Fachhochschule Nordwestschweiz

07 Graphen

Algorithmen und Datenstrukturen 2

1. **Teil**
    - Graphrepräsentation
    - Topologisches Sortieren

2. **Teil**
    - Graphtraversierungen (DFS & BFS)
    - Anwendungen von DFS & BFS (Zusammenhangskomponenten & Spannbaum)

3. **Teil**
    - Kürzeste Wege

Grundprinzip - “nimm einen (beliebigen) Knoten v im Rand, folge einer unbenutzten Kante <v,w>”

1. Füge Startknoten **s** zum Rand **R** und setze **s.visited = true**
2. Solange **R** nicht leer ist,
   betrachte einen (beliebigen) Knoten **v** in **R**
   a. Falls aus **v** keine unbenutzten Kanten führt, lösche **v** aus **R**.
   b. Sonst, folge einer noch unbenutzten Kante **<v,w>**.
      i. Falls !**w.visited**, füge **w** zu **R** hinzu und setze **w.visited=true**

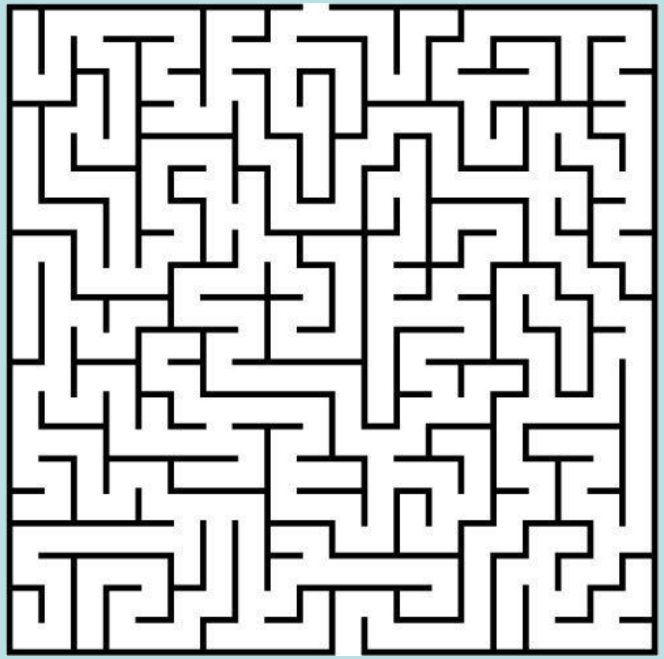
Tiefensuche

- ***Stack*** für das Speichern der Randknoten
- Setzt Traversierung im ***zuletzt*** gefundenen Knoten fort

Breitensuche

- ***Queue*** für das Speichern der Randknoten
- Setzt Traversierung im ***zuerst*** gefundenen Knoten fort

Kürzeste Wege



Breitensuche

```
void BFS(Vertex s) {
     Queue<Vertex<K>> R = new LinkedList<Vertex<K>>();
     print(v); s.visited = true;
     R.add(s);

     while(!R.isempty()) {
       Vertex v = R.remove();
          for(Vertex w : v.adjList) { // benutze alle Kanten
          if(!w.visited) {
             print(w); w.visited = true;
             R.add(w);
}  }  }  }
```

Definitionen

Distanz zweier Knoten:

D(x, y) = Länge eines kürzesten Weges von *x* nach *y*, falls einer existiert; ∞ sonst

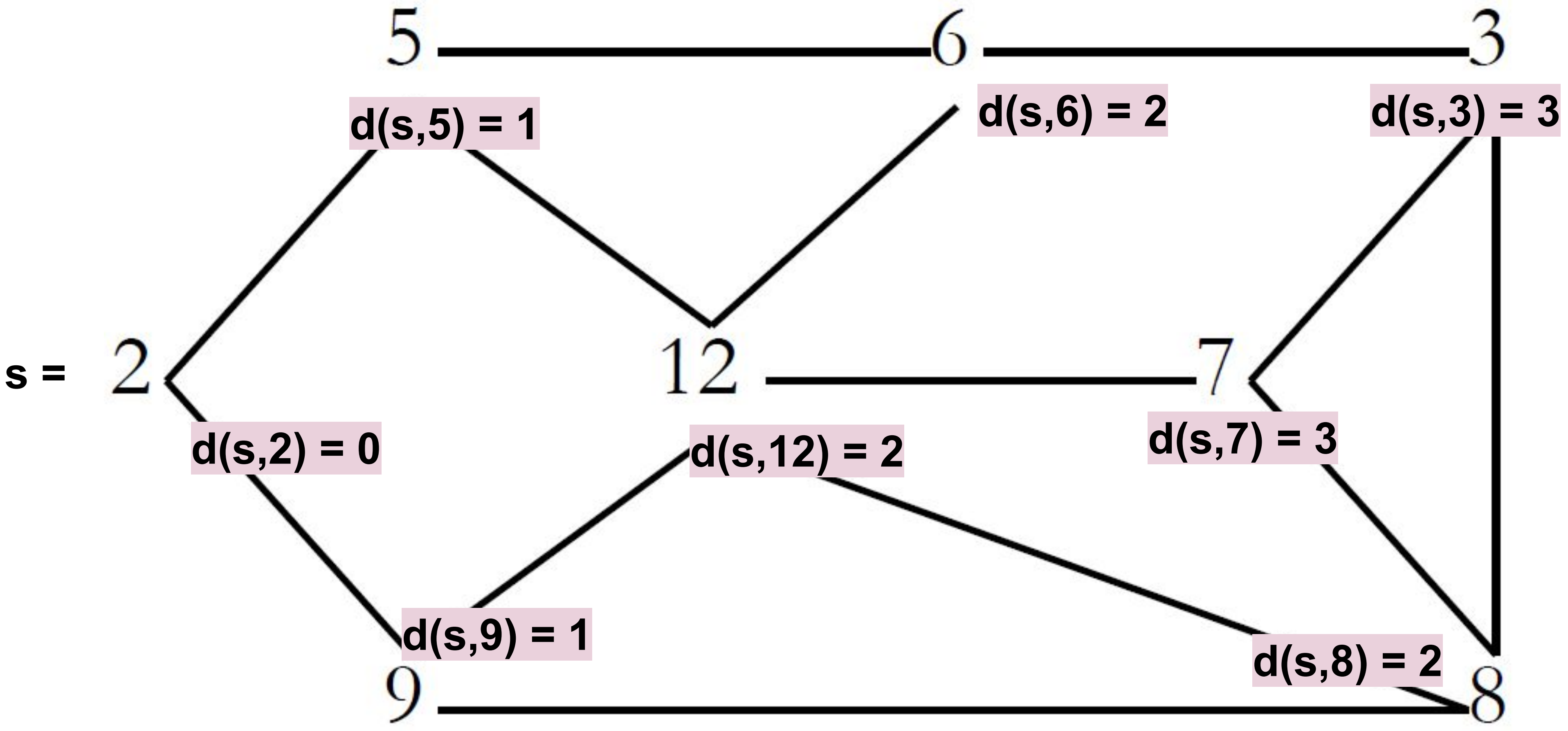
Länge eines Weges:

1. In ungewichteten Graphen: Anzahl Kanten
2. In gewichteten Graphen: Summe aller Kantengewichte

SSSP (Single Source Shortest Path)

gesucht sind die Kürzesten Wege von einem Knoten s zu allen anderen Knoten

Kürzeste Wege in ungewichteten Graphen

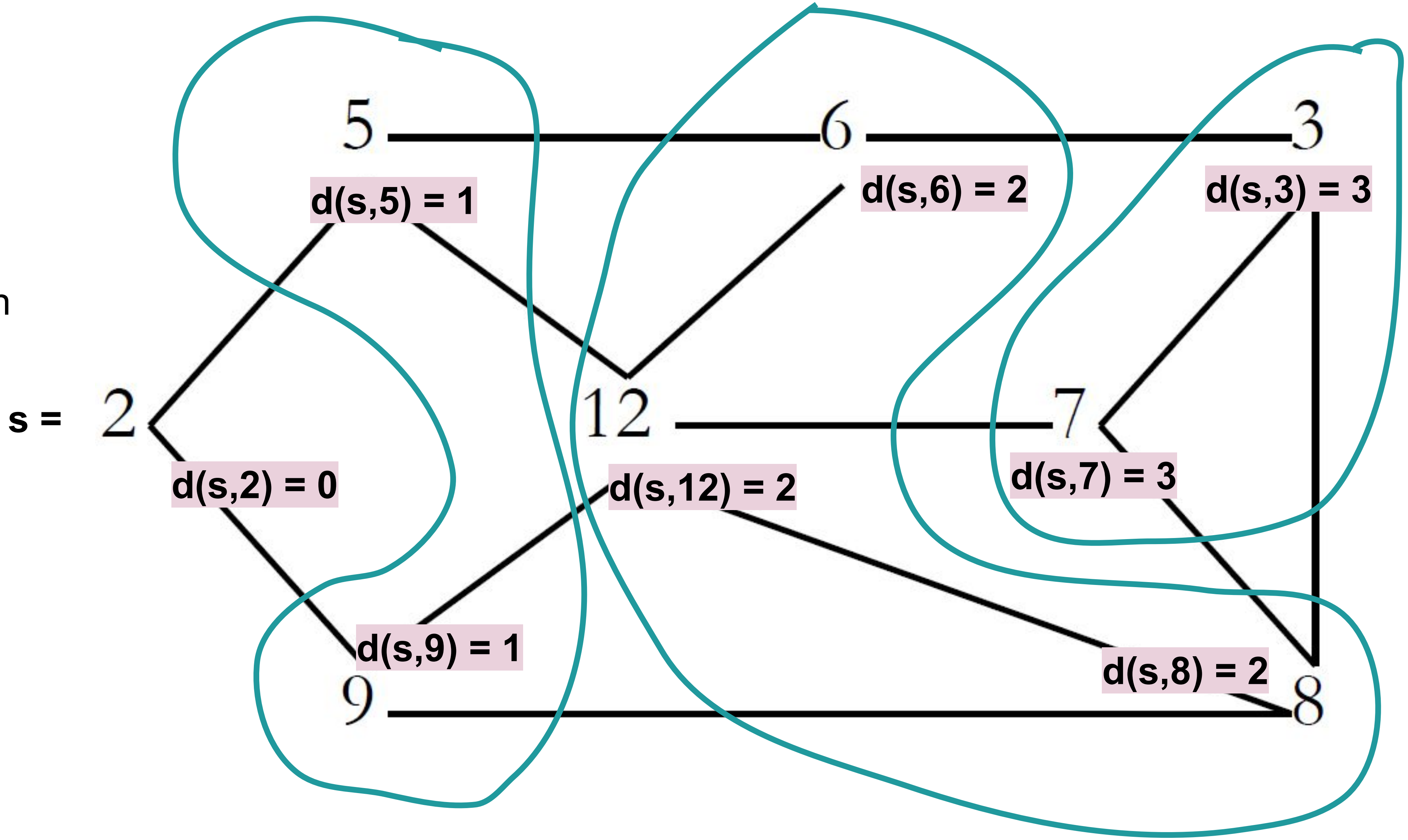


Kürzeste Wege in ungewichteten Graphen

BFS geht schichtweise vor:

Ausgehend vom Startpunkt werden zuerst alle Knoten mit Distanz 1 markiert, dann jene mit Distanz 2 usw.

Die Knoten werden "schichtweise" zum Rand R hinzugefügt und bearbeitet.



Kürzeste Wege in ungewichteten Graphen

Algorithmus im Pseudo-Code:

1. Bei allen Knoten die Distanzangabe auf ∞ setzen
2. Beim Startknoten die Distanzangabe auf 0 setzen
3. Startknoten hinten an die Queue R anfügen
4. Solange Queue R nicht leer ist:
    1. Vordersten Knoten *v* aus der Queue R entnehmen
    2. Für alle Knoten *w* zu denen eine Kante von *v* führt:

        Falls die Distanzangabe dieses Knotens ∞ ist:

            Distanzangabe von *w* = 1 + Distanzangabe von *v*

            Knoten *w* hinten an Queue R anfügen

Kürzeste Wege in ungewichteten Graphen

Algorithmus im Pseudo-Code:

1. Bei allen Knoten die Distanzangabe auf ∞ setzen
2. Beim Startknoten die Distanzangabe auf 0 setzen
3. Startknoten hinten an die Queue R anfügen
4. Solange Queue R nicht leer ist:
    1. Vordersten Knoten *v* aus der Queue R entnehmen
    2. Für alle Knoten *w* zu denen eine Kante von *v* führt:

        Falls die Distanzangabe dieses Knotens ∞ ist:

          Distanzangabe von *w* = 1 + Distanzangabe von *v*

          Knoten *w* hinten an Queue R anfügen

Vergleich zu BFS letzte Woche:

∞ entspricht genau dem boolean visited == false

Kürzeste Wege in ungewichteten Graphen

Algorithmus im Pseudo-Code:

1. Bei allen Knoten die Distanzangabe auf ∞ setzen
2. Beim Startknoten die Distanzangabe auf 0 setzen
3. Startknoten hinten an die Queue R anfügen
4. Solange Queue R nicht leer ist:
   1. Vordersten Knoten *v* aus der Queue R entnehmen
   2. Für alle Knoten *w* zu denen eine Kante von *v* führt:

      Falls die Distanzangabe dieses Knotens ∞ ist:

        Distanzangabe von *w* = 1 + Distanzangabe von *v*

        Knoten *w* hinten an Queue R anfügen

Vergleich zu BFS letzte Woche:

∞ entspricht genau dem boolean visited == false

Implementierungs-Möglichkeiten

- Tabelle mit Distanzen
- Attribut in Klasse Vertex

Kürzeste Wege in ungewichteten Graphen

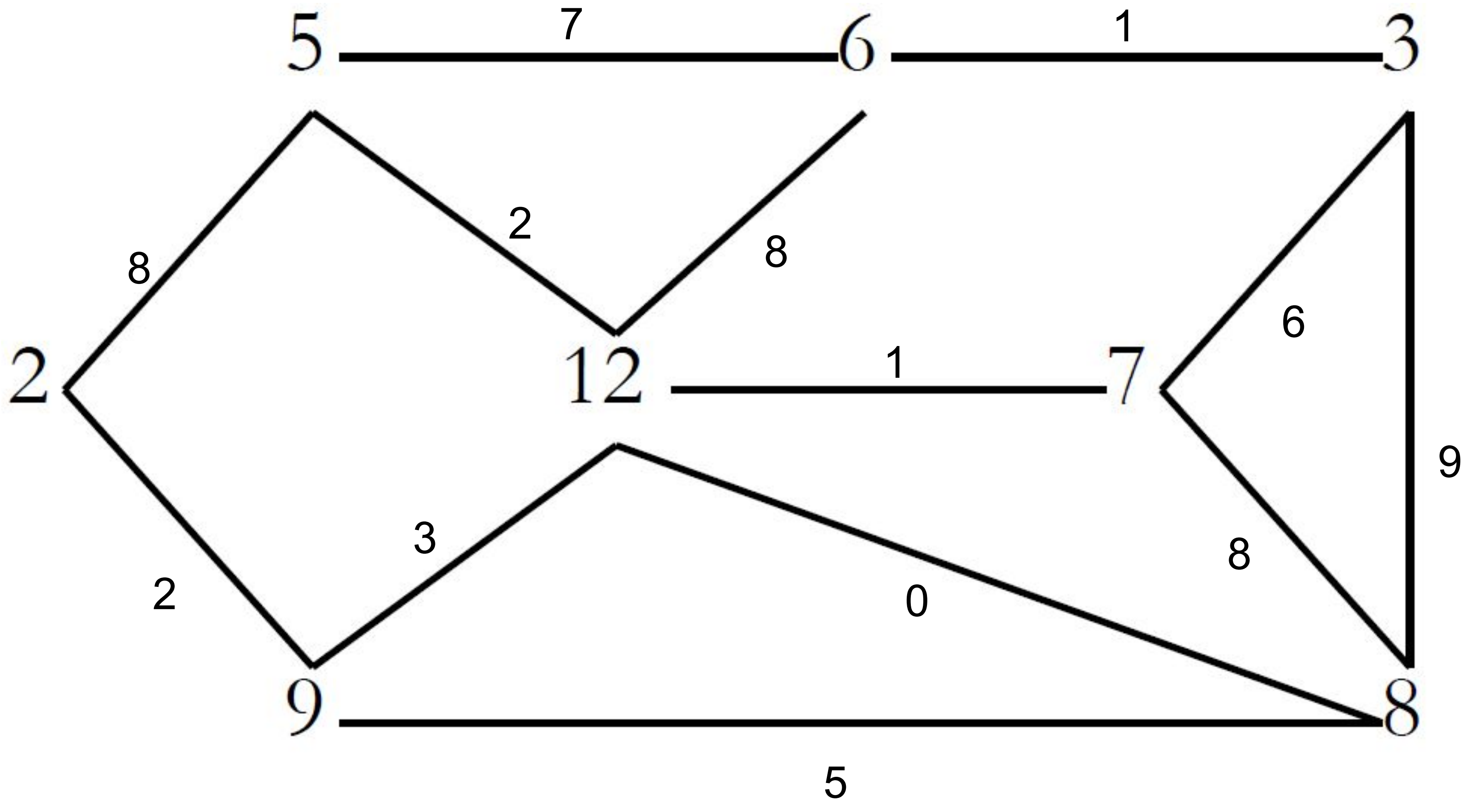
```
void BFS(Vertex s) {
     Queue<Vertex<K>> R = new LinkedList<Vertex<K>>();
     print(v); s.visited = true; s.dist = 0;
     R.add(s);

     while(!R.isempty()) {
       Vertex v = R.remove();
          for(Vertex w : v.adjList) {
              if (!w.visited) { (w.dist == Integer.MAX_VALUE) {
                 print(w); w.visited = true; w.dist = v.dist + 1;
                 R.add(w);
}  }  }  }
```

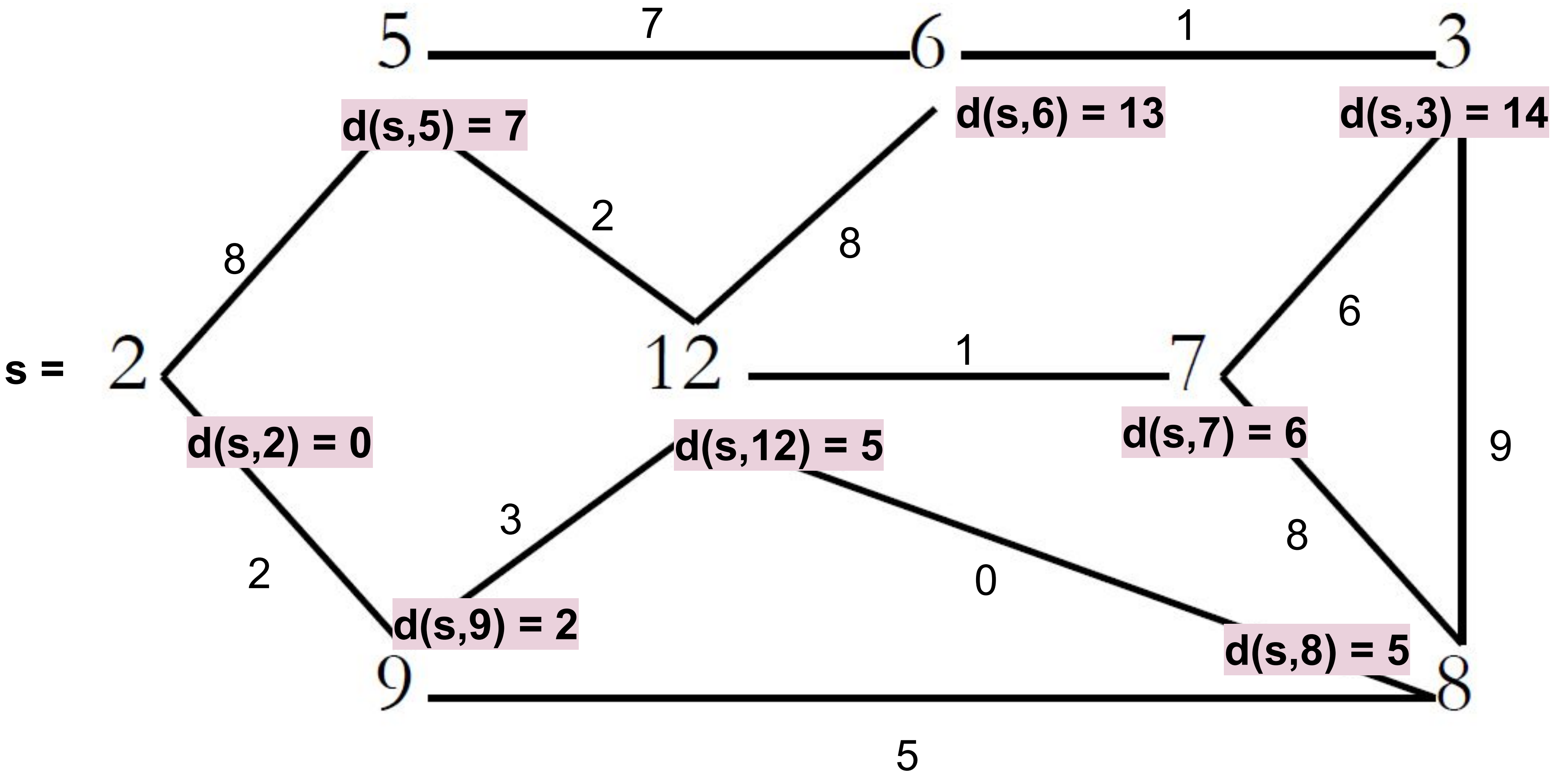
Fachhochschule Nordwestschweiz

Kürzeste Wege in gewichteten Graphen

s =



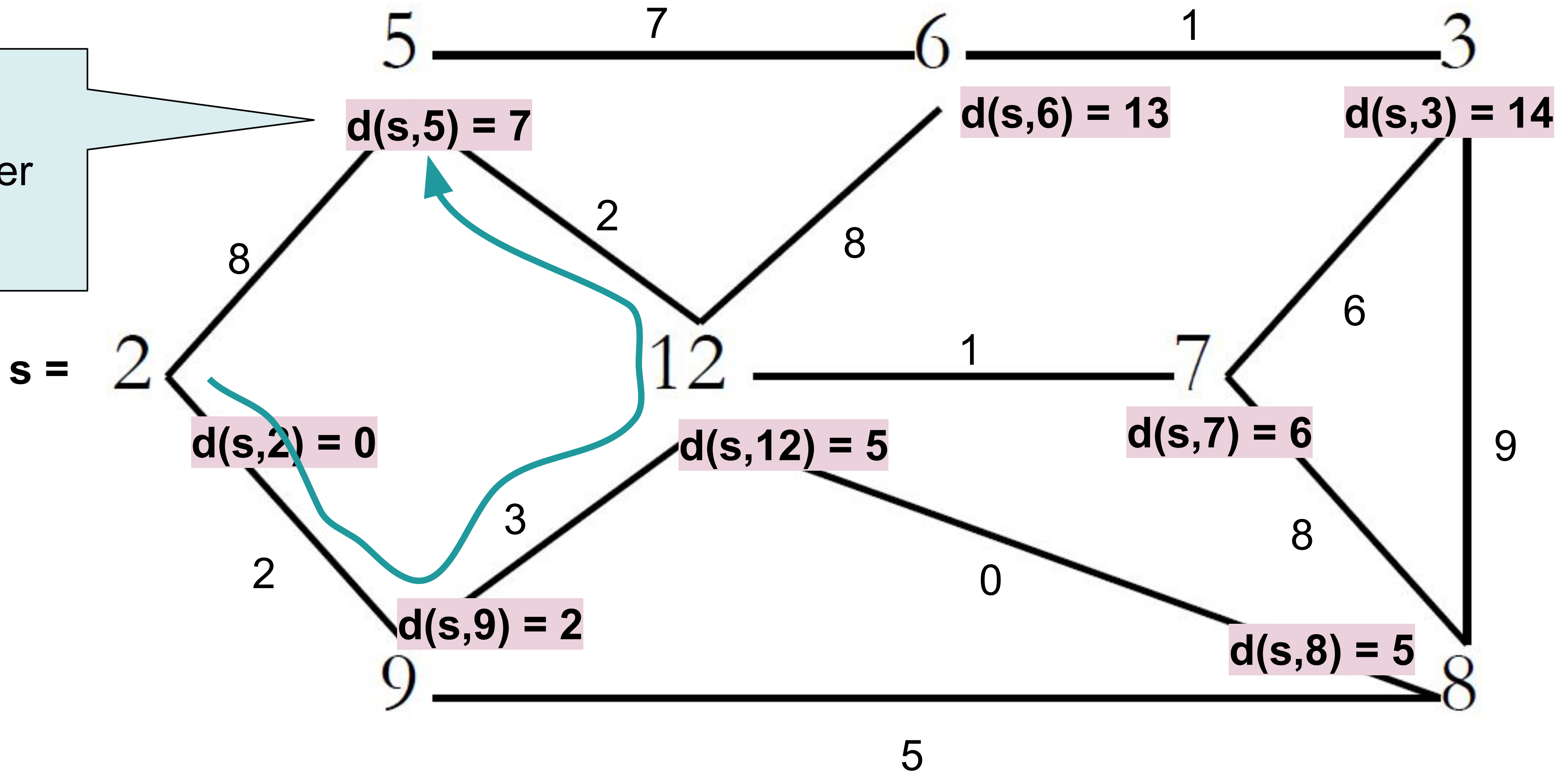
Kürzeste Wege in gewichteten Graphen



Fachhochschule
Nordwestschweiz

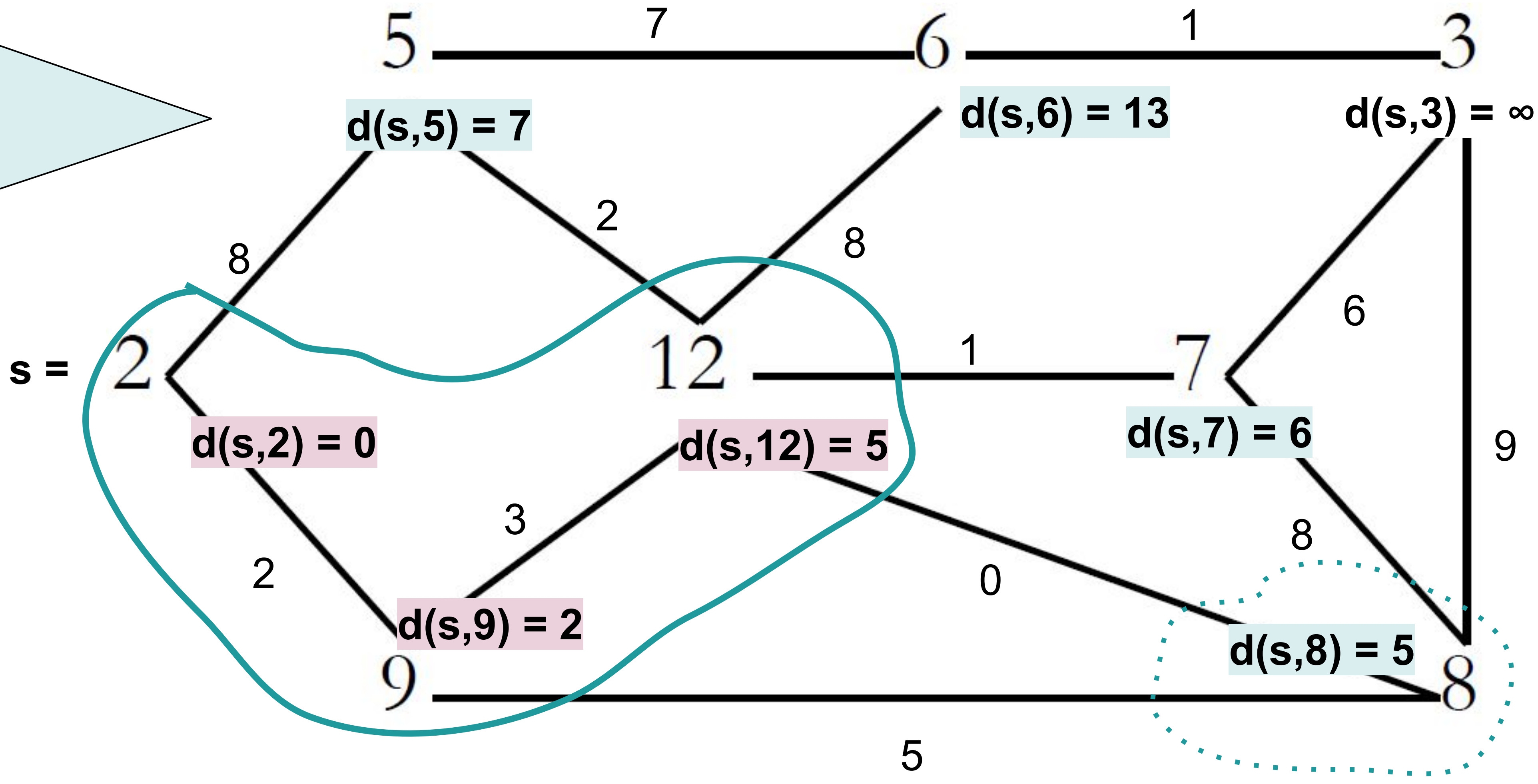
Kürzeste Wege in gewichteten Graphen

Nicht der Weg mit den wenigsten Kanten ist der kürzeste von 2 zu 5.



Kürzeste Wege in gewichteten Graphen

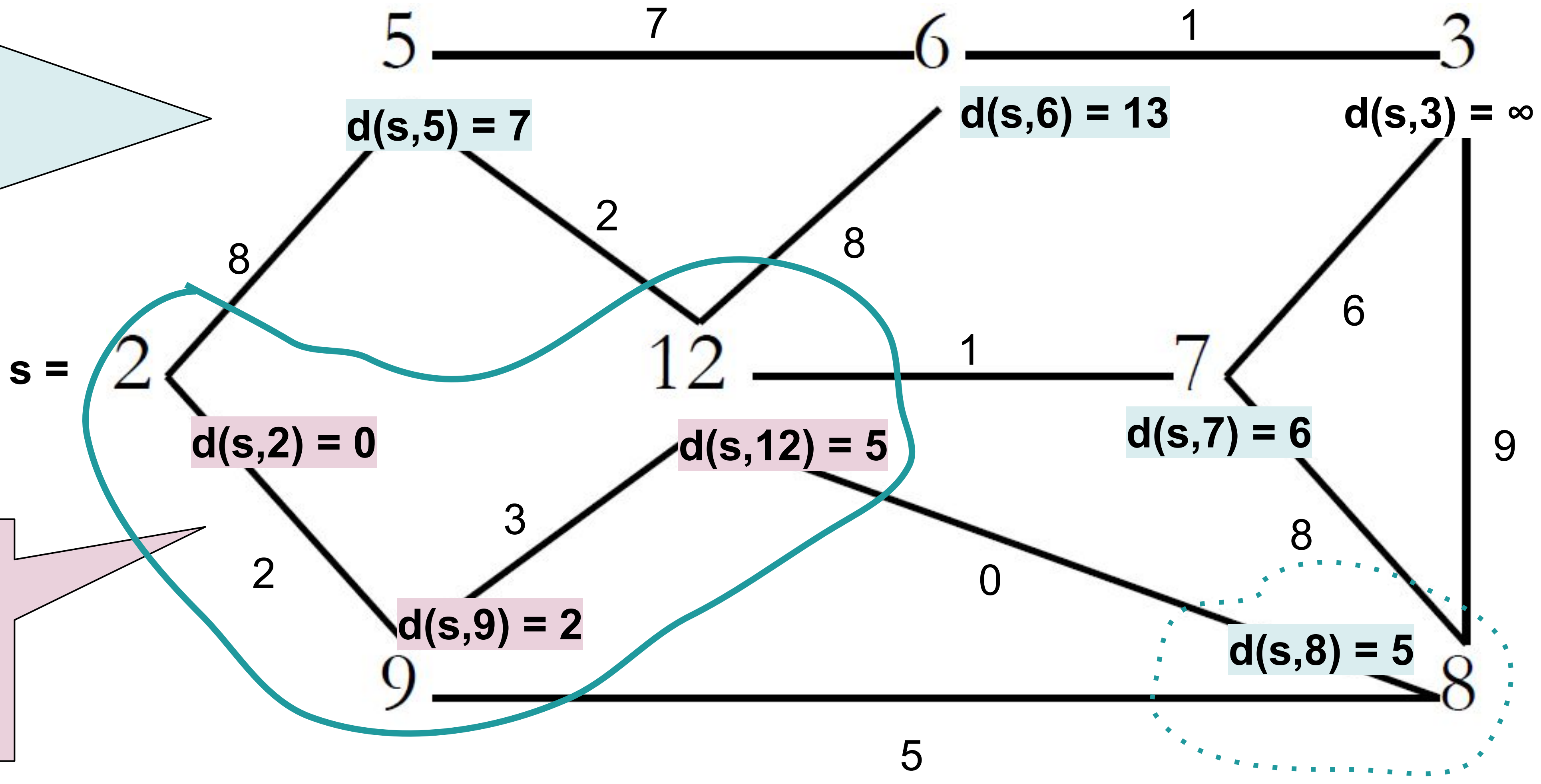
Idee: Nicht der BFS-Ordnung folgen, sondern die Menge der “Fertigen Knoten” (rot) erweitern mit dem Rand-Knoten (gepunktet), der momentan die *kürzeste Distanz* zum Startknoten hat. Danach die Distanzen seiner Nachbarn updaten.



Kürzeste Wege in gewichteten Graphen

Idee: Nicht der BFS-Ordnung folgen, sondern die Menge der “Fertigen Knoten” (rot) erweitern mit dem Rand-Knoten (gepunktet), der momentan die *kürzeste Distanz* zum Startknoten hat. Danach die Distanzen seiner Nachbarn updaten.

Achtung! Funktioniert nur, wenn alle Kantengewichte nicht negativ sind.



Kürzeste Wege in gewichteten Graphen

Genau diese Idee verfolgt der Algorithmus von ***Dijkstra***:

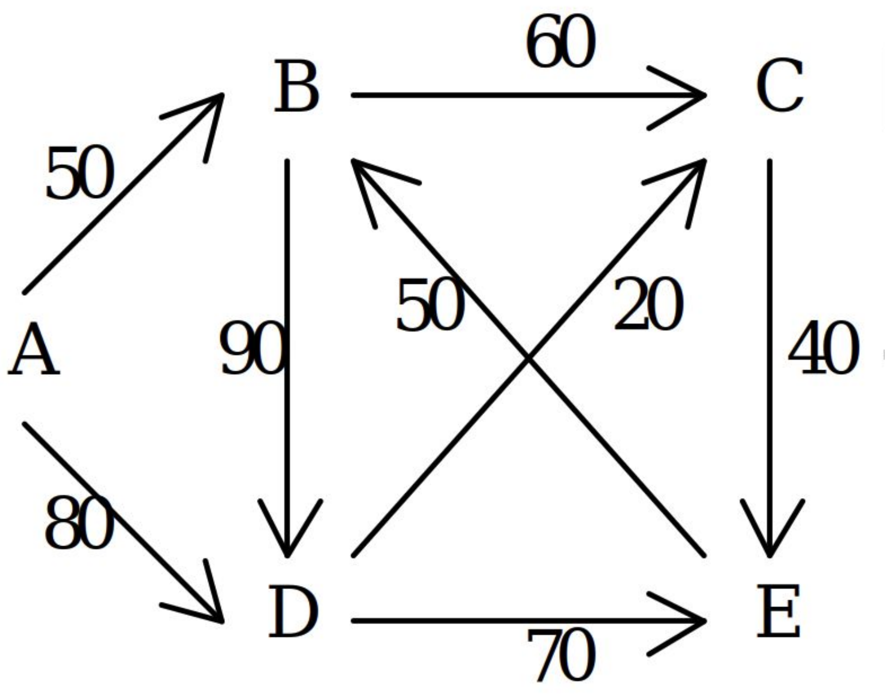
1. Tabelle für alle Knoten:
   fertig: (init: *false*) // *kürzester Weg von s definitiv bekannt? (rote Knoten)*
   dist: (init: *0* für s, ∞ sonst) // *Länge des bis jetzt bekannten kürzesten Wegs von Startknoten s*
   via: (init: ? oder *null*) // *vorhergehender Knoten auf dem bisher kürzesten Weg von s*

2. ***While*** es in der Tabelle Knoten mit (*fertig* = *false* und *dist* < ∞) gibt:
   a. Knoten *v* suchen der unter allen Knoten mit *fertig* = *false* den kleinsten Wert für *dist* hat.
   b. *v.fertig* = *true* setzen // es kann keinen noch kürzeren Weg zu *v* geben
   c. Für jeden *Knoten w* zu dem es eine Kante <*v*, *w*> gibt:
      i. *int d* = *v.dist* + Kantengewicht von <*v*, *w*> // entspricht Distanz des kürzesten Wegs via v
      ii. if (*d* < *w.dist*) { *w.dist* = *d*; *w.via* = *v*; } // Update Tabelle, wenn neuer Weg kürzer ist

Kürzeste Wege in gewichteten Graphen

Initialisierung: Startknoten A

Knoten	fertig	Distanz	via
A	false	0	?
B	false	∞	null
C	false	∞	null
D	false	∞	null
E	false	∞	null



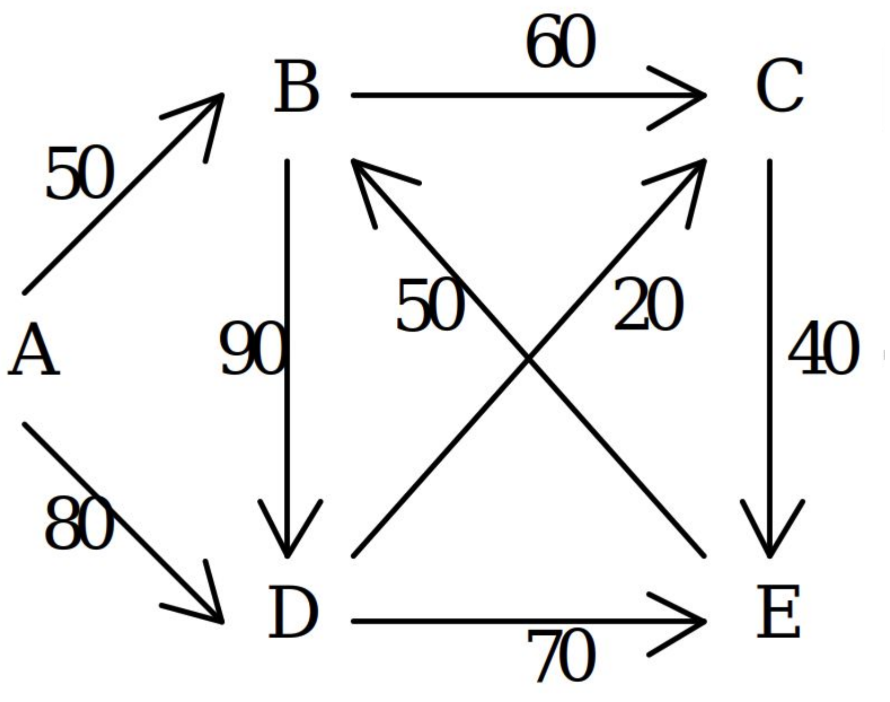
Kürzeste Wege in gewichteten Graphen

Initialisierung: Startknoten A

Knoten	fertig	Distanz	via
A	false	0	?
B	false	∞	null
C	false	∞	null
D	false	∞	null
E	false	∞	null

null: aufpassen bei Rückverfolgung
A: stoppen, wenn Knoten == via

∞ ist keine Zahl.
Alternative: max int

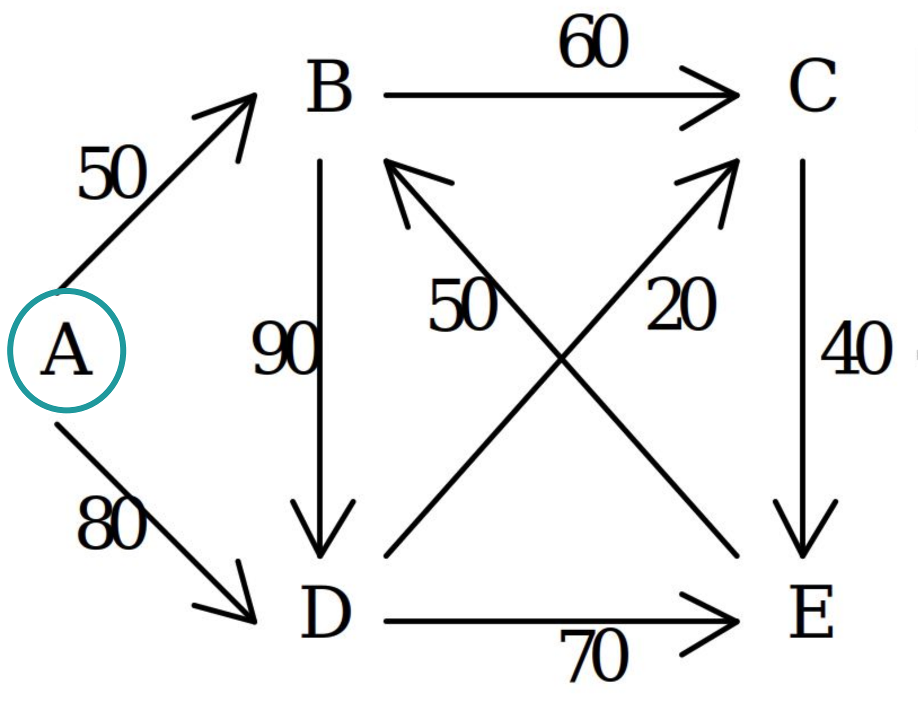


Kürzeste Wege in gewichteten Graphen

Erster Schritt:

Knoten	fertig	Distanz	via
A	false	0	?
B	false	∞	null
C	false	∞	null
D	false	∞	null
E	false	∞	null

einziger **!fertig**
mit Distanz < ∞



Kürzeste Wege in gewichteten Graphen

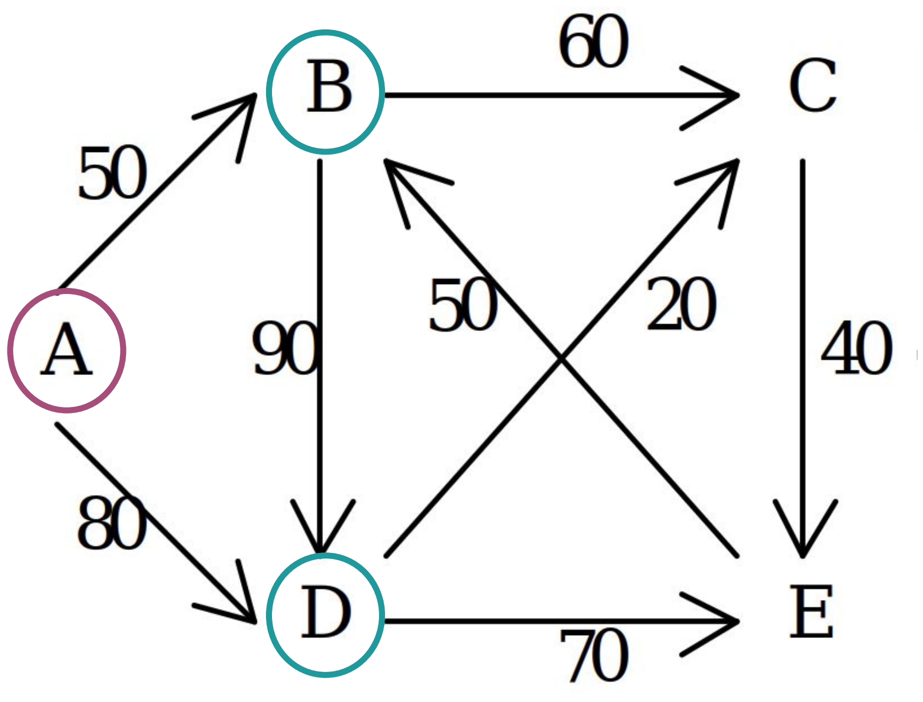
Erster Schritt:

Knoten	fertig	Distanz	via
A	**true**	0	?
B	false	**50**	**A**
C	false	∞	null
D	false	**80**	**A**
E	false	∞	null

einziger **!fertig**
mit Distanz < ∞

<A,B>
d = 50 + dist(A) = 50

<A,D>
d = 80 + dist(A) = 80



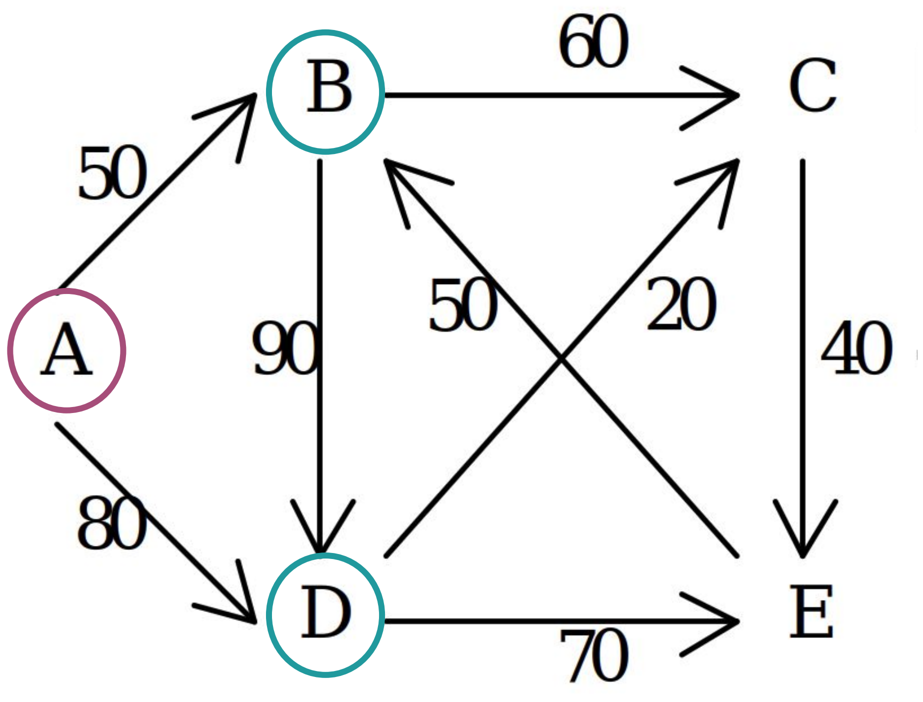
Fachhochschule
Nordwestschweiz

Kürzeste Wege in gewichteten Graphen

Zweiter Schritt:

Knoten	bekannt	Distanz	via
A	true	0	?
B	false	50	A
C	false	∞	null
D	false	80	A
E	false	∞	null

!fertig und kleinste bis jetzt bekannte Distanz



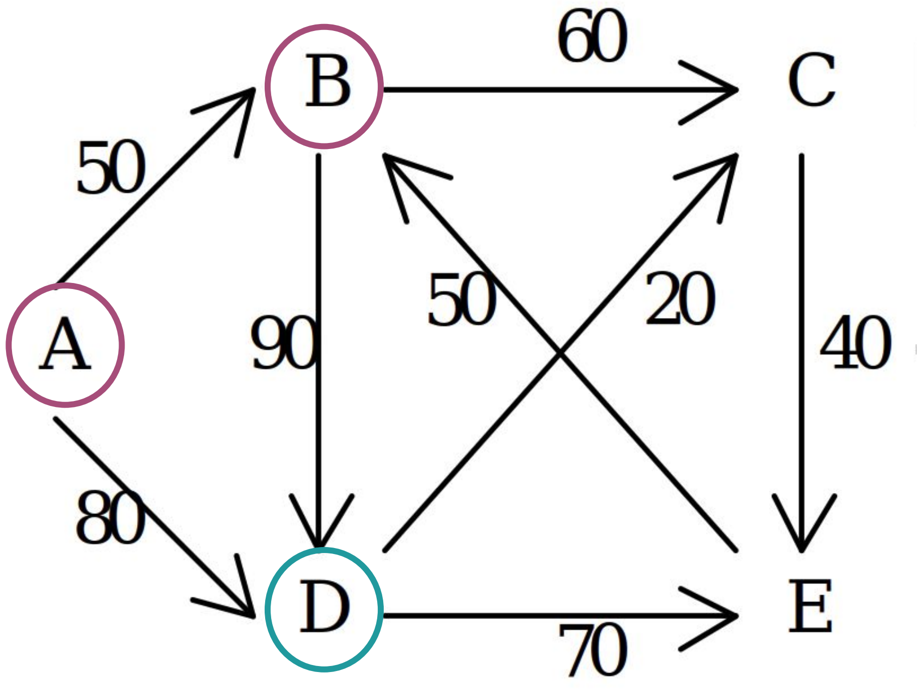
Kürzeste Wege in gewichteten Graphen

Zweiter Schritt:

Knoten	bekannt	Distanz	via
A	true	0	?
B	false	50	A
C	false	∞	null
D	false	80	A
E	false	∞	null

!fertig und kleinste bis jetzt bekannte Distanz

<B,C>
d = 60 + dist(B) = 110
d < dist(C) -> update
<B,D>
d = 90 + dist(B) = 140
d > dist(D) -> ok



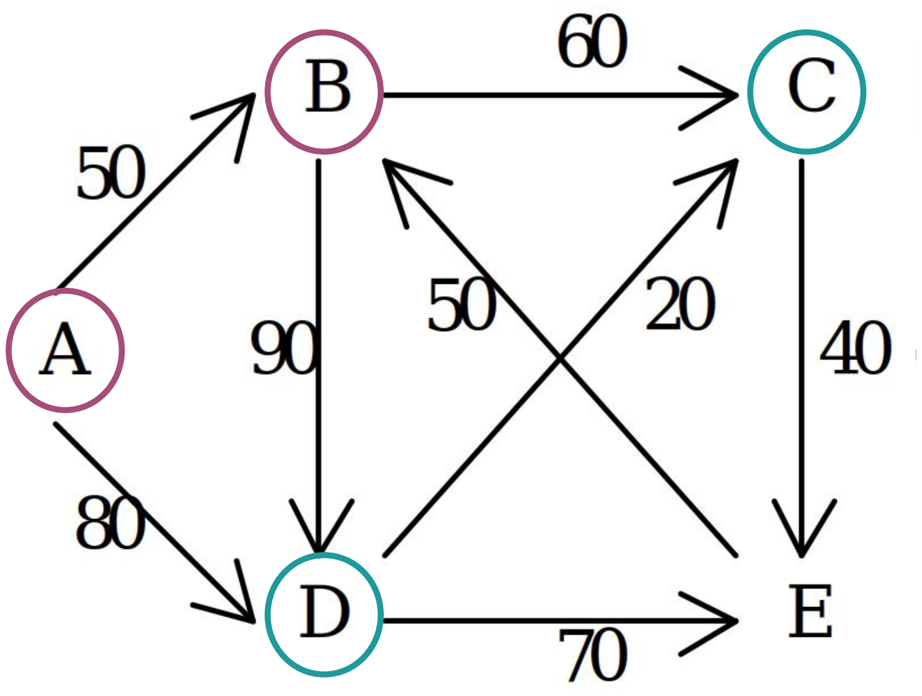
Kürzeste Wege in gewichteten Graphen

Zweiter Schritt:

Knoten	bekannt	Distanz	via
A	true	0	?
B	**true**	50	A
C	false	**110**	**B**
D	false	80	A
E	false	∞	null

!fertig und kleinste bis jetzt bekannte Distanz

<B,C>
d = 60 + dist(B) = 110
d < dist(C) -> update
<B,D>
d = 90 + dist(B) = 140
d > dist(D) -> ok



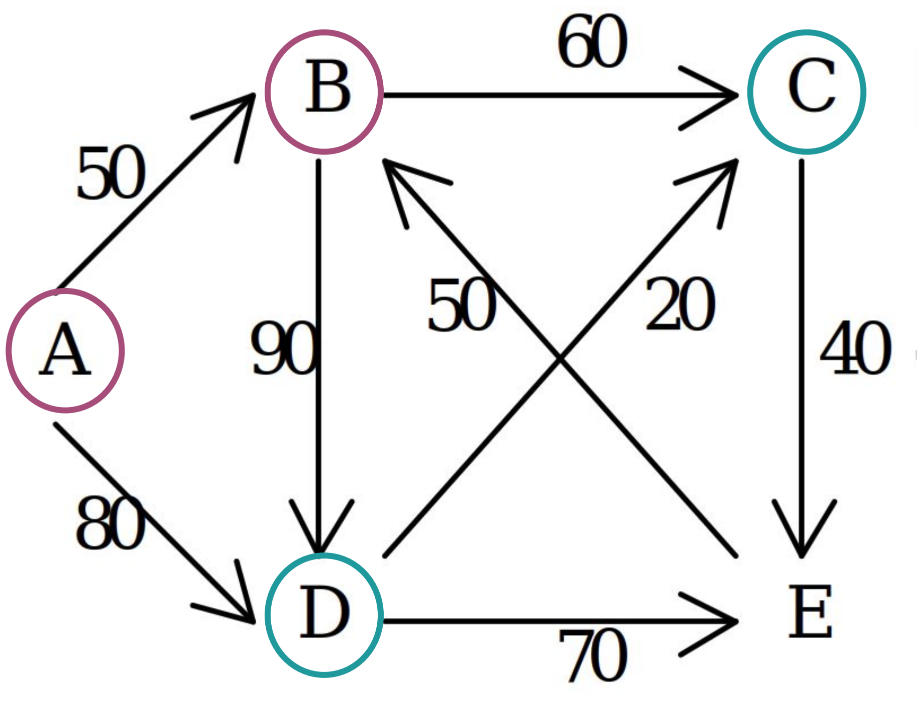
Kürzeste Wege in gewichteten Graphen

Zweiter Schritt:

Knoten	bekannt	Distanz	via
A	true	0	?
B	true	50	A
C	false	110	B
D	false	80	A
E	false	∞	null

!fertig und kleinste bis jetzt bekannte Distanz

<D,C>
d = 20 + dist(D)= 100
d < dist(C) -> update
<D,E>
d = 70 + dist(D)= 150
d < dist(E) -> update



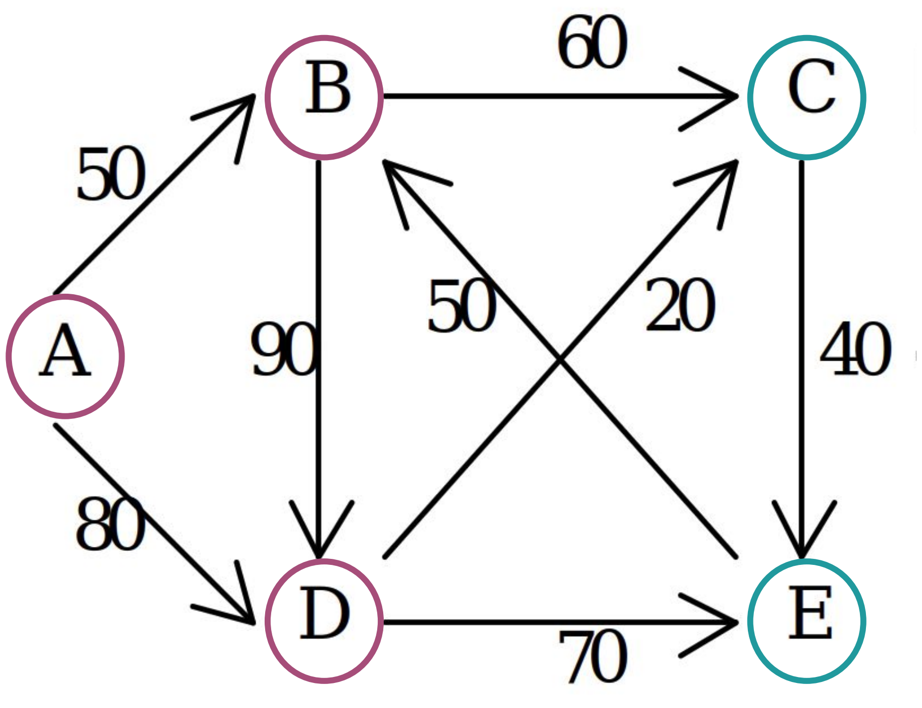
Kürzeste Wege in gewichteten Graphen

Dritter Schritt:

Knoten	bekannt	Distanz	via
A	true	0	?
B	true	50	A
C	false	**100**	**D**
D	**true**	80	A
E	false	**150**	**D**

!bekannt und kleinste bis jetzt bekannte Distanz

<D,C>
d = 20 + dist(D)= 100
d < dist(C) -> update
<D,E>
d = 70 + dist(D) =150
d < dist(E) -> update

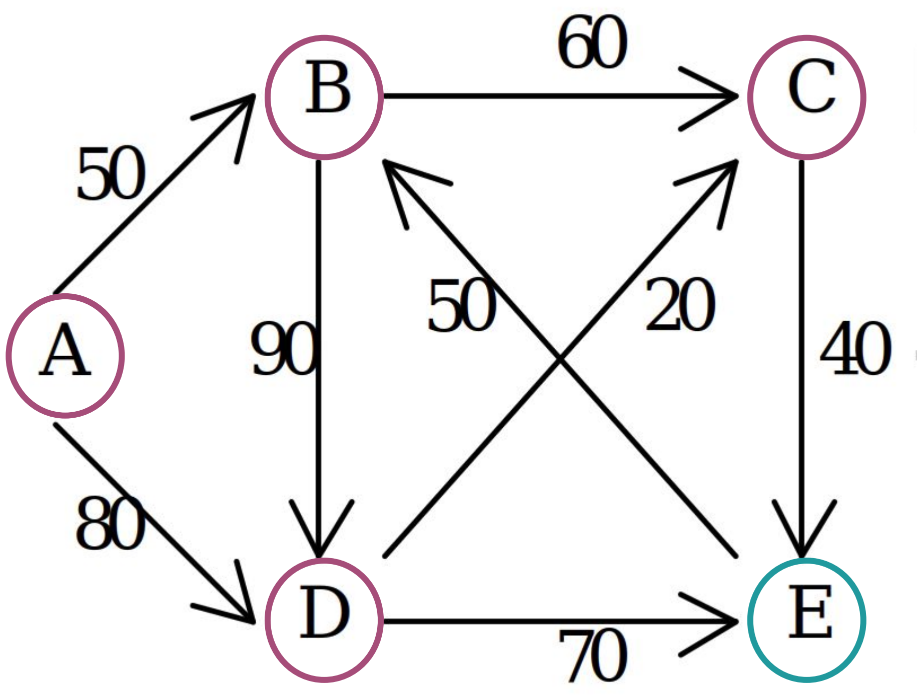


Kürzeste Wege in gewichteten Graphen

Vierter Schritt:

Knoten	bekannt	Distanz	via
A	true	0	?
B	true	50	A
C	**true**	100	D
D	true	80	A
E	false	**140**	**C**

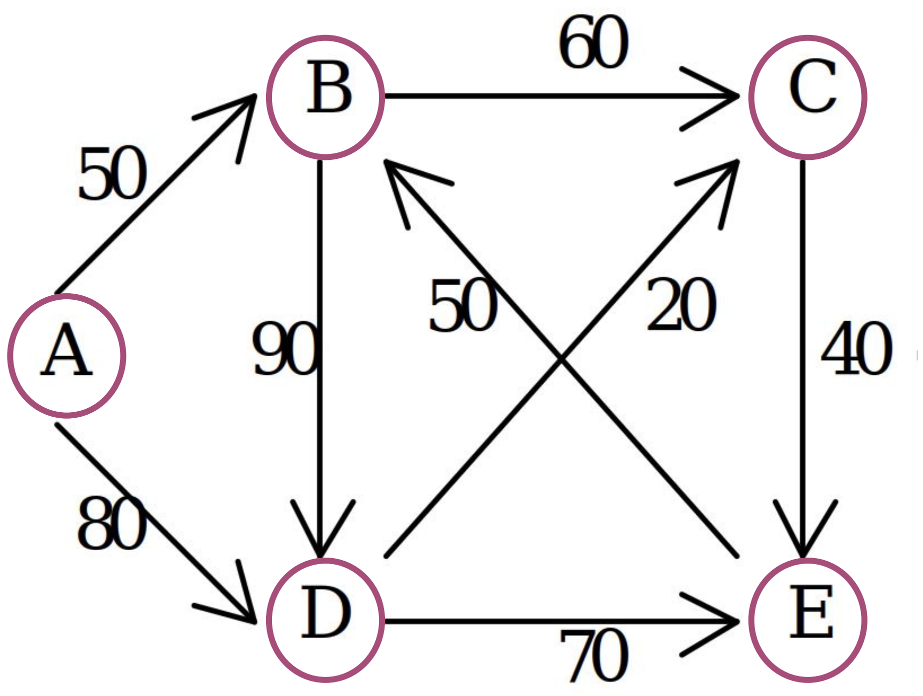
<C,E>
d = 40 + dist(C)= 140
d < dist(C) -> Update



Kürzeste Wege in gewichteten Graphen

Letzter Schritt:

Knoten	bekannt	Distanz	via
A	true	0	?
B	true	50	A
C	true	100	D
D	true	80	A
E	**true**	140	C



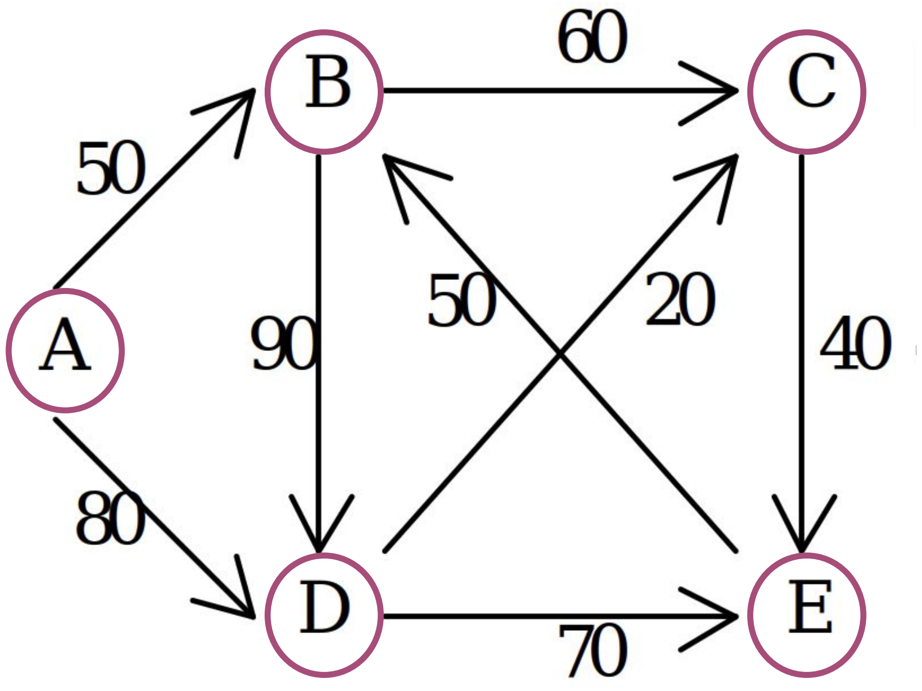
Fachhochschule Nordwestschweiz

Kürzeste Wege in gewichteten Graphen

Rückverfolgung: Der kürzeste Weg von A nach X lässt sich mit den “via” rekonstruieren.

Weitere Visualisierungen mit: [Dijkstra Visualzation (usfca.edu)]

Knoten	bekannt	Distanz	via
A	true	0	?
B	true	50	A
C	true	100	D
D	true	80	A
E	**true**	140	C



Aufwand 1x *While*:
a: O(n)
b: O(1)
c: O(v.outdegree)

Kürzeste Wege in gewichteten Graphen

Eine Möglichkeit ist der Algorithmus von *Dijkstra*:

1. Tabelle für alle Knoten:
   - *fertig:* (init: *false*)
   - *dist:* (init: *0* für s, ∞ sonst)
   - *via:* (init: ? oder *null*)
2. ***While*** es in der Tabelle Knoten mit (*fertig* = *false* und *dist* < ∞) gibt:
   a. Knoten *v* suchen der unter allen Knoten mit *fertig*= *false* den kleinsten Wert für *dist* hat.
   b. *v.fertig* = *true* setzen
   c. Für jeden *Knoten w* zu dem es eine Kante <*v*, *w*> gibt:
      i. *int d* = *v.dist* + Kantengewicht von <*v*, *w*>
      ii. if (*d* < *w.dist*) { *w.dist* = *d*; *w.via* = *v*; }

Fachhochschule Nordwestschweiz

Kürzeste Wege in gewichteten Graphen

Aufwand 1x *While*:	Aufwand nx *While*:
a: O(n)	a: $O(n^2)$
b: O(1)	b: O(n)
c: O(v.outdegree)	c: O(m) (mit Adj.listen)
	Total $O(n^2 + m)$

Eine Möglichkeit ist der Algorithmus von *Dijkstra*:

1. Tabelle für alle Knoten:
   - *fertig:* (init: *false*)
   - *dist:* (init: *0* für s, ∞ sonst)
   - *via:* (init: ? oder *null*)
2. ***While*** es in der Tabelle Knoten mit (*fertig* = *false* und *dist* < ∞) gibt:
   a. Knoten *v* suchen der unter allen Knoten mit *fertig* = *false* den kleinsten Wert für *dist* hat.
   b. *v.fertig* = *true* setzen
   c. Für jeden *Knoten w* zu dem es eine Kante <*v*, *w*> gibt:
      i. *int d* = *v.dist* + Kantengewicht von <*v*, *w*>
      ii. if (*d* < *w.dist*) { *w.dist* = *d*; *w.via* = *v*; }

Kürzeste Wege in gewichteten Graphen

Verbesserungsideen

Knoten mit (fertig == false) in **Set** speichern, dass nicht die ganze Tabelle durchsucht werden muss
-> hilft asymptotisch nichts

Knoten mit (fertig == false) in **Priority Queue** speichern, so dass zuvorderst derjenige Knoten mit der bisher kleinsten dist steht. z.B. mit einem Min-Heap:
-> Init: *Add* alle Knoten O(n)
a: *DeleteMin* O(log n),
c: *DecreaseKey* outdeg Mal O(log n)
-> **Total O((n+m) log n)**

Fachhochschule Nordwestschweiz

Kürzeste Wege - Übersicht

ungewichtete Kanten:	Breitensuche	$O(n + m)$
gewichtete Kanten:	Algorithmus von Dijkstra	$O(n^2 + m)$ oder $O((n+m)\log n)$ mit Heap

beide Algorithmen funktionieren für gerichtete und ungerichtete Graphen
die Laufzeiten beziehen sich auf Graphen in Adjazenzlisten-Darstellung

Fachhochschule Nordwestschweiz

Aufgabe

- Programmieraufgabe 2 (Dijkstra)

Prüfung

- 90 Minuten
- 2 Seiten Zusammenfassung
- Stoff: alles; Schwerpunkte: Priority Queues, Hashing, Graphen

Fragen im Teams-Chat

- Fragen zum Stoff werden bis 1 Woche vor Prüfung sicher beantwortet.

Vielen Dank für die tolle Mitarbeit!