Opción 4:

Disclaimer: El código para Dijsktra fue realizado en conjunto con Anthony Guimarey. El código para Vector-Distancia fue realizado únicamente por mi desde cero, no se utilizó código de otras fuentes para Vector-Distancia, a excepción de la función de Fibonacci.

Respuesta:

```python
class Router:
    def __init__(self, id):
        name_of_file = "file" + str(id) + ".txt"
        self.file = open(name_of_file,"w+")
        self.tabla = {}
        self.vecinos = []
        self.id = id
```

La clase *Router* fue creada para manejar de forma sencilla las tablas con las distancias de los otros routers. Las tablas se modifican y actualizan en tiempo real por cada operación realizada, es decir, que para cada inciso de la pregunta 2, se pueden ver cada una de las tablas de cada uno de los routers en un archivo de texto. A estos routers se les asigna la tabla que luego pasara al archivo asignado, un ID y los vecinos correspondientes para cada topología.

```python
def add_vecinos(self, r1, r2):
    self.vecinos.insert(0,r1)
    self.vecinos.insert(1,r2)
    if self.id != 0:
        self.tabla[r1.id][self.id] = Fibonacci(self.id+2)
        self.tabla[r2.id][self.id] = Fibonacci(self.id+3)
    else:
        self.tabla[r1.id][self.id] = Fibonacci(r1.id+3)
        self.tabla[r2.id][self.id] = Fibonacci(self.id+3)

    with open(self.file.name, "w+") as f:
        for k in range(0, len(self.tabla)):
            f.write(str(self.tabla[k]) + "\n")
    f.close()
```

```python
def fill_file(self):

    for j in range(0, 9):
        self.tabla[j] = []
        for k in range(0,9):
            self.tabla[j].insert(j,pow(2,30))
    self.tabla[self.id][self.id] = 0

    with open(self.file.name, "w+") as f:
        for k in range(0, len(self.tabla)):
            f.write(str(self.tabla[k]) + "\n")
    f.close()
```

La función *add_vecinos* se encarga de enlazar a los nodos adyacentes según la estructura correspondiente, así mismo se asigna el peso de dicho enlace el cual está determinado por el *id* del router y el número de Fibonacci a partir del tercer número (el segundo uno en la serie). La función *fill_file* llena el archivo con el máximo valor entero posible en Python, para representar el “infinito”, ya que luego se buscan los caminos con menor peso, además la función también coloca los ceros en la posición en la que los routers se apuntan a si mismos. Cabe mencionar que las tablas son diccionarios, las llaves son los *id*s de los routers y el contenido son arrays que almacenan los pesos con respecto al router que se esté usando como llave.

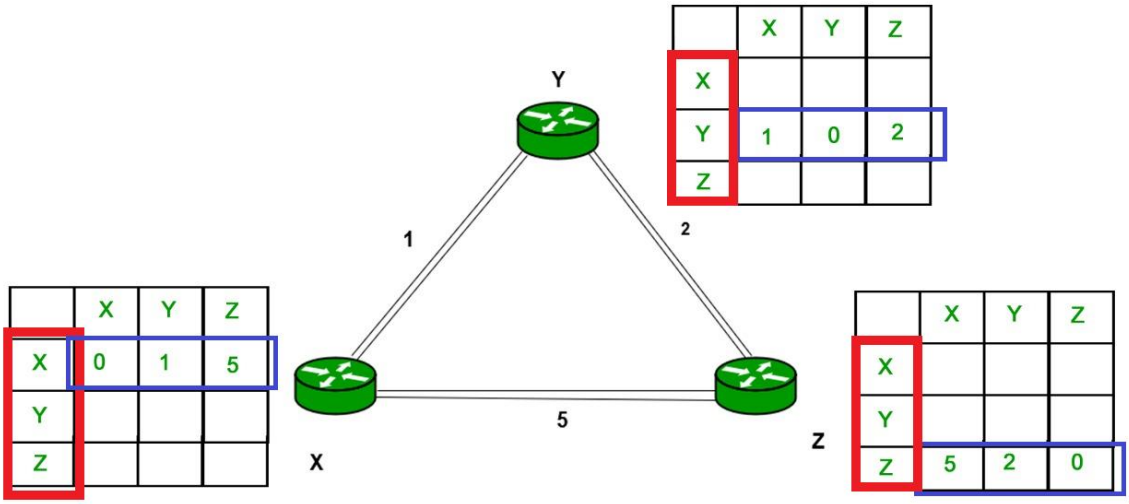


Imagen sacada de GeekforGeeks (adaptación propia)

En la imagen se puede ver en rojo lo que se usan como llaves de los diccionarios y en azul los arrays que están vinculados a cada llave (hay un array por llave, y los espacios en blanco se llenan con el máximo valor entero posible con la función *fill_file*).

La función actualizar es la más importante, ya que se encarga de mantener las tablas de los routers en sincronía. Cuando un router ejecuta esta función, esta compara los valores de su tabla con la de uno de sus vecinos (*vecino[0]*), luego se actualizan ambas tablas para tener exactamente la misma información. Luego el router busca a su otro vecino (*vecino[1]*) y realiza el mismo proceso. Al terminar, nuevamente regresa con *vecino[0]*, para que los 3 tengan la misma información optimizada.

```
def actualizar(self):

    r1 = self.vecinos[0]
    r2 = self.vecinos[1]

    for x in r1.tabla:
        for i in range(0, len(r1.tabla)):
            self.tabla[i][x] = min(self.tabla[i][x
            r1.tabla[i][x] = self.tabla[i][x]

    for x in r2.tabla:
        for i in range(0, len(r2.tabla)):
            self.tabla[i][x] = min(self.tabla[i][x
            r2.tabla[i][x] = self.tabla[i][x]

    for x in r1.tabla:
        for i in range(0, len(r1.tabla)):
            self.tabla[i][x] = min(self.tabla[i][x
            r1.tabla[i][x] = self.tabla[i][x]

    with open(self.file.name, "w+") as f:
        for j in range(0, len(self.tabla)):
            f.write(str(self.tabla[j]) + "\n")
```

Luego de esto, se abren los archivos de cada router visto anteriormente y se escribe la tabla, reemplazando todo lo que tuviese contenido.

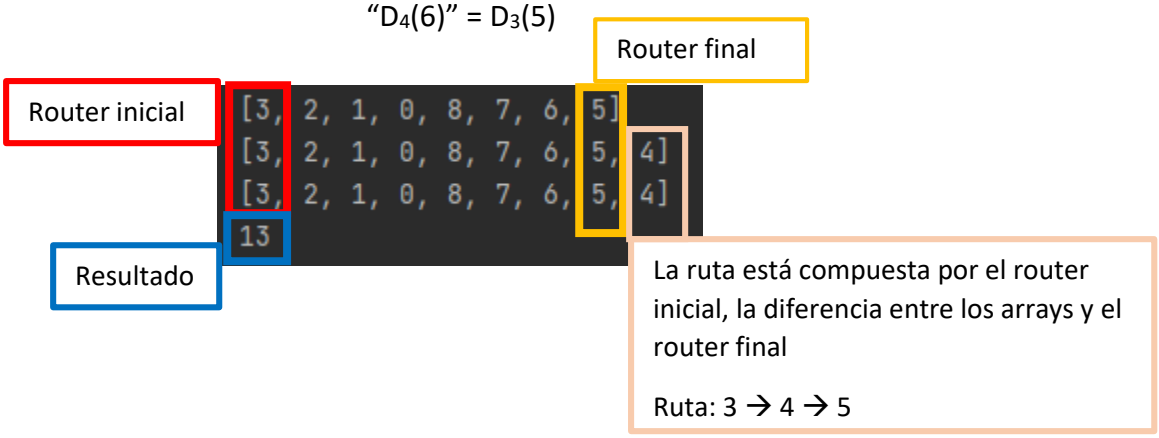
```python
def DistVectorial(src,dest,path):
    if (src.id in path):
        return 0
    if src.id == dest.id:
        path.append(src.id)
        return 0
    else:
        path.append(src.id)
        DistVectorial(src.vecinos[0], dest, path)
        DistVectorial(src.vecinos[1], dest, path)
        src.actualizar()
        resultado = src.tabla[dest.id][src.id]
        print(path)
        return resultado
```

Todas las funciones anteriores eran parte de la clase *Router*, pero esta no. La función *DistVectorial* es la que se encarga de resolver la pregunta 2. Como Python tiene un límite de profundidad para la recursividad, tuve que cambiar un poco el cómo funciona esta función a comparación de una red de verdad con un sistema de Distance Vector Routing. En este caso *DistVectorial* necesita un inicio, un destino y un array que servirá para que no repita el camino por el que ya pasó y entre en bucle. Primero busca si el inicio está dentro del *path,* si es así solo retorna un cero y ahí muere esa rama, luego compara si el inicio es el mismo que el destino, si es así, la rama también se detiene y retorna cero. Por último, en caso nada de lo anterior ocurriera, se agrega el router actual para agregarlo a la lista del *path*, luego se ejecuta la misma función, pero con sus vecinos. El detalle es la función *actualizar*, ya que cada vez que se llega a un router antes del destino, las tablas en los nodos más alejados al router inicial y a su vez más cercanos al router destino se actualizan y modifican, y a medida que regresan de la recursividad, van cambiando las tablas de sus vecinos y van sumando los pesos por los caminos por los que pasaron. Estos pesos se van comparando y solo se queda el menor, igual que en la fórmula original, para al final tener el resultado y el camino, aunque el camino terminó quedando de una forma poco convencional. Veamos un ejemplo usando ya los ejercicios de la pregunta 2:

$$\text{Calcule } D_1(9),\ D_4(6),\ D_3(7)\ D_5(6)$$

En el código tuve que usar los valores desde 0 hasta 8, así que restaré 1 a los valores del problema para la ejecución y que sea el resultado correcto:

"$D_4(6)$" = $D_3(5)$



Este es el formato que tiene el output para los ejercicios.

"$D_1(9)$" = $D_0(8)$

```
[0, 8, 1, 2, 3, 4, 5, 6, 7]
[0, 8, 1, 2, 3, 4, 5, 6, 7]
55
```

Aquí como la ruta solo son 2 elementos, no hay una diferencia entre los arrays

Ruta: 0 → 8

"$D_3(7)$" = $D_2(6)$

```
[2, 1, 0, 8, 7, 6]
[2, 1, 0, 8, 7, 6, 3, 4, 5]
[2, 1, 0, 8, 7, 6, 3, 4, 5]
[2, 1, 0, 8, 7, 6, 3, 4, 5]
[2, 1, 0, 8, 7, 6, 3, 4, 5]
29
```

Ruta: 2→ 3 → 4 → 5 → 6

"$D_5(6)$" = $D_4(5)$

```
[4, 3, 2, 1, 0, 8, 7, 6, 5]
[4, 3, 2, 1, 0, 8, 7, 6, 5]
8
```

Ruta: 4 → 5

Para Dijsktra:

```
def imprimirCamino(verticeActual, ancestros, caminoString):
    if verticeActual == -1:
        return caminoString
    caminoString = imprimirCamino(ancestros[verticeActual], ancestros, caminoString)
    caminoString += str(verticeActual) + ' -> '
    return caminoString
```

La función *imprimirCamino* forma parte del formato para el output, la variable *caminoString* guarda path que usan el vértice inicial, en este caso como variable *verticeActual,* para llegar desde este punto a cualquier otro.

```
def imprimirSolucion(verticeInicial, distancias, ancestros):
    numeroVertices = len(distancias)
    print('-' * 60)
    print('|{0: ^13}|{1: ^13}|{2: ^30}|'.format('Vértice', 'Distancia', 'Camino'))
    print('-' * 60, end='')
    for indiceVertice in range(numeroVertices):
        if indiceVertice != verticeInicial:
            verticeString = str(verticeInicial) + ' -> ' + str(indiceVertice)
            distanciaString = str(distancias[indiceVertice])
            caminoString = ' '
            caminoString = imprimirCamino(indiceVertice, ancestros, caminoString)
            print()
            print('|{0: ^13}|{1: ^13}|{2: <30}|'.format(verticeString, distanciaS
    print()
    print('-' * 60, end='')
```

Ahora, para el output como tal, se emplea la función *imprimirSolucion* que usa la función anterior para mostrar el resultado que estamos buscando para la pregunta 2. Esta consiste de un *verticeInicial,* que es como nuestro router inicial, las distancias que es un array que las contiene, y los ancestros son los routers previamente usados que son parte del path.

```python
def dijkstra(matriz, verticeInicial):
    numeroVertices = len(matriz[0])
    distanciasMenores = [float('inf')] * numeroVertices
    agregados = [False] * numeroVertices

    for indiceVertice in range(numeroVertices):
        distanciasMenores[indiceVertice] = float('inf')
        agregados[indiceVertice] = False

    distanciasMenores[verticeInicial] = 0
    ancestros = [-1] * numeroVertices
    ancestros[verticeInicial] = -1

    for i in range(1, numeroVertices):
        verticeCercano = -1
        distanciaMinima = float('inf')
        for indiceVertice in range(numeroVertices):
            if not agregados[indiceVertice] and distancias
                verticeCercano = indiceVertice
                distanciaMinima = distanciasMenores[indice
```

Y para terminar, el núcleo del programa, el algoritmo de dijkstra. Este requiere de la matriz de adyacencia en la cual están las relaciones entre los routers con sus respectivos pesos. Con la matriz de adyacencia es posible generar las distancias menores de acuerdo a un vértice inicial. De la misma forma que con Vector Distancia aquí también se inicia la enumeración de lo routers con 0 y culmina en 8. Con todas estas funciones, el output quedaría de la siguiente manera:

$$\text{Calcule } D_1(9),\ D_4(6),\ D_3(7)\ D_5(6)$$

"$D_1(9)$" = $D_0(8)$

Vértice	Distancia	Camino
0 -> 1	1	0 -> 1 ->
0 -> 2	3	0 -> 1 -> 2 ->
0 -> 3	6	0 -> 1 -> 2 -> 3 ->
0 -> 4	11	0 -> 1 -> 2 -> 3 -> 4 ->
0 -> 5	19	0 -> 1 -> 2 -> 3 -> 4 -> 5 ->
0 -> 6	32	0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 ->
0 -> 7	53	0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 ->
0 -> 8	55	0 -> 8 ->

“$D_4(6)$” = $D_3(5)$

Vértice	Distancia	Camino
3 -> 0	6	3 -> 2 -> 1 -> 0 ->
3 -> 1	5	3 -> 2 -> 1 ->
3 -> 2	3	3 -> 2 ->
3 -> 4	5	3 -> 4 ->
3 -> 5	13	3 -> 4 -> 5 ->
3 -> 6	26	3 -> 4 -> 5 -> 6 ->
3 -> 7	47	3 -> 4 -> 5 -> 6 -> 7 ->
3 -> 8	61	3 -> 2 -> 1 -> 0 -> 8 ->

“$D_3(7)$” = $D_2(6)$

Vértice	Distancia	Camino
2 -> 0	3	2 -> 1 -> 0 ->
2 -> 1	2	2 -> 1 ->
2 -> 3	3	2 -> 3 ->
2 -> 4	8	2 -> 3 -> 4 ->
2 -> 5	16	2 -> 3 -> 4 -> 5 ->
2 -> 6	29	2 -> 3 -> 4 -> 5 -> 6 ->
2 -> 7	50	2 -> 3 -> 4 -> 5 -> 6 -> 7 ->
2 -> 8	58	2 -> 1 -> 0 -> 8 ->

“$D_5(6)$” = $D_4(5)$

Vértice	Distancia	Camino
4 -> 0	11	4 -> 3 -> 2 -> 1 -> 0 ->
4 -> 1	10	4 -> 3 -> 2 -> 1 ->
4 -> 2	8	4 -> 3 -> 2 ->
4 -> 3	5	4 -> 3 ->
4 -> 5	8	4 -> 5 ->
4 -> 6	21	4 -> 5 -> 6 ->
4 -> 7	42	4 -> 5 -> 6 -> 7 ->
4 -> 8	66	4 -> 3 -> 2 -> 1 -> 0 -> 8 ->

Luego de obtener los resultados, para ambos métodos podemos ver que el resultado es el mismo. Esto tal vez se deba a que ambos tienen una matriz de adyacencia, o por lo menos conocen los valores de sus vecinos. Sin embargo, en Vector-Distancia se requiere que los routers actualicen “manualmente” sus tablas para conocer la red en su totalidad y esto requiere que realicen operaciones (al menos así fue planteado en el código). Para Dijkstra, no es necesario esto ya que se trabaja con una matriz de adyacencia “global”, entonces los nodos siempre saben a cuánto costaría ir de un router a otro, y aunque esto ahorre tiempo a la hora de ejecutar en su totalidad ambos métodos (ya que un router no tiene que ir descubriendo la red, sino que ya está “descubierta” y lista para trabajar sobre ella), la consecuencia sería que esta matriz no se puede editar. Por otro lado, como cada router en Vector-Distancia posee su propia tabla, esta se puede cambiar y mientras más procesos de actualización se realicen, más rápido es el cálculo de peso desde un router cualquiera a otro, ya que cada router luego de cierta cantidad de hops (dependiendo de la topología) conocen toda la red.

Repositorio de Github: https://github.com/LuisUTEC/Redes_Final