Programmierbeispiel 3

Für die Lehrveranstaltung Algorithmen und Datenstrukturen Sommer-Semester 2025

Jana Allgäuer und Lena Gampenrieder

Inhalt

[Datenstruktur: 2](#_Toc199079250)

[Aufbau Graph – Ablauf und O-Notation 2](#_Toc199079251)

[Pfadsuche – Ablauf und O-Notation 3](#_Toc199079252)

[Pfad ausgeben – Ablauf und O-Notation 6](#_Toc199079253)

# Datenstruktur:

Wir verwenden eine unordered\_map, welche einer Hashtabelle entspricht (<https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/>), um unseren Graphen zu speichern . Der Key über welchen zugegriffen wird entspricht dem Namen einer Station. Die Daten welche in der unordered\_map gespeichert werden liegen in Form eines Vectors welcher Structs vom Typ „KantenInfo“ enthält.

Ein Bild, das Text, Schrift, Reihe, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein. Jedes KantenInfo-Objekt enthält einen string *to*, welcher dem Namen der anderen Station entspricht, eine int *weight*, welche dem Gewicht der Kante, also der Zeit zwischen den Beiden Stationen, entspricht und einen string *line*, welcher den Namen der Linie, mit welcher die beiden Stationen verbunden sind, entspricht. Ein Bild, das Text, Screenshot, Kreis, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

# Aufbau Graph – Ablauf und O-Notation

Als erstes wird die Datei, aus welcher ausgelesen werden soll, geöffnet O(1).

Dann wird auf der Datei jeweils eine Zeile ausgelesen O(L) (wobei L für die Zeilen in der Datei und somit die Anzahl der Linien steht) und folgendes gemacht:

Die Line wird in einen iStringStream gespeichert (für bessere Lesbarkeit, da sie nur aus- und nicht eingelesen werden soll). O(1)

Der Linienname wird in die String-Variable lineName eingelesen O(1).

Der Linienname wird von führende und folgende Whitespaces befreit O(1).

Es wird solange aus dem iStringStream in die string Variable „dummy“ gelesen bis das erste “ entdeckt wird.

Dann wird der Name der Ersten Station, auf dieser Linie, eingelesen. O(1)

Nun sind wir bei der inneren While-Schleife, diese wird so lange ausgeführt, wie Kanten-Gewichte gefunden werden. O(k)

In ihr wird der Name der nächsten Station eingelesen O(1).

Und im Anschluss werden die beiden Stationen als jeweilige Nachbar\*Innen in den Graph eingespeichert:



Der Zugrif auf einen Index von graph erfolgt (da es sich um eine unordered\_map und somit um eine Hashtabelle handelt) mit O(1)

(Quelle: <https://www.geeksforgeeks.org/map-vs-unordered_map-c/>)

.push\_back() erfolgt mit O(1).

(Quellen:<https://runestone.academy/ns/books/published/cppds/AlgorithmAnalysis/VectorAnalysis.html> | <https://www.quora.com/Do-C-vectors-allocate-memory-for-empty-elements> )

Schließlich wird noch currentStation = prevStation gesetzt O(1).

Somit haben wir eine äußere Schleife mit O(L) und eine innere Schleife mit O(k).

Grundsätzlich gibt es in jeder Zeile zwei Stationen mehr als Kanten (keine Kante führt ins nichts / jede Kante verbindet zwei Stationen)

Die Erste Station ist immer Teil der äußeren Schleife.

Die Anzahl der Kanten (k) ist aber Abhängig von der Linie in der wir uns befinden. Daher kommen wir nicht auf eine O-Notation von O(L\*k), sondern wir erhalten eine O-Notation von O(L+k) .

# Pfadsuche – Ablauf und O-Notation

Ich gehen zur Berechnung der O-Notation mit folgender Legende vor:

S= Anzahl an Stationen | K = Anzahl an Kanten

Für die Suche wird der implementierte dijkstra Algorithmus ausgeführt.

Dieser beginnt damit, dass eine unordered\_map *node\_data* ,in welcher über den Stationsnamen immer ein KnotenInfo-Struct gefunden werden kann, initialisiert wird. O(1)

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

*node\_data* entspricht der Liste an noch nicht besuchten Knoten.

Des Weiteren wird eine priority\_queue *heap* initialisiert O(1). priority\_queue enstprechen einem binär-Baum (Quelle <https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/> ).

Unsere priority\_queque speichert Structs vom Typ HeapElement, als Vector und sortiert immer das kleinste nach vorne – es handelt sich also um einen min-Heap.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

*Heap* entspricht den momentan erreichbaren Nachbar\*Innen mit den Kosten zu ihnen.

Für jeden Eintrag im Graphen wird der korrelierenden Eintrag in der unordered\_map node\_data auf unbesucht, mit einer distanz von der größtmöglichen Zahl und keinem Vorgängerkonten und keiner Linie mit der der Eintrag erreicht werden kann, gesetzt. - O(S), wobei S die Anzahl der eingelesenen Stationen /“keys“ im Graphen ist.

Der Start-Knoten erhält das Gewicht 0, da es uns keine Zeit kostet vom Start-Knoten zum Start-Knoten zu gelangen – O(1).

In die priority\_queque *heap* wird ein HeapElement mit station: startKnoten und total\_weight: 0 gepushed -O(log n), wobei n die Anzahl an vorhandenen Knoten in der priority\_queque ist, also in diesem Fall O(1), da startKnoten das erste Objekt ist, das wir in die priority\_queque pushen.

(Quelle: <https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/> )

Nun wird noch ein string *currentStation* mit dem Wert des startKnoten belegt O(1)

Eine int Variable *currentWeight* angelegt O(1)

Sowie der bool *zielKnotenErreicht* auf false gesetzt O(1).

Nun sind die Vorbereitungen für das Durchführen des Dijkstra-Algorithmus abgeschlossen.

Wir starten jetzt mit einer do-while schleife, deren Abbruchbedingungen sind, dass 1. Der Zielknoten noch nicht erreicht wurde und 2. Unsere priority\_queque nicht leer ist.

Da Jede Runde neue Kanten zur priority\_queque hinzugefügt werden können (insgesammt sogar maximal alle Kanten), könnten wir eine maximale Laufzeit von O(K) erhalten.

Wir nehmen uns innerhalb der Do-While schleife den Obersten Eintrag aus *heap,* Im ersten Durchlauf ist das offensichtlich startKnoten, da es der einzige Eintrag ist, in Zukunft wird es immer jener mit den geringsten Kosten sein. O(1)

Unser Gesamtgewicht wird gleich dem Gewicht dieses Eintrags gesetzt – O(1)

Dann wird er Eintrag aus *heap* entfernt – O(log n), wobei n der momentanen Größe des Heap entspricht.

Nun wird das Gewicht um diesen Eintrag zu erreichen mit dem Gewicht, das wir für diesen Eintrag in *node\_data* – also der Liste noch nicht in unseren Weg aufgenommener Knoten- gespeichert haben, verglichen. O(1)

Falls das so ist, wird der Schleifendurchlauf übersprungen.

Andernfalls, wird der Knoten *node\_data* hinzugefügt. O(1).

Nun wird überprüft ob wir bereits den Zielknoten erreicht haben, falls ja wird abgebrochen O(1).

Ansonsten wird die momentan betrachtete Station in *node\_data* auf besucht gestellt. O(1)

Nun kommen wir in eine For-Schleife, innerhalb unserer Do-While Schleife, in welcher wir nun alle, noch nicht besuchten, Nachbar\*Innen, unseres soeben besuchten Knotens in *heap* aufnehmen. Allerdings nur jene Nachbar\*Innen bei denen das Gewicht - um sie über den soeben hinzugefügten Knoten zu erreichen - kürzer ist als ein Weg den wir bereits gefunden haben (welcher ja zu Beginn immer den größtmöglichen Wert hat).

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Diese For-Schleife wird also O(N) mal ausgeführt, wobei N gleich der Anzahl an Nachbar\*Innen der soeben besuchten Station ist. Da die For-Schleife innerhalb der Do-While Schleife ausgeführt wird, könnte es schlimmstenfalls passieren, dass wir alle Kanten vergleichen. Also O(K). Jede Kante verursacht maximal einen Heap-Push: O(K \* log(S))

Wenn der Zielknoten gefunden wurde, wird der Pfad ausgegeben.

Wir kommen also insgesamt auf eine O-Notation von O((S+K) \* log S)

# Pfad ausgeben – Ablauf und O-Notation

Zunächst wird geprüft ob in node\_data der Endknoten noch den maximal-Wert als Weg-kosten hat, in diesem Fall wäre er nicht besucht worden, es wird ein Logic\_error geworfen und die Funktion abgebrochen. O(1)

Dann wird ein Vektor aus String-Paaren mit dem Namen *pfadSchritte* deklariert. O(1).

Außerdem wird ein string mit dem Namen *aktuell* und dem Wert von Endknoten initialisiert. O(1)

Dann wird er Pfad vom Endknoten aus Rückwärts gesucht indem immer zum Vorgänger-Knoten gegangen wird. O(L), wobei L der Länge des Pfades entspricht.

Dann wird der Pfad mit dem Algorithmus reverse() in die richtige Reihenfolge gebracht O(L) (Quelle: <https://cplusplus.com/reference/algorithm/reverse/> )

Dann wird für jedes Element im Pfad ausgegeben ob ein Umstieg stattgefunden hat oder nicht und jedenfalls der Name der Station O(L)

Insgesamt kommen wir also auf eine Laufzeitkomplexität von O(L), wobei L die Länge des Pfades ist.