Basiswissen ▷ Netzwerkalgorithmen

Version – 16. April 2021

1 Graphen

Ein (einfacher) *Graph* G ist ein geordnetes Paar (V, E), wobei E eine Teilmenge von $\binom{V}{2}$ (die Menge aller 2-elementigen Teilmengen von V) ist. Wir nennen V die Menge der *Knoten* und E die Menge der *Kanten* von G.

Es gibt verschiedene Möglichkeiten Graphen darzustellen. Bei einer typischen Darstellung in der Ebene werden die Knoten durch Punkte repräsentiert und die Kanten durch Kurven, die ihre zwei Knoten verbinden, siehe Abbildung 1.

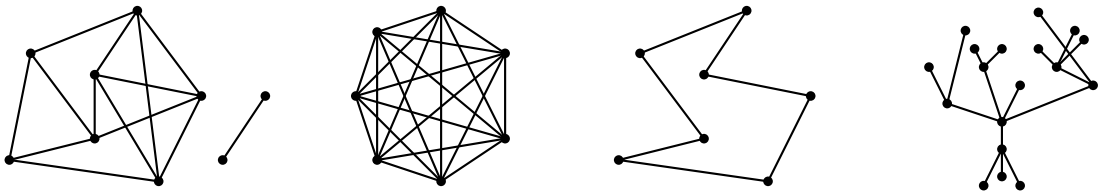


Abbildung 1: Ein nicht zusammenhängender Graph, ein vollständiger Graph, ein Kreis und ein Baum.

Ein Graph $H = (V', E')$ ist ein *Teilgraph* von $G = (V, E)$ wenn $V' \subset V$ und $E' \subset E$ gilt.
Der *Grad* eines Knotens v ist die Anzahl der zu v inzidenten Kanten und wird durch $d(v)$ notiert.

Einige spezielle Graphen

- Ein Graph ist vollständig, wenn $E = \binom{V}{2}$.
- Ein Graph ist bipartit, wenn die Knotenmenge in zwei disjunkte Mengen X und Y zerlegt werden kann, sodass jede Kante einen Knoten aus X und einen Knoten aus Y enthält.
- Eine Knoten-Kanten-Folge $v_0, e_1, v_1, e_2 \ldots, e_k, v_k$ in G heißt *Weg* wenn $e_i = \{v_{i-1}, v_i\}$.
- Ein *Pfad* in G ist ein Weg dessen Knoten paarweise verschieden sind.
- Ein Graph G ist *zusammenhängend*, wenn es für alle $u, w \in V$ einen Weg von u nach w gibt.
- Ein Graph ist ein *Kreis*, wenn er zusammenhängend ist und alle Knoten Grad 2 haben.
- Ein Graph ist ein *Wald*, wenn er keinen Kreis als Teilgraph enthält.
- Ein zusammenhängender Wald ist ein *Baum*.

Gerichtete Graphen In vielen Anwendungen ist es nützlich *gerichtete* Kanten zu betrachten. Ein *gerichteter Graph* D, auch *Digraph*, ist ein geordnetes Paar (V, A), wobei A eine Teilmenge von $V \times V$ (die Menge aller 2-Tupel von V) ist. Wir nennen A die Menge der *gerichteten Kanten* von G. Zur Darstellung gerichteter Graphen verwendet man Pfeile für die gerichteten Kanten, siehe Abbildung 2. Viele Konzepte und Eigenschaften (ungerichteter) Graphen lassen sich mit geeigneten Anpassungen auf gerichtete Graphen übertragen.

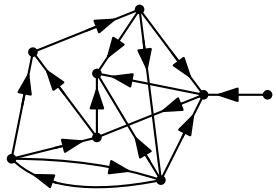


Abbildung 2: Beispiel eines gerichteten Graphen.

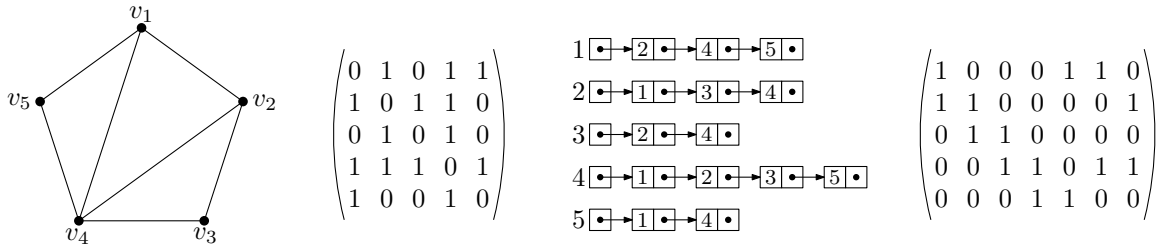
Speicherung von Graphen Drei wichtige Datenstrukturen zur Darstellung eines Graphen im Rechner sind die Adjazenzmatrix, die Adjazenlisten und die Inzidenzmatrix. Sei $G = (V, E)$ ein Graph mit $V = \{v_1, \ldots, v_n\}$ und $E = \{e_1, \ldots, e_m\}$.

- Die *Adjazenzmatrix* des Graphen G ist eine $(n \times n)$-Matrix, wobei der Eintrag $a_{i,j}$ in Zeile i und Spalte j gegeben ist durch
$$a_{i,j} = \begin{cases} 1 & \text{falls } \{v_i, v_j\} \in E, \\ 0 & \text{sonst.} \end{cases}$$
Die Größe der Matrix ist n^2. Ist die Anzahl der Kanten klein im Vergleich zur Anzahl an Knoten, so sind viele Einträge der Matrix 0. Daher kann es sinnvoll sein, eine andere Datenstruktur zu verwenden. Der Test, ob $\{v_i, v_j\} \in E$, bedarf nur das Überprüfen des Eintrags $a_{i,j}$, was in konstanter Zeit möglich ist.

- Die *Adjazenzlisten* von G ist eine Menge von (einfach verketteten) Listen (je eine Liste pro Knoten), in der die jeweils adjazenten Knoten (in alphabetischer oder numerischer Reihenfolge) gespeichert werden. Der Speicherbedarf der Adjazenzlisten ist $(n + 2m)$ (jeder Knoten besitzt eine Liste, jede Kante wird zweimal gespeichert). Dies ist bei *dünnen* Graphen, also Graphen mit wenigen Kanten (im Verhältnis zur Anzahl an Knoten) deutlich sparsamer, als bei der Adjazenzmatrix. Für den Kantentest muss im schlechtesten Fall die Liste eines Knotens komplett durchlaufen werden, weshalb sich eine Laufzeit in $O(\min\{n, m\})$ ergibt.

- Die *Inzidenzmatrix* von G ist eine $(n \times m)$-Matrix, wobei der Eintrag $a_{i,j}$ in Zeile i und Spalte j gegeben ist durch
$$a_{i,j} = \begin{cases} 1 & \text{falls } v_i \in e_j, \\ 0 & \text{sonst.} \end{cases}$$
Die Größe der Matrix ist $(n \cdot m)$. Die Zeit für den Test, ob $\{v_i, v_j\} \in E$, ist in $O(m)$, da zuerst die Zeile v_i durchlaufen und dann bei jeder 1 geschaut werden muss, ob v_j in dieser Spalte 1 oder 0 ist.

Die obigen Datenstrukturen lassen sich leicht auf Digraphen erweitern: Die Adjazenzlisten und die Adjazenzmatrix eines Digraphen sind völlig analog definiert; allerdings ist die Adjazenzmatrix in der Regel nicht symmetrisch. Bei der Indizenzmatrix speichert man eine 1 falls der Knoten Startknoten und -1 falls er Endknoten der gerichteten Kante ist.



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

Abbildung 3: Ein Graph und die Darstellungen als Adjazenzmatrix, Adjazenzlisten und Inzidenzmatrix.

2 Laufzeiten von Algorithmen

Bei der Analyse von Algorithmen ist die Laufzeit ein essentieller Parameter. Für einen konkreten Algorithmus ist die Anzahl der Rechenschritte unmittelbar von der Eingabe abhängig. Man betrachtet deshalb Eingaben fester Größe und unterscheidet zwischen folgenden Fällen:

- Worst-Case – ungünstigster Fall
  Maximum aller möglichen Laufzeiten bei einer Eingabe fester Länge n.
- Average-Case – durchschnittlicher Fall
  Arithmetisches Mittel aller möglichen Laufzeiten bei einer Eingabe fester Länge n.
- Best-Case – günstigster Fall
  Minimum aller möglichen Laufzeiten bei einer Eingabe fester Länge n.

In der Regel ist es schwierig oder gar unmöglich, die genaue Gleichung für den Zeitaufwand für einen Algorithmus anzugeben. Man versucht daher, den Aufwand asymptotisch abzuschätzen, d. h. eine möglichst einfache Funktion zu finden, die für große n die Funktion des Zeitaufwandes $T(n)$ nach oben hin beschränkt. Es soll also $T(n) \leq c \cdot f(n)$ gelten, ab einem bestimmten Element n_0, wobei c eine positive Konstante ist. Insbesondere sind die genauen Konstanten nicht von Bedeutung.

O-Notation – eine obere Schranke

$O(f(n))$ ist die Menge aller Funktionen die asymptotisch höchstens so schnell wachsen wie $f(n)$. Die formale Definition lautet wie folgt:

$$O(f(n)) := \{g : \mathbb{N} \mapsto \mathbb{R} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \text{ sodass } g(n) \leq c \cdot f(n) \text{ für alle } n \geq n_0\}$$

In Worten: Es gilt $g(n) \in O(f(n))$ genau dann wenn zwei Konstanten $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ existieren, sodass für alle $n \geq n_0$ die Ungleichung $0 \leq g(n) \leq c \cdot f(n)$ gilt.

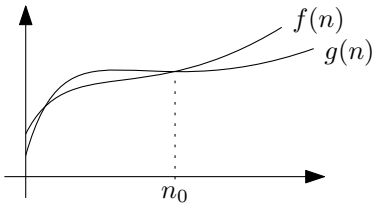


Abbildung 4: Es gilt $g(n) \in O(f(n))$, da $g(n) \leq 1 \cdot f(n)$ für alle $n \geq n_0$.

Ω-Notation – eine untere Schranke

$\Omega(f(n))$ ist die Menge aller Funktionen die asymptotisch mindestens so schnell wachsen wie $f(n)$.

$$\Omega(f(n)) := \{g : \mathbb{N} \mapsto \mathbb{R} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \text{ sodass } c \cdot f(n) \leq g(n) \text{ für alle } n \geq n_0\}$$

In Worten: Es gilt $g(n) \in \Omega(f(n))$, genau dann wenn zwei Konstanten $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ existieren, sodass für alle $n \geq n_0$ die Ungleichung $0 \leq c \cdot f(n) \leq g(n)$ gilt.

Θ-Notation – eine obere und untere Schranke

$\Theta(f(n))$ ist die Menge aller Funktionen die asymptotisch genauso so schnell wachsen wie $f(n)$.

$$\Theta(f(n)) := O(f(n)) \cap \Omega(f(n))$$

In Worten: Es gilt $g(n) \in \Theta(f(n))$ genau dann, wenn $g(n) \in O(f(n))$ und $g(n) \in \Omega(f(n))$ gelten.

Beispiele Für $p(n) = 3n^2 + 9n - 1$ gilt $p(n) \in \Theta(n^2)$, aber auch $p(n) \in \Omega(n)$ und $p(n) \in O(n^3)$. Auf der anderen Seite gilt $3^n \notin O(2^n)$. Einen Beweis und Weiteres findet ihr auf dem Merkzettel[1].

3 Graphen-Algorithmen

Es gibt zwei einfache Algorithmen um die Knoten eines Graphen $G = (V, E)$ systematisch von einem Startknoten v aus zu durchsuchen – die Breitensuche und die Tiefensuche. Beide Methoden finden Verwendung als Subroutinen in weiteren, komplexeren Graphen-Algorithmen. Sie haben eine Laufzeit von $O(|V| + |E|)$ und unterscheiden sich durch die jeweils verwendete Datenstruktur grundlegend in der Suchreihenfolge.

Breitensuche Die bei der Breitensuche verwendete Datenstruktur ist eine Warteschlange. Für den gerade aktiven Knoten u wird für jeden adjazenten Knoten von u (also jeden Nachbarn) getestet, ob er schon entdeckt wurde, oder nicht. Falls ja, wird w übersprungen, falls nicht, wird w zur Warteschlange hinzugefügt. Die Breitensuche „entdeckt“ also zuerst alle Knoten mit einem Abstand von 1 zum Startknoten v (die Nachbarn von v), danach alle Knoten mit einem Abstand von 2 (also alle noch nicht entdeckten Nachbarn der Nachbarn von v), dann mit Abstand 3 und so weiter. Der Algorithmus liest sich wie folgt:

Algorithmus 1 Breitensuche

```
1:  procedure BREITENSUCHE(G, v)
2:      Initialisiere eine Warteschlange Q          ▷ Initialisiere eine Warteschlange
3:      Q.enqueue(v)                                ▷ Initialisiere Q mit dem Startknoten v
4:      while Q ist nicht leer do
5:          v = Q.dequeue()                         ▷ Entferne v aus der Warteschlange
6:          for all Nachbarn w von v do
7:              if w ist nicht markiert then        ▷ Wenn w noch nicht entdeckt wurde...
8:                  markiere w                      ▷ Markiere w als entdeckt
9:                  Q.enqueue(w)                    ▷ Füge w zur Warteschlange hinzu
10:             end if
11:         end for
12:     end while
13: end procedure
```

[1] https://www.ibr.cs.tu-bs.de/alg/Merkzettel/runtime-booklet.pdf

Tiefensuche Die Tiefensuche verwendet als Datenstruktur einen Stapel. Im Gegensatz zur Breitensuche läuft die Tiefensuche zuerst vollständig entlang eines Pfades in die Tiefe, bis abzweigende Pfade genutzt werden. Für den gerade aktiven Knoten u wird für eine Kante $\{u, w\}$ getestet, ob w schon entdeckt wurde oder nicht. Ist dies nicht der Fall, wird auf w rekursiv die Tiefensuche durchgeführt. Wird vom aktuellen Knoten kein unentdeckter Knoten mehr über eine Kante direkt erreicht, prüft der Algorithmus ob noch unentdeckte Knoten für den Vorgänger existieren und so weiter. Der Algorithmus liest sich wie folgt:

Algorithmus 2 Tiefensuche

```
1:  procedure TIEFENSUCHE(G, v)
2:      Sei S ein Stapel                          ▷ Initialisiere den Stapel S als Datenstruktur
3:      S.push(v)                                 ▷ Initialisiere S mit dem Startknoten v
4:      while S ist nicht leer do
5:          u = S.pop()                           ▷ Entferne u vom Stapel
6:          if u ist nicht markiert then          ▷ Wenn u noch nicht entdeckt wurde...
7:              markiere u
8:              for all Kanten e = {u, w} do      ▷ Für jeden Nachbarn von u
9:                  S.push(w)                     ▷ Lege w auf den Stapel
10:             end for
11:         end if
12:     end while
13: end procedure
```

Abbildung 5 illustriert einen Graph G und die beiden Suchbäume.

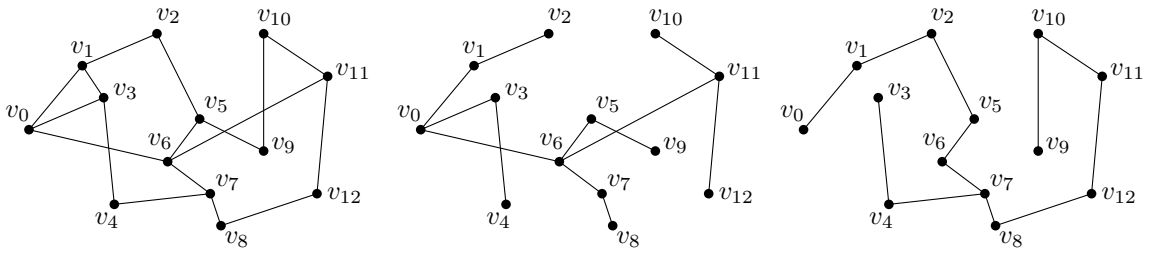


Abbildung 5: Ein Graph G und sein Breitensuche- und Tiefensuche-Baum je mit Startknoten v_0.

4 Beweistechniken

Eine *Aussage* ist ein Satz, der entweder wahr oder falsch ist. Beispiele:

- Braunschweig ist 200km von Hamburg entfernt.
- Wenn es regnet, dann ist die Straße nass.
- 391 ist durch 7 teilbar.

Aussagen lassen sich verknüpfen: $\wedge$ ('und'), $\vee$ ('oder'), $\neg$ ('nicht'), $\Longrightarrow$ ('daraus folgt'), $\Longleftrightarrow$ ('äquivalent').

Um zu zeigen, dass eine Aussage wahr oder falsch ist, ist eine logisch vollständige Begründung, ein sogenannter *Beweis*, anzugeben. Dazu gibt es verschiedene Techniken.

Direkter Beweis Zu beweisende mathematische Aussagen haben häufig die Form *„Wenn ..., dann ...“*. Aussagenlogisch ist dies $A \implies B$ (aus A folgt B). Dabei nennen wir A die Voraussetzung und B die Folgerung. Bei dem direkten Beweis geht es darum, aus A über eine logische Folge auf B zu schließen.

- Wenn eine Zahl durch 10 teilbar ist, dann ist sie auch durch 5 teilbar.

  Beweis. Sei x eine Zahl, die durch 10 teilbar ist. Dann gibt es also eine ganze Zahl k, sodass gilt $x = 10 \cdot k$. Durch Umformung erhalten wir $x = 10 \cdot k = 2 \cdot 5 \cdot k = 5 \cdot k'$ für $k' = 2 \cdot k$. Nach der Definition der Teilbarkeit ist also x auch durch 5 teilbar. □

Indirekter Beweis Manchmal ist es schwierig eine Aussage direkt zu zeigen. Die Beweistechnik des indirekten Beweises basiert auf der Tatsache, dass die Aussage $(A \implies B)$ äquivalent zur Aussage $(\neg B \implies \neg A)$ ist. Diese Tatsache kann man sich sehr leicht am Beispiel intuitiv klar machen (und zum Beispiel mit einer Wahrheitstafel beweisen). Sei A die Aussage ‘Es regnet.’ und B die Aussage ‘Die Straße ist nass.’. Dann ist die Aussage *‘Wenn es regnet, dann ist die Straße nass.’* $(A \implies B)$ äquivalent zu der Aussage *‘Wenn die Straße nicht nass ist, dann regnet es nicht.’* $(\neg B \implies \neg A)$.

- Für alle $x, y \in \mathbb{N}$ mit $x \neq y$ gilt: Der Bruch $\frac{x+y}{x-y}$ ist gekürzt $\implies$ der Bruch $\frac{x}{y}$ ist gekürzt.

  Beweis. Angenommen $\frac{x}{y}$ kann gekürzt werden. Dann besitzen x und y einen gemeinsamen Teiler m, so dass $r, s \in \mathbb{N}$ existieren mit $x = m \cdot r$ und $y = m \cdot s$. Damit ist aber $x - y = m(r - s)$ und $x + y = m(r + s)$. Also besitzen auch $(x - y)$ und $(x + y)$ den gemeinsamen Teiler m und der Bruch $\frac{x+y}{x-y}$ kann gekürzt werden. □

Äquivalenzen Äquivalenzbeweise sind für Aussagen der Form $A \iff B$ gefordert, d.h. A gilt genau dann, wenn B gilt. Manchmal geht dies direkt über Äquivalenzumformungen. Funktioniert dies nicht, teilt man die Äquivalenzaussage in die beiden Implikationen $(A \implies B)$ und $(B \implies A)$ auf und zeigt diese mit einer geeigneten Beweismethode.

Induktion Bei der vollständigen Induktion handelt es sich um ein Prinzip bei dem man in zwei Schritten Aussagen für unendlich viele natürliche Zahlen zeigen kann. Sei $n, n_0 \in \mathbb{N}$. Um zu zeigen, dass eine Aussage $A(n)$ für alle $n \geq n_0$ gilt, genügt es folgende zwei Dinge zu zeigen: Erstens $A(n_0)$ ist wahr. Zweitens, für alle $n \geq n_0$ gilt: $A(n) \implies A(n+1)$.

Theorem 1 (Vollständige Induktion)**.** *Sei $A(n)$ eine Aussage für $n \in \mathbb{N}$. Wenn $A(n_0)$ für ein $n_0 \in \mathbb{N}$ und $A(n) \implies A(n+1)$ für alle $n \geq n_0$ gilt, so gilt $A(n)$ für alle $n \geq n_0$.*

- Für einen Graphen $G = (V, E)$ gilt: $2|E| = \sum_{v \in V} d(v)$.

  Beweis. Wir beweisen diese Behauptung mit vollständiger Induktion über n.

  Induktionsanfang: $n = 1$. Ein Graph mit $n = 1$ Knoten hat keine Kante, also gilt $2|E| = 0$ und $\sum_{v \in V} d(v) = 0$.

  Induktionsvoraussetzung: Die Behauptung gilt für beliebiges aber festes n.

  Induktionsschritt: Wir betrachten einen Graph $G = (V, E)$ mit $n + 1$ Knoten und konstruieren daraus einen Graph $G' = (V - v, E')$ mit n Knoten wie folgt: Wir wählen einen beliebig aber festen Knoten $v \in V$ und löschen v und alle inzidenten Kanten von v. Nach Konstruktion gilt $|E| = |E'| + d(v)$. Nach Induktionsvoraussetzung gilt: $2|E'| = \sum_{u \in V} d'(u)$, wobei $d'(u)$ den Knotengrad in G' beschreibt. Im Vergleich zu G hat sich der Knotengrad in G' wie folgt geändert. Für alle Nachbarn von v (Knoten, die sich eine Kante mit v teilen) ist der Knotengrad um 1

geringer; für alle anderen Knoten hat sich der Grad nicht geändert. Wir notieren die Menge aller Nachbarn von v durch $N(v)$. Daher gilt:

$$\begin{aligned}
\sum_{u \in V} d(u) &= d(v) + \sum_{u \in N(v)} d(u) + \sum_{u \in V - v - N(v)} d(u) \\
&= d(v) + \sum_{u \in N(v)} (d'(u) + 1) + \sum_{u \in V - v - N(v)} d'(u) \\
&= d(v) + \sum_{u \in V - v} d'(u) + |N(v)| \\
&\overset{\text{I.V.}}{=} d(v) + 2|E'| + |N(v)| \\
&\overset{|N(v)| = d(v)}{=} 2(|E'| + d(v)) \\
&\overset{|E| = |E'| + d(v)}{=} 2|E|
\end{aligned}$$

Somit haben wir die Aussage für Graphen mit $n+1$ Knoten bewiesen unter der Annahme ihrer Gültigkeit für Graphen mit n Knoten. □

Weiteres zum Thema findet ihr auf dem Merkzettel[2] und zwei Foliensätzen[3,4].

5 Sonstiges

Problem vs. Instanz

Wir betrachten folgende Fragestellung:

Problem.
Gegeben: Ein Graph $G = (V, E)$.
Gesucht: Ein Kreis in G, der jeden Knoten genau einmal besucht.

Dies ist wie folgt zu verstehen. Für jede beliebige Eingabe eines Graphen wollen wir einen Kreis finden, der jeden Knoten genau einmal besucht. Wenn wir einen konkreten Graphen betrachten, wie zum Beispiel den Graph aus Abbildung 6, dann betrachten wir eine *Instanz* des Problems und versuchen die allgemeine Frage für diesen einen Fall zu lösen.

Zur Lösung des Problems ist also die Angabe einer Strategie / Algorithmus der für jede Eingabe eine korrekte Lösung berechnet. Für die Lösung der Instanz des Problems genügt die Angabe eines Kreises, in diesem Fall also eine Reihenfolge, in der die Knoten besucht werden.

Entscheidungsproblem vs. Optimierungsproblem.

Betrachten wir die folgenden zwei Fragestellungen:

Problem. Hamiltonkreis-Problem
Gegeben: Ein Graph $G = (V, E)$.
Gesucht: Ein Kreis in G, der jeden Knoten genau einmal besucht.

Problem. Das Problem des Handlungsreisenden – Traveling-Salesman-Problem
Gegeben: Ein Graph $G = (V, E)$ und eine Kostenfunktion $c \colon E \to \mathbb{R}$.
Gesucht: Einen Kreis minimalen Gewichts in G, der jeden Konten genau einmal besucht.

[2] `https://www.ibr.cs.tu-bs.de/alg/Merkzettel/proof-booklet.pdf`
[3] `https://www.ibr.cs.tu-bs.de/courses/ws1819/aud/uebungen/U2.pdf`
[4] `https://www.ibr.cs.tu-bs.de/courses/ws1819/aud/uebungen/U3.pdf`

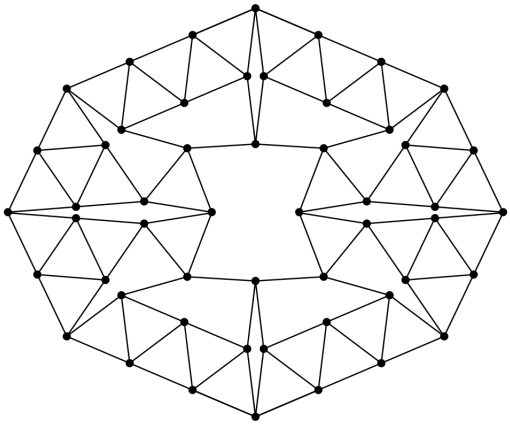


Abbildung 6: Ein Graph G. Es handelt sich hierbei um den *Harborth-Graphen*.

Das Hamiltonkreis-Problem ist ein Entscheidungsproblem. Die Eingabe ist (in diesem Fall) ein beliebiger Graph G und die Ausgabe ist eine spezielle Struktur (in diesem Fall ein Kreis, der jeden Knoten genau einmal besucht), oder ein Argument, dass eine solche Struktur in G nicht existiert.

Das Problem des Handlungsreisenden ist ein Optimierungsproblem. Die Eingabe ist (in diesem Fall) wieder ein beliebiger Graph G und die Ausgabe ist die Beste unter allen gültigen Strukturen (in diesem Fall ein Kreis minimalen Gewichts, der jeden Knoten genau einmal besucht), oder ein Argument, dass eine solche Struktur in G nicht existiert.