Greedy-Algorithmen

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 23. März 2023
Vorlesungsfolien

ALGORITHMS AND COMPLEXITY GROUP
TU WIEN Informatics

Einleitung

Algorithmen: Paradigmen

Greedy: Erstelle inkrementell eine Lösung, bei der nicht vorausschauend ein lokales Kriterium zur Wahl der jeweils nächsten hinzuzufügenden Lösungskomponente verwendet wird.

Divide-and-Conquer: Teile ein Problem in Teilprobleme auf. Löse jedes Teilproblem unabhängig und kombiniere die Lösung für die Teilprobleme zu einer Lösung für das ursprüngliche Problem.

Greedy: Einführendes Beispiel

Geld wechseln: Gegeben sei eine Stückelung von Münzen (z.B. Euromünzen in Cent): 1, 2, 5, 10, 20, 50, 100, 200.

Gesucht: Methode, um einen Betrag mit der kleinstmöglichen Anzahl an Münzen herauszugeben.

Beispiel:

- 37 Cent
- Optimale Lösung: 1×20, 1×10, 1×5, 1×2

Hinweis: Es kann auch mehr als eine Lösung geben.

- Stückelung von Münzen: 1, 5, 10, 20, 25, 50
- Betrag: 30
- 1×20 und 1×10 sowie 1×25 und 1×5 sind optimale Lösungen.

Geld wechseln: Greedy-Algorithmus

Greedy-Ansatz: Für Betrag S.

```
while S ≠ 0
    Finde die Münze mit größtem Wert x, sodass x ≤ S
    Benutze ⌊S/x⌋ Münzen von Wert x
    S ← S mod x
```

Geld wechseln: Greedy-Algorithmus

Greedy-Ansatz konkreter:

- Werte von m Münzen in einem Array w.
- Es gilt w$[0] >$ w$[1] > \ldots >$ w$[m-1] = 1$.
- Betrag S gegeben.
- Anzahl jeder einzelnen Münze, um S zu wechseln, wird in einem Array num gespeichert.
- num$[i]$ enthält Anzahl der Münzen von Wert w$[i]$.

```
for i ← 0 bis m − 1
    num[i] ← ⌊S / w[i]⌋
    S ← S mod w[i]
```

Greedy-Algorithmus: Allgemeines

Greedy-Algorithmus:

- Eine Lösung wird schrittweise aufgebaut, in jedem Schritt wird das Problem auf ein kleineres Problem reduziert.
- Greedy-Prinzip: Füge jeweils eine lokal am attraktivsten erscheinende Lösungskomponente hinzu.
- Einmal getätigte Entscheidungen werden nicht mehr zurückgenommen.
- Meist einfach zu konstruieren und zu implementieren.
- Kann eine optimale Lösung liefern, muss es i.A. aber nicht.

Greedy-Algorithmus: Optimalität

Optimale Lösung: Für eine Stückelung von 1, 5 und 10 kann gezeigt werden, dass der Greedy-Algorithmus eine optimale Lösung liefert.

Beweis:

- Wir gehen von irgendeiner optimalen Lösung aus.
- Die Lösung kann nicht mehr als vier 1er haben, da fünf davon durch einen 5er ersetzt werden können.
- Die Lösung kann auch nicht mehr als einen 5er haben, da zwei davon durch einen 10er ersetzt werden können.
- Daher muss die Anzahl der 10er im Greedy-Algorithmus und in einem optimalen Algorithmus gleich sein.
- Die Anzahl der restlichen Münzen kann dann maximal 9 ergeben.
- Daher muss man nur den Fall ≤ 9 betrachten.

Greedy-Algorithmus: Optimalität

Beweis (Fortsetzung):

- Jeder Betrag < 5 kann nur durch 1er abgedeckt werden und der optimale Algorithmus und der Greedy-Algorithmus benutzen die gleiche Anzahl von 1er.
- Wenn der Betrag zwischen 5 und 9 (beide inklusive) ist, dann haben der optimale Algorithmus und der Greedy-Algorithmus genau einen 5er und der Rest wird mit 1ern aufgefüllt.
- Der Greedy-Algorithmus liefert daher die gleiche Anzahl an Münzen wie der optimale Algorithmus.

Hinweis: Für Euromünzen kann ähnlich gezeigt werden, dass der Greedy-Algorithmus optimal ist.

Greedy-Algorithmus: Optimalität

Nicht optimal:

- Gegeben sei eine Stückelung von 1, 5, 10, 20, 25.
- Bei dieser Stückelung liefert der Greedy-Algorithmus nicht immer eine optimale Lösung.

Beispiel: Mit $S = 40$.

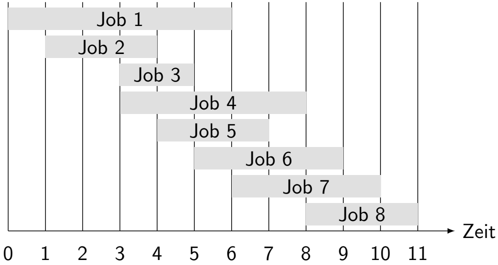
- Greedy-Algorithmus liefert 1 $\times$ 25, 1 $\times$ 10, 1 $\times$ 5.
- Optimale Lösung ist 2 $\times$ 20.

Zeitplanung von Jobs (*Interval Scheduling*)

Interval Scheduling

Interval Scheduling:

- Gegeben: Jobs $j = 1, \ldots, n$.
- Job j startet zum Zeitpunkt s_j und endet zum Zeitpunkt f_j.
- Zwei Jobs sind kompatibel, wenn sie sich nicht überlappen.
- Ziel: Finde größte Teilmenge von paarweise kompatiblen Jobs.



Beispiele: Job 2 und 5 sind kompatibel, Job 2 und 3 sind nicht kompatibel.

Interval Scheduling: Greedy-Algorithmus

Greedy-Ansatz: Betrachte die Jobs in einer natürlichen Ordnung. Wähle einen Job wenn er kompatibel (nicht überlappend) mit den bisher gewählten Jobs ist.

Mögliche Greedy-Strategien:

- [Früheste Startzeit] Berücksichtige Jobs in aufsteigender Reihenfolge von s_j.
- [Früheste Beendigungszeit] Berücksichtige Jobs in aufsteigender Reihenfolge von f_j.
- [Kleinstes Intervall] Berücksichtige Jobs in aufsteigender Reihenfolge von $f_j - s_j$.
- [Wenigste Konflikte] Zähle für jeden Job j die Anzahl c_j der nicht kompatiblen Jobs. Berücksichtige Jobs in aufsteigender Reihenfolge von c_j.

Interval Scheduling: Greedy-Algorithmus

Greedy-Ansatz: Betrachte die Jobs in einer natürlichen Ordnung. Wähle einen Job wenn er kompatibel (nicht überlappend) mit den bisher gewählten Jobs ist.

Gegenbeispiel für früheste Startzeit

Gegenbeispiel für kleinstes Intervall

Gegenbeispiel für wenigste Konflikte

Früheste Beendigungszeit: Gegenbeispiel? Nein!

Interval Scheduling: Greedy-Algorithmus

Greedy-Algorithmus: Berücksichtige Jobs in aufsteigender Reihenfolge der Beendigungszeit.
Wähle einen Job, wenn er kompatibel mit den bisher gewählten Jobs ist.

```
Sortiere Jobs nach Beendigungszeit, sodass f_1 ≤ f_2 ≤ ··· ≤ f_n
A ← ∅
for j ← 1 bis n
    if Job j ist kompatibel zu A
        A ← A ∪ {j}
return A
```

Menge der ausgewählten Jobs

Interval Scheduling: Greedy-Algorithmus

Greedy-Algorithmus: Pseudocode mit angepassten Indexwerten und Array.

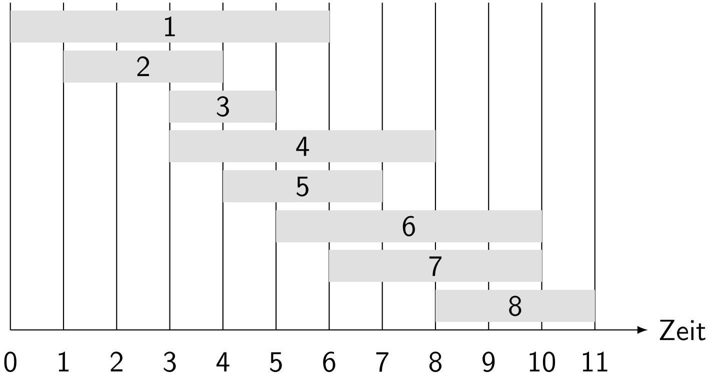
```
Sortiere Jobs nach Beendigungszeit, sodass f_1 ≤ f_2 ≤ ··· ≤ f_n
A ← ∅
t ← 0
for j ← 1 bis n
    if t ≤ s_j
        A ← A ∪ {j}
        t ← f_j
return A
```

Interval Scheduling: Greedy-Algorithmus

Implementierung: Laufzeit in $O(n \log n)$.

- Jobs werden nach Beendigungszeit sortiert und nummeriert. Wenn $f_i \leq f_j$, dann $i < j$. Die Sortierung läuft in $O(n \log n)$.
- Jobs werden vom ersten Job beginnend in der Reihenfolge ansteigender Werte für f_i ausgewählt.
- Sei die Beendigungszeit des aktuellen Jobs t:
  - Dann wird in den nachfolgenden Jobs der erste Job j gesucht, für den gilt: $s_j \geq t$.
  - Dieser Job wird der neue aktuelle Job und die Suche wird von diesem Job aus fortgesetzt.
- Der Greedy-Algorithmus kann in einem Durchlauf realisiert werden, d.h. die Laufzeit ohne Sortieren liegt in $O(n)$.
- Somit liegt die Gesamtlaufzeit in $O(n \log n)$.

Interval Scheduling: Beispiel



Jobs: Nach Beendigungszeit sortiert

Job i	2	3	1	5	4	6	7	8
s_i	1	3	0	4	3	5	6	8
f_i	4	5	6	7	8	10	10	11

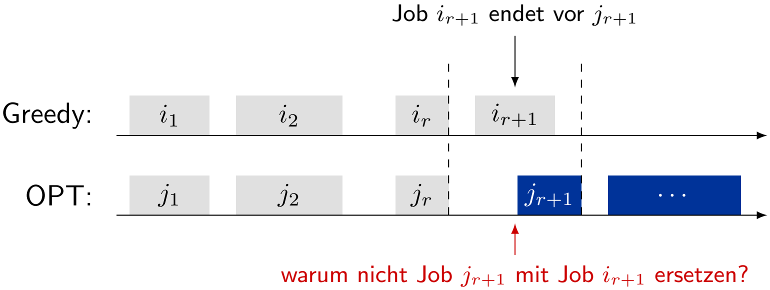
Lösung: Jobs 2, 5 und 8

Interval Scheduling: Analyse

Theorem: Der Greedy-Algorithmus liefert immer eine optimale Lösung.

Beweis: (durch Widerspruch)

- Angenommen, der Algorithmus liefert keine optimale Lösung.
- Sei $i_1, i_2, \ldots, i_k$ die Menge von Jobs, die vom Algorithmus ausgewählt wird.
- Sei $j_1, j_2, \ldots, j_m$ die Menge von Jobs in einer optimalen Lösung mit $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ für größtmögliches r.

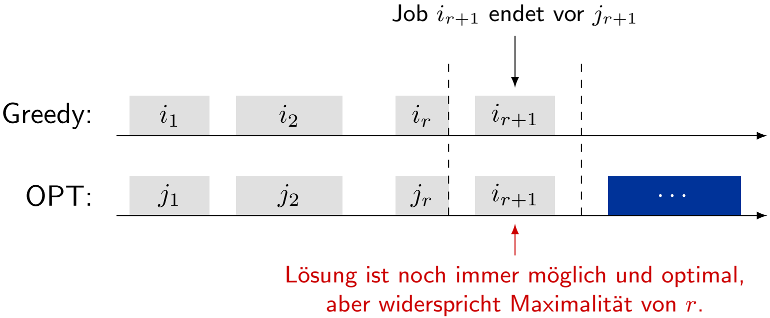


Interval Scheduling: Analyse

Theorem: Der Greedy-Algorithmus liefert immer eine optimale Lösung.

Beweis: (durch Widerspruch)

- Angenommen, der Algorithmus liefert keine optimale Lösung.
- Sei $i_1, i_2, \ldots, i_k$ die Menge von Jobs, die vom Algorithmus ausgewählt wird.
- Sei $j_1, j_2, \ldots, j_m$ die Menge von Jobs in einer optimalen Lösung mit $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ für größtmögliches r.

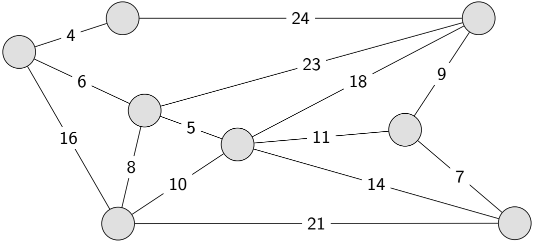


Minimaler Spannbaum

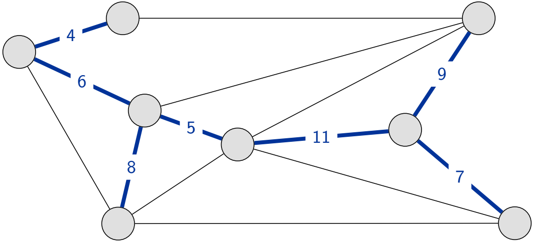
Minimaler Spannbaum

Gegeben: Ein zusammenhängender schlichter Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv} = c_{vu}$ für $e = (u, v) \in E$.

Minimaler Spannbaum: Ein minimaler Spannbaum (*Minimum Spanning Tree, MST*) ist ein Teilgraph $G_T = (V, T)$ von G mit gleicher Knotenmenge und einer Teilmenge der Kanten $T \subseteq E$, sodass er ein aufspannender Baum mit minimaler Summe der Kantengewichte ist.



$G = (V, E)$



$T, \sum_{e \in E} c_e = 50$

MST-Problem

MST-Problem: Finde in einem zusammenhängenden schlichten Graph $G = (V, E)$ mit reellwertigen Kantengewichten c_e einen minimalen Spannbaum, d.h. einen zusammenhängenden, zyklenfreien Untergraphen $G_T = (V, T)$ mit $T \subseteq E$, dessen Kanten alle Knoten aufspannen und für den $cost(T) = \sum_{e \in T} c_e$ so klein wie möglich ist.

Aufwand: Es gibt exponentiell viele Spannbäume und daher wäre ein Brute-Force-Durchprobieren aller Spannbäume nicht effizient.

Lösung: Algorithmen, die in diesem Abschnitt vorgestellt werden.

Anwendungen

Das MST-Problem ist ein fundamentales Problem mit vielen unterschiedlichen Anwendungen:

- Basis für den Entwurf von Netzwerken.
  - Telefonie, Elektrizität, Kabelfernsehen, Computernetze, Straßenverkehrsnetze
- Approximationsalgorithmen für schwere Probleme.
  - Problem des Handlungsreisenden (*Travelling Salesman Problem*), Steinerbaum Problem

Greedy-Algorithmen

Algorithmen:

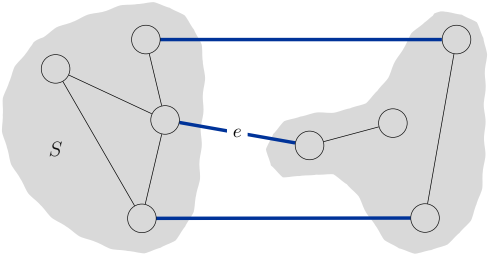
- Algorithmus von Prim: Starte mit einem beliebigen Startknoten s. Füge in jedem Schritt eine billigste Kante e zu T hinzu, die genau einen noch nicht angebundenen Knoten mit dem bisherigen Baum verbindet.
- Algorithmus von Kruskal: Starte mit $T = \emptyset$. Betrachte die Kanten in aufsteigender Reihenfolge ihrer Kosten. Füge Kante e nur dann zu T hinzu, wenn dadurch kein Kreis erzeugt wird.

Beide Algorithmen erzeugen immer einen MST.

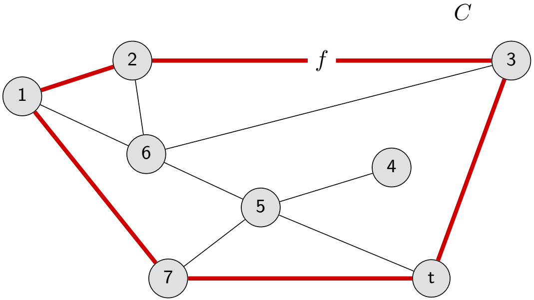
Greedy-Algorithmen: Lemmata

Kantenschnittlemma: Sei S eine beliebige Teilmenge von Knoten und sei e die minimal gewichtete Kante mit genau einem Endknoten in S. Dann enthält der MST die Kante e.

Kreislemma: Sei C ein beliebiger Kreis und sei f die maximal gewichtete Kante in C. Dann enthält der MST f nicht.



e ist im MST

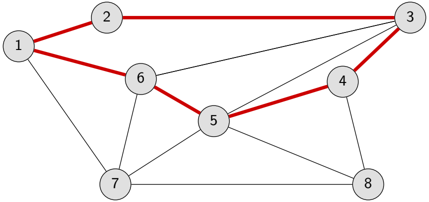


f ist nicht im MST

Vereinfachende Annahme: Alle Kantengewichte sind unterschiedlich, dadurch ist der MST eindeutig.

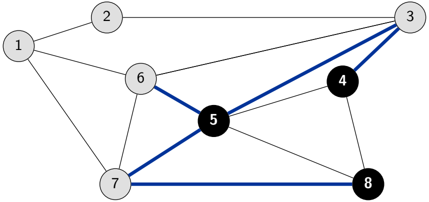
Kreise und Schnitte

Kreis: Ein Kreis ist ein Kantenzug $v_1, v_2, \ldots, v_{k\text{-}1}, v_k$ in dem $v_1 = v_k$, $k \geq 4$, und die ersten $k-1$ Knoten alle unterschiedlich sind. Alternativ kann ein Kreis als Menge $E(C)$ von Kanten der Form a-b, b-c, c-d, …, y-z, z-a gesehen werden.



Kreis $E(C) = \{$1-2, 2-3, 3-4, 4-5, 5-6, 6-1$\}$

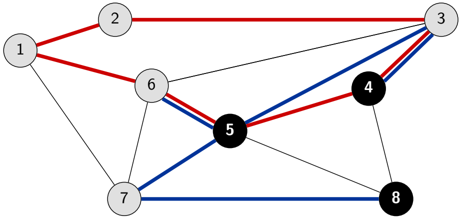
Kantenschnittmenge: Sei S eine Teilmenge der Knoten. Die dazugehörige Kantenschnittmenge D ist die Menge jener Kanten, die genau einen Endpunkt in S haben.



S =$\{$4,5,8$\}$
Schnittmenge $D = \{$5-6, 5-7, 3-4, 3-5, 7-8$\}$

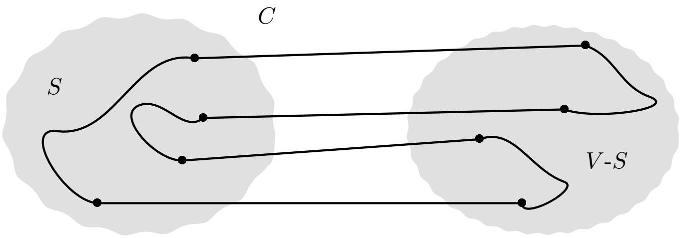
Kreise und Schnitte: Paritätslemma

Behauptung: Ein beliebiger Kreis und eine beliebige Kantenschnittmenge haben eine gerade Anzahl von Kanten gemeinsam.



Kreis $E(C) = \{$1-2, 2-3, 3-4, 4-5, 5-6, 6-1$\}$
Schnittmenge $D = \{$3-4, 3-5, 5-6, 5-7, 7-8$\}$
Durchschnitt $= \{$3-4, 5-6$\}$

Beweis: (durch Bild)



Beweis des Kantenschnittlemmas

Kantenschnittlemma: Sei S eine beliebige Teilmenge von Knoten und sei e die minimal gewichtete Kante mit genau einem Endknoten in S. Dann enthält der MST T^* die Kante e.

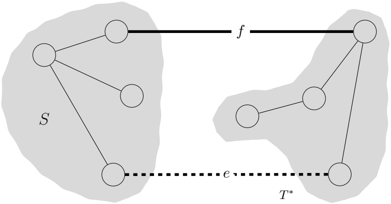
Annahme für Beweis: Alle Kantengewichte c_e sind unterschiedlich, vereinfacht Beweis.

Hinweis: Man kann zu allen Kosten kleine Störwerte hinzufügen, um die Annahme, dass alle Kanten unterschiedliches Gewicht haben müssen, zu vermeiden.

Beweis des Kantenschnittlemmas

Beweis: (Austauschargument)

- Angenommen e gehört nicht zu T^*.
- Das Hinzufügen von e zu T^* erzeugt einen Kreis $E(C)$ in T^*.
- Kante e ist sowohl im Kreis $E(C)$ als auch in der Schnittmenge D von S.
- Paritätslemma $\Rightarrow$ es existiert eine andere Kante, sagen wir f, die sich sowohl in $E(C)$ als auch in D befindet.
- $T' = T^* \cup \{e\} - \{f\}$ ist auch ein aufspannender Baum.
- Da $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- Das ist ein Widerspruch zur Annahme, dass T^* minimal ist. $\square$



Beweis des Kreislemmas

Kreislemma: Sei $E(C)$ ein beliebiger Kreis in G und sei f die maximal gewichtete Kante in $E(C)$. Dann enthält kein MST die Kante f.

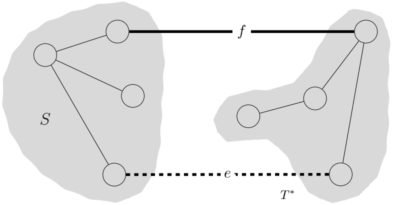
Annahme für Beweis: Alle Kantengewichte c_e sind unterschiedlich, vereinfacht Beweis.

Hinweis: Man kann zu allen Kosten kleine Störwerte hinzufügen, um die Annahme, dass alle Kanten unterschiedliches Gewicht haben müssen, zu vermeiden.

Beweis des Kreislemmas

Beweis: (Austauschargument)

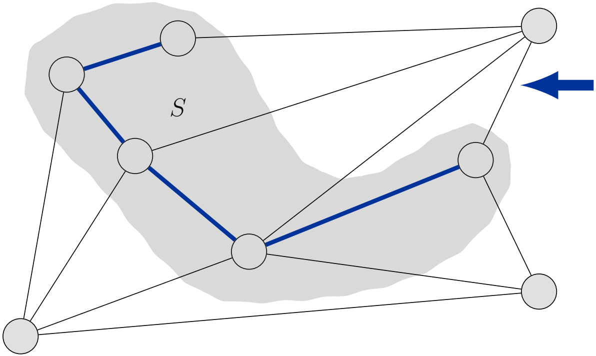
- Angenommen f gehört zu T^*
- Löschen von f aus T^* erzeugt eine Teilmenge S von Knoten in T^*.
- Kante f ist sowohl im Kreis $E(C)$ als auch in der Schnittmenge D von S.
- Paritätslemma $\Rightarrow$ es existiert eine andere Kante, sagen wir e, die sich sowohl in $E(C)$ als auch in D befindet.
- $T' = T^* \cup \{e\} - \{f\}$ ist auch ein aufspannender Baum.
- Da $c_e < c_f$, cost$(T') <$ cost(T^*).
- Das ist ein Widerspruch zur Annahme, dass T^* minimal ist. $\square$



Algorithmus von Prim

Algorithmus von Prim: [Jarnìk 1930, Dijkstra 1957, Prim 1959]

- Initialisiere S mit einem beliebigen Knoten.
- Wende das Kantenschnittlemma auf S an.
- Füge die minimal gewichtete Kante e in der Schnittmenge von S zu T hinzu und füge den Knoten u (Endknoten von e der sich noch nicht in S befindet) zu S hinzu.

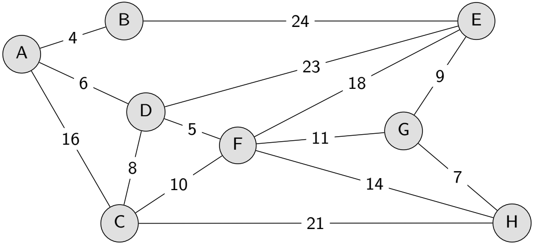


Algorithmus von Prim: Implementierung

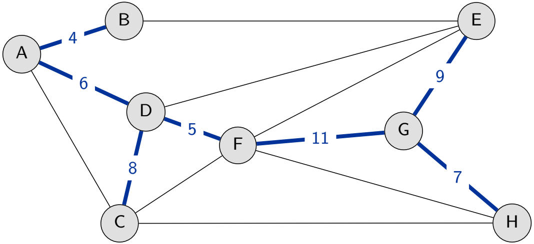
Annahme: Alle Kantengewichte sind unterschiedlich.

```
Prim(G, c):
foreach (v ∈ V)
    A[v] ← ∞
Initialisiere eine leere Priority Queue Q
foreach (v ∈ V)
    Füge v in Q ein
S ← ∅
while Q ist nicht leer
    u ← entnehme minimales Element aus Q
    S ← S ∪ {u}
    foreach Kante e = (u, v) inzident zu u
        if v ∉ S und c_e < A[v]
            Verringere die Priorität A[v] auf c_e
```

Algorithmus von Prim: Beispiel



$$G = (V, E)$$



$$T, \sum_{e \in E} c_e = 50$$

Start:

- Start bei A (willkürlich gewählt, alle Knoten gleiche Priorität)
- Priority Queue zu Beginn: A, B, C, D, E, F, G, H

Ausgewählt	Resultierende Priority Queue	Knotenmenge S	Gewicht
A	B, D, C, E, F, G, H	A	0
B	D, C, E, F, G, H	A, B	4
D	F, C, E, G, H	A, B, D	10
F	C, G, H, E	A, B, D, F	15
C	G, H, E	A, B, D, F, C	23
G	H, E	A, B, D, F, C, G	34
H	E	A, B, D, F, C, G, H	41
E		A, B, D, F, C, G, H, E	50

Algorithmus von Prim: Analyse

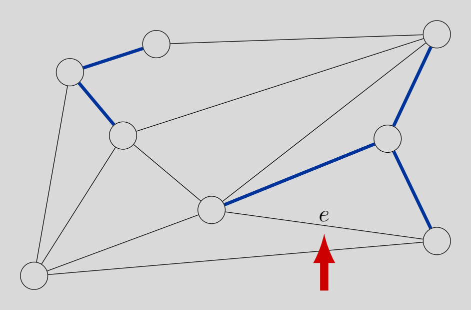
Implementierung: Benutze eine Priority Queue wie bei Dijkstra.

- Verwalte eine Menge von bearbeiteten Knoten S.
- Verwalte jeden unbearbeiteten Knoten v mit Kosten A$[v]$ in der Priority Queue.
- A$[v]$ sind die Kosten der billigsten Kante von v zu einem Knoten in S.
- Laufzeit in $O(n^2)$, wenn die Priority Queue mit einem Array implementiert ist.
- Laufzeit in $O(m \log n)$ mit einem binären Heap (Min-Heap).

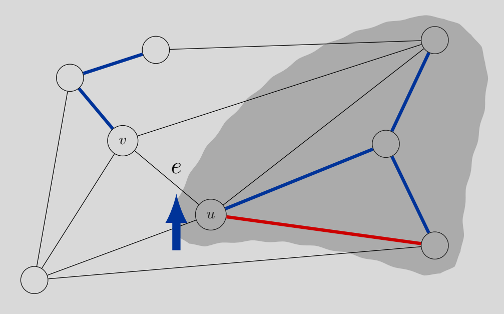
Algorithmus von Kruskal

Algorithmus von Kruskal: [Kruskal, 1956]

- Bearbeite Kanten in aufsteigender Reihenfolge der Kantengewichte.
- Fall 1: Wenn das Hinzufügen von e zu T einen Kreis erzeugt, verwerfe e gemäß des Kreislemmas.
- Fall 2: Sonst füge $e = (u, v)$ in T gemäß des Kantenschnittlemmas ein.



Fall 1



Fall 2

Algorithmus von Kruskal: Implementierung

Implementierung:

```
Kruskal(G, c):
Sortiere Kantengewichte so, dass c_1 ≤ c_2 ≤ ··· ≤ c_m
T ← ∅
foreach (u ∈ V) erzeuge eine einelementige Menge mit u
for i ← 1 bis m
    (u, v) = e_i
    if  u und v sind in verschiedenen Mengen
        T ← T ∪ {e_i}
        Vereinige die Mengen mit u und v
return T
```

- *sind u und v in unterschiedlichen Zusammenhangskomponenten?*
- *Vereinige zwei Komponenten*

Algorithmus von Kruskal: Implementierung

Sind u und v in verschiedenen Zusammenhangskomponenten?

Einfache Möglichkeit: Verwende Tiefen- oder Breitensuche

Effizienter: Benutze die sogenannte Union-Find-Datenstruktur.

- Verwalte die Teilmenge aller Knoten für jede Zusammenhangskomponente.
- $O(m \log n)$ für die Sortierung ($m \leq n^2 \Rightarrow \log m$ ist $O(\log n)$).

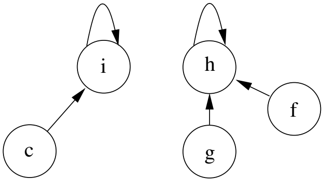
Union-Find-Datenstruktur: Abstrakter Datentyp

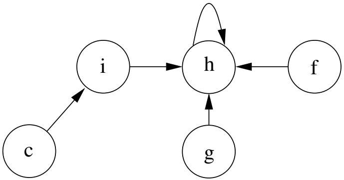
Abstrakter Datentyp: Dynamische Disjunkte Mengen (DDM)

Familie $S = \{S_1, S_2, \ldots, S_k\}$ disjunkter Teilmengen einer Menge M.
Jedes S_i hat einen Repräsentanten.

- makeset(v): Erzeugt eine Menge $\{v\} = S_v$; v ist Repräsentant von S_v
- union(v, w): Vereinigt Mengen S_v und S_w deren Repräsentanten v und w sind; neuer Repräsentant ist ein beliebiges $u \in S_v \cup S_w$
- findset(v): Liefert Repräsentanten der Menge S mit $v \in S$

Die Union Find Datenstruktur





v	a	b	c	d	e	f	g	h	i
parent$[v]$	a	b	i	d	e	h	h	h	h

Die Union Find Datenstruktur: Implementierung

Einfache Implementierung:

- makeset (v):

$$\texttt{parent}[v] = v$$

- union (v, w):

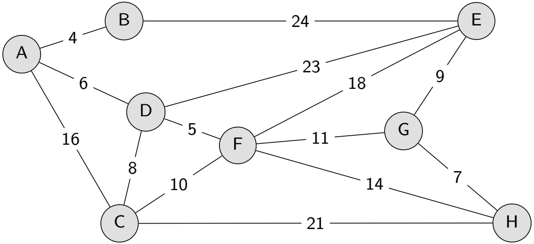
$$\texttt{parent}[v] = w$$

- findset (v):

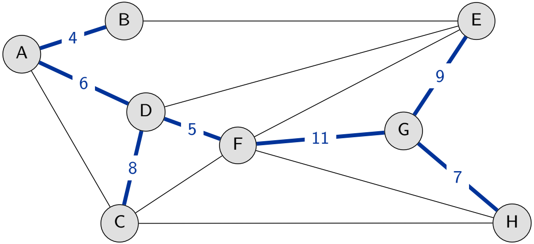
```
h = v
while parent[h] ≠ h
    h = parent[h]
return h;
```

Laufzeit: Mit einer verbesserten Implementierung kann eine in der Praxis nahezu konstante Laufzeit für jede der drei Operationen erreicht werden.

Algorithmus von Kruskal: Beispiel



$$G = (V, E)$$



$$T, \sum_{e \in E} c_e = 50$$

Start: Kanten sortiert nach Gewicht (kleinstes zuerst): (A,B), (D,F), (A,D), (G,H), (C,D), (E,G), (C,F), (F,G), (F,H), (A,C) ...

Mengen	Kante	Hinzu?	T
{**A**},{**B**},{C},{D},{E},{F},{G},{H}	(A,B)	Ja	{(A,B)}
{A,B},{C},{**D**},{E},{**F**},{G},{H}	(D,F)	Ja	{(A,B), (D,F)}
{**A**,B},{C},{**D**,F},{E},{G},{H}	(A,D)	Ja	{(A,B), (D,F), (A,D)}
{A,B,D,F},{C},{E},{**G**},{**H**}	(G,H)	Ja	{(A,B), (D,F), (A,D), (G,H)}
{A,B,**D**,F},{**C**},{E},{G,H}	(C,D)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D)}
{A,B,C,D,F},{**E**},{**G**,H}	(E,G)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G)}
{A,B,**C**,D,**F**},{E,G,H}	(C,F)	Nein	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G)}
{A,B,C,D,**F**},{E,**G**,H}	(F,G)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G), (F,G)}
{A,B,C,D,E,F,G,H}	...	...	...

- *Ab jetzt werden keine weiteren Kanten mehr aufgenommen!*

Kruskal und Prim im Vergleich

Laufzeit von Kruskal:

- Union-Find-Operation ist praktisch in konstanter Zeit möglich, d.h. der zweite Teil des Kruskal-Algorithmus hat nahezu lineare Laufzeit.
- Der Gesamtaufwand wird nun durch das Kantensortieren bestimmt und ist somit $O(m \log n)$.

Laufzeit von Prim:

- Wird als Priority Queue ein klassischer Heap verwendet, dann ist der Gesamtaufwand $O(m \log n)$.
- Wird ein sogenannter Fibonacci-Heap verwendet, so reduziert sich die Laufzeit auf $O(m + n \log n)$.

Anwendung in der Praxis:

- Für dichte Graphen ($m = \Theta(n^2)$) ist Prims Algorithmus besser geeignet.
- Für dünne Graphen ($m = \Theta(n)$) ist Kruskals Algorithmus besser geeignet.