Graphentheorie

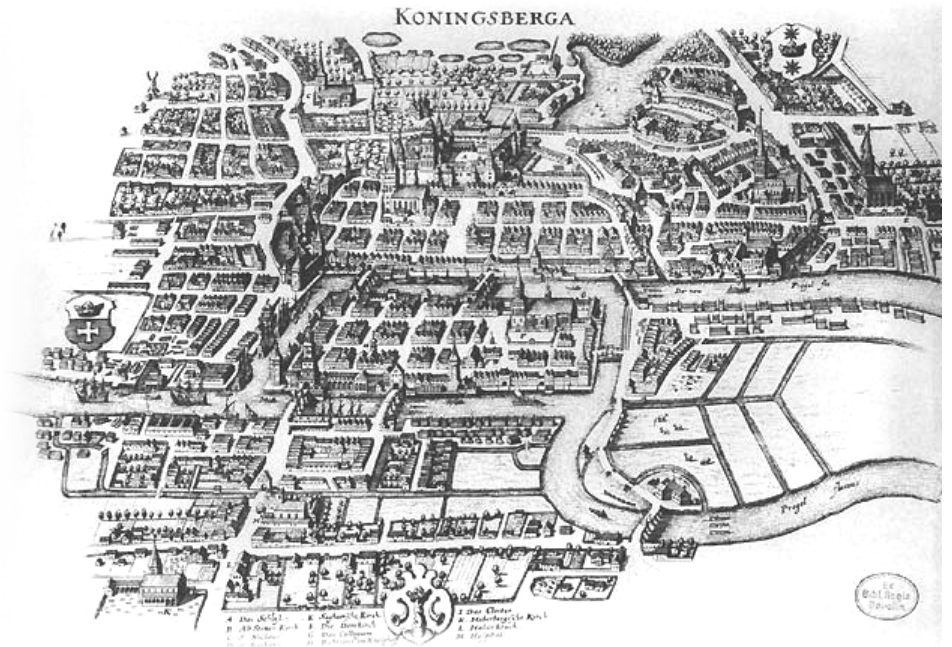
Yichuan Shen

10. Oktober 2013

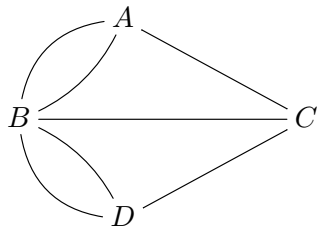
1 Was ist ein Graph?

Ein Graph ist eine kombinatorische Struktur, die bei der Modellierung zahlreicher Probleme Verwendung findet. Er besteht ganz allgemein aus einer Menge von Objekten, auch Knoten genannt, und Relationen zwischen diesen Objekten, auch Kanten genannt.

Beispiel. *(Das Königsberger Brückenproblem)* Als Geburtsstunde der Graphentheorie kann eine Arbeit von Leonard Euler aus dem Jahr 1736 angesehen werden: Die Stadt Königsberg in Preußen (heute Kaliningrad in Russland) liegt am Zusammenfluss zweier Arme der Pregel. Die Stadt besitzt sieben Brücken, die die einzelnen Stadtteile verbinden, die an den verschiedenen Ufern dieser Flussarme und auf einer Insel im Fluss liegen. Kann nun ein Spaziergänger an einem beliebigen Punkt der Stadt aus starten, über alle sieben Brücken genau einmal zu spazieren, und schließlich wieder den Ausgangspunkt erreichen?



Um dieses Problem zu lösen, bezeichnet Euler die einzelnen Stadtteile mit A, B, C und D und verbindet die Punkte mit einer Kante, wenn eine Brücke zwischen denen existiert. Der Stadtplan von Königsberg hat unter diesem Blickwinkel nun die folgende Gestalt:



Gibt es einen Weg, die von einem beliebigen Knoten ausgeht, genau einmal über jede Kante verläuft und zum Ausgangspunkt zurückführt? Auf dieser Basis konnte Euler nun sehr einfach zeigen, dass es in Königsberg keinen Rundgang der gewünschten Art gibt: Auf einem solchen Rundgang müsste nämlich die Anzahl der Kanten in jedem Zwischenknoten gerade sein.

Definition. Ein *(ungerichteter) Graph* $G = (V, E)$ ist ein Tupel, bestehend aus einer Menge V und einer Teilmenge:

$$E \subset \{\{u, v\} \subset V \mid u \neq v\}$$

Die Elemente der Menge V werden *Knoten* genannt, die Elemente $e = \{u, v\}$ der Menge E sind die *Kanten* des Graphen. Die Kante e verbindet die Knoten v und u. In diesem Fall sagen wir, dass v und u *adjazent* sind.

Ein Tupel $G = (V, E)$ heißt *gerichteter Graph* oder auch *Digraph*, wenn die Kantenmenge $E \subset V \times V$ aus Tupeln besteht.

G heißt *endlich*, wenn die Knotenmenge V endlich ist. Wir werden nur von nun an nur endliche Graphen betrachten.

Bemerkung. Tatsächlich kann man mit unserer Definition das Königsberger Brückenproblem nicht modellieren, da wir keine Möglichkeiten haben, doppelte Kanten darzustellen. Ein Graph, das Mehrfachkanten erlaubt, heißt *Multigraph*.

Beispiel. Ein Graph kann zur Modellierung verschiedenster Gebiete verwendet werden.

- $V \subset \{\text{Webseiten}\}$, $(u, v) \in E \iff$ Es existiert ein Link auf Seite u nach Seite v. Dann ist $G = (V, E)$ ein Digraph.
- $V \subset \{\text{Ortschaften}\}$, $\{u, v\} \in E \iff$ Es existiert eine Straße, die u und v auf direktem Wege verbindet. Dann ist $G = (V, E)$ ein Graph.
- $V \subset \{\text{Medikamente}\}$, $\{u, v\} \in E \iff$ Die gleichzeitige Einnahme von u und v ist tödlich. Dann ist $G = (V, E)$ ein Graph.

- $V \subset \{\text{Länder}\}$, $\{u, v\} \in E \iff$ Länder u und v haben auf der Landkarte eine gemeinsame Grenze. Dann ist $G = (V, E)$ ein Graph.
- $V \subset \{\text{Personen}\}$, $\{u, v\} \in E \iff u$ und v sind befreundet. $G = (V, E)$ ist meistens ein ungerichteter Graph.

Definition. Für einen Graphen $G = (V, E)$ definieren wir die *Nachbarschaft* $\Gamma(v)$ eines Knotens $v \in V$ durch:

$$\Gamma(v) := \{u \in V \mid \{u, v\} \in E\}$$

Der *Grad* $\deg(v) := \#\Gamma(v)$ von v bezeichnet die Größe der Nachbarschaft von v.

Satz 1. Für jeden Graphen $G = (V, E)$ gilt:

$$\sum_{v \in V} \deg(v) = 2 \cdot \#E$$

Beweis. Wir führen den Beweis durch vollständige Induktion über die Kantenzahl $m = \#E$. Ist $m = 0$, so auch $\Gamma(v) \subset E = \varnothing$ und somit $\deg(v) = 0$ für alle $v \in V$.

Sei die Behauptung für alle Graphen mit höchstens m Kanten bewiesen und $G = (V, E)$ ein Graph mit $m + 1$ Kanten. Sei $e = \{u', v'\} \in E$ eine Kante und betrachte den Graphen $G' = (V, E \setminus \{e\})$. Nach der Induktionsvoraussetzung gilt:

$$\sum_{v \in V} \deg_{G'}(v) = 2m$$

Nun ist $\deg_G(v) = \deg_{G'}(v)$ für alle $v \in V \setminus \{u', v'\}$ und $\deg_G(v) = \deg_{G'}(v) + 1$ für alle $v \in \{u', v'\}$. Also folgt:

$$\sum_{v \in V} \deg_G(v) = 2 + \sum_{v \in V} \deg_{G'}(v) = 2(m + 1) = 2 \cdot \#E$$

□

Korollar 2. In jedem Graphen ist die Zahl der Knoten mit ungeradem Grad gerade.

Beweis. Aufgrund des letzten Satzes ist die Gesamtsumme der Knotengrade in jedem Graphen gerade. Da sich gerade Knotengrade stets zu einer geraden Summe addieren, ungerade aber nur, wenn die Zahl der Summanden gerade ist, muss die Zahl der ungeraden Knotengrade gerade sein. □

Korollar 3. Für einen *k-regulären* Graphen $G = (V, E)$, d.h. $\deg(v) = k$ für alle $v \in V$ gilt:

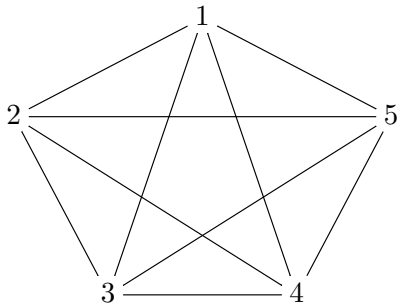
$$k \cdot \#V = 2 \cdot \#E$$

Definition. Ein Graph $H = (V', E')$ heißt *Teilgraph* eines Graphen $G = (V, E)$, wenn $V' \subset V$ und $E' \subset E$ gilt. Gilt sogar

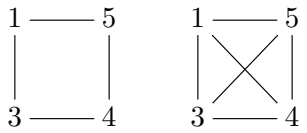
$$u, v \in V' \iff \{u, v\} \in E',$$

so nennt man H einen *induzierten Teilgraphen* von G und schreibt $H = G[V']$.

Beispiel. Betrachte den 4-regulären Graphen K_5:



Dann sind der linke Graph ein Teilgraph und der rechte der induzierte Teilgraph $K_5[\{1, 3, 4, 5\}]$:



Definition. Sei $G = (V, E)$ ein Graph und $u, v \in V$.

- Ein *Weg* W von u nach v ist eine endliche Folge $W = (w_0, \ldots, w_\ell)$ mit $u = w_0$ und $v = w_\ell$, so dass $\{w_i, w_{i-1}\} \in E$ für alle $i = 1, \ldots, \ell$ gilt. Die *Länge* dieses Weges ist ℓ.
- W heißt *geschlossen*, wenn $w_0 = w_\ell$ gilt.
- W heißt *Pfad*, wenn alle Knoten $w_0, \ldots, w_\ell$ paarweise verschieden sind.
- W heißt *Kreis*, wenn er geschlossen ist und $(w_1, \ldots, w_\ell)$ ein Pfad ist.

Definition. Zwei Knoten $u, v \in V$ eines Graphen $G = (V, E)$ heißen *zusammenhängend*, wenn es in G einen Weg von u nach v gibt. Diese Zusammenhangsrelation ist eine Äquivalenzrelation.

G heißt *zusammenhängend*, wenn jedes Paar von Knoten $u, v \in V$ zusammenhängend ist.

Die Äquivalenzklassen der Zusammenhangsrelation heißen *Zusammenhangskomponenten* von G.

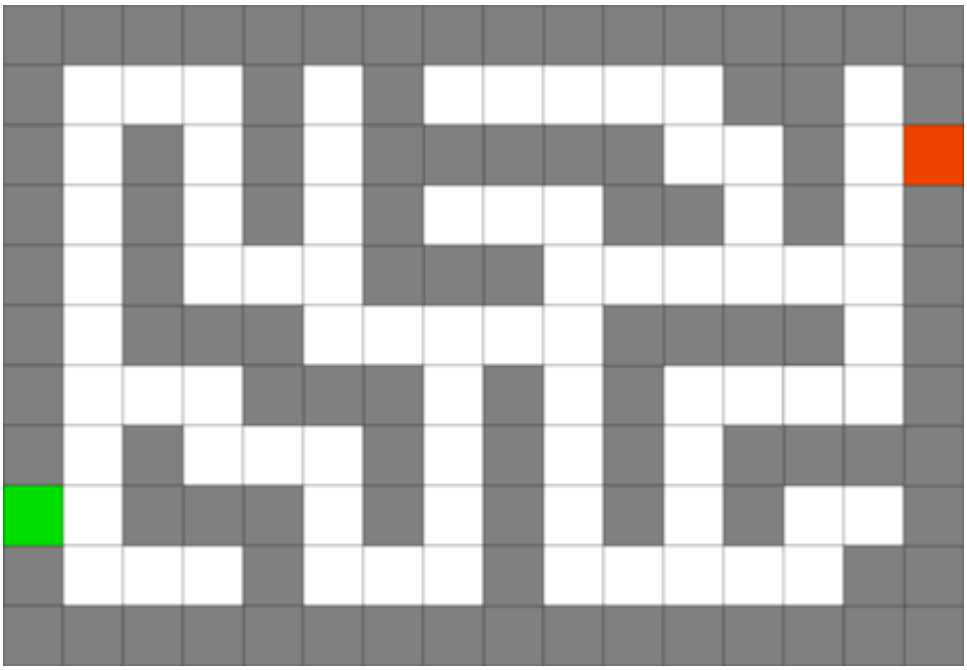
Bemerkung. Eine Zusammenhangskomponente eines Graphen ist also ein maximal zusammenhängender induzierter Teilgraph.

Lemma 4. Sei $G = (V, E)$ ein zusammenhängender Graph und C ein Kreis in G. Dann ist für jede im Kreis C enthaltende Kante e der Graph $G_e := (V, E \setminus \{e\})$ zusammenhängend.

Beweis. Angenommen, G_e wäre nicht zusammenhängend. Dann gibt es mindestens zwei Zusammenhangskomponenten. Da G zusammenhängend ist, müssen die Knoten der Kante $e = \{u, v\}$ in verschiedenen Zusammenhangskomponenten liegen. Weil e in einem Kreis C liegt, gibt es einen Pfad von u nach v, der die Kante e nicht enthält. Also liegen u, v in der selben Zusammenhangskomponente, ein Widerspruch. □

2 Datenstruktur

Wir wollen nun anhand des folgenden Beispiels Graphenprobleme mithilfe eines Rechners lösen. Gegeben ist ein Labyrinth, ein Eingang und ein Ausgang. Das Ziel ist einen Weg durch das Labyrinth zu finden.



Als Erstes unterteilen wir das Labyrinth in gleich große Quadrate wie im Bild angedeutet. Die nicht passierbaren Wände werden mit grau hinterlegt und die Gänge des Labyrinths mit weiß. Wir erhalten unser Labyrinth-Graph $G = (V, E)$, wobei:

$$V = \{\text{Nicht-graue Quadrate}\},\ \{u, v\} \in E \iff u \text{ und } v \text{ liegen nebeneinander}$$

Bevor wir anfangen können, müssen wir einen Graphen im Rechner darstellen können. Für eine Repräsentation von G im Rechner ist es hilfreich, wenn die Knotenmenge aus natürlichen Zahlen besteht. Wir identifizieren jedes Quadrat aus V mit einer natürlichen Zahl, also $V = \{0, 1, \ldots, n\}$.

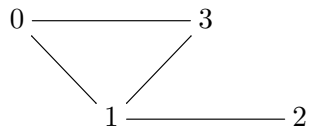
Definition. Die *Adjazenzmatrix* eines Graphen $G = (V, E)$ ist definiert als $A = (a_{uv})_{0 \le u,v \le n}$ mit:

$$a_{uv} := \begin{cases} 1, & \{u, v\} \in E \\ 0, & \text{sonst} \end{cases}$$

A ist symmetrisch, d.h. $a_{uv} = a_{vu}$.

Man kann Graphen auch in einer *Adjazenzliste* Γ speichern, d.h. wir speichern für jeden Knoten seine Nachbarschaft als eine Liste.

Beispiel. Die folgende Abbildung illustriert die Speicherung eines Graphen mit einer Adjazenzmatrix bzw. Adjazenzliste:



$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{array}, \qquad \Gamma = \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} [1,3] & [0,3,2] & [1] & [1,0] \end{bmatrix} \end{array}$$

Beide Darstellungen haben Vor- und Nachteile. So ist der Platz, den eine Adjazenzmatrix im Speicher benötigt, proportional zu der Anzahl der Einträge in der Matrix, also proportional zu $\#V^2$. Diese Darstellung eignet sich besonders für dichte Graphen. Im Gegensatz dazu ist die Darstellung durch eine Adjazenzliste effizienter, wenn der Graph nicht dicht ist, so dass viele Einträge der Adjazenzmatrix Null wären.

Definition. Um den Graphen effizient durchsuchen zu können, ist es praktisch zu wissen, ob man einen Knoten schon besucht hat oder nicht. Ferner wollen wir beim Durchlaufen des Graphen zu jedem Knoten seinen Vorgänger speichern, damit wir am Ende den Weg zurückverfolgen und somit rekonstruieren können. Wir speichern diese Daten in einem *Zustandsarray* s.

Für alle $v \in V$ setzen wir $s[v] := -1$, wenn v noch nicht besucht ist. Andernfalls setzen wir $s[v] := u \ge 0$, wenn $u \in V$ der Vorgänger von v ist und $s[v] := v \ge 0$, wenn v kein Vorgänger hat, d.h. der Startknoten ist.

3 Durchlaufen von Graphen

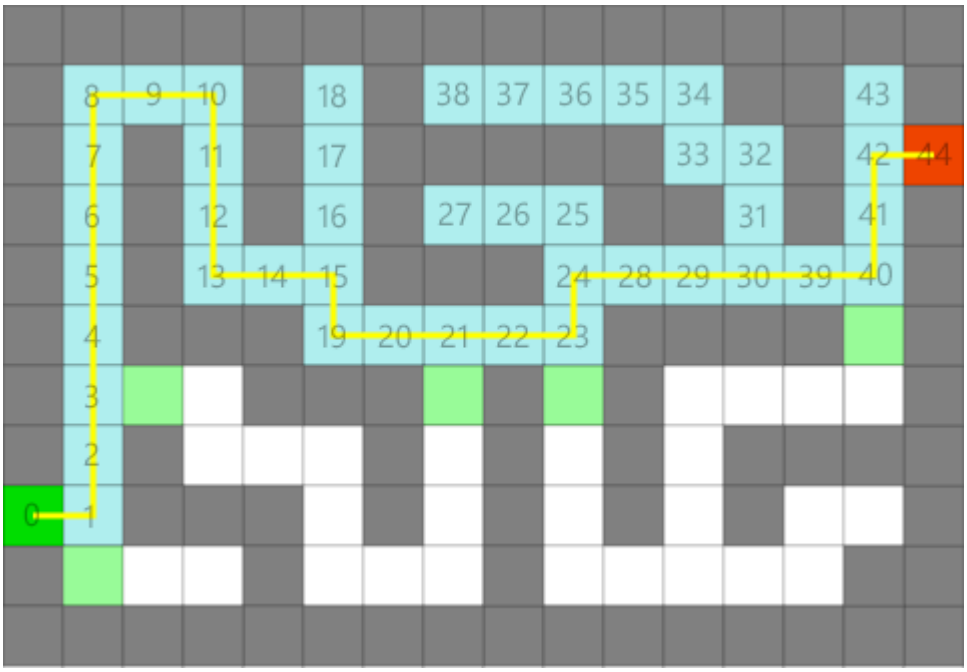
Unsere erste Idee ist es, ein Divide & Conquer-Algorithmus zu entwerfen. Wenn wir es rekursiv machen, geht es schon fast von selbst:

Wenn der Zielknoten in der Nachbarschaft vom Startknoten ist, sind wir fertig. Wenn nicht, entfernen wir den Startknoten vom Graph und erhalten einen kleineren Teilgraphen, den wir rekursiv mit der Tiefensuche durchsuchen.

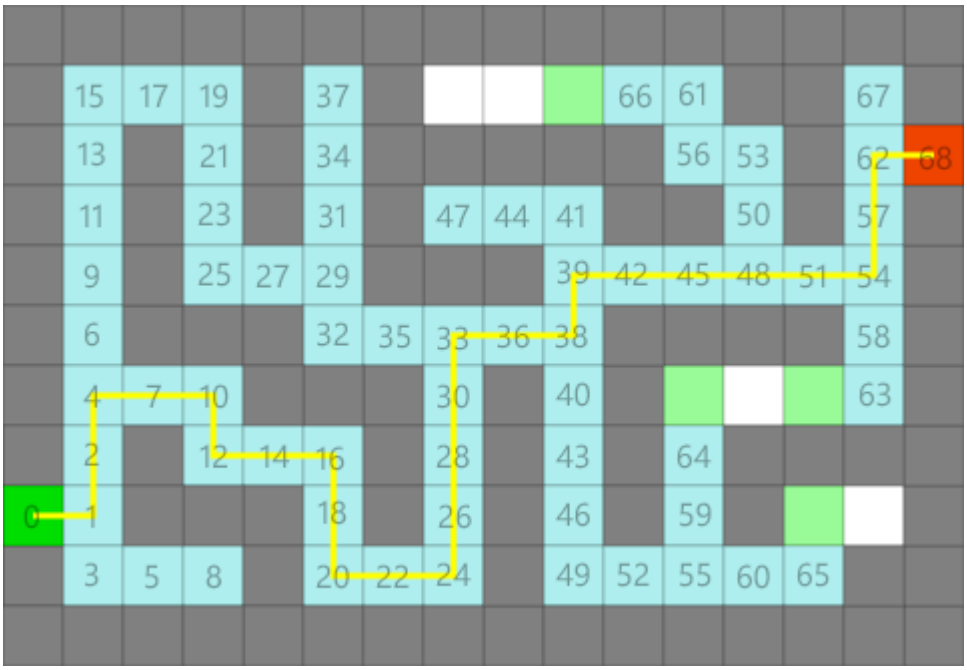
Algorithmus 5. *(Tiefensuche)*

1: $u \in V$ ist der Startknoten, $v \in V$ der Zielknoten, s der Zustandsarray.
2:
3: **function** DepthFirstSearch($u, v, s = \text{None}$)
4: **if** $s = \text{None}$ **then** ▷ Initializierung
5: $s \leftarrow [-1, \ldots, -1]$, so dass $\#s = \#V$
6: $s[u] \leftarrow u$
7: **end if**
8:
9: **for** $u' \in \Gamma[u]$, so dass $s[u'] = -1$ **do**
10: $s[u'] = u$ ▷ Besuche Knoten
11: **if** $u' = v$ **then** ▷ Ziel gefunden
12: Weg aus s zusammensetzen und zurückgeben.
13: **end if**
14:
15: DepthFirstSearch(u', v, s) ▷ Rekursion
16: **end for**
17:
18: **return** None ▷ Knoten nicht erreichbar
19: **end function**

Korollar 6. Mit der Tiefensuche kann man leicht testen, ob der Graph zusammenhängend ist. Man wählt einfach einen nicht vorhandenen Endknoten $v = -2$ und prüft am Ende, ob $s[x] \neq -1$ für alle $x \in V$ ist. Falls nicht, bilden alle Knoten $x \in V$ mit $s[x] \neq -1$ eine Zusammenhangskomponente.



Anstatt in die Tiefe zu suchen, können wir auch in die Breite suchen, d.h. sind wir bei einem Knoten $v \in V$ angelangt, so erforschen wir zunächst alle Nachbarn $u \in \Gamma(v)$, bevor wir bei tiefer weitersuchen. Diese Methode können wir verwenden, um den kürzesten Pfad durch das Labyrinth zu finden.



Algorithmus 7. *(Breitensuche)*

```
1: u ∈ V ist der Startknoten, v ∈ V der Zielknoten.
2:
3: function BreadthFirstSearch(u, v)
4:     s ← [−1, . . . , −1], so dass #s = #V                ▷ Zustandsarray
5:     s[u] ← u
6:     Q ← [u]                                              ▷ Warteschlange
7:
8:     while #Q > 0 do
9:         u ← Q[0] und Q.RemoveFirst( )
10:        if u = v then                                    ▷ Ziel erreicht
11:            Weg aus s zusammensetzen und zurückgeben.
12:        end if
13:
14:        for u′ ∈ Γ[u], so dass s[u′] = −1 do
15:            s[u′] ← u
16:            Q.Append(u′)
17:        end for
18:    end while
19:
20:    return None                                          ▷ Knoten nicht erreichbar
21: end function
```

4 Was man noch alles anstellen kann

- Man kann mit einer veränderten Tiefensuche Kreise oder Zusammenhangskomponenten im Graphen finden.
- Man kann die Knoten eines Graphen $G = (V, E)$ *einfärben*. Mathematisch ist eine *Färbung* eine Abbildung $f : V \to \mathbb{N}$. Eine Farbe entspricht eine natürliche Zahl. Man ist interessiert an *gültige* Färbungen, bei der zwei benachbarte Knoten unterschiedliche Farben haben, d.h:

  $$\forall u, v \in V,\ \{u, v\} \in E :\ f(u) \neq f(v)$$

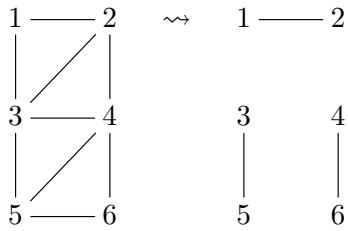
  Wie kann man einen Graphen mit möglichst wenig Farben gültig färben? Anwendungsbeispiel:

  $$V \subset \{\text{Vorlesungen}\}$$

  $$\{u, v\} \in E \iff u \text{ und } v \text{ können/sollen nicht gleichzeitig stattfinden (weil z.B. gleicher Professor)}$$

  Dann gibt uns eine gültige Färbung f von G eine Stundenplaneinteilung, wenn wir die verschiedenen Farben mit zuteilbaren Zeitfenstern assoziieren.

- Ist ein Graph $G = (V, E)$ gegeben, so ist ein *Matching* eine Teilmenge $M \subset E$, so dass keine zwei Kanten aus M einen Knoten gemeinsam haben.



  Gesucht ist ein großtmögliches Matching, d.h. ein Matching M, das als Menge eine maximale Kardinalität unter allen gültigen Matchings von G hat. Anwendungsbeispiel:

$$V \subset \{\text{Arbeitssuchende}\} \cup \{\text{Stellenangebote}\}$$

$$\{u, v\} \in E \iff \text{Arbeitssuchender } u \text{ ist für die Stelle } v \text{ geeignet}$$

  Ein großtmögliches Matching liefert dann eine optimale Zuordnung.

5 Appendix: Der Dijkstra-Algorithmus

Die Breitensuche liefert genau dann den kürzesten Pfad, wenn alle Kanten gleich lang sind wie im Labyrinth oben. Will man verschiedene Längen verschiedene Kanten zuweisen, müssen wir diese Information unserer Struktur hinzufügen. Ein Graph wird zu einem *gewichteten Graphen*, d.h. ein Graph $G = (V, E)$ zusammen mit einer Abbildung $\omega : E \to \mathbb{R}$, das jede Kante $e \in E$ einem Gewicht $\omega(e) \in \mathbb{R}$ zuordnet. Gesucht ist dann ein Pfad $W = (w_0, w_1, \ldots, w_\ell)$ zwischen u und v, so dass der *Gewicht* von W

$$\omega(W) := \sum_{i=1}^{\ell} \omega(\{w_i, w_{i-1}\})$$

minimal ist. Wir können diese Information in einer erweiterten Adjazenzmatrix speichern:

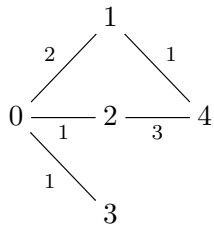
$$A = (a_{uv})_{0 \le u,v \le n}, \qquad a_{uv} = \begin{cases} \omega(\{u, v\}), & \{u, v\} \in E \\ \infty, & \text{sonst} \end{cases}$$

Wir werden zusätzlich den Graph im Form einer Adjazenzliste Γ bereitstellen. Der Algorithmus ähnelt der Breitensuche; der Unterschied liegt darin, in welcher Reihenfolge die Warteschlange abgearbeitet wird. Zusätzlich zu unserer Warteschlange Q, speichern wir in einem weiteren Array P den Vorgänger sämtlicher Knoten in Q und in Ω die Gewichte der gefunden Wege.

Algorithmus 8. *(Dijkstra-Algorithmus)*

```
1: u ∈ V ist der Startknoten, v ∈ V der Zielknoten.
2:
3: function DIJKSTRA(u, v)
4:     s ← [−1, . . . , −1], so dass #s = #V                ▷ Zustandsarray
5:     Q ← [u], P ← [u], Ω ← [0]                            ▷ Warteschlange
6:
7:     while #Q > 0 do
8:         Suche i, so dass Ω[i] = min{Ω[j] | 0 ≤ j < #Ω}
9:         u ← Q[i] und Q.REMOVE(i)
10:
11:        if s[u] ≠ −1 then
12:            P.REMOVE(i) und Ω.REMOVE(i)
13:            continue                                     ▷ Schon besucht
14:        end if
15:
16:        s[u] ← P[i] und P.REMOVE(i)                      ▷ Besuche Knoten
17:        ℓ ← Ω[i] und Ω.REMOVE(i)
18:
19:        if u = v then                                    ▷ Ziel erreicht
20:            Weg aus s zusammensetzen und zurückgeben.
21:        end if
22:
23:        for u′ ∈ Γ[u], so dass s[u′] = −1 do
24:            Q.APPEND(u′)
25:            P.APPEND(u)
26:            Ω.APPEND(ℓ + A[u′, u])
27:        end for
28:    end while
29:
30:    return None                                          ▷ Knoten nicht erreichbar
31: end function
```

Beispiel. Gegeben ist der folgende Graph und wir führen DIJKSTRA$(0, 4)$ aus.

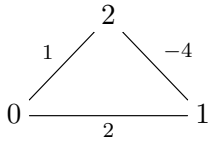


1. $u = 0,\ s[u] = 0,\ \ell = 0,\ Q = [1, 2, 3],\ P = [0, 0, 0],\ \Omega = [2, 1, 1]$
2. $u = 2,\ s[u] = 0,\ \ell = 1,\ Q = [1, 3, 4],\ P = [0, 0, 2],\ \Omega = [2, 1, 4]$

3. $u = 3,\ s[u] = 0,\ \ell = 1,\ Q = [1, 4],\ P = [0, 2],\ \Omega = [2, 4]$
4. $u = 1,\ s[u] = 0,\ \ell = 2,\ Q = [4, 4],\ P = [2, 1],\ \Omega = [4, 3]$
5. $u = 4,\ s[u] = 1,\ \ell = 3$, Ziel erreicht

Bemerkung. Der Dijkstra-Algorithmus funktioniert im Allgemeinen nicht für Graphen mit negativen Kantengewichte.

Beispiel.



Führen wir DIJKSTRA$(0, 2)$ an den Graphen aus, so erhalten wir:

- $u = 0,\ s[u] = 0,\ \ell = 0,\ Q = [2, 1],\ P = [0, 0],\ \Omega = [1, 2]$
- $u = 2,\ s[u] = 0,\ \ell = 1$, Ziel erreicht

Es gibt aber einen kürzeren Weg, nämlich $W = (0, 1, 2)$ mit $\omega(W) = -2$.