Graphentheorie

by

Dr. Günter Kolousek

© Dr. Günter Kolousek

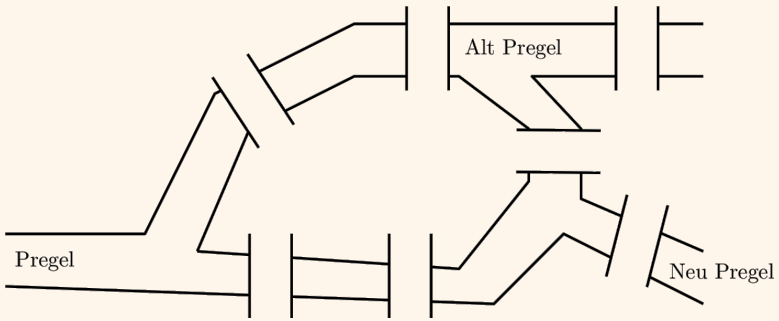
Creative Commons Namensnennung - Nicht kommerziell - Keine Bearbeitungen 4.0 International Lizenz

Motivation

- ”Alle Wege führen nach Rom!”
  - **Gibt** es wirklich einen Weg?
  - Wie findet man solch **einen** Weg?
  - Wie findet man den **kürzesten** Weg?
  - Wie findet man **alle** Wege?
  - Was ist, wenn es auch **Einbahnen** gibt?
  - Wie erkennt man **Schleifen**?
  - Was macht man **ohne** Karte?
  - Wie findet man wieder **zurück**?

Geschichte und Anwendungen

- 1736 Leonhard Euler: Königsberger Brückenproblem
  - Finden eines Weges (oder sogar Rundweg) für einen Spaziergang: jede Brücke genau einmal überqueren!



- (wichtigster) Teil der diskreten Mathematik
  - weiters: Kombinatorik, Informationstheorie, Kodierungstheorie, Zahlentheorie, Kryptographie
- Anwendungen: Routenplaner, Netzwerkoptimierung, Automatische Fahrzeuge zur Beladung und Entladung von Schiffen, Ablaufpläne,…

Definitionen

- *Graph* $G = (V, E)$
  - Menge V von Knoten (engl. vertex, pl. vertices)
  - Menge E (engl. edge) von Kanten
    - jede Kante $e \in E$ verbindet zwei Knoten u, v aus V
  - Ist $u = v$, dann spricht man von einer *Schlinge* (*Schleife*).
  - 2 Kanten haben dieselben Endknoten, dann: *parallel* (*Mehrfachkante*).
- Graph heißt *vollständig*, wenn alle seine Knoten miteinander durch Kanten verbunden sind.
  - Vollständiger Graph mit n Knoten: K_n.
  - Zeichne K_1, K_2, K_3, K_4, K_5

Arten von Graphen

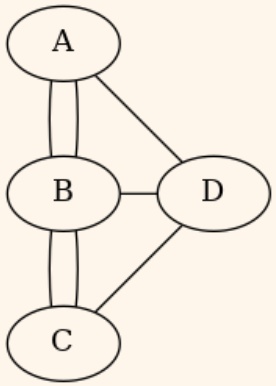
- ungerichteter Graph: Kanten sind Linien
  - Einfacher Graph: keine Schlingen, keine Mehrfachkanten
    - $E \subseteq \{V_2 | V_2 \subseteq V, |V_2| = 2\}$ V_2 ... 2-Teilmenge
      z.B.: $E = \{\{A, D\}, \{B, D\}\}$
  - Multigraph: Schlingen und/oder Mehrfachkanten
    - E, V_2 ... Multimengen!
    - $E : \{V_2 | V_2 : V \to \mathbb{N}, |V_2| = 2\} \to \mathbb{N}$
      z.B.: $E = \{\{A, A\}, \{A, B\}, \{A, B\}, ...\}$
- gerichteteter Graph: Kanten sind Pfeile, mit oder ohne Schlingen
  - ohne Mehrfachkanten: $E \subseteq V \times V$
  - mit Mehrfachkanten: $E : V \times V \to \mathbb{N}$
- gewichtete Graphen: Kante hat ein Gewicht
- Hypergraphen: Kante verbindet mehr als 2 Knoten

Definitionen – 2

- $G = (V, E)$ ein Graph mit $u, v \in V$ und $e, f \in E$.
  - v und e *inzidieren* miteinander und heißen *inzident*, wenn v ein Endknoten von e ist.
  - $u, v \in e$, dann sind u und v *adjazent* (oder *benachbart*) in G und heißen Nachbarn.
  - e und f heißen *adjazent* (oder *benachbart*) in G, wenn sie einen gemeinsamen Knoten haben.
  - $d(v)$: Grad des Knotens v ist die Anzahl des Auftretens von v als Endknoten einer Kante.
  - $\delta(G)$: Minimalgrad in G
  - $\Delta(G)$: Maximalgrad in G
  - Die Anzahl der Knoten in G bezeichnen wir mit n.
  - Die Anzahl der Kanten in G bezeichnen wir mit m.
    - m jeweils für K_1, K_2, K_3, K_4, K_5?
    - Allgemeine Formel?

Königsberger Brückenproblems

- Graph
  - Modellierung der Landmassen als Knoten
  - Modellierung der Brücken als Kanten



- Fragen
  - Um welche Art von Graph handelt es sich?
  - Gib $G = (V, E)$ an.
  - Welche Knoten und Kanten inzidieren miteinander?
  - Welche Knoten/Kanten sind zueinander adjazent?
  - Welchen Grad haben jeweils die einzelnen Knoten?
  - $\delta(G), \Delta(G)$?

Speicherungsformen von Graphen

- Adjazenzmatrix (Nachbarschaftsmatrix):

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

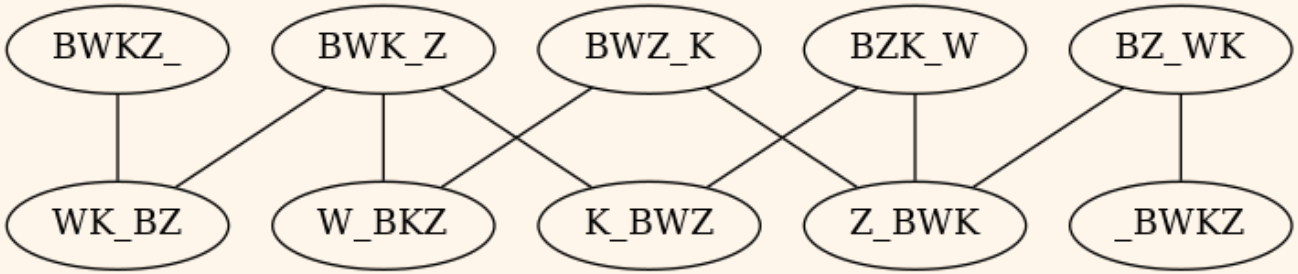
  - bei: dicht besetzten (engl. dense) Graphen
  - wenn: schnelle Überprüfung, ob zwei Knoten adjazent
  - Durchlaufen der Nachbarschaft eines Knotens: **–**
- Adjazenzliste (Nachbarschaftsliste):
  - zu jedem v_i, $1 \leq i \leq n$: Liste der adjazenten Knoten.
  - bei spärlich besetzten (engl. sparse) Graphen
  - Durchlaufen der Nachbarschaft eines Knotens: **+**
- Aufgaben für Königsberger Brückenproblem
  - ges.: Adjazenzmatrix (Redundante Information?)
  - ges.: Adjazenzliste (Redundanz?)

Rätsel

- Bauer, Wolf, Ziege, Kohl über Fluss
  - Boot: Bauer & max. ein Element von {Wolf, Ziege, Kohl}
  - Einschränkung: Wolf & Ziege oder Ziege & Kohl nie alleine!
    - Wie kommt der Bauer mit seiner Habe heil an das andere Ufer?

Rätsel

- Bauer, Wolf, Ziege, Kohl über Fluss
  - Boot: Bauer & max. ein Element von {Wolf, Ziege, Kohl}
  - Einschränkung: Wolf & Ziege oder Ziege & Kohl nie alleine!
    - Wie kommt der Bauer mit seiner Habe heil an das andere Ufer?
    - Modellierung als Graph

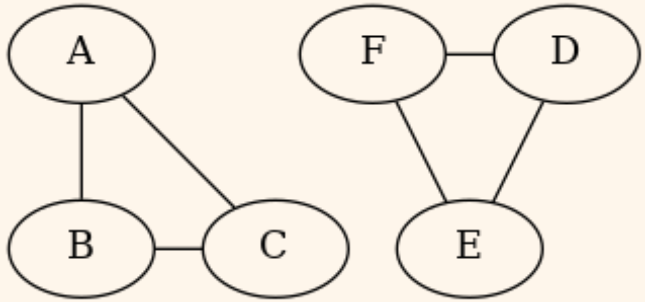


Definitionen – 3

- Ein *Kantenzug* Z in G ist eine Folge von Knoten und Kanten aus G: $Z = v_0 e_0 v_1 e_1 \ldots e_{k-1} v_k$
  - auch doppelte Kanten und doppelte Knoten!
  - v_0 und v_k heißen Endknoten
  - Anzahl der Kanten k gibt die Länge an
- Z heißt *Weg*, wenn alle Kanten verschieden sind.
- *Einfacher* Weg: keine Knoten werden doppelt durchlaufen.
- *Kreis* ist ein Weg mit gleichem Anfangs- und Endknoten.
  - *Einfacher* Kreis: außer den Endknoten kein Knoten doppelt.
- Aufgaben:
  - #Wege für den Bauern, um zum anderen Ufer zu kommen?
    - Wie viele davon sind einfach?
  - Gibt es Kreise? Wenn ja: welche? Wenn ja: auch Einfache?

Zusammenhang

- Kann man in einem Graphen von einem beliebigen Punkt zu einem anderen beliebigen Punkt kommen?



- 2 Knoten heißen verbunden, wenn sie identisch sind oder durch einen Kantenzug verbunden sind.
  - ”verbunden mit” ist eine Äquivalenzrelation
- Äquivalenzklasse dieser Relation heißt Zusammenhangskomponente.
- Besitzt *G* nur eine Zusammenhangskomponente, dann heißt G *zusammenhängend*.

Zusammenhang – 2

- Wie kann man feststellen, ob G zusammenhängend ist?
- Prinzip
  1. Wähle beliebigen Knoten v von G
  2. Markiere sukzessive (schrittweise) alle Knoten, die mit v verbunden sind.
     Markiere danach alle Knoten, die mit den schon markierten Knoten verbunden sind, usw.
  3. Sind zum Schluss alle Knoten markiert, ist der Graph zusammenhängend.

Zusammenhang – 3: Algorithmus

```
def is_connected(G):
    # start node goes into empty agenda:
    agenda = [anyitem(G[0])]
    marked = set()  # used to mark nodes

    while agenda:
        node = agenda.pop(0)
        marked.add(node)
        for other in adjacent_nodes(G, node):
            if other not in marked:
                agenda.append(other)

    return marked == G[0]

G = ({1, 2, 3, 4, 5, 6},
     {E(1,2),E(2,3),E(3,4),E(4,5),E(5,6),E(6,1)})
print(is_connected(g))
```

Zusammenhang – 4: Hilfsfunktionen

```
import collections
E = collections.namedtuple("E", "u v")

def anyitem(iterable):
    try:
        return next(iter(iterable))
    except StopIteration:
        return None
```

Zusammenhang – 5: Adjazente Knot.

```
def adjacent_nodes(G, node):
    res = set()
    for e in G[1]:
        # test if one end of the edges is same as
        # node and other one is in set of nodes
        if e.u == node and e.v in G[0]:
            res.add(e.v)
        elif e.v == node and e.u in G[0]:
            res.add(e.u)
    return res
```

Eulersche Wege

- Königsberger Brückenproblem…
- Eulerscher Weg (bzw. eulerscher Kreis): Weg (bzw. Kreis), der sämtliche Kanten des Graphen enthält.
  - Eulerscher Graph: Graph, der einen eulerschen Kreis besitzt.
- Satz: Ein ungerichteter zusammenhängender Graph ist genau dann eulersch, wenn alle Knoten gerade sind.
  - Gibt es im Königsberger Brückenproblem einen Rundgang (also mit Anfangspunkt dem Endpunkt)?
- Satz: Ein ungerichteter zusammenhängender Graph besitzt genau dann einen eulerschen Weg, wenn genau 2 oder keiner seiner Knoten ungerade ist.
  - Keiner $\rightarrow$ eulerscher Kreis

Algorithmus von Hierholzer

- (eulerscher Kreis)
- Prinzip: sukzessive Unterkreise bilden und zusammenfügen
- Algorithmus
  1. beliebigen Knoten v_0 wählen
  2. von v_0 ausgehend, sukzessive aufeinanderfolgende unbesuchte Kanten wählen bis wieder bei v_0 angekommen.
  3. Testen, ob entstandener Kreis eulersch ist. Wenn eulersch, dann fertig, anderenfalls weiter bei 4.
  4. Aus dem entstandenen Kreis einen Knoten wählen, der noch nicht besuchte inzidente Kanten hat und von diesen wieder einen Kreis konstruieren.
  5. Neuen Kreis in alten Kreis einfügen und weiter zu 3.

Suchen und kürzeste Wege

- aufspannende Bäume
  - Tiefen und Breitensuche $\rightarrow$ Suchen
  - Bestensuche (Algorithmus von Prim) $\rightarrow$ minimal aufspannender Baum
  - Dijkstra-Algorithmus $\rightarrow$ optimale (kürzeste) Wege
- A*-Alogorithmus $\rightarrow$ *Heuristik* zum Finden *eines* optimalen Weges zwischen zwei Knoten
- Traveling Salesman Problem $\rightarrow$ kürzester Rundweg

Bäume

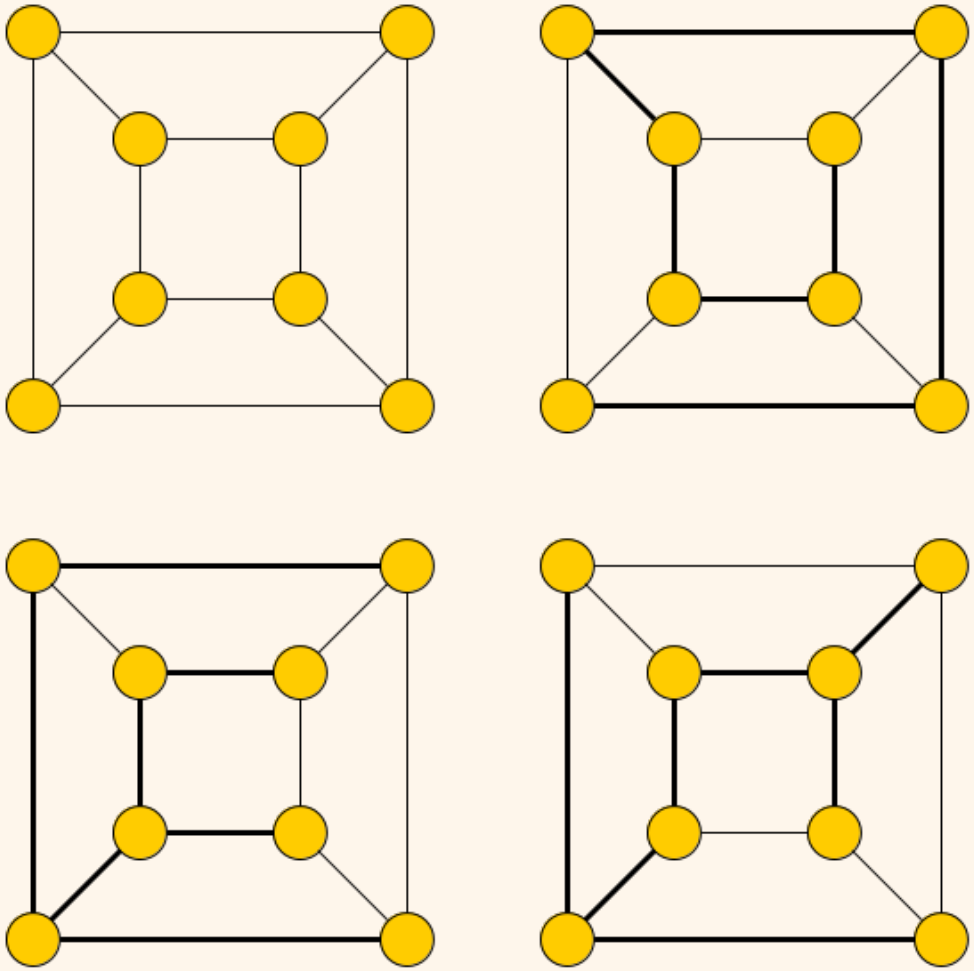
- Definition: Baum ist ein Graph, der zusammenhängend und kreisfrei ist.
- Satz: Sei G ein Graph mit n Knoten, dann sind folgende Aussagen äquivalent:
  - G ist ein Baum.
  - G ist kreisfrei und hat $n - 1$ Kanten.
  - G ist zusammenhängend und hat $n - 1$ Kanten.

Aufspannende Bäume

- engl.: spanning tree
- Def.: Ein Graph $G' = (V', E')$ ist ein Teilgraph von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$.
- Def.: Ein Teilgraph G' von G, der ein Baum ist und für den $V = V'$ gilt, heißt *aufspannender* Baum.
  - Idee: Aus G solange Kanten löschen, wie dieser noch zusammenhängend ist. Kann keine weitere Kante gelöscht werden $\rightarrow$ aufspannender Baum!
- Satz: Jeder zusammenhängende Graph enthält einen aufspannenden Baum.
- Vgl. ”spanning tree protocol”

Aufspannende Bäume – 2

Beispiele



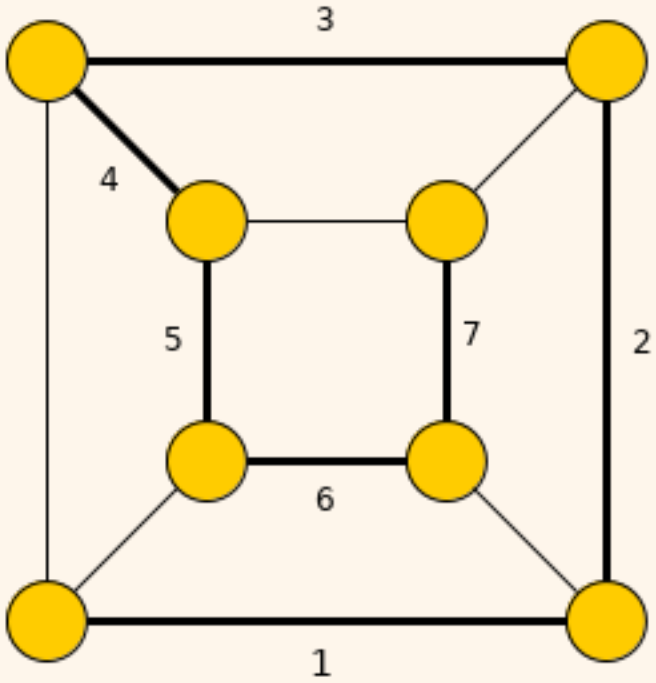
Aufsp. Bäume – Konstruktion

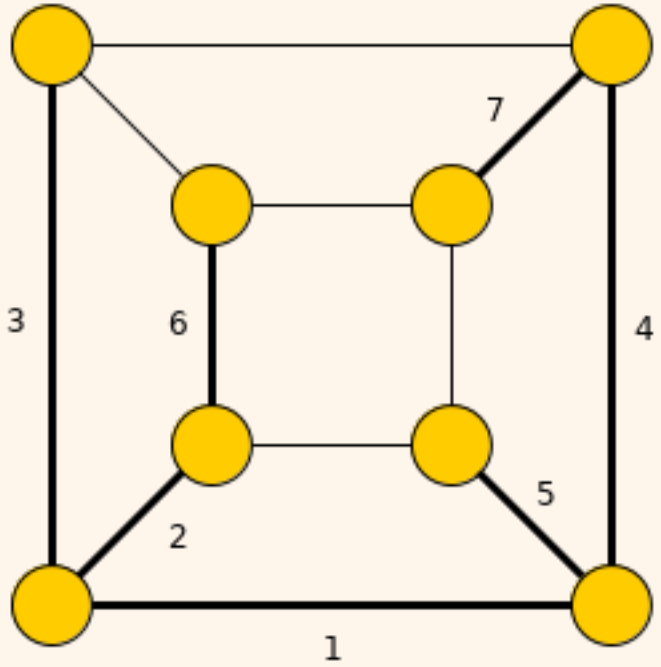
- spannender Baum $B = (V, T)$ von $G = (V, E)$ mit $T \subseteq E$
- Algorithmus
  1. Initialisierung: Agenda $= \{\}$, $T = \{\}$
  2. Startknoten wählen und zur Agenda hinzufügen
  3. Solange Agenda nicht leer:
     - Einen Knoten u aus Agenda entfernen
     - Ist u schon markiert, dann zu 3 gehen, anderenfalls u markieren.
     - Hat u einen Vaterknoten v, dann Kante (u, v) (vom Kind zum Elter) zu T hinzufügen.
     - Alle zu u adjazenten und nicht markierten Knoten in die Agenda einfügen.

Tiefensuche vs. Breitensuche

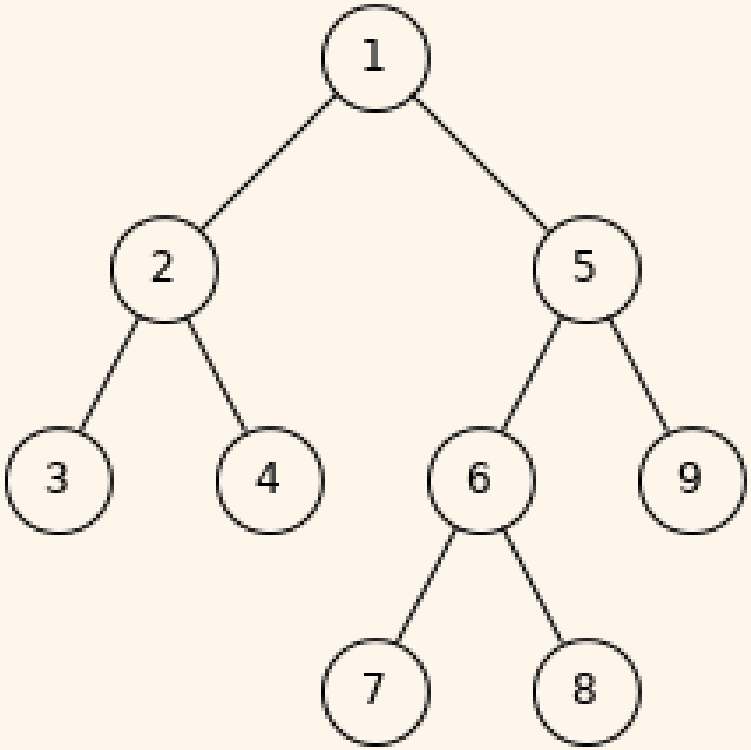
- basierend auf aufspannendem Baum
- Tiefensuche (engl. depth-first-search)
  - Prinzip: Jeweils den letzten gefundenen Knoten expandieren bis man an einen Knoten gelangt, an dem man nicht mehr weiter kommt, weil dessen sämtliche Nachbarn schon markiert sind. Dann zurück zum letzten Verzweigungspunkt (engl. backtracking).
  - Agenda ist als Stack organisiert
- Breitensuche (engl. breadth-first-search)
  - Prinzip: Jeweils eine Schicht vollständig abarbeiten, dann zur nächsten Schicht wechseln bis man alle Schichten abgearbeitet hat.
  - Agenda ist als Queue organisiert

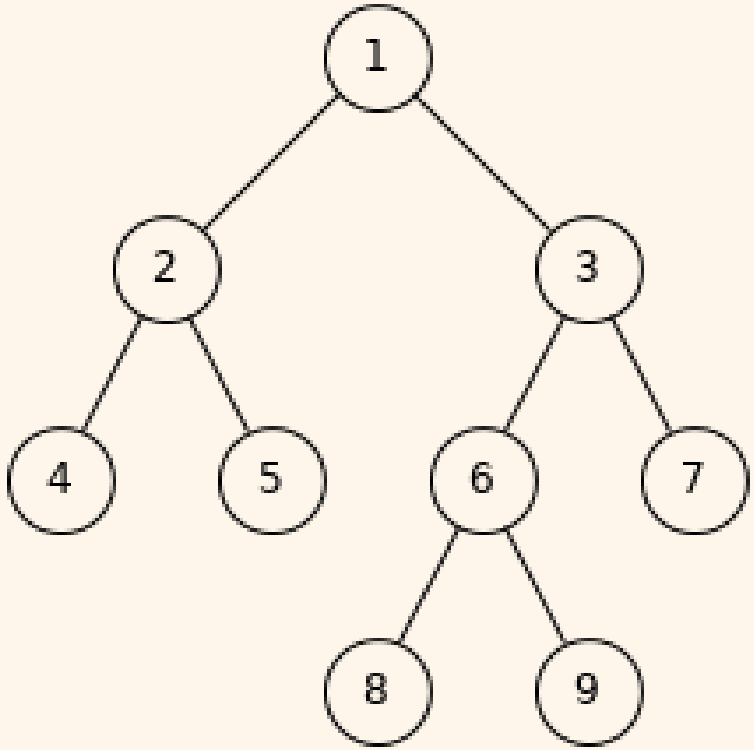
Tiefensuche vs. Breitensuche – 2





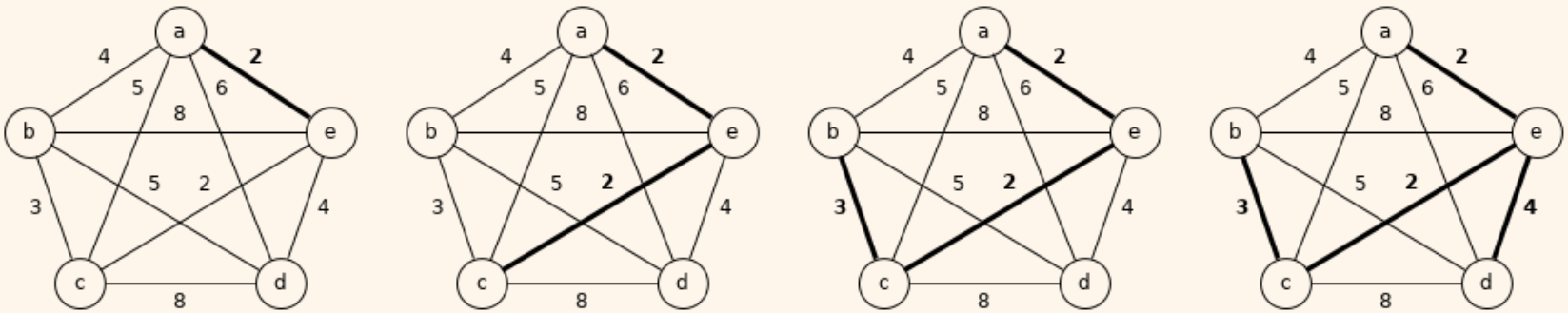
Tiefensuche vs. Breitensuche – 3



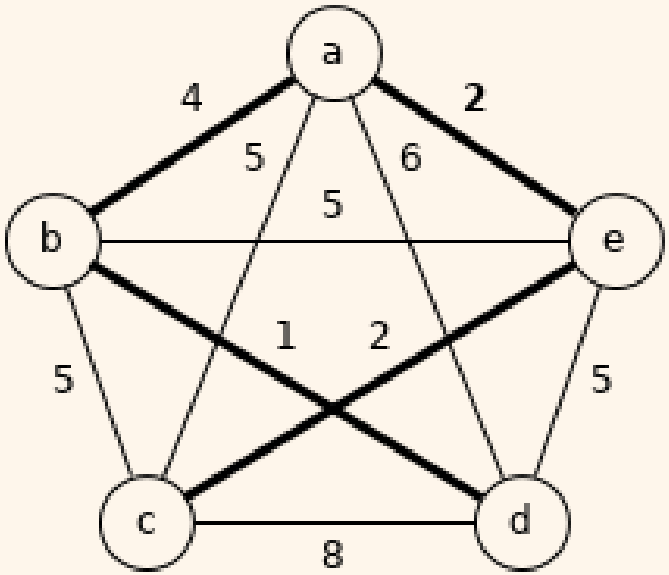


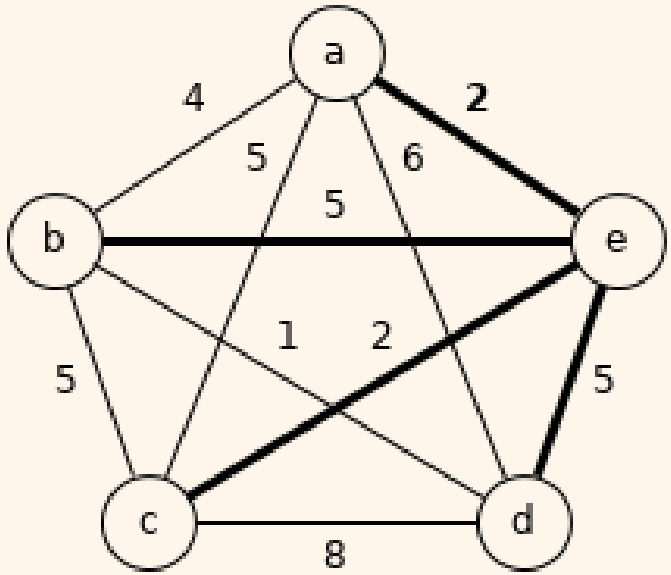
Minimaler aufspannender Baum

- Minimum spanning tree (MST)
- Ungerichteter, gewichteter Graph (Kosten)
- ges.: aufspannender Baum, der *Gesamtkosten* minimiert
- Bestensuche
  - Problem: 5 Städte sollen durch ein Straßennetz verbunden werden, sodass jede Stadt von jeder anderen aus erreichbar ist. Straßenbaukosten sollen *minimal* sein.
  - Aufspannenden Baum aufbauen jedoch anstatt Stack bzw. Queue wird jetzt eine *Priority Queue* verwendet (basierend auf Min-Heap) $\rightarrow$ Bestensuche (Algo. von Prim)



MST vs. kürzeste Wege





Dijkstra-Algorithmus

- Ungerichteter, gewichteter Graph (Kosten)
- ges.: aufspannender Baum, der vom Startknoten die *kürzesten* Wege zu allen anderen Knoten enthält
- Wir definieren:
  - N: Menge der Knoten für die noch kein kürzester Weg berechnet ist.
  - Startknoten s
  - $w(u, v)$: Kosten von u nach v, liefert ∞, wenn keine Verbindung zwischen u und v existiert.
  - $c[u]$: Gesamtkosten von u nach s

Dijkstra-Algorithmus – 2

Grundstruktur:

```
N = V - {s}
# Kosten fuer alle Knoten in G initialisieren
for n in N:
    c[n] = w(G, s, n)
# solange N nicht leer
while N:
    # waehle f aus N, sodass c[f] minimal:
    cmin = float("infinity")
    f = None
    for n in N:
        if c[n] < cmin:
            cmin = c[n]
            f = n
    # ausgewaehltes f aus N entfernen und Kosten berechnen
    N -= {f}
    for n in N:
        c[n] = min(c[n], c[f] + w(G, f, n))
```

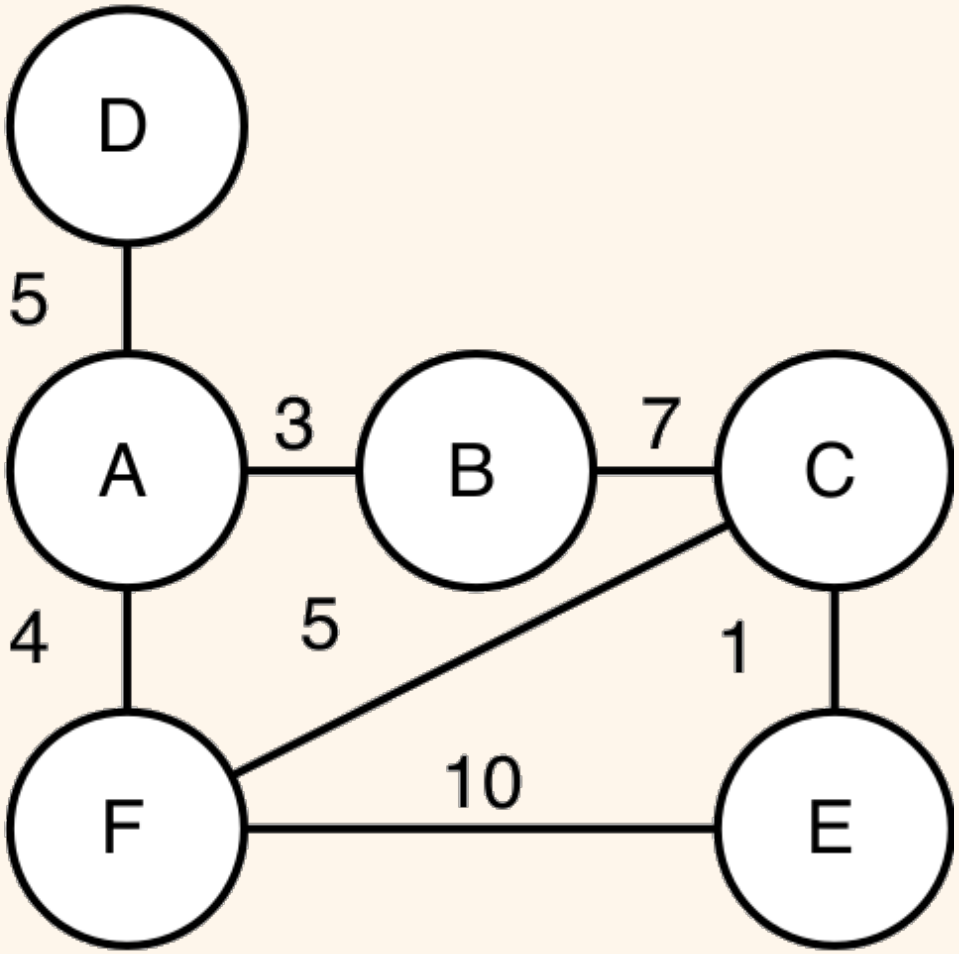
Dijkstra-Algorithmus – 3

- Verbesserung, da
  - bis jetzt nur die minimalen Kosten ermittelt werden
  - Wahl von f betrachtet alle Knoten von N
- Deshalb: Menge weiter unterteilen
  - Menge der Randknoten B (engl. border)
  - Menge der Ergebnisknoten R (engl. results)
- $p[n]$: Vorgänger des Knoten n

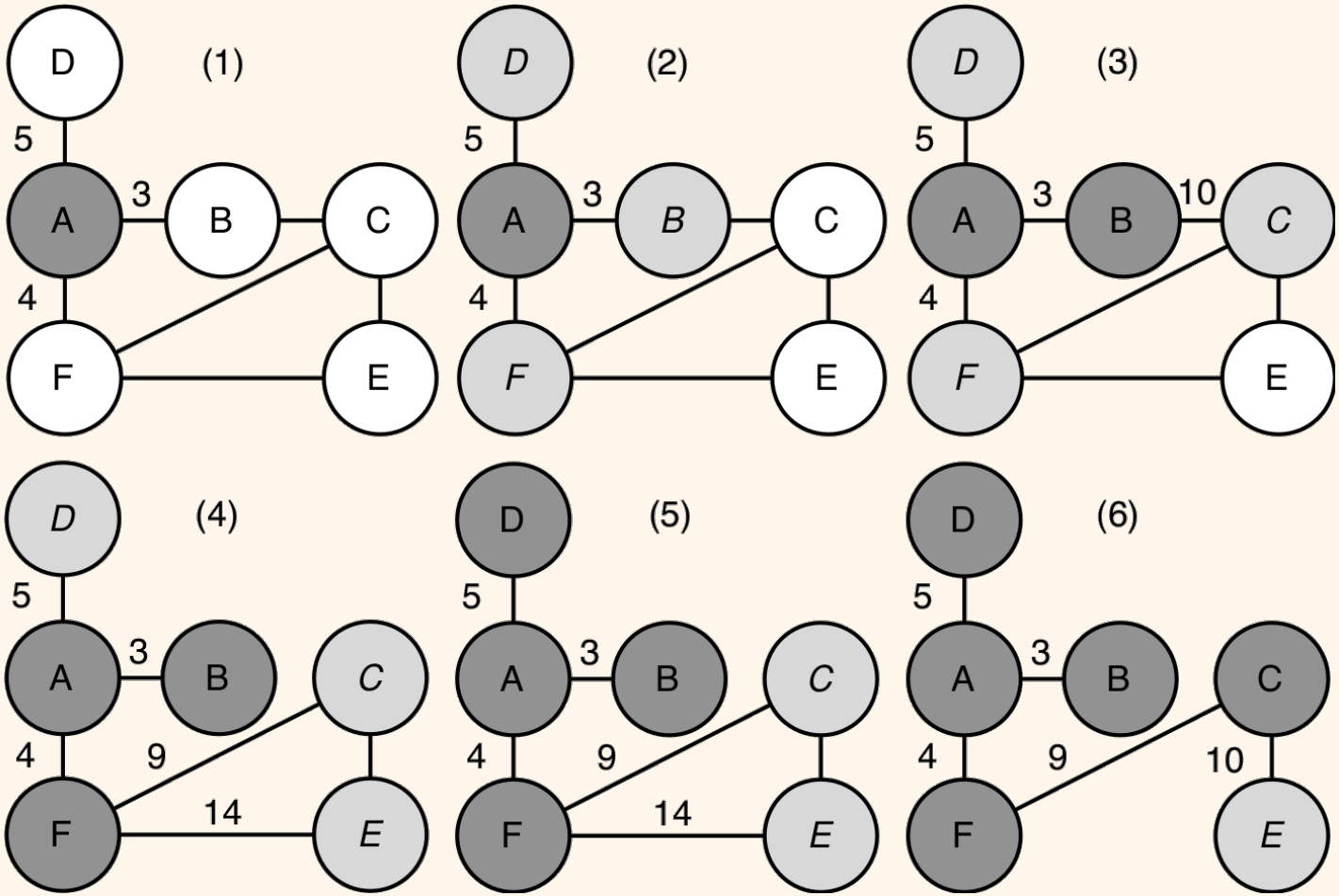
Dijkstra-Algorithmus – 4

```
R = {s}  # Menge der Ergebnisknoten
N = V - {s}  # alle anderen Knoten
for n in N:
    c[n] = w(s, n)  # Kosten des Knotens (relativ zu s)
    p[n] = None  # Vorgaenger gibt es noch keinen
B = adjacent_nodes(G, s) # B mit Nachbarn von s init.
for b in B:  # Vorgaenger von b setzen
    p[b] = s
while B:  # solange B nicht leer
    waehle f aus B, sodass c(f) minimal ist
    B = B - {f}  # aus Rand entfernen
    R = R | {f}  # zu den Gefundenen hinzufuegen
    for n in adjacent_nodes(G, f):
        if n not in R:
            B = B | {n}  # zum Rand hinzufuegen
            if c[f] + w(f,n) < c[n]: # Gesamtkost. kleiner?
                c[n] = c[f] + w(f,n)
                p[n] = f
```

Dijkstra-Algorithmus – 5



Dijkstra-Algorithmus – 6



A^*

- Verallgemeinerung von Dijkstra
- Suche eines optimalen Weges zwischen zwei Knoten
  - vollständig und optimaler Algorithmus, d.h. findet *eine optimale* Lösung, falls diese existiert
- verwendet eine Schätzfunktion ($\rightarrow$ Heuristik) um zielgerichtet zu suchen
  - $\rightarrow$ informierter Suchalgorithmus
  - Heuristik
    - Heureka: ”ich habe [es] gefunden”
    - Heuristik: ”Kunst mit begrenztem Wissen und wenig Zeit denoch zu wahrscheinlichen Aussage oder praktikalen Lösungen zu kommen.”
    - $\rightarrow$ Erfahrung
  - darf echte Kosten nie überschätzen!!!
    - geeignet wäre z.B. Luftlinie bei Entfernungen

A^* – 2

- Vorteile/Nachteile
  - ist schneller (…um *einen* Weg zu suchen)
  - benötigt viel Speicher!

A* – 3

```
open_list = []
closed_list = {}  # closed list *and* dedicated parent
c = {s : 0}  # costs for node
# open_list consists of (costs, (node, parent))
heapq.heappush(open_list, (0, (s, None)))

while open_list:
    _, f = heapq.heappop(open_list) # priority doesn't matter
    if f[0] in closed_list:
        # current node is already expanded...
        continue
    # insert current node into closed list and remember parent
    closed_list[f[0]] = f[1]
    if f[0] == g:  # current node *is* the goal
        break
    for n in adjacent_nodes(G, f[0]):
        if n not in closed_list:
            c[n] = c[f[0]] + w(G, f[0], n) # actual costs
            # extend with estimated costs to goal and node
            heapq.heappush(open_list, (c[n]+h(n,g), (n,f[0])))
```

A^* – 4

```
# construct path if solution has been found
path = []
f = f[0]
if f == g:
    path.insert(0, f)
    while f and f != s:
        f = closed_list[f]
        path.insert(0, f)
```

Traveling Salesman Problem

- Motivation
  - Handlungsreisender
  - zu besuchenden Orte
  - Anfangsort = Endort
  - Reihenfolge, sodass Reisestrecke minimal
- ges.: Kreis, der alle Knoten umfasst und minimale Summe der Gewichte umfasst
- Anwendungen
  - Tourenplanung
  - Logistik
  - Entwurf von integrierten Schaltkreisen

TSP – 2

- Problem
  - alle Kreise müssen berechnet werden!
  - insgesamt gibt es $\frac{(n-1)!}{2}$ Möglichkeiten!
    - bei 10 Orten: 181440
    - bei 15 Orten: 43589145600 (~43 Milliarden)
    - bei 20 Orten: 6082255020441600 (~60 Billiarden)
- $\rightarrow$ für exakte Lösung unpraktikabel
  - daher: Vielzahl an Algorithmen basierend auf Heuristik!
  - z.B. immer den nächsten Nachbarort besuchen
    - d.h. den Ort mit der geringsten Distanz