Tarea 4 - Grafos

Santiago Rodriguez Mora - 202110332

April 11, 2025

Problema 1: Conversión entre Estructuras de Datos

1.1. De Matriz de Adyacencias a Lista de Adyacencias

Dada una matriz A de dimensiones $n \times n$ en la que $A[i][j] \neq 0$ indica la existencia de una arista entre i y j, se recorre la matriz para construir la lista de adyacencias L de la siguiente forma:

Pseudocódigo en GCL:

```
procedure ConvertAdjMatrixToList(A: matrix of integer, n: integer) returns L: array of
    list of integer
    for i from 1 to n do
        L[i] = new list
    end for
    for i from 1 to n do
        for j from 1 to n do
            if A[i][j] != 0 then
                Append(L[i], j)
            end if
        end for
    end for
    return L
end procedure
```

Se recibe una matriz de adyacencia y la transforma en una lista de adyacencia, creando primero una lista vacía para cada vértice, y luego, mediante dos ciclos anidados, revisa cada elemento de la matriz; si en la posición A[i][j] se encuentra un valor distinto de 0, se interpreta como la existencia de una conexión del vértice i al vértice j, por lo que se agrega j a la lista correspondiente a i, y al finalizar se devuelve el arreglo L que contiene, para cada vértice, los vértices a los que está conectado

Análisis de complejidad: El algoritmo recorre los n^2 elementos de la matriz, por lo que su complejidad es $O(n^2)$.

1.2. De Lista de Adyacencias a Matriz de Adyacencias

Dada una lista de adyacencias L, se debe construir una matriz A de dimensiones $n \times n$ inicializada en 0 y, para cada vértice i, se marca $A[i][j] = 1$ para cada $j \in L[i]$.

Pseudocódigo en GCL:

```
procedure ConvertAdjListToMatrix(L: array of list of integer, n: integer) returns A:
    matrix of integer
    Create matrix A[n][n] and initialize all elements to 0
    for i from 1 to n do
        for each j in L[i] do
            A[i][j] = 1
        end for
    end for
    return A
end procedure
```

Se convierte una lista de adyacencia en una matriz de adyacencia creando primero una matriz de n×n inicializada en 0, luego para cada vértice i recorre la lista L[i] y, por cada vértice j en esa lista, asigna A[i][j] = 1 para indicar la existencia de una conexión, devolviendo finalmente la matriz resultante

Análisis de complejidad: Para grafos dispersos, el recorrido de las listas cuesta $O(n + m)$, aunque en el peor caso (grafo denso) la complejidad se aproxima a $O(n^2)$.

Problema 2: Implementación de Prim's y Kruskal

2.1. Algoritmo de Kruskal (Lista de Adyacencias)

El algoritmo de Kruskal selecciona las aristas en orden creciente de peso y utiliza una estructura `Union-Find` para asegurar que no se formen ciclos.

Código en Python:

```python
class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

    def find(self, i):
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        root_i = self.find(i)
        root_j = self.find(j)
        if root_i == root_j:
            return False
        if self.rank[root_i] < self.rank[root_j]:
            self.parent[root_i] = root_j
        else:
            self.parent[root_j] = root_i
            if self.rank[root_i] == self.rank[root_j]:
                self.rank[root_i] += 1
        return True

def kruskal(graph, n):
    edges = []
    for u in graph:
        for v, weight in graph[u]:
            if u < v:
                edges.append((u, v, weight))
    edges.sort(key=lambda edge: edge[2])

    uf = UnionFind(n)
    mst = []
    for u, v, weight in edges:
        if uf.union(u, v):
            mst.append((u, v, weight))
    return mst
```

Análisis de complejidad: El paso de ordenamiento de las aristas tiene complejidad $O(m \log m)$ y utilizando operaciones casi constantes en la estructura `Union-Find`, la complejidad global es $O(m \log m)$.

2.2. Algoritmo de Prim (Matriz de Adyacencias)

El algoritmo de Prim utiliza un arreglo `key` para almacenar el peso mínimo de conexión y un arreglo `parent` para reconstruir el MST, seleccionando en cada iteración el vértice con el valor mínimo de `key`.

Código en Python:

```python
import sys

def prim(adj_matrix):
    n = len(adj_matrix)
    selected = [False] * n
    key = [sys.maxsize] * n
    parent = [-1] * n

    key[0] = 0
    mst = []

    for _ in range(n):
        min_val = sys.maxsize
        u = -1
        for v in range(n):
            if not selected[v] and key[v] < min_val:
                min_val = key[v]
                u = v
        selected[u] = True

        for v in range(n):
            if adj_matrix[u][v] != 0 and not selected[v] and adj_matrix[u][v] < key[v]:
                key[v] = adj_matrix[u][v]
                parent[v] = u

    for v in range(1, n):
        if parent[v] != -1:
            mst.append((parent[v], v, adj_matrix[parent[v]][v]))
    return mst
```

Análisis de complejidad: La selección del vértice mínimo en cada iteración cuesta $O(n)$ y, al realizarse n iteraciones, la complejidad total es $O(n^2)$.

Problema 3: Verificación de Grafo Bipartito

Se utiliza un recorrido en anchura (BFS) para asignar uno de dos colores (0 o 1) a cada vértice. Durante el recorrido, si se encuentra que dos vértices adyacentes tienen el mismo color, se concluye que el grafo no es bipartito.

Código en Python:

```
from collections import deque

def is_bipartite(graph, n):
    colors = {}
    partitions = (set(), set())

    for start in range(n):
        if start not in colors:
            colors[start] = 0
            partitions[0].add(start)
            queue = deque([start])

            while queue:
                u = queue.popleft()
                for v in graph.get(u, []):
                    if v not in colors:
                        colors[v] = 1 - colors[u]
                        partitions[colors[v]].add(v)
                        queue.append(v)
                    elif colors[v] == colors[u]:
                        return (False, None)
    return (True, partitions)
```

Análisis de complejidad: El algoritmo recorre cada vértice y cada arista una única vez, obteniéndose una complejidad de $O(n + m)$.

Casos de Prueba y Consideraciones Finales

Se han diseñado y documentado casos de prueba para cada algoritmo, considerando distintas configuraciones en cuanto a tamaño y densidad de grafos. Para **Kruskal** se han generado casos con listas de adyacencia en grafos de tamaños variados ($10 \leq n \leq 1000$) y densidades que van desde $n-1$ hasta $\binom{n}{2}$ aristas; para **Prim** se han creado casos utilizando matrices de adyacencia que representan grafos con diferentes densidades; y para la verificación de **grafo bipartito** se han incluido tanto ejemplos clásicos de grafos bipartitos como casos con ciclos impares que garantizan la identificación de grafos no bipartitos.

Automatización de Casos de Prueba y Verificación de Correctitud

La automatización de la generación de casos de prueba se implementó mediante el módulo `src/utils/test_generator.py`, el cual permite:

- **Generación aleatoria controlada:** Se generan grafos con diferentes características mediante funciones especializadas, como `generate_random_weighted_graph()` para grafos ponderados, `generate_connected_graph()` para garantizar la conexidad en casos de MST, y para la verificación de bipartición se utiliza `generate_bipartite_graph()` para casos positivos y `generate_non_bipartite_graph()` que introduce ciclos impares.

- **Diversidad y formato estandarizado:** Los casos generados siguen un formato consistente. Así, para Kruskal se utiliza una lista de adyacencias con pesos, para Prim se emplea una matriz de adyacencias, y para la verificación de bipartición se utiliza una lista de adyacencias sin pesos. Además, se puede controlar la cantidad de casos mediante un comando específico, permitiendo generar pruebas robustas a nivel de escenarios extremos.

La correcta ejecución de las implementaciones se verificó de diversas formas:

- **Diseño basado en teoría:** Los algoritmos se implementaron siguiendo fielmente sus definiciones teóricas. En el caso de Kruskal, se asegura la selección de aristas mediante un ordenamiento correcto y la detección de ciclos a través de la estructura Union-Find; en Prim, se garantiza la selección del vértice de menor peso en cada iteración; y en la verificación de bipartición se utiliza un recorrido BFS con coloración alterna.

- **Propiedades invariantes:** Se comprueba que los MST obtenidos tengan exactamente $n-1$ aristas, y en el algoritmo de bipartición se verifica que las particiones de vértices sean completamente disjuntas.

- **Comparación y anotación:** Se anotan las características esperadas de los casos generados, especialmente para la verificación de bipartición, lo que permite realizar una verificación automática de los resultados.

Ejemplos de Ejecución Correcta

```
$ python3 -m src.main 1 tests/problema2/kruskal/case_1.txt
Ejecutando Kruskal en tests/problema2/kruskal/case_1.txt...
MST encontrado con 499 aristas
Peso total del MST: 684
Tiempo de ejecución: 0.010140 segundos

$ python3 -m src.main 2 tests/problema2/prim/case_1.txt
Ejecutando Prim en tests/problema2/prim/case_1.txt...
MST encontrado con 499 aristas
Peso total del MST: 684
Tiempo de ejecución: 0.024564 segundos

$ python3 -m src.main 3 tests/problema3/bipartite/case_1.txt
Verificando bipartición en tests/problema3/bipartite/case_1.txt...
El grafo NO es bipartito.
Tiempo de ejecución: 0.000029 segundos
```

En estos ejemplos se evidencia que, incluso en grafos relativamente grandes (por ejemplo, 500 vértices), el algoritmo de Kruskal obtiene un MST correcto con $n-1$ aristas en tiempos muy reducidos, mientras que Prim arroja resultados equivalentes en cuanto al peso total del MST, confirmando la convergencia de ambos algoritmos. Por otra parte, la verificación de la bipartición se realiza de manera extremadamente rápida, detectando eficazmente la presencia de ciclos impares cuando el grafo no es bipartito.

Conclusiones

La integración de la generación automatizada de casos de prueba, con una verificación sistemática basada en propiedades teóricas e invariantes, demuestra la robustez y efectividad de las implementaciones para Kruskal, Prim y la verificación de grafos bipartitos. Los resultados obtenidos muestran que los algoritmos producen resultados correctos en un amplio rango de escenarios, que la ejecución es consistente con las complejidades teóricas esperadas, y que la automatización de pruebas garantiza una cobertura exhaustiva incluso en casos extremos.