Simulazione del Protocollo di Routing Distance Vector

Giacomo Biagioni
Matricola: 12345678
Email: giacomo.biagioni@studio.unibo.it

Obiettivo del Progetto

L'obiettivo del progetto è stato quello di implementare una simulazione del protocollo di **Distance Vector Routing (DVR)** in linguaggio Python. Questo protocollo è largamente utilizzato nelle reti di comunicazione per calcolare i percorsi più brevi tra i nodi di una rete. La simulazione permette di osservare il comportamento dell'algoritmo durante l'aggiornamento delle tabelle di routing e la convergenza verso lo stato stabile.

Descrizione dell'Algoritmo

Il protocollo Distance Vector funziona attraverso i seguenti passaggi:

1. **Inizializzazione della Tabella di Routing:** Ogni nodo dispone inizialmente di una tabella di routing in cui:
   - La distanza verso se stesso è 0.
   - La distanza verso tutti gli altri nodi è impostata a infinito (∞).
   - La distanza verso i nodi vicini è uguale al costo diretto del collegamento.
2. **Scambio di Informazioni:** I nodi comunicano periodicamente la propria tabella di routing ai vicini.
3. **Aggiornamento delle Tabelle:** Quando un nodo riceve una tabella di routing da un vicino, calcola la distanza minima verso ciascun nodo della rete sommando il costo del collegamento al vicino e la distanza presente nella tabella ricevuta.
4. **Convergenza:** L'algoritmo si arresta quando le tabelle di routing non subiscono più modifiche, raggiungendo così uno stato stabile.

Implementazione

La simulazione è stata implementata in Python con le seguenti componenti principali:

- **Inizializzazione della Rete:** La funzione `initialize_routing_table` crea una tabella di routing per ciascun nodo. Ogni tabella contiene:
  - I costi iniziali verso sé stesso (0), verso i vicini (costo diretto) e verso tutti gli altri nodi (∞).
  - Il *next hop* per ogni destinazione, che inizialmente è impostato al vicino diretto o a `None` per i nodi non raggiungibili.

Listing 1: Inizializzazione della tabella di routing

```
def initialize_routing_table(nodes, edges):
    routing_table = {}
    for node in nodes:
        routing_table[node] = {n: float('inf') for n in nodes}
        routing_table[node][node] = 0

    for edge in edges:
```

```
        nodo1, nodo2, costo = edge
        routing_table[nodo1][nodo2] = costo
        routing_table[nodo2][nodo1] = costo

    return routing_table
```

- **Aggiornamento Iterativo delle Tabelle:** La funzione `distance_vector_routing` implementa l'algoritmo principale, che:
  - Scorre iterativamente attraverso i nodi e i vicini.
  - Calcola il costo dei percorsi passando attraverso ogni vicino.
  - Aggiorna la tabella di routing solo se viene trovato un percorso più economico.
  - Si arresta quando le tabelle non subiscono più modifiche, segnalando la convergenza dell'algoritmo.

Listing 2: Aggiornamento iterativo delle tabelle di routing

```
def distance_vector_routing(nodes, edges, max_iterations=10):
    routing_table = initialize_routing_table(nodes, edges)

    for iteration in range(max_iterations):
        updated = False
        new_table = copy.deepcopy(routing_table)

        for node in nodes:
            for neighbor in nodes:
                if neighbor == node or routing_table[node][neighbor] ==
                    continue

                for dest in nodes:
                    if dest == node:
                        continue

                    new_cost = routing_table[node][neighbor] + routing_t
                    if new_cost < routing_table[node][dest]:
                        new_table[node][dest] = new_cost
                        updated = True

        routing_table = new_table
        if not updated:
            break

    return routing_table
```

- **Visualizzazione:** La funzione `print_routing_table` permette di osservare lo stato della tabella di routing per ciascun nodo. Stampa i costi e i *next hop*, aiutando a tracciare l'evoluzione dell'algoritmo.

Listing 3: Visualizzazione delle tabelle di routing

```
def print_routing_table(routing_table):
```

```
    for node, table in routing_table.items():
        print(f"Tabella di routing per il nodo {node}:")
        for dest, cost in table.items():
            print(f"  {dest}: {cost}")
        print("\n")
```

Esempio di Simulazione

Input:

- Nodi: `Nodo A, Nodo B, Nodo C, Nodo D`
- Collegamenti:
  - Nodo A ↔ Nodo B, costo: 1
  - Nodo B ↔ Nodo C, costo: 2
  - Nodo A ↔ Nodo C, costo: 4
  - Nodo C ↔ Nodo D, costo: 1

Output: Le tabelle di routing aggiornate ad ogni iterazione e quelle finali sono come segue:

```
Nodo A:
  Nodo A: 0
  Nodo B: 1
  Nodo C: 3
  Nodo D: 4

Nodo B:
  Nodo A: 1
  Nodo B: 0
  Nodo C: 2
  Nodo D: 3

Nodo C:
  Nodo A: 3
  Nodo B: 2
  Nodo C: 0
  Nodo D: 1

Nodo D:
  Nodo A: 4
  Nodo B: 3
  Nodo C: 1
  Nodo D: 0
```

Analisi e Conclusioni

- **Convergenza:** L'algoritmo converge rapidamente, con un numero limitato di iterazioni.
- **Efficienza:** La complessità computazionale è $O(N^3)$, dove N è il numero di nodi, poiché ogni nodo aggiorna le proprie distanze per tutti gli altri nodi.
- **Robustezza:** Il protocollo gestisce bene le variazioni dei costi, ma è suscettibile al problema dei *count-to-infinity*, non affrontato in questa implementazione.