Vorlesungstermin 12:

Minimale Spannbäume

Markus Püschel
David Steurer

talks2.dsteurer.org/mst.pdf

Algorithmen und Datenstrukturen, Herbstsemester 2018, ETH Zürich

Plan für heute

- minimiale Spannbäume mit gierigen Algorithmen
- effiziente Implementierung mit Hilfe von `Union-Find` Datenstruktur (neues Konzept: amortisierte Analyse)

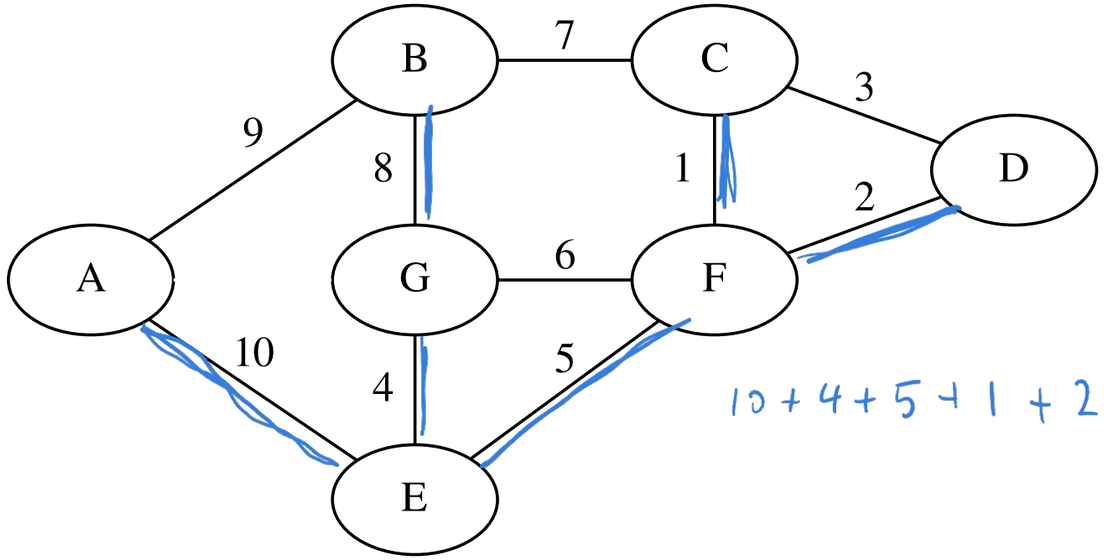
Plan für heute

- minimiale Spannbäume mit gierigen Algorithmen
- effiziente Implementierung mit Hilfe von `Union-Find` Datenstruktur (neues Konzept: amortisierte Analyse)

Unterrichtsbeurteilung: bitte Fragebogen ausfüllen
(siehe E-Mail)

minimale Spannbäume

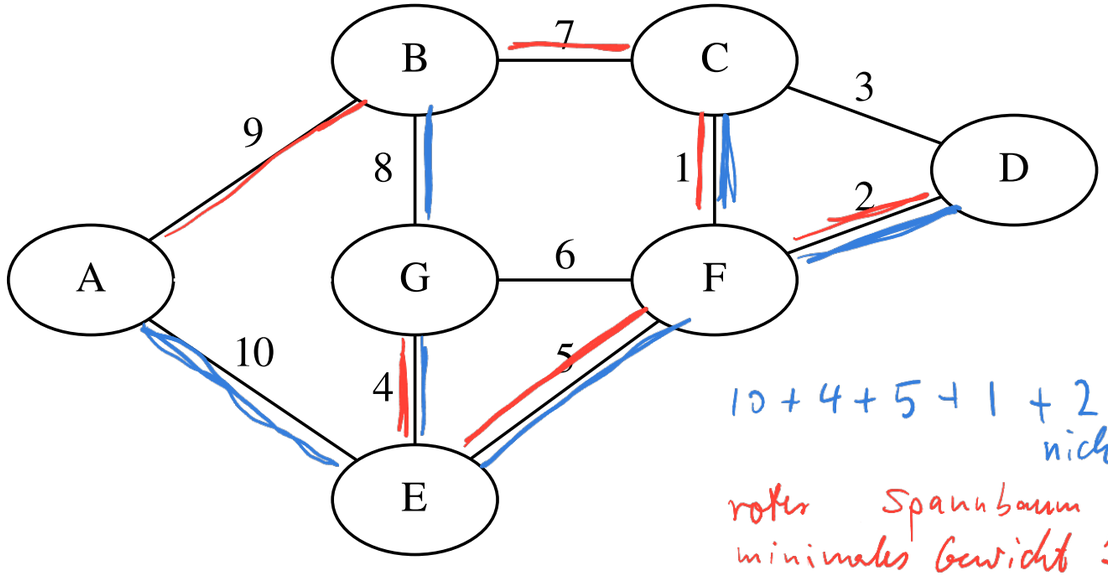
gegeben: unger. Graph $G = (V, E)$, Gewichte $w\colon E \to \mathbb{R}$



Definition: Teilgraph T von G ist ein *Spannbaum*, falls T ein Baum ist und alle Knoten von G enthält

minimale Spannbäume

gegeben: unger. Graph $G = (V, E)$, Gewichte $w\colon E \to \mathbb{R}$



10 + 4 + 5 + 1 + 2 + 8 = 30
nicht minimal

roter Spannbaum hat minimales Gewicht 1 + 2 + 4 + 5 + 7 + 9 = 28

Definition: Teilgraph T von G ist ein *Spannbaum*, falls T ein Baum ist und alle Knoten von G enthält

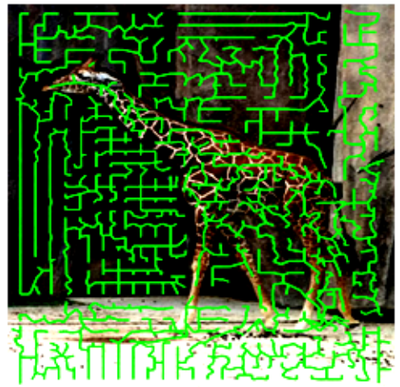
gesucht: Spannbaum mit *minimalem Gewicht*

Anwendungsbeispiele

- vernetze n Computer eines globalen Unternehmens mit *minimalen Kosten* (allgemeiner Begriff: Netzwerkdesign)

Anwendungsbeispiele

- vernetze n Computer eines globalen Unternehmens mit *minimalen Kosten* (allgemeiner Begriff: Netzwerkdesign)
- Bildsegmentierung (image segmentation)



Anwendungsbeispiele

- vernetze n Computer eines globalen Unternehmens mit *minimalen Kosten* (allgemeiner Begriff: Netzwerkdesign)
- Bildsegmentierung (image segmentation)
- Clustering (z.B. für genetische Distanz)

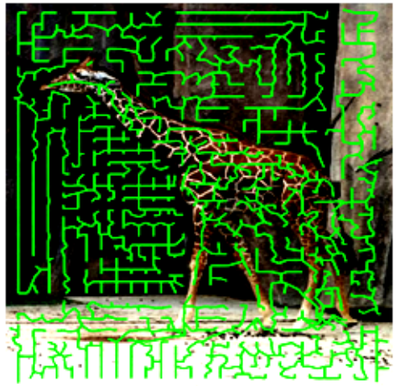
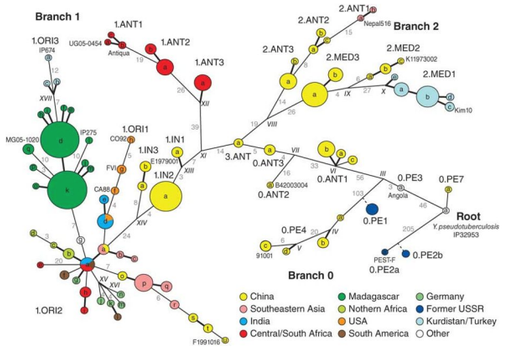


Figure 2 : Fully parsimonious minimal spanning tree of 933 SNPs for 282 isolates of Y. *pestis* colored by location.

From: Yersinia pestis genome sequencing identifies patterns of global phylogenetic diversity



Large, bold text, branches 1, 2 and 0; smaller letters, populations (for example, 1.ORI3); lower case letters, nodes (for example, 1.ORI3.a). Strain designations near terminal nodes, genomic sequences. Roman numbers, hypothetical nodes. Gray text on lines between nodes, numbers of SNPs, except that one or two SNPs are indicated by thick and thin black lines, respectively. Six additional isolates in 0.PE1 and 0.PE2b (blue dashes) were tested only for selected, informative SNPs.

Anwendungsbeispiele

- vernetze n Computer eines globalen Unternehmens mit *minimalen Kosten* (allgemeiner Begriff: Netzwerkdesign)
- Bildsegmentierung (image segmentation)
- Clustering (z.B. für genetische Distanz)
- Bestandteil anderer Algorithmen

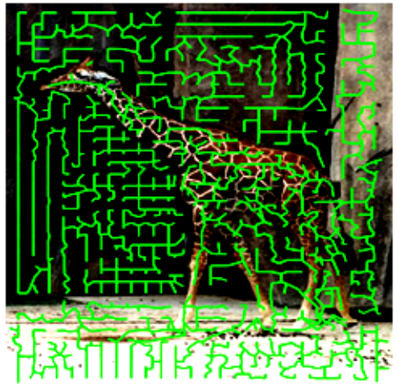
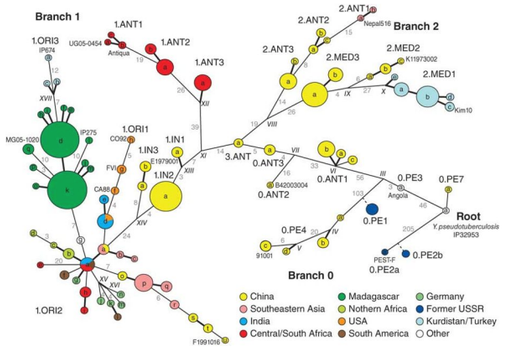


Figure 2 : Fully parsimonious minimal spanning tree of 933 SNPs for 282 isolates of Y. *pestis* colored by location.

From: Yersinia pestis genome sequencing identifies patterns of global phylogenetic diversity



Large, bold text, branches 1, 2 and 0; smaller letters, populations (for example, 1.ORI3); lower case letters, nodes (for example, 1.ORI3.a). Strain designations near terminal nodes, genomic sequences. Roman numbers, hypothetical nodes. Gray text on lines between nodes, numbers of SNPs, except that one or two SNPs are indicated by thick and thin black lines, respectively. Six additional isolates in 0.PE1 and 0.PE2b (blue dashes) were tested only for selected, informative SNPs.

gegeben: unger. Graph $G = (V, E)$, Gewichte $w \colon E \to \mathbb{R}$

gesucht: Spannbaum mit minimalem Gewicht

gegeben: unger. Graph $G = (V, E)$, Gewichte $w\colon E \to \mathbb{R}$

gesucht: Spannbaum mit minimalem Gewicht

sei $n = |V|$ und $m = |E|$

kein Kreis <=> Kanten bilden Wald
Wald mit k Komponent
hat n-k Kanten
=> Wald m.t n-1 Kanten ist Baum

drei äquivalente Formulierungen:

- wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gegeben: unger. Graph $G = (V, E)$, Gewichte $w \colon E \to \mathbb{R}$

gesucht: Spannbaum mit minimalem Gewicht

sei $n = |V|$ und $m = |E|$

kein Kreis $\Leftrightarrow$ Kanten bilden Wald
Wald mit k Komponent
hat n-k Kanten
$\Rightarrow$ Wald m.t n-1 Kanten ist Baum

drei äquivalente Formulierungen:

- wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*
- wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gegeben: unger. Graph $G = (V, E)$, Gewichte $w\colon E \to \mathbb{R}$

gesucht: Spannbaum mit minimalem Gewicht

sei $n = |V|$ und $m = |E|$

kein Kreis <=> Kanten bilden Wald
Wald mit k Komponent
hat n-k Kanten
=> Wald m.t n-1 Kanten ist Baum

drei äquivalente Formulierungen:

- wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*
- wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*
- entferne $m - n + 1$ Kanten mit maximalem Gewicht so, dass *alle Knoten zusammenhängend bleiben*

gegeben: unger. Graph $G = (V, E)$, Gewichte $w\colon E \to \mathbb{R}$

gesucht: Spannbaum mit minimalem Gewicht

sei $n = |V|$ und $m = |E|$

kein Kreis <=> Kanten bilden Wald
Wald mit k Komponent
hat n-k Kanten
=> Wald m.t n-1 Kanten ist Baum

drei äquivalente Formulierungen:

- wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*
- wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*
- entferne $m - n + 1$ Kanten mit maximalem Gewicht so, dass *alle Knoten zusammenhängend bleiben*

später: *jede dieser Formulierungen entspricht einem "gierigen Algorithmus" für minimale Spannbäume*

gierige Algorithmen

gierige Algorithmen

- berechne Lösung als Sequenz von Entscheidung
  Beispiel: Entscheidung ob Kante zum minimalen Spannbaum gehört oder nicht

gierige Algorithmen

- berechne Lösung als Sequenz von Entscheidung
  Beispiel: Entscheidung ob Kante zum minimalen Spannbaum gehört oder nicht
- treffe jede Entscheidung *"best-möglich"* im Hinblick auf vorher getroffene Entscheidungen (ignoriere die Zukunft!)

gierige Algorithmen

- berechne Lösung als Sequenz von Entscheidung
  Beispiel: Entscheidung ob Kante zum minimalen Spannbaum gehört oder nicht
- treffe jede Entscheidung *"best-möglich"* im Hinblick auf vorher getroffene Entscheidungen (ignoriere die Zukunft!)
- *ändere niemals schon getroffene Entscheidungen!*

Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus:

Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

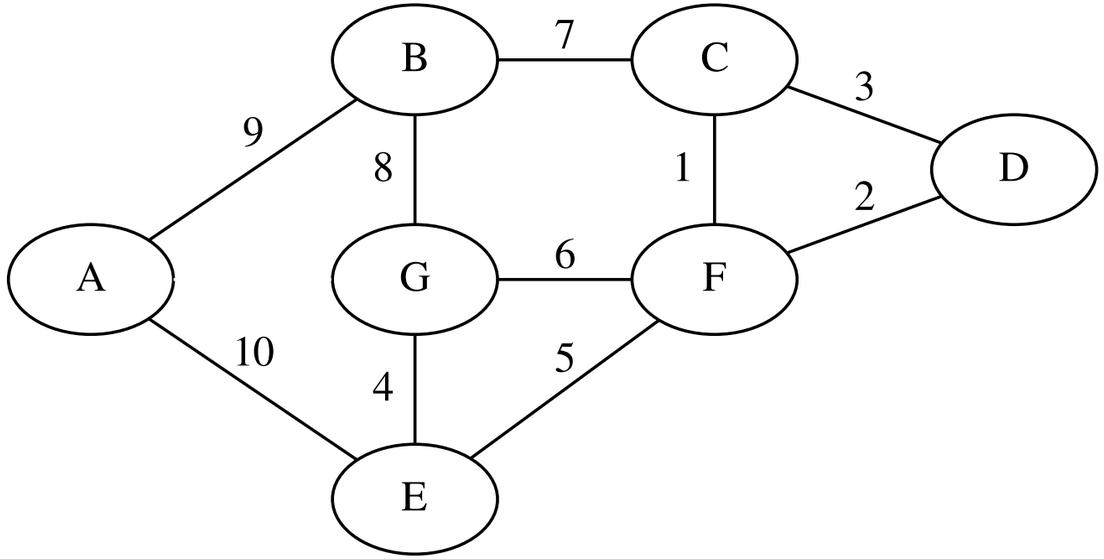
gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*

Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*

Iteration: 0

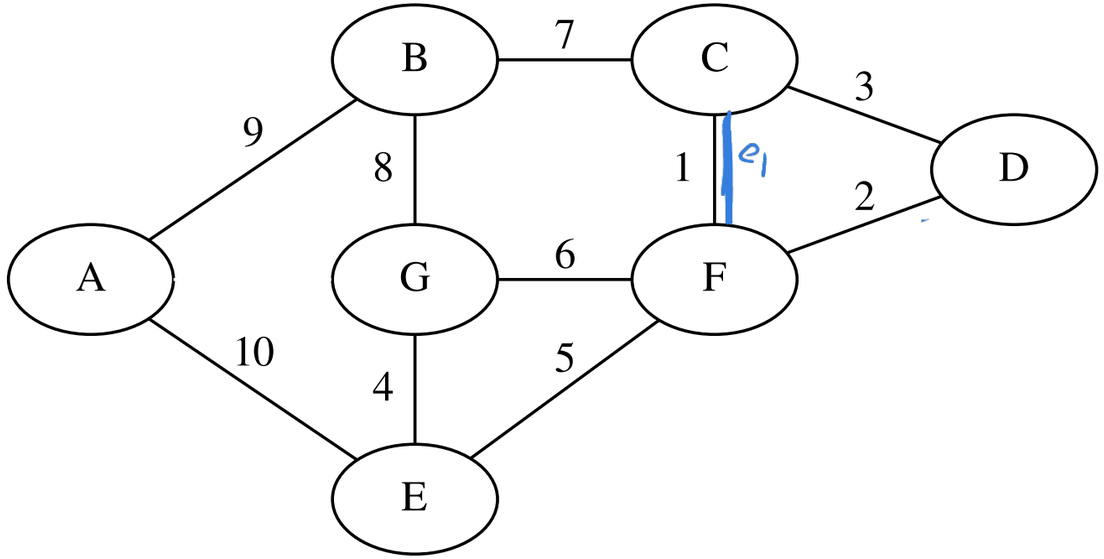


Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*

Iteration: 1

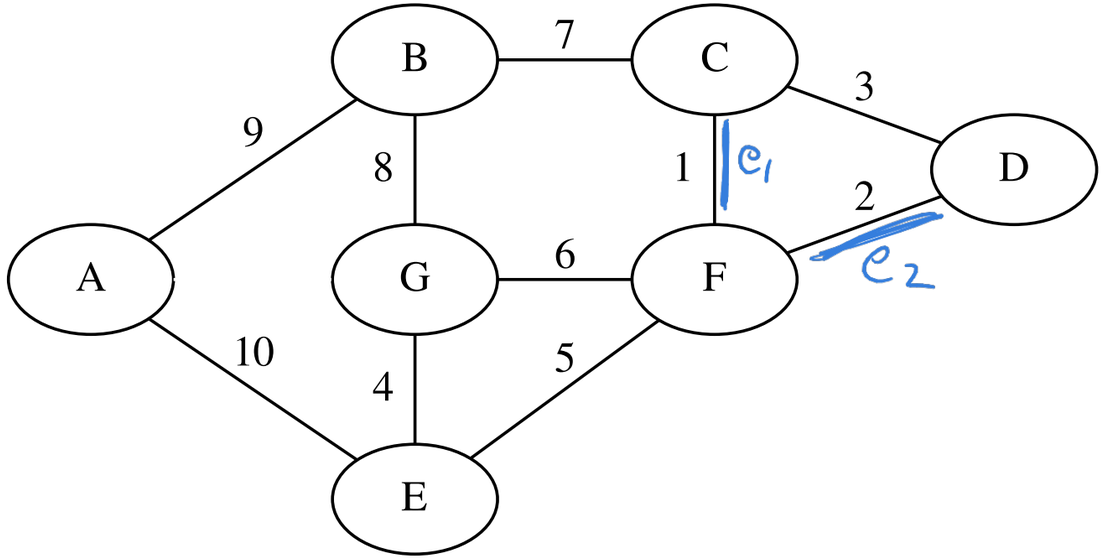


Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*

Iteration: 2

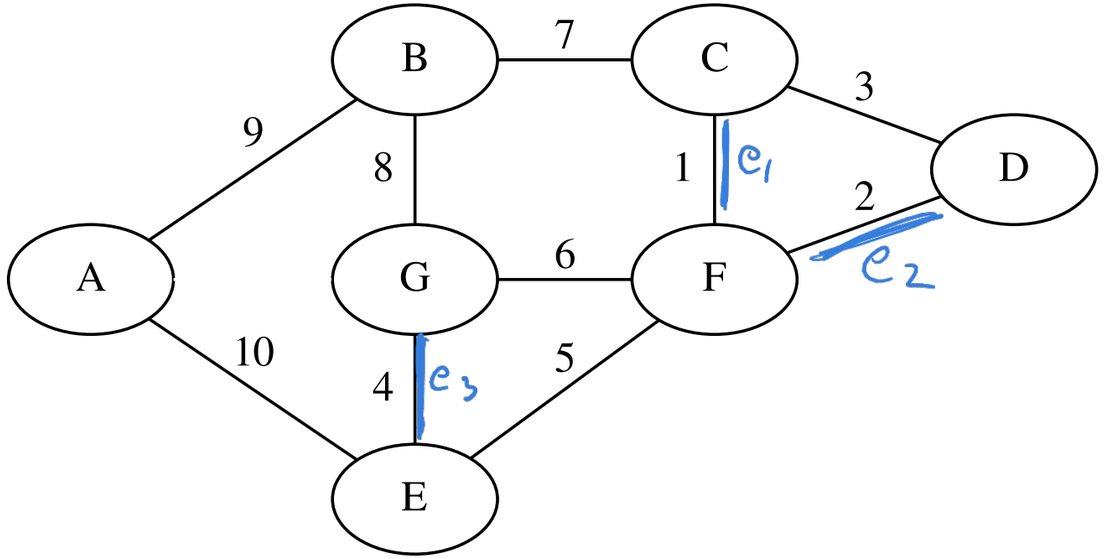


Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*

Iteration: 3

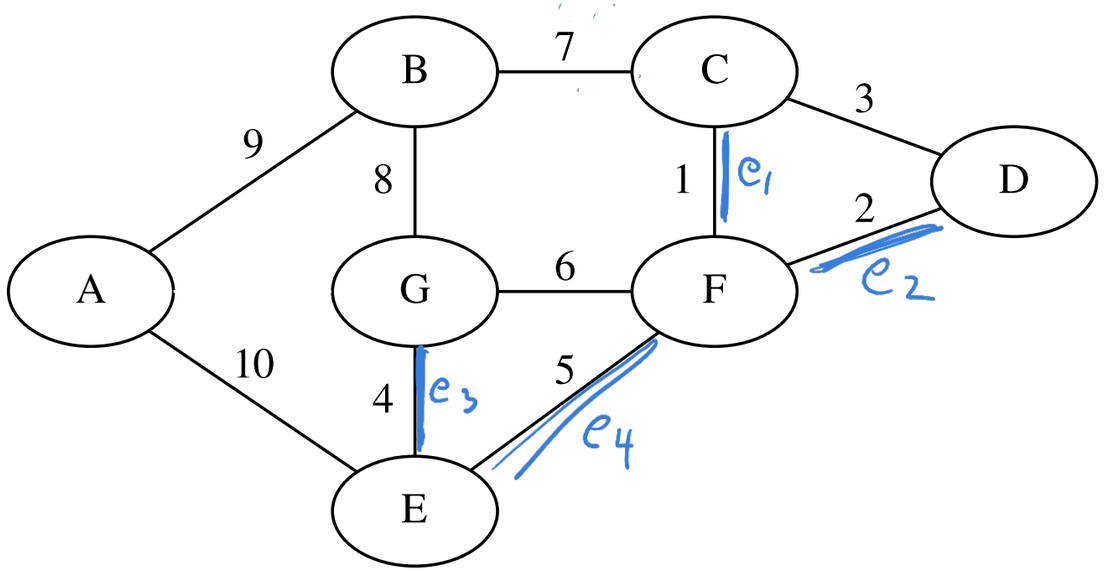


Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*

Iteration: 4

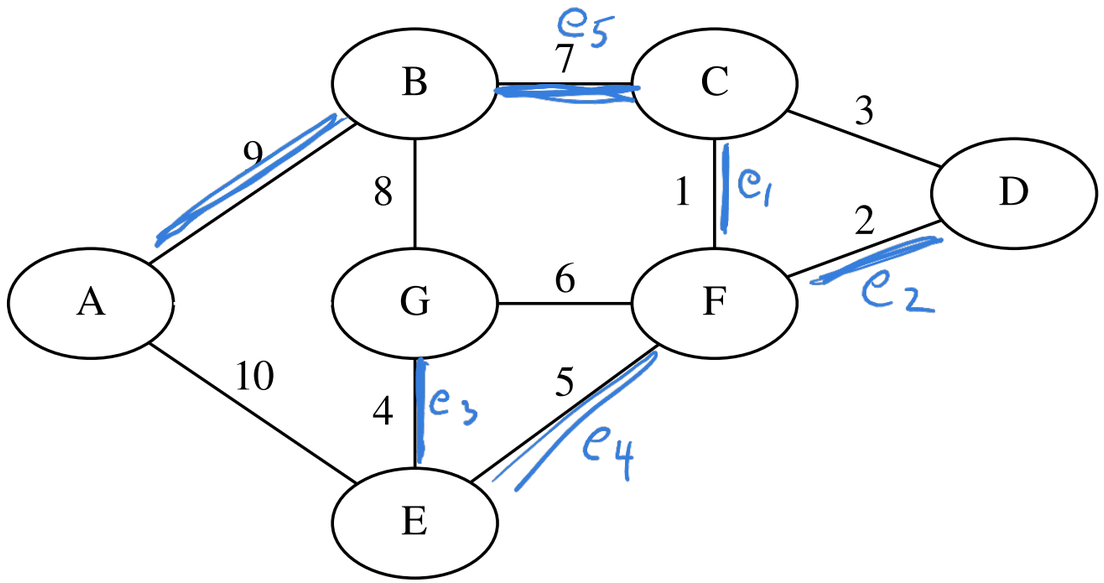


Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*

Iteration: 5

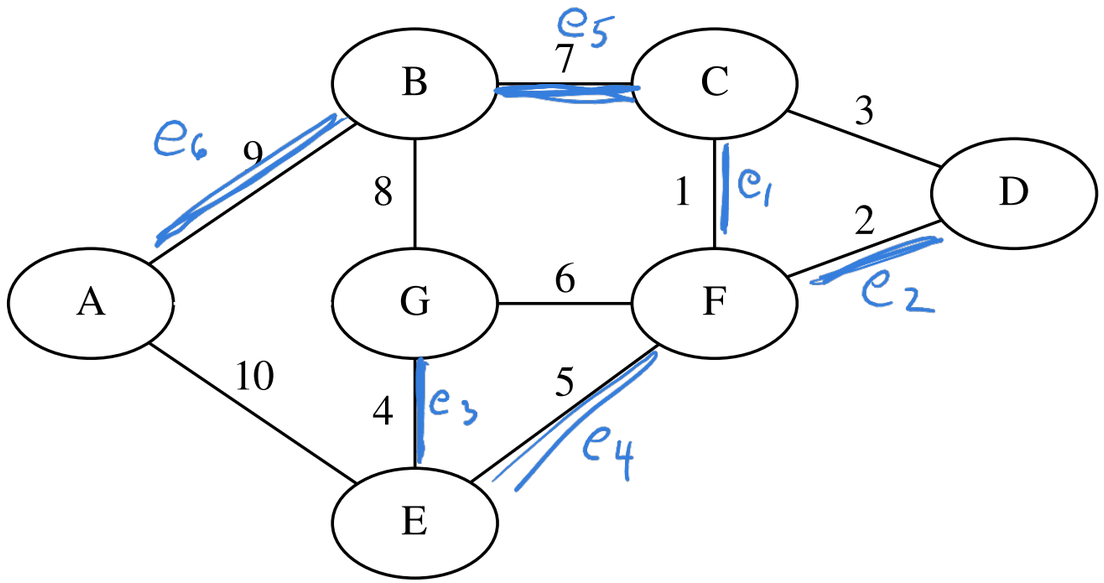


Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*

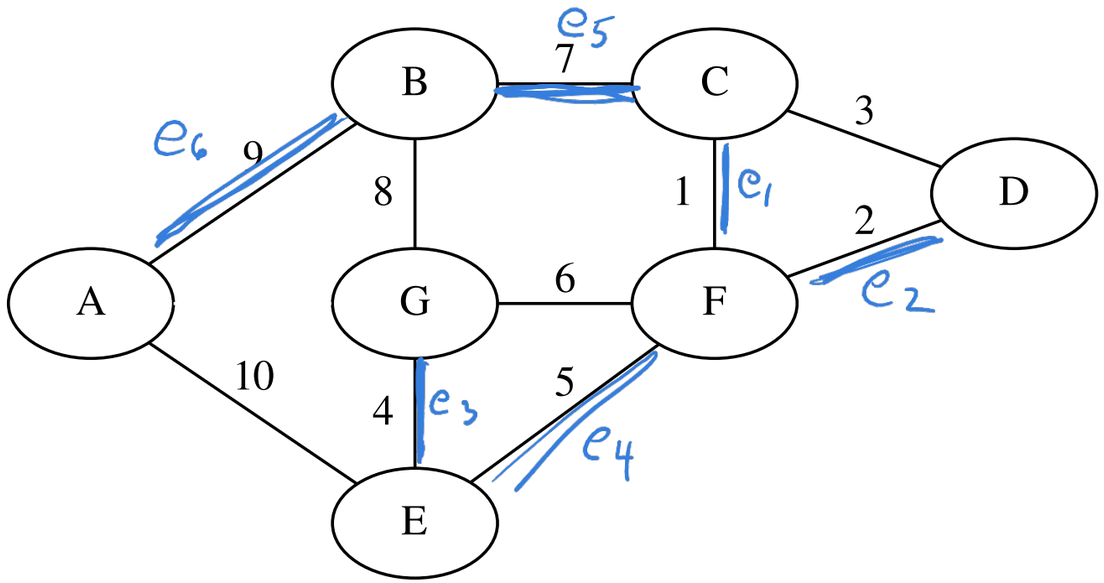
Iteration: 6



Kruskals Algorithmus

Problemformulierung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *kein Kreis entsteht*

gieriger Algorithmus: beginne mit leerer Kantenmenge; füge immer die *günstigste Kante hinzu, die keinen Kreis verursacht*



Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gieriger Algorithmus:

Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

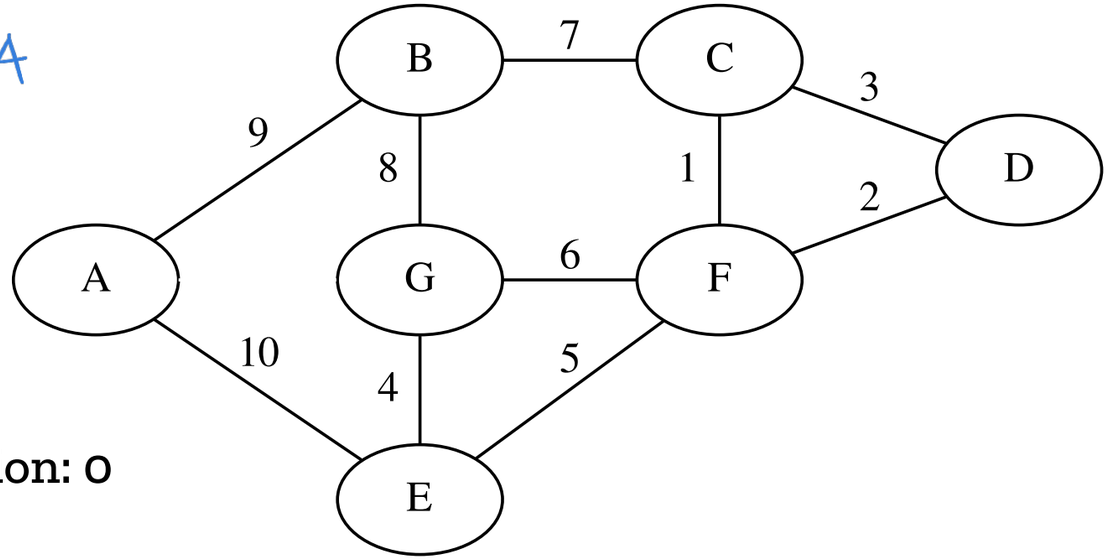
gieriger Algorithmus: beginne mit beliebigen Startknoten s; füge immer die *günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt*

Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gieriger Algorithmus: beginne mit beliebigen Startknoten s; füge immer die *günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt*

$s = A$

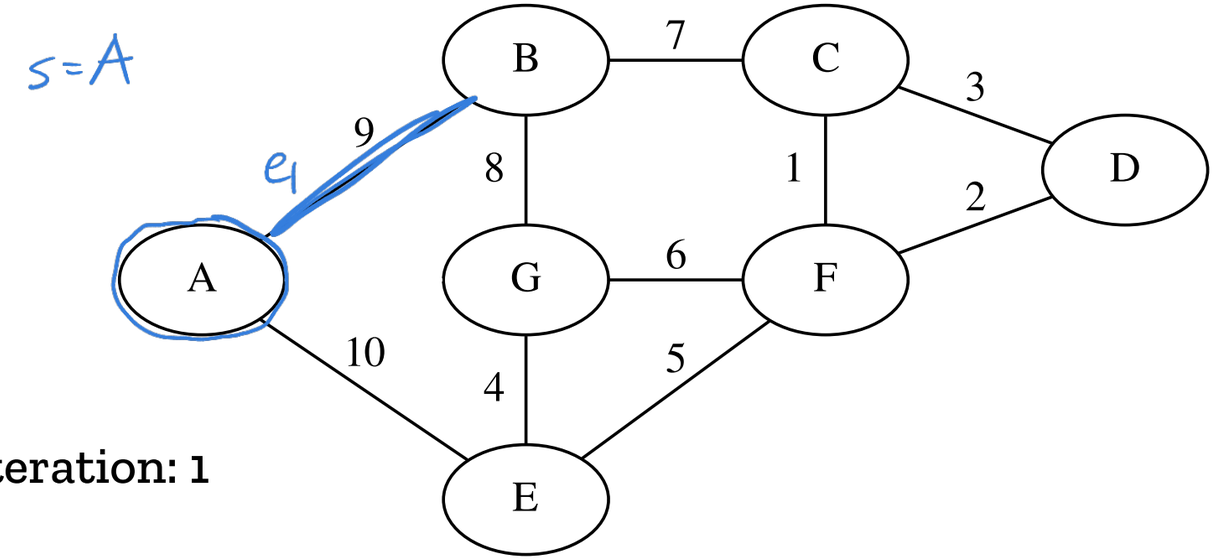


Iteration: 0

Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gieriger Algorithmus: beginne mit beliebigen Startknoten s; füge immer die *günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt*

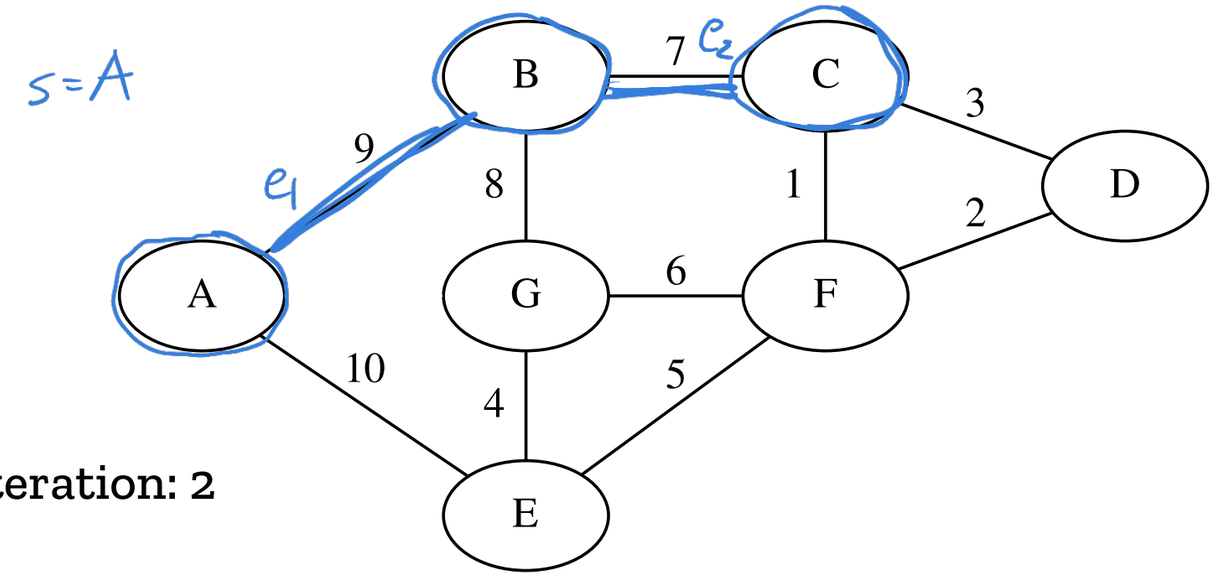


Iteration: 1

Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gieriger Algorithmus: beginne mit beliebigen Startknoten s; füge immer die *günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt*

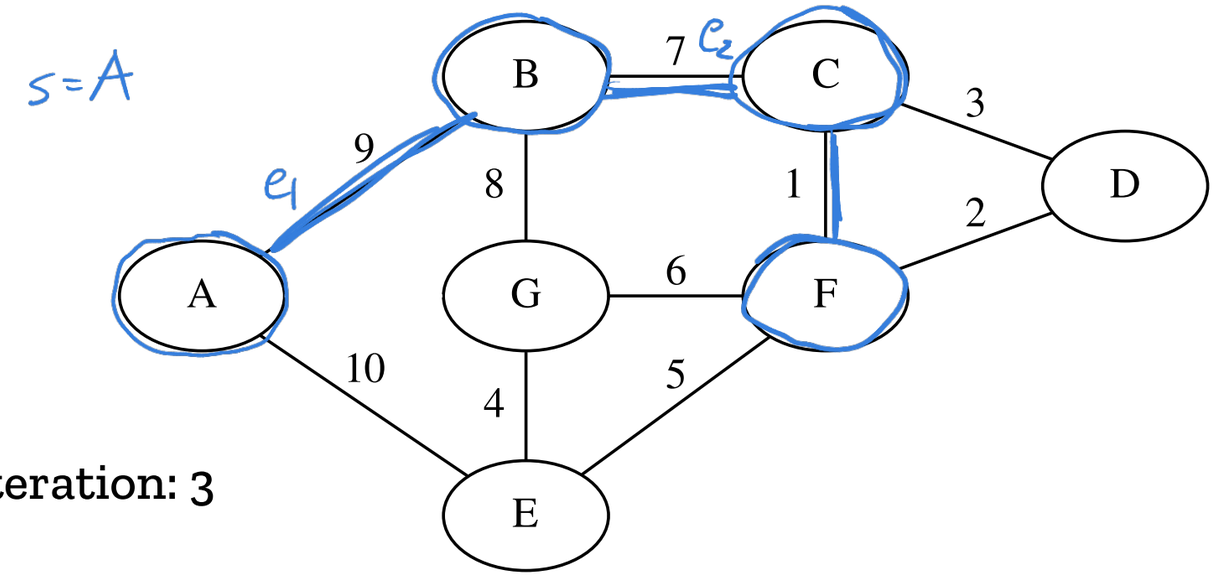


Iteration: 2

Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gieriger Algorithmus: beginne mit beliebigen Startknoten s; füge immer die *günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt*

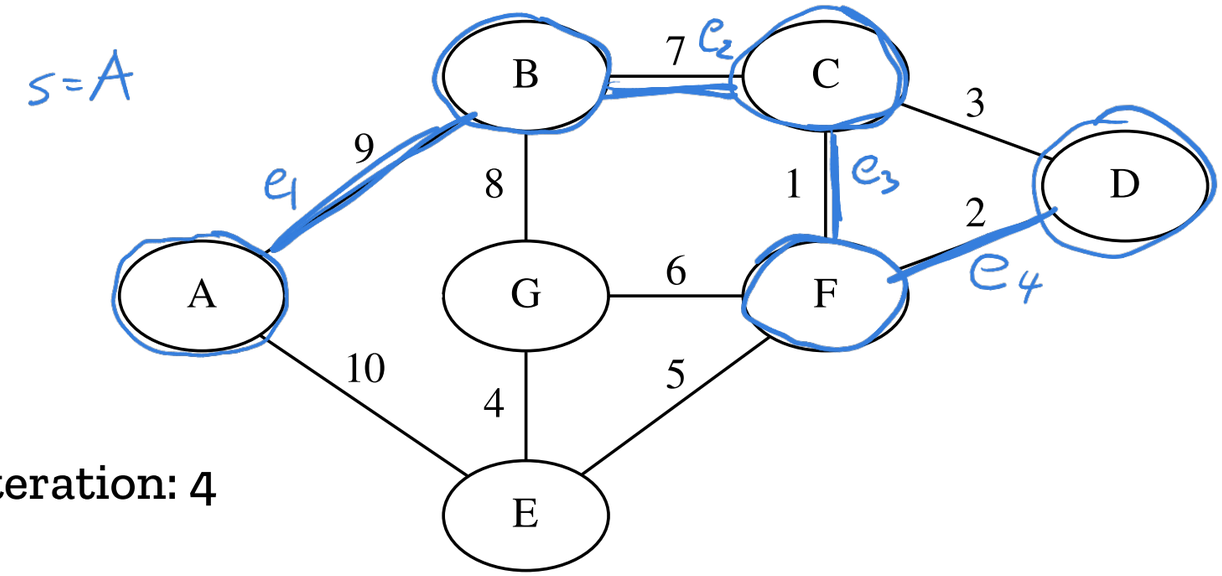


Iteration: 3

Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gieriger Algorithmus: beginne mit beliebigen Startknoten s; füge immer die *günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt*

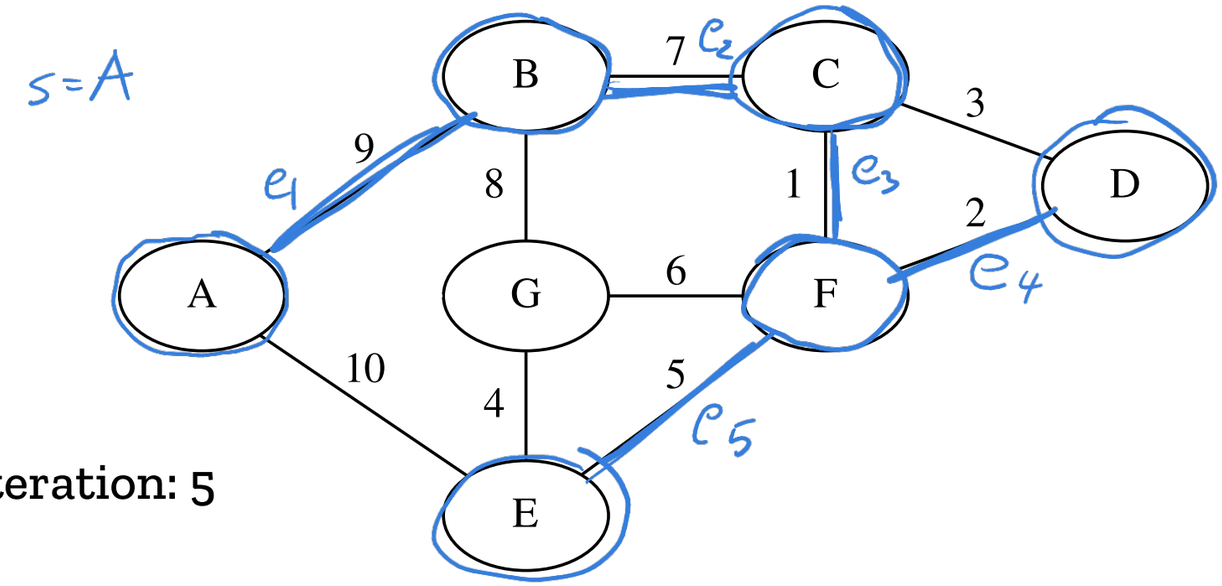


Iteration: 4

Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gieriger Algorithmus: beginne mit beliebigen Startknoten s; füge immer die *günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt*

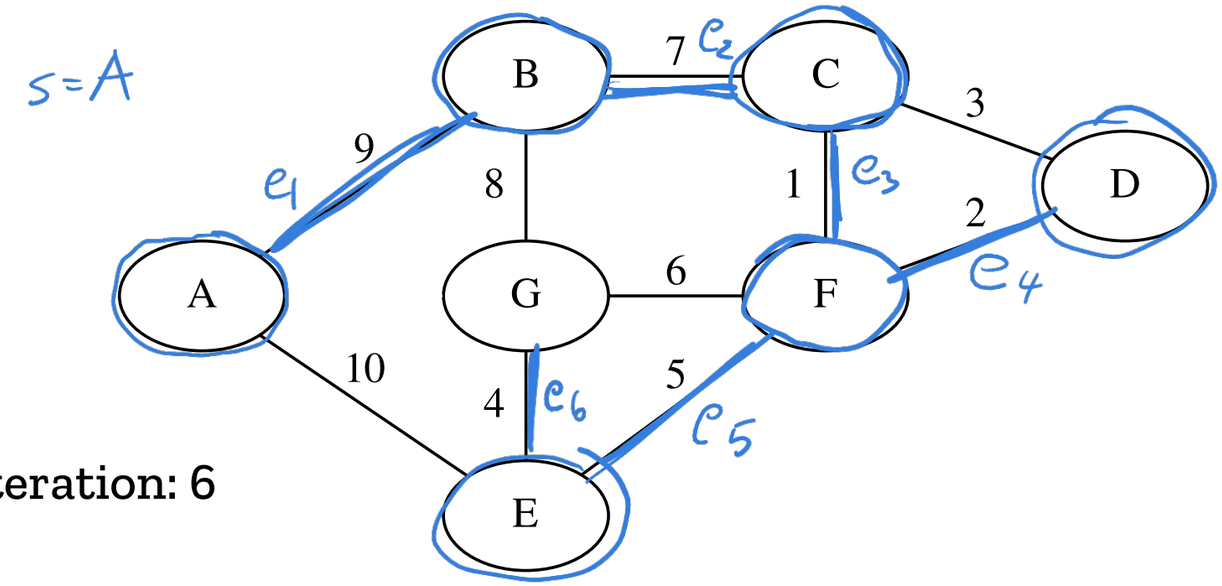


Iteration: 5

Prims Algorithmus

Problemstellung: wähle $n - 1$ Kanten mit minimalen Gewicht so aus, dass *alle Knoten zusammenhängend werden*

gieriger Algorithmus: beginne mit beliebigen Startknoten s; füge immer die *günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt*



Iteration: 6

"Rückwärts-Kruskal" Algorithmus

Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus:

"Rückwärts-Kruskal" Algorithmus

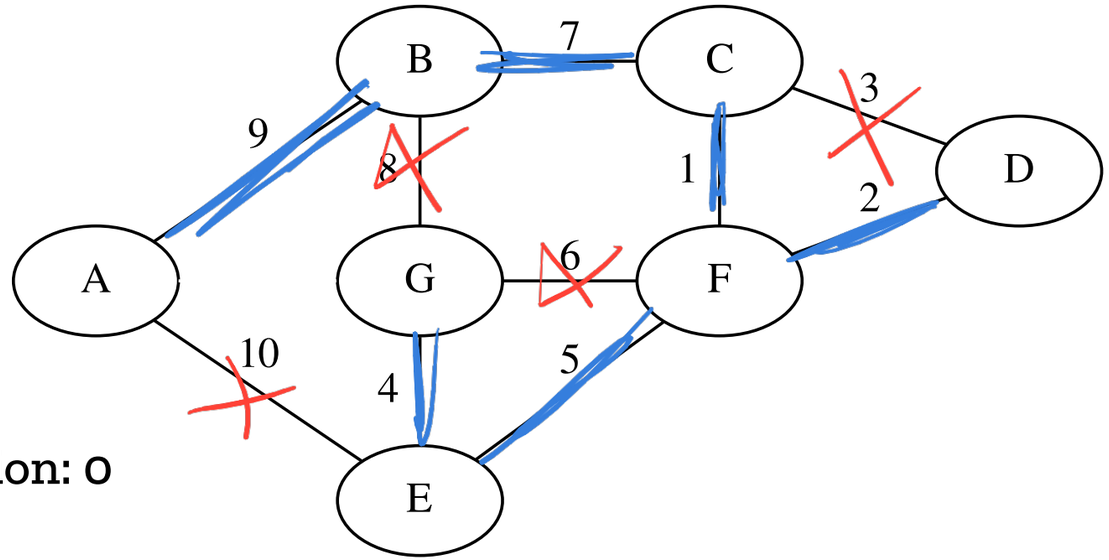
Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus: beginne mit voller Kantenmenge; entferne immer die *teuerste Kante, so dass alle Knoten zusammenhängend bleiben*

"Rückwärts-Kruskal" Algorithmus

Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus: beginne mit voller Kantenmenge; entferne immer die *teuerste Kante, so dass alle Knoten zusammenhängend bleiben*

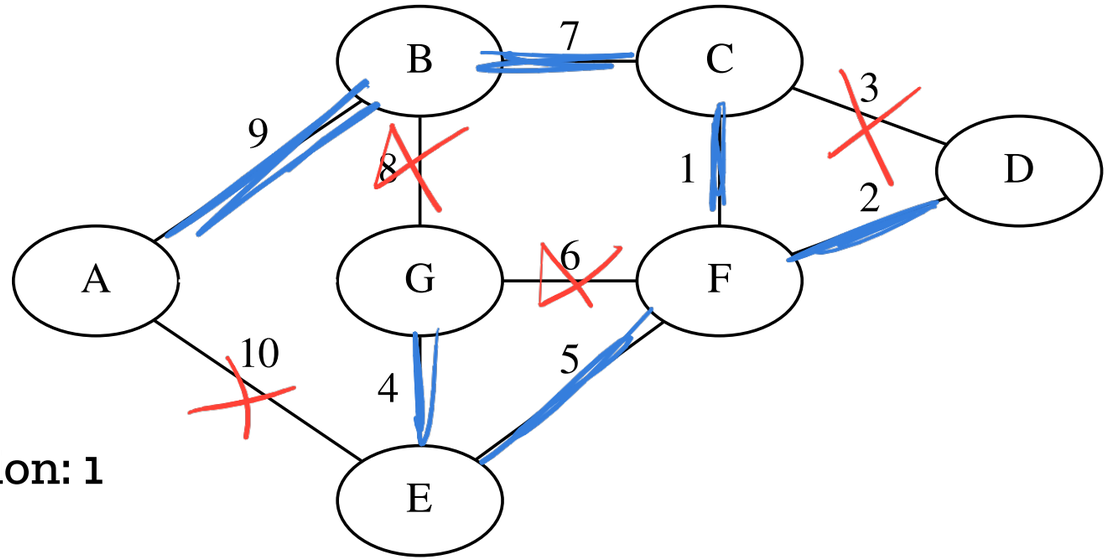


Iteration: 0

"Rückwärts-Kruskal" Algorithmus

Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus: beginne mit voller Kantenmenge; entferne immer die *teuerste Kante, so dass alle Knoten zusammenhängend bleiben*

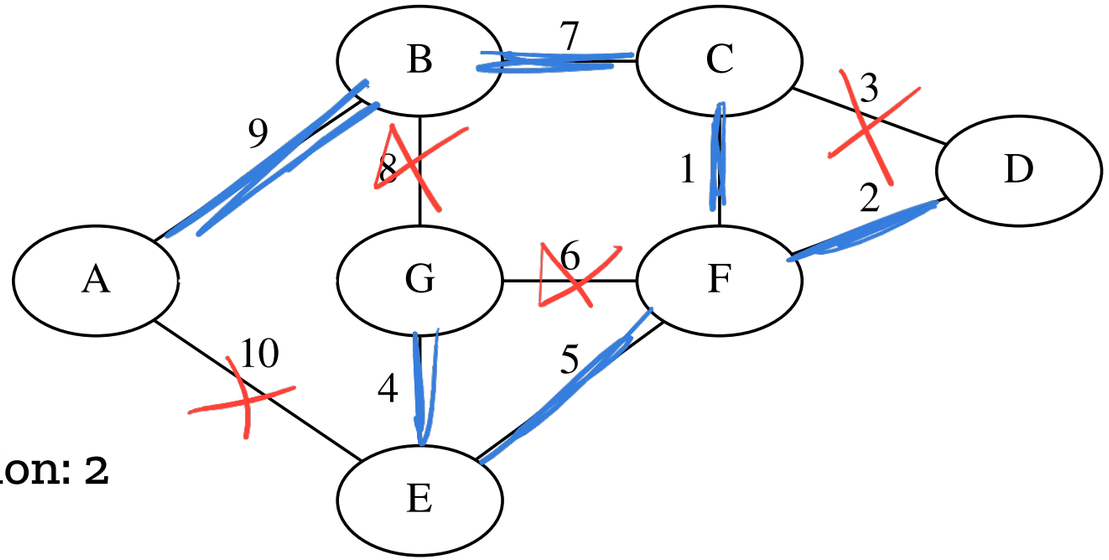


Iteration: 1

"Rückwärts-Kruskal" Algorithmus

Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus: beginne mit voller Kantenmenge; entferne immer die *teuerste Kante, so dass alle Knoten zusammenhängend bleiben*

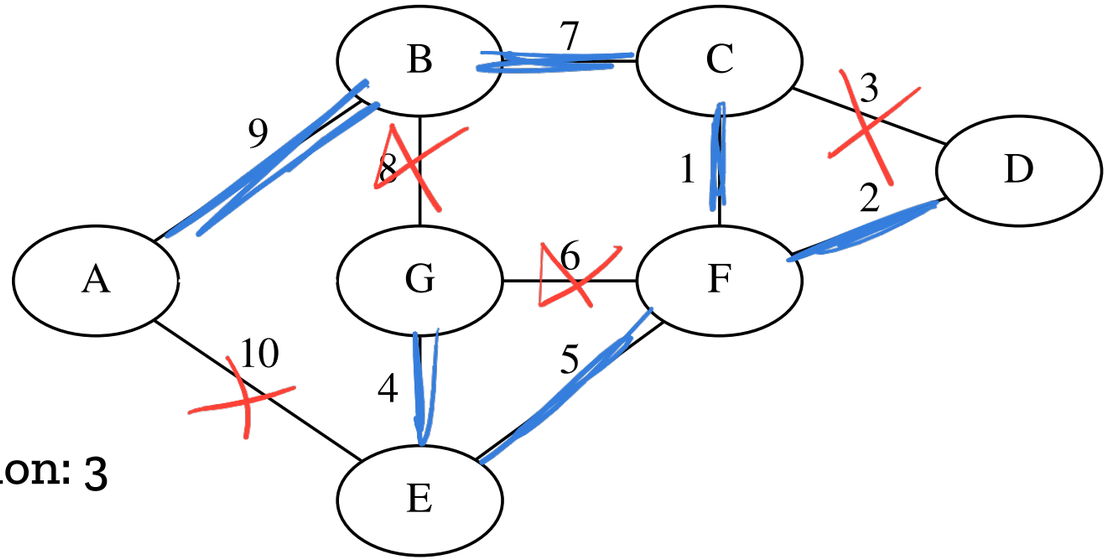


Iteration: 2

"Rückwärts-Kruskal" Algorithmus

Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus: beginne mit voller Kantenmenge; entferne immer die *teuerste Kante, so dass alle Knoten zusammenhängend bleiben*

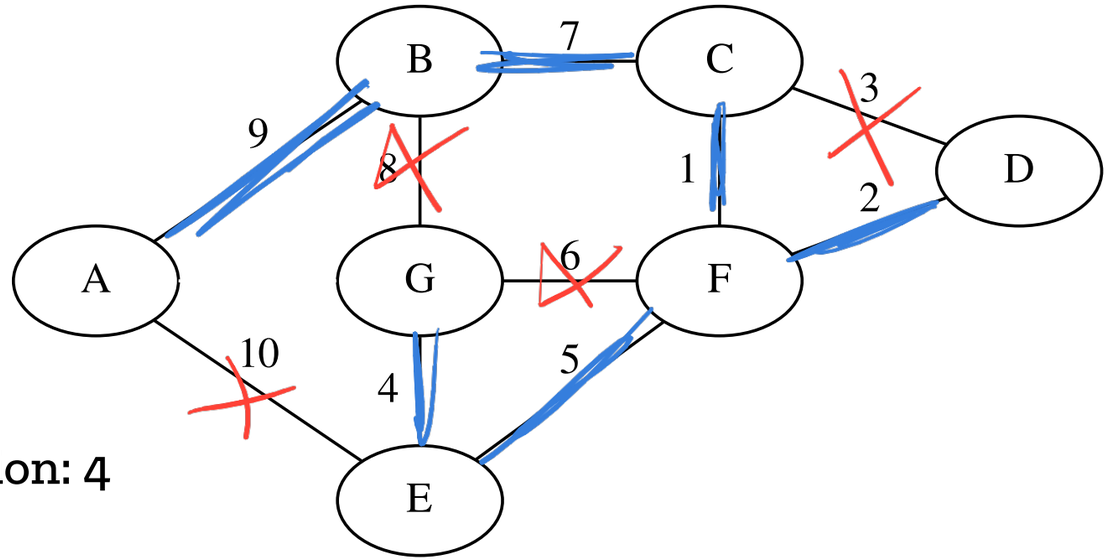


Iteration: 3

"Rückwärts-Kruskal" Algorithmus

Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus: beginne mit voller Kantenmenge; entferne immer die *teuerste Kante, so dass alle Knoten zusammenhängend bleiben*

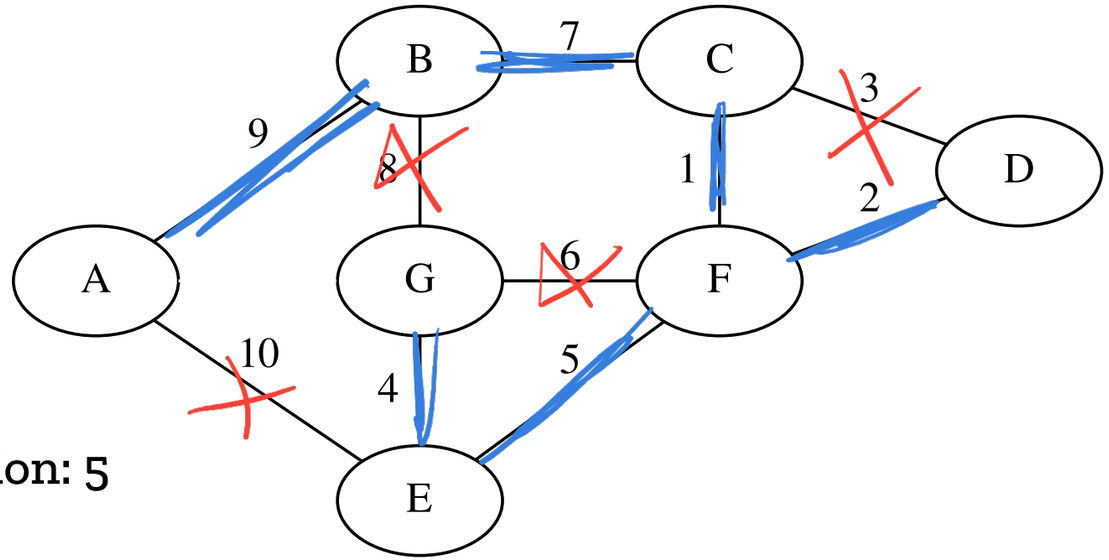


Iteration: 4

"Rückwärts-Kruskal" Algorithmus

Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus: beginne mit voller Kantenmenge; entferne immer die *teuerste Kante, so dass alle Knoten zusammenhängend bleiben*

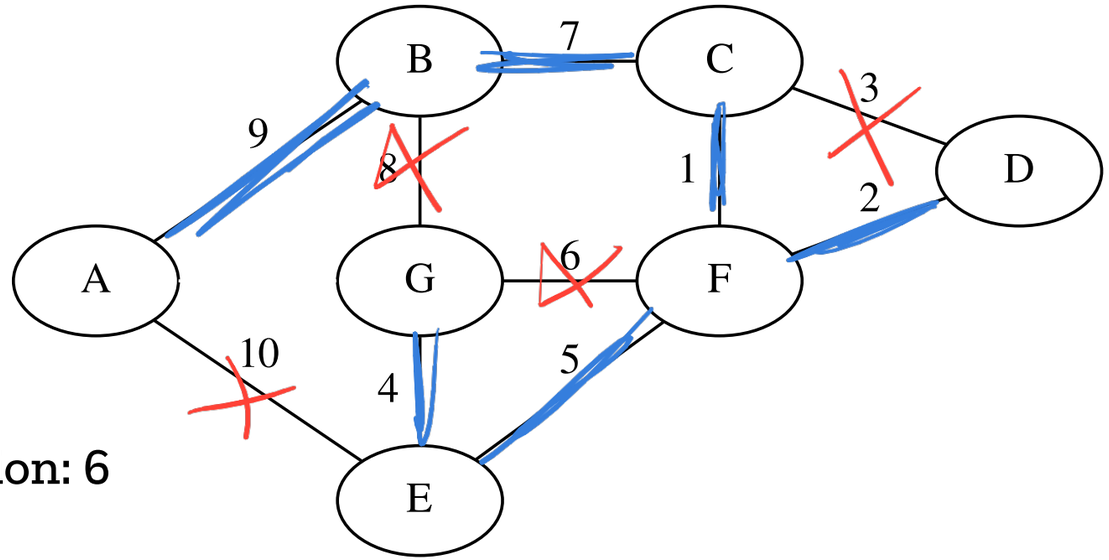


Iteration: 5

"Rückwärts-Kruskal" Algorithmus

Problemstellung: entferne $m - n + 1$ Kanten mit max. Gewicht so, dass alle Knoten *zusammenhängend bleiben*

gieriger Algorithmus: beginne mit voller Kantenmenge; entferne immer die *teuerste Kante, so dass alle Knoten zusammenhängend bleiben*



Iteration: 6

Korrektheit

Korrektheit

alle drei gierigen Algorithmen sind korrekt, finden also immer einen minimalen Spannbaum

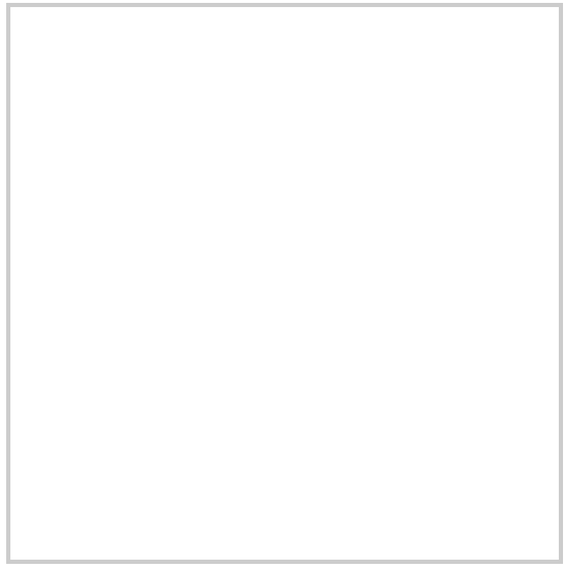
Korrektheit

alle drei gierigen Algorithmen sind korrekt, finden also immer einen minimalen Spannbaum

zwei allgemeine Prinzipien erlauben uns die Korrektheit dieser Algorithmen zu zeigen

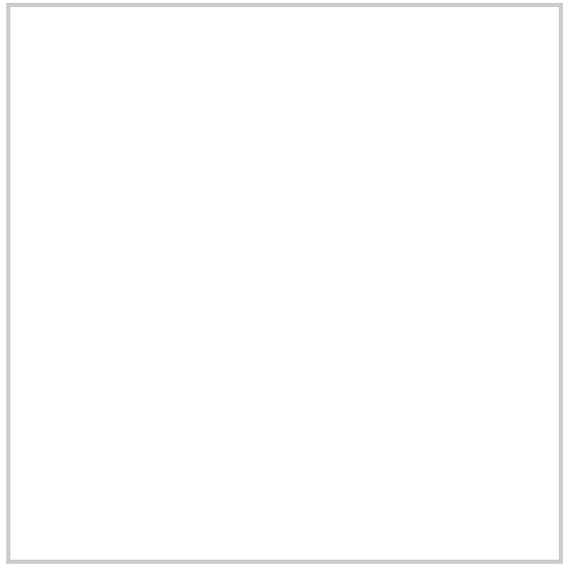
Annahme: alle Kantengewichte sind verschieden (wie zuvor)

Kreisprinzip: (cycle property)
betrachte Kreis C im Graphen;



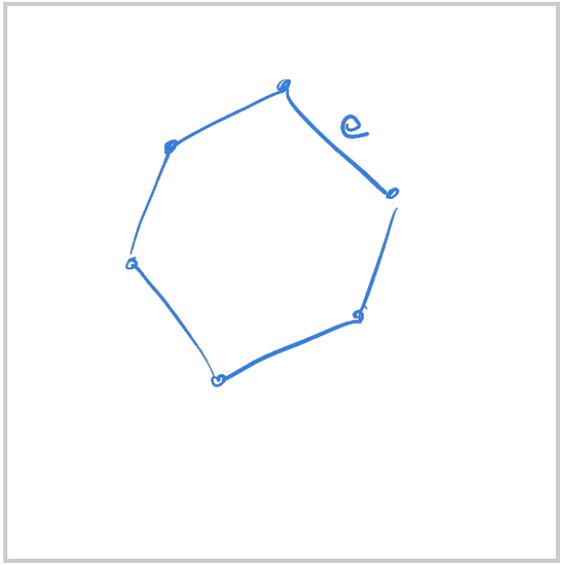
Annahme: alle Kantengewichte sind verschieden (wie zuvor)

Kreisprinzip: (cycle property)
betrachte Kreis C im Graphen;
sei e die Kante mit maximalem Gewicht in C;



Annahme: alle Kantengewichte sind verschieden (wie zuvor)

Kreisprinzip: (cycle property)
betrachte Kreis C im Graphen;
sei e die Kante mit maximalem Gewicht in C;
dann enthält *kein minimaler Spannbaum* die Kante e

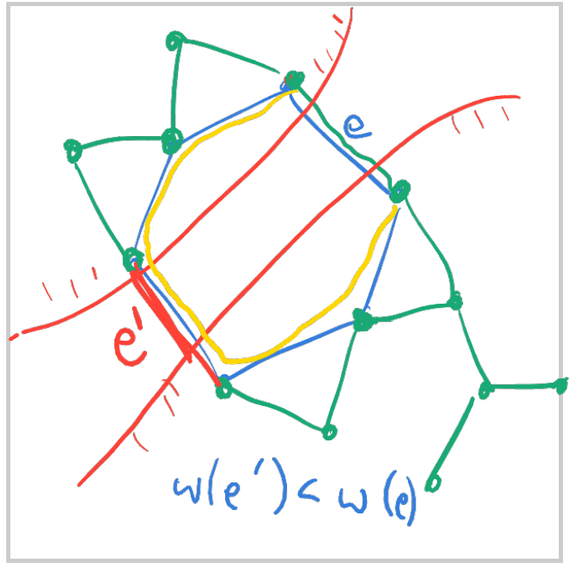


Annahme: alle Kantengewichte sind verschieden (wie zuvor)

Kreisprinzip: (cycle property)
betrachte Kreis C im Graphen;
sei e die Kante mit maximalem Gewicht in C;
dann enthält *kein minimaler Spannbaum* die Kante e

Strategie: betrachte Baum T mit e und konstruiere Baum T' mit kleinerem Gewicht

Beweis:

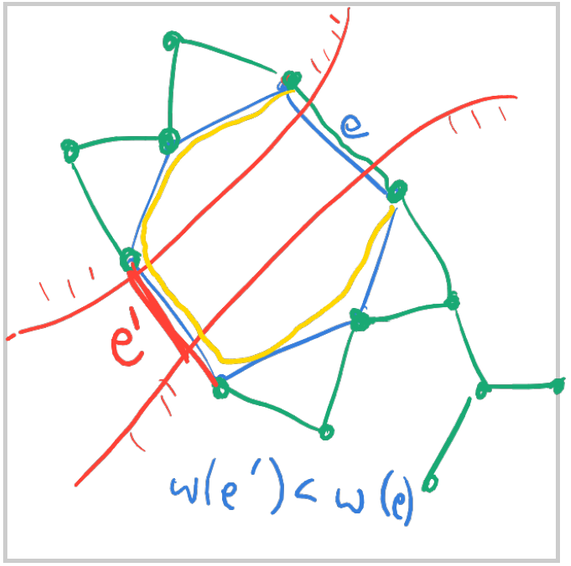


Annahme: alle Kantengewichte sind verschieden (wie zuvor)

Kreisprinzip: (cycle property)
betrachte Kreis C im Graphen;
sei e die Kante mit maximalem Gewicht in C;
dann enthält *kein minimaler Spannbaum* die Kante e

Strategie: betrachte Baum T mit e und konstruiere Baum T' mit kleinerem Gewicht

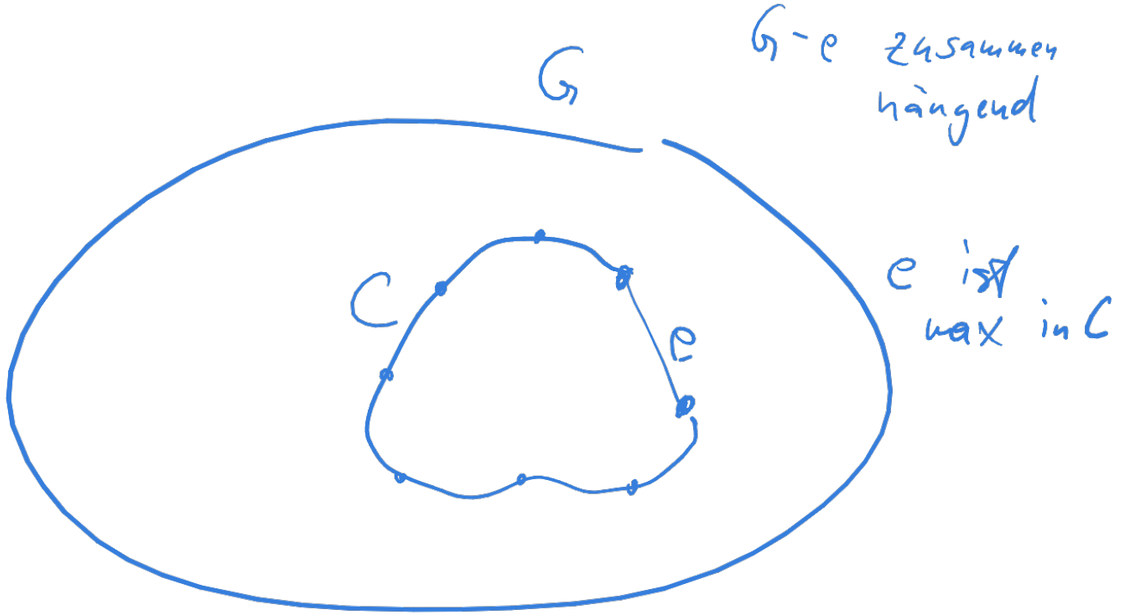
Beweis: wähle $T' = T - e + e'$ wobei $e' \in C - e$ beide Zusammenhangskomponenten in $T - e$ miteinander verbindet



Korrektheit von Rückwärts-Kruskal durch Kreisprinzip

Satz: jede Kante, die der Rückwärts-Kruskal Algorithms entfernt, erfüllt das Kreisprinzip

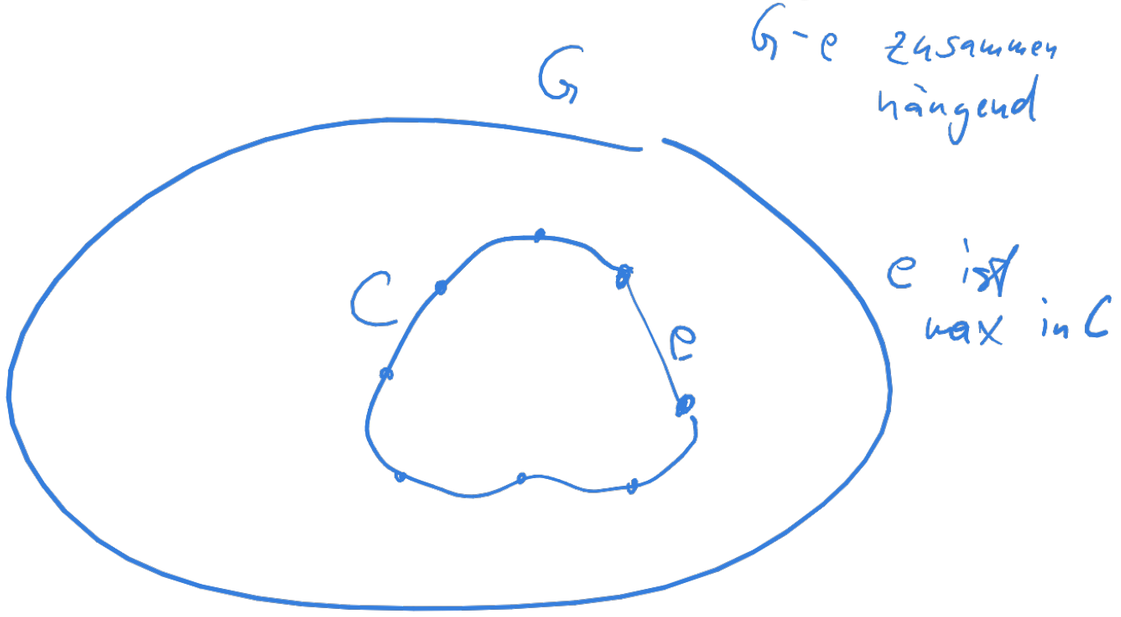
Beweis:



Korrektheit von Rückwärts-Kruskal durch Kreisprinzip

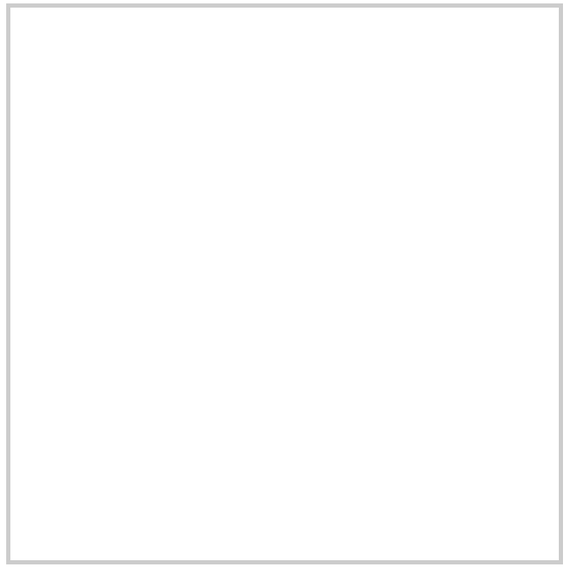
Satz: jede Kante, die der Rückwärts-Kruskal Algorithms entfernt, erfüllt das Kreisprinzip

Beweis: in jeder Iteration, entfernt Rückwärts-Kruskal die teuerste Kante unter all denen, die auf einem Kreis liegen



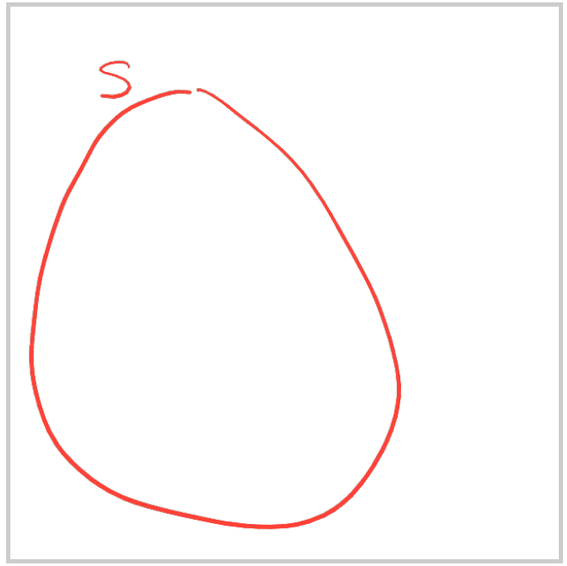
Annahme: alle Kantengewichte sind verschieden (warum OK?)

Schnittprinzip: (cut property)



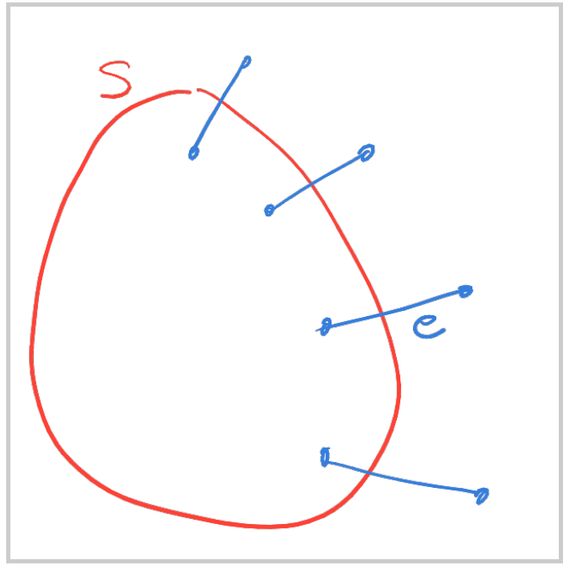
Annahme: alle Kantengewichte sind verschieden (warum OK?)

Schnittprinzip: (cut property)
betrachte Knotenmenge $S \subseteq V$ mit $0 < |S| < |V|$;



Annahme: alle Kantengewichte sind verschieden (warum OK?)

Schnittprinzip: (cut property)
betrachte Knotenmenge $S \subseteq V$ mit $0 < |S| < |V|$;
unter allen Kanten mit genau einem Endpunkt in S, sei e
diejenige mit kleinstem Gewicht;

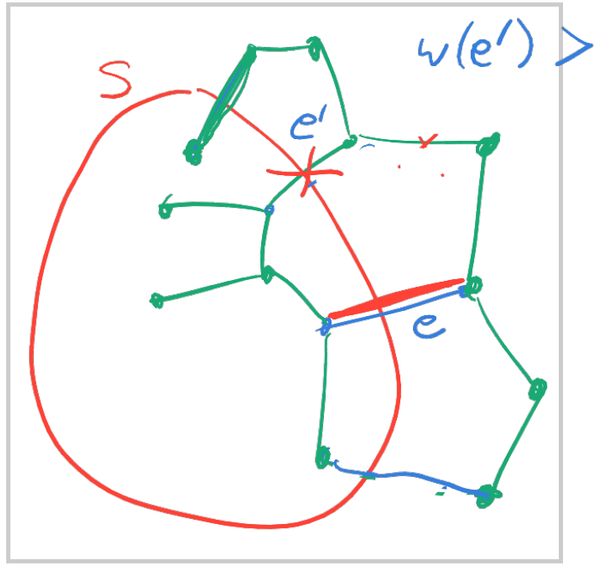


Annahme: alle Kantengewichte sind verschieden (warum OK?)

Schnittprinzip: (cut property)
betrachte Knotenmenge $S \subseteq V$ mit $0 < |S| < |V|$;
unter allen Kanten mit genau einem Endpunkt in S, sei e
diejenige mit kleinstem Gewicht;
dann enthält *jeder minimale Spannbaum* die Kante e.

Strategie: betrachte Baum T ohne e
und konstruiere Baum T' mit
kleinerem Gewicht

Beweis:

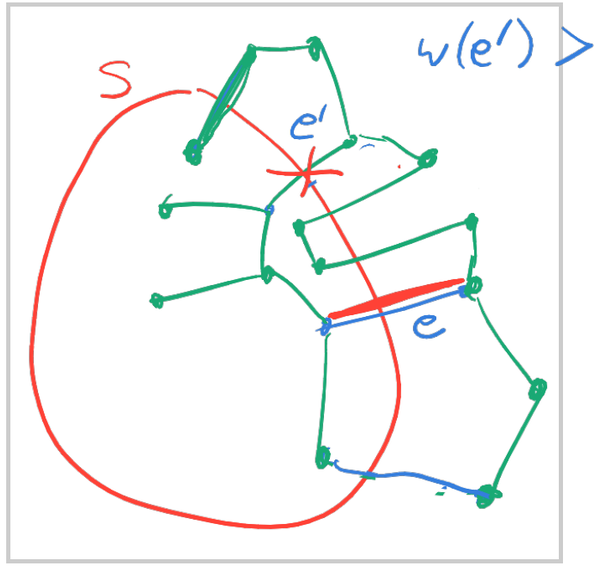


Annahme: alle Kantengewichte sind verschieden (warum OK?)

Schnittprinzip: (cut property)
betrachte Knotenmenge $S \subseteq V$ mit $0 < |S| < |V|$;
unter allen Kanten mit genau einem Endpunkt in S, sei e diejenige mit kleinstem Gewicht;
dann enthält *jeder minimale Spannbaum* die Kante e.

Strategie: betrachte Baum T ohne e und konstruiere Baum T' mit kleinerem Gewicht

Beweis: wähle $T' = T + e - e'$ wobei $e' \in T$ eine Kante auf dem Kreis in $T + e$ mit genau einem Endpunkt in S ist



Korrektheit von Prim durch Schnittprinzip

Satz: jede Kante, die Prims Algorithmus auswählt, erfüllt das Schnittprinzip

Korrektheit von Prim durch Schnittprinzip

Satz: jede Kante, die Prims Algorithmus auswählt, erfüllt das Schnittprinzip

Beweis:
angenommen Prim wählt Kanten $e_1, \ldots, e_{n-1}$ nacheinander

Korrektheit von Prim durch Schnittprinzip

Satz: jede Kante, die Prims Algorithmus auswählt, erfüllt das Schnittprinzip

Beweis:
angenommen Prim wählt Kanten $e_1, \ldots, e_{n-1}$ nacheinander

erfüllt e_i das Schnittprinzip?

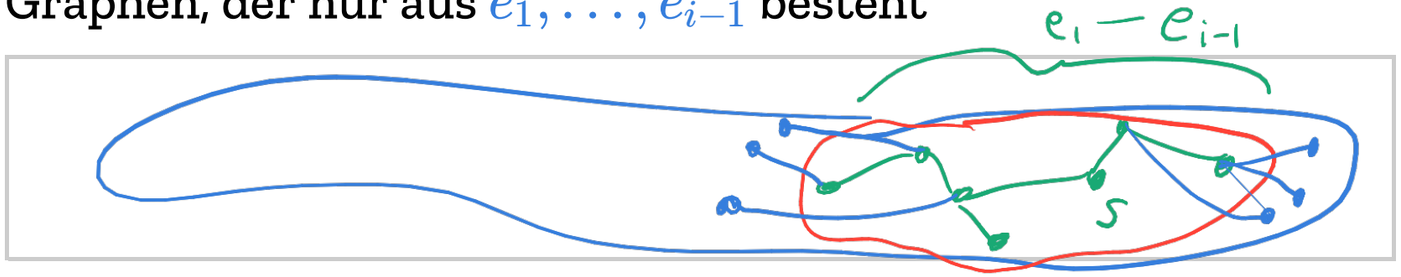
Korrektheit von Prim durch Schnittprinzip

Satz: jede Kante, die Prims Algorithmus auswählt, erfüllt das Schnittprinzip

Beweis:
angenommen Prim wählt Kanten $e_1, \ldots, e_{n-1}$ nacheinander

erfüllt e_i das Schnittprinzip?

betrachte Zusammenhangskomponente S des Startknoten im Graphen, der nur aus $e_1, \ldots, e_{i-1}$ besteht



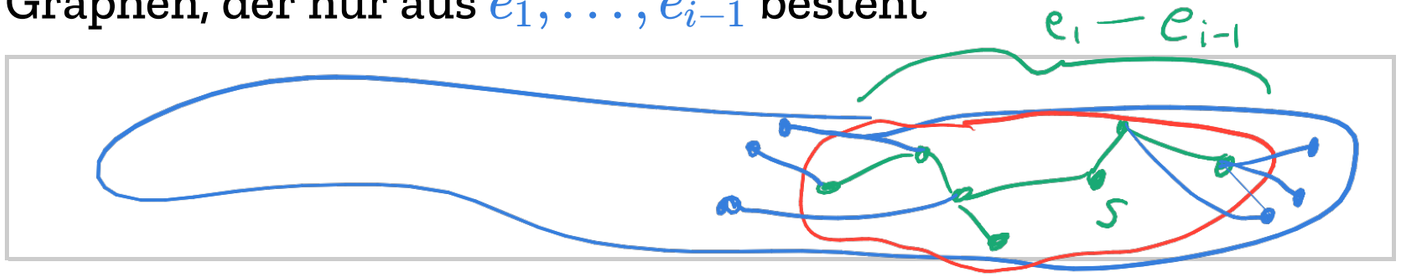
Korrektheit von Prim durch Schnittprinzip

Satz: jede Kante, die Prims Algorithmus auswählt, erfüllt das Schnittprinzip

Beweis:
angenommen Prim wählt Kanten $e_1, \ldots, e_{n-1}$ nacheinander

erfüllt e_i das Schnittprinzip?

betrachte Zusammenhangskomponente S des Startknoten im Graphen, der nur aus $e_1, \ldots, e_{i-1}$ besteht



dann hat e_i das kleinste Gewicht unter allen Kanten mit genau einem Knoten in S $\square$

Korrektheit von Kruskal durch Schnittprinzip

Satz: jede Kante, die Kruskals Algorithmus auswählt, erfüllt das Schnittprinzip

Korrektheit von Kruskal durch Schnittprinzip

Satz: jede Kante, die Kruskals Algorithmus auswählt, erfüllt das Schnittprinzip

Beweis:
angenommen Kruskal wählt $e_1, \ldots, e_{n-1} \in E$ nacheinander

erfüllt e_i das Schnittprinzip?

Korrektheit von Kruskal durch Schnittprinzip

Satz: jede Kante, die Kruskals Algorithmus auswählt, erfüllt das Schnittprinzip

Beweis:
angenommen Kruskal wählt $e_1, \ldots, e_{n-1} \in E$ nacheinander

erfüllt e_i das Schnittprinzip?

betrachte Zusammenhangskomponenten $S_1, \ldots, S_k$ im Graphen, der nur aus $e_1, \ldots, e_{i-1}$ besteht

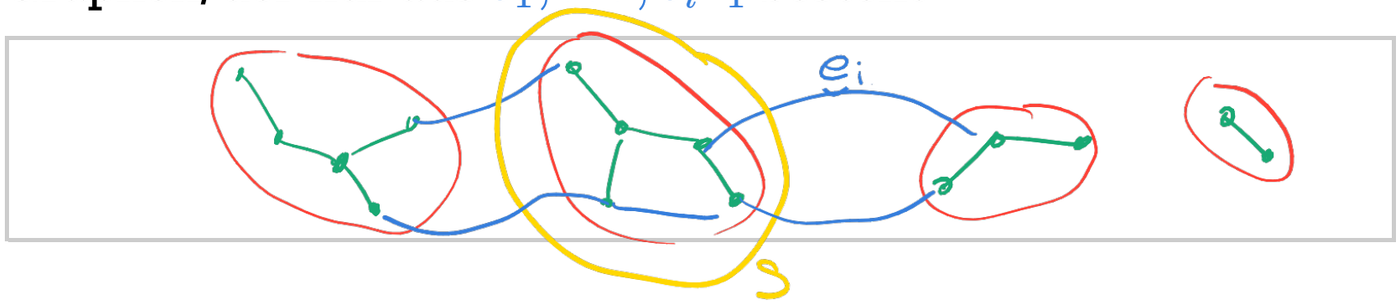
Korrektheit von Kruskal durch Schnittprinzip

Satz: jede Kante, die Kruskals Algorithmus auswählt, erfüllt das Schnittprinzip

Beweis:
angenommen Kruskal wählt $e_1, \ldots, e_{n-1} \in E$ nacheinander

erfüllt e_i das Schnittprinzip?

betrachte Zusammenhangskomponenten $S_1, \ldots, S_k$ im Graphen, der nur aus $e_1, \ldots, e_{i-1}$ besteht



Korrektheit von Kruskal durch Schnittprinzip

Satz: jede Kante, die Kruskals Algorithmus auswählt, erfüllt das Schnittprinzip

Beweis:
angenommen Kruskal wählt $e_1, \ldots, e_{n-1} \in E$ nacheinander

erfüllt e_i das Schnittprinzip?

betrachte Zusammenhangskomponenten $S_1, \ldots, S_k$ im Graphen, der nur aus $e_1, \ldots, e_{i-1}$ besteht

dann ist e_i die Kante mit kleinstem Gewicht unter allen Kanten mit Endpunkten in zwei versch. Komponenten $\square$

Laufzeit

Laufzeit

leicht einzusehen: alle drei kennengelernten Algorithmen lassen sich mit polynomieller Laufzeit implementieren
(tatsächlich reicht jeweils $O(n \cdot m)$ Zeit aus)

Laufzeit

leicht einzusehen: alle drei kennengelernten Algorithmen lassen sich mit polynomieller Laufzeit implementieren
(tatsächlich reicht jeweils $O(n \cdot m)$ Zeit aus)

zumindest die Algorithmen von Prim und Kruskal lassen sich sehr effizient implementieren, mit Laufzeit $O(m \cdot \log n)$

Laufzeit

leicht einzusehen: alle drei kennengelernten Algorithmen lassen sich mit polynomieller Laufzeit implementieren (tatsächlich reicht jeweils $O(n \cdot m)$ Zeit aus)

zumindest die Algorithmen von Prim und Kruskal lassen sich sehr effizient implementieren, mit Laufzeit $O(m \cdot \log n)$

im Folgenden: effiziente Implementierung von Kruskals Algorithmus

Pseudocode für Kruskals Algorithm

gegeben: Graph $G = (V, E)$ mit Gewichten $w\colon E \to \mathbb{R}$

gesucht: Spannbaum $T \subseteq E$ mit minimalem Gewicht

Pseudocode für Kruskals Algorithm

gegeben: Graph $G = (V, E)$ mit Gewichten $w\colon E \to \mathbb{R}$
gesucht: Spannbaum $T \subseteq E$ mit minimalem Gewicht

Prozedur:

$m = |E|$

- $T \leftarrow \emptyset$ $\quad O(m \cdot \log m)$
- für alle Kanten $e \in E$ *sortiert nach aufsteigendem Gewicht*
  - falls $T + e$ keinen Kreis enthält ?
    - setze $T \leftarrow T + e$

Pseudocode für Kruskals Algorithm

gegeben: Graph $G = (V, E)$ mit Gewichten $w\colon E \to \mathbb{R}$
gesucht: Spannbaum $T \subseteq E$ mit minimalem Gewicht

Prozedur:

$m = |E|$

- $T \leftarrow \emptyset$ $\quad O(m \cdot \log m)$
- für alle Kanten $e \in E$ *sortiert nach aufsteigendem Gewicht*
  - falls $T + e$ keinen Kreis enthält ?
    - setze $T \leftarrow T + e$

Beobachtung: $T + e$ enthält keinen Kreis $\Leftrightarrow$ die Endpunkte von e sind in verschiedenen Z.-komponenten von T

gegeben: Graph $G = (V, E)$ mit Gewichten $w : E \to \mathbb{R}$
gesucht: Spannbaum $T \subseteq E$ mit minimalem Gewicht

Prozedur:

$m = |E|$

- $T \leftarrow \emptyset$ $\quad O(m \cdot \log m)$
- für alle Kanten $e \in E$ *sortiert nach aufsteigendem Gewicht*
  - falls $T + e$ keinen Kreis enthält ?
    - setze $T \leftarrow T + e$

Beobachtung: $T + e$ enthält keinen Kreis $\Leftrightarrow$ die Endpunkte von e sind in verschiedenen Z.-komponenten von T

Idee: Datenstruktur für Zusammenhangskomponenten (ZHK)

gegeben: Graph $G = (V, E)$ mit Gewichten $w : E \to \mathbb{R}$
gesucht: Spannbaum $T \subseteq E$ mit minimalem Gewicht

Prozedur:

$m = |E|$

- $T \leftarrow \emptyset$, make(V) $O(m \cdot \log m)$
- für alle Kanten $e \in E$ sortiert nach aufsteigendem Gewicht $e = \{u, v\}$
  - falls $T + e$ keinen Kreis enthält find(u) ≠ find(v)
    - setze $T \leftarrow T + e$; union(u, v)

Beobachtung: $T + e$ enthält keinen Kreis $\Leftrightarrow$ die Endpunkte von e sind in verschiedenen Z.-komponenten von T

Idee: Datenstruktur für Zusammenhangskomponenten (ZHK)

make(V): erstelle Datenstruktur für leeren Graphen auf V
find(u): gib *eindeutigen Namen* der ZHK von u zurück
union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

Union-Find Datentyp / Datenstruktur

make(V): erstelle Datenstruktur für leeren Graphen auf V
find(u): gib *eindeutigen Namen* der ZHK von u zurück
union(u,v): füge Kante $\{u, v\}$ hinzu / *vereinige ZHKs* von u und v

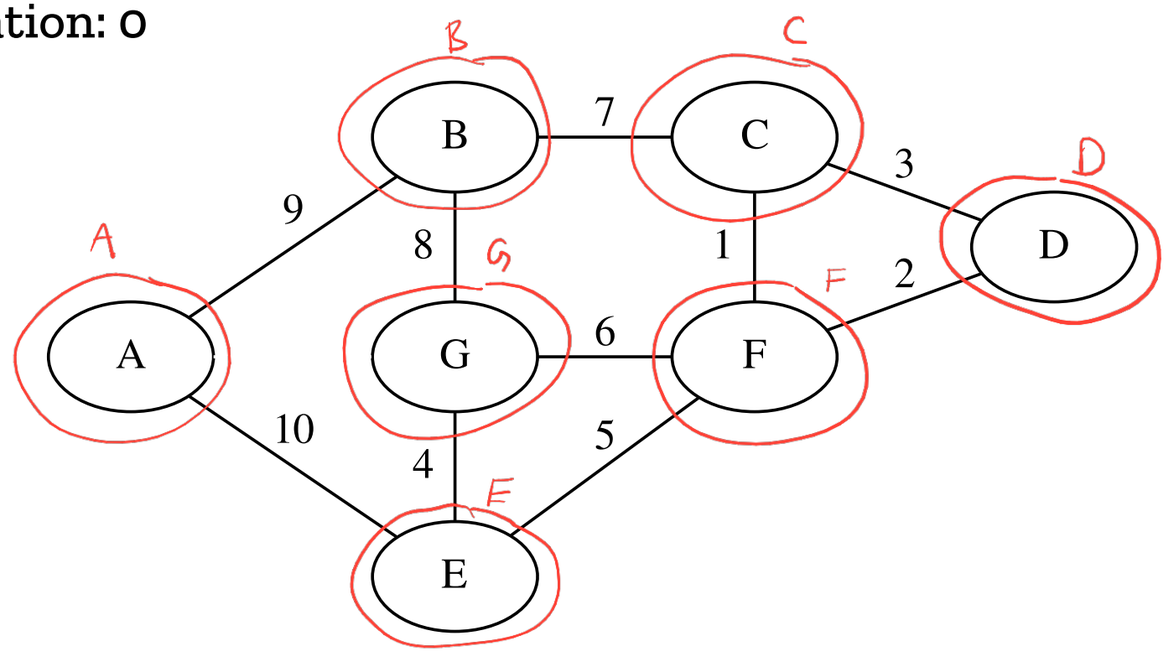
Union-Find Datentyp / Datenstruktur

make(V): erstelle Datenstruktur für leeren Graphen auf V
find(u): gib *eindeutigen Namen* der ZHK von u zurück
union(u,v): füge Kante $\{u, v\}$ hinzu / *vereinige ZHKs* von u und v

ausgehend von naiver Datenstruktur,
werden wir effiziente Datenstruktur erarbeiten,
die diese Operationen unterstützt

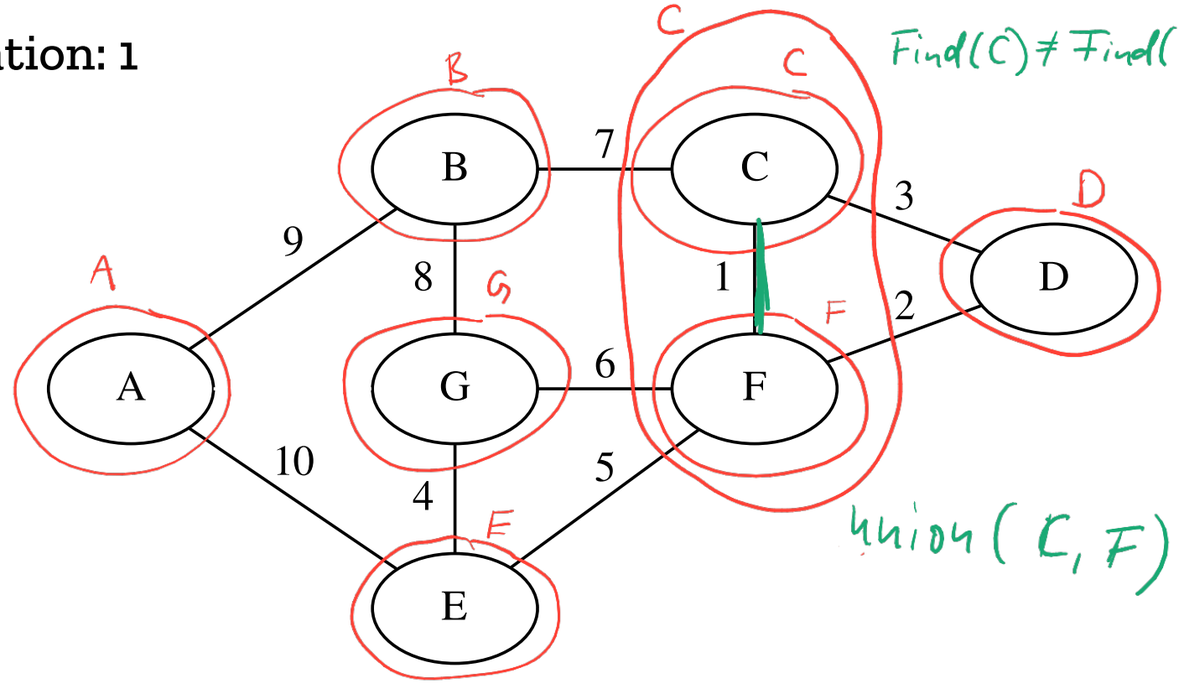
Beispiel: Kruskal mit Hilfe von Union-Find Datentyp

Iteration: 0



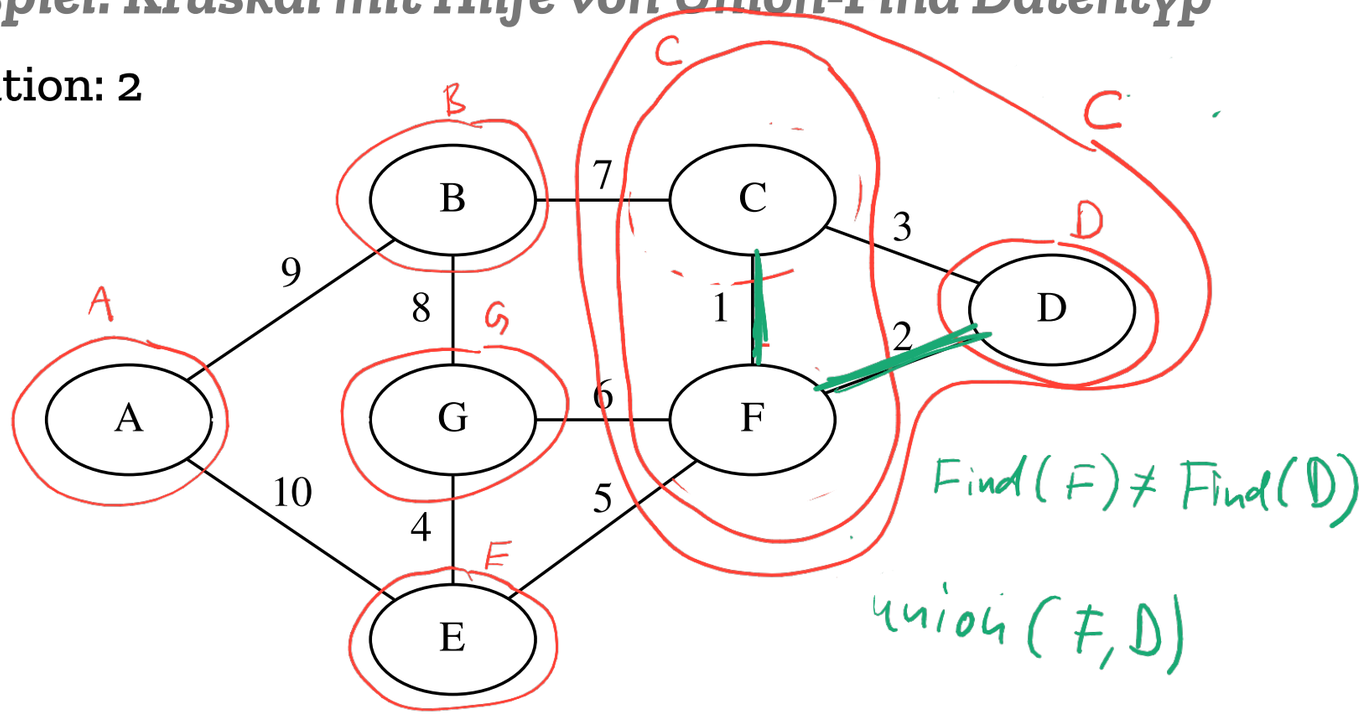
Beispiel: Kruskal mit Hilfe von Union-Find Datentyp

Iteration: 1



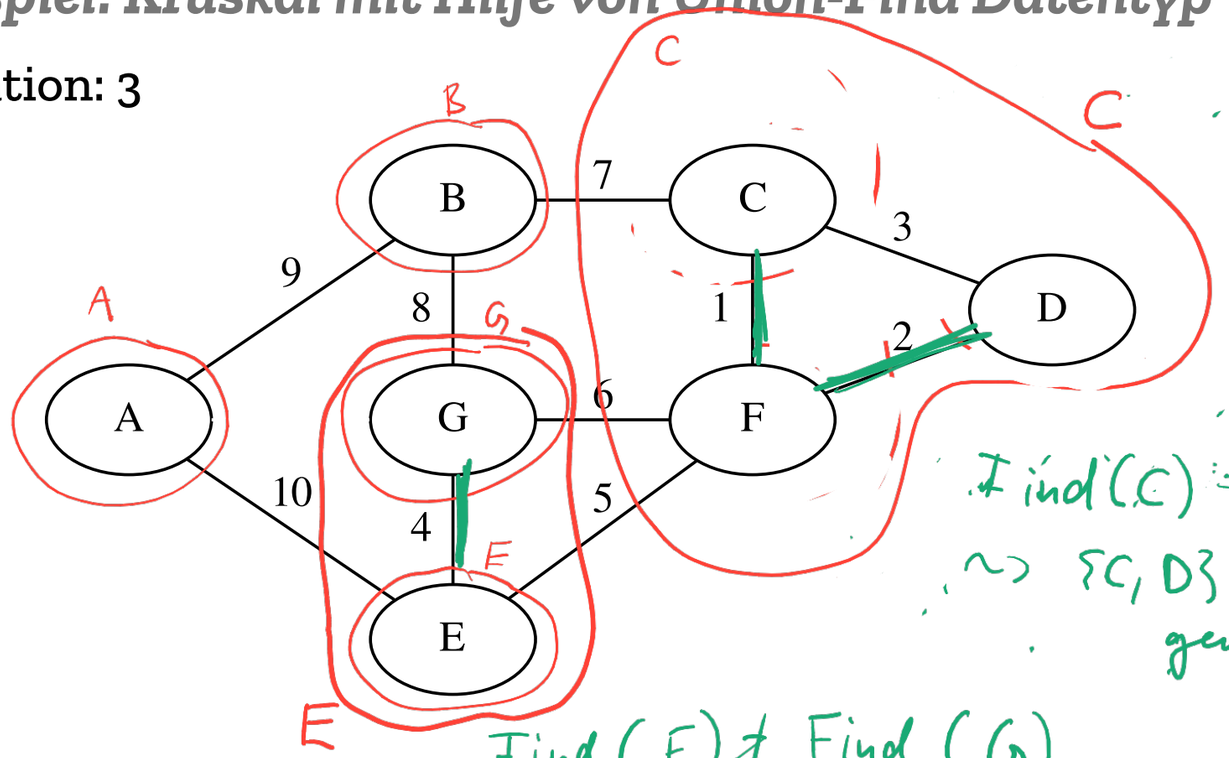
Beispiel: Kruskal mit Hilfe von Union-Find Datentyp

Iteration: 2



Beispiel: Kruskal mit Hilfe von Union-Find Datentyp

Iteration: 3

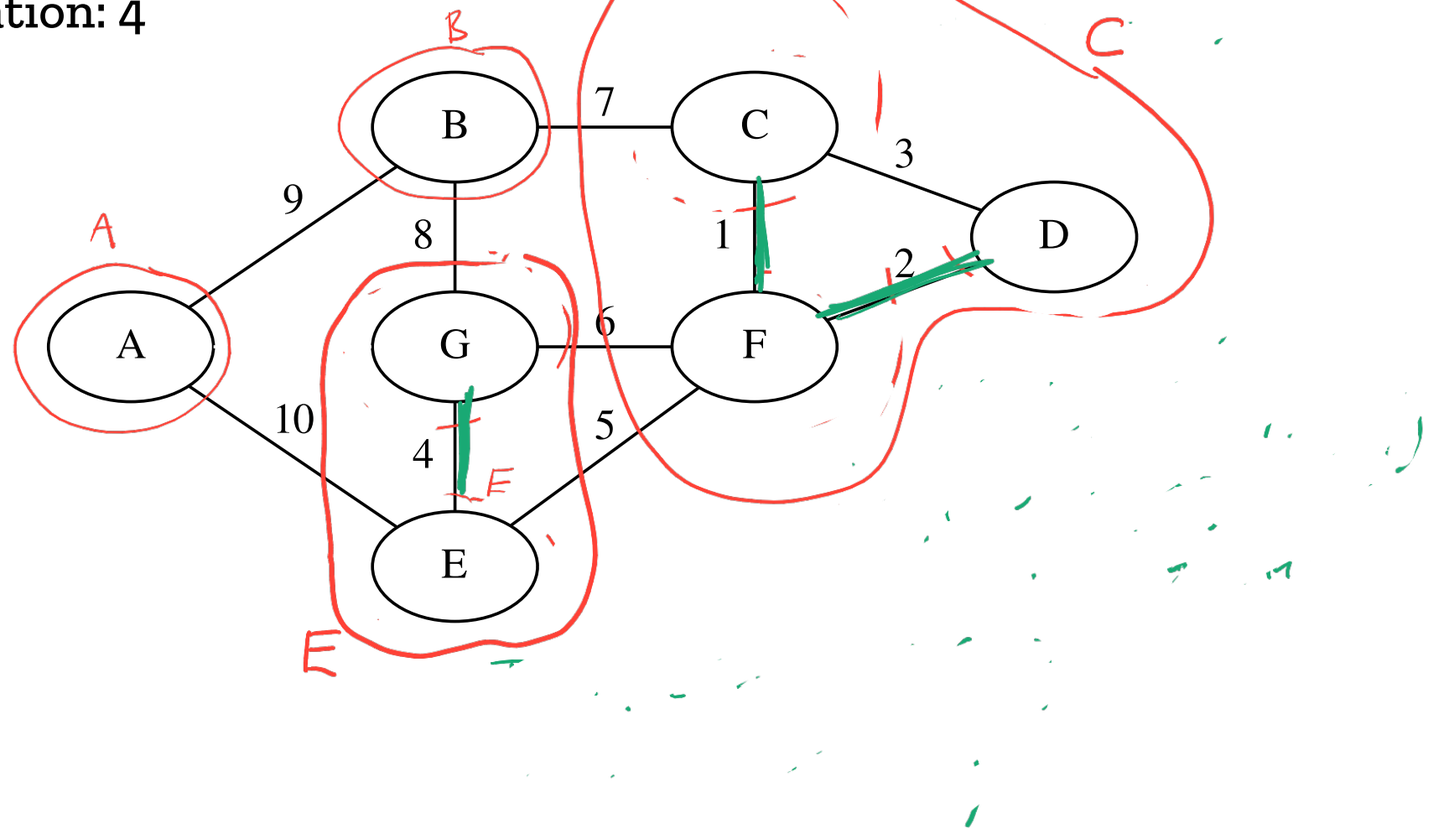


Find(C) = Find(D)
⇝ {C, D} nicht gewählt

Find(E) ≠ Find(G)
Union(E, G)

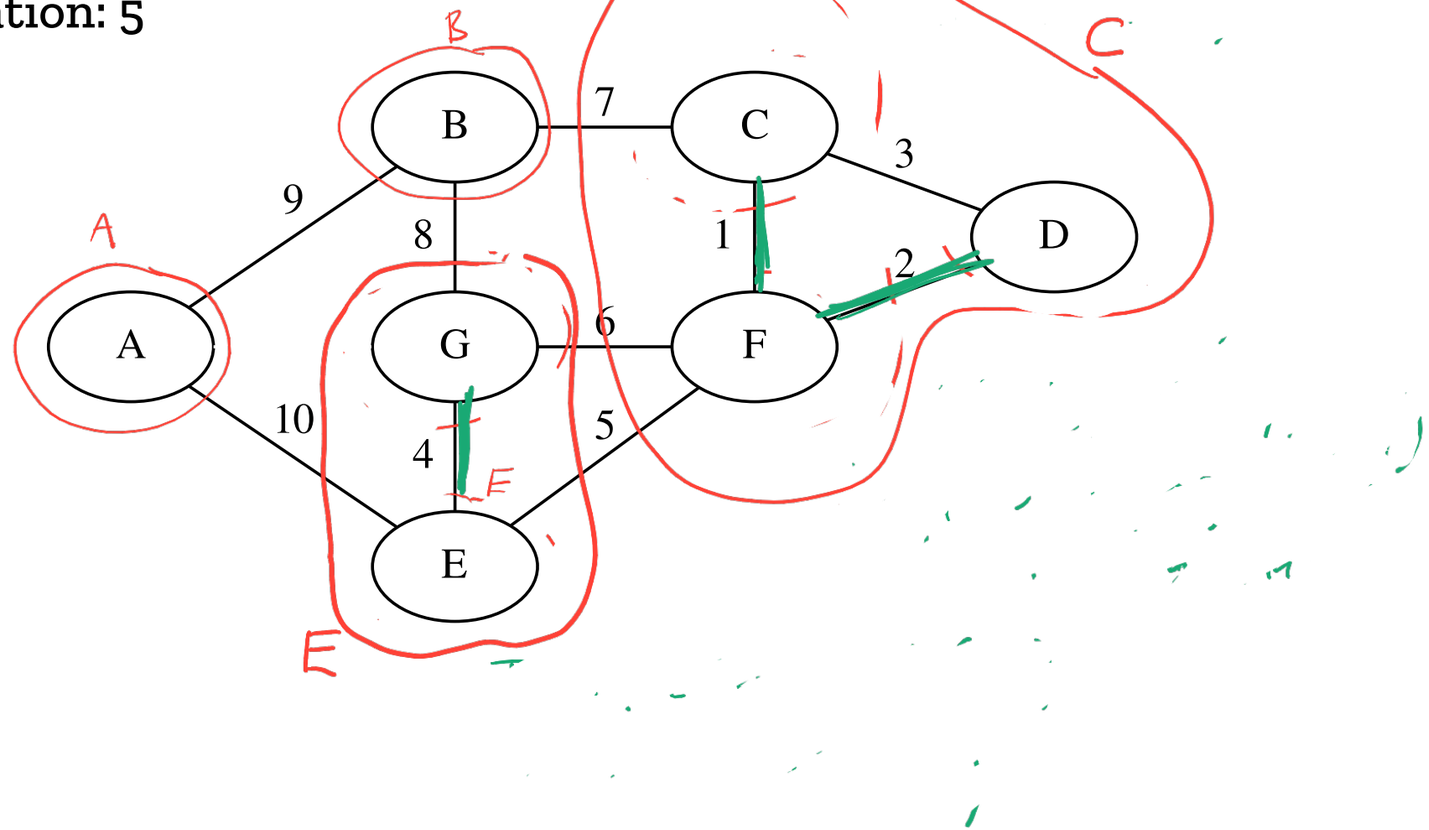
Beispiel: Kruskal mit Hilfe von Union-Find Datentyp

Iteration: 4



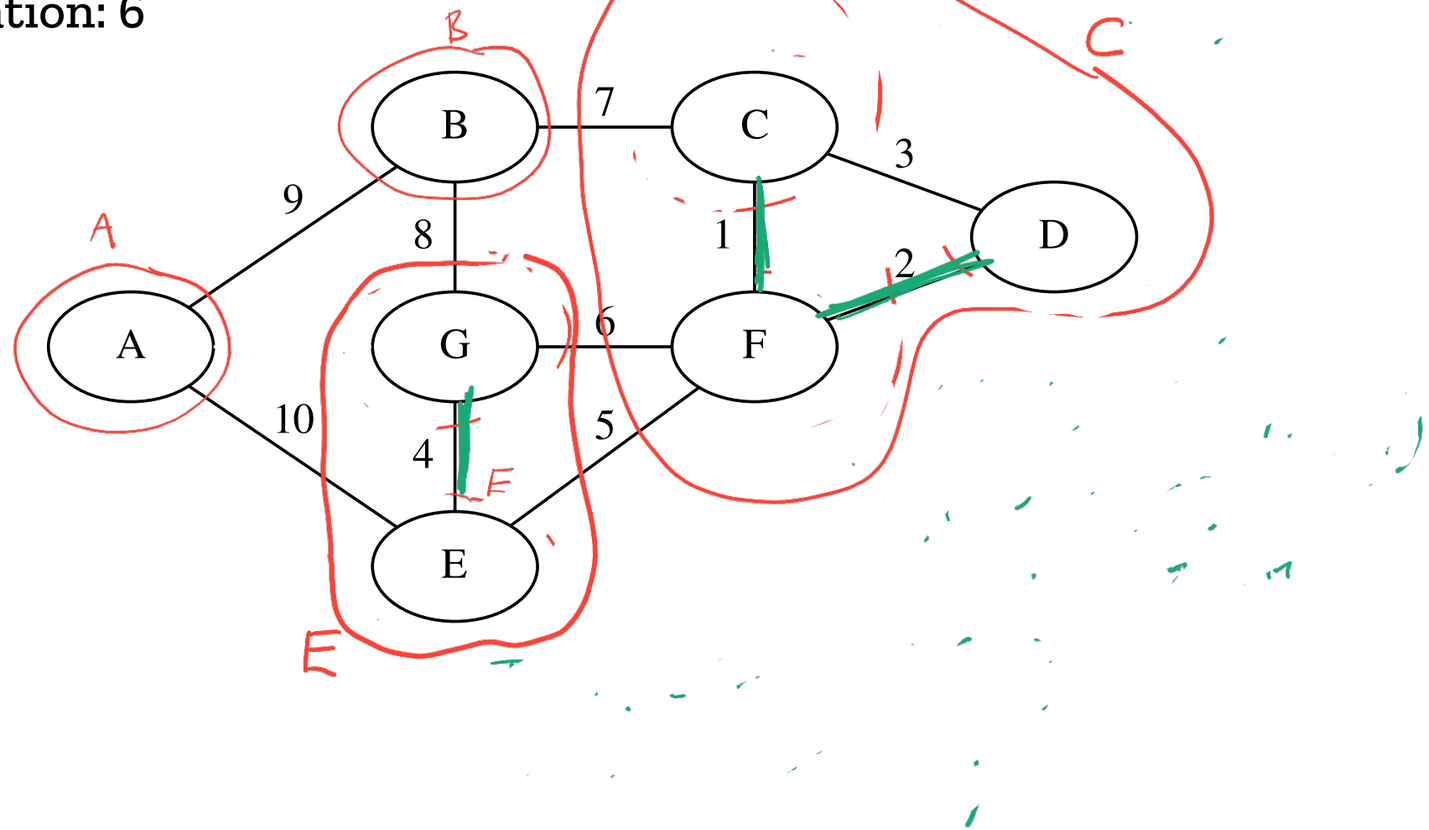
Beispiel: Kruskal mit Hilfe von Union-Find Datentyp

Iteration: 5



Beispiel: Kruskal mit Hilfe von Union-Find Datentyp

Iteration: 6



Union-Find: erster Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Operationen:

Union-Find: erster Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Operationen:

make(V): erstelle Datenstruktur für leeren Graphen auf V

- setze $\text{rep}[u] \leftarrow u$ für alle $u \in V$

Union-Find: erster Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Operationen:

make(V): erstelle Datenstruktur für leeren Graphen auf V

- setze rep[u] $\leftarrow$ u für alle $u \in V$ // O(n) Laufzeit

find(u): gib *eindeutigen Namen* der ZHK von u zurück

- return rep[u] // Laufzeit O(1) (Gesamt: O(m))

Union-Find: erster Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Operationen:

make(V): erstelle Datenstruktur für leeren Graphen auf V

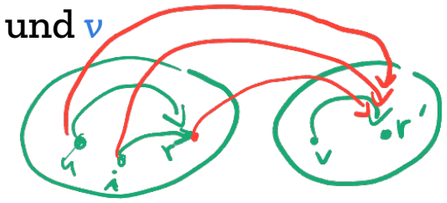
- setze rep[u] $\leftarrow$ u für alle $u \in V$ // O(n) Laufzeit

find(u): gib *eindeutigen Namen* der ZHK von u zurück

- return rep[u] // Laufzeit O(1) (Gesamt: O(m))

union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- für alle $i \in V$
  - falls rep[i]=rep[u], setze rep[i] $\leftarrow$ rep[v]



Union-Find: erster Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Operationen:

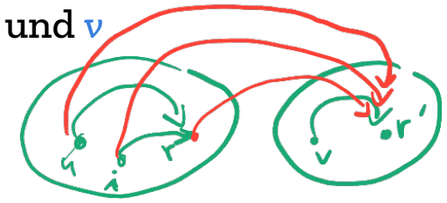
make(V): erstelle Datenstruktur für leeren Graphen auf V

- setze rep[u] $\leftarrow$ u für alle $u \in V$ // O(n) Laufzeit

find(u): gib *eindeutigen Namen* der ZHK von u zurück

- return rep[u] // Laufzeit O(1) (Gesamt: O(m))

union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v



- für alle $i \in V$
  - falls rep[i]=rep[u], setze rep[i] $\leftarrow$ rep[v]

Laufzeit:

Union-Find: erster Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Operationen:

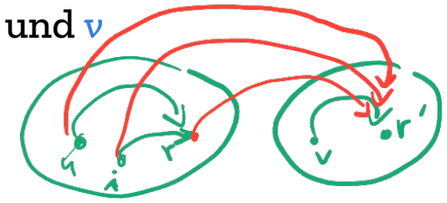
make(V): erstelle Datenstruktur für leeren Graphen auf V

- setze $\text{rep}[u] \leftarrow u$ für alle $u \in V$ // O(n) Laufzeit

find(u): gib *eindeutigen Namen* der ZHK von u zurück

- return rep[u] // Laufzeit O(1) (Gesamt: O(n))

union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v



- für alle $i \in V$
  - falls $\text{rep}[i]$=rep[u], setze $\text{rep}[i] \leftarrow$ rep[v]

Laufzeit: immer $\Theta(n)$ Schritte für `union(u,v)`

Union-Find: erster Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Operationen:

make(V): erstelle Datenstruktur für leeren Graphen auf V

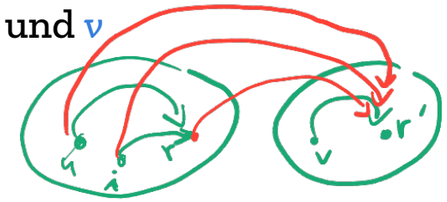
- setze rep[u] $\leftarrow$ u für alle $u \in V$ // O(n) Laufzeit

find(u): gib *eindeutigen Namen* der ZHK von u zurück

- return rep[u] // Laufzeit O(1) (Gesamt: O(m))

union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- für alle $i \in V$
  - falls rep[i]=rep[u], setze rep[i] $\leftarrow$ rep[v]



Laufzeit: immer $\Theta(n)$ Schritte für `union(u,v)` ⇝ *zu teuer!* $O(n^2)$ Gesamt

Union-Find: zweiter Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

*Union-Find: **zweiter** Versuch*

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Array members[r] = Liste der ZHK representiert von r

*Union-Find: **zweiter** Versuch*

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Array members[r] = Liste der ZHK representiert von r

Operationen:

`make` und `find` wie zuvor (im wesentlichen)

union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

*Union-Find: **zweiter** Versuch*

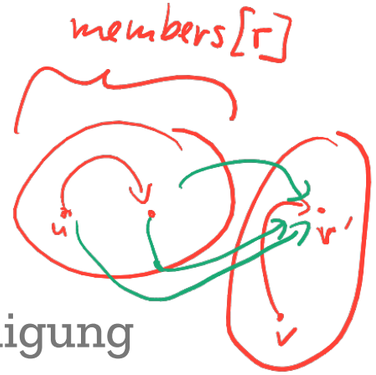
Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Array members[r] = Liste der ZHK representiert von r

Operationen:

`make` und `find` wie zuvor (im wesentlichen)



union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- $r \leftarrow$ rep[u] und $r' \leftarrow$ rep[v]
- für alle $i \in$ members$[r]$ // r' wird Rep. der Vereinigung
  - rep$[i] \leftarrow r'$
- members$[r'] \leftarrow$ members$[r'] \cup$ members$[r]$ // schnell genug
- members$[r] \leftarrow \emptyset$

*Union-Find: **zweiter** Versuch*

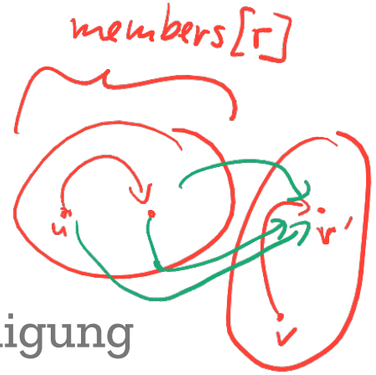
Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Array members[r] = Liste der ZHK representiert von r

Operationen:

`make` und `find` wie zuvor (im wesentlichen)



union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- $r \leftarrow$ rep[u] und $r' \leftarrow$ rep[v]
- für alle $i \in$ members$[r]$ // r' wird Rep. der Vereinigung
  - rep$[i] \leftarrow r'$
- members$[r'] \leftarrow$ members$[r'] \cup$ members$[r]$ // schnell genug
- members$[r] \leftarrow \emptyset$

Laufzeit:

*Union-Find: **zweiter** Versuch*

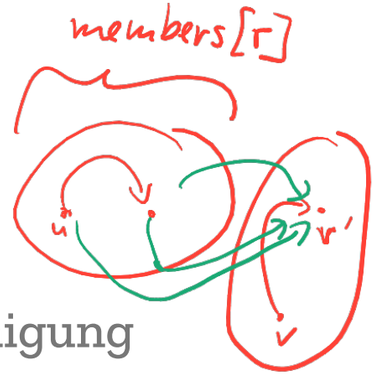
Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Array members[r] = Liste der ZHK representiert von r

Operationen:

`make` und `find` wie zuvor (im wesentlichen)



union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- $r \leftarrow$ rep[u] und $r' \leftarrow$ rep[v]
- für alle $i \in$ members$[r]$ // r' wird Rep. der Vereinigung
  - rep$[i] \leftarrow r'$
- members$[r'] \leftarrow$ members$[r'] \cup$ members$[r]$ // schnell genug
- members$[r] \leftarrow \emptyset$

Laufzeit: $O(|\mathrm{ZHK}(u)|)$ für `union(u,v)`

*Union-Find: **zweiter** Versuch*

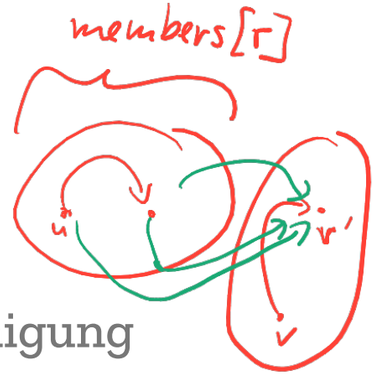
Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Array members[r] = Liste der ZHK representiert von r

Operationen:

`make` und `find` wie zuvor (im wesentlichen)



`union(u,v)`: füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- $r \leftarrow$ `rep[u]` und $r' \leftarrow$ `rep[v]`
- für alle $i \in$ members$[r]$ // r' wird Rep. der Vereinigung
  - rep$[i] \leftarrow r'$
- members$[r'] \leftarrow$ members$[r'] \cup$ members$[r]$ // schnell genug
- members$[r] \leftarrow \emptyset$

Laufzeit: $O(|\mathrm{ZHK}(u)|)$ für `union(u,v)` ⇝ *auch zu teuer!* ☹

Union-Find: dritter (letzter) Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u

Array members[r] = Liste der ZHK representiert von r

Union-Find: dritter (letzter) Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u
Array members[r] = Liste der ZHK representiert von r
Array size[r] = Grösse der ZHK representiert von r

Union-Find: dritter (letzter) Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u
Array members[r] = Liste der ZHK representiert von r
Array size[r] = Grösse der ZHK representiert von r

Operationen:

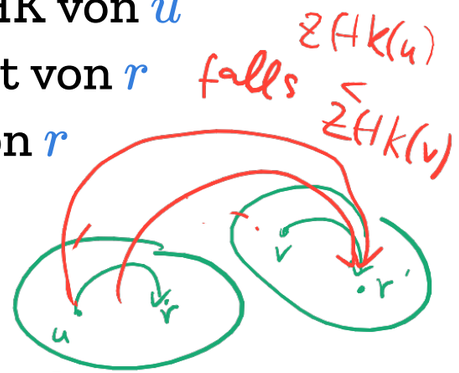
`make` und `find` wie zuvor (im wesentlichen)

union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

Union-Find: dritter (letzter) Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u
Array members[r] = Liste der ZHK representiert von r
Array size[r] = Grösse der ZHK representiert von r



Operationen:

`make` und `find` wie zuvor (im wesentlichen)

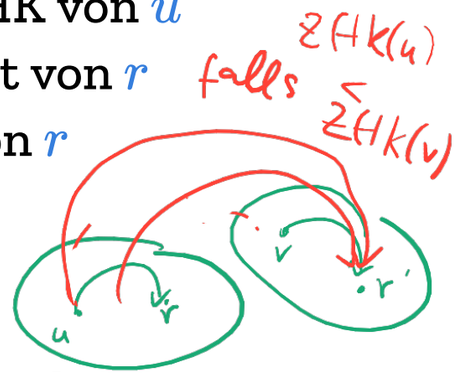
union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- betrachte Grössen der ZHKs von u und v
- mache Rep. der grösseren ZHK zum Rep. der Vereinigung
  (dazu muss nur kleinere ZHK durchlaufen werden)

Union-Find: dritter (letzter) Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u
Array members[r] = Liste der ZHK representiert von r
Array size[r] = Grösse der ZHK representiert von r



Operationen:

`make` und `find` wie zuvor (im wesentlichen)

union(u,v): füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

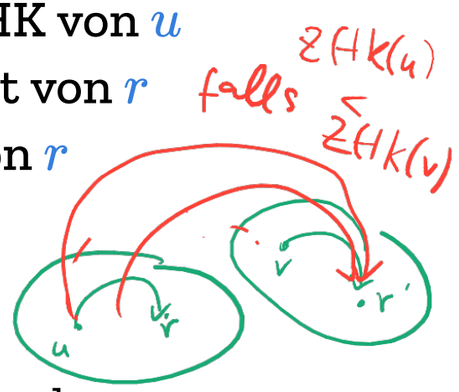
- betrachte Grössen der ZHKs von u und v
- mache Rep. der grösseren ZHK zum Rep. der Vereinigung
  (dazu muss nur kleinere ZHK durchlaufen werden)

Laufzeit:

Union-Find: dritter (letzter) Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u
Array members[r] = Liste der ZHK representiert von r
Array size[r] = Grösse der ZHK representiert von r



Operationen:

`make` und `find` wie zuvor (im wesentlichen)

`union(u,v)`: füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

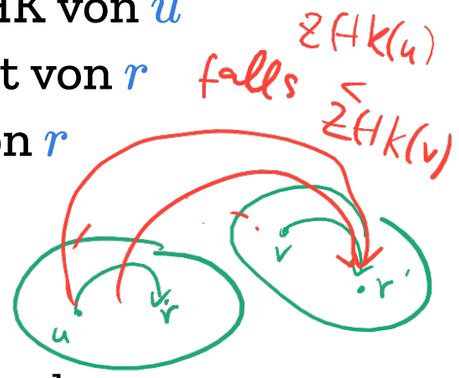
- betrachte Grössen der ZHKs von u und v
- mache Rep. der grösseren ZHK zum Rep. der Vereinigung
  (dazu muss nur kleinere ZHK durchlaufen werden)

Laufzeit: $O(\min\{|\mathrm{ZHK}(u)|, |\mathrm{ZHK}(v)|\})$ für `union(u,v)`

Union-Find: dritter (letzter) Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u
Array members[r] = Liste der ZHK representiert von r
Array size[r] = Grösse der ZHK representiert von r



Operationen:

`make` und `find` wie zuvor (im wesentlichen)

`union(u,v)`: füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- betrachte Grössen der ZHKs von u und v
- mache Rep. der grösseren ZHK zum Rep. der Vereinigung
  (dazu muss nur kleinere ZHK durchlaufen werden)

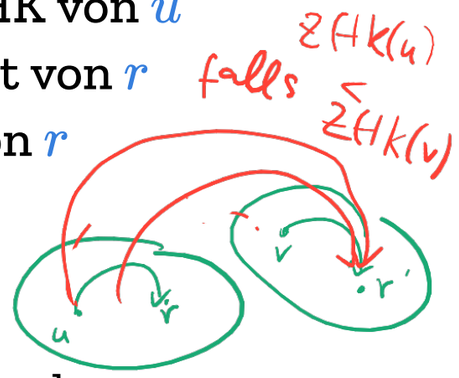
Laufzeit: $O(\min\{|\mathrm{ZHK}(u)|, |\mathrm{ZHK}(v)|\})$ für `union(u,v)`

schlimmster Fall: $\Theta(n)$

Union-Find: dritter (letzter) Versuch

Speicher:

Array rep[u] = eindeutiger Repräsentant der ZHK von u
Array members[r] = Liste der ZHK representiert von r
Array size[r] = Grösse der ZHK representiert von r



Operationen:

`make` und `find` wie zuvor (im wesentlichen)

`union(u,v)`: füge Kante {u, v} hinzu / *vereinige ZHKs* von u und v

- betrachte Grössen der ZHKs von u und v
- mache Rep. der grösseren ZHK zum Rep. der Vereinigung
  (dazu muss nur kleinere ZHK durchlaufen werden)

Laufzeit: $O(\min\{|\mathrm{ZHK}(u)|, |\mathrm{ZHK}(v)|\})$ für `union(u,v)`

schlimmster Fall: $\Theta(n)$
werden zeigen: kann höchstens $O(\log n)$ *mal eintreten*

Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n - 1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n - 1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

beachte: *einzelne Operationen* können Laufzeit $\Omega(\log n)$ haben

Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n-1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

beachte: *einzelne Operationen* können Laufzeit $\Omega(\log n)$ haben

Beweis:

wieviel Zeit verwenden wir insgesamt für einen Knoten u?

Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n - 1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

beachte: *einzelne Operationen* können Laufzeit $\Omega(\log n)$ haben

Beweis:

wieviel Zeit verwenden wir insgesamt für einen Knoten u?

Hängt davon ab wie oft `rep[u]` *verändert wird!*

Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n - 1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

beachte: *einzelne Operationen* können Laufzeit $\Omega(\log n)$ haben

Beweis:

wieviel Zeit verwenden wir insgesamt für einen Knoten u?

Hängt davon ab wie oft `rep[u]` *verändert wird!*

sei N_u = Anzahl der Änderungen von `rep[u]`

Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n-1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

beachte: *einzelne Operationen* können Laufzeit $\Omega(\log n)$ haben

Beweis:

wieviel Zeit verwenden wir insgesamt für einen Knoten u?

Hängt davon ab wie oft `rep[u]` *verändert wird!*

sei N_u = Anzahl der Änderungen von `rep[u]`

dann gilt $\sum_{i=1}^{n-1} \min\{|\mathrm{ZHK}_i(u_i)|, |\mathrm{ZHK}_i(v_i)|\} = \sum_{u=1}^{n} N_u$

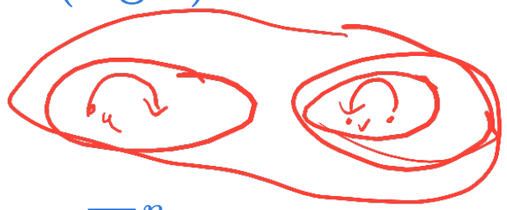
Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n-1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

beachte: *einzelne Operationen* können Laufzeit $\Omega(\log n)$ haben

Beweis:



sei N_u = Anzahl der Änderungen von `rep[u]`

dann gilt $\sum_{i=1}^{n-1} \min\{|\mathrm{ZHK}_i(u_i)|, |\mathrm{ZHK}_i(v_i)|\} = \sum_{u=1}^{n} N_u$

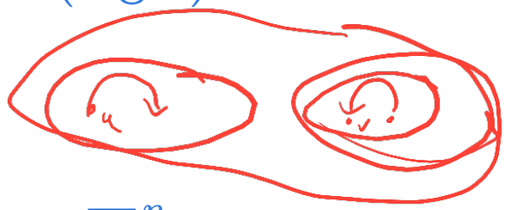
jede Änderungen von `rep[u]` *verdoppelt* Grösse der ZHK von u
(da wir immer nur Rep. der kleineren ZHK ändern)

Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n-1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

beachte: *einzelne Operationen* können Laufzeit $\Omega(\log n)$ haben



Beweis:

sei N_u = Anzahl der Änderungen von `rep[u]`

dann gilt $\sum_{i=1}^{n-1} \min\{|\mathrm{ZHK}_i(u_i)|, |\mathrm{ZHK}_i(v_i)|\} = \sum_{u=1}^{n} N_u$

jede Änderungen von `rep[u]` *verdoppelt* Grösse der ZHK von u
(da wir immer nur Rep. der kleineren ZHK ändern)

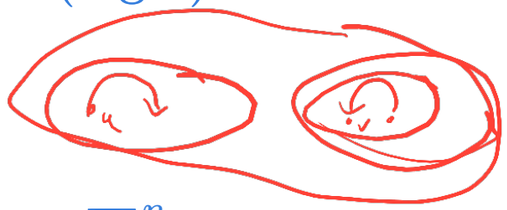
also gilt $N_u \leqslant \log_2 n$ für alle u

Amortisierte Analyse

erzeuge Datenstruktur durch `make(V)` (Laufzeit: $O(|V|)$)

Satz: jede Sequenz von $n-1$ `union` Operationen hat *Gesamtlaufzeit* $O(n \log n)$

beachte: *einzelne Operationen* können Laufzeit $\Omega(\log n)$ haben



Beweis:

sei N_u = Anzahl der Änderungen von `rep[u]`

dann gilt $\sum_{i=1}^{n-1} \min\{|\mathrm{ZHK}_i(u_i)|, |\mathrm{ZHK}_i(v_i)|\} = \sum_{u=1}^{n} N_u$

jede Änderungen von `rep[u]` *verdoppelt* Grösse der ZHK von u
(da wir immer nur Rep. der kleineren ZHK ändern)

also gilt $N_u \leqslant \log_2 n$ für alle u

Gesamtlaufzeit: $O(\sum_{u=1}^{n} N_u) \leqslant O(n \log n)$

Ende