Angewandte Mathematik für das Web

Entwurf — Skriptum 2019

Georg Grasegger, Jan Legerský
(Helene Laimer, Mario Neumüller)

Juni 2019

Inhaltsverzeichnis

1 Grundlagen

1.1 Ungerichtete Graphen

Notation 1.1.
Sei A eine Menge und sei $k \in \mathbb{N}$. Dann bezeichnen wir mit $[A]^k$ die Menge aller k-elementigen Teilmengen von A.

Definition 1.2.
Ein (ungerichteter) Graph *(engl. undirected graph) G ist ein 2-Tupel $G = (V, E)$, wobei V eine endliche Menge ist und $E \subseteq [V]^2$. V heißt* Knotenmenge*, die Elemente aus V sind Knoten (engl. vertices). E ist die* Kantenmenge *und deren Elemente heißen Kanten (engl. edges).*

Sei $e = \{u, v\}$. Wir sagen „die Kante e verbindet die Knoten u und v“ oder „die Kante e hat u und v als Endknoten“.

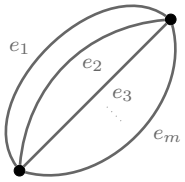
In manchen Fällen benötigen wir eine allgemeinere Definition in der mehrfache Kanten und Schleifen erlaubt sind.

Definition 1.3.
Ein ungerichteter allgemeiner Multigraph *G ist ein Tripel $G = (V, E, i)$ mit*

1. *V, E zwei endliche Mengen.*
2. *$i \colon E \to [V]^2 \cup [V]^1$, das heißt für $e \in E$:*
   - *$i(e) = \{u, v\}$, mit $u \neq v$, $u, v \in V$ oder*
   - *$i(e) = \{v\}$, mit $v \in V$.*

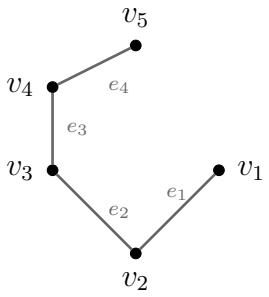
Ist $i(e) = \{u, v\}$, sagt man „die Kante e verbindet u und v“. Ist $i(e) = \{v\}$, dann sagt man „die Kante e verbindet v mit sich selbst“ (Schleife, engl. loop). Falls i injektiv ist, das heißt es gibt keine Mehrfachkanten und falls es außerdem keine Schleifen gibt, also $i \colon E \to [V]^2$, spricht man von einem (schlichten, ungerichteten) Graphen. Statt schlicht wird oft auch die Bezeichnung einfach *(engl. simple graph) verwendet.*

Im Folgenden gehen wir meist davon aus, dass unsere Graphen einfach sind.



$i(e_1) = \ldots = i(e_m) = \{u, v\}$ $\qquad$ $i(e_1) = i(e_2) = \{v\}$

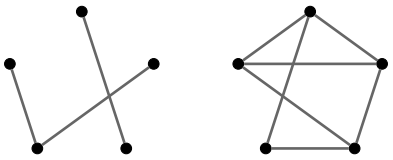
Definition 1.4.
Sei $G = (V, E)$ ein Graph. Zwei Knoten $u, v \in V$ mit $u \neq v$ heißen benachbart (adjazent, engl. adjacent), falls es eine Kante gibt, die u und v verbindet. Wir schreiben $u \sim v$. Zwei Kanten e_1, e_2 mit $e_1 \neq e_2$ heißen benachbart (inzident, engl. incident), falls es einen Knoten gibt, der auf beiden Kanten liegt.



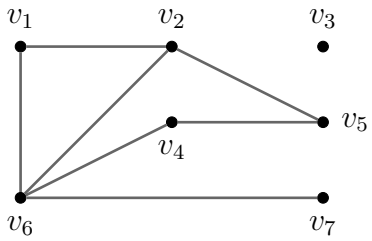
v_4 und v_5 sind benachbart, $v_4 \sim v_5$
e_1 und e_2 sind benachbart
e_2 und e_4 sind nicht benachbart
v_1 und v_5 sind nicht benachbart, $v_1 \not\sim v_5$

Definition 1.5.
Sei $G = (V, E)$ ein Graph. $|V|$ ist die Ordnung *(engl. order) von G und $|E|$ ist die Größe (engl. size) von G. Klarerweise gilt: $|E| \leq \binom{|V|}{2} = |[V]^2|$.*

$|\cdot|$ wird hier als Schreibweise für die Kardinalität verwendet.



Definition 1.6.
Sei $G = (V, E)$ ein Graph und sei $v \in V$. Der Knotengrad *(engl. vertex degree)* $\deg(v)$ *ist die Anzahl der Kanten, auf denen v liegt. Die Nachbarschaftsmenge von v ist definiert durch $N(v) = \{u \in V : u \text{ ist benachbart zu } v\}$. Klarerweise gilt $N(v) \subseteq V$. Es gilt $|N(v)| = \deg(v)$ für alle $v \in V$. Ein Knoten mit leerer Nachbarschaftsmenge,* $\deg(v) = 0$, *heißt* isolierter *Knoten (engl. isolated). $\delta(G) := \min_{v \in V} \deg(v)$ heißt minimaler Knotengrad und $\Delta(G) := \max_{v \in V} \deg(v)$ heißt maximaler Knotengrad. G heißt r-regulär (engl. regular), falls $\deg(v) = r$ für alle $v \in V$.*



$d(v_1) = 2$, $d(v_2) = 3$, $d(v_3) = 0$, $d(v_4) = 2$, $d(v_5) = 2$, $d(v_6) = 4$, $d(v_7) = 1$
$N(v_1) = \{v_2, v_6\}$, $N(v_2) = \{v_1, v_5, v_6\}$
$\delta(G) = 0$
$\Delta(G) = 4$

Satz 1.7.
Sei $G = (V, E)$ ein Graph. Dann gilt

$$\sum_{v \in V} \deg(v) = 2\,|E|\,.$$

Beweis. $\deg(v)$ ist die Anzahl aller Kanten, auf denen v liegt. In $\sum_{v \in V} \deg(v)$ wird also jede Kante genau zweimal gezählt. □

Korollar 1.8. (Handshaking Lemma)
Sei $G = (V, E)$ ein Graph. Die Anzahl der Knoten mit ungeradem Knotengrad ist gerade.

Beweis. Wir wissen von Satz 1.7:

$$\underbrace{2\,|E|}_{\text{gerade}} = \sum_{v \in V} \deg(v) = \underbrace{\sum_{\substack{v \in V \\ \deg(v) \text{ gerade}}} \deg(v)}_{\text{gerade}} + \sum_{\substack{v \in V \\ \deg(v) \text{ ungerade}}} \deg(v).$$

Daraus folgt:

$$\sum_{\substack{v \in V \\ \deg(v) \text{ ungerade}}} \deg(v)$$

ist gerade, woraus man direkt schließen kann, dass die Anzahl der Knoten mit ungeradem Knotengrad, $|\{v \in V \mid \deg(v) \text{ ungerade}\}|$, gerade ist. □

1.2 Gerichtete Graphen

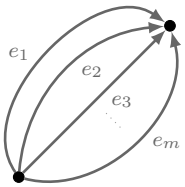
Definition 1.9.
Ein gerichteter (allgemeiner) Graph G *(engl. directed graph) ist ein Tupel $G = (V, E)$ mit*

1. *V, E zwei endliche Mengen.*

2. *$E \subseteq V \times V$, das heißt für $e \in E$:*
   - *$e = (u, v)$, mit $u \neq v$, $u, v \in V$ oder*
   - *$e = (v, v)$, mit $v \in V$. Die Kante ist also eine Schleife.*

Eine Kante $e = (u, v)$ bedeutet, dass die Knoten u und v durch die gerichtete Kante e verbunden werden, wobei u als *Startknoten* und v als *Endknoten* der Kante bezeichnet wird. Es wird u auch *Vorgänger* von v und umkehrt v als *Nachfolger* von u bezeichnet. Für die Definition des Grades eines Knoten müssen wir nun die Richtung der Kanten berücksichtigen.

Definition 1.10.
Unter Eingangsgrad *(engl. indegree) eines Knoten v verstehen wir die Anzahl der Kanten, für die v Endknoten ist. Wenn der Graph keine Mehrfachkanten hat, ist das gleich der Anzahl der Vorgänger von v. Unter* Ausgangsgrad *(engl. outdegree) eines Knoten v verstehen wir die Anzahl der Kanten für die v Startknoten ist. Wenn der Graph keine Mehrfachkanten hat, ist das gleich der Anzahl der Nachfolger von v.*



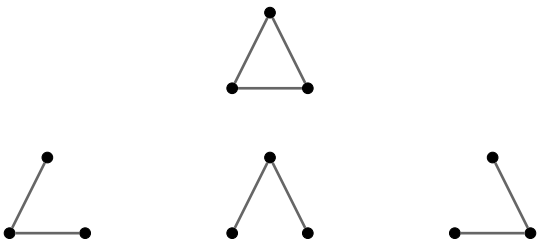
$i(e_1) = \ldots = i(e_m) = (u, v)$

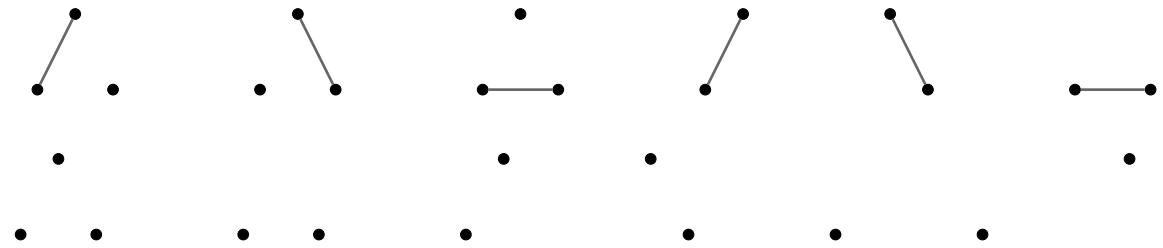
$i(e_1) = i(e_2) = (v, v)$

1.3 Unterstrukturen in Graphen

Für eine detaillierte Beschreibung von Graphen ist es notwendig, deren Unterstrukturen genauer zu beschreiben.

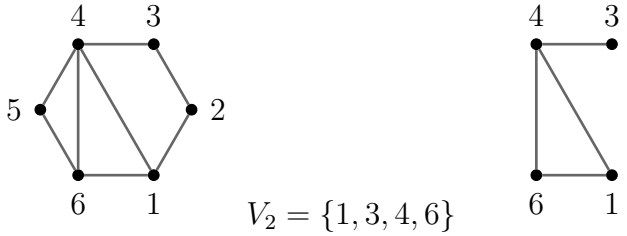
Definition 1.11.
Ein Graph $G_2 = (V_2, E_2)$ heißt Teilgraph (Untergraph, engl. subgraph) eines Graphen $G_1 = (V_1, E_1)$, falls $V_2 \subseteq V_1$ und $E_2 \subseteq E_1$.





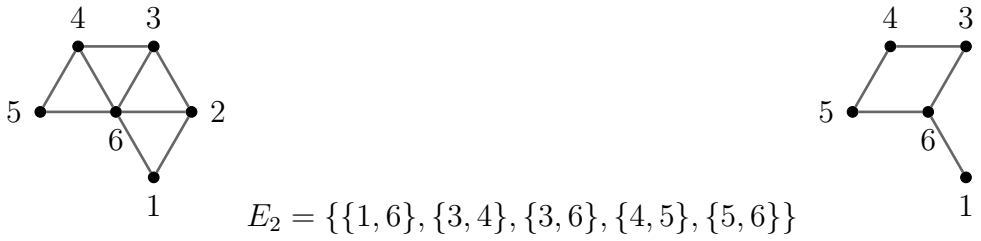
Definition 1.12.
Sei $G = (V_1, E_1)$ ein Graph und sei $V_2 \subseteq V_1$. Der Teilgraph $\langle V_2 \rangle := (V_2, [V_2]^2 \cap E_1)$ heißt knoteninduzierter Teilgraph *(genauer: der von der Knotenmenge V_2 induzierte Teilgraph; engl. vertex-induced subgraph).*

In Worten: Man wählt eine Teilmenge V_2 von V_1 aus und übernimmt alle Kanten von G zwischen Knoten aus V_2.



Definition 1.13.
Sei $G = (V_1, E_1)$ ein Graph und sei $E_2 \subseteq E_1$. Der Teilgraph $\langle E_2 \rangle := (V_2, E_2)$ mit $V_2 = \{v \in V_1 :$ es gibt $k \in E_2$ mit v ist Endknoten von $k\}$ heißt kanteninduzierter Teilgraph *(genauer: durch die Kantenmenge E_2 induzierter Teilgraph; engl. edge-induced subgraph).*

In Worten: Man wählt eine Teilmenge E_2 von Kanten aus G aus und nimmt als Knotenmenge alle Endknoten von Kanten aus E_2.



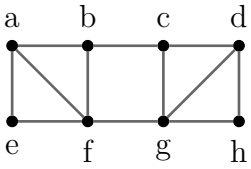
Definition 1.14.
Sei $G = (V, E)$ ein Graph. Ein Teilgraph $H = (V_1, E_1)$ heißt spannender Teilgraph von G, falls $V_1 = V$ ist.

1.4 Wanderungen, Wege und Pfade in Graphen

Definition 1.15.
Sei $G = (V, E)$ ein Graph und seien $u, v \in V$. Eine $(u - v)$-Wanderung W in G (engl. walk) ist eine endliche Abfolge von Knoten und Kanten:
$W = (v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n)$ *mit*

1. $v_i \in V \ \forall i \in \{0, \ldots, n\}$
2. $e_i \in E \ \forall i \in \{1, \ldots, n\}$
3. $e_i = \{v_{i-1}, v_i\} \ \forall i \in \{1, \ldots, n\}$
4. $v_0 = u, v_n = v$.

Der Knoten u heißt Anfangsknoten von W, v heißt Endknoten von W. Die Länge $|W|$ von W ist die Kantenanzahl n.



$W = (a, \{a, b\}, b, \{b, f\}, f, \{f, e\}, e, \{e, a\}, a, \{a, f\}, f, \{f, g\}, g, \{g, h\}, h)$
$W = (a, b, f, e, a, f, g, h)$

Bemerkung 1.16.
In einer Wanderung können sowohl Knoten als auch Kanten mehrfach vorkommen. Da in einem Graphen keine Mehrfachkanten beziehungsweise Schlingen existieren, genügt es die Wanderung als Knotenfolge zu beschreiben, also $W = (u_0, u_1, \ldots, u_n)$.

Definition 1.17.
Eine Wanderung heißt geschlossen (engl. closed), falls Anfangs- und Endpunkt übereinstimmen ($u = v$). Andernfalls heißt eine Wanderung offen (engl. open). Für einen Knoten $v \in V$ heißt $W = (v)$ triviale Wanderung. Eine $(u - v)$-Wanderung heißt Weg (engl. trail), falls alle Kanten in W verschieden sind. Eine $(u - v)$-Wanderung heißt Pfad (engl. path), falls alle Knoten in W verschieden sind.

Bemerkung 1.18.
Nach unserer Definition existieren keine geschlossenen Pfade, außer dem trivialen Pfad. Es gilt Pfad $\Rightarrow$ Weg $\Rightarrow$ Wanderung.

Definition 1.19.
Ein geschlossener Weg $W = (u_1, u_2, \ldots, u_n, u_1)$, $u_i \in V$ heißt Zyklus oder Kreis (engl. cycle), falls $u_i \neq u_j$ für alle $1 \leq i \neq j \leq n$.

Wanderungen, Wege, Pfade und Zyklen lassen sich in ähnlicher Weise auch für gerichtete Graphen formulieren. Für den unten definierten Begriff des Zusammenhangs müssen verschiedene neue Begriffe eingeführt werden.

Definition 1.20.
Sei $G = (V, E)$ ein Graph. G heißt zusammenhängend (engl. connected), falls es zu je zwei Knoten aus V eine Wanderung von dem einen zum anderen gibt. Andernfalls ist G nicht zusammenhängend.

Die Menge der Wege bezeichnen wir mit $\mathcal{W}$. Es gilt $\mathcal{W} = \bigcup_{j=0}^{|V|} V^j$. Auf der Menge an Wegen definieren wir eine Operation $\circ$, die zwei Wege aneinanderfügt.

Definition 1.21.
Seien $(v_1, \ldots, v_k)$ und $(w_1, \ldots, w_j)$ zwei Wege in einem Graphen mit $v_k \sim w_1$. Dann ist

$$(v_1, \ldots, v_k) \circ (w_1, \ldots, w_j) := (v_1, \ldots, v_k, w_1, \ldots, w_j)$$

Manchmal benötigen wir auch das Konstrukt eines leeren Wegs (engl. empty), also ein Weg/Pfad ohne einen Knoten, und schreiben hierfür $(\,)$.

Sei $G = (V, E)$ ein Graph. Wir definieren eine Äquivalenzrelation auf der Knotenmenge: Seien $u, v \in V$

$$uRv \Leftrightarrow \text{ Es existiert eine Wanderung von } u \text{ nach } v.$$

Das ist tatsächlich eine Äquivalenzrelation:

- reflexiv: uRu da $W = (u)$
- symmetrisch: trivial, da man Wanderungen auch „von hinten lesen“ kann
- transitiv: uRv und $vRw \Rightarrow uRw$.

Diese Äquivalenzrelation teilt die Knotenmenge in disjunkte Äquivalenzklassen ein: $V = V_1 \cup \cdots \cup V_k$, wobei $V_i \cap V_j = \{\,\}$ für alle $i \neq j$. Die knoteninduzierten Teilgraphen $\langle V_1 \rangle, \ldots, \langle V_k \rangle$ heißen Zusammenhangskomponenten (engl. connected component) von G. Alle $\langle V_i \rangle$ sind zusammenhängend. G ist zusammenhängend genau dann, wenn es aus genau einer Zusammenhangskomponente besteht.

Definition 1.22.
Sei $G = (V, E)$ ein zusammenhängender Graph. Eine Kante $e \in E$ heißt Brücke (engl. bridge), wenn der Graph $(V, E \setminus \{e\})$ nicht zusammenhängend ist.

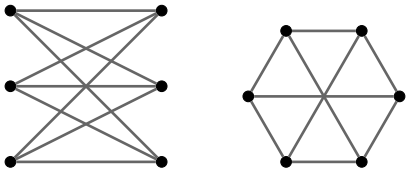
Sei $G = (V, E)$ ein Graph. Eine Kante $e \in E$ heißt Brücke, wenn der Graph $(V, E \setminus \{e\})$ mehr Zusammenhangskomponenten als G hat.

Satz 1.23.
Sei $G = (V, E)$ ein Graph und seien $u, v \in V$ und sei weiters W eine $(u-v)$-Wanderung in G. Dann gibt es auch einen $(u - v)$-Pfad in G.

1.5 Isomorphie von Graphen

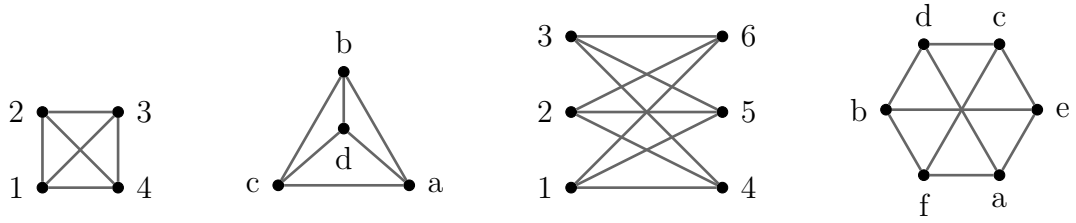
Beispiel 1.24.
Verschiedene zeichnerische Darstellungen eines Graphen.

$$G = (V, E),$$
$$V = \{1, 2, 3, 4, 5, 6\},$$
$$E = \{\{1, 4\}, \{1, 5\}, \{1, 6\}, \{2, 4\}, \{2, 5\}, \{2, 6\}, \{3, 4\}, \{3, 5\}, \{3, 6\}\}.$$



Es stellt sich die Frage, ob sich G so in die Ebene so zeichnen lässt, dass sich Kanten ausschließlich in Knoten treffen. Darauf wollen wir an dieser Stelle aber nicht näher eingehen. Hier betrachten wir die Frage, wie wir feststellen können, ob zwei Beschreibungen von Graphen tatsächlich den gleichen oder zumindest einen strukturgleichen Graphen darstellen.

Motivation der nächsten Definition



G_1 G_1' G_2 G_2'

In den obigen Beispielen sind die beiden Graphen G_1 und G_1' beziehungsweise G_2 und G_2' zwar nicht identisch aber strukturgleich.

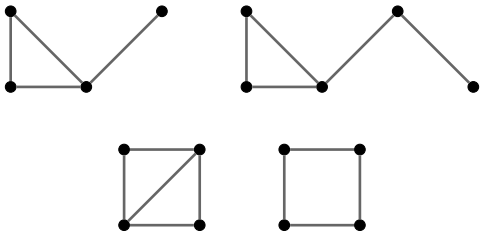
Seien im zweiten Beispiel $V_2 = \{1, \ldots, 6\}$ und $V_2' = \{a, \ldots, f\}$. Wir betrachten die Abbildung $f\colon V_2 \to V_2'$ mit $1 \mapsto a, 2 \mapsto c, 3 \mapsto e, 4 \mapsto b, 5 \mapsto d, 6 \mapsto f$.

Definition 1.25.
Zwei Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ heißen isomorph (engl. isomorphic), falls es eine Abbildung $f\colon V_1 \to V_2$ gibt, sodass

1. *f bijektiv*
2. *f „erhält Nachbarschaftsverhältnisse“, das heißt für alle $u, v \in V_1$ gilt $u \sim v \Leftrightarrow f(u) \sim f(v)$.*

G_1 und G_2 haben also bis auf eine Umbenennung der Knoten die gleiche Struktur.

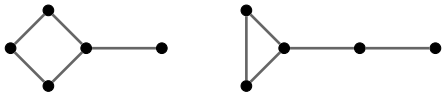
Wir können für das obige Beispiel leicht nachprüfen, dass G_1 und G_1' beziehungsweise G_2 und G_2' isomorph sind. Die folgenden Graphen sind jedoch nicht isomorph.



Satz 1.26.
Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei isomorphe Graphen. Dann gelten:

- $|V_1| = |V_2|$
- $|E_1| = |E_2|$
- *für alle $v \in V_1$ gilt* $\deg(f(v)) = \deg(v)$ *für Isomorphismen $f\colon V_1 \to V_2$.*

Achtung: Die Tatsache, dass zwei Graphen gleiche Knoten- und Kantenzahl und gleiche Knotengrade aufweisen, reicht nicht aus, um schließen zu können, dass diese isomorph sind, wie das folgende Beispiel zeigt.



1.6 Darstellung eines Graphen

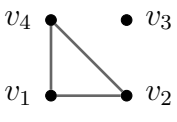
Bisher haben wir einen Graphen immer durch die Knotenmenge und Kantenmenge bestimmt. Neben dieser Möglichkeit gibt es noch andere eindeutige Darstellungen von Graphen, etwa über die sogenannte Adjazenzmatrix.

Definition 1.27.
Sei $G = (V, E)$ ein Graph mit Knoten $v_1, \ldots, v_n$ und Kanten $e_1, \ldots, e_m$. Dann heißt die $(n \times n)$-Matrix $A = (a_{ij})_{i,j=1,\ldots,n}$ mit

$$a_{ij} = \begin{cases} 1 & \text{falls } \{v_i, v_j\} \in E \\ 0 & \text{sonst} \end{cases}$$

eine Adjazenzmatrix *(engl. adjacency matrix). Für gerichtete Graphen ist die Matrix nicht symmetrisch.*

Beispiel 1.28.
Adjazenzmatrix für einen einfachen Graph:



$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Eine andere Nummerierung der Knoten führt zu einer anderen Adjazenzmatrix, welche sich aber aus der ursprünglichen durch eine Permutation von Zeilen und Spalten erzeugen lässt.

Adjazenzmatrizen von ungerichteten Graphen sind immer symmetrisch und bei einfachen Graphen sind die Einträge in der Hauptdiagonale alle Null. Zeilen- und Spaltensummen entsprechen den Knotengraden der jeweiligen Knoten.

Eine weitere Möglichkeit Graphen darzustellen bietet die sogenannte Inzidenzmatrix.

Definition 1.29.
Sei $G = (V, E)$ ein Graph mit Knoten $v_1, \dots, v_n$ und Kanten $e_1, \dots, e_m$. Dann heißt die $(n \times m)$-Matrix $B = (b_{ij})_{\substack{i=1,\dots,n \\ j=1,\dots,m}}$ mit

$$b_{ij} = \begin{cases} 1 & \textit{falls } v_i \textit{ liegt auf } e_j \\ 0 & \textit{sonst} \end{cases}$$

eine Inzidenzmatrix (engl. incidence matrix).

Sei $B^\top$ die transponierte Matrix zu B. Dann gilt für die $(n \times n)$ Matrix $M = BB^\top = (m_{i,j})_{i,j=1,\dots,n}$, dass

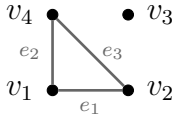
$$m_{ij} = \begin{cases} d(v_i) & \text{falls } i = j \\ a_{ij} & \text{sonst.} \end{cases}$$

Also:

$$M = \begin{pmatrix} d(v_1) & 0 & \cdots & 0 \\ 0 & d(v_2) & 0 & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \cdots & \cdots & d(v_n) \end{pmatrix} + A,$$

wobei $A = (a_{ij})_{i,j=1,\dots,n}$ die Adjazenzmatrix ist.

Beispiel 1.30.
Inzidenzmatrix für einen einfachen Graph:



$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

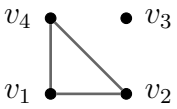
Vor allem wenn ein Graph relativ wenig Kanten im Vergleich zur Anzahl der Knoten hat, benötigt die Darstellung mittels einer Matrix viel Speicherplatz. Ein Ausweg dafür kann in manchen Fällen eine Listendarstellung sein.

Definition 1.31.
Sei $G = (V, E)$ ein Graph mit Knoten $v_1, \dots, v_n$ und Kanten $e_1, \dots, e_m$. Dann heißt der Vektor $L = (L_i)_{i=1,\dots,n}$ Adjazenzliste *(engl. adjacency list), wobei*

$$L_i = \{v \in V \mid v \sim v_i\} \ .$$

Für gerichtete Graphen werden statt der Nachbarn die Nachfolger in die Liste geschrieben.

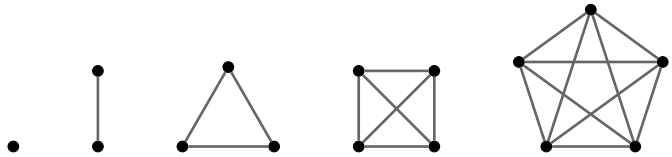
Beispiel 1.32.
Adjazenzliste für einen einfachen Graph:



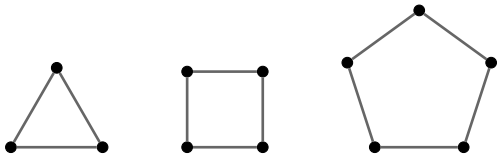
$$(\{v_2, v_4\}, \{v_1, v_4\}, \{\}, \{v_1, v_2\})$$

1.7 Wichtige spezielle Graphen

1. **Der vollständige Graph K_n mit n Knoten** (engl. complete graph)
   $V = \{1, \dots, n\}$, $E = [V]^2$



2. **Der n-Zyklus C_n**
   $V = \{1, \dots, n\}, n \geq 3$, $E = \{\{i, i+1\} \colon 1 \leq i \leq n-1\} \cup \{\{1, n\}\}$

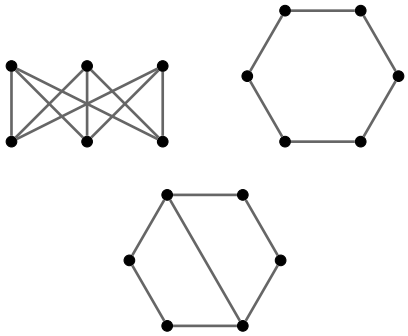


3. **Der n-Pfad P_n**
   $V = \{0, \dots, n\}$, $E = \{\{i, i+1\} \colon 0 \leq i \leq n-1\}$

Definition 1.33.
Sei $G = (V, E)$ ein Graph. G heißt bipartit (engl. bipartite), falls es eine Partition, $\{V_1, V_2\}$ mit $V = V_1 \cup V_2$, $V_1 \cap V_2 = \{\}$, der Knotenmenge V der folgenden Form gibt:

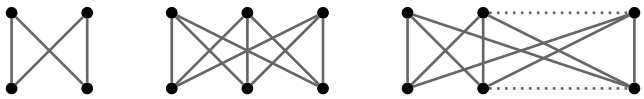
$$\textit{für alle } u, v \in V_1 \Rightarrow u \not\sim v \textit{ und für alle } u, v \in V_2 \Rightarrow u \not\sim v,$$

das heißt Kanten existieren höchstens zwischen Knoten aus V_1 und V_2.



Bipartite Graphen sind zum Beispiel gut geeignet für die Behandlung von Zuordnungsproblemen. Sei etwa V_1 eine Menge von Jobs und V_2 eine Menge von Bewerbern. So setzen wir $\{u, v\} \in E$, wenn Person u gut geeignet ist für Job v. Der so entstehende bipartite Graph modelliert das Personen-Zuordnungsproblem, das in der kombinatorischen Optimierung von großer Bedeutung ist.

4. **Der vollständige bipartite Graph** $K_{n,m}$
   $V = \{u_1, \ldots, u_n\} \cup \{v_1, \ldots v_m\}$,
   $E = \{\{u_i, v_j\} \colon u_i \in \{u_1, \ldots, u_n\}, v_j \in \{v_1, \ldots v_m\}\}$.



1.8 Eulergraphen

Das wohl berühmteste Rätsel der Graphentheorie entstand im 18. Jahrhundert. Der Mathematiker Leonhard Euler wurde gefragt, ob es denn eine Möglichkeit gebe, in Königsberg einen Rundweg zu finden, der über jede Brücke genau einmal führt. Von diesen Brücken über die Pregel gibt es genau sieben. In Abbildung 1.1 ist ein Plan der Stadt.

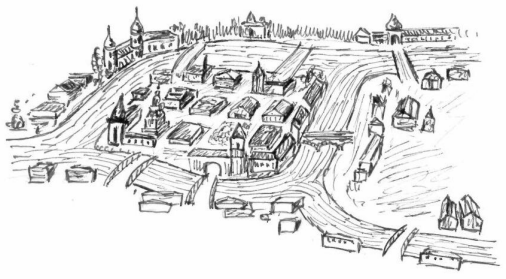


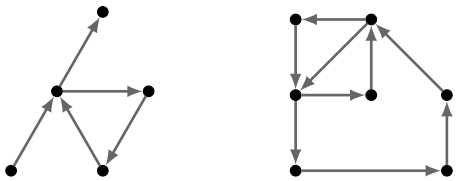
Abbildung 1.1: Königsberg (© Jana Kyšková)

Dieses Problem kann man mit einem Graphen modellieren. Jede Brücke wird zu einer Kante. Jeder Landesteil ist ein Knoten.

Abbildung 1.2: Graph von Königsberg

Definition 1.34.
Sei $G = (V, E)$ ein Graph. Ein Weg W heißt Eulerweg in G (engl. Euler trail), falls dieser Weg alle Kanten von G enthält.

Bemerkung 1.35.
Ein Eulerweg ist also eine Wanderung in G, welche jede Kante genau einmal benützt.



Definition 1.36.
Ein Graph heißt eulersch oder Eulergraph (engl. eulerian), falls es in diesem Graphen einen geschlossenen Eulerweg gibt.

Bemerkung 1.37.
Man nennt einen geschlossenen Eulerweg oft Eulerkreis (engl. Euler tour), selbst wenn es sich nicht um einen Kreis handelt, das heißt wenn Knoten mehrfach vorkommen.

Euler war natürlich nicht damit zufrieden, eine Lösung für das spezielle Problem in Königsberg zu finden, sondern suchte nach einer allgemein gültigen Aussage.

Satz 1.38.
Sei $G = (V, E)$ ein Graph und sei $W = (u_0, e_1, u_1, \ldots, e_n, u_n)$ ein Weg von u_0 nach u_n in G. Dann gelten:

1. *Falls W geschlossen ist ($u_0 = u_n$), dann ist für alle $0 \leq i \leq n$ die Anzahl der Kanten, die u_i als Endknoten haben, gerade, das heißt der Knotengrad von u_i ist gerade.*

2. *Falls W offen ist ($u_0 \neq u_n$), dann ist für alle $1 \leq i \leq n-1$ die Anzahl der Kanten von W, die u_i als Endknoten haben, gerade und die Anzahl der Kanten von W, auf denen u_0 beziehungsweise u_n liegen, ist ungerade.*

Daraus kann man folgenden Satz ableiten.

Satz 1.39.
Sei G ein zusammenhängender Graph. Dann sind folgende Aussagen äquivalent.

1. *G ist eulersch.*
2. *Jeder Knoten von G hat einen geraden Grad.*

Bemerkung 1.40.
Die Eigenschaft „alle Knotengrade sind gerade“ ist ein Zusammenschluss lokaler Eigenschaften von G; die Eigenschaft „G ist eulersch“ ist eine globale Eigenschaft von G. Dennoch sind die beiden Eigenschaften äquivalent.

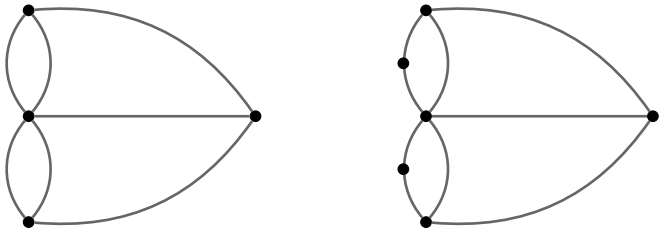
Eine ähnliche Aussage gilt für Eulerwege.

Satz 1.41.
Sei G ein zusammenhängender Graph. Dann sind folgende Aussagen äquivalent.

1. *In G gibt es einen Eulerweg.*
2. *Es gibt genau zwei oder keinen Weg mit ungeradem Grad. Alle anderen haben geraden Grad.*

Bemerkung 1.42.
Falls es genau zwei Knoten mit ungeradem Grad gibt, existiert ein offener Eulerweg; falls alle Knotengrade gerade sind, existiert ein geschlossener Eulerweg.

Satz 1.41 gilt auch für Graphen mit Mehrfachkanten, denn: Wir betrachten zunächst jenen Graphen, bei dem auf Mehrfachkanten jeweils ein weiterer Knoten eingefügt wurde. Diese Knoten haben geraden Grad. Alle anderen Knotengrade bleiben unverändert und der so entstandene Graph hat keine Mehrfachkanten. Offensichtlich gibt es in dem neuen Graphen genau dann einen Eulerweg, wenn es im ursprünglichen Graphen einen solchen gibt. In Königsberg gibt es also keinen Eulerweg.



Ein ähnliches Problem ist das *Haus des Nikolaus*. Bei diesem Beispiel findet sich also ein offener Eulerweg.

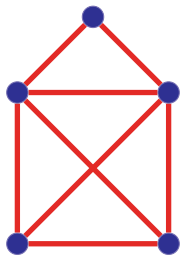


Abbildung 1.3: Haus des Nikolaus

Um Eulerkreise zu finden gibt es zwei grundlegende Algorithmen, der Zwiebelschalen-Algorithmus und den Fleury-Algorithmus. Wir lernen hier nur zweiteren kennen.

Algorithmus 1 Fleury-Algorithmus

Eingabe: Ein zusammenhängender eulerscher Graph, $G = (V, E)$
Ausgabe: Ein Eulerkreis

1: $F \longleftarrow E$
2: $v \longleftarrow$ ein zufälliger Knoten
3: $K \longleftarrow (v)$
4: $e \longleftarrow$ zufällig gewählte Kante, die zu v benachbart ist
5: **while** $F \neq \{\}$ **do**
6: $\quad F \longleftarrow F \setminus \{e\}$
7: $\quad v \longleftarrow$ anderes Ende von e
8: $\quad K \longleftarrow K \circ (v)$
9: $\quad e \longleftarrow$ zufällig gewählte Kante, die zu v benachbart ist und nicht in K liegt;
$\quad\quad\quad\quad$ e darf nur dann eine Brücke im kanteninduzierten Graph $\langle F \rangle$ sein,
$\quad\quad\quad\quad$ wenn es die letzte Möglichkeit ist
10: **end while**
11: **return** K

Beispiel 1.43.
Wir betrachten den Fleury-Algorithmus an einem Beispiel.

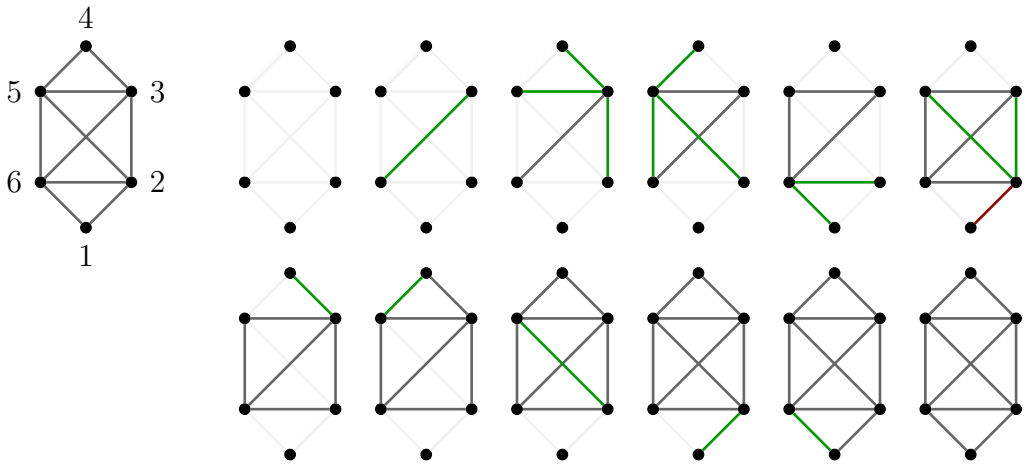


Abbildung 1.4: Eulerkreis im Haus des Nikolaus

Der Eulerkreis ist also $(6, 3, 5, 6, 2, 3, 4, 5, 2, 1, 6)$.

1.9 Hamiltongraphen

Definition 1.44.
Sei $G = (V, E)$ ein Graph. Ein Pfad in G, der alle Knoten von G enthält, heißt Hamiltonpfad (engl. Hamilton path); das heißt, ein Hamiltonpfad ist eine Wanderung in G, in der jeder Knoten genau einmal vorkommt.

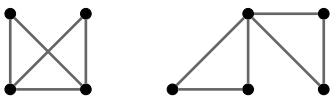
Ein Kreis in G, der alle Knoten von G enthält, heißt Hamiltonkreis (Hamiltonzyklus, engl. Hamilton cycle).

Wandert man längs eines Hamiltonkreises, so besucht man jeden Knoten genau einmal - mit Ausnahme des Startknoten.

Bemerkung 1.45.
Enthält ein Graph einen Hamiltonkreis, dann erhält man durch Weglassen einer Kante einen Hamiltonpfad.

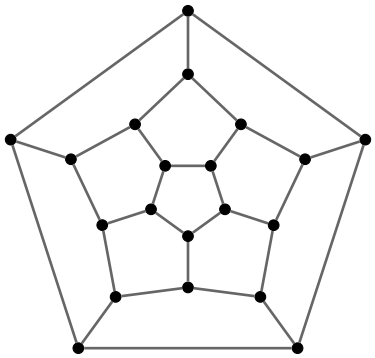
Definition 1.46.
Besitzt ein Graph G einen Hamiltonkreis, so heißt er hamiltonsch (engl. hamiltonian).

Beispiel 1.47.
Wir betrachten je ein einfaches Beispiel für einen hamiltonschen und einen nicht hamiltonschen Graphen.



Historisch: Sir William Hamilton (1805–1865) benannte die Knoten eines Dodekaeders mit verschiedenen Städtenamen und stellte die Frage, ob es einen Pfad entlang der Kanten gibt, der jede Stadt außer jener, in der gestartet wurde, genau einmal besucht?

Beispiel 1.48.
Der Dodekaeder-Graph.



Beispiel 1.49.
Weitere hamiltonsche Graphen:

- C_n *ist hamiltonsch* ($n \geq 3$)
- K_n *ist hamiltonsch* ($n \geq 3$)
- *Ist ein Graph* G *hamiltonsch und fügt man Kanten zwischen nicht benachbarten Knoten hinzu, so ist der so entstandene Graph auch hamiltonsch.*

Satz 1.50. (Ore)
Sei $G = (V, E)$ *ein Graph mit mindestens 3 Knoten und sei* n *die Anzahl der Knoten in* G*. Weiters gelte für alle nicht-benachbarten Knoten* u *und* v*, dass* $d(u) + d(v) \geq n$*. Dann folgt:* G *ist hamiltonsch.*

Korollar 1.51. (Dirac)
Sei $G = (V, E)$ *ein Graph mit* $|V| \geq 3$ *und* $d(v) \geq \frac{|V|}{2}$ *für alle* $v \in V$*. Dann folgt:* G *ist hamiltonsch.*

2 Kürzeste Wege

Um kürzeste Wege zu bestimmen, benötigen wir zuerst einen Begriff der Länge eines Weges. Wir können die Länge eines Weges durch die Anzahl an Kanten, die durchlaufen werden, definieren. Im Allgemeinen sind wir aber daran interessiert, verschiedenen Kanten unterschiedliche Längenwerte zuzuordnen.

Definition 2.1.
Sei $G = (V, E)$ ein Graph und $\omega : E \to \mathbb{N}$. Wir nennen ω eine Gewichtsfunktion *(engl. weight function). Das Paar (G, ω) ist dann ein* gewichteter Graph *(engl. weighted graph).*

Als Länge eines Weges bezeichnen wir die Summe der Kantengewichte, der Kanten die im Weg vorkommen. Die Länge des leeren Weges fixieren wir mit ∞. Die Länge eines Weges (v) mit nur einem Knoten ist 0.

2.1 Dijkstra

Der Dijkstra-Algorithmus ist eine nach dem niederländischen Informatiker Edsger Wybe Dijkstra benannte Methode, um einen kürzesten Weg von einem Startpunkt zu einem Ziel zu finden. Dieser Algorithmus ist ein Vertreter der sogenannten gierigen (engl. greedy) Algorithmen. Das sind Algorithmen, die im jeweiligen Schritt den größtmöglichen Gewinn für die Wahl des nächsten Zustands wählen.

Im Algorithmus verwenden wir die folgenden Funktionen, die im Verlauf regelmäßig aktualisiert werden.

$\text{dist} : V \to \mathbb{N} \cup \{\infty\}$	Distanz eines Knotens zum Startknoten $\text{dist}(v)$ ist die Länge von $W(v)$
$\ell : V \times V \to \mathbb{N}$	Gewicht/Länge der Kante zwischen zwei Knoten für $(v, w) \notin E$ gilt $\ell(v, w) = \infty$
$W : V \to \mathcal{W}$	Weg eines Knotens vom Startpunkt aus

Algorithmus 2 Dijkstra-Algorithmus

Eingabe: Ein gewichteter zusammenhängender Graph, $G = (V, E)$, ein Startknoten s und ein Zielknoten z

Ausgabe: Ein kürzester Weg von s nach z

1: $F \longleftarrow \{\,\}$
2: $\text{dist}(s) \longleftarrow 0$
3: $W(s) \longleftarrow (s)$
4: **for all** $v \in V \setminus \{s\}$ **do**
5: $\quad \text{dist}(v) \longleftarrow \infty$
6: $\quad W(v) \longleftarrow (\,)$
7: **end for**
8: **while** $z \notin F$ **do**
9: $\quad v^* \longleftarrow \text{argmin}_{v \in V \setminus F}\{\text{dist}(v)\}$
10: $\quad F \longleftarrow F \cup \{v^*\}$
11: $\quad$ **for all** $v \in V \setminus F$ **do**
12: $\quad\quad$ **if** $\text{dist}(v^*) + \ell(v^*, v) < \text{dist}(v)$ **then**
13: $\quad\quad\quad \text{dist}(v) \longleftarrow \text{dist}(v^*) + \ell(v^*, v)$
14: $\quad\quad\quad W(v) \longleftarrow W(v^*) \circ (v)$
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: **end while**
18: **return** $W(z)$

Beispiel 2.2.
Wir betrachten den Dijkstra-Algorithmus anhand eines Beispiels.

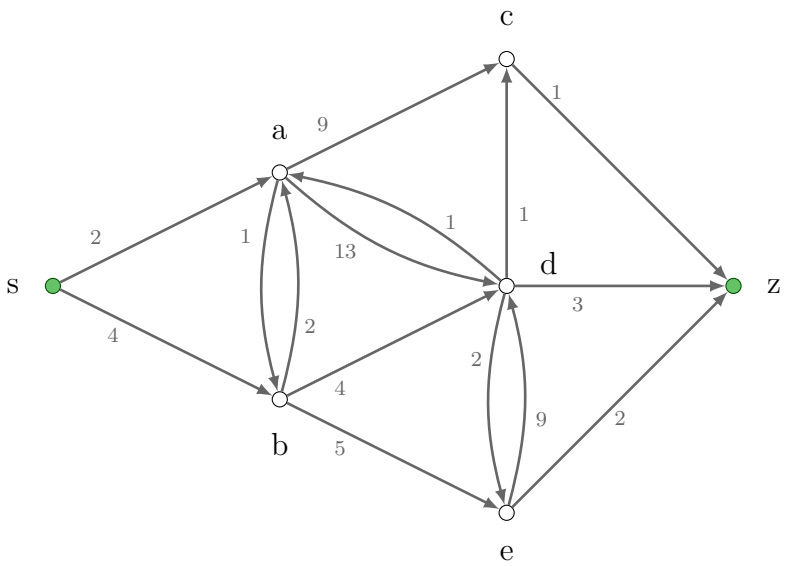
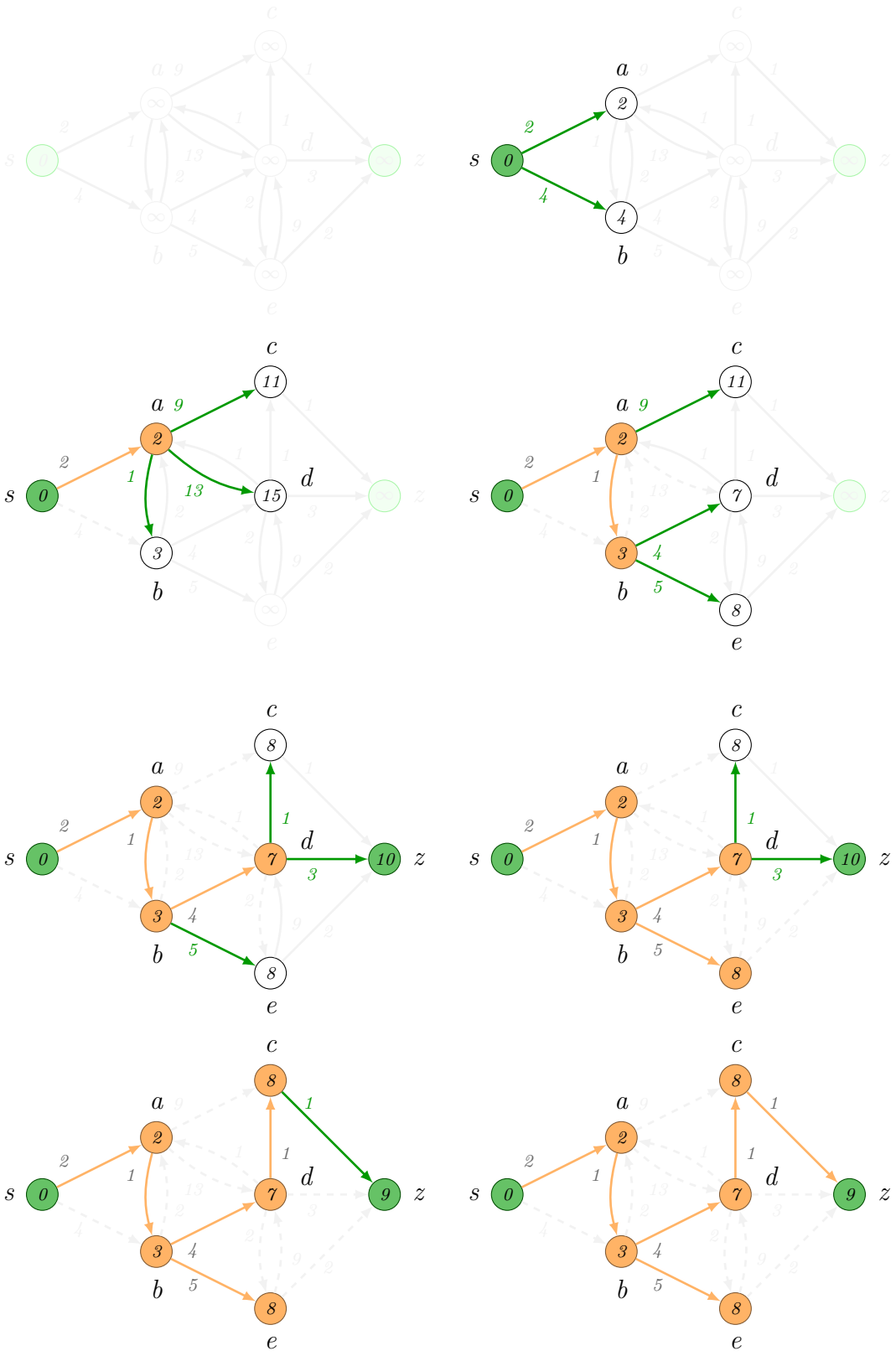


Abbildung 2.1: Ausgangsbeispiel



Die folgenden Tabellen betrachten die einzelnen Schleifendurchläufe des Algorithmus für

unser Beispiel. Die For-Schleife bezeichnen wir dabei mit dem Index 0*. Die Werte gelten dann für das Ende des jeweiligen Schleifendurchlaufs.*

Schritt	v^*	s	a	b	c	d	e	z	F
Beginn	–	0	∞	∞	∞	∞	∞	∞	$\{\,\}$
1	s	0	2	4	∞	∞	∞	∞	$\{s\}$
2	a	0	2	3	11	15	∞	∞	$\{s, a\}$
3	b	0	2	3	11	7	8	∞	$\{s, a, b\}$
4	d	0	2	3	8	7	8	10	$\{s, a, b, d\}$
5	e	0	2	3	8	7	8	10	$\{s, a, b, d, e\}$
6	c	0	2	3	8	7	8	9	$\{s, a, b, d, e, c\}$
7	z	0	2	3	8	7	8	9	$\{s, a, b, d, e, c, z\}$

Schritt	$W(s)$	$W(a)$	$W(b)$	$W(c)$	$W(d)$	$W(e)$	$W(z)$
Beginn	(s)	$(\,)$	$(\,)$	$(\,)$	$(\,)$	$(\,)$	$(\,)$
1	(s)	(s, a)	(s, b)	$(\,)$	$(\,)$	$(\,)$	$(\,)$
2	(s)	(s, a)	(s, a, b)	(s, a, c)	(s, a, d)	$(\,)$	$(\,)$
3	(s)	(s, a)	(s, a, b)	(s, a, c)	(s, a, b, d)	(s, a, b, e)	$(\,)$
4	(s)	(s, a)	(s, a, b)	(s, a, b, d, c)	(s, a, b, d)	(s, a, b, e)	(s, a, b, d, z)
5	(s)	(s, a)	(s, a, b)	(s, a, b, d, c)	(s, a, b, d)	(s, a, b, e)	(s, a, b, d, z)
6	(s)	(s, a)	(s, a, b)	(s, a, b, d, c)	(s, a, b, d)	(s, a, b, e)	(s, a, b, d, c, z)
7	(s)	(s, a)	(s, a, b)	(s, a, b, d, c)	(s, a, b, d)	(s, a, b, e)	(s, a, b, d, c, z)

Wir müssen noch erklären, warum der Dijkstra-Algorithmus korrekt ist.

Lemma 2.3. (Korrektheitslemma)
Sei $G = (V, E)$ ein gerichteter Graph und sei s der Startknoten, mit dem der Dijkstra-Algorithmus aufgerufen wird. Wir verwenden die Notation wie im Algorithmus. Wir nehmen an, dass folgende zwei Eigenschaften gelten.

(i) $\forall v \in F : W(v)$ *ist der minimale Weg von s nach v und alle Knoten in $W(v)$ liegen in F.*

(ii) $\forall u \in V \setminus F$ *: Sei* $\bar{w} = (\bar{u}_1, \ldots, \bar{u}_j, u)$ *mit* $\bar{u}_i \in F$, $\bar{u}_1 = s$ *und* $j \geq 1$*. Dann ist $\bar{w}$ länger als oder gleich lang wie $W(u)$.*

Sei v^* ein Knoten mit $\mathrm{dist}(v^*) = \min_{v \in V \setminus F}\{\mathrm{dist}(v)\}$*. Sei W' die Wegfunktion nach dem Schleifendurchlauf, also*

$$W'(v) = \begin{cases} W(v) & \text{wenn } \mathrm{dist}(v^*) + \ell(v^*, v) \geq \mathrm{dist}(v) \\ W(v^*) \circ (v) & \text{wenn } \mathrm{dist}(v^*) + \ell(v^*, v) < \mathrm{dist}(v) \end{cases}$$

Sei weiters $F' = F \cup \{v^*\}$ und dist' *die aktualisierte Distanzfunktion. Dann gilt*

(i') $\forall v \in F' : W'(v)$ *ist der minimale Weg von s nach v und alle Knoten in $W'(v)$ liegen in F'.*

(ii') $\forall u \in V \setminus F' :$ *Sei* $\bar{w} = (\bar{u}_1, \ldots, \bar{u}_j, u)$ *mit* $\bar{u}_i \in F'$, $\bar{u}_1 = s$ *und* $j \geq 1$. *Dann ist* $\bar{w}$ *länger als oder gleich lang wie* $W'(u)$.

Die Eigenschaften aus (i) und (ii) heißen Schleifeninvarianten. Wir zeigen, dass die Schleifeninvarianten zu Beginn des Algorithmus erfüllt sind und dann auch während des Algorithmus nach jedem Schleifendurchlauf gelten. In Worten lauten die Schleifeninvarianten wie folgt.

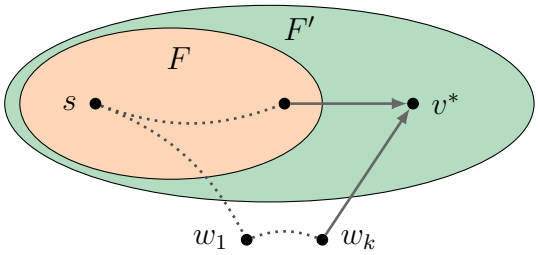
(i) Für alle Knoten v in F haben wir einen kürzesten Weg von s nach v gefunden.

(ii) Für alle Knoten u, die nicht in F liegen ist der gespeicherte Weg $W(u)$ ein kürzester unter all jenen Wegen von s nach u, deren Kanten alle bis auf die letzte in $\langle F \rangle$ liegen.

Beweis. Zu Beginn gilt (i), da F die leere Menge ist. Auch (ii) gilt, da noch kein Weg über Knoten aus F führen kann. Wir nehmen nun an, dass (i) und (ii) gelten.

(i'): Sei $v \in F'$. Wir wollen zeigen, dass $W'(v)$ minimal ist und unterscheiden zwei Fälle.

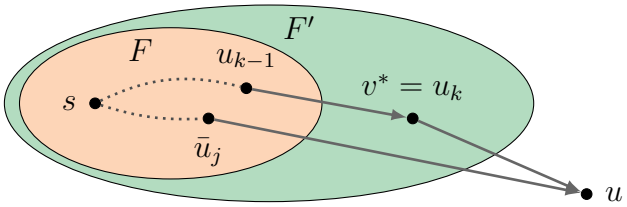
1. Wenn $v \in F$, dann ist $W(v)$ minimal nach (i). Das heißt $W'(v) = W(v)$ ist minimal und alle Knoten in $W'(v)$ liegen in F'.
2. Wenn $v \notin F$ also $v = v^*$. Es gilt $\text{dist}(v^*)$ ist minimal in $V \setminus F$. Weiters gilt $W'(v^*) = W(v^*)$. Wir wissen von (ii), dass $W(v^*)$ minimal ist unter allen Wegen in denen der vorletzte Knoten in F liegt. Alle Knoten in $W(v^*)$ liegen in F'. Nehmen wir an, es gibt einen anderen Weg $\bar{w} = (s, v_1, \ldots, v_j, w_1, \ldots, w_k, v^*)$ mit $v_i \in F$ und $w_1 \notin F$. Dann ist $\bar{w}$ klarerweise echt länger als $(s, v_1, \ldots, v_j, w_1)$ und dieser Weg wiederum ist länger als $W(w_1)$ nach (ii). Da $\text{dist}(v^*) \leq \text{dist}(w_1)$, weil v^* ja so gewählt war, dass $\text{dist}(v^*)$ minimal ist, folgt, dass $\bar{w}$ länger als $W'(v^*)$ ist.



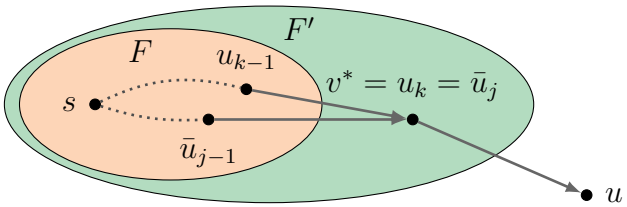
(ii'): Sei $u \in V \setminus F'$. Sei $W'(u) = (s, u_1, \ldots, u_k, u)$ mit $u_i \in F'$. Daher gilt auch $W(u_k) = (s, u_1, \ldots, u_k)$. Sei weiters $\bar{w} = (s, \bar{u}_1, \ldots, \bar{u}_j, u)$ mit $\bar{u}_i \in F'$ ein Weg von s nach u. Wir unterscheiden wieder zwei Fälle mit Unterfällen.

1. Wir nehmen an, dass $u_k = v^*$.
   1.1. Sei $\bar{u}_j \neq v^*$, also $\bar{u}_j \in F$. Wir können ohne Beschränkung der Allgemeinheit annehmen, dass $\bar{u}_i \in F$ auch für $1 \leq i \leq j-1$. In der letzten Schleife hatten wir $\text{dist}'(u) = \text{dist}(v^*) + \ell(v^*, u) < \text{dist}(u)$ nach (ii). Es gilt, dass $W(u)$ der kürzeste Weg von s nach u ist unter allen Wegen mit

der Eigenschaft, dass der vorletzte Knoten in F liegt. Also ist $\bar{w}$ länger als oder gleich lang wie $W(u)$. Da aber $W'(u)$ kürzer als $W(u)$ ist, folgt (ii').

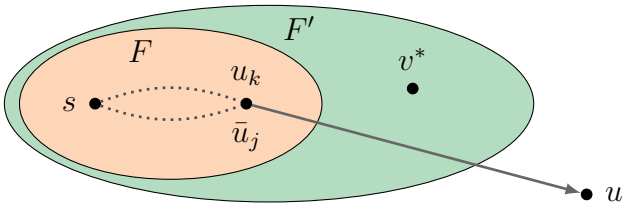


1.2. Sei $\bar{u}_j = v^*$. Nach (i') ist $W'(v^*)$ der kürzeste Weg von s nach v^*. Da $W'(u) = W'(v^*) \circ (u)$, gilt, dass $\bar{w}$ länger ist.

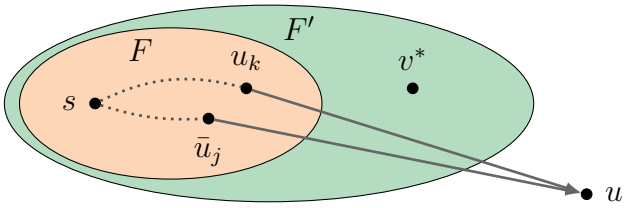


2. Wir nehmen an, dass $u_k \neq v^*$.

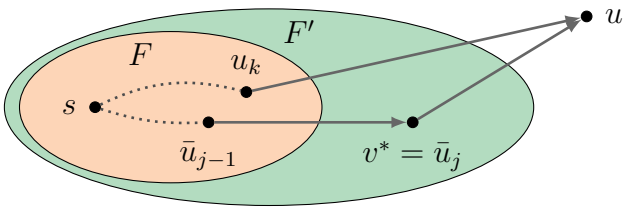
2.1. Sei $u_k = \bar{u}_j$. Dann ist nach (i') $W'(u_k) = W'(\bar{u}_j)$ minimal und $W'(u) = W'(u_k) \circ (u)$ also kürzer oder gleich lang wie $\bar{w}$.



2.2. Sei $u_k \neq \bar{u}_j$. Das heißt, $u_k \in F$. Es gilt also $\text{dist}'(u) = \text{dist}(u)$ und $W'(u) = W(u)$ und somit ist nach (ii) $\bar{w}$, wenn $\bar{u}_j \neq v^*$, da $\bar{u}_i \in F$ für $1 \leq i \leq j$ ohne Beschränkung der Allgemeinheit.



Es bleibt noch zu ermitteln, was im Fall $\bar{u}_j = v^*$ passiert. Nach (ii) ist $W'(v^*)$ minimal. Also können wir ohne Beschränkung der Allgemeinheit annehmen, dass $(s, \bar{u}_1, \ldots, \bar{u}_j) = W'(v^*)$. Es gilt dann $W'(v^*) = W(v^*)$. Angenommen $W(v^*) \circ (u)$ wäre kürzer als $W(u)$. Dann wäre aber $\text{dist}'(u) \neq \text{dist}(u)$, was aber nicht sein kann.



□

Der Dijkstra-Algorithmus kann leicht erweitert werden, sodass er minimale Wege von einem Knoten zu allen anderen Knoten findet. In der bisherigen Version findet er nur minimale Wege zu einer gewissen Teilmenge der Knoten. Die Änderung betrifft nur die Bedingung, wann wir die Schleife abbrechen dürfen. Bisher haben wir abgebrochen, wenn $z \in F$. Nun brechen wir erst ab, wenn alle Knoten besucht sind.

Algorithmus 3 Dijkstra-Algorithmus (alle Knoten)

Eingabe: Ein gewichteter zusammenhängender Graph $G = (V, E)$, ein Startknoten s
Ausgabe: Ein kürzester Weg von s zu allen Knoten in V

```
1:  F ⟵ { }
2:  dist(s) ⟵ 0
3:  W(s) ⟵ (s)
4:  for all v ∈ V \ {s} do
5:      dist(v) ⟵ ∞
6:      W(v) ⟵ ( )
7:  end for
8:  while F ≠ V do
9:      v* ⟵ argmin_{v∈V\F}{dist(v)}
10:     F ⟵ F ∪ {v*}
11:     for all v ∈ V \ F do
12:         if dist(v*) + ℓ(v*, v) < dist(v) then
13:             dist(v) ⟵ dist(v*) + ℓ(v*, v)
14:             W(v) ⟵ W(v*) ∘ (v)
15:         end if
16:     end for
17: end while
18: return W
```

2.2 A*

Der Dijkstra-Algorithmus hat den Nachteil, dass er gleichmäßig in alle Richtungen sucht. Häufig haben wir aber Informationen, die uns erlauben bevorzugt in eine bestimmte Richtung zu suchen. Bei der Wegsuche auf Landkarten ist dies etwa die Luftlinienentfernung. Der A*-Algorithmus verwendet solche Zusatzinformationen. Wir nennen sie

Heuristik. Der Begriff kommt daher, dass häufig (im Gegensatz zu Luftlinien) nur unvollständige oder geschätzte Informationen vorliegen. Wir verlangen daher zusätzliche Eigenschaften um zu garantieren, dass der Algorithmus funktioniert.

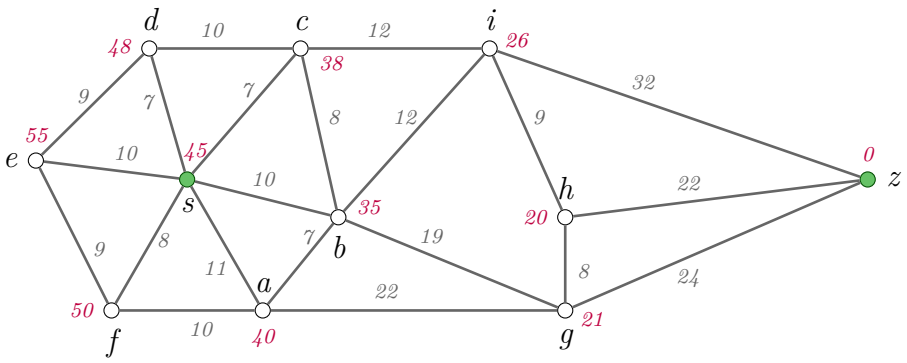
Definition 2.4.
Sei $G = (V, E)$ ein Graph mit einer Gewichtsfunktion $\omega : E \to \mathbb{N}$ und $z \in V$. Eine Funktion $h : V \to \mathbb{R}^+$ heißt monotone Heuristik (engl. monotone heuristic), falls $h(u) \leq \omega(\{u, v\}) + h(v)$ für alle $\{u, v\} \in E$ und $h(z) = 0$.

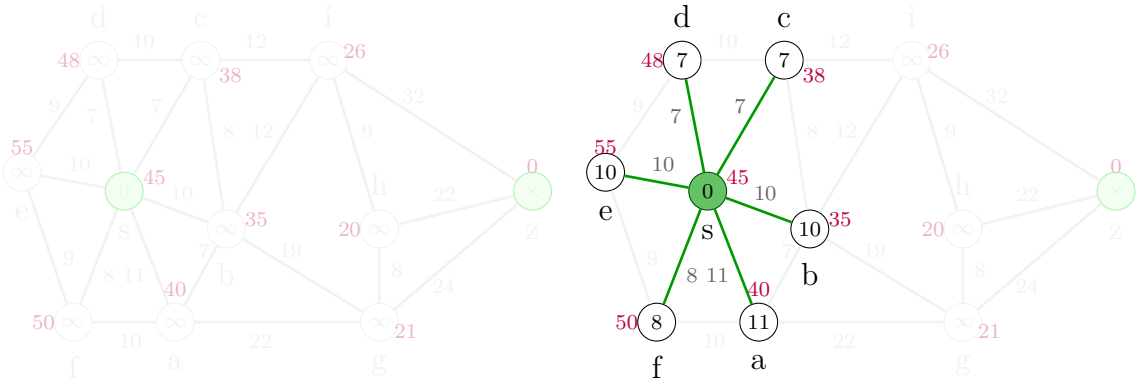
Algorithmus 4 A*-Algorithmus

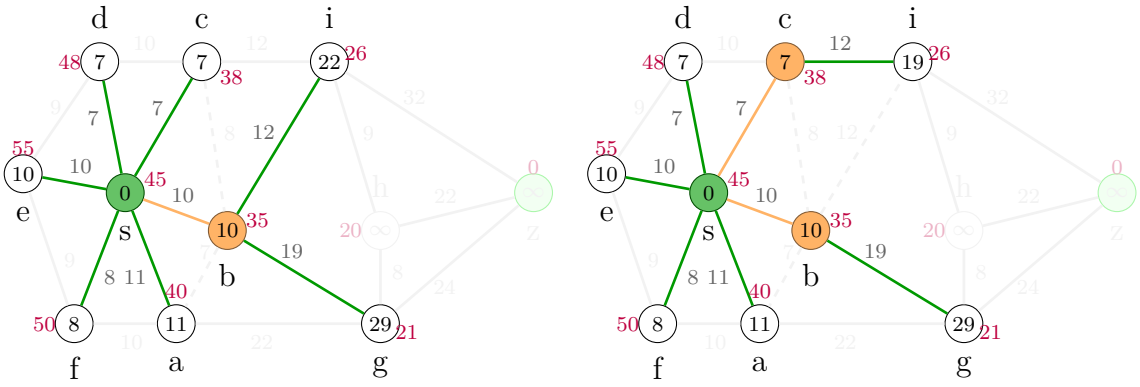
Eingabe: Ein gewichteter zusammenhängender Graph, $G = (V, E)$, ein Startknoten s und ein Zielknoten z, eine monotone Heuristik h
Ausgabe: Ein kürzester Weg von s nach z

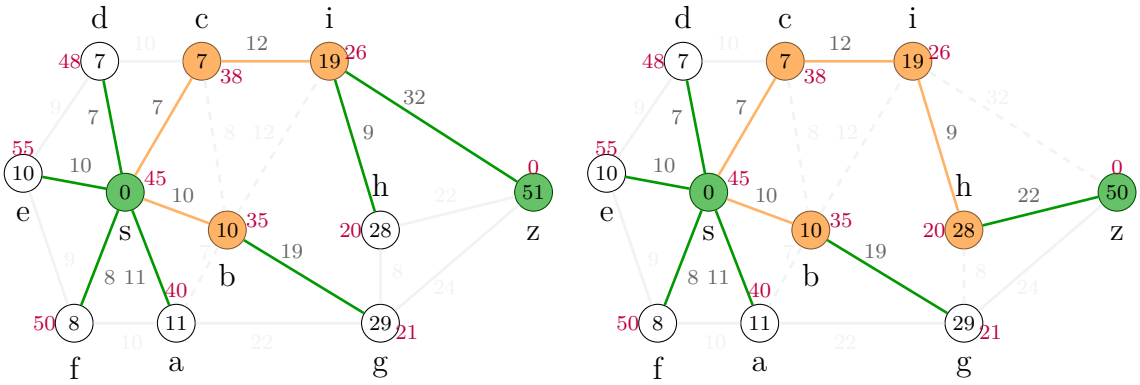
```
1: F ⟵ { }
2: dist(s) ⟵ 0
3: W(s) ⟵ (s)
4: for all v ∈ V \ {s} do
5:     dist(v) ⟵ ∞
6:     W(v) ⟵ ( )
7: end for
8: while z ∉ F do
9:     v* ⟵ argmin_{v∈V\F}{dist(v) + h(v)}
10:    F ⟵ F ∪ {v*}
11:    for all v ∈ V \ F do
12:        if dist(v*) + ℓ(v*, v) < dist(v) then
13:            dist(v) ⟵ dist(v*) + ℓ(v*, v)
14:            W(v) ⟵ W(v*) ∘ (v)
15:        end if
16:    end for
17: end while
18: return W(z)
```

Beispiel 2.5.
Wir sehen uns den Unterschied von A und Dijkstra anhand eines Beispiels an.*

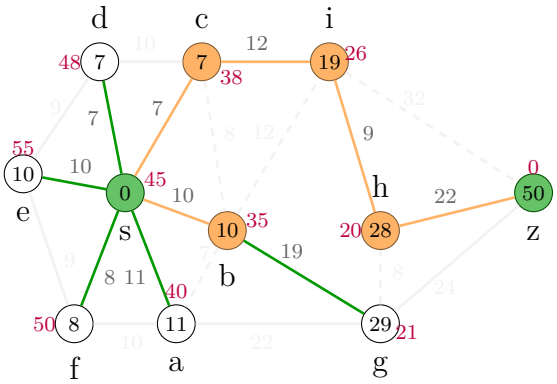

d
c
i
48
38
26
10
12
32
9
7
7
8
12
9
55
e
10
45
10
0
z
s
35
h
22
20
9
8
7
b
19
11
8
24
a
22
50
10
40
21
f
g


d
c
i
48 7
7
38
26
9
7
7
55
10
10
0
45
10
10
35
e
s
b
h
22
0
z
20
8 11
40
50 8
11
f
a
21
g


d
c
i
48 7
7
38
22
26
12
7
7
55
10
10
0
45
10
35
e
s
b
h
22
0
z
20
19
8 11
40
50 8
11
29
21
f
a
g


d
c
i
48 7
7
38
19
26
12
7
7
55
10
10
0
45
10
35
e
s
b
h
22
0
z
20
19
8 11
40
50 8
11
29
21
f
a
g

d
c
i
48 7
7
38
19
26
12
7
7
55
10
10
0
45
10
35
e
s
b
h
22
50
z
20
28
19
8 11
40
50 8
11
29
21
f
a
g


d
c
i
48 7
7
38
19
26
12
32
7
7
9
55
10
10
0
45
10
35
51
e
s
b
h
22
z
20
28
19
8 11
40
50 8
11
29
21
f
a
g

d
c
i
48 7
7
38
19
26
12
32
7
7
9
55
10
10
0
45
10
35
50
e
s
b
h
22
z
20
28
19
8 11
40
50 8
11
29
21
f
a
g

3 Spannbäume

Einen zusammenhängenden ungerichteten Graphen, der keine Kreise enthält, nennt man *Baum* (engl. tree). Ein *Spannbaum* (engl. spanning tree) ist ein Baum, der alle Knoten eines Graphen erreicht. Ein *minimaler Spannbaum* (engl. minimum spanning tree) ist ein solcher, der minimales Kantengewicht erreicht. Wir lernen nun zwei Algorithmen kennen, die einen minimalen Spannbaum errechnen.

Spannbäume spielen etwa eine Rolle bei der Betrachtung von Rechnernetzen. Diese sind mit einer Switch-Infrastruktur miteinander verbunden. Redundante Verbindungen erlauben Ausfallsicherheit. Sie führen aber dazu, dass Pakete in einer Schleife gefangen werden können und nie ankommen. Um dies zu vermeiden, wird ein Spannbaum der Netz-Struktur erzeugt und alle Verbindungen, die nicht im Baum sind, werden blockiert. Details dazu findet man unter dem Begriff *Spanning-Tree-Protocol*.

Der Dijkstra-Algorithmus generiert auch einen Spannbaum, allerdings ist dieser nicht immer minimal.

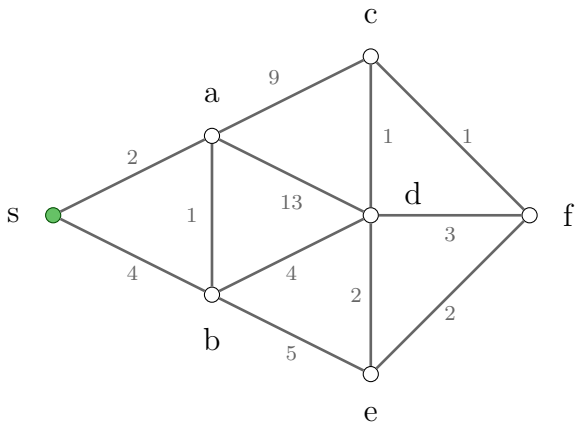


Abbildung 3.1: Ausgangsbeispiel für Spannbäume

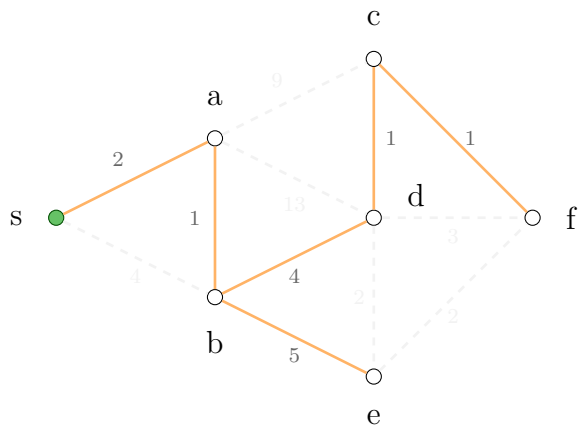


Abbildung 3.2: Spannbaum erzeugt vom Dijkstra-Algorithmus.

3.1 Prim-Algorithmus

Um einen minimalen Spannbaum zu generieren reicht eine kleine Änderung. Der Algorithmus von Prim ist das Resultat.

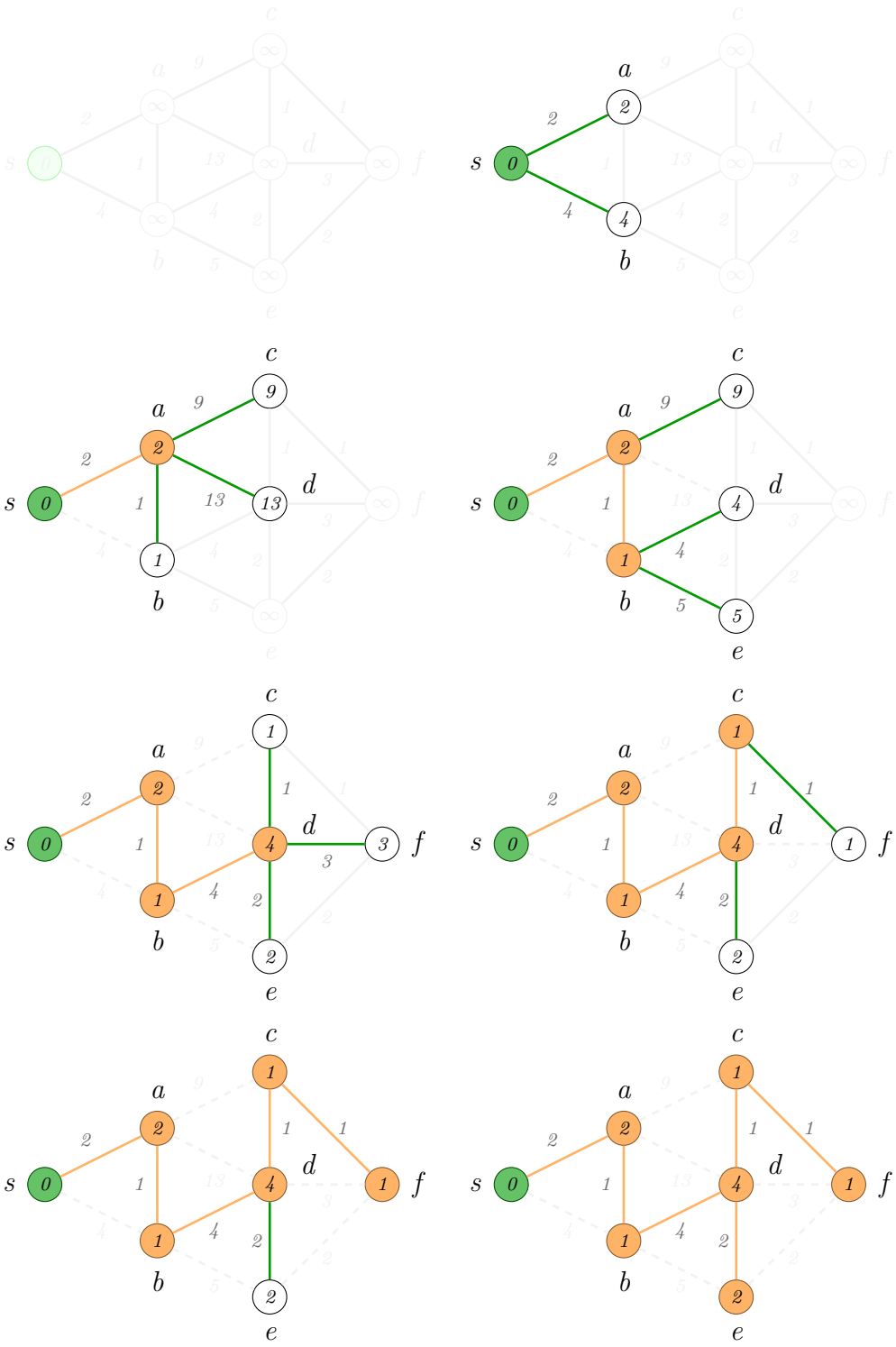
Algorithmus 5 Prim-Algorithmus

Eingabe: Ein gewichteter zusammenhängender Graph, $G = (V, E)$ und ein beliebiger Startknoten s

Ausgabe: Ein minimaler Spannbaum von G

1: $F \longleftarrow \{\}$
2: $\text{dist}(s) \longleftarrow 0$
3: $W(s) \longleftarrow ()$
4: **for all** $v \in V \setminus \{s\}$ **do**
5: $\quad \text{dist}(v) \longleftarrow \infty$
6: $\quad W(v) \longleftarrow ()$
7: **end for**
8: **while** $F \neq V$ **do**
9: $\quad v^* \longleftarrow \text{argmin}_{v \in V \setminus F}\{\text{dist}(v)\}$
10: $\quad F \longleftarrow F \cup \{v^*\}$
11: $\quad$ **for all** $v \in V \setminus F$ **do**
12: $\quad\quad$ **if** $\ell(v^*, v) < \text{dist}(v)$ **then**
13: $\quad\quad\quad \text{dist}(v) \longleftarrow \ell(v^*, v)$
14: $\quad\quad\quad W(v) \longleftarrow (v^*)$
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: **end while**
18: **return** $\langle \{e \in E \mid \exists v \in V : (W(v), v) = e\} \rangle$

Beispiel 3.1.
In diesem Beispiel wollen wir den Prim-Algorithmus kennen lernen. Wir gehen die einzelnen Schritte durch.



Wir sehen also, dass dieser Spannbaum eine geringere Länge hat als der durch den Dijkstra-Algorithmus erzeugte.

3.2 Kruskal

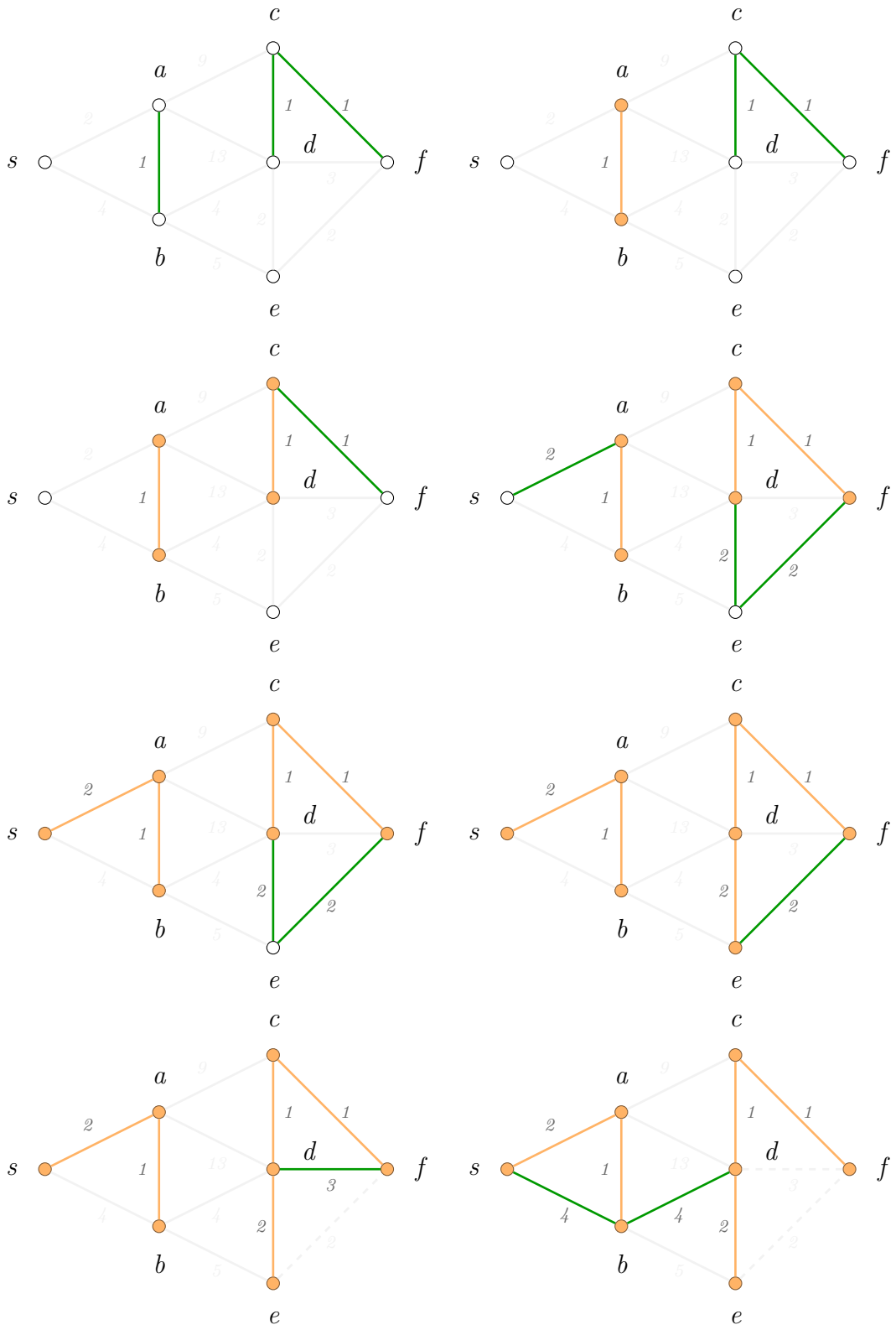
Der Prim-Algorithmus ist wieder ein Greedy-Algorithmus, aber in diesem Fall gibt es einen noch gierigeren Algorithmus. Algorithmus 6 beschreibt diesen.

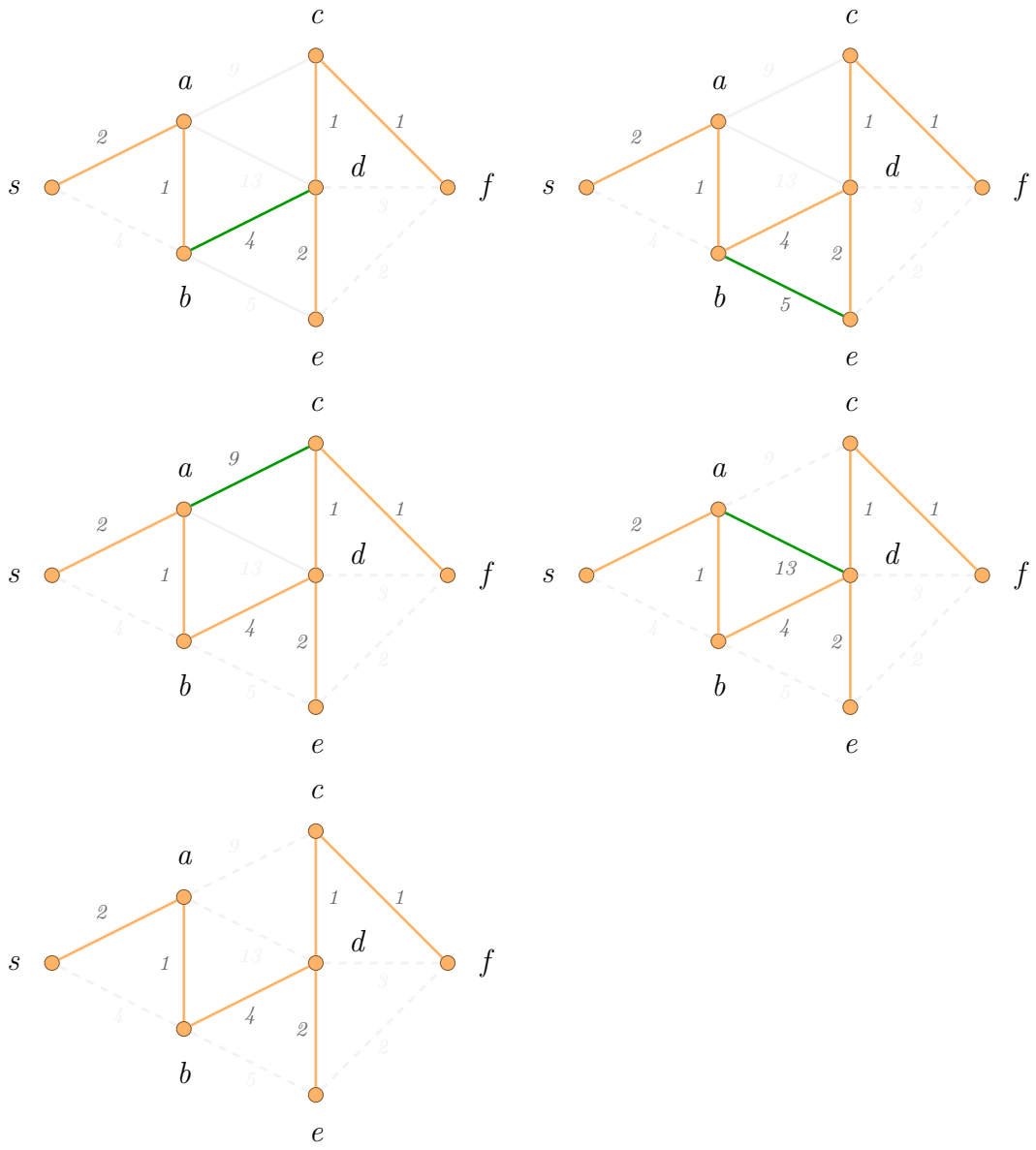
Algorithmus 6 Kruskal-Algorithmus

Eingabe: Ein gewichteter zusammenhängender Graph, $G = (V, E)$
Ausgabe: Ein minimaler Spannbaum von G

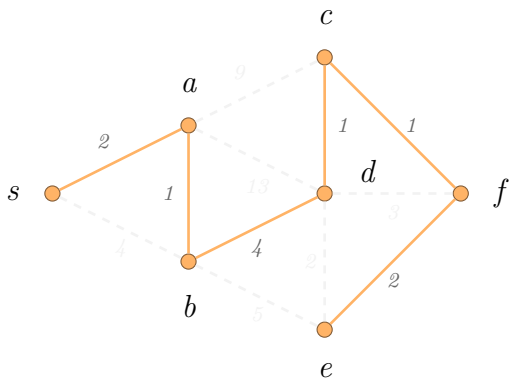
1: $B \longleftarrow E$
2: $S \longleftarrow \{\,\}$
3: **while** $B \neq \{\,\}$ **do**
4: $\quad e^* \longleftarrow \operatorname{argmin}_{e \in B}\{\omega(e)\}$
5: $\quad$ **if** $\langle S \cup \{e^*\}\rangle$ enthält keinen Kreis **then**
6: $\quad\quad S \longleftarrow S \cup \{e^*\}$
7: $\quad$ **end if**
8: $\quad B \longleftarrow B \setminus \{e^*\}$
9: **end while**
10: **return** $\langle S \rangle$

Beispiel 3.2.
Wir betrachten das Beispiel von oben anhand des Kruskal-Algorithmus.


c
a
9
2
1
1
1
d
s
1
13
f
3
4
4
2
2
b
5
e



In dem Schritt, in dem wir Kanten der Länge 2 betrachtet haben, konnten wir wählen. Hätten wir uns anders entschieden, wäre das Resultat der folgende minimale Spannbaum.



Ein minimaler Spannbaum ist also nicht eindeutig.

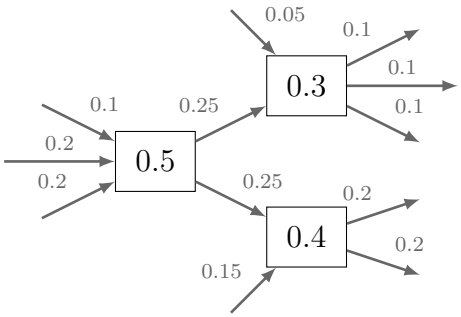
4 PageRank

Historisch: Im Jahr 1996 kreierten die zwei Doktoratsstudenten der Stanford Universität Lawrence Page und Sergey Brin eine neue Idee für eine Internet-Suchmaschine. Konventionelle damalige Suchmaschine verwendeten meist die Anzahl an Vorkommnissen des Suchbegriffs zur Sortierung. Der PageRank-Algorithmus von Page und Brin verwendet statt dessen die Anzahl an Links als Kriterium für die Sortierung.

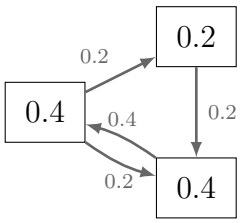
Wir definieren einen *Webgraph* (V, E) als gerichteter Graph, dessen Knoten die Webseiten repräsentieren und die Kanten Links darstellen. Eine gerichtete Kanten (u, v) heißt also, dass die Webseite u auf die Webseite v verlinkt. Die Kante wird oft auch als *Backlink* von v bezeichnet. Das Ziel des PageRank-Algorithmus ist eine Sortierung $R = (R_w)_{w \in V}, R_w \in [0, 1]$ zu finden. Wir nennen R den *Rangvektor* (engl. rank vector). Die Intuition dahinter, ist dass Seiten mit vielen Backlinks beziehungsweise mit bedeutenden Backlinks weiter vorne sortiert werden. Ein Analogon dazu ist ein zufälliger Surfer. Der Rank bestimmt die Wahrscheinlichkeit, mit welcher der Surfer auf einer Webseite landet, wenn er immer einen zufälligen Link klickt. Sei $B_u = \{v \in V \mid (v, u) \in E\}$ die Menge an Webseiten, die auf u verlinken. Wir definieren die Sortierung durch

$$R_u = \sum_{v \in B_u} \frac{R_v}{\deg_{\text{out}}(v)} .$$

Das folgende Bild erklärt, wie die Reihung von einer Seite auf die nächste übertragen wird.



Ein erster Vorschlag zur Berechnung wäre es, zu Beginn jeder Webseite den gleichen Wert zuzuordnen und dann iterativ die Werte neu zu berechnen bis sich diese stabilisieren. Für manche Graphen funktioniert das auch.



Im Allgemeinen funktioniert das aber nicht. Die Formel kann auch in Matrixschreibweise gegeben werden.

$$R = \widetilde{A} \cdot R\,, \tag{4.1}$$

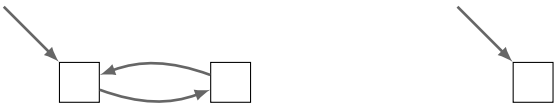
wobei die Matrix $\widetilde{A} = (\widetilde{A}_{u,v})_{u,v \in V}$ definiert ist durch

$$\widetilde{A}_{u,v} = \begin{cases} \frac{1}{\deg_{\text{out}}(v)} & \text{wenn } v \in B_u\,, \\ 0 & \text{sonst.} \end{cases}$$

Für das Beispiel von oben bekommen wir also

$$\widetilde{A} = \begin{pmatrix} 0 & 0 & 1 \\ 0.5 & 0 & 0 \\ 0.5 & 1 & 0 \end{pmatrix}.$$

Die Matrix $\widetilde{A}$ kann von der Adjazenzmatrix von G hergeleitet werden, indem diese transponiert und normalisiert wird, sodass die Summe der Spalteneinträge immer eins ergibt. Die Matrixdarstellung (4.1) stellt das Problem als Eigenwertproblem dar. Wir sehen, dass R ein Eigenvektor von $\widetilde{A}$ mit Eigenwert 1 ist. Der Rangvektor R existiert nur dann, wenn die Matrix einen Eigenwert 1 hat und der dazugehörige Eigenvektor nur nicht-negative Einträge hat. Das ist etwa nicht der Fall, wenn der Graph Senken (engl. sink) oder hängende Knoten, (engl. dangling node) hat. Senken repräsentieren ein Teil eines Netzwerks, der nicht in den Rest des Netzes verlinkt, aber von dort aus erreichbar ist. Hängende Knoten sind jene Knoten v mit $\deg_{\text{out}}(v) = 0$. Achtung, diese Bezeichnungen stammen aus den Ursprüngen von PageRank. In der Graphentheorie wird meist letzteres als Senke bezeichnet. Diese Knoten bringen ihre Reihung nicht mehr ins System zurück wodurch sie die einzigen bleiben, die einen Eintrag größer 0 haben.



Aus diesem Grund beinhaltet der PageRank-Algorithmus noch einen *Dämpfungsfaktor d* (engl. damping factor). Für den zufälligen Surfer bedeutet dieser Dämpfungsfaktor die Wahrscheinlichkeit weiterhin Links zu klicken, während mit Wahrscheinlichkeit $d-1$ auf eine zufällige Webseite gesprungen wird. Ein üblicher Wert für den Dämpfungsfaktor wäre etwa 0.85. Weiters nehmen wir an, dass Senken ihren Wert gleichmäßig auf alle anderen Knoten verteilen. Im Fall, dass der zufällige Surfer auf einer solchen Webseite

landet, springt er also zufällig auf eine beliebige andere Seite. Mathematisch bedeutet das

$$R_u = \frac{1-d}{|V|} + d\left(\sum_{v \in B_u} \frac{R_v}{\deg_{\text{out}}(v)} + \sum_{\substack{v \notin B_u \\ \deg_{\text{out}}(v)=0}} \frac{R_v}{|V|}\right).$$

Wieder schreiben wir dies als Vektorgleichung.

$$R = \frac{1-d}{|V|} \cdot \mathbb{J} \cdot R + d \cdot \widehat{A} \cdot R = \underbrace{\left(\frac{1-d}{|V|} \cdot \mathbb{J} + d \cdot \widehat{A}\right)}_{M} \cdot R,$$

wobei $\mathbb{J}$ die Matrix mit $|V| \times |V|$ Eins-Einträgen ist und $\widehat{A}$ wie folgt definiert ist

$$\widehat{A}_{u,v} = \begin{cases} \frac{1}{|V|} & \text{wenn } \deg_{\text{out}}(v) = 0, \\ \frac{1}{\deg_{\text{out}}(v)} & \text{wenn } v \in B_u, \\ 0 & \text{sonst.} \end{cases}$$

Die Matrix M wird manchmal auch *Google-Matrix* genannt. Aus der Tatsache, dass sich die Einträge jeder Spalte in M auf eins summieren, kann der folgende Satz bewiesen werden.

Satz 4.1.
Sei $d \in (0,1)$ und M wie oben. Dann hat M einen Eigenwert 1 und es gelten:

1. *1 hat den echt größten Absolutbetrag aller Eigenwerte. 1 ist also ein dominanter Eigenwert.*
2. *1 ist ein einfacher Eigenwert, der zugehörige Eigenraum ist also eindimensional*
3. *Es gibt einen Eigenvektor R zum Eigenwert 1 mit nur positiven Einträgen.*

Daher ist die Existenz des Rangvektors R garantiert. Die direkte Berechnung der Eigenwerte und Eigenvektoren von M ist nur für kleine Matrizen sinnvoll. Das Internet hingegen besteht aus vielen Knoten. Der Beweis von Satz 4.1 liefert aber auch einen Algorithmus (Algorithmus 7) um den Rangvektor effizient zu berechnen. Dabei wird die sogenannte *Power-Method* aus der numerischen linearen Algebra verwendet. Diese Methode berechnet den dominanten Eigenwert einer Matrix und den dazugehörigen Eigenvektor unter der Annahme, dass der Eigenwert einfach ist.

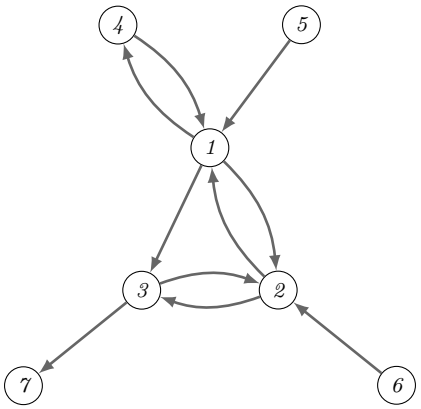
Algorithmus 7 PageRank-Algorithmus

Eingabe: Matrix M mit Dämpfungsfaktor $d \in (0,1)$, Toleranz $\varepsilon < 1/2$
Ausgabe: Rangvektor R

1: $R \longleftarrow \frac{1}{|V|} \cdot (1,\ldots,1)^T$
2: $R_{\text{prev}} \longleftarrow (1,\ldots,1)^T$
3: **while** $\|R - R_{\text{prev}}\|_1 > \varepsilon$ **do**
4: $\quad R_{\text{prev}} \longleftarrow R$
5: $\quad R \longleftarrow M \cdot R$
6: **end while**
7: **return** R

Das Abbruchkriterium ist von der Toleranz abhängig. Wir verwenden hier die Norm $||(x_1,\ldots,x_n)||_1 = \sum_{k=1}^{n} |x_k|$. Die Matrix M kann in einer Weise gespeichert werden, dass die Multiplikation $M \cdot R$ nicht so teuer kommt. Weiters ist es im Durchschnittsfall ausreichend die Schleife $O(\log |V|)$ zu durchlaufen. Dadurch kann der Algorithmus auch große Webgraphen behandeln.

Beispiel 4.2.
Sei G der folgende Webgraph.



Die transponierte Adjazenzmatrix und die Matrix $\widehat{A}$ für diesen Graph sehen wie folgt aus.

$$A^T = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad \widehat{A} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & 1 & 1 & 0 & \frac{1}{7} \\ \frac{1}{3} & 0 & \frac{1}{2} & 0 & 0 & 1 & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{7} \\ \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & \frac{1}{7} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{7} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{7} \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{7} \end{pmatrix}.$$

Wir konstruieren die Matrix M mit Dämpfungsfaktor $d = 0.85$.

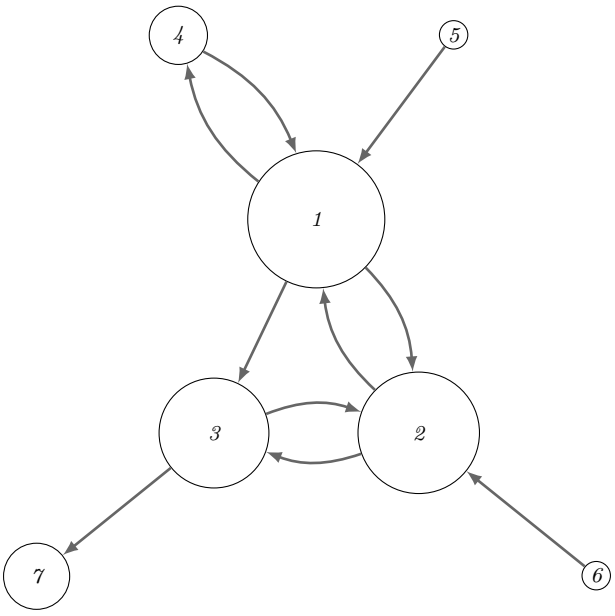
$$M = \left(\frac{0.15}{7} \cdot \mathbb{J} + 0.85 \cdot \widehat{A}\right)$$

$$\doteq \begin{pmatrix} 0.0214 & 0.4464 & 0.0214 & 0.8714 & 0.8714 & 0.0214 & 0.1429 \\ 0.3048 & 0.0214 & 0.4464 & 0.0214 & 0.0214 & 0.8714 & 0.1429 \\ 0.3048 & 0.4464 & 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.1429 \\ 0.3048 & 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.1429 \\ 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.1429 \\ 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.0214 & 0.1429 \\ 0.0214 & 0.0214 & 0.4464 & 0.0214 & 0.0214 & 0.0214 & 0.1429 \end{pmatrix}$$

Algorithmus 7 mit $\varepsilon = 0.001$ liefert nach 11 Iterationen den Rangvektor

$$R = (0.2578, 0.2285, 0.2066, 0.1096, 0.0365, 0.0365, 0.1244)^T .$$

Der Graph mit Knotengröße entsprechend seines Rangvektors sieht dann so aus:



5 Flussnetzwerke

Wir betrachten ein Netzwerk mit Knoten und Leitungen, die diese verbinden. Dabei nehmen wir an, dass Datenpakete von einem Quellknoten zu einem Zielknoten versendet werden sollen. Jede Leitung hat dabei eine gewisse Kapazität und wir wollen das Netz natürlich so effizient wie möglich verwenden. Mathematisch gesehen betrachten wir einen gerichteten Graphen $G = (V, E)$ mit der Eigenschaft, dass $(v, u) \in E$ automatisch $(u, v) \notin E$ impliziert. Wir nennen solche Graphen *orientiert* (engl. oriented). Weiters repräsentiert eine Gewichtsfunktion $\omega\colon E \longrightarrow \mathbb{N}$ die Kapazität der Leitungen (engl. capacity function). Wir benötigen noch zwei Knoten $s, t \in V$ für die Quelle und das Ziel, sodass $\deg_{\text{in}}(s) = 0$ und $\deg_{\text{out}}(t) = 0$. Das Tupel (G, ω, s, t) nennen wir ein *Netzwerk* (engl. network).

Definition 5.1.
Sei $N = (G, \omega, s, t)$ mit $G = (V, E)$ ein Netzwerk und f eine Abbildung $f\colon E \longrightarrow \mathbb{N}_0$ mit $f(e) \leq \omega(e)$ für alle $e \in E$. Die Abbildung f ist ein Fluss, wenn sie das Flusserhaltungsgesetz *(engl. conservation law) erfüllt, also, wenn für alle $v \in V \setminus \{s, t\}$*

$$\sum_{u \text{ Vorgänger von } v} f((u, v)) = \sum_{u \text{ Nachfolger von } v} f((v, u)).$$

Der Wert *des Flusses (engl. value) ist dann definiert durch*

$$\sum_{u \text{ Vorgänger von } t} f((u, t)) = \sum_{u \text{ Nachfolger von } s} f((s, u)).$$

Wir nennen so einen Fluss auch $(s - t)$-Fluss. Ein maximaler Fluss (engl. maximum flow) ist ein Fluss mit maximalem Wert unter allen $(s - t)$-Flüssen.

Unsere Problemstellung können wir also dadurch definieren für ein gegebenes Netzwerk einen $(s - t)$-Fluss mit maximalem Wert zu finden. Um einen Algorithmus für dieses Problem zu beschreiben, benötigen wir noch ein paar neue Begriffe.

Definition 5.2.
Für eine Kante $e = (u, v) \in E$, nennen wir die Kante $\overleftarrow{e} = (v, u)$ die gegenläufige Kante *(engl. reverse edge). Mit $\overrightarrow{\overleftarrow{E}}$ bezeichnen wir die Vereinigung von Kanten und gegenläufigen Kanten, als $\overrightarrow{\overleftarrow{E}} = E \cup \left\{\overleftarrow{e} \mid e \in E\right\}$. Wir definieren weiters eine neue Gewichtsfunktion $\omega_f\colon \overrightarrow{\overleftarrow{E}} \longrightarrow \mathbb{N}$ durch $\omega_f(e) = \omega(e) - f(e)$ und $\omega_f(\overleftarrow{e}) = f(e)$ für alle Kanten $e \in E$.*

Dadurch erhalten wir den sogenannten Residualgraphen *(engl. residual graph)* $G_f = (V, E_f)$ *mit Gewichten* ω_f *und* $E_f = \left\{ e \in \overrightleftharpoons{E} \mid \omega_f(e) \neq 0 \right\}$.

Wir betrachten nun einen Fluss f *und einen Weg* W *im Residualgraph* G_f. *Seien* F *die Kanten des Weges. Ein* augmentierender Weg *(engl. augmenting path) ist ein* $(s-t)$*-Weg im Residualgraph. Wir sagen ein Fluss wird* mit λ entlang W augmentiert, *wenn für alle* $e \in E \cap F$ *der Fluss* $f(e)$ *um* λ *erhöht wird und für alle* $e \in F \setminus E$ *der Fluss* $f(e)$ *um* λ *verringert wird.*

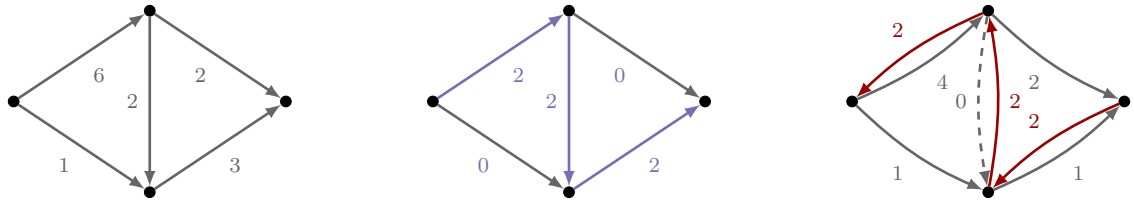
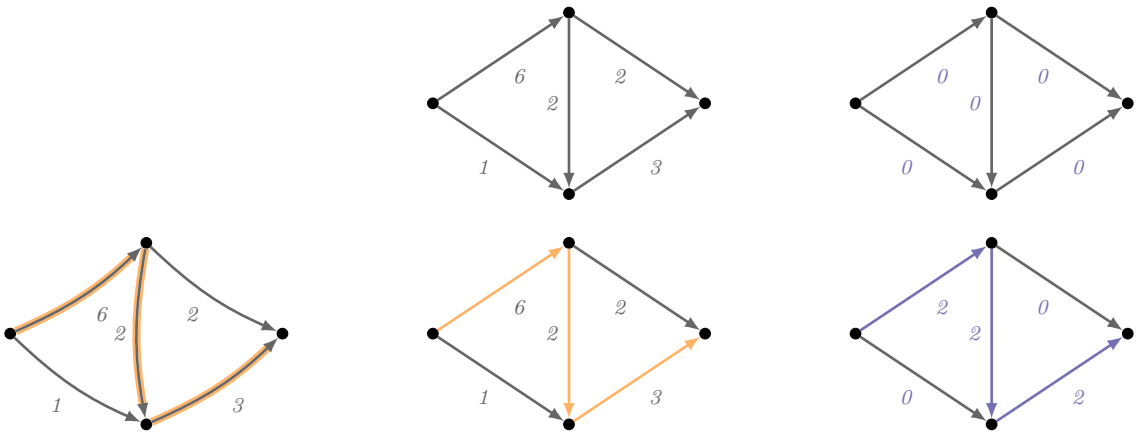


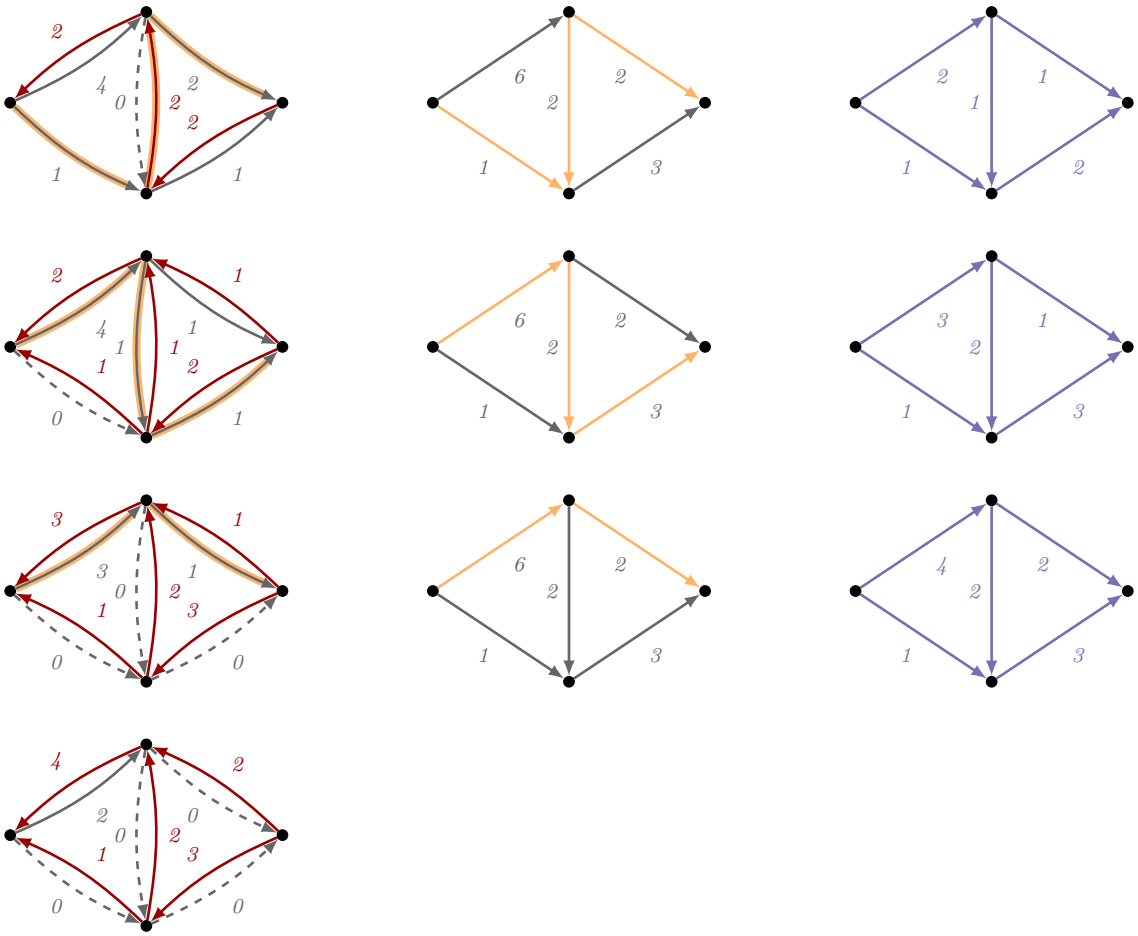
Abbildung 5.1: Graph (links), Fluss (Mitte) und Residualgraph (rechts)

Algorithmus 8 Ford-Fulkerson-Algorithmus

Eingabe: Netzwerk (G, ω, s, t)
Ausgabe: maximaler $(s-t)$-Fluss
  Setze $f(e) = 0$ für alle $e \in E$
  while Es gibt einen augmentierenden Weg W **do**
    $F \longleftarrow$ Kanten von W
    $\lambda \longleftarrow \min_{e \in F}(\omega_f(e))$
    Augmentiere f entlang W um λ
  end while
  return f

Beispiel 5.3.
Wir sehen uns den Algorithmus anhand des obigen Beispiels an.





Satz 5.4.
Ein $(s-t)$-Fluss ist genau dann maximal, wenn keine augmentierenden Wege mehr existieren.

Ein *Schnitt* eines zusammenhängenden Graphen (engl. cut) ist eine Menge an Kanten $S \subseteq E$, sodass der Graph $\langle E \setminus S \rangle$ nicht mehr zusammenhängt, aber für jede echte Teilmenge $S' \subset$ der Graph $\langle E \setminus S' \rangle$ noch zusammenhängend ist. Der Wert eines Schnittes ist die Summe der Kantengewichte in S.

Satz 5.5. (Max-Flow-Min-Cut-Theorem)
Der Wert eines maximalen $(s-t)$-Fluss in einem Netzwerk ist gleich dem minimalen Wert der Schnitte.

Index