Progetto Python: Simulazione di Protocollo di Routing

Spiegazione sul codice

Classe Node

Questa classe, contenuta nel file `node.py`, rappresenta un nodo del network, in questo caso può essere vista come un router che può sfruttare DV.

AttributiNode

- **name**: identificativo univoco del nodo, può essere visto come l'indirizzo IP del router, nella simulazione non è nè necessario nè verificato che corrisponda un indirizzo IP, per far funzionare il programma è sufficiente che ogni nodo abbia un nome diverso.
- **routing_table**: rappresenta la routing table del router, che contiene le informazione sulla distanza da tutti i nodi nella rete raggiungibili dal router. In particolare è un dizionario python con chiave la destinazione (o un qualsiasi nodo raggiungibile) e valori una tupla contenente il peso e il prossimo nodo da raggiungere per arrivare alla destinazione.

MetodiNode Il costruttore prende in input l'identificativo univoco `name` del nodo.

`update_routing_table(self, neighbors)`

Il metodo principale dell'intera classe, questo metodo permette ai nodi di aggiornare le proprie informazioni sulla base delle informazioni in possesso dei nodi vicini. oltre a questo permette alla rete di capire se si è giunti alla convergenza grazie al valore di ritorno.

`neighbors` è un dizionario python che contiene informazioni su ognuno dei vicini del nodo che sta eseguendo il metodo in sè. in particolare la chiave è l'identificatore del vicino e il valore è una tupla con all'interno il peso dell'arco che collega il nodo al vicino e la tabella di routing del vicino stesso.

Il metodo si compone di 2 cicli for principali: il ciclo esterno scorre tutti i vicini, invece il ciclo interno valuta tutte i nodi nella routing table del nodo vicino, se il nodo viene raggiunto più velocemente passando dal vicino o non è nella routing table del nodo principale allora la routing table viene aggiornata e la variabile updated che segnala se sono avvenute delle modifiche viene aggiornata al valore di true.

```
updated = False
for neighbor, (cost_to_neighbor, neighbor_table) in neighbors.items():
    for dest, (neighbor_distance, neighbor_next_hop) in neighbor_table.items():
        new_distance = cost_to_neighbor + neighbor_distance
        if dest not in self.routing_table or new_distance < self.routing_table[dest][0]:
            self.routing_table[dest] = (new_distance, neighbor)
            updated = True
return updated
```

`print_routing_table(self)`

Metodo molto semplice che stampa la tabella di routing del nodo sul terminale.

Classe Network

Questa classe rappresenta la rete stessa, con tutti router collegati.

AttributiNetwork

- **nodes**: dizionario python dei nodi della rete, ognuno con chiave l'identificativo del nodo e valore il nodo stesso.

- **edges**: questa classe rappresenta i collegamenti tra i nodi della rete, si considera che ogni collegamento è bidirezionale e quindi il codice è adattato per seguire questa regola, tuttavia un singolo edge va da un nodo1 a un nodo2, quindi per ogni collegamento esistono 2 edges speculari. La classe è un dizionario con chiave il nodo di partenza e valore un secondo dizionario con chiave il vicino al primo nodo e il peso dell'arco. Quindi può essere visto come una lista di adiacenza del grafo. `{node_name1: {vicino1: weight1, ...}, ...}`.

MetodiNetwork

`add_node_after(self, node_name)`

Il metodo aggiunge un nodo al Network e prende in input l'identificativo del nodo, in più crea l'arco di collegamento del nodo con se stesso con peso 0. Questo metodo funziona anche se un nodo viene aggiunto dopo che la rete è già stata creata come se venisse aggiunto un altro router al network.

`add_edge(self, node1, node2, weight) and add_edge_and_update(self, node1, node2, weight)`

Questi due metodi permettono semplicemente di aggiungere un collegamento tra 2 nodi, la differenza è che il secondo metodo fa anche partire la simulazione immediatamente dopo aggiornando le routing table. Il metodo `add_edge` prende in input 2 id dei nodi e il peso dell'arco poi crea, o aggiorna, l'arco con le nuove caratteristiche e modifica le routing table in modo che i nodi coinvolti usino il nuovo arco, per aggiornare nuovamente le routing table sarà necessario rifare la simulazione dopo aver chiamato add_edge (anche per non usare il nuovo arco nel caso sia un peggioramento). `add_edge_and_update` si limita a rifare la simulazione dopo aver aggiunto un arco.

```
self.edges.setdefault(node1, {})[node2] = weight
self.edges.setdefault(node2, {})[node1] = weight

self.nodes[node1].routing_table[node2] = (weight, node2)
self.nodes[node2].routing_table[node1] = (weight, node1)
```

`remove_node_and_update(self, node_name)`

Preso in input l'id di un nodo lo rimuove dalla rete e aggiorna le tabelle di routing di conseguenza, simula un guasto a un router.

`remove_edge_and_update(self, node1, node2)`

Questo metodo rimuove un arco dalla rete e aggiorna le routing table di conseguenza, simulando un problema sulla linea.

`simulate(self)`

Algoritmo al centro del calcolo delle distanze dei router. Questo algoritmo si compone di due cicli principali, il primo procede fin tanto che non si raggiunge la convergenza (la convergenza in particolare è sempre raggiunta all'iterazione precedente all'ultima eseguita), ma questo criterio può cambiare in reti più grandi, rischiando però di non scrivere nella routing table dei nodi tutti i nodi raggiungibili. Qualora si voglia cambiare questo criterio va comunque considerato che la convergenza è matematicamente sempre raggiunta in al più n-2 iterazioni.

Il ciclo interno invece scorre ogni nodo nella rete e crea il dizionario `neighbors` da passare al metodo `.update_routing_table` dei nodi, quindi il ciclo interno passa ai nodi della rete le informazioni in possesso dei vicini dei nodi della rete e update routing table aggiorna queste informazioni, se questo scambio avviene n-2 volte è garantita la convergenza. N-2 deriva dal fatto che prima della prima iterazione i nodi hanno già alcune informazioni sui vicini e che la distanza massima tra 2 nodi è n-1, comunque nella simulazione ci si

aspetta che ci vogliano sempre molto meno di n-2 iterazioni. Senza variare il funzionamento attuale della gui ad esempio sarebbe sempre sufficiente al più una sola iterazione per aggiornare le routing table correttamente.

Considerazione: attualmente il metodo simulate aggiorna prima le routing table del primo nodo e poi del secondo e così via, quindi gli ultimi nodi a essere aggiornati avranno delle routing table più nuove. Per evitare questo sarebbe necessario salvarsi tutti i valori di `neighbors` in un buffer del tipo `{idNodo: neighbors}` e poi alla fine del ciclo for interno eseguire un altro ciclo for per passare a tutti i nodi i vari neighbors. Questa idea non è stata implementata, ma permetterebbe di dividere il ciclo interno in 2 cicli embarrassingly parallel e nel caso di uso di multithread su un numero molto elevato di nodi potrebbe dare un aumento di performance.

Il metodo simulate si occupa anche di stampare a terminale le routing table nelle varie iterazioni.

```
iteration = 0
changes = True
while changes:
    iteration += 1
    print(f"Iterazione {iteration}:")
    changes = False
    for node_name, node in self.nodes.items():
        neighbors = {
            neighbor: (self.edges[node_name][neighbor], self.nodes[neighbor].routing_table)
            for neighbor in self.edges.get(node_name, {})
        }
        if node.update_routing_table(neighbors):
            changes = True

    if changes:
        for node in self.nodes.values():
            node.print_routing_table()
    else:
        print("Nessun cambiamento, convergenza ottenuta")
    print("-" * 50)  # Separatore tra le iterazioni
```

Classe NetworkGUI

La classe che crea la gui per interagire con la rete. Al momento la gui aggiorna la rete al click di ogni singolo pulsante tuttavia è molto semplice cambiare questa caratteristica rimuovendo il commento

```
#self.simulate_button = tk.Button(self.command_frame, text="Simula", command=self.simulate_network, bg=
#self.simulate_button.grid(row=0, column=2, padx=5, pady=5)
```

a riga 41 e sostituendo il metodo `add_edge_and_update(node1, node2, weight)` con il metodo `add_edge(node1, node2, weight)` a riga 96.

Nella fase iniziale nella gui non è presente alcun nodo, sta all'utente posizionare degli archi e dei nodi di collegamento.

La GUI si compone di una text area `routing_text`, che si aggiorna all'uso dei pulsanti, in cui compaiono le informazioni sulle routing table dei nodi.

E 4 o 5 pulsanti:

- `add_edge_button` che (attualmente) chiama `add_edge_and_update` e permette di aggiungere degli archi tra i nodi esistenti.
- `add_node_button` che permette di aggiungere dei nodi senza alcun collegamento se non a loro stessi.
- `remove_edge_button` permette di rimuovere un arco dalla rete, simulando ad esempio un problema di linea.
- `remove_node_button` permette di rimuovere un nodo simulando un problema ad esempio a un router.

- `simulate_button` pulsante per simulare CV. **Attualmente NON presente nella gui** può essere facilmente introdotto seguendo le istruzioni descritte sopra, questo tasto pur essendo sempre e comunque funzionante non ha alcuno scopo se `add_edge_button` chiama il metodo `add_edge_and_update` poiché allo stato attuale cliccare su qualunque pulsante a eccezione di `add_node_button` fa già partire la simulazione del CV.