5 Graphalgorithmen

Wir lernen in diesem Abschnitt ...

- ... was Graphen sind und wozu man sie braucht (Abschnitt 5.1.1).
- ... wie man Graphen in einer Programmiersprache repräsentiert (Abschnitt 5.1.2).
- ... wie man einen Graphen systematisch durchlaufen kann (Abschnitt 5.2).
- ... wie man den kürzesten Weg zwischen zwei (oder mehreren) Knoten berechnet (Abschnitt 5.3).
- ... wie man einen minimalen Spannbaum – eine Art „kostengünstigsten“ Verbindungsgraphen – berechnet (Abschnitt 5.4).
- ... wie man einen maximal möglichen (Waren-)Fluss in einem Netzwerk aus Knoten und Flusskapazitäten berechnet (Abschnitt 5.5).

Voraussetzung für das Verständnis der in diesem Kapitel vorstellten Algorithmen ist die Kenntnis der grundlegenden mathematischen Konzepte die der Graphentheorie zugrunde liegen. Anhang B.4 liefert den notwendigen Überblick.

5.1 Grundlegendes

5.1.1 Wozu Graphen?

Ein Graph ist ein mathematisches Objekt, bestehend aus *Knoten* und Verbindungen zwischen Knoten, genannt *Kanten*. Weitere mathematische Details zu Graphen finden sich in Anhang B.4.

Graphen sind in der Informatik das Mittel der Wahl um eine Vielzahl von Phänomenen der realen Welt zu repräsentieren. Es gibt eine Vielzahl von Beispielen für „Dinge“, die sich angemessen durch Graphen repräsentieren lassen, etwa ein Straßennetz (Knoten: Städte, Kanten: Verbindungen zwischen Städten), Mobilfunkteilnehmer (Knoten: Handys oder Basisstationen; Kanten: Verbindungen zwischen Handy und Basisstation), ein Ablaufplan (Knoten: Zustand; Kanten: möglicher Übergang von einem Zustand zu einem anderen) oder das Internet (Knoten: Websites; Kanten: Link einer Website zu einer anderen) oder Hierarchische Beziehungen (Knoten: Begriffe; Kanten: Beziehungen zwischen Begriffen wie etwa „ist ein“), usw. Als Beispiel ist in Abbildung 5.1 ein Graph zu sehen, der einen Teil des Semantic Web zeigt; in Abbildung 5.2 ist ein kleiner

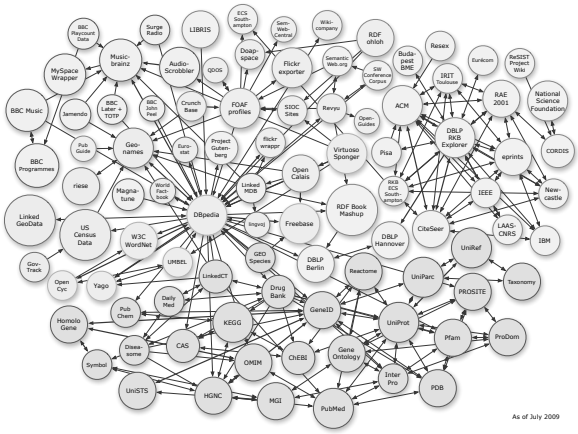


Abb. 5.1: *Ein Ausschnitt des sog. Semantic Web, einem Teil des WWW, in dem sich Informationen befinden über die Bedeutung verschiedener Begriffe und deren Beziehungen untereinander; die Knoten stellen Gruppen von Begriffen dar; die Kanten geben an, zwischen welchen Begriffsgruppen Beziehungen bestehen.*

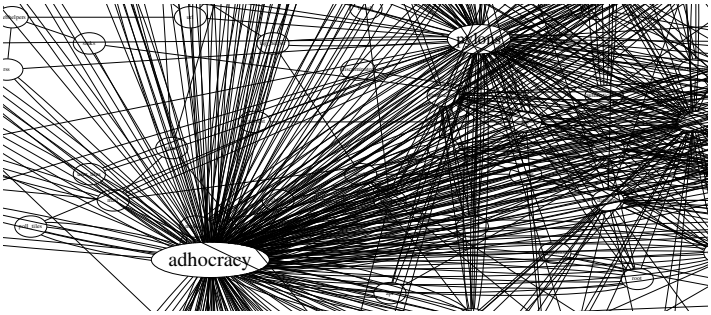


Abb. 5.2: *Ein Aussschnitt aus den als Graph modellierten Importbeziehungen eines größeren Python-Projektes, des Liquid-Democracy-Tools „Adhocracy', modelliert als ungerichteter Graph.*

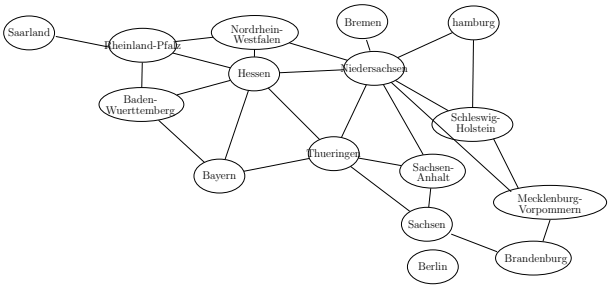


Abb. 5.3: *Ein Graph der die Nachbarschaftsbeziehung der Bundesländer modelliert.*

Teil der Importbeziehungen der Module eines größeren Softwareprojektes zu sehen; der Graph aus Abbildung 5.3 zeigt die Nachbarschaftsbeziehung der Bundesländer. Wichtig ist dabei sich vor Augen zu halten, dass die mathematische Struktur „Graph“ i. A. von der räumlichen Anordnung der Knoten abstrahiert, d. h. es spielt keine Rolle, ob ein Knoten v_i links von einem Knoten v_j gezeichnet wird oder rechts. Alleine entscheidend ist nur die Information, welche Knoten miteinander verbunden sind.

5.1.2 Repräsentation von Graphen

Es gibt zwei grundsätzlich verschiedene Möglichkeiten der Darstellung eines Graphen im Rechner; jede hat Ihre Vor- und Nachteile und man muss sich je nach anzuwendendem Algorithmus und je nach „Dichte“ des Graphen von Fall zu Fall neu entscheiden, welche der beiden Darstellungsformen man für die Repräsentation eines Graphen $G = (V, E)$ verwendet, wobei V die Menge der Knoten und E die Menge der Kanten darstellt.

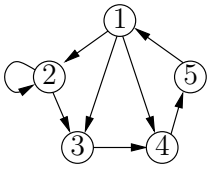


Abb. 5.4: *Ein einfacher gerichteter Graph.*

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

Abb. 5.5: *Repräsentation des in Abbildung 5.4 gezeigten Graphen als Adjazenzmatrix.*

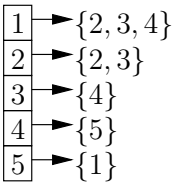


Abb. 5.6: *Repräsentation des in Abbildung 5.4 gezeigten Graphen als Adjazenzliste.*

1. Darstellung als *Adjazenzmatrix*:
Der Graph wird in Form einer Matrix A repräsentiert, wobei der Eintrag in der i-ten Zeile und der j-ten Spalte 1 ist, falls es eine Verbindung von i nach j im Graphen G gibt; formaler ausgedrückt muss für die Adjazenzmatrix $A = (a_{ij})$ gelten:

$$a_{ij} = \begin{cases} 1, \text{ falls } (i,j) \in E \\ 0, \text{ sonst} \end{cases}$$

Abbildung 5.5 zeigt ein Beispiel.

2. Darstellung als *Adjazenzliste*:
Der Graph wird als Liste seiner Knoten gespeichert. Jeder Eintrag in der Liste zeigt auf die zum jeweiligen Knoten benachbarten (d. h. adjazenten) Knoten. Abbildung 5.6 zeigt ein Beispiel.

Besitzt der Graph relativ „wenige“ Kanten (im Vergleich zum vollständigen Graphen $K = (V, V \times V)$), so ist die Repräsentation als Adjazenzmatrix sehr verschwenderisch, was den Speicherbedarf betrifft, und die Adjazenzmatrix wäre eine sog. *dünn besetzte* Matrix, d. h. eine Matrix, in der die meisten Einträge 0 sind. In solchen Fällen, insbesondere dann, wenn der Graph viele Knoten hat, empfiehlt sich die Repräsentation als Adjazenzliste.

Bestimmte grundlegende Operationen sind je nach Darstellungsform unterschiedlich

aufwändig. Der Test, ob eine bestimmte Kante (i,j) im Graphen enthalten ist, braucht nur $O(1)$ Schritte, wenn der Graph als Adjazenzmatrix repräsentiert ist, jedoch $O(deg(i))$, wenn der Graph als Adjazenzliste gespeichert ist. Andererseits benötigt das Durchlaufen der Nachbarschaft eines Knotens i – eine häufig durchgeführte Operation bei der Breiten- und Tiefensuche – nur $O(deg(i))$ Schritte, wenn der Graph als Adjazenzliste gespeichert ist, jedoch $O(n)$ Schritte, wenn der Graph als Adjazenzmatrix gespeichert ist, wobei i. A. $deg(i) \ll n$ gilt.

In Python sind diese Repräsentationen einfach zu übertragen. Eine Adjazenzmatrix kann einfach als Liste von Zeilen (die wiederum Listen sind) definiert werden. Eine Adjazenzliste ist entsprechend eine Liste von Nachbarschaften der jeweiligen Knoten. Eine „Nachbarschaft“ kann man nun wiederum als Liste darstellen. Um einen schnelleren Zugriff auf einen bestimmten Nachbarn zu gewährleisten ist es jedoch günstiger die Nachbarschaft eines Knotens in einem *dict*-Objekt zu speichern.

Wir wollen definieren einen Graphen mittels einer Klasse *Graph*:

```
1 class Graph(object):
2   def __init__(self,n):
3     self.vertices = []
4     self.numNodes = n
5     for i in range(0,n+1):
6       self.vertices.append({})
```

Wir legen uns schon bei der Initialisierung des Graphen auf dessen Größe fest und übergeben der *__init__*-Funktion die Anzahl n der Knoten im Graphen. Neben dem Attribut *numNodes*, enthält der Graph noch die Adjazenzliste *vertices*; jeder Eintrag dieser Adjazenzliste wird zunächst mit einer leeren Knotenmenge {} (in Python durch ein leeres Dictionary repräsentiert) initialisiert.

Listing 5.1 zeigt die Implementierung der wichtigsten Graphmethoden.

```
1 class Graph(object):
2   ...
3   def addEdge(self,i,j,weight=None):
4     self.vertices[i][j] = weight
5   def isEdge(self,i,j):
6     return j in self.vertices[i]
7   def G(self,i):
8     return self.vertices[i].keys()
9   def V(self):
10    return [i for i in range(0,self.numNodes+1)]
11  def E(self):
12    return [(i,j) for i in self.V() for j in self.G(i)]
```

Listing 5.1: *Implementierung der wichtigsten Graphmethoden.*

Die Methode *Graph.addEdge(i,j)* fügt dem Graphen eine Kante (i,j) hinzu – optional mit einem Gewicht *weight*; die Methode *Graph.isEdge(i,j)* testet, ob die Kante (i,j)

im Graphen enthalten ist; die Methode *Graph.G*(i) liefert die Liste der Nachbarn des Knotens i zurück. Und schließlich wird die Methode *Graph.V*() implementiert, die einfach die Liste aller Knoten zurückliefert und die Methode *Graph.E*(), die die Liste aller Kanten des Graphen zurückliefert.

Um nun etwa den Beispielgraphen in Abbildung 5.7 zu erzeugen, kann man die folgenden Anweisungen verwenden:

```
g2 = Graph(11)
for i,j in [  (1,2),(1,4),(1,5),(2,3),(3,6),(6,5),
              (6,9),(5,9),(5,8),(8,7),(8,11),(11,10)  ]:
  g2.addEdge(i,j)
```

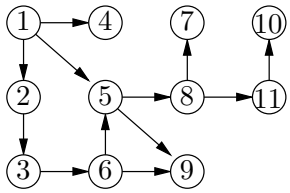


Abb. 5.7: *Ein Beispielgraph.*

Aufgabe 5.1

Erweitern Sie die Klasse *Graph* um die Methode *Graph.w*(i,j), die das Gewicht der Kante (i, j) zurückliefert (bzw. *None*, falls die Kante kein Gewicht besitzt).

Aufgabe 5.2

Erweitern Sie die Klasse *Graph* um die folgenden Methoden:

(a) Eine Methode *Graph.isPath*(*vs*), die eine Knotenliste *vs* übergeben bekommt und prüft, ob es sich hierbei um einen Pfad handelt.

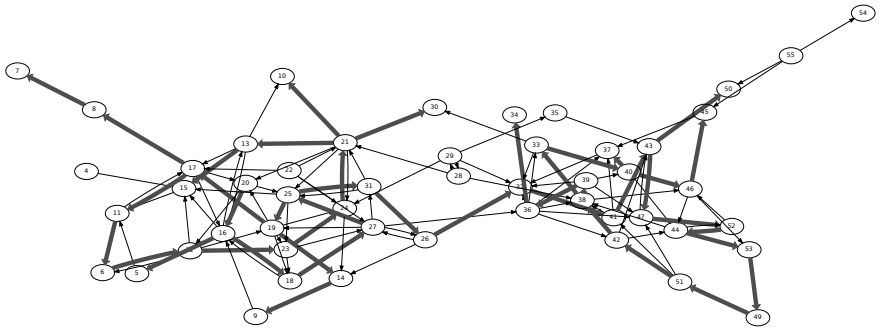
(b) Eine Methode *Graph.pathVal*(*vs*), die eine Knotenliste *vs* übergeben bekommt. Handelt es sich dabei um einen gültigen Pfad, so wird der „Wert“ dieses Pfades (d. h. die Summe der Gewichte der Kanten des Pfades) zurückgeliefert. Andernfalls soll der Wert ∞ (in Python: *float*(`'inf'`)) zurückgeliefert werden. Verwenden Sie hierbei das folgende „Gerüst“ und fügen Sie an der mit „ ... “ markierten Stelle die passende Listenkomprehension ein.

```
def pathVal( self,  xs ):
  if  len(xs)<2: return 0
  return sum( [...] )
```

Aufgabe 5.3

Schreiben Sie eine Klasse *GraphM*, die dieselbe Schnittstelle wie die Klasse *Graph* bereitstellt (also ebenfalls Methoden *addEdge*, *isEdge*, *G*, und die einen Graphen als Adjazenzmatrix implementiert.

5.2 Breiten- und Tiefensuche



Mit einer Breiten- bzw. Tiefensuche kann man einen Graphen in systematischer Weise durchlaufen. Viele Algorithmen verwenden als „Gerüst“ eine Breiten- oder Tiefensuche, wie etwa die in späteren Abschnitten behandelte Topologische Sortierung, oder das Finden von Zyklen in einem Graphen.
Obige Abbildung zeigt eine Tiefensuche durch einen größeren Beispielgraphen mit $|V| = 60$ Knoten.

5.2.1 Breitensuche

Queues

Für die Implementierung einer Breitensuche empfiehlt es sich, eine Warteschlange zu verwenden, auch im Deutschen oft als eine *Queue* bezeichnet. Eine Queue ist eine Datenstruktur, die überlicherweise die folgenden Operationen unterstützt:

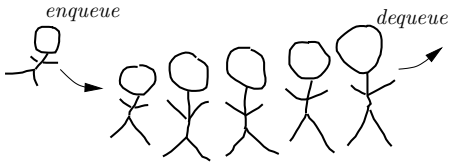


Abb. 5.8: *Eine Queue; neue Elemente (bzw. Leute) müssen sich „hinten“ einreihen; „vorne“ werden Elemente entnommen.*

1. Das Einfügen *enqueue*(x) eines Elementes x; **2.** das Entfernen *dequeue*() desjenigen Elementes, das sich am längsten in der Queue befindet; **3.** einen Test *isEmpty*() ob die Queue leer ist. Entscheidend ist die folgende Eigenschaft von Queues: Es wird immer dasjenige Element als Nächstes zur Bearbeitung aus der Queue entfernt, das sich am *längsten* in der Queue befindet, das also als erstes in die Queue eingefügt wurde. Eine Queue zeigt also das gleiche Verhalten, das jede Warteschlange im alltäglichen Leben auch zeigen sollte. Da das Element, das zeitlich gesehen als erstes eingefügt wurde auch

als erstes an der Reihe ist, wird eine Queue auch als FIFO (= first-in, first-out) Datenstruktur bezeichnet. Queues werden etwa bei der Abarbeitung von Druckaufträgen verwendet, oder auch bei der „gerechten“ Zuteilung sonstiger Ressourcen, wie Rechenzeit, Speicher usw.

Aufgabe 5.4

Implementieren Sie eine Klasse *Queue*, die die Operationen *enqueue*(x), *dequeue*() und *isEmpty* unterstützt.

Implementierung der Breitensuche. Eine Breitensuche erhält als Eingabe einen Graphen $G = (V, E)$ und einen Startknoten $s \in V$. Als Ergebnis der Breitensuche werden die Listen d und *pred* zurückgeliefert. Nach Ausführung der Breitensuche enthält der Eintrag $d[i]$ den „Abstand“ des Knotens i vom Startknoten s; der Eintrag $pred[i]$ enthält den Vorgänger zu Knoten i auf einem Breitensuche-Durchlauf durch den Graphen.

Listing 5.2 zeigt die Implementierung der Breitensuche (engl: Breadth First Search oder kurz: BFS).

```
1  def bfs(s,graph):
2    q = Queue()
3    d = [-1 if i≠s else 0 for i in range(graph.numNodes)]
4    pred = [None for _ in range(graph.numNodes)]]
5    v = s
6    while v ≠None:
7      for u in [u for u in graph.G(v) if d[u]==-1]:
8        d[u] = d[v] +1
9        pred[u] = v
10       q.enqueue(u)
11     if not q.isEmpty():
12       v = q.dequeue()
13     else:
14       v = None
15   return d,pred
```

Listing 5.2: *Implementierung der Breitensuche.*

Jeder Knoten v durchläuft hierbei in der **for**-Schleife in Zeile 7 diejenigen seiner Nachbarn, die bisher noch nicht besucht wurden, d. h. deren Distanzwert d noch den Wert -1 hat. Jeder der noch nicht besuchten Nachbarn wird durch Setzen des Distanzwertes und des *pred*-Arrays als besucht markiert. Schließlich „merkt“ sich die Breitensuche den Knoten u in der Queue, um zu einem späteren Zeitpunkt (nachdem die restlichen Nachbarn von v abgearbeitet wurden) die Breitensuche beim Knoten u fortzufahren. Nach Beendigung der **for**-Schleife gibt es keine Nachbarn von v mehr, die noch nicht

besucht wurden. Die Breitensuche holt sich nun den nächsten in der Queue vorgemerkten Knoten und fährt mit diesem fort. Sollte die Queue allerdings leer sein, so gibt es für die Breitensuche nichts mehr zu tun; der Algorithmus bricht ab.

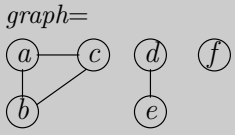
Nach Durchlauf der Breitensuche befindet sich in Eintrag $d[i]$ die Länge des kürzesten Pfades vom Startknoten s zum Knoten i und die Kantenmenge $\{(i,j) \mid pred[i] = j\}$ bildet einen Spannbaum des Graphen.

Abbildung 5.9 zeigt den Ablauf einer Breitensuche für den Beispielgraphen aus Abbildung 5.7.

Aufgabe 5.5

Verwenden Sie die Breitensuche, um alle Zusammenhangskomponenten eines Graphen zu bestimmen; implementieren Sie eine entsprechende Funktion *allComps*, die eine Liste aller Zusammenhangskomponenten zurückliefert. Eine Zusammenhangskomponenten soll hierbei wiederum als Menge (etwa repräsentiert als Liste oder *set*-Objekt) von Knoten repräsentiert sein, die die entsprechende Zusammenhangskomponente bilden. Beispiel:

```
>>> allComps(graph)
>>> [[a,b,c],[d,e],[f]]
```



5.2.2 Tiefensuche

Stacks

Für eine (iterative) Implementierung der Tiefensuche empfiehlt es sich einen *Stapelspeicher*, auch in der deutschen Literatur oft mit dem englischen Wort *Stack* bezeichnet, zu verwenden. Einen Stack kann man sich vorstellen als einen Stapel Papier auf einem Schreibtisch; jedes Papier bedeutet gewisse Arbeit, die durchzuführen ist. Kommt neue Arbeit hinzu, so legt man diese üblicherweise – wie in Abbildung 5.10 angedeutet – oben auf dem Stapel ab und will man ein neues Blatt bearbeiten, so entnimmt man dieses auch von oben.

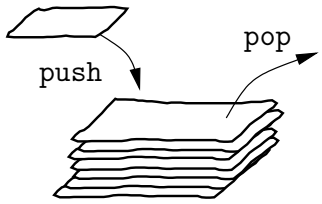
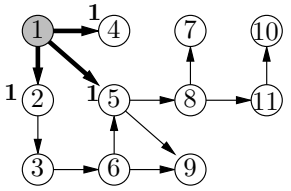


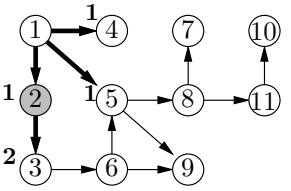
Abb. 5.10: *Ein Stapelspeicher; neue Elemente (bzw. Blätter) werden immer oben abgelegt und von oben entnommen.*

In der Informatik ist ein Stack eine Datenstruktur, die üblicherweise die folgenden Operationen unterstützt. **1.** Das Einfügen *push*(x) eines Elementes in einen Stack; **2.** Das Entnehmen *pop*() des obersten Elements; **3.** Der Test *isEmpty*(), ob der Stack leer ist. Entscheidend ist die folgende Eigenschaft von Stacks: Es wird immer dasjenige Element als Nächstes zur Bearbeitung vom Stack entfernt, das sich am kürzesten im Stack befindet, d. h. das als letztes in den Stack gelegt wurde. Aus diesem Grund wird diese Datenstruktur gerne als LIFO (= last-in, first-out) bezeichnet.



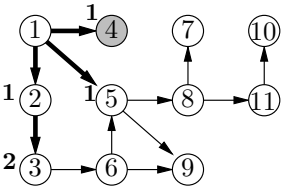
$v = 1$
unvisitedNeighb = [2,4,5]
q = -

(a)



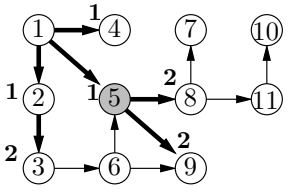
$v = 2$
unvisitedNeighb = [3]
q = ← 2 | 4 | 5 ←

(b)



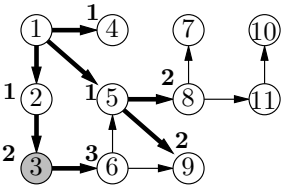
$v = 4$
unvisitedNeighb = []
q = ← 4 | 5 | 3 ←

(c)



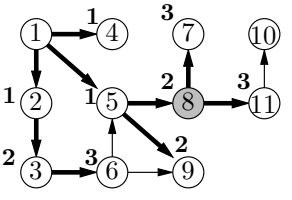
$v = 5$
unvisitedNeighb = [8,9]
q = ← 4 | 5 | 3 ←

(d)



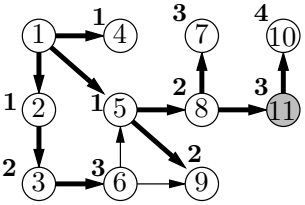
$v = 3$
unvisitedNeighb = [6]
q = ← 3 | 8 | 9 ←

(e)



$v = 8$
unvisitedNeighb = [7,11]
q = ← 8 | 9 | 7 | 11 ←

(f)



$v = 11$
unvisitedNeighb = [10]
q = ← 11 ←

(g)

. . . . . .

Abb. 5.9: *Ablauf einer Breitensuche durch den in Abbildung 5.7 dargestellten Beispielgraphen. Für jeden Durchlauf ist der Wert des aktuellen Knotens v, seine noch nicht besuchten Nachbarn unvisitedNeighb und der Wert der Warteschlange q angegeben. Die fett gezeichneten Kanten sind die in Liste pred aufgeführten Kanten, also Kanten, die im bisherigen Verlauf der Breitensuche gegangen wurden. Neben bisher besuchten Knoten sind die jeweiligen Werte der d-Liste aufgeführt, also der Liste, die im Laufe der Breitensuche für jeden Knoten den Abstandswert berechnet.*

Aufgabe 5.6

Implementieren Sie eine Klasse *Stack*, die die Operationen *push*(x), *pop*(x) und *isEmpty*() unterstützt.

Implementierung der Tiefensuche. Die Tiefensuche erhält als Eingabe einen Graphen $G = (V, E)$ und einen Startknoten $s \in V$. Als Ergebnis der Tiefensuche wird die Liste *pred* zurückgeliefert. Die Kantenmenge $\{(i, j) \mid pred[i] = j\}$ beschreibt hierbei den von der Tiefensuche gegangenen Weg durch den Graphen G.

Im Gegensatz zur Breitensuche, läuft die Tiefensuche ausgehend vom Startknoten einem Pfad solange als möglich nach; wenn es nicht mehr „weitergeht“ (weil der betreffende Knoten keine nicht besuchten Nachbarn mehr hat) so setzt die Tiefensuche zurück, d. h. sie läuft den gegangenen Pfad solange rückwärts, bis sie wieder einen Knoten findet, für den es noch etwas zu tun gibt. Dieses „Zurücksetzen“ nennt man in der Informatik auch *Backtracking*.

Listing 5.3 zeigt die Implementierung der Tiefensuche.

```
1  def dfs(s,graph):
2    pred = []
3    n = graph.numNodes
4    pred = [None for _ in range(n)]
5    st = Stack()
6    v = s
7    while True:
8      unvisitedNeighb = [u for u in graph.G(v) if pred[u]==None and u ≠s]
9      if unvisitedNeighb ≠ []:
10       u = unvisitedNeighb[0]
11       st.push(v)
12       pred[u] = v
13       v = u
14     elif not st.isEmpty():
15       v = st.pop()
16     else:
17       break
18   return pred
```

Listing 5.3: *Implementierung der Tiefensuche*

Zunächst werden die verwendeten Variablen *pred*, *st* und v initialisiert. Der eigentliche Algorithmus beginnt ab Zeile 7. In Zeile 8 werden zunächst die Nachbarn des aktuellen Knotens v gesucht, die noch nicht besucht wurden und in der Liste *unvisitedNeighb* gespeichert. Es gibt drei Fälle: **1.** Die Liste *unvisitedNeighb* enthält mindestens ein Element, d. h. es gibt einen noch nicht besuchten Nachbarn u von v. In diesem Fall wird v auf den Stack gelegt, in der Annahme, es könne zu einem späteren Zeitpunkt ausgehend von v noch mehr zu tun geben. Die Kante (v, u) wird anschließend zur „Menge“ *pred*

der durch die Tiefensuche gegangenen Kanten hinzugefügt; schließlich wird mit dem Knoten u fortgefahren. **2.** Die Liste *unvistedNeigb* ist leer, d. h. es gibt keinen noch nicht besuchten Nachbarn von v, d. h. ausgehend vom Knoten v gibt es für die Tiefensuche nichts mehr zu tun. Falls es noch auf dem Stack *st* hinterlegte „Arbeit“ gibt, wird diese vom Stack geholt. **3.** Falls sowohl die Liste *invisitedNeigb*, als auch der Stack leer ist, ist die Tiefensuche beendet und die **while**-Schleife wird verlassen.

Aufgabe 5.7

Es gibt eine entscheidende Ineffizienz in der in Listing 5.3 vorgestellten Implementierung der Tiefensuche: Obwohl in jedem Schleifendurchlauf der **while**-Schleife nur *ein einziger* noch nicht besuchter Nachbar von v zur weiteren Bearbeitung benötigt wird, wird in der Listenkomprehension in Zeile 8 immer die gesamte Menge der noch nicht besuchten Nachbarn berechnet.

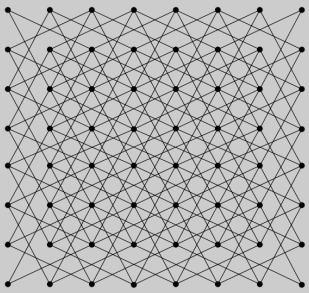
Verbessern sie die Implementierung der Tiefensuche, indem sie diese Ineffizienz entfernen.

Abbildung 5.11 zeigt den Ablauf einer Tiefensuche für den Beispielgraphen aus Abbildung 5.7.

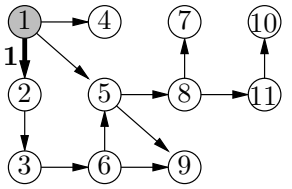
Die „nackte“ Tiefensuche liefert zwar keine eigentlich nützliche Information zurück, jedoch dient die Tiefensuche als „Gerüst“ für eine Vielzahl wichtiger Graphenalgorithmen, unter Anderem der topologischen Sortierung, der Suche nach Zyklen in einem Graphen oder der Auswertung als Bäume repräsentierter arithmetischer Ausdrücke.

Aufgabe 5.8

Das sog. Springerproblem besteht darin, auf einem sonst leeren $n \times n$ Schachbrett eine Tour für einen Springer zu finden, auf der dieser jedes Feld genau einmal besucht. Wir wählen zunächst besser $n < 8$ (andernfalls sind sehr lange Rechenzeiten zu erwarten). Finden Sie eine Lösung für das Springerproblem, indem sie wie folgt vorgehen:
1: Repräsentieren Sie das Problem als Graph. Jedes Feld des Schachbretts sollte einen Knoten darstellen und jeder mögliche Zug sollte als Kante zwischen zwei Knoten dargestellt werden; sie können entweder die Kanten von Hand eintragen oder ein Programm schreiben, das das erledigt. **2:** Verwenden Sie eine Variante der Tiefensuche, die verbietet, dass ein Knoten mehr als einmal besucht wird und finden Sie damit eine Lösung des Springerproblems.

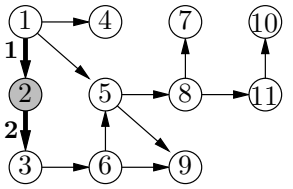


Ein Graph der alle möglichen Züge eines Springers auf einem 8×8 Schachbrett repräsentiert.



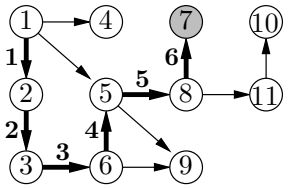
$v = 1$
unvisitedNeighb = [2,4,5]
s = -

(a)



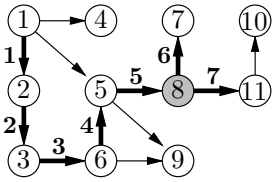
$v = 2$
unvisitedNeighb = [3]
s = | 1 |

(b)



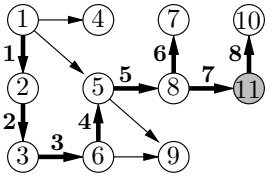
$v = 7$
unvisitedNeighb = []
s = | 1 | 2 | 3 | 6 | 5 | 8 |

(c)



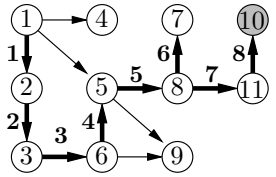
$v = 8$
unvisitedNeighb = [11]
s = | 1 | 2 | 3 | 6 | 5 | 8 |

(d)



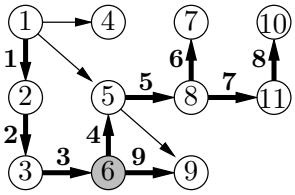
$v = 11$
unvisitedNeighb = [10]
s = | 1 | 2 | 3 | 6 | 5 | 8 | 11 |

(e)



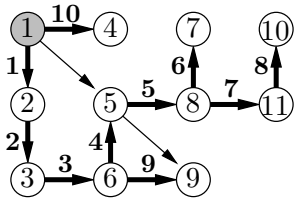
$v = 10$
unvisitedNeighb = []
s = | 1 | 2 | 3 | 6 | 5 | 8 | 11 |

(f)



$v = 6$
unvisitedNeighb = [9]
s = | 1 | 2 | 3 | 6 |

(g)



$v = 1$
unvisitedNeighb = [4]
s = | 1 |

(h)

Abb. 5.11: *Ablauf einer Tiefensuche durch den in Abbildung 5.7 dargestellten Beispielgraphen. Für jede Situation ist der Wert des aktuellen Knotens v, seine noch nicht besuchten Nachbarn unvisitedNeighb und der Wert des Stacks s angegeben. Die fett gezeichneten Kanten sind die in Liste pred aufgeführten Kanten, also Kanten, die im bisherigen Verlauf der Tiefensuche gegangen wurden. Der Übersichtlichkeit halber wurden die Kanten in der von der Tiefensuche gegangenen Reihenfolge nummeriert – diese Nummerierung erfolgt lediglich der Anschaulichkeit halber; sie wird im Algorithmus selbst nicht protokolliert. Man beachte, dass einige Schritte in der Darstellung übersprungen wurden, und zwar drei Schritte zwischen 5.11(b) und 5.11(c), vier Schritte zwischen 5.11(f) und 5.11(g) und vier Schritte zwischen 5.11(g) und 5.11(h).*

Aufgabe 5.9

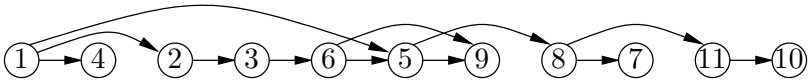
Statt explizit einen Stack zu verwenden, lässt sich die Tiefensuche elegant rekursiv implementieren. Implementieren Sie eine rekursive Variante *dfsRek*, des in Listing 5.3 gezeigten Algorithmus *dfs*.

5.2.3 Topologische Sortierung

Eine *topologische Sortierung* ist eine Anordnung der Knoten eines DAG, d. h. eines gerichteten azyklischen Graphen $G = (V, E)$, so dass für jede Kante $(i, j) \in E$ gilt, dass Knoten j nach Knoten i angeordnet ist. DAGs werden oft verwendet, wenn man eine Rangordnungen zwischen bestimmten Elementen oder Ereignissen darstellen will. Beispielsweise ließe sich der Graph aus Abbildung 5.7 auf Seite 151 topologisch sortieren durch die folgende Anordnung seiner Knoten:

$$1, 4, 2, 3, 6, 5, 9, 8, 7, 11, 10$$

Der Graph ließe sich dann entsprechend so zeichnen, dass jede Kante von links nach rechts verläuft:



Man kann eine topologische Sortierung folgendermaßen einfach berechnen: Man beginnt eine Tiefensuche durch einen Graphen mit einem Knoten, der keinen Vorgänger besitzt; solch ein Knoten muss existieren, wenn der Graph keinen Zyklus besitzt. Sobald bei solch einem Tiefensuche-Durchlauf ein bestimmter Knoten v „abgeschlossen“ wurde, füge diesen mittels *append* hinten an eine Liste an. Oder genauer formuliert: Sobald für einen Knoten v der während der Tiefensuche in Listing 5.3 berechneten Liste *unvisitedNeighb* (Zeile 8) leer ist, wird dieser Knoten v an die eine topologische Anordnung der Knoten repräsentierende Ergebnisliste hinten angehängt. Nach der Tiefensuche enthält diese Ergebnisliste die für die topologische Sortierung erforderliche Rangordnung. Dies kann ganz einfach folgendermaßen implementiert werden (in den mit ... markierten Bereichen befindet sich Code der identisch zu dem Code der Tiefensuche aus Listing 5.3 ist):

```
1  def topSort(s, graph): #s: Knoten ohne Vorgänger
2    topLst = []
3    ...
4    while True:
5      ...
6      if unvisitedNeighb ≠ []:
7        ...
8      elif not st.isEmpty():
9        topLst.append(v)
10       v = st.pop()
```

```
11      else:
12          topLst.append(v)
13          break
14      topLst.reverse()
15  return topLst
```

Listing 5.4: *Berechnung einer topologischen Sortierung eines DAG. Der Startknoten s muss hierbei so gewählt sein, dass s keinen Vorgänger besitzt.*

Der **elif**- und **else**-Zweig wird gegangen, wenn der betreffende Knoten v abgeschlossen ist; genau zu diesem Zeitpunkt wird v in die Liste *topLst* der topologisch sortierten Knoten eingefügt.

Aufgabe 5.10

(a) Welche Anordnung der Knoten liefert der in Listing 5.4 dargestellte Algorithmus als topologische Sortierung?

(b) Versuchen Sie herauszufinden, wie viele verschiedene topologische Sortierungen es für den in Abbildung 5.7 dargestellten Graphen gibt.

Aufgabe 5.11

Beim Anziehen von Kleidungsstücken müssen manche Teile unbedingt vor anderen angezogen werden. Die folgenden Beziehungen sind vorgegeben:

- Das *Unterhemd* vor dem *Pullover*
- Die *Unterhose* vor der *Hose*
- Den *Pullover* vor dem *Mantel*
- Die *Hose* vor dem *Mantel*
- Die *Hose* vor den *Schuhen*
- Die *Socken* vor den *Schuhen*

(a) Modellieren Sie diese Abhängigkeiten als Graphen.

(b) Nummerieren Sie die Knoten so, dass sich die daraus ergebende Rangordnung der Knoten eine topologische Sortierung darstellt – gibt hier mehrere Lösungen.

(c) Bestimmen Sie diejenige topologische Sortierung, die sich durch Ausführung von dem in Listing 5.4 gezeigten Algorithmus ergibt.

Aufgabe 5.12

Die topologische Sortierung erwartet als Eingabe einen Knoten, der keinen Vorgänger besitzt. Implementieren Sie eine Funktion *startNodes*(*graph*), die alle Knoten des Graphen *graph* zurückliefert, die keinen Vorgängerknoten besitzen.

Aufgabe 5.13

Der in Listing 5.4 gezeigte Algorithmus funktioniert nur auf zusammenhängenden DAGs. Erweitern Sie den Algorithmus so, dass er auch auf nicht zusammenhängenden DAGs funktioniert.

Aber warum liefert dieser Algorithmus eine topologische Sortierung? Wir müssen dazu Folgendes zeigen: Befindet sich eine Kante (u,v) im Graphen $G = (V, E)$, so wird zuerst *topLst.append*(v) und danach erst *topLst.append*(u) ausgeführt. Durch die Anweisung *topLst.reverse*() in Zeile 14 in Listing 5.4 werden dann schließlich u und v in die richtige Reihenfolge gebracht – nämlich u *vor* v. Warum also wird *topLst.append*(v) vor *topLst.append*(u) ausgeführt?

Wird im Rahmen der Tiefensuche der Knoten u erstmalig betrachtet, so gibt es zwei Möglichkeiten. **1.** Es gilt: v **in** *unvisitedNeighb*. In diesem Fall wird u auf den Stack gelegt (Zeile 11) und die Tiefensuche mit dem Knoten v weiter durchlaufen, und zwar so lange, bis v abgeschlossen wird und keine nicht besuchten Nachbarn mehr besitzt (d. h. *unvisitedNeighb* == [] gilt) und somit *topLst.append*(v) ausgeführt wird. Erst danach wird der Knoten u fertig bearbeitet und somit folgt erst danach die Anweisung *topLst.append*(u).

2. Es gilt: v **not in** *unvisitedNeighb*. Der Knoten v wurde also schon besucht. Kann es dann sein, dass v noch nicht abgeschlossen ist (und folglich *topLst.append*(u) *vor* *topList.append*(v) ausgeführt werden würde)? Wäre dem so, dann würde sich in diesem Fall v noch im Stack *st* befinden, d. h. *st* hätte folgendes Aussehen:

$$[\ldots, v, \ldots, u]$$

Folglich müsste es einen Pfad von v nach u geben. Zusammen mit der Kante (u,v) würde dies einen Kreis ergeben, was aber nach Voraussetzung (es handelt sich um einen DAG, also einen kreisfreien Graphen) unmöglich ist.

5.3 Kürzeste Wege

Eine der offensichtlichsten Anwendungen der Graphentheorie besteht in der Aufgabe, die kürzest möglichen Wege in einem kantenbewerteten Graphen $G = (V, E)$ mit der Gewichtsfunktion $w : E \to \mathbb{R}^+$ zwischen zwei Knoten zu berechnen. Die Funktion w ordnet jeder Kante eine (positive) Zahl zu; so kann man etwa den Abstand zwischen zwei Städten abbilden. Es ist nicht zuletzt der Effizienz und Eleganz des Dijkstra-Algorithmus zu verdanken, dass die Berechnung des kürzesten Weges zwischen zwei

Ortschaften durch ein Navigationssystem oder ein Online-Routenplanungssystem so schnell und unkompliziert möglich ist.

Wir stellen in diesem Abschnitt zwei unterschiedliche Algorithmen zur Berechnung der kürzesten Wege in einem Graphen $G = (V, E)$ vor: Zum Einen den Dijkstra-Algorithmus, der die kürzesten Wege ausgehend von einem bestimmten Knoten $u \in V$ zu allen anderen Knoten im Graphen berechnet; zum Anderen den Warshall-Algorithmus, der die kürzesten Wege zwischen allen Knotenpaaren $u, v \in V$ berechnet – in der englischsprachigen Literatur wird diese Aufgabe auch als „All Pairs Shortest Paths“ bezeichnet.

5.3.1 Der Dijkstra-Algorithmus

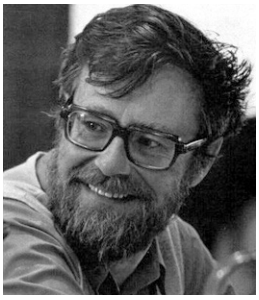


Abb. 5.12: *Edsger Dijkstra (1930 - 2002).*

Will man einen kürzesten Pfad von einem Knoten u zu einem anderen Knoten v berechnen, so könnte dieser Pfad im Allgemeinen alle anderen Knoten berühren. Es macht daher durchaus Sinn, für die Lösung dieses Problems einen Algorithmus zu entwerfen, der die kürzesten Wege von Knoten u zu *jedem anderen* Knoten des Graphen berechnet. Der sog. Dijkstra-Algorithmus, entdeckt von dem niederländischen Informatik-Pionier Edsger Dijkstra, ist ein effizienter Algorithmus der alle von u ausgehenden kürzesten Wege berechnet. Dijkstra war unter Anderem auch der Wegbereiter der strukturierten Programmierung und der parallelen Programmierung (er verwendete erstmals Semaphoren, eine spezielle Datenstruktur, die dazu eingesetzt wird, parallel laufende Prozesse zu synchronisieren).

Der Dijkstra Algorithmus ist ein typischer sog. *Greedy-Algorithmus*. Greedy-Algorithmen schlagen zum Finden einer optimalen Lösung eine einfache Vorgehensweise ein: Es wird in einem Schritt immer nur eine bestimmte Teillösung berechnet. Um die Teillösungen zu erweitern und sich dadurch einen Schritt Richtung Gesamtlösung zu bewegen, werden nur diejenigen Möglichkeiten in Betracht gezogen, die „lokal“ zum jeweiligen Zeitpunkt am günstigsten erscheinen. Nicht immer führt die Strategie eines Greedy-Algorithmus zur Berechnung des Optimums – jedoch im Falle des Dijkstra-Algorithmus schon.

Dies ist genau die Vorgehensweise des Dijkstra-Algorithmus zum Finden der kürzesten Wege ausgehend von einem bestimmten Knoten u in einem Graphen $G = (V, E)$. In jedem Schritt wird immer derjenige noch nicht fertig bearbeitete Knoten betrachtet, der den momentan geringsten Abstandswert zu u hat.

Der Dijkstra-Algorithmus liefert als Ergebnis die Abstände $l[v]$ aller Knoten $v \in V$ zu Knoten u und zusätzlich in Form der Menge F alle Kanten, aus denen die kürzesten Wege bestehen. In der Menge W merkt sich der Algorithmus die noch zu bearbeitenden Knoten; in jedem Durchlauf des Dijkstra-Algorithmus wird ein Knoten aus W entfernt und zwar immer derjenige mit dem momentan geringsten Abstand zu u. Nach $|V|$ vielen Durchläufen hat der Algorithmus also alle kürzesten Wege berechnet. In jedem der $|V|$ Durchläufe wird immer derjenige Knoten v als Nächstes bearbeitet, der den momentan

geringsten Abstand $l[v]$ vom Startknoten u besitzt – genau dieser Schritt macht den Algorithmus zu einem Greedy-Algorithmus. In diesem Schritt wird immer jeweils die gesamte Nachbarschaft $\Gamma(v)$ des Knotens v durchlaufen und versucht die Abstandswerte der Nachbarn zu verbessern. Hierbei wird der Abstandswert eines Nachbarn $v' \in \Gamma(v)$ genau dann angepasst, falls entweder noch kein Abstandswert berechnet wurde oder falls

$$l[v] + w(v, v') < l[v']$$

gilt, d. h. falls ein Weg über v zu v' kürzer ist als der bisher berechnete Weg.

Listing 5.5 zeigt die Implementierung des Dijkstra-Algorithmus.

```
1  def dijkstra(u,graph):
2    n = graph.numNodes
3    l = {u:0} ; W = graph.V()
4    F = [] ; k = {}
5    for i in range(n):
6      lv,v = min([ (l[node],node) for node in l  if  node in W ])
7      W.remove(v)
8      if v≠u: F.append(k[v])
9      for neighb in  filter (lambda x:x in W, graph.G(v)):
10       if neighb not in l or l[v] + graph.w(v,neighb) < l[neighb]:
11         l[neighb] = l[v]  + graph.w(v,neighb)
12         k[neighb] = (v,neighb)
13   return l,F
```

Listing 5.5: *Der Dijkstra-Algorithmus*

Die **for**-Schleife ab Zeile 5 implementiert die $|V|$ vielen Durchläufe. In Zeile 6 wird bestimmt, welcher Knoten in dem aktuellen Durchlauf bearbeitet wird, nämlich derjenige Knoten v, mit minimalem Abstandswert $l[v]$. Dieser Knoten wird aus der Menge W der zu bearbeitenden Knoten gelöscht (Zeile 7) und die entsprechende aus Richtung u kommende Kante $k[v]$ zur Kantenmenge F hinzugefügt. In Zeile 12 beginnt die **for**-Schleife, die die Nachbarschaft des Knotens v durchläuft und alle suboptimalen Abstandswerte anpasst. Für jeden angepassten Nachbarknoten *neighb* merkt sich der Algorithmus zusätzlich in Zeile 12 die Kante $(v, neighb)$, die zu dieser Anpassung führte; diese Kante wird später eventuell (falls diese Anpassung später nicht noch weiter optimiert wird) zur Kantenmenge F der kürzesten Wege hinzugefügt.

Abbildung 5.13 zeigt den Ablauf des Dijkstra-Algorithmus für einen gewichteten ungerichteten Beispielgraphen.

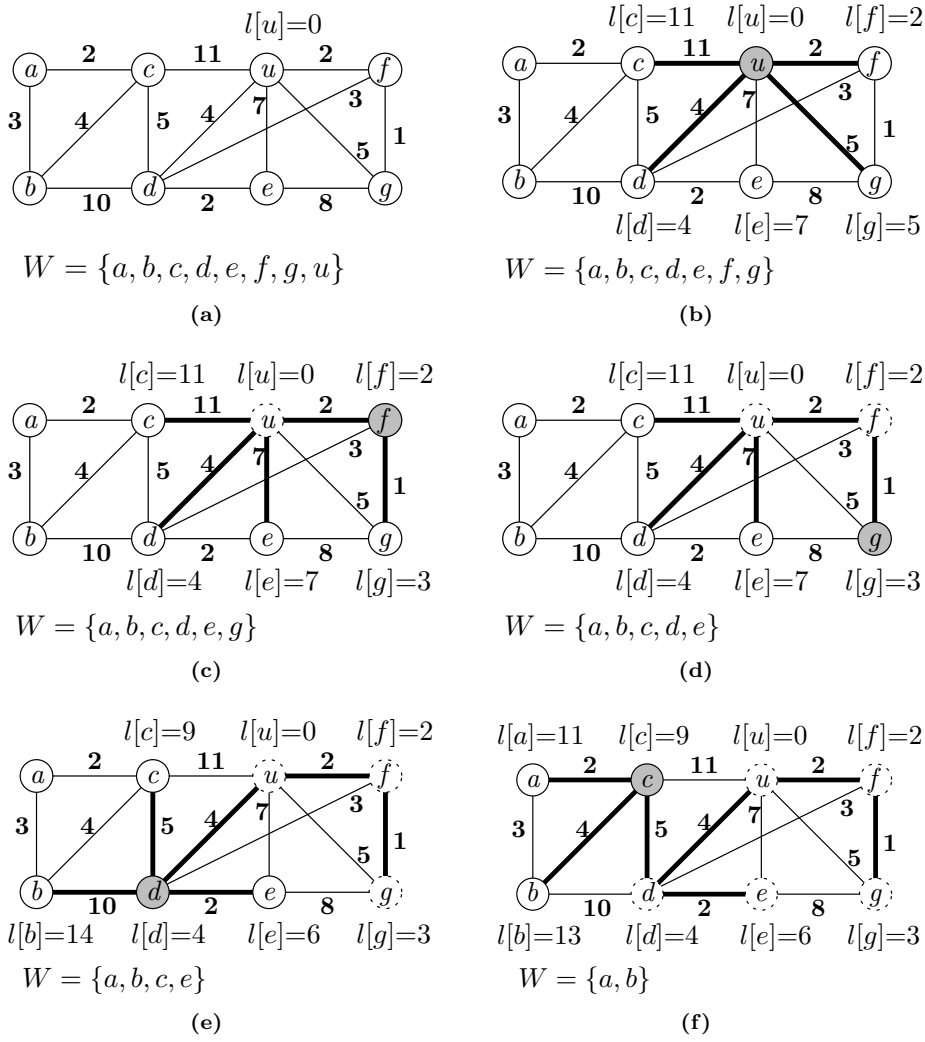


Abb. 5.13: *Ablauf des Dijkstra-Algorithmus auf einem gewichteten ungerichteten Graphen. Abbildung 5.13(a) zeigt die Ausgangssituation: Es existiert nur für den Startknoten ein Abstandswert von 0. Im ersten Schritt, gezeigt in Abbildung 5.13(b), wird der Startknoten u bearbeitet da dieser initial den kleinsten Abstandswert hatte. Die Nachbarschaft von u wird durchlaufen, die Abstandswerte aller Nachbarn werden angepasst und die entsprechenden Kanten in k vorgemerkt. Diese Kanten sind in den Abbildungen immer fett gezeichnet. Im nächsten Schritt (Abbildung 5.13(c)) wird, da $l[f]$ minimal ist, der Knoten f bearbeitet. Die Nachbarschaft des Knotens f wird also durchlaufen; hierbei wird der Abstandswert des Knotens g angepasst, denn $l[f] + w(f,g)$ ist kleiner als $l[g]$; die Abstandswerte der restlichen Nachbarn bleiben gleich. Als Nächstes (Abbildung 5.13(d)) wird der Knoten g bearbeitet, da $l[g]$ minimal ist, usw.*

Aufgabe 5.14

In jedem der $|V|$ vielen Durchläufe des Dijkstra-Algorithmus muss der Knoten mit minimalem Abstandswert l bestimmt werden. Dies geschieht in Algorithmus 5.5 mittels der *min*-Anweisung in Zeile 9.

(a) Welche Laufzeit hat diese *min*-Anweisung?

(b) Statt das Minimum aus einer Liste zu bestimmen ist es i. A. effizienter ein Heap-Datenstruktur zu verwenden und mittels *minExtract* das Minimum zu extrahieren. Welche Laufzeit hätte das Finden des Knotens mit minimalem Abstandswert, falls statt einer einfachen Liste eine Heap-Datenstruktur verwendet wird?

(c) Geben sie eine Python-Implementierung des Dijkstra-Algorithmus an, zum Finden des minimalen Abstandswertes Heaps verwendet.

5.3.2 Der Warshall-Algorithmus

Gegeben sei, genau wie beim Dijkstra-Algorithmus, ein kantenbewerteter Graph $G = (V, E)$ mit der Gewichtsfunktion $w : E \to \mathbb{R}^+$. Der Warshall-Algorithmus berechnet die kürzesten Wege zwischen *allen* Knotenpaaren in G. Wir gehen von einer Knotenmenge $V = \{1, \ldots, n\}$ aus.

Entscheidend für den Warshall-Algorithmus ist folgende Überlegung. Man betrachtet zunächst kürzeste Wege, für die gewisse Einschränkungen gelten. Diese „Einschränkungen“ sollten optimalerweise zwei Eigenschaften erfüllen: **1:** Die Berechnung der kürzesten Wege, für die diese Einschränkungen (die wir gleich genau erläutern) gelten, sollte sinnvollerweise einfacher sein, als die Berechnung der kürzesten Wege ohne Einschränkungen. **2:** Es sollte möglich sein, diese Einschränkungen schrittweise zu entfernen, bis man schließlich die kürzesten Wege (für die gar keine Einschränkungen mehr gelten) erhält.

Wir sehen nun diese Einschränkungen im Falle des Warshall-Algorithmus aus? Anfänglich berechnen wir die kürzesten Wege, die keine Zwischenknoten enthalten (also nur Direktverbindungen); diese „Berechnung“ ist sehr einfach, denn diese Direktverbindungen sind in Form der Adjazenzmatrix des Graphen schon vorhanden. Im nächsten Schritt berechnen wir die kürzesten Wege, deren Zwischenknoten aus der Knotenmenge $\{1\}$ kommen. Im folgenden Schritt berechnen wir, aus den im vorigen Schritt berechneten Informationen, die kürzesten Wege, deren Zwischenknoten aus der Knotenmenge $\{1, 2\}$ kommen, usw. Im letzten Schritt berechnen wir schließlich die kürzesten Wege, deren Zwischenknoten aus der Knotenmenge $\{1, \ldots, n\}$ kommen, d. h. für diese kürzesten Wege gibt es keine Einschränkungen mehr. In diesem letzten Schritt werden also die gesuchten kürzesten Wege berechnet.

Wir müssen uns nur noch überlegen, wie man vom $(k-1)$-ten Schritt zum k-ten Schritt „kommen“ kann, d. h. wie man aus dem kürzesten Pfad zwischen Knoten $i \in V$ und Knoten $j \in V$, dessen innere Knoten ausschließlich aus der Knotenmenge $\{1, \ldots, k-1\}$ kommen, den kürzesten Pfad zwischen i und j berechnen kann, dessen innere Knoten aus der Knotenmenge $\{1, \ldots, k\}$ kommen. Bei der Konstruktion dieser Berechnung ist es sinnvoll zwei Fälle zu unterschieden.

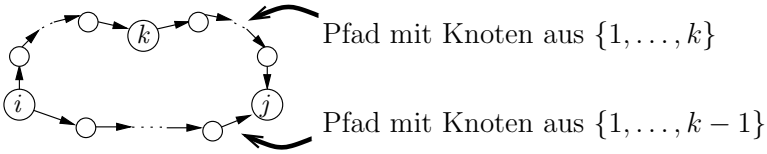


Abb. 5.14: *Darstellung der beiden Möglichkeiten für die Konstruktion eines kürzesten Pfades zwischen Knoten i und Knoten j der ausschließlich Knoten aus $\{1,\ldots k\}$ enthält. Entweder enthält dieser Pfad tatsächlich k als inneren Knoten, oder solch ein Pfad enthält den Knoten k nicht. Der Warshall-Algorithmus wählt in jedem Schritt immer den kürzeren dieser beiden möglichen Pfade.*

1: Der kürzeste Pfad zwischen i und j mit inneren Knoten aus $\{1,\ldots,k\}$ enthält den inneren Knoten k *nicht*. In diesem Fall gilt einfach, dass der kürzeste Pfad zwischen Knoten i und Knoten j mit inneren Knoten aus $\{1,\ldots,k-1\}$ gleich dem kürzesten Pfad zwischen i und j mit inneren Knoten aus $\{1,\ldots,k\}$ ist.
2: Der kürzeste Pfad zwischen i und j mit inneren Knoten aus $\{1,\ldots k\}$ enthält den inneren Knoten k; dieser setzt sich zusammen aus dem kürzesten Pfad von i nach k mit inneren Knoten aus $\{1,\ldots,k-1\}$ und dem kürzesten Pfad von k nach j mit inneren Knoten aus $\{1,\ldots,k-1\}$. Abbildung 5.14 veranschaulicht diesen Sachverhalt graphisch.

Wir bezeichnen mit $W_k[i,j]$ die Länge des kürzesten Pfads zwischen Knoten i und Knoten j mit inneren Knoten ausschließlich aus $\{1,\ldots,k\}$. Dann gilt nach den vorigen Überlegungen also folgende Beziehung:

$$W_k[i,j] := \min\{\ W_{k-1}[i,j],\ W_{k-1}[i,k] + W_{k-1}[k,j]\ \} \tag{5.1}$$

Wir sind also in der Lage W_k aus W_{k-1} zu berechnen. Die gewünschte Lösung, also alle kürzesten Wege, erhalten wir durch Berechnung von W_n, das bzgl. der inneren Knoten eines jeden Pfades keine Beschränkung mehr auferlegt. Wir beginnen die Berechnungen mit der Matrix W_0, die nichts anderes ist als die Adjazenzmatrix des Graphen G. Es ergibt sich also folgender Algorithmus:

```
1 def warshall(graph):
2   n = graph.numNodes+1
3   W = [[graph.w(i,j) for j in graph.V()] for i in graph.V() ] # W_0
4   for k in range(1,n): # Berechnung von W_k
5     for i in range(1,n):
6       for j in range(1,n):
7         W[i][j] = min( W[i][j] , W[i][k]+W[k][j] )
8   return W
```

Listing 5.6: *Implementierung des Warshall-Algorithmus*

Die geschachtelte Listenkomprehension in Zeile 3 erzeugt zunächst die Matrix W_0, also die Adjazenzmatrix von *graph*. Wichtig zu wissen ist hier, dass die Methode $V()$ der Klasse *Graph* die Liste der im Graph vorhandenen Knoten zurückliefert; die Methode $w(i,j)$ der Klasse *Graph* muss so implementiert sein, dass $graph.w(i,i)$ den Wert

0 zurückliefert (der Abstand eines Knotens i zu sich selbst ist sinnvollerweise 0) und *graph.w*(i, j) den Wert ∞ zurückliefert (in Python i. A. repräsentiert durch den speziellen Wert *inf*[1]), falls $(i, j) \notin E$; in allen anderen Fällen soll *graph.w*(i, j) das Gewicht der Kante (i, j) zurückliefern. Die Matrix W wird nun in n-1 Schleifendurchläufen schrittweise erweitert. Zeile 7 entspricht einer direkten Umsetzung der Formel (5.1). Der Algorithmus liefert in Zeile 8 die Matrix W_n in Form der Variablen W zurück; $W[i][j]$ enthält dann die Länge des kürzesten Weges von Knoten i zu Knoten j. Abbildung 5.15 zeigt die Zwischenergebnisse des Warshall-Algorithmus, d. h. die Matrizen W_k für die Berechnung der kürzesten Wege eines Beispielgraphen.

Aufgabe 5.15

Implementieren Sie eine Methode $w(i, j)$ der Klasse *Graph* in der für den Warshall-Algorithmus erforderlichen Weise.

Aufgabe 5.16

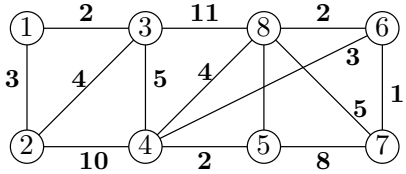
Die *transitive Hülle* eines gerichteten Graphen $G = (V, E)$ ist definiert als die Matrix $H = (h_{ij})$ mit

$$h_{ij} = \begin{cases} 1, \text{ Falls es einen gerichteten Pfad von } i \text{ nach } j \text{ in } G \text{ gibt} \\ 0, \text{ sonst} \end{cases}$$

Implementieren Sie eine Funktion *transHuelle*(*graph*) die die transitive Hülle des Graphen *graph* als Ergebnis zurückliefert.
Tipp: Sie können *transHuelle* relativ einfach dadurch programmieren, indem sie *warshall* an geeigneter Stelle etwas modifizieren.

Die Laufzeit des Warshall-Algorithmus ist aus offensichtlichen Gründen $O(|V|^3)$, denn die $|V| \times |V|$ große Adjazenzmatrix muss genau $|V|$ mal durchlaufen werden.

[1] Dieser spezielle Wert *inf* kann in Python durch den Aufruf *float*(`'inf'`) erzeugt werden; dies sollte in den meisten Python-Installationen möglich sein; es ist jedoch möglich, dass ältere Python-Versionen (Versionsnummer ¡ 2.4) diesen speziellen Wert noch nicht unterstützen.



$$\begin{bmatrix} 0 & 3 & 2 & \infty & \infty & \infty & \infty & \infty \\ 3 & 0 & 4 & 10 & \infty & \infty & \infty & \infty \\ 2 & 4 & 0 & 5 & \infty & \infty & \infty & 11 \\ \infty & 10 & 5 & 0 & 2 & 3 & \infty & 4 \\ \infty & \infty & \infty & 2 & 0 & \infty & 8 & 7 \\ \infty & \infty & \infty & 3 & \infty & 0 & 1 & 2 \\ \infty & \infty & \infty & \infty & 8 & 1 & 0 & 5 \\ \infty & \infty & 11 & 4 & 7 & 2 & 5 & 0 \end{bmatrix}$$
$k = 1$

$$\begin{bmatrix} 0 & 3 & 2 & \infty & \infty & \infty & \infty & \infty \\ 3 & 0 & 4 & 10 & \infty & \infty & \infty & \infty \\ 2 & 4 & 0 & 5 & \infty & \infty & \infty & 11 \\ \infty & 10 & 5 & 0 & 2 & 3 & \infty & 4 \\ \infty & \infty & \infty & 2 & 0 & \infty & 8 & 7 \\ \infty & \infty & \infty & 3 & \infty & 0 & 1 & 2 \\ \infty & \infty & \infty & \infty & 8 & 1 & 0 & 5 \\ \infty & \infty & 11 & 4 & 7 & 2 & 5 & 0 \end{bmatrix}$$
$k = 2$

$$\begin{bmatrix} 0 & 3 & 2 & \mathbf{13} & \infty & \infty & \infty & \infty \\ 3 & 0 & 4 & 10 & \infty & \infty & \infty & \infty \\ 2 & 4 & 0 & 5 & \infty & \infty & \infty & 11 \\ \mathbf{13} & 10 & 5 & 0 & 2 & 3 & \infty & 4 \\ \infty & \infty & \infty & 2 & 0 & \infty & 8 & 7 \\ \infty & \infty & \infty & 3 & \infty & 0 & 1 & 2 \\ \infty & \infty & \infty & \infty & 8 & 1 & 0 & 5 \\ \infty & \infty & 11 & 4 & 7 & 2 & 5 & 0 \end{bmatrix}$$
$k = 3$

$$\begin{bmatrix} 0 & 3 & 2 & \mathbf{7} & \infty & \infty & \infty & \mathbf{13} \\ 3 & 0 & 4 & \mathbf{9} & \infty & \infty & \infty & \mathbf{15} \\ 2 & 4 & 0 & 5 & \infty & \infty & \infty & 11 \\ \mathbf{7} & \mathbf{9} & 5 & 0 & 2 & 3 & \infty & 4 \\ \infty & \infty & \infty & 2 & 0 & \infty & 8 & 7 \\ \infty & \infty & \infty & 3 & \infty & 0 & 1 & 2 \\ \infty & \infty & \infty & \infty & 8 & 1 & 0 & 5 \\ \mathbf{13} & \mathbf{15} & 11 & 4 & 7 & 2 & 5 & 0 \end{bmatrix}$$
$k = 4$

$$\begin{bmatrix} 0 & 3 & 2 & 7 & \mathbf{9} & \mathbf{10} & \infty & \mathbf{11} \\ 3 & 0 & 4 & 9 & \mathbf{11} & \mathbf{12} & \infty & \mathbf{13} \\ 2 & 4 & 0 & 5 & \mathbf{7} & \mathbf{8} & \infty & \mathbf{9} \\ 7 & 9 & 5 & 0 & 2 & 3 & \infty & 4 \\ \mathbf{9} & \mathbf{11} & \mathbf{7} & 2 & 0 & \mathbf{5} & 8 & \mathbf{6} \\ \mathbf{10} & \mathbf{12} & \mathbf{8} & 3 & \mathbf{5} & 0 & 1 & 2 \\ \infty & \infty & \infty & \infty & 8 & 1 & 0 & 5 \\ \mathbf{11} & \mathbf{13} & \mathbf{9} & 4 & \mathbf{6} & 2 & 5 & 0 \end{bmatrix}$$
$k = 5$

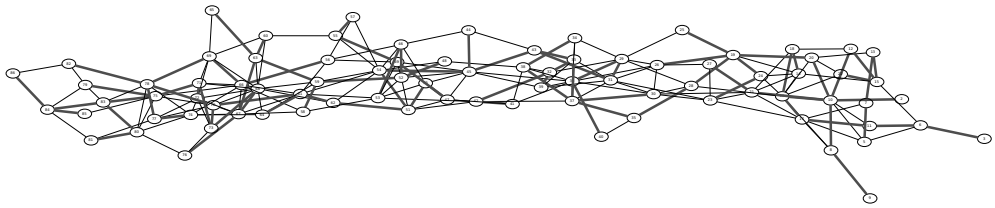
$$\begin{bmatrix} 0 & 3 & 2 & 7 & 9 & 10 & \mathbf{17} & 11 \\ 3 & 0 & 4 & 9 & 11 & 12 & \mathbf{19} & 13 \\ 2 & 4 & 0 & 5 & 7 & 8 & \mathbf{15} & 9 \\ 7 & 9 & 5 & 0 & 2 & 3 & \mathbf{10} & 4 \\ 9 & 11 & 7 & 2 & 0 & 5 & 8 & 6 \\ 10 & 12 & 8 & 3 & 5 & 0 & 1 & 2 \\ \mathbf{17} & \mathbf{19} & \mathbf{15} & \mathbf{10} & 8 & 1 & 0 & 5 \\ 11 & 13 & 9 & 4 & 6 & 2 & 5 & 0 \end{bmatrix}$$
$k = 6$

$$\begin{bmatrix} 0 & 3 & 2 & 7 & 9 & 10 & \mathbf{11} & 11 \\ 3 & 0 & 4 & 9 & 11 & 12 & \mathbf{13} & 13 \\ 2 & 4 & 0 & 5 & 7 & 8 & \mathbf{9} & 9 \\ 7 & 9 & 5 & 0 & 2 & 3 & \mathbf{4} & 4 \\ 9 & 11 & 7 & 2 & 0 & 5 & \mathbf{6} & 6 \\ 10 & 12 & 8 & 3 & 5 & 0 & 1 & 2 \\ \mathbf{11} & \mathbf{13} & \mathbf{9} & \mathbf{4} & \mathbf{6} & 1 & 0 & \mathbf{3} \\ 11 & 13 & 9 & 4 & 6 & 2 & \mathbf{3} & 0 \end{bmatrix}$$
$k = 7$

$$\begin{bmatrix} 0 & 3 & 2 & 7 & 9 & 10 & 11 & 11 \\ 3 & 0 & 4 & 9 & 11 & 12 & 13 & 13 \\ 2 & 4 & 0 & 5 & 7 & 8 & 9 & 9 \\ 7 & 9 & 5 & 0 & 2 & 3 & 4 & 4 \\ 9 & 11 & 7 & 2 & 0 & 5 & 6 & 6 \\ 10 & 12 & 8 & 3 & 5 & 0 & 1 & 2 \\ 11 & 13 & 9 & 4 & 6 & 1 & 0 & 3 \\ 11 & 13 & 9 & 4 & 6 & 2 & 3 & 0 \end{bmatrix}$$
$k = 8$

Abb. 5.15: *Die vom Warshall-Algorithmus berechneten Matrizen W_k für $k = 1, \ldots, 8$ für den oben dargestellten Beispielgraphen. Die fett gedruckten Einträge wurden im jeweiligen Schritt angepasst. Ist also ein Eintrag $W_k[i, j]$ fett gedruckt dargestellt, so gilt, dass $W_{k-1}[i, k] + W_{k-1}[k, j] < W_{k-1}[i, j]$ ist, d. h. es gilt dass es einen Weg über den Knoten k gibt der kleiner als der bisher berechnete Weg ist.*

5.4 Minimaler Spannbaum



Neben dem systematischen Durchlaufen eines Graphen und dem Finden von kürzesten Wegen ist das Finden von minimalen (bzw. maximalen) Spannbäumen das in der Praxis wichtigste graphentheoretische Problem. Die Anwendungsbeispiele hierfür sind vielfältig, etwa das Finden eines möglichst preisgünstigen zusammenhängenden Netzwerkes.

Wir stellen in diesem Abschnitt den Algorithmus von Kruskal vor, der wie der Algorithmus von Dijkstra, auch ein Greedy-Algorithmus ist. Im Verlauf des Algorithmus von Kruskal muss eine Kantenmenge eines Graphen wiederholt daraufhin überprüft werden, ob sie Zyklen enthält. Dieser Test ist zwar relativ einfach durch eine Tiefensuche realisierbar; es gibt jedoch eine effizientere Möglichkeit, als diese wiederholte Durchführung der Tiefensuche. Wir stellen hierzu eine Implementierung der sog. Union-Find-Operationen vor (in der deutschen Literatur manchmal auch als *Vereinigungs-Suche* bezeichnet) mit deren Hilfe ein effizienterer Test auf Zyklenfreiheit möglich ist.

5.4.1 Problemstellung

Gegeben sei wiederum ein kantengewichteter Graph $G = (V, E)$ mit Gewichtsfunktion $w : E \to \mathbb{R}^+$. Gesucht ist nun die mit den geringsten Kosten verbundene Möglichkeit, alle Knoten in G mit Kanten aus E zu verbinden. Man kann sich leicht überlegen, dass solch ein Verbindungsgraph ein Spannbaum sein *muss*[2]. Abbildung 5.16 gibt ein Beispiel eines minimalen Spannbaums – der übrigens nicht immer eindeutig bestimmt ist; es kann durchaus mehrere minimale Spannbäume geben.

Aufgabe 5.17

(a) Finden Sie einen weiteren minimalen Spannbaum des Beispielgraphen aus Abbildung 5.16.

(b) Finden Sie einen maximalen Spannbaum des Beispielgraphen aus Abbildung 5.16.

[2]Ein einfacher Beweis über Widerspruch: Angenommen solch eine kostengünstigste Verbindung würde einen Kreis enthalten; entfernt man aber eine Kante e mit $w(e) > 0$ aus diesem Kreis, so ist der Graph immer noch zusammenhängend, verbindet also alle Knoten miteinander, und hat ein geringeres Gewicht. Folglich war diese ursprüngliche Verbindungsmöglichkeit auch nicht die kostengünstigste; was ein Widerspruch zur Annahme ist. Die kostengünstigste Verbindungsmöglichkeit kann also keinen Kreis enthalten.

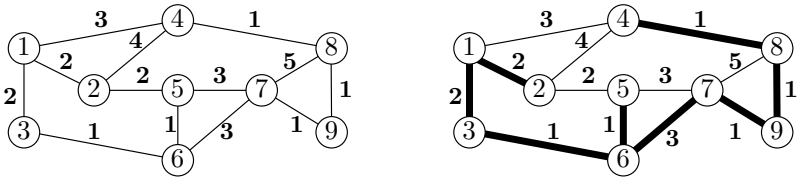


Abb. 5.16: *Ein ungerichteter gewichteter Beispielgraph zusammen mit einem minimalen Spannbaum.*

Es gibt wichtige Anwendungen für dieses Problem. Wir geben zwei Beispiele hierfür an. **1:** Das Finden eines möglichst preisgünstigen zusammenhängenden Netzwerkes. Die Kantengewichte geben hierbei jeweils Auskunft darüber, wie teuer es ist, zwischen zwei Orten eine Netzwerkverbindung zu installieren. Die Suche nach einem minimalen Spannbaum würde dann der Suche nach der kostengünstigsten Netzwerkinstallation entsprechen, die alle Teilnehmer verbindet.
2: Für einige Netzwerkprotokolle stellt es ein Problem dar, wenn es mehrere mögliche Pfade für das Versenden eines Datenpaketes von einem Netzknoten i zu einem anderen Netzknoten j gibt. In bestimmten Netzwerken können aus dieser Redundanz Inkonsistenzen entstehen. Um solche redundanten Pfade zu vermeiden, muss ein Spannbaum (vorzugsweise ein minimaler Spannbaum) über alle beteiligten Netzwerkknoten gefunden werden.

5.4.2 Der Algorithmus von Kruskal

Der Kruskal-Algorithmus verwendet eine typische Greedy-Strategie: „größere“ Lösungen werden schrittweise aus „kleineren“ Lösungen aufgebaut. In jedem dieser Schritte wird eine Lösung immer aus der in diesem Moment am besten erscheinenden Erweiterung angereichert. Im Falle des Kruskal-Algorithmus sieht diese Strategie konkret folgendermaßen aus: In jedem Schritt wird immer diejenige Kante mit dem minimalen Gewicht zur Menge der Kanten hinzugefügt, die am Ende den minimalen Spannbaum bilden sollen – jedoch nur dann, wenn durch dieses Hinzufügen kein Kreis entsteht (ein Spannbaum muss ja ein zusammenhängender *kreisfreier* Teilgraph sein; siehe hierzu auch Anhang B.4.1). Abbildung 5.17 zeigt ein Beispiel für den Ablauf des Kruskal-Algorithmus auf einem Beispielgraphen.

Korrektheit. Die folgenden beiden Eigenschaften (mit Hilfe derer die Korrektheit des Kruskal-Algorithmus leicht zu zeigen ist) gelten für jeden minimalen Spannbaum.

1. Die Kreiseigenschaft. Sei C ein beliebiger Kreis und e eine Kante aus C mit maximalem Gewicht. Dann gilt, dass der minimale Spannbaum e *nicht* enthält.
Beweis: Wir nehmen an, e wäre im minimalen Spannbaum enthalten. Entfernen wir e, so zerfällt der Spannbaum in zwei Komponenten K und K'. In C gibt es jedoch (da C ein Kreis ist) eine andere Kante e', die K und K' miteinander verbindet. Durch Wahl von e' erhalten wir also wiederum einen Spannbaum. Da $w(e) > w(e')$ hat jedoch der neue Spannbaum ein geringeres Gewicht als der ursprüngliche; somit konnte der ursprüngliche Spannbaum nicht minimal gewesen sein.

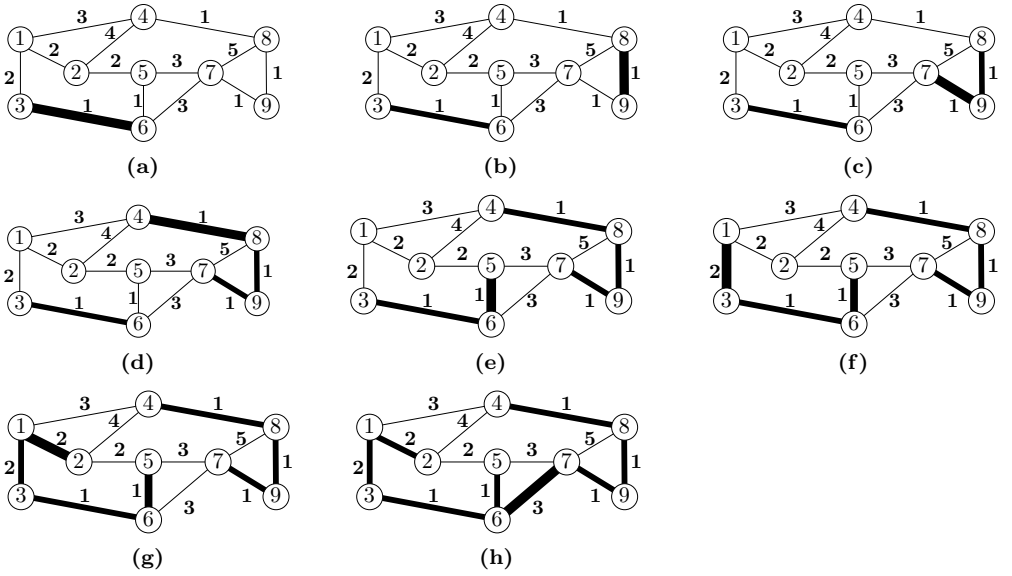


Abb. 5.17: *Ablauf des Kruskal-Algorithmus für den Beispielgraphen aus Abbildung 5.16. Wie man sieht, wird in jedem Schritt immer diejenige Kante (aus der Menge der verbleibenden Kanten) ausgewählt die das minimale Gewicht besitzt und die zusammen mit den bisher ausgewählten Kanten keinen Zyklus bildet. Zunächst werden im Beispiel alle Kanten mit Gewicht 1 ausgewählt; anschließend wird mit den Kanten mit Gewicht 2 fortgefahren. In Schritt 5.17(h) wird jedoch die Kante mit minimalem Gewicht* $(2, 5)$ *nicht ausgewählt, da sie zusammen mit den bisher ausgewählten Kanten einen Zyklus bilden würde. Stattdessen muss eine Kante mit Gewicht 3 ausgewählt werden – in diesem konkreten Fall wird* $(6, 7)$ *gewählt; es wäre aber hier ebenso möglich gewesen die Kante* $(1, 4)$ *auszuwählen.*

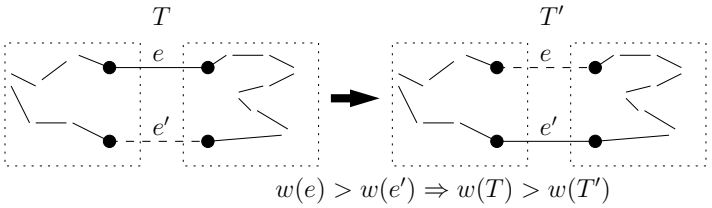


Abb. 5.18: *Durch Ersetzen der Kante e mit maximalem Gewicht durch die „kleinere“ Kante e' entsteht ein „kleinerer“ Spannbaum T'.*

2. Die Schnitteigenschaft. Sei S eine beliebige Teilmenge von Knoten. Es sei e diejenige Kante mit minimalem Gewicht, die genau einen Endpunkt in S besitzt. Dann gilt, dass der minimale Spannbaum e enthalten muss.
Beweis: Wir nehmen an, e wäre im minimalen Spannbaum *nicht* enthalten. Fügen wir nun die Kante e dem Spannbaum hinzu, so erhalten wir einen Kreis C. Entfernen wir nun eine andere Kante e' mit genau einem Endpunkt in S aus dem Kreis C, so erhalten

wir wiederum einen Spannbaum, der jedoch ein geringeres Gewicht als der ursprüngliche Spannbaum hat (da $w(e') > w(e)$); der ursprüngliche Spannbaum konnte also nicht minimal gewesen sein.

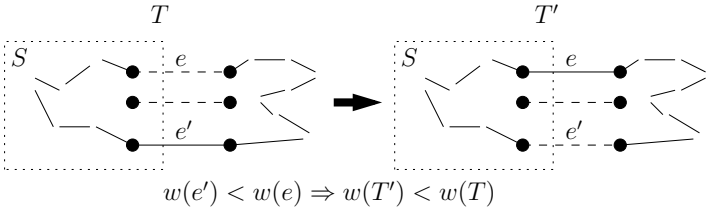


Abb. 5.19: *Durch Ersetzen der Kante e' mit nicht minimalem Gewicht durch die „kleinere" Kante e entsteht ein „kleinerer" Spannbaum T'.*

Mit Hilfe dieser beiden Eigenschaften können wir zeigen, dass jede Kante, die vom Kruskal-Algorithmus ausgewählt wird, tatsächlich zum minimalen Spannbaum gehören *muss*. Wir unterscheiden zwei Fälle:
1. Angenommen, die ausgewählte Kante e erzeugt einen Kreis C. Da alle anderen Kanten dieses Kreises zu einem früheren Zeitpunkt ausgewählt wurden, ist e die Kante mit maximalem Gewicht in C, kann also nicht zum minimalen Spannbaum gehören, wird also vom Kruskal-Algorithmus zu Recht nicht ausgewählt.
2. Angenommen, die ausgewählte Kante $e = \{i, j\}$ erzeugt keinen Kreis. Sei K die Menge der Knoten der (Zusammenhangs-)Komponente der i angehört. Die Kante e besitzt genau einen Endpunkt in K und ist gleichzeitig die Kante mit minimalem Gewicht, die genau einen Endpunkt in K hat, wird also nach der Schnitteigenschaft zu Recht vom Kruskal-Algorithmus ausgewählt.

Implementierung. Listing 5.7 zeigt eine einfache Implementierung des Kruskal-Algorithmus.

```
1 def kruskal(graph):
2   allEdges = [(graph.w(i,j), i, j) for i,j in graph.E_undir()]
3   allEdges.sort(reverse=True) # absteigend sortieren
4   spannTree = []
5   while len(spannTree) < len(graph.V()) -1 and allEdges≠[]:
6     (w,i,j) = allEdges.pop()
7     if not buildsCircle(spannTree,(i,j)):
8       spannTree.append((i,j))
9   return spannTree
```

Listing 5.7: *Einfache Implementierung des Kruskal-Algorithmus*

Mittels der Listenkomprehension in Zeile 2 wird die Liste *allEdges* aller Kante inklusive ihrer Gewichte erzeugt und in Zeile 3 nach ihren Gewichten absteigend sortiert. In jedem **while**-Schleifendurchlauf wird dann mittels *allEdges.pop*() immer diejenige noch nicht betrachtete Kante mit minimalem Gewicht ausgewählt und genau dann zum Spannbaum *spannTree* hinzugefügt, falls dadurch kein Kreis erzeugt wird.

Zwei Punkte sind jedoch an dieser Implementierung zu bemängeln bzw. unvollständig:
1: Das Sortieren aller Kanten nach deren Gewicht hat eine Laufzeit von $O(|E| \log |E|)$ und ist damit ineffizienter als die Verwendung einer Heap-Struktur: Der Aufbau des Heaps benötigt $O(|E|)$ Schritte; es werden jedoch nur $|V| - 1$ Elemente aus dem Heap entnommen und wir erhalten daher eine Laufzeit von $O(|E| + |V| \log |E|)$; für den häufigen Fall, dass $|E| \gg |V|$ ist dies wesentlich günstiger als die Laufzeit von $O(|E| \log |E|)$. Zur Implementierung siehe Aufgabe 5.18.
2: Wir haben in Listing 5.7 offen gelassen, wie die Funktion *buildsCircle* zu implementieren ist, die testet, ob durch das Hinzufügen der Kante (i, j) zur Kantenmenge *spannTree* ein Kreis entsteht. Es wäre möglich diesen Test mit Hilfe einer Tiefensuche durchzuführen; es geht jedoch schneller über eine sog. *Union-Find*-Datenstruktur.

Aufgabe 5.18

Eine verbesserte Implementierung des Kruskal-Algorithmus würde es vermeiden die gesamte Kantenmenge zu sortieren, sondern stattdessen einen Heap verwenden, um in jedem Durchlauf effizient die Kante mit dem minimalen Gewicht auszuwählen.

Passen Sie die Implementierung des in Listing 5.7 gezeigten Skripts entsprechend an.

Aufgabe 5.19

Implementieren Sie eine Funktion *buildsCircle* (*tree* ,(i, j)), die testet, ob der Graph *graph* einen Zyklus enthält. Verwenden Sie hierzu als Basis eine Tiefensuche.

Aufgabe 5.20

Welche Laufzeit hat die in Listing 5.7 gezeigte Implementierung des Kruskal-Algorithmus, falls *buildsCircle* über eine Tiefensuche implementiert wird und ...

(a) ... die Kante mit dem geringsten Gewicht durch eine entsprechende Sortierung der Kantenmenge erhalten wird.

(b) ... die Kante mit dem geringsten Gewicht durch Aufbau einer Heapstruktur über die Kantenmenge erhalten wird.

Aufgabe 5.21

(a) Kann man den minimalen Spannbaum auch finden, indem man genau umgekehrt wie der Kruskal-Algorithmus vorgeht, d. h. man beginne mit allen im Graphen enthaltenen Kanten und entfernt Kanten mit dem momentan höchsten Gewicht – aber nur dann, wenn man dadurch den Graphen nicht auseinanderbricht?

(b) Geben Sie eine Implementierung des „umgekehrten“ Kruskal-Algorithmus an.

5.4.3 Union-Find-Operationen

Über eine effiziente Implementierung der sog. *Union-Find*-Operationen, d. h. der Mengenoperationen „Vereinigung“ zweier Mengen und „Suche“ eines Elementes in einer Menge, erhält man sogleich eine effiziente Methode zum Testen, ob durch das Hinzufügen einer Kante $\{i, j\}$ zu einer kreisfreien Kantenmenge S ein Zyklus entsteht; genau dieser Test muss im Verlaufe des Kruskal-Algorithmus wiederholt durchgeführt werden.

Die effizientesten Implementierungen der Union-Find-Operationen modellieren die Mengenzugehörigkeit durch Graphen und sehen die Relation „gehört zur selben Menge wie“ im Graphen modelliert als „gehört zur selben (Zusammenhangs-)Komponente wie“.

In einer Union-Find-Datenstruktur wird eine Menge von Objekten $v_1, \ldots v_n$ verwaltet. Anfangs sieht man die Objekte als einelementige Mengen. Im Verlauf der Benutzung der Datenstruktur können die Mengen vereinigt werden; es wird also immer eine Menge von disjunkten[3] Teilmengen verwaltet. Es werden die folgende beiden Operationen unterstützt:

- *find*(v): Diese Funktion liefert eine eindeutige Repräsentation derjenigen Menge zurück, zu der v gehört.
- *union*(x,y): Vereinigt die beiden Mengen, deren eindeutige Repräsentationen x und y sind.

Abbildung 5.20 zeigt ein Beispiel für den Aufbau einer Union-Find-Datenstruktur; diese spezielle Folge von Vereinigungsschritten würde sich während der in Abbildung 5.17 gezeigten Ausführung des Kruskal-Algorithmus ergeben.

Mit der Union-Find-Datenstruktur kann man während der Ausführung des Kruskal-Algorithmus protokollieren, welche Zusammenhangskomponenten sich aus dem bisher berechneten (Teil-)Spannbaum ergeben; aus dieser Information wiederum kann man in jedem Schritt des Kruskal-Algorithmus leicht nachprüfen, ob durch das Hinzufügen einer Kante ein Kreis entsteht. Zu Beginn enthält *spannTree* keine Kanten, alle Knoten stehen daher einzeln da und *spannTree* hat folglich 9 Zusammenhangskomponenten. Dies entspricht dem in Abbildung 5.20 gezeigten Anfangszustand. In jedem Schritt wird durch den Kruskal-Algorithmus nun die Kante $\{i, j\}$ mit dem geringsten Gewicht ausgewählt. Es gibt zwei Fälle:

1. Es gilt *find*(i)==*find*(j). D. h. i und j befinden sich schon in derselben Zusammenhangskomponente (d. h. es gibt in *spannTree* einen Weg von i nach j). Ein Hinzufügen der Kante $\{i, j\}$ würde daher einen Kreis entstehen lassen.
2. Es gilt *find*$(i)\neq$*find*(j). D. h. das Hinzufügen der Kante $\{i, j\}$ würde zwei bisher getrennte Komponenten verbinden, d. h. *spannTree* würde kreisfrei bleiben. Der Algorithmus würde also die Kante zu *spannTree* hinzufügen und durch Ausführen von *union*(*find*(i), *find*(j)) in der Union-Find-Datenstruktur protokollieren, dass sich nun i und j in der gleichen Komponente (bzw. Menge) befinden.

[3] Die Mengen M_1 und M_2 heißen *disjunkt*, wenn sie keine gemeinsamen Elemente besitzen, d. h. wenn Ihr Schnitt gleich der leeren Menge ist. In Formeln: wenn $M_1 \cap M_2 = \emptyset$ gilt.

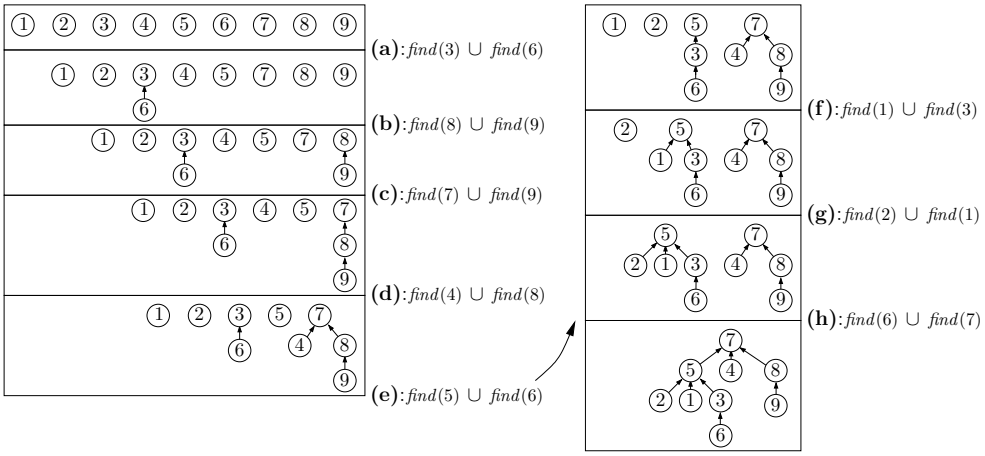


Abb. 5.20: *Ein Beispiel für den Aufbau einer Union-Find-Datenstruktur. Es werden 9 Elemente verwaltet, die zu Beginn einzeln stehen. Wie man sieht, wird die Mengenzugehörigkeit durch die Union-Find-Datenstruktur als Menge von Bäumen repräsentiert. Befinden sich zwei Elemente im selben Baum, so heißt dies, dass die beiden Elemente derselben Menge angehören. Nach jedem union-Schritt werden (falls die zu vereinigenden Elemente sich in verschiedenen Mengen befinden) zwei Bäume miteinander verschmolzen. Beispielsweise wird in Schritt* **(d)** *die Menge, der 4 angehört (also find*(4)*), was in diesem Falle einfach der Menge* $\{4\}$ *entspricht, vereinigt mit der Menge, der 8 angehört (also find*(8)*, was in diesem Falle der Menge* $\{7,8,9\}$ *entspricht); als Folge werden die beiden entsprechenden Bäume verschmolzen. Man beachte, dass dieser „Verschmelzungsprozess“ nicht eindeutig ist. Es gibt immer zwei Möglichkeiten, wie zwei Bäume* B_1 *und* B_2 *miteinander verschmolzen werden können: Entweder man hängt* B_1 *als Kind unter die Wurzel von* B_2 *oder man hängt* B_2 *als Kind unter die Wurzel von* B_1*.*

Angenommen *uf* sei eine Instanz der Klasse *UF* (deren Implementierung wir weiter unten in Listing 5.8 präsentieren), erzeugt mittels

$$uf = UF(graph.numNodes)$$

Wir sollten also die Zeilen 7 und 8 in Listing 5.7 folgendermaßen ersetzen:

```
7   if not buildsCircle(spannTree,(i,j)):
8     spannTree.append((i,j))
```

$\Longrightarrow$

```
7     Mi = uf.find(i)
8     Mj = uf.find(j)
9     if Mi ≠ Mj:
10      spannTree.append((i,j))
11      uf.union(Mi,Mj)
```

Um festzustellen, ob durch Hinzunahme der Kante $\{i,j\}$ ein Kreis entsteht, wird also geprüft, ob i und j zur selben Menge gehören. Ist dies nicht der Fall (falls nämlich *Mi*≠*Mj*), so wird die Kante $\{i,j\}$ zum Spannbaum hinzugefügt (Zeile 10) und anschließend die Menge, der j angehört, und die Menge, der i angehört, vereinigt (Zeile 11).

Listing 5.8 zeigt die Implementierung der Klasse *UF*.

```
1 class UF(object):
2   def __init__(self,n):
3     self.parent = [0]*n
4   def find(self,x):
5     while self.parent[x] > 0: x = self.parent[x]
6     return x
7   def union(self,x,y):
8     self.parent[y] = x
```

Listing 5.8: *Implementierung der Union-Find-Datenstruktur.*

Eine Kante in dem „Wald“ der durch die Union-Find-Datenstruktur dargestellt wird, wird durch die Liste *parent* repräsentiert. Der i-te Eintrag in *parent* enthält den Vater des Knotens i. Falls $parent[i]$ gleich 0 ist, heißt dies, dass i die Wurzel des Baumes ist. Initial werden alle *parent*-Einträge auf 0 gesetzt (Zeile 3), d. h. alle verwalteten Elemente sind Wurzeln, d. h. initial haben wir es mit einem Wald aus n Bäumen zu tun, die jeweils nur ein Element (nämlich das Wurzelelement) enthalten. Ein Aufruf von *union*(x,y) fügt zwei Bäume zusammen, indem die Wurzel des einen Baumes (der y enthält) als Kind unter die Wurzel des anderen Baumes (der x enthält) gehängt wird. Der Aufruf *find*(x) liefert die Wurzel des Baumes zurück, der x enthält.

Aufgabe 5.22

Implementieren Sie für die in Listing 5.8 gezeigte Klasse *UF* die *str*-Funktion, die ein Objekt der Klasse in einen String umwandelt. Die Ausgabe sollte gemäß folgendem Beispiel erfolgen:

```
>>> uf = UF(10)
>>> uf.union(1,2) ; uf.union(1,3) ; uf.union(5,6) ; uf.union(8,9)
>>> str(uf)
>>> '{1, 2, 3} {4} {5, 6} {7} {8, 9} '
```

Wir betrachten zwei Möglichkeiten, die Union-Find-Datenstruktur zu optimieren:

Balancierung. Im ungünstigsten Falle entwickeln sich in der Union-Find-Datenstruktur entartete (d. h. stark unbalancierte) Bäume. Ein ungünstiger Fall tritt immer dann ein, wenn ein Baum der Höhe h unter die Wurzel eines Baumes mit geringerer Höhe h' gehängt wird, d. h. wenn *union*(x,y) ausgeführt wird, und die Höhe des Baumes, in dem sich x befindet kleiner ist als die Höhe des Baumes, in dem sich y befindet. Wir können dies einfach dadurch vermeiden, indem wir prüfen, welcher Baum höher ist. Wir wollen aus Performance-Gründen vermeiden, wiederholt die Höhe zu berechnen. Daher speichern wir immer die Höhe jedes Baumes im *parent*-Eintrag der Wurzel – jedoch als negative Zahl, um weiterhin in der Lage zu sein, die Wurzel eines Baumes „erkennen“ zu können. Damit bleibt auch die **while**-Schleife in Listing 5.8 in Zeile 5 gültig.

Aufgabe 5.23

Verbessern Sie die in Abbildung 5.8 gezeigte Implementierung dadurch, dass Sie auf die Balancierung der in der Union-Find-Datenstruktur verwalteten Bäume achten. Der Baum *find*(x) sollte also nur dann als Kind unter die Wurzel des Baums *find*(y) gehängt werden, wenn die Höhe von *find*(x) kleiner ist als die Höhe von *find*(y); andernfalls sollte *find*(y) unter die Wurzel von *find*(x) gehängt werden.

Pfad-Komprimierung. Ein Aufruf von *find*(x) findet immer den Pfad von x zur Wurzel des Baumes in dem sich x befindet. Nach solch einem Aufruf ist es günstig eine direkte Kante von x zur Wurzel einzufügen, um bei einem späteren erneuten Aufruf von *find*(x) zu vermeiden, dass wiederum der gleiche Pfad bis zur Wurzel gelaufen werden muss. Diese Technik nennt man *Pfadkomprimierung*. Zur Implementierung der Pfadkomprimierung muss lediglich die *find*-Methode der Klasse *UF* angepasst werden. Listing 5.9 zeigt die Implementierung der *find*-Methode, die zusätzlich eine Pfadkomprimierung durchführt.

```
1 class UF(object):
2   ...
3   def find( self ,x):
4     i=x
5     while self.parent[x] > 0: x = self.parent[x]
6     while self.parent[i] > 0:
7       tmp=i ; i=self.parent[i] ; self.parent[tmp]=x
8     return x
```

Listing 5.9: *Implementierung der Pfadkomprimierung in der find-Methode.*

Zunächst wird, wie in der ursprünglichen Implementierung der *find*-Methode, die Wurzel des als Parameter übergebenen Elements x gesucht. Anschließend wird in den Zeilen 6 und 7 der gegangene Pfad nochmals abgelaufen und die *parent*-Einträge aller Knoten auf diesem Pfad direkt auf die Wurzel x des Baumes gesetzt. Dadurch wird eine Erhöhung der Laufzeit für spätere *find*-Aufrufe ermöglicht.

Laufzeit. Obwohl die Funktionsweise der Union-Find-Datenstruktur verhältnismäßig einfach nachvollziehbar ist, ist eine Laufzeitanalyse komplex. Wir beschränken uns hier deshalb darauf, lediglich die Ergebnisse der Laufzeitanalyse zu präsentieren. Die Kombination der beiden vorgestellten Optimierungen, Pfad-Komprimierung und Balancierung, ermöglicht eine (zwar nicht ganz, aber nahezu) lineare Laufzeit für die Erzeugung eine Union-Find-Datenstruktur aus $|E|$ Kanten.

Damit ergibt sich für den Kruskal-Algorithmus eine Laufzeit von $O(|E| \log(|E|))$: Die **while**-Schleife wird im ungünstigsten Fall $|E|$ mal ausgeführt; in jedem Durchlauf wird die Kante mit dem geringsten Gewicht aus der Heap-Struktur entfernt, was $O(\log(|E|))$ Schritte benötigt; insgesamt ergibt sich daraus die Laufzeit von $O(|E| \log(|E|))$. Die Tests auf Entstehung der Kreise brauchen insgesamt (wie eben erwähnt) $O(|E|)$ und der

anfängliche Aufbau des Heaps ebenfalls $O(|E|)$ Schritte (was aber durch $O(|E|\log(|E|))$ „geschluckt“ wird).

Aufgabe 5.24

Schreiben Sie die folgenden Funktionen, um Performance-Tests auf dem Kruskal-Algorithmus durchzuführen:

(a) Schreiben Sie eine Funktion *genRandGraph*(*n*,*m*,*k*), die einen zufälligen Graphen $G = (V, E)$ generiert mit $|V| = n$, $|E| = m$ und $w : E \to \{1, \ldots, k\}$.

(b) Testen Sie nun die Laufzeit des Kruskal-Algorithmus auf einem Graphen *genGraph*(1000,5000,1000), dessen Implementierung ...

1. ... die Kanten sortiert (statt Heaps zu verwenden) und die Tiefensuche verwendet.
2. ... die Kanten sortiert und statt der Tiefensuche eine einfache Union-Find-Struktur verwendet.
3. ... die Kanten sortiert und eine optimierte Union-Find-Struktur verwendet.
4. ... Heaps verwendet und eine optimierte Union-Find-Struktur verwendet.

5.5 Maximaler Fluss in einem Netzwerk.

Wir behandeln hier in diesem Abschnitt eine sowohl in wirtschaftswissenschaftlichen als auch in naturwissenschaftlichen Kontexten häufig auftretende Fragestellung. Es geht um das Problem, wie und wie viel „Material“ (das kann je nach Kontext Waren, Mitarbeiter, elektrischer Strom oder eine Flüssigkeit sein) durch ein Netzwerk von Knoten gelenkt werden kann.

5.5.1 Netzwerke und Flüsse

Ein Netzwerk ist ein gewichteter gerichteter Graph $G = (V, E)$ mit Gewichtsfunktion $w : E \to \mathbb{R}^+$, d. h. jeder Kante ist eine positive reelle Zahl zugeordnet. Wir interpretieren die einer Kante zugeordnete Zahl als *Kapazität*. Diese Kapazität sagt uns, wie viel Material (bzw. Strom, Flüssigkeit, usw.) maximal über diese Kante „fließen“ kann. Es seien zwei Kanten $s, t \in V$ speziell ausgezeichnet und wir nennen s die *Quelle* und t die *Senke* des Netzwerkes. Außerdem sei ein *Fluss* gegeben, modelliert als Funktion $f : V \times V \to \mathbb{R}^+$, der die folgenden Bedingungen erfüllen sollte:

1. Aus der Kapazität ergibt sich die maximal möglich Menge „Material“, die über eine Kante fließen kann, d. h.

$$f(u, v) \leq w(u, v) \text{ für alle } (u, v) \in E$$

2. Der Fluss in Rückwärtsrichtung hat immer den negativen Wert des Flusses in Vorwärtsrichtung, d. h.

$$f(u,v) = -f(v,u) \text{ für alle } (u,v) \in E$$

3. Das „Material“, das in einen Knoten hineinfließt, muss auch wieder hinausfließen, d. h.

$$\text{Für jeden Knoten } v \in V \setminus \{s,t\} \text{ muss gelten: } \sum_{u \in V} f(u,v) = 0$$

Diese Bedingung wird manchmal auch als das *Kirchhoff'sche Gesetz* oder das *Gesetz der Flusserhaltung* bezeichnet. Wir wollen also ein Szenario modellieren, in dem alle Knoten (ausgenommen s und t) lediglich das hineinfließende „Material“ weitergeben, also weder „Material“ konsumieren, noch neues „Material“ erzeugen können. Lediglich die Quelle s kann „Material“ produzieren und die Senke t kann „Material“ konsumieren.

Aufgabe 5.25

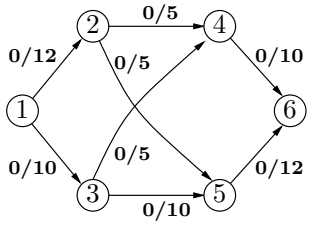
Warum hat die den Fluss modellierende Funktion f nicht den „Typ“ $f : E \to \mathbb{R}^+$, sondern den Typ $f : V \times V \to \mathbb{R}^+$?

Der Wert eines Flusses ist definiert als $\sum_{u \in V} f(s,u)$ also die Menge an Material, die von der Quelle erzeugt wird. Da für alle Knoten (aus s und t) Flusserhaltung gilt, muss genau dieser Fluss auch bei der Senke wieder ankommen, d. h. es muss gelten, dass $\sum_{u \in V} f(s,u) = \sum_{u \in V} f(u,t)$. In vielen Anwendungen ist der maximal mögliche Fluss gesucht, d. h. die maximal mögliche Menge an Material, die (unter Berücksichtigung der Kapazitäten der Kanten) durch ein Netzwerk geschleust werden kann. Abbildung 5.21 zeigt ein Beispiel, das zeigt, wie man sich diesem maximalen Fluss annähern kann.

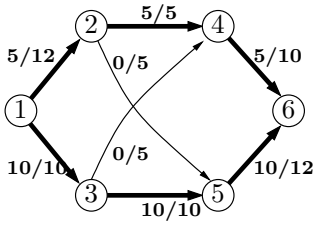
5.5.2 Der Algorithmus von Ford-Fulkerson

Die Idee des sog. Algorithmus von Ford-Fulkerson ist recht einfach und schon in Abbildung 5.21 angedeutet: Solange es einen Pfad von der Quelle zur Senke gibt, mit noch verfügbarer Kapazität auf allen Kanten des Pfades, so schicken wir (möglichst viel) „Material“ über diesen Pfad. Genauer: Wurde im letzten Schritt ein gültiger Fluss f des Netzwerks $G = (V,E)$ (mit Kapazitätsfunktion w) gefunden, so wird im nächsten Schritt zunächst das sog. Restnetzwerk $G_f = (V, E_f)$ berechnet, das man einfach aus dem „alten“ Netzwerk G durch Berechnung der neuen Kapazitätsfunktion $w_f(i,j) = w(i,j) - f(i,j)$[4] erhält. Anschließend versucht der Algorithmus in G_f einen Pfad p von s nach t in G_f zu finden, so dass $w_f(i,j) > 0$ für alle $(i,j) \in p$; einen solchen Pfad nennt man auch *Erweiterungspfad*. Gibt es keinen Erweiterungspfad, so bricht der

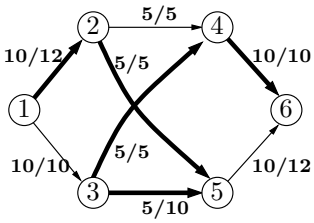
[4] Es kann hierbei sogar passieren, dass das Restnetzwerk G_f einen Fluss von j nach i erlaubt, auch wenn G keinen Fluss von j nach i erlaubt hatte: Falls $f(i,j) > 0$ und $w(j,i) = 0$ dann ist nämlich $w_f(j,i) = w(j,i) - f(j,i) = -f(j,i) = f(i,j) > 0$; die Rückrichtung hat somit in G_f eine positive Kapazität und ein Fluss von j nach i wäre in G_f möglich.



(a) Fluss mit Wert 0.



(b) Verbesserter Fluss mit Wert 15.



(c) Der maximale Fluss mit Wert 20.

Abb. 5.21: *Drei verschiedene sukzessiv vergrößerte Flüsse in einem Netzwerk. Wie man sieht kann man aus dem „leeren“ Fluss (dargestellt in Abbildung 5.21(a)) relativ einfach einen Fluss mit Wert 15 generieren: Über den Pfad* $(1, 3, 5, 6)$ *kann man einen Fluss mit Wert 10 (entsprechend dem minimalen Kantengewicht auf diesem Pfad) fließen lassen und über den Pfad* $(1, 2, 4, 6)$ *kann man einen Fluss mit Wert 5 (wiederum entsprechend dem minimalen Kantengewicht auf diesem Pfad) fließen lassen; dies ergibt zusammengenommen den in Abbildung 5.21(b) gezeigten Fluss mit Wert 10+5=15. Nicht ganz so offensichtlich ist die in Abbildung 5.21(c) gezeigte Möglichkeit, diesen Fluss zu vergrößern. Über den Pfad* $(1, 2, 5, 3, 4, 6)$ *kann man einen zusätzlichen Fluss mit Wert 5 schicken; man beachte, dass dieser Pfad die Kante* $(5, 3)$ *beinhaltet, also die im ursprünglichen Graphen vorhandenen Kante* $(3, 5)$ *in Rückwärtsrichtung durchlaufen wird. Laut Bedingung 2 gilt für den Fluss aus Abbildung 5.21(b) über diese Kante:* $f(3, 5) = -f(5, 3) = -10$*; dieser Fluss über diese Kante auf dem Pfad* $(1, 2, 5, 3, 4, 6)$ *kann von -10 auf den Wert -5 vergrößert werden. Insgesamt ergibt sich also ein Fluss mit Wert 20, dargestellt in Abbildung 5.21(c).*

Algorithmus ab und die bisher gefundenen Flüsse zusammengenommen bilden einen maximalen Fluss. Konnte dagegen ein Erweiterungspfad p gefunden werden, so wird für alle Kanten $(i, j) \in p$ der Fluss f' auf den Wert $min\{w_f(i, j) \mid (i, j) \in p\}$ gesetzt, anschließend wieder das Restnetzwerk berechnet, usw.

Listing 5.10 zeigt die Implementierung in Python. In jedem Durchlauf der **while**-Schleife wird zunächst der Fluss f über den (im letzten Schritt) berechneten Erweiterungspfad *path* bestimmt; wie schon oben beschrieben entspricht der Wert dieses Flusses dem minimalen in *path* befindlichen Kantengewicht. Dieser Fluss f wird zum bisherigen Gesamtfluss *flow* hinzuaddiert (Zeile 7). Anschließend wird in der **for**-Schleife (Zeile 9 bis 15) das Restnetzwerk $graph_f$ des zu Beginn der **while**-Schleife betrachteten Netzwerkes *graph* berechnet, d. h. für jede Kante $(i, j) \in path$ müssen die Kapazitäten entsprechend des Flusses f folgendermaßen angepasst werden: Besitzt eine Kante (i, j) im Graphen *graph* die Kapazität w, so erhält diese im Graphen $graph_f$ die Kapazität $w - f$; falls $w - f == 0$ (d. h. die Kapazität verschwindet), so wird die Kante mittels *delEdge* aus dem Graphen entfernt. Der Grund dafür, dass wir die Kante in diesem Falle löschen, liegt darin, dass im Falle von Netzwerken die Tatsache, dass eine Kante (i, j) nicht existiert gleichbedeutend ist mit der Tatsache, dass eine Kante (i, j) die Kapazität 0 hat. Aus dem gleichen Grund weisen wir dem Gewicht der Rückwärtskante (j, i) in Zeile 11 den Wert 0 zu, falls diese nicht existiert. In den Zeilen 14 und 15 wird die Rückwärtskante entsprechend angepasst und – analog zur Vorwärtskante – gelöscht, falls deren Wert 0 wird.

```
1  def maxFlow(s,t,graph):
2    path = findPath(s,t,graph)
3    flow = 0
4    while path ≠ []:
5      # Bestimme größtmöglichen Fluss über path
6      f = min(graph.w(i,j) for i,j in path)
7      flow += f
8      # Restnetzwerk berechnen
9      for i,j in path:
10       w = graph.w(i,j)
11       wBack = graph.w(j,i) if graph.isEdge(j,i) else 0
12       if w-f == 0: graph.delEdge(i,j)
13       else:        graph.addEdge(i,j,w-f)
14       if wBack+f == 0: graph.delEdge(j,i)
15       else:            graph.addEdge(j,i,wBack+f)
16     # Pfad im Restnetzwerk finden
17     path = findPath(s,t,graph)
18   return flow
```

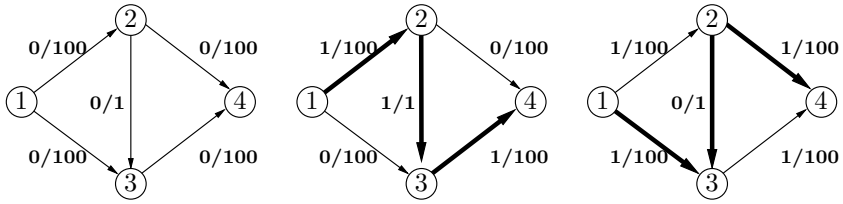
Listing 5.10: *Implementierung des Ford-Fulkerson-Algorithmus.*

In Zeile 17 wird schließlich nach einem Erweiterungspfad von s nach t durch das eben berechnete Restnetzwerk gesucht und mit diesem dann im nächsten **while**-Schleifendurchlauf analog verfahren.

Dieser Algorithmus funktioniert im Allgemeinen gut. Gibt es aber mehrere Pfade von s nach t dann kann es, abhängig davon welcher Pfad gewählt wird, zu einer sehr schlechten Worst-Case-Laufzeit kommen. Im ungünstigsten Fall kann die Laufzeit sogar vom *Wert* des größten Flusses selbst abhängen. Abbildung 5.22 zeigt ein Beispiel eines solchen problematischen Falles. Man kann zeigen, dass dieser ungünstige Fall einfach vermieden werden kann, indem man als Erweiterungspfad grundsätzlich einen Pfad mit möglichst wenig Kanten wählt.

Aufgabe 5.26

Für die in Listing 5.10 gezeigte Implementierung des Ford-Fulkerson-Algorithmus wird eine Funktion benötigt, die eine Kante eines Graphen löschen kann – siehe Zeilen 12 und 14.
Fügen Sie der Klasse *Graph* eine Methode *delEdge*(i,j) hinzu, die die Kante (i,j) aus dem Graphen löscht.



(a) Fluss mit Wert 0. **(b)** Fluss mit Wert 1. **(c)** Fluss mit Wert 2.

Abb. 5.22: *Dieses Beispiel zeigt einen ungünstigen Verlauf des Ford-Fulkerson-Algorithmus, der zwar letztendlich zum richtigen Ergebnis führt, jedoch eine (unnötig) langen Laufzeit aufweist. Gesucht ist ein maximaler Fluss von der Quelle 1 zur Senke 4. Wird* $(1, 2, 3, 4)$ *als erster Erweiterungspfad gewählt, so kann der Fluss nur um den Wert „1“ verbessert werden (denn:* $max(w(1, 2), w(2, 3), w(3, 4)) = 1$*), gezeigt in Abbildung 5.22(b). Wird im nächsten Schritt der gültige Pfad* $(1, 2, 3, 4)$ *des Restnetzwerkes (das sich aus dem im vorigen Schritt gefundenen Flusses ergibt) gewählt, so kann der Fluss wiederum nur um den Wert „1“ erhöht werden, gezeigt in Abbildung 5.22(c). Verfährt man so weiter, so würde der Algorithmus 200 Schritte benötigen. Durch Wahl der Pfade* $(1, 2, 4)$ *und* $(1, 3, 4)$ *hätte man den maximalen Fluss aber in lediglich zwei Schritten berechnen können.*

Aufgabe 5.27

Implementieren Sie die in Zeile 17 in Listing 5.10 benötigte Funktion *findPath*, die nach einem gültigen Pfad von s nach t im Restnetzwerk sucht.

Hinweis: Um das in Abbildung 5.22 erwähnte Problem zu vermeiden, muss eine Breitensuche verwendet werden – erklären Sie warum!

5.5.3 Korrektheit des Ford-Fulkerson-Algorithmus

Dass ein Erweiterungspfad p mit $f(i, j) > 0$ für alle $(i, j) \in p$ den bestehenden Fluss verbessern kann, ist leicht einzusehen. Die entscheidende Frage ist aber: Falls es keinen Erweiterungspfad mehr gibt, ist dann auch garantiert der maximal mögliche Fluss gefunden? Dass diese Antwort „Ja“ ist, ist nicht ganz so leicht einzusehen; dies kann am einfachsten über einen „Umweg“ gezeigt werden, der uns über das sog. *Max-Flow-Min-Cut-Theorem* führt. Dieses Theorem besagt, dass der maximale Fluss gleich dem minimalen Schnitt des Netzwerkes ist, oder in anderen Worten: Es besagt, dass der maximale Fluss genau gleich der Größe des „Flaschenhalses“ des Netzwerkes ist.

Definieren wir zunächst, was wir formal unter einem *Schnitt* (in einem Graphen) verstehen. Ein *Schnitt* eines Graphen $G = (V, E)$ ist eine Knotenmenge $S \subset V$. Die *Kanten des Schnittes* sind defininiert als

$$e(S) := \{\ (i, j) \in E \mid i \in S \text{ und } j \in V \setminus S\ \}$$

also als die Menge aller Kanten mit genau einem Endpunkt in S. Der Wert (bzw. die

Kapazität) eines Schnittes S ist definiert als

$$w(S) := \sum_{e \in e(S)} w(e)$$

also als die Summe aller Gewichte (bzw. Kapazitäten) aller im Schnitt enthaltenen Kanten. Als s-t-Schnitt bezeichnet man einen Schnitt S, für den $s \in S$ und $t \in V \setminus S$ gilt. Der Fluss $f(S)$ eines s-t-Schnittes S ist definiert als die Summe der Flüsse aller Kanten des Schnittes, also $f(S) := \sum_{e \in e(S)} f(e)$ Abbildung 5.23 zeigt ein Beispiel eines Schnittes (der übrigens nicht der minimale Schnitt ist) in einem Graphen.

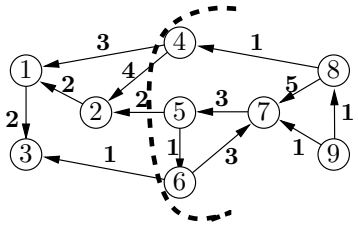


Abb. 5.23: *Der Schnitt $S = \{4, 5, 6, 7, 8, 9\}$ durch einen Beispielgraphen.*

Aufgabe 5.28

Betrachten Sie den in Abbildung 5.23 dargestellten Graphen und den Schnitt S und beantworten Sie die folgenden Fragen:

(a) Geben Sie $e(S)$ an, d. h. die zu dem Schnitt gehörige Kantenmenge.

(b) Geben Sie $w(S)$ an, d. h. die Kapazität des Schnittes S.

Aufgabe 5.29

(a) Definieren Sie eine Python-Funktion *cut*(*C*,*graph*), die eine den Schnitt definierende Knotenmenge C und einen Graphen *graph* übergeben bekommt und eine Liste aller Kanten zurückliefert die sich im Schnitt befinden. Versuchen Sie eine Implementierung als „Einzeiler“, also in der Form

```
def cut(C,graph):
  return ...
```

(b) Definieren Sie eine Python-Funktion *cutVal*(*C*,*graph*), die den Wert des Schnittes zurückliefert, der durch die Knotenmenge C definiert ist. Versuchen Sie wiederum eine Implementierung als Einzeiler.

Man kann zeigen: Für jeden beliebigen s-t-Schnitt A eines Netzwerkes $G = (V, E)$ gilt immer, dass $f(A) = f$, d. h. egal welchen s-t-Schnitt durch das Netzwerk man betrachtet, der Fluss des Schnittes hat immer den selben Wert, nämlich den des Flusses. Diese Aussage kann man leicht durch Induktion über die Anzahl der Knoten im Schnitt zeigen; wir überlassen den Beweis dem interessierten Leser.

Aufgabe 5.30

Zeigen Sie die eben aufgestellte Behauptung, die besagt dass – für einen gegebenen Fluss f – der Fluss jedes beliebigen Schnittes S immer denselben Wert hat.

Außerdem ist klar, dass für jeden s-t-Schnitt A des Netzwerkes gilt: $f(A) \leq w(A)$, d. h. für jeden Schnitt gilt, dass der Fluss des Schnittes kleiner oder gleich der Kapazität des Schnittes ist, einfach deshalb, weil für jede einzelne Kante e des Schnittes gilt, dass $f(e) \leq w(e)$. Es ist aber nicht offensichtlich, dass es immer einen Fluss und einen Schnitt gibt, für die $f(A) = w(A)$ gilt.

Endlich haben wir die Voraussetzungen, das Max-Flow-Min-Cut-Theorem zu beweisen. Wir zeigen, dass die folgenden beiden Aussagen äquivalent[5] sind:

(1) $f(A) = w(A)$ für einen s-t-Schnitt A und einen Fluss f.

(2) Es gibt keinen Erweiterungspfad von s nach t in G_f

Können wir zeigen, dass diese beiden Aussagen äquivalent sind, haben wir die Korrektheit des Ford-Fulkerson-Algorithmus gezeigt: Kann der Algorithmus keinen Erweiterungspfad mehr finden, so können wir sicher sein, dass der maximale Fluss gefunden wurde.

Der Beweis gliedert sich in zwei Teile:

(1)⇒(2) Wir nehmen also an, $f(A) = w(A)$. Im Restnetzwerk G_f gilt folglich, dass $w_f(i, j) = 0$ für alle (i, j) mit $i \in A$ und $j \in V \setminus A$. Folglich ist kein Knoten in $V \setminus A$ von einem Knoten aus A aus erreichbar, insbesondere ist t nicht von s aus erreichbar, d. h. es gibt keinen Erweiterungspfad von s nach t in G_f.

(2)⇒(1) Gibt es keinen Erweiterungspfad von s nach t in G_f, so wähle man $A = \{i \in V \mid i \text{ ist von } s \text{ aus erreichbar}\}$, d. h. der Schnitt A bestehe aus allen von s aus erreichbaren Knoten. Für alle Knoten $i \in A$ und $j \in V \setminus A$ muss also $w_f(i, j) = 0$ sein. Aus der Art und Weise wie das Restnetzwerk G_f konstruiert wird, folgt auch, dass $w_f(i, j) = w(i, j) - f(i, j)$. Also gilt $w(i, j) - f(i.j) = 0 \Leftrightarrow w(i, j) = f(i, j)$ für alle Kanten $(i, j) \in e(A)$. Also ist auch $w(A) = f(A) = f$ und somit ist f der maximal mögliche Fluss in G.

[5] Wenn man behauptet zwei Aussagen A und B seien *äquivalent*, so meint man, dass beide Aussagen „gleichbedeutend“ seien, d. h. wenn die Aussage A wahr ist, dann ist auch B wahr, und wenn die Aussage B wahr ist, dann ist auch A wahr.