Algorithmische Graphentheorie

Kapitel 6: Kreise und Wege

Babeş-Bolyai Universität, Fachbereich Informatik, Klausenburg

Wiederholung: Algorithmus von Dijkstra

- Es sei $G = (V, A)$ ein gerichteter Graph mit nichtnegativen Kantengewichten und $s \in V$.
- Bei diesem Algorithmus bauen wir den sogenannten *Vorgängergraphen* $G_\pi = (V_\pi, A_\pi)$ auf, der wie folgt definiert ist:

$$V_\pi := \{v \in V : \pi[v] \neq \texttt{nil}\} \cup \{s\}$$

und

$$A_\pi := \{(\pi[v], v) \in A : v \in V_\pi \ \wedge \ v \neq s\},$$

wobei die Funktion π zu einem Knoten den entsprechenden Vorgängerknoten speichert und `nil` für einen leeren Eintrag steht.

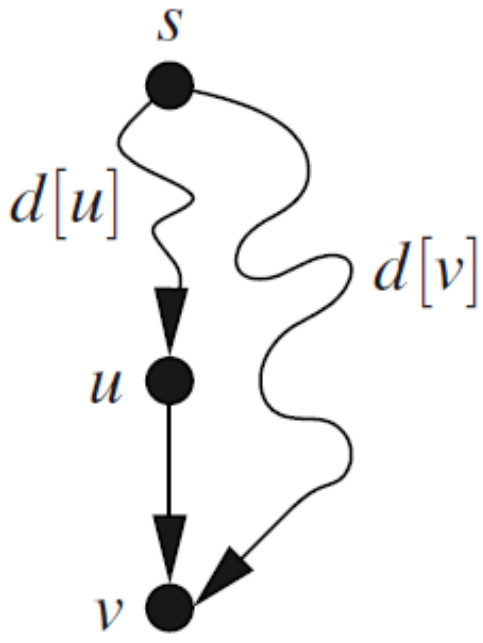
- G_π ist ein Baum kürzester Wege bezüglich s.

INIT(G, s)

```
1 for all v ∈ V do
2     d[v] := +∞
3     π[v] := nil
4 d[s] := 0
```

TEST(u, v)

```
1 if d[v] > d[u] + c(u, v) then
2     d[v] := d[u] + c(u, v)
3     π[v] := u
```



Testschritt TEST(u, v)

Initialisierung und Test: Algorithmus, der für eine gerichtete Kante $(u, v) \in A$ prüft, ob diese benutzt werden kann, um einen kürzeren Weg von s nach v zu finden als der bisher bekannte mit Länge $d[v]$. π baut den Vorgängergraphen G_π auf.

DIJKSTRA(G, c, s)

Input: Ein gerichteter Graph $G = (V, A)$ in Adjazenzlistendarstellung, eine nichtnegative Gewichtsfunktion $c\colon A \to \mathbb{R}_+$ und ein Knoten $s \in V$ mit allen Knoten aus s erreichbar

Output: Für alle $v \in V$ die Distanz $d[v] = \mathrm{dist}_c(s, v)$ sowie ein Baum G_π kürzester Wege von s aus

```
1 INIT(G, s)
2 PERM := ∅        { PERM ist die Menge der »permanent markierten« Knoten }
3 while PERM ≠ V do
4    Wähle u ∈ Q := V \ PERM mit minimalem Schlüsselwert d[u].
5    PERM := PERM ∪ {u}
6    for all v ∈ Adj[u] \ PERM do
7       TEST(u, v)
8 return d[] und G_π     { G_π wird durch die Vorgängerzeiger π »aufgespannt« }
```

Man beachte: Gewichte dürfen nicht negativ sein!

Algorithmus von Dijkstra: Korrektheit

Satz. *Bei Abbruch des Algorithmus von Dijkstra gilt* $d[v] = \mathrm{dist}_c(s, v)$ *für alle* $v \in V$. *Der Graph* G_π *ist ein Baum kürzester Wege bezüglich* s.

Beweis (1/7)

Lemma 1. *Es sei $s \in V$ ein ausgezeichneter Knoten. Dann gilt*

$$\mathrm{dist}_c(s,v) \leq \mathrm{dist}_c(s,u) + c(u,v)$$

für alle $(u,v) \in A$.

Beweis. Ein kürzester sv-Weg ist höchstens so lang wie ein *beliebiger* sv-Weg. Insbesondere gilt dies auch für alle kürzesten su-Wege, die um die gerichtete Kante (u,v) zu einem sv-Weg verlängert werden.

Ein kürzester su-Weg der Länge $\mathrm{dist}_c(s,u)$ lässt sich durch die Kante (u,v) zu einem sv-Weg der Länge $\mathrm{dist}_c(s,u) + c(u,v)$ ergänzen.

Beweis (2/7)

Lemma 2. *Ein Algorithmus initialisiere mithilfe von INIT und führe dann eine beliebige Anzahl von Testschritten TEST aus. Dann gilt während des Algorithmus:* $d[v] \geq \text{dist}_c(s, v)$ *für jedes* $v \in V$.

Beweis. Wir beweisen die Behauptung durch Induktion nach der Anzahl der Testschritte. Vor dem ersten Testschritt gilt $V_\pi = \{s\}, A_\pi = \emptyset, d[s] = 0$ und $d[v] = +\infty$ für $v \neq s$. Die Aussagen sind somit alle offenbar richtig.

Es werde nun ein weiterer Schritt $\text{TEST}(u, v)$ ausgeführt. Wenn $d[v] \leq d[u] + c(u, v)$ war, so wird nichts an G_π oder den Werten π und d geändert. Somit ist in diesem Fall nichts zu zeigen.

Beweis (3/7)

Wir können also $d[v] > d[u] + c(u, v)$ annehmen.

Es gilt nach Induktionsvoraussetzung

$$d[u] + c(u, v) \geq \text{dist}_c(s, u) + c(u, v) \geq \text{dist}_c(s, v),$$

wobei die letzte Ungleichung aus Lemma 1 folgt. Damit ist Lemma 2 bewiesen.

Beweis (4/7)

Beweis des Korrektheitssatzes (Folie 5):

- Nach Lemma 2 gilt $d[v] \geq \text{dist}_c(s, v)$ für alle $v \in V$.
- Wir zeigen durch Induktion nach der Anzahl der Durchläufe der **while**-Schleife, dass $d[u] = \text{dist}_c(s, u)$ für alle $u \in \text{PERM}$ gilt.
- Vor dem ersten Durchlauf ist dies offenbar richtig, da $\text{PERM} = \{s\}$ und $d[s] = \text{dist}_c(s, s) = 0$.
- Für den Induktionsschritt genügt es zu beweisen, dass für den Knoten u, der in Schritt 4 aus $Q := V \setminus \text{PERM}$ gewählt wird, $d[u] \leq \text{dist}_c(s, u)$ gilt.

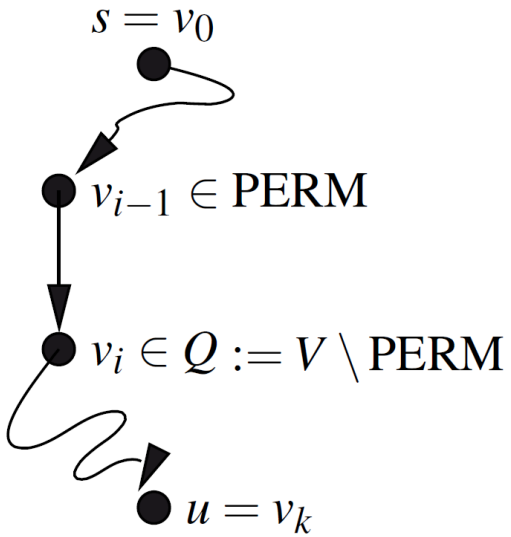
Beweis (5/7)

- Wir nehmen an, dass $d[u] > \text{dist}_c(s, u)$. (Wir führen also einen Widerspruchsbeweis.)
- Es sei $P = v_0 \dots v_k$ (wobei $s = v_0$ und $v_k = u$) ein kürzester su-Weg der Länge $\text{dist}_c(s, u)$.
- Es sei $i \in \{0, \dots, k\}$ minimal mit $v_i \in Q$. Dann gilt $i \geq 1$, da $v_0 = s \in \text{PERM}$ bereits vor der ersten Iteration galt und niemals Knoten aus PERM aus Q entfernt werden.

Beweis (6/7)

- Es ist $v_{i-1} \in$ PERM nach Wahl von i.
- Dann wurde v_{i-1} in einer früheren Iteration als Minimum aus Q entfernt.
- Nach Induktionsvoraussetzung war dann $d[v_{i-1}] \leq \text{dist}_c(s, v_{i-1})$.
- Nach den Testschritten in Schritt 7 gilt zu Ende dieser Iteration dann

$$d[v_i] = d[v_{i-1}] + c(v_{i-1}, v_i) \overset{I.V.}{\leq} \text{dist}_c(s, v_{i-1}) + c(v_{i-1}, v_i) \overset{P\ min.}{=} \text{dist}_c(s, v_i)$$



Beweis (7/7)

Also ist

$$d[v_i] \leq \text{dist}_c(s, v_i) \leq \text{dist}_c(s, u) \overset{\text{Folie } 10}{<} d[u]$$

im Widerspruch zur Wahl von u als Knoten in Q mit minimalem Schlüsselwert (s. Schritt 4). Damit ist der Beweis abgeschlossen.

Beweis (7/7)

Also ist

$$d[v_i] \leq \text{dist}_c(s, v_i) \leq \text{dist}_c(s, u) \overset{\text{Folie } 10}{<} d[u]$$

im Widerspruch zur Wahl von u als Knoten in Q mit minimalem Schlüsselwert (s. Schritt 4). Damit ist der Beweis abgeschlossen.

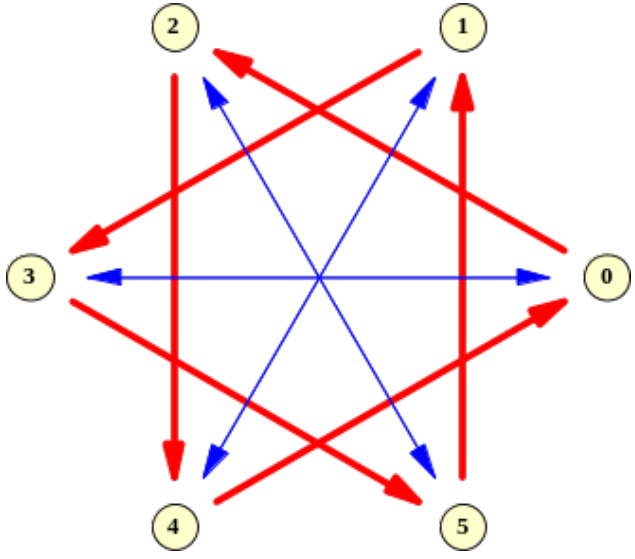
Illustration des Dijkstra-Algorithmus für euklidische Abstände:
`https://tinyurl.com/ybp87nro`

Anwendungen

- Routenplaner sind ein wichtiges Beispiel, bei dem dieser Algorithmus eingesetzt werden kann. Der Graph repräsentiert hier das Verkehrswegenetz, das verschiedene Punkte miteinander verbindet. Gesucht ist die kürzeste Route zwischen zwei Punkten.
- Einige topologische Indizes, etwa der J-Index von Balaban, benötigen gewichtete Distanzen zwischen den Atomen eines Moleküls. Die Gewichtung ist in diesen Fällen die Bindungsordnung.
- Dijkstras Algorithmus wird auch im Internet als Routing-Algorithmus in verschiedenen Protokollen eingesetzt.
- Auch bei der Lösung des Münzproblems, eines zahlentheoretischen Problems, das auf den ersten Blick nichts mit Graphen zu tun hat, kann der Dijkstra-Algorithmus eingesetzt werden. Wir geben als Beispiel eine Variante hiervon:

Das McNugget-Problem (1/2)

- Gefragt ist, welche Anzahlen Chicken McNuggets man kaufen kann, wenn man sie aus den klassischen Verpackungsgrößen zu 6, 9 und 20 Stück zusammenstellt.
- Wir modellieren das Problem mit einem Graphen mit sechs Knoten (Anzahl Restklassen bei Division mit 6). Jene gerichteten Kanten, die zur Zahl 9 gehören, sind in der Darstellung blau gezeichnet und teilen die Knoten in drei Paare ein, die roten Kanten gehören zur Zahl 20 und bilden zwei Dreiergruppen.



Das McNugget-Problem (2/2)

Mit dem Dijkstra-Algorithmus bestimmen wir die kürzesten Wege vom Knoten 0 aus (von mehreren Möglichkeiten ist eine angegeben):

- $0 \to 3 \to 5 \to 1$ (Länge $9 + 20 + 20 = 49$)
- $0 \to 2$ (Länge 20)
- $0 \to 3$ (Länge 9)
- $0 \to 2 \to 4$ (Länge $20 + 20 = 40$)
- $0 \to 3 \to 5$ (Länge $9 + 20 = 29$)

Der Graph liefert auch zu jeder darstellbaren Zahl eine mögliche Darstellung. Will man etwa 47 Chicken McNuggets, so sucht man wegen

$$47 \equiv 5 \bmod 6$$

nach dem kürzesten Weg von 0 nach 5. Dieser besteht aus einer blauen und einer roten Kante, sodass man eine Packung zu 9 und eine zu 20 Chicken McNuggets kaufen sollte. Für die restlichen 18 nimmt man drei 6er Packungen.

Bellman-Ford-Algorithmus

- Wie der Dijkstra-Algorithmus berechnet der Bellman-Ford-Algorithmus die kürzesten Wegen von einem Startknoten aus zu allen anderen Knoten in einem gewichteten, gerichteten Graphen.
  (Vorsicht: *Zwischenschritte* bei Bellman-Ford betrachten Kantenzüge, nicht nur Wege.)
- Der Algorithmus wurde 1955 von Alfonso Shimbel veröffentlicht, ist aber in der Literatur nach Richard Bellman and Lester Ford Jr. benannt, die ihn 1958 bzw. 1956 publizierten.
- Geeignet auch für negative Gewichte. (Nicht jedoch negative Kreise! Kann diese aber aufspüren.)
- Der Dijkstra-Algorithmus ist greedy und sucht nach dem kürzesten Weg. Der Bellman-Ford-Algorithmus durchsucht alle Kanten und merkt sich die leichteste.
- Zeitkomplexität: $O(|E| \cdot |V|)$.

Es sei G ein gerichteter Graph mit Kantengewichten $\in \mathbb{R}$, ohne negative Kreise, und ein Knoten $s \in V$. Es sei Distanz(v) das Gewicht eines bereits bestimmten sv-Kantenzuges.

für jedes v in V

    Distanz(v) := ∞, Vorgänger(v) := keiner

Distanz(s) := 0

wiederhole $|V| - 1$ mal

    für jedes (u, v) aus E

        wenn (Distanz(u) + Gewicht(u, v)) $<$ Distanz(v) **dann**

            Distanz(v) := Distanz(u) + Gewicht(u, v)

            Vorgänger(v) := u

für jedes (u, v) aus E

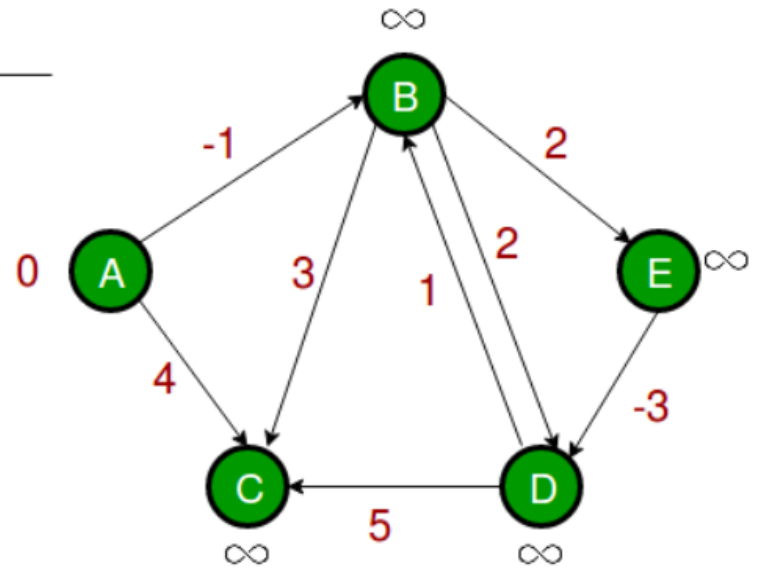
    wenn (Distanz(u) + Gewicht(u, v)) $<$ Distanz(v) dann

        STOPP mit Ausgabe *Es gibt Kreis negativen Gewichtes.*

Ausgabe *Distanz, Vorgänger*

Beispiel (1/3)

A	B	C	D	E
0	∞	∞	∞	∞

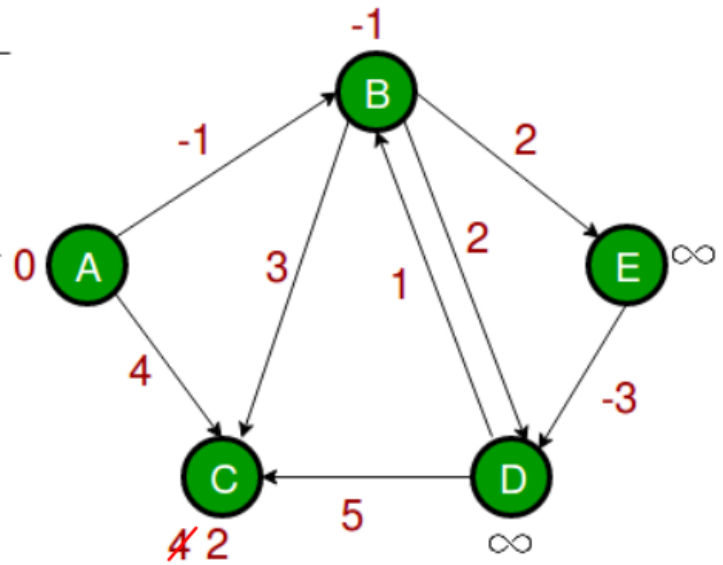


$|V| = 5$, wir werden also $|V| - 1 = 4$ Durchläufe haben. Start in A. Wir durchlaufen Kanten in dieser Reihenfolge:
(B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D)

Wir wenden nun den Bellman-Ford-Algorithmus an und notieren die jeweils aktuellen Abstände in einer Tabelle.

BEISPIEL (2/3)

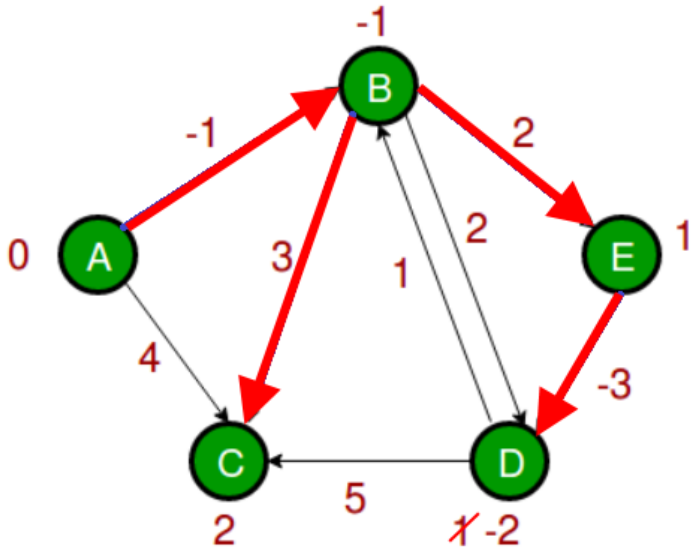
A	B	C	D	E
0	∞	∞	∞	∞
0	-1_A	∞	∞	∞
0	-1_A	4_A	∞	∞
0	-1_A	2_B	∞	∞



Zeile 1 zeigt die ursprünglichen Distanzen.
Zeile 2 zeigt Distanzen nach Betrachtung der Kanten (B, E), (D, B), (B, D), (A, B). Vorgänger in blau.
Zeile 3 zeigt Distanzen nach Betrachtung der Kante (A,C).
Zeile 4 zeigt Dist. nach Betrachtung von (D, C), (B, C), (E, D).
Dieser 1. Durchgang liefert alle kürzesten Kantenzüge der Länge höchstens 1.

BEISPIEL (3/3)

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1_A	2_B	∞	1_B
0	-1_A	2_B	1_B	1_B
0	-1_A	2_B	-2_E	1_B



Wir betrachten jede Kante ein zweites Mal:
(B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D)
Dieser 2. Durchgang liefert alle kürzesten Kantenzüge der Länge ≤ 2. Der Algorithmus betrachtet jede Kante weitere 2 Male, es werden jedoch keine Distanzen verbessert, da diese nach dem 2. Durchgang bereits minimal waren. Erhalten den Baum (in rot) aller kürzesten Wege von A aus. (Jeder kürzeste Kantenzug im Output ist ein Weg.)

Floyd-Warshall-Algorithmus

- Es sei G ein gerichteter Graph.
- Dijkstra und Bellman-Ford berechnen einen kürzesten Weg **von einem Wurzelknoten** zu allen anderen Knoten von G.
- Ziel: Berechnung **aller** kürzesten Wege in G.
- Wende Dijkstra und Bellman-Ford für alle Knotenpaare an:
  - Zeitkomplexität Dijkstra: $O(|V|^3)$.
  - Zeitkomplexität Bellman-Ford: $O(|V|^2 \cdot |E|)$.

Floyd-Warshall-Algorithmus

- Der Floyd-Warshall-Algorithmus berechnet kürzeste (gerichtete) Wege in gewichteten, gerichteten Graphen.
- Negative Gewichte sind zulässig.
- Wir setzen voraus, dass die betrachteten Graphen keinen Kreis negativer Länge enthalten.
- Der Floyd-Warshall-Algorithmus basiert auf dem Prinzip der dynamischen Programmierung: Spalte eine Berechnung in mehrere einfache Schritte mithilfe einer rekursiven Prozedur.
- Rekursive Prozeduren werden auch in der logischen Programmierung intensiv genutzt.

- Dieser Algorithmus wurde 1962 von Robert Floyd in seiner heute anerkannten Form veröffentlicht.
- Er entspricht jedoch im Wesentlichen den Algorithmen, die zuvor 1959 von Bernard Roy und 1962 auch von Stephen Warshall veröffentlicht wurden (um den sogenannten transitiven Abschluss eines Graphen zu finden), und ist eng mit einem 1956 veröffentlichten Algorithmus von Kleene verwandt.
- Die moderne Formulierung des Algorithmus als drei verschachtelte **for**-Schleifen wurde erstmals von Peter Ingerman beschrieben, ebenfalls 1962.

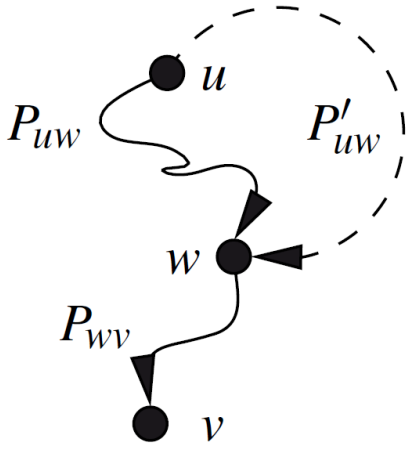
- Die Idee für den Floyd-Warshall-Algorithmus ist eine Rekursion für die Distanzen.
- Es sei $G = (V, A)$ ein Digraph mit $|V| = n$ und c eine Gewichtsfunktion (negative Gewichte zugelassen), sodass G keine Kreise negativen Gewichts enthält.
- Setze $V = \{v_1, \ldots, v_n\}$.
- Definiere für $v_i, v_j \in V$ und $k = 0, 1, \ldots, n$ den Wert $D_k(v_i, v_j)$ als die Länge (bzgl. c) eines kürzesten $v_i v_j$-Weges, dessen Knoten in $\{v_i, v_j, v_1, \ldots, v_k\}$ enthalten sind. Ist $k = 0$, so dürfen nur die Knoten v_i, v_j benutzt werden.

- Die Idee für den Floyd-Warshall-Algorithmus ist eine Rekursion für die Distanzen.
- Es sei $G = (V, A)$ ein Digraph mit $|V| = n$ und c eine Gewichtsfunktion (negative Gewichte zugelassen), sodass G keine Kreise negativen Gewichts enthält.
- Setze $V = \{v_1, \ldots, v_n\}$.
- Definiere für $v_i, v_j \in V$ und $k = 0, 1, \ldots, n$ den Wert $D_k(v_i, v_j)$ als die Länge (bzgl. c) eines kürzesten $v_i v_j$-Weges, dessen Knoten in $\{v_i, v_j, v_1, \ldots, v_k\}$ enthalten sind. Ist $k = 0$, so dürfen nur die Knoten v_i, v_j benutzt werden.
- Falls kein solcher Weg existiert, so sei $D_k(v_i, v_j) := \infty$.
- Die Distanz $\text{dist}_c(v_i, v_j)$ entspricht dann $D_n(v_i, v_j)$ (da wir hier alle Knoten des Graphen verwenden dürfen).
- Da G nach Annahme keine Kreise negativer Länge enthält, können wir in dieser gesamten Betrachtung ohne Beschränkung der Allgemeinheit davon ausgehen, dass all unsere Kantenzüge Wege sind, also keine Kanten- oder Knotenwiederholungen vorkommen.

Wir benötigen das folgende Lemma
(Verallgemeinerung der Bemerkung aus Kap. 3, Folie 9):

Lemma (Teilwege kürzester Wege sind kürzeste Wege). *Es sei P ein kürzester uv-Weg und $w \in V(P)$. Dann ist der Teilweg $P_{uw} \subseteq P$ von u nach w ein kürzester Weg von u nach w. Ferner ist der Teilweg $P_{wv} \subseteq P$ ein kürzester Weg von w nach v.*

Beweis. Ein kürzerer uw-Weg P'_{uw} kann benutzt werden – indem wir P_{uw} durch P'_{uw} ersetzen –, um einen uv-Weg zu erhalten, der kürzer als P ist: Widerspruch. Analoge Argumentation für den Teilweg von w nach v.



Wir zeigen nun, wie man die Werte $D_k(v_i, v_j)$ rekursiv berechnen kann. Für $k = 0$ darf der Weg nur v_i und v_j verwenden, wir haben also

$$(\star) \qquad D_0(v_i, v_j) = \begin{cases} c(v_i, v_j) & \text{falls } (v_i, v_j) \in A, \\ \infty & \text{sonst.} \end{cases}$$

Für $0 < k < n$ sei nun P ein kürzester Weg von v_i nach v_j, dessen Knoten in $\{v_i, v_j, v_1, \ldots, v_{k+1}\}$ enthalten sind.

- Falls $v_{k+1} \notin V(P)$, so ist $c(P) = D_k(v_i, v_j)$.
- Falls $v_{k+1} \in V(P)$, so können wir P in einen $v_i v_{k+1}$-Weg $P_{v_i, v_{k+1}}$ und einen $v_{k+1} v_j$-Weg P_{v_{k+1}, v_j} **aufspalten**.
- Dann gilt $c(P_{v_i, v_{k+1}}) = D_k(v_i, v_{k+1})$ und $c(P_{v_{k+1}, v_j}) = D_k(v_{k+1}, v_j)$, vgl. das Lemma auf Folie 25.

- Falls $v_{k+1} \notin V(P)$, so ist $c(P) = D_k(v_i, v_j)$.
- Falls $v_{k+1} \in V(P)$, so können wir P in einen $v_i v_{k+1}$-Weg $P_{v_i, v_{k+1}}$ und einen $v_{k+1} v_j$-Weg P_{v_{k+1}, v_j} **aufspalten**.
- Dann gilt $c(P_{v_i, v_{k+1}}) = D_k(v_i, v_{k+1})$ und $c(P_{v_{k+1}, v_j}) = D_k(v_{k+1}, v_j)$, vgl. das Lemma auf Folie 25.
- Also gilt

$$(\dagger) \qquad D_{k+1}(v_i, v_j) = \min\{D_k(v_i, v_j), D_k(v_i, v_{k+1}) + D_k(v_{k+1}, v_j)\}.$$

- Dies ist die Kernidee des Floyd-Warshall-Algorithmus.
- Mithilfe der Gleichungen $(\star)$ und $(\dagger)$ lassen sich die Distanzen $\mathrm{dist}_c(u, v)$ in $O(n^3)$ Zeit berechnen: Falls wir alle Werte D_k kennen, so benötigt die Berechnung aller Werte D_{k+1} mittels $(\dagger)$ nur $O(n^2)$ Zeit.

Es folgt der Pseudocode zum Floyd-Warshall-Algorithmus – für ein Beispiel, siehe Kap. 2, Folien 119–120.

FLOYD-WARSHALL(G, c)

Input: Ein gerichteter Graph $G = (V, A)$ in Adjazenzlistendarstellung ohne gerichtete Kreise negativer Länge, eine Gewichtsfunktion $c\colon A \to \mathbb{R}$

Output: Für alle $u, v \in V$ die Distanz $D_n(u, v) = \text{dist}_c(u, v)$

1 **for all** $v_i, v_j \in V$ **do**
2 $D_0(v_i, v_j) := +\infty$
3 **for all** $(v_i, v_j) \in A$ **do**
4 $D_0(v_i, v_j) := c(v_i, v_j)$
5 **for** $k = 0, \ldots, n-1$ **do**
6 **for** $i = 1, \ldots, n$ **do**
7 **for** $j = 1, \ldots, n$ **do**
8 $D_{k+1}(v_i, v_j) := \min \{ D_k(v_i, v_j), D_k(v_i, v_{k+1}) + D_k(v_{k+1}, v_j) \}$
9 **return** $D_n[]$

Damit haben wir folgendes Ergebnis bewiesen:

Satz. *Falls G keinen Kreis negativer Länge enthält, so berechnet der Floyd-Warshall-Algorithmus korrekt die Distanzen zwischen allen Knotenpaaren in $O(n^3)$ Zeit.*

Der Bellman-Ford-Algorithmus für alle Paare braucht $O(n^4)$ Zeit. Der Dijkstra-Algorithmus braucht zwar auch nur $O(n^3)$ Zeit für alle Paare, lässt sich jedoch nur anwenden, wenn die Gewichtsfunktion nicht negativ ist.

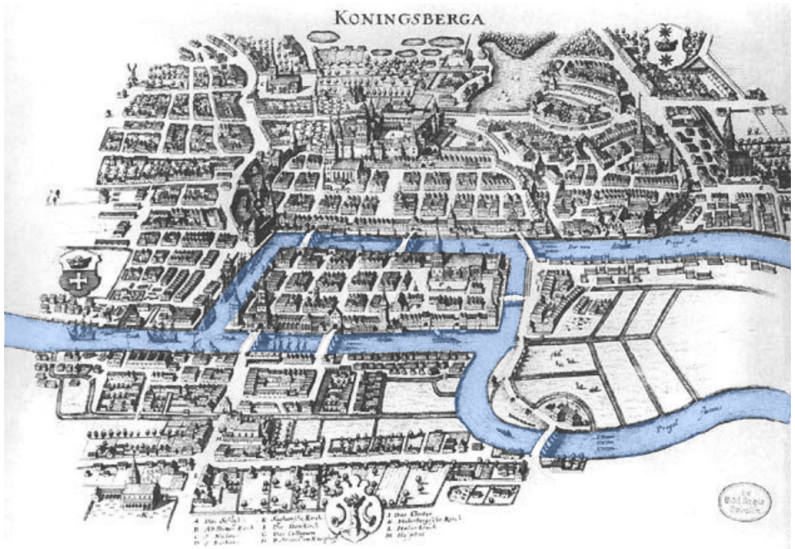
Eulersche Kantenzüge

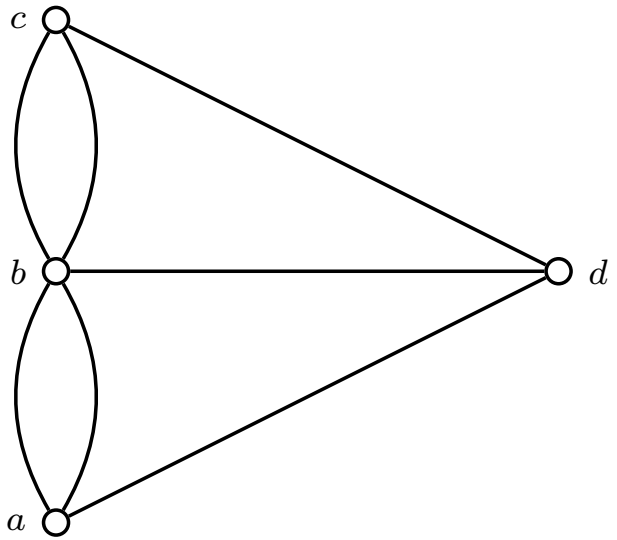
Es sei G ein Graph.

- Ein Kantenzug – geschlossen oder nicht geschlossen –, der jede Kante von G genau einmal enthält, heißt *eulersch*.
- G heißt *eulersch* gdw. G einen geschlossenen eulerschen Kantenzug enthält.

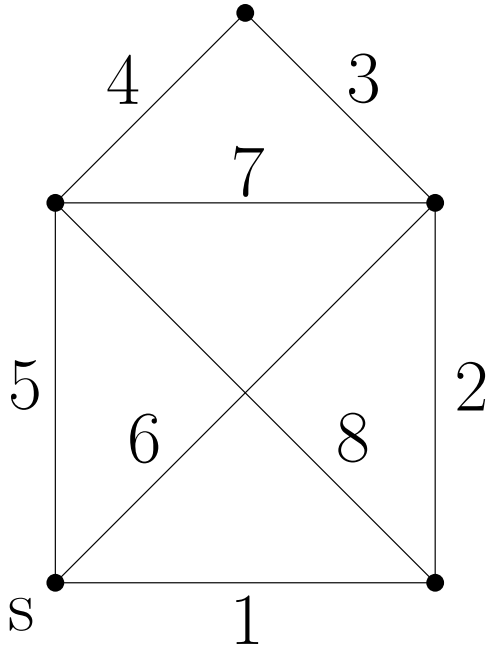
Beispiel

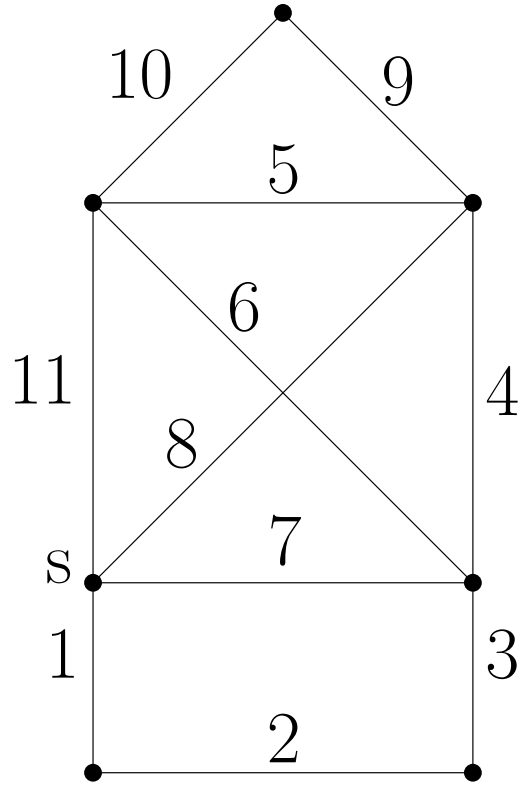
Der Graph des Königsberger Brückenproblems scheint nicht eulersch zu sein; er scheint auch keinen eulerschen Kantenzug zu enthalten:





Das Haus vom Nikolaus enthält einen eulerschen Kantenzug, das Haus vom Nikolaus mit Keller sogar einen geschlossenen.





Wir starten in s und besuchen die Kanten in der durch die Zahlen angegebenen Reihenfolge.

Satz von Euler-Hierholzer

Ein Graph G hat einen eulerschen Kantenzug gdw.

- G bis auf isolierte Knoten zusammenhängend ist und
- für die Zahl u der Knoten mit ungeradem Grad gilt: $u = 0$ oder $u = 2$.

Die Existenz eines geschlossenen eulerschen Kantenzugs ist äquivalent mit $u = 0$.

Eine weitere übliche Formulierung des Satzes lautet:
Es sei G ein Graph mit Kantenmenge E, bei dem höchstens eine Zusammenhangskomponente Kanten enthält. Dann sind folgende Aussagen äquivalent:

- G ist eulersch.
- Jeder Knoten in G hat geraden Grad.
- E ist die Vereinigung der Kantenmengen paarweise disjunkter Kreise.

Wir beweisen nun, direkt in algorithmischer Formulierung (Algorithmus von Hierholzer), die Fassung des Satzes von der vorherigen Folie für geschlossene eulersche Kantenzüge.

Zunächst besprechen wir jedoch kurz zwei Anwendungen des Satzes.

Bemerkung

Haus vom Nikolaus

Besitzt einen eulerschen Kantenzug, weil es genau zwei Knoten mit ungeradem Grad gibt (Fall $u = 2$). Beim Zeichnen des Kantenzugs muss man also immer in einem dieser beiden Knoten starten und in dem anderen enden – tut man dies nicht, so kann der Kantenzug unmöglich eulersch sein! Das Haus vom Nikolaus mit Keller besitzt sogar einen geschlossenen eulerschen Kantenzug, da $u = 0$.

Das Königsberger Brückenproblem

Alle vier Knoten sind ungeraden Grades, es gibt also keinen eulerschen Kantenzug und folglich auch keinen geschlossenen eulerschen Kantenzug.

Algorithmus von Hierholzer (1873)

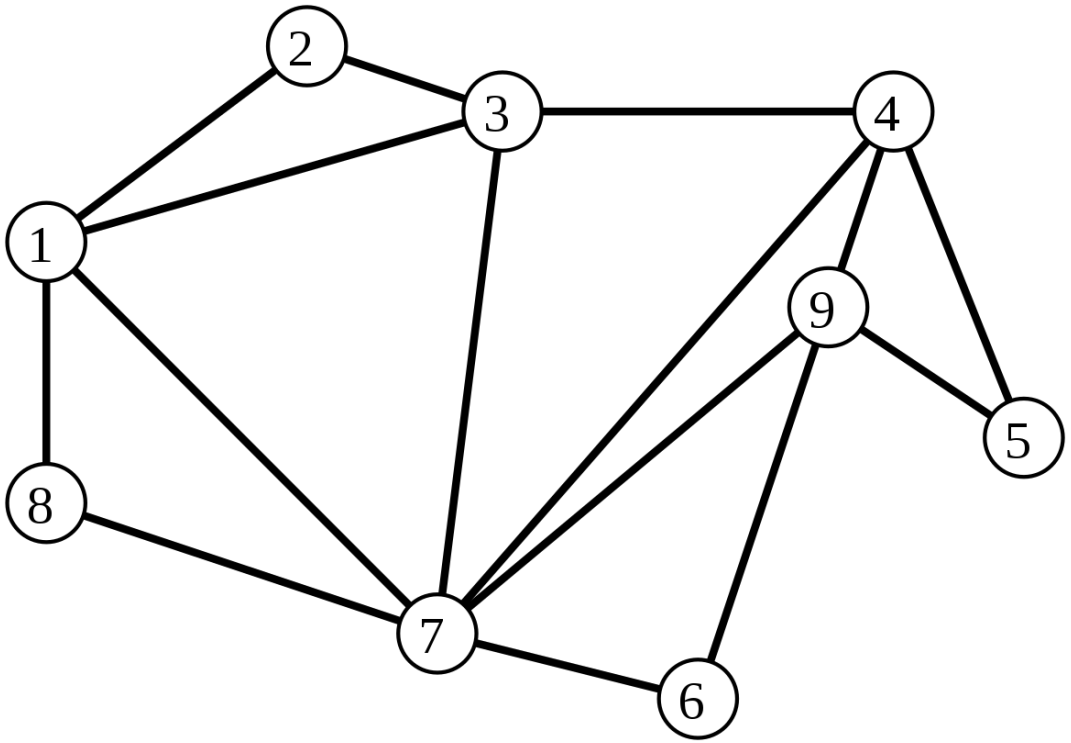
Es sei $G = (V, E)$ ein zusammenhängender Graph, der nur Knoten mit geradem Grad aufweist.

1. Wähle beliebigen Knoten $v_0 \in V$ und konstruiere von v_0 ausgehend einen geschlossenen Kantenzug K in G, der keine Kante in G zweimal durchläuft.
2. Wenn K ein geschlossener eulerscher Kantenzug ist, breche ab (wir sind fertig). Andernfalls:
3. An einem beliebigen Knoten v_1 von K, dessen Grad > 0 ist (in $G - E(K)$), lässt man nun einen weiteren geschlossenen Kantenzug K' entstehen, der keine Kante in K durchläuft und keine Kante in G zweimal enthält.
4. Füge in K den zweiten geschlossenen Kantenzug K' ein, indem der Startpunkt v_1 von K' durch alle Knoten von K' in der richtigen Reihenfolge ersetzt wird.
5. Nenne den so erhaltenen geschlossenen eulerschen Kantenzug K und fahre bei Schritt 2 fort.

In diesem algorithmischen Beweis verwenden wir implizit die Tatsache, dass zusammenhängende Graphen in denen alle Knoten geraden Grad haben, keine Brücken enthalten (in einem zusammenhängenden Graphen $G = (V, E)$ heißt eine Kante $e \in E$ *Brücke*, falls $(V, E \setminus \{e\})$ nicht mehr zusammenhängend ist).

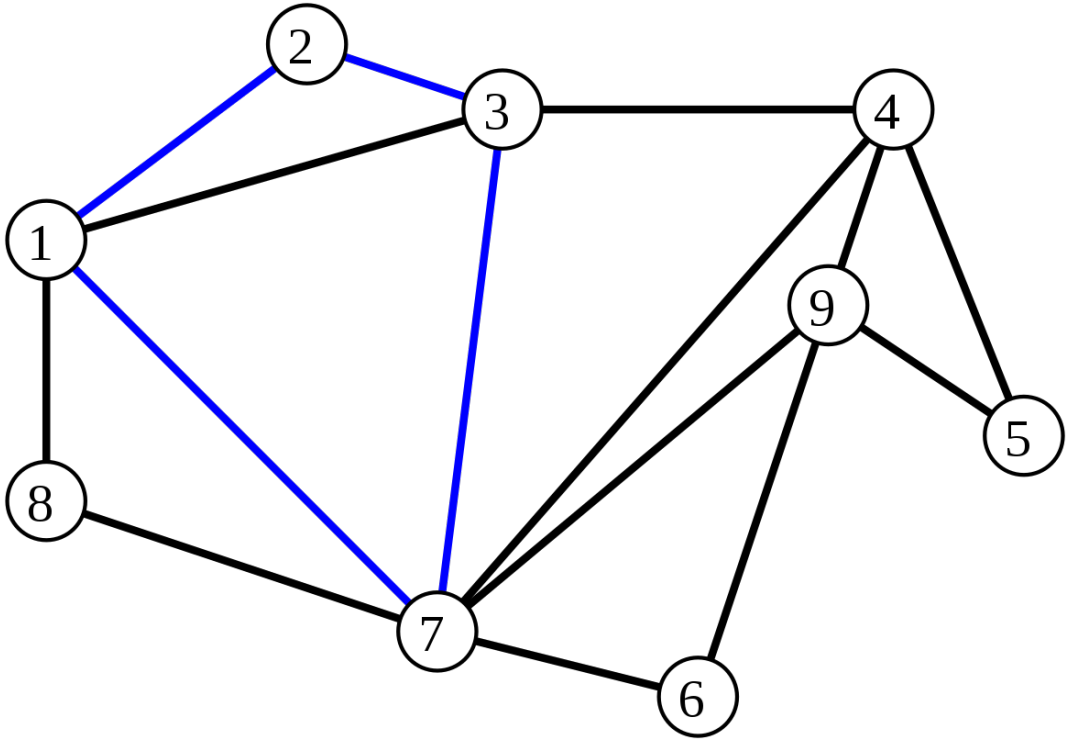
Für einen zusammenhängenden Graphen $G = (V, E)$, in dem jeder Knoten geraden Grad hat, beträgt die Zeitkomplexität des Algorithmus von Hierholzer $O(|V| + |E|)$.

Beispiel (1/4)



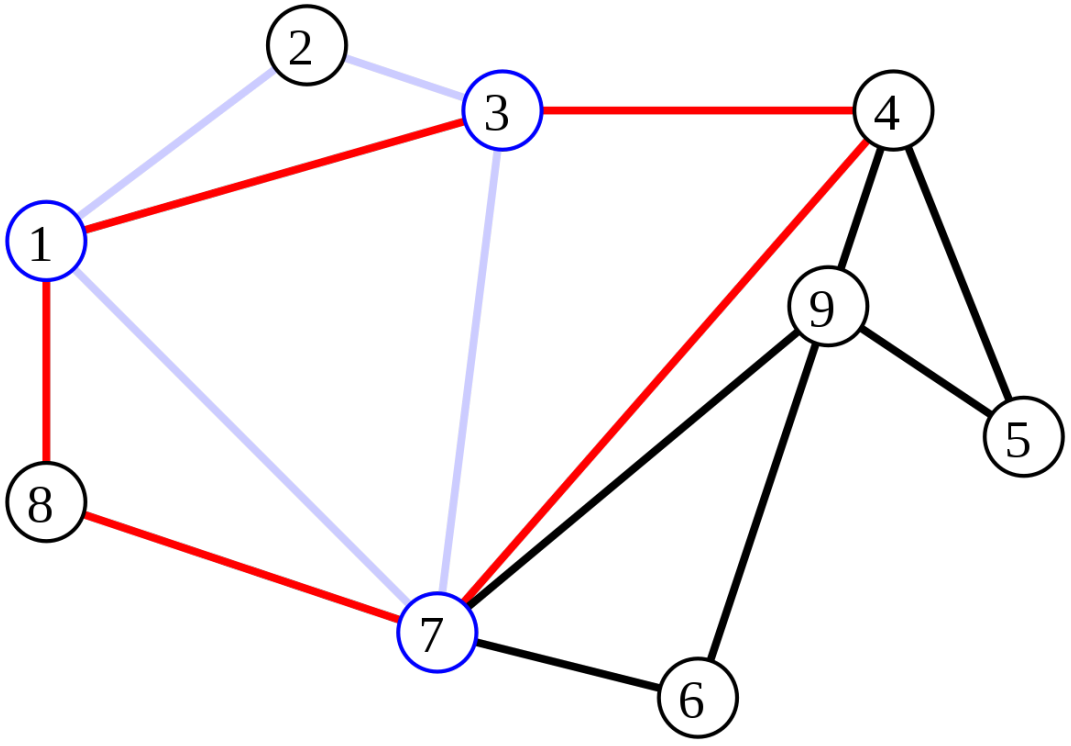
Aufgabe. Man prüfe zunächst, ob man auf obigen Graphen G den Hierholzer-Algorithmus $\mathfrak{H}$ anwenden kann. Wenn ja, so wende man $\mathfrak{H}$ auf G an.

Beispiel (2/4)



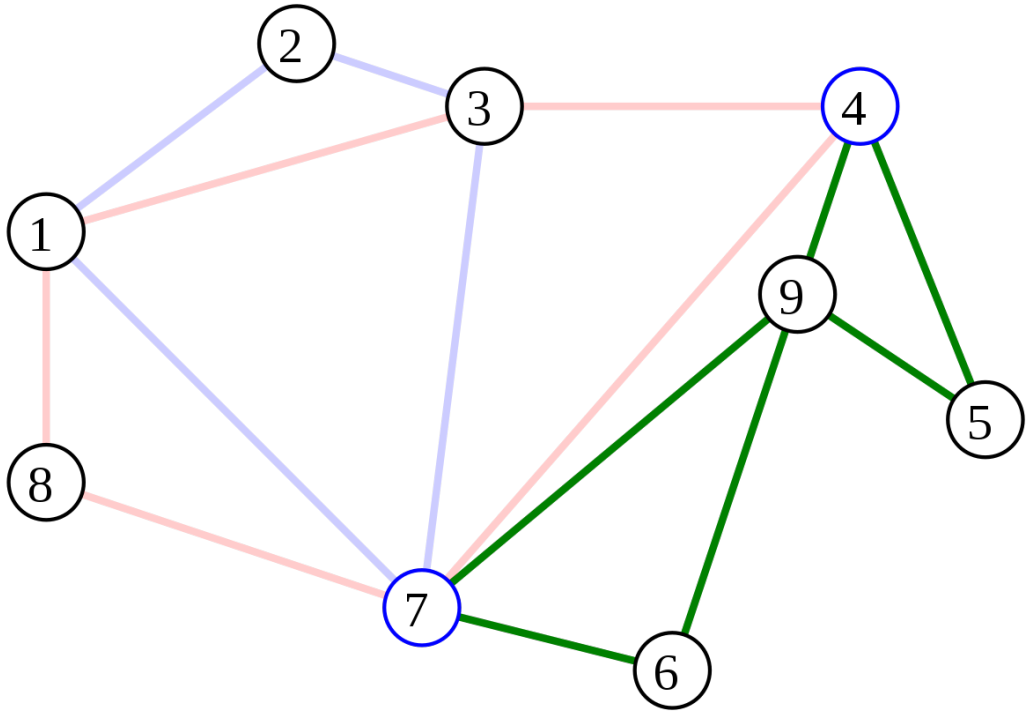
$\mathfrak{H}$ ist auf G anwendbar, da G zusammenhängend ist und alle Knoten von G geraden Grad haben. Wir starten in Knoten 1 und finden den geschlossenen Kantenzug $K^1 = 1 - 2 - 3 - 7 - 1$.

Beispiel (3/4)



Knoten 1 hat in $G - E(K^1)$ Grad $2 > 0$, wir können also wieder dort starten und finden den geschlossenen Kantenzug $K^2 = 1 - 3 - 4 - 7 - 8 - 1$ in $G - E(K^1)$. Nach Einfügen von K^2 in K^1 erhalten wir den geschlossenen Kantenzug $K = 1 - 2 - 3 - 7 - 1 - 3 - 4 - 7 - 8 - 1$.

Beispiel (4/4)



Knoten 4 hat in $G - E(K^1) - E(K^2)$ Grad $2 > 0$. Starten dort und finden den geschlossenen Kantenzug
$K^3 = 4 - 5 - 9 - 6 - 7 - 9 - 4$ in $G - E(K^1) - E(K^2)$. Nach Einfügen von K^3 in K erhalten wir den geschl. Kantenzug
$1 - 2 - 3 - 7 - 1 - 3 - 4 - 5 - 9 - 6 - 7 - 9 - 4 - 7 - 8 - 1$,
der eulersch ist. Algorithmus terminiert.

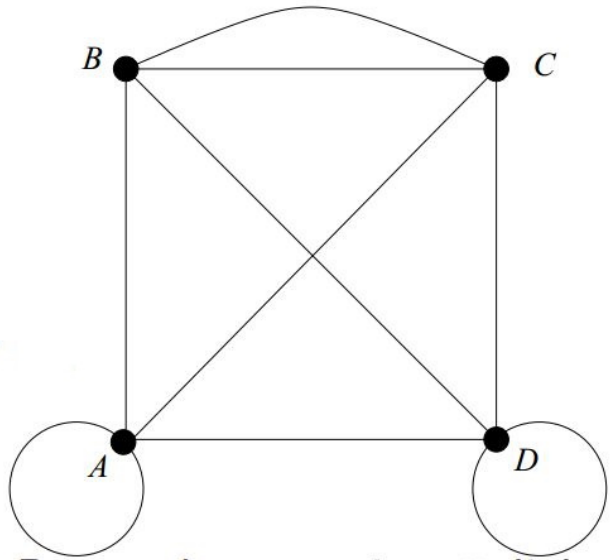
Anwendungen

- Dominospiel: Gegeben ist eine Menge S von Spielsteinen, die auf jeder Seite mit einem Symbol markiert sind.
- Einen Spielstein $[A : B]$ kann man sowohl in dieser Form, als auch als $[B : A]$ verwenden.
- Man darf zwei Spielsteine aneinander legen, wenn die sich berührenden Hälften dasselbe Symbol aufweisen.
- Kann man die Steine einer Menge S zu einer ununterbrochenen Kette zusammenlegen?

BEISPIEL

Gegeben ist die folgende Menge an Spielsteinen:
[A:B], [A:D], [B:C], [C:D], [A:A], [D:D], [B:C], [A:C], [B:D]

Zugehöriger Graph:



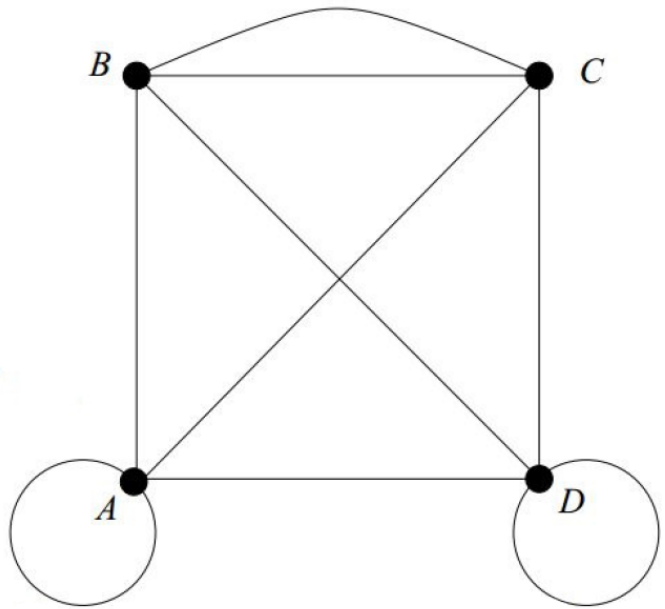
- Jede Kante entspricht einem Spielstein.
- **Eulerscher Kantenzug** entspricht einer ununterbrochenen Dominokette.

Anwendung in der Praxis: Berechnung von Prozessketten mit möglichst wenigen Unterbrechungen.

Spielsteine:

[A:B], [A:D], [B:C], [C:D], [A:A], [D:D], [B:C], [A:C], [B:D]

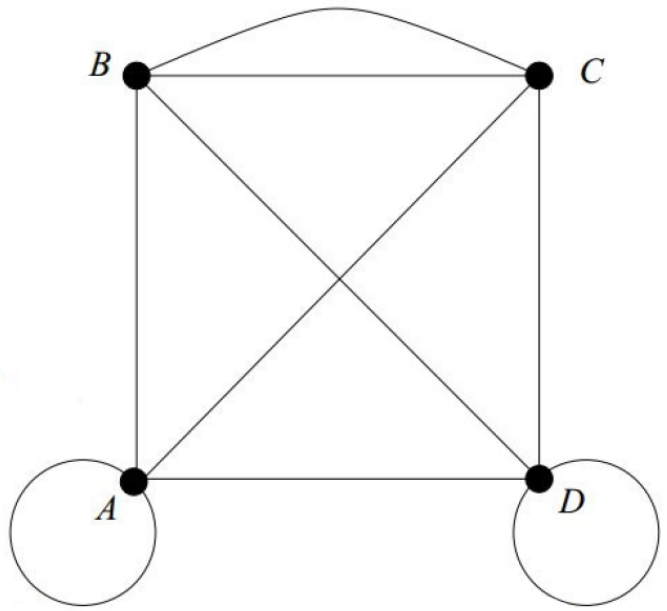
Frage. Gibt es eine ununterbrochene Dominokette?



Spielsteine:

[A:B], [A:D], [B:C], [C:D], [A:A], [D:D], [B:C], [A:C], [B:D]

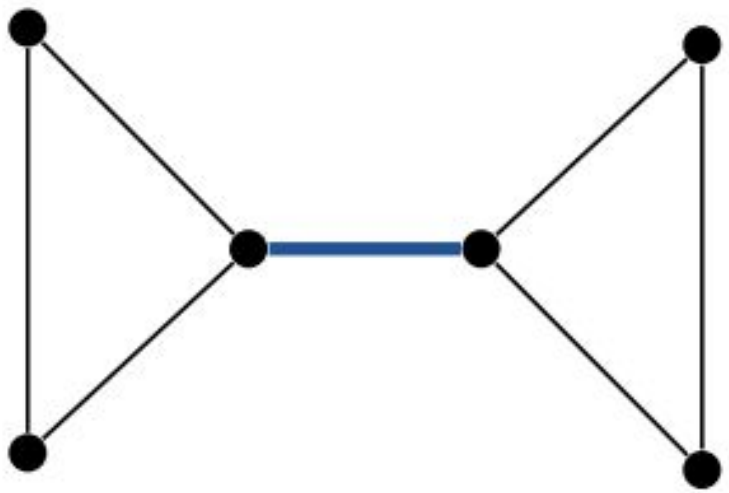
Frage. Gibt es eine ununterbrochene Dominokette?



Ja: [A:A] [A:D] [D:D] [D:C] [C:B] [B:C] [C:A] [A:B] [B:D]

Algorithmus von Fleury (1883)

Wiederholung: Es sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. Eine Kante $e \in E$ heißt *Brücke*, wenn $(V, E \setminus \{e\})$ nicht mehr zusammenhängend ist.



In diesem Beispiel ist die blaue Kante eine Brücke.

Algorithmus von Fleury

- Ob eine Kante $e \in E$ eine Brücke ist, kann man bestimmen, indem man e aus dem Graphen G entfernt und anschließend Breiten- oder Tiefensuche mit einem beliebigen Startknoten s durchführt.
- Findet diese Suche Wege von s zu allen anderen Knoten, so ist e keine Brücke.
- Gibt es hingegen Knoten, zu denen kein Weg mehr gefunden wird, so hat man eine Brücke entfernt.

Algorithmus von Fleury

Der Algorithmus fügt einer anfangs leeren Kantenfolge alle Kanten eines eulerschen Graphen hinzu, sodass ein geschlossener eulerscher Kantenzug entsteht.

1. Wähle einen beliebigen Knoten als aktuellen Knoten.
2. Wähle unter den unmarkierten, mit dem aktuellen Knoten inzidenten Kanten eine beliebige Kante aus. Dabei sind zuerst Kanten zu wählen, die im unmarkierten Graphen **keine Brückenkanten** sind.
3. Markiere die gewählte Kante und füge sie der Kantenfolge hinzu.
4. Wähle den anderen Knoten der gewählten Kante als neuen aktuellen Knoten.
5. Wenn noch unmarkierte Kanten existieren, gehe zu Schritt 2.

Algorithmus von Fleury

Äquivalent:

Input: Ein ungerichteter eulerscher Graph $G = (V, E)$.

1 Wähle einen Startknoten $v_0 \in V$.
2 Initialisiere den **geschlossenen eulerschen Kantenzug** $P := (v_0)$.
3 Setze $v := v_0$ und den Restgraphen $G_R = (V_R, E_R) := G$.
4 **while** $E_R \neq \emptyset$ **do**
5 Wähle eine Kante $e = \{v, w\} \in E_R$ mit Anfangsknoten v. Wähle hierbei nur dann eine Brücke, wenn es keine andere Kanten mit Anfangsknoten v mehr in E_R gibt.
6 Füge e und w an P an.
7 Entferne e und alle isolierten Knoten aus G_R.
8 Setze $v := w$.
9 **end while**

Output: Der **geschlossene eulersche Kantenzug** P.

Der chinesische Postbote

Ein Postbote bekommt Straßen zugeteilt, in denen er Briefe austeilen muss. Um Zeit und Wege zu sparen, versucht er, die Briefe möglichst effizient auszuteilen. Geht man davon aus, dass er in einer Straße gleichzeitig Briefe auf beiden Straßenseiten einwirft und am Ende seiner Runde wieder seine Posttasche an seiner Dienststelle abliefern muss, so sucht er einen möglichst kurzen Rundweg, auf dem er jede Straße in seinem Revier mindestens einmal durchläuft. Woher weiß der Postbote, wie er laufen muss?

Dieses Problem wird als chinesisches Postbotenproblem (*Chinese postman problem* oder *Route inspection problem*) bezeichnet, weil es 1960 erstmals von dem chinesischen Mathematiker Mei Ko Kwan untersucht wurde.

- Ist ein Graph eulersch, so wird eine Lösung für das chinesische Postbotenproblem von den Algorithmen von Hierholzer oder Fleury geliefert.
- Was jedoch, wenn der Graph nicht eulersch ist?
- Ist keine Änderung am Graphen zugelassen, so ist das Problem nicht lösbar. Wir besprechen jetzt eine Änderung, die die Struktur des Graphen *weitestgehend* unverändert lässt: wir fügen parallele Kanten hinzu.
- Dies entspricht beim Postbotenbeispiel dem mehrmaligen Durchlaufen derselben Straße.

EULERISIERUNG

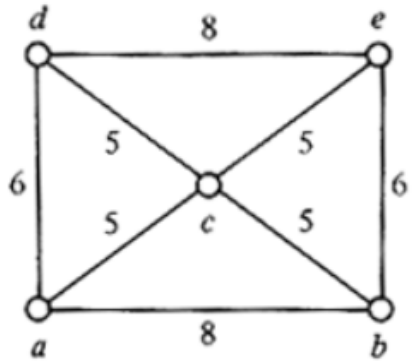
- Das Einfügen einer neuen Kante in einem Graphen ist *legal*, falls die verbundenen Knoten bereits durch eine alte Kante – also eine Kante, die im ursprünglichen Graphen bereits vorhanden war – verbunden sind.
- Es ist üblich, dass wenn wir legale Kanten einfügen müssen, wir deren Anzahl so gering wie nur möglich halten wollen (im Postbotenbeispiel: möglichst wenige Straßen mehrfach ablaufen).
- Der Prozess des Einfügens von legalen Kanten in einem Graphen G, bis ein Graph entsteht, der einen geschlossenen eulerschen Kantenzug enthält, heißt *Eulerisierung* des Graphen G.

Eulerisierung

- Das Einfügen einer neuen Kante in einem Graphen ist *legal*, falls die verbundenen Knoten bereits durch eine alte Kante – also eine Kante, die im ursprünglichen Graphen bereits vorhanden war – verbunden sind.
- Es ist üblich, dass wenn wir legale Kanten einfügen müssen, wir deren Anzahl so gering wie nur möglich halten wollen (im Postbotenbeispiel: möglichst wenige Straßen mehrfach ablaufen).
- Der Prozess des Einfügens von legalen Kanten in einem Graphen G, bis ein Graph entsteht, der einen geschlossenen eulerschen Kantenzug enthält, heißt *Eulerisierung* des Graphen G.
- Man kann jeden zusammenhängenden Graphen eulerisieren: Verdoppele **alle** Kanten.
- (Dies ist häufig nicht die effizienteste Eulerisierung.)

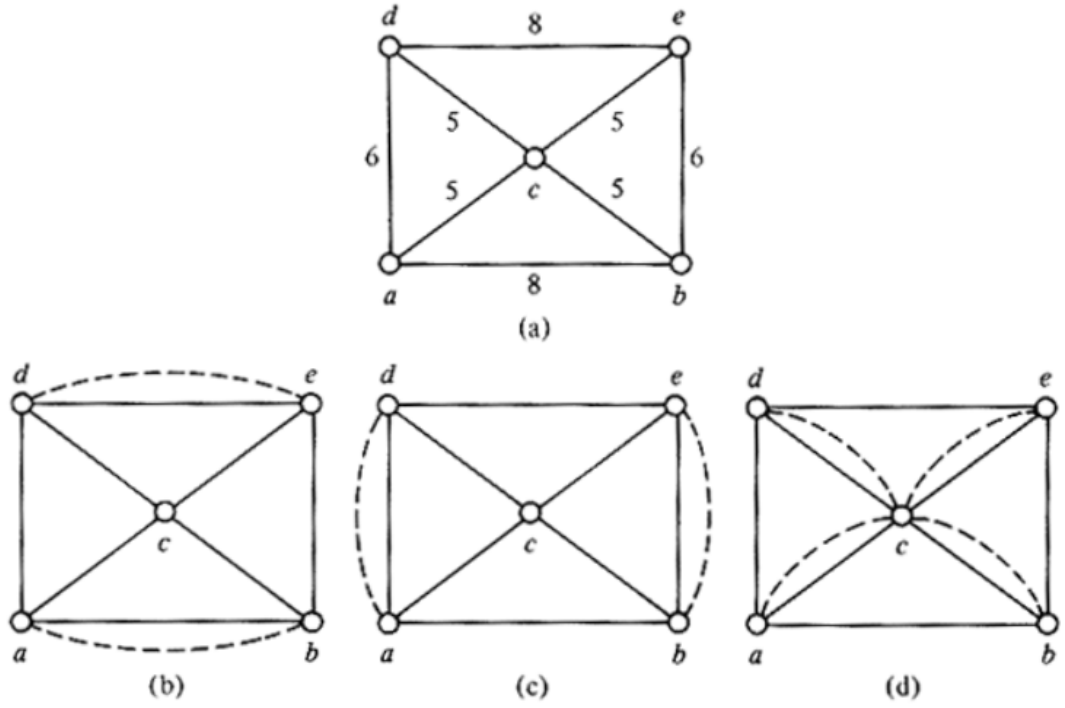
Wir betrachten nun einen Algorithmus zum Problem des chinesischen Postboten. Gegeben sei ein (ungerichteter) Graph G mit Kostenfunktion c.

1. Finde alle Knoten ungeraden Grades in G. Es sei U die Menge dieser Knoten.
2. Da $k := |U|$ gerade ist [warum?], können wir U in $k/2$ Paare aufteilen. Wir wollen dies derart tun, dass wenn wir die zwei Knoten in jedem Paar verbinden (über Wege, bei denen kein Konten bis auf die Endknoten in U liegt), *die Gesamtkosten C der $k/2$ Wege zwischen all diesen Paaren minimal sind*.
3. Auf jedem dieser $k/2$ Wege verdoppeln wir die entsprechenden Kanten in G und erhalten den Graphen G'.
4. G' enthält keine Knoten ungeraden Grades und ist daher eulersch. Die Lösung, die wir finden (z.B. mit dem Hierholzer-Algorithmus), ist optimal, und ihre Kosten entsprechen $\sum_{e \in E} c(e) + C$.



(a)

Aufgabe. Bestimmen Sie die optimale Eulerisierung des obigen Graphen.



Wir geben hier drei nicht triviale Varianten an, den Graphen zu eulerisieren. Lösung (c) ist optimal und hat Gesamtkosten

$$\underbrace{4 \cdot 5 + 2 \cdot 6 + 2 \cdot 8}_{alt} + \underbrace{2 \cdot 6}_{neu} = 60.$$

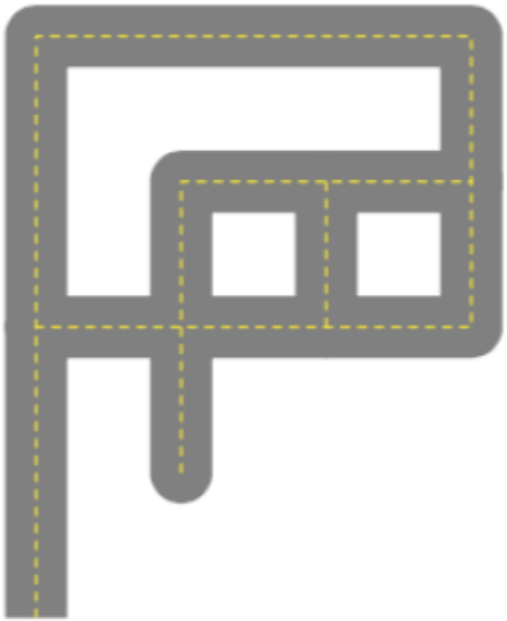
- Es ist leicht, einzusehen, dass der Algorithmus von Folie 52 auch im Allgemeinen eine optimale Lösung liefert.
- Schritte 1, 3 und 4 sind unproblematisch.
- Schritt 2 jedoch bedarf, a priori, der Berechnung von

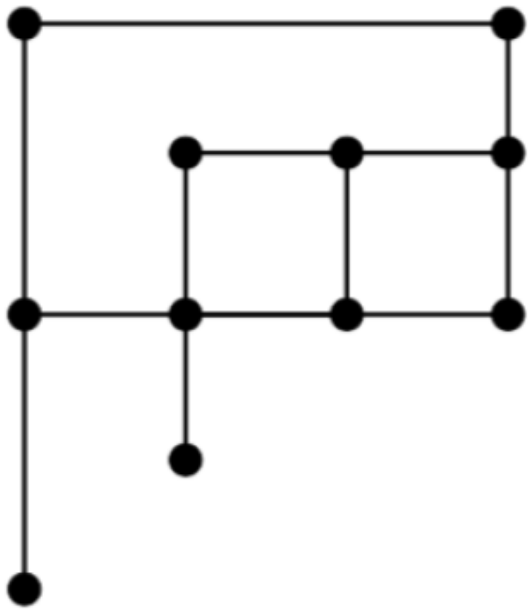
$$S = \prod_{i=1}^{k/2} (2i - 1)$$

  möglichen Paaren (k ist die Anzahl der Knoten ungeraden Grades). Leider steigt S sehr schnell.
- Glücklicherweise gibt es einen effizienten (jedoch komplizierten) Algorithmus von Jack Edmonds (1973), der das gesamte Problem in $O(n^3)$ Zeit löst.
- Es ist aber so, dass man in kleinen Graphen auch *von Hand* die beste Lösung finden kann.

Beispiel: Eulerisierung (1/4)

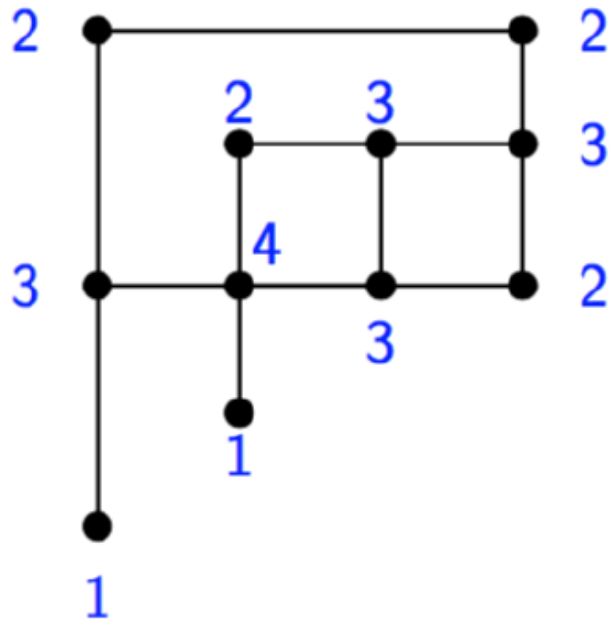
Ein Schneepflug muss den Schnee von den Straßen entfernen.
Wie kann man diesen Prozess effizient gestalten?





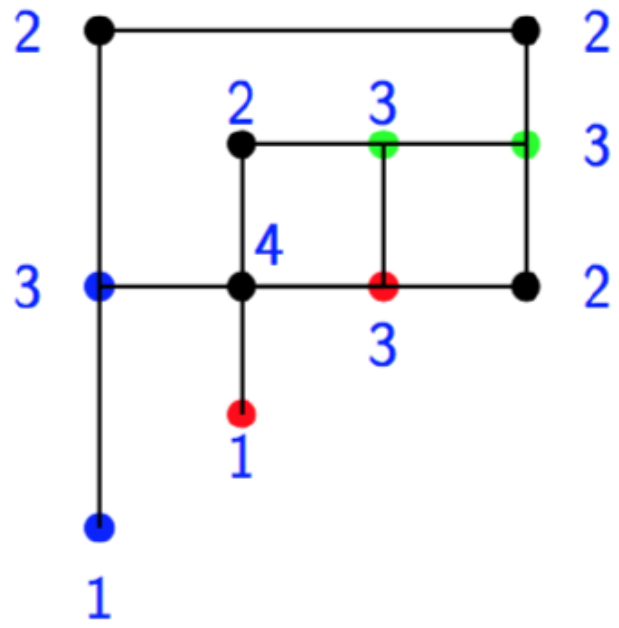
Berechne den Grad aller Knoten.

Beispiel: Eulerisierung (2/4)



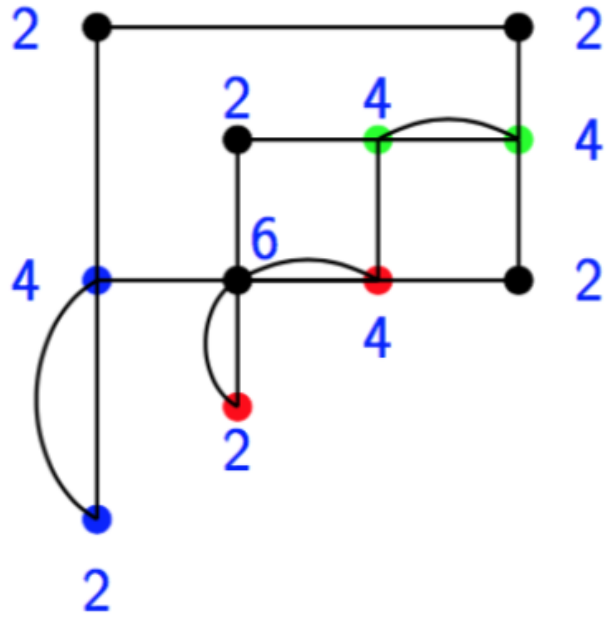
Bilde Paare von Knoten ungeraden Grades, sodass sie möglichst *nah* aneinander liegen. (Hier sind die Kantengewichte konstant, es gilt also: nah = wenig Kanten.)
Jedes Paar bekommt eine Farbe.

Beispiel: Eulerisierung (3/4)



Füge nun Kanten hinzu, um die gefärbten Knoten zu verbinden. Es kann sein, dass die Verbindung durch einen oder mehrere Zwischenknoten laufen muss.

Beispiel: Eulerisierung (4/4)



Ist dies optimal? Ja, denn jede neue Kante deckt höchstens zwei Knoten ungeraden Grades ab. Dies ergibt in unserem Fall, dass wir mindestens drei neue Kanten benötigen. Gibt es jedoch einen Knoten ungeraden Grades, der keinen Nachbarn ungeraden Grades hat, so kann die Anzahl der neuen Kanten nicht gleich der Anzahl der Knoten ungeraden Grades sein. Dies ist hier der Fall, es sind also mindestens vier Kanten nötig.