# Simulazione di un Protocollo di Routing: Distance Vector Routing

Federico Morsucci

December 11, 2024

**Abstract**

In questo documento presentiamo l'implementazione e la simulazione di un protocollo di routing basato sull'algoritmo di Distance Vector Routing, sviluppato in Python. La simulazione evidenzia come il protocollo calcoli le rotte ottimali in una rete, aggiornando iterativamente le tabelle di routing fino a convergenza. Valutiamo la soluzione in base ai criteri di completezza, chiarezza della documentazione, accuratezza tecnica e capacità di diagnosi dei problemi.

# Contents

**1 Introduzione** **2**
- 1.1 Obiettivi . . . 2
- 1.2 Vantaggi e Limitazioni . . . 2

**2 Struttura del Codice** **2**
- 2.1 Classe Node . . . 3
- 2.2 Classe Network . . . 4
  - 2.2.1 Funzione `initialize_routing_tables` . . . 4
  - 2.2.2 Funzione `run_distance_vector` . . . 5

**3 Analisi e Problemi Risolti** **6**

**4 Conclusioni** **6**

# 1 Introduzione

Il Distance Vector Routing (DVR) è un algoritmo di routing per reti di computer che consente a ciascun nodo di determinare il percorso più breve verso ogni altro nodo. Ogni nodo condivide periodicamente la propria tabella di routing con i vicini per aggiornare le informazioni.

## 1.1 Obiettivi

La simulazione soddisfa i seguenti obiettivi:

1. Implementazione di un modello di rete con nodi interconnessi.
2. Calcolo iterativo delle tabelle di routing usando l'algoritmo DVR.
3. Analisi della convergenza e gestione di problemi tipici del DVR.

## 1.2 Vantaggi e Limitazioni

Il DVR è semplice e scalabile per reti di dimensioni moderate, ma soffre di limitazioni come:

- Convergenza lenta in reti grandi.
- Oscillazioni dovute a cambiamenti dinamici.
- Problemi di loop in assenza di meccanismi aggiuntivi come *Split Horizon*.

# 2 Struttura del Codice

Il codice Python che implementa l'algoritmo di Distance Vector Routing è composto da due classi principali: `Node` e `Network`.

## 2.1 Classe Node

Ogni nodo nella rete è rappresentato dalla classe `Node`. Un nodo mantiene una tabella di routing, che associa ogni destinazione a una tupla contenente la distanza e il prossimo nodo per arrivare a quella destinazione.

Listing 1: Definizione della classe Node

```
class Node:
    def __init__(self, name):
        self.name = name
        self.routing_table = {}  # Destinazione: (Distanza, Prossimo Nodo)

    def update_routing_table(self, neighbor, neighbor_table, weight):
        """
        Aggiorna la tabella di routing di un nodo sulla base delle informazioni
        ricevute da un nodo vicino.
        """
        updated = False
        for dest, (dist, _) in neighbor_table.items():
            new_dist = weight + dist
            if dest not in self.routing_table or new_dist < self.routing_table[dest][0]:
                self.routing_table[dest] = (new_dist, neighbor)
                updated = True
        return updated

    def __str__(self):
        """
        Rappresentazione leggibile della tabella di routing del nodo.
        """
        table = f"Routing Table for {self.name}:\n"
        table += f"{'Destinazione':<15} {'Distanza':<10} {'Prossimo Nodo'}\n"
        table += "-" * 40 + "\n"
        for dest, (dist, next_hop) in self.routing_table.items():
            table += f"{dest:<15} {dist:<10} {next_hop}\n"
        return table
```

Nel metodo `update_routing_table`, ogni nodo aggiorna la propria tabella di routing confrontando le informazioni dei nodi vicini, determinando se un percorso più breve è disponibile. La distanza di ciascun nodo viene aggiornata se un percorso più breve è trovato.

## 2.2 Classe Network

La classe `Network` gestisce l'intera rete e la logica di aggiornamento delle tabelle di routing. Gestisce i collegamenti tra i nodi e esegue l'algoritmo di Distance Vector, iterando fino a quando tutte le tabelle di routing convergono. Le due funzioni principali di questa classe sono descritte di seguito:

### 2.2.1 Funzione `initialize_routing_tables`

La funzione `initialize_routing_tables` è responsabile dell'inizializzazione delle tabelle di routing per tutti i nodi della rete. All'inizio, ogni nodo ha una distanza di zero verso sé stesso, una distanza diretta ai nodi vicini e una distanza infinita (`float('inf')`) per i nodi che non sono direttamente collegati. In altre parole, ogni nodo conosce i percorsi diretti a sé stesso e ai suoi vicini immediati, mentre le rotte verso gli altri nodi sono inizialmente sconosciute.

Listing 2: Funzione initialize_routing_tables della classe Network

```
1  def initialize_routing_tables(self):
2      """
3      Inizializza le tabelle di routing per ogni nodo
           nella rete.
4      """
5      for node in self.nodes.values():
6          for other in self.nodes:
7              if other == node.name:
8                  node.routing_table[other] = (0, other) #
                       Distanza a se' stesso
9              elif (node.name, other) in self.links:
10                 node.routing_table[other] = (self.links
                       [(node.name, other)], other)
11             else:
12                 node.routing_table[other] = (float('inf'
                       ), None)
```

### 2.2.2 Funzione `run_distance_vector`

La funzione `run_distance_vector` implementa l'algoritmo di Distance Vector. Durante ogni iterazione, ogni nodo confronta la propria tabella di routing con quella dei nodi vicini e aggiorna la propria tabella se trova percorsi migliori. Questo processo viene ripetuto fino a quando le tabelle di routing non subiscono più cambiamenti, segnalando che la rete ha raggiunto la convergenza.

Listing 3: Funzione run_distance_vector della classe Network

```
def run_distance_vector(self):
    """
    Esegue l'algoritmo di Distance Vector fino a convergenza.
    """
    self.initialize_routing_tables()
    converged = False
    iteration = 0
    prev_routing_tables = {node.name: dict(node.routing_table) for node in self.nodes.values()}

    while not converged:
        converged = True
        for (node1, node2), weight in self.links.items():
            n1 = self.nodes[node1]
            n2 = self.nodes[node2]

            # Aggiorna le tabelle di routing in entrambe le direzioni
            updated1 = n1.update_routing_table(node2, n2.routing_table, weight)
            updated2 = n2.update_routing_table(node1, n1.routing_table, weight)

            if updated1 or updated2:
                converged = False
        iteration += 1
```

# 3 Analisi e Problemi Risolti

Durante la simulazione, abbiamo identificato i seguenti problemi:

1. **Convergenza Lenta**: In reti grandi, il DVR richiede molte iterazioni. Per mitigare, possiamo implementare tecniche come *Split Horizon*.

2. **Cicli Temporanei**: Per evitare loop, si possono aggiungere controlli durante l'aggiornamento delle tabelle.

# 4 Conclusioni

Il DVR è un algoritmo efficace per reti moderate, ma richiede ottimizzazioni per garantire una convergenza rapida e stabile. La simulazione ha mostrato come implementarlo correttamente in Python e risolvere le sfide associate.