Informe municipios

Ángelo Ibáñez - 20212020007

Brayan Sierra – 20212020036

Andrés Acevedo – 2022102010

Universidad Distrital Francisco José de Caldas

Bogotá D.C.

24 de septiembre del 2024

Información general

- Lenguaje de programación: Python
- Herramienta de control de versiones: Git
- Librerías: ‘matplotlib’, ‘networkx’ y ‘disjoint_set’

Diagrama (inicial) de clases

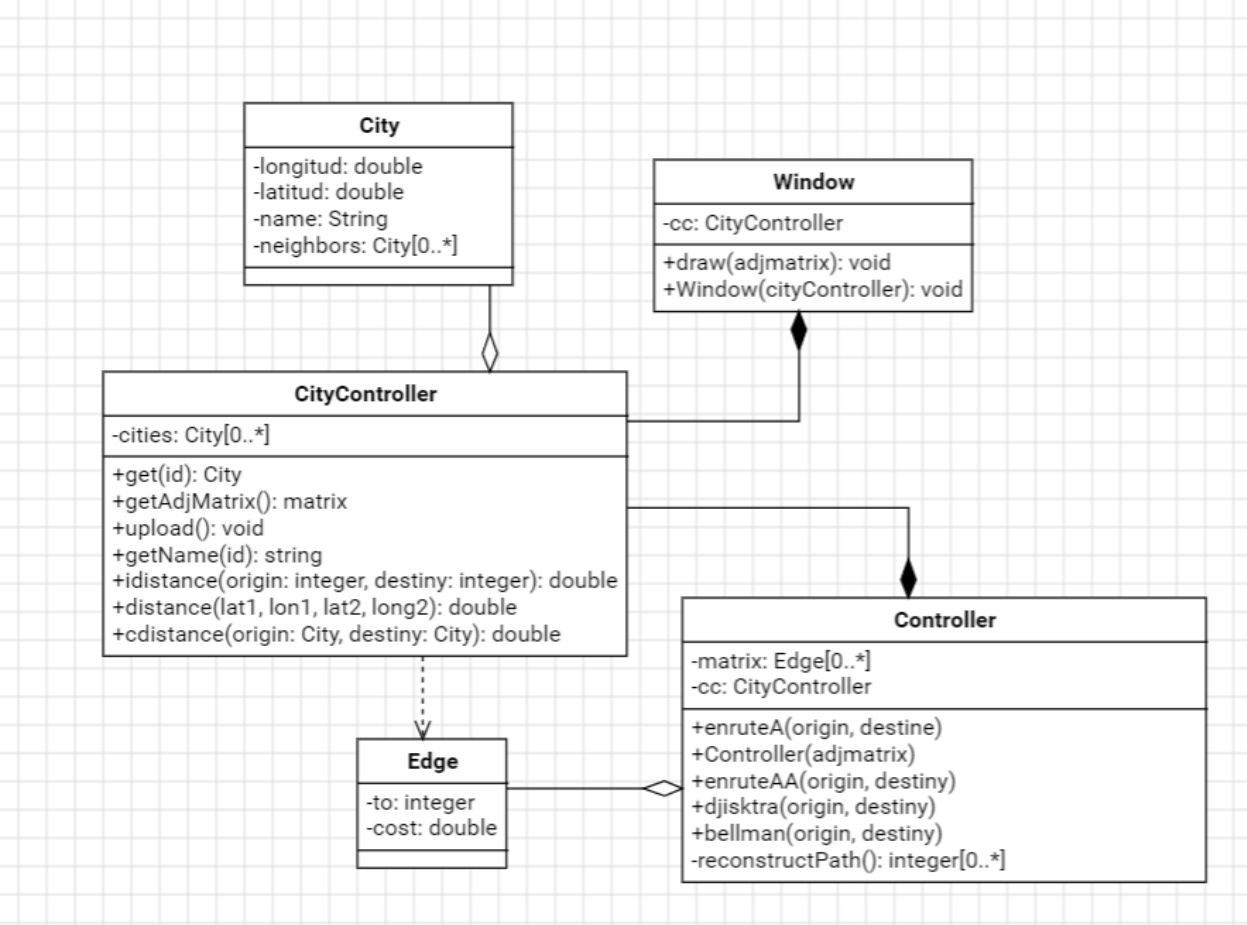
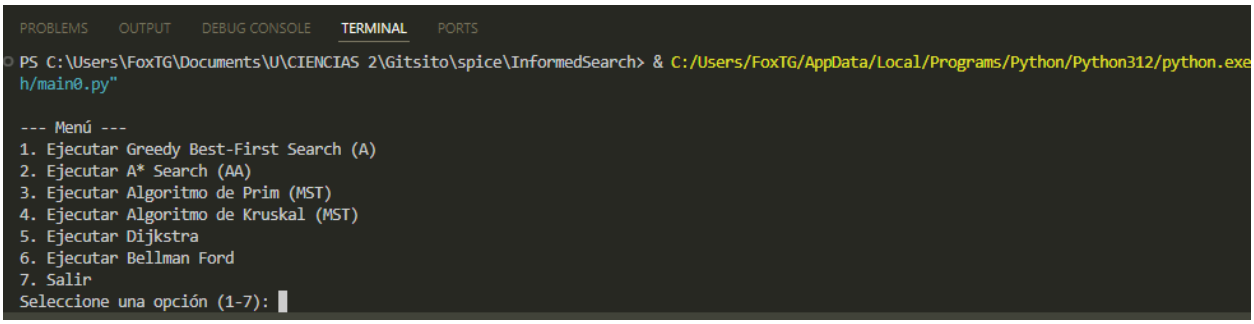


Figura 1: Diagrama de clases

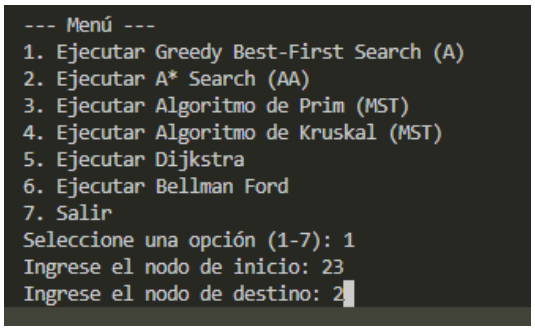
Informe

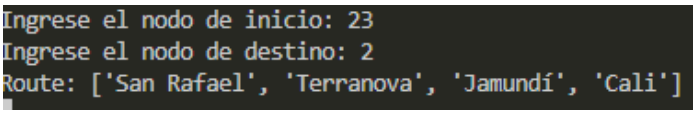
Para la construcción del grafo se utilizó una matriz de adyacencia para así generar la estructura de un total de 30 nodos. En este caso el concepto son Cali y sus municipios vecinos.

En cuanto a ejecución, se dispone de un menú en consola para poder elegir el algoritmo deseado:



En este caso, se eligió la opción número uno, es decir el algoritmo A. Se debe ingresar entonces el nodo de inicio y el nodo destino. El nodo 23 representa al municipio de San Rafael y el destino, el nodo 2, sería Cali.





El resultado, en cuanto a la consola, se imprime la ruta utilizada por el algoritmo. Para poder visualizar de una mejor manera la ruta, una ventana dibuja el grafo en su totalidad, con cada nodo con su nombre y arista con su peso. Los nodos en general son dibujados de color azul y sus aristas en negro. Los nodos de la ruta son de color verde y sus aristas de color rojo.

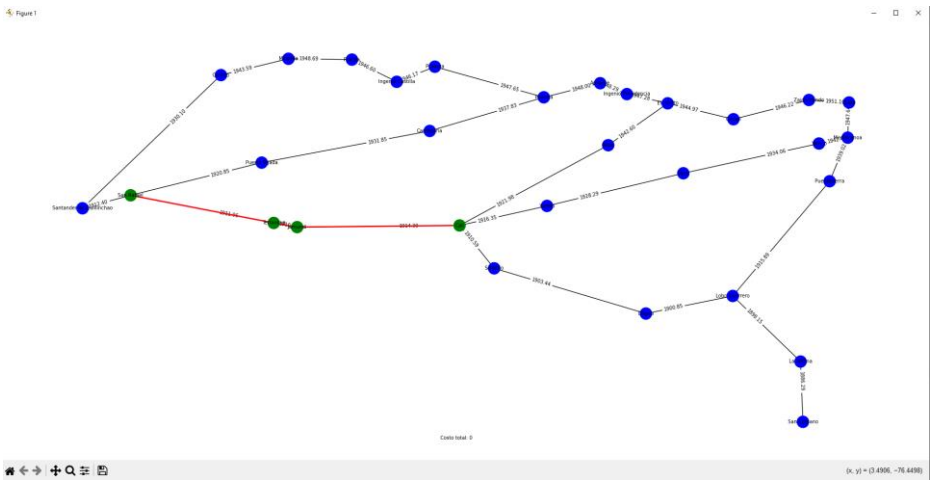


Figura 2. Impresión del grafo con los algoritmos A, A*, Dijkstra y

El resultado es prácticamente igual para los algoritmos A*, Dijkstra y Bellman Ford.

En el caso de los algoritmos Prim y Kruskal, en cuando a consola, se nos muestra la ruta utilizada para general el árbol de expansión mínima:

```
Seleccione una opción (1-7): 3
Route: ['Terranova', 'Dagua', 'Saladito', 'Gucari', 'Ingenio Castilla', 'Pradera', 'Santander de Quillinchao', 'Cali', 'Yumbo', 'Lobo Guerrero', 'Buga', 'Ingenio Providencia', 'Miranda', 'Florida', 'Amaime', 'San Cipriano', 'La delfina', 'Jamundí', 'Candelaria', 'Zajon Hondo', 'Mediacanoa', 'Puentetierra', 'Rozo', 'Vijes', 'Palmira', 'San Rafael', 'El cerrito', 'Puerto Tejada', 'Corinto', 'Yotoco']
```

Y en cuanto a la ventana, se nos muestra en la parte inferior el costo total de la ruta utilizada. Además, como es evidente, todos los nodos están coloreados de color verde, así como las aristas utilizadas para el árbol de expansión mínima están de color rojo.

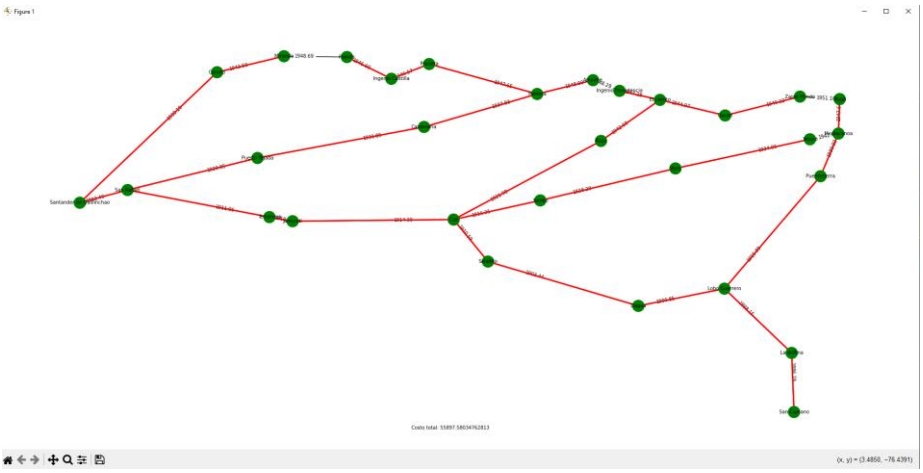
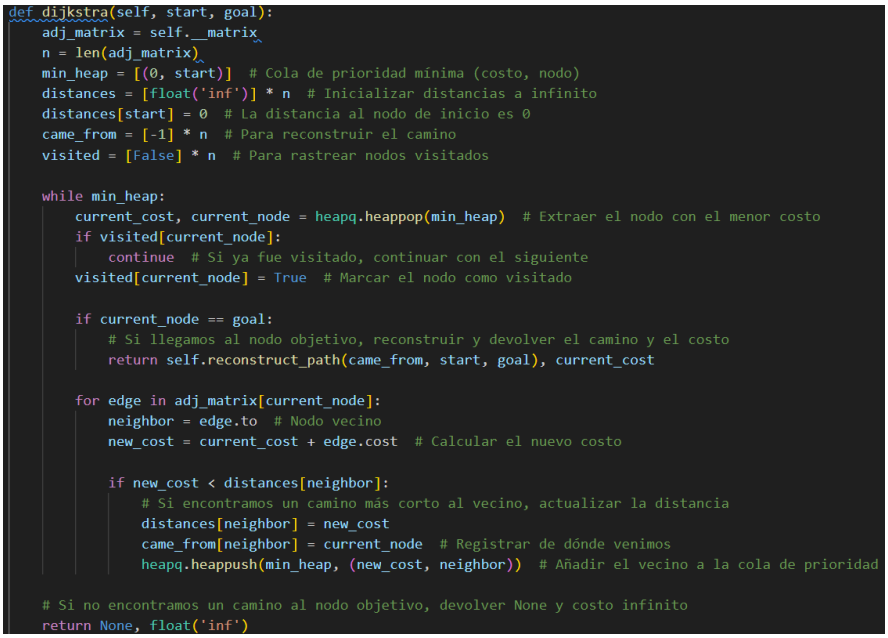


Figura 3. Impresión con los algoritmos de Prim y

En el siguiente código se implementa el algoritmo de Dijsktra, inicializamos varias estructuras, manejando una cola de prioridad para los nodos que serán recorridos, una

lista para las distancias más cortas, y otra para explorar los nodos visitados. El nodo inicial inicia con una distancia de 0, y se empieza a explorar los nodos vecinos, actualizando las distancias cuando se encuentran caminos más cortos. Durante el recorrido, el nodo con menor costo se extrae de la cola de prioridad y se marca como visitado, si se llega al nodo deseado, se reconstruye el camino seguido, y se devuelve con el costo total. Si no encuentra un camino al nodo objetivo el algoritmo retorna None, y un costo infinito.



```python
def dijkstra(self, start, goal):
    adj_matrix = self.__matrix
    n = len(adj_matrix)
    min_heap = [(0, start)]  # Cola de prioridad mínima (costo, nodo)
    distances = [float('inf')] * n  # Inicializar distancias a infinito
    distances[start] = 0  # La distancia al nodo de inicio es 0
    came_from = [-1] * n  # Para reconstruir el camino
    visited = [False] * n  # Para rastrear nodos visitados

    while min_heap:
        current_cost, current_node = heapq.heappop(min_heap)  # Extraer el nodo con el menor costo
        if visited[current_node]:
            continue  # Si ya fue visitado, continuar con el siguiente
        visited[current_node] = True  # Marcar el nodo como visitado

        if current_node == goal:
            # Si llegamos al nodo objetivo, reconstruir y devolver el camino y el costo
            return self.reconstruct_path(came_from, start, goal), current_cost

        for edge in adj_matrix[current_node]:
            neighbor = edge.to  # Nodo vecino
            new_cost = current_cost + edge.cost  # Calcular el nuevo costo

            if new_cost < distances[neighbor]:
                # Si encontramos un camino más corto al vecino, actualizar la distancia
                distances[neighbor] = new_cost
                came_from[neighbor] = current_node  # Registrar de dónde venimos
                heapq.heappush(min_heap, (new_cost, neighbor))  # Añadir el vecino a la cola de prioridad

    # Si no encontramos un camino al nodo objetivo, devolver None y costo infinito
    return None, float('inf')
```

Ahora en el siguiente código se ve la implementación del algoritmo de Bellman-Ford. Primero asignamos distancias infinitas a todos los nodos excepto al de inicio; luego realizamos un proceso de relajación repetido, en donde se actualizan las distancias mínimas si se encuentra un camino más corto a través de un nodo intermedio. También se tienen en cuenta distancias negativas, aunque en este caso, todas las distancias son positivas. Finalmente, si encuentra un camino válido hacia el nodo final, se reconstruye y devuelve el camino junto a la distancia mínima, si ese camino no existe, retorna un valor infinito.

```python
def bellman(self, start, goal):
    adj_matrix = self.__matrix
    n = len(adj_matrix)
    # Inicializar las distancias de todos los nodos a infinito (excepto el nodo de inicio)
    distances = [float('inf')] * n  # Lista de distancias mínimas desde el nodo de inicio a cada nodo
    came_from = [-1] * n  # Lista que almacena desde qué nodo llegamos al nodo actual
    distances[start] = 0  # La distancia al nodo de inicio es 0
    # Proceso de relajación n - 1 veces (donde n es el número de nodos)
    for _ in range(n - 1):
        # Para cada nodo en el grafo
        for u in range(n):
            # Iterar sobre todas las aristas del nodo actual u
            for edge in adj_matrix[u]:
                v = edge.to  # Nodo adyacente a u
                weight = edge.cost  # Peso de la arista entre u y v
                # Si la distancia a u no es infinita y encontramos una distancia más corta a v
                if distances[u] != float('inf') and distances[u] + weight < distances[v]:
                    distances[v] = distances[u] + weight  # Actualizamos la distancia mínima a v
                    came_from[v] = u  # Registramos que llegamos a v desde u
                # Considerar la arista en la dirección opuesta (para grafos no dirigidos)
                if distances[v] != float('inf') and distances[v] + weight < distances[u]:
                    distances[u] = distances[v] + weight  # Actualizamos la distancia mínima a u
                    came_from[u] = v  # Registramos que llegamos a u desde v
    # Después de la relajación, verificamos si hay ciclos de peso negativo
    for u in range(n):
        for edge in adj_matrix[u]:
            v = edge.to  # Nodo adyacente a u
            weight = edge.cost  # Peso de la arista entre u y v
            # Si todavía se puede reducir una distancia, significa que hay un ciclo de peso negativo
            if distances[u] != float('inf') and distances[u] + weight < distances[v]:
                raise ValueError("Graph contains a negative-weight cycle")  # Lanzamos error
            # También comprobamos la dirección opuesta para grafos no dirigidos
            if distances[v] != float('inf') and distances[v] + weight < distances[u]:
                raise ValueError("Graph contains a negative-weight cycle")  # Lanzamos error
    # Si la distancia al nodo de destino es infinita, significa que no hay un camino posible
    if distances[goal] == float('inf'):
        return None, float('inf')
    # Si existe un camino válido, devolvemos el camino reconstruido y la distancia mínima al nodo objetivo
    return self.reconstruct_path(came_from, start, goal), distances[goal]
```