Documentazione progetto di Reti di Telecomunicazione - Simulazione di un protocollo di routing

Michele Farneti: 0001080677

December 2024

Contents

Chapter 1

Introduzione

1.1 Generalità studente

- Studente: Michele Farneti
- Matricola: 0001080677
- Email: michele.farneti@studio.unibo.it

1.2 Traccia

Progetto Python: Simulazione di Protocollo di Routing: creare uno script Python che simuli un protocollo di routing semplice, come il Distance Vector Routing. Gli studenti implementeranno gli aggiornamenti di routing tra i nodi, con il calcolo delle rotte più brevi. Obiettivi: Implementare la logica di aggiornamento delle rotte, gestione delle tabelle di routing e calcolo delle distanze tra nodi. Consegne richieste: Codice Python ben documentato, output delle tabelle di routing per ogni nodo e relazione finale che spieghi il funzionamento dello script.

Chapter 2

Analisi soluzione

2.1 Descrizone

Al fine di simulare il funzionamento di un protocollo di routing, ho scelto di realizzare uno script python che simulasse il funzionamento del routing **Distance Vector**. In particolare, lo script permetterà di:

- Generare una network formata da un numero di nodi passato in input, interconnessi in casualmente da archi di peso random.
- Visualizzare mediante un grafo una rappresentazione della rete.
- Simulare una serie di iterazioni del protocollo di routing, le quali comprendono lo scambio dei distance vector e gli aggiornamenti delle tabelle di routing.
- Visualizzazione, per ciascun iterazione, l'evoluzione della tabella di routing di ciascun nodo.

2.2 Cenni teorici

2.2.1 Grafo di rete e shortest path routing

La rete può essere rappresentata sotto forma di un grafo nel quale **ogni nodo rappresenta un terminale o commutatore ed ogni arco rappresenta un collegamento. Il peso dell'arco rappresenterà il costo del**

collegamento, che può essere espresso mediante diverse metriche, come numero di hop o distanza geografica. Nel mondo IP, ciascun nodo manterrà una propria tabella di instradamento, per mezzo della quale avrà modo di tener traccia, per ogni destinazione, del **next hop**, ossia del nodo a cui inviare il pacchetto per instradarlo verso la direzione, e del **costo del collegamento**. Al momento dell'accessione dei nodi, ciascuno conoscerà soltanto le proprie interfacce, e ne utilizzerà le informazioni per popolare la propria tabella di routing. Al fine di implementare il **routing shortest path** verso una qualsiasi destinazione, ossia la popoloazione delle routing table di ciascun nodo con informazioni che permettano di raggiungere ogni altro nodo al costo minore posssibile, vengono implementati i **protocolli di routing**.

2.2.2 Routing distance vector

Questo protocollo di routing è semplice e bassa richiesta di risorse. È basato su Bellman-Ford in una versione dinamica e distribuita proposta da Ford-Fulkerson. L'algoritmo prevede che ciascun nodo scopra i propri vicini e la relativa distanza da se, dopodichè, ad ogni succesivo passo, ogni nodo comunicherà ai propri vicini il proprio distance vector: il vettore contenente la stima della sua distanza da tutti gli altri nodi della rete (quelli di cui è a conoscenza). La semplicità dell'algoritmo lo rende tuttavia possibile soggetto di diverse problematiche. Le principali sono:

- **Convergenza lenta**:L'algoritmo converge al più dopo un numero di passi pari al numero di nodi della rete, dunque se la rete è molto grande il tempo di convergenza può essere lungo. Ciò può essere un problema soprattuto qualora lo stato della rete cambi in un tempo inferiore a quello di convergenza dell'algoritmo.
- **Bouncing effect**: La rottura di un collegamento e la non immediatezza delle comunicazioni può portare all ricezione di distance vector con incongruenze temporanee, causando possibili rimpalli continui di pacchetti tra i router.
- **Bouncing effect**: La rottura di un collegamento ed il rimpallo dei dv tra due nodi può causare l'incremento infinito della distanza da un nodo.

Tra le possibili soluzioni sottoforma di meccanismi migliorativi che abbiamo per mitigare gli effetti di questi problematiche troviamo:

- **Split horizon** Qualora un nodo A utilizzi un nodo B per raggiungere una determinata destinazione, ometterà la propria dal dv che invierà da B la propria distanza da quella destinazione. Nella versione **wiht poisonus reverse** invece, la distanza viene posta all'infnito nel dv inviato.

- **Triggered update**: Un nodo invia immediatamente le informazioni a tutti i vicini qualora si verifichi una modifica della propria tabella di instradamento.

Chapter 3

Analisi del codice

3.1 Componenti modellate

3.1.1 Router

I nodi di cui si compone la rete sono rappresentati mediante oggetti di classe **Router**. Ogni router sarà caratterizzato da un nome che lo identificherà e conterrà al suo interno la tabella di routing, un set contenente i nomi dei router adiacenti ed un buffer per le routing table ricevute ad ogni iterazione dell'algoritmo.

Campi:

- **routing_table**: implementata sotto forma di dizionario *(destinazione: (costo, next hop))*. Costo e next hop sono espressi come nome del nodo.
- **neighbours**: Set contenente i nomi dei router adiacenti.
- **communciations_buffer**: implementato sotto forma di dizionario *(name, (destinazione: (costo, next hop)))*, tiene memorizzati i dv ricevuti di un nodo in un'iterazione di algoritmo.

```
self.name = name
self.routing_table = {}
self.neighbours = set()
self.communications_buffer = {}
```

Aggiunta vicini: Ad ogni nodo, in fase di inizializzazione, è possibile comunicare la presenza di un router a cui è direttamente collegato mediante il metodo **add_neighbour**, specificando il nome del nodo e il costo del collegamento. Il nodo viene così aggiunto al set *dei neighbours* e la relativa entry memorizzata nella *routing_table*. In questo caso, non verrà memorizzato un effetivo next hop ma nel campo verrà memorizzato **"DIRECT"**.

```
def add_neighbour(self, neighbour_name, cost):
        self.neighbours.add(neighbour_name)
        self.routing_table[neighbour_name] = (cost, "DIRECT")
```

Gestione DV Ricevuti: La funzione **handle_route** simula la gestione da parte del router della ricezione e di un entry dal dv di un altro router. Questo metodo può portare alla modifica o ad un aggiornamento della routing table. In particolare, oltre alla comunicazione della distanza e del costo, viene passato come parametro all'invocazione della funzione sotto il nome di **next_hop** il router da cui proviene il distance vector. Quest'ultimo verrà infatti impostato come next hop qualora la route comunicata sarà ritenuta conveniente. Il nuovo costo della route proposta è calcolato aggiungendo al costo nella entry del dv quello per raggiungere l'hop da cui é arrivato il dv. Se il nuovo costo sarà inferiore a quello della route giá presente nella tabella, la entry verrá aggiornata. In caso la entry non fosse presente, verrá semplicemente aggiunta. Il valore di ritorno della funzione è true se la tabella subisce un aggiornamento, false altrimenti.

Comunicazione: Al fine di simulare l'invio simultaneo dei dv tra i nodi, scelgo di processare le comunicazioni solo al termine dell'iterazione dell'algoritmo, così da simulare in maniera distinta ogni passo.
Soluzione: Implementazione di un buffer che mantiene i dv ricevuti da altri nodi e di un metodo che procede col processarli chiamato solo al termine delle comunicazioni dell'iterazione. Il valore di ritorno, True o False, permette di verificare se l'iterazione a portato o meno alla modifca della tabella di routing del nodo (True se ci sono state modifiche).

```
1     def handle_communications(self):
2         changes = False
3         for sender,distance_vector in
              self.communications_buffer.items():
4             for dest,(cost, next_hop) in
                  distance_vector.items():
5                 if self.handle_route(dest,cost,sender):
6                     changes = True
7         self.communications_buffer.clear
8         return changes
```

3.1.2 DV_Network

Classe che rapprsenta una network sulla quale sará possibile simulare il funzionamento del protocollo distance vector. Verranno mantenuti i singoli router che la costituiscono ed un grafo per rappresentare graficamente la distribuzione dei collegamenti.

Campi: Scelgo di memorizzare i router sotto forma di un dizionario che li indicizza per nome. Per il grafo utilizzo la classe **Graph** della libreria **networkx**.

```
1 self.routers = {} # Router nella network: {nome: Router}
2 self.graph = nx.Graph()  # Grafo di rete (ha solo scopi
      grafici)
```

Generazione network Il metodo **generate_netowrk** permette di generare una network di n nodi interconnessi in maniera casuale. Il parametro n va specificato alla chiamata della funzione , viene garantito che ogni router abbia almeno un collegamento con un altro router. I pesi dei collegamenti sono

assegnati casualmente. L'interconnessione tra i router, una volta aggiunti alla rete avviene mediante il metodo **connect_routers()**:

```
    def connect_routers(self, router1, router2, cost):
        self.routers[router1].add_neighbour(router2,cost)
        self.routers[router2].add_neighbour(router1,cost)
```

Parallelamente alla gestione degli oggetti router, nel metodo di generazione della network viene anche realizzata la rappresentazione grafica della stessa. Metodo **generate_network**:

```
    def generate_network(self,n):
        # Creazione degli n nodi del grafo e dei router
        for i in range(n):
            node_name = f"R{i+1}"
            new_router = Router(node_name)
            new_router.add_route_entry(node_name,0,"")
            self.add_router(new_router)
            self.graph.add_node(node_name)

        # Costruisco le connessioni iniziali per garantire
            ogni router abbia almeno una connsessione (peso
            random)
        for i in range(n - 1):
            node1_name = f"R{i+1}"
            node2_name = f"R{i+2}"
            cost = random.randint(1, 10)
            self.graph.add_edge(node1_name, node2_name,
                weight=cost)
            self.connect_routers(node1_name,node2_name,cost)

        # Aggiunta di archi casuali extra alla rete
        for i in range(n):
            for j in range(i + 1, n):
                node1_name = f"R{i+1}"
                node2_name = f"R{j+1}"
                if not self.graph.has_edge(node1_name,
                    node2_name) and random.choice([True,
                    False, False, False,False,False]): #Una
                    possibilita su 7 di generare un arco
                    extra tra due nodi
                    cost = random.randint(1, 10)
                    self.graph.add_edge(node1_name,
                        node2_name, weight=cost)
                    self.connect_routers(node1_name,
                        node2_name, cost)
```

Metodo **show** per la visualizzazione del grafo di rete:

```
    def show(self):
        # Layout del grafo
        pos = nx.spring_layout(self.graph)
        labels = nx.get_edge_attributes(self.graph, 'weight')

        # Creazione della griglia per il layout della
            finestra
        fig = plt.figure(figsize=(12, 80))

        # Disegno del grafo
        nx.draw(self.graph, pos, with_labels=True,
            node_color='lightblue', node_size=500,
            font_size=10)
        nx.draw_networkx_edge_labels(self.graph, pos,
            edge_labels=labels, font_color='red')

        plt.tight_layout()
```

Simulazione protocollo La simulazione del protocollo di routing viene effetuata mediante il metodo simulate della network. Viene posto un numero massimo max_iterations di iterazioni oltre al quale la simulazione verrà interrotta, altrimenti verranno eseguiti nuovi passi fino a quando non si giungerà a convergenza. La convergenza, verrà dichiarata raggiunta solo qualora un iterazione non porterà all'aggiornamento della tabella di routing di alcun nodo. In ciascuna iterazione, viene fatto comunicare ad ogni router il propio distance vector ad ogni router adiacente (metodo **communicate_dv()**). Al termine della comunicazione, ogni router procede ad elaboare i messaggi ed aggiornare la propria routing table (metodo **handle_communications()**). Viene inoltre stampato su riga di comando il contenuto delle routing table di ciascun nodo.

```
   def simulate(self):
        changes = True
        max_it = 20
        iterations = 0

        print(f"################### PARTENZA
            ###################")

        #Gestione delle comunicazioni ed update delle
            tabelle di routing di ciascuno
```

```
        for router in self.routers.values():
            print(router.get_printable_routing_table())

        while iterations < max_it and changes:
            #Comunicazione dei dv tra i router adiacenti
            for router in self.routers.values():
                for neighbour_name in
                    router.get_neighbours():
                     self.routers[neighbour_name]
                     .communicate_dv(router.get_name(),router
                     .get_routing_table())

            iterations = iterations + 1
            print(f"################### Iterazione:
                {iterations} ###################")

            #Gestione delle comunicazioni ed update delle
                tabelle di routing di ciascuno
            changes = False
            for router in self.routers.values():
                if router.handle_communications():
                    changes = True
                print(router.get_printable_routing_table())

            if not changes :
                print(f"Convergenza raggiunta in
                    {iterations} iterazioni!")
            else :
                print("Convergenza non ancora raggiunta!\n")

        print("--Fine simulazione")
```

3.1.3 Main

Il main del programma gestisce l'input da parte dell'utente del numero di nodi richiesto per la networkda generare, verificando che rientri in una soglia compresa tra i 2 ed i 16. Dopodichè procede con la generazione della network, stampa del grafo e simulazione del protocollo comprensiva di stampa delle tabelle di routing per ciascuna iterazione.

```
n = int(input("Inserisci il numero di nodi del grafo (min 2,
    max 16): "))

    if n > 2 and n <= 16:
```

```
4           network = DV_network()
5           network.generate_network(n)
6           network.show()
7           network.simulate()
8           plt.show()
9       else:
10          print("Il numero di nodi deve essere maggiore di
                zero.")
11      sys.exit(0)
```

Chapter 4

Esecuzione di prova

Viene ora mostrato un esempio di esecuzione dello script. In particolare viene simulato il protocollo su una rete di 6 router.

```
Inserisci il numero di nodi del grafo (min 2, max 16): 6
```

Figure 4.1: Input d tasiera

Una volta ricevuto l'input, viene generata la network ed il relativo grafo:

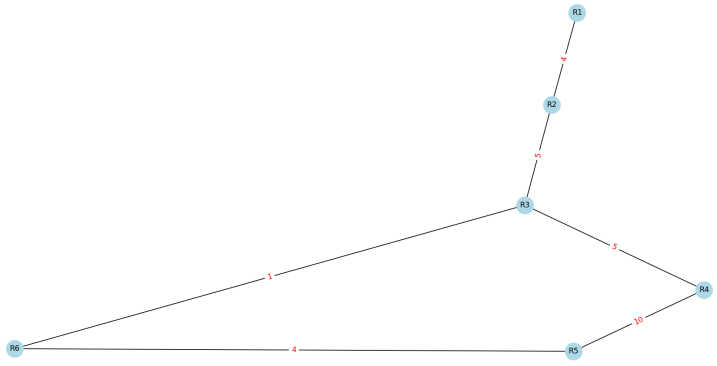

Figure 4.2: Grafo rappresentante la network dei 6 nodi generata casualmente

Di seguito viene mostrata la tabella di routing inziale dei vari nodi. Ciascuno arvrà come sole entry se stesso, a distanza zero, ed i router confinanti.

```
################### PARTENZA ###################
Routing table di R1:
Destinazione    Costo      Next Hop
---------------------------------------
R1              0
R2              4          DIRECT

Routing table di R2:
Destinazione    Costo      Next Hop
---------------------------------------
R2              0
R1              4          DIRECT
R3              5          DIRECT

Routing table di R3:
Destinazione    Costo      Next Hop
---------------------------------------
R3              0
R2              5          DIRECT
R4              5          DIRECT
R6              1          DIRECT

Routing table di R4:
Destinazione    Costo      Next Hop
---------------------------------------
R4              0
R3              5          DIRECT
R5              10         DIRECT

Routing table di R5:
Destinazione    Costo      Next Hop
---------------------------------------
R5              0
R4              10         DIRECT
R6              4          DIRECT

Routing table di R6:
Destinazione    Costo      Next Hop
---------------------------------------
R6              0
R5              4          DIRECT
R3              1          DIRECT
```

Figure 4.3: Tabelle di routing iniziali

Dopo la prima iterazione dell'algoritmo, la convergenza non è stata ancora raggiunta, le tabelle sono le seguenti.

```
Routing table di R1:
Destinazione    Costo      Next Hop
-----------------------------------------
R1              0
R2              4          DIRECT
R3              9          R2

Routing table di R2:
Destinazione    Costo      Next Hop
-----------------------------------------
R2              0
R1              4          DIRECT
R3              5          DIRECT
R4              10         R3
R6              6          R3

Routing table di R3:
Destinazione    Costo      Next Hop
-----------------------------------------
R3              0
R2              5          DIRECT
R4              5          DIRECT
R6              1          DIRECT
R1              9          R2
R5              5          R6
```

Figure 4.4: Tabelle di routing iterazione 1 (pt.1)

```
Routing table di R4:
Destinazione    Costo       Next Hop
----------------------------------------
R4              0
R3              5           DIRECT
R5              10          DIRECT
R2              10          R3
R6              6           R3
R1              14          R3

Routing table di R5:
Destinazione    Costo       Next Hop
----------------------------------------
R5              0
R4              10          DIRECT
R6              4           DIRECT
R3              5           R6
R2              20          R4
R1              24          R4

Routing table di R6:
Destinazione    Costo       Next Hop
----------------------------------------
R6              0
R5              4           DIRECT
R3              1           DIRECT
R2              6           R3
R4              6           R3
R1              10          R3

Convergenza non ancora raggiunta!
```

Figure 4.5: Tabelle di routing iterazione 1 (pt.2)

L'algoritmo raggiunge la convergenza alla quarta iterazione, le tabelle di routing definitive sono le seguenti:

```
################## Iterazione: 4 ##################
Routing table di R1:
Destinazione    Costo      Next Hop
---------------------------------------
R1              0
R2              4          DIRECT
R3              9          R2
R4              14         R2
R6              10         R2
R5              14         R2

Routing table di R2:
Destinazione    Costo      Next Hop
---------------------------------------
R2              0
R1              4          DIRECT
R3              5          DIRECT
R4              10         R3
R6              6          R3
R5              10         R3

Routing table di R3:
Destinazione    Costo      Next Hop
---------------------------------------
R3              0
R2              5          DIRECT
R4              5          DIRECT
R6              1          DIRECT
R1              9          R2
R5              5          R6
```

Figure 4.6: Tabelle di routing definitive (pt.1)

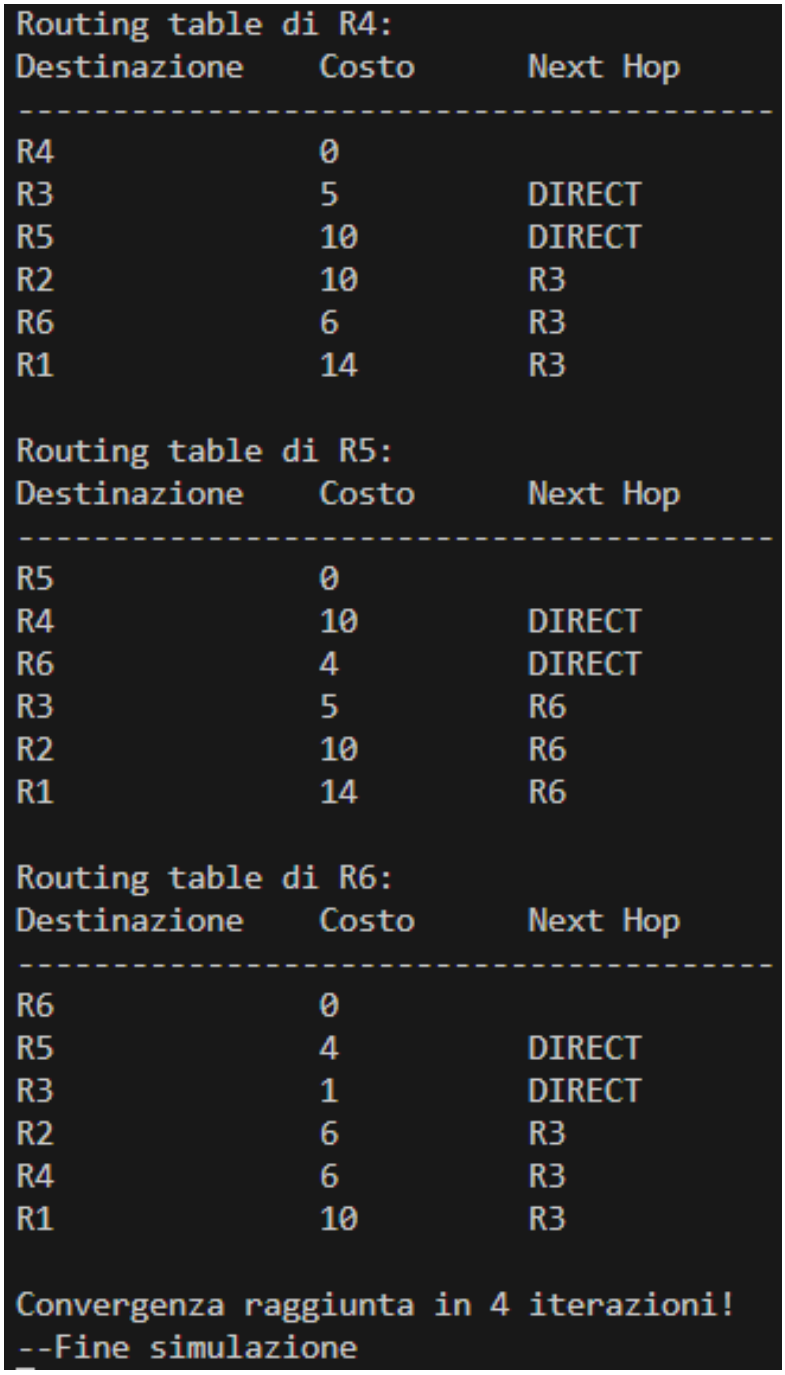

Figure 4.7: Tabelle di routing definitive (pt.2)