*Graphen Erstellung*

```
def create_network_from_file(filename):
  with open(filename, 'r') as file:
    stations = set()
    routes = []

    for line in file:
      parts = line.strip().split(': ')
      if len(parts) > 1:
        line_name = parts[0]
        line = parts[1]
        line_parts = re.findall(r'\".*?\"(?: \d)?', line)

        for i in range(len(line_parts) - 1):
          station1 = line_parts[i].split('"')[1]
          station2 = line_parts[i + 1].split('"')[1]
          weight = int(line_parts[i].split(' ')[-1])
          line_name = line_name

          stations.add(station1)
          stations.add(station2)
          routes.append((station1, station2, weight, line_name))

    graph = Graph(sorted(list(stations)))
    for route in routes:
      graph.add_route(*route)

    return graph
```

Die Funktion **create_network_from_file(filename)** erzeugt ein Graph-Objekt aus den Daten in der angegebenen Datei. Die Datei wird eingelesen und 2 Sets für die Stationen und Routen erstellt. Jede Zeile wird gelesen und korrekt gespalten. Alle Stationen werden in das Stationen-Set hinzugefügt und alle Routen (2 verbundene Stationen und deren Gewicht) in das Routen-Set hinzugefügt. Anschließend wir dein Graph-Objekt erstellt mit allen Stationen aus dem Stationen-Set. Für alle Routen im Routen-Set wird die Methode add_route(*route) aufgerufen. Der Stern-Operator teilt die route in ihre 4 Variablen (Station1, Station2, Weight, Line) auf und gibt sie an die Methode weiter.

```python
class Graph:

  def __init__(self, stations):
    self.stations = stations
    self.index_to_station = {index: station for index, station in enumerate(stations)}
    self.station_to_index = {station: index for index, station in enumerate(stations)}
    self.graph = [[None] * len(stations) for _ in range(len(stations))]
    for i in range(len(stations)):
      self.graph[i][i] = 0
      self.lines = [[None] * len(stations) for _ in range(len(stations))]

  def add_route(self, station1, station2, weight=1, line=None):
    i = self.station_to_index[station1]
    j = self.station_to_index[station2]
    self.graph[i][j] = weight
    self.graph[j][i] = weight
    self.lines[i][j] = line
    self.lines[j][i] = line
```

Der Konstruktor **__init__(self, stations)** initialisiert den Graphen. Er nimmt eine Liste von Stationen entgegen und erzeugt daraus zwei Dictionaries, die die Zuordnung zwischen den Stationsnamen und ihren Indexen in der Liste speichern. Danach wird eine quadratische Matrix der Größe n*n erstellt, in der n die Anzahl der Stationen ist. Jeder Wert in der Matrix wird auf None gesetzt, um anzuzeigen, dass es zunächst keine Verbindung zwischen den Stationen gibt. Schließlich wird die Hauptdiagonale auf Null gesetzt, um anzuzeigen, dass die Entfernung einer Station zu sich selbst Null ist.

Die Methode **add_route(self, station1, station2, weight=1, line=None)** fügt eine Route zum Graphen hinzu. Sie nimmt zwei Stationsnamen, ein Gewicht und optional einen Linienamen entgegen. Sie findet die Indizes der beiden Stationen in der Liste und setzt den entsprechenden Wert in der Matrix auf das gegebene Gewicht. Sie speichert auch den Liniennamen in einer separaten Matrix, um die Linieninformationen für jede Kante im Graphen zu speichern.

*Laufzeit für das Erstellen eines Graphen*

Die Funktion **create_network_from_file(filename)** liest eine Datei, teilt jede Zeile auf, fügt Stationen zu einem Set hinzu und Routen zu einer Liste. Jede dieser Operationen hat eine Zeitkomplexität von **O(m), wobei m die Anzahl der Zeilen in der Datei ist**, da jede Zeile einzeln gelesen und verarbeitet wird.

Das Erzeugen des Graphen dauert **$O(n^2)$, wobei n die Anzahl der Stationen ist**, da es eine quadratische Matrix erstellt.

Das Hinzufügen der Routen zur Graph-Matrix hat eine Zeitkomplexität von **O(k), wobei k die Anzahl der Routen ist**, die in der Datei aufgeführt sind.

Daher ist die gesamte Zeitkomplexität der create_network_from_file(filename)-Funktion **$O(m + n^2 + k)$**. Im Allgemeinen kann die Laufzeit auf **$O(n^2)$** vereinfacht werden können.

*„Außerdem ist der Platzbedarf im Gegensatz zu den drei vorhergehenden Datenstrukturen $\Theta(n^2)$, unabhängig von der Anzahl der Kanten. Damit ist die Adjazenzmatrix vor allem für dichte Graphen interessant, also für Graphen, die, gemessen an der Anzahl ihrer Knoten, viele Kanten besitzen."*

Prof. Justus Piater, Ph.D. (30.04.2023): Algorithmen und Datenstrukturen – Graphen (Seite 12), https://iis.uibk.ac.at/public/piater/courses/IIS/703010/notes/graphs/graphs.pdf

```
def find_shortest_route(self, start, end):
  shortest_distances = {station: float('inf') for station in self.stations}
  previous_stations = {station: None for station in self.stations}
  previous_line = {station: None for station in self.stations}

  shortest_distances[start] = 0
  unvisited = set(self.stations)

  while len(unvisited) > 0:
    current_node = min(unvisited, key=lambda node: shortest_distances[node])
    unvisited.remove(current_node)

    if current_node == end:
      break
```

```
    current_index = self.station_to_index[current_node]

    for neighbor_index, weight in enumerate(self.graph[current_index]):
      if weight != float('inf'):
        neighbor = self.index_to_station[neighbor_index]
        new_distance = shortest_distances[current_node] + weight

        if new_distance < shortest_distances[neighbor]:
          shortest_distances[neighbor] = new_distance
          previous_stations[neighbor] = current_node
          previous_line[neighbor] = self.lines[current_index][neighbor_index]

  shortest_route = []
  current_node = end
  while current_node is not None:
    shortest_route.append((current_node, previous_line[current_node],
                           shortest_distances[current_node]))
    current_node = previous_stations[current_node]

  shortest_route.reverse()
  return shortest_route, shortest_distances[end]
```

Die Methode **find_shortest_route(self, start, end)** ist Teil der Klasse Graph und sucht nach der kürzesten Route zwischen zwei eingegebenen Stationen.

Zuerst werden drei Dictionaries initialisiert. Das Dictionary “shortest_distances” wird mit einer Distanz von unendlich und die Dictionaries “previous_stations “und “previous_line” werden mit None initialisiert.

Anschließend wird die kürzeste Distanz zur Startstation gleich 0 gesetzt und ein Set (= ungeordnete Sammlung eindeutiger Elemente, jedes Element kommt also nur einmal vor) von allen Stationen im Graphen in die Variable “unvisited” gespeichert, um besser prüfen zu können, ob eine Station schon einmal besucht wurde oder nicht.

Nun beginnt eine Schleife, die so lange durchgelaufen wird, bis es keine unbesuchten Stationen mehr gibt.

Zu Beginn dieser Schleife wird die nächste unbesuchte Station mit der geringsten Distanz in eine Variable gespeichert und aus dem Set der unbesuchten Stationen entfernt. Falls diese Station schon die Endstation ist, wird die Schleife unterbrochen.

Dann wird der Index dieser Station ebenfalls in eine Variable gespeichert und in einer Schleife alle Nachbarn des aktuellen Knotens durchgegangen.

Wenn eine Verbindung zu diesem Nachbarn besteht, werden der Nachbar selbst und die Distanz zu ihm in jeweils einer Variable zwischengespeichert. Im nächsten Schritt wird überprüft, ob die neu gefundene Distanz zu einem Nachbarn kürzer ist als die bisher gefundene kürzeste Distanz. Ist das der Fall, wird die kürzeste Distanz, die vorhergehende Station und die Linie aktualisiert.

Sobald die Endstation erreicht wird, wird eine Liste für die gefundene kürzeste Route erstellt und die aktuelle Station gleich der Endstation gesetzt. Dann werden alle Stationen samt vorhergehender Station und kürzester Distanz zur Liste hinzugefügt, bis die Startstation erreicht wird. Zuletzt wird die Liste reversed und dann zusammen mit der kürzesten Distanz zwischen Start- und Endstation zurückgegeben.

Die gesamte Funktion hat im Worst Case eine Laufzeit von **$O(n^2)$**, wobei n die Anzahl der Stationen im Graphen ist, da im schlimmsten Fall jede Station mit all ihren Nachbarn besucht werden muss.

```
class ShortestRoutes:
  def __init__(self):
    self.routes = {}

  def get_route(self, start, end):
    if start in self.routes and end in self.routes[start]:
      return self.routes[start][end]
    else:
      return None

  def add_route(self, start, end, route):
    if start not in self.routes:
      self.routes[start] = {}
      self.routes[start][end] = route
```

Die Klasse Shortest Routes erstellt mit ihrem Konstruktor ein Dictionary, in das alle bereits gefundenen Routen zwischen zwei Stationen gespeichert werden.

Die Funktion **get_route(self, start, end)** überprüft, ob eine Route von der eingegebenen Startstation bereits in das Dictionary gespeichert wurde. Wenn das der Fall ist, wird diese Route zurückgegeben, andernfalls gibt die Funktion None zurück. Diese Funktion hat eine Laufzeit von **O(1)**, da auf einen bestimmten Schlüssel im Dictionary zugegriffen wird.

Die Methode **add_route(self, start, end, route)** fügt eine neue Route zu dem Dictionary hinzu. Sie überprüft, ob die Startstation bereits in den gespeicherten Routen vorhanden ist und erstellt ein neues Dictionary für diesen Startort, wenn er noch nicht existiert. Anschließend wird die Route zum Dictionary hinzugefügt, wobei die Zielstation als Schlüssel und die Route als Wert gespeichert wird. Diese Funktion hat eine Laufzeit von **O(1)**, da das Hinzufügen eines Eintrages in ein Dictionary eine Zeitkomplexität von O(1) hat.

*Zeitmessung*

```
# Erstellung des Graphens
start = time.time()
graph = create_network_from_file("wv.txt")
end = time.time()
print(f"Die Ausführung der Funktion create_network_from_file() hat {end - start} Sekunden gedauert.")

# Wegsuche
start = time.time()
route, total_time = graph.find_shortest_route("Floridsdorf",
"Praterstern")
end = time.time()
print(f"Die Ausführung der Funktion find_shortest_route hat {end - start} Sekunden gedauert.")
```

**Vollständige File:**

Für die Erstellung der Matrix des Wiener Verkehrsnetzes wird folgendes ausgegeben:

```
Die Ausführung der Funktion create_network_from_file() hat 0.00700068473815918 Sekunden gedauert.
```

Für die Wegsuche im Graphen wird folgendes ausgegeben:

```
Die Ausführung der Funktion find_shortest_route hat 0.024719715118408203 Sekunden gedauert.
```

**Unvollständige File (50% der Stationen fehlen):**

Für die Erstellung der Matrix des Wiener Verkehrsnetzes wird folgendes ausgegeben:

```
Die Ausführung der Funktion create_network_from_file() hat 0.002980947494506836 Sekunden gedauert.
```

Für die Wegsuche im Graphen wird folgendes ausgegeben:

```
Die Ausführung der Funktion find_shortest_route hat 0.006997823715209961 Sekunden gedauert.
```

**Unvollständige File (75% der Stationen fehlen):**

Für die Erstellung der Matrix des Wiener Verkehrsnetzes wird folgendes ausgegeben:

```
Die Ausführung der Funktion create_network_from_file() hat 0.0009851455688476562 Sekunden gedauert.
```

Für die Wegsuche im Graphen wird folgendes ausgegeben:

```
Die Ausführung der Funktion find_shortest_route hat 0.0020987987518310547 Sekunden gedauert.
```

| Anzahl der Stationen | Matrix Erstellung in Sekunden | Wegsuche in Sekunden |
|---|---|---|
| 100% | 0.00700068473815918 | 0.02471971511848203 |
| 50% | 0,002980947494506836 | 0,006997823715209961 |
| 25% | 0,0009851455688476562 | 0,0020987987518310547 |

Es ist klar ersichtlich, dass die Laufzeit der Funktionen mit geringeren Stationen abnimmt.