Relazione sul Protocollo di Routing Distance Vector

Kevin Shimaj, 0001070938
Lisa Vandi, 0001071265

a.a. 2024/2025

Contents

1 Introduzione

Il protocollo di routing Distance Vector è uno dei principali protocolli utilizzati per determinare i percorsi più brevi in una rete.
Tale protocollo è la versione dinamica e distribuita dell'algoritmo di Bellman-Ford tale per cui:

- ogni nodo scopre i suoi vicini e ne calcola la distanza da sè stesso
- ad ogni passo, ogni nodo invia ai propri vicini un vettore contenente la stima della sua distanza da tutti gli altri nodi della rete. All'interno delle tabelle di routing, ogni nodo inserisce: nodo di destinazione, distanza da esso e next hop.

In questa relazione, esploreremo l'implementazione di questo protocollo in Python, suddividendo il codice in metodi chiave e analizzandone il funzionamento. Verranno inoltre discussi i possibili miglioramenti e i limiti del protocollo.

2 Idea Generale del Codice

L'obiettivo del codice è simulare il comportamento di un protocollo Distance Vector in una rete composta da nodi connessi da collegamenti con costi specifici. Il codice consente di:

- Definire una rete come grafo con costi.
- Inizializzare le tabelle di routing per ogni nodo.
- Simulare lo scambio di informazioni tra i nodi per aggiornare le tabelle di routing.
- Stampare le tabelle di routing finali e intermedie.

Il protocollo utilizza iterazioni successive per propagare le informazioni attraverso la rete fino a raggiungere la convergenza.

3 Spiegazione Dettagliata del Codice

In questa sezione, analizziamo i metodi principali del codice con screenshot e relative spiegazioni.

3.1 Definizione della rete

La rete viene definita come un dizionario Python, dove ogni nodo ha una lista dei suoi vicini e dei costi associati:

Listing 1: Definizione della rete

```
rete = {
    "A": {"B": 1, "C": 4},
    "B": {"A": 1, "C": 2, "D": 5},
    "C": {"A": 4, "B": 2, "D": 1},
    "D": {"B": 5, "C": 1}
}
```

In fase di testing, abbiamo provato a definire la rete con un numero elevato di nodi, notando di conseguenza una convergenza molto lenta e con numerose iterazioni. Questo è in linea con quanto già studiato a lezione.
Abbiamo quindi deciso di optare per un numero ridotto di nodi, all'interno della rete. Ulteriori problemi riguardanti il protocollo verranno discussi nei capitoli successivi.

3.2 Inizializzazione delle tabelle di routing

Ogni nodo inizializza una tabella di routing in cui:

- La distanza verso sé stesso è 0.
- Le distanze verso gli altri nodi sono inizializzate a infinito (`float('inf')`).

Listing 2: Inizializzazione delle tabelle di routing

```
tabelle_routing = {
    nodo: {dest: (float('inf'), None) for dest in rete}
        for nodo in rete
}

for nodo in tabelle_routing:
    tabelle_routing[nodo][nodo] = (0, nodo)
```

3.3 Aggiornamento delle tabelle di routing

La funzione `aggiorna_tabella` è responsabile dell'aggiornamento della tabella di routing di un nodo basandosi sulle informazioni ricevute dai vicini. È una delle parti centrali del protocollo Distance Vector e funziona iterando sui vicini e sulle loro tabelle di routing per trovare percorsi più brevi.

Listing 3: Funzione di aggiornamento delle tabelle di routing

```
def aggiorna_tabella(nodo):
    cambiato = False
    for vicino, costo_vicino in rete[nodo].items():
        for destinazione, (costo, _) in tabelle_routing[
           vicino].items():
            nuovo_costo = costo_vicino + costo
            if nuovo_costo < tabelle_routing[nodo][
               destinazione][0]:
                tabelle_routing[nodo][destinazione] = (
                   nuovo_costo, vicino)
                cambiato = True
    return cambiato
```

Funzionamento

- Ogni nodo esamina i suoi vicini diretti.
- Calcola un nuovo costo per raggiungere ciascuna destinazione passando attraverso il vicino.
- Se trova un percorso più breve, aggiorna la tabella di routing.
- Restituisce `True` se ci sono stati aggiornamenti, `False` altrimenti.

3.4 Esecuzione del protocollo

La funzione `esegui_protocollo` simula lo scambio di informazioni tra i nodi per un numero massimo di iterazioni:

Listing 4: Simulazione del protocollo

```
def esegui_protocollo(num_iterazioni=5):
    for iterazione in range(num_iterazioni):
        print(f"\nIterazione {iterazione + 1}:")
        cambiamenti = any(aggiorna_tabella(nodo) for
           nodo in rete)
        if not cambiamenti:
            print("Convergenza raggiunta!")
            break
```

3.5 Visualizzazione delle tabelle di routing

Le tabelle di routing finali vengono stampate in un formato leggibile utilizzando la libreria `tabulate`:

Listing 5: Visualizzazione delle tabelle di routing

```
def mostra_tabelle_routing():
    for nodo, tabella in tabelle_routing.items():
        dati = [
            [destinazione, costo, next_hop if next_hop
               is not None else "None"]
            for destinazione, (costo, next_hop) in
               tabella.items()
        ]
        print(f"Tabella di Routing per {nodo}:")
        print(tabulate(dati, headers=["Destinazione", "
           Costo", "Next Hop"], tablefmt="grid"))
        print()
```

4 Contro del Protocollo di Routing Distance Vector

- **Lentezza nella convergenza**: in reti grandi, il protocollo può richiedere molte iterazioni.
- **Cold start**: inizialmente, ciascun nodo, ha informazioni riguardanti i diretti vicini.
- **Problemi di conteggio infinito**: in caso di guasti, i costi possono aumentare indefinitamente.
- **Limitazione delle dimensioni della rete**: non è scalabile come i protocolli di routing basati su *Link-State*.

5 Installazione delle Dipendenze e Esecuzione del Codice

5.1 Installazione della libreria `tabulate`

Per utilizzare il codice, è necessario installare la libreria `tabulate`, che serve per stampare le tabelle di routing in un formato leggibile. Per installarla,

eseguire il seguente comando nel terminale:

```
pip install tabulate
```

5.2 Esecuzione del Codice

Per eseguire il codice:

```
python distance_vector.py
```