ESTADISTICA EN PHYTON

Electiva - II

Jaime Andres Martinez

2
0
2
5

Taller Nº4

Cristhian Cañar
21/03/2025

1. Dijkstra

El código proporcionado implementa el algoritmo de Dijkstra para calcular las distancias más cortas desde un nodo origen en un grafo ponderado. Utiliza **heapq** para gestionar prioridades, **timeit** para medir el tiempo promedio de ejecución, **networkx** para crear el grafo y **matplotlib** para visualizarlo. El algoritmo actualiza las distancias mientras explora los nodos, y al final muestra las distancias mínimas. La Figura 1 y 2 corresponde a este código, que incluye la definición del grafo, la lógica del algoritmo, la función para graficar y la medición del tiempo promedio de ejecución mediante **timeit**. Por otro lado, la Figura 3 es la representación gráfica generada por el código, que muestra visualmente el grafo con sus nodos, aristas y los pesos asociados a cada conexión. Esta gráfica ayuda a entender la estructura del grafo y cómo se relacionan las distancias calculadas por el algoritmo.

```python
import heapq
import timeit  # Importamos timeit para medir el tiempo de ejecución
import networkx as nx
import matplotlib.pyplot as plt

def dijkstra(grafo, nodo_origen):
    # Inicializar las distancias más cortas a infinito
    distancias = {nodo: float('inf') for nodo in grafo}
    distancias[nodo_origen] = 0  # Se le indica que la distancia al nodo origen es 0
    # Cola para determinar la exploración de nodos
    cola_prioridad = [(0, nodo_origen)]
    # Mientras haya nodos, va a explorar
    while cola_prioridad:
        # Obtención del nodo con menor distancia, como primera vez sabemos que es 0
        distancias_actual, nodo_actual = heapq.heappop(cola_prioridad)
        # Si la distancia actual es mayor a la registrada, continuar
        if distancias_actual > distancias[nodo_actual]:
            continue
        # Explorar los vecinos del nodo actual (cercanos)
        for vecino, peso in grafo[nodo_actual].items():
            # Calcular la distancia al vecino a través del nodo actual
            nueva_distancia = distancias_actual + peso
            # Si la nueva distancia es menor que la registrada, actualizar
            if nueva_distancia < distancias[vecino]:
                distancias[vecino] = nueva_distancia
                # Agregar al vecino a la cola de prioridad
                heapq.heappush(cola_prioridad, (nueva_distancia, vecino))
    return distancias
def graficar_grafo(grafo):
    # Instancia de la clase para graficar nodos
    Grafox = nx.DiGraph()
    # Añadir los nodos y las aristas con pesos
    for nodo, vecinos in grafo.items():
```

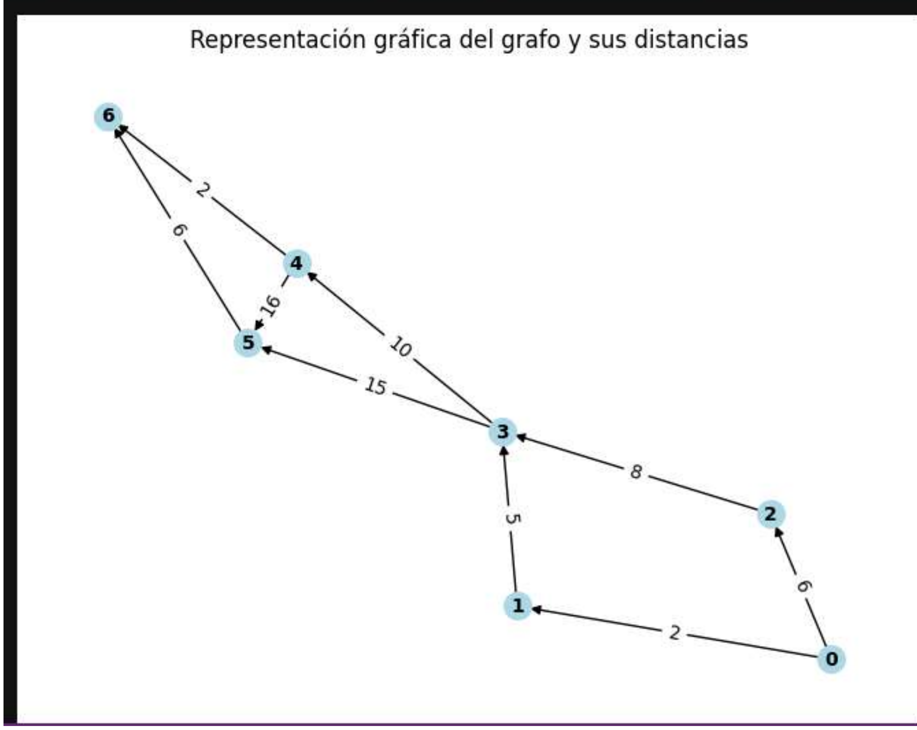
Figura 1: Codigo Pyhton “Ejercicio”

```python
    # Dibujar las etiquetas de las aristas (pesos)
    labels = nx.get_edge_attributes(Grafox, 'weight')
    nx.draw_networkx_edge_labels(Grafox, pos, edge_labels=labels, font_size=10)

    # Mostrar la gráfica
    plt.title("Representación gráfica del grafo y sus distancias")
    plt.show()
# Definir el grafo
grafo = {
    '0': {'1': 2, '2': 6},
    '1': {'3': 5},
    '2': {'3': 8},
    '3': {'5': 15, '4': 10},
    '4': {'5': 16, '6': 2},
    '5': {'6': 6},
    '6': {},
}
# Medir tiempo de ejecución de Dijkstra con timeit
tiempo_ejecucion = timeit.timeit(
    lambda: dijkstra(grafo, '0'),  # Función a medir
    number=1000  # Número de repeticiones para mayor precisión
)
# Ejecución del algoritmo Dijkstra
resultado = dijkstra(grafo, '0')
# Recorrido de la lista de resultados
print("Distancia más corta desde el nodo 0:")
for nodo, distancia in resultado.items():
    print(f"Nodo: {nodo}, distancia: {distancia}")
# Mostrar el tiempo de ejecución promedio
print(f"\nTiempo de ejecución de Dijkstra (promedio de 1000 ejecuciones): {tiempo_ejecucion / 1000:.6f} segundos")
# Gráfica
graficar_grafo(grafo)
```

Figura 2: Código Pyhton "Ejercicio"

1.1. Resultado del código

```
Tiempo de ejecución de Dijkstra (promedio de 1000 ejecuciones): 0.000006 segundos
```




```
Distancia mas corta desde el nodo 0
Nodo: 0, distancia: 0
Nodo: 1, distancia: 2
Nodo: 2, distancia: 6
Nodo: 3, distancia: 7
Nodo: 4, distancia: 17
Nodo: 5, distancia: 22
Nodo: 6, distancia: 19
```

Figura 3: visualización de la grafica

2. ALGORITMO A* (ESTRELLA)

El código proporcionado implementa el algoritmo A* (A estrella) para encontrar la ruta más corta desde un nodo origen hasta un nodo objetivo en un grafo ponderado, utilizando una heurística que puede ajustarse según el caso (en este código se define como 0 por simplicidad). Utiliza **heapq** para gestionar la cola de prioridad y explorar los nodos según la suma del costo acumulado y la estimación heurística. Además, incluye una función para graficar el grafo con **networkx** y **matplotlib**, permitiendo visualizar las conexiones, aristas y pesos entre nodos. La Figura 4 y 5 corresponde a este código, que incluye la definición del grafo, la lógica del algoritmo y la función de grafica. Por otro lado, la Figura 6 es la representación gráfica generada por el código, mostrando visualmente el grafo con sus nodos, aristas y pesos asociados. Esta gráfica ayuda a comprender la estructura del grafo y cómo se relacionan las distancias calculadas por el algoritmo. Finalmente, el tiempo de ejecución del algoritmo A* se mide utilizando **timeit**, repitiendo la ejecución 1000 veces para obtener un promedio preciso.

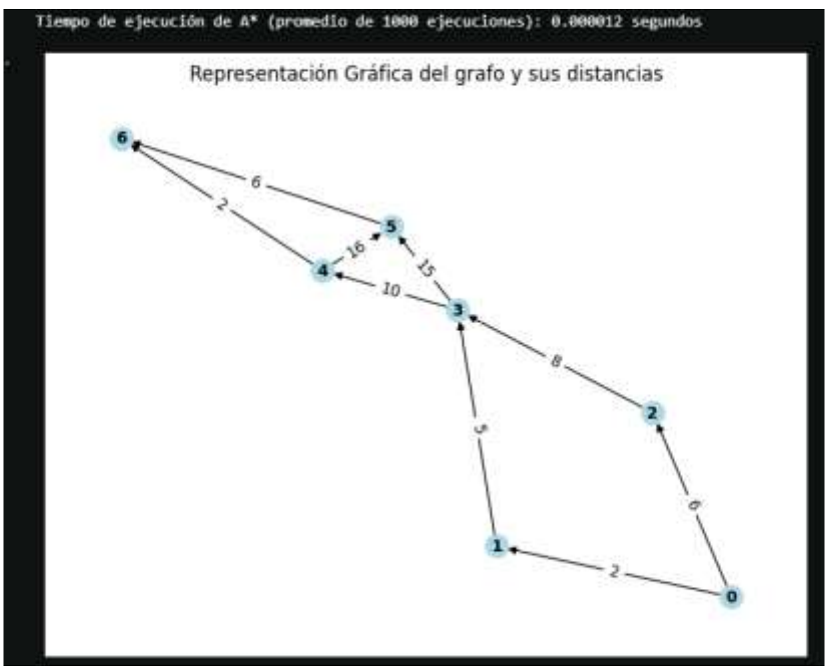
```python
import heapq
import timeit  # Usamos timeit para mayor precisión
import networkx as nx
import matplotlib.pyplot as plt
# Función heurística (en este caso, la distancia en línea recta o una estimación)
def heuristica(nodo_actual, nodo_objetivo):
    return 0  # Heurística simple
def a_estrella(grafo, nodo_origen, nodo_objetivo):
    distancias = {nodo: float('inf') for nodo in grafo}
    distancias[nodo_origen] = 0
    cola_prioridad = [(0 + heuristica(nodo_origen, nodo_objetivo), 0, nodo_origen)]
    while cola_prioridad:
        f_actual, g_actual, nodo_actual = heapq.heappop(cola_prioridad)
        if nodo_actual == nodo_objetivo:
            break
        if g_actual > distancias[nodo_actual]:
            continue
        for vecino, peso in grafo[nodo_actual].items():
            nueva_distancia = g_actual + peso
            if nueva_distancia < distancias[vecino]:
                distancias[vecino] = nueva_distancia
                f = nueva_distancia + heuristica(vecino, nodo_objetivo)
                heapq.heappush(cola_prioridad, (f, nueva_distancia, vecino))
    return distancias

def graficar_grafo(grafo):
    Grafox = nx.DiGraph()
    for nodo, vecinos in grafo.items():
        for vecino, pesos in vecinos.items():
            Grafox.add_edge(nodo, vecino, weight=pesos)
    pos = nx.spring_layout(Grafox)
    nx.draw(Grafox, pos, with_labels=True, node_color='lightblue', node_size=200,
            font_size=10, font_weight='bold')
```

Figura 4: Codigo Pyhton “Algoritmo A*”

```python
            font_size=10, font_weight='bold')
    labels = nx.get_edge_attributes(Grafox, 'weight')
    nx.draw_networkx_edge_labels(Grafox, pos, edge_labels=labels, font_size=10)
    plt.title("Representación Gráfica del grafo y sus distancias")
    plt.show()
# Definición del grafo
grafo = {
    '0': {'1': 2, '2': 6},
    '1': {'3': 5},
    '2': {'3': 8},
    '3': {'5': 15, '4': 10},
    '4': {'5': 16, '6': 2},
    '5': {'6': 6},
    '6': {},
}
# Nodo objetivo para A*
nodo_objetivo = '6'
# Medir tiempo de ejecución de A* con timeit
tiempo_ejecucion = timeit.timeit(
    lambda: a_estrella(grafo, '0', nodo_objetivo),  # Función a medir
    number=1000  # Número de repeticiones para mayor precisión
)
# Recorrido de la lista de resultados
resultado = a_estrella(grafo, '0', nodo_objetivo)
print(f"Distancia más corta desde el nodo 0 al nodo {nodo_objetivo}")
for nodo, distancia in resultado.items():
    print(f"Nodo: {nodo}, distancia: {distancia}")
# Mostrar el tiempo de ejecución promedio
print(f"\nTiempo de ejecución de A* (promedio de 1000 ejecuciones): {tiempo_ejecucion / 1000:.6f} segundos")
# Graficar el grafo
graficar_grafo(grafo)
```

Figura 5: Código Pyhton "Ejercicio"

2.1. Resultado del código




```
Distancia más corta desde el nodo 0 al nodo 6
Nodo: 0, distancia: 0
Nodo: 1, distancia: 2
Nodo: 2, distancia: 6
Nodo: 3, distancia: 7
Nodo: 4, distancia: 17
Nodo: 5, distancia: 22
Nodo: 6, distancia: 19
```

Figura 6: visualización de la grafica

3. TABLA COMPARATIVA

Comparación de tiempos de ejecución:		
Algoritmo		
Tiempo Promedio		
N#	***A****	***Dijktra***
1	*0.000012s*	*0.000009s*
2	*0.000007s*	*0.000007s*
3	*0.000006s*	*0.000008s*
4	*0.000011s*	*0.000009s*
5	*0.000013s*	*0.000010s*
6	*0.000009s*	*0.000010s*
7	*0.000010s*	*0.000013s*
8	*0.000010s*	*0.000012s*
9	*0.000013s*	*0.000011s*
10	*0.000008s*	*0.000006s*

Figura 7: visualización de la tabla de excel – comparación

4. Minimax con poda alfa-beta en un juego sencillo (Piedra-Papel-Tijera).

El código de Piedra-Papel-Tijera que implementa **Minimax con poda alfa-beta** utiliza este algoritmo para que la computadora elija siempre la mejor opción basada en la elección del jugador. En este caso, la computadora actúa como el "minimizador", buscando contrarrestar la jugada del jugador para minimizar su posibilidad de ganar. Sin embargo, debido a la simplicidad del juego, el algoritmo siempre lleva a la computadora a ganar o empatar, ya que puede predecir y responder de manera óptima a la elección del jugador. Esto hace que el juego sea predecible y poco divertido, por lo que, en la práctica, no se recomienda usar **Minimax para Piedra-Papel-Tijera.** Una alternativa más justa es que la computadora elija una opción al azar, lo que equilibra las posibilidades y hace el juego más entretenido.

Juego:

N: finaliza juego S: inicia nueva partida

Comandos:

Piedra – Papel - Tijera

```python
def evaluar(jugador, computadora):
    if jugador == computadora:
        return 0  # Empate
    elif (jugador == "piedra" and computadora == "tijera") or \
         (jugador == "papel" and computadora == "piedra") or \
         (jugador == "tijera" and computadora == "papel"):
        return 1  # Jugador gana
    else:
        return -1  # Computadora gana
# Función Minimax con poda alfa-beta
def minimax_alfa_beta(jugador, es_maximizador, alfa, beta):
    opciones = ["piedra", "papel", "tijera"]
    if es_maximizador:
        mejor_valor = -float('inf')
        mejor_opcion = ""
        for opcion in opciones:
            valor = evaluar(jugador, opcion)
            mejor_valor = max(mejor_valor, valor)
            alfa = max(alfa, mejor_valor)
            if beta <= alfa:
                break  # Poda alfa-beta
            if mejor_valor == valor:
                mejor_opcion = opcion
        return mejor_opcion
    else:
        mejor_valor = float('inf')
        mejor_opcion = ""
        for opcion in opciones:
            valor = evaluar(jugador, opcion)
            mejor_valor = min(mejor_valor, valor)
```

Figura 8: Código Pyhton "Juego"

```python
            if beta <= alfa:
                break  # Poda alfa-beta
            if mejor_valor == valor:
                mejor_opcion = opcion
        return mejor_opcion
# Juego de Piedra-Papel-Tijera
def jugar_piedra_papel_tijera():
    while True:
        print("\n--- Piedra, Papel o Tijera ---")
        jugador = input("Elige (piedra, papel, tijera): ").lower()
        if jugador not in ["piedra", "papel", "tijera"]:
            print("Opción inválida. Intenta de nuevo.")
            continue
        # Turno de la computadora (usa Minimax con poda alfa-beta)
        computadora = minimax_alfa_beta(jugador, False, -float('inf'), float('inf'))
        print(f"Computadora elige: {computadora}")
        # Determinar el resultado
        resultado = evaluar(jugador, computadora)
        if resultado == 0:
            print("¡Empate!")
        elif resultado == 1:
            print("¡Ganaste!")
        else:
            print("¡La computadora gana!")
        # Preguntar si desea jugar de nuevo
        jugar_de_nuevo = input("¿Jugar de nuevo? (s/n): ").lower()
        if jugar_de_nuevo != "s":
            break
# Iniciar el juego
jugar_piedra_papel_tijera()
```

Figura 9: Código Pyhton "Juego"

4.1.1. Resultado del código

Figura 10: Comandos para "Jugar"

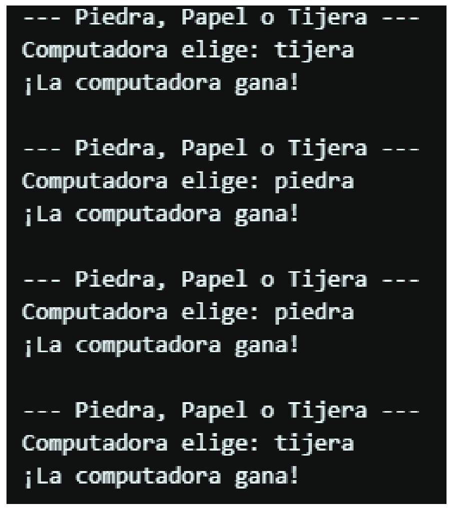


Figura 11: Resultado "Juego"

5. CONCLUSIONES

- El algoritmo de Dijkstra es un método robusto para encontrar el camino más corto en grafos con pesos no negativos. Su enfoque de exploración uniforme garantiza que siempre encuentre la solución óptima, ya que evalúa todos los nodos sin priorizar direcciones específicas. Sin embargo, esta exhaustividad lo hace menos eficiente en grafos grandes, ya que puede explorar nodos innecesarios. Cuando se mide con timeit, su tiempo de ejecución tiende a ser mayor, especialmente en comparación con algoritmos heurísticos como A*
- El algoritmo A* mejora la eficiencia de Dijkstra al incorporar una función heurística que guía la búsqueda hacia el nodo objetivo. Esta heurística permite priorizar nodos prometedores, reduciendo el espacio de búsqueda y el tiempo de ejecución. Sin embargo, su rendimiento depende de la calidad de la heurística: si no es admisible (sobreestima la distancia), A*

puede no garantizar la optimalidad. Al medir con timeit, A* suele mostrar un tiempo de ejecución menor que Dijkstra, especialmente en grafos grandes o cuando la heurística está bien diseñada.

- El algoritmo Minimax con poda alfa-beta se implementó en el código para que la computadora tome decisiones óptimas basadas en la elección del jugador. Sin embargo, en un juego simple como Piedra-Papel-Tijera, donde no hay un estado del juego que evolucione con el tiempo, este algoritmo resulta excesivo y poco práctico. La computadora siempre elige la mejor opción para contrarrestar al jugador, lo que lleva a que siempre gane o empate, eliminando el factor de aleatoriedad y diversión que caracteriza a este juego.
- Piedra-Papel-Tijera es un juego basado en la suerte y la elección aleatoria, sin un componente estratégico profundo. Aunque el código utiliza técnicas avanzadas como Minimax con poda alfa-beta, estas no son necesarias para este tipo de juego. Una implementación más adecuada y justa sería que la computadora elija una opción al azar, lo que mantendría la esencia del juego y lo haría más equilibrado y entretenido para el jugador.

6. BIBLIOGRAFIA

- https://www.geeksforgeeks.org/a-search-algorithm/
- https://docs.python.org/3/library/timeit.html
- https://aima.cs.berkeley.edu/
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/
- https://realpython.com/python-rock-paper-scissors/