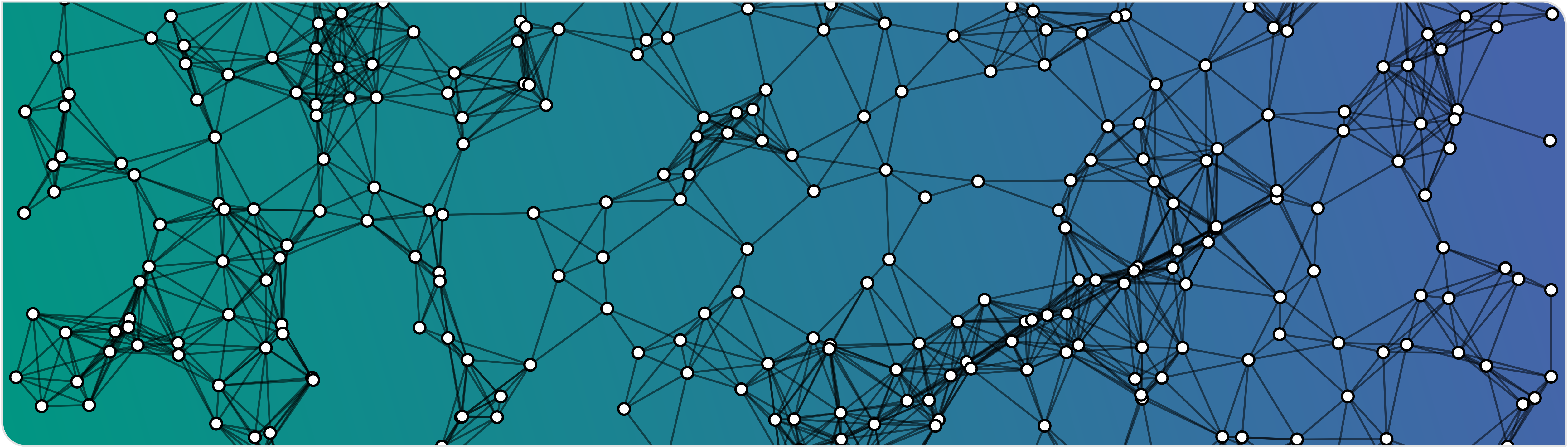
# Algorithmen 1

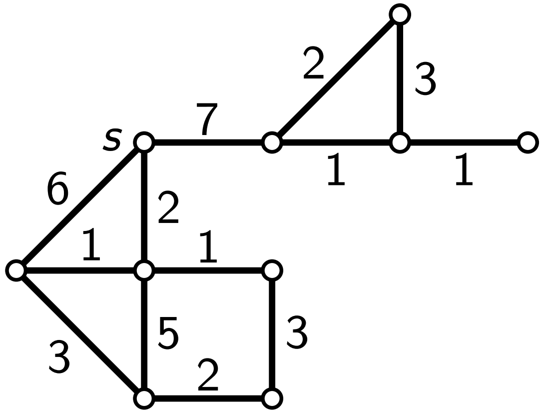
## Kürzeste Wege in gewichteten Graphen – Dijkstras Algorithmus



# Problemstellung

## Gegeben

- ein Graph $G = (V, E)$ (ungerichtet oder gerichtet)
- Gewichtsfunktion $\text{len}\colon E \to \mathbb{Z}$, die jeder Kante $e \in E$ eine **Länge** $\text{len}(e)$ zuordnet
- Startknoten $s$

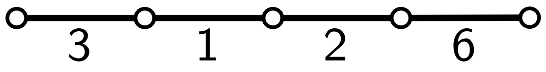


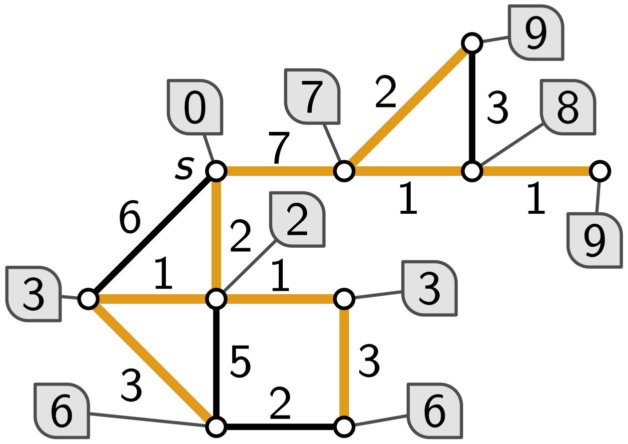

## Pfadlängen und Distanzen

(Kurzform für $\text{len}(\{v_{i-1}, v_i\})$)

- **Länge eines Pfades** $\langle v_0, \ldots, v_k \rangle$: $\sum_{i=1}^{k} \text{len}\,(v_{i-1}, v_i)$
- **Distanz** $\text{dist}(s, t)$: Länge des kürzesten $st$-Pfades

Pfad der Länge 12



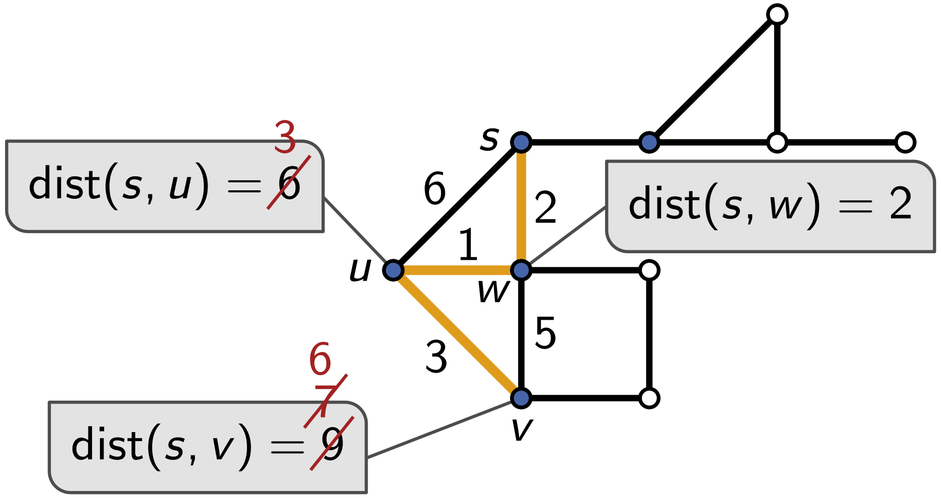




> **Problem: Single-Source Shortest Path (SSSP)**
> Gegeben einen gewichtete Graphen $G = (V, E)$ und einen Knoten $s \in V$, berechne $\text{dist}(s, t)$ für alle $t \in V$.

# Erinnerung: BFS

**Vorgehen**

- arbeite die Knoten layerweise ab
- **Explorieren** eines Knotens $u$ aus Layer $i-1$:
  - betrachte alle Nachbarn $v$ von $u$
  - $v$ ungefärbt $\Rightarrow$ $v$ gehört zu Layer $i$




**Kürzeste Pfade: $v$ in Layer $i \Rightarrow \mathrm{dist}(s,v) = i$**

- $sv$-Pfad der Länge $i$ ($\mathrm{dist}(s,v) \le i$)
  - induktiv: es gibt einen Pfad der Länge $i-1$ zu $u$
  - zusammen mit $\{u, v\} \Rightarrow sv$-Pfad der Länge $i$
- kein kürzerer Pfad möglich ($\mathrm{dist}(s,v) \ge i$):
  - gäbe es einen kürzeren $sv$-Pfad, dann hätte $v$ einen Nachbarn in einem vorherigen Layer
  - layerweise Abarbeitung $\Rightarrow$ $v$ wäre schon gefärbt

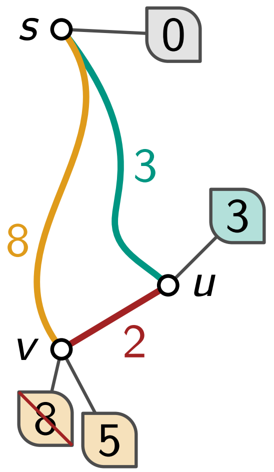
**Mit Kantenlängen?**

- speichere $\mathrm{dist}(s,v)$ statt Layer
- $\mathrm{dist}(s,u) \le 6 \Rightarrow \mathrm{dist}(s,v) \le 9$
- ggf. findet man später einen kürzeren Pfad zu $v$
- oder sogar zu $u$

# Kernproblem beim Umgang mit Kantenlängen

## Plan für das generelle Vorgehen

- speichere $d[v]$ für jeden Knoten $v$: Länge des kürzesten bekannten $sv$-Pfades (Invariante: $\mathrm{dist}(s, v) \leq d[v]$)
- **Explorieren** eines Knotens $u$:
  - betrachte alle Nachbarn $v$ von $u$
  - falls $d[v] > d[u] + \mathrm{len}(u, v)$, setze $d[v] = d[u] + \mathrm{len}(u, v)$




## Anders als bei BFS

- $v$ gefunden $\not\Rightarrow$ kürzester $sv$-Pfad gefunden
- $d[v]$ wird ggf. mehrfach aktualisiert

**nicht schlimm:** keine Zusatzkosten verglichen mit dem Testen ob $v$ schon gefärbt ist

- $v$ schon exploriert, wenn $d[v]$ sich ändert $\to$ $v$ muss erneut exploriert werden

**problematisch:** könnte öfters passieren und ggf. kaskadieren

## Ziel im Folgenden

- $v$ exploriert $\Rightarrow$ kürzester $sv$-Pfad gefunden

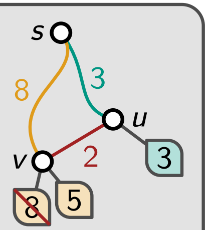
**super:** jeden Knoten einmal explorieren kostet insgesamt $\Theta(m)$ (wie bei BFS)

# Explorierungsreihenfolge → Dijkstras Algorithmus

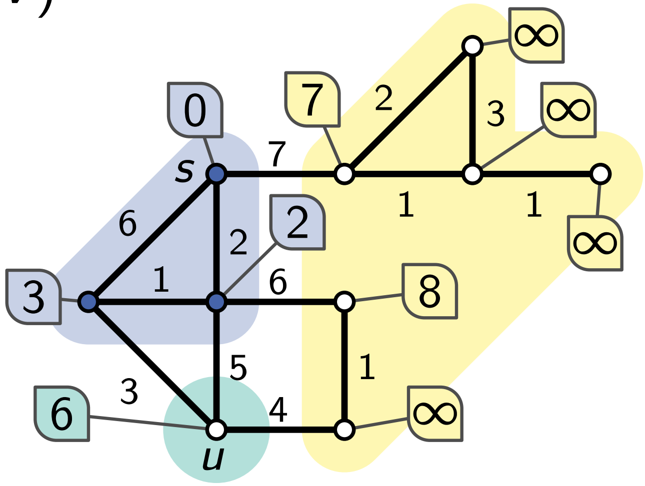
**Ziel:** $v$ exploriert ⇒ kürzester $sv$-Pfad gefunden

**Strategie:** exploriere Knoten $u$ mit minimalem $d[u]$
(unter allen noch nicht explorierten Knoten)

**Erinnerung: Explorieren von $u$**
- betrachte alle Nachbarn $v$ von $u$
- falls $d[v] > d[u] + \text{len}(u, v)$,
  setze $d[v] = d[u] + \text{len}(u, v)$




## Zielführende Strategie
- Annahme: für alle vorher explorierten Knoten $v$ gilt $d[v] = \text{dist}(s, v)$
- zeige: für $u$ gilt ebenfalls $d[u] = \text{dist}(s, u)$
- Intuition:
  - $su$-Pfad über $v$ mit $d[v] < d[u]$ schon gefunden ($v$ exploriert)
  - Knoten $w$ mit $d[w] > d[u]$ explorieren kann $d[u]$ nicht senken
- **Achtung:** wir nehmen hier nicht-negative Kantenlängen an




● bereits exploriert
○ noch nicht exploriert
X aktueller Wert für $d[\cdot]$

## Gleich
- Wrap-Up: Algo-Beschreibung, Umsetzung in Pseudocode, Beispiel
- formaler Korrektheitsbeweis

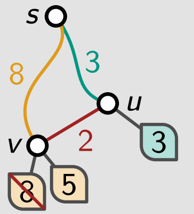
# Dijkstras Algorithmus

## Algorithmus

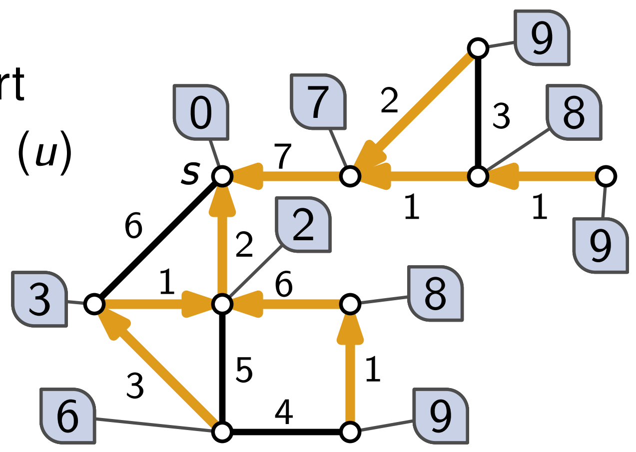
- wiederhole bis alle Knoten exploriert:
  - wähle unexplorierten Knoten $u$ mit minimalem $d[u]$
  - exploriere $u$
- Initialisierung: $d[v] = \infty$ für $v \in V$ und $d[s] = 0$

**Erinnerung: Explorieren von $u$**

- betrachte alle Nachbarn $v$ von $u$
- falls $d[v] > d[u] + \text{len}(u, v)$, setze $d[v] = d[u] + \text{len}(u, v)$




schon exploriert
am explorieren ($u$)
unexploriert
X $d[\cdot]$




**Dijkstra**(*Graph G*, *Node s*)

```
d := Array of size n initialized with ∞
d[s] := 0
while there is an unexplored node do
    Node u := unexp. node with min. d[u]
    for Node v in N(u) do
        if d[v] > d[u] + len(u, v) then
            d[v] := d[u] + len(u, v)
```

## Konstruktion der kürzesten Pfade (wie bei BFS)

- jeder Knoten merkt sich seinen Vorgänger

# Ausblick: Priority-Queue

**Wie finden wir den unexplorierten Knoten $u$ mit minimalem $d[u]$?**

- naiv: einmal alle Knoten anschauen $\rightarrow \Theta(n)$ pro Iteration $\rightarrow \Theta(n^2)$ insgesamt
- Datenstruktur: Priority-Queue (Prioritätswarteschlange)
  - **push**$(v, 7)$: füge $v$ mit Priorität 7 ein
  - **popMin**$()$: extrahiere Element mit kleinster Priorität
  - **decPrio**$(v, 4)$: verkleinere Priorität von $v$ auf 4

**Wie schnell gehen die Operationen?**

- unterschiedliche Varianten mit unterschiedlichen Laufzeiten
- viele schnelle Varianten: $O(\log n)$ pro Operation
- Fibonacci-Heap: $\Theta(\log n)$ für **popMin**, $\Theta(1)$ für **push** und **decPrio**

**Wie geht das?**

- clevere Verwaltung der Elemente in einer Baumstruktur
- werden wir bald genauer kennen lernen

# Dijkstras Algorithmus mit Priority-Queue

**Dijkstra**(*Graph* $G$, *Node* $s$)

```
d := Array of size n initialized with ∞
d[s] := 0
PriorityQueue Q := empty priority queue
for Node v in V do
    Q.push(v, d[v])
while Q ≠ ∅ do
    u := Q.popMin()
    for Node v in N(u) do
        if d[v] > d[u] + len(u, v) then
            d[v] := d[u] + len(u, v)
            Q.decPrio(v, d[v])
```

## Anmerkung

- man könnte die Knoten auch beim ersten Entdecken erst in die Queue einfügen
- dann muss man sich merken, welche schon eingefügt sind (ähnlich zu BFS)

## Laufzeitanalyse

- $n$ Mal **push**
- jeder Knoten wird nur einmal aus der Queue entfernt $\Rightarrow$ $n$ Mal **popMin**
- jede Kante nur zweimal betrachtet (einmal von jeder Seite) $\Rightarrow$ $m$ Mal **decPrio**
- $\Theta(n \log n + m)$, wenn **decPrio** in $\Theta(1)$
- $\Theta((n + m) \log n)$, wenn **decPrio** in $\Theta(\log n)$

# Korrektheit: Überblick

Karlsruher Institut für Technologie

**Theorem**
Gegeben einen Graphen $G = (V, E)$, Kantenlängen $\mathrm{len}\colon E \to \mathbb{N}$ und $s \in V$, dann berechnet Dijkstras Algorithmus in $\Theta(n \log n + m)$ Zeit die Distanzen $\mathrm{dist}(s, v)$ für alle $v \in V$.

**Anmerkung:** die Kantenlängen sind hier nicht-negativ

**Beweis**
- Laufzeit: gerade gesehen
- Korrektheit: zeige die folgenden zwei Lemmas

**Lemma** (der Algo behauptet nicht Pfade zu kennen, die es gar nicht gibt)
Wenn $d[v] < \infty$, dann gibt es einen $sv$-Pfad der Länge $d[v]$. Das heißt $d[v] \geq \mathrm{dist}(s, v)$ gilt zu jedem Zeitpunkt.

**Lemma** (der Algo findet tatsächlich den kürzesten Pfad)
In dem Moment, in dem $u$ aus der Queue entfernt und exploriert wird gilt $d[u] = \mathrm{dist}(s, u)$.

# Korrektheit: Algo kennt nur existiernede Pfade

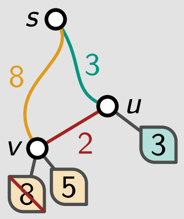
**Lemma** (der Algo behauptet nicht Pfade zu kennen, die es gar nicht gibt)
Wenn $d[v] < \infty$, dann gibt es einen $sv$-Pfad der Länge $d[v]$. Das heißt $d[v] \geq \mathrm{dist}(s, v)$ gilt zu jedem Zeitpunkt.

## Mehr oder weniger offensichtlich

- Invariante gilt am Anfang
- bleibt beim Explorieren eines Knotens erhalten

**Erinnerung: Explorieren von $u$**

- betrachte alle Nachbarn $v$ von $u$
- falls $d[v] > d[u] + \mathrm{len}(u, v)$, setze $d[v] = d[u] + \mathrm{len}(u, v)$




## Beweis im Detail

- Invariante gilt nach der Initialisierung: $d[s] = 0$ und $d[v] = \infty$ für alle $v \in V \setminus \{s\}$
- $d[v]$ kann nur durch das Explorieren eines Nachbarn $u$ geändert werden
  - da Invariante vorher galt: es gibt $su$-Pfad der Länge $d[u]$
  - Invariante bleibt erhalten: es gibt $sv$-Pfad der Länge $d[u] + \mathrm{len}(u, v)$

# Korrektheit: Algo findet kürzesten Pfad

**Lemma** (der Algo findet tatsächlich den kürzesten Pfad)
In dem Moment, in dem $u$ aus der Queue entfernt und exploriert wird gilt $d[u] = \text{dist}(s, u)$.

## Intuition

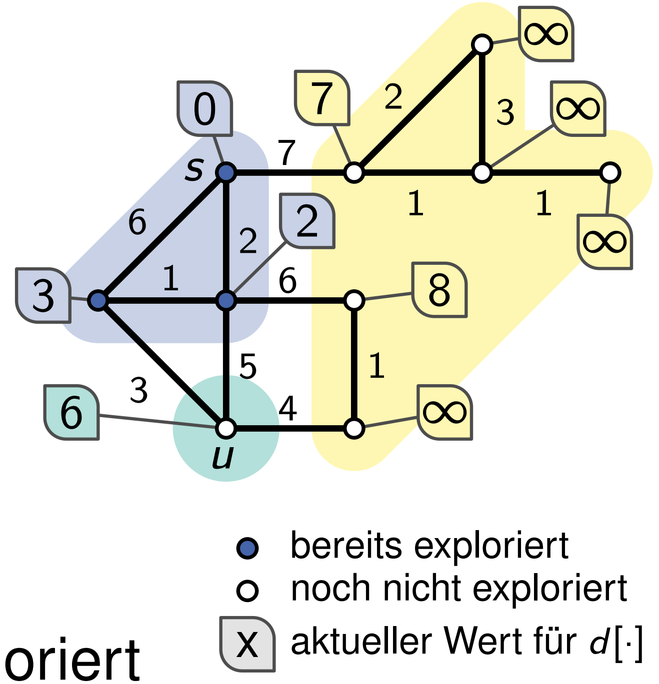
- Erinnerung: wir explorieren immer den unexplorierten Knoten $u$ mit minimalem $d[u]$
- $su$-Pfad über $v$ mit $d[v] < d[u]$ schon gefunden ($v$ exploriert)
- Knoten $w$ mit $d[w] > d[u]$ explorieren kann $d[u]$ nicht senken

## Problem: das ist noch kein Beweis

- nur weil wir später keinen kürzeren Pfad mehr finden können heißt das nicht, dass wir jetzt den kürzesten Pfad haben

## Nützliche Beobachtung

- Knoten werden in nicht-absteigender Reihenfolge exploriert
- also: wenn $d[v] < d[u]$ beim Explorieren, dann wird $v$ vor $u$ exploriert



# Korrektheit: Algo findet kürzesten Pfad

**Lemma** (der Algo findet tatsächlich den kürzesten Pfad)
In dem Moment, in dem $u$ aus der Queue entfernt und exploriert wird gilt $d[u] = \mathrm{dist}(s, u)$.

**Beweis:** angenommen es stimmt nicht

- wähle Gegenbeispiel $u$ mit $d[u] > \mathrm{dist}(s, u)$, sodass $u$ der erste solche Knoten auf einem kürzesten $su$-Pfad ist

**Notation**
- $d[v]$: Länge des kürzesten bekanntesten Pfades von $s$ nach $v$ **beim Explorieren**
- $\mathrm{dist}(s, v)$: tatsächliche Distanz von $s$ nach $v$
- $\mathrm{len}(v, u)$: Länge der Kante $\{v, u\}$

**Einschub: Beweistechnik**
- naheliegendes Argument: $u$ ist Gegenbeispiel $\Rightarrow$ ein Nachbar von $u$ ist Gegenbeispiel
- damit hangeln wir uns von Gegenbeispiel zu Gegenbeispiel
- und müssen zeigen: irgendwann endet das in einem Widerspruch
- schöner: starte mit einem **minimalen Gegenbeispiel**

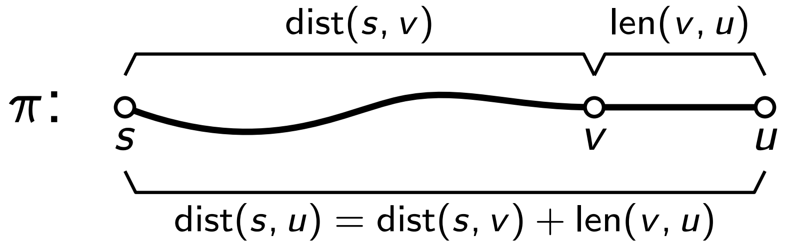
# Korrektheit: Algo findet kürzesten Pfad

**Lemma** (der Algo findet tatsächlich den kürzesten Pfad)
In dem Moment, in dem $u$ aus der Queue entfernt und exploriert wird gilt $d[u] = \text{dist}(s, u)$.

**Beweis:** angenommen es stimmt nicht

- wähle Gegenbeispiel $u$ mit $d[u] > \text{dist}(s, u)$, sodass $u$ der erste solche Knoten auf einem kürzesten $su$-Pfad ist
- sei $v$ der Vorgänger von $u$ auf diesem Pfad

(${v}$ ist also kein Gegenbeispiel)




**Notation**
- $d[v]$: Länge des kürzesten bekanntesten Pfades von $s$ nach $v$ **beim Explorieren**
- $\text{dist}(s, v)$: tatsächliche Distanz von $s$ nach $v$
- $\text{len}(v, u)$: Länge der Kante $\{v, u\}$

**Nützliche Beobachtung**
- Knoten werden in nicht-absteigender Reihenfolge exploriert
- also: wenn $d[v] < d[u]$ beim Explorieren, dann wird $v$ vor $u$ exploriert

- es gilt: $d[v] = \text{dist}(s, v) \leq \text{dist}(s, u) < d[u]$
- damit wird $v$ vor $u$ exploriert
- dabei wird $d[u]$ auf $d[v] + \text{len}(v, u) = \text{dist}(s, v) + \text{len}(v, u) = \text{dist}(s, u)$ gesetzt

# Dijkstras Algo: Zusammenfassung und Anmerkungen

**Theorem**
Gegeben einen Graphen $G = (V, E)$, Kantenlängen $\mathrm{len}\colon E \to \mathbb{N}$ und $s \in V$, dann berechnet Dijkstras Algorithmus in $\Theta(n \log n + m)$ Zeit die Distanzen $\mathrm{dist}(s, v)$ für alle $v \in V$.

## Anmerkungen

- funktioniert für gerichtete und ungerichtete Graphen
- wir brauchen nicht-negativen Kantenlängen
- das Betrachten der Kante $e = \{u, v\}$ beim Explorieren von $u$ heißt auch **relaxieren** von $e$

## Nächstes Mal

- Umgang mit negativen Kantenlängen: Bellman–Ford
- Berechnung der Distanzen zwischen allen Paaren: Floyd–Warshall

## Später

- effiziente Priority-Queues

# Ausblick: Routenplanung in der Praxis

**Kürzeste Wege auf Straßengraphen**

- Europa hat ca. 18 M Knoten und 42 M Kanten
- Dijkstra braucht mehrere Sekunden
- ungeeignet für viele Anwendungen

**Problem:** viel besser als $O(n \log n + m)$ kann es nicht werden

**Beobachtung:** der Eingabegraph ist immer der gleiche

**Idee**

- zwei Phasen:
  1. generiere Zusatzinformationen über $G$
  2. **beschleunigte Anfrage** dank dieser Information
- drei Kriterien: Beschleunigung, Vorberechnungszeit, Speicherplatz

# Ausblick: Routenplanung in der Praxis (Beispiele)

### Zielgerichtetes Routing (Arc-Flags)

- manche Straßen in Karlsruhe sind nicht relevant, wenn man nach Berlin möchte
- Vorberechnung: berechne für jede Kante, ob sie für ein Ziel in Berlin relevant ist
- Anfrage: Zielknoten liegt in Berlin $\to$ Dijkstra, aber ignoriere irrelevanten Kanten

### Ausnutzung von Transitknoten

- Situation: wir wollen von Karlsruhe weit weg fahren
- kürzester Weg führt dann immer erstmal auf die Autobahn
- es gibt nicht so viele Autobahnauffahrten $\to$ wenige Transitknoten
- berechne Distanzen zwischen allen Paaren von Transitkonten vor

### Anmerkungen

- es lohnt sich: bis zu $1\,000\,000$ mal schneller als Dijkstra
- viele der Algorithmen wurden hier in Karlsruhe entwickelt
- mehr davon? $\to$ Vorlesung Algorithmen für Routenplanung (Master)

# Zusammenfassung

Karlsruher Institut für Technologie

## Dijkstras Algorithmus

- vermutlich der bekannteste Graphalgorithmus (vielleicht nach BFS und DFS)
- effiziente Berechnung kürzester Wege
- Wunschzettel: eine schnelle Priority-Queue

## Beweistechniken

- Invariante (Algo kennt nur Pfade, die es wirklich gibt)
- Widerspruch mit minimalem Gegenbeispiel (Algo findet den kürzesten Pfad)

## Ausblick Routenplanung

- Algorithmische Technik: Vorberechnung, wenn bei vielen Anfragen auf den selben Daten
- kann Anfragen sehr beschleunigen