Relazione Elaborato Reti di Telecomunicazioni

Simulazione del Protocollo di Routing Distance Vector

Filippo Ferretti
Matricola 0001077544

11 dicembre 2024

Indice

1 Introduzione **2**
1.1 Obiettivi 2

2 Implementazione **3**
2.1 Inizializzazione della rete 3
2.2 Funzioni principali 3
2.3 Esecuzione del programma 5

3 Risultati **6**
3.1 Nodo A 6
3.2 Nodo B 6
3.3 Nodo C 6
3.4 Nodo D 7
3.5 Nodo E 7

4 Conclusioni **8**

Capitolo 1

Introduzione

Il protocollo di routing Distance Vector è uno dei metodi più semplici e basilari utilizzati per calcolare i percorsi più brevi tra nodi in una rete basato su Bellman-Ford. Questa relazione descrive la simulazione del protocollo utilizzando un programma Python. L'obiettivo è calcolare e aggiornare le tabelle di routing per ciascun nodo, mostrando il processo di convergenza alla rete ottimale.

1.1 Obiettivi

Gli obiettivi principali di questa simulazione sono:

- Implementare la logica di aggiornamento delle tabelle di routing per ciascun nodo.
- Gestire le tabelle di routing in base al protocollo Distance Vector.
- Simulare una rete con nodi e archi definiti e osservare la convergenza del protocollo.
- Fornire output delle tabelle di routing di tutti i nodi della rete.

Capitolo 2

Implementazione

La simulazione è stata realizzata in Python. Di seguito sono descritte le principali componenti del codice.

2.1 Inizializzazione della rete

La rete è rappresentata da:

- **Nodi**: Una lista di identificatori dei nodi (ad esempio, `["A", "B", "C", "D", "E"]`).
- **Archi**: Una lista di tuple che specificano i collegamenti tra nodi e i relativi costi (ad esempio, `[("A", "B", 2), ("B", "C", 1)]`).

2.2 Funzioni principali

- **print_routing_table**: Stampa la tabella di routing per un nodo, contente le destinazioni, ovvero gli altri nodi della rete, i next hop per raggiungere la relativa destinazione e il costo di tale operazione.

```
def print_routing_table(node, routing_table):
    print(f"Routing Table for Node {node}:")
    print("Destination\t\tCost\t\tNext Hop")
    for dest in sorted(routing_table.keys()):
        next_hop, cost = routing_table[dest]
        print(f"{dest}\t\t\t{cost}\t\t\t{next_hop}")
    print("\n")
```

- **update_routing_table**: Aggiorna la tabella di routing di un nodo basandosi sulle informazioni ricevute dai vicini.
  Per ogni vicino, controlla i percorsi disponibili e i relativi costi. Se trova un percorso più economico per raggiungere una destinazione passando per un vicino, aggiorna la propria tabella di routing.

```
def update_routing_table(node, neighbors, routing_table, tables):
    updated = False
```

```
3
4       # Itera sui vicini e il loro costo diretto
5       for neighbor, cost_to_neighbor in neighbors.items():
6           # Ottieni la tabella di routing del vicino
7           neighbor_table = tables[neighbor]
8           # Itera su ciascuna destinazione conosciuta dal vicino
9           for dest, (next_hop, cost_from_neighbor) in neighbor_table.
                items():
10              # Calcola il nuovo costo passando attraverso il vicino
11              new_cost = cost_to_neighbor + cost_from_neighbor
12
13              # Aggiorna la tabella di routing se:
14              # - La destinazione non e' nella tabella
15              # - Oppure il nuovo costo e' migliore
16              if dest not in routing_table or new_cost <
                    routing_table[dest][1]:
17                  routing_table[dest] = (neighbor, new_cost)
18                  updated = True
19
20      return updated
```

- **simulate_distance_vector_routing**: Simula il protocollo su una rete definita, iterando fino alla convergenza o al raggiungimento di un numero massimo di iterazioni. Inizializza le tabelle di routing di ogni nodo, poi per ogni iterazione passa attraverso ogni nodo e aggiorna la sua tabella di routing usando i dati dei vicini.
  Se non ci sono più aggiornamenti (convergenza), termina la simulazione in anticipo. Dopo ogni iterazione, stampa lo stato attuale delle tabelle di routing per tutti i nodi. Alla fine, restituisce le tabelle di routing finali
  All'interno di questa funzione sono stati inseriti anche controlli d'errore per quanto riguarda nodi e archi, per evitare sia che abbiano valori negativi, sia che appartengano alla rete.

```
1   def simulate_distance_vector_routing(nodes, edges):
2       # Controlla che ci siano nodi ed archi
3       if not nodes:
4           raise ValueError("La lista dei nodi non puo' essere vuota."
                )
5       if not edges:
6           raise ValueError("La lista degli archi non puo' essere
                vuota.")
7
8       # Calcolare le iterazioni massime come il numero di nodi
9       iterations = len(nodes)
10
11      # Inizializza la rete
12      neighbors = {node: {} for node in nodes}  # Vicini di ciascun
            nodo
13      tables = {node: {node: (node, 0)} for node in nodes}  # Tabelle
            di routing di ciascun nodo
14
15      for node1, node2, cost in edges:
16          if node1 not in nodes or node2 not in nodes:
17              raise ValueError(f"Edge ({node1}, {node2}) contains
                    unknown nodes.")
18          if cost < 0:
```

```
19              raise ValueError(f"Edge ({node1}, {node2}) has a
                    negative cost: {cost}.")
20          neighbors[node1][node2] = cost
21          neighbors[node2][node1] = cost
22
23      # Simula l'algoritmo per un massimo di `iterations` iterazioni
24      for _ in range(iterations):
25          updated = False
26          for node in nodes:
27              if update_routing_table(node, neighbors[node], tables[
                    node], tables):
28                  updated = True
29          # Se nessuna tabella e' stata aggiornata, termina prima
30          if not updated:
31              break
32
33      # Stampa le tabelle di routing finali
34      for node in nodes:
35          print_routing_table(node, tables[node])
```

2.3 Esecuzione del programma

Il codice Python è stato eseguito con i seguenti parametri stabiliti nel main:

- **Nodi**: `["A", "B", "C", "D", "E"]`
- **Archi**: `[("A", "B", 2), ("A", "C", 5), ("A", "E", 4), ("B", "C", 1), ("B", "D", 3), ("C", "D", 1), ("C", "E", 6), ("D", "E", 3),]`

Capitolo 3

Risultati

Dopo l'esecuzione del programma, le tabelle di routing finali sono state calcolate per ciascun nodo:

3.1 Nodo A

Destinazione	Costo	Next Hop
A	0	A
B	2	B
C	3	B
D	4	B
E	4	E

3.2 Nodo B

Destinazione	Costo	Next Hop
A	2	A
B	0	B
C	1	C
D	2	C
E	5	C

3.3 Nodo C

Destinazione	Costo	Next Hop
A	3	B
B	1	B
C	0	C
D	1	D
E	4	D

3.4 Nodo D

Destinazione	Costo	Next Hop
A	4	C
B	2	C
C	1	C
D	0	D
E	3	E

3.5 Nodo E

Destinazione	Costo	Next Hop
A	4	A
B	5	D
C	4	D
D	3	D
E	0	E

Capitolo 4

Conclusioni

La simulazione ha dimostrato che il protocollo Distance Vector è in grado di calcolare correttamente i percorsi più brevi tra nodi in una rete. Dopo alcune iterazioni, le tabelle di routing di tutti i nodi convergono a valori stabili. Il programma Python sviluppato può essere ulteriormente esteso per supportare reti più complesse o altri protocolli di routing.