Progetto Python: Simulazione di Protocollo di Routing

Nome: Nikolai
Cognome: Zanni
Matricola: 0001069041

Nome: Emily
Cognome: Pedini
Matricola: 0001081969

Nome: Benedetta
Cognome: Barone
Matricola: 0001081612

DESCRIZIONE:
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.

INTRODUZIONE:
Questo progetto, scritto in Python, ha l'obiettivo di simulare un protocollo di Distance Vector Routing (DVR), ricreando un ambiente di rete in cui i nodi determinano autonomamente i percorsi più brevi tra loro. Nel Distance Vector Routing, ogni nodo utilizza una metodologia distribuita per aggiornare e mantenere le proprie tabelle di instradamento tramite lo scambio periodico di informazioni con i suoi vicini.
Ogni nodo è implementato come un'istanza della classe Nodo e mantiene una propria tabella di instradamento che include i costi e i percorsi per raggiungere tutti gli altri nodi.
La simulazione si evolve fino a raggiungere uno stato di convergenza, cioè quando tutte le tabelle di instradamento si stabilizzano. Questo significa che ogni nodo ha la conoscenza del percorso ottimale per raggiungere ogni altro nodo.
Abbiamo progettato questa simulazione in modo che ogni passaggio del processo sia chiaro e ben documentato, rendendo il progetto un utile strumento per capire concretamente il funzionamento del protocollo di routing distribuito.

FUNZIONAMENTO DEL DISTANCE VECTOR:
Il protocollo Distance Vector Routing, che abbiamo implementato, si basa sull'algoritmo di Bellman-Ford per determinare i percorsi più brevi all'interno di una rete distribuita.
Vediamo le fasi principali che caratterizzano il funzionamento di questo protocollo:

1) *Algoritmo Bellman-Ford*: ogni nodo aggiorna iterativamente la sua stima delle distanze a tutti gli altri nodi basandosi sulle informazioni ricevute dai suoi vicini.

La tabella di instradamento viene mantenuta sotto forma di mappa in cui ogni chiave è una destinazione e ogni valore è una coppia contenente il costo e il prossimo passo. Ad ogni aggiornamento ricevuto da un vicino, il nodo verifica se il nuovo percorso rappresenta un miglioramento e aggiorna la sua tabella di conseguenza tramite il metodo *aggiorna_tabella.*

2) *Distribuzione locale*: ogni nodo invia periodicamente ai suoi vicini il proprio vettore delle distanze tramite il metodo *invia_vettore_distanze*. Questo permette di condividere le informazioni più aggiornate sui percorsi. Questo processo distribuito assicura che i nodi collaborino per trovare le rotte più efficienti.
3) *Convergenza graduale*: la rete raggiunge uno stato di convergenza dopo un certo numero di iterazioni. All'inizio ogni nodo conosce solo le distanze dei suoi vicini diretti. Con ogni iterazione, i nodi aggiornano le proprie tabelle di instradamento fino a quando non ci sono più cambiamenti. Questo stato stabile rappresenta la convergenza.

STRUTTURA DEL CODICE:

Il progetto è organizzato in modo modulare per facilitare la comprensione e l'espandibilità. Di seguito vengono descritti i componenti principali del codice:

1) *Classe Nodo*:
    - *Inizializzazione*: ogni nodo è rappresentato da un'istanza della classe Nodo, contenente un nome, una tabella di instradamento inizialmente vuota, e una lista di vicini diretti con i relativi costi.
    - *Aggiornamento della tabella*: la funzione *aggiorna_tabella* consente al nodo di aggiornare i percorsi verso tutti gli altri nodi sulla base delle informazioni ricevute dai vicini.
    - *Invio vettore delle distanze*: ogni nodo, tramite il metodo *invia_vettore_distanze*, genera il proprio vettore delle distanze da condividere con i vicini.
2) *Funzioni Principali*:
    - *simula_instradamento*: questa funzione simula il protocollo DVR sull'intero grafo della rete. Viene eseguita un'interazione continua di aggiornamenti delle tabelle fino al raggiungimento della convergenza.
    - *costruisci_vettore_distanze*: funzione che permette di convertire la tabella di instradamento in un formato vettoriale per semplificare l'invio delle informazioni.
    - *log_tabella*: utilizzata per registrare e stampare lo stato corrente delle tabelle di instradamento per ciascun nodo, facilitando il debug e la comprensione del processo di instradamento.
3) *Gestione del Grafo*:
    - *nodi*: ogni nodo è una chiave.
    - *archi e pesi*: ogni coppia nodo-vicino ha un peso che rappresenta il costo del collegamento.

EVOLUZIONE DELLE TABELLE:

1) *Inizializzazione*:
    - all'inizio della simulazione, ogni nodo conosce solo le distanze verso se stesso (costo 0) e verso i suoi vicini diretti. Tutte le altre distanze sono impostate su infinito (DISTANZA_MASSIMA).

2) *Iterazione*:
    - *passo 1*: ogni nodo condivide il suo Distance Vector con i vicini.
    - *passo 2*: i nodi ricevono i vettori dei vicini e aggiornano le proprie tabelle, ricalcolando i percorsi più economici. Questo avviene tramite il metodo *aggiorna_tabella.*
    - *passo 3*: questo processo continua finché non ci sono più aggiornamenti nelle tabelle di instradamento, segnalando il raggiungimento della convergenza.
3) *Convergenza*:
    - la rete raggiunge la convergenza quando ogni nodo conosce i percorsi più brevi verso tutti gli altri nodi. A questo punto, le tabelle non subiscono più cambiamenti.

ANALISI:

Output iniziale: le tabelle di instradamento iniziali mostrano esclusivamente le connessioni dirette:

DV(A) = {'A': 0, 'B': 1, 'C': 6, 'D': inf, 'E': inf}
DV(B) = {'B': 0, 'A': 1, 'C': 2, 'D': 1, 'E': inf}
DV(C) = {'C': 0, 'A': 6, 'B': 2, 'D': 3, 'E': 7}
DV(D) = {'D': 0, 'B': 1, 'C': 3, 'E': 2, 'A': inf}
DV(E) = {'E': 0, 'C': 7, 'D': 2, 'A': inf, 'B': inf}

iterazione 1:

i nodi aggiornano le loro tabelle di instradamento in base ai vettori delle distanze ricevuti dai vicini.

ad esempio, il nodo A riceve:

A riceve DV(B): {'B': 0, 'A': 1, 'C': 2, 'D': 1, 'E': inf}
A Tabella prima: [A: 0,A; B: 1,B; C: 6,C; D: inf; E: inf]
A Tabella dopo: [A: 0,A; B: 1,B; C: 3,B; D: 2,B; E: inf]

iterazione 2:

le informazioni vengono propagate ulteriormente. Le tabelle si avvicinano allo stato di convergenza.

iterazione 3:

la rete raggiunge la convergenza, e non si verificano ulteriori aggiornamenti

Output finale: le tabelle di instradamento finali mostrano i percorsi ottimali calcolati:

DV(A): {'A': 0, 'B': 1, 'C': 3, 'D': 2, 'E': 4}
DV(B): {'B': 0, 'A': 1, 'C': 2, 'D': 1, 'E': 3}
DV(C): {'C': 0, 'A': 3, 'B': 2, 'D': 3, 'E': 5}
DV(D): {'D': 0, 'B': 1, 'C': 3, 'E': 2, 'A': 2}
DV(E): {'E': 0, 'C': 5, 'D': 2, 'A': 4, 'B': 3}

CAPACITÀ DI DIAGNOSI E PROBLEM SOLVING:

Durante lo sviluppo del progetto sono emersi alcuni problemi e sfide che sono stati affrontati con una metodologia sistematica. Ogni errore veniva analizzato isolando le componenti coinvolte e testandole singolarmente. Ogni modifica al codice veniva testata su diverse configurazioni del grafo per garantire che il problema fosse risolto in modo generale. Attraverso log dettagliati e test incrementali, abbiamo identificato la causa principale dei problemi. Di seguito sono descritti i principali ostacoli incontrati, i tentativi di risoluzione e il ragionamento dietro le scelte effettuate.

- *Gestione delle tabelle di instradamento*:

Descrizione:
durante l'implementazione iniziale, si è verificato un problema nella rappresentazione delle tabelle di instradamento. I percorsi venivano sovrascritti anche quando il nuovo percorso non era più conveniente. Ciò ha causato incongruenze nei risultati finali.
Soluzione:
abbiamo aggiunto una condizione per verificare se il nuovo costo era effettivamente inferiore al costo già registrato nella tabella prima di aggiornare la voce corrispondente.
Abbiamo introdotto la funzione *aggiorna_tabella* per centralizzare la logica di aggiornamento delle tabelle e ridurre gli errori derivanti dalla duplicazione del codice.
Risultato:
la modifica ha garantito che i percorsi venissero aggiornati solo quando era disponibile una soluzione più efficiente, migliorando la coerenza dei dati.

- *Convergenza lenta del protocollo*:

Descrizione:
in alcune configurazioni del grafo, il processo di convergenza richiedeva molte iterazioni, aumentando il tempo di simulazione.
Soluzione:
abbiamo implementato un meccanismo per monitorare le modifiche alle tabelle di instradamento in ogni iterazione. Se nessuna tabella veniva aggiornata, la simulazione si interrompeva anticipatamente. La logica è stata migliorata utilizzando un flag (ci_sono_aggiornamenti) per verificare in modo globale se si erano verificati cambiamenti in una iterazione.
Risultato:
il numero di iterazioni necessarie è stato significativamente ridotto senza compromettere la correttezza del protocollo, rendendo la simulazione più efficiente.

- *Errore nella propagazione dei vettori delle distanze*:

Descrizione:
durante i primi test, alcune informazioni non venivano propagate correttamente tra i nodi. Questo problema era dovuto a errori nella funzione che costruiva il vettore delle distanze da inviare ai vicini.
Soluzione:
abbiamo isolato il problema verificando ogni parte della funzione invia_vettore_distanze. Abbiamo riscritto la funzione per garantire che venissero inclusi tutti i nodi nella rete, anche quelli con distanza infinita.
Risultato:
la funzione è stata corretta, e i dati sono stati propagati correttamente tra i nodi, garantendo che ogni nodo ricevesse informazioni complete per aggiornare la propria tabella di instradamento.

- *Debug e interpretazione dell'output*:

Descrizione:
durante la simulazione, era difficile comprendere i risultati a causa della mancanza di un formato leggibile per l'output delle tabelle di instradamento.
Soluzione:
abbiamo implementato un metodo __str__ nella classe Nodo per fornire una rappresentazione chiara e leggibile delle tabelle di instradamento.
Abbiamo introdotto la funzione log_tabella per registrare lo stato delle tabelle in ogni fase del processo, sia nel terminale che in un file di log.

Risultato:
l’output è diventato leggibile e facilmente interpretabile, semplificando il debug e la valutazione del processo di routing.