UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

Análisis de algoritmos

Mtro. Jorge Ernesto Lopez Arce Delgado
Act. Técnica Voraz

Integrantes:
Braulio Hurtado Escoto 220426225
Jorge Daniel Hernández Reyes 220027797
Juan Pablo Solis Regin 220468416

Introducción

Definiciones
Algoritmo de Prim: El algoritmo de Prim es un algoritmo de tipo voraz. Se caracteriza por empezar con un solo nodo y se mueve por varios nodos adyacentes, esto con el fin de explorar todas las aristas que se conectan en el camino. Las características de este algoritmo son las siguientes:

- Comienza con un árbol de expansión vacío.
- Se mantienen dos conjuntos de vértices. El primero contiene los vértices ya incluidos en el MST y el otro contiene los vértices que aún no se han incluido.
- Se consideran todas las aristas de menor peso.

Básicamente, empieza por un nodo raíz, donde toma todos los caminos hasta terminar de recorrerlos todos sin saltarse ninguno de los vértices o nodos.

Algoritmo de Kruskal: Es un algoritmo de árbol de expansión mínima donde se busca recorrer todos los nodos pero buscando la ruta con el menor costo posible. Los pasos para su implementación son los siguientes:

- Ordena todas las aristas de menor a mayor peso.
- Se debe seleccionar la arista con el menor peso posible y se agrega al árbol. Si cuando se agrega la arista se crea un ciclo, debe ser descartado.
- Se añaden aristas hasta recorrer todos los nodos o vértices.

A través de esta actividad, se busca conocer el funcionamiento en código de estos algoritmos y entender cómo se comporta cada uno y que ventajas o desventajas nos ofrecen cada uno de estos. Siendo ambos algoritmos de técnica voraces, pueden ser similares, pero cada uno funciona con sus propias reglas y propósito para el cual fue diseñado.

Objetivo

Nuestro objetivo para esta actividad es el entender mejor el uso de cada uno de estos algoritmos, para comparar resultados a la hora de ejecutarlos y elegir cual fue el algoritmo más eficiente en términos generales, ya que al ser similares, podrían tener resultados igualmente similares, por lo que a la hora de ejecutarlos se determinará con distintos datos cual es mejor.

Desarrollo:

Esta actividad se dividirá de la siguiente manera:

Integrante	Funciones principales	Actividades específicas	Entregables / Resultados
Jorge Daniel Hernández Reyes	Ejecución y testing del programa.	Se harán las pruebas de ejecución necesarias para comprobar el correcto funcionamiento del programa y de cada uno de los algoritmos.	Evidencias de la ejecución del programa y sus respectivos algoritmos a probar (Prim y Kruskal).
Braulio Hurtado Escoto	Implementación del código.	Por medio de la investigación se hará el desarrollo final del código para poder ejecutarlo y hacer las respectivas pruebas.	Desarrollo de algoritmos de Prim y Kruskal.
Juan Pablo Solis Regin	Investigación y armado de código.	Se realizará una investigación acerca de cada algoritmo, sus posibles implementaciones y definiciones de variables y recursos que se requerirán para el desarrollo del código.	Investigación sobre algoritmos de Prim y Kruskal y su implementación en Python.

Código en Python:

Python

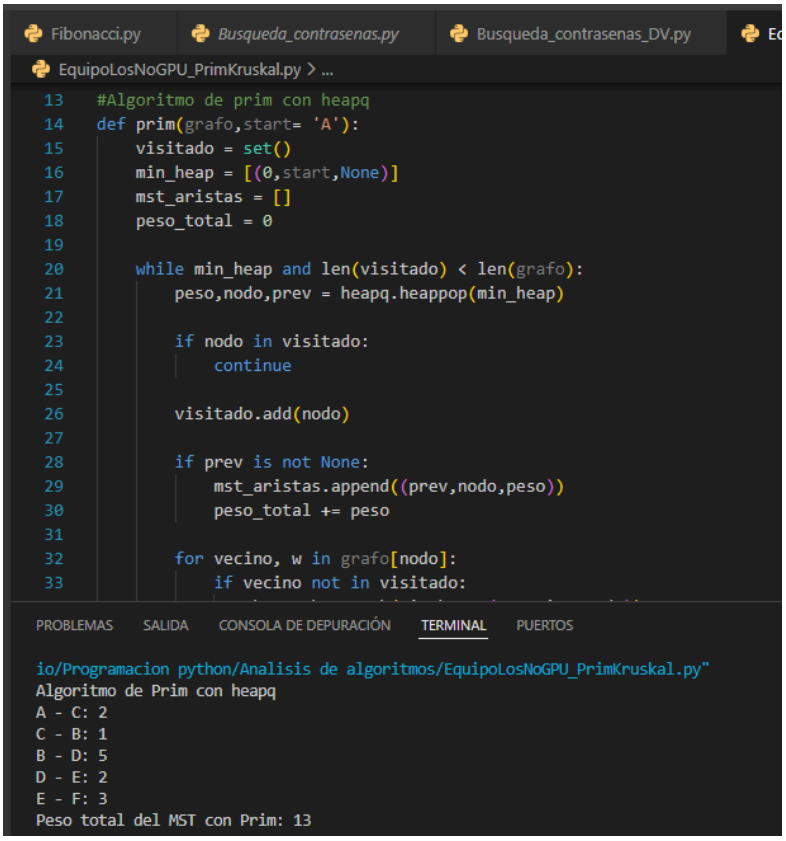
```python
import heapq

#Grafo ponderado no dirigido con 6 nodos y 9 aristas
grafo = {
    'A': [('B',4),('C',2)],
    'B': [('A',4),('C',1),('D',5)],
    'C': [('A',2),('B',1),('D',8),('E',10)],
    'D': [('B',5),('C',8),('E',2),('F',6)],
    'E': [('C',10),('D',2),('F',3)],
    'F': [('D',6),('E',3)]
}

#Algoritmo de prim con heapq
def prim(grafo,start= 'A'):
    visitado = set()
    min_heap = [(0,start,None)]
    mst_aristas = []
    peso_total = 0

    while min_heap and len(visitado) < len(grafo):
        peso,nodo,prev = heapq.heappop(min_heap)

        if nodo in visitado:
            continue

        visitado.add(nodo)

        if prev is not None:
            mst_aristas.append((prev,nodo,peso))
            peso_total += peso

        for vecino, w in grafo[nodo]:
            if vecino not in visitado:
                heapq.heappush(min_heap,(w,vecino,nodo))
```

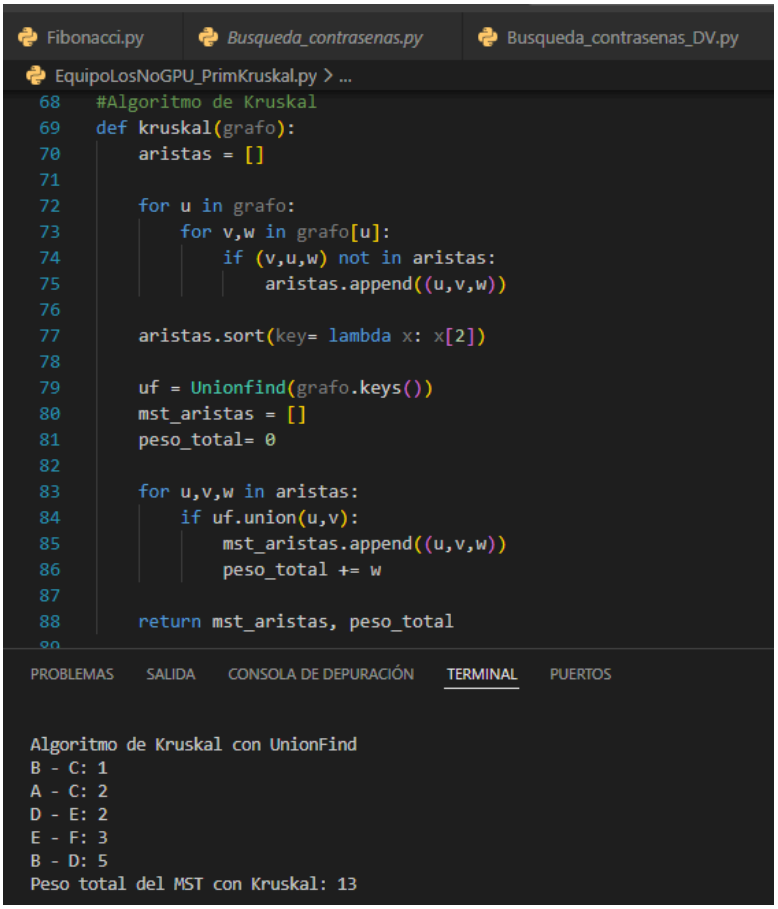
```
    return mst_aristas,peso_total


#Union-Find
class Unionfind:
    def __init__(self,nodos):
        self.parent = {x: x for x in nodos}
        self.rank = {x: 0 for x in nodos}

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

    def union(self,a,b):
        rootA = self.find(a)
        rootB = self.find(b)

        if rootA == rootB:
            return False

        if self.rank[rootA] < self.rank[rootB]:
            self.parent[rootA] = rootB
        elif self.rank[rootA] > self.rank[rootB]:
            self.parent[rootB] = rootA
        else:
            self.parent[rootB] = rootA
            self.rank[rootA] += 1

        return True


#Algoritmo de Kruskal
def kruskal(grafo):
```

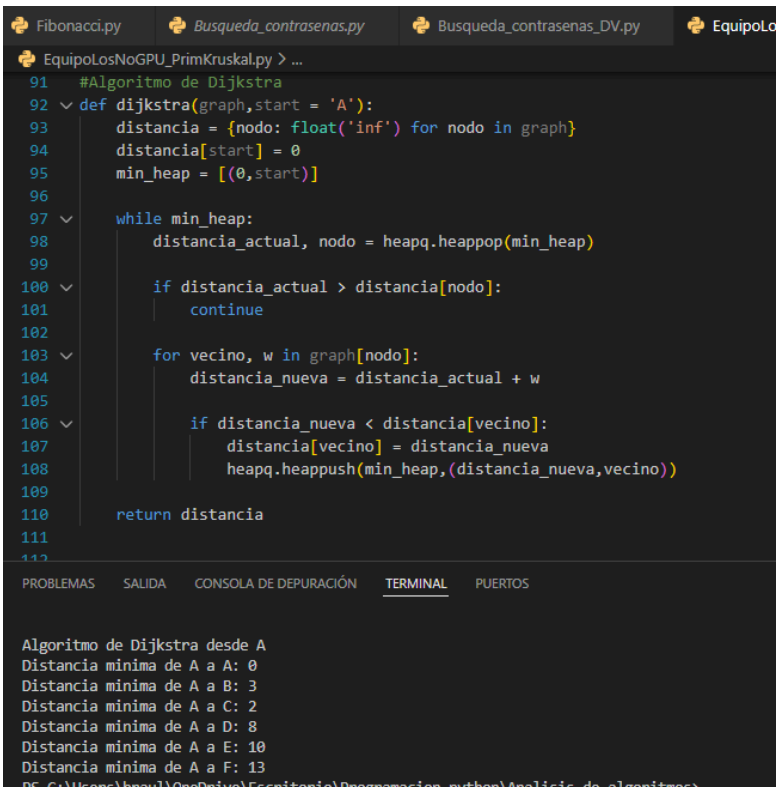
```
    aristas = []

    for u in grafo:
        for v,w in grafo[u]:
            if (v,u,w) not in aristas:
                aristas.append((u,v,w))

    aristas.sort(key= lambda x: x[2])

    uf = Unionfind(grafo.keys())
    mst_aristas = []
    peso_total= 0

    for u,v,w in aristas:
        if uf.union(u,v):
            mst_aristas.append((u,v,w))
            peso_total += w

    return mst_aristas, peso_total


#Algoritmo de Dijkstra
def dijkstra(graph,start = 'A'):
    distancia = {nodo: float('inf') for nodo in graph}
    distancia[start] = 0
    min_heap = [(0,start)]

    while min_heap:
        distancia_actual, nodo = heapq.heappop(min_heap)

        if distancia_actual > distancia[nodo]:
            continue

        for vecino, w in graph[nodo]:
            distancia_nueva = distancia_actual + w
```

```
            if distancia_nueva < distancia[vecino]:
                distancia[vecino] = distancia_nueva
                heapq.heappush(min_heap,(distancia_nueva,vecino))

    return distancia


#MAIN
if __name__ == "__main__":
    print("Algoritmo de Prim con heapq")
    prim_aristas, prim_peso = prim(grafo,'A')
    for u,v,w in prim_aristas:
        print(f"{u} - {v}: {w}")
    print(f"Peso total del MST con Prim: {prim_peso}\n")

    print("Algoritmo de Kruskal con UnionFind")
    kruskal_aristas, kruskal_peso = kruskal(grafo)
    for u,v,w in kruskal_aristas:
        print(f"{u} - {v}: {w}")
    print(f"Peso total del MST con Kruskal: {kruskal_peso}\n")

    print("Algoritmo de Dijkstra desde A")
    distancia = dijkstra(grafo,'A')
    for nodo in distancia:
        print(f"Distancia minima de A a {nodo}:
{distancia[nodo]}")
```

Capturas de pantalla:







Comparación entre Prim y Kruskal

En Prim, se usa una cola de prioridad (con heapq) para seleccionar en cada paso la arista de menor peso que conecte un nodo nuevo al árbol ya construido. Este método es eficiente cuando el grafo está representado mediante listas de adyacencia y su complejidad es O(E log V).

Por otro lado, Kruskal utiliza una estructura Union-Find para unir partes y evitar ciclos. El algoritmo selecciona las aristas de menor peso una por una, ya ordenadas, sin importar el nodo inicial, pues trabaja sobre un conjunto de componentes disjuntos y los va uniendo hasta formar el árbol de expansión mínima además tiene una complejidad de O(E log E) que puede llegar a ser O(E log V).

Conclusión

Durante la actividad se logró comprender de manera más profunda el funcionamiento y la diferencias entre estos dos algoritmos vistos, que son el algoritmo de Prim y de Kruskal, estos ambos son usados para la construcción de árboles de expansión mínima dentro de un grafo, estos mismos pertenecen a los algoritmos de técnica voraces y el objetivo es encontrar el camino de menor costo que pueda conectar todos los nodos, pero cada uno su resultado puede ser mejor en diferentes escenarios.

Dentro de las pruebas realizadas y la investigación determinamos que Prim es más eficiente cuando el grafo es denso y tiene una estructura como una matriz de adyacencia y por parte de Kruskal tiende a comportarse de una mejor manera con grafos dispersos debido a que trabaja de manera más directa con las aristas ordenadas.

El algoritmo que más nos resultó interesante fue el algoritmo de Prim, conectamos muy bien con este y además fue más sencillo de implementar y entender para todos nosotros. Es un algoritmo bastante completo a pesar de su sencillez, pero que tiene grandes aplicaciones y usos prácticos, como en este caso que lo implementamos en nuestro código.

Fuentes de consulta:

- *Kruskal's Algorithm*. (n.d.). Www.programiz.com. https://www.programiz.com/dsa/kruskal-algorithm
- *ÁRBOL DE EXPANSIÓN MÍNIMA: ALGORITMO DE KRUSKAL*. (2012, April 19). Algorithms and More. https://jariasf.wordpress.com/2012/04/19/arbol-de-expansion-minima-algoritmo-de-kruskal/
- GeeksforGeeks. (2012, November 18). *Prim's Algorithm for Minimum Spanning Tree (MST)*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/prims-minimum-spanning-tree-mst-greedy-algo-5/
- Chris, K. (2023, February 14). *Prim's Algorithm – Explained with a Pseudocode Example*. FreeCodeCamp.org. https://www.freecodecamp.org/news/prims-algorithm-explained-with-pseudocode/