5 Graphalgorithmen

5.1 Definitionen

5.1.1 Gerichtete und ungerichtete Graphen

- Ein *Graph* ist ein Paar $G = (V, E)$ mit einer endlichen Menge V von *Knoten* oder *Ecken* (engl. vertices) und einer zugehörigen Menge E von *Kanten* (engl. edges).

- Wenn E eine Teilmenge von $V \times V = \{ (u, v) \mid u, v \in V \}$ ist, heißt der Graph *gerichtet*. (Eine Kante $(u, v) \in E$ geht von u nach v, aber nicht umgekehrt.)

- Wenn E eine Teilmenge von $\{ \{ u, v \} \mid u, v \in V \}$ ist, heißt der Graph *ungerichtet*. (Eine Kante $\{ u, v \} = \{ v, u \} \in E$ geht sowohl von u nach v als auch umgekehrt.)

- Ein ungerichteter Graph kann auch als gerichteter Graph aufgefasst werden, in dem es zu jeder Kante (u, v) auch die entgegengesetzte Kante (v, u) gibt.

- Eine Kante (v, v) oder $\{ v, v \} = \{ v \}$ von einem Knoten v zu sich selbst heißt *Schlinge*.

5.1.2 Adjazenzlisten und -matrizen

- Wenn es in einem Graphen eine Kante von einem Knoten u zu einem Knoten v gibt, heißt u *Vorgänger* von v und v *Nachfolger* von u.

- Die *Adjazenzliste* eines Knotens enthält alle Nachfolger dieses Knotens in irgendeiner Reihenfolge.

- Die *Adjazenzlistendarstellung* eines Graphen besteht aus den Adjazenzlisten aller Knoten des Graphen.

- Die *Adjazenzmatrix* eines Graphen mit $N = |V|$ Knoten ist eine Matrix A mit $N \times N$ Elementen a_{uv}. Jedes a_{uv} ist 1, wenn es eine Kante von u nach v gibt, andernfalls 0.

  Formal: $A: V \times V \to \{0, 1\}$ mit $A(u, v) = \begin{cases} 1, \text{ wenn } (u, v) \in E \text{ bzw. } \{u, v\} \in E \\ 0 \text{ sonst} \end{cases}$

  Folgerung: $|E| \leq |V|^2$

- Die Adjazenzmatrix eines ungerichteten Graphen ist symmetrisch.

- Die Adjazenzlistendarstellung eines Graphen $G = (V, E)$ hat die Größe $O(|V| + |E|)$, die Adjazenzmatrix $O\left(|V|^2\right)$.

5.1.3 Weitere Begriffe

- Ein (einfacher) *Weg* oder *Pfad* von einem Knoten u zu einem Knoten v ist eine Folge paarweise verschiedener Knoten $w_0, \ldots, w_n$ mit $u = w_0$, Kanten von w_{i-1} nach w_i für $i = 1, \ldots, n$ und $w_n = v$.

- Die Anzahl n der Kanten heißt *Länge* des Wegs.
  (Der Fall $n = 0$ als *leerer Weg* von einem Knoten zu sich selbst ist zulässig.)

- Wenn es einen Weg von u nach v gibt, heißt v von u aus *erreichbar*.
  (Insbesondere ist jeder Knoten von sich selbst aus erreichbar.)

- Die *Distanz* $\delta(u, v)$ zwischen u und v ist entweder die Länge eines kürzesten Wegs von u nach v, falls v von u aus erreichbar ist, oder andernfalls ∞.
  (Insbesondere ist $\delta(v, v) = 0$ für jeden Knoten $v \in V$.)

- Wenn $w_0, \ldots, w_n$ ein Weg ist und es eine Kante von w_n nach w_0 gibt, heißt die Knotenfolge $w_0, \ldots, w_n, w_0$ *Zyklus* oder *Kreis*.
  (Beachte: Die Knotenfolge $w_0, \ldots, w_n, w_0$ ist kein Weg, weil ihre Knoten nicht alle verschieden sind.)

- Ein Graph ohne Zyklen heißt *azyklisch* oder *zyklenfrei*.

5.2 Breitensuche (breadth-first search)

5.2.1 Problemstellung

Gegeben

- Graph $G = (V, E)$
- Startknoten $s \in V$

Gesucht

- Alle Knoten $v \in V$, die vom Startknoten s aus erreichbar sind

Genauer

- $\delta(s, v)$ für alle Knoten $v \in V$
- Ein kürzester Weg von s nach v für alle von s aus erreichbaren Knoten $v \in V$

5.2.2 Algorithmus

1 Für jeden Knoten $v \in V \setminus \{ s \}$:

  Setze $\delta(v) = \infty$ und $\pi(v) = \perp$.

2 Setze $\delta(s) = 0$ und $\pi(s) = \perp$.

3 Füge s in eine FIFO-Warteschlange ein.

4 Solange die Warteschlange nicht leer ist:

  1 Entnimm den ersten Knoten u aus der Warteschlange.

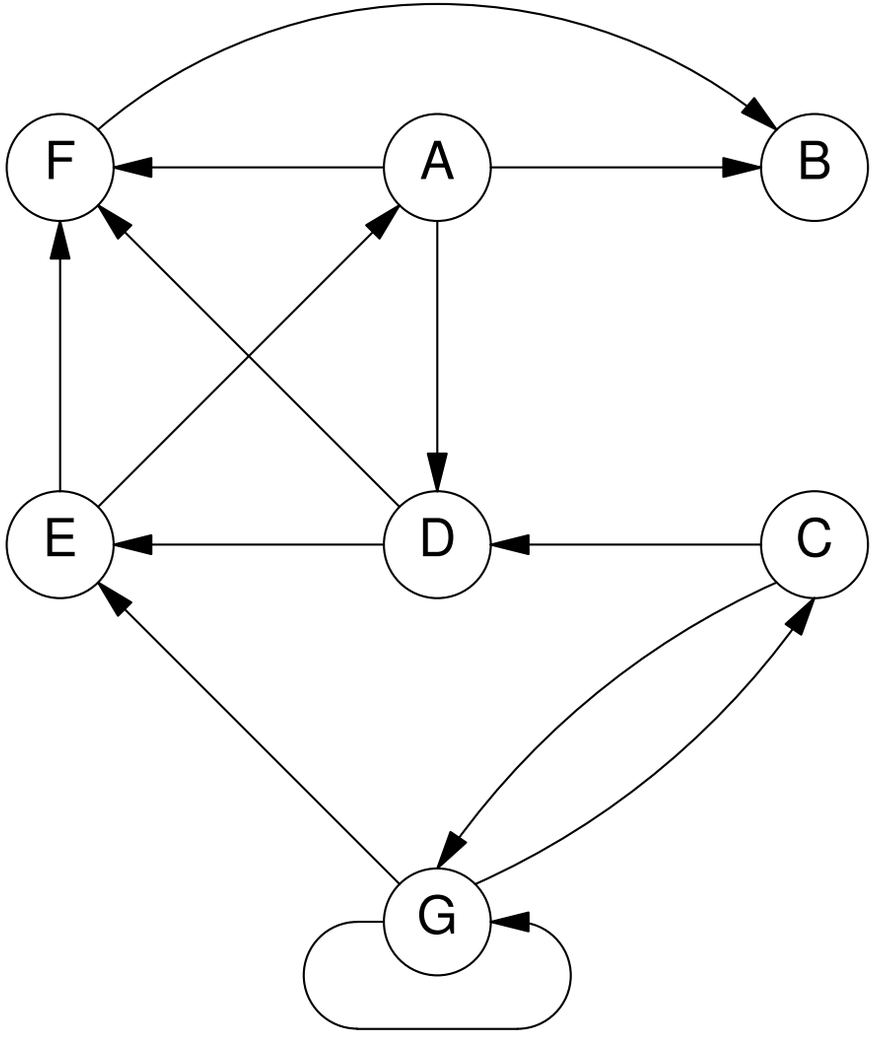
  2 Für jeden Nachfolger v von u:

    Wenn $\delta(v) = \infty$ ist:

    1 Setze $\delta(v) = \delta(u) + 1$ und $\pi(v) = u$.

    2 Füge v am Ende der Warteschlange an.

5.2.3 Beispiel



5.2.4 Laufzeit

- Initialisierung (Schritt 1 und 2): $O(|V|)$
- Eigentliche Suche (Schritt 3 und 4): $O(|V| + |E|)$
  - Die äußere Schleife wird für jeden Knoten höchstens einmal durchlaufen, da jeder Knoten höchstens einmal in die Warteschlange eingefügt wird.
  - Damit wird die innere Schleife insgesamt höchstens $\sum_{u \in V} \gamma(u) = |E|$ mal durchlaufen, wobei $\gamma(u)$ = Grad des Knotens u = Anzahl der Nachfolger von u = Anzahl der von u ausgehenden Kanten.
- Insgesamt also: $O(|V| + |E|)$

5.2.5 Ergebnis

Nach Ausführung des Algorithmus gilt:

- $\delta(v) = \delta(s, v)$ für alle $v \in V$
- Wenn $v \neq s$ von s aus erreichbar ist, dann ist $\pi(v)$ der Vorgänger von v auf einem kürzesten Weg von s nach v, das heißt: Der kürzeste Weg von s nach v lautet $w_n, \ldots, w_0$ mit $w_0 = v$ und $w_i = \pi(w_{i-1})$ für $i = 1, \ldots, n$ mit $n = \delta(v)$.

5.2.6 Vorgängergraph einer Breitensuche (Breitensuchebaum)

- Der Graph $G_\pi = (V_\pi, E_\pi)$ mit
  - $V_\pi = \{ v \in V \mid \delta(v) < \infty \}$ = Menge aller vom Startknoten s aus erreichbaren Knoten
  - $E_\pi = \{ (\pi(v), v) \mid v \in V_\pi \setminus \{ s \} \} = \{ (\pi(v), v) \mid \pi(v) \neq \perp \} \subseteq E$

  ist ein Baum mit Wurzel s,
  der alle von s aus erreichbaren Knoten des Graphen G enthält.

- Auf Ebene d dieses Baums befinden sich alle Knoten v mit $\delta(s, v) = d$.

5.2.7 Anwendungsbeispiel

Automatische Speicherbereinigung (garbage collection), z. B. in Java

- $V = \{ v \mid$ Objekt v wurde mit `new` erzeugt $\}$
- $E = \{ (u, v) \in V \times V \mid$ eine Objektvariable von u verweist auf $v \}$
- $S = \{ s \in V \mid$ eine Klassenvariable oder eine lokale Variable verweist auf Objekt $s \}$
- Die Breitensuche wird (nach einmaliger Initialisierung aller Knoten/Objekte) für alle Objekte $s \in S$ nacheinander ausgeführt.
- $\pi(v)$ wird nicht benötigt.
- Statt $\delta(v)$ genügt die binäre Information, ob v von irgendeinem Objekt $s \in S$ aus erreichbar ist oder nicht.
- Objekte, die auf diese Weise nicht erreichbar sind, können gelöscht werden.

5.3 Tiefensuche (depth-first search) und topologische Sortierung

5.3.1 Problemstellung

- Ordne die Knoten eines Graphen $G = (V, E)$ hierarchisch von links nach rechts an, sodass keine Kanten von links nach rechts zeigen.
- Nebenbei kann überprüft werden, ob der Graph einen Zyklus enthält oder nicht.
- Wenn er azyklisch ist, können die Knoten auch sequentiell von links nach rechts angeordnet werden, sodass alle Kanten von rechts nach links zeigen (*topologische Sortierung*).
- Während der Suche erhält jeder Knoten $v \in V$ einen *Vorgänger* $\pi(v)$ im *Tiefensuche-wald* sowie eine *Entdeckungszeit* $\delta(v) \in \mathbb{N}$ und eine *Abschlusszeit* $\varphi(v) \in \mathbb{N}$ mit $1 \leq \delta(v) < \varphi(v) \leq 2\,|V|$.
- Damit besitzt jeder Knoten außerdem zu jedem Zeitpunkt eine gedachte *Farbe*:
  - Ein Knoten ist *weiß*, wenn er noch nicht entdeckt wurde, d. h. wenn er noch keine Entdeckungs- und Abschlusszeit besitzt.
  - Ein Knoten ist *grau*, wenn er gerade bearbeitet wird, d. h. wenn er eine Entdeckungs-, aber noch keine Abschlusszeit besitzt.
  - Ein Knoten ist *schwarz*, wenn seine Bearbeitung abgeschlossen ist, d. h. wenn er sowohl eine Entdeckungs- als auch eine Abschlusszeit besitzt.

5.3.2 Algorithmus

Für jeden Knoten $u \in V$:

Wenn u weiß ist:

1. Setze $\pi(u) = \perp$.
2. Durchsuche den zu u gehörenden Teilgraphen, das heißt:
   1. Setze $\delta(u)$ auf den nächsten Zeitwert aus der Menge $\{1, \ldots, 2 \cdot |V|\}$.
   2. Für jeden Nachfolger v von u:

      Wenn v weiß ist:

      1. Setze $\pi(v) = u$.
      2. Durchsuche rekursiv den zu v gehörenden Teilgraphen.
   3. Setze $\varphi(u)$ auf den nächsten Zeitwert aus der Menge $\{1, \ldots, 2 \cdot |V|\}$.

5.3.3 Beispiel

❒ Siehe § 5.2.3.

5.3.4 Laufzeit

- Die äußere Schleife wird für jeden Knoten genau einmal durchlaufen.
- Die rekursive Operation „Durchsuchen“ wird für jeden Knoten genau einmal ausgeführt.
- Damit wird die innere Schleife in dieser Operation insgesamt $\sum_{u \in V} \gamma(u) = |E|$ mal durchlaufen.
- Damit ergibt sich als Laufzeit: $O(|V| + |E|)$

5.3.5 Vorgängergraph einer Tiefensuche (Tiefensuchewald)

- Der Graph $G_\pi = (V, E_\pi)$ mit $E_\pi = \{ (\pi(v), v) \mid v \in V, \pi(v) \neq \perp \} \subseteq E$ ist eine Menge von Bäumen, d. h. ein Wald.
- Jeder Knoten v mit $\pi(v) = \perp$ ist der Wurzelknoten eines dieser Bäume.

5.3.6 Klassifikation der Kanten

Nach der Ausführung einer Tiefensuche gilt für die Bearbeitungszeiträume $\Sigma(u) = [\delta(u), \varphi(u)]$ und $\Sigma(v) = [\delta(v), \varphi(v)]$ zweier beliebiger Knoten $u, v \in V$ mit $u \neq v$ immer genau eine der folgenden Beziehungen:

- $\Sigma(u) \supset \Sigma(v)$, d. h. $\delta(u) < \delta(v) < \varphi(v) < \varphi(u)$
  - v wurde während der Bearbeitung von u entdeckt.
  - Damit ist v ein direkter oder indirekter Nachfolger von u in einem Baum des Tiefensuchewalds.
  - Wenn es eine Kante von u nach v gibt, geht sie im Tiefensuchewald von oben nach unten und wird deshalb entweder als *Baumkante* (tree edge) – wenn sie zur Menge E_π gehört – oder andernfalls als *Abwärts-* oder *Vorwärtskante* (forward edge) bezeichnet.
- $\Sigma(u) \subset \Sigma(v)$, d. h. $\delta(v) < \delta(u) < \varphi(u) < \varphi(v)$
  - u wurde während der Bearbeitung von v entdeckt.
  - Damit ist v ein direkter oder indirekter Vorgänger von u in einem Baum des Tiefensuchewalds.
  - Wenn es eine Kante von u nach v gibt, geht sie im Tiefensuchewald von unten nach oben und wird deshalb als *Aufwärts-* oder *Rückwärtskante* (back edge) bezeichnet.
  - Eine Schlinge wird ebenfalls so bezeichnet.

- $\Sigma(u) > \Sigma(v)$, d. h. $\delta(v) < \varphi(v) < \delta(u) < \varphi(u)$
  - u wurde nach der Bearbeitung von v entdeckt.
  - Damit ist v weder ein Nachfolger noch ein Vorgänger von u in einem Baum des Tiefensuchewalds.
  - Wenn es eine Kante von u nach v gibt, geht sie im Tiefensuchewald von rechts nach links und wird deshalb als *Querkante* (cross edge) bezeichnet.
- $\Sigma(u) < \Sigma(v)$, d. h. $\delta(u) < \varphi(u) < \delta(v) < \varphi(v)$
  - v wurde nach der Bearbeitung von u entdeckt.
  - Damit ist v weder ein Nachfolger noch ein Vorgänger von u in einem Baum des Tiefensuchewalds.
  - Wenn es eine Kante von u nach v gäbe, müsste sie im Tiefensuchewald von links nach rechts gehen.
  - Aber wenn es eine solche Kante gäbe, dann hätte der Algorithmus den Knoten v spätestens während der Bearbeitung von u über diese Kante entdeckt, d. h. dann würde entweder $\Sigma(v) < \Sigma(u)$ (Entdeckung und Bearbeitung von v bereits vor der Bearbeitung von u) oder $\Sigma(v) \subset \Sigma(u)$ (Entdeckung und Bearbeitung von v während der Bearbeitung von u) gelten.
  - Daher gibt es im Tiefensuchewald niemals Kanten von links nach rechts.

Aufgrund der Rekursionsstruktur des Algorithmus ist eine Überlappung der Bearbeitungszeiträume, d. h. $\delta(u) < \delta(v) < \varphi(u) < \varphi(v)$ oder $\delta(v) < \delta(u) < \varphi(v) < \varphi(u)$, nicht möglich.

5.3.7 Topologische Sortierung

Problemstellung

- Die Kanten eines azyklischen gerichteten Graphen können als Abhängigkeitsbeziehungen zwischen den Knoten interpretiert werden, d. h. $(u, v) \in E$ bedeutet, dass Knoten u irgendwie von Knoten v abhängt.
- Gesucht ist eine sequentielle Anordnung der Knoten von links nach rechts, in der jeder Knoten erst nach den Knoten kommt, von denen er abhängt, d. h. in der alle Kanten von rechts nach links verlaufen.

Lösung: Erweiterte Tiefensuche

- Für jeden Nachfolger v von u wird in Schritt 2.2 der Tiefensuche zusätzlich überprüft, ob er grau ist. Wenn ja, handelt es sich bei der Kante (u, v) um eine Rückwärtskante. In diesem Fall enthält der Graph einen Zyklus und kann daher nicht topologisch sortiert werden.
- Wenn jeder Knoten u nach Abschluss seiner Verarbeitung am Ende einer linearen Liste angefügt wird, enthält diese Liste nach Beendigung der Tiefensuche die Knoten in der gewünschten Reihenfolge.

Anwendungsbeispiele

- Abhängigkeiten zwischen Software-Komponenten, Begriffsdefinitionen, ...

5.4 Zusammenhangskomponenten

5.4.1 Definitionen

- Eine *Verbindung* zweier Knoten u und v ist eine Folge paarweise verschiedener Knoten $w_0, \ldots, w_n$ mit $u = w_0$, Kanten von w_{i-1} nach w_i oder umgekehrt für $i = 1, \ldots, n$ und $w_n = v$. (In einem ungerichteten Graphen sind Verbindung und Weg gleichbedeutend.)
- Ein Graph heißt *zusammenhängend*, wenn es von jedem Knoten eine Verbindung zu jedem anderen Knoten gibt.
- Eine *Zusammenhangskomponente* eines Graphen ist eine Äquivalenzklasse der Relation „es gibt eine Verbindung von u nach v“, d. h. zwei Knoten gehören genau dann zur gleichen Zusammenhangskomponente, wenn es eine Verbindung zwischen ihnen gibt.
- Ein gerichteter Graph heißt *stark zusammenhängend*, wenn es von jedem Knoten einen Weg zu jedem anderen Knoten gibt, d. h. wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- Eine *starke Zusammenhangskomponente* eines gerichteten Graphen ist eine Äquivalenzklasse der Relation „u ist von v aus erreichbar und umgekehrt“, d. h. zwei Knoten u und v gehören genau dann zur gleichen starken Zusammenhangskomponente, wenn u von v aus erreichbar ist und umgekehrt.

- Ein Graph ist genau dann (stark) zusammenhängend, wenn er genau eine (starke) Zusammenhangskomponente besitzt.
- Für einen gerichteten Graphen $G = (V, E)$ heißt der Graph $G^T = \left(V, E^T\right)$ mit $E^T = \{ (v, u) \mid (u, v) \in E \}$ *transponierter Graph* von G.
  (Die Adjazenzmatrix von G^T ist die transponierte Matrix der Adjazenzmatrix von G.)
- Offensichtlich besitzen G und G^T die gleichen starken Zusammenhangskomponenten.

5.4.2 Bestimmung von Zusammenhangskomponenten

- Die Zusammenhangskomponenten eines ungerichteten Graphen können direkt durch eine Tiefensuche bestimmt werden:
  Jeder Baum des resultierenden Tiefensuchewalds entspricht direkt einer Zusammenhangskomponente.

- Die Zusammenhangskomponenten eines gerichteten Graphen $G = (V, E)$ können analog durch eine Tiefensuche im Graphen $G' = \left(V, E \cup E^{\mathrm{T}}\right)$ bestimmt werden.

- Die starken Zusammenhangskomponenten eines gerichteten Graphen $G = (V, E)$ können wie folgt durch zwei aufeinanderfolgende Tiefensuchen bestimmt werden:
  - Führe eine erste Tiefensuche auf G aus, um die Abschlusszeiten $\varphi(u)$ aller Knoten $u \in V$ zu bestimmen.
  - Führe eine zweite Tiefensuche auf G^{T} aus, in der die Knoten u in der äußeren Schleife in absteigender Reihenfolge dieser Abschlusszeiten $\varphi(u)$ durchlaufen werden.
  - Jeder Baum des resultierenden Tiefensuchewalds der zweiten Tiefensuche entspricht einer starken Zusammenhangskomponente von G (und von G^{T}).
  - Beispiel: Siehe § 5.2.3

5.4.3 Korrektheit der Bestimmung starker Zusammenhangskomponenten

Behauptung

Jeder Baum des zweiten Tiefensuchewalds ist eine starke Zusammenhangskomponente des Graphen G.

Beweis durch vollständige Induktion

nach der Anzahl n der bis jetzt von der zweiten Tiefensuche ermittelten Bäume

Induktionsanfang $n = 0$: Hier ist nichts zu zeigen.

Induktionsschritt $n \to n + 1$:

- Nach Induktionsvoraussetzung sind die bis jetzt ermittelten n Bäume starke Zusammenhangskomponenten des Graphen.
- Zu zeigen:
  Der nächste ermittelte Baum ist ebenfalls eine starke Zusammenhangskomponente.
- Sei r der Knoten, für den in der äußeren Schleife der zweiten Tiefensuche als nächstes die Operation „Durchsuchen“ ausgeführt wird und der somit die Wurzel dieses nächsten Tiefensuchebaums wird.

- Die zweite Tiefensuche findet ausgehend von diesem Wurzelknoten r
  - keine Knoten, die im ersten Tiefensuchewald links von r liegen, weil es in diesem Tiefensuchewald gemäß § 5.3.6 keine Kanten von links nach rechts und dementsprechend nach Transponierung des Graphen keine Kanten von rechts nach links gibt;
  - keine Knoten, die im ersten Tiefensuchewald rechts oder oberhalb von r liegen, weil diese eine größere Abschlusszeit als r besitzen und deshalb von der zweiten Tiefensuche (deren Hauptschleife nach absteigenden Abschlusszeiten durchlaufen wird) bereits früher gefunden wurden;
  - folglich nur Knoten, die im ersten Tiefensuchewald unterhalb von r liegen und deshalb auch in G von r aus erreichbar sind und somit zur gleichen starken Zusammenhangskomponente wie r gehören.
- Außerdem findet eine Tiefensuche immer *alle* erreichbaren Knoten, die nicht schon früher gefunden wurden.
  Das bedeutet umgekehrt für alle Knoten v, die dabei nicht gefunden werden:
  - Entweder ist v in G^T nicht von r aus erreichbar und gehört deshalb nicht zur starken Zusammenhangskomponente von r.
  - Oder v wurde bereits früher gefunden und gehört deshalb nach Induktionsvoraussetzung zu einer anderen starken Zusammenhangskomponente.
- Also werden *genau* diejenigen Knoten gefunden, die zur starken Zusammenhangskomponente von r gehören.

5.5 Minimale Spannbäume und Gerüste

5.5.1 Bäume und Wälder

- Ein zusammenhängender Graph $G = (V, E)$ heißt *Baum*, wenn $|E| = |V| - 1$ gilt. (Zu jedem Knoten außer der Wurzel gibt es genau eine Kante, die ihn mit seinem übergeordneten Knoten verbindet. Je nach Art des Baums, können diese Kanten ungerichtet, „von oben nach unten“ gerichtet oder „von unten nach oben“ gerichtet sein. Bei einem ungerichteten Baum kann jeder Knoten die Rolle des Wurzelknotens spielen.)

- Eine Menge von Bäumen heißt *Wald*.

- *Anmerkung:* Für jeden zusammenhängenden Graphen gilt: $|E| \geq |V| - 1$.
  Damit ist ein Baum ein zusammenhängender Graph mit möglichst wenig Kanten.

Alternative Definition

- Ein Graph ohne Schlingen heißt Baum, wenn es zwischen je zwei Knoten genau eine Verbindung gibt.

- Ein Graph ohne Schlingen heißt Wald, wenn es zwischen je zwei Knoten höchstens eine Verbindung gibt.

Anmerkung

- Da ein ungerichteter Graph mit mindestens einer Kante $\{u, v\} = \{v, u\}$ gemäß § 5.1.3 immer Zyklen u, v, u und v, u, v enthält, ist die Charakterisierung eines Baums als zusammenhängender, azyklischer Graph (vgl. Wikipedia) für ungerichtete Graphen falsch.

- Um solche „unerwünschten“ Zyklen zu vermeiden, könnte man zusätzlich verlangen, dass die Kanten auf einem Zyklus paarweise verschieden sind.

- Dann bestünde jedoch ein feiner Unterschied zwischen einem ungerichteten Graphen und dem entsprechenden gerichteten Graphen, in dem es zu jeder Kante auch die entgegengesetzte Kante gibt: Der ungerichtete Graph könnte dann azyklisch sein, der gerichtete Graph wäre jedoch immer zyklisch (sofern es mindestens eine Kante gibt).

Lemma

1. Wenn man aus einem Baum eine Kante entfernt, zerfällt er in zwei Bäume.

2. Wenn man zwei Bäume durch eine Kante verbindet, entsteht ein einziger Baum.

3. Wenn man zu einem Baum eine Kante von einem Knoten u zu einem Knoten v hinzufügt und gleichzeitig eine Kante auf der Verbindung von u nach v entfernt, entsteht wieder ein Baum.

4. Wenn man zu einem Wald eine Kante von einem Knoten u zu einem Knoten v hinzufügt, die bereits miteinander verbunden sind, und gleichzeitig eine Kante auf dieser Verbindung entfernt, entsteht wieder ein Wald mit gleich vielen Bäumen.

Beweis jeweils mit der Definition, dass es in einem Baum zwischen je zwei Knoten genau eine Verbindung gibt.

5.5.2 Spannbäume und Gerüste

- Ein *Gerüst* oder *Spannwald* (spanning forest) eines ungerichteten Graphen $G = (V, E)$ ist ein Wald (V, F) mit $F \subseteq E$, der die gleichen Zusammenhangskomponenten wie G besitzt.

- Das heißt:
  - Ein Gerüst eines Graphen enthält alle Knoten des Graphen und eine möglichst kleine Teilmenge seiner Kanten.
  - Jeder Zusammenhangskomponente des Graphen entspricht ein Baum im Gerüst.
  - Insbesondere gibt es für jede Kante $\{u, v\} \in E$ des Graphen einen Weg im Gerüst von u nach v.

- Abkürzend wird auch die Kantenmenge F eines Gerüsts (V, F) als Gerüst bezeichnet.

- Wenn der Graph G zusammenhängend ist, ist jedes Gerüst von G ein Baum, der dann als *Spannbaum* (spanning tree) von G bezeichnet wird.

5.5.3 Minimale Spannbäume und Gerüste

- Ein *gewichteter Graph* $G = (V, E, \rho)$ ist ein Graph (V, E) mit einer zugehörigen *Gewichtsfunktion* $\rho: E \to \mathbb{R}$, die jeder Kante $e \in E$ ein *Gewicht* $\rho(e) \in \mathbb{R}$ zuordnet.

- Abkürzend wird auch $\rho(u, v)$ statt $\rho((u, v))$ (Gewicht der gerichteten Kante (u, v)) bzw. $\rho(\{u, v\})$ (Gewicht der ungerichteten Kante $\{u, v\}$) geschrieben.

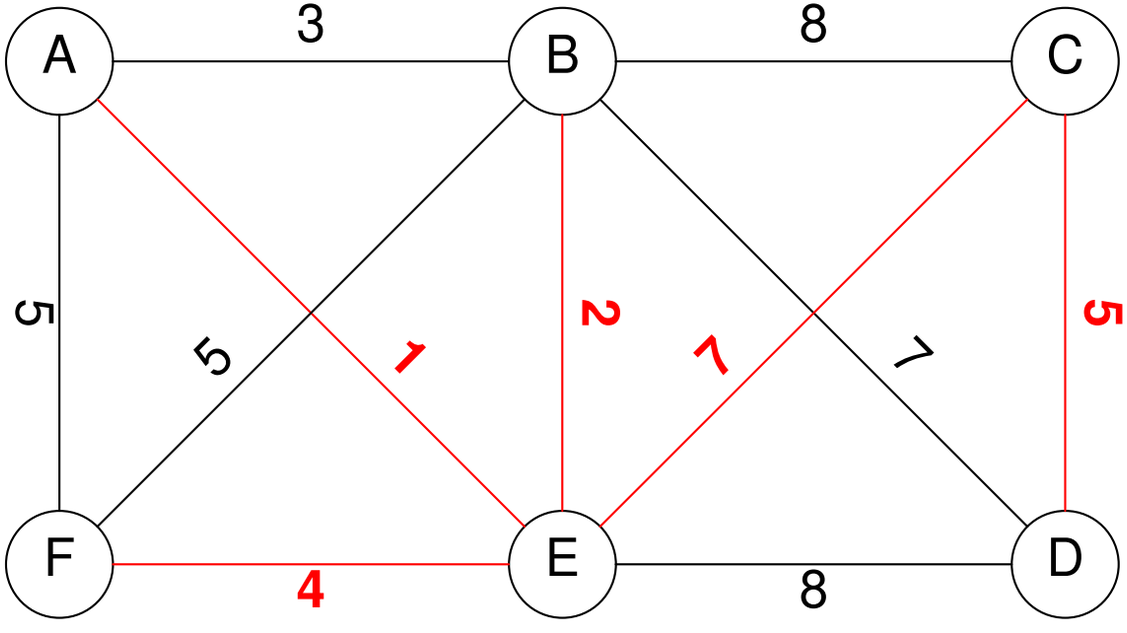
- Sofern nichts anderes gesagt wird, sind prinzipiell auch negative Kantengewichte zulässig.

- Das Gewicht einer Kantenmenge $F \subseteq E$ ist die Summe $\rho(F) = \sum_{e \in F} \rho(e)$ der Gewichte aller Kanten $e \in F$.

- Ein *Minimalgerüst* (oder *minimales Gerüst*, *minimaler Spannwald*; minimum spanning forest) eines ungerichteten, gewichteten Graphen $G = (V, E, \rho)$ ist ein Gerüst F des Graphen mit minimalem Gewicht, d. h. $\rho(F) \leq \rho(F')$ für jedes Gerüst F' von G.

- Wenn der Graph zusammenhängend ist, ist jedes Minimalgerüst ein Baum, der dann als *minimaler Spannbaum* (minimum spanning tree, MST) des Graphen bezeichnet wird.

5.5.4 Beispiel



Die roten Kanten bilden einen von zwei möglichen minimalen Spannbäumen des Graphen.

5.5.5 Anwendungsbeispiele

- Verbindung elektrischer Bauteile mit möglichst wenig Draht
- Approximationsverfahren zur Lösung des Problems des Handlungsreisenden (vgl. § 5.7.3)

5.5.6 Schleifeninvarianten

Definition

- Eine *Schleifeninvariante* ist eine Aussage, die an folgenden Stellen gilt:
  - Vor Beginn einer Schleife
  - Am Anfang und am Ende jedes Schleifendurchlaufs
  - Nach Beendigung der Schleife

Verwendung in Beweisen

- Um zu beweisen, dass eine bestimmte Aussage tatsächlich eine Schleifeninvariante darstellt, genügt es (ähnlich wie bei vollständiger Induktion), zwei Dinge zu zeigen (unter der Annahme, dass die Auswertung der Schleifenbedingung keine Nebeneffekte verursacht):
  - *Initialisierung:*
    Die Invariante gilt vor Beginn der Schleife.
    (Dann gilt sie auch am Anfang des ersten Durchlaufs, sofern es überhaupt einen Durchlauf gibt.)
  - *Aufrechterhaltung:*
    Wenn die Invariante am Anfang eines Durchlaufs gilt, dann gilt sie auch am Ende dieses Durchlaufs (und damit auch am Anfang des nächsten Durchlaufs, sofern es einen solchen gibt).
    (Zusammen mit der Initialisierung folgt dann durch vollständige Induktion, dass die Invariante nach *jedem* Durchlauf gilt.)

- Daraus folgt dann automatisch:
  - *Terminierung:*
    Wenn die Schleife terminiert (was ggf. anderweitig gezeigt werden muss, sofern es nicht offensichtlich ist), dann gilt die Invariante auch nach ihrer Beendigung.

Beispiel

```
int pow (int x, int n) {
  int p = 1, k = 0;
  while (k < n) {
    p = p * x;
    k = k + 1;
  }
  return p;
}
```

- Um zu zeigen, dass diese Funktion für $n \geq 0$ tatsächlich x^n berechnet, kann die Schleifeninvariante $p = x^k$ verwendet werden:

- Initialisierung:
  Vor Beginn der Schleife gilt: $p = 1 = x^0 = x^k$
- Aufrechterhaltung:
  Wenn die Aussage $p = x^k$ am Anfang eines Durchlaufs gilt, dann gilt am Ende dieses Durchlaufs: $p' = p \cdot x = x^k \cdot x = x^{k+1} = x^{k'}$.
  Dabei bezeichnen p und k die Werte der entsprechenden Variablen am Anfang des Durchlaufs, p' und k' ihre Werte am Ende des Durchlaufs.
- Terminierung:
  Nach Beendigung der Schleife ist $k = n$ (sofern $n \geq 0$ ist) und somit gilt:
  $p = x^k = x^n$.

Anmerkung

- for-Schleifen sind für derartige Beweise aus mehreren Gründen schlecht geeignet:
  - Die Veränderung der Laufvariablen erfolgt (je nach konkreter syntaktischer Form) mehr oder weniger „versteckt“.
  - Vor Beginn und nach Beendigung der Schleife existiert die Laufvariable meist gar nicht.

  Deshalb sollten for-Schleifen in äquivalente while-Schleifen umgeschrieben werden, wenn die Laufvariable in der Schleifeninvariante gebraucht wird.

5.5.7 Algorithmus von Kruskal

Gegeben

- Ungerichteter, gewichteter Graph $G = (V, E, \rho)$

Algorithmus

1. Setze $F = \emptyset$.
2. Sortiere die Kantenmenge E nach aufsteigenden Gewichten.
3. Für jede Kante $\{u, v\} \in E$ in dieser sortierten Reihenfolge:

   Wenn die Knoten u und v noch nicht durch Kanten aus F verbunden sind:

   Füge die Kante $\{u, v\}$ zur Menge F hinzu.

Ergebnis

- Nach Ausführung des Algorithmus ist die Kantenmenge F ein Minimalgerüst von G (das aus $|V| - |F|$ Bäumen besteht).

- Wenn G zusammenhängend ist (d. h. wenn $|V| - |F| = 1$ ist), ist F folglich ein minimaler Spannbaum von G.

Beispiel

❒ Siehe § 5.5.4.

Korrektheit

1. Nachdem eine Kante $\{u, v\} \in E$ in Schritt 3 des Algorithmus verarbeitet wurde, sind die Knoten u und v durch Kanten aus F verbunden.

2. Die Kantenmenge F ist zu jedem Zeitpunkt Teilmenge eines Minimalgerüsts von G, d. h. es gibt immer ein Minimalgerüst M mit $F \subseteq M$.

3. Nach Ausführung des Algorithmus ist die Kantenmenge F ein Minimalgerüst von G.

Beweis von Aussage 1:

❒ Wenn u und v vor der Verarbeitung der Kante $\{u, v\}$ bereits durch Kanten aus F verbunden sind, ist nichts zu zeigen.

❒ Andernfalls wird die Kante zur Menge F hinzugefügt, sodass die Behauptung anschließend erfüllt ist.

Beweis von Aussage 2 als Schleifeninvariante:

- Initialisierung:
  Vor Beginn der Schleife in Schritt 3 des Algorithmus ist $F = \emptyset$, und somit gilt trivialerweise $F \subseteq M$ für jedes Minimalgerüst M.

- Aufrechterhaltung:
  - Bevor in Schritt 3 eine Kante $e = \{u, v\}$ zur Menge F hinzugefügt wird, ist die ursprüngliche Menge F aufgrund der Invariante Teilmenge eines Minimalgerüsts M.
  - Da M ein Gerüst von G ist, gibt es in M einen Weg von u nach v.
  - Da u und v noch nicht durch Kanten aus F verbunden sind, muss es auf diesem Weg mindestens eine Kante $e' = \{u', v'\}$ geben, deren Knoten u' und v' ebenfalls noch nicht durch Kanten aus F verbunden sind.
    Insbesondere muss $e' \notin F$ und somit $F \subseteq M \setminus \{e'\}$ gelten.
  - Deshalb gilt für die neue Menge $F' = F \cup \{e\}$: $F' \subseteq M \setminus \{e'\} \cup \{e\} =: M'$, wobei M' aufgrund des Lemmas in § 5.5.1 ebenfalls ein Gerüst von G ist.
  - Da u' und v' noch nicht durch Kanten aus F verbunden sind, wurde die Kante e' wegen Aussage 1 vom Algorithmus noch nicht verarbeitet.
    Da die Kanten nach aufsteigendem Gewicht verarbeitet werden, gilt folglich $\rho(e) \leq \rho(e')$ und somit $\rho(M') = \rho(M) - \rho(e') + \rho(e) \leq \rho(M)$.

- Da M bereits ein Minimalgerüst von G ist, muss somit auch M' ein Minimalgerüst sein.
- Wegen $F' \subseteq M'$ ist die neue Menge F' also wiederum Teilmenge eines Minimalgerüsts von G.

Beweis von Aussage 3:

- Nach Aussage 2 gilt nach Ausführung des Algorithmus:
  $F \subseteq M$ für ein Minimalgerüst M von G.
- Für jede Kante $\{u, v\} \in M$ gilt:
  - Nach Aussage 1 sind die Knoten u und v durch Kanten aus F verbunden.
  - Wegen $F \subseteq M$ sind u und v in M durch dieselben Kanten verbunden.
  - Da es im Gerüst M höchstens eine Verbindung von u und v gibt, kann diese Verbindung nur aus der Kante $\{u, v\}$ bestehen.
  - Also muss die Kante $\{u, v\}$ auch zur Menge F gehören.
- Also gilt $F = M$, d. h. F ist ein Minimalgerüst von G.

Hilfsdatenstruktur zur effizienten Implementierung

- Um effizient überprüfen zu können, ob zwei Knoten u und v bereits durch Kanten aus F verbunden sind, werden die Knoten des Graphen sukzessive zu Bäumen zusammengefasst (deren Kanten jedoch nichts mit den Kanten des Graphen G bzw. des vom Algorithmus konstruierten Minimalgerüsts zu tun haben).

- Für jeden Knoten $v \in V$ werden $\pi(v)$, $\tau(v)$ und $\delta(v)$ hierfür wie folgt definiert:

- Wenn v der Wurzelknoten eines Baums ist, ist $\pi(v) = \perp$.
  Andernfalls ist $\pi(v)$ der Vorgänger von v im entsprechenden Baum.

- Durch Verfolgen der Vorgängerkette findet man zu einem Knoten v den zugehörigen Wurzelknoten $\tau(v) = \begin{cases} v, & \text{falls } \pi(v) = \perp \\ \tau(\pi(v)) & \text{sonst} \end{cases}$

- Für einen Knoten v ist $\delta(v)$ die Tiefe des Teilbaums mit Wurzel v.

- Initialisierung der Hilfsdatenstruktur:
  Zu Beginn des Algorithmus wird $\pi(v) = \perp$ und $\delta(v) = 0$ für jeden Knoten $v \in V$ gesetzt, d. h. jeder Knoten stellt einen einelementigen Baum dar.

- Immer, wenn in Schritt 3 des Algorithmus eine Kante $\{u, v\}$ zur Menge F hinzugefügt wird, werden die Bäume, zu denen u und v gehören, zu einem einzigen Baum zusammengefasst, indem einer der beiden (im Zweifelsfall der mit der geringeren Tiefe) in den anderen eingehängt wird (auf diese Weise bleibt die Tiefe der Bäume möglichst klein), das heißt:
  - Seien $u' = \tau(u)$ und $v' = \tau(v)$ die Wurzelknoten der beiden Bäume.
  - Wenn $\delta(u') < \delta(v')$: Setze $\pi(u') = v'$,
    d. h. hänge den Baum mit Wurzel u' in den Baum mit Wurzel v' ein,
    dessen Tiefe dabei unverändert bleibt.
  - Wenn $\delta(u') > \delta(v')$: Setze $\pi(v') = u'$ (also gerade umgekehrt).
  - Wenn $\delta(u') = \delta(v')$: Setze $\pi(u') = v'$ und $\delta(v') = \delta(v') + 1$,
    d. h. hänge den Baum mit Wurzel u' in den Baum mit Wurzel v' ein,
    dessen Tiefe dabei um eins größer wird.

- Dann gilt offensichtlich zu jedem Zeitpunkt:
  Zwei Knoten u und v sind genau dann durch Kanten in F verbunden, wenn $\tau(u) = \tau(v)$ gilt, d. h. wenn u und v zum selben Baum gehören.

Optimierungsmöglichkeit

- Jedesmal, wenn mittels $\tau(v)$ der zu einem Knoten v gehörende Wurzelknoten r bestimmt wurde, wird für alle dabei durchlaufenen Knoten u ihr Vorgänger $\pi(u)$ auf r gesetzt, d. h. diese Knoten werden direkt in den Wurzelknoten r eingehängt.

- Damit werden anschließende Berechnungen von $\tau(u)$ für diese Knoten u. U. erheblich beschleunigt.

Laufzeit der Operationen auf der Hilfsdatenstruktur

Behauptung:

- Ein Baum der Hilfsdatenstruktur mit Tiefe k enthält mindestens 2^k Knoten.

Beweis durch vollständige Induktion nach k:

- Induktionsanfang $k = 0$:
  - Ein Baum mit Tiefe $k = 0$ enthält genau $1 = 2^0 = 2^k$ Knoten.
- Induktionsschritt $k \to k + 1$:
  - Ein Baum mit Tiefe $k + 1$ ist ursprünglich aus zwei Bäumen mit Tiefe k entstanden (Fall $\delta(u') = \delta(v')$ beim Zusammenfassen zweier Bäume), zu dem später eventuell weitere Bäume mit Tiefe $\leq k$ hinzugefügt wurden (Fälle $\delta(u') < \delta(v')$ und $\delta(u') > \delta(v')$).
  - Deshalb enthält ein Baum mit Tiefe $k + 1$ nach Induktionsvoraussetzung mindestens $2 \cdot 2^k = 2^{k+1}$ Knoten.

Daraus folgt:

- Ein Baum mit N Knoten hat höchstens Tiefe $\log_2 N$.
- Die Länge der Vorgängerkette eines Knotens ist $O(\log |V|)$.
- Sowohl der Test $\tau(u) = \tau(v)$ zur Überprüfung, ob u und v bereits durch Kanten in F verbunden sind, als auch das Zusammenfassen zweier Bäume hat Laufzeit $O(\log |V|)$.

Laufzeit des Algorithmus

- Sortieren der Kantenmenge E mit einem geeigneten Verfahren:
  $O(|E| \log |E|) = O\left(|E| \log |V|^2\right) = O(|E| \, 2 \log |V|) = O(|E| \log |V|)$
  (Für jeden Graphen gilt: $|E| \leq |V|^2$.)
- Operationen auf der Hilfsdatenstruktur:
  $|E|$-mal Test $\tau(u) = \tau(v)$ und höchstens $|E|$-mal Zusammenfassen zweier Bäume, insgesamt also $O(|E| \log |V|)$.
- Gesamtlaufzeit somit: $O(|E| \log |V|)$

5.5.8 Algorithmus von Prim

Gegeben

- Ungerichteter, gewichteter Graph $G = (V, E, \rho)$

Algorithmus

1. Für jeden Knoten $v \in V$:
   1. Füge v mit Priorität $\delta(v) = \infty$ in eine Minimum-Vorrangwarteschlange Q ein.
   2. Setze $\pi(v) = \perp$.
2. Solange Q nicht leer ist:
   1. Entnimm einen Knoten u mit minimaler Priorität.
   2. Für jeden Nachfolger v von u:

      Wenn $v \in Q$ und $\rho(u, v) < \delta(v)$:
      1. Erniedrige die Priorität $\delta(v)$ auf $\rho(u, v)$.
      2. Setze $\pi(v) = u$.

Laufzeit

- Operationen auf der Vorrangwarteschlange:
  - $|V|$-mal Einfügen eines Knotens
  - $|V|$-mal Test, ob die Warteschlange leer ist
  - $|V|$-mal Entnehmen eines Knotens mit minimaler Priorität
  - $|E|$-mal Test, ob ein Knoten enthalten ist
    (Der Rumpf der inneren Schleife wird insgesamt $|E|$-mal ausgeführt.)
  - Maximal $|E|$-mal Erniedrigen der Priorität eines Knotens

  Insgesamt: $O(|V| + |E|)$

- Laufzeit jeder solchen Operation: $O(\log |V|)$,
  da die Warteschlange maximal $|V|$ Einträge enthält.

- Gesamtlaufzeit somit: $O((|V| + |E|) \log |V|)$

- Für zusammenhängende Graphen: $O(|E| \log |V|)$, da für sie $|E| \geq |V| - 1$ gilt.

Ergebnis

- Nach Ausführung des Algorithmus ist die Kantenmenge $F = \{\{\pi(v), v\} \mid \pi(v) \neq \perp\}$ ein Minimalgerüst von G (das aus $|V| - |F|$ Bäumen besteht).

- Wenn G zusammenhängend ist (d. h. wenn $|V| - |F| = 1$ ist), ist F folglich ein minimaler Spannbaum von G.

- Andernfalls kann der Algorithmus dies bei Bedarf feststellen:
  Wenn mehr als einmal ein Knoten u mit $\delta(u) = \infty$ entnommen wird, ist der Graph nicht zusammenhängend.

Beispiel

- Siehe § 5.5.4.

Korrektheit

- Bezeichnungen (jeweils am Anfang eines Schleifendurchlaufs):
  - Q = Menge aller Knoten, die sich noch in der Vorrangwarteschlange befinden
  - $P = V \setminus Q$ = Menge aller Knoten, die bereits entnommen wurden
  - $PQ = \{\{p, q\} \in E \mid p \in P, q \in Q\} =$
    Menge aller Kanten, die Knoten aus P mit Knoten aus Q verbinden
  - $F = \{\{\pi(v), v\} \mid v \in P, \pi(v) \neq \perp\}$

- Schleifeninvariante für Schritt 2 des Algorithmus:
  1. Für alle Knoten $q \in Q$ gilt entweder $\pi(q) = \perp$ und $\delta(q) = \infty$
     oder $\pi(q) \in P, \{\pi(q), q\} \in E$ und $\delta(q) = \rho(\pi(q), q)$
  2. Für alle Kanten $\{p, q\} \in PQ$ gilt: $\delta(q) \leq \rho(p, q)$
  3. Die Kantenmenge F ist Teilmenge eines Minimalgerüsts von G,
     d. h. es gibt ein Minimalgerüst M mit $F \subseteq M$
  4. Für alle Knoten $p_1, p_2 \in P$ gilt:
     Wenn es in G einen Weg von p_1 nach p_2 gibt,
     dann gibt es auch in F einen Weg von p_1 nach p_2

- Initialisierung: Vor Beginn der Schleife gilt:
  1. $\pi(q) = \perp$ und $\delta(q) = \infty$ für alle $q \in Q$
  2. $P = \emptyset$ und somit auch $PQ = \emptyset$
  3. $F = \emptyset$ und somit $F \subseteq M$ für jedes Minimalgerüst M
  4. $P = \emptyset$

- Aufrechterhaltung: Wenn die Invariante am Anfang eines Schleifendurchlaufs gilt, gilt am Ende dieses Durchlaufs:
  1. Wenn $\pi(v)$ und $\delta(v)$ in Schritt 2.2 verändert werden, bleibt Teil 1 der Invariante jeweils erhalten.
  2. Weil der Knoten u zur Menge P hinzukommt, muss Teil 2 der Invariante zusätzlich für alle Kanten $\{u, v\}$ mit $v \in Q$ gelten.
     Dies wird durch Schritt 2.2 sichergestellt.

     Für alle anderen Kanten $\{p, q\} \in PQ$ gilt Teil 2 der Invariante weiterhin, weil $\delta(q)$ während eines Schleifendurchlaufs nur kleiner, aber niemals größer werden kann.
  3. Wenn $\pi(u) = \perp$ ist, ist nichts zu zeigen, weil die Menge F in diesem Fall unverändert bleibt.

     Wenn $\pi(u) \neq \perp$ ist, kommt die Kante $e = \{\pi(u), u\}$ zur Menge F hinzu, und es gilt:

- Aufgrund von Teil 3 der Invariante ist die ursprüngliche Menge F Teilmenge eines Minimalgerüsts M von G.
- Da M ein Gerüst von G ist, gibt es in M einen Weg von $\pi(u) \in P$ nach $u \in Q$, der mindestens eine Kante $e' = \{ p, q \} \in PQ$ enthalten muss.
  (Beachte: u gehört noch zur ursprünglichen Menge Q.)
- Es gilt: $\rho(e) = \rho(\pi(u), u) \underset{\text{(a)}}{=} \delta(u) \underset{\text{(b)}}{\leq} \delta(q) \underset{\text{(c)}}{\leq} \rho(p, q) = \rho(e')$, denn:
  a) Teil 1 der Invariante
  b) Es wird ein Knoten u mit minimaler Priorität $\delta(u)$ entnommen
  c) Teil 2 der Invariante
- Deshalb gilt für $M' := M \setminus \{ e' \} \cup \{ e \}$, das aufgrund des Lemmas in § 5.5.1 ebenfalls ein Gerüst von G ist:
  $\rho(M') = \rho(M) - \rho(e') + \rho(e) \leq \rho(M)$.
- Da M bereits ein Minimalgerüst von G ist, muss somit auch M' ein Minimalgerüst sein.
- Wegen $q \notin P$ gilt $e' = \{ p, q \} \notin F$ (F enthält nur Kanten $\{ \pi(v), v \}$ mit $v \in P$ und $\pi(v) \in P$) und somit $F' = F \cup \{ e \} = F \setminus \{ e' \} \cup \{ e \} \subseteq M \setminus \{ e' \} \cup \{ e \} = M'$, d. h. die neue Menge F' ist wiederum Teilmenge eines Minimalgerüsts von G.

4. Weil der Knoten u zur Menge P hinzukommt, muss Teil 4 der Invariante zusätzlich für alle Wege von irgendeinem Knoten $p \in P$ zum Knoten u gelten.

   Wenn $\delta(u) = \infty$ ist:

   - Weil ein Knoten u mit minimaler Priorität $\delta(u)$ entnommen wird, ist $\delta(q) = \infty$ für alle Knoten $q \in Q$.
   - Deshalb gibt es wegen Teil 2 der Invariante zu diesem Zeitpunkt keine Kanten in PQ und deshalb keinen Weg von $p \in P$ nach $u \in Q$.

   Wenn $\delta(u) < \infty$ (und deshalb nach Teil 1 der Invariante $\pi(u) \in P$) ist:

   - Wenn es in G einen Weg von $p \in P$ nach u gibt, gibt es auch einen Weg von p nach $\pi(u)$ (entweder weil der Weg von p nach u bereits über $\pi(u)$ führt oder weil er sonst über die Kante $\{u, \pi(u)\}$ nach $\pi(u)$ fortgesetzt werden kann).
   - Wegen $\pi(u) \in P$ gibt es aufgrund von Teil 4 der Invariante auch einen Weg in F von p nach $\pi(u)$ und somit auch einen Weg von p nach u in $F' = F \cup \{\{\pi(u), u\}\}$ (entweder weil der Weg von p nach $\pi(u)$ bereits über u führt oder weil er sonst über die Kante $\{\pi(u), u\}$ nach u fortgesetzt werden kann).

   Für alle übrigen Knoten $p_1, p_2 \in P$ gilt Teil 4 der Invariante weiterhin, weil die Kantenmenge F während eines Schleifendurchlaufs nur größer, aber niemals kleiner werden kann.

- Terminierung: Nach Beendigung der Schleife gilt:

  3. Die endgültige Kantenmenge F ist Teilmenge eines Minimalgerüsts von G.
  4. Für alle Knoten $p_1, p_2 \in P = V$ gilt:
     Wenn es in G einen Weg von p_1 nach p_2 gibt,
     dann gibt es auch in F einen Weg von p_1 nach p_2.

     Damit ist F ein vollständiges Minimalgerüst von G.

Modifizierter Algorithmus mit vorgegebenem Startknoten

Gegeben

- Ungerichteter, gewichteter Graph $G = (V, E, \rho)$
- Startknoten $s \in V$

Algorithmus

1 Für jeden Knoten $v \in V \setminus \{ s \}$:
   1 Füge v mit Priorität $\delta(v) = \infty$ in eine Minimum-Vorrangwarteschlange Q ein.
   2 Setze $\pi(v) = \perp$.
2 Setze $\pi(s) = \perp$.
3 Setze $u = s$.
4 Solange Q nicht leer ist:
   1 Für jeden Nachfolger v von u:

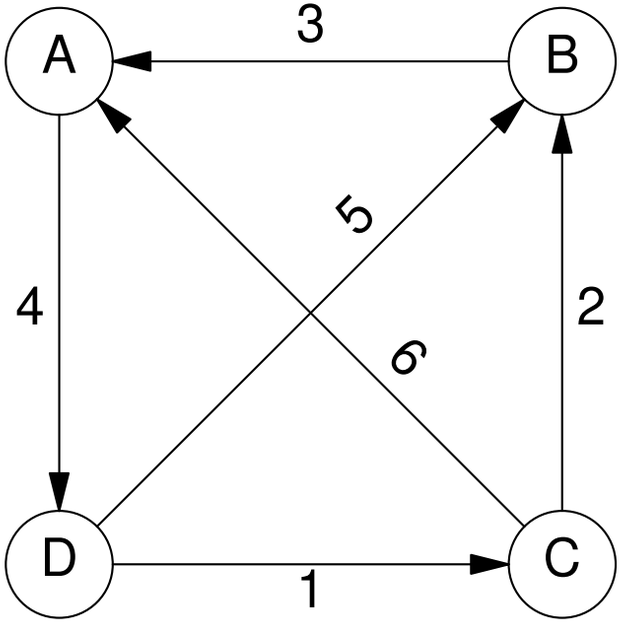
     Wenn $v \in Q$ und $\rho(u, v) < \delta(v)$:
      1 Erniedrige die Priorität $\delta(v)$ auf $\rho(u, v)$.
      2 Setze $\pi(v) = u$.
   2 Entnimm einen Knoten u mit minimaler Priorität.

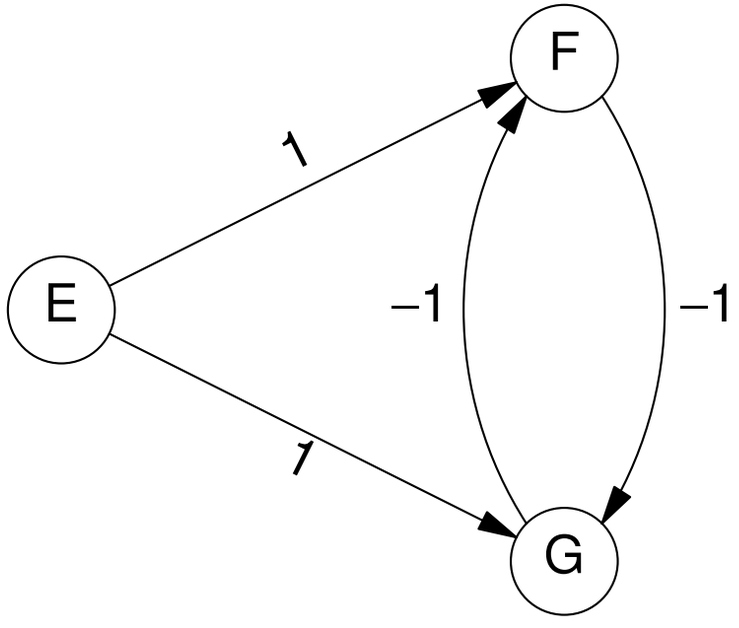
5.6 Kürzeste Wege (shortest paths)

5.6.1 Definitionen

- Gegeben sei ein (gerichteter oder ungerichteter) gewichteter Graph $G = (V, E, \rho)$.
- Das *Gewicht* eines Wegs $w = w_0, \ldots, w_n$ ist die Summe $\rho(w) = \sum_{i=1}^{n} \rho(w_{i-1}, w_i)$ der Gewichte aller Kanten auf dem Weg.
- Wenn es einen Weg von einem Knoten u zu einem Knoten v gibt, ist ein *kürzester Weg* von u nach v ein Weg von u nach v mit minimalem Gewicht.
  (Weil ein Weg keinen Knoten mehrfach enthalten kann, kann es nur endlich viele verschiedene Wege von u nach v geben, sodass das Minimum wohldefiniert ist. Wenn es unendlich viele verschiedene Wege geben könnte, gäbe es u. U. keinen mit minimalem Gewicht.)
- In diesem Fall ist der *Abstand* $\delta(u, v)$ von u nach v das Gewicht eines kürzesten Wegs von u nach v. Andernfalls ist $\delta(u, v) = \infty$.
- Anmerkung: Da Kanten und Wege ein „Gewicht“ besitzen, müsste man eigentlich von „leichtesten“ statt von „kürzesten“ Wegen sprechen. Auch der Begriff des „Abstands“ passt eigentlich nicht zu dem des „Gewichts“. Trotzdem werden die Bezeichnungen üblicherweise so verwendet.

Beispiel





- In diesem Graphen gilt zum Beispiel:

  $\delta(A, B) = 7$

  $\delta(C, A) = 5$

  $\delta(A, E) = \delta(E, A) = \infty$

  $\delta(E, F) = \delta(E, G) = 0$

  $\delta(F, G) = \delta(G, F) = -1$

5.6.2 Problemstellungen

- Ein Knotenpaar:
  Finde einen kürzesten Weg von einem bestimmten Startknoten s zu einem bestimmten Zielknoten t.

- Fester Startknoten:
  Finde jeweils einen kürzesten Weg von einem bestimmten Startknoten s zu allen Knoten des Graphen.

- Fester Zielknoten:
  Finde jeweils einen kürzesten Weg von allen Knoten des Graphen zu einem bestimmten Zielknoten t.

- Alle Knotenpaare:
  Finde jeweils einen kürzesten Weg von jedem Knoten des Graphen zu jedem anderen.

Anmerkungen

- Das Problem mit festem Zielknoten lässt sich auf das Problem mit festem Startknoten zurückführen, indem man den transponierten Graphen betrachtet.

- Es sind keine Algorithmen bekannt, die das Problem für ein einzelnes Knotenpaar effizienter lösen als das Problem mit festem Startknoten.

- Das Problem für alle Knotenpaare lässt sich prinzipiell auf das Problem mit festem Startknoten zurückführen, indem man jeden Knoten des Graphen einmal als Startknoten wählt.
  Hier gibt es jedoch spezielle Algorithmen, die das Problem für alle Knotenpaare effizienter lösen.

- Aufgrund dieser Beobachtungen werden im folgenden nur Algorithmen für das Problem mit festem Startknoten sowie solche für alle Knotenpaare betrachtet.

5.6.3 Hilfsmittel für das Problem mit festem Startknoten

- Für jeden Knoten $v \in V$ wird das Gewicht $\delta(v)$ des kürzesten bis jetzt gefundenen Wegs vom Startknoten s zu v sowie der Vorgänger $\pi(v)$ von v auf diesem Weg gespeichert.

- Solange noch kein Weg von s nach v gefunden wurde, ist $\delta(v) = \infty$ und $\pi(v) = \perp$.

- Initialisierung:
  - Für alle Knoten $v \in V$: Setze $\delta(v) = \infty$ und $\pi(v) = \perp$.
  - Setze dann $\delta(s) = 0$.

- Verwerten einer Kante von u nach v:

  Wenn $\delta(u) + \rho(u, v) < \delta(v)$ ist,
  d. h. wenn der Weg von s nach v über u kürzer als der kürzeste bis jetzt gefundene Weg von s nach v ist:

  Setze $\delta(v) = \delta(u) + \rho(u, v)$ und $\pi(v) = u$,
  d. h. speichere den Weg über u als neuen kürzesten Weg nach v.

5.6.4 Algorithmus von Bellman und Ford

Gegeben

- Gewichteter Graph $G = (V, E, \rho)$, Startknoten $s \in V$
  (Das heißt, der Algorithmus löst das Problem mit festem Startknoten.)

- Einschränkung:

  Der Graph darf keine *negativen Zyklen* enthalten (d. h. Zyklen $w_0, \ldots, w_n$ mit $w_0 = w_n$, deren Gewicht $\sum_{i=1}^{n} \rho(w_{i-1}, w_i)$ negativ ist), die vom Startknoten s aus erreichbar sind.

- Die Einhaltung dieser Einschränkung wird vom Algorithmus selbst überprüft.
  Wenn sie verletzt ist, bricht der Algorithmus ab.

- Abgesehen davon, sind Kanten mit negativem Gewicht erlaubt.

Algorithmus

1 Für alle Knoten $v \in V$: Setze $\delta(v) = \infty$ und $\pi(v) = \perp$.
Setze dann $\delta(s) = 0$.

2 Wiederhole $(|V| - 1)$-mal:
Für jede Kante $(u, v) \in E$:
Verwerte die Kante (vgl. § 5.6.3).

3 Für jede Kante $(u, v) \in E$:
Wenn $\delta(u) + \rho(u, v) < \delta(v)$:
Abbruch, weil der Graph einen von s aus erreichbaren negativen Zyklus enthält.

Ergebnis

- Wenn der Graph einen von s aus erreichbaren negativen Zyklus enthält, bricht der Algorithmus in Schritt 3 ab.

- Andernfalls gilt nach Ausführung des Algorithmus für jeden Knoten $v \in V$:
  - $\delta(v) = \delta(s, v)$
  - Wenn $\pi(v) \neq \perp$ ist, ist $\pi(v)$ der Vorgänger von v auf einem kürzesten Weg von s nach v.

Laufzeit

- Initialisierung (Schritt 1): $O(|V|)$
- Eigentlicher Algorithmus (Schritt 2): $O(|V| \cdot |E|)$
- Überprüfung auf negative Zyklen (Schritt 3): $O(|E|)$
- Gesamtlaufzeit also: $O(|V| \cdot |E|)$

Beispiel

- Siehe § 5.6.1.

Korrektheit

Definition

- Eine *Route* (oder ein *Multiweg*) der Länge n von einem Knoten u zu einem Knoten v ist eine Folge von Knoten $w_0, \ldots, w_n$ mit $u = w_0$, Kanten von w_{i-1} nach w_i für $i = 1, \ldots, n$ und $w_n = v$.
  (Im Gegensatz zu einem Weg, darf ein Knoten in einer Route auch mehrmals vorkommen, d. h. eine Route darf auch Zyklen enthalten.)

Lemma 1

- Für jeden Knoten $v \in V$ gilt zu jedem Zeitpunkt entweder $\delta(v) = \infty$ oder $\delta(v) = \rho(w) = \sum_{i=1}^{n} \rho(w_{i-1}, w_i)$ für eine Route $w = w_0, \ldots, w_n$ vom Startknoten s zum Knoten v.

Beweis durch vollständige Induktion nach der Anzahl der bis jetzt verwerteten Kanten

- Am Anfang gilt für $v = s$: $\delta(v) = 0 = \rho(w)$ für die leere Route w von s nach v, und $\delta(v) = \infty$ für alle anderen Knoten $v \neq s$.

- Nach der Verwertung einer Kante (u, v) in Schritt 2 ist $\delta(v)$ entweder unverändert – dann folgt die Behauptung für diesen Knoten v direkt aus der Induktionsvoraussetzung –, oder es gilt: $\delta(v) = \delta(u) + \rho(u, v) = \rho(w) + \rho(u, v) = \rho(w')$ für eine Route $w = s, \ldots, u$ – die es nach Induktionsvoraussetzung gibt – und die Route $w' = s, \ldots, u, v$.
  Für alle anderen Knoten v bleibt $\delta(v)$ unverändert.

Lemma 2

- Nach dem n-ten Durchlauf der Schleife in Schritt 2 gilt für jede Route $w = s, \ldots, v$ der Länge n vom Startknoten s zu irgendeinem Knoten v: $\delta(v) \leq \rho(w)$.

Beweis durch vollständige Induktion nach n

- Induktionsanfang $n = 0$
  - Die einzige Route w der Länge 0 vom Startknoten s zu irgendeinem Knoten v ist die leere Route von s nach s, für die gilt: $\rho(w) = 0$.
  - Also gilt für $v = s$: $\delta(v) = 0 \leq \rho(w)$.
- Induktionsschritt $n \to n + 1$
  - Sei $w = s, \ldots, u, v$ eine Route der Länge $n + 1$ und $w' = s, \ldots, u$.
  - Nach dem n-ten Durchlauf der Schleife gilt nach Induktionsvoraussetzung: $\delta(u) \leq \rho(w')$.
  - Im $(n + 1)$-ten Durchlauf der Schleife wird u. a. die Kante (u, v) verwertet, woraus $\delta(v) \leq \delta(u) + \rho(u, v) \leq \rho(w') + \rho(u, v) = \rho(w)$ folgt.
  - Durch die Verwertung weiterer Kanten im selben oder späteren Durchläufen wird der Wert von $\delta(v)$ eventuell noch kleiner, aber niemals größer.

Lemma 3

- Wenn G keinen vom Startknoten s aus erreichbaren negativen Zyklus enthält, gilt für jeden Knoten $v \in V$ zu jedem Zeitpunkt: $\delta(v) \geq \delta(s, v)$.

Beweis

- Für $\delta(v) = \infty$ gilt trivialerweise $\delta(v) \geq \delta(s, v)$, unabhängig vom Wert von $\delta(s, v)$.
- Für $\delta(v) < \infty$ gilt nach Lemma 1: $\delta(v) = \rho(w)$ für eine Route w von s nach v.
- Da es keinen von s aus erreichbaren negativen Zyklus gibt, kann diese Route keinen negativen Zyklus enthalten.
- Für den Weg w' von s nach v, der entsteht, wenn man aus der Route w eventuell vorhandene Zyklen entfernt, gilt nach Definition von $\delta(s, v)$: $\rho(w') \geq \delta(s, v)$.
- Da das Gewicht der eventuell entfernten Zyklen nicht negativ ist, gilt: $\delta(v) = \rho(w) \geq \rho(w') \geq \delta(s, v)$.

Satz 1

- Wenn G keinen vom Startknoten s aus erreichbaren negativen Zyklus enthält, gilt nach Ausführung von Schritt 2 für jeden Knoten $v \in V$: $\delta(v) = \delta(s, v)$.

Beweis:

- Nach Lemma 3 gilt für jeden Knoten $v \in V$: $\delta(v) \geq \delta(s, v)$.
  Deshalb genügt es zu zeigen, dass für jeden Knoten $v \in V$ gilt: $\delta(v) \leq \delta(s, v)$.

- Wenn v von s aus nicht erreichbar ist, ist $\delta(s, v) = \infty$ und somit trivialerweise $\delta(v) \leq \delta(s, v)$.

- Wenn v von s aus erreichbar ist, gibt es einen Weg und damit auch einen kürzesten Weg $w = s, \ldots, v$ von s nach v, d. h. $\delta(s, v) = \rho(w)$.

- Da ein Weg keinen Knoten mehrmals enthalten kann, ist die Länge dieses Wegs höchstens $|V| - 1$.

- Deshalb gilt nach dem $(|V| - 1)$-ten Durchlauf der Schleife gemäß Lemma 2: $\delta(v) \leq \rho(w) = \delta(s, v)$.

Korollar

- Wenn G keinen vom Startknoten s aus erreichbaren negativen Zyklus enthält, bricht der Algorithmus in Schritt 3 nicht ab.

Beweis

- Annahme: Es gibt eine Kante $(u, v) \in E$, für die in Schritt 3 die Abbruchbedingung $\delta(u) + \rho(u, v) < \delta(v)$ erfüllt ist.
- Nach Satz 1 gilt zu diesem Zeitpunkt: $\delta(u) = \delta(s, u)$ und $\delta(v) = \delta(s, v)$.
  Also muss für die Kante (u, v) die Bedingung $\delta(s, u) + \rho(u, v) < \delta(s, v)$ erfüllt sein, woraus insbesondere $\delta(s, u) < \infty$ folgt.
- Deshalb gibt es einen kürzesten Weg $w = s, \ldots, u$ von s nach u mit $\rho(w) = \delta(s, u)$.
- Dann ist $w' = s, \ldots, u, v$ ein Weg von s nach v mit Gewicht $\delta(s, u) + \rho(u, v) < \delta(s, v)$.
- Widerspruch zur Definition von $\delta(s, v)$!
- Also gibt es keine Kante, für die in Schritt 3 die Abbruchbedingung erfüllt ist.

Satz 2

- Wenn G einen von s aus erreichbaren negativen Zyklus $w = w_0, \ldots, w_n$ enthält, bricht der Algorithmus in Schritt 3 ab.

Beweis

- Da $w_0, \ldots, w_n$ ein Zyklus ist, gilt $w_0 = w_n$ und deshalb $S = \sum_{i=1}^{n} \delta(w_i) = \sum_{i=1}^{n} \delta(w_{i-1})$.
- Da der Zyklus von s aus erreichbar ist, gibt es für jeden seiner Knoten w_i einen Weg w von s nach w_i mit Länge $\leq |V| - 1$.
  Deshalb gilt nach Lemma 2: $\delta(w_i) \leq \rho(w) < \infty$ für jeden Knoten w_i des Zyklus und somit auch $S = \sum_{i=1}^{n} \delta(w_i) < \infty$.
- Annahme: In Schritt 3 gilt für keine Kante $(u, v) \in E$ die Abbruchbedingung $\delta(u) + \rho(u, v) < \delta(v)$, d. h. für jede Kante $(u, v) \in E$ gilt: $\delta(v) \leq \delta(u) + \rho(u, v)$.
- Insbesondere gilt $\delta(w_i) \leq \delta(w_{i-1}) + \rho(w_{i-1}, w_i)$ für jede Kante (w_{i-1}, w_i) des Zyklus und somit: $S = \sum_{i=1}^{n} \delta(w_i) \leq \sum_{i=1}^{n} (\delta(w_{i-1}) + \rho(w_{i-1}, w_i)) = \sum_{i=1}^{n} \delta(w_{i-1}) + \sum_{i=1}^{n} \rho(w_{i-1}, w_i) = S + \rho(w)$.
- Daraus folgt wegen $S < \infty$: $0 \leq \rho(w)$. Widerspruch, da w ein negativer Zyklus ist.

5.6.5 Algorithmus von Dijkstra

Gegeben

- Gewichteter Graph $G = (V, E, \rho)$, Startknoten $s \in V$
  (Das heißt, der Algorithmus löst das Problem mit festem Startknoten.)

- Einschränkung:
  Der Graph darf keine Kanten mit negativem Gewicht enthalten.

- Die Einhaltung dieser Einschränkung wird vom Algorithmus *nicht* überprüft.
  Wenn sie verletzt ist, ist das Ergebnis des Algorithmus undefiniert.

Algorithmus

1 Für alle Knoten $v \in V$: Setze $\delta(v) = \infty$ und $\pi(v) = \perp$.
Setze dann $\delta(s) = 0$.

2 Für alle Knoten $v \in V$:
Füge v mit Priorität $\delta(v)$ in eine Minimum-Vorrangwarteschlange ein.

3 Solange die Warteschlange nicht leer ist:

   1 Entnimm einen Knoten u mit minimaler Priorität.

   2 Für jeden Nachfolger v von u, der sich noch in der Warteschlange befindet:

      1 Verwerte die Kante (u, v) (vgl. § 5.6.3).

      2 Wenn $\delta(v)$ dadurch erniedrigt wurde:

         Erniedrige die Priorität von v in der Warteschlange entsprechend.

Ergebnis

- Wenn der Graph keine Kanten mit negativem Gewicht enthält, gilt nach Ausführung des Algorithmus für alle Knoten $v \in V$:
  - $\delta(v) = \delta(s, v)$
  - Wenn $\pi(v) \neq \perp$ ist, ist $\pi(v)$ der Vorgänger von v auf einem kürzesten Weg von s nach v.
- Andernfalls ist das Ergebnis des Algorithmus undefiniert.

Beispiel

- Siehe § 5.6.1.

Laufzeit

- Initialisierung (Schritt 1): $O(|V|)$
- Operationen auf der Vorrangwarteschlange:
  - $|V|$-mal Einfügen eines Knotens
  - $|V|$-mal Test, ob die Warteschlange leer ist
  - $|V|$-mal Entnehmen eines Knotens mit minimaler Priorität
  - $|E|$-mal Test, ob ein Knoten enthalten ist
  - Maximal $|E|$-mal Erniedrigen der Priorität eines Knotens
    (Der Rumpf der inneren Schleife wird höchstens $|E|$-mal ausgeführt.)

  Insgesamt $O(|V| + |E|)$
- Laufzeit jeder solchen Operation: $O(\log |V|)$,
  da die Warteschlange maximal $|V|$ Einträge enthält.
- Gesamtlaufzeit somit: $O((|V| + |E|) \log |V|)$

Korrektheit

1. Sei Q die Menge der Knoten, die sich zu einem bestimmten Zeitpunkt noch in der Warteschlange befinden, und $P = V \setminus Q$.

2. Am Ende jedes Durchlaufs durch die Schleife in Schritt 3 gilt: $\delta(v) \leq \delta(u) + \rho(u, v)$ für jeden Nachfolger v von u, der sich noch in der Warteschlange befindet.
(Dies folgt unmittelbar aus der Definition der Operation „Verwerten“.)

3. Zum Zeitpunkt der Entnahme eines Knotens u aus der Warteschlange gilt: $\delta(u) = \delta(s, u)$.

4. Da $\delta(u)$ für Knoten $u \in P$ nicht mehr verändert wird und $\delta(v)$ für Knoten $v \in Q$ später höchstens noch verkleinert wird, gelten diese Aussagen auch zu jedem späteren Zeitpunkt.

5. Damit gilt insbesondere nach Ausführung des Algorithmus: $\delta(u) = \delta(s, u)$ für jeden Knoten $u \in V$.

Beweis von Aussage 3 durch „Schleifeninduktion“:

- Zum Zeitpunkt der Entnahme von $u = s$ gilt: $\delta(s) = 0 = \delta(s, s)$
- Zum Zeitpunkt der Entnahme eines anderen Knotens u gilt:

- Da es keine negativen Kantengewichte und somit auch keine negativen Zyklen gibt, gilt nach Lemma 3 in § 5.6.4: $\delta(u) \geq \delta(s, u)$.
- Wenn u von s aus nicht erreichbar ist, gilt somit: $\delta(u) \geq \delta(s, u) = \infty \Rightarrow \delta(u) = \delta(s, u)$.
- Wenn u von s aus erreichbar ist, gibt es einen kürzesten Weg $w = s, \ldots, u$ von s nach u mit Gewicht $\rho(w) = \delta(s, u)$.
- Sei p der letzte Knoten auf diesem Weg, für den $p \in P$ gilt (eventuell ist $p = s$), und q der nächste Knoten auf diesem Weg (eventuell ist $q = u$), d. h. q ist ein Nachfolger von p, und es gilt $q \in Q$.
- Somit gilt: $\delta(s, u) = \rho(w) = \rho(s, \ldots, p, q, \ldots, u) = \rho(s, \ldots, p) + \rho(p, q) + \rho(q, \ldots, u) \underset{\text{(a)}}{\geq}$

  $\rho(s, \ldots, p) + \rho(p, q) \underset{\text{(b)}}{\geq} \delta(s, p) + \rho(p, q) \underset{\text{(c)}}{=} \delta(p) + \rho(p, q) \underset{\text{(d)}}{\geq} \delta(q) \underset{\text{(e)}}{\geq} \delta(u)$, denn:

  a) $\rho(q, \ldots, u) \geq 0$, da es keine negativen Kantengewichte gibt.

  b) $\rho(s, \ldots, p) \geq \delta(s, p)$ für jeden Weg von s nach p

  c) Induktionsvoraussetzung

  d) Aussage 2

  e) u ist der Knoten mit minimaler Priorität, d. h. $\delta(u) \leq \delta(q)$ für alle $q \in Q$.
- Damit gilt einerseits $\delta(u) \geq \delta(s, u)$ und andererseits $\delta(u) \leq \delta(s, u)$ und somit $\delta(u) = \delta(s, u)$.

5.6.6 Algorithmus von Floyd und Warshall

Gegeben

- Gewichteter Graph $G = (V, E, \rho)$
  (Das heißt, der Algorithmus löst das Problem für alle Knotenpaare.)

- Einschränkung: Der Graph darf keine negativen Zyklen enthalten.

- Die Einhaltung dieser Einschränkung wird vom Algorithmus *nicht* überprüft.
  Wenn sie verletzt ist, ist das Ergebnis des Algorithmus undefiniert.

- Abgesehen davon, sind Kanten mit negativem Gewicht erlaubt.

Definitionen

- Zur Vereinfachung der Notation sei die Knotenmenge des Graphen $V = \{1, \ldots, m\}$.

- Für $k = 0, \ldots, m$ sei ein Weg von u nach v *via* k ein Weg $w_0, \ldots, w_n$ mit $w_0 = u$, $w_n = v$ und $w_1, \ldots, w_{n-1} \leq k$, d. h. alle *Zwischenknoten* (sofern es welche gibt) gehören zur Menge $\{1, \ldots, k\}$ (die für $k = 0$ leer ist).

- Für $k = 0, \ldots, m$ und $u, v \in V$ sei
  - $\Delta_k(u, v)$ entweder das Gewicht eines kürzesten Wegs von u nach v via k oder ∞ (falls es keinen solchen Weg gibt)
  - $\Pi_k(u, v)$ entweder der Vorgänger von v auf diesem Weg oder $\perp$.

Rekursionsgleichungen

- Für $k = 0$ gilt:

$\Delta_0(u, v) =$	$\Pi_0(u, v) =$	wenn
0	$\perp$	$u = v$
$\rho(u, v)$	u	$u \neq v$ und $(u, v) \in E$ bzw. $\{u, v\} \in E$
∞	$\perp$	sonst

- Für $k = 1, \ldots, m$ gilt:

$\Delta_k(u, v) =$	$\Pi_k(u, v) =$	wenn
$\Delta_{k-1}(u, v)$	$\Pi_{k-1}(u, v)$	$\Delta_{k-1}(u, v) \leq \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)$
$\Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)$	$\Pi_{k-1}(k, v)$	$\Delta_{k-1}(u, v) > \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)$

Begründung

Mit den Bezeichnungen $d = \Delta_k(u, v)$, $d_1 = \Delta_{k-1}(u, v)$ und $d_2 = \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)$ gelten folgende Aussagen:

1. $d = d_1$ oder $d = d_2$

   Beweis:

   - Wenn es keinen Weg von u nach v via k gibt, ist $d = \infty$.
     In diesem Fall gibt es auch keinen Weg von u nach v via $k - 1$, d. h. es ist auch $d_1 = \infty$ und somit $d = d_1$.
   - Wenn es einen kürzesten Weg von u nach v via k (mit Gewicht d) gibt und dieser den Knoten k *nicht* enthält, stimmt er mit einem kürzesten Weg von u nach v via $k - 1$ (mit Gewicht d_1) überein, d. h. es gilt wiederum $d = d_1$.
   - Andernfalls besteht dieser Weg (mit Gewicht d) aus einem (eventuell leeren) kürzesten Teilweg von u nach k via $k - 1$ (mit Gewicht $\Delta_{k-1}(u, k)$) und einem (eventuell leeren) kürzesten Teilweg von k nach v via $k - 1$ (mit Gewicht $\Delta_{k-1}(k, v)$), d. h. es gilt $d = d_2$.
     (Beachte: Jeder Teilweg eines kürzesten Wegs ist ebenfalls ein kürzester Weg, sofern es keine negativen Zyklen gibt.)

2. $d = \min(d_1, d_2)$

   Anmerkung:

   - Diese Aussage folgt *nicht* unmittelbar aus der vorigen Aussage 1!
   - Wenn ein Graph einen negativen Zyklus enthält, kann $d = d_1$ gelten, obwohl $d_1 > d_2$ ist.
   - Deshalb verwendet der folgende Beweis an einer entscheidenden Stelle die Voraussetzung, dass der Graph *keinen* negativen Zyklus enthält.

   Beweis:

   - Wenn $d_1 = d_2$ ist (insbesondere auch, wenn $d_1 = d_2 = \infty$ ist), folgt die Behauptung unmittelbar aus der vorigen Aussage 1.
   - Wenn $d_1 < d_2$ ist (woraus insbesondere $d_1 < \infty$ folgt), dann gibt es einen kürzesten Weg von u nach v via $k-1$ mit Gewicht d_1, der auch ein Weg von u nach v via k ist.
     Daraus folgt $d \leq d_1$, und zusammen mit $d_1 < d_2$ und Aussage 1:
     $d = d_1 = \min(d_1, d_2)$.

- Wenn $d_2 < d_1$ ist (woraus insbesondere $d_2 < \infty$ folgt), dann gibt es einen kürzesten Weg von u nach k via $k-1$ mit Gewicht $\Delta_{k-1}(u, k)$ sowie einen kürzesten Weg von k nach v via $k-1$ mit Gewicht $\Delta_{k-1}(k, v)$.
  - Wenn man diese Wege zusammensetzt, entsteht eine *Route* von u nach v via k mit Gewicht d_2.
  - Annahme: Diese Route $u, \ldots, k, \ldots, v$ ist kein Weg, d. h. sie enthält (mindestens) einen Zyklus $w, \ldots, w$.
  - Da die Teilrouten $u, \ldots, k$ und $k, \ldots, v$ Wege sind, d. h. keinen Knoten mehrfach enthalten, muss der Zyklus den Knoten k enthalten, d. h. die Route muss $u, \ldots, w, \ldots, k, \ldots, w, \ldots, v$ lauten.
  - Wenn man den Zyklus (bzw. die Zyklen) entfernt, entsteht somit ein Weg von u nach v via $k-1$ mit Gewicht $\leq d_2$, da das Gewicht der Zyklen nicht negativ ist.
  - Zusammen mit $d_2 < d_1$ ist dies ein Widerspruch dazu, dass d_1 das Gewicht eines kürzesten Wegs von u nach v via $k-1$ ist.
  - Also muss die o. g. Route immer ein *Weg* von u nach v via k mit Gewicht d_2 sein.
  - Daraus folgt $d \leq d_2$, und zusammen mit $d_2 < d_1$ und Aussage 1: $d = d_2 = \min(d_1, d_2)$.

Praktische Berechnung

- Zur Berechnung des Werts $\Delta_k(u, v)$ in Zeile u und Spalte v der Matrix Δ_k werden neben dem korrespondierenden Wert $\Delta_{k-1}(u, v)$ der „vorigen“ Matrix Δ_{k-1} noch die Werte $\Delta_{k-1}(u, k)$ und $\Delta_{k-1}(k, v)$ benötigt, die sich in Spalte bzw. Zeile k dieser Matrix befinden.

- Diese Werte in Spalte und Zeile k bleiben beim Übergang von Δ_{k-1} zu Δ_k unverändert, denn es gilt:

  - Für $v = k$ (d. h. Spalte k):
    $$\begin{aligned}\Delta_k(u, v) &= \min(\Delta_{k-1}(u, v), \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)) = \\ &= \min(\Delta_{k-1}(u, k), \Delta_{k-1}(u, k) + \Delta_{k-1}(k, k)) = \\ &= \min(\Delta_{k-1}(u, k), \Delta_{k-1}(u, k) + 0) = \Delta_{k-1}(u, k) = \Delta_{k-1}(u, v)\end{aligned}$$

  - Für $u = k$ (d. h. Zeile k):
    $$\begin{aligned}\Delta_k(u, v) &= \min(\Delta_{k-1}(u, v), \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)) = \\ &= \min(\Delta_{k-1}(k, v), \Delta_{k-1}(k, k) + \Delta_{k-1}(k, v)) = \\ &= \min(\Delta_{k-1}(k, v), 0 + \Delta_{k-1}(k, v)) = \Delta_{k-1}(k, v) = \Delta_{k-1}(u, v)\end{aligned}$$

  (Beachte: $\Delta_{k-1}(k, k) = 0$, weil ein kürzester Weg von k nach k immer Gewicht 0 besitzt.)

- Somit können die Werte der Matrix Δ_{k-1} (und analog Π_{k-1}) jeweils gefahrlos durch die korrespondierenden Werte der Matrix Δ_k (und analog Π_k) überschrieben werden.

Algorithmus

1 Für $u = 1, \ldots, m$:

  1 Für $v = 1, \ldots, m$: Setze $\Delta(u, v) = \infty$ und $\Pi(u, v) = \perp$.

  2 Für jeden Nachfolger v von u: Setze $\Delta(u, v) = \rho(u, v)$ und $\Pi(u, v) = u$.

  3 Setze $\Delta(u, u) = 0$ und $\Pi(u, u) = \perp$.

2 Für $k = 1, \ldots, m$:

  Für $u = 1, \ldots, m$ und $v = 1, \ldots, m$:

  Wenn $\Delta(u, v) > \Delta(u, k) + \Delta(k, v)$:

  Setze $\Delta(u, v) = \Delta(u, k) + \Delta(k, v)$ und $\Pi(u, v) = \Pi(k, v)$.

Ergebnis

- Nach Ausführung des Algorithmus gilt für alle Knoten $u, v \in V$:
  - $\Delta(u, v) = \Delta_m(u, v)$ = Gewicht eines kürzesten Wegs von u nach v via m = $\delta(u, v)$ = Gewicht eines beliebigen kürzesten Wegs von u nach v, falls ein solcher Weg existiert, andernfalls ∞
  - $\Pi(u, v) = \Pi_m(u, v)$ = Vorgänger von v auf diesem Weg bzw. $\perp$

Laufzeit

- Offensichtlich $O(m^3) = O(|V|^3)$

Beispiel

- Siehe § 5.6.1.

5.6.7 Laufzeitvergleich

Fester Startknoten

- Das Problem mit festem Startknoten kann mit folgenden Algorithmen gelöst werden, sofern die jeweilige Voraussetzung erfüllt ist:
  - Dijkstra, wenn es keine negativen Kantengewichte gibt
  - Bellman-Ford,
    wenn es keinen vom Startknoten aus erreichbaren negativen Zyklus gibt
  - Floyd-Warshall, wenn es überhaupt keinen negativen Zyklus gibt
- Die nachfolgende Tabelle zeigt die jeweiligen Laufzeiten im Vergleich

Algorithmus	*Laufzeit*		
	allgemein	wenn $\|E\| \approx \|V\|$	wenn $\|E\| \approx \|V\|^2$
Dijkstra	$O((\|V\| + \|E\|) \log \|V\|)$	$O(\|V\| \log \|V\|)$	$O\left(\|V\|^2 \log \|V\|\right)$
Bellman-Ford	$O(\|V\| \cdot \|E\|)$	$O\left(\|V\|^2\right)$	$O\left(\|V\|^3\right)$
Floyd-Warshall	$O\left(\|V\|^3\right)$	$O\left(\|V\|^3\right)$	$O\left(\|V\|^3\right)$

Alle Knotenpaare

- Das Problem für alle Knotenpaare kann mit folgenden Algorithmen gelöst werden, sofern die jeweilige Voraussetzung erfüllt ist:
  - $|V|$-mal Dijkstra, wenn es keine negativen Kantengewichte gibt
  - $|V|$-mal Bellman-Ford, wenn es keinen negativen Zyklus gibt
  - 1-mal Floyd-Warshall, wenn es keinen negativen Zyklus gibt
- Die nachfolgende Tabelle zeigt die jeweiligen Gesamtlaufzeiten im Vergleich

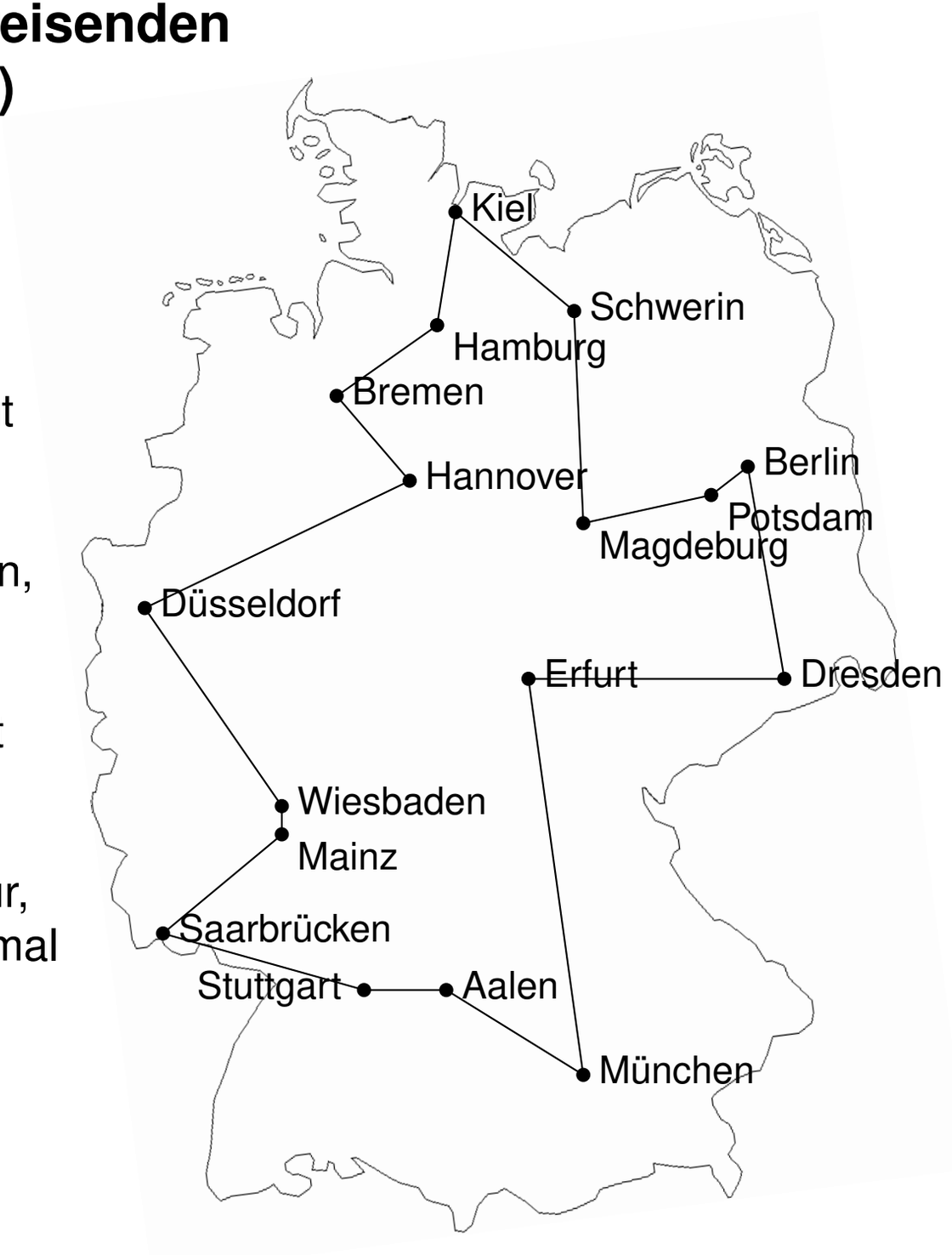
Algorithmus	*Laufzeit*		
	allgemein	wenn $\|E\| \approx \|V\|$	wenn $\|E\| \approx \|V\|^2$
Dijkstra	$O(\|V\|\,(\|V\| + \|E\|)\log\|V\|)$	$O\left(\|V\|^2 \log\|V\|\right)$	$O\left(\|V\|^3 \log\|V\|\right)$
Bellman-Ford	$O\left(\|V\|^2 \cdot \|E\|\right)$	$O\left(\|V\|^3\right)$	$O\left(\|V\|^4\right)$
Floyd-Warshall	$O\left(\|V\|^3\right)$	$O\left(\|V\|^3\right)$	$O\left(\|V\|^3\right)$

5.7 Das Problem des Handlungsreisenden (Traveling Salesman Problem)

5.7.1 Problemstellung

Anschauliche Formulierung

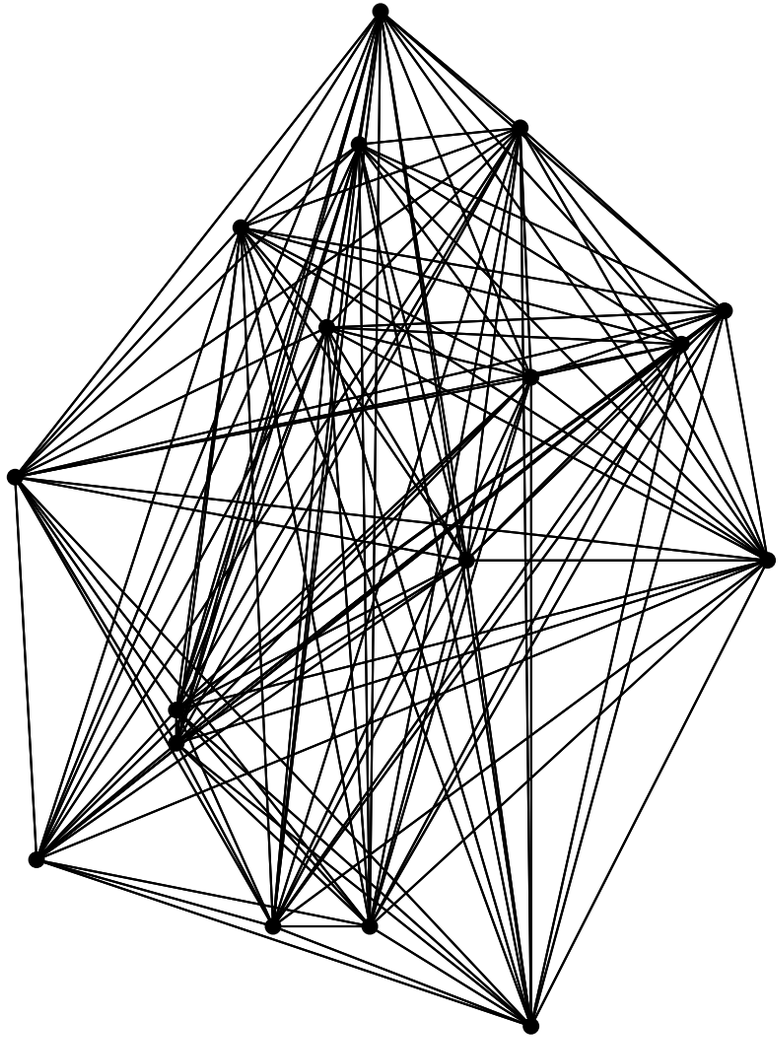
- Ein Vertreter einer Aalener Firma betreut Kunden in vielen Städten Deutschlands.
- Um ihnen ein neues Produkt vorzustellen, muss er alle Kunden besuchen.
- Die Distanz zwischen je zwei Städten ist bekannt (z. B. Entfernungstabelle).
- Wie findet der Vertreter die kürzeste Tour, um von Aalen aus jede Stadt genau einmal zu besuchen und am Schluss wieder in Aalen zu sein (d. h. eine Rundtour)?



Mathematische Formulierung

Gegeben

- Ungerichteter, gewichteter Graph $G = (V, E, \rho)$
- mit $|V| = N$ Knoten,
- vollständiger Kantenmenge $E = \{\{u, v\} \mid u, v \in V\}$
- und Gewichtsfunktion $\rho: E \to \mathbb{R}^+$



Gesucht

- Zyklus $v_1, \ldots, v_N, v_1$, der jeden Knoten des Graphen genau einmal enthält,
- mit minimalem Gewicht,
- d. h. $\sum_{i=1}^{N-1} \rho(v_i, v_{i+1}) + \rho(v_N, v_1)$ ist minimal

Anwendungsmöglichkeiten

- Routenplanung
- Schaltungsentwurf/-verdrahtung
- Umrüsten von Produktionsmaschinen
- Usw.
- Musteranwendung und Benchmark für Approximationsverfahren

5.7.2 Exakte Lösungsverfahren

- Naiv: Alle Kombinationen ausprobieren: $O(N!)$
- Verbesserung durch sog. dynamisches Programmieren: $O\left(N^2 \cdot 2^N\right)$
- Weitere Verbesserungen z. B. durch Ausnutzen geometrischer Eigenschaften
- Trotzdem: Problem ist *NP-schwierig*,
  d. h. nach heutigem Wissensstand nicht effizient lösbar

5.7.3 Approximationsverfahren

Grundprinzip

- Finde effiziente Algorithmen,
  die eine möglichst gute *Näherungslösung* des Problems liefern

- Betrachte zwei Parameter zum Vergleich von Verfahren:
  - *Laufzeit* in Abhängigkeit von der Knotenzahl N
  - *Qualitätsfaktor* $q = \dfrac{\text{max. Gewicht der gefundenen Tour}}{\text{Gewicht einer optimalen Tour}}$

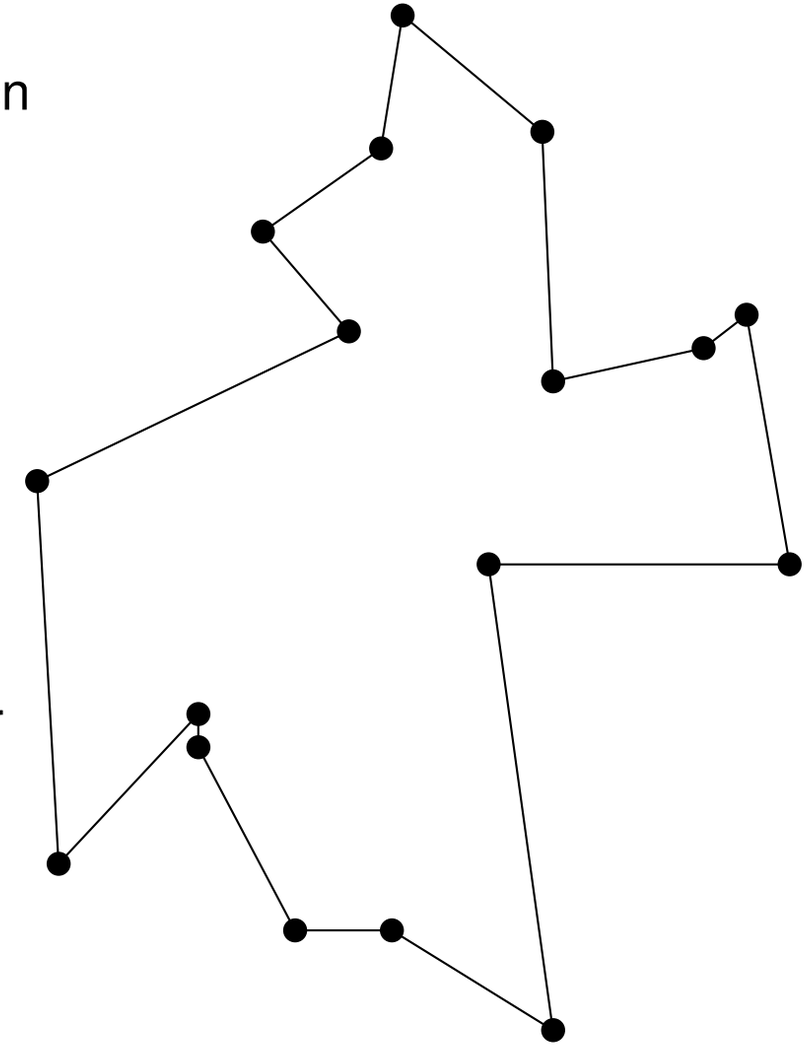
    ($q = 1$ bei exakten Verfahren)

- In der Regel wird die *Dreiecksungleichung* ausgenutzt:

$$\rho(u, w) \leq \rho(u, v) + \rho(v, w) \qquad \text{für alle } u, v, w \in V$$

Das Nearest-Neighbour-Verfahren

- Beginne mit dem Startknoten v_1.
- Wähle als zweiten Knoten v_2 den nächstgelegenen Nachbarn von v_1 ($N - 1$ Kandidaten).
- Wähle als dritten Knoten v_3 den nächstgelegenen Nachbarn von v_2 aus der Menge der verbleibenden Knoten ($N - 2$ Kandidaten).
- Usw.
- Wähle als vorletzten Knoten v_{N-1} den nächstgelegenen Nachbarn von v_{N-2} aus der Menge der verbleibenden Knoten (2 Kandidaten).
- Füge den letzten verbleibenden Knoten v_N hinzu (1 Kandidat).
- Typisches Beispiel eines Nächstbest-Algorithmus



Laufzeit

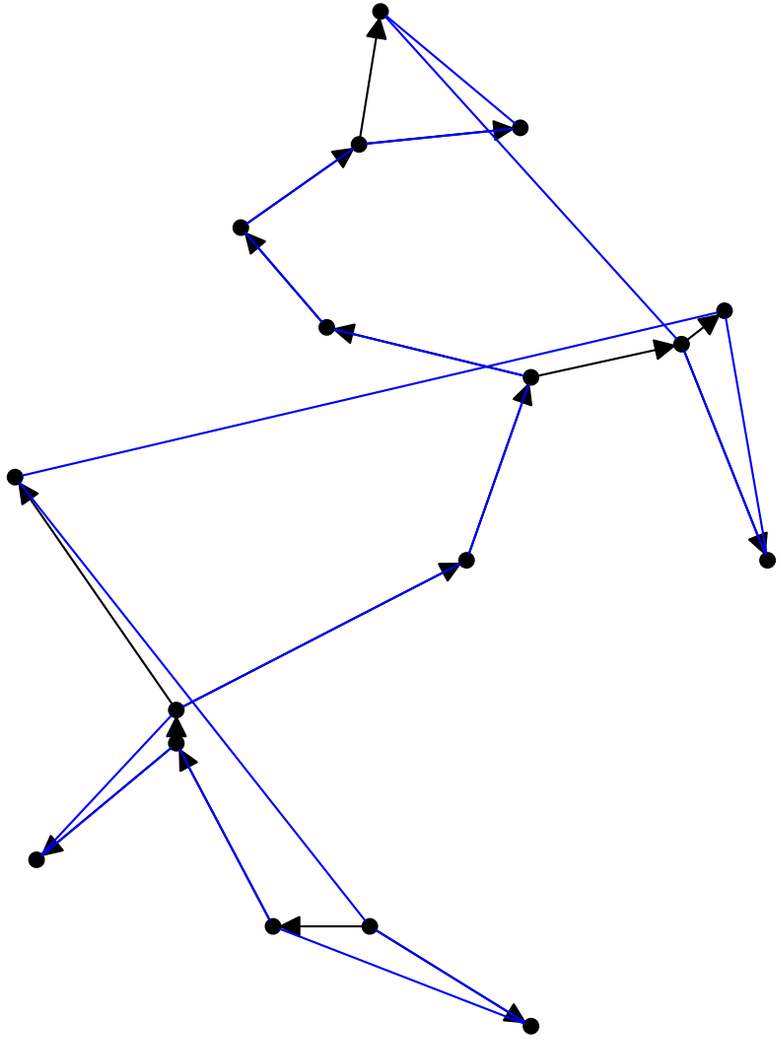
- $(N-1)+(N-2)+\ldots+2+1=\frac{(N-1)\,N}{2}=\mathrm{O}\left(N^2\right)$

Qualitätsfaktor

- Allgemein: $q(N)=\frac{\lfloor \log_2 N+1 \rfloor}{2}$
- Konkret z. B.: $q(10)=2.5$ $q(100)=4$ $q(1000)=5.5$ $q\left(10^6\right)=10.5$

Das Spannbaum-Verfahren

- Konstruiere einen minimalen Spannbaum des Graphen (schwarze Pfeile).
- Konstruiere eine Tour, die jede Kante des Spannbaums genau zweimal durchläuft, einmal „abwärts“ und einmal „aufwärts“.
- Überspringe alle Knoten, die bereits besucht wurden.



Laufzeit

- Konstruktion des minimalen Spannbaums mit dem Algorithmus von Prim: $O(|E| \log |V|) = O\left(N^2 \log N\right)$
- Durchlaufen des Spannbaums: $O(N)$
- Insgesamt also: $O\left(N^2 \log N\right)$

Qualitätsfaktor

- $L_{\text{Tour}} \leq 2\, L_{\text{MST}}$ (Dreiecksungleichung)
- $L_{\text{MST}} \leq L_{\text{ST}} \leq L_{\text{Opt}}$ (jede Tour impliziert einen Spannbaum)
- Somit: $L_{\text{Tour}} \leq 2\, L_{\text{Opt}}$
- Also: $q = 2$