# Implementazione del Protocollo Distance Vector

Luca Venturini

December 11, 2024

## Contents

# 1 Introduzione

Il progetto implementa il protocollo di routing *Distance Vector*, un algoritmo utilizzato nei sistemi di rete per determinare i percorsi più brevi tra nodi. L'algoritmo sfrutta la trasmissione periodica delle informazioni di distanza tra nodi adiacenti, aggiornando progressivamente le tabelle di routing fino al raggiungimento di una convergenza.

Lo scopo di questo progetto è:

- Simulare il funzionamento del protocollo Distance Vector su un grafo casuale.
- Verificare la correttezza delle tabelle di routing utilizzando l'algoritmo di Dijkstra come riferimento.

# 2 Metodologia

L'implementazione è stata sviluppata in *Python* utilizzando la libreria `networkx` per la gestione del grafo e `matplotlib` per la sua visualizzazione.

Il programma segue i seguenti passaggi:

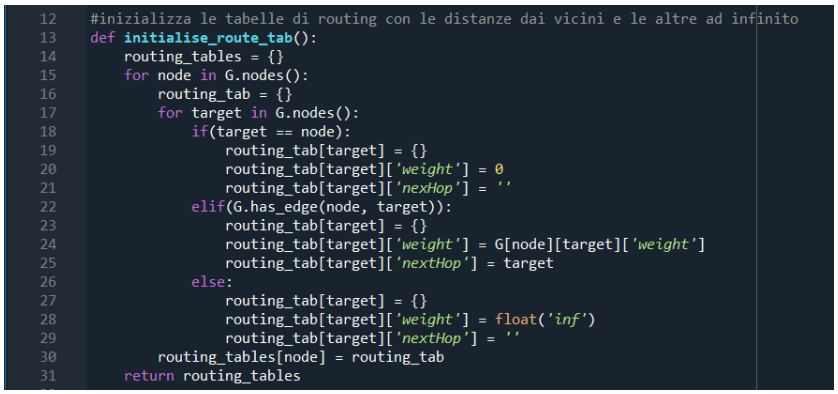
1. **Creazione del Grafo:** Viene generato un grafo casuale con il modello *Erdős-Rényi*. Gli archi sono assegnati con pesi casuali compresi tra 1 e 10.

```
80  G = nx.erdos_renyi_graph(n=num_nodes, p=probability)
81
82  while True:
83      #Crea un grafo casuale con 10 nodi
84      G = nx.erdos_renyi_graph(n=num_nodes, p=probability)
85
86      # Aggiunge dei pesi casuali agli archi appena creati
87      for (u, v) in G.edges():
88          G[u][v]['weight'] = random.randint(1, 10)  # Pesi casuali tra 1 e 10
89      #verifica che il grafo sia connesso
90      if(nx.is_connected(G)):
91          break
```

Figure 1: Codice creazione grafo

2. **Inizializzazione delle Tabelle di Routing:** Ogni nodo inizializza una tabella di routing dove:

   - La distanza verso se stesso è 0.
   - La distanza verso i nodi vicini è pari al peso dell'arco che li collega.
   - La distanza verso tutti gli altri nodi è impostata a infinito ($\infty$).



```
#inizializza le tabelle di routing con le distanze dai vicini e le altre ad infinito
def initialise_route_tab():
    routing_tables = {}
    for node in G.nodes():
        routing_tab = {}
        for target in G.nodes():
            if(target == node):
                routing_tab[target] = {}
                routing_tab[target]['weight'] = 0
                routing_tab[target]['nexHop'] = ''
            elif(G.has_edge(node, target)):
                routing_tab[target] = {}
                routing_tab[target]['weight'] = G[node][target]['weight']
                routing_tab[target]['nextHop'] = target
            else:
                routing_tab[target] = {}
                routing_tab[target]['weight'] = float('inf')
                routing_tab[target]['nextHop'] = ''
        routing_tables[node] = routing_tab
    return routing_tables
```

Figure 2: Codice inizializzazione delle tabelle di routing

3. **Aggiornamento delle Tabelle di Routing:** Per ogni nodo nel grafo vengono comparate le tabelle di routing con quelle dei vicini, se per una certa destinazione viene trovato un cammino piu corto allora la distanza dalla data destinazione si aggiornata e viene impostato il vicino come next hop.

```
#Aggiorna le tabelle di routing
def update_routing_tab():
    changes = False
    for node in G.nodes():
        for neightbour in G.neighbors(node):
            for target in G.nodes():
                #Se la nuova distanza è minore di quella presente la cambia
                if(routing_tables[node][target]['weight'] > routing_tables[neightbour][target]['weight'] + routing_tables[node][neightbour]['weight']):
                    routing_tables[node][target]['weight'] = routing_tables[neightbour][target]['weight'] + routing_tables[node][neightbour]['weight']
                    routing_tables[node][target]['nextHop'] = neightbour
                    changes = True;
    #ritorna vero tutte le volte che viene eseguito almeno un cambiamento
    return changes
```

Figure 3: Codice di aggiornamento delle distanze delle tabelle

4. **Convergenza:** Gli aggiornamenti delle tabelle sono eseguiti ogni 2 secondi e continuano fino a quando non vengono più rilevati aggiornamenti.

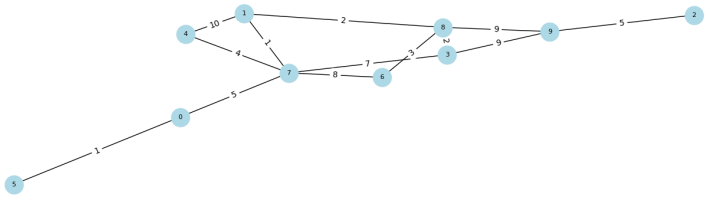
5. **Verifica:** I risultati vengono confrontati con i percorsi calcolati dall'algoritmo di Dijkstra, fornito dalla libreria `networkx`. Alla fine del programma viene fornito il disegno del grafo e le risultanti tabelle di routing.

```
66  #Verifica che te tabele risultanti siano corrette
67  def verify_tables():
68      print("\nverifica in corso")
69      for source in G.nodes():
70          for target in G.nodes():
71              #compara il risultato con quello della funzione dijkstra_path_length
72              if(nx.dijkstra_path_length(G, source, target) != routing_tables[source][target]['weight']):
73                  print("distanza sbagliata", nx.dijkstra_path_length(G, source, target), "!=" , routing_tables[source][target]['weight'])
74      print("verifica completata")
75
```

Figure 4: Codice di verifica delle distanze calcolate

# 3 Risultati e Verifica

Il programma genera un grafo casuale, aggiorna le tabelle di routing fino alla convergenza e verifica i risultati confrontandoli con l'algoritmo di Dijkstra. In seguito, riportato un esempio di grafo e tabelle di routing:




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | {'weight': 0, 'nextHop': ''} | {'weight': 6, 'nextHop': 7} | {'weight': 22, 'nextHop': 7} | {'weight': 10, 'nextHop': 7} | {'weight': 9, 'nextHop': 7} | {'weight': 1, 'nextHop': 5} | {'weight': 11, 'nextHop': 7} | {'weight': 5, 'nextHop': 7} | {'weight': 8, 'nextHop': 7} | {'weight': 17, 'nextHop': 7} |
| 1 | {'weight': 6, 'nextHop': 7} | {'weight': 0, 'nextHop': ''} | {'weight': 16, 'nextHop': 8} | {'weight': 4, 'nextHop': 8} | {'weight': 5, 'nextHop': 7} | {'weight': 7, 'nextHop': 7} | {'weight': 5, 'nextHop': 8} | {'weight': 1, 'nextHop': 7} | {'weight': 2, 'nextHop': 8} | {'weight': 11, 'nextHop': 8} |
| 2 | {'weight': 22, 'nextHop': 9} | {'weight': 16, 'nextHop': 9} | {'weight': 0, 'nextHop': ''} | {'weight': 14, 'nextHop': 9} | {'weight': 21, 'nextHop': 9} | {'weight': 23, 'nextHop': 9} | {'weight': 17, 'nextHop': 9} | {'weight': 17, 'nextHop': 9} | {'weight': 14, 'nextHop': 9} | {'weight': 5, 'nextHop': 9} |
| 3 | {'weight': 10, 'nextHop': 8} | {'weight': 4, 'nextHop': 8} | {'weight': 14, 'nextHop': 9} | {'weight': 0, 'nextHop': ''} | {'weight': 9, 'nextHop': 8} | {'weight': 11, 'nextHop': 7} | {'weight': 5, 'nextHop': 8} | {'weight': 5, 'nextHop': 8} | {'weight': 2, 'nextHop': 8} | {'weight': 9, 'nextHop': 9} |
| 4 | {'weight': 9, 'nextHop': 7} | {'weight': 5, 'nextHop': 7} | {'weight': 21, 'nextHop': 1} | {'weight': 9, 'nextHop': 1} | {'weight': 0, 'nextHop': ''} | {'weight': 10, 'nextHop': 7} | {'weight': 10, 'nextHop': 1} | {'weight': 4, 'nextHop': 7} | {'weight': 7, 'nextHop': 1} | {'weight': 16, 'nextHop': 1} |
| 5 | {'weight': 1, 'nextHop': 0} | {'weight': 7, 'nextHop': 0} | {'weight': 23, 'nextHop': 0} | {'weight': 11, 'nextHop': 0} | {'weight': 10, 'nextHop': 0} | {'weight': 0, 'nextHop': ''} | {'weight': 12, 'nextHop': 0} | {'weight': 6, 'nextHop': 0} | {'weight': 9, 'nextHop': 0} | {'weight': 18, 'nextHop': 0} |
| 6 | {'weight': 11, 'nextHop': 8} | {'weight': 5, 'nextHop': 8} | {'weight': 17, 'nextHop': 8} | {'weight': 5, 'nextHop': 8} | {'weight': 10, 'nextHop': 8} | {'weight': 12, 'nextHop': 7} | {'weight': 0, 'nextHop': ''} | {'weight': 6, 'nextHop': 8} | {'weight': 3, 'nextHop': 8} | {'weight': 12, 'nextHop': 8} |
| 7 | {'weight': 5, 'nextHop': 0} | {'weight': 1, 'nextHop': 1} | {'weight': 17, 'nextHop': 1} | {'weight': 5, 'nextHop': 1} | {'weight': 4, 'nextHop': 4} | {'weight': 6, 'nextHop': 0} | {'weight': 6, 'nextHop': 1} | {'weight': 0, 'nextHop': ''} | {'weight': 3, 'nextHop': 1} | {'weight': 12, 'nextHop': 1} |
| 8 | {'weight': 8, 'nextHop': 1} | {'weight': 2, 'nextHop': 1} | {'weight': 14, 'nextHop': 9} | {'weight': 2, 'nextHop': 3} | {'weight': 7, 'nextHop': 1} | {'weight': 9, 'nextHop': 1} | {'weight': 3, 'nextHop': 6} | {'weight': 3, 'nextHop': 1} | {'weight': 0, 'nextHop': ''} | {'weight': 9, 'nextHop': 9} |
| 9 | {'weight': 17, 'nextHop': 8} | {'weight': 11, 'nextHop': 8} | {'weight': 5, 'nextHop': 2} | {'weight': 9, 'nextHop': 3} | {'weight': 16, 'nextHop': 8} | {'weight': 18, 'nextHop': 8} | {'weight': 12, 'nextHop': 8} | {'weight': 12, 'nextHop': 8} | {'weight': 9, 'nextHop': 8} | {'weight': 0, 'nextHop': ''} |

Figure 5: Grafo generato con pesi casuali e tabella di `Distance` Vector

# 4 Conclusioni

L'implementazione dimostra il funzionamento del protocollo di routing Distance Vector anche se non vengono presi in considerazione gli svantaggi di questo, ovvero una partenza lenta e il tempo di convergenza.