Kapitel 4
Bäume und Matchings

Wir haben im letzten Kapitel Bäume implizit als Ergebnis unserer Suchverfahren kennengelernt. In diesem Kapitel wollen wir diese Graphenklasse ausführlich untersuchen.

4.1 Definition und Charakterisierungen

Die in den Suchverfahren konstruierten Graphen waren zusammenhängend und enthielten keine Kreise. Also vereinbaren wir:

Definition 4.1. Ein zusammenhängender Graph $T = (V,E)$, der keinen Kreis enthält, heißt *Baum (*engl. *tree*).

Wenn ein Graph keinen Kreis enthält, muss jeder maximale Weg zwangsläufig in einer „Sackgasse“ enden. Eine solche Sackgasse in einem Graphen nennen wir ein *Blatt*.

Definition 4.2. Sei $G = (V,E)$ ein Graph und $v \in V$ mit $\deg(v) = 1$. Dann nennen wir v ein *Blatt* von G.

Genauer haben wir in einem Graphen ohne Kreis sogar immer mindestens zwei Blätter.

Lemma 4.1. *Jeder Baum mit mindestens zwei Knoten hat mindestens zwei Blätter.*

Beweis. Da der Baum zusammenhängend ist und mindestens zwei Knoten hat, enthält er Wege der Länge mindestens 1. Sei $P = (v_1, \ldots, v_k)$ ein möglichst langer Weg in T. Da T kreisfrei ist, ist v_1 zu keinem von $v_3, \ldots, v_k$ adjazent. Dann muss $\deg(v_1)$ aber schon 1 sein, da man ansonsten P_k verlängern könnte. Die gleiche Argumentation gilt für v_k. □

Wenn wir an einem Baum ein Blatt „abzupfen“, bleibt er immer noch ein Baum. Gleiches gilt, wenn wir ein Blatt „ankleben“.

Lemma 4.2. *Sei $G = (V,E)$ ein Graph und v ein Blatt in G. Dann ist G ein Baum genau dann, wenn $G \setminus v$ ein Baum ist.*

Beweis.

„$\Rightarrow$“ Sei G ein Baum und v ein Blatt von G. Dann enthält kein Weg in G den Knoten v als inneren Knoten. Also ist $G \setminus v$ immer noch zusammenhängend und gewiss weiterhin kreisfrei.

W. Hochstättler, *Algorithmische Mathematik*, Springer-Lehrbuch
DOI 10.1007/978-3-642-05422-8_4,

„⇐“ Sei umgekehrt nun vorausgesetzt, dass $G \setminus v$ ein Baum ist. Da v ein Blatt ist, hat es einen Nachbarn u, von dem aus man in $G \setminus v$ alle Knoten erreichen kann, also ist G zusammenhängend. Offensichtlich kann v auf keinem Kreis liegen.

□

Lemma 4.2 ist nun ein wesentliches Hilfsmittel um weitere Eigenschaften, die Bäume charakterisieren, induktiv zu beweisen.

Satz 4.1. *Sei $T = (V,E)$ ein Graph und $|V| \geq 2$. Dann sind paarweise äquivalent:*

a) T ist ein Baum.
b) Zwischen je zwei Knoten $v,w \in V$ gibt es genau einen Weg von v nach w.
c) T ist zusammenhängend und für alle $e \in E$ ist $T \setminus e$ unzusammenhängend.
d) T ist kreisfrei und für alle $\bar{e} \in \binom{V}{2} \setminus E$ enthält $T + \bar{e}$ einen Kreis.
e) T ist zusammenhängend und $|E| = |V| - 1$.
f) T ist kreisfrei und $|E| = |V| - 1$.

Beweis. Ist $|V| = 2$, so sind alle Bedingungen dann und nur dann erfüllt, wenn G isomorph zum K_2 ist. Man beachte, dass es in der Bedingung d) eine Kante $\bar{e} \in \binom{V}{2} \setminus E$ nicht gibt, weswegen die Bedingung trivialerweise erfüllt ist.

Wir fahren fort per Induktion und nehmen an, dass $|V| \geq 3$ und die Gültigkeit der Äquivalenz für Graphen mit höchstens $|V| - 1$ Knoten bewiesen sei.

$a) \Rightarrow b)$ Seien also $v,w \in V$. Ist v oder w ein Blatt in G, so können wir o. E. annehmen, dass v ein Blatt ist, ansonsten vertauschen wir die Namen. Sei x der eindeutige Nachbar des Blattes v in T. Nach Lemma 4.2 ist $T \setminus v$ ein Baum. Also gibt es nach Induktionsvoraussetzung genau einen Weg von w nach x in $T \setminus v$. Diesen können wir mit (x,v) zu einem Weg von w nach v verlängern. Umgekehrt setzt sich jeder Weg von v nach w aus der Kante (x,v) und einem xw-Weg in $T \setminus v$ zusammen. Also gibt es auch höchstens einen vw-Weg in T.
Ist weder v noch w ein Blatt, so folgt die Behauptung per Induktion, wenn wir ein beliebiges Blatt aus G entfernen.

$b) \Rightarrow c)$ Wenn es zwischen je zwei Knoten einen Weg gibt, ist der Graph zusammenhängend. Sei $e = (v,w) \in E$. Gäbe es in $T \setminus e$ einen vw-Weg, dann gäbe es in T deren zwei, da e schon einen vw-Weg bildet. Also muss $T \setminus e$ unzusammenhängend sein.

$c) \Rightarrow d)$ Wenn es in T einen Kreis gibt, so kann man jede beliebige Kante dieses Kreises entfernen, ohne den Zusammenhang zu zerstören, da diese Kante in jedem Spaziergang durch den Rest des Kreises ersetzt werden kann. Da aber das Entfernen einer beliebigen Kante nach Voraussetzung in $c)$ den Zusammenhang zerstört, muss T kreisfrei sein. Die Aussage in $c)$ verbietet also die Existenz eines Kreises.
Sei $\bar{e} = (v,w) \in \binom{V}{2} \setminus E$. Da T zusammenhängend ist, gibt es in T einen vw-Weg, der mit der Kante $\bar{e}$ einen Kreis in $T + \bar{e}$ bildet.

$d) \Rightarrow a)$ Wir müssen zeigen, dass T zusammenhängend und kreisfrei ist. Letzteres wird in $d)$ explizit vorausgesetzt. Wenn T nicht zusammenhängend wäre, so könnte man eine Kante zwischen zwei Komponenten einfügen, ohne einen Kreis zu erzeugen. Also muss T zusammenhängend sein.

$a) \Rightarrow e), f)$ Nach Voraussetzung ist T sowohl zusammenhängend als auch kreisfrei. Sei v ein Blatt in T. Nach Lemma 4.2 ist $T \setminus v$ ein Baum. Nach Induktionsvoraussetzung ist also $|E(T \setminus v)| = |V(T \setminus v)| - 1$. Nun ist aber $|E(T)| = |E(T \setminus v)| + 1 = |V(T \setminus v)| = |V(T)| - 1$.

$a) \Leftarrow e)$ Da T nach Voraussetzung zusammenhängend ist und $|V| \geq 3$, haben wir $\deg(v) \geq 1$ für alle $v \in V$ und nach dem Handshake Lemma Proposition 3.6

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

Angenommen alle Knoten hätten Valenz $\deg_G(v) \geq 2$, so müsste

$$\sum_{v \in V} \deg(v) \geq 2|V|$$

sein. Da dies nicht so ist, aber G zusammenhängend und nicht trivial ist, muss es einen Knoten mit $\deg(v) = 1$, also ein Blatt in T, geben. Dann ist $T \setminus v$ weiterhin zusammenhängend und $|E(T \setminus v)| = |V(T \setminus v)| - 1$. Nach Induktionsvoraussetzung ist also $T \setminus v$ ein Baum und somit nach Lemma 4.2 auch T ein Baum.

$a) \Leftarrow f)$ Da T nach Voraussetzung kreisfrei und $|V| \geq 3$ ist, hat T mindestens eine Kante. Angenommen T hätte kein Blatt. Dann könnten wir an einem beliebigen Knoten einen Weg starten und daraufhin jeden Knoten durch eine andere Kante verlassen, als wir sie betreten haben. Da $|V|$ endlich ist, müssen dabei Knoten wiederholt auftreten, was wegen der Kreisfreiheit nicht möglich ist. Also hat T ein Blatt und wir können wie in $a) \Leftarrow e)$ schließen.

□

Aufgabe 4.2. Sei $T = (V, E)$ ein Baum und $\bar{e} \in \binom{V}{2} \setminus E$. Zeigen Sie:

a) $\bar{e}$ schließt genau einen Kreis C mit T, den wir mit $C(T, \bar{e})$ bezeichnen.
b) Für alle $e \in C(T, \bar{e}) \setminus \bar{e}$ ist $(T + \bar{e}) \setminus e$ ein Baum.

Lösung siehe Lösung 9.33.

4.2 Isomorphismen von Bäumen

Im Gegensatz zu der Situation bei allgemeinen Graphen, bei denen angenommen wird, dass die Isomorphie ein algorithmisch schweres Problem ist, kann man bei Bäumen (und einigen anderen speziellen Graphenklassen) die Isomorphie zweier solcher Graphen effizient testen.

Wir stellen in diesem Abschnitt einen Algorithmus vor, der zu jedem Baum mit n Knoten einen $2n$-stelligen Klammerausdruck berechnet, den wir als den *Code* des Graphen bezeichnen. Dieser Code zweier Bäume ist genau dann gleich, wenn die Bäume isomorph sind.

Zunächst ist folgendes Konzept hilfreich, das wir implizit schon bei der Breitensuche kennengelernt haben.

Definition 4.3. Ein *Wurzelbaum* oder eine *Arboreszenz* ist ein Paar (T, r) bestehend aus einem Baum T und einem ausgezeichneten Knoten $r \in V$, den wir als *Wurzelknoten* bezeichnen. Wir denken uns dann alle Kanten des Baumes so orientiert, dass die Wege von r zu allen anderen Knoten v

gerichtete Wege sind. Ist dann (v,w) ein Bogen, so sagen wir v ist *Elternteil* von w und w ist *Kind* oder *direkter Nachfahre* von v.

Aufgabe 4.3. Zeigen Sie: Ein zusammenhängender, gerichteter Graph $D = (V,A)$ ist genau dann ein Wurzelbaum, wenn es genau einen Knoten $r \in V$ gibt, so dass $\deg^+(r) = 0$ und für alle anderen Knoten $v \in V \setminus \{r\}$ gilt

$$\deg^+(v) = 1.$$

Lösung siehe Lösung 9.34.

Wir werden in unserem Algorithmus zunächst in einem Baum einen Knoten als Wurzel auszeichnen, so dass wir bei isomorphen Bäumen isomorphe Wurzelbäume erhalten. Diese Wurzelbäume pflanzen wir dann in die Zeichenebene, wobei wir wieder darauf achten, dass wir isomorphe Wurzelbäume isomorph einpflanzen. Gepflanzten Bäumen sieht man dann die Isomorphie fast sofort an.

Definition 4.4. Ein *gepflanzter Baum* (T,r,ρ) ist ein Wurzelbaum, bei dem an jedem Knoten $v \in V$ eine Reihenfolge $\rho(v)$ der direkten Nachfahren vorgegeben ist. Dadurch ist eine „Zeichenvorschrift“ definiert, wie wir den Graphen in die Ebene einzubetten haben.

Wie wir eben schon angedeutet haben, kann man für jede dieser Baumklassen mit zusätzlicher Struktur Isomorphismen definieren. Ein Isomorphismus zweier Wurzelbäume $(T,r),(T',r')$ ist ein Isomorphismus von T und T', bei dem r auf r' abgebildet wird. Ein Isomorphismus gepflanzter Bäume ist ein Isomorphismus der Wurzelbäume, bei dem zusätzlich die Reihenfolge der direkten Nachfahren berücksichtigt wird.

Die Bäume in Abbildung 4.1 sind alle paarweise isomorph als Bäume, die beiden rechten sind isomorph als Wurzelbäume, und keine zwei sind isomorph als gepflanzte Bäume.

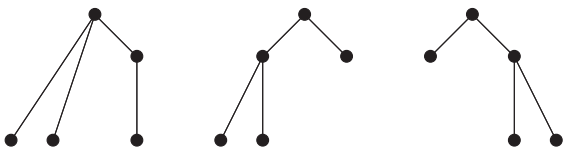


Abb. 4.1 Gepflanzte Bäume

Wie angekündigt gehen wir nun in drei Schritten vor.

a) Zu einem gegebenen Baum bestimmen wir zunächst eine Wurzel.
b) Zu einem Wurzelbaum bestimmen wir eine kanonische Pflanzung.
c) Zu einem gepflanzten Baum bestimmen wir einen eindeutigen Code.

Da sich der erste und der zweite Schritt leichter darstellen lassen, wenn der dritte bekannt ist, stellen wir dieses Verfahren von hinten nach vorne vor.

Sei also (T,r,ρ) ein gepflanzter Baum. Wir definieren den Code „Bottom-Up“ für jeden Knoten, indem wir ihn zunächst für Blätter erklären und dann für gepflanzte Bäume, bei denen alle Knoten außer der Wurzel schon einen Code haben. Dabei identifizieren wir den Code eines Knotens x mit

dem Code des gepflanzten Baumes, der durch den Ausgangsbaum auf x und allen seinen (nicht notwendigerweise direkten) Nachfahren induziert wird.

- Alle Blätter haben den Code $()$.
- Ist x ein Knoten mit Kindern in der Reihenfolge $y_1, \ldots, y_k$, deren Codes $C_1, \ldots, C_k$ sind, so erhält x den Code $(C_1 C_2 \ldots C_k)$.

Wir können nun aus dem Code den gepflanzten Baum wieder rekonstruieren. Dazu stellen wir zunächst fest, dass wir durch obiges Verfahren nur wohlgeklammerte Ausdrücke erhalten.

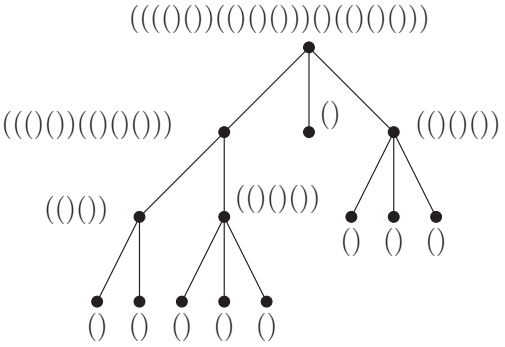
Definition 4.5. Sei $C \in \{(,)\}^{2m}$ eine Zeichenkette aus Klammern. Dann nennen wir C *wohlgeklammert*, wenn C gleich viele öffnende wie schließende Klammern enthält und mit einer öffnenden Klammer beginnt, welche erst mit der letzten Klammer geschlossen wird.

Beispiel 4.4. Der Ausdruck $C_1 = (((\,)(\,)))$ ist wohlgeklammert, aber $C_2 = ((\,))((\,)(\,))$ und $C_3 = (\,))((\,$ sind nicht wohlgeklammert.

Aufgabe 4.5. Der Code eines gepflanzten Baumes ist ein wohlgeklammerter Ausdruck.
Lösung siehe Lösung 9.35.

Wenn wir nun einen wohlgeklammerten Ausdruck haben, erhalten wir rekursiv einen gepflanzten Baum wie folgt.

- Zeichne eine Wurzel r.
- Streiche die erste (öffnende) Klammer.
- Solange das nächste Zeichen eine öffnende Klammer ist
  - Suche die entsprechende schließende Klammer, schreibe die so definierte Zeichenkette C_i bis hierhin raus, hänge die Wurzel y_i des durch C_i definierten gepflanzten Wurzelbaums als rechtes Kind an r an und lösche C_i aus C.
  - Streiche die letzte (schließende) Klammer.



In dem Beispiel in der Abbildung erhalten wir als Codes der Kinder der Wurzel auf diese Weise völlig zu Recht $C_1 = (((\,)(\,))(\,)(\,)))$, $C_2 = (\,)$, $C_3 = ((\,)(\,))$. Beachten Sie, dass dies nicht der Code des obigen Baumes ist, da der obige Baum nicht kanonisch gepflanzt ist.

Aufgabe 4.6. Sei (T,r,ρ) ein gepflanzter Baum und C der Code von (T,r,ρ). Zeigen Sie: Mittels der soeben beschriebenen rekursiven Prozedur erhalten wir einen gepflanzten Baum, der isomorph zu (T,r,ρ) ist.
Lösung siehe Lösung 9.36.

Eine anschauliche Interpretation des Codes eines gepflanzten Baumes erhält man, wenn man den geschlossenen Weg betrachtet, der an der Wurzel mit der Kante nach links unten beginnt und dann außen um den Baum herumfährt. Jedesmal, wenn wir eine Kante abwärts fahren, schreiben wir eine öffnende Klammer, und eine schließende Klammer, wenn wir eine Kante aufwärts fahren. Schließlich machen wir um den ganzen Ausdruck noch ein Klammerpaar für die Wurzel.

Da wir nach Aufgabe 4.6 so den gepflanzten Baum (bis auf Isomorphie) aus C rekonstruieren können, haben nicht isomorphe gepflanzte Bäume verschiedene Codes. Umgekehrt bleibt der Code eines gepflanzten Baumes unter einem Isomorphismus offensichtlich invariant, also haben isomorphe gepflanzte Bäume den gleichen Code.

Wir übertragen diesen Code nun auf Wurzelbäume, indem wir die Vorschrift modifizieren. Zunächst erinnern wir an die lexikographische Ordnung aus Aufgabe 3.10. Durch „(“ $<$ „)“ erhalten wir eine Totalordnung auf $\{(,)\}$ und damit eine lexikographische Ordnung auf den Klammerstrings.

Dann ist ein Klammerstring A lexikographisch kleiner als ein anderer B, in Zeichen $A \preceq B$, wenn entweder A der Anfang von B ist oder die erste Klammer, in der die beiden Wörter sich unterscheiden, bei A öffnend und bei B schließend ist. Z. B. ist $(()) \preceq ()$.

Die Wahl dieser Totalordnung ist hier willkürlich. Unser Algorithmus funktioniert mit jeder Totalordnung auf den Zeichenketten.

Wir definieren nun unseren Code auf Wurzelbäumen Bottom-Up wie folgt:

- Alle Blätter haben den Code $()$.
- Ist x ein Knoten mit Kindern, deren Codes bekannt sind, so sortiere die Kinder so zu $y_1,\ldots,y_k$, dass für die zugehörigen Codes gilt $C_1 \preceq C_2 \preceq \ldots \preceq C_k$.
- x erhält dann den Code $(C_1C_2\ldots C_k)$.

Diese Vereinbarung definiert auf den Knoten eine Reihenfolge der Kinder, macht also auf eindeutige Weise aus einem Wurzelbaum einen gepflanzten Baum.

Aufgabe 4.7. Zeigen Sie: Isomorphe Wurzelbäume erhalten so den gleichen Code.
Lösung siehe Lösung 9.37.

Kommen wir nun zu den Bäumen. Wir versuchen zunächst von einem gegebenen Baum einen Knoten zu finden, der sich als Wurzel aufdrängt und unter Isomorphismen fix bleibt. Ein solcher Knoten soll in der Mitte des Baumes liegen. Das zugehörige Konzept ist auch auf allgemeinen Graphen sinnvoll.

Definition 4.6. Sei $G=(V,E)$ ein Graph und $v \in V$. Als *Exzentrizität* $ex_G(v)$ bezeichnen wir die Zahl

$$ex_G(v) = \max\{dist_G(v,w) \mid w \in V\}, \tag{4.1}$$

also den größten Abstand zu einem anderen Knoten.

Das *Zentrum* $Z(G)$ ist die Menge der Knoten minimaler Exzentrizität

$$Z(G) = \{v \in V \mid ex_G(v) = \min\{ex_G(w) \mid w \in V\}\}. \tag{4.2}$$

Ist das Zentrum unseres Baumes ein Knoten, so wählen wir diesen als Wurzel. Ansonsten nutzen wir aus:

Lemma 4.3. *Sei $T = (V,E)$ ein Baum. Dann ist $|Z(T)| \leq 2$. Ist $Z(T) = \{x,y\}$ mit $x \neq y$, so ist $(x,y) \in E$.*

Beweis. Wir beweisen dies mittels vollständiger Induktion über $|V|$. Die Aussage ist sicherlich richtig für Bäume mit einem oder zwei Knoten. Ist nun $|V| \geq 3$, so ist nach Satz 4.1 e) $\sum_{v \in V} \deg(v) = 2|V| - 2 > |V|$, also können nicht alle Knoten Blätter sein. Entfernen wir alle Blätter aus T, so erhalten wir einen nicht leeren Baum T' auf einer Knotenmenge $V' \subset V$, die echt kleiner geworden ist. In einem Graphen mit mindestens drei Knoten kann kein Blatt im Zentrum liegen, da die Exzentrizität seines Nachbarn um genau 1 kleiner ist. Also ist $Z(T) \subseteq V'$, und für alle Knoten in $w \in V'$ gilt offensichtlich

$$ex_{T'}(w) = ex_T(w) - 1.$$

Folglich ist $Z(T) = Z(T')$ und dieses hat nach Induktionsvoraussetzung höchstens zwei Elemente. Sind es genau zwei Elemente, so müssen diese adjazent sein. □

Besteht das Zentrum aus zwei Knoten $\{x_1,x_2\}$, so entfernen wir die verbindende Kante (x_1,x_2), bestimmen die Codes der in x_1 bzw. x_2 gewurzelten Teilbäume und wählen den Knoten als Wurzel von T, dessen Teilbaum den lexikographisch kleineren Code hat. Wir fassen zusammen:

- Ist $Z(G) = \{v\}$, so ist der Code von T der Code von (T,v).
- Ist $Z(G) = \{x_1,x_2\}$ mit $x_1 \neq x_2$, so sei $e = (x_1,x_2)$. Seien T_1,T_2 die Komponenten von $T \setminus e$ mit $x_1 \in T_1$ und $x_2 \in T_2$. Sei C_i der Code des Wurzelbaumes (T_i,x_i) und die Nummerierung der Bäume so gewählt, dass $C_1 \preceq C_2$. Dann ist der Code von T der Code des Wurzelbaumes (T,x_1).

Satz 4.8. *Zwei Bäume haben genau dann den gleichen Code, wenn sie isomorph sind.*

Beweis. Sind zwei Bäume nicht isomorph, so sind auch alle zugehörigen gepflanzten Bäume nicht isomorph, also die Codes verschieden. Sei für die andere Implikation $\varphi : V \to V'$ ein Isomorphismus von $T = (V,E)$ nach $T' = (V,E')$. Seien r,r' die bei der Konstruktion der Codes ausgewählten Wurzeln von T bzw. T'. Ist $\varphi(r) = r'$, so sind die Wurzelbäume (T,r) und (T',r') isomorph und haben nach Aufgabe 4.7 den gleichen Code. Andernfalls besteht das Zentrum von T aus zwei Knoten r,s und $\varphi(s) = r'$. Dann ist aber der Code des in r' gewurzelten Teilbaumes (T'_1,r') von $T' \setminus (\varphi(r),r')$ lexikographisch kleiner als der des in $\varphi(r)$ gewurzelten Teilbaumes $(T'_2,\varphi(r))$. Letzterer ist aber isomorph zu dem in r gewurzelten Teilbaum (T_2,r) von $T \setminus (r,s)$, welcher also nach Aufgabe 4.7 den gleichen Code wie $(T'_2,\varphi(r))$ hat. Da aber r als Wurzel ausgewählt wurde, ist dieser Code lexikographisch kleiner als der des in s gewurzelten Teilbaumes (T_1,s). Dessen Code ist aber wiederum nach Aufgabe 4.7 gleich dem Code von (T'_1,r'). Also müssen alle diese Codes gleich sein. Somit sind (T_1,r) und (T_2,s) nach Aufgabe 4.7 isomorphe Wurzelbäume. Sei $\psi(V_1,V_2) : V_1 \to V_2$ ein entsprechender Isomorphismus. Betrachten wir $V = V_1 \dot\cup V_2$ als (V_1,V_2) bzw. (V_2,V_1), so vermittelt $(\psi,\psi^{-1}) : (V_1,V_2) \to (V_2,V_1) = V$ einen Automorphismus von T und $\varphi \circ \psi$ ist ein Isomorphismus von (T,r) nach (T,r'), also haben nach dem bereits Gezeigten T und T' denselben Code. □

4.3 Aufspannende Bäume

In diesem Abschnitt werden wir unter Anderem den fehlenden Teil des Beweises, dass der BFS die Komponenten eines Graphen berechnet, nachholen. Die dort berechneten Teilgraphen spannen die Ausgangsgraphen auf. Solche minimalen aufspannenden Teilgraphen bezeichnet man manchmal auch als Gerüste. Aber erst noch mal zur Definition:

Definition 4.7. Ein kreisfreier Graph heißt *Wald*. Sei $G = (V,E)$ ein Graph und $T = (V,F)$ ein Teilgraph, der die gleichen Zusammenhangskomponenten wie V hat. Dann sagen wir T ist G *aufspannend*. Ist T darüberhinaus kreisfrei, so heißt T ein G *aufspannender Wald* oder ein *Gerüst von* G. Ist G zusammenhängend und T ein Baum, so heißt T ein G *aufspannender Baum*.

Diese Definition ist offensichtlich auch für Multigraphen sinnvoll. Wir werden im Folgenden auch bei Multigraphen von aufspannenden Bäumen sprechen.

Wir analysieren nun zwei schnelle Algorithmen, die in einem zusammenhängenden Graphen einen aufspannenden Baum berechnen. Die im vorhergehenden Kapitel betrachteten Algorithmen BFS und DFS kann man als Spezialfälle des zweiten Verfahrens betrachten.

Die Methode `T.CreatingCycle(e)` überprüfe zu einer kreislosen Kantenmenge T mit $e \notin T$, ob $T+e$ einen Kreis enthält, `T.AddEdge(e)` füge zu T die Kante e hinzu.

Algorithmus 4.9. Sei E eine (beliebig sortierte) Liste der Kanten des Graphen (V,E) und zu Anfang $T = \emptyset$.

```
for e in E:
    if not T.CreatingCycle(e):
        T.AddEdge(e)
```

Lemma 4.4. *Algorithmus 4.9 berechnet einen G aufspannenden Wald.*

Beweis. Zu Anfang enthält T gewiss keinen Kreis. Da nie eine Kante hinzugefügt wird, die einen Kreis schließt, berechnet der Algorithmus eine kreisfreie Menge, also einen Wald T. Wir haben zu zeigen, dass zwischen zwei Knoten u,v genau dann ein Weg in T existiert, wenn er in G existiert. Eine Implikation ist trivial: wenn es einen Weg in T gibt, so gab es den auch in G. Sei also $u = v_0, v_1, \ldots, v_k = v$ ein uv-Weg P in G. Angenommen u und v lägen in unterschiedlichen Komponenten von T. Sei dann v_i der letzte Knoten auf P, der in T in der gleichen Komponente wie u liegt. Dann ist $e = (v_i v_{i+1}) \in E \setminus T$. Als e im Algorithmus abgearbeitet wurde, schloss e folglich mit T einen Kreis. Also gibt es in T einen Weg von v_i nach v_{i+1} im Widerspruch dazu, dass sie in verschiedenen Komponenten liegen. □

Betrachten wir die Komplexität des Algorithmus, so hängt diese von einer effizienten Implementierung des Kreistests ab. Eine triviale Implementierung dieser Subroutine in $O(|V|)$ Zeit labelt ausgehend von einem Endknoten u von $e = (u,v)$ alle Knoten, die in T von u aus erreichbar sind. Wird v gelabelt, so schließt e einen Kreis mit T und sonst nicht. Dies führt aber zu einer Gesamtlaufzeit von $O(|V| \cdot |E|)$. Um effizienter zu werden, müssen wir folgendes Problem schneller lösen:

Problem 4.1 (UNION-FIND). Sei $V = \{1,\ldots,n\}$ und eine initiale Partition in n triviale einelementige Klassen $V = \{1\} \dot\cup \ldots \dot\cup \{n\}$ gegeben. Wie sieht eine geeignete Datenstruktur aus, so dass man folgende Operationen effizient auf einer gegebenen Partition ausführen kann?

UNION Gegeben seien x, y aus verschiedenen Klassen, vereinige diese Klassen.
FIND Gegeben seien $x, y \in V$. Stelle fest, ob x und y in der gleichen Klasse liegen.

Was hat dieses Problem mit einer effizienten Implementierung von Algorithmus 4.9 zu tun? Zu jedem Zeitpunkt besteht T_i aus den Kanten eines Waldes. Die Knotenmengen seiner Zusammenhangskomponenten liefern die Klassen unserer Partition. Für den Kreistest genügt es dann zu prüfen, ob die Endknoten x, y der Kante (x, y) in der gleichen Klasse liegen. Ist dies nicht der Fall, so nehmen wir e in T_{i+1} auf und müssen die Klassen von x und y vereinigen.

Also benötigen wir für unseren Algorithmus zur Berechnung eines aufspannenden Waldes $|E|$ FIND und höchstens $|V| - 1$ UNION Operationen.

Wir stellen eine einfache Lösung dieses Problems vor. Jede Klasse hat eine Nummer, jeder Knoten die Nummer seiner Klasse. In einem Array speichern wir einen Zeiger auf die Klasse jedes Knotens. Die Klasse enthält eine Liste ihrer Knoten und zusätzlich einen Eintrag für die Anzahl der Elemente der Klasse. Bei einer nicht mehr existierenden Nummer ist der Eintrag 0. Für eine FIND-Operation benötigen wir dann nur einen Vergleich der Nummern der Klassen, also konstante Zeit. Bei einer UNION-Operation erbt die kleinere Komponente die Nummer der größeren, wir datieren die Nummern der Knoten in der kleineren Komponente auf und verschmelzen die Listen.

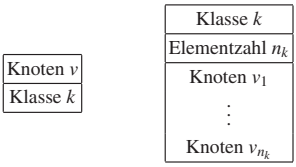


Abb. 4.2 Eine einfache UNION-FIND Datenstruktur

Lemma 4.5. *Die Kosten des Komponentenverschmelzens über den gesamten Lauf des Algorithmus betragen akkumuliert* $O(|V| \log |V|)$.

Beweis. Wir beweisen dies mit vollständiger Induktion über $n = |V|$. Für $n = 1$ ist nichts zu zeigen. Verschmelzen wir zwei Komponenten T_1 und T_2 der Größe $n_1 \leq n_2$ mit $n = n_1 + n_2$, dann ist $n_1 \leq \frac{n}{2}$ und das Update kostet cn_1. Addieren wir dies zu den Kosten für das Verschmelzen der einzelnen Knoten zu T_1 und T_2, die nach Induktionsvoraussetzung bekannt sind, erhalten wir

$$\begin{aligned} cn_1 + cn_1 \log_2 n_1 + cn_2 \log_2 n_2 &\leq cn_1 + cn_1 \log_2 \frac{n}{2} + cn_2 \log_2 n \\ &= cn_1 + cn_1 (\log_2 n - 1) + cn_2 \log_2 n \\ &= cn \log_2 n. \end{aligned}$$

□

Bemerkung 4.10. Die beste bekannte UNION-FIND Struktur geht auf R. Tarjan zurück. Die Laufzeit ist dann beinahe linear. Man hat als Laufzeitkoeffizienten zusätzlich noch die so genannte *Inverse der Ackermann-Funktion*, eine Funktion, die zwar gegen Unendlich wächst, aber viel langsamer als $\log n$, $\log \log n$ etc. (siehe etwa [9]).

Unser zweiter Algorithmus sieht in etwa aus wie eine allgemeinere Version des Breadth-First-Search Algorithmus. Auf Grund dieser Allgemeinheit beschreiben wir ihn nur verbal.

Algorithmus 4.11. Sei $v \in V$.

- Setze $V_0 = \{v\}$, $T_0 = \emptyset$, $i = 0$
- Solange es geht

  Wähle eine Kante $e = (x,y) \in E$ mit $x \in V_i$, $y \notin V_i$ und setze $V_{i+1} = V_i \cup \{y\}$, $T_{i+1} = T_i \cup \{e\}$, $i = i+1$.

Lemma 4.6. *Wenn Algorithmus 4.11 endet, dann ist $T = T_i$ aufspannender Baum der Komponente von G, die v enthält.*

Beweis. Die Kantenmenge T ist offensichtlich zusammenhängend und kreisfrei und verbindet alle Knoten in V_i. Nach Konstruktion gibt es keine Kante mehr, die einen Knoten aus V_i mit einem weiteren Knoten verbindet. □

Zwei Möglichkeiten, diesen Algorithmus zu implementieren, haben wir mit BFS und DFS kennengelernt und damit an dieser Stelle deren Korrektheitsbeweise nachgeholt. Der zu BFS angegebene Algorithmus startet allerdings zusätzlich in jedem Knoten und überprüft, ob dieser in einer neuen Komponente liegt. Indem wir Lemma 4.6 in jeder Komponente anwenden, haben wir auch den fehlenden Teil des Beweises von Satz 3.27 nachgeholt.

4.4 Minimale aufspannende Bäume

Wir wollen nun ein einfaches Problem der „Kombinatorischen Optimierung“ kennenlernen. Die Kanten unseres Graphen sind zusätzlich mit Gewichten versehen. Sie können sich diese Gewichte als Längen oder Kosten der Kanten vorstellen.

Betrachten wir etwa das Problem, eine Menge von Knoten kostengünstigst durch ein Netzwerk zu verbinden. Dabei sind zwei Knoten miteinander verbunden, wenn es im Netzwerk einen Weg – eventuell mit Zwischenknoten – vom einen zum anderen Knoten gibt.

Uns sind die Kosten der Verbindung zweier Nachbarn im Netzwerk bekannt, und wir wollen jeden Knoten von jedem aus erreichbar machen und die Gesamtkosten minimieren.

Als abstraktes Problem erhalten wir dann das Folgende:

Problem 4.2. Sei $G = (V,E)$ ein zusammenhängender Graph und $w : E \to \mathbb{N}$ eine nichtnegative Kantengewichtsfunktion. Bestimme einen aufspannenden Teilgraphen $T = (V,F)$, so dass

$$w(F) := \sum_{e \in F} w(e) \tag{4.3}$$

minimal ist.

Bemerkung 4.12. Bei den Überlegungen zu Problem 4.2 macht es keinen wesentlichen Unterschied, ob die Gewichtsfunktion ganzzahlig oder reell ist. Da wir im Computer mit beschränkter Stellenzahl rechnen, können wir im praktischen Betrieb sowieso nur mit rationalen Gewichtsfunktionen umgehen. Multiplizieren wir diese mit dem Hauptnenner, ändern wir nichts am Verhältnis der Kosten,

insbesondere bleiben Optimallösungen optimal. Also können wir in der Praxis o. E. bei Problem 4.2 stets von ganzzahligen Daten ausgehen.

Da die Gewichtsfunktion nicht-negativ ist, können wir, falls eine Lösung Kreise enthält, aus diesen so lange Kanten entfernen, bis die Lösung kreisfrei ist, ohne höhere Kosten zu verursachen. Also können wir uns auf folgendes Problem zurückziehen:

Problem 4.3 (Minimaler aufspannender Baum (MST, von engl. Minimum Spanning Tree)). Sei $G = (V,E)$ ein zusammenhängender Graph und $w : E \to \mathbb{N}$ eine nichtnegative Kantengewichtsfunktion. Bestimme einen G aufspannenden Baum $T = (V,F)$ minimalen Gewichts $w(F)$.

Der vollständige Graph K_n mit n Knoten hat n^{n-2} aufspannende Bäume, wie wir in Abschnitt 4.6 sehen werden. Eine vollständige Aufzählung ist also kein effizientes Verfahren. Ein solches gewinnen wir aber leicht aus Algorithmus 4.9. Der folgende Algorithmus heißt Greedy-Algorithmus (greedy ist englisch für gierig), weil er stets lokal den besten nächsten Schritt tut. Eine solche Strategie ist nicht immer zielführend, in diesem Falle aber schon, wie wir sehen werden. Der lokal beste nächste Schritt ist hier die leichteste Kante, die mit dem bereits erzeugten Graphen keinen Kreis schließt. Also sortieren wir zunächst die Kanten nicht-absteigend und wenden dann Algorithmus 4.9 an.

Algorithmus 4.13 (Greedy-Algorithmus (Kruskal)). Sortiere die Kanten so, dass

$$w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$$

und führe Algorithmus 4.9 aus.

Satz 4.14. *Der Greedy-Algorithmus berechnet einen minimalen aufspannenden Baum.*

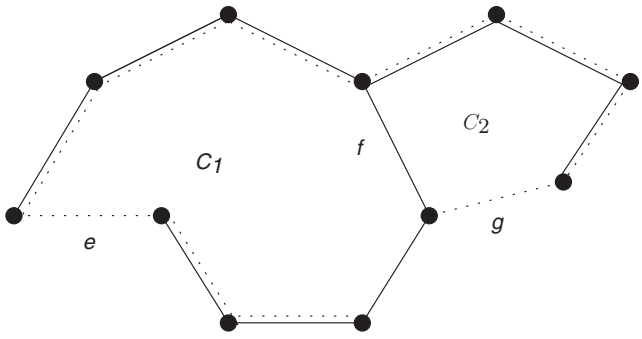


Abb. 4.3 Zum Beweis von Satz 4.14. Kanten in $\tilde{T}$ sind durchgezogen gezeichnet, die in T gestrichelt.

Beweis. Als spezielle Implementierung von Algorithmus 4.9 berechnet der Greedy-Algorithmus einen aufspannenden Baum T. Angenommen es gäbe einen aufspannenden Baum $\tilde{T}$ mit $w(\tilde{T}) < w(T)$. Sei dann ein solches $\tilde{T}$ so gewählt, dass $|T \cap \tilde{T}|$ maximal ist. Sei e die Kante mit kleinstem Gewicht in $T \setminus \tilde{T}$. Dann schließt e in $\tilde{T}$ nach Satz 4.1 einen Kreis C_1 (siehe Abbildung 4.3). Nach Wahl von $\tilde{T}$ muss nun $w(f) < w(e)$ für alle $f \in C_1 \setminus T$ sein, denn sonst könnte man durch Ersetzen eines solchen f mit $w(f) \geq w(e)$ durch e einen Baum $\hat{T} = (\tilde{T} \setminus f) + e$ konstruieren mit

$w(\hat{T}) \leq w(\tilde{T}) < w(T)$ und $|T \cap \hat{T}| > |T \cap \tilde{T}|$. Sei nun $f \in C_1 \setminus T$. Da f vom Greedy-Algorithmus verworfen wurde, schließt es mit T einen Kreis C_2. Da die Kanten nach aufsteigendem Gewicht sortiert wurden, gilt für alle $g \in C_2 : w(g) \leq w(f)$. Sei $g \in C_2 \setminus \tilde{T}$. Dann ist $g \in T \setminus \tilde{T}$ und $w(g) \leq w(f) < w(e)$. Also hat g ein kleineres Gewicht als e im Widerspruch zur Wahl von e.
□

Aufgabe 4.15. Sei $G = (V,E)$ ein zusammenhängender Graph, $w : E \to \mathbb{Z}$ eine Kantengewichtsfunktion und $H = (V,T)$ ein G aufspannender Baum. Zeigen Sie: H ist genau dann ein minimaler G aufspannender Baum, wenn

$$\forall \bar{e} \in E \setminus T \; \forall e \in C(T,\bar{e}) : w(e) \leq w(\bar{e}),$$

wenn also in dem nach Aufgabe 4.2 eindeutigen Kreis $C(T,\bar{e})$, den $\bar{e}$ mit T schließt, keine Kante ein größeres Gewicht als $\bar{e}$ hat. Diese Bedingung ist als *Kreiskriterium* bekannt.
Lösung siehe Lösung 9.38.

Bemerkung 4.16. Man kann im Allgemeinen n Zahlen in $O(n \log n)$ Zeit sortieren, und man kann zeigen, dass es schneller im Allgemeinen nicht möglich ist. Wir wollen im Folgenden diese Aussage ohne Beweis voraussetzen und benutzen (siehe etwa [19, 9]).

Also benötigen wir hier für das Sortieren der Kanten $O(|E| \log(|E|))$, und erhalten wegen

$$O(\log(|E|)) = O(\log(|V|^2)) = O(\log(|V|))$$

mit unserer Implementierung von UNION-FIND ein Verfahren der Komplexität $O((|E| + |V|) \log(|V|))$.

4.5 Die Algorithmen von Prim-Jarnik und Borůvka

Auch aus Algorithmus 4.11 können wir ein Verfahren ableiten, um minimale aufspannende Bäume zu berechnen. Dieser Algorithmus ist nach Robert C. Prim benannt, der ihn 1957 wiederentdeckte. Die erste Veröffentlichung dieses Verfahrens von Vojtech Jarnik war auf Tschechisch.

Algorithmus 4.17 (Prims Algorithmus). Sei $v \in V$.

- Setze $V_0 = \{v\}$, $T_0 = \emptyset$, $i = 0$
- Solange es geht
  - Wähle eine Kante $e = (x,y) \in E$ mit $x \in V_i$, $y \notin V_i$ von minimalem Gewicht und setze $V_{i+1} = V_i \cup \{y\}$, $T_{i+1} = T_i \cup \{e\}$, $i = i+1$.

Bevor wir diskutieren, wie wir effizient die Kante minimalen Gewichts finden, zeigen wir zunächst einmal die Korrektheit des Verfahrens.

Satz 4.18. *Prims Algorithmus berechnet einen minimalen aufspannenden Baum.*

Beweis. Als Spezialfall von Algorithmus 4.11 berechnet Prims Algorithmus einen aufspannenden Baum. Im Verlauf des Algorithmus haben wir auch stets einen Baum, der v enthält. Dieser erhält in jeder Iteration eine neue Kante. Wir zeigen nun mittels Induktion über die Anzahl der Iterationen:

Zu jedem Zeitpunkt des Algorithmus ist T_i in einem minimalen aufspannenden Baum enthalten.

Diese Aussage ist sicherlich zu Anfang für $T_0 = \emptyset$ wahr. Sei nun soeben die Kante e zu T_i hinzugekommen. Nach Induktionsvoraussetzung ist $T_i \setminus e$ in einem minimalen aufspannenden Baum $\hat{T}$ enthalten. Wenn dieser e enthält, so sind wir fertig. Andernfalls schließt e einen Kreis mit $\hat{T}$. Dieser enthält neben e mindestens eine weitere Kante f, die die Knotenmenge von $T_i \setminus e$ mit dem Komplement dieser Knotenmenge verbindet. Nach Wahl von e ist

$$w(e) \leq w(f)$$

und nach Aufgabe 4.2 ist $(\hat{T}+e)\setminus\{f\}$ ein aufspannender Baum und

$$w\big((\hat{T}+e)\setminus f\big) = w(\hat{T}) + w(e) - w(f) \leq w(\hat{T}).$$

Da $\hat{T}$ ein minimaler aufspannender Baum war, schließen wir $w(e) = w(f)$ und somit ist $(\hat{T}+e)\setminus f$ ein minimaler aufspannender Baum, der T_i enthält. □

Aufgabe 4.19. Sei $G = (V,E)$ ein zusammenhängender Graph. Ist $S \subseteq V$, so nennen wir die Kantenmenge

$$\partial_G(S) := \{e \in E \mid |e \cap S| = 1\}$$

den von S *induzierten Schnitt*. Allgemein nennen wir eine Kantenmenge D einen *Schnitt in* G, wenn es ein $S \subseteq V$ gibt mit $D = \partial_G(S)$. Sei nun ferner $w : E \to \mathbb{Z}$ eine Kantengewichtsfunktion und $H = (V,T)$ ein G aufspannender Baum. Zeigen Sie:

a) Für alle $e \in T$ ist die Menge

$$D(T,e) := \{\bar{e} \in E \mid (T \setminus e) + \bar{e} \text{ ist ein Baum}\}$$

ein Schnitt in G.

b) H ist genau dann ein minimaler G aufspannender Baum, wenn

$$\forall e \in T \, \forall \bar{e} \in D(T,e) : w(e) \leq w(\bar{e}),$$

wenn also e eine Kante mit kleinstem Gewicht ist, die die Komponenten von $T \setminus e$ miteinander verbindet. Diese Bedingung ist als *Schnittkriterium* bekannt.

Lösung siehe Lösung 9.39.

Kommen wir zur Diskussion der Implementierung von Prims Algorithmus. Sicherlich wollen wir nicht in jedem Schritt alle Kanten überprüfen, die aus T herausführen. Statt dessen merken wir uns stets die kürzeste Verbindung aus T zu allen Knoten außerhalb von T in einer Kantenmenge `F`. Zu dieser Kantenmenge haben wir als neue Methode `F.MinimumEdge(weight)`, die aus `F` eine Kante minimalen Gewichts liefert. Wenn wir diese Datenstruktur aufdatieren, müssen wir nur alle Kanten, die aus dem neuen Baumknoten herausführen, daraufhin überprüfen, ob sie eine kürzere Verbindung zu ihrem anderen Endknoten aus T heraus herstellen. Wir erhalten also folgenden Code:

```
T=[]
F=[]
for w in G.Neighborhood(v):
```

```
    F.AddEdge((v,w))
    pred[w] = v
while not T.IsSpanning():
    (u,v) = F.MinimumEdge(weight)
    F.DeleteEdge((u,v))
    T.AddEdge((u,v))
    for w in G.Neighborhood(v):
        if not T.Contains(w) and weight[(pred[w],w)] > weight[(w,v)]:
            F.DeleteEdge((pred[w],w))
            F.AddEdge((w,v))
            pred[w] = v
```

Dabei gehen wir davon aus, dass vor Ausführung des Algorithmus die Felder `pred` für alle Knoten mit `pred[v]=v` und `weight[(v,v)]` mit unendlich initialisiert worden sind. Die Kanten fassen wir hier als gerichtet auf, dass heißt in `(u,v) = F.MinimumEdge(weight)` ist `u` ein Knoten innerhalb des Baumes und `v` ein Knoten auf der „anderen“ Seite.

Auf diese Weise wird jede Kante in genau einer der beiden **for**-Schleifen genau einmal untersucht. Die Kosten für das Aufdatieren von F sind also $O(|E|)$. Die **while**-Schleife wird nach Satz 4.1 genau $(|V|-1|)$- mal durchlaufen. Für die Laufzeit bleibt die Komplexität von `F.minimumEdge(weight)` zu betrachten. Hier wird aus einer Menge von $O(|V|)$ Kanten das Minimum bestimmt. Man kann nun die Daten, etwa in einer so genannten *Priority Queue* so organisieren, dass $|F|$ stets geordnet ist. Das Einfügen einer Kante in F kostet dann $O(\log|F|)$ und das Löschen und Finden des Minimums benötigt ebenso $O(\log|F|)$. Für Details verweisen wir auf [9]. Als Gesamtlaufzeit erhalten wir damit

$$O(|E|\log|V|).$$

Aufgabe 4.20. Zeigen Sie: In jeder Implementierung ist die Laufzeit von Prims Algorithmus von unten durch $\Omega(|V|\log|V|)$ beschränkt. Weisen Sie dazu nach, dass man mit Prims Algorithmus $|V|$ Zahlen sortieren kann.

Lösung siehe Lösung 9.40.

Als letztes stellen wir das älteste Verfahren vor, das schon 1926 von Otakar Borůvka, ebenfalls auf Tschechisch, publiziert wurde. Dazu zunächst noch eine vorbereitende Übungsaufgabe.

Aufgabe 4.21. Sei $G=(V,E)$ ein zusammenhängender Graph und $w: E \to \mathbb{Z}$ eine Kantengewichtsfunktion. Zeigen Sie:

a) Ist T ein minimaler G aufspannender Baum und $S \subseteq E(T)$, so ist $T \setminus S$ ein minimaler aufspannender Baum von G/S (vgl. Definition 3.16).
b) Ist darüberhinaus w injektiv, und sind also alle Kantengewichte verschieden so ist die Menge S der Kanten, die aus den (eindeutigen) Kanten kleinsten Gewichts an jedem Knoten besteht, in dem eindeutigen minimalen aufspannenden Baum enthalten.

Lösung siehe Lösung 9.41.

Der Algorithmus von Borůvka verfährt nun wie folgt. Wir gehen zunächst davon aus, dass $G=(V,E)$ ein Multigraph mit einer injektiven Gewichtsfunktion ist.

Algorithmus 4.22 (Borůvkas Algorithmus). Setze $T = \emptyset$.

- Solange G noch mehr als einen Knoten hat:

  Jeder Knoten markiert die Kante minimalen Gewichts, die zu ihm inzident und keine Schleife ist.
  Füge alle markierten Kanten S zu T hinzu und setze $G = G/S$.

Hierbei interpretieren wir die Kanten in T am Ende als Kanten des ursprünglichen Graphen G.

Satz 4.23. *Borůvkas Algorithmus berechnet den eindeutigen minimalen aufspannenden Baum von G.*

Beweis. Wir zeigen per Induktion über die Anzahl der Iterationen, dass T in jedem minimalen aufspannenden Baum enthalten ist. Solange T leer ist, ist dies gewiss richtig. Sei also T in jedem minimalen aufspannenden Baum enthalten und S wie beschrieben. Nach Aufgabe 4.21 b) ist S in dem eindeutigen aufspannenden Baum $\tilde{T}$ von G/T enthalten. Sei nun $\hat{T}$ ein minimaler aufspannender Baum von G, also $T \subseteq \hat{T}$. Nach Aufgabe 4.21 a) ist $\hat{T} \setminus T$ ein minimaler aufspannender Baum von G/T. Wir schließen $\hat{T} \setminus T = \tilde{T}$. Insgesamt erhalten wir wie gewünscht $S \cup T \subseteq \tilde{T}$.

Da in jedem Schritt die Anzahl der Knoten mindestens halbiert wird, berechnet man in höchstens $\log_2 |V|$ Schritten eine Kantenmenge T, die ein aufspannender Baum ist und in jedem minimalen aufspannenden Baum enthalten ist. Also ist dieser Baum eindeutig. □

Bemerkung 4.24. Die Bedingung, dass w injektiv ist, ist keine wirkliche Einschränkung. Wird ein Kantengewicht mehrfach angenommen, so kann man z. B. die Nummer der Kante dazu benutzen, in der Ordnung auf den Kantengewichten überall eine „echte Ungleichung“ zu haben, was das Einzige ist, was in Aufgabe 4.21 b) benutzt wurde.

Wie wir oben bereits bemerkt hatten, wird in jeder Iteration die Anzahl der Knoten mindestens halbiert, also haben wir höchstens $\log_2(|V|)$ Iterationen. Für die Kontraktion müssen wir bei $O(|E|)$ Kanten die Endknoten aufdatieren und erhalten (wenn wir davon ausgehen, dass $|V| = O(|E|)$ ist) als Gesamtlaufzeit

$$O(|E| \log |V|).$$

Beispiel 4.25. Wir betrachten die geometrische Instanz in Abbildung 4.4 mit 13 Knoten. Darauf betrachten wir den vollständigen Graphen, wobei die Kantengewichte durch die Entfernung in der Zeichnung gegeben seien. In der linken Grafik haben wir einige Kanten eingezeichnet. Diejenigen, die wir weggelassen haben sind so lang, dass sie für einen minimalen aufspannenden Baum auch nicht in Frage kommen.

In der mittleren Figur haben wir die Kanten in der Reihenfolge nummeriert, in der sie der Greedy-Algorithmus in den minimalen aufspannenden Baum aufnimmt. Dabei ist die Reihenfolge der zweiten, dritten und vierten sowie der achten und neunten vertauschbar, da diese alle jeweils die gleiche Länge haben. Wir gehen im Folgenden davon aus, dass die früher gewählten Kanten eine kleinere Nummer haben.

Die Nummerierung im Baum ganz rechts entspricht der Reihenfolge, in der Prims Algorithmus die Kanten in den Baum aufnimmt, wenn er im obersten Knoten startet. Hier ist die Reihenfolge eindeutig.

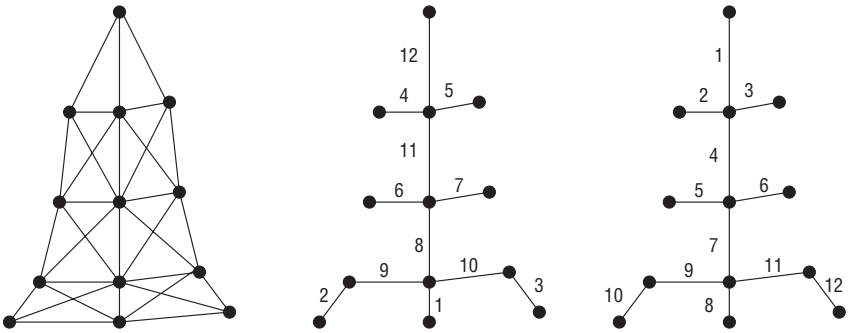


Abb. 4.4 Kruskal und Prim

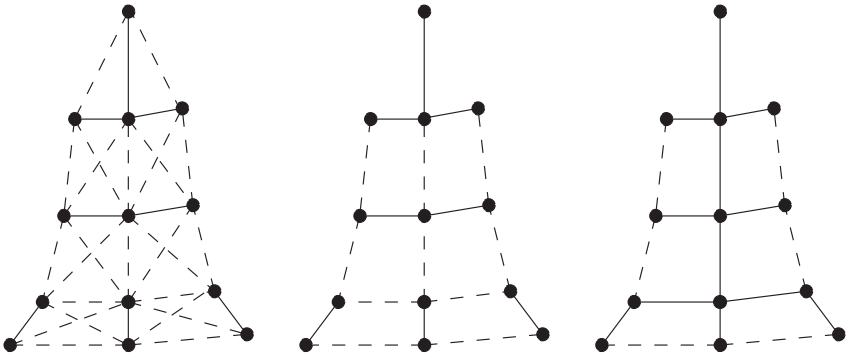


Abb. 4.5 Borůvka

In Abbildung 4.5 haben wir den Verlauf des Algorithmus von Borůvka angedeutet. Es kommt hier nicht zum Tragen, dass die Kantengewichtsfunktion nicht injektiv ist. In der ersten Iteration sind die markierten Kanten, die fett gezeichneten Kanten in den ersten beiden Bäumen. Im mittleren Baum haben wir Kanten eliminiert, die nach der Kontraktion entweder Schleifen oder parallel zu kürzeren Kanten sind. Im Baum rechts erkennt man, dass der Algorithmus bereits nach der zweiten Iteration den minimalen aufspannenden Baum gefunden hat.

Aufgabe 4.26. Sei $V = \{1, 2, \ldots, 30\}$ und $G = (V, E)$ definiert durch

$$e = (i, j) \in E \iff i \mid j \text{ oder } j \mid i$$

der Teilbarkeitsgraph. Die Gewichtsfunktion w sei gegeben durch den ganzzahligen Quotienten $\frac{j}{i}$ bzw. $\frac{i}{j}$.

Geben Sie die Kantenmengen und die Reihenfolge ihrer Berechnung an, die die Algorithmen von Kruskal, Prim und Borůvka berechnen. Bei gleichen Kantengewichten sei die mit den kleineren Knotennummern die Kleinere.

Lösung siehe Lösung 9.42.

4.6 Die Anzahl aufspannender Bäume

Wir hatten zu Anfang unserer Überlegungen zu minimalen aufspannenden Bäumen angekündigt nachzuweisen, dass der K_n n^{n-2} aufspannende Bäume hat. Da Kanten unterschiedliche Gewichte haben können, betrachten wir dabei isomorphe aber nicht identische Bäume als verschieden, wir nennen diese *knotengelabelte Bäume*.

Die Formel wurde 1889 von Cayley entdeckt und ist deswegen auch unter dem Namen Cayley-Formel bekannt. Der folgende Beweis ist allerdings 110 Jahre jünger, er wurde erst 1999 von Jim Pitman publiziert und benutzt die *Methode des doppelten Abzählens*. Anstatt knotengelabelte Bäume zu zählen, zählen wir knoten- und kantengelabelte Wurzelbäume. Darunter verstehen wir einen Wurzelbaum zusammen mit einer Nummerierung seiner Kanten. Da es $(n-1)!$ Möglichkeiten gibt, die Kanten zu nummerieren und weitere n Möglichkeiten gibt, die Wurzel auszuwählen, entspricht ein knotengelabelter Baum insgesamt $n!$ knoten- und kantengelabelten Wurzelbäumen. Dies halten wir fest:

Proposition 4.1. *Jeder knotengelabelte Baum mit n Knoten gibt Anlass zu genau $n!$ knoten- und kantengelabelten Wurzelbäumen.*

Wir zählen nun die knoten- und kantengelabelten Wurzelbäume, indem wir die Nummerierung der Kanten als dynamischen Prozess interpretieren. Im ersten Schritt haben wir n isolierte Knoten. Diese interpretieren wir als n (triviale) Wurzelbäume und fügen eine gerichtete Kante hinzu, so dass daraus $n-1$ Wurzelbäume entstehen. Im k-ten Schritt haben wir $n-k+1$ Wurzelbäume und fügen die k-te gerichtete Kante hinzu (siehe Abbildung 4.6). Diese Kante darf von einem beliebigen Knoten in einem der Wurzelbäume ausgehen, darf aber wegen Aufgabe 4.3 nur in der Wurzel eines der $n-k$ übrigen Wurzelbäume enden.

Lemma 4.7. *Es gibt genau $n!\,n^{n-2}$ knoten- und kantengelabelte Wurzelbäume mit n Knoten.*

Beweis. Die eben beschriebenen Wahlmöglichkeiten waren alle unabhängig voneinander. Also erhalten wir die Anzahl der knoten- und kantengelabelte Wurzelbäume als

$$\prod_{k=1}^{n-1} n(n-k) = n^{n-1}\,(n-1)! = n^{n-2}\,n!.$$

□

Fassen wir Proposition 4.1 und Lemma 4.7 zusammen, so erhalten wir:

Satz 4.27 (Cayley-Formel). *Die Anzahl der knotengelabelten Bäume mit n Knoten ist n^{n-2}.*

□

Aufgabe 4.28. Sei $G = K_n$ der vollständige Graph mit n Knoten und e eine feste Kante. Zeigen Sie: Die Anzahl der knotengelabelten Bäume von G, die die Kante e enthalten, ist $2n^{n-3}$.
Lösung siehe Lösung 9.43.

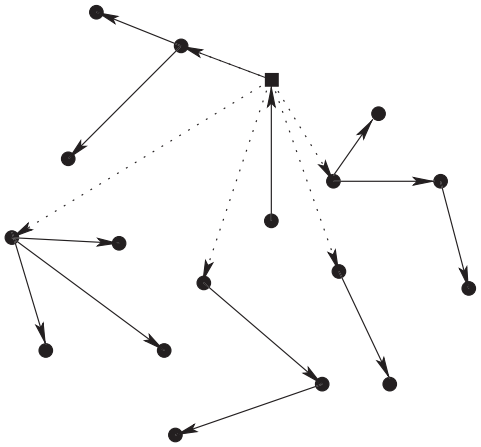


Abb. 4.6 Wenn man den quadratischen Knoten als Startknoten für die dreizehnte Kante auswählt, hat man $5 = 18 - 13 = 6 - 1$ Möglichkeiten, einen Endknoten auszusuchen, die wir durch die gepunkteten Kanten angedeutet haben.

4.7 Bipartites Matching

Wir betrachten nun Zuordnungsprobleme. Der Einfachheit halber betrachten wir nur Aufgabenstellungen, bei denen Elementen aus einer Menge U jeweils ein Element aus einer Menge V unter gewissen Einschränkungen zugeordnet werden soll.

Beispiel 4.29. a) In einer geschlossenen Gesellschaft gibt es m heiratsfähige Männer und n heiratsfähige Frauen. Die Frauen haben jeweils eine Liste der akzeptablen Partner. Verheirate möglichst viele Paare unter Beachtung der Akzeptanz und des Bigamieverbots.

b) An einer Universität bewerben sich Studenten für verschiedene Studiengänge, wobei die Individuen sich für mehrere Studiengänge bewerben. Die Anzahl der Studienplätze in jedem Fach ist begrenzt. Finde eine Zuordnung der Studenten zu den Studiengängen, so dass die Wünsche der Studenten berücksichtigt werden und möglichst viele Studienplätze gefüllt werden.

In beiden Situationen haben wir es mit einem bipartiten Graphen zu tun, der im ersten Fall die Neigungen der Damen und im zweiten die Wünsche der Studenten modelliert:

Definition 4.8. Sei $G = (W, E)$ ein Graph. Dann heißt G *bipartit*, wenn es eine Partition $W = U \dot\cup V$ gibt, so dass alle Kanten je einen Endknoten in beiden Klassen haben. Wir nennen dann U und V die *Farbklassen von* G.

Sie haben in Beispiel 3.12 bereits die vollständigen bipartiten Graphen $K_{m,n}$ kennen gelernt. Allgemeine bipartite Graphen lassen sich aber auch sehr leicht charakterisieren.

Proposition 4.2. *Ein Graph $G = (W, E)$ ist bipartit genau dann, wenn er keinen Kreis ungerader Länge hat.*

Beweis. Ist G bipartit, so müssen die Knoten jedes Kreises abwechselnd in U und in V liegen. Da der Kreis geschlossen ist, muss er also gerade Länge haben.

Sei nun G ein Graph, in dem alle Kreise gerade Länge haben. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass G zusammenhängend ist. Ansonsten machen wir das Folgende in jeder Komponente. Sei $v \in W$. Wir behaupten zunächst, dass für alle $w \in W$ jeder vw-Weg entweder stets gerade oder stets ungerade Länge hat. Denn angenommen P wäre ein vw-Weg der Länge $2k$ und Q ein vw-Weg der Länge $2k'+1$. Dann ist die *symmetrische Differenz*

$$P \Delta Q := (P \cup Q) \setminus (P \cap Q)$$

eine Menge mit ungerade vielen Elementen, denn

$$|P \Delta Q| = |P| + |Q| - 2|P \cap Q| = 2(k + k' - |P \cap Q|) + 1.$$

Wir untersuchen nun, welche Knotengrade in dem von $P \Delta Q$ gebildeten Teilgraphen von G auftreten können. Sowohl in P als auch in Q haben u und w jeweils den Knotengrad 1 und alle anderen Knoten entweder Knotengrad 0 oder Knotengrad 2. Also haben in $P \Delta W$ alle Knoten den Knotengrad $0, 2$ oder 4, insbesondere ist $H = (W, P \Delta Q)$ eulersch und somit nach Satz 3.40 kantendisjunkte Vereinigung von Kreisen. Da die Gesamtzahl der Kanten in $P \Delta Q$ aber ungerade ist, muss unter diesen Kreisen mindestens einer ungerader Länge sein im Widerspruch zur Voraussetzung. Also hat für alle $w \in W$ jeder vw-Weg entweder stets gerade oder stets ungerade Länge.

Seien nun $U \subseteq W$ die Knoten u, für die alle uv-Wege ungerade Länge haben und V die Knoten mit gerader Distanz von v. Angenommen, es gäbe eine Kante e zwischen zwei Knoten u_1, u_2 in U. Ist dann P ein vu_1 Weg, so ist $P \Delta (u_1, u_2)$ ein vu_2-Weg gerader Länge im Widerspruch zum Gezeigten. Analog gibt es auch keine Kanten zwischen Knoten in V. Also ist G bipartit. □

Die Zuordnungsvorschriften in Beispiel 4.29 kann man auch für beliebige Graphen definieren.

Definition 4.9. Sei $G = (V, E)$ ein Graph. Eine Kantenmenge $M \subseteq E$ heißt ein *Matching* in G, falls für den Graphen $G_M = (V, M)$ gilt

$$\forall v \in V : \deg_{G_M}(v) \leq 1. \tag{4.4}$$

Wir sagen u ist mit v *gematched*, wenn $(u, v) \in M$, und nennen einen Knoten *gematched*, wenn er mit einer Matchingkante inzident ist, und ansonsten *ungematched.*

Gilt in (4.4) stets Gleichheit, so nennen wir das Matching *perfekt.* Wir bezeichnen (4.4) auch als *Bigamieverbot*.

Bei der Bestimmung von Matchings mit maximal vielen Kanten ist nun die Greedy-Strategie, die bei aufspannenden Bäumen so erfolgreich war, kein probates Mittel. Betrachten wir etwa den Graphen in Abbildung 4.7 mit der fett gezeichneten Kante als Matching, so kann man zu dieser Kante keine weitere Kante hinzunehmen, ohne das Bigamieverbot zu verletzen.

Hingegen gibt es offensichtlich Matchings mit zwei Kanten. Um eine intelligentere Strategie zu entwickeln, betrachten wir den Unterschied zwischen einem Matching und einem Matching mit einer Kante mehr.

Seien also M, M' Matchings in $G = (V, E)$ und $|M'| > |M|$. Wiederum analysieren wir die Knotengrade in $(V, M \Delta M')$. Da die Knotengrade in M wie in M' nur 0 und 1 sind, kommen als Knotengrade in $(V, M \Delta M')$ nur $0, 1$ und 2 in Frage. Also besteht $(V, M \Delta M')$ aus isolierten Knoten, Pfaden und Kreisen. In den Kreisen müssen sich aber stets Kanten aus M mit Kanten aus M'

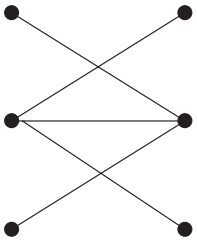


Abb. 4.7 Ein inklusionsmaximales Matching

abwechseln, also müssen diese immer gerade Länge haben. Da aber M' mehr Elemente als M hat, muss es unter den Wegen einen geben, der mehr Kanten in M' als in M hat. Auch umgekehrt kann man aus einem solchen Weg stets ein größeres Matching konstruieren. Dafür führen wir zunächst einmal den Begriff des augmentierenden Weges ein:

Definition 4.10. Sei $G = (V,E)$ ein (nicht notwendig bipartiter) Graph und $M \subseteq E$ ein Matching. Ein Weg $P = v_0v_1v_2\dots v_k$ heißt *M-alternierend*, wenn seine Kanten abwechselnd in M und außerhalb von M liegen. Der Weg P ist *M-augmentierend*, wenn darüber hinaus die beiden Randknoten v_0 und v_k ungematched sind. Insbesondere ist dann k ungerade und $v_iv_{i+1} \in M$ genau dann, wenn $1 \leq i \leq k-2$ und i ungerade.

Satz 4.30. *Sei $G = (V,E)$ ein (nicht notwendig bipartiter) Graph und $M \subseteq E$ ein Matching. Dann ist M genau dann von maximaler Kardinalität, wenn es keinen M-augmentierenden Weg in G gibt.*

Beweis. Wir zeigen die Kontraposition dieses Satzes, also, dass M genau dann nicht maximal ist, wenn es einen M-augmentierenden Weg gibt. Wir haben vor der letzten Definition bereits gezeigt, dass, wenn M' ein Matching von G mit $|M'| > |M|$ ist, $M'\Delta M$ einen M-augmentierenden Weg enthält. Sei also nun umgekehrt P ein M-augmentierender Weg in G. Wir untersuchen die Kantenmenge $M' := M\Delta P$. Da Anfangs- und Endknoten von P ungematched und verschieden sind, aber P ansonsten zwischen Matching- und Nichtmatchingkanten alterniert, hat $H = (V,M')$ überall Knotengrad 0 oder 1, also ist M' ein Matching und enthält eine Kante mehr als M. □

Wir haben nun das Problem, ein maximales Matching zu finden, auf das Bestimmen eines augmentierenden Weges reduziert. Wie findet man nun einen augmentierenden Weg? In allgemeinen Graphen wurde dieses Problem erst 1965 von Jack Edmonds gelöst. Wir wollen darauf hier nicht näher eingehen. In bipartiten Graphen ist die Lage viel einfacher, da ein M-augmentierender Weg stets die Endknoten in unterschiedlichen Farbklassen haben muss:

Proposition 4.3. *Sei $G = (U\dot\cup V,E)$ ein bipartiter Graph, M ein Matching in G und $P = v_0v_1v_2\dots v_k$ ein M-augmentierender Weg in G. Dann gilt*

$$v_0 \in U \Leftrightarrow v_k \in V.$$

Beweis. Wie oben bemerkt, ist k ungerade. Ist $v_0 \in U$, so ist $v_1 \in V$ und induktiv schließen wir, dass alle Knoten mit geradem Index in U und alle mit ungeradem Index in V liegen. Insbesondere gilt letzteres für v_k. Analog impliziert $v_0 \in V$ auch $v_k \in U$. □

Wenn wir also alle Kanten, die nicht in M liegen, von U nach V richten, und alle Kanten in M von V nach U, so wird aus P ein gerichteter Weg von einem ungematchten Knoten in U zu einem ungematchten Knoten in V. Da P beliebig war, ist dies unser Mittel der Wahl. Das Schöne an dem folgenden Verfahren ist, dass es, wenn es keinen augmentierenden Weg findet, einen „Beweis“ dafür liefert, dass es einen solchen auch nicht geben kann.

Algorithmus 4.31 (Find-Augmenting-Path). Input des Algorithmus ist ein bipartiter Graph $G = (U \dot\cup V, E)$ und ein Matching `M` $\subseteq E$. Output ist entweder ein Endknoten eines `M`-augmentierenden Weges oder ein „Zertifikat“ `C` für die Maximalität von M. In der Queue `Q` merken wir uns die noch zu bearbeitenden Knoten. Wir gehen davon aus, dass in einer Initialisierung alle Zeiger des Vorgängerfeldes `pred` mit None initialisiert worden sind und `C` die leere Liste ist. Bei Rückgabe eines Endknotens, kann man aus diesem durch Rückverfolgen der Vorgänger den augmentierenden Weg konstruieren.

```
for u in U:
    if not M.matches(u):
        Q.Append(u)
        pred[u]=u
while not Q.IsEmpty():
    u=Q.Top()
    for v in G.Neighborhood(u):
        if pred[v]==None:
            pred[v]=u
            if not M.IsMatched(v):
                return v
            else:
                s=M.Partner(v)
                Q.Append(s)
                pred[s]=v
for all u in U:
    if pred[u]==None:
        C.Append(u)
for all v in V:
    if pred[v]!=None:
        C.Append(v)
```

Zunächst initialisieren wir `Q` mit allen ungematchten Knoten `u` in U. Dann untersuchen wir deren Nachbarn `v` und setzen `u` als ihren Vorgänger ein. Finden wir darunter ein ungematchtes `v`, so wurde schon ein augmentierender Weg gefunden. Ansonsten sei `s` sein Matchingpartner. Der Knoten `s` kann bisher noch nicht bearbeitet worden sein, wir hängen ihn an die Warteschlange und setzen seinen Vorgänger auf `v`. So fahren wir fort, bis wir entweder einen ungematchten Knoten in V finden oder die Schlange `Q` leer ist.

Findet das Verfahren keinen augmentierenden Weg mehr, so sammeln wir in `C` den „Beweis“ der Maximalität des Matchings. Warum dies ein Beweis der Maximalität ist, werden wir in Lemma 4.8 und Satz 4.33 erfahren.

Beispiel 4.32. Wir starten unseren Algorithmus mit dem Matching in Abbildung 4.7.

Zunächst stellen wir u_1 und u_3 in die Warteschlange und setzen `pred(`u_1`)`$=u_1$ und `pred(`u_3`)`$=u_3$. Ausgehend vom Knoten u_1 finden wir v_2, setzen `pred(`v_2`)`$=u_1$ und stellen dessen Matchingpartner u_2 in die Schlange mit `pred(`u_2`)`$=v_2$. Von u_3 aus finden wir keinen neuen Knoten, aber von u_2 aus den ungematchten Knoten v_1, dessen Vorgänger wir auf `pred(`v_1`)`$=u_2$ setzen und den wir zurückliefern. Durch Rückverfolgen der Vorgängerfunktion finden wir $u_1v_2u_2v_1$ als M-augmentierenden Weg und ersetzen die bisherige Matchingkante durch (u_1v_2) und (u_2v_1).

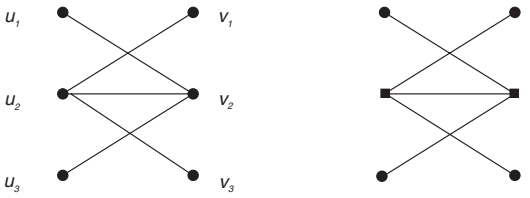


Abb. 4.8 Zwei Durchläufe der Suche nach einem erweiternden Weg

Wir löschen nun wieder alle Vorgänger, d.h. wir setzen `pred` auf `NULL` und stellen u_3 in die Schlange, finden von dort aus v_2, dessen Matchingpartner u_1 in die Schlange aufgenommen wird. Von u_1 aus finden wir nichts Neues und die Warteschlange ist erfolglos abgearbeitet worden. Der einzige Knoten ohne gesetzten Vorgänger in U ist u_2 und der einzige mit gesetztem Vorgänger in V ist v_2. Also ist `C`$=\{u_2, v_2\}$.

Proposition 4.4. *Findet die Prozedur „Find Augmenting Path“ einen ungematchten Knoten* `v`*, so erhält man durch Rückverfolgen des* `pred`*-Arrays einen M-augmentierenden Weg.*

Beweis. Der Knoten `w` wurde von einem Knoten `u` in U aus gelabelt. Dieser ist entweder selber ungematched, also `uw` ein M-augmentierender Weg, oder wir suchen vom Matchingpartner von `u` an Stelle von `w` weiter. Da die Knotenmenge endlich ist und das Verfahren wegen der Vorgängerabfrage in der 8. Zeile nicht zykeln kann, muss es in einem augmentierenden Weg enden. □

Im Folgenden wollen wir klären, inwiefern die Liste `C` ein Beweis dafür ist, dass es keinen augmentierenden Weg mehr gibt. Dafür zunächst noch eine Definition:

Definition 4.11. Sei $G=(V,E)$ ein (nicht notwendig bipartiter) Graph und $C \subseteq V$. Dann heißt C *kantenüberdeckende Knotenmenge* oder kürzer *Knotenüberdeckung (engl. vertex cover)*, wenn für alle $e \in E$ gilt: $C \cap e \neq \emptyset$.

Lemma 4.8. *Liefert das Verfahren eine Knotenliste* `C` *zurück, so ist* $|\texttt{C}| = |M|$ *und* `C` *ist eine kantenüberdeckende Knotenmenge.*

Beweis. `C` besteht aus allen unerreichten Knoten in U und allen erreichten Knoten in V. Also sind alle Knoten in $\texttt{C} \cap U$ gematched, da ungematchte Knoten zu Beginn in `Q` aufgenommen werden, und alle Knoten in $\texttt{C} \cap V$ sind gematched, da kein ungematchter Knoten in V gefunden wurde. Andererseits sind die Matchingpartner von Knoten in $V \cap \texttt{C}$ nicht in `C`, da diese ja in `Q` aufgenommen wurden. Somit gilt

$$\forall m \in M : |\texttt{C} \cap m| \leq 1.$$

Da M ein Matching ist und somit kein Knoten zu 2 Kanten in M inzident sein kann, schließen wir hieraus

$$|\mathtt{C}| \leq |M|.$$

Wir zeigen nun, dass `C` eine kantenüberdeckende Knotenmenge ist. Angenommen, dies wäre nicht so und $e = (\mathtt{u},\mathtt{v})$ eine Kante mit $\{\mathtt{u},\mathtt{v}\} \cap \mathtt{C} = \emptyset$. Dann ist `pred[v]=None`, aber `pred[u]`$\neq$`None`. Als aber `pred[u]` gesetzt wurde, wurde `u` gleichzeitig in `Q` aufgenommen, also irgendwann auch mal abgearbeitet. Dabei wurde bei allen Nachbarn, die noch keinen Vorgänger hatten, ein solcher gesetzt, insbesondere auch bei `v`, im Widerspruch zu `pred[v]=None`. Also ist `C` eine Knotenüberdeckung.

Schließlich folgt $|\mathtt{C}| \geq |M|$ aus der Tatsache, dass kein Knoten zwei Matchingkanten überdecken kann. □

Den folgenden Satz haben wir damit im Wesentlichen schon bewiesen:

Satz 4.33 (Satz von König 1931). *In bipartiten Graphen ist*

$$\max\{|M| \mid M \text{ ist Matching }\} = \min\{|C| \mid C \text{ ist Knotenüberdeckung}\}.$$

Beweis. Da jeder Knoten einer Knotenüberdeckung C höchstens eine Matchingkante eines Matchings M überdecken kann, gilt stets $|M| \leq |C|$, also auch im Maximum. Ist nun M ein maximales Matching, so liefert die Anwendung von Algorithmus 4.31 eine Knotenüberdeckung C mit $|C| = |M|$. Also ist eine minimale Knotenüberdeckung höchstens so groß wie ein maximales Matching. □

Bemerkung 4.34. In allgemeinen Graphen ist das Problem der minimalen Knotenüberdeckung **NP**-vollständig.

Wir stellen nun noch zwei Varianten des Satzes von König vor. Dafür führen wir den Begriff der Nachbarschaft von Knoten ein.

Definition 4.12. Ist $G = (V,E)$ ein (nicht notwendig bipartiter) Graph und $H \subseteq V$, so bezeichnen wir mit $N_G(H)$ bzw. $N(H)$ die *Nachbarschaft von H*

$$N(H) := \{v \in V \mid \exists u \in H : (u,v) \in E\}.$$

Korollar 4.35 (Heiratssatz von Frobenius 1917). *Sei $G = (U \dot\cup V, E)$ ein bipartiter Graph. Dann hat G genau dann ein perfektes Matching, wenn $|U| = |V|$ und*

$$\forall H \subseteq U : |N(H)| \geq |H|. \tag{4.5}$$

Beweis. Hat G ein perfektes Matching und ist $H \subseteq U$, so liegt der Matchingpartner jedes Knotens in H in der Nachbarschaft von H, die also gewiss mindestens so groß wie H sein muss. Die Bedingung $|U| = |V|$ ist bei Existenz eines perfekten Matchings trivialerweise erfüllt.

Die andere Implikation zeigen wir mittels Kontraposition. Wir nehmen an, dass G keine isolierten Knoten hat, denn sonst ist (4.5) offensichtlich verletzt. Hat G kein perfektes Matching und ist $|U| = |V|$ so hat G nach dem Satz von König eine Knotenüberdeckung C mit $|C| < |U|$. Wir setzen $H = U \setminus C$. Da C eine Knotenüberdeckung ist, ist

$$N(H) \subseteq C \cap V.$$

Also ist

$$|N(H)| \leq |C \cap V| = |C| - |C \cap U| < |U| - |C \cap U| = |U \setminus C| = H.$$

Also verletzt H (4.5). □

Der Name Heiratssatz kommt von der Interpretation wie in Beispiel 4.29. Wenn alle Frauen nur Supermann heiraten wollen, bleiben einige ledig.

Die letzte Variante des Satzes von König ist eine asymmetrische Version des Satzes von Frobenius:

Satz 4.36 (Heiratssatz von Hall). *Sei $G = (U \dot\cup V, E)$ ein bipartiter Graph. Dann hat G ein Matching, in dem alle Knoten in U gematched sind, genau dann, wenn*

$$\forall H \subseteq U : |N(H)| \geq |H|. \tag{4.6}$$

Beweis. Wie im Satz von Frobenius ist die Notwendigkeit der Bedingung offensichtlich. Wir können ferner davon ausgehen, dass $|U| \leq |V|$ ist, da ansonsten sowohl die Nichtexistenz eines gewünschten Matchings als auch die Verletzung von (4.6) offensichtlich ist. Wir fügen nun $|V| - |U|$ Dummyknoten zu U hinzu, die alle Knoten in V kennen, und erhalten den bipartiten Graphen $\tilde{G} = (\tilde{U} \dot\cup V, \tilde{E})$, der offensichtlich genau dann ein perfektes Matching hat, wenn G ein Matching hat, das alle Knoten in U matched. Hat $\tilde{G}$ kein perfektes Matching, so gibt es nach dem Satz von Frobenius eine Menge $H \subseteq \tilde{U}$ mit $|N_{\tilde{G}}(H)| < |H|$. Da die Dummyknoten alle Knoten in V kennen, muss $H \subseteq U$ sein und also

$$|N_G(H)| = |N_{\tilde{G}}(H)| < |H|.$$

□

Wir wollen diesen Abschnitt beschließen mit dem nun hoffentlich offensichtlichen Algorithmus zur Bestimmung eines maximalen Matchings in einem bipartiten Graphen und der Analyse seiner Laufzeit.

Algorithmus 4.37 (Bipartites Matching). Starte mit einem leeren Matching und setze `C=[]`.

```
while C==[]:
    (C,w)=FindAugmentingPath(M)
    if C==[]:
        P=BackTrackPath(w)
        Augment(M,P)
```

Wir können offensichtlich höchstens $\min\{|U|, |V|\}$ Matchingkanten finden, also wird die while-Schleife $O(\min\{|U|, |V|\})$-mal ausgeführt. Die Prozedur Find-Augmenting-Path besteht im Wesentlichen aus einer Breitensuche in dem Digraphen, der aus G entsteht, wenn Matchingkanten „Rückwärtskanten“ und die übrigen Kanten „Vorwärtskanten“, also von U nach V orientiert sind. Der Aufwand beträgt also $O(|E|)$. Für das Backtracking und die Augmentierung zahlen wir nochmal je $O(\min\{|U|, |V|\})$, wenn wir davon ausgehen, dass $\min\{|U|, |V|\} = O(|E|)$ ist, geht dieser Term in $O(|E|)$ auf und wir erhalten als Gesamtlaufzeit

$$O(\min\{|U|, |V|\}|E|).$$

Bemerkung 4.38. Mit etwas, aber nicht viel mehr, Aufwand berechnet ein Algorithmus von Hopcroft und Tarjan ein maximales bipartites Matching in $O(\sqrt{|V|}|E|)$.

Aufgabe 4.39. Bestimmen Sie in dem Graphen in Abbildung 4.9 ein maximales Matching und eine minimale Knotenüberdeckung. Lösung siehe Lösung 9.44.

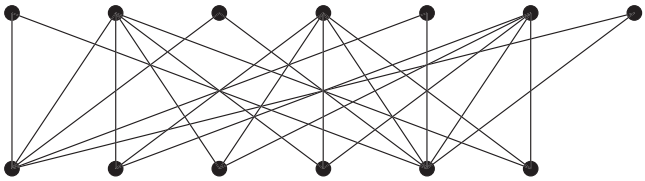


Abb. 4.9 Ein bipartiter Graph

Aufgabe 4.40. Betrachten Sie ein Schachbrett, auf dem einige Felder markiert sind. Auf den markierten Feldern sollen Sie nun möglichst viele Türme platzieren, so dass sich keine zwei davon schlagen können. Zeigen Sie:

Die Maximalzahl der Türme, die man auf den markierten Feldern platzieren kann, ohne dass zwei sich schlagen können ist gleich der minimalen Summe der Anzahl der Zeilen und Spalten, die man auswählen kann, so dass jedes markierte Feld in einer ausgewählten Spalte oder in einer ausgewählten Zeile liegt.
Lösung siehe Lösung 9.45.

Aufgabe 4.41. Eine *Permutationsmatrix* ist eine Matrix $P \in \{0,1\}^{n\times n}$, bei der in jeder Zeile und Spalte jeweils genau eine 1 und sonst nur Nullen stehen.

Eine Matrix $A \in \mathbb{R}^{n\times n}$, bei der für alle Einträge a_{ij} gilt $0 \leq a_{ij} \leq 1$, heißt *doppelt stochastisch*, wenn die Summe aller Einträge in jeder Zeile und Spalte gleich 1 ist.

Zeigen Sie (etwa per Induktion über die Anzahl $k \geq n$ der von Null verschiedenen Einträge in A): Jede doppelt stochastische Matrix ist eine *Konvexkombination* von Permutationsmatrizen, d. h. es gibt $l \in \mathbb{N}$ und Permutationsmatrizen $P_1, \ldots, P_l$ sowie Koeffizienten $\lambda_1, \ldots, \lambda_l$ mit $0 \leq \lambda_i \leq 1$ und $\sum_{i=1}^{l} \lambda_i = 1$ so, dass

$$A = \sum_{i=1}^{l} \lambda_i P_i.$$

Lösung siehe Lösung 9.46.

4.8 Stabile Hochzeiten

Beschließen wollen wir dieses Kapitel mit einer Variante des Matchingproblems, die der Spieltheorie zugeordnet und durch einen einfachen Algorithmus gelöst wird. Es fängt ganz ähnlich wie beim Matching an.

Beispiel 4.42. In einer geschlossenen Gesellschaft gibt es je n heiratsfähige Männer und Frauen. Sowohl Frauen als auch Männer haben Präferenzen, was die Personen des anderen Geschlechts angeht. Aufgabe ist es nun, Männer und Frauen so zu verheiraten, dass es kein Paar aus Mann und Frau gibt, die nicht miteinander verheiratet sind, sich aber gegenseitig Ihren Ehepartnern vorziehen.

Wir werden uns im Folgenden in der Darstellung an diesem Beispiel orientieren. Das liegt einerseits daran, dass es so in den klassischen Arbeiten präsentiert wird und außerdem die Argumentation dadurch anschaulicher wird. Ähnlichkeiten mit Vorkommnissen bei lebenden Personen oder Personen der Zeitgeschichte werden von uns weder behauptet noch gesehen.

Unsere Modellierung sieht wie folgt aus:

Definition 4.13. Seien U, V Mengen mit $|U| = |V|$ und für alle $u \in U$ sei $\prec_u$ eine Totalordnung von V, sowie für alle $v \in V$ sei $\prec_v$ eine Totalordnung von U. Eine bijektive Abbildung von $\tau : U \to V$ heißt *stabile Hochzeit*, wenn für alle $u \in U$ und $v \in V$ gilt,

$$\text{entweder } \tau(u) = v \text{ oder } v \prec_u \tau(u) \text{ oder } u \prec_v \tau^{-1}(v).$$

In Worten: entweder u und v sind miteinander verheiratet oder mindestens einer zieht seinen Ehepartner dem anderen (d.h. u oder v) vor.

Wir nennen U die Menge der Männer und V die Menge der Frauen.

Es ist nun nicht ohne Weiteres klar, dass es für alle Präferenzlisten stets eine stabile Hochzeit gibt. Dass dies so ist, wurde 1962 von den Erfindern dieses „Spiels“ Gale und Shapley algorithmisch gezeigt. Der Algorithmus, mit dem sie eine stabile Hochzeit berechnen, trägt seine Beschreibung schon im Namen: „Men propose – Women dispose“.

Algorithmus 4.43 (Men propose – Women dispose). Eingabedaten wie eben. Zu Anfang ist niemand verlobt. Die verlobten Paare bilden stets ein Matching im vollständigen bipartiten Graphen auf U und V. Der Algorithmus terminiert, wenn das Matching perfekt ist.

- Solange es einen Mann gibt, der noch nicht verlobt ist, macht dieser der besten Frau auf seiner Liste einen Antrag.
- Wenn die Frau nicht verlobt ist oder ihr der Antragsteller besser gefällt als ihr Verlobter, nimmt sie den Antrag an und löst, falls existent, ihre alte Verlobung. Ihr Ex-Verlobter streicht sie von seiner Liste.
- Andernfalls lehnt Sie den Antrag ab, und der Antragsteller streicht sie von seiner Liste.

Zunächst stellen wir fest, dass wir alle soeben aufgeführten Schritte in konstanter Zeit durchführen können, wenn wir davon ausgehen, dass die Präferenzen als Liste gegeben sind und wir zusätzlich zwei Elemente in konstanter Zeit vergleichen können. Die unverlobten Männer können wir in einer Queue verwalten, deren erstes Element wir in konstanter Zeit finden. Ebenso können wir später einen Ex-Verlobten in konstanter Zeit ans Ende der Queue stellen. Der Antragsteller findet seine Favoritin in konstanter Zeit am Anfang seiner Liste und diese braucht nach Annahme auch nicht länger, um ihn mit Ihrem Verlobten zu vergleichen.

Für die Laufzeit des Algorithmus ist also folgende Feststellung ausschlaggebend:

Proposition 4.5. *Kein Mann macht der gleichen Frau zweimal einen Antrag.*

Beweis. Wenn ein Mann beim ersten Antrag abgelehnt wird, streicht er die Frau von seiner Liste. Wird er angenommen, so streicht er sie von seiner Liste, wenn sie ihm den Laufpass gibt. □

Nun überlegen wir uns, dass der Algorithmus zulässig ist, dass also stets ein Mann, der nicht verlobt ist, noch mindestens eine Kandidatin auf seiner Liste hat. Dazu beobachten wir:

Proposition 4.6. *Eine Frau, die einmal verlobt ist, bleibt es und wird zu keinem späteren Zeitpunkt mit einem Antragsteller verlobt sein, der ihr schlechter als ihr derzeitiger Verlobter gefällt.*

Beweis. Eine Frau löst eine Verlobung nur, wenn sie einen besseren Antrag bekommt. Der Rest folgt induktiv, da die Ordnung der Präferenzen transitiv ist. □

Da die Anzahl der verlobten Frauen und Männer stets gleich ist, gibt es mit einem unverlobten Mann auch stets noch mindestens eine unverlobte Frau, die also auch noch auf der Liste unseres Antragstellers stehen muss. Setzen wir nun $|U| = |V| = n$, so haben wir damit fast schon gezeigt:

Satz 4.44. *a) Der Algorithmus „Men propose – Women dispose“ terminiert in $O(n^2)$.*
b) Wenn er terminiert, sind alle verlobt.
c) Die durch die Verlobungen definierte bijektive Abbildung ist eine stabile Hochzeit.

Beweis.

a) Eine Antragstellung können wir in konstanter Zeit abarbeiten. Jeder Mann macht jeder Frau höchstens einen Antrag, also gibt es höchstens n^2 Anträge und somit ist die Laufzeit $O(n^2)$.
b) Da der Algorithmus erst terminiert, wenn jeder Mann verlobt ist und $|U| = |V|$ ist, sind am Ende alle verlobt.
c) Bezeichnen wir die Verlobungsabbildung wieder mit τ. Seien $u \in U$ und $v \in V$ nicht verlobt, aber $\tau(u) \prec_u v$. Als u $\tau(u)$ einen Antrag machte, stand v nicht mehr auf seiner Liste, muss also vorher gestrichen worden sein. Als u v von seiner Liste strich, war sie entweder mit einem Mann verlobt, den sie u vorzog oder hatte soeben von einem entsprechenden Kandidaten einen Antrag bekommen. Nach Proposition 4.6 gilt also auch zum Zeitpunkt der Terminierung $u \prec_v \tau^{-1}(v)$. Also bildet τ eine stabile Hochzeit.

□

Aufgabe 4.45. Zeigen Sie: Der Algorithmus „Men propose – Women dispose“ liefert eine *männeroptimale* stabile Hochzeit, d. h. ist $u \in U$ ein beliebiger Mann und τ das Ergebnis von „Men propose – Women dispose“, so gibt es keine stabile Hochzeit σ, in der u mit einer Frau verheiratet wird, die er seiner gegenwärtigen vorzieht, d. h.

$$\forall \sigma \text{ stabile Hochzeit } \forall u \in U : \sigma(u) \preceq_u \tau(u).$$

Lösung siehe Lösung 9.47.

Selbstverständlich kann man aus Symmetriegründen im gesamten Abschnitt die Rollen von Männern und Frauen vertauschen. Dann liefert „Women propose – Men dispose“ eine frauenoptimale, stabile Hochzeit. Mischformen dieser beiden Ansätze, die eine stabile Hochzeit liefern, sind uns aber nicht bekannt.

http://www.springer.com/978-3-642-05421-1

Algorithmische Mathematik
Hochstättler, W.
2010, XIII, 298 S., Softcover
ISBN: 978-3-642-05421-1