Distance Vector Routing

Funzionamento del sistema

Il sistema si compone di un solo file, chiamato Distance_Vector_Routing, scritto in linguaggio Python. Il file si può dividere in tre macroaree:

1. La classe Node;
2. La classe Network;
3. La creazione della rete, dei nodi, dei collegamenti e la parte di esecuzione dello script.

Lo script ha come obiettivo quello di implementare correttamente il protocollo Distance Vector Routing (DVR), comprendente la logica di aggiornamento delle rotte, di gestione delle tabelle di routing e del calcolo delle distanze tra i nodi.

Proseguendo nella spiegazione del funzionamento del sistema, concentriamoci adesso sulla classe Node. Questa ha il compito di rappresentare un nodo qualsiasi della rete e di fornire anche i metodi per gestirlo. I metodi che implementa sono 4: __init__, add_neighbor, update_routing_table e print_routing_table. Il primo è il costruttore del nodo e si occupa sostanzialmente di crearlo assegnandoli un nome passato come parametro, di creare un dizionario vuoto per i nodi vicini e di inserire nella sua tabella di routing il nodo stesso con costo di raggiungimento pari a 0.

```python
#Inizializza un nodo con un nome che deve essere specificato
def __init__(self, name):
    #Assegna il nome al nodo corrente
    self.name = name
    #Dizionario dei nodi vicini e del costo per raggiungerli
    self.neighbors = {}
    #Inizializza la tabella di routing nel seguente modo {destinazione: (costo, next hop)}
    #All'inizio la tabella contiene solo il nodo stesso con costo 0
    self.routing_table = {name: (0, name)}
```

Metodo __init__ della classe Node

Il metodo add_neighbor, invece, deve creare un collegamento tra il nodo su cui viene chiamato e il nodo che gli viene passato come parametro, assegnandogli il costo passato come secondo parametro di input. Realizza questo aggiungendo il costo passato nel dizionario dei vicini e poi inserisce la coppia costo-nome vicino nella tabella di routing.

```python
#Aggiunge un vicino con il costo associato
def add_neighbor(self, neighbor, cost):
    #Aggiorna i vicini con il costo
    self.neighbors[neighbor] = cost
    #Aggiorna la tabella di routing inserendo il nodo vicino e il costo per raggiungerlo
    self.routing_table[neighbor.name] = (cost, neighbor.name)
```

Metodo add_neighbor della classe Node

Il terzo metodo, si deve occupare dell'aggiornamento della tabella di routing di ogni singolo nodo. Per fare questo implementa un doppio ciclo for, il primo sul dizionario dei vicini del nodo che stiamo considerando, mentre il secondo viene eseguito sul contenuto della tabella di routing di un nodo vicino. Inoltre, calcola il nuovo costo della rotta come il costo per arrivare al nodo (cost_to_neighbor) sommato a quello per raggiungere il vicino (neighbor_cost). Successivamente esegue un controllo per verificare se il vicino è contenuto o meno nella tabella di routing del nodo in questione. Se non è presente, allora il nodo vicino viene aggiunto con il costo calcolato precedentemente, altrimenti, controlla se il nuovo costo calcolato è strettamente minore di quello già contenuto nella tabella. In caso affermativo, aggiorna il percorso con la nuova distanza e il nuovo nodo successivo nel percorso.

```python
#Aggiorna la tabella di routing basandosi sulle tabelle dei vicini
def update_routing_table(self):
    for neighbor, cost_to_neighbor in self.neighbors.items():
        #Scansiona la tabella di routing del vicino
        for destination, (neighbor_cost, next_hop) in neighbor.routing_table.items():
            #Calcola il costo per raggiungere la destinazione tramite il vicino
            new_cost = cost_to_neighbor + neighbor_cost
            #Aggiunge o aggiorna la rotta nella tabella di routing
            if(destination not in self.routing_table or
               new_cost < self.routing_table[destination][0]):
                self.routing_table[destination] = (new_cost, neighbor.name)
```

Metodo update_routing_table della classe Node

L'ultimo metodo della classe Node ha il compito di stampare la tabella di routing di un singolo nodo della rete con la destinazione, il costo per raggiungerla e il next hop del percorso. In particolare il metodo print(), il quale si occupa di stampare del testo inserito tra doppi apici, permette di anteporre un carettere "f" prima dei doppi apici. In questo modo si possono inserire valori di variabili o espressioni, racchiuse tra parentesi graffe, che verranno valutate e sostituite con il loro valore direttamente all'interno della stringa.

```python
#Stampa la tabella di routing di un singolo nodo mostrando
#il costo e il next hop per raggiungere ogni destinazione
def print_routing_table(self):
    print(f"Routing table for {self.name}:")
    for destination, (cost, next_hop) in sorted(self.routing_table.items()):
        print(f"  To {destination}: Cost = {cost}; Next hop = {next_hop}")
    print()
```

Metodo print_routing_table della classe Node

Ora, invece, occupiamoci del contenuto della classe Network. Questa, che deve gestire l'intera rete, si compone di 5 metodi: __init__, add_node, add_link, run_distance_vector_routing e print_all_routinf_tables. Il primo metodo è il costruttore della rete e inizializza un dizionario vuoto che andrà a contenere tutti i nodi della rete.

```python
#Inizializza un dizionario che rappresenta una rete vuota
def __init__(self):
    self.nodes = {}
```

Metodo __init__ della classe Network

Il metodo add_node si occupa di aggiungere un nodo alla rete, dopo averlo creato tramite la chiamata al costruttore della classe Node e avergli passato come parametro il nome assegnato al nodo. Ovviamente, lo crea e lo aggiunge al dizionario solo se non ne esiste già uno con lo stesso nome.

```python
#Aggiunge un nodo alla rete solo se uno con il suo stesso nome non è già presente
def add_node(self, name):
    if name not in self.nodes:
        #Chiama automaticamente il metodo __init__ della classe Node
        self.nodes[name] = Node(name)
```

Metodo add_node della classe Network

Il terzo metodo di questa classe ha il compito di creare un collegamento tra una coppia di nodi passati come parametri. In realtà, vengono passati come input i nomi di due nodi e poi si accede al dizionario dei nodi della rete per estrarre le istanze dei nodi. Queste istanze verranno passate al metodo add_neighbor appartenente alla classe Node per costruire effettivamente il collegamento. Visto che il metodo viene chiamato sia sul primo nodo in input sia sul secondo, il collegamento che si andrà a creare sarà bidirezionale.

```python
#Aggiunge un collegamento tra 2 nodi con il costo associato
def add_link(self, node1_name, node2_name, cost):
    if node1_name in self.nodes and node2_name in self.nodes:
        node1 = self.nodes[node1_name]
        node2 = self.nodes[node2_name]
        node1.add_neighbor(node2, cost)
        node2.add_neighbor(node1, cost)
```

Metodo add_link della classe Network

Il metodo run_distance_vector_routing è il metodo che fa partire effettivamente il protocollo DVR perchè, ad ogni iterazione del ciclo for più esterno, chiama tutti i nodi che compongono la rete e su ognuno esegue l'aggiornamento della tabella di routing. L'aggiornamento della tabella viene svolto dal metodo update_routing_table contenuto nella classe Node.

```python
#Esegue il protocollo DVR, per un massimo di iterazioni specificato
def run_distance_vector_routing(self, iterations):
    for i in range(iterations):
        for node in self.nodes.values():
            #Chiama su un nodo il metodo per aggiornare la sua tabella di routing
            node.update_routing_table()
```

Metodo run_distance_vector_routing della classe Network

Infine, l'ultimo metodo della classe Network si occupa semplicemente di stampare tutte le tabelle di routing richiamando, per ogni nodo della rete, il metodo print_routing_table della classe Node.

```python
#Stampa le tabelle di routing di tutti i nodi della rete
def print_all_routing_tables(self):
    for node in self.nodes.values():
        node.print_routing_table()
```

Metodo print_all_routing_tables della classe Network

Considerazioni aggiuntive

Un punto da sottolineare è l'utilizzo all'interno del file dell'elemento "self". Nel linguaggio Python, questo elemento è molto importante perché rappresenta l'istanza corrente di una classe. Infatti, tramite questo ci possiamo riferire in modo chiaro all'oggetto su cui stiamo lavorando e ci permette anche di accedere ai suoi attributi e metodi. Tutti i metodi di una classe hanno come primo parametro in ingresso proprio l'elemento "self", e questo permette al metodo di sapere su quale istanza si sta lavorando. Nonostante sia un parametro di input, quando chiamo i metodi l'oggetto a cui si deve riferire l'elemento non viene passato, perché automaticamente riconosce l'oggetto su cui lo chiamo e utilizza quello come istanza su cui eseguirlo.

Conclusioni

Da riga 81 a riga 108 sono presenti le istruzioni che svolgono le seguenti operazioni: creare la rete, i nodi che la compongono e gli archi che li collegano con i pesi associati. La rete di esempio, rappresentata nella figura successiva, è composta da 5 nodi e da 6 archi.

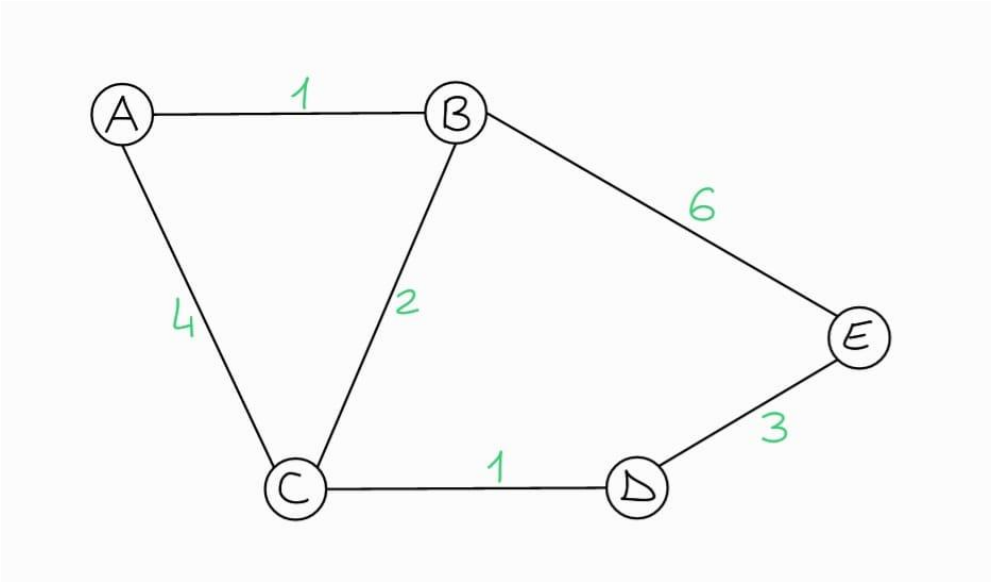

Immagine del grafo della rete di esempio

L'istruzione a riga 111 è quella che fa eseguire il protocollo Distance Vector Routing sulla rete per un numero massimo di iterazioni specificato. Nell'esempio, il numero di iterazioni è stato fissato a 10 che, grazie alle prove effettuate, siamo sicuri porti l'algoritmo a convergenza con la rete di esempio. Infine, l'istruzione a riga 114 è quella che si occupa di stampare le tabelle di routing di ciascun nodo della rete, le quali sono state calcolate grazie alla chiamata al metodo dell'istruzione precedente.
Quindi, alla pressione del pulsante "Run" con la configurazione di default, otterremo l'output seguente.

```
Routing table for A:
  To A: Cost = 0; Next hop = A
  To B: Cost = 1; Next hop = B
  To C: Cost = 3; Next hop = B
  To D: Cost = 4; Next hop = B
  To E: Cost = 7; Next hop = B

Routing table for B:
  To A: Cost = 1; Next hop = A
  To B: Cost = 0; Next hop = B
  To C: Cost = 2; Next hop = C
  To D: Cost = 3; Next hop = C
  To E: Cost = 6; Next hop = E

Routing table for C:
  To A: Cost = 3; Next hop = B
  To B: Cost = 2; Next hop = B
  To C: Cost = 0; Next hop = C
  To D: Cost = 1; Next hop = D
  To E: Cost = 4; Next hop = D

Routing table for D:
  To A: Cost = 4; Next hop = C
  To B: Cost = 3; Next hop = C
  To C: Cost = 1; Next hop = C
  To D: Cost = 0; Next hop = D
  To E: Cost = 3; Next hop = E

Routing table for E:
  To A: Cost = 7; Next hop = B
  To B: Cost = 6; Next hop = B
  To C: Cost = 4; Next hop = D
  To D: Cost = 3; Next hop = D
  To E: Cost = 0; Next hop = E
```

Output dello script con la rete di esempio