

Relazione Progetto Reti di Telecomunicazioni

Distance Vector Routing (DVR)

Brighi Federico matr. 0001070887
Saponaro Mattia matr. 0001071328

December 7, 2024

1 Introduzione

Il **Distance Vector Routing** è uno degli algoritmi fondamentali per la determinazione dei percorsi ottimali in una rete. L'algoritmo si basa sulla propagazione delle informazioni di routing tra i vari **Nodi** appartenenti alla medesima **Rete**. Ogni nodo conserva una **Tabella di Routing** che indica il costo per raggiungere ogni altro nodo della rete. Durante l'esecuzione dell'algoritmo, ogni nodo invia periodicamente la propria tabella di routing ai nodi vicini e aggiorna la sua tabella in base alle informazioni ricevute.

L'obiettivo del nostro progetto è implementare una rete composta da 4 nodi utilizzando l'algoritmo di Distance Vector Routing. Ogni nodo mantiene la propria tabella di routing e la aggiorna in base alle informazioni provenienti dai nodi vicini.

2 Descrizione del problema

Il problema consiste nella creazione di una rete composta da quattro nodi denominati A , B , C e D , collegati tra loro da collegamenti/link con relativi costi. L'algoritmo di Distance Vector Routing deve calcolare le tabelle di routing per ciascun nodo, in cui sono indicati i costi per raggiungere ogni destinazione e il nodo successivo da percorrere (Next Hop).

Le tabelle di routing vengono aggiornate iterativamente fino a quando non ci sono più modifiche. Ogni nodo inizialmente conosce solo i propri collegamenti diretti, ma grazie alla comunicazione tra nodi, imparerà i percorsi più brevi per ogni destinazione.

3 Descrizione del codice

Il nostro codice che implementa l'algoritmo di Distance Vector Routing è suddiviso in due classi principali:

- **Node:** rappresenta un nodo della rete. Ogni nodo mantiene una tabella di routing, una lista di vicini e il nodo successivo (Next Hop) per ogni destinazione.
- **Network:** gestisce la rete, i nodi e i collegamenti tra di essi. Si occupa anche di aggiornare le tabelle di routing e di stampare il risultato finale.

3.1 Classe Node

La classe **Node** è responsabile per la gestione della tabella di routing di ciascun nodo, dei vicini e del next hop. Ogni nodo inizia con una tabella che contiene solo se stesso (con costo 0) e un next hop impostato a *None*. Man mano che le informazioni vengono ricevute dai vicini, la tabella viene aggiornata.

Attributi della Classe:

- **name:** Il nome identificativo del nodo.
- **routing_table:** Un dizionario che rappresenta la tabella di routing del nodo. Le chiavi sono i nomi delle destinazioni, e i valori sono i costi per raggiungere tali destinazioni.
 - Inizialmente, contiene solo il costo per raggiungere sé stesso, che è 0.
- **next_hop:** Un dizionario che memorizza il prossimo nodo ("next hop") per ogni destinazione nella tabella di routing.
 - Inizialmente, il next hop per sé stesso è **None**.
- **neighbors:** Un dizionario che memorizza i nodi vicini e i costi per raggiungerli.
 - Le chiavi sono i nomi dei vicini, e i valori sono tuple (**neighbor_node**, **cost**).

Metodi Principali:

- **`__init__()`:**
 - Inizializza il nodo con il proprio nome, una tabella di routing vuota, e una lista di vicini.
- **`add_neighbor(neighbor, cost)`:**
 - Aggiunge un nodo vicino alla lista dei vicini con il relativo costo.
 - Aggiorna la tabella di routing con il costo iniziale per raggiungere il vicino e imposta il next hop al vicino stesso.
- **`receive_routing_info(neighbor_name, neighbor_table)`:**
 - Riceve una tabella di routing da un vicino e aggiorna la propria tabella se trova percorsi migliori o nuove destinazioni.
 - Calcola il nuovo costo per una destinazione passando attraverso il vicino e confronta con il costo attuale. Se il nuovo costo è inferiore, aggiorna la tabella e imposta il next hop al vicino.
 - Restituisce **True** se la tabella è stata aggiornata, altrimenti **False**.
- **`send_routing_info()`:**
 - Restituisce il nome del nodo e la sua tabella di routing, che può essere propagata ai vicini.

```
class Node:
    def __init__(self, name):
        self.name = name
        self.routing_table = {name: 0} # Costo a sé stesso è 0
        self.next_hop = {name: None} # Next hop a sé stesso è None
        self.neighbors = {}

    def add_neighbor(self, neighbor, cost):
        """Aggiungi un nodo vicino alla lista dei vicini con il relativo costo."""
        self.neighbors[neighbor.name] = (neighbor, cost) # Memorizza il nodo vicino e il costo
        self.routing_table[neighbor.name] = cost # Aggiungi il costo per il vicino
        self.next_hop[neighbor.name] = neighbor.name # Il next hop inizialmente è il vicino stesso

    def receive_routing_info(self, neighbor_name, neighbor_table):
        """
        Riceve le informazioni di routing da un vicino e aggiorna la propria tabella
        se necessario, restituendo True se la tabella è stata aggiornata.
        """
        updated = False
        for dest, cost_to_dest in neighbor_table.items():
            # Calcola il nuovo costo passando attraverso il vicino
            new_cost = self.neighbors[neighbor_name][1] + cost_to_dest
            if dest not in self.routing_table or self.routing_table[dest] > new_cost:
                self.routing_table[dest] = new_cost
                self.next_hop[dest] = neighbor_name # Aggiorna il next hop
                updated = True
        return updated # Ritorna True se la tabella è stata aggiornata

    def send_routing_info(self):
        """Restituisce il nome del nodo e la sua tabella di routing."""
        return self.name, self.routing_table
```

Figure 1: Classe Node con i metodi principali.

3.2 Classe Network

La classe **Network** gestisce la nostra rete, aggiungendo nodi e link, e si occupa dell'aggiornamento delle tabelle di routing dei 4 nodi.

Attributi della Classe:

- **nodes**: Un dizionario che mappa i nomi dei nodi agli oggetti **Node**.

Metodi Principali:

- **__init__()**:
 - Inizializza la rete con un dizionario vuoto di nodi.
- **add_node(node_name)**:
 - Aggiunge un nodo alla rete. Se il nodo esiste già, stampa un messaggio di avviso.
- **add_link(node1_name, node2_name, cost)**:
 - Crea un collegamento tra due nodi esistenti con un determinato costo.
 - Aggiunge reciprocamente ogni nodo alla lista dei vicini dell'altro.
- **update_routing_tables()**:
 - Aggiorna iterativamente le tabelle di routing di tutti i nodi finché non ci sono più modifiche.
 - Utilizza l'algoritmo Distance Vector Routing: ogni nodo riceve informazioni di routing dai vicini e aggiorna la propria tabella se trova percorsi migliori.
- **get_routing_tables()**:
 - Restituisce le tabelle di routing di tutti i nodi in un formato strutturato, utile per visualizzazione o debugging.
 - Ogni tabella contiene due dizionari: uno per i costi di destinazione e uno per i next hop.

- `print_routing_tables()`:
 - Stampa il risultato delle tabelle di Routing su terminale.

```
class Network:
    def __init__(self):
        self.nodes = {}

    def add_node(self, node_name):
        """Aggiungi un nodo alla rete se non esiste già."""
        if node_name not in self.nodes:
            self.nodes[node_name] = Node(node_name)
        else:
            print(f"Attenzione: Nodo {node_name} esiste già!")

    def add_link(self, node1_name, node2_name, cost):
        """Aggiungi un collegamento tra due nodi con il relativo costo."""
        if node1_name not in self.nodes or node2_name not in self.nodes:
            print(f"Errore: Uno o entrambi i nodi {node1_name}, {node2_name} non esistono.")
            return
        node1, node2 = self.nodes[node1_name], self.nodes[node2_name]
        node1.add_neighbor(node2, cost)
        node2.add_neighbor(node1, cost)

    def update_routing_tables(self):
        """
        Aggiorna le tabelle di routing per tutti i nodi finché non ci sono più cambiamenti.
        Ottimizzazione: esce quando nessuna tabella è cambiata.
        """
        updates = True
        while updates:
            updates = False
            for node in self.nodes.values():
                # Propagazione delle tabelle dai vicini
                for neighbor in node.neighbors.values():
                    neighbor_name, neighbor_table = neighbor[0].send_routing_info()
                    if node.receive_routing_info(neighbor_name, neighbor_table):
                        updates = True

    def print_routing_tables(self):
        """Stampa le tabelle di routing finali per ogni nodo."""
        for node in self.nodes.values():
            print(f"TABELLA DI ROUTING DEL NODO {node.name}:")
            print(f"{'DESTINAZIONE':<15}{'COSTO':<10}{'Next Hop'}") # Header
            for dest in sorted(node.routing_table):
                cost = node.routing_table[dest]
                next_hop = node.next_hop[dest]
                print(f"dest:<15}{cost:<10}{next_hop}") # Formattazione tabellare
            print()
```

Figure 2: Classe Network con i suoi metodi principali.

3.3 Aggiornamento delle tabelle di routing

Le tabelle di routing vengono aggiornate attraverso un processo iterativo. Ogni nodo invia la propria tabella ai vicini, e ogni nodo riceve e aggiorna la propria tabella in base alle informazioni ricevute.

Il processo continua fino a quando non ci sono più modifiche nelle tabelle.

4 Soluzione e risultato

La rete finale consiste in quattro nodi, A, B, C, D, con i seguenti collegamenti:

- A è connesso a B con costo 2 e a C con costo 3.
- B è connesso a C con costo 1 e a D con costo 6.
- C è connesso solamente a D con costo 5.

Le tabelle di routing finali per ogni nodo vengono stampate sul terminale al termine del processo di aggiornamento delle tabelle e anche mostrate su una GUI personalizzata, realizzata tramite la funzione **show_routing_tables_gui** presente nella classe **Network** e grazie all'importazione della libreria **tkinter**.

```
def show_routing_tables_gui(self):
    """Mostra una GUI con le tabelle di routing."""
    root = tk.Tk()
    root.title("Tabelle di Routing")

    # Crea un notebook per i tab
    notebook = ttk.Notebook(root)

    for node_name, table in self.get_routing_tables().items():
        frame = ttk.Frame(notebook)
        notebook.add(frame, text=f"Nodo {node_name}")

        tree = ttk.Treeview(frame, columns=("Destinazione", "Costo", "Next Hop"), show="headings")
        tree.heading("Destinazione", text="Destinazione")
        tree.heading("Costo", text="Costo")
        tree.heading("Next Hop", text="Next Hop")

        # Inserisce i dati nella tabella
        for dest, cost in table["destination"].items():
            next_hop = table["next_hop"][dest]
            tree.insert("", "end", values=(dest, cost, next_hop))

        tree.pack(expand=True, fill="both")

    notebook.pack(expand=True, fill="both")
    root.mainloop()
```

Figure 3: Funzione per creare un output delle tabelle tramite GUI.

4.1 Tabelle di Routing Finali

• Nodo A:	Destinazione	Costo	Next Hop
	A	0	None
	B	2	B
	C	3	C
	D	8	B

• Nodo B:	Destinazione	Costo	Next Hop
	A	2	A
	B	0	None
	C	1	C
	D	6	D

• Nodo C:	Destinazione	Costo	Next Hop
	A	3	A
	B	1	B
	C	0	None
	D	5	D

• Nodo D:	Destinazione	Costo	Next Hop
	A	8	B
	B	6	B
	C	5	C
	D	0	None

5 Conclusioni e Problematiche Riscontrate

Come si può notare dalle tabelle mostrate in output, l'algoritmo di Distance Vector Routing è stato implementato correttamente per una rete composta da quattro nodi. Le tabelle di routing sono state aggiornate in modo iterativo e il risultato finale mostra i percorsi ottimali tra i nodi, con i costi e i next hop appropriati. Questo tipo di algoritmo è utile per reti dinamiche dove i costi dei collegamenti possono cambiare nel tempo, ma potrebbe non essere la scelta migliore per reti di grandi dimensioni a causa del numero elevato di aggiornamenti necessari.

Durante lo sviluppo del progetto abbiamo incontrato qualche difficoltà tecnica che ci ha messo alla prova : una delle principali sfide è stata garantire la consistenza delle tabelle di routing durante cambiamenti dinamici nella

topologia della rete. Durante i test, abbiamo notato che alcune tabelle temporaneamente contenevano dati non coerenti, causando percorsi non ottimali. Per mitigare questo comportamento, abbiamo implementato una soluzione che garantisce la corretta decisione del miglior percorso possibile tra i vari nodi della rete in base ai livelli di efficienza.