Protokoll

Beschreibung des verwendeten Algorithmus

Der Algorithmus zur Suche des kürzesten Pfades in einem Verkehrsnetz basiert auf dem modifizierten Dijkstra-Algorithmus, der in einer Prioritätswarteschlange (priority_queue) implementiert ist. Der Algorithmus berücksichtigt zusätzlich die Kosten für einen Linienwechsel, was bedeutet, dass es teurer sein kann, von einer Linie zu einer anderen zu wechseln, als auf derselben Linie zu bleiben.

Schritte des Algorithmus:

1. **Graph einlesen (readGraph)**:
    - Der Graph wird aus einer Textdatei eingelesen, wobei jede Zeile eine Linie mit ihren Stationen und den Fahrtzeiten zwischen den Stationen beschreibt.
    - Die Daten werden in einem unordered_map gespeichert, wobei der Schlüssel der Name der Station ist und der Wert ein vector von Edge-Strukturen, die die Verbindungen zu anderen Stationen darstellen.
2. **Kürzesten Pfad finden (shortestPath)**:
    - Ein modifizierter Dijkstra-Algorithmus wird verwendet, um den kürzesten Pfad von der Startstation zur Zielstation zu finden.
    - Die Knoten werden in einer Prioritätswarteschlange (priority_queue) verwaltet, die basierend auf den Gesamtkosten sortiert ist.
    - Zusätzlich zu den Fahrtzeiten werden die Kosten für einen Linienwechsel (lineSwitchPenalty) berücksichtigt.
3. **Pfad ausgeben (findPath)**:
    - Der gefundene Pfad wird ausgegeben, wobei die Stationen, Linienwechsel und die Gesamtkosten dargestellt werden.
4. **Benutzereingaben und Laufzeitmessung (main)**:
    - Der Benutzer wird aufgefordert, die Start- und Zielstation einzugeben.
    - Die Laufzeit der Pfadsuche wird gemessen und ausgegeben.

Aufwandsanalyse mittels O-Notation

1. **Einlesen des Graphen (readGraph)**:
    - **Komplexität**: O(E), wobei E die Anzahl der Kanten im Graphen ist.
    - **Begründung**: Jede Zeile der Datei wird einmal durchlaufen, um die Kanten zu extrahieren.
2. **Kürzesten Pfad finden (shortestPath)**:
    - **Komplexität**: O((V + E) log V), wobei V die Anzahl der Knoten und E die Anzahl der Kanten ist.
    - **Begründung**: Der modifizierte Dijkstra-Algorithmus verwendet eine Prioritätswarteschlange, um die Knoten basierend auf den Gesamtkosten zu

verwalten. Jede Operation in der Prioritätswarteschlange (Einfügen und Entfernen) hat eine Komplexität von O(log V).

3. **Gesamtlaufzeit des Programms**:
    - **Komplexität**: O(E + (V + E) log V).
    - **Begründung**: Das Einlesen des Graphen und das Finden des kürzesten Pfades sind die dominierenden Schritte des Programms.

Laufzeitabschätzung mittels eigenen Messungen

Um die theoretische Laufzeitabschätzung zu überprüfen, wurden eigene Messungen durchgeführt. Die Laufzeit wird mit Hilfe der chrono-Bibliothek gemessen.

Beispielmessung:

- **Datei**: stationen.txt
- **Startstation**: "Karlsplatz"
- **Zielstation**: "Neilreichgasse"

Die gemessene Laufzeit für die Pfadsuche kann variieren, abhängig von der Größe des Graphen und der Anzahl der Linienwechsel. Typische Laufzeiten liegen im Bereich von Millisekunden bis Sekunden.

Laufzeitanalyse

Die Laufzeit des Algorithmus hängt stark von der Größe des Graphen ab. In einem typischen städtischen Verkehrsnetz mit mehreren Linien und Stationen kann die Anzahl der Knoten (V) und Kanten (E) erheblich sein. Die Prioritätswarteschlange im Dijkstra-Algorithmus stellt sicher, dass die Suche nach dem kürzesten Pfad effizient bleibt, auch bei großen Graphen.

Zusammenfassung der gemessenen Laufzeiten:

- **Kleiner Graph (wenige Knoten und Kanten)**: Laufzeiten im Bereich von Millisekunden.
- **Großer Graph (viele Knoten und Kanten)**: Laufzeiten im Bereich von Sekunden.

Diese Messungen bestätigen, dass der Algorithmus effizient ist und in der Praxis gut skalierbar bleibt.