

# Algorithmen und Datenstrukturen

## Vorlesung #09 – Kürzeste Wege in Graphen

Benjamin Blankertz



Lehrstuhl für Neurotechnologie, TU Berlin

[benjamin.blankertz@tu-berlin.de](mailto:benjamin.blankertz@tu-berlin.de)



13 · Jun · 2023

# Themen der heutigen Vorlesung

- ▶ Topologische Sortierung (siehe VL #07)
- ▶ Kantengewichtete Digraphen
- ▶ Baum der kürzesten Wege von einem Startknoten
- ▶ Relaxation
- ▶ Bestimmung kürzester Wege:
  - ▶ in azyklischen Digraphen (mit Topologischer Sortierung)
  - ▶ in Digraphen mit nicht-negativen Gewichten (Dijkstra Algorithmus)
  - ▶ in Digraphen ohne negative Zyklen (Bellman-Ford Algorithmus)

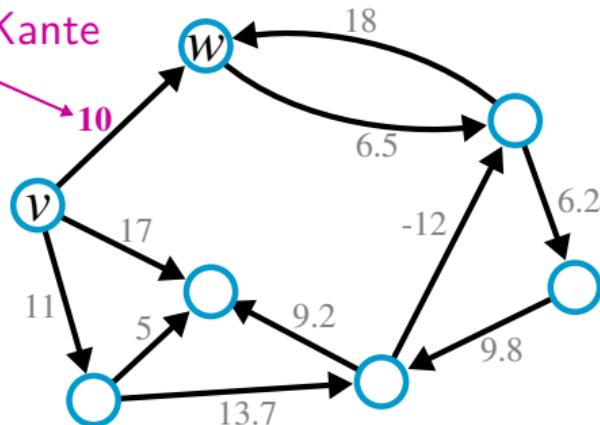
# Kürzeste Wege in gewichteten Graphen

- ▶ Breitensuche: Wege von einem Startknoten zu Endknoten mit **minimaler Anzahl von Kanten**
- ▶ Für Navigation u. Ä.: **kürzeste Wege** im Sinne der Gesamtstrecke oder -fahrzeit
- ▶ Allgemein: bestimme Weg mit kleinsten Gesamtgewicht (Summer der Gewichte aller benutzten Kanten)
- ▶ So kann z. B. Verbrauch oder Kosten für Wegstrecken berücksichtigt werden.
- ▶ Das Finden kürzester Wege hat viele wichtige Anwendungen, z. B. in Bildbearbeitung, *Scheduling*, *Routing*, ...

# Kantengewichtete Digraphen

Gewicht der Kante

$$g(v,w) = 10$$



- ▶ Formal:  $g(v,w)$  oder  $weight(v,w)$  für das **Gewicht der Kante**  $v \rightarrow w$ .
- ▶ Auch negative Gewichte sind erlaubt.

**Änderung in der Implementation** gegenüber ungewichteten Digraphen:

- ▶ Neue Klasse für gewichtete Kanten, die Start- und Endknoten, sowie das Gewicht enthalten.
- ▶ Die Inzidenzlisten enthalten für jeden Knoten die **ausgehenden Kanten**.
- ▶ Dabei ist die Information des Startknotens redundant. Dies wird in Kauf genommen, da es die Implementierung mancher Algorithmen vereinfacht.

# API für einen kantengewichteten Digraphen

## API für eine gewichtete gerichtete Kante

```
public class DirectedEdge implements Comparable<DirectedEdge>
```

DirectedEdge(int v, int w, double weight)	Kante v->w mit Gewicht weight
---	-------------------------------

double weight()	Gewicht der Kante
-----------------	-------------------

int from()	Startknoten dieser Kante
------------	--------------------------

int to()	Endknoten dieser Kante
----------	------------------------

int compareTo(DirectedEdge that)	Vergleicht die Kantengewichte
----------------------------------	-------------------------------

## API für einen kantengewichteten Digraphen

```
public class EdgeWeightedDigraph
```

EdgeWeightedDigraph(int V)	Erzeugt leeren Digraphen mit V Knoten
----------------------------	---------------------------------------

int V()	Anzahl der Knoten
---------	-------------------

void addEdge(DirectedEdge e)	Füge Kante e zum Digraphen hinzu
------------------------------	----------------------------------

Iterable<DirectedEdge> incident(int v)	Kanten inzident zu v
--	----------------------

## Kürzester Weg

Finde den Weg mit kleinstem Gewicht in einem kantengewichteten Digraphen von einem Startknoten zu einem Zielknoten.

### Variante:

- ▶ Finde kürzeste Wege von einem Startknoten zu **allen** erreichbaren Knoten (*single-source shortest paths; SSSP*).

### Mögliche Einschränkungen:

- ▶ Der Graph ist azyklisch (DAG).
- ▶ Die Kantengewichte sind euklidsche Abstände in der Ebene.
- ▶ Die Kantengewichte sind nicht-negativ.
- ▶ Die Gewichte sind beliebig und der Graph besitzt keine negativen Zyklen.

Für die unterschiedlichen Einschränkungen sind unterschiedliche Algorithmen geeignet.

## Single Source Shortest Paths (SSSP)

- ▶ Herausforderung: Finde kürzeste Wege zu allen erreichbaren Knoten
- ▶ Für speziellen Zielknoten: Führe SSSP Algorithmus aus und stoppe, sobald der Zielknoten erreicht wurde.
- ▶ Kürzeste Wege sind nur dann definiert, wenn man auf dem Weg von Start- zu Endknoten keine **negative Zyklen** (= Zyklen mit negativem Gesamtgewicht) passieren kann.
  - Dann würde jede zusätzliche Runde durch diesen Zyklus das Gewicht des Weges verringern, es gäbe also keinen kürzesten Weg.
- ▶ Wir setzen im Folgenden voraus, dass die Graphen keine negativen Zyklen besitzen.

## Baum der kürzesten Wege (*Shortest Paths Tree, SPT*)

- ▶ Zunächst: Datenstrukturen zur Speicherung der Information für die kürzeste Wege
- ▶ Das Gewicht eines kürzesten Weges zu einem Zielknoten  $t$  wird in einem knotenindizierten Array  $\text{dist}$  unter Index  $t$  gespeichert.
- ▶ Für die Speicherung der Wege hilft uns der folgende Satz:

Die SSSP Wege können als Baum dargestellt werden

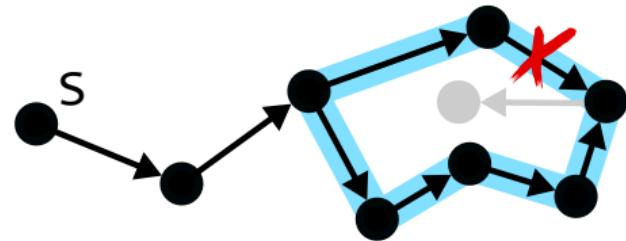
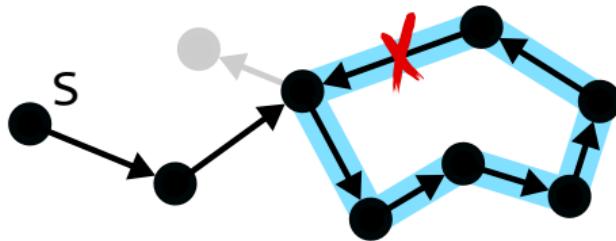
Kürzeste Wege von einem Startknoten können als ein Baum repräsentiert werden. Falls es zu einem Zielknoten mehrere kürzeste Wege gibt, wird nur einer repräsentiert.

- ▶ Dieser Baum wird **Baum der kürzesten Wege** (*Shortest Paths Tree, SPT*) genannt.

# Baum der kürzesten Wege

## Beweis.

- ▶ Z.z: Im Untergraphen der kürzesten Wege können Zyklen vermieden werden.
- ▶ Nehmen wir an, dass es einen Zyklus im Untergraphen gibt.
- ▶ Dies kann kein gerichteter Zyklus sein, da darin ein Pfeil keinen Beitrag zu einem kürzesten Weg haben kann (siehe Abbildung links).
- ▶ In dem ungerichteten Zyklus muss es einen Knoten geben, auf den zwei Pfeile zeigen. Beide Pfeile müssen dasselbe Gewicht haben. Einer der beiden Pfeile kann entfernt werden. Damit ist auch der Zyklus entfernt (siehe Abbildung rechts).



# Baum der kürzesten Wege (SPT)

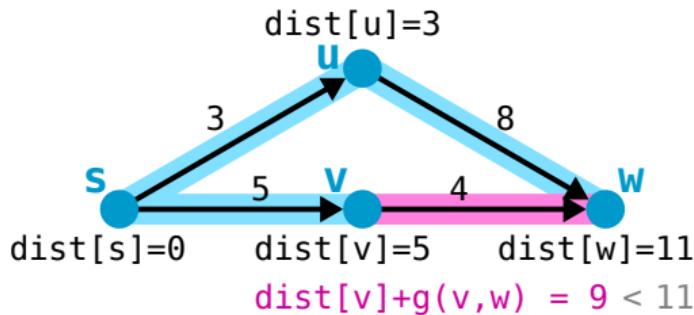
- ▶ Kürzeste Wege können also effizient als Baum gespeichert werden, wie bei der Graphendurchsuchung.
- ▶ `dist[v]` speichert die Länge eines kürzesten, aktuell bekannten Weges von `s` nach `v`
- ▶ `parent[v]` speichert den Vorgänger auf diesem Weg

```
public class SingleSourceShortestPaths
{
    private double[] dist;      // Länge eines aktuell kürzesten Weges,
                                // oder INF falls keiner bekannt
    private int[] parent;       // Vorgängerknoten im Baum der aktuell kürzesten Wege

    public double dist(int v)   { return dist[v]; }
    public boolean hasPathTo(int v) { return dist[v] < Double.POSITIVE_INFINITY; }
    public Iterable<int> pathTo(int v) // wie zuvor
}
```

# Relaxation

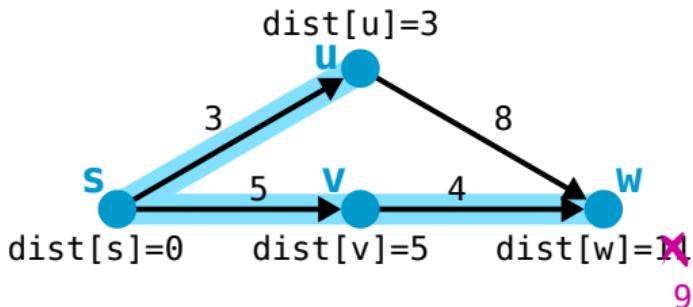
- Viele SSSP Algorithmen beruhen auf der **Relaxation**:  
Wenn eine Kante  $v \rightarrow w$  zu einem kürzeren Weg führt, nutze sie! Genauer:
  - Falls der kürzeste bekannte Weg zu  $w$  länger ist als der kürzeste bekannte Weg zu  $v$  plus dem Gewicht der Kante  $v \rightarrow w$ ,
  - dann aktualisiere den Weg nach  $w$ : Nehme den Weg nach  $v$  und dann Pfeil  $v \rightarrow w$ .



```
public void relax(DirectedEdge e)
{
    int v = e.from();
    int w = e.to();
    if (dist[w] > dist[v] + e.weight()) {
        dist[w] = dist[v] + e.weight();
        parent[w] = v;
    }
}
```

# Relaxation

- ▶ Viele SSSP Algorithmen beruhen auf der **Relaxation**:  
Wenn eine Kante  $v \rightarrow w$  zu einem kürzeren Weg führt, nutze sie! Genauer:
  - Falls der kürzeste bekannte Weg zu  $w$  länger ist, als der kürzeste bekannte Weg zu  $v$  plus dem Gewicht der Kante  $v \rightarrow w$ ,
  - dann aktualisiere den Weg nach  $w$ : Nehme den Weg nach  $v$  und dann Pfeil  $v \rightarrow w$ .



```
public void relax(DirectedEdge e)
{
    int v = e.from();
    int w = e.to();
    if (dist[w] > dist[v] + e.weight()) {
        dist[w] = dist[v] + e.weight();
        parent[w] = v;
    }
}
```

- ▶ Hintergrund des Begriffs *Relaxation*: Gespanntes Gummi von  $s$  über  $u$  nach  $w$  wird entspannt, wenn es über  $v$  anstatt über  $u$  geleitet wird.

## Bedingungen zur Feststellung kürzester Wege

- ▶ Die Erfüllung der Relaxationsbedingung für jede **einzelne Kante**, impliziert die Kürzeste-Wege-Eigenschaft für den **ganzen Digraphen**!
- ▶ Dies gibt Hoffnung auf einen **greedy Algorithmus** für SSSP:
- ▶ Wähle immer einen Schritt mit lokaler Verbesserung (Relaxierung) und erreiche dadurch ein globales Optimum!

## Bedingung für kürzeste Wege

Für einen kantengewichteten Digraphen gilt die folgende Äquivalenz:

Die Werte  $d(v)$  sind genau dann kürzeste Wege von Knoten  $s$ , wenn gilt:

- ▶ Für jeden Knoten  $v$  ist  $d(v)$  die Länge eines Weges von  $s$  nach  $v$  und
- ▶ für jede Kante  $v \rightarrow w$  gilt  $d(w) \leq d(v) + g(v, w)$ .

**Beweis.**  $\Rightarrow$

- ▶ Wir nehmen an, dass die letzte Bedingung *nicht* stimmt und zeigen, dass dies zu einem Widerspruch mit der Voraussetzung an  $d(v)$  führt.
- ▶ Es gibt also eine Kante  $v \rightarrow w$  mit Relaxierungsbedarf:  $d(w) > d(v) + g(v, w)$ .
- ▶ Die Länge des Wegs  $s \rightsquigarrow v \rightarrow w$  ist  $d(v) + g(v, w)$ , also kürzer als  $d(w)$ .
- ▶ Widerspruch dazu, dass die Werte  $d(\cdot)$  die Längen der kürzesten Wege sind. □

# Bedingungen zur Feststellung kürzester Wege

## Bedingung für kürzeste Wege

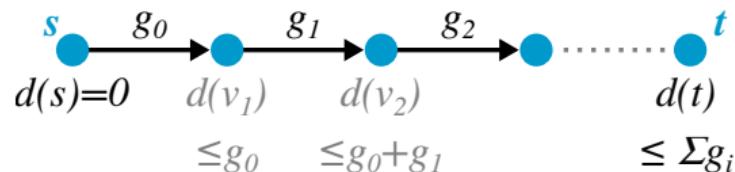
Für einen kantengewichteten Digraphen gilt die folgende Äquivalenz:

Die Werte  $d(v)$  sind genau dann kürzeste Wege von Knoten  $s$ , wenn gilt:

- ▶ Für jeden Knoten  $v$  ist  $d(v)$  die Länge eines Weges von  $s$  nach  $v$  und
- ▶ für jede Kante  $v \rightarrow w$  gilt  $d(w) \leq d(v) + g(v, w)$ .

**Beweis.**  $\Leftarrow$

- ▶ Wir betrachten einen kürzesten Weg von  $s$  nach  $t$ :  $v_0 (= s), v_1, \dots, v_K (= t)$ .
- ▶ Aus  $d(w) \leq d(v) + g(v, w)$  für alle Kanten des Weges folgt:  $d(t) \leq \sum_{i < K} g_i$
- ▶ Die Summe über  $g_i$  ist die Länge des kürzesten Weges. Also muss  $d(t)$  gleich dieser Länge sein, was zu zeigen war.  $\square$



# Allgemeiner Algorithmus für kürzeste Wege

```
1 // dist: Array der Größe V
2 // parent: Array der Größe V
3 for all nodes  $v \in V$ 
4    $dist[v] \leftarrow \infty$ 
5    $dist[s] \leftarrow 0$ 
6 while exists edge  $e=(v,w) \in E$  with  $dist[w] > dist[v] + e.weight$  do
7   relax( $e$ )           // wie auf Folie 'Relaxation' angegeben
```

- ▶ In jedem Schritt ist  $dist[v]$  die Länge eines einfachen Weges von  $s$  nach  $v$  oder  $\infty$ , falls  $v$  (noch) nicht erreicht wurde.
- ▶ Für erreichte Knoten ist  $parent[v]$  der Vorgänger von  $v$  auf jenem Weg.
- ▶ **Korrektheit:** Nach voriger Bedingung (S. 12) ist am Ende des Algorithmus  $dist[v]$  die Länge des kürzesten Weges von  $s$  nach  $v$  für alle erreichbaren  $v$ .
- ▶ **Terminierung:** In jedem Schritt wird ein  $dist$  Wert verkleinert. Es kann nur endlich viele Schritte geben, da es eine untere Schranke für die Distanzverkleinerung gibt (kleinste Differenz zweier Kantengewichte), Ausnahme nächste Seite.

# Konkrete Algorithmen für kürzeste Wege

- ▶ Gibt es immer kürzeste Wege?
- ▶ **Nein**, wenn der Digraph negative Zyklen hat, die von  $s$  erreichbar sind, nicht. Jeder weitere Durchlauf des Zyklus ergibt einen Weg mit geringerem Gewicht.
- ▶ Wir beschränken uns im Folgenden auf Digraphen **ohne negative Zyklen**.
- ▶ Je nachdem welche weiteren Einschränkungen des Graphen bekannt sind, sind unterschiedliche Algorithmen geeignet, um kürzeste Wege zu finden:
- ▶ **Azyklische Digraphen (DAG)**: mit Topologischer Sortierung
- ▶ **Digraphen ohne negative Gewichte**: Dijkstra Algorithmus
- ▶ **Allgemeine Digraphen** (ohne negative Zyklen): Bellman-Ford Algorithmus

# Kürzeste Wege in gewichteten azyklischen Digraphen

- ▶ Man nehme eine topologische Reihenfolge des DAG und gehe in der Reihenfolge bis Startknoten  $s$ .
- ▶ Von dort werden alle ausgehenden Kanten relaxiert.
- ▶ Dieser Prozess wird iterativ mit dem folgenden Knoten in der topologischen Reihenfolge fortgesetzt.

```
1 for each node  $v$ 
2    $dist[v] \leftarrow \infty$ 
3 end
4  $dist[s] \leftarrow 0$ 
5 determine a topological sorting
6 for each node  $v$  in topological order // beginnend bei  $s$ 
7   for each  $w$  with  $v \rightarrow w \in E$ 
8     relax( $v \rightarrow w$ )           // wie auf Folie 'Relaxation' angegeben
9   end
10 end
```

## Korrektheit und Laufzeit

Der Algorithmus mit topologischer Sortierung bestimmt kürzeste Wege in jedem gewichteten DAG von Startknoten  $s$  in Laufzeit  $O(V + E)$

### Korrektheit:

- ▶ Jede Kante  $e = v \rightarrow w$  wird nur einmal relaxiert, und zwar wenn der Knoten  $v$  an der Reihe ist.
- ▶ Die bei der Relaxation von  $e = v \rightarrow w$  hergestellte Bedingung  $dist[w] \leq dist[v] + e.weight$  bleibt gültig:
  - ▶  $dist[v]$  bleibt unverändert, da alle Kanten mit  $v$  als Endknoten nach der topologischen Sortierung schon vorher bearbeitet worden sein müssen.
  - ▶  $dist[w]$  kann durch weitere Relaxierungen nur kleiner werden.
- ▶ Die Bedingung für kürzeste Wege S. 12 ist also erfüllt.  $\square$

## Korrektheit und Laufzeit

Der Algorithmus mit topologischer Sortierung bestimmt kürzeste Wege in jedem gewichteten DAG von Startknoten  $s$  in Laufzeit  $\mathcal{O}(V + E)$ .

### Laufzeit:

- ▶ Jede Kante, die in der Sortierung nach  $s$  kommt, wird nur einmal relaxiert, und *relax* hat konstante Laufzeit.
- ▶ Also hat die Schleife eine Laufzeit in  $\mathcal{O}(E)$ .
- ▶ Die Initialisierung der Arrays benötigt  $\mathcal{O}(V)$  und die topologische Sortierung  $\mathcal{O}(V + E)$ .
- ▶ Da diese drei Schritte nacheinander ausgeführt werden, ergibt sich für die Gesamtlaufzeit ebenfalls  $\mathcal{O}(V + E)$ . □

# Anwendung des topologischen Algorithmus für kürzeste Wege

- Mit Hilfe des gerade vorgestellten Algorithmus kann eine interessante Anwendung aus der Bildbearbeitung realisiert werden: *Content-aware image resizing* durch *seam carving*.

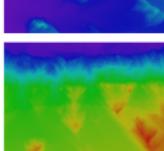
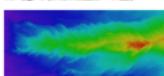
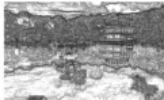
## Seam Carving for Content-Aware Image Resizing

Shai Avidan

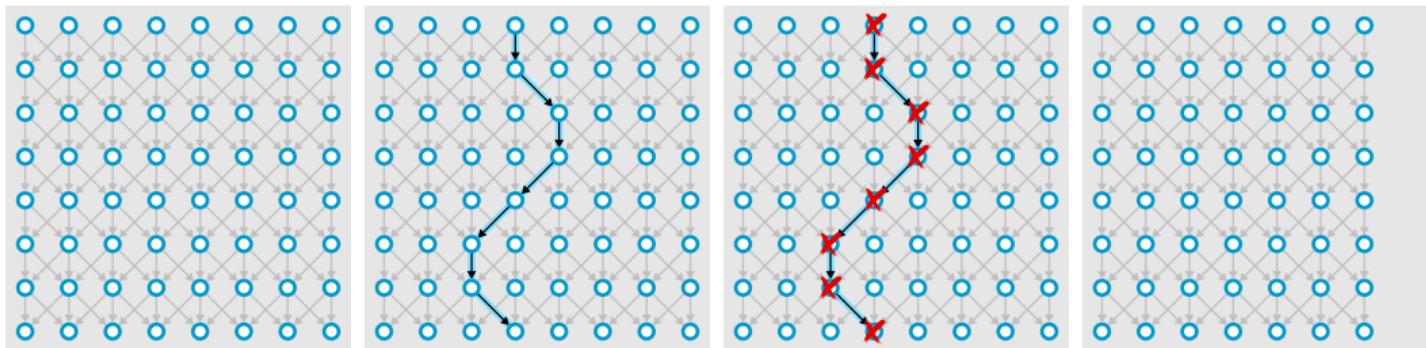
Mitsubishi Electric Research Labs

Ariel Shamir

The Interdisciplinary Center & MERL



## Ansatz für Graphen-basiertes *Seam Carving*



- ▶ Jeder Pixel wird als Knoten in einem Digraphen dargestellt.
- ▶ Für vertikale Nähte gehen von jedem Knoten drei Kanten zu den unteren Nachbarn.
- ▶ Als Gewicht wird der Kontrast zwischen den verbundenen Pixel genommen.
- ▶ Die Naht ist der kürzeste Pfad von oben nach unten.
- ▶ Die Pixel entlang der Naht werden gelöscht.
- ▶ Die tatsächliche Implementierung benutzt eine Gewichtung der **Knoten**, die die Nachbarschaft der Pixel einbezieht.

## Kürzeste Wege in nicht-negativ gewichteten Digraphen

- ▶ Die Voraussetzung der **Zyklenfreiheit** ist sehr einschränkend.
- ▶ Im Folgenden erlauben wir Zyklen.
- ▶ Der **Dijkstra Algorithmus** ist effizient, wenn alle Gewichte **nicht-negativ** sind.
- ▶ Auch er folgt dem allgemeinen Ansatz von S. 13.
- ▶ Es wird ein SPT Suchbaum ausgehend von einem Startknoten  $s$  aufgebaut.
- ▶ In jedem Schritt wird ein Knoten über eine **kreuzende Kante** ausgewählt.

# Dijkstra Algorithmus für kürzeste Wege

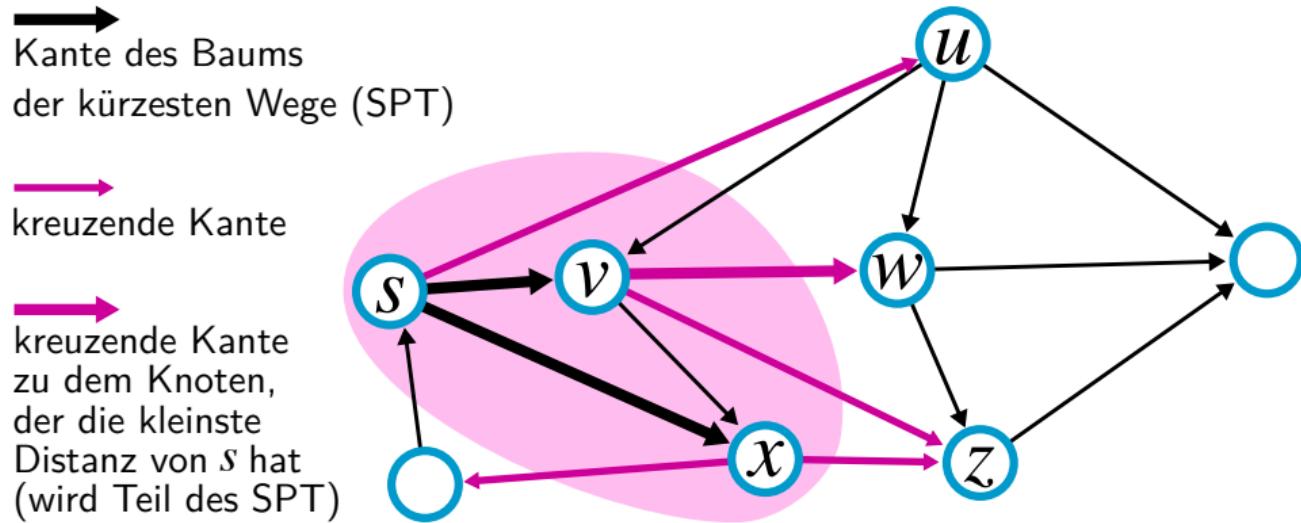
- ▶ Bei Dijkstra wird derjenige Knoten ausgewählt, der die kleinste Distanz von  $s$  hat. Die benutzte Kante wird relaxiert und der Knoten markiert.
- ▶ Der markierte Bereich muss nicht explizit gespeichert werden. Im folgenden Pseudocode ist  $M$  der Rand des markierten Bereiches.

```
1 // initialisiere 'dist' wie zuvor
2  $M \leftarrow \{s\}$ 
3 while  $M \neq \emptyset$ 
4    $v \leftarrow$  element of  $M$  with minimum dist
5   delete  $v$  from  $M$ 
6   for each  $w$  with  $v \rightarrow w \in E$ 
7     relax( $v \rightarrow w$ )    //  $w$  wurde entdeckt
8   end                  //  $v$  fertig bearbeitet
9 end
```

```
procedure relax( $v \rightarrow w$ )
10  if  $dist(w) > dist(v) + weight(v \rightarrow w)$ 
11     $dist(w) = dist(v) + weight(v \rightarrow w)$ 
12    parent( $w$ ) =  $v$ ;
13    add  $w$  to  $M$ 
14  end
15
```

- ▶ Die Terminierung wird durch das allgemeine Verfahren (S. 13) impliziert.

## Auswahl der kreuzenden Kante im Dijkstra Algorithmus



- ▶ Zeile 4 des Algorithmus wählt die kreuzende Kante zu demjenigen Knoten  $w$  aus, der in der aktuellen Schätzung der Distanzen von  $s$  den kleinsten Wert hat.
- ▶ Diese Kante  $v \rightarrow w$  muss im SPT enthalten sein, und die aktuelle Distanzschätzung für  $w$  ist die endgültige, ist also die Länge des kürzesten Weges von  $s$  nach  $w$ .

## Korrektheit des Dijkstra Algorithmus

Der Algorithmus von Dijkstra (S. 22) bestimmt kürzeste Wege ausgehend von Startknoten  $s$  in jedem Graphen mit nicht-negativen Gewichten.

- 1 Die Distanzen, über die  $v$  in Zeile 4 ausgewählt wird, sind **monoton steigend**.  
Begründung:

- ▶  $dist(v)$  wurde als kleinster  $dist$ -Wert ausgewählt (Zeile 4).
- ▶ Alle Knoten  $u$ , die zu der Zeit in  $M$  sind, haben also einen größeren  $dist$ -Wert:  
 $dist(u) \geq dist(v)$ .
- ▶ Für Knoten  $w$ , die später in  $M$  eingefügt werden, gilt für ein solches  $u$ :  
 $dist(w) = dist(u) + weight(u \rightarrow w) \geq dist(v)$ , da die Kantengewichte als nicht-negativ vorausgesetzt wurden.

## Korrektheit des Dijkstra Algorithmus

- 2 Nachdem  $v$  in Zeile 4 ausgewählt wurde, wird es weder wieder in  $M$  eingefügt, noch wird sein  $dist$  Wert geändert, da dies nur unter der Bedingung  $dist(v) > dist(u) + weight(u \rightarrow v)$  für ein später ausgewähltes  $u$  passieren würde, was 1 widerspricht.
- 3 Jeder Pfeil  $v \rightarrow w$  wird folglich nur einmal relaxiert. Die dabei hergestellte Bedingung  $dist(w) \leq dist(v) + weight(v \rightarrow w)$  bleibt bis zur Terminierung gültig:
  - ▶  $dist[w]$  kann durch weitere Relaxierungen nur kleiner werden.
  - ▶  $dist[v]$  bleibt nach 2 unverändert.
  - ▶ Die Bedingung für kürzeste Wege (siehe S. 12) ist laut 3 erfüllt. □

# Überlegungen zur Implementation des Dijkstra Algorithmus

- ▶ Es bleibt zu entscheiden, welche Datenstruktur für  $M$  verwendet wird.
- ▶ Die kritische Operation ist das **Entnehmen des kleinsten Elements** bezüglich eines Schlüssels ( $dist$ ).
- ▶ Daher wäre eine Implementation mit einer minPQ **Vorrangwarteschlange** möglich. Die Schlüssel der PQ sind die Knoten, wobei die zugehörigen  $dist$  Werte die Ordnung (**comparable**) definieren.
- ▶ Allerdings muss berücksichtigt werden, dass die Distanzwerte (in *relax*) verändert werden.
- ▶ Das Ändern von Schlüsseln bzw. ihrer Ordnung ist, wie das Löschen, **nicht effizient** in Vorrangwarteschlangen. Daher hatten wir diese Methoden nicht in der API.
- ▶ Für dieses Anforderungsprofil bietet sich die **indizierte Vorrangwarteschlange** an. Die  $dist$  Werte (Schlüssel) werden über Knoten indiziert.

# Pseudocode für den Dijkstra Algorithmus mit IndexMinPQ

```
1   $Q : IndexMinPQ$ 
2  for each node  $v$ 
3       $dist(v) \leftarrow \infty$ 
4  end
5   $dist(s) \leftarrow 0$ 
6   $add(Q, s, 0)$ 
7  while  $Q \neq \emptyset$ 
8       $v \leftarrow poll(Q)$ 
9      for each  $w$  with  $v \rightarrow w \in E$ 
10          $relaxDijkstra(v \rightarrow w)$ 
11     end
12 end
```

```
13
14  procedure  $relaxDijkstra(v \rightarrow w)$ 
15      if  $dist(w) > dist(v) + weight(v \rightarrow w)$ 
16           $dist(w) = dist(v) + weight(v \rightarrow w)$ 
17           $parent(w) = v$ 
18          if  $contains(Q, w)$ 
19               $decreaseKey(Q, w, dist(w))$ 
20          else
21               $add(Q, w, dist(w))$ 
22          end
23      end
```

# Laufzeit des Dijkstra Algorithmus mit IndexMinPQ

- Die maximale Länge der PQ ist  $V$ . Daher haben *poll* (Zeile 8), *decreaseKey* (Zeile 18) und *add* (Zeile 20) eine Laufzeit in  $\mathcal{O}(\log V)$ .
- Gemäß 2 des Korrektheitsbeweises wird die *while* Schleife max.  $V$ -mal ausgeführt.
- Gemäß 3 wird die Relaxierung höchstens  $E$ -mal ausgeführt.
- Damit ergibt sich eine Gesamtlaufzeit in  $\mathcal{O}(E \log V)$ .

## Weitere Verbesserung:

- Mit einem *Fibonacci-Heap* kann die Laufzeit des Dijkstra Algorithmus sogar im *worst-case* auf  $\mathcal{O}(E + V \log V)$  gesenkt werden.
- Allerdings ist die Implementierung eines *Fibonacci-Heaps* aufwändig, siehe [Cormen et al, S. 511ff].

## Nachtrag: Dijkstra Algorithmus bei negativen Gewichten

- ▶ Der Korrektheitsbeweis auf S. 24 gilt nur für Graphen, die **keine negativen Kantengewichte** haben.
- ▶ Die Beweisführung wurde in Hinblick auf die Laufzeitabschätzung gewählt.
- ▶ Durch Verwendung der Voraussetzung, dass die Gewichte nicht-negativ sind, konnte gezeigt werden, dass jede Kante nur einmal relaxiert wird.
- ▶ Diese Garantie der Effizienz des Dijkstra Algorithmus geht verloren, wenn man negative Gewicht zulässt. Die *worst-case* Laufzeit ist dann exponentiell.
- ▶ Der folgende einfachere Beweis zeigt, dass die Korrektheit in diesem Fall trotzdem erhalten bleibt.

## Korrektheit des Dijkstra Algorithmus

Der Algorithmus von Dijkstra bestimmt kürzeste Wege ausgehend von Startknoten  $s$  in jedem gewichteten Graphen ohne negative Zyklen.

- ▶ Für jeden Knoten  $v$ , der von  $s$  erreichbar ist, werden alle Kanten  $v \rightarrow w$  relaxiert.
- ▶ Dabei wird die Bedingung  $dist[w] \leq dist[v] + weight(v \rightarrow w)$  hergestellt.
- ▶ Sie ist auch bei der Terminierung gültig, da
  - $dist[w]$  im Weiteren höchstens verringert wird und
  - falls  $dist[v]$  verringert wird (Zeile 13), wird  $v$  in  $M$  eingefügt (Zeile 14), also wird die Kante  $v \rightarrow w$  nochmals relaxiert und die Bedingung wird wiederhergestellt.
- ▶ Die Terminierung folgt aus dem generellen Ansatz, S. 13.

## Kürzeste Wege in Digraphen mit beliebigen Gewichten

- ▶ Die garantierter Effizienz des Dijkstra Algorithmus gilt nur für **nicht-negative** Gewichte.
- ▶ In einigen Anwendungsfällen kommen diese allerdings vor.
- ▶ So werden z.B. bei der Ablaufplanung über negative Gewichte Vorbedingungen modelliert. (Teilprojekt X muss 2 Monate vor Beginn von Teilprojekt Y beendet sein. Der kürzeste Weg ergibt die minimale Projektlaufzeit.)
- ▶ Wenn auch negative Gewichte erlaubt werden, ist es wichtig zu beachten, dass keine negativen Zyklen auftreten dürfen.
- ▶ In diesem Fall kommt der **Bellman-Ford Algorithmus** zum Einsatz.

## Kürzeste Wege mit dem Bellman-Ford Algorithmus

- ▶ Auch der Bellman-Ford Algorithmus folgt dem allgemeinen Ansatz von S. 13.
- ▶ Er relaxiert  $V - 1$  mal alle Kanten in beliebiger Reihenfolge.
- ▶ Ein Pfeil kann effektiv nur relaxiert werden, wenn für seinen Anfangsknoten schon eine Distanz bestimmt wurde ( $dist(v) < \infty$ ).
- ▶ Daher wächst auch hier ein Suchbaum schrittweise, ausgehend vom Startknoten  $s$ .

# Bellman-Ford Algorithmus für kürzeste Wege

```
1 // initialisiere 'dist' wie zuvor
2 for  $k \leftarrow 1$  to  $V-1$ 
3   for each node  $v$ 
4     for each node  $w$  with  $v \rightarrow w \in E$ 
5        $relax(v \rightarrow w)$ 
6     end
7   end
8 end
9
10 procedure  $relax(v \rightarrow w)$ 
11   if  $dist(w) > dist(v) + weight(v \rightarrow w)$ 
12      $dist(w) = dist(v) + weight(v \rightarrow w)$ 
13      $parent(w) = v;$ 
14 end
```

## Korrektheit des Bellman-Ford Algorithmus

Bei Anwendung des Bellman-Ford Algorithmus auf einen Digraphen ohne negative Zyklen gilt: Nach  $k$  Schritten sind kürzeste Wege von dem Startknoten  $s$  für alle Knoten  $v$  bestimmt und in  $dist(v)$  gespeichert, die maximal  $k$  Kanten lang sind.

- ▶ **Beweis** durch Induktion nach  $k$ . Für  $k = 0$  ist die Behauptung offensichtlich erfüllt.
- ▶ Sei  $v_0 (= s), \dots, v_k (= t)$  ein kürzester Weg von  $s$  nach  $t$  mit  $k > 0$  Kanten. Nach IV ist nach Schritt  $k - 1$  der Wert  $dist(v_{k-1})$  die Länge des kürzesten Weges von  $s$  nach  $v_{k-1}$ . Die Relaxierung der Kante  $v_{k-1} \rightarrow v_k$  in Schritt  $k$  ergibt:

$$dist(v_k) \leq dist(v_{k-1}) + weight(v_{k-1} \rightarrow v_k)$$

- ▶ Der Wert von  $dist(v_k)$  kann nicht echt kleiner sein als die rechte Seite, da der gegebene Pfad als kürzester Weg angenommen wurde. Somit gilt Gleichheit und die IB wurde für Schritt  $k$  gezeigt. □

## Laufzeit des Bellman-Ford Algorithmus

Der Bellman-Ford Algorithmus bestimmt die kürzesten Wege in einem Digraphen ohne negative Zyklen von einem Startknoten  $s$  in einer Laufzeit in  $\mathcal{O}(EV)$ .

### Beweis:

- ▶ Alle  $E$  Kanten werden  $V - 1$  mal relaxiert. Die Relaxierung hat konstante Laufzeit.  
□

## Erkennung von negativen Zyklen

In einem gewichteten Digraphen können negative Zyklen in einer Laufzeit in  $O(EV)$  erkannt werden.

### Beweis:

- Man wendet den Bellman-Ford Algorithmus an. Wenn die Bedingung

$$dist(w) \leq dist(v) + weight(v \rightarrow w)$$

für mindestens eine Kante **nicht** erfüllt ist, muss ein negativer Zyklus vorliegen.

- Wenn der Digraph keine negativen Zyklen hätte, müsste nach dem Korrektheitssatz obige Bedingung für alle Pfeile  $v \rightarrow w$  erfüllt sein.
- Der Zyklus kann dann ausgehend von  $w$  über die in *parent* gespeicherten Kanten zurückverfolgt werden.

## Effiziente Version des Bellman-Ford Algorithmus

- ▶ Der Algorithmus hat eine Laufzeit in  $O(EV)$ . Im *worst case* gibt es keine schnellere Variante.
- ▶ Mit einer relativ leichten Änderung ergibt sich aber in der Praxis laut [Sedgewick & Wayne] meist eine wesentlich kürzere Laufzeit.
- ▶ Eine Kante  $v \rightarrow w$  muss nur relaxiert werden, wenn  $dist(v)$  im letzten Schritt aktualisiert wurde.
- ▶ Speichere Knoten  $w$  in eine Warteschlange, deren Distanzwert durch eine Relaxierung vermindert wurde.
- ▶ Allerdings sollten keine Knoten mehrfach in die Schlange kommen.
  - Um dies auszuschließen, verwenden wir ein boole'sches Array  $onQ$ , das anzeigt, welche Knoten sich in der Warteschlange befinden.

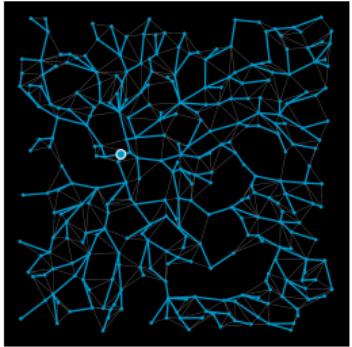
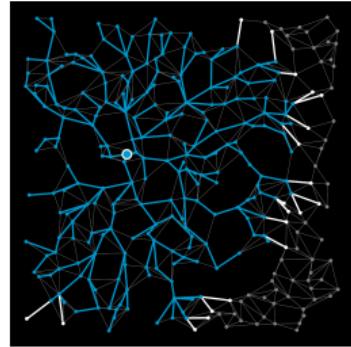
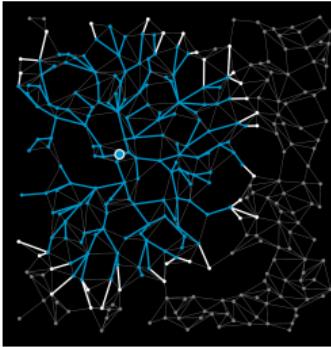
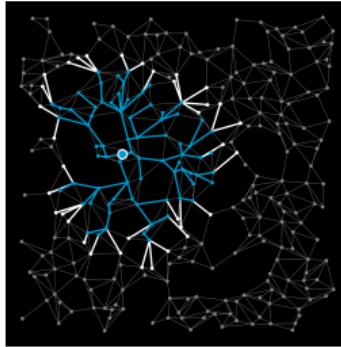
# Pseudocode für Bellman-Ford mit Warteschlange

```
1  $Q : \text{Queue}$ 
2  $onQ : \text{boolean array of size } V$ 
3 // initialisiere 'dist' wie zuvor
4 enqueue( $Q, s$ )
5  $onQ(s) \leftarrow \text{true}$ 
6 while  $Q \neq \emptyset$ 
7    $v = \text{dequeue}(Q)$ 
8    $onQ(v) \leftarrow \text{false}$ 
9   for each node  $w$  with  $v \rightarrow w \in E$ 
10    relaxBF( $v \rightarrow w$ )
11  end
12 end
```

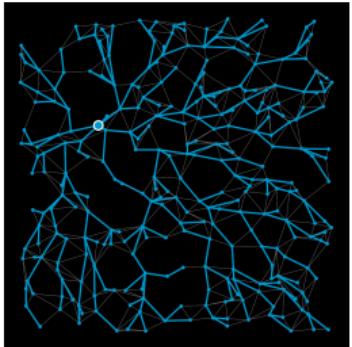
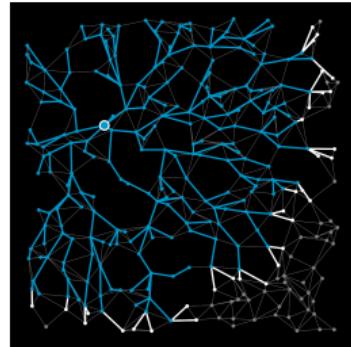
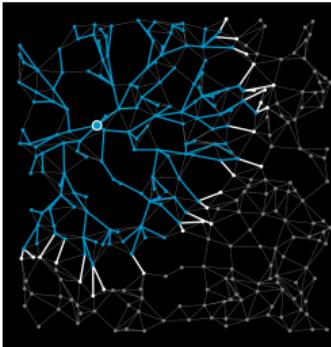
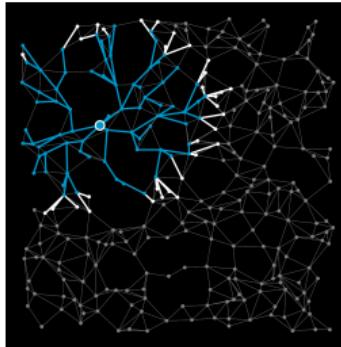
```
13 procedure relaxBF( $v \rightarrow w$ )
14   if  $dist(w) > dist(v) + \text{weight}(v \rightarrow w)$ 
15      $dist(w) = dist(v) + \text{weight}(v \rightarrow w)$ 
16      $parent(w) = v;$ 
17     if  $\text{!}onQ(w)$ 
18       enqueue( $Q, w$ )
19        $onQ(w) \leftarrow \text{true}$ 
20     end
21   end
```

# Reihenfolge der Knotenauswahl bei Dijkstra und Bellman-Ford

**Dijkstra Algorithmus** (Kanten in der Warteschlange sind weiß dargestellt)

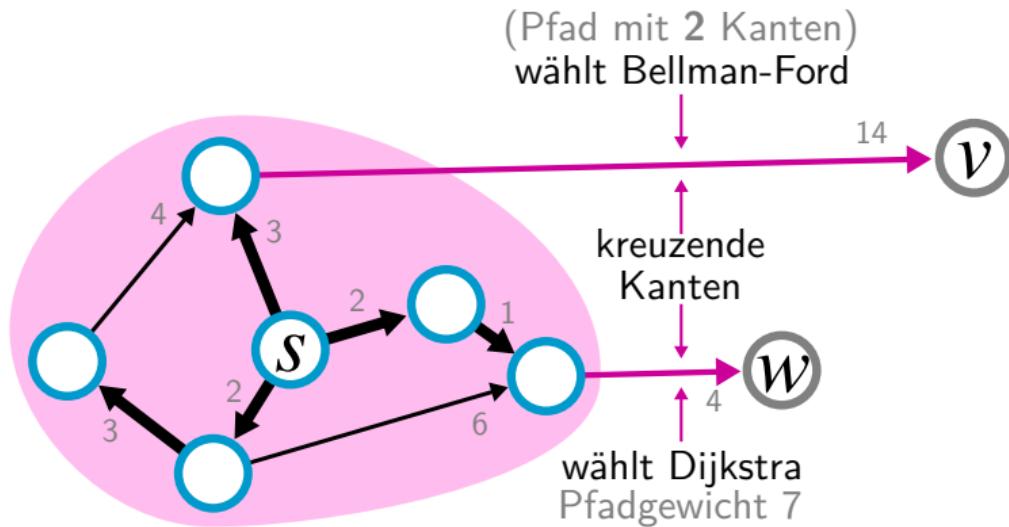


**Bellman-Ford Algorithmus** (Kanten in der Warteschlange sind weiß dargestellt)



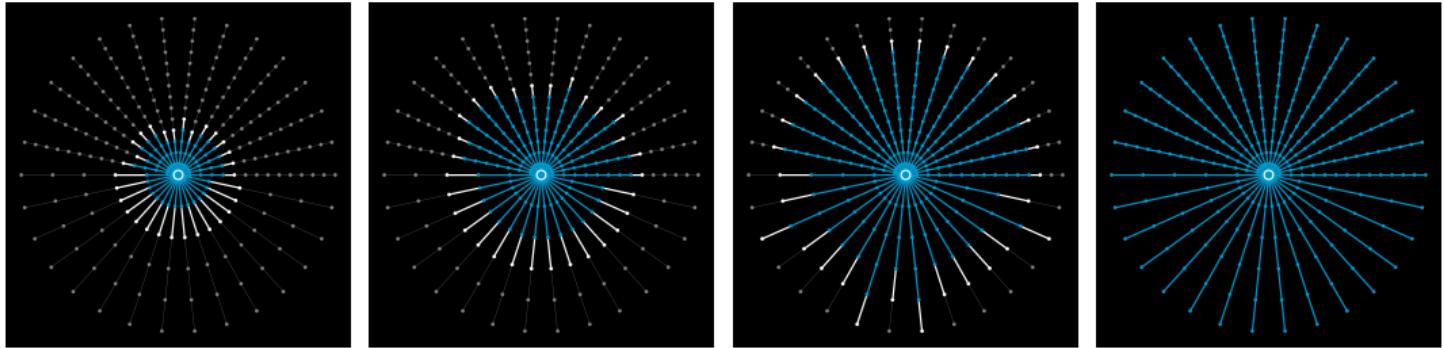
# Unterschied im Vorgehen bei Dijkstra und Bellman-Ford?

- ▶ Bei dem **Dijkstra** Algorithmus werden die Knoten in der Reihenfolge ihrer Gewichtsdistanz (= Länge des kürzesten Weges definiert durch Kantengewichte) von  $s$  bearbeitet.
- ▶ Bei dem **Bellman-Ford** Algorithmus werden die Knoten in der Reihenfolge ihrer Kantendistanz (= Anzahl der Kanten des Weges mit den wenigsten Kanten) von  $s$  bearbeitet.

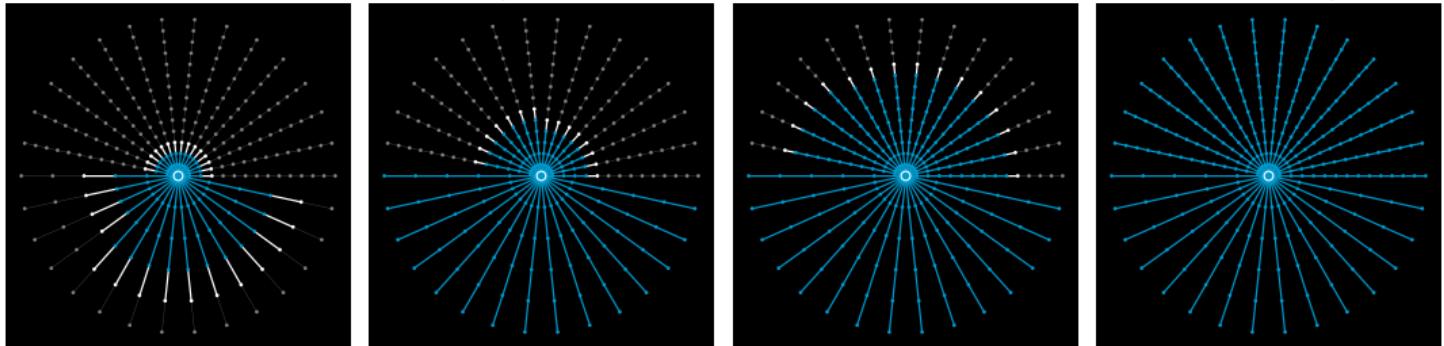


# Graph, der den Unterschied besser sichtbar macht

**Dijkstra Algorithmus** (Kanten in der Warteschlange sind weiß dargestellt)



**Bellman-Ford Algorithmus** (Kanten in der Warteschlange sind weiß dargestellt)



# Unterschied im Vorgehen bei Dijkstra und Bellman-Ford?

## Dijkstra Algorithmus

- ▶ Kantengewichte  $\geq 0$
- ▶ wählt Kanten nach Gewichtsdistanz
- ▶ Auswahl ist geschickt (greedy):
- ▶ besucht Knoten nur einmal
- ▶ Relax hat Laufzeit  $O(\log V)$
- ▶ Laufzeit  $O(E \log V)$
- ▶ Funktioniert auch bei negativen Kantengewichten, ist dann aber im *worst-case* deutlich ineffizienter als Bellman-Ford

## Bellman-Ford Algorithmus

- ▶ negative Kantengewichte erlaubt
- ▶ wählt Kanten nach Kantendistanz
- ▶ Auswahl ist einfach:
- ▶ muss Knoten ggf. mehrfach besuchen
- ▶ Relax hat Laufzeit  $O(1)$
- ▶ Laufzeit  $O(EV)$  (*worst case*)

## Executive Summary

- ▶ Je nach Art des Digraphen sind unterschiedliche SSSP Algorithmen geeignet.
- ▶ Für Dijkstra sollte eine **indizierte** Vorrangwarteschlange oder ein Fibonacci Heap benutzt werden.
- ▶ Bellman-Ford sollte mit einer Warteschlange implementiert werden.
- ▶ Für Digraphen mit Euklidschen Distanzen gibt es noch effizientere Algorithmen.

Algorithmen zum Finden kürzester Wege in Digraphen			
Algorithmus	Einschränkung	worst-case	Praxis (?)
Topologisches Sortieren	DAG	$O(V + E)$	
Dijkstra Algorithmus	Gewichte $\geq 0$	$O(E \log V)$	
mit Fibonacci-Heap	Gewichte $\geq 0$	$O(E + V \log V)$	
Bellman-Ford	(keine neg. Zyklen)	$O(VE)$	$O(V + E)$

## Generell:

- ▶ Ottmann T & Widmayer P. *Algorithmen und Datenstrukturen*. Springer Verlag, 5. Auflage; 2011. ISBN: 978-3827428042
- ▶ Segdewick R & Wayne K, *Algorithmen: Algorithmen und Datenstrukturen*, Pearson Studium, 4. Auflage, 2014. ISBN: 978-3868941845; in Teilen auch auf <http://www.cs.princeton.edu/IntroAlgsDS>
- ▶ TH Cormen, CE Leiserson, R Rivest, C Stein, *Algorithmen - Eine Einführung* . De Gruyter Oldenbourg, 4. Auflage; 2013. ISBN: 978-3486748611

## Originalveröffentlichungen:

- ▶ Avidan S, Shamir A. *Seam carving for content-aware image resizing*. In: ACM Transactions on graphics (TOG) 2007 Aug 5 (Vol. 26, No. 3, p. 10).

# Index

- Content-aware image resizing*, 18
- API**
- gerichtete gewichtete Kante*, 4
  - kantengewichteter Digraph*, 4
- Baum der kürzesten Wege*, 7
- Bellman-Ford*
- Korrektheitsbeweis*, 34
  - Laufzeit*, 35
- Bellman-Ford Algorithmus*, 31, 33
- Dijkstra*
- negative Kantengewichte*, 30
- Dijkstra Algorithmus*, 21
- Korrektheitsbeweis*, 24
  - Laufzeit*, 28
- DirectedEdge*, 4
- EdgeWeightedDigraph*, 4
- Kantengewichte*, 3
- Kürzeste Wege*
- Dijkstra*, 22
  - in gewichteten DAG*, 15
  - mit topologischer Sortierung*,
- 15*
- Kürzeste Wege in DAG*
- Korrektheitsbeweis*, 16
- Kürzeste Wege*
- Bellman-Ford*, 33
- negative Zyklen*
- Erkennung*, 36
- negativer Zyklus*, 6
- Relaxation*, 10
- Shortest Paths Tree*, 7
- SPT*, 6, 7