Elaborato Reti di Telecomunicazioni

Samuele Casadei, 0001097772

Dicembre 2024

Indice

0.1 Introduzione

La tipologia di elaborato scelta è l'implementazione del protocollo RIP **Distance Vector** tramite uno script in Python.

0.2 Spiegazione del protocollo

Il protocollo Distance Vector è un metodo di routing, ampiamente utilizzato per ottenere il percorso ottimale tra i nodi di una rete. Ogni nodo della rete possiede una tabella di routing contenente informazioni relative alla distanza del nodo da ogni altro nodo della rete. Ogni nodo parte avendo solamente la distanza dai nodi adiacenti (detti neighbors) e, ad ogni iterazione, informa i suoi vicini fornendo il contenuto della propria tabella di routing. Il vicino che riceve le informazioni, in base al contenuto della propria tabella, decide se aggiornare i percorsi (se più convenienti o non ancora conosciuti) e procede ad inviare la propria tabella. Nella pratica, lo scambio di informazioni avviene ad intervalli regolari, in genere di 30 secondi; tuttavia, nell'elaborato, ho eliminato l'intervallo di attesa per rendere l'algoritmo più pratico da verificare.

0.3 Definizione delle Strutture Dati

La struttura dati chiave dell'algoritmo è la classe Node, Node rappresenta il singolo nodo di commutazione all'interno della rete. Ciascun nodo dispone di

un nome, acquisito al momento della costruzione (per comodità userò delle lettere maiuscole), un dictionary **routingTable** le cui entry sono strutturate come segue: (destinazione, (distanza, prossimo nodo));
e, infine, un altro dictionary **neighbors**, le cui entry sono strutturate come: (nodo adiacente, distanza).
Notare che **neighbors** parte inizialmente vuoto, dato che le connessioni tra i nodi non vengono immediatamente definite; la topologia è generata in modo "casuale" ad ogni esecuzione. la classe Node definisce al suo interno tre metodi, oltre al costruttore:

- **addNeighbor**:
  Questa funzione prende tre parametri: **self**, riferimento all'istanza chiamante, in modo da poter accedere al proprio campo **neighbors**; **neighbor** e **distance**, sono le informazioni corrispondenti al vicino da aggiungere alla lista di adiacenza.

- **updateNode**:
  Questa funzione prende solamente **self** come parametro, in modo da poter aggiornare la propria **routingTable**, la funzione itera attraverso la lista di adiacenza e confronta la propria routig table con quella dei vicini, eventualmente modificando la propria in caso trovasse percorsi a nuovi nodi o percorsi a nodi già noti, ma con distanze minori.
  La funzione ritorna un valore booleano che è **true se il nodo a aggiornato la propria tabella** e false altrimenti.

- **printTable**:
  La funzione printTable è una semplice utility che stampa la tabella di routing del nodo chiamante, serve a verificare i cambiamenti dopo ogni iterazione.

```python
# la classe Nodo rappresenta l'istanza del nodo di commutazione
class Node:
    # costruttore di classe
    def __init__(self, name):
        # Inizializza un nodo con un nome specifico e una tabella di routing.
        self.name = name
        """
        la routing table è utilizzata come dictionary, dove:
        le key sono i nomi delle destinazioni, e ad ogni key è associata una
        coppia di valori (distanza, nextHop).
        Ovviamente, per ogni nodo, viene inizializzato con distanza 0 da se stesso
        """
        self.routingTable = {name: (0, name)}  # Formato: {destinazione: (distanza, next_hop)}
        """
        Neighbors è una dictionary dove le chiavi sono i nodi adiacenti, e il valore
        corrispondenete ad ogni chiave è la distanza da esso.
        """
        self.neighbors = {}

    # funzione che aggiunge un nuovo nodo alla lista di adiacenza di un altro nodo
    def addNeighbor(self, neighbor, distance):
        # Aggiunge un nodo vicino con una distanza specifica.
        self.neighbors[neighbor] = distance
        self.routingTable[neighbor.name] = (distance, neighbor.name)

    def updateNode(self):
        """
        Aggiorna la tabella di routing utilizzando il protocollo Distance Vector.
        Restituisce True se la tabella è stata aggiornata, False altrimenti.
        """
        # flag che descrive se l'update è avvenuto o meno
        updated = False
        # per ogni vicino
        for neighbor, distance_to_neighbor in self.neighbors.items():
            # esamino la routing table
            neighbor_table = neighbor.routingTable
            """
            se il vicino riesce a raggiungere una destinazione nota percorrendo
            una distanza minore, oppure riesce a raggiungere una destinazione non nota,
            aggiorno la mia routing table con il nuovo percorso (tramite il vicino).
            """
            for dest, (dist, next_hop) in neighbor_table.items():
                new_distance = distance_to_neighbor + dist
                if dest not in self.routingTable or new_distance < self.routingTable[dest][0]:
                    self.routingTable[dest] = (new_distance, neighbor.name)
                    # se faccio modifiche alla routing table, update diventa True
                    updated = True
        return updated

    def printTable(self):
        # Stampa la tabella di routing del nodo.
        print(f"Routing table for {self.name}:")
        print("dest\t\t  dist\t\t\tnextHop")
        for dest, (dist, next_hop) in self.routingTable.items():
            print(f"{dest}\t\t/\t\t{dist}\t\t/\t\t{next_hop}")
        print()
```

Figura 1: classe Node

0.4 Creazione della Rete

La creazione della rete di nodi avviene al momento dell'esecuzione tramite la

chiamata del metodo **randomNetwork**, un metodo che accetta come parametri: il numero di nodi da creare **num˙nodes** e la distanza massima di cui si può generare un collegamento **max˙distance**.
La prima istruzione inizializza **num˙nodes** nodi, assegnando a ciascuno una lettera, dopodiché inizia ad iterare tra i nodi: fissato un nodo, per ciascun altro nodo nella rete crea un collegamento con una distanza tra 1 e **max˙distance** con una probabilità del 50%; una volta creato il collegamento aggiorna i vicini di entrambi i nodi.
Infine, per ogni nodo, controlla che abbia almeno un vicino, nel caso in cui il nodo sia isolato (**not node.neighbors**) crea un collegamento con un nodo casuale; questo viene fatto perché inizializzando in modo casuale la rete, può capitare che dei nodi rimangano isolati, questo è controproducente al fine di applicare l'algoritmo.

```python
def randomNetwork(num_nodes, max_distance=10):
    """
    Genera una rete casuale con un dato numero di nodi e distanze massime,
    garantendo che ogni nodo abbia almeno un collegamento.
    """
    # crea nodi per il numero indicato, dando a ciascuno (come nome) una lettera (A, B, C, ...)
    nodes = [Node(chr(65 + i)) for i in range(num_nodes)]  # Nodi con nomi A, B, C, ...

    # crea collegamenti random tra i nodi, con probabilità del 50%
    for i in range(num_nodes):
        for j in range(i + 1, num_nodes):
            if random.random() < 0.5:
                distance = random.randint(1, max_distance)
                nodes[i].addNeighbor(nodes[j], distance)
                nodes[j].addNeighbor(nodes[i], distance)

    '''
    Dato un nodo isolato non sarebbe utile per il test, questo ciclo fa sì che ogni nodo
    abbia al meno un collegamento, scelto sempre in modo casuale.
    '''
    for node in nodes:
        if not node.neighbors:  # Nodo isolato
            # Collega il nodo a un altro nodo casuale
            other_node = random.choice([n for n in nodes if n != node])
            distance = random.randint(1, max_distance)
            node.addNeighbor(other_node, distance)
            other_node.addNeighbor(node, distance)

    return nodes
```

Figura 2: Metodo randomNetwork

0.5 Aggiornamento delle Tabelle

L'aggiornamento delle tabelle avviene dentro al metodo **updateTables**, un metodo che accetta come parametri la lista dei nodi inizializzati **nodes** e il numero di iterazioni **iterations**.

Il metodo è molto semplice: per ogni nodo, chiama il metodo **updateNode**, se il nodo viene aggiornato, una variabile **updated** viene settata a true (N.B. basta che un singolo nodo venga aggiornato perché **updated** sia true), se nessun nodo viene aggiornato, **updated** è false e significa che è stata raggiunta la convergenza, altrimenti si continua fino al numero di iterazioni prestabilito.

```
def tablesUpdate(nodes, iterations=10):
    '''
    Simula l'aggiornamento delle tabelle di routing tra i nodi
    per un dato numero di iterazioni.
    '''
    for i in range(iterations):
        print(f"---------- Iteration {i + 1} ----------")
        # flage che indica se viene raggiunta la convergenza
        updated = False
        for node in nodes:
            if node.updateNode():
                # se almeno una tabella viene aggiornata, updated è true
                updated = True
        for node in nodes:
            # per ogni nodo stampa la sua routing table
            node.printTable()
        # se in questa iterazione nessuna tabella viene aggiornata, ferma il loop
        if not updated:
            print("Convergence accomplished.\n")
            break

# Configurazione della rete
num_nodes = 6  # Cambia per avere più o meno nodi
max_distance = 15  # Cambia per variare le distanze massime

# Generazione della rete
nodes = randomNetwork(num_nodes, max_distance)

# Simulazione del routing
tablesUpdate(nodes)
```

Figura 3: Metodo updateTables e avvio dello script