Distance Vector: simulazione

Obiettivo del progetto

L'obiettivo del progetto é la creazione di uno script in Python che permetta di simulare l'aggiornamento dinamico delle tabelle di routing dei nodi all'interno di un grafo. L'aggiornamento delle tabelle avviene tramite distance vector, una metodologia in cui ogni nodo della rete condivide periodicamente le proprie informazioni di routing con i propri vicini fino al raggiungimento della convergenza, con la conseguente finalizzazione delle tabelle di routing per i singoli nodi.

Script

1. **Utilizzo**

   E' possibile eseguire lo script nel seguente modo:

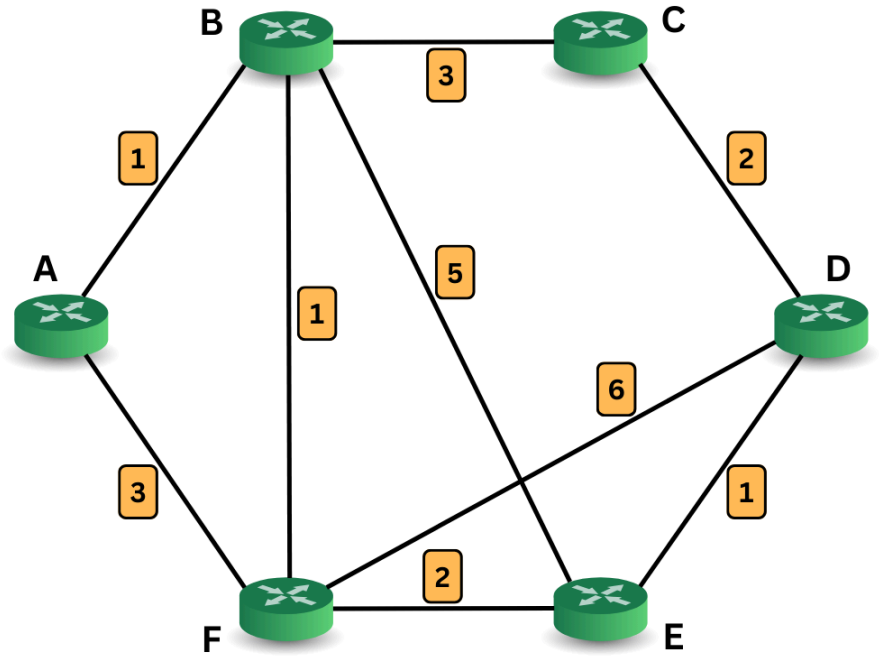
   > *Nota: é necessaria una versione di Python superiore o uguale alla 3.6*

   ```
   python3 distance_vector.py
   ```

   A questo punto verrà chiesto all'utente se egli vuole utilizzare un grafo già configurato o se ne vuole creare uno inserendo i singoli archi che collegano due nodi e il relativo peso.

   Inserire `0` per utilizzare il grafo preconfigurato, `1` per creare un nuovo grafo.

   Il grafo preconfigurato é il seguente:



Nel caso in cui si scelga di creare un grafo, lo script chiederà in input gli archi del grafo, inseribili nel formato `NomeNodo1 NomeNodo2 CostoArco`.

> *I nomi dei nodi sono case sensitive, quindi il nodo a ed il nodo A sono contati come nodi separati.*
>
> *Il grafo non é orientato: l'aggiunta di un arco N1→N2 presuppone che esista anche il senso di percorrenza N2→N1*

Una volta selezionato il grafo verrà chiesto all'utente se egli vuole vedere le tabelle di routing dei nodi ad ogni loro aggiornamento. Nel caso venga inserito `0`, per ogni nodo `X` verrà visualizzato il distance vector inviato ai nodi vicini e il risultato dell'aggiornamento delle tabelle di routing di quest'ultimi. Se viene inserito `1` vengono visualizzate solo le tabelle di routing finali dopo la convergenza.

2. **Analisi del codice**

   Nell'analisi del codice verranno omesse le porzioni di codice dedicate alla stampa del distance vector, la formattazione delle stampe e la configurazione dell'utente per mantenere l'attenzione sull'aspetto logico del programma.

Classe DistanceVector

```python
class DistanceVector:
    def __init__(self, name: str):
        self.name = name
        self.distance_vector = {
            name: (0, "")
        }  # Maps the destination to (cost, nextHop)

    def update_distance_vector(self, other: "DistanceVector", weight: int) -> bool:
        updated = False
        for dest, data in other.distance_vector.items():
            cost = data[0]
            if (
                dest not in self.distance_vector
                or cost + weight < self.distance_vector[dest][0]
            ):
                self.distance_vector[dest] = (cost + weight, other.name)
                updated = True
        return updated
```

La classe `DistanceVector` modella un distance vector del nodo `name`.
Nell'inizializzazione del distance vector l'unica informazione disponibile é la presenza del nodo `name` stesso come destinazione, alla quale é associato un costo e next hop (prossimo nodo da visitare per arrivare alla destinazione nel percorso di costo minimo) nulli.

Sia il `X` il nodo corrente e `Y` un nodo adiacente a `X`. Il metodo `update_distance_vector` aggiorna le informazioni della routing table di `X` con `other`, il distance vector di `Y`. Per ogni destinazione in `other` si controlla se quest'ultima non é presente nel distance vector di `X` oppure se, per arrivare alla destinazione, in `other` sia descritto un costo minore di quello conosciuto tenendo in considerazione anche il costo dell'arco che collega `X` ed `Y`; in entrambi i casi, si aggiunge la rotta col costo aggiornato. Il metodo `update_distance_vector` ritorna `True` se una qualsiasi rotta nel distance vector corrente (di `X`) é stata aggiornata: questo é utile per capire se il grafo é arrivato a convergenza.

Classe Graph

```python
class Graph:
    def __init__(self):
        # Maps the node_name to a list((destination, edge_weight))
        self.edges = {}
        # Maps the node_name to a DistanceVector
        self.vectors = {}

    def add_node(self, node: str):
        if node not in self.edges:
            self.edges[node] = []
            self.vectors[node] = DistanceVector(node)

    def add_edge(self, src: str, dst: str, weight: int):
        self.add_node(src)
        self.add_node(dst)
        self.edges[src].append((dst, weight))
        self.edges[dst].append((src, weight))

    def get_nodes(self) -> list[str]:
        return list(self.edges.keys())

    def get_edges_from(self, src: str) -> list[(str, int)]:
        return self.edges[src][:]
```

La classe `Graph` modella un grafo non orientato in cui ogni nodo viene associato ad un `DistanceVector`. Per ogni nodo viene mantenuta anche l'informazione degli archi che partono da quest'ultimo in una lista contenente delle coppie `(destination, edge_weight)`. Rispettivamente, gli archi vengono associati al nome del nodo in una mappa `edges` e i distance vector in una mappa `vectors`. Il metodo `add_edge` é il fulcro della classe: ad ogni chiamata vengono inseriti i nodi `src` e `dst` (se non già presenti) e creato il relativo arco a doppia percorrenza (viene creato sia `src`→`dst` sia `dst`→`src`).

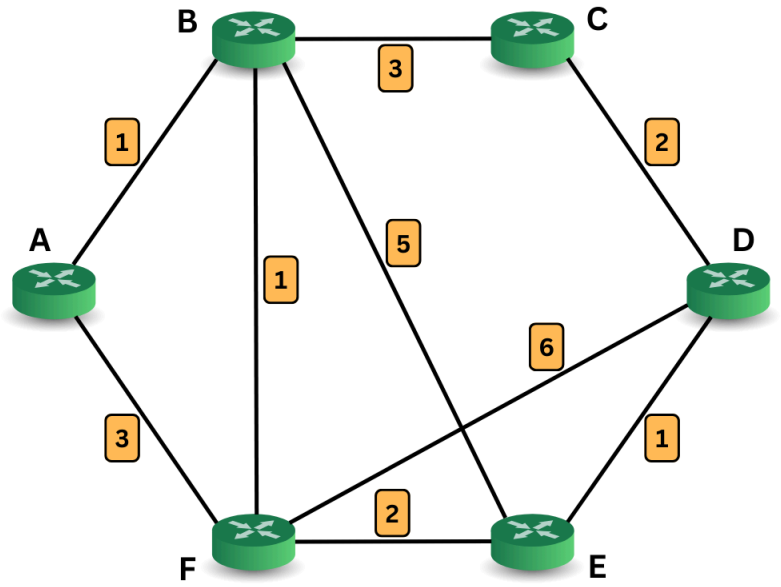
Main

```
while True:
    changes = False
    for src in g.get_nodes():
        src_dv = g.vectors[src]
        # share the distance vector with all the adjacent nodes
        for edge in g.get_edges_from(src):
            dst, weight = edge
            dst_dv = g.vectors[dst]
            if dst_dv.update_distance_vector(src_dv, weight):
                # another full cycle must be done
                changes = True
    # stable configuration reached
    if not changes:
        break
for node in g.vectors:
    g.vectors[node].print_routing_table()
```

Questo é l'unico punto logico all'interno del main: l'effettivo scambio di distance vector tra nodi adiacenti (non viene qui riportata tutta la parte di configurazione utente o delle stampe opzionali ad ogni aggiornamento). Si opera nel seguente modo: per ogni nodo `src` all'interno del grafo se ne estrapola il distance vector `src_dv`. Per ogni nodo `dst` adiacente a `src` si prende il suo distance vector `dst_dv` e si aggiorna con le informazioni contenute in `src_dv` (vedere la sezione precedente per capire come opera `update_distance_vector`). A questo punto, se un qualsiasi distance vector `dst_dv` di un nodo adiacente a `src` riporta una modifica é necessario ripetere l'intero ciclo di aggiornamenti per ogni possibile nodo poiché non si é arrivati a convergenza (é possibile che il nodo adiacente appena aggiornato possa contenere informazioni utili per aggiornare nodi a sua volta adiacenti), si salva quindi la presenza di cambiamenti in uno/più distance vector in `changes`. Quando la rete arriverà a convergenza (cioé quando tutti i nodi raggiungeranno una configurazione stabile) allora `changes` varrà `False` poiché nessun nodo avrà ricevuto più aggiornamenti utili da nodi vicini, si smette dunque di inviare distance vector ai nodi adiacenti e si procede con la stampa delle tabelle di routing finali dei nodi. (`print_routing_table()` effettua la stampa della singola routing table)

Esempio con routing table finali

Per l'esempio prendiamo in considerazione il grafo preconfigurato nello script:



L'output delle tabelle finali dato dallo script é il seguente:

[A] Routing table

Destination	Cost	Next Hop
A	0	
B	1	B
F	2	B
E	4	B
C	4	B
D	5	B

[E] Routing table

Destination	Cost	Next Hop
E	0	
B	3	F
A	4	F
F	2	F
D	1	D
C	3	D

[B] Routing table

Destination	Cost	Next Hop
B	0	
A	1	A
F	1	F
E	3	F
C	3	C
D	4	F

[D] Routing table

Destination	Cost	Next Hop
D	0	
F	3	E
A	5	E
B	4	E
E	1	E
C	2	C

[F] Routing table

Destination	Cost	Next Hop
F	0	
A	2	B
B	1	B
E	2	E
D	3	E
C	4	B

[C] Routing table

Destination	Cost	Next Hop
C	0	
B	3	B
A	4	B
D	2	D
F	4	B
E	3	D