcula la nueva distancia desde el inicio hasta esa ciudad pasando por la ciudad actual. Si

esta nueva distancia es menor que la distancia actual conocida para la ciudad vecina, se actualiza la distancia y se agrega la ciudad vecina a la cola de prioridad.

Una vez que se ha explorado todo el grafo y se han actualizado las distancias mínimas, el algoritmo devuelve la distancia mínima desde el inicio hasta el destino.

0.2 Binary Search Tree: *BST*.

0.2.1 Definición.

El Árbol Binario de Búsqueda, *BST*, es una estructura jerárquica de datos que consiste en nodos finitos, cada uno de los cuales puede estar vacío o contener un elemento. Este árbol organiza los nodos de manera ordenada y define una relación de orden entre ellos. Un nodo se compone de tres entidades: un valor con dos punteros, izquierda y derecha, considerados hijos; nodo padre, componente principal de cada subárbol o el más alto de un árbol; nodos hoja, son los elementos base de un árbol binario. La clave característica del *BST* es que, para cada nodo, los valores de los nodos en el subárbol izquierdo son menores que el valor del nodo, y los valores en el subárbol derecho son mayores.

Esta organización permite búsquedas eficientes, ya que el *BST* aprovecha su estructura ordenada para encontrar elementos rápidamente. Además, es útil para la inserción y eliminación de elementos, manteniendo siempre su propiedad de orden.

0.2.2 Clase BST.

Para ello creamos una nueva clase con los métodos y atributos correspondientes al nodo de una BST. Al separar la lógica del BST en una clase aparte, evita reescribir todo el código si se quiere aplicar en otro contexto. Además, implementa una clase Nodo en vez de una clase Árbol ya que estos arboles binarios como definimos son un puntero a un nodo padre que lo conecta con sus hijos.

Como atributos principales definimos a su hijo izquierdo y derecho, junto con los datos específicos del nodo: distancia, ciudad1 y ciudad2.

A continuación, necesitamos una manera para insertar nuevos datos en el árbol, al insertarlos se añaden como nodos hoja en el lugar correspondiente. Para ello definimos el método de inserción, que consta de distintos condicionales, que dependiendo de lo que se cumple ocurre una cosa u otra.

Posteriormente se muestra el pseudocódigo de este método.

Algorithm 3: Método insertar(distancia, ciudad1, ciudad2)

```
Data: distancia, ciudad1, ciudad2
Result: Nodo insertado en el árbol
if nodo actual está vacío then
    Crear un nuevo nodo con la distancia, ciudad1 y ciudad2;
    Establecer este nodo como el nodo actual;
    return
end
if distancia nodo actual == distancia a insertar then
    No hacer nada, ya que existe un nodo con esa distancia;
    return
end
if distancia a insertar < distancia nodo actual then
    if ∃ nodo a la izquierda del nodo actual then
        Insertar recursivamente en el nodo izquierdo;
        return
    end
    else
        Crear un nuevo nodo con la distancia, ciudad1 y ciudad2;
        Establecer este nodo como el nodo izquierdo del actual;
        return
    end
end
if distancia a insertar > distancia nodo actual then
    if ∃ nodo a la derecha del nodo actual then
        Insertar recursivamente en el nodo derecho;
        return
    end
    else
        Crear un nuevo nodo con la distancia, ciudad1 y ciudad2;
        Establecer este nodo como el nodo derecho del actual;
        return
    end
end
```

El otro método que definimos es *inorder_traversal* que permite realizar un recorrido *inorden* en *BST*. Este tipo de recorrido visita primero el subárbol izquierdo, luego el nodo actual y finalmente el subárbol derecho, siguiendo una secuencia ascendente en términos de las claves de los nodos.

En este método, se distingue sobre si hay un nodo a la izquierda o un nodo a la derecha. Primero verifica el nodo a la izquierda, en caso de haberlo se realiza recursivamente un recorrido *inorden* en este subárbol izquierdo, agregando los elementos obtenidos a la lista del resultado. Tras finalizar la condición esa, se añade al resultado el elemento correspondiente al nodo actual, que consta de la distancia, ciudad 1 y ciudad 2. Después cumple la otra condición, si existe el nodo derecho, si lo hay, se realiza recursivamente un recorrido *inorden* en este subárbol derecho, agregando los elementos obtenidos a la lista *result*.

Al finalizar, retorna la lista de los resultados con los elementos organizados.

0.2.3 Implementación en la clase Grafo.

Todo lo anterior, debemos de implementarlo en la clase Grafo, creando una nueva instancia dedicada a las distancias mediante *BST* que llame a esta clase definida anteriormente referenciada como *distancias_bst*. Además, añadimos dos métodos nuevos: *agregar_distancia_bst*, encargado de insertar nuevas distancias con sus ciudades asociadas en el *BST* dedicado a las distancias cuando se le llama, se asegura que cualquier nueva conexión entre ciudades agregada al grafo también actualice el registro de distancias en el *BST*; *mostrar_registro_ordenado*, llama al método *inorder_traversal* definido en *BST* de la instancia de distancias *BST*, devolviendo un registro ordenado de las distancias junto con las ciudades asociadas, siguiendo un orden ascendente.

Al agregar una nueva conexión entre ciudades mediante el método *agregar_conexion* en la clase Grafo, también se invoca automáticamente el método *agregar_distancia_bst*. Esto asegura que cada nueva conexión actualice el registro de distancias en el *BST*.

0.3 Minimum Spanning Tree, MST.

0.3.1 Definición.

El Minimum Spanning Tree, MST, representa la estructura de árbol que conecta todos los nodos de un grafo ponderado con el costo mínimo. Es decir, se trata de una red de conexiones mínima que garantiza la conectividad total con la menor suma de pesos en sus aristas.

Existen varios algoritmos para encontrar el MST, como Prim y Kruskal. Kruskal es especialmente útil en grafos no dirigidos, donde su lógica se basa en ordenar todas las aristas, conexión de ciudades, por peso, en nuestro caso su distancia, y seleccionar la más liviana para conectar nodos, siempre que no forme un ciclo, hasta que todos los nodos estén conectados.

Hemos optado por la selección del algoritmo de Kruskal porque en este grafo no se considera la dirección de las rutas. La aplicación de Kruskal es correcta debido a que este grafo no dirigido representa las conexiones entre ciudades sin considerar la dirección de las carreteras o rutas. Kruskal se adapta bien al problema propuesto al seleccionar las conexiones más cortas entre ciudades, creando así un árbol de recubrimiento mínimo que minimice la distancia total entre todas las ciudades conexionadas.

0.3.2 Algoritmo de Kruskal.

El algoritmo de Kruskal consiste en lo siguiente:

1. Ordenamiento de todas las aristas en orden creciente basándose en sus pesos.

2. Creación de un nuevo conjunto vacío para la representación del MST.

3. Iterar a través de cada arista en la lista ordenada de aristas. Se debe comprobar si al agregar la arista al MST se crea un ciclo.

4. En caso de que no se cree un ciclo, agregar esta arista al MST. Repetir este procedimiento hasta que todos los nodos estén unidos.

Se trata de un algoritmo de complejidad temporal eficaz para la búsqueda de un árbol de expansión mínimo de grafos grandes.

0.3.3 Implementación en la clase Grafo.

Para implementarlo a nuestro problema, debemos añadir tres métodos nuevos: encontrar, unir y KruskalMST. El método *KruskalMST* se encarga de encontrar el árbol de recubrimiento mínimo. Utiliza una lista de aristas ordenadas por la distancia de las ciudades y va seleccionando las aristas en orden ascendente. Para cada arista, verifica si unir los nodos a los que pertenece no crea un ciclo en el árbol (mediante la función unir que hace uso del método encontrar). Si no se forma un ciclo, la arista se agrega al resultado final, que es el *MST*. La variable *coste_min* se utiliza para calcular la suma de las distancias de las aristas que forman el árbol mínimo. Finalmente haciendo uso de la biblioteca *NetworkX*, permite la visualización del árbol de recubrimiento para un entendimiento mejor.