Kürzester Wege in Graphen

Zur **Bestimmung von kürzesten Wegen in Graphen** betrachten wir zwei unterschiedliche Algorithmen:

- **Single Source Shortest Path Problem**
- **All Pairs Shortest Path-Problem**

Single Source Shortest Path-Problem I

- Wie komme ich möglichst schnell nach …?

Wegsuche und Navigation sind mittlerweile zu allgegenwärtigen Service-Anwendungen geworden

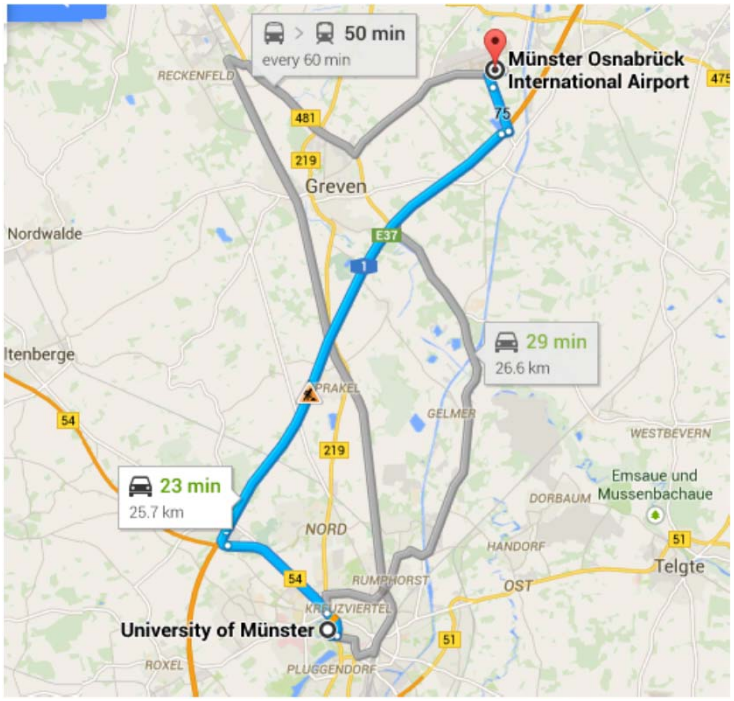
- GPS-Navigation im Auto
- Wegsuche auf dem Smart-Phone
- Reiseplanung und Verkehrsverbindung, z.B. DB-Navigator

- Suche nach Freunde in Sozialen Nezwerken, z.B. Twitter, Facebook, …

Single Source Shortest Path-Problem II

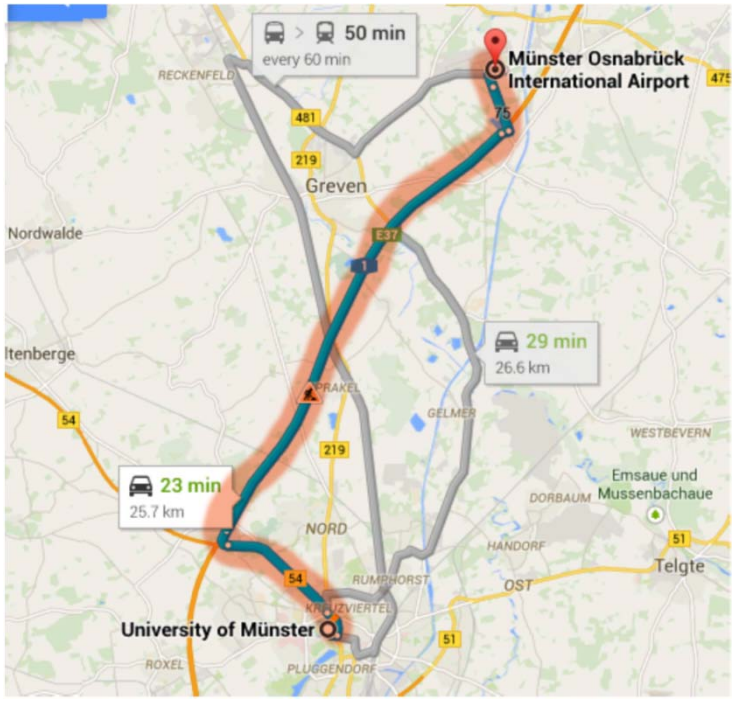
- Problemstellung & Anwendungen …
  - Was ist der kostengünstigste Weg von der Uni-Münster zum Flughafen?
  - Eingabe:
    1) Verkehrsnetzwerk G
    2) Startpunkt S
    3) Zielpunkt T
  - Ausgabe:

    Der kürzeste Weg von S nach T in G



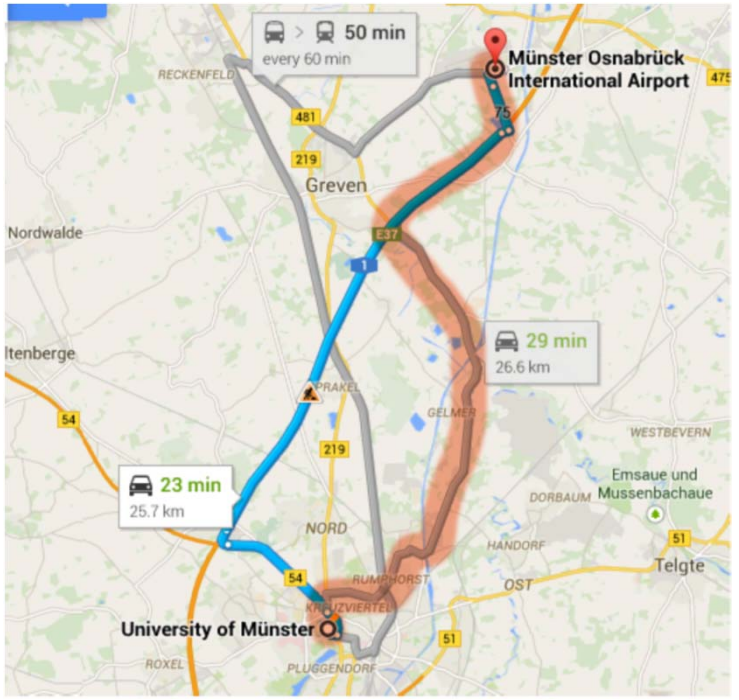
Single Source Shortest Path-Problem III

- Problemstellung & Anwendungen …
  - Was ist der kostengünstigste Weg von der Uni-Münster zum Flughafen?
  - Kosten:
    - Strecke
    - Zeit
    - CO_2-Emission
    - Sprit
  - Netzwerk:
    - Straßennetzwerk
    - ÖVM (Bus,Bahn,…)
    -



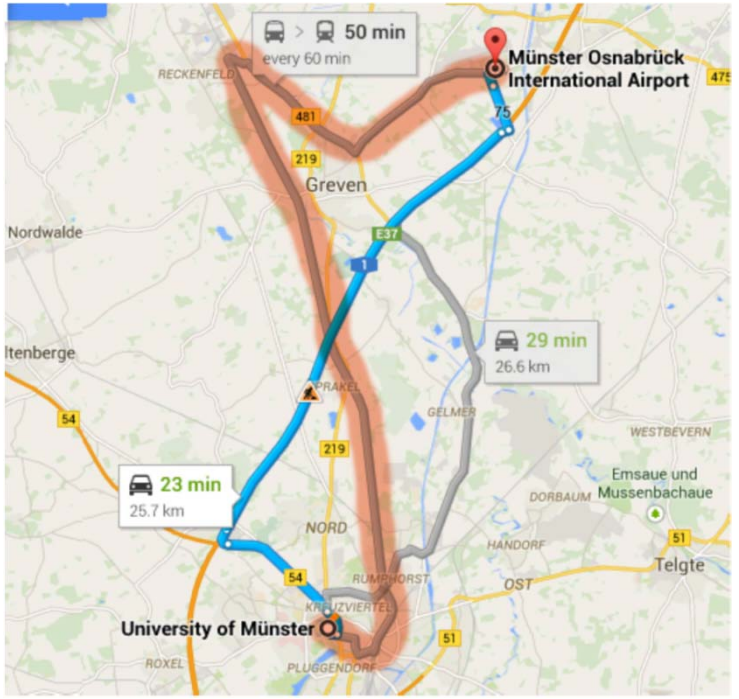
Single Source Shortest Path-Problem III

- Problemstellung & Anwendungen …
  - Was ist der kostengünstigste Weg von der Uni-Münster zum Flughafen?
  - Kosten:
    - Strecke
    - Zeit
    - CO_2-Emission
    - Sprit
  - Netzwerk:
    - Straßennetzwerk
    - ÖVM (Bus,Bahn,…)
    -



Single Source Shortest Path-Problem III

- Problemstellung & Anwendungen ...
  - Was ist der kostengünstigste Weg von der Uni-Münster zum Flughafen?
  - Kosten:
    - Strecke
    - Zeit
    - CO_2-Emission
    - Sprit
  - Netzwerk:
    - Straßennetzwerk
    - ÖVM (Bus,Bahn,...)
    -



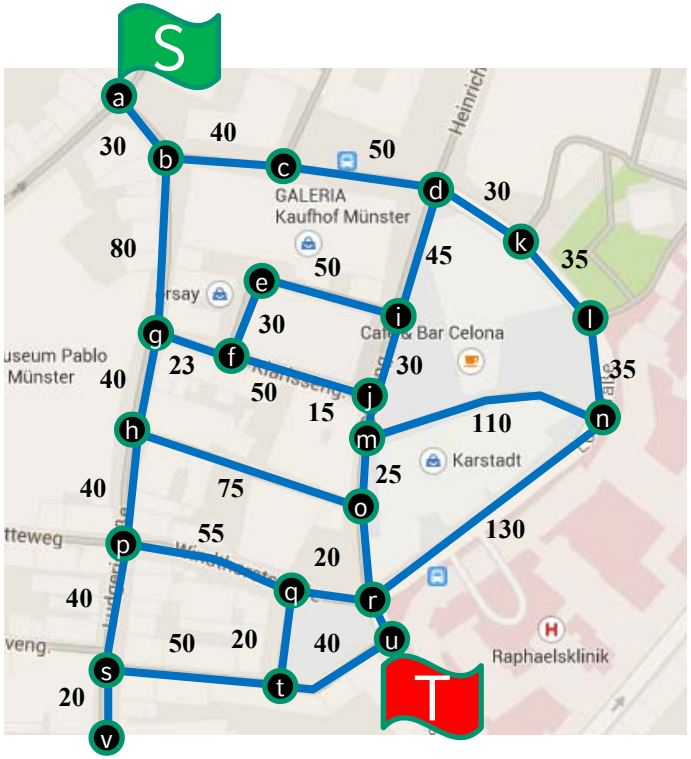
Single Source Shortest Path-Problem IV

- Problemstellung & Anwendungen …

Beispiel:

Berechnung des kürzesten Weg vom Startknoten S (a) zum Zielknoten T (u).

- Überführung der Straßenkarte in einen **gewichteten gerichteten Graph mit nicht-negativen Kantengewichten**
  - Kreuzungen → Knoten
  - Streckenabschnitt zwischen zwei Kreuzungen → Kante zwischen zwei Knoten
  - Länge eines Streckenabschnitts → Gewicht der entsprechenden Kante
  - Weitere Kantengewichte möglich die sich über einen Pfad hinweg linear akkumulieren, wie z.B. Zeit, Kosten, CO2-Emission, Benzinverbrauch, Anzahl der Ampeln, etc., und die wir minimieren wollen.
- Lösen des Problems anhand von Graphalgorithmen.
- Im Folgenden: Formale Betrachtung des Problems



Single Source Shortest Path-Problem V

- Definitionen
  - Gewichteter Gerichteter Graph:

Ein gewichteter gerichteter Graph ist ein Triple $G = (V, E, w)$ mit
- einer Menge V von Knoten
- einer Menge $E \subseteq \{(u,v) | u,v \in V\}$ von Kanten
- eine Gewichtsfunktion $w: E \rightarrow IR$

  - Pfad:

Ein Pfad $p_{s,t}$ der Länge k von einem Startknoten $s \in V$ zu einem Zielknoten $t \in V$ in $G(V,E,w)$ ist eine Folge $p_{s,t} = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ von Knoten, für die gilt:
- $s = v_0$, $t = v_k$
- $(v_i, v_{i+1}) \in E$, für alle $0 \leq i \leq k-1$

Single Source Shortest Path-Problem VI

- Definitionen
  - Gewicht eines Pfades und kürzester Pfad

Das Gewicht eines Pfades $p=\langle v_0, v_1, v_2, \ldots, v_k \rangle$ ist die Summe der Gewichte aller enthaltenen Kanten:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Das Gewicht des kürzesten Pfades von s nach t sei durch die Funktion

$$\delta(s,t) = \begin{cases} \min\{w(p_{s,t}) \mid p_{s,t} \in P(s,t)\}, & falls\ t\ von\ s\ aus\ erreichbar \\ \infty & sonst \end{cases}$$

definiert.

P(s,t) := Menge aller Pfade von s nach t.

Ein Pfad $p_{s,t}$ mit dem Gewicht $\delta(s,t)$ wird als kürzester Pfad von s nach t bezeichnet.

Single Source Shortest Path-Problem VII

- Eigenschaften kürzester Pfade
  - Optimale-Teilstruktur-Eigenschaft:
    - Jeder Teilpfad eines kürzesten Pfades ist ebenfalls ein kürzester Pfad

Theorem 1:
Gegeben: gewichteter gerichteter Graph G = (V,E,w), kürzester Pfad p_{v_0,v_k} in G;
Jeder Teilpfad p_{v_i,v_j} von p_{v_0,v_k} $(0 \leq i \leq j \leq k)$ erfüllt ebenfalls die kürzester-Pfad-Eigenschaft, d.h.

$$w(p_{v_i,v_j}) = \delta(v_i, v_j).$$

- Beweis: Annahme p_{v_0,v_k} ist kürzester Pfad von v_0 nach v_k

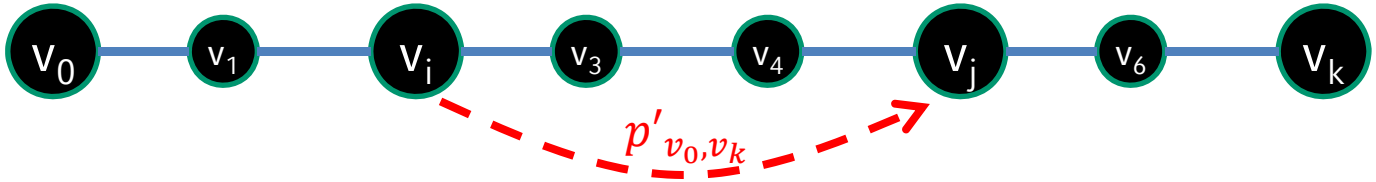
Single Source Shortest Path-Problem VII

- Eigenschaften kürzester Pfade
  - Optimale-Teilstruktur-Eigenschaft:
    - Jeder Teilpfad eines kürzesten Pfades ist ebenfalls ein kürzester Pfad

Theorem 1:
Gegeben: gewichteter gerichteter Graph G = (V,E,w), kürzester Pfad p_{v_0,v_k} in G;
Jeder Teilpfad p_{v_i,v_j} von p_{v_0,v_k} $(0 \leq i \leq j \leq k)$ erfüllt ebenfalls die kürzester-Pfad-Eigenschaft, d.h.

$$w(p_{v_i,v_j}) = \delta(v_i, v_j).$$

  - Beweis: Sei der Pfad p'_{v_i,v_j} kürzer als Teilpfad p_{v_i,v_j}



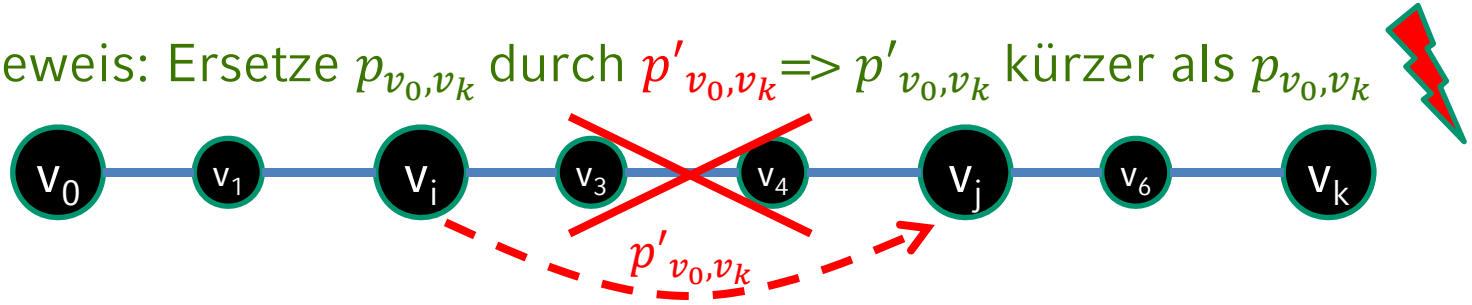
Single Source Shortest Path-Problem VII

- Eigenschaften kürzester Pfade
  - Optimale-Teilstruktur-Eigenschaft:
    - Jeder Teilpfad eines kürzesten Pfades ist ebenfalls ein kürzester Pfad

Theorem 1:
Gegeben: gewichteter gerichteter Graph G = (V,E,w), kürzester Pfad p_{v_0,v_k} in G;
Jeder Teilpfad p_{v_i,v_j} von p_{v_0,v_k} $(0 \leq i \leq j \leq k)$ erfüllt ebenfalls die kürzester-Pfad-Eigenschaft, d.h.

$$w(p_{v_i,v_j}) = \delta(v_i, v_j).$$

  - Beweis: Ersetze p_{v_0,v_k} durch p'_{v_0,v_k} => p'_{v_0,v_k} kürzer als p_{v_0,v_k}



Single Source Shortest Path-Problem VII

- Eigenschaften kürzester Pfade
  - Optimale-Teilstruktur-Eigenschaft:
    - Jeder Teilpfad eines kürzesten Pfades ist ebenfalls ein kürzester Pfad

Theorem 1:
Gegeben: gewichteter gerichteter Graph G = (V,E,w), kürzester Pfad p_{v_0,v_k} in G;
Jeder Teilpfad p_{v_i,v_j} von p_{v_0,v_k} $(0 \leq i \leq j \leq k)$ erfüllt ebenfalls die kürzester-Pfad-Eigenschaft, d.h.

$$w(p_{v_i,v_j}) = \delta(v_i, v_j).$$

  - Theorem 1 ist Hauptindikator dafür dass der kürzeste Pfad effizient (über Greedy-Ansatz oder dynamische Programmierung) ermittelt werden kann.

Single Source Shortest Path-Problem VIII

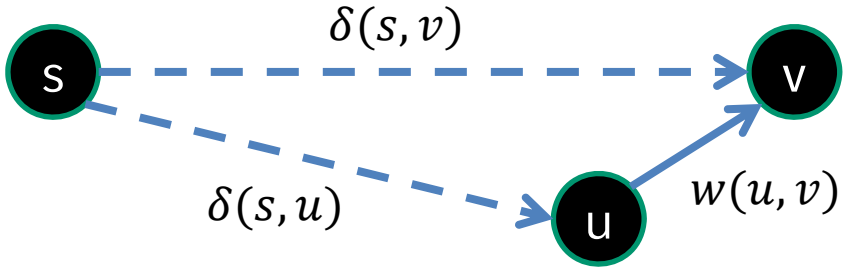
- Eigenschaften kürzester Pfade
  - Dreiecksungleichung

Theorem 2:
Gegeben: gewichteter gerichteter Graph G = (V,E,w) und Startknoten s;

Für alle Kanten (u,v)∈E gilt:

$$\delta(s,v) \leq \delta(s,u) + w(u,v).$$

- Beweis:

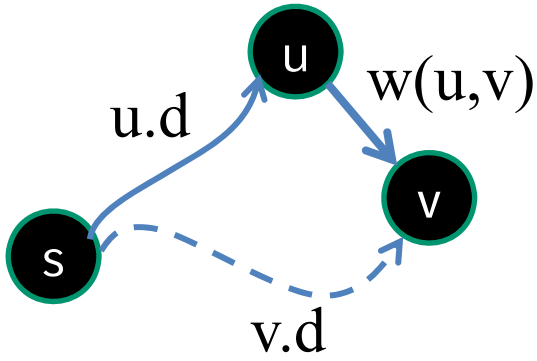


  - Falls δ(s,v) > δ(s,u) + w(u,v), dann wäre das Gewicht des Pfades von Knoten s nach v über Knoten u kleiner als δ(s,v) und würde somit der Definition von δ(s,v) widersprechen.

Single Source Shortest Path-Problem IX

- Relaxation
  - Die Methode der Relaxation spielt eine zentrale Rolle für die folgenden SSSP-Algorithmen.
  - Prinzip:
    - Verwalte für jeden Knoten $v \in V$ eine konservative Abschätzung (obere Schranke) v.d für das Gewicht $\delta(s,v)$ des kürzesten Pfades von Startknoten $s \in V$ nach v.
    - Relaxieren einer Kante $(u,v) \in E$:

```
RELAX(u,v,w)
1 if v.d > u.d + w(u,v) then
2    v.d = u.d + w(u,v)
3    v.π = u
4 endif
```



Single Source Shortest Path-Problem X

- Das kürzeste-Pfade-Problem (SSSP)
  - Gegeben sei
    - ein gewichteter gerichteter Graph G=(V,E,w)
    - mit nicht-negativen!!! Kantengewichten, d.h. $w: E \rightarrow \mathbb{R}_0^+$
    - und ein Startknoten s∈V
  - Gesucht ist
    - zu jedem Knoten t∈V, jeweils ein kürzester Pfad von s nach t

$$SSSP(s,t) = \{p_{s,t} \mid t \in \mathrm{V}, w(p_{s,t}) = \delta(s,t)\}$$

SSSP = single source shortest path

Single Source Shortest Path-Problem XI

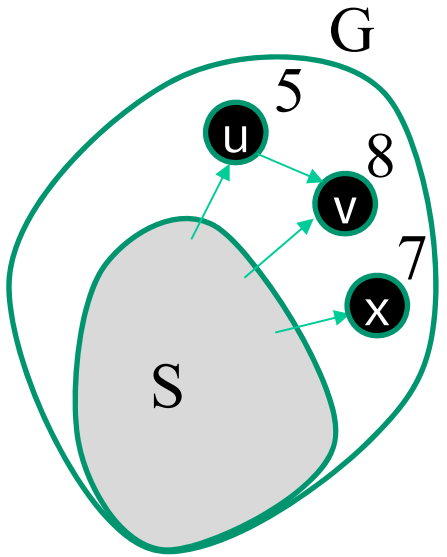
- Das kürzeste-Pfade-Problem (SSSP)
  - 1. Lösungsansatz:

    Naive Lösung (erschöpfende Suche)
    - Berechnung aller Pfade von s nach t
    - Auswahl und Ausgabe des kürzesten Pfades
  - Problem:
    - Selbst unter Ausschluss aller Zyklen gibt es eine sehr große Anzahl an möglichen Pfaden (Exponentiell in der Pfadlänge).
  - Im Folgenden wird gezeigt wie Probleme dieser Problemklasse effizient gelöst werden können.

Single Source Shortest Path-Problem XII

- Das kürzeste-Pfade-Problem (SSSP)
  - 2. Lösungsansatz: Greedy-Ansatz

1. Betrachte eine Menge $S \subseteq V$ von Knoten für die jeweils das Gewicht des kürzesten Pfades von s bekannt sei;



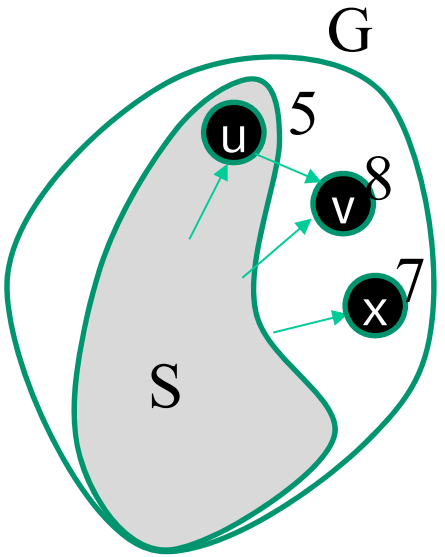
Single Source Shortest Path-Problem XII

- Das kürzeste-Pfade-Problem (SSSP)
  - 2. Lösungsansatz: Greedy-Ansatz

1. Betrachte eine Menge $S \subseteq V$ von Knoten für die jeweils das Gewicht des kürzesten Pfades von s bekannt sei;

Schrittweise:

2. Füge der Menge S einen Knoten $u \in V - S$ hinzu, für den das abgeschätzte Gewicht $u.d$ des kürzesten Pfades von s minimal ist, d.h. $\forall v \in V - S: u.d \leq v.d$;



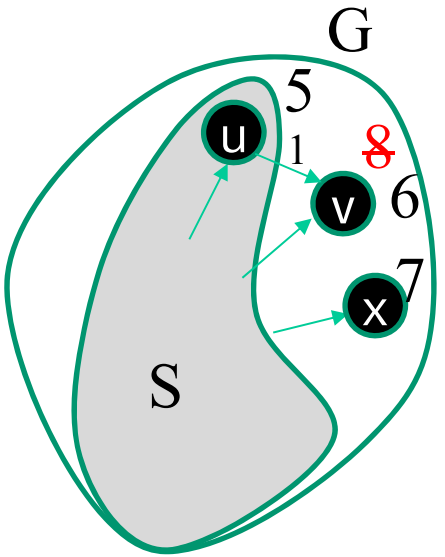
Single Source Shortest Path-Problem XII

- Das kürzeste-Pfade-Problem (SSSP)
  - 2. Lösungsansatz: Greedy-Ansatz

1. Betrachte eine Menge $S \subseteq V$ von Knoten für die jeweils das Gewicht des kürzesten Pfades von s bekannt sei;

Schrittweise:

2. Füge der Menge S einen Knoten $u \in V - S$ hinzu, für den das abgeschätzte Gewicht $u.d$ des kürzesten Pfades von s minimal ist, d.h. $\forall v \in V - S: u.d \leq v.d$;
3. Relaxierung: Erneuere die Gewichtsabschätzung $v.d$ für alle Knoten $v \in V$ die von u aus direkt erreichbar sind, d.h. für die gilt $(u, v) \in E$



Single Source Shortest Path-Problem XIII

- Das kürzeste-Pfade-Problem (SSSP)
  - 2. Lösungsansatz: Dijkstra's Algorithmus

```
DIJKSTRA(G,s)
1   for each v ∈ G.V-{s} do
2     v.d ← ∞
3     v.π ← null
4   endfor
5   s.d ← 0
6   S ← ∅
7   Q ← G.V      \\ Q := Prioritäts-Warteschlange gemäß Attribut d
8   while Q ≠ ∅ do
9     u ← get_first(Q) \\ entnimmt v aus Q mit niedrigstem v.d
10    S ← S ∪ {u}
11    for each v ∈ G.Adj(u) do
12      RELAX(u,v,w)
13    end for
14  end while
```

Initialisierung

Single Source Shortest Path-Problem XIII

- Das kürzeste-Pfade-Problem (SSSP)
  - 2. Lösungsansatz: Dijkstra's Algorithmus

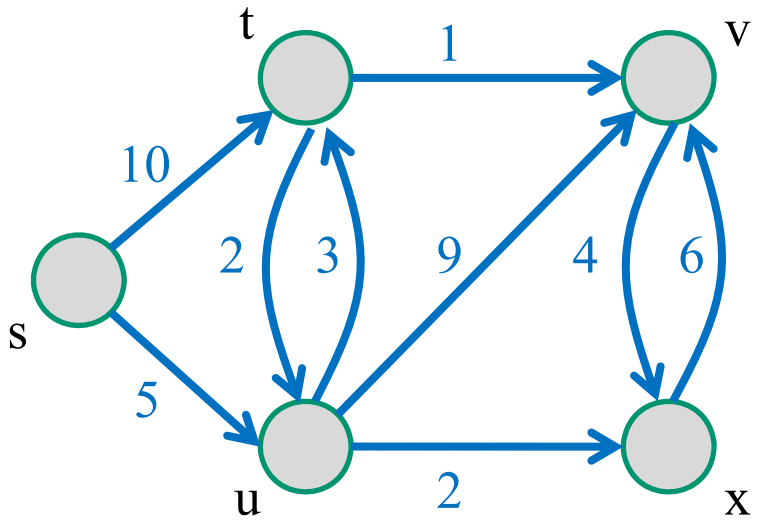
```
DIJKSTRA(G,s)
1  for each v ∈ G.V-{s} do
2    v.d ← ∞
3    v.π ← null
4  endfor
5  s.d ← 0
6  S ← ∅
7  Q ← G.V      \\ Q := Prioritäts-Warteschlange gemäß Attribut d
8  while Q ≠ ∅ do
9    u ← get_first(Q) \\ entnimmt v aus Q mit niedrigstem v.d
10   S ← S ∪ {u}
11   for each v ∈ G.Adj(u) do
12     RELAX(u,v,w)
13   end for
14 end while
```

Initialisierung

```
1 if v.d > u.d + w(u,v) then
2   v.d = u.d + w(u,v)
3   v.π = u
```

Single Source Shortest Path-Problem XIV

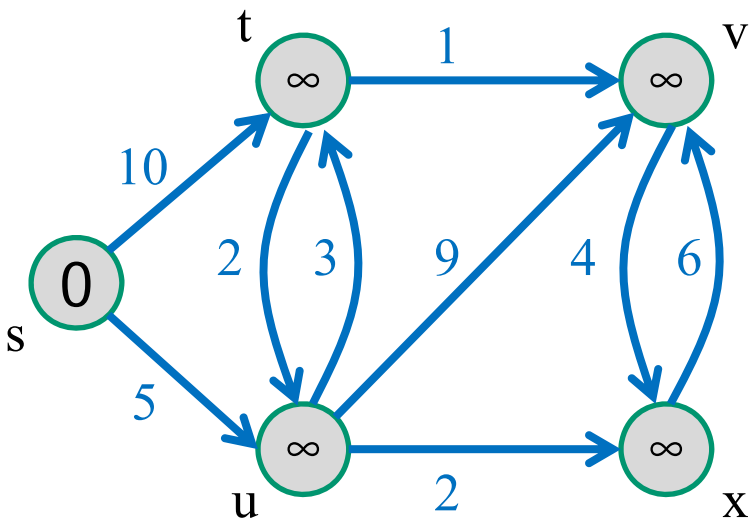
- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



	s.d	t.d	u.d	v.d	x.d

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



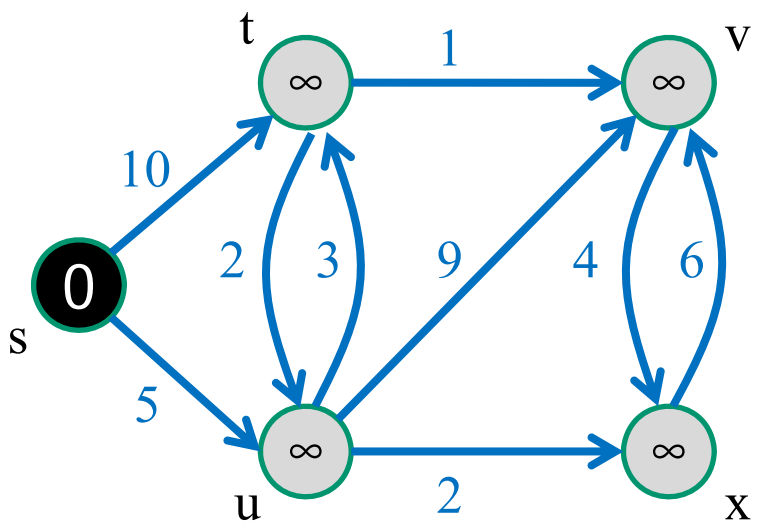
s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞

Initialisierung

$S = \{\ \}$ $\quad Q = \{ s, t, u, v, x \}$

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



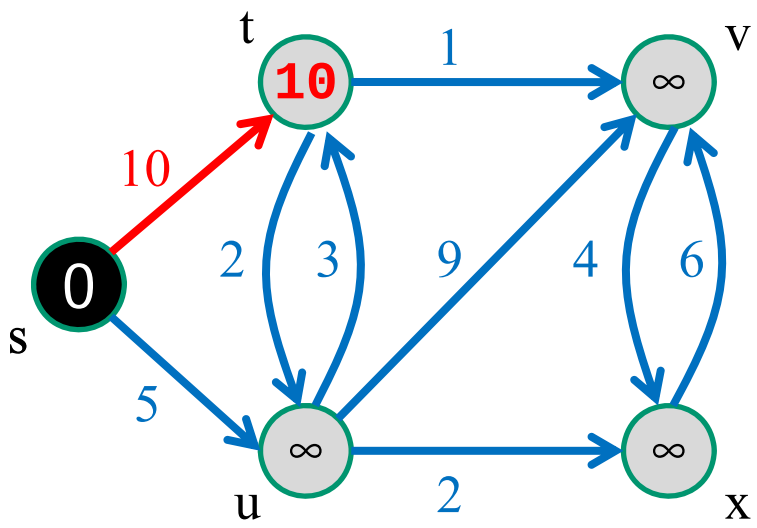
s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞

1. Iteration

$S = \{ s \}$ $Q = \{ s, t, u, v, x \}$

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞
	10	∞	∞	∞

1. Iteration

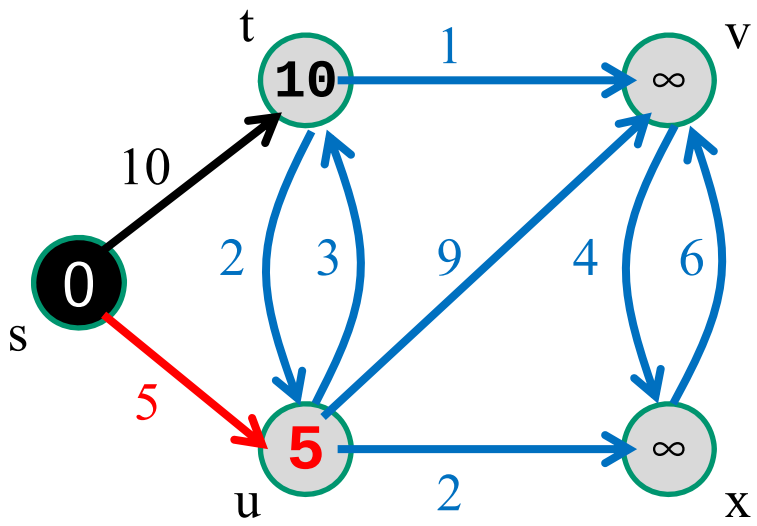
RELAX(s , t , w)

S = { s }

Q = { t , u , v , x }

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞
	10	**5**	∞	∞

1. Iteration

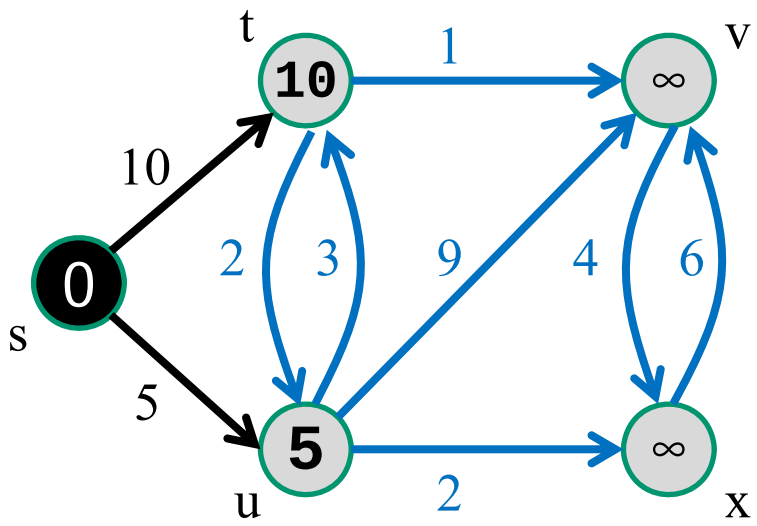
RELAX(s , t , w)
RELAX(s , u , w)

S = { s }

Q = { t , u , v , x }

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞
	10	5	∞	∞

1. Iteration

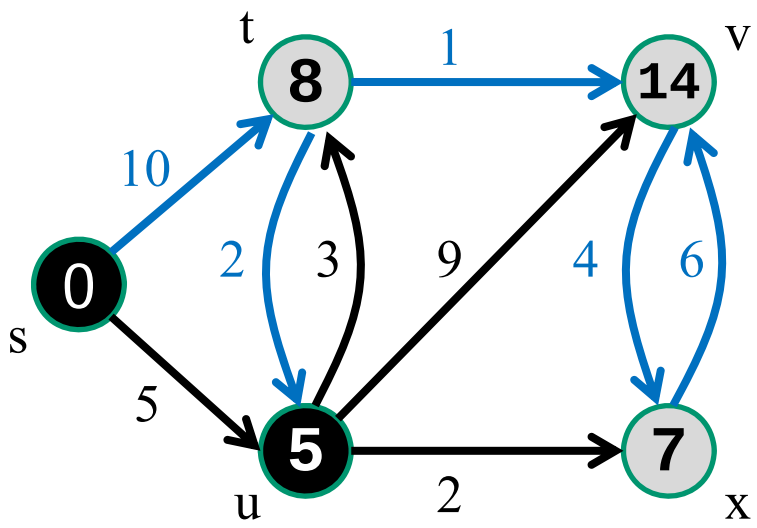
RELAX(s , t , w)
RELAX(s , u , w)

S = { s }

Q = { t , u , v , x }

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞
	10	5	∞	∞
	8		14	7

2. Iteration

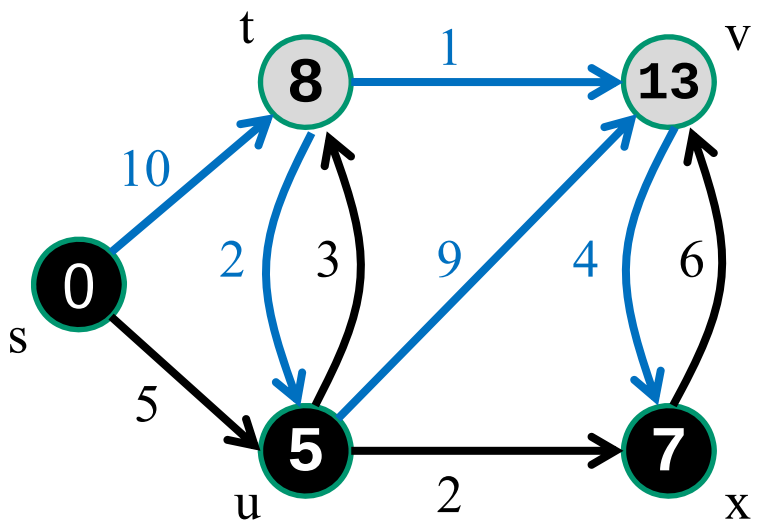
RELAX(u , t , w)
RELAX(u , v , w)
RELAX(u , x , w)

S = { s , u }

Q = { t , v , x }

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞
	10	5	∞	∞
	8		14	7
	8		13	

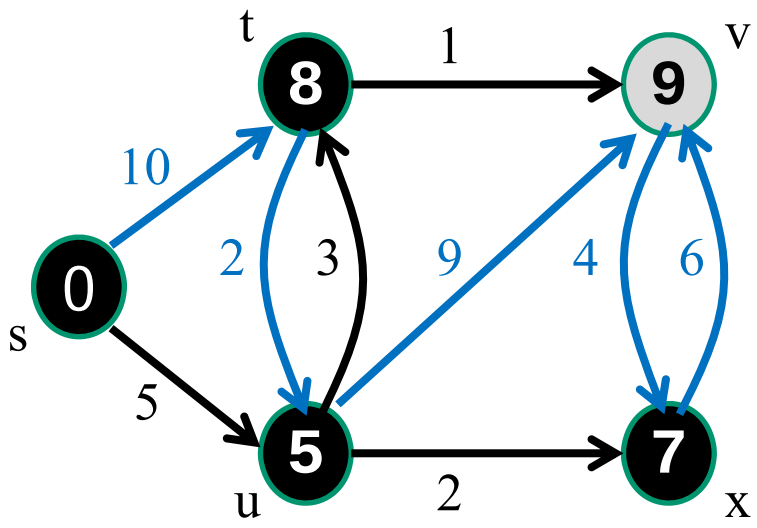
3. Iteration

RELAX(x , v , w)

S = { s , u , x } Q = { t , v }

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞
	10	5	∞	∞
	8		14	7
	8		13	
			9	

4. Iteration

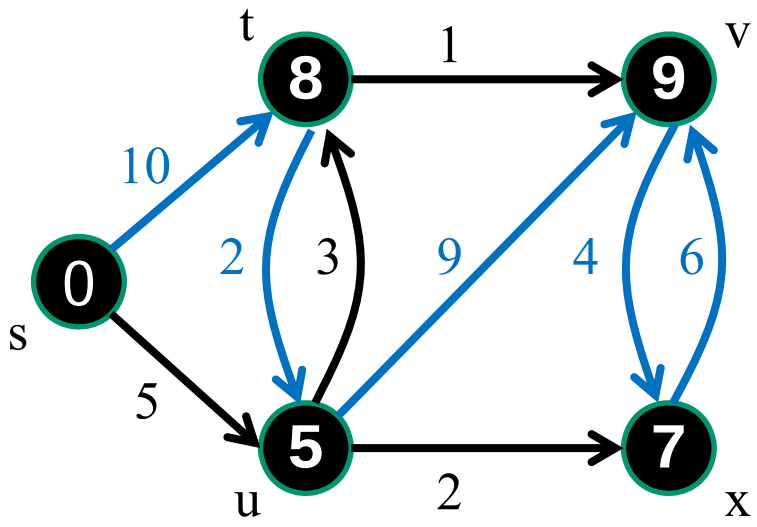
RELAX(t , u , w)

RELAX(t , v , w)

$S = \{ s , u , x , t \}$ $Q = \{ v \}$

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞
	10	5	∞	∞
	8		14	7
	8		13	
			9	

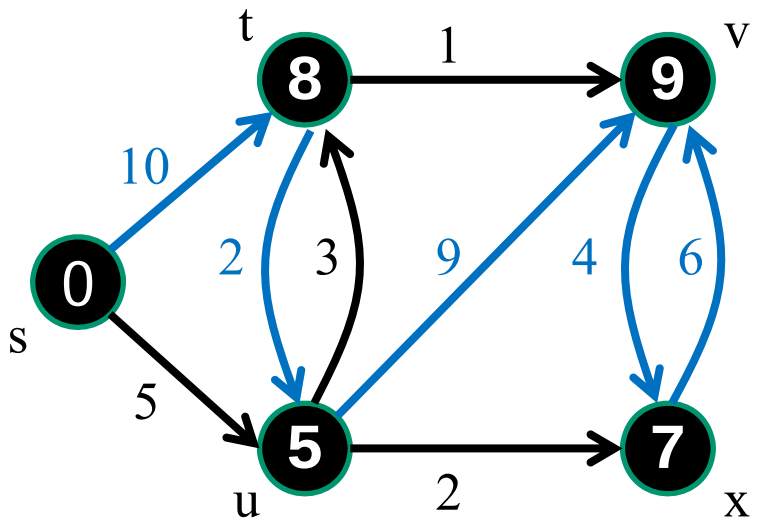
5. Iteration

RELAX(v , x , w)

S = { s , u , x , t , v } Q = { }

Single Source Shortest Path-Problem XIV

- Das kürzeste-Pfade-Problem (SSSP)
  - Beispiel:



s.d	t.d	u.d	v.d	x.d
0	∞	∞	∞	∞
	10	5	∞	∞
	8		14	7
	8		13	
			9	

$S = \{ s, u, x, t, v \}$ $Q = \{ \}$ → Abbruch der while-Schleife

Single Source Shortest Path-Problem XV

- Das kürzeste-Pfade-Problem (SSSP)
  - Korrektheit:
    - Algorithmus terminiert mit $u.d = \delta(s,u)$ für alle $u \in V$, falls G ein gerichteter gewichteter Graph mit nicht–negativen Kantengewichten ist.
    - Für den Beweis …

      … müsste gezeigt werden dass für alle Knoten $v \in V$ die zur Menge S hinzugenommen werden folgendes gilt:

      $$v.d = \delta(s,v)$$

Single Source Shortest Path-Problem XVI

- Das kürzeste-Pfade-Problem (SSSP)
  - Laufzeitanalyse des Algorithmus von Dijkstra
    - Initialisierung:

```
DIJKSTRA(G,s)
1  for each v ∈ G.V-{s} do
2    v.d ← ∞
3    v.π ← null
4  endfor
5  s.d ← 0
6  S ← ∅
7  Q ← G.V      \\ Q := Prioritäts-Warteschlange gemäß Attribut d
```

      - Für jeden Knoten: Zuweisung (O(1)) + Einfügen in Q (O(1))
      - Gesamtkomplexität: O(V)

Single Source Shortest Path-Problem XVI

- Das kürzeste-Pfade-Problem (SSSP)
  - Laufzeitanalyse des Algorithmus von Dijkstra
    - Schleifendurchlauf:

```
8  while Q ≠ ∅ do
9    u ← get_first(Q)
10   S ← S ∪ {u}
11   for each v ∈ G.Adj(u) do
12     RELAX(u,v,w)
13   end for
14 end while
```

|G.Adj(u)| mal

|V| mal

    - U ← get_first(Q) wird |V| mal aufgerufen
    - RELAX(u,v,w) wird insgesamt |E| mal ausgeführt

Single Source Shortest Path-Problem XVI

- Das kürzeste-Pfade-Problem (SSSP)
  - Laufzeitanalyse des Algorithmus von Dijkstra
    - Schleifendurchlauf:

```
8  while Q ≠ ∅ do
9    u ← get_first(Q)
10   S ← S ∪ {u}
11   for each v ∈ G.Adj(u) do
12     RELAX(u,v,w)
13   end for
14 end while
```

|G.Adj(u)| mal

|V| mal

Zeitrelevante Operationen:

- Zeile 9: Ermittlung des minimalen Schlüssels: |V|-mal
- Zeile 12: Aktualisierung des Schlüssels: insgesamt |E|-mal

Single Source Shortest Path-Problem XVII

- Das kürzeste-Pfade-Problem (SSSP)
  - Laufzeitanalyse des Algorithmus von Dijkstra
    - Laufzeit über $|V| \cdot T_{get\_first(Q)} + |E| \cdot T_{decrease\_key}$ bestimmt.

Q:	$T_{get\_first(Q)}$:	$T_{decrease\_key}$:	Gesamtlaufzeit:
Array	O(V)	O(1)	$O(E + V^2)$
Heap	O(log V)	O(log V)	O(E log V)
Fibo.-Heap	O(log V)	O(1)	O(E + V log V)

    - Speicherplatzbedarf = O(V+E) (je nach Implementierung)

All Pairs Shortest Path-Problem I

Problem:

- Vorgaben wie beim single source shortest path-Problem
- Bestimmung der kürzesten Wege zwischen allen Paaren von Knoten des Graphen G zwischen denen eine Verbindung besteht.

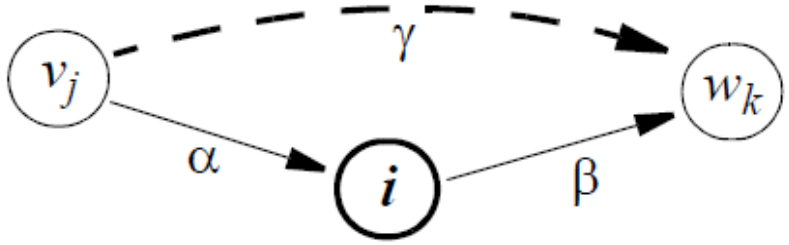
Das Problem kann trivialerweise durch iterative Anwendung des Algorithmus von Dijkstra auf alle Knoten des Graphen gelöst werden. Wir stellen hier jedoch einen wesentlich einfacheren Algorithmus vor, der das Problem direkt löst:

All Pairs Shortest Path-Problem II

Der Algorithmus von Floyd:

- Knoten des Graphen G sind mit 1 bis n durchnummeriert.
- Der Algorithmus berechnet eine Folge $G_0 = G, G_1, \ldots, G_n$ von Graphen, wobei Graph G_i, $1 \leq i \leq n$, wie folgt definiert ist:
  - G_i besitzt die gleiche Knotenmenge wie G
  - In G_i existiert eine Kante von Knoten v nach Knoten w mit Kosten α $\Leftrightarrow$ es gibt einen Pfad von v nach w, der nur Knoten aus $\{1, \ldots, i\}$ verwendet und der kürzeste dieser Pfade hat die Länge α.
- G_i entsteht durch folgende Modifikation von G_{i-1} im i-ten Schritt des Algorithmus
- Seien $v_1, \ldots, v_r$ die Vorgänger und $w_1, \ldots, w_s$ die Nachfolger von Knoten i im Graphen G_{i-1}, so betrachtet man alle Paare (v_j, w_k), $1 \leq j \leq r$ und $1 \leq k \leq s$.

All Pairs Shortest Path-Problem III

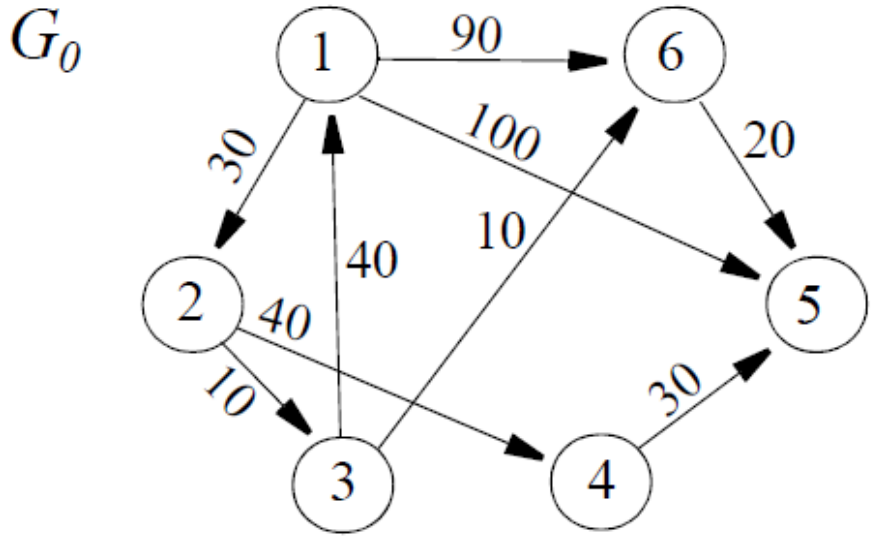


- Falls noch keine Kante von v_j nach w_k existiert, so erzeuge eine entsprechend Kante.
- Existiert bereits eine solche Kante mit Kosten γ, so ersetze γ durch $\alpha + \beta$, falls $\alpha + \beta < \gamma$.
- G_i erfüllt dabei wiederum obige Eigenschaften, denn:
  - In G_{i-1} waren alle kürzesten Pfade durch Kanten repräsentiert, die nur auf Basis der Zwischenknoten $\{1, \ldots, i-1\}$ entstanden sind.
  - Jetzt sind alle Pfade bekannt, die Knoten aus $\{1, \ldots, i\}$ benutzen
- Nach n Schritten sind in Graph G_n die Kosten alle kürzesten Wege von Graph G repräsentiert.

All Pairs Shortest Path-Problem IV

Beispiel:

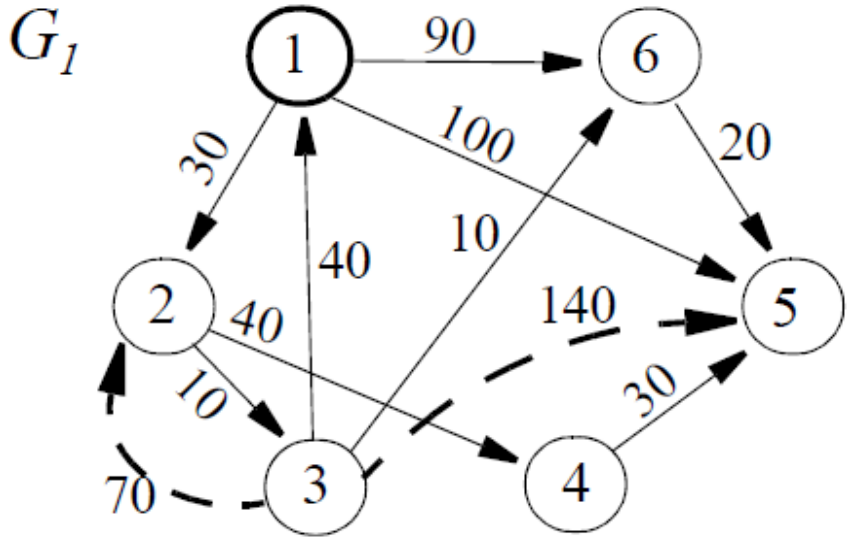
- Gegeben sei der folgende Beispielsgraph, wobei die Knoten nummeriert sind:



All Pairs Shortest Path-Problem V

- Im ersten Schritt werden alle Paare von Vorgängern (3) und Nachfolgern (2,5,6) von Knoten 1 betrachtet. In den Graphen G_0 werden dabei die folgenden Kanten eingefügt, resultierend im Graphen G_1:

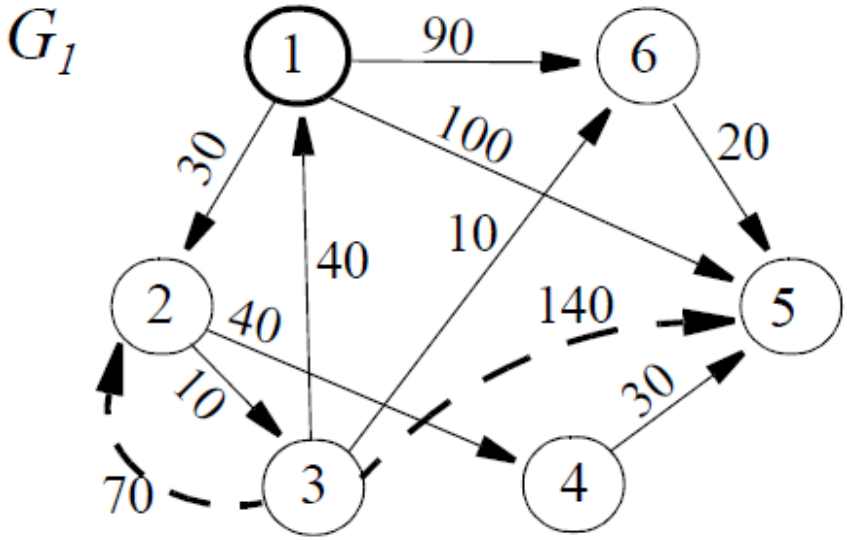
  Kante (3,2) mit Kosten 70 und
  Kante (3,5) mit Kosten 140:



All Pairs Shortest Path-Problem VI

- Im nächsten Schritt betrachten wir innerhalb von G_1 die Vorgänger (1,3) und Nachfolger (3,4) von Knoten 2. Hinzugefügte Kanten (resultierend in Graph G_2) sind hierbei:

  Kante (1,3) (Kosten 40).
  Kante (1,4) (Kosten 70).
  Kante (3,4) (Kosten 110).



All Pairs Shortest Path-Problem VII

- Denselben Vorgang wiederholen wir nun für die Knoten 3, 4, 5 und 6:

Knoten	Vorgänger	Nachfolger	Neue Kanten	Veränderte Kanten
'3'	1, 2	1, 2, 4, 5, 6	(2,1): Kosten 50 (2,5): Kosten 150 (2,6): Kosten 20	(1,6): Kosten 50
'4'	1, 2, 3	5		(2,5): Kosten 70
'5'	1, 2, 3, 4, 6	--		
'6'	1, 2, 3	5		(1,5): Kosten 70 (2,5): Kosten 40 (3,5): Kosten 30

All Pairs Shortest Path-Problem VIII

- Als Resultat ergibt sich ein gerichteter Graph G_6, der durch die folgende Kostenmatrix beschrieben wird:

G_6	1	2	3	4	5	6
1	--	30	40	70	70	50
2	50	--	10	40	40	20
3	40	70	--	110	30	10
4	--	--	--	--	30	--
5	--	--	--	--	--	--
6	--	--	--	--	20	--

Anmerkung: Die Knoten 1, 2, 3 und 6 sind von Knoten 4 aus und die Knoten 1 bis 4 sind von Knoten 6 aus nicht erreichbar; von Knoten 5 aus erreicht man keinen anderen Knoten. Die entsprechenden Kanten existieren daher im Ergebnisgraphen G_6 nicht; es gibt keine (und damit keine kürzeste) zugehörige Verbindung.

All Pairs Shortest Path-Problem VIX

Konkrete, vereinfachte **Implementierung des Algorithmus von Floyd** auf Basis der Implementierung des Graphen mit Hilfe einer Kostenmatrix:

- Sei C die Kostenmatrix für den Ausgangsgraphen $G = (V, E)$.
- Diese sei als globale Variable vorgegeben.
- Dabei sei vereinbart, dass $C[i, i] = 0$, $1 \leq i \leq n$ und $C[i, j] = \infty$, falls es keine Ante von Knoten i nach Knoten j in G gibt.

All Pairs Shortest Path-Problem VX


```
double [][] floyd () {
   double [][]a = new double[n][n];
   // Zuweisung der Kostenmatrix an Matrix a (by value !)
   for (int i = 1; i < n; i++) { // aktueller Knoten i
      for (int j = 1; j < n; j++) { // Vorgänger j betrachten
         for (int k = 1; k < n; k++) // Nachfolger k betrachten
            if (a[j][i] + a[i][k] < a[j][k]) {
               // kürzerer Weg über i gefunden
               a[j][k] = a[j][i] + a[i][k];
            }
         }
      }
      return a;
   }
}
```

Die Laufzeit für diese Implementierung ist offensichtlich $O(n^3)$.

All Pairs Shortest Path-Problem VXI

- Im Laufe des Algorithmus erhöht sich die Kantenzahl zum Teil deutlich und kommt in der Regel der Maximalzahl von n^2 recht nahe. Daher kann auch für eine Darstellung über Adjazenzlisten kein entscheidend verbessertes Laufzeitverhalten erwartet werden.
- Aufgrund ihrer Einfachheit ist daher die obige Implementierung im Allgemeinen zu bevorzugen.

Bisher:

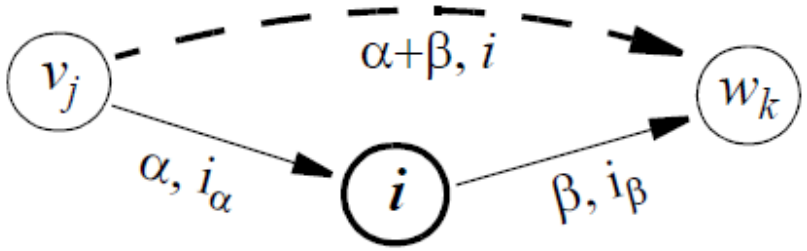
Berechnung ausschließlich der Kosten der kürzesten Pfade

Jetzt:

Repräsentation der Pfade selbst mittels geeigneter zusätzlicher Kantenmarkierungen.

All Pairs Shortest Path-Problem VXII

- Konkret: Markierung beim Übergang von Graph G_{i-1} auf G_i (s.o.) die neu hinzugekommenden bzw. geänderten Kanten mit Knoten i, d.h. mit dem Knoten, über den der (neue) minimale Weg verläuft.



- Zusätzliche (zu Beginn mit 0-Feldern initialisierte) Matrix:

```
int[][] pathCost = new int[n][n];
```

- In der `if`-Anweisung des obigen Algorithmus von Floyd wird der folgende zusätzliche Befehl eingefügt:

```
pathCost[j][k] = i;
```

All Pairs Shortest Path-Problem VXIII

- Nach Terminierung des Algorithmus erhält die Matrix ‚`pathCost`' die folgende Information:
  - `pathCost[i][j] = 0`
    → Kante zwischen Knoten i und j ist kostenminimal, falls sie existiert; ansonsten: kein Weg von Knoten i nach Knoten j vorhanden.
  - `pathCost[i][j] = k > 0`
    → gibt an, über welchen Knoten k die kürzeste Verbindung verläuft

- Aus dieser Matrix lässt sich der kürzeste Weg zwischen zwei beliebigen Knoten i und j in $O(r)$ Zeit konstruieren, wobei r die Anzahl der Knoten auf dem kürzesten Weg bezeichnet.

Transitive Hülle I

Häufig (unabhängig vom Knoten) nur von Interesse:
„existiert ein Weg im Graph G von Knoten i nach Knoten j?

Allgemeines Problem:

Welche Knoten eines Graphen sind von welchem anderen Knoten aus durch einen Weg erreichbar?
(**Zusammenhangsproblem** oder **Erreichbarkeitsproblem**)

Definition: Transitive Hülle

Gegeben sei ein Graph $G = (V, E)$. Die **transitive Hülle** von G ist der Graph $\bar{G} = (V, \bar{E})$, wobei $\bar{E}$ definiert ist durch: $(v, w) \in \bar{E} \Leftrightarrow$ es gibt einen Weg in G von v nach w.

Lösung: Vereinfachung des Algorithmus von Floyd.

Transitive Hülle II

Der Algorithmus von Warshall

Annahme:

G ist durch eine (boolesche) Adjazenzmatrix $A[i,j]$ gegeben. Hieraus wird eine Adjazenzmatrix $\bar{A}[i,j]$ für $\bar{G}$ generiert.

Prinzip:

- Ähnlich zum Algorithmus von Floyd wird beginnend mit G eine Folge von Graphen $G, G_1, \ldots, G_n$ mit einer zunehmenden Menge von Kanten generiert.

...

Transitive Hülle III

...

- Die Einträge $A_i[j,k]$ der Adjazenzmatrix A_{i-1} von Graph G_i, $1 \leq i \leq n$, entstehen dabei aus der Adjazenzmatrix A_{i-1} von Graph G_{i-1} wie folgt:

$$A_i[j,k] := A_{i-1}[j,k]\ OR\ (A_{i-1}[j,i]\ AND\ A_{i-1}[i,k])$$

- In der resultierenden Adjazenzmatrix A_n schließlich sind genau diejenigen Einträge $A_n[j,k]$ mit **true** belegt, für die gilt: ‚es gibt einen Weg in G von Knoten j nach Knoten k'. G_n repräsentiert somit die transitive Hülle von G.

Transitive Hülle IV

Folgende Implementierung entspricht der des Algorithmus von Floyd mit Ausnahme der `if`-Anweisung im Inneren der geschachtelten Schleifen:

```
boolean[][] warshall () {
   boolean a[][] = new boolean[n][n];
   // Zuweisung der Adjazenzmatrix an Matrix a (by value !)
   for (int i = 1; i < n; i++) {          // aktueller Knoten i
      for (int j = 1; j < n; j++) {       // Vorgänger j betrachten
         for (int k = 1; k < n; k++)      // Nachfolger k betrachten
            if (!a[j][k]) {   // Kante (j, k) existiert noch nicht
               a[j][k] = (a[j][i] && a[i][k]);
            }
         }
      }
   return a;
}
```

Minimaler Spannbaum I

Definition: Spannbaum eines Graphen:

Gegeben sei ein zusammenhängender, ungerichteter Graph $G = (V, E)$.

Ein **Spannbaum** von G ist ein Teilgraph $\bar{G} = (V, \bar{E})$, bei dem zwei (und somit alle drei) der folgenden Bedingungen erfüllt sind:

(1) $\bar{G}$ ist zusammenhängend.

(2) $\bar{G}$ besitzt $n - 1$ Kanten ($n = |V|$).

(3) $\bar{G}$ ist zyklenfrei.

Bemerkungen:

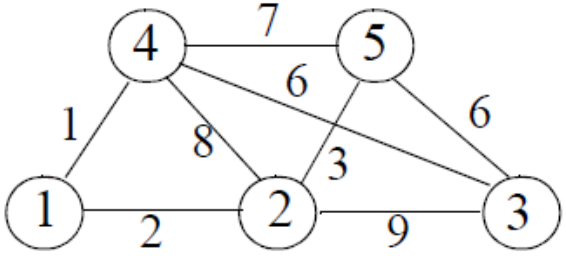
- Jeder zusammenhängende Graph besitzt mind. 1 Spannbaum.
- Die Anzahl verschiedener Spannbäume von n Knoten beträgt n^{n-2}. (ohne Beweis)

Minimaler Spannbaum II

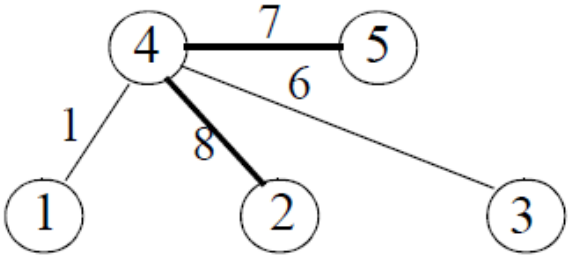
Definition: Minimaler Spannbaum:

Sind den Kanten des Graphen G Kosten zugeordnet, so besitzt jeder Spannbaum Kosten, die sich aus der Summe seiner Kantenkosten ergeben.

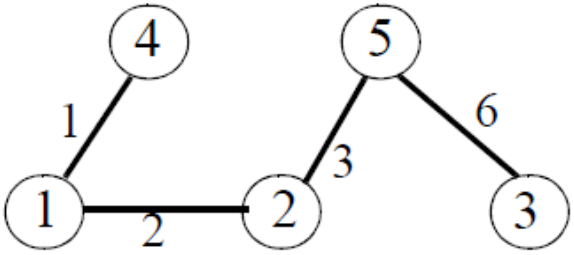
Man spricht dann von einem **minimalen Spannbaum** (**kostenminimaler**, **zyklenfreier**, **zusammenhängender Teilgraph**), falls dessen Kosten minimal unter allen möglichen Spannbäumen von G sind.



Graph G_1



ein Spannbaum
Gesamtkosten = 22



ein *minimaler* Spannbaum
Gesamtkosten = 12

Minimaler Spannbaum III

Zwei Ansätze, einen minimalen Spannbaum zu ermitteln:

Grow-Algorithmus:

Beginne mit einer leeren Kantenmenge;

Solange noch kein Spannbaum gefunden ist:

→ füge der Kantenmenge die Kante mit minimalen Kosten aus G hinzu, die keinen Zyklus erzeugt.

Shrink-Algorithmus:

Beginne mit allen Kanten des Graphen als Kantenmenge;

Solange noch kein Spannbaum gefunden ist:

→ entferne aus der Kantenmenge die Kante mit maximalen Kosten, die nicht die Zusammenhangseigenschaft verletzt.

Minimaler Spannbaum IV

Der Algorithmus von Kruskal:

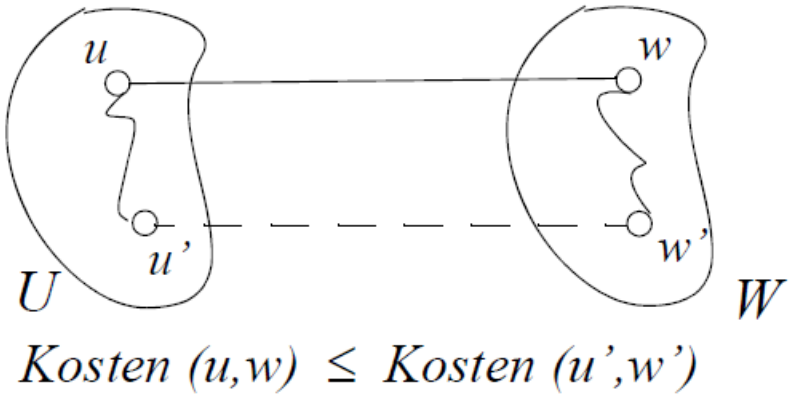
Grundlage: das ‚grow-Prinzip'.

Vorbemerkung: (ohne formalen Beweis)

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph und $\{U, W\}$ eine Zerlegung der Knotenmenge V.
Sei (u, w), $u \in U$ und $w \in W$, eine Kante in G mit minimalen Kosten unter allen Kanten $\{(u', w') | u' \in U, w' \in W\}$.
Dann gibt es einen minimalen Spannbaum für G, der (u, w) enthält.

Veranschaulichung:

(u',w') beliebig!

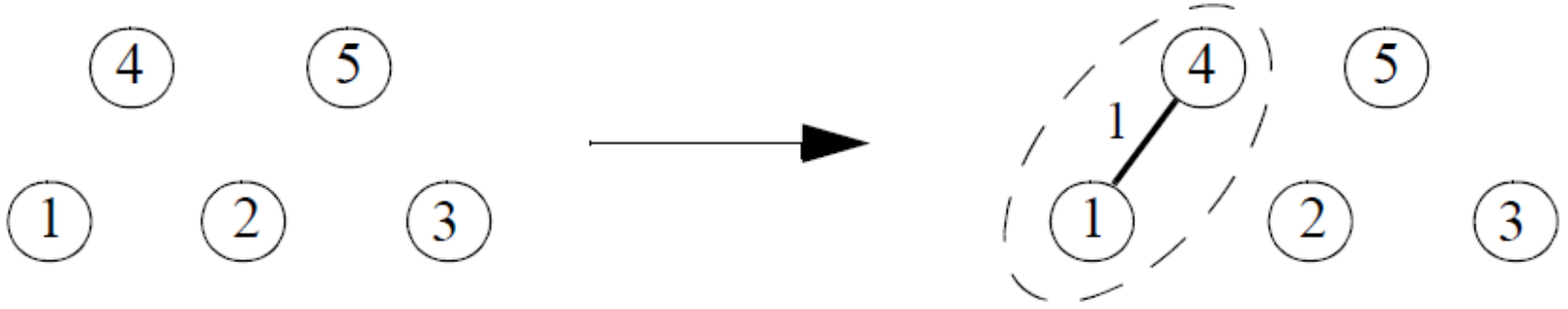


Minimaler Spannbaum V

Hieraus ergibt sich der Ansatz des Kruskal'schen Algorithmus: Schrittweise anwachsende Teilmengen der Knotenmenge V zu betrachten und deren minimale Verbindungskante zu ermitteln:

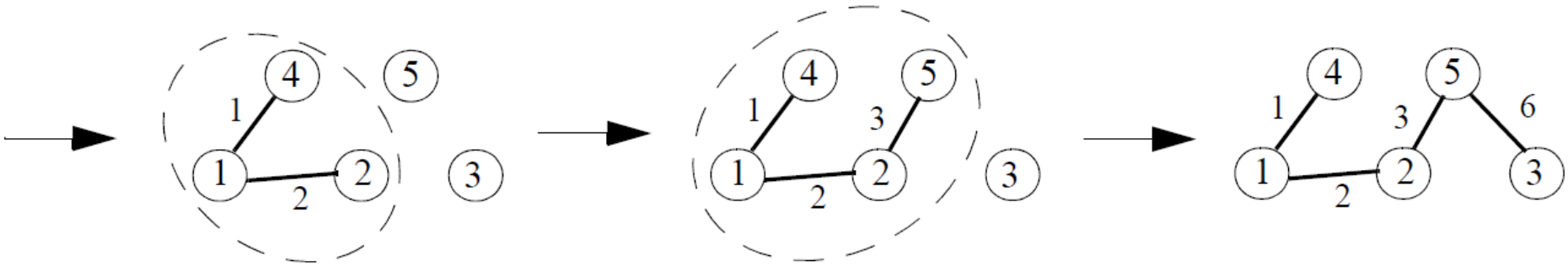
Prinzip:

- Jeder Knoten wird als einelementige Teilmenge von V betrachtet.
- Algorithmus betrachtet die Kanten in E nach aufsteigenden Kosten.
- Erster Schritt: Die beiden durch die kostenminimalen Kante verbundenen Knoten werden zu einer zweielementigen Teilmenge verschmolzen.



Minimaler Spannbaum VI

- Dieser Schritt wird $(n-1)$-mal iteriert, wobei jeweils diejenige Kante mit minimalen Kosten gewählt wird, die zwei bisher getrennte Knotenmengen verbindet.
- Diese beiden Mengen werden verschmolzen.
- Schließlich sind alle Knoten zu einer einzigen Knotenmenge vereint.
- Die schrittweise eingefügten Kanten bilden einen minimalen Spannbaum von G.



Minimaler Spannbaum VII

Informelle Beschreibung des Algorithmus:

- Kostenminimale Kanten können schnell gefunden werden.
- Dazu werden die Kanten nach aufsteigenden Kosten in einem dynamischen Heap ‚`K_Heap`' organisiert.
- Zugriff erfolgt mit Hilfe der Methode ‚`get_minimum`', wobei die kostenminimale Kante zurückgeliefert und gleichzeitig aus der Struktur gelöscht wird.

- Weitere benötigte Datenstrukturen:
  - ‚`Knoten_Sets`': organisiert die aktuellen Teilmengen der Knotenmenge.
    Methoden: ‚`find_component`' und ‚`merge_components`'.
  - ‚`Spann_Graph`': zweite Graphstruktur mit gleicher Knotenmenge wie G.
    Aufbau des minimalen Spannbaumes.
    Methode: ‚`insert_edge`'.

Minimaler Spannbaum VIII


```
ALGORITHMUS Kruskal;
   VAR Schrittzahl : CARDINAL;
       v, w : Knoten;
       a, b : Component; // abstrakter Datentyp für 'Knoten_Sets'
BEGIN
   Initialisierung Knoten_Sets → jeder Knoten bildet eine eigene Komponente;
   füge alle Kanten aus E gemäß ihrer Kosten in K_Heap ein;
   initialisiere die Kantenmenge von Spann_Graph als leer;
   Schrittzahl := 1;
WHILE Schrittzahl < n DO
   get_minimum(K_Heap, v, w);
   a := find_component(Knoten_Sets, v);
   b := find_component(Knoten_Sets, w);
   IF a ≠ b THEN // andernfalls wird die Kante einfach übergangen
      insert_edge(Spann_Graph, v, w); // Aufbau minimaler Spannbaum
      merge_components(Knoten_Sets, a, b);
      INC(Schrittzahl);
END
END
END Kruskal;
```

Minimaler Spannbaum VIX

Laufzeitanalyse:
Sei $n = |V|$ und $e = |E|$, dann besitzt vorheriger Algorithmus das folgende Laufzeitverhalten:

- Initialisierung von ‚`Spann_Graph`' und ‚`Knoten_Sets`': $O(n)$
- Initialisierung von ‚`K_Heap`': $O(e)$
- Maximal e Operationen ‚`get_minimum`': $O(e \cdot \log e)$
- Maximal $2e$ Operationen ‚`find_component`': $O(e \cdot \log e)$
- $n - 1$ Operationen ‚merge_components': $O(e \cdot \log e)$

→ Gesamtlaufzeit: $O(e \cdot \log e)$,
da $e \geq n - 1$ (G ist zusammenhängend).