Spanning Tree

Dies ist ein Laborprojekt bei Friedmann Stockmayer für das Kommunikations- und Netztechnik Labor.

Requirements

Python: ^3.7

Ausführung

Um dieses Programm auszuführen, muss in die Konsole `py main.py` eingegeben werden (Vom Projekt-Ordner aus).

Aufbau

Input

Um ein Spanning Tree zu erstellen muss der Aufbau des Netzwerks mit den gegeben Kantenkosten definiert sein. Dafür wird eine JSON-Datei genutzt.

JSON-Datei

- **nodes**: Eine Liste von Knoten im Graphen. Jeder Knoten hat zwei Eigenschaften:
  - *name*: Der Name des Knotens.
  - *id*: Die ID des Knoten. Die kleinste ID wird zum Root!
- **edges**: Ein Objekt, das die Kanten im Graphen definiert. Jede Kante wird durch eine Zeichenkette beschrieben, die die beiden verbundenen Knoten durch einen Unterstrich (_) trennt (z.B. "A_B"). Der Wert der Kante ist das Gewicht der Kante

Um einen eigenen Spanning Tree zu definieren, muss die `input.json` Datei verändert werden.

Programm

Das Programm besteht aus 3 Python-Datein.

- **main.py**: Hier wird das Programm ausgeführt
- **edge.py**: Hier wird eine Kanten Klasse definiert, welche eine Kante zwischen zwei Knoten repräsentiert.
- **node.py**: Hier wird eine Knoten Klasse definiert, welche einen Knoten repräsentiert.

Output

Der Spanning Tree wird auf der Konsole ausgegeben auf folgender Weise:

```
D: -> E
F: -> E
A: -> B
```

```
C: -> D
B: ROOT
```

---

main.py

`main.py` – Hauptprogramm zur Berechnung des Spanning Tree

Das Skript `main.py` implementiert den **Spanning Tree Algorithmus** unter Verwendung der `Node`- und `Edge`-Klassen. Es liest die Knotendaten und Kanten aus einer JSON-Datei ein, erstellt den Graphen und berechnet den Spanning Tree mithilfe des BPDU-Protokolls.

Funktionen

`read_input(file_path)`

Liest die Knoten- und Kantendaten aus einer JSON-Datei und gibt sie als Dictionaries zurück.

Parameter:

- `file_path`: Pfad zur JSON-Datei mit den Graphendaten.

Rückgabe:

- `node_data`: Liste mit Knotendaten (`[{"name": "A", "id": 1}, ...]`).
- `edge_data`: Dictionary mit Kanteninformationen (`{"A_B": 3, "B_C": 2, ...}`).

`executeAlgorithm(nodeList)`

Führt den Spanning Tree Algorithmus aus.

Ablauf:

1. Die Knotenliste wird zufällig durchmischt.
2. Das BPDU-Protokoll läuft so lange, bis keine Änderungen mehr auftreten oder die maximale Iterationszahl erreicht ist.
3. Die aktualisierte Knotenliste wird zurückgegeben.

Parameter:

- `nodeList`: Liste aller `Node`-Objekte.

Rückgabe:

- `nodeList`: Aktualisierte Liste der Knoten nach der Berechnung des Spanning Trees.

`result(nodeList)`

Gibt das Ergebnis des Spanning Tree Algorithmus aus.

Ablauf:

- Falls ein Knoten `nextHop == self` ist, wird er als **ROOT** ausgegeben.
- Andernfalls wird die Verbindung zum `nextHop` angezeigt.

Parameter:

- `nodeList`: Liste aller `Node`-Objekte.

Hauptablauf

1. **Einlesen der JSON-Daten** (`read_input`).
2. **Erstellung der Knoten** (`create_nodes`).
3. **Erstellung der Kanten** (`create_edges`).
4. **Zuordnung der Kanten zu Knoten** (`add_edges_to_nodes`).
5. **Berechnung des Spanning Tree** (`executeAlgorithm`).
6. **Ausgabe des Ergebnisses** (`result`).

Edge.py

Klasse `Edge` – Repräsentation einer Kante im Graphen

Die Klasse `Edge` dient zur Modellierung einer gewichteten Kante zwischen zwei Knoten in einem Graphen. Jede Kante verbindet genau zwei Knoten und besitzt eine bestimmte Gewichtung.

Attribute:

- `node1`: Der erste Knoten der Kante.
- `node2`: Der zweite Knoten der Kante.
- `weight`: Das Gewicht der Kante.

Methoden:

`__init__(self, node1, node2, weight)`

Konstruktor zur Initialisierung einer Kante mit den beiden Knoten `node1`, `node2` sowie dem Gewicht `weight`.

`getNeighbour(self, node)`

- Gibt den benachbarten Knoten der Kante zurück.
- Nimmt als Parameter einen Knoten `node` und liefert den jeweils anderen Knoten der Kante.

`nodeInEdge(self, nodeName)`

- Prüft, ob ein Knoten mit dem Namen `nodeName` Teil der Kante ist.
- Gibt `True` zurück, falls `node1` oder `node2` den gesuchten Namen trägt, ansonsten `False`.

`__str__(self)`

- Gibt eine lesbare Zeichenkettendarstellung der Kante zurück.

Funktion `create_edges(edge_data, nodes)`

Diese Funktion erstellt eine Liste von `Edge`-Objekten aus einer gegebenen Kanten-Datenstruktur.

Parameter:

- `edge_data`: Ein Dictionary, in dem die Schlüssel die Kanten als Zeichenketten (z. B. `"A_B"`) und die Werte die jeweiligen Kantengewichte enthalten.
- `nodes`: Ein Dictionary, das Knotennamen als Schlüssel und die entsprechenden Knotenobjekte als Werte enthält.

Ablauf:

1. Eine leere Liste `edges` wird initialisiert.
2. Jede Kante aus `edge_data` wird verarbeitet:
    - Der Schlüssel (z. B. `"A_B"`) wird an `_` gesplittet, um die beiden Knotennamen zu extrahieren.
    - Ein neues `Edge`-Objekt wird mit den zugehörigen `nodes` und dem Gewicht erzeugt.
    - Das `Edge`-Objekt wird der Liste hinzugefügt.
3. Die fertige Liste aller `Edge`-Objekte wird zurückgegeben.

Node.py

Klasse `Node` – Repräsentation eines Knotens im Graphen

Die Klasse `Node` stellt einen Knoten in einem Graphen dar. Jeder Knoten kann mit anderen Knoten über Kanten (`Edge`-Objekte) verbunden sein und wird über eine eindeutige ID sowie einen Namen identifiziert.

Attribute:

- `name`: Der Name des Knotens (z. B. „A“, „B“).
- `id`: Eine eindeutige ID zur Identifikation des Knotens.
- `pEdges`: Eine Liste von Kanten (`Edge`-Objekte), die mit diesem Knoten verbunden sind.
- `nextHop`: Der nächste Knoten auf dem Weg zur Root im Spanning Tree (initialisiert mit sich selbst).
- `root`: Ein Dictionary mit:
    - `"node"`: Der aktuelle Root-Knoten.
    - `"weight"`: Das Gewicht des Pfades zum Root-Knoten (anfangs `0`).

Methoden:

`__init__(self, name, id, pEdges=None)`

- Erstellt einen Knoten mit einem Namen, einer ID und einer optionalen Kantenliste.
- Initialisiert sich selbst als Root im Spanning-Tree-Protokoll.

`addEdges(self, edges)`

- Fügt eine Liste von Kanten (`Edge`-Objekte) zum Knoten hinzu.

`getEdgeToNeighbour(self, nodeName)`

- Sucht die Kante zu einem direkt verbundenen Nachbarn mit dem Namen `nodeName`.
- Gibt die Kante zurück oder `None`, falls keine existiert.

`sendBPDU(self)`

- Sendet ein **BPDU (Bridge Protocol Data Unit)** an alle benachbarten Knoten.
- Falls sich der Root-Knoten durch den Empfang einer BPDU ändert, gibt die Methode `True` zurück, sonst `False`.

`receiveBPDU(self, node, toRoot)`

- Verarbeitet eine empfangene BPDU von einem Nachbarknoten.
- Entscheidet, ob der aktuelle Knoten eine bessere Root-Route gefunden hat:
  - Falls der empfangene Root-Knoten eine niedrigere ID hat, wird er übernommen.
  - Falls die Root-ID gleich ist, aber der neue Weg ein geringeres Gewicht hat, wird er übernommen.
- Gibt `True` zurück, wenn sich die Root-Information geändert hat.

`__str__(self)`

- Gibt eine String-Repräsentation des Knotens zurück (`"A: 1"`).

Statische Funktionen zur Initialisierung von Knoten und Kanten

`create_nodes(node_data)`

- Erstellt ein Dictionary mit `Node`-Objekten basierend auf einer Liste von Knotendaten.
- Erwartet `node_data` als eine Liste von Dictionaries mit `"name"` und `"id"`.

`add_edges_to_nodes(nodes, edges)`

- Geht durch alle Knoten und fügt ihnen die entsprechenden Kanten hinzu.
- Verwendet die Funktion `edgeOfNode`, um verbundene Kanten zu finden.

`edgeOfNode(nodeName, edges)`

- Sucht alle Kanten, die mit einem bestimmten Knoten (`nodeName`) verbunden sind.
- Gibt eine Liste der entsprechenden `Edge`-Objekte zurück.