Simulazione di Protocollo di Routing

Margherita Zanchini - 0001081989

December 2024

1 Introduzione

Il progetto simula il Distance Vector Routing, un protocollo semplice basato sull'algoritmo di Bellman-Ford. Il programma mostra la configurazione della rete e come le tabelle di routing di ogni nodo cambiano fino a trovare il percorso più breve per ogni router della rete.

2 Configurazione iniziale e generazione della rete

Il programma può essere avviato da terminale con il comando *python main.py*

Di deafult verrà generato un grafo della rete strutturato come nella seguente immagine:

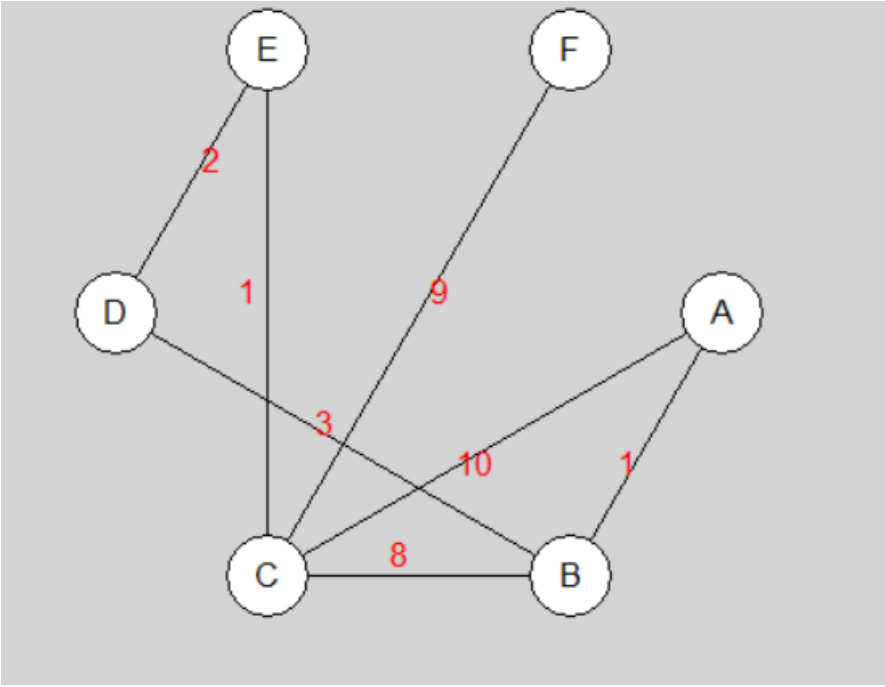

Figura 1: Grafo Rete Default

Se invece si vuole generare una rete casuale, aprire il file *main.py*, trovare la funzione chiamata *main*:

```
7  def main():
8      global routers  # dichiarazione della variabile globale
9      network = Network()
10
11     # comment the second line and uncomment the first line to create a random network
12     #routers = network.create_random_network()
13     routers = network.create_network()
14
```

Figura 2: Creazione Network

poi commentare la riga 13 dove è chiamata la funzione *create_network()* e decommentare la riga 12 dove è chiamata la funzione *create_random_network()*, in questo modo il programma genererà una rete casuale con un numero di router N compreso tra 3 e 10, il numero di archi sarà invece compreso tra *N-1* e $\frac{N(N-1)}{2}$. I collegamenti tra i router e i pesi di ogni arco sono anch'essi assegnati in modo randomico.

3 Svolgimento del Programma

Dopo aver avviato il programma con il comando *python main.py* si aprirà un'interfaccia grafica che si presenta in questo modo:

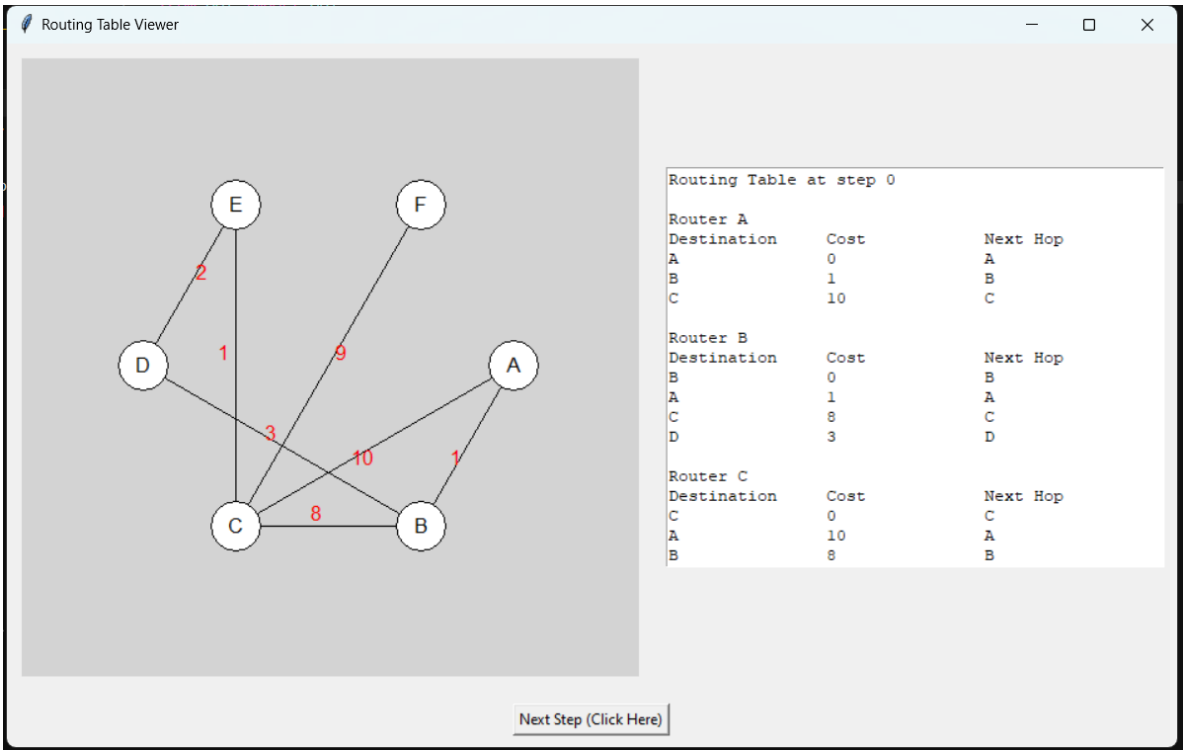

Figura 3: GUI

A sinistra è rappresentato il grafo della rete, mentre a destra si trovano le tabelle di routing di ogni nodo a ogni "step" (scorrere per vedere tutte le tabelle).

Allo step 0, come previsto dall'algoritmo, ogni router conosce solo sè stesso e i suoi vicini diretti. Per avanzare allo step successivo cliccare il pulsante in basso *Next Step*. Durante ogni "step" ogni router riceve i distance vector dai suoi vicini immediati e aggiorna la propria tabella di routing di conseguenza.

Si continua a cliccare il pulsante per aggiornare le tabelle di routing dei router fino a raggiungere lo stato finale, ovvero lo stato in cui ogni nodo ha calcolato il percorso più breve per raggiungere tutti gli altri router della rete. A questo punto il pulsante *Next Step* viene disabilitato.

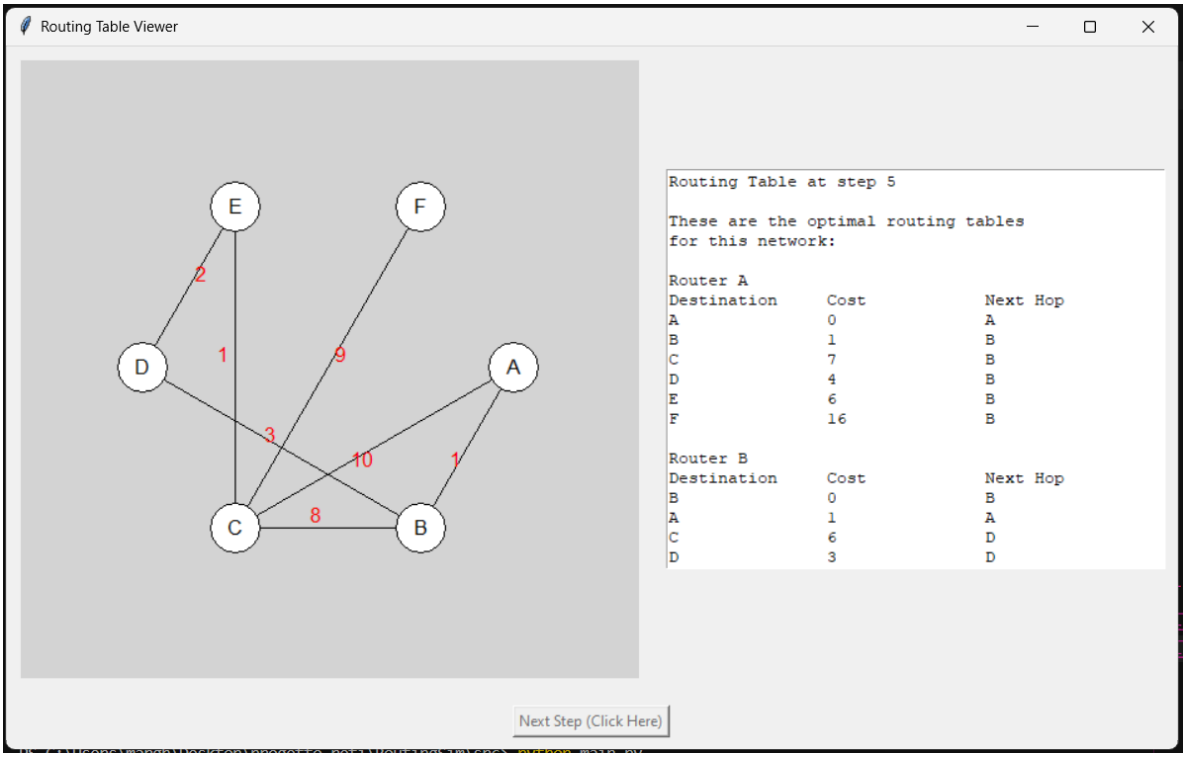

Figura 4: GUI con tabelle ottimali

4 Algoritmo Distance Vector e Implementazione

L'algoritmo del Distance Vector funziona così:

- Per prima cosa ogni nodo scopre i propri vicini e calcola la distanza tra sè stesso e ciascuno di loro
- Ad ogni passo ciascun nodo riceve poi dai vicini un vettore contenente la stima delle loro distanze da tutti gli altri nodi della rete (quelli di cui sono a conoscenza).
- Utilizzando queste informazioni calcola il percorso minimo da sè stesso a tutti gli altri nodi

Guardiamo ora come il programma implementa l'algoritmo del Distance Vector. La logica dei router è definita nella classe *Router*, contenuta nel file *router.py*. La gestione della tabella di routing è invece compresa nel file *routing_table.py* che contiene la classe *RoutingTable*, essa ha un metodo *add_route*, il quale consente sia l'aggiunta di una nuova rotta, sia la modifica di quelle già esistenti.

Il cuore dell'algoritmo è implementato nella classe *Router* ed è il metodo chiamato *update_distance_vector* e si occupa di aggiornare la tabella di routing del router stesso:

```python
#update the distance vector of the router with the Dv of the neighbors
def update_distance_vector(self) -> bool:
    update = False
    for router in self.neighbors:
        neighbor_distance_vector = router.get_distance_vector() #takes the DV of the neighbor
        for destination, cost in neighbor_distance_vector.items():
            if destination in self.my_table.table: #chek if the destination is already in the routing table
                if self.my_table.table[destination][0] > cost[0] + self.neighbors[router]: #if it is in the
                    self.my_table.add_route(destination, cost[0] + self.neighbors[router], router.name) #upd
                    update = True
            else: #if the destination is not in the table add it
                self.my_table.add_route(destination, cost[0] + self.neighbors[router], router.name)
                update = True
    return update

def get_distance_vector(self):
    return self.my_table.table
```

Figura 5: *update_distance_vector*

Questo metodo analizza i distance vector degli immediati vicini, aggiungendo alla propria tabella i nodi che ancora non conosce e aggiornando i percorsi per i nodi già raggiungibili se trova un percorso più breve. Ogni volta che si clicca il pulsante *Next Step* viene attivata una funzione:

```python
#function called when the button is clicked, it updates the distance vector of all routers
def distance_vector_routing(self):
    self.number_of_steps += 1
    #every roueter updates his DV
    self.update = False
    for router in self.routers:
        if(router.update_distance_vector()):
            self.update = True #update is true if at least one DV changed
    self.print_routing_tables()
    if(not self.update):
        self.button_next.config(state=tk.DISABLED)
```

Figura 6: distance_vector_routing

questa fa sì che tutti i router eseguano il loro metodo *update_distance_vector*, permettendo a tutti i router di aggiornare le loro distanze con gli altri nodi in base alle informazioni ricevute dai vicini.