Università di Bologna - Campus di Cesena
Ingegneria e Scienze Informatiche (8615)

"Reti di telecomunicazione"

Simulazione Distance Vector Routing

Matricola: 0001080182
Manuele D'Ambrosio

Anno Accademico 2024 - 2025

Indice

1 Introduzione

Il protocollo di routing *Distance Vector*, basato sull'algoritmo Bellman-Ford, rappresenta uno dei metodi fondamentali per la determinazione dei percorsi in reti di comunicazione. La sua semplicità e il meccanismo intuitivo di scambio di informazioni tra router lo rendono una scelta adatta per reti di dimensioni contenute, dove i requisiti di configurazione e manutenzione devono rimanere minimi.

2 Caratteristiche principali

Il protocollo fa in modo che per ogni nodo si conosca la distanza da ogni altro nodo da esso raggiungibile, inoltre per ogni destinazione viene memorizzato il *next-hop* ossia il prossimo nodo da attraversare per raggiungere tale destinazione. La distanza considerata può essere espressa in diversi modi, ad esempio nel caso più semplice la distanza rappresenta il numero di salti da compiere per raggiungere la destinazione, ma potrebbe essere espressa anche in termini di tempo o costi di altra natura.

3 Funzionamento

3.1 Inizializzazione

Inizialmente ogni nodo è a conoscenza dei soli nodi direttamente connessi e delle relative distanze, in questo modo anche il next-hop coincide con il nodo di destinazione. La distanza del nodo da sé stesso è ovviamente inizializzata a 0.

3.2 Aggiornamenti

Periodicamente (o quando c'è un cambiamento significativo), ogni router invia il suo distance vector ai nodi vicini. Successivamente i vicini aggiornano il proprio distance vector basandosi sulle informazioni ricevute, utilizzando l'algoritmo:

$$D(d) = min\{D(d), D(v) + c(v, d)\}$$

Dove:

- D(d) -> distanza verso la destinazione d
- D(v) -> distanza verso il vicino v

- c(v,d) -> costo del collegamento tra v e d

3.3 Convergenza

Dopo un certo numero di iterazioni, tutti router completano il proprio distance vector (compreso di next-hop) in cui saranno presenti le indicazioni per raggiungere ogni router appartenente alla stessa *network* (Se questa è fortemente connessa). Le iterazioni terminano nel momento in cui si smette di effettuare nuovi aggiornamenti. Nel caso in cui il grafo della topologia non sia fortemente connesso, ossia non sia sempre presente un percorso che congiunge ogni nodo con ogni altro, al termine nella convergenza i distance vector potrebbero non contenere tutti i nodi presenti nella network.

4 Problematiche del protocollo distance vector

4.1 Problemi tipici

- **Loop di routing**: I router possono formare cicli logici a causa di aggiornamenti non sincronizzati.
- **Conteggio infinito (Count to infinity)**: La distanza verso una certa destinazione potrebbe crescere indefinitamente prima che i router apprendano che tale destinazione non sia raggiungubile.
- **Lentezza nella convergenza**: In caso di cambiamenti topologici, in particolare in network di grandi dimensioni, possono essere necessari molti aggiornamenti per raggiungere una nuova stabilità.

4.2 Meccanismi per mitigare i problemi

- **Split Horizon**: Questo metodo viene utilizzato per prevenire dei loop e consiste nell'impedire a un router di inviare le informazioni di routing al router da cui le ha ricevute.
- **Poison Reverse**: Quando un router scopre che una certa destinazione non è più raggiungibile, invia un aggiornamento indicando un distanza infinita per quella destinazione. Nel caso del protocollo RIP questa distanza viene impostata a 16.
- **Holddown Timer**: Vengono temporaneamente bloccati gli aggiornamenti per una rete considerata instabile.

5 Protocolli basati su Distance Vector

- **RIP (Routing Information Protocol)**: Usa il numero di hop come valutazione della distanza e consente un massimo di 15 hop, la distanza 16 è utilizzata per indicare una destinazione non raggiungibile (distanza infinita).
- **IGRP (Interior Gateway Routing Protocol)**: Versione avanzata sviluppata da *Cisco*.

6 Script Python

Lo script python (denominato *DistanceVectorRouting.py*) simula il funzionamento di un protocollo di routing basato su *Distance Vector*. In seguito saranno mostrate e descritte le parti del codice più importanti.

6.1 Classe NetworkNode

```
6  class NetworkNode:
7      def __init__(self, name, neighbors):
8          self.name = name
9          self.neighbors = neighbors
10         self.distance_vector = {name: 0}
11         self.next_hop = {name: None}
12         for neighbor, cost in neighbors.items():
13             self.distance_vector[neighbor] = cost
14             self.next_hop[neighbor] = neighbor
15
16     def update_distance_vector(self, network_nodes):
17         updated = False
18         for neighbor in self.neighbors:
19             if neighbor not in network_nodes:
20                 continue
21             neighbor_vector = network_nodes[neighbor].distance_vector
22             for dest, cost in neighbor_vector.items():
23                 if dest == self.name:
24                     continue
25                 new_cost = self.neighbors[neighbor] + cost
26                 if dest not in self.distance_vector or new_cost <
                    ↪  self.distance_vector[dest]:
27                     self.distance_vector[dest] = new_cost
28                     self.next_hop[dest] = neighbor
```

```
                updated = True
        return updated
```

La classe **NetworkNode** rappresenta un nodo (o router) appartenente alla network. Per inizializzare ogni nodo è necessario conoscere il suo nome oltre che al nome dei nodi vicini e la loro distanza (di default è impostata distanza 1). Nella classe è anche presente un metodo (*update_distance_vector*) che ha lo scopo di simulare l'aggiornamento del distance vector del nodo.

6.2 Classe Network

```
class Network:
    def __init__(self, topology):
        self.nodes = {name: NetworkNode(name, neighbors) for name, neighbors in
        ↪  topology.items()}

    def simulate_routing(self):
        iteration = 0
        print(f"Iterazione {iteration}: Stato iniziale")
        self.print_distance_vectors()

        while True:
            updated = False
            iteration += 1
            for node in self.nodes.values():
                if node.update_distance_vector(self.nodes):
                    updated = True

            print(f"\nIterazione {iteration}: Aggiornamento")
            self.print_distance_vectors()

            if not updated:
                print("\nConvergenza raggiunta!")
                break

        print("\nTabelle di routing finali dopo la convergenza:")
        self.print_distance_vectors()

    def print_distance_vectors(self):
        for name, node in self.nodes.items():
            print(f"Nodo {name}:")
```

```
62            print(f"  Distance Vector: {node.distance_vector}")
63            print(f"  Next Hop: {node.next_hop}")
```

La classe **Network** viene costruita mediante la topologia scelta e rappresenta il funzionamento dell'intera network. il metodo *simulate_routing* avvia lo scambio di informazioni tra nodi ed esegue un certo numero di iterazioni fino a che non vengono più effettuati aggiornamenti. Il metodo *print_print_distance_vectors* è utilizzato per stampare la situazione della network ad ogni iterazione.

6.3 Main

```
106 def main():
107     hop_count=True
108     if len(sys.argv) > 1:
109         if sys.argv[1] == '0':
110             hop_count=False
111             topology = default_topology(hop_count)
112             print("Utilizzo della topologia di default con costi randomizzati:")
113         else:
114             input_file = sys.argv[1]
115             topology = load_topology(input_file, hop_count)
116     else:
117         topology = default_topology(hop_count)
118         print("Nessuna topologia specificata. Utilizzo della topologia di default:")
119 
120     try:
121         for node, neighbors in topology.items():
122             print(f"{node}: {neighbors}")
123         print("\n")
124     except (AttributeError, UnboundLocalError):
125         print("Errore: La topologia caricata non è valida. Terminazione forzata")
126         sys.exit(1)
127 
128     network = Network(topology)
129     network.simulate_routing()
```

La funzione **Main** si occupa di verificare se siano stati passati eventuali parametri da riga di comando prima di eseguire il codice. Nel caso non venga passato nessun parametro da riga di comando la network sarà costruita

mediante una topologia di default rappresentata da un grafo fortemente connesso, non orientato e non pesato. Nel caso si scelga invece si scelga di passare il parametro '0', la topologia caricata sarà equivalente a quella di default, con la differenza che i costi degli archi saranno inizializzato casualmente con un valore tra 1 e 5 (compresi). Se si vuole invece utilizzare una topologia diversa da quella di default sarà possibile passarla da riga di comando come stringa in formato JSON oppure passare direttamente un file JSON contenente la topologia che si vuole utilizzare (Nella repository è presente un file JSON che si può utilizzare).

7 Esempi di esecuzione del codice

7.1 Topologia di default

Se si vuole lanciare lo script utilizzando la topologia di default, che considera sempre uguale a 1 la distanza tra due nodi adiacenti, bisogna posizionarsi nella cartella contenente lo script e lanciare il seguente comando:

```
python DistanceVectorRouting.py
```

```
Tabelle di routing finali dopo la convergenza:
Nodo A:
  Distance Vector: {'A': 0, 'B': 1, 'D': 1, 'C': 2, 'F': 2, 'E': 2, 'G': 2, 'H': 3}
  Next Hop: {'A': None, 'B': 'B', 'D': 'D', 'C': 'B', 'F': 'B', 'E': 'D', 'G': 'D', 'H': 'D'}
Nodo B:
  Distance Vector: {'B': 0, 'A': 1, 'C': 1, 'F': 1, 'D': 2, 'E': 3, 'G': 2, 'H': 4}
  Next Hop: {'B': None, 'A': 'A', 'C': 'C', 'F': 'F', 'D': 'A', 'E': 'A', 'G': 'C', 'H': 'A'}
Nodo C:
  Distance Vector: {'C': 0, 'B': 1, 'D': 1, 'F': 1, 'G': 1, 'A': 2, 'E': 2, 'H': 3}
  Next Hop: {'C': None, 'B': 'B', 'D': 'D', 'F': 'F', 'G': 'G', 'A': 'B', 'E': 'D', 'H': 'D'}
Nodo D:
  Distance Vector: {'D': 0, 'A': 1, 'C': 1, 'E': 1, 'G': 1, 'B': 2, 'F': 2, 'H': 2}
  Next Hop: {'D': None, 'A': 'A', 'C': 'C', 'E': 'E', 'G': 'G', 'B': 'A', 'F': 'C', 'H': 'E'}
Nodo E:
  Distance Vector: {'E': 0, 'D': 1, 'H': 1, 'A': 2, 'C': 2, 'G': 2, 'B': 3, 'F': 3}
  Next Hop: {'E': None, 'D': 'D', 'H': 'H', 'A': 'D', 'C': 'D', 'G': 'D', 'B': 'D', 'F': 'D'}
Nodo F:
  Distance Vector: {'F': 0, 'B': 1, 'C': 1, 'A': 2, 'D': 2, 'E': 3, 'G': 2, 'H': 4}
  Next Hop: {'F': None, 'B': 'B', 'C': 'C', 'A': 'B', 'D': 'C', 'E': 'C', 'G': 'C', 'H': 'C'}
Nodo G:
  Distance Vector: {'G': 0, 'C': 1, 'D': 1, 'B': 2, 'F': 2, 'A': 2, 'E': 2, 'H': 3}
  Next Hop: {'G': None, 'C': 'C', 'D': 'D', 'B': 'C', 'F': 'C', 'A': 'D', 'E': 'D', 'H': 'D'}
Nodo H:
  Distance Vector: {'H': 0, 'E': 1, 'D': 2, 'A': 3, 'C': 3, 'G': 3, 'B': 4, 'F': 4}
  Next Hop: {'H': None, 'E': 'E', 'D': 'E', 'A': 'E', 'C': 'E', 'G': 'E', 'B': 'E', 'F': 'E'}
```

Figura 1: Esempio di output con topologia di default

7.2 Topologia di default con pesi casuali

Se si vuole lanciare lo script utilizzando la topologia di default con pesi casuali, bisogna posizionarsi nella cartella contenente lo script e lanciare il seguente comando:

```
python DistanceVectorRouting.py 0
```

```
Tabelle di routing finali dopo la convergenza:
Nodo A:
  Distance Vector: {'A': 0, 'B': 4, 'D': 2, 'C': 5, 'F': 6, 'E': 4, 'G': 7, 'H': 9}
  Next Hop: {'A': None, 'B': 'B', 'D': 'D', 'C': 'D', 'F': 'B', 'E': 'D', 'G': 'D', 'H': 'D'}
Nodo B:
  Distance Vector: {'B': 0, 'A': 4, 'C': 4, 'F': 2, 'D': 6, 'E': 8, 'G': 7, 'H': 13}
  Next Hop: {'B': None, 'A': 'A', 'C': 'C', 'F': 'F', 'D': 'A', 'E': 'A', 'G': 'C', 'H': 'A'}
Nodo C:
  Distance Vector: {'C': 0, 'B': 4, 'D': 3, 'F': 2, 'G': 3, 'A': 5, 'E': 5, 'H': 10}
  Next Hop: {'C': None, 'B': 'B', 'D': 'D', 'F': 'F', 'G': 'G', 'A': 'D', 'E': 'D', 'H': 'D'}
Nodo D:
  Distance Vector: {'D': 0, 'A': 2, 'C': 3, 'E': 2, 'G': 5, 'B': 6, 'F': 5, 'H': 7}
  Next Hop: {'D': None, 'A': 'A', 'C': 'C', 'E': 'E', 'G': 'G', 'B': 'A', 'F': 'C', 'H': 'E'}
Nodo E:
  Distance Vector: {'E': 0, 'D': 2, 'H': 5, 'A': 4, 'C': 5, 'G': 7, 'B': 8, 'F': 7}
  Next Hop: {'E': None, 'D': 'D', 'H': 'H', 'A': 'D', 'C': 'D', 'G': 'D', 'B': 'D', 'F': 'D'}
Nodo F:
  Distance Vector: {'F': 0, 'B': 2, 'C': 2, 'A': 6, 'D': 5, 'E': 7, 'G': 5, 'H': 12}
  Next Hop: {'F': None, 'B': 'B', 'C': 'C', 'A': 'B', 'D': 'C', 'E': 'C', 'G': 'C', 'H': 'C'}
Nodo G:
  Distance Vector: {'G': 0, 'C': 3, 'D': 5, 'B': 7, 'F': 5, 'A': 7, 'E': 7, 'H': 12}
  Next Hop: {'G': None, 'C': 'C', 'D': 'D', 'B': 'C', 'F': 'C', 'A': 'D', 'E': 'D', 'H': 'D'}
Nodo H:
  Distance Vector: {'H': 0, 'E': 5, 'D': 7, 'A': 9, 'C': 10, 'G': 12, 'B': 13, 'F': 12}
  Next Hop: {'H': None, 'E': 'E', 'D': 'E', 'A': 'E', 'C': 'E', 'G': 'E', 'B': 'E', 'F': 'E'}
```

Figura 2: Esempio di output con topologia di default con pesi casuali

7.3 Topologia su file JSON

Se si vuole lanciare lo script utilizzando un file JSON contenente la topologia che si vuole utilizzare (In questo caso è utilizzato il file di esempio *ExampleTopology.json* presente nella directory), bisogna posizionarsi nella cartella contenente lo script e lanciare il seguente comando:

```
python DistanceVectorRouting.py ExampleTopology.json
```

```
Tabelle di routing finali dopo la convergenza:
Nodo A:
  Distance Vector: {'A': 0, 'B': 1, 'D': 1, 'C': 3}
  Next Hop: {'A': None, 'B': 'B', 'D': 'D', 'C': 'B'}
Nodo B:
  Distance Vector: {'B': 0, 'A': 1, 'C': 2, 'D': 2}
  Next Hop: {'B': None, 'A': 'A', 'C': 'C', 'D': 'A'}
Nodo C:
  Distance Vector: {'C': 0, 'B': 2, 'D': 2, 'A': 3}
  Next Hop: {'C': None, 'B': 'B', 'D': 'D', 'A': 'B'}
Nodo D:
  Distance Vector: {'D': 0, 'A': 1, 'C': 2, 'B': 2}
  Next Hop: {'D': None, 'A': 'A', 'C': 'C', 'B': 'A'}
```

Figura 3: Esempio di output con topologia su file JSON