# Distance Vector Routing

Andrea Baldazzi : 0001071149

10 dicembre 2024

# Indice

# Capitolo 1

# Introduzione

L'elaborato simula il funzionamento del protocollo *Distance Vector Routing*, implementato con un semplice programma Python.

# Capitolo 2

# Funzionamento

Ogni router virtuale tiene traccia delle rotte migliori associando ad ogni nodo (router) da esso raggiungibile un costo e il primo nodo da cui passare per raggiungerlo. La simulazione prevede che i router scambino con i vicini il proprio Distance Vector e cerchino di migliorare il proprio in seguito alla ricezione degli altri Distance Vector.

Sono proposte due soluzioni, una semplice e una con meccanismi *Split Horizon* e *Triggered Update*.

## 2.1 Soluzione semplice

In questa implementazione viene proposto il meccanismo di *Distance Vector Routing* così com'è senza meccanismi per facilitare la convergenza. Appropriata documentazione è presente nel codice Python sia per le classi che per i metodi.

### Creazione Router

Nel programma un router è gestito come un oggetto con diverse funzionalità. Al momento della creazione viene creata una tabella di routing vuota, dove l'unica rotta è quella verso se stesso.

```
1 class Router:
2
3     def __init__(self, name: str)
```

2.1: Creazione router

### Aggiunta router vicino

Il router accetta l'aggiunta di un router vicino che viene subito aggiunto alla propria tabella di routing.

```
def add_neighbour(self, neighbour: str, distance: int)
```

2.2: Aggiunta router vicino

### Aggiornamento tabella

Il router accetta un Distance Vector proveniente da un vicino e lo utilizza per aggiornare la propria tabella di routing. In particolare, vengono aggiunte le destinazioni non conosciute e aggiornate quelle già presenti nel caso venga fornita una rotta con minore distanza. Viene restituito True nel caso la tabella è stata modificata, False altrimenti.

```
def update_table(self, neighbour: str, neighbour_table: dict[str,
    tuple[int, str]])
```

2.3: Aggiornamento tabella

### Creazione rete

Nel programma fornito la rete è implementata con un oggetto con diverse funzionalità. Al momento della creazione è priva di router e di collegamenti.

```
class Network:

    def __init__(self)
```

2.4: Creazione rete

### Aggiunta collegamento tra router

La rete permette di creare un nuovo collegamento tra due router. Nel caso essi non siano già presenti nella rete, vengono creati.

```
def add_edge(self, router1: str, router2: str, distance: int)
```

2.5: Aggiunta colegamento tra router

### Scambio dei distance vector

La rete permette di far scambiare tra i router vicini i propri Distance Vector. Il processo viene ripetuto un numero di volte pari al numero di nodi nel caso il valore `steps` non venga fornito.

```
def update_tables(self, steps=None)
```

2.6: Scambio dei Distance Vector

## 2.2 Soluzione avanzata

In questa implementazione viene modificato il programma semplice sopra descritto aggiungendo due meccanismi della rete per facilitare la convergenza della rete:

**Split Horizon:** Sfruttando la versione con Poisonous Reverse, se A per raggiungere C deve passare da B, dice a B che la sua distanza da C è infinita.

**Triggered Update:** Ogni volta che un router aggiorna il suo Distance Vector lo invia a tutti i suoi vicini.

### Invio di un Distance Vector

Con questo metodo un solo Distance Vector è inviato ai vicini, ovvero quello di `router`. Il metodo tiene traccia di quali vicini hanno modificato la tabella, richiamandosi ricorsivamente su di essi per implementare *Triggered Update*. Inoltre, ogni Distance Vector inviato è "avvelenato" per implementare *Split Horizon*.

```
1 def send_table(self, router: str)
```

2.7: Invio di un Distance Vector

### Invio di tutti i Distance Vector

Richiama `send_table()` su tutti i nodi della rete un numero `steps` di volte, pari al numero di router nella rete se non fornito.

```
1 def update_tables(self, steps=None)
```

2.8: Invio di tutti i Distance Vector

# Capitolo 3

# Utilizzo

## 3.1 Avvio applicazione

Per avviare l'applicazione è sufficiente, previa installazione di un interprete Python sulla propria macchina, avviare uno dei due script `DV-routing-simulation.py` o `DV+SH+TU-routing-simulation.py`. I due script sono stati testati con Python 3.11.8.

## 3.2 Esempio dimostrativo

Nei due script è precaricato un esempio di utilizzo, identico in entrambi. La rete è configurata nel seguente modo:

```
net = Network()
net.add_edge("A", "B", 7)
net.add_edge("B", "C", 5)
net.add_edge("C", "D", 20)
net.add_edge("A", "D", 2)
```

Ne consegue che la struttura risultante sia la seguente:

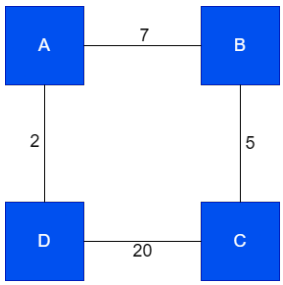


Figura 3.1: Schema della rete di esempio

Forziamo ora lo scambio dei Distance Vector:

```
net.update_tables()
net.print_tables()
```

L'output in console è il seguente:

```
Routing table for node A:
  Destination: A, Distance: 0, Next Hop: A
  Destination: B, Distance: 7, Next Hop: B
  Destination: D, Distance: 2, Next Hop: D
  Destination: C, Distance: 12, Next Hop: B
Routing table for node B:
  Destination: B, Distance: 0, Next Hop: B
  Destination: A, Distance: 7, Next Hop: A
  Destination: C, Distance: 5, Next Hop: C
  Destination: D, Distance: 9, Next Hop: A
Routing table for node C:
  Destination: C, Distance: 0, Next Hop: C
  Destination: B, Distance: 5, Next Hop: B
  Destination: D, Distance: 14, Next Hop: B
  Destination: A, Distance: 12, Next Hop: B
Routing table for node D:
  Destination: D, Distance: 0, Next Hop: D
  Destination: C, Distance: 14, Next Hop: A
  Destination: A, Distance: 2, Next Hop: A
  Destination: B, Distance: 9, Next Hop: A
```

Proviamo adesso a cambiare il peso di un arco e ad aggiornare nuovamente le tabelle:

```
net.add_edge("C", "D", 1) # modifies an existing edge if it exists
net.update_tables()
net.print_tables()
```

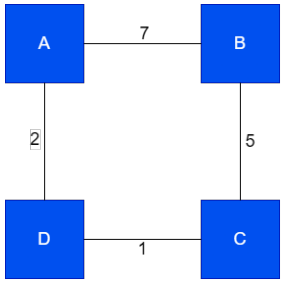


Figura 3.2: Schema della rete dopo la modifica

L'output in console dopo la modifica è il seguente:

```
Routing table for node A:
  Destination: A, Distance: 0, Next Hop: A
  Destination: B, Distance: 7, Next Hop: B
  Destination: D, Distance: 2, Next Hop: D
  Destination: C, Distance: 3, Next Hop: D
Routing table for node B:
  Destination: B, Distance: 0, Next Hop: B
  Destination: A, Distance: 7, Next Hop: A
  Destination: C, Distance: 5, Next Hop: C
  Destination: D, Distance: 6, Next Hop: C
Routing table for node C:
  Destination: C, Distance: 0, Next Hop: C
  Destination: B, Distance: 5, Next Hop: B
  Destination: D, Distance: 1, Next Hop: D
  Destination: A, Distance: 3, Next Hop: D
Routing table for node D:
  Destination: D, Distance: 0, Next Hop: D
  Destination: C, Distance: 1, Next Hop: C
  Destination: A, Distance: 2, Next Hop: A
  Destination: B, Distance: 6, Next Hop: C
```

Come si può notare, dopo l'abbassamento del costo dell'arco tra C e D tutti i router hanno aggiornato il proprio Distance Vector per tenere traccia di rotte più brevi.

# Capitolo 4

# Considerazioni finali

Il corretto funzionamento dei due programmi è stato testato su una macchina Windows con interprete Python 3.11.8. Si è verificata la creazione di nuovi router, il collegamento tra essi, l'invio dei Distance Vector e la modifica del peso degli archi già presenti. Per semplicità non è stato implementato un meccanismo di rimozione degli archi, tuttavia con la soluzione fornita è facilmente implementabile.