Instituto Politécnico Nacional

Unidad Profesional Interdisciplinaria en Ingeniería Campus Zacatecas

Unidad de Aprendizaje

Práctica 1: Análisis de Casos

Realizado por:

Boleta	Nombre	Apellidos
2023670031	Antonio	Valdés Hernández

1 de diciembre de 2023

Índice

1. Introducción

El objetivo principal del algoritmo de Dijkstra es calcular las distancias mínimas desde el nodo de inicio a todos los demás nodos del grafo. A través de su enfoque voraz, el algoritmo selecciona iterativamente el nodo más cercano según la distancia actualmente conocida y actualiza las distancias a sus nodos adyacentes si se encuentra una ruta más corta.

La ejecución del algoritmo de Dijkstra se compone de tres fases clave: inicialización, iteración y finalización. En la fase de inicialización, se establecen las distancias iniciales, considerando el nodo de inicio con distancia cero y todos los demás nodos con distancia infinita. Durante la iteración, el algoritmo selecciona y explora nodos en orden de distancia creciente, actualizando las distancias de los nodos adyacentes según sea necesario. El proceso continúa hasta que se visitan todos los nodos o se vacía la cola de prioridad. Finalmente, al concluir, se obtienen las distancias mínimas desde el nodo de inicio a todos los demás nodos.

La eficiencia del algoritmo de Dijkstra radica en su capacidad para manejar grafos ponderados, priorizando la exploración de las rutas más cortas en lugar de explorar todo el espacio de búsqueda. Sin embargo, es esencial destacar que el algoritmo asume que los pesos de las aristas son no negativos, ya que la presencia de aristas negativas podría provocar comportamientos impredecibles.

2. Desarrollo

2.1. Descripción del Algoritmo de Dijkstra

El algoritmo de Dijkstra, desarrollado por Edsger Dijkstra en 1956, es un algoritmo voraz que resuelve el problema de caminos mínimos en grafos dirigidos con pesos no negativos. A continuación, se detalla su funcionamiento:

2.2. Inicialización

Se asigna una distancia inicial de 0 al nodo de inicio y de infinito a todos los demás nodos. Se mantiene una cola de prioridad (min-heap) que contiene los nodos no visitados, ordenados por sus distancias actuales.

2.3. Iteración

Mientras la cola de prioridad no esté vacía, se extrae el nodo con la distancia mínima. Se actualizan las distancias de los nodos adyacentes si se encuentra una ruta más corta a través del nodo actual.

2.4. Finalización

Una vez que todos los nodos han sido visitados o la cola de prioridad está vacía, el algoritmo termina.

2.5. Resultado

Al finalizar, las distancias mínimas desde el nodo de inicio a todos los demás nodos se han calculado y se pueden utilizar para determinar las rutas más cortas.

2.6. Implementación en Python con ejemplo

```
import heapq
import time
#El siguiente código es el código con el ejemplo pedido en la práctica:
class Grafoo:
    def __init__(xd):
        xd.vertiC = set() #Conjunto par aalmacenar vértices
        xd.aristocrata = {} #Diccionario para almacenar las aristas de cada vertice uwu
    def add_vertex(xd, nodo):
        xd.vertiC.add(nodo) #Agregar un vértice al conjunto de vértices
        xd.aristocrata[nodo] = [] #Inicializar una lista vacía para las aristas del vértice
```

```
    def add_edge(xd, inicio, finite, pesao):
        xd.aristocrata[inicio].append((finite, pesao))
        #Agregar una arista con peso desde un vértice de inicio a uno final

def Sigismund(gragrafo, inicio):
    dist = {nodo: float('infinity') for nodo in gragrafo.vertiC}
    #Inicializar distancias a infinito para todos los vértices
    dist[inicio] = 0
    #Establecer la distancia del vértice de inicio a sí mismo como 0
    lista_priori = [(0, inicio)]
    #Se tiene la cola de prioridad para manejar los nodos no visitados ordenados por distancia
    while lista_priori:
        current_distance, current_vertex = heapq.heappop(lista_priori)
        #Obtener el nodo con la distancia mínima
        if current_distance > dist[current_vertex]:
            continue
            #Si la distancia actual es mayor que la almacenada, ignorar este nodo
        for laVecinita, pesao in gragrafo.aristocrata[current_vertex]:
            dizt = current_distance + pesao #Calcular la nueva distancia
            if dizt < dist[laVecinita]:
                dist[laVecinita] = dizt #Actualizar la distancia mínima
                heapq.heappush(lista_priori, (dizt, laVecinita))
                #Agregar el vecino a la cola de prioridad
    return dist
#Devolver las distancias mínimas desde el vértice de inicio a todos los demás vértices

def DIO(gragrafo, inicio):
    za = time.time() #Tiempo inicial
    Sigismund(gragrafo, inicio) #Llamar a la función Dijkstra
    warudo = time.time() #Tiempo final
    return warudo - za #Devolver el tiempo de ejecución

#Crear un grafo de ejemplo
grafo = Grafoo()
grafo.add_vertex("A")
grafo.add_vertex("B")
grafo.add_vertex("C")
grafo.add_edge("A", "B", 3)
grafo.add_edge("A", "C", 5)
grafo.add_edge("B", "C", 2)

#Probar el algoritmo de Dijkstra con el grafo de ejemplo dado en la práctica
inicio = "A"
distancias_minimas = Sigismund(grafo, inicio)
print(f"Distancias mínimas desde el vértice {inicio}: {distancias_minimas}")

#Y ya para terminar con esto, medimos el tiempo de ejecución :D
toki_wo_tomare = DIO(grafo, inicio)
print(f"El tiempo de ejecución es: {toki_wo_tomare} byou ga sugita")
print("Ari ari ari arivedercci")
#Arrivedercci
```

2.7. Implementación en Python sin ejemplo

```
import heapq
import time
import random
import matplotlib.pyplot as plt
#Para este código, ya no se usará el ejemplo pedido en la práctica,
# sino que se usarán muchos grafos aleatorios
# y e graficará su tiempo de ejecución :D

# Definición de la clase para representar el grafo
class Grafoo:
    def __init__(xd):
        xd.vertiC = set() #Conjunto para almacenar vértices
        xd.aristocrata = {}  #Diccionario para almacenar las arisas de cada vértice
    def aniadirVertice(xd, nodo):
        xd.vertiC.add(nodo) #Agregar un vértice al conjunto de vértices
        xd.aristocrata[nodo] = []
        #Inicializar una lita vacía para las aristas del vértice
    def aniadirArista(xd, inicio, finite, pesao):
        xd.aristocrata[inicio].append((finite, pesao))
        #Agregar una aristac on peso desde un vértice de inicio a uno final

#Algoritmo de Dijkstra para encontrar los  caminos mínimos en un grafo ponderado dirigido
def Sigismund(gragrafo, inicio):
    dist = {nodo: float('infinity') for nodo in gragrafo.vertiC}
    #Inicialaizar distancias a infinito para todos los vértices
    dist[inicio]= 0
    #Establecer la distancia del vértice de inicio a sí mismo como 0
    lista_priori =[(0, inicio)]
    #Cola de p rioridad para manejar los nodos no visitados ordenados por distancia

    while lista_priori:
        current_distance, current_vertex = heapq.heappop(lista_priori)
        #Obtener el nodo con la distancia mínima
        if current_distance > dist[current_vertex]:
            continue
        #Si la distancia actual es mayor que la almacenada, ignorar este nodo
        for laVecinita, pesao in gragrafo.aristocrata[current_vertex]:
            dizt = current_distance + pesao #Calculr la nueva distancia
            if dizt < dist[laVecinita]:
                dist[laVecinita] = dizt #Actualizar la distancia mínima
                heapq.heappush(lista_priori, (dizt, laVecinita))
                #Agregar el vecino a la cola de prioridad
    return dist
#Devolver las distancias mínimas desde el vértice de inicio a todos los demás vértices

#Función para medir el tiempo y ejecutar el algoritmo en varios grafos aleatorios
def medir_tiempo_y_graficar(NoGrafos, NoVertices, NoAristas):
    tiempoEjecucionuwu = []
    for _ in range(NoGrafos):
```

```
        grafo = Grafoo()
        for i in range(NoVertices):
            grafo.aniadirVertice(str(i))
        for _ in range(NoAristas):
            start = str(random.randint(0, NoVertices - 1))
            end = str(random.randint(0, NoVertices - 1))
            weight = random.randint(1, 10)
            grafo.aniadirArista(start, end, weight)
        empezar = str(random.randint(0, NoVertices - 1))

        tiempo_ejecucion = DIO(grafo, empezar)
        tiempoEjecucionuwu.append(tiempo_ejecucion)

    #Graficar los tiempos de ejecución
    plt.plot(tiempoEjecucionuwu, marker='o', linestyle='-', color='b')
    plt.title(f'Tiempos de ejecución para {NoGrafos} grafos aleatorios')
    plt.xlabel('Número de grafos')
    plt.ylabel('Tiempo de ejecución (en segundos xd)')
    plt.show()

def DIO(gragrafo, inicio):
    za = time.time() #Tiempo inicial
    Sigismund(gragrafo, inicio) #Llamar a la función Dijkstra
    warudo = time.time() #Tiempo final
    return warudo - za #Devolver el tiempo de ejecución

#Parámetros para la función
NoGrafos=100
NoVertices = 10
NoAristas = 15

#Llamar a la función para medir el tiempo y graficar
medir_tiempo_y_graficar(NoGrafos, NoVertices, NoAristas)
```

2.8. Gráfico

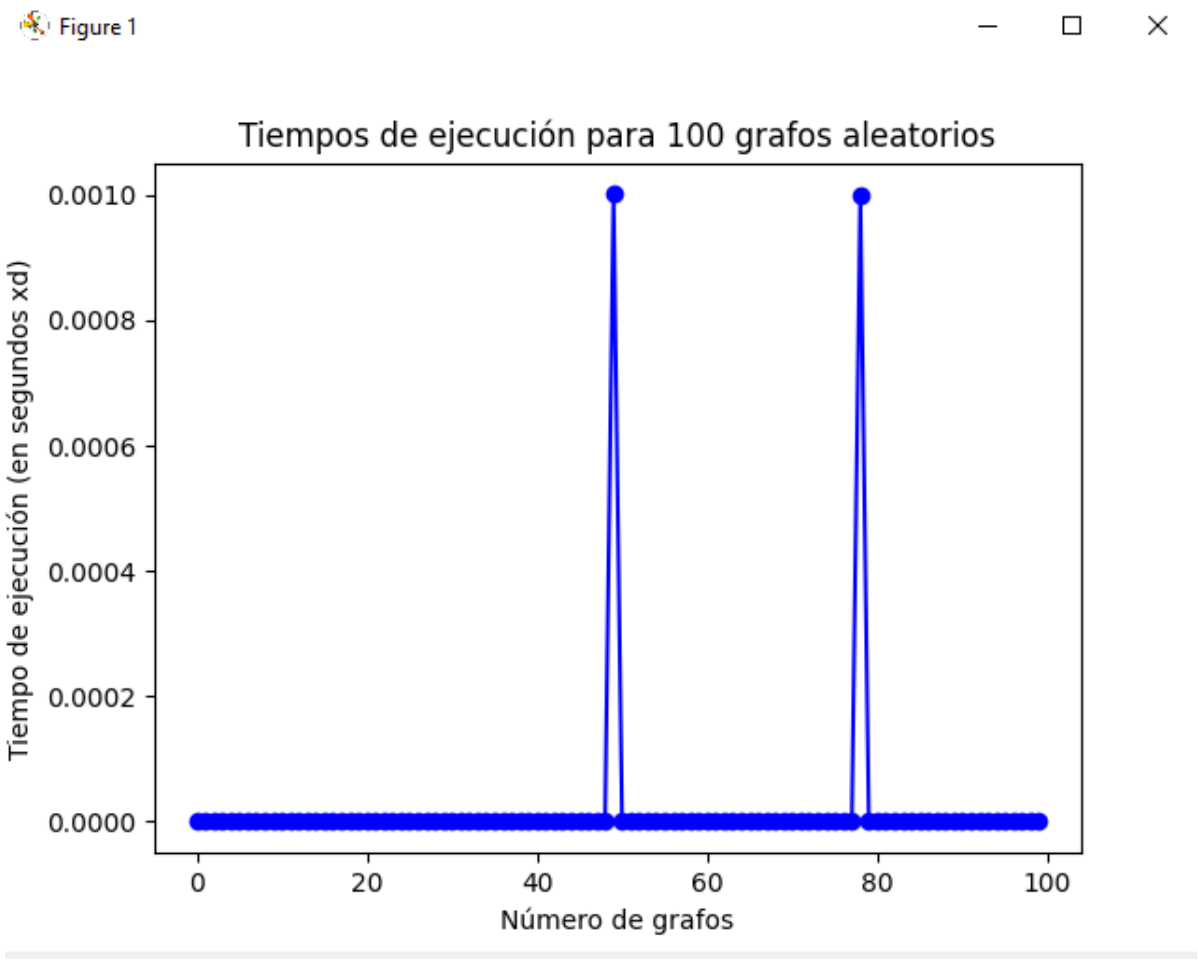
Figure 1
Tiempos de ejecución para 100 grafos aleatorios
Tiempo de ejecución (en segundos xd)
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
0
20
40
60
80
100
Número de grafos

2.8.1. Complejidad BigO

La complejidad Big O del algoritmo de Dijkstra depende de cómo se implemente y de la estructura de datos utilizada. En este caso, la implementación utiliza un heap para manejar la cola de prioridad, que es esencial para la eficiencia del algoritmo.

La implementación de Dijkstra utiliza un heap binario para mantener los nodos no visitados ordenados por distancia. Las operaciones principales que contribuyen a la complejidad son:

Inicialización de distancias: La inicialización de las distancias toma O(V) (linea dist = nodo: float('infinity') for nodo in gragrafo.vertiC), donde V es el número de vértices.

Bucle principal: El bucle principal ejecutará en el peor caso $O(V + E * log(V))$, donde V es el número de vértices y E es el número de aristas (lineas del bucle principal). Esto se debe a las operaciones de extracción y adición al heap.

Extracción de la cola de prioridad: Cada extracción de la cola de prioridad tiene una complejidad de $O(logV)$.

Por lo tanto, la complejidad total es $O(V + E * log(V))$. En el peor caso, cuando el grafo es denso y E es cercano a V^2, esto puede simplificarse a $O(V^2 * log(V))$.

3. Conclusión

El algoritmo de Dijkstra es una herramienta eficaz para encontrar los caminos más cortos en grafos ponderados no negativos. Su eficiencia se destaca en aplicaciones como sistemas de rutas y redes, donde minimizar distancias es esencial. Sin embargo, una limitación importante es su dependencia de pesos no negativos en las aristas, ya que no maneja adecuadamente los pesos negativos.

La clave de la eficiencia de Dijkstra radica en el uso de una cola de prioridad, que selecciona los vértices de manera óptima. Esto se traduce en un rendimiento rápido y preciso en la búsqueda de rutas mínimas. No obstante, su complejidad puede aumentar en grafos densos, siendo menos eficiente en tales casos.

La aplicabilidad del algoritmo se extiende a diversas áreas, desde logística hasta planificación, destacando su versatilidad. Sin embargo, es crucial considerar la elección de estructuras de datos eficientes, como el heap binario, para garantizar un rendimiento óptimo.

En términos prácticos, Dijkstra es esencial para optimizar rutas y minimizar costos en situaciones donde la distancia juega un papel crítico. Para grafos grandes o con características específicas, puede ser necesario evaluar variantes como A* o Bellman-Ford para adaptarse a las necesidades del problema.

4. Referencias

- *-*, F. (s/f). Algoritmo-de-Dijkstra: implementacion del Algoritmo de Dijkstra en python utilizando min heap.
- Coding games and programming challenges to code better. (s/f). CodinGame. Recuperado el 1 de diciembre de 2023, de https://www.codingame.com/playgrounds/7656/los-caminos-mas-cortos-con-el-algoritmo-de-dijkstra/el-algoritmo-de-dijkstra
- de Ingeniería UNAM, L.-F. [@licad-facultaddeingenieria5339]. (2020, septiembre 19). 13 . Algoritmo de Dijkstra en Python. Youtube. https://www.youtube.com/watch?v=wYyrMnfPmMw
- Imprimir el camino mas corto con Algoritmo de Dijkstra (repetidas veces). (s/f). Stack Overflow en español. Recuperado el 1 de diciembre de 2023, de https://es.stackoverflow.com/questions/473973/imprimir-el-camino-mas-corto-con-algoritmo-de-dijkstra-repetidas-veces
- Khetarpal, V. (2021, octubre 22). Algoritmo de Dijkstra en Python. Delft Stack. https://www.delftstack.com/es/howto/python/dij algorithm-python/
- Navone, E. C. (2022, octubre 24). Algoritmo de la ruta más corta de Dijkstra - Introducción gráfica y detallada. freecodecamp.org. https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/
- Rutas más cortas de fuente única: algoritmo de Dijkstra. (s/f). Techiedelight.com. Recuperado el 1 de diciembre de 2023, de https://www.techiedelight.com/es/single-source-shortest-paths-dijkstras-algorithm/
- (S/f). Linkedin.com. Recuperado el 1 de diciembre de 2023, de https://es.linkedin.com/pulse/dijkstra-en-python-con-pyvis-y-networkx-diego-alejandro-gonzález